5 Machine Learning

5.1 Introduction

Machine learning algorithms are general purpose tools for generalizing from data. They have proven to be able to solve problems from many disciplines without detailed domain-specific knowledge. To date they have been highly successful for a wide range of tasks including computer vision, speech recognition, document classification, automated driving, computational science, and decision support.

The core problem. A core problem underlying many machine learning applications is learning a good classification rule from labeled data. This problem consists of a domain of interest $\mathcal{X}$, called the *instance space*, such as the set of email messages or patient records, and a classification task, such as classifying email messages into spam versus non-spam or determining which patients will respond well to a given medical treatment. We will typically assume our instance space $\mathcal{X} = \{0,1\}^d$ or $\mathcal{X} = \mathbb{R}^d$, corresponding to data that is described by d Boolean or real-valued features. Features for email messages could be the presence or absence of various types of words, and features for patient records could be the results of various medical tests. To perform the learning task, our learning algorithm is given a set S of labeled *training examples*, which are points in $\mathcal{X}$ along with their correct classification. This training data could be a collection of email messages, each labeled as spam or not spam, or a collection of patients, each labeled by whether or not they responded well to the given medical treatment. Our algorithm then aims to use the training examples to produce a classification rule that will perform well over new data, i.e., new points in $\mathcal{X}$. A key feature of machine learning, which distinguishes it from other algorithmic tasks, is that our goal is *generalization*: to use one set of data in order to perform well on new data we have not seen yet. We focus on *binary classification* where items in the domain of interest are classified into two categories (called the positive class and the negative class), as in the medical and spam-detection examples above, but nearly all the techniques described here will also apply to multi-way classification.

How to learn. A high-level approach that many algorithms we discuss will follow is to try to find a "simple" rule with good performance on the training data. For instance, in the case of classifying email messages, we might find a set of highly indicative words such that every spam email in the training data has at least one of these words and none of the non-spam emails has any of them; in this case, the rule "if the message has any of these words then it is spam, else it is not" would be a simple rule that performs well on the training data. Or, we might find a way of weighting words with positive and negative weights such that the total weighted sum of words in the email message is positive on the spam emails in the training data, and negative on the non-spam emails. We will then argue that so long as the training data is representative of what future data will look like, we can be confident that any sufficiently "simple" rule that performs well on the training data will also perform well on future data. To make this into a formal math-

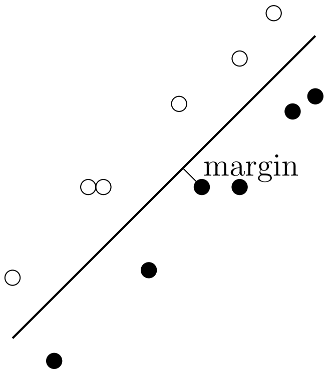


Figure 5.1: Margin of a linear separator.

ematical statement, we need to be precise about what we mean by “simple” as well as what it means for training data to be “representative” of future data. In fact, we will see several notions of complexity, including bit-counting and VC-dimension, that will allow us to make mathematical statements of this form. These statements can be viewed as formalizing the intuitive philosophical notion of Occam’s razor.

5.2 The Perceptron algorithm

To help ground our discussion, we begin by describing a specific interesting learning algorithm, the *Perceptron algorithm*, for the problem of assigning positive and negative weights to features (such as words) so that each positive example has a positive sum of feature weights and each negative example has a negative sum of feature weights.

More specifically, the Perceptron algorithm is an efficient algorithm for finding a linear separator in d-dimensional space, with a running time that depends on the *margin of separation* of the data. We are given as input a set S of training examples (points in d-dimensional space), each labeled as positive or negative, and our assumption is that there exists a vector $\mathbf{w}^*$ such that for each positive example $\mathbf{x} \in S$ we have $\mathbf{x}^T\mathbf{w}^* \geq 1$ and for each negative example $\mathbf{x} \in S$ we have $\mathbf{x}^T\mathbf{w}^* \leq -1$. Note that the quantity $\mathbf{x}^T\mathbf{w}^*/|\mathbf{w}^*|$ is the distance of the point $\mathbf{x}$ to the hyperplane $\mathbf{x}^T\mathbf{w}^* = 0$. Thus, we can view our assumption as stating that there exists a linear separator through the origin with all positive examples on one side, all negative examples on the other side, and all examples at distance at least $\gamma = 1/|\mathbf{w}^*|$ from the separator. This quantity γ is called the *margin* of separation (see Figure 5.1).

The goal of the Perceptron algorithm is to find a vector $\mathbf{w}$ such that $\mathbf{x}^T\mathbf{w} > 0$ for all positive examples $\mathbf{x} \in S$, and $\mathbf{x}^T\mathbf{w} < 0$ for all negative examples $\mathbf{x} \in S$. It does so via the following update rule:

The Perceptron Algorithm: Start with the all-zeroes weight vector $\mathbf{w} = \mathbf{0}$. Then repeat the following until $\mathbf{x}^T\mathbf{w}$ has the correct sign for all $\mathbf{x} \in S$ (positive for positive examples and negative for negative examples):

1. Let $\mathbf{x} \in S$ be an example for which $\mathbf{x}^T\mathbf{w}$ does not have the correct sign.

2. Update as follows:

   (a) If $\mathbf{x}$ is a positive example, let $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}$.

   (b) If $\mathbf{x}$ is a negative example, let $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}$.

While simple, the Perceptron algorithm indeed will find a linear separator whenever one exists, making at most $(R/\gamma)^2$ updates where $R = \max_{\mathbf{x}\in S} |\mathbf{x}|$. Thus, if there exists a hyperplane through the origin that correctly separates the positive examples from the negative examples by a large margin relative to the radius of the smallest ball enclosing the data, then the total number of updates will be small.

Theorem 5.1 *If there exists a vector $\mathbf{w}^*$ such that $\mathbf{x}^T\mathbf{w}^* \geq 1$ for all positive examples $\mathbf{x} \in S$ and $\mathbf{x}^T\mathbf{w}^* \leq -1$ for all negative examples $\mathbf{x} \in S$ (i.e., a linear separator of margin $\gamma = 1/|\mathbf{w}^*|$), then the number of updates made by the Perceptron algorithm is at most $R^2|\mathbf{w}^*|^2$, where $R = \max_{\mathbf{x}\in S} |\mathbf{x}|$.*

To get a feel for this bound, notice that if we multiply all entries in all the $\mathbf{x} \in S$ by 100, we can divide all entries in $\mathbf{w}^*$ by 100 and it will still satisfy the "if"condition. So the bound is invariant to this kind of scaling, i.e., to our "units of measurement".

Proof of Theorem 5.1: Fix some $\mathbf{w}^*$ satisfying the "if" condition of the theorem. We will keep track of two quantities, $\mathbf{w}^T\mathbf{w}^*$ and $|\mathbf{w}|^2$. First of all, each time we make an update, $\mathbf{w}^T\mathbf{w}^*$ increases by at least 1. That is because if $\mathbf{x}$ is a positive example, then

$$(\mathbf{w} + \mathbf{x})^T\mathbf{w}^* = \mathbf{w}^T\mathbf{w}^* + \mathbf{x}^T\mathbf{w}^* \geq \mathbf{w}^T\mathbf{w}^* + 1,$$

by definition of $\mathbf{w}^*$. Similarly, if $\mathbf{x}$ is a negative example, then

$$(\mathbf{w} - \mathbf{x})^T\mathbf{w}^* = \mathbf{w}^T\mathbf{w}^* - \mathbf{x}^T\mathbf{w}^* \geq \mathbf{w}^T\mathbf{w}^* + 1.$$

Next, on each update, we claim that $|\mathbf{w}|^2$ increases by at most R^2. Let us first consider updates on positive examples. If we update on a positive example $\mathbf{x}$ then we have

$$(\mathbf{w} + \mathbf{x})^T(\mathbf{w} + \mathbf{x}) = |\mathbf{w}|^2 + 2\mathbf{x}^T\mathbf{w} + |\mathbf{x}|^2 \leq |\mathbf{w}|^2 + |\mathbf{x}|^2 \leq |\mathbf{w}|^2 + R^2,$$

where the middle inequality comes from the fact that we only perform an update on a positive example when $\mathbf{x}^T\mathbf{w} \leq 0$. Similarly, if we update on a negative example $\mathbf{x}$ then we have

$$(\mathbf{w} - \mathbf{x})^T(\mathbf{w} - \mathbf{x}) = |\mathbf{w}|^2 - 2\mathbf{x}^T\mathbf{w} + |\mathbf{x}|^2 \leq |\mathbf{w}|^2 + |\mathbf{x}|^2 \leq |\mathbf{w}|^2 + R^2.$$

Note that it is important here that we only update on examples for which $\mathbf{x}^T\mathbf{w}$ has the incorrect sign.

So, if we make M updates, then $\mathbf{w}^T\mathbf{w}^* \geq M$, and $|\mathbf{w}|^2 \leq MR^2$, or equivalently, $|\mathbf{w}| \leq R\sqrt{M}$. Finally, we use the fact that $\mathbf{w}^T\mathbf{w}^*/|\mathbf{w}^*| \leq |\mathbf{w}|$ which is just saying that the projection of $\mathbf{w}$ in the direction of $\mathbf{w}^*$ cannot be larger than the length of $\mathbf{w}$. This gives us:

$$\begin{aligned} M/|\mathbf{w}^*| &\leq R\sqrt{M} \\ \sqrt{M} &\leq R|\mathbf{w}^*| \\ M &\leq R^2|\mathbf{w}^*|^2 \end{aligned}$$

as desired. ■

What if there is no $\mathbf{w}^*$ that *perfectly* separates the positive and negative examples? In Section 5.8 we will address this in the context of an online learning model, and we will see that the Perceptron algorithm enjoys strong guarantees even if the best $\mathbf{w}^*$ is not quite perfect, as a function of a quantity called the "hinge loss" of $\mathbf{w}^*$.

In the next section, we consider a related issue. Suppose the positive and negative examples are not *linearly* separable (there is no hyperplane with the positives on one side and the negatives on the other) but they are separable by some other simple curve such as a circle. In that case, we can use a technique known as *kernel functions*.

5.3 Kernel Functions

Suppose that instead of a linear separator decision boundary, the boundary between important emails and unimportant emails looks more like a circle, for example as in Figure 5.2.

A powerful idea for addressing situations like this is to use what are called *kernel functions*, or sometimes the "kernel trick". Here is the idea. Suppose you have a function K, called a "kernel", over pairs of data points such that for some function $\phi : \mathbb{R}^d \to \mathbb{R}^N$, where perhaps $N \gg d$, we have $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}')$. In that case, if we can write the Perceptron algorithm so that it only interacts with the data via dot-products, and then replace every dot-product with an invocation of K, then we can act as if we had performed the function ϕ explicitly without having to actually compute ϕ.

For example, consider $K(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T\mathbf{x}')^k$ for some integer $k \geq 1$. It turns out this corresponds to a mapping ϕ into a space of dimension $N \approx d^k$. For example, in the case $d = 2, k = 2$ we have (using x_i to denote the ith coordinate of $\mathbf{x}$):

$$\begin{aligned} K(\mathbf{x}, \mathbf{x}') &= (1 + x_1x_1' + x_2x_2')^2 \\ &= 1 + 2x_1x_1' + 2x_2x_2' + x_1^2x_1'^2 + 2x_1x_2x_1'x_2' + x_2^2x_2'^2 \\ &= \phi(\mathbf{x})^T\phi(\mathbf{x}') \end{aligned}$$

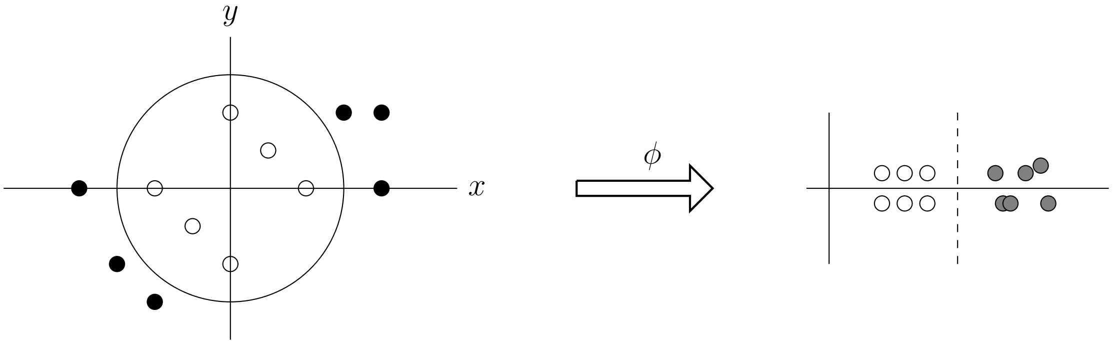


Figure 5.2: Data that is not linearly separable in the input space $\mathbb{R}^2$ but that is linearly separable in the "ϕ-space," $\phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2)$, corresponding to the kernel function $K(\mathbf{x}, \mathbf{x}') = (1 + x_1x_1' + x_2x_2')^2$.

for $\phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2)$. Notice also that a linear separator in this space could correspond to a more complicated decision boundary such as an ellipse in the original space. For instance, the hyperplane $\phi(\mathbf{x})^T\mathbf{w}^* = 0$ for $\mathbf{w}^* = (-4, 0, 0, 1, 0, 1)$ corresponds to the circle $x_1^2 + x_2^2 = 4$ in the original space, such as in Figure 5.2.

The point of this is that if in the higher-dimensional "ϕ-space" there is a $\mathbf{w}^*$ such that the bound of Theorem 5.1 is small, then the algorithm will halt after not too many updates (and later we will see that under reasonable assumptions on the data, this implies we can be confident in its ability to perform well on new data as well). But the nice thing is we didn't have to computationally perform the mapping ϕ!

So, how can we view the Perceptron algorithm as only interacting with data via dot-products? Notice that $\mathbf{w}$ is always a linear combination of data points. For example, if we made updates on examples $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_5$, and these examples were positive, positive, and negative respectively, we would have $\mathbf{w} = \mathbf{x}_1 + \mathbf{x}_2 - \mathbf{x}_5$. So, if we keep track of $\mathbf{w}$ this way, then to classify a new example $\mathbf{x}$, we can write $\mathbf{x}^T\mathbf{w} = \mathbf{x}^T\mathbf{x}_1 + \mathbf{x}^T\mathbf{x}_2 - \mathbf{x}^T\mathbf{x}_5$. So if we just replace each of these dot-products with "K", we are running the algorithm as if we had explicitly performed the ϕ mapping. This is called "kernelizing" the algorithm.

Many different pairwise functions on examples are legal kernel functions. One easy way to create a kernel function is by combining other kernel functions together, via the following theorem.

Theorem 5.2 *Suppose K_1 and K_2 are kernel functions. Then*

1. *For any constant $c \geq 0$, cK_1 is a legal kernel. In fact, for any scalar function f, the function $K_3(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})f(\mathbf{x}')K_1(\mathbf{x}, \mathbf{x}')$ is a legal kernel.*

2. *The sum $K_1 + K_2$, is a legal kernel.*

3. The product, K_1K_2, is a legal kernel.

You will prove Theorem 5.2 in Exercise 5.9. Notice that this immediately implies that the function $K(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T\mathbf{x}')^k$ is a legal kernel by using the fact that $K_1(\mathbf{x}, \mathbf{x}') = 1$ is a legal kernel, $K_2(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}'$ is a legal kernel, then adding them, and then multiplying that by itself k times. Another popular kernel is the Gaussian kernel, defined as:

$$K(\mathbf{x}, \mathbf{x}') = e^{-c|\mathbf{x}-\mathbf{x}'|^2}.$$

If we think of a kernel as a measure of similarity, then this kernel defines the similarity between two data objects as a quantity that decreases exponentially with the squared distance between them. The Gaussian kernel can be shown to be a true kernel function by first writing it as $f(\mathbf{x})f(\mathbf{x}')e^{2c\mathbf{x}^T\mathbf{x}'}$ for $f(\mathbf{x}) = e^{-c|\mathbf{x}|^2}$ and then taking the Taylor expansion of $e^{2c\mathbf{x}^T\mathbf{x}'}$, applying the rules in Theorem 5.2. Technically, this last step requires considering countably infinitely many applications of the rules and allowing for infinite-dimensional vector spaces.

5.4 Generalizing to New Data

So far, we have focused on the problem of finding a classification rule that performs well on a given set S of training data. But what we really want our classification rule to do is to perform well on new data we have not seen yet. To make guarantees of this form, we need some assumption that our training data is somehow representative of what new data will look like; formally, we will assume they are drawn from the same probability distribution. Additionally, we will see that we will want our algorithm's classification rule to be "simple" in some way. Together, these two conditions will allow us to make *generalization guarantees*: guarantees on the ability of our learned classification rule to perform well on new unseen data.

Formalizing the problem. To formalize the learning problem, assume there is some probability distribution D over the instance space $\mathcal{X}$, such that (a) our training set S consists of points drawn independently at random from D, and (b) our objective is to predict well on new points that are also drawn from D. This is the sense in which we assume that our training data is representative of future data. Let c^*, called the *target concept*, denote the subset of $\mathcal{X}$ corresponding to the positive class for the binary classification we are aiming to make. For example, c^* would correspond to the set of all patients who respond well to a given treatment in a medical scenario, or it could correspond to the set of all spam emails in a spam-detection scenario. So, each point in our training set S is labeled according to whether or not it belongs to c^* and our goal is to produce a set $h \subseteq \mathcal{X}$, called our *hypothesis*, which is close to c^* with respect to distribution D. The *true error* of h is $err_D(h) = \text{Prob}(h \triangle c^*)$ where "$\triangle$" denotes symmetric difference, and probability mass is according to D. In other words, the true error of h is the probability it incorrectly classifies a data point drawn at random from D. Our goal is to produce h of low true error. The *training error* of h, denoted $err_S(h)$, is the fraction of points in S on which h and

c^* disagree. That is, $err_S(h) = |S \cap (h \triangle c^*)|/|S|$. Training error is also called *empirical error*. Note that even though S is assumed to consist of points randomly drawn from D, it is possible for a hypothesis h to have low training error or even to completely agree with c^* over the training sample, and yet have high true error. This is called *overfitting* the training data. For instance, a hypothesis h that simply consists of listing the positive examples in S, which is equivalent to a rule that memorizes the training sample and predicts positive on an example if and only if it already appeared positively in the training sample, would have zero training error. However, this hypothesis likely would have high true error and therefore would be highly overfitting the training data. More generally, overfitting is a concern because algorithms will typically be optimizing over the training sample. To design and analyze algorithms for learning, we will have to address the issue of overfitting.

To analyze overfitting, we introduce the notion of an hypothesis class, also called a concept class or set system. An hypothesis class $\mathcal{H}$ over $\mathcal{X}$ is a collection of subsets of $\mathcal{X}$, called hypotheses. For instance, the class of *intervals* over $\mathcal{X} = \mathbb{R}$ is the collection $\{[a, b] | a \leq b\}$. The class of *linear separators* over $\mathcal{X} = \mathbb{R}^d$ is the collection

$$\{\{\mathbf{x} \in \mathbb{R}^d | \mathbf{w} \cdot \mathbf{x} \geq w_0\} | \mathbf{w} \in \mathbb{R}^d, w_0 \in \mathbb{R}\};$$

that is, it is the collection of all sets in $\mathbb{R}^d$ that are linearly separable from their complement. In the case that $\mathcal{X}$ is the set of 4 points in the plane $\{(-1,-1), (-1,1), (1,-1), (1,1)\}$, the class of linear separators contains 14 of the $2^4 = 16$ possible subsets of $\mathcal{X}$.[17] Given an hypothesis class $\mathcal{H}$ and training set S, what we typically aim to do algorithmically is to find the hypothesis in $\mathcal{H}$ that most closely agrees with c^* over S. For example, we saw that the Perceptron algorithm will find a linear separator that agrees with the target function over S so long as S is linearly separable. To address overfitting, we argue that if S is large enough compared to some property of $\mathcal{H}$, then with high probability all $h \in \mathcal{H}$ have their training error close to their true error, so that if we find a hypothesis whose training error is low, we can be confident its true error will be low as well.

Before giving our first result of this form, we note that it will often be convenient to associate each hypotheses with its $\{-1, 1\}$-valued indicator function

$$h(x) = \begin{cases} 1 & x \in h \\ -1 & x \notin h \end{cases}$$

In this notation the true error of h is $err_D(h) = \text{Prob}_{x \sim D}[h(x) \neq c^*(x)]$ and the training error is $err_S(h) = \text{Prob}_{x \sim S}[h(x) \neq c^*(x)]$.

5.5 Overfitting and Uniform Convergence

We now present two generalization guarantees that explain how one can guard against overfitting. To keep things simple, we assume our hypothesis class $\mathcal{H}$ is *finite*. Later, we

[17] The only two subsets that are not in the class are the sets $\{(-1,-1), (1,1)\}$ and $\{(-1,1), (1,-1)\}$.

will see how to extend these results to infinite classes as well. Given a class of hypotheses $\mathcal{H}$, the first result states that for any given ϵ greater than zero, so long as the training data set is large compared to $\frac{1}{\epsilon}\ln(|\mathcal{H}|)$, it is unlikely any hypothesis $h \in \mathcal{H}$ will have zero training error but have true error greater than ϵ. This means that with high probability, any hypothesis that our algorithms finds that agrees with the target hypothesis on the training data will have low true error. The second result states that if the training data set is large compared to $\frac{1}{\epsilon^2}\ln(|\mathcal{H}|)$, then it is unlikely that the training error and true error will differ by more than ϵ for any hypothesis in $\mathcal{H}$. This means that if we find an hypothesis in $\mathcal{H}$ whose training error is low, we can be confident its true error will be low as well, even if its training error is not zero.

The basic idea is the following. If we consider some h with large true error, and we select an element $x \in \mathcal{X}$ at random according to D, there is a reasonable chance that x will belong to the symmetric difference $h \triangle c^*$. If we select a large enough training sample S with each point drawn independently from $\mathcal{X}$ according to D, the chance that S is completely disjoint from $h \triangle c^*$ will be incredibly small. This is just for a single hypothesis h but we can now apply the union bound over all $h \in \mathcal{H}$ of large true error, when $\mathcal{H}$ is finite. We formalize this below.

Theorem 5.3 *Let $\mathcal{H}$ be an hypothesis class and let ϵ and δ be greater than zero. If a training set S of size*

$$n \geq \frac{1}{\epsilon}\big(\ln|\mathcal{H}| + \ln(1/\delta)\big),$$

is drawn from distribution D, then with probability greater than or equal to $1-\delta$ every h in $\mathcal{H}$ with true error $err_D(h) \geq \epsilon$ has training error $err_S(h) > 0$. Equivalently, with probability greater than or equal to $1-\delta$, every $h \in \mathcal{H}$ with training error zero has true error less than ϵ.

Proof: Let $h_1, h_2, \ldots$ be the hypotheses in $\mathcal{H}$ with true error greater than or equal to ϵ. These are the hypotheses that we don't want to output. Consider drawing the sample S of size n and let A_i be the event that h_i is consistent with S. Since every h_i has true error greater than or equal to ϵ

$$\text{Prob}(A_i) \;\leq\; (1-\epsilon)^n.$$

In other words, if we fix h_i and draw a sample S of size n, the chance that h_i makes no mistakes on S is at most the probability that a coin of bias ϵ comes up tails n times in a row, which is $(1-\epsilon)^n$. By the union bound over all i we have

$$\text{Prob}\left(\cup_i A_i\right) \;\leq\; |\mathcal{H}|(1-\epsilon)^n.$$

Using the fact that $(1-\epsilon) \leq e^{-\epsilon}$, the probability that any hypothesis in $\mathcal{H}$ with true error greater than or equal to ϵ has training error zero is at most $|\mathcal{H}|e^{-\epsilon n}$. Replacing n by the sample size bound from the theorem statement, this is at most $|\mathcal{H}|e^{-\ln|\mathcal{H}|-\ln(1/\delta)} = \delta$ as desired. ■

$$\begin{array}{cccccccccccccccc l}
\multicolumn{8}{c}{\overbrace{\hphantom{x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8}}^{\text{Not spam}}} & \multicolumn{8}{c}{\overbrace{\hphantom{x_9\ x_{10}\ x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}}}^{\text{Spam}}} & \\
x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & \text{emails} \\
 & & & & & \downarrow & & & & \downarrow & & \downarrow & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \text{target concept} \\
 & & & & & \updownarrow & & & & \updownarrow & & \updownarrow & & & & & \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \text{hypothesis } h_i \\
 & & & & & \uparrow & & & & \uparrow & & \uparrow & & & & &
\end{array}$$

Figure 5.3: The hypothesis h_i disagrees with the truth in one quarter of the emails. Thus with a training set $|S|$, the probability that the hypothesis will survive is $(1 - 0.25)^{|S|}$

The conclusion of Theorem 5.3 is sometimes called a "PAC-learning guarantee" since it states that if we can find an $h \in \mathcal{H}$ consistent with the sample, then this h is *P*robably *A*pproximately *C*orrect.

Theorem 5.3 addressed the case where there exists a hypothesis in $\mathcal{H}$ with zero training error. What if the best h_i in $\mathcal{H}$ has 5% error on S? Can we still be confident that its true error is low, say at most 10%? For this, we want an analog of Theorem 5.3 that says for a sufficiently large training set S, every $h_i \in \mathcal{H}$ has training error within $\pm\epsilon$ of the true error with high probability. Such a statement is called *uniform convergence* because we are asking that the training set errors converge to their true errors uniformly over all sets in $\mathcal{H}$. To see intuitively why such a statement should be true for sufficiently large S and a single hypothesis h_i, consider two strings that differ in 10% of the positions and randomly select a large sample of positions. The number of positions that differ in the sample will be close to 10%.

To prove uniform convergence bounds, we use a tail inequality for sums of independent Bernoulli random variables (i.e., coin tosses). The following is particularly convenient and is a variation on the Chernoff bounds in Section 12.6.1 of the appendix.

Theorem 5.4 (Hoeffding bounds) *Let $x_1, x_2, \ldots, x_n$ be independent $\{0,1\}$-valued random variables with Prob$(x_i = 1) = p$. Let $s = \sum_i x_i$ (equivalently, flip n coins of bias p and let s be the total number of heads). For any $0 \le \alpha \le 1$,*

$$\begin{aligned}
Prob(s/n > p + \alpha) &\le e^{-2n\alpha^2} \\
Prob(s/n < p - \alpha) &\le e^{-2n\alpha^2}.
\end{aligned}$$

Theorem 5.4 implies the following uniform convergence analog of Theorem 5.3.

Theorem 5.5 (Uniform convergence) *Let $\mathcal{H}$ be a hypothesis class and let ϵ and δ be greater than zero. If a training set S of size*

$$n \ge \frac{1}{2\epsilon^2}\big(\ln|\mathcal{H}| + \ln(2/\delta)\big),$$

is drawn from distribution D, then with probability greater than or equal to $1-\delta$, every h in $\mathcal{H}$ satisfies $|err_S(h) - err_D(h)| \leq \epsilon$.

Proof: First, fix some $h \in \mathcal{H}$ and let x_j be the indicator random variable for the event that h makes a mistake on the j^{th} example in S. The x_j are independent $\{0,1\}$ random variables and the probability that x_i equals 1 is the true error of h, and the fraction of the x_j's equal to 1 is exactly the training error of h. Therefore, Hoeffding bounds guarantee that the probability of the event A_h that $|err_D(h) - err_S(h)| > \epsilon$ is less than or equal to $2e^{-2n\epsilon^2}$. Applying the union bound to the events A_h over all $h \in \mathcal{H}$, the probability that there *exists* an $h \in \mathcal{H}$ with the difference between true error and empirical error greater than ϵ is less than or equal to $2|\mathcal{H}|e^{-2n\epsilon^2}$. Using the value of n from the theorem statement, the right-hand-side of the above inequality is at most δ as desired. ■

Theorem 5.5 justifies the approach of optimizing over our training sample S even if we are not able to find a rule of zero training error. If our training set S is sufficiently large, with high probability, good performance on S will translate to good performance on D.

Note that Theorems 5.3 and 5.5 require $|\mathcal{H}|$ to be finite in order to be meaningful. The notion of growth functions and VC-dimension in Section 5.11 extend Theorem 5.5 to certain infinite hypothesis classes.

5.6 Illustrative Examples and Occam's Razor

We now present some examples to illustrate the use of Theorem 5.3 and 5.5 and also use these theorems to give a formal connection to the notion of Occam's razor.

5.6.1 Learning Disjunctions

Consider the instance space $\mathcal{X} = \{0,1\}^d$ and suppose we believe that the target concept can be represented by a *disjunction* (an OR) over features, such as $c^* = \{x | x_1 = 1 \vee x_4 = 1 \vee x_8 = 1\}$, or more succinctly, $c^* = x_1 \vee x_4 \vee x_8$. For example, if we are trying to predict whether an email message is spam or not, and our features correspond to the presence or absence of different possible indicators of spam-ness, then this would correspond to the belief that there is some subset of these indicators such that every spam email has at least one of them and every non-spam email has none of them. Formally, let $\mathcal{H}$ denote the class of disjunctions, and notice that $|\mathcal{H}| = 2^d$. So, by Theorem 5.3, it suffices to find a consistent disjunction over a sample S of size

$$|S| = \frac{1}{\epsilon}\big(d\ln(2) + \ln(1/\delta)\big).$$

How can we efficiently find a consistent disjunction when one exists? Here is a simple algorithm.

Simple Disjunction Learner: Given sample S, discard all features that are set to 1 in any negative example in S. Output the concept h that is the OR of all features that remain.

Lemma 5.6 *The Simple Disjunction Learner produces a disjunction h that is consistent with the sample S (i.e., with $err_S(h) = 0$) whenever the target concept is indeed a disjunction.*

Proof: Suppose target concept c^* is a disjunction. Then for any x_i that is listed in c^*, x_i will not be set to 1 in any negative example by definition of an OR. Therefore, h will include x_i as well. Since h contains all variables listed in c^*, this ensures that h will correctly predict positive on all positive examples in S. Furthermore, h will correctly predict negative on all negative examples in S since by design all features set to 1 in any negative example were discarded. Therefore, h is correct on all examples in S. ∎

Thus, combining Lemma 5.6 with Theorem 5.3, we have an efficient algorithm for PAC-learning the class of disjunctions.

5.6.2 Occam's Razor

Occam's razor is the notion, stated by William of Occam around AD 1320, that in general one should prefer simpler explanations over more complicated ones.[18] Why should one do this, and can we make a formal claim about why this is a good idea? What if each of us disagrees about precisely which explanations are simpler than others? It turns out we can use Theorem 5.3 to make a mathematical statement of Occam's razor that addresses these issues.

First, what do we mean by a rule being "simple"? Let's assume that each of us has some way of describing rules, using bits (since we are computer scientists). The methods, also called *description languages*, used by each of us may be different, but one fact we can say for certain is that in any given description language, there are at most 2^b rules that can be described using fewer than b bits (because $1 + 2 + 4 + \ldots + 2^{b-1} < 2^b$). Therefore, by setting $\mathcal{H}$ to be the set of all rules that can be described in fewer than b bits and plugging into Theorem 5.3, we have the following:

Theorem 5.7 (Occam's razor) *Fix any description language, and consider a training sample S drawn from distribution $\mathcal{D}$. With probability at least $1 - \delta$, any rule h with $err_S(h) = 0$ that can be described using fewer than b bits will have $err_D(h) \leq \epsilon$ for $|S| = \frac{1}{\epsilon}[b\ln(2) + \ln(1/\delta)]$. Equivalently, with probability at least $1 - \delta$, all rules with $err_S(h) = 0$ that can be described in fewer than b bits will have $err_D(h) \leq \frac{b\ln(2)+\ln(1/\delta)}{|S|}$.*

For example, using the fact that $\ln(2) < 1$ and ignoring the low-order $\ln(1/\delta)$ term, this means that if the number of bits it takes to write down a rule consistent with the training data is at most 10% of the number of data points in our sample, then we can be confident

[18] The statement more explicitly was that "Entities should not be multiplied unnecessarily."

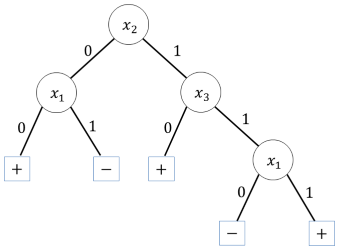


Figure 5.4: A decision tree with three internal nodes and four leaves. This tree corresponds to the Boolean function $\bar{x_1}\bar{x_2} \vee x_1x_2x_3 \vee x_2\bar{x}_3$.

it will have error at most 10% with respect to $\mathcal{D}$. What is perhaps surprising about this theorem is that it means that we can each have different ways of describing rules and yet all use Occam's razor. Note that the theorem does not say that complicated rules are necessarily bad, or even that given two rules consistent with the data that the complicated rule is necessarily worse. What it does say is that Occam's razor is a good policy in that simple rules are unlikely to fool us since there are just not that many simple rules.

5.6.3 Application: Learning Decision Trees

One popular practical method for machine learning is to learn a *decision tree*; see Figure 5.4. While finding the smallest decision tree that fits a given training sample S is NP-hard, there are a number of heuristics that are used in practice.[19] Suppose we run such a heuristic on a training set S and it outputs a tree with k nodes. Such a tree can be described using $O(k \log d)$ bits: $\log_2(d)$ bits to give the index of the feature in the root, $O(1)$ bits to indicate for each child if it is a leaf and if so what label it should have, and then $O(k_L \log d)$ and $O(k_R \log d)$ bits respectively to describe the left and right subtrees, where k_L is the number of nodes in the left subtree and k_R is the number of nodes in the right subtree. So, by Theorem 5.7, we can be confident the true error is low if we can produce a consistent tree with fewer than $\epsilon|S|/\log(d)$ nodes.

[19] For instance, one popular heuristic, called ID3, selects the feature to put inside any given node v by choosing the feature of largest *information gain*, a measure of how much it is directly improving prediction. Formally, using S_v to denote the set of examples in S that reach node v, and supposing that feature x_i partitions S_v into S_v^0 and S_v^1 (the examples in S_v with $x_i = 0$ and $x_i = 1$, respectively), the information gain of x_i is defined as: $Ent(S_v) - [\frac{|S_v^0|}{|S_v|}Ent(S_v^0) + \frac{|S_v^1|}{|S_v|}Ent(S_v^1)]$. Here, $Ent(S')$ is the binary entropy of the label proportions in set S'; that is, if a p fraction of the examples in S' are positive, then $Ent(S') = p\log_2(1/p) + (1-p)\log_2(1/(1-p))$, defining $0\log_2(0) = 0$. This then continues until all leaves are pure—they have only positive or only negative examples.

5.7 Regularization: Penalizing Complexity

Theorems 5.5 and 5.7 suggest the following idea. Suppose that there is no simple rule that is perfectly consistent with the training data, but we notice there are very simple rules with training error 20%, say, and then some more complex rules with training error 10%, and so on. In this case, perhaps we should optimize some combination of training error and simplicity. This is the notion of *regularization*, also called *complexity penalization*.

Specifically, a *regularizer* is a penalty term that penalizes more complex hypotheses. Given our theorems so far, a natural measure of complexity of a hypothesis is the number of bits we need to write it down.[20] Consider now fixing some description language, and let $\mathcal{H}_i$ denote those hypotheses that can be described in i bits in this language, so $|\mathcal{H}_i| \leq 2^i$. Let $\delta_i = \delta/2^i$. Rearranging the bound of Theorem 5.5, we know that with probability at least $1 - \delta_i$, all $h \in \mathcal{H}_i$ satisfy $err_D(h) \leq err_S(h) + \sqrt{\frac{\ln(|\mathcal{H}_i|)+\ln(2/\delta_i)}{2|S|}}$. Now, applying the union bound over all i, using the fact that $\delta_1 + \delta_2 + \delta_3 + \ldots = \delta$, and also the fact that $\ln(|\mathcal{H}_i|) + \ln(2/\delta_i) \leq i\ln(4) + \ln(2/\delta)$, gives the following corollary.

Corollary 5.8 *Fix any description language, and consider a training sample S drawn from distribution $\mathcal{D}$. With probability greater than or equal to $1 - \delta$, all hypotheses h satisfy*

$$err_D(h) \quad \leq \quad err_S(h) + \sqrt{\frac{\text{size}(h)\ln(4) + \ln(2/\delta)}{2|S|}}$$

where size(h) *denotes the number of bits needed to describe h in the given language.*

Corollary 5.8 gives us the tradeoff we were looking for. It tells us that rather than searching for a rule of low training error, we instead may want to search for a rule with a low right-hand-side in the displayed formula. If we can find one for which this quantity is small, we can be confident true error will be low as well.

5.8 Online Learning

So far we have been considering what is often called the *batch learning* scenario. You are given a "batch" of data—the training sample S—and your goal is to use it to produce a hypothesis h that will have low error on new data, under the assumption that both S and the new data are sampled from some fixed distribution D. We now switch to the more challenging *online learning* scenario where we remove the assumption that data is sampled from a fixed probability distribution, or from any probabilistic process at all.

Specifically, the online learning scenario proceeds as follows. At each time $t = 1, 2, \ldots$, two events occur:

[20] Later we will see support vector machines that use a regularizer for linear separators based on the margin of separation of data.

1. The algorithm is presented with an arbitrary example $x_t \in \mathcal{X}$ and is asked to make a prediction ℓ_t of its label.
2. The algorithm is told the true label of the example $c^*(x_t)$ and is charged for a mistake if $c^*(x_t) \neq \ell_t$.

The goal of the learning algorithm is to make as few mistakes as possible in total. For example, consider an email classifier that when a new email message arrives must classify it as "important" or "it can wait". The user then looks at the email and informs the algorithm if it was incorrect. We might not want to model email messages as independent random objects from a fixed probability distribution, because they often are replies to previous emails and build on each other. Thus, the online learning model would be more appropriate than the batch model for this setting.

Intuitively, the online learning model is harder than the batch model because we have removed the requirement that our data consists of independent draws from a fixed probability distribution. Indeed, we will see shortly that any algorithm with good performance in the online model can be converted to an algorithm with good performance in the batch model. Nonetheless, the online model can sometimes be a cleaner model for design and analysis of algorithms.

5.8.1 An Example: Learning Disjunctions

As a simple example, let's revisit the problem of learning disjunctions in the online model. We can solve this problem by starting with a hypothesis $h = x_1 \vee x_2 \vee \ldots \vee x_d$ and using it for prediction. We will maintain the invariant that every variable in the target disjunction is also in our hypothesis, which is clearly true at the start. This ensures that the only mistakes possible are on examples x for which $h(x)$ is positive but $c^*(x)$ is negative. When such a mistake occurs, we simply remove from h any variable set to 1 in x. Since such variables cannot be in the target function (since x was negative), we maintain our invariant *and* remove at least one variable from h. This implies that the algorithm makes at most d mistakes total on any series of examples consistent with a disjunction.

In fact, we can show this bound is tight by showing that no deterministic algorithm can guarantee to make fewer than d mistakes.

Theorem 5.9 *For any deterministic algorithm A there exists a sequence of examples σ and disjunction c^* such that A makes at least d mistakes on sequence σ labeled by c^*.*

Proof: Let σ be the sequence $e_1, e_2, \ldots, e_d$ where e_j is the example that is zero everywhere except for a 1 in the jth position. Imagine running A on sequence σ and telling A it made a mistake on every example; that is, if A predicts positive on e_j we set $c^*(e_j) = -1$ and if A predicts negative on e_j we set $c^*(e_j) = +1$. This target corresponds to the disjunction of all x_j such that A predicted negative on e_j, so it is a legal disjunction. Since A is deterministic, the fact that we constructed c^* by running A is not a problem: it would make the same mistakes if re-run from scratch on the same sequence and same target. Therefore, A makes d mistakes on this σ and c^*. ∎

5.8.2 The Halving Algorithm

If we are not concerned with running time, a simple algorithm that guarantees to make at most $\log_2(|\mathcal{H}|)$ mistakes for a target belonging to any given class $\mathcal{H}$ is called the *halving algorithm.* This algorithm simply maintains the *version space* $\mathcal{V} \subseteq \mathcal{H}$ consisting of all $h \in \mathcal{H}$ consistent with the labels on every example seen so far, and predicts based on majority vote over these functions. Each mistake is guaranteed to reduce the size of the version space $\mathcal{V}$ by at least half (hence the name), thus the total number of mistakes is at most $\log_2(|\mathcal{H}|)$. Note that this can be viewed as the number of bits needed to write a function in $\mathcal{H}$ down.

5.8.3 The Perceptron Algorithm

Earlier we described the Perceptron algorithm as a method for finding a linear separator consistent with a given training set S. However, the Perceptron algorithm also operates naturally in the online setting as well.

Recall that the basic assumption of the Perceptron algorithm is that the target function can be described by a vector $\mathbf{w}^*$ such that for each positive example $\mathbf{x}$ we have $\mathbf{x}^T\mathbf{w}^* \geq 1$ and for each negative example $\mathbf{x}$ we have $\mathbf{x}^T\mathbf{w}^* \leq -1$. Recall also that we can interpret $\mathbf{x}^T\mathbf{w}^*/|\mathbf{w}^*|$ as the distance of $\mathbf{x}$ to the hyperplane $\mathbf{x}^T\mathbf{w}^* = 0$. Thus, we can view our assumption as stating that there exists a linear separator through the origin with all positive examples on one side, all negative examples on the other side, and all examples at distance at least $\gamma = 1/|\mathbf{w}^*|$ from the separator, where γ is called the margin of separation.

The guarantee of the Perceptron algorithm will be that the total number of mistakes is at most $(R/\gamma)^2$ where $R = \max_t |\mathbf{x}_t|$ over all examples $\mathbf{x}_t$ seen so far. Thus, if there exists a hyperplane through the origin that correctly separates the positive examples from the negative examples by a large margin relative to the radius of the smallest ball enclosing the data, then the total number of mistakes will be small. The algorithm, restated in the online setting, is as follows.

The Perceptron Algorithm: Start with the all-zeroes weight vector $\mathbf{w} = \mathbf{0}$. Then, for $t = 1, 2, \ldots$ do:

1. Given example $\mathbf{x}_t$, predict $\operatorname{sgn}(\mathbf{x}_t^T \mathbf{w})$.
2. If the prediction was a mistake, then update:
   (a) If $\mathbf{x}_t$ was a positive example, let $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}_t$.
   (b) If $\mathbf{x}_t$ was a negative example, let $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}_t$.

The Perceptron algorithm enjoys the following guarantee on its total number of mistakes.

Theorem 5.10 *On any sequence of examples $\mathbf{x}_1, \mathbf{x}_2, \ldots$, if there exists a vector $\mathbf{w}^*$ such that $\mathbf{x}_t^T \mathbf{w}^* \geq 1$ for the positive examples and $\mathbf{x}_t^T \mathbf{w}^* \leq -1$ for the negative examples (i.e., a linear separator of margin $\gamma = 1/|\mathbf{w}^*|$), then the Perceptron algorithm makes at most $R^2|\mathbf{w}^*|^2$ mistakes, where $R = \max_t |\mathbf{x}_t|$.*

Proof: Fix some consistent $\mathbf{w}^*$. We will keep track of two quantities, $\mathbf{w}^T\mathbf{w}^*$ and $|\mathbf{w}|^2$. First of all, each time we make a mistake, $\mathbf{w}^T\mathbf{w}^*$ increases by at least 1. That is because if $\mathbf{x}_t$ is a positive example, then

$$(\mathbf{w} + \mathbf{x}_t)^T \mathbf{w}^* = \mathbf{w}^T \mathbf{w}^* + \mathbf{x}_t^T \mathbf{w}^* \geq \mathbf{w}^T \mathbf{w}^* + 1,$$

by definition of $\mathbf{w}^*$. Similarly, if $\mathbf{x}_t$ is a negative example, then

$$(\mathbf{w} - \mathbf{x}_t)^T \mathbf{w}^* = \mathbf{w}^T \mathbf{w}^* - \mathbf{x}_t^T \mathbf{w}^* \geq \mathbf{w}^T \mathbf{w}^* + 1.$$

Next, on each mistake, we claim that $|\mathbf{w}|^2$ increases by at most R^2. Let us first consider mistakes on positive examples. If we make a mistake on a positive example $\mathbf{x}_t$ then we have

$$(\mathbf{w} + \mathbf{x}_t)^T(\mathbf{w} + \mathbf{x}_t) = |\mathbf{w}|^2 + 2\mathbf{x}_t^T\mathbf{w} + |\mathbf{x}_t|^2 \leq |\mathbf{w}|^2 + |\mathbf{x}_t|^2 \leq |\mathbf{w}|^2 + R^2,$$

where the middle inequality comes from the fact that we made a mistake, which means that $\mathbf{x}_t^T\mathbf{w} \leq 0$. Similarly, if we make a mistake on a negative example $\mathbf{x}_t$ then we have

$$(\mathbf{w} - \mathbf{x}_t)^T(\mathbf{w} - \mathbf{x}_t) = |\mathbf{w}|^2 - 2\mathbf{x}_t^T\mathbf{w} + |\mathbf{x}_t|^2 \leq |\mathbf{w}|^2 + |\mathbf{x}_t|^2 \leq |\mathbf{w}|^2 + R^2.$$

Note that it is important here that we only update on a mistake.

So, if we make M mistakes, then $\mathbf{w}^T\mathbf{w}^* \geq M$, and $|\mathbf{w}|^2 \leq MR^2$, or equivalently, $|\mathbf{w}| \leq R\sqrt{M}$. Finally, we use the fact that $\mathbf{w}^T\mathbf{w}^*/|\mathbf{w}^*| \leq |\mathbf{w}|$ which is just saying that the projection of $\mathbf{w}$ in the direction of $\mathbf{w}^*$ cannot be larger than the length of $\mathbf{w}$. This gives us:

$$\begin{aligned} M/|\mathbf{w}^*| &\leq R\sqrt{M} \\ \sqrt{M} &\leq R|\mathbf{w}^*| \\ M &\leq R^2|\mathbf{w}^*|^2 \end{aligned}$$

as desired. ■

5.8.4 Extensions: Inseparable Data and Hinge Loss

We assumed above that there exists a perfect $\mathbf{w}^*$ that correctly classifies all the examples, e.g., correctly classifies all the emails into important versus non-important. This is rarely the case in real-life data. What if even the best $\mathbf{w}^*$ isn't quite perfect? We can see what this does to the above proof: if there is an example that $\mathbf{w}^*$ doesn't correctly classify, then while the second part of the proof still holds, the first part (the dot product of $\mathbf{w}$ with $\mathbf{w}^*$ increasing) breaks down. However, if this doesn't happen too often, and also $\mathbf{x}_t^T\mathbf{w}^*$ is just a "little bit wrong" then we will only make a few more mistakes.

To make this formal, define the *hinge-loss* of $\mathbf{w}^*$ on a positive example $\mathbf{x}_t$ as $\max(0, 1-\mathbf{x}_t^T\mathbf{w}^*)$. In other words, if $\mathbf{x}_t^T\mathbf{w}^* \geq 1$ as desired then the hinge-loss is zero; else, the hinge-loss is the amount the LHS is less than the RHS.[21] Similarly, the hinge-loss of $\mathbf{w}^*$ on a negative example $\mathbf{x}_t$ is $\max(0, 1+\mathbf{x}_t^T\mathbf{w}^*)$. Given a sequence of labeled examples S, define the total hinge-loss $L_{hinge}(\mathbf{w}^*, S)$ as the sum of hinge-losses of $\mathbf{w}^*$ on all examples in S. We now get the following extended theorem.

Theorem 5.11 *On any sequence of examples $S = \mathbf{x}_1, \mathbf{x}_2, \ldots$, the Perceptron algorithm makes at most*

$$\min_{\mathbf{w}^*}\left(R^2|\mathbf{w}^*|^2 + 2L_{hinge}(\mathbf{w}^*, S)\right)$$

mistakes, where $R = \max_t |\mathbf{x}_t|$.

Proof: As before, each update of the Perceptron algorithm increases $|\mathbf{w}|^2$ by at most R^2, so if the algorithm makes M mistakes, we have $|\mathbf{w}|^2 \leq MR^2$.

What we can no longer say is that each update of the algorithm increases $\mathbf{w}^T\mathbf{w}^*$ by at least 1. Instead, on a positive example we are "increasing" $\mathbf{w}^T\mathbf{w}^*$ by $\mathbf{x}_t^T\mathbf{w}^*$ (it could be negative), which is at least $1 - L_{hinge}(\mathbf{w}^*, \mathbf{x}_t)$. Similarly, on a negative example we "increase" $\mathbf{w}^T\mathbf{w}^*$ by $-\mathbf{x}_t^T\mathbf{w}^*$, which is also at least $1 - L_{hinge}(\mathbf{w}^*, \mathbf{x}_t)$. If we sum this up over all mistakes, we get that at the end we have $\mathbf{w}^T\mathbf{w}^* \geq M - L_{hinge}(\mathbf{w}^*, S)$, where we are using here the fact that hinge-loss is never negative so summing over all of S is only larger than summing over the mistakes that $\mathbf{w}$ made.

[21] This is called "hinge-loss" because as a function of $\mathbf{x}_t^T\mathbf{w}^*$ it looks like a hinge.

Finally, we just do some algebra. Let $L = L_{hinge}(\mathbf{w}^*, S)$. So we have:

$$\begin{array}{rcl} \mathbf{w}^T\mathbf{w}^*/|\mathbf{w}^*| & \leq & |\mathbf{w}| \\ (\mathbf{w}^T\mathbf{w}^*)^2 & \leq & |\mathbf{w}|^2|\mathbf{w}^*|^2 \\ (M-L)^2 & \leq & MR^2|\mathbf{w}^*|^2 \\ M^2 - 2ML + L^2 & \leq & MR^2|\mathbf{w}^*|^2 \\ M - 2L + L^2/M & \leq & R^2|\mathbf{w}^*|^2 \\ M & \leq & R^2|\mathbf{w}^*|^2 + 2L - L^2/M \ \leq \ R^2|\mathbf{w}^*|^2 + 2L \end{array}$$

as desired. ■

5.9 Online to Batch Conversion

Suppose we have an online algorithm with a good mistake bound, such as the Perceptron algorithm. Can we use it to get a guarantee in the distributional (batch) learning setting? Intuitively, the answer should be yes since the online setting is only harder. Indeed, this intuition is correct. We present here two natural approaches for such online to batch conversion.

Conversion procedure 1: Random Stopping. Suppose we have an online algorithm $\mathcal{A}$ with mistake-bound M. Say we run the algorithm in a single pass on a sample S of size M/ϵ. Let X_t be the indicator random variable for the event that $\mathcal{A}$ makes a mistake on example $\mathbf{x}_t$. Since $\sum_{t=1}^{|S|} X_t \leq M$ for *any* set S, we certainly have that $\mathbf{E}[\sum_{t=1}^{|S|} X_t] \leq M$ where the expectation is taken over the random draw of S from $\mathcal{D}^{|S|}$. By linearity of expectation, and dividing both sides by $|S|$ we therefore have:

$$\frac{1}{|S|}\sum_{t=1}^{|S|} \mathbf{E}[X_t] \ \leq \ M/|S| = \epsilon. \qquad (5.1)$$

Let h_t denote the hypothesis used by algorithm $\mathcal{A}$ to predict on the tth example. Since the tth example was randomly drawn from $\mathcal{D}$, we have $\mathbf{E}[err_{\mathcal{D}}(h_t)] = \mathbf{E}[X_t]$. This means that if we choose t at random from 1 to $|S|$, i.e., stop the algorithm at a random time, the expected error of the resulting prediction rule, taken over the randomness in the draw of S and the choice of t, is at most ϵ as given by equation (5.1). Thus we have:

Theorem 5.12 (Online to Batch via Random Stopping) *If an online algorithm $\mathcal{A}$ with mistake-bound M is run on a sample S of size M/ϵ and stopped at a random time between 1 and $|S|$, the expected error of the hypothesis h produced satisfies $\mathbf{E}[err_{\mathcal{D}}(h)] \leq \epsilon$.*

Conversion procedure 2: Controlled Testing. A second natural approach to using an online learning algorithm $\mathcal{A}$ in the distributional setting is to just run a series of controlled tests. Specifically, suppose that the initial hypothesis produced by algorithm $\mathcal{A}$ is h_1. Define $\delta_i = \delta/(i+2)^2$ so we have $\sum_{i=0}^{\infty} \delta_i = (\frac{\pi^2}{6} - 1)\delta \leq \delta$. We draw a set of

$n_1 = \frac{1}{\epsilon}\log(\frac{1}{\delta_1})$ random examples and test to see whether h_1 gets all of them correct. Note that if $err_D(h_1) \geq \epsilon$ then the chance h_1 would get them all correct is at most $(1-\epsilon)^{n_1} \leq \delta_1$. So, if h_1 indeed gets them all correct, we output h_1 as our hypothesis and halt. If not, we choose some example x_1 in the sample on which h_1 made a mistake and give it to algorithm $\mathcal{A}$. Algorithm $\mathcal{A}$ then produces some new hypothesis h_2 and we again repeat, testing h_2 on a fresh set of $n_2 = \frac{1}{\epsilon}\log(\frac{1}{\delta_2})$ random examples, and so on.

In general, given h_t we draw a fresh set of $n_t = \frac{1}{\epsilon}\log(\frac{1}{\delta_t})$ random examples and test to see whether h_t gets all of them correct. If so, we output h_t and halt; if not, we choose some x_t on which $h_t(x_t)$ was incorrect and give it to algorithm $\mathcal{A}$. By choice of n_t, if h_t had error rate ϵ or larger, the chance we would mistakenly output it is at most δ_t. By choice of the values δ_t, the chance we *ever* halt with a hypothesis of error ϵ or larger is at most $\delta_1 + \delta_2 + \ldots \leq \delta$. Thus, we have the following theorem.

Theorem 5.13 (Online to Batch via Controlled Testing) *Let $\mathcal{A}$ be an online learning algorithm with mistake-bound M. Then this procedure will halt after $O(\frac{M}{\epsilon}\log(\frac{M}{\delta}))$ examples and with probability at least $1-\delta$ will produce a hypothesis of error at most ϵ.*

Note that in this conversion we cannot re-use our samples: since the hypothesis h_t depends on the previous data, we need to draw a fresh set of n_t examples to use for testing it.

5.10 Support-Vector Machines

In a batch setting, rather than running the Perceptron algorithm and adapting it via one of the methods above, another natural idea would be just to solve for the vector $\mathbf{w}$ that minimizes the right-hand-side in Theorem 5.11 on the given dataset S. This turns out to have good guarantees as well, though they are beyond the scope of this book. In fact, this is the Support Vector Machine (SVM) algorithm. Specifically, SVMs solve the following convex optimization problem over a sample $S = \{\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_n\}$ where c is a constant that is determined empirically.

$$\begin{array}{ll} \text{minimize} & c|\mathbf{w}|^2 + \sum_i s_i \\ \text{subject to} & \mathbf{w}\cdot\mathbf{x}_i \geq 1 - s_i \text{ for all positive examples } \mathbf{x}_i \\ & \mathbf{w}\cdot\mathbf{x}_i \leq -1 + s_i \text{ for all negative examples } \mathbf{x}_i \\ & s_i \geq 0 \text{ for all } i. \end{array}$$

The variables s_i are called *slack variables*, and notice that the sum of the slack variables is the total hinge loss of $\mathbf{w}$. So, this convex optimization is minimizing a weighted sum of $1/\gamma^2$, where γ is the margin, and the total hinge loss. If we were to add the constraint that all $s_i = 0$ then this would be solving for the maximum margin linear separator for the data. However, in practice, optimizing a weighted combination generally performs better. SVMs can also be kernelized, by using the dual of the above optimization problem (the key idea is that the optimal $\mathbf{w}$ will be a weighted combination of data points, just as in the Perceptron algorithm, and these weights can be variables in the optimization problem); details are beyond the scope of this book.

5.11 VC-Dimension

In Section 5.5 we presented several theorems showing that so long as the training set S is large compared to $\frac{1}{\epsilon}\log(|\mathcal{H}|)$, we can be confident that every $h \in \mathcal{H}$ with $err_D(h) \geq \epsilon$ will have $err_S(h) > 0$, and if S is large compared to $\frac{1}{\epsilon^2}\log(|\mathcal{H}|)$, then we can be confident that every $h \in \mathcal{H}$ will have $|err_D(h) - err_S(h)| \leq \epsilon$. In essence, these results used $\log(|\mathcal{H}|)$ as a measure of complexity of class $\mathcal{H}$. VC-dimension is a different, tighter measure of complexity for a concept class, and as we will see, is also sufficient to yield confidence bounds. For any class $\mathcal{H}$, $\text{VCdim}(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$ but it can also be quite a bit smaller. Let's introduce and motivate it through an example.

Consider a database consisting of the salary and age for a random sample of the adult population in the United States. Suppose we are interested in using the database to answer questions of the form: "what fraction of the adult population in the United States has age between 35 and 45 and salary between \$50,000 and \$70,000?" That is, we are interested in queries that ask about the fraction of the adult population within some axis-parallel rectangle. What we can do is calculate the fraction of the database satisfying this condition and return this as our answer. This brings up the following question: How large does our database need to be so that with probability greater than or equal to $1-\delta$, our answer will be within $\pm\epsilon$ of the truth for *every* possible rectangle query of this form?

If we assume our values are discretized such as 100 possible ages and 1,000 possible salaries, then there are at most $(100 \times 1,000)^2 = 10^{10}$ possible rectangles. This means we can apply Theorem 5.5 with $|\mathcal{H}| \leq 10^{10}$. Specifically, we can think of the target concept c^* as the empty set so that $err_S(h)$ is exactly the fraction of the sample inside rectangle h and $err_D(h)$ is exactly the fraction of the whole population inside h.[22] This would tell us that a sample size of $\frac{1}{2\epsilon^2}(10\ln 10 + \ln(2/\delta))$ would be sufficient.

However, what if we do not wish to discretize our concept class? Another approach would be to say that if there are only N adults total in the United States, then there are at most N^4 rectangles that are truly different with respect to D and so we could use $|\mathcal{H}| \leq N^4$. Still, this suggests that S needs to grow with N, albeit logarithmically, and one might wonder if that is really necessary. VC-dimension, and the notion of the *growth function* of concept class $\mathcal{H}$, will give us a way to avoid such discretization and avoid any dependence on the size of the support of the underlying distribution D.

[22] Technically D is the uniform distribution over the adult population of the United States, and we want to think of S as an independent identically distributed sample from this D.

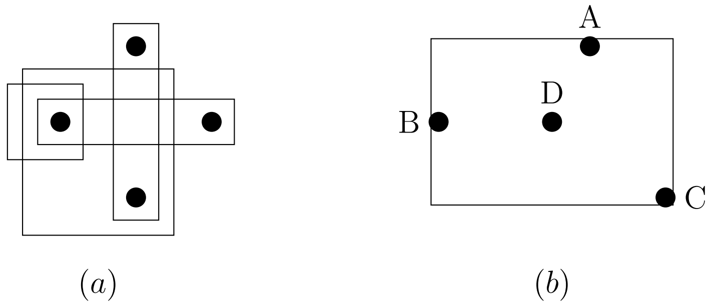


Figure 5.5: (a) shows a set of four points that can be shattered by rectangles along with some of the rectangles that shatter the set. Not every set of four points can be shattered as seen in (b). Any rectangle containing points A, B, and C must contain D. No set of five points can be shattered by rectangles with axis-parallel edges. No set of three collinear points can be shattered, since any rectangle that contains the two end points must also contain the middle point. More generally, since rectangles are convex, a set with one point inside the convex hull of the others cannot be shattered.

5.11.1 Definitions and Key Theorems

Definition 5.1 *Given a set S of examples and a concept class $\mathcal{H}$, we say that S is* **shattered** *by $\mathcal{H}$ if for every $A \subseteq S$ there exists some $h \in \mathcal{H}$ that labels all examples in A as positive and all examples in $S \setminus A$ as negative.*

Definition 5.2 *The* **VC-dimension** *of $\mathcal{H}$ is the size of the largest set shattered by $\mathcal{H}$.*

For example, there exist sets of four points in the plane that can be shattered by rectangles with axis-parallel edges, e.g., four points at the vertices of a diamond (see Figure 5.5). Given such a set S, for any $A \subseteq S$, there exists a rectangle with the points in A inside the rectangle and the points in $S \setminus A$ outside the rectangle. However, rectangles with axis-parallel edges cannot shatter any set of five points. To see this, assume for contradiction that there is a set of five points shattered by the family of axis-parallel rectangles. Find the minimum enclosing rectangle for the five points. For each edge there is at least one point that has stopped its movement. Identify one such point for each edge. The same point may be identified as stopping two edges if it is at a corner of the minimum enclosing rectangle. If two or more points have stopped an edge, designate only one as having stopped the edge. Now, at most four points have been designated. Any rectangle enclosing the designated points must include the undesignated points. Thus, the subset of designated points cannot be expressed as the intersection of a rectangle with the five points. Therefore, the VC-dimension of axis-parallel rectangles is four.

We now need one more definition, which is the *growth function* of a concept class $\mathcal{H}$.

Definition 5.3 *Given a set S of examples and a concept class $\mathcal{H}$, let $\mathcal{H}[S] = \{h \cap S :$*

$h \in \mathcal{H}\}$. That is, $\mathcal{H}[S]$ is the concept class $\mathcal{H}$ restricted to the set of points S. For integer n and class $\mathcal{H}$, let $\mathcal{H}[n] = \max_{|S|=n} |\mathcal{H}[S]|$; this is called the **growth function** *of $\mathcal{H}$.*

For example, we could have defined shattering by saying that S is shattered by $\mathcal{H}$ if $|\mathcal{H}[S]| = 2^{|S|}$, and then the VC-dimension of $\mathcal{H}$ is the largest n such that $\mathcal{H}[n] = 2^n$. Notice also that for axis-parallel rectangles, $\mathcal{H}[n] = O(n^4)$. The growth function of a class is sometimes called the shatter function or shatter coefficient.

What connects these to learnability are the following three remarkable theorems. The first two are analogs of Theorem 5.3 and Theorem 5.5 respectively, showing that one can replace $|\mathcal{H}|$ with its growth function. This is like replacing the number of concepts in $\mathcal{H}$ with the number of concepts "after the fact", i.e., after S is drawn, and is subtle because we cannot just use a union bound after we have already drawn our set S. The third theorem relates the growth function of a class to its VC-dimension. We now present the theorems, give examples of VC-dimension and growth function of various concept classes, and then prove the theorems.

Theorem 5.14 (Growth function sample bound) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, if a training sample S is drawn from $\mathcal{D}$ of size*

$$n \geq \frac{2}{\epsilon}[\log_2(2\mathcal{H}[2n]) + \log_2(1/\delta)]$$

then with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$ (equivalently, every $h \in \mathcal{H}$ with $err_S(h) = 0$ has $err_{\mathcal{D}}(h) < \epsilon$).

Theorem 5.15 (Growth function uniform convergence) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, if a training sample S is drawn from $\mathcal{D}$ of size*

$$n \geq \frac{8}{\epsilon^2}[\ln(2\mathcal{H}[2n]) + \ln(1/\delta)]$$

then with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ will have $|err_S(h) - err_{\mathcal{D}}(h)| \leq \epsilon$.

Theorem 5.16 (Sauer's lemma) *If $\text{VCdim}(\mathcal{H}) = d$ then $\mathcal{H}[n] \leq \sum_{i=0}^{d} \binom{n}{i} \leq (\frac{en}{d})^d$.*

Notice that Sauer's lemma was fairly tight in the case of axis-parallel rectangles, though in some cases it can be a bit loose. E.g., we will see that for linear separators in the plane, their VC-dimension is 3 but $\mathcal{H}[n] = O(n^2)$. An interesting feature about Sauer's lemma is that it implies the growth function switches from taking the form 2^n to taking the form of roughly $n^{\text{VCdim}(\mathcal{H})}$ when n exceeds the VC-dimension of the class $\mathcal{H}$.

Putting Theorems 5.14 and 5.16 together, with a little algebra we get the following corollary (a similar corollary results by combining Theorems 5.15 and 5.16):

Corollary 5.17 (VC-dimension sample bound) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, a training sample S of size*

$$O\left(\frac{1}{\epsilon}[\text{VCdim}(\mathcal{H})\log(1/\epsilon)+\log(1/\delta)]\right)$$

is sufficient to ensure that with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$ (equivalently, every $h \in \mathcal{H}$ with $err_S(h) = 0$ has $err_{\mathcal{D}}(h) < \epsilon$).

For any class $\mathcal{H}$, $\text{VCdim}(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$ since $\mathcal{H}$ must have at least 2^k concepts in order to shatter k points. Thus Corollary 5.17 is never too much worse than Theorem 5.3 and can be much better.

5.11.2 Examples: VC-Dimension and Growth Function

Rectangles with axis-parallel edges

As we saw above, the class of axis-parallel rectangles in the plane has VC-dimension 4 and growth function $\mathcal{H}[n] = O(n^4)$.

Intervals of the reals

Intervals on the real line can shatter any set of two points but no set of three points since the subset of the first and last points cannot be isolated. Thus, the VC-dimension of intervals is two. Also, $\mathcal{H}[n] = O(n^2)$ since we have $O(n^2)$ choices for the left and right endpoints.

Pairs of intervals of the reals

Consider the family of pairs of intervals, where a pair of intervals is viewed as the set of points that are in at least one of the intervals, in other words, their set union. There exists a set of size four that can be shattered but no set of size five since the subset of first, third, and last point cannot be isolated. Thus, the VC-dimension of pairs of intervals is four. Also we have $\mathcal{H}[n] = O(n^4)$.

Convex polygons

Consider the set system of all convex polygons in the plane. For any positive integer n, place n points on the unit circle. Any subset of the points are the vertices of a convex polygon. Clearly that polygon will not contain any of the points not in the subset. This shows that convex polygons can shatter arbitrarily large sets, so the VC-dimension is infinite. Notice that this also implies that $\mathcal{H}[n] = 2^n$.

Halfspaces in d-dimensions

Define a halfspace to be the set of all points on one side of a linear separator, i.e., a set of the form $\{\mathbf{x}|\mathbf{w}^T\mathbf{x} \geq w_0\}$. The VC-dimension of halfspaces in d-dimensions is $d+1$.

There exists a set of size $d+1$ that can be shattered by halfspaces. Select the d unit-coordinate vectors plus the origin to be the $d+1$ points. Suppose A is any subset of these $d+1$ points. Without loss of generality assume that the origin is in A. Take a 0-1 vector $\mathbf{w}$ which has 1's precisely in the coordinates corresponding to vectors not in A. Clearly A lies in the half-space $\mathbf{w}^T\mathbf{x} \leq 0$ and the complement of A lies in the complementary halfspace.

We now show that no set of $d+2$ points in d-dimensions can be shattered by halfspaces. This is done by proving that any set of $d+2$ points can be partitioned into two disjoint subsets A and B of points whose convex hulls intersect. This establishes the claim since any linear separator with A on one side must have its entire convex hull on that side,[23] so it is not possible to have a linear separator with A on one side and B on the other.

Let $convex(S)$ denote the convex hull of point set S.

Theorem 5.18 (**Radon**)*: Any set $S \subseteq R^d$ with $|S| \geq d+2$, can be partitioned into two disjoint subsets A and B such that $convex(A) \cap convex(B) \neq \phi$.*

Proof: Without loss of generality, assume $|S| = d+2$. Form a $d \times (d+2)$ matrix with one column for each point of S. Call the matrix A. Add an extra row of all 1's to construct a $(d+1) \times (d+2)$ matrix B. Clearly the rank of this matrix is at most $d+1$ and the columns are linearly dependent. Say $\mathbf{x} = (x_1, x_2, \ldots, x_{d+2})$ is a nonzero vector with $B\mathbf{x} = 0$. Reorder the columns so that $x_1, x_2, \ldots, x_s \geq 0$ and $x_{s+1}, x_{s+2}, \ldots, x_{d+2} < 0$. Normalize $\mathbf{x}$ so $\sum\limits_{i=1}^{s} |x_i| = 1$. Let $\mathbf{b_i}$ (respectively $\mathbf{a_i}$) be the i^{th} column of B (respectively A). Then, $\sum\limits_{i=1}^{s} |x_i|\mathbf{b_i} = \sum\limits_{i=s+1}^{d+2} |x_i|\mathbf{b_i}$ from which it follows that $\sum\limits_{i=1}^{s} |x_i|\mathbf{a_i} = \sum\limits_{i=s+1}^{d+2} |x_i|\mathbf{a_i}$ and $\sum\limits_{i=1}^{s} |x_i| = \sum\limits_{i=s+1}^{d+2} |x_i|$. Since $\sum\limits_{i=1}^{s} |x_i| = 1$ and $\sum\limits_{i=s+1}^{d+2} |x_i| = 1$ each side of $\sum\limits_{i=1}^{s} |x_i|\mathbf{a_i} = \sum\limits_{i=s+1}^{d+2} |x_i|\mathbf{a_i}$ is a convex combination of columns of A which proves the theorem. Thus, S can be partitioned into two sets, the first consisting of the first s points after the rearrangement and the second consisting of points $s+1$ through $d+2$. Their convex hulls intersect as required. ■

Radon's theorem immediately implies that half-spaces in d-dimensions do not shatter any set of $d+2$ points.

Spheres in d-dimensions

[23] If any two points $\mathbf{x}_1$ and $\mathbf{x}_2$ lie on the same side of a linear separator, so must any convex combination: if $\mathbf{w} \cdot \mathbf{x}_1 \geq b$ and $\mathbf{w} \cdot \mathbf{x}_2 \geq b$ then $\mathbf{w} \cdot (a\mathbf{x}_1 + (1-a)\mathbf{x}_2) \geq b$.

A *sphere* in d-dimensions is a set of points of the form $\{\mathbf{x} |\ |\mathbf{x} - \mathbf{x_0}| \leq r\}$. The VC-dimension of spheres is $d+1$. It is the same as that of halfspaces. First, we prove that no set of $d+2$ points can be shattered by spheres. Suppose some set S with $d+2$ points can be shattered. Then for any partition A_1 and A_2 of S, there are spheres B_1 and B_2 such that $B_1 \cap S = A_1$ and $B_2 \cap S = A_2$. Now B_1 and B_2 may intersect, but there is no point of S in their intersection. It is easy to see that there is a hyperplane perpendicular to the line joining the centers of the two spheres with all of A_1 on one side and all of A_2 on the other and this implies that halfspaces shatter S, a contradiction. Therefore no $d+2$ points can be shattered by hyperspheres.

It is also not difficult to see that the set of $d+1$ points consisting of the unit-coordinate vectors and the origin can be shattered by spheres. Suppose A is a subset of the $d+1$ points. Let a be the number of unit vectors in A. The center $\mathbf{a_0}$ of our sphere will be the sum of the vectors in A. For every unit vector in A, its distance to this center will be $\sqrt{a-1}$ and for every unit vector outside A, its distance to this center will be $\sqrt{a+1}$. The distance of the origin to the center is $\sqrt{a}$. Thus, we can choose the radius so that precisely the points in A are in the hypersphere.

Finite sets

The system of finite sets of real numbers can shatter any finite set of real numbers and thus the VC-dimension of finite sets is infinite.

5.11.3 Proof of Main Theorems

We begin with a technical lemma. Consider drawing a set S of n examples from $\mathcal{D}$ and let A denote the event that there exists $h \in \mathcal{H}$ with zero training error on S but true error greater than or equal to ϵ. Now draw a second set S' of n examples from $\mathcal{D}$ and let B denote the event that there exists $h \in \mathcal{H}$ with zero error on S but error greater than or equal to $\epsilon/2$ on S'. We claim that $\text{Prob}(B) \geq \text{Prob}(A)/2$.

Lemma 5.19 *Let $\mathcal{H}$ be a concept class over some domain $\mathcal{X}$ and let S and S' be sets of n elements drawn from some distribution $\mathcal{D}$ on $\mathcal{X}$, where $n \geq 8/\epsilon$. Let A be the event that there exists $h \in \mathcal{H}$ with zero error on S but true error greater than or equal to ϵ. Let B be the event that there exists $h \in \mathcal{H}$ with zero error on S but error greater than or equal to $\frac{\epsilon}{2}$ on S'. Then Prob$(B) \geq$ Prob$(A)/2$.*

Proof: Clearly, $\text{Prob}(B) \geq \text{Prob}(A, B) = \text{Prob}(A)\text{Prob}(B|A)$. Consider drawing set S and suppose event A occurs. Let h be in $\mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ but $err_S(h) = 0$. Now, draw set S'. $\mathbf{E}(\text{error of } h \text{ on } S') = err_{\mathcal{D}}(h) \geq \epsilon$. So, by Chernoff bounds, since $n \geq 8/\epsilon$, $\text{Prob}(err_{S'}(h) \geq \epsilon/2) \geq 1/2$. Thus, $\text{Prob}(B|A) \geq 1/2$ and $\text{Prob}(B) \geq \text{Prob}(A)/2$ as desired. ■

We now prove Theorem 5.14, restated here for convenience.

Theorem 5.14 (Growth function sample bound) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, if a training sample S is drawn from $\mathcal{D}$ of size*

$$n \;\geq\; \frac{2}{\epsilon}[\log_2(2\mathcal{H}[2n]) + \log_2(1/\delta)]$$

then with probability $\geq 1-\delta$, every $h \in \mathcal{H}$ with $err_{\mathcal{D}}(h) \geq \epsilon$ has $err_S(h) > 0$ (equivalently, every $h \in \mathcal{H}$ with $err_S(h) = 0$ has $err_{\mathcal{D}}(h) < \epsilon$).

Proof: Consider drawing a set S of n examples from $\mathcal{D}$ and let A denote the event that there exists $h \in \mathcal{H}$ with true error greater than ϵ but training error zero. Our goal is to prove that $\text{Prob}(A) \leq \delta$.

By Lemma 5.19 it suffices to prove that $\text{Prob}(B) \leq \delta/2$. Consider a third experiment. Draw a set S'' of $2n$ points from $\mathcal{D}$ and then randomly partition S'' into two sets S and S' of n points each. Let B^* denote the event that there exists $h \in \mathcal{H}$ with $err_S(h) = 0$ but $err_{S'}(h) \geq \epsilon/2$. $\text{Prob}(B^*) = \text{Prob}(B)$ since drawing $2n$ points from $\mathcal{D}$ and randomly partitioning them into two sets of size n produces the same distribution on (S, S') as does drawing S and S' directly. The advantage of this new experiment is that we can now argue that $\text{Prob}(B^*)$ is low by arguing that for any set S'' of size $2n$, $\text{Prob}(B^*|S'')$ is low, with probability now taken over just the random partition of S'' into S and S'. The key point is that since S'' is fixed, there are at most $|\mathcal{H}[S'']| \leq \mathcal{H}[2n]$ events to worry about. Specifically, it suffices to prove that for any fixed $h \in \mathcal{H}[S'']$, the probability over the partition of S'' that h makes zero mistakes on S but more than $\epsilon n/2$ mistakes on S' is at most $\delta/(2\mathcal{H}[2n])$. We can then apply the union bound over $\mathcal{H}[S''] = \{h \cap S'' | h \in \mathcal{H}\}$.

To make the calculations easier, consider the following specific method for partitioning S'' into S and S'. Randomly put the points in S'' into pairs: (a_1, b_1), (a_2, b_2), $\ldots$, (a_n, b_n). For each index i, flip a fair coin. If heads put a_i into S and b_i into S', else if tails put a_i into S' and b_i into S. Now, fix some partition $h \in \mathcal{H}[S'']$ and consider the probability over these n fair coin flips that h makes zero mistakes on S but more than $\epsilon n/2$ mistakes on S'. First of all, if for any index i, h makes a mistake on both a_i and b_i then the probability is zero (because it cannot possibly make zero mistakes on S). Second, if there are fewer than $\epsilon n/2$ indices i such that h makes a mistake on either a_i or b_i then again the probability is zero because it cannot possibly make more than $\epsilon n/2$ mistakes on S'. So, assume there are $r \geq \epsilon n/2$ indices i such that h makes a mistake on exactly one of a_i or b_i. In this case, the chance that all of those mistakes land in S' is exactly $1/2^r$. This quantity is at most $1/2^{\epsilon n/2} \leq \delta/(2\mathcal{H}[2n])$ as desired for n as given in the theorem statement. ■

We now prove Theorem 5.15, restated here for convenience.

Theorem 5.15 (Growth function uniform convergence) *For any class $\mathcal{H}$ and distribution $\mathcal{D}$, if a training sample S is drawn from $\mathcal{D}$ of size*

$$n \;\geq\; \frac{8}{\epsilon^2}[\ln(2\mathcal{H}[2n]) + \ln(1/\delta)]$$

then with probability $\geq 1-\delta$*, every* $h \in \mathcal{H}$ *will have* $|err_S(h) - err_{\mathcal{D}}(h)| \leq \epsilon$*.*

Proof: This proof is identical to the proof of Theorem 5.14 except B^* is now the event that there exists a set $h \in \mathcal{H}[S'']$ such that the error of h on S differs from the error of h on S' by more than $\epsilon/2$. We again consider the experiment where we randomly put the points in S'' into pairs (a_i, b_i) and then flip a fair coin for each index i, if heads placing a_i into S and b_i into S', else placing a_i into S' and b_i into S. Consider the difference between the number of mistakes h makes on S and the number of mistakes h makes on S' and observe how this difference changes as we flip coins for $i = 1, 2, \ldots, n$. Initially, the difference is zero. If h makes a mistake on both or neither of (a_i, b_i) then the difference does not change. Else, if h makes a mistake on exactly one of a_i or b_i, then with probability 1/2 the difference increases by one and with probability 1/2 the difference decreases by one. If there are $r \leq n$ such pairs, then if we take a random walk of $r \leq n$ steps, what is the probability that we end up more than $\epsilon n/2$ steps away from the origin? This is equivalent to asking: if we flip $r \leq n$ fair coins, what is the probability the number of heads differs from its expectation by more than $\epsilon n/4$. By Hoeffding bounds, this is at most $2e^{-\epsilon^2 n/8}$. This quantity is at most $\delta/(2\mathcal{H}[2n])$ as desired for n as given in the theorem statement. ■

Finally, we prove Sauer's lemma, relating the growth function to the VC-dimension.

Theorem 5.16 (Sauer's lemma) *If* $\text{VCdim}(\mathcal{H}) = d$ *then* $\mathcal{H}[n] \leq \sum_{i=0}^{d}\binom{n}{i} \leq (\frac{en}{d})^d$.

Proof: Let $d = \text{VCdim}(\mathcal{H})$. Our goal is to prove for any set S of n points that $|\mathcal{H}[S]| \leq \binom{n}{\leq d}$, where we are defining $\binom{n}{\leq d} = \sum_{i=0}^{d}\binom{n}{i}$; this is the number of distinct ways of choosing d or fewer elements out of n. We will do so by induction on n. As a base case, our theorem is trivially true if $n \leq d$.

As a first step in the proof, notice that:

$$\binom{n}{\leq d} = \binom{n-1}{\leq d} + \binom{n-1}{\leq d-1} \tag{5.2}$$

because we can partition the ways of choosing d or fewer items into those that do not include the first item (leaving $\leq d$ to be chosen from the remainder) and those that do include the first item (leaving $\leq d-1$ to be chosen from the remainder).

Now, consider any set S of n points and pick some arbitrary point $x \in S$. By induction, we may assume that $|\mathcal{H}[S \setminus \{x\}]| \leq \binom{n-1}{\leq d}$. So, by equation (5.2) all we need to show is that $|\mathcal{H}[S]| - |\mathcal{H}[S \setminus \{x\}]| \leq \binom{n-1}{\leq d-1}$. Thus, our problem has reduced to analyzing how many *more* partitions there are of S than there are of $S \setminus \{x\}$ using sets in $\mathcal{H}$.

If $\mathcal{H}[S]$ is larger than $\mathcal{H}[S \setminus \{x\}]$, it is because of pairs of sets in $\mathcal{H}[S]$ that differ only on point x and therefore collapse to the same set when x is removed. For set $h \in \mathcal{H}[S]$

containing point x, define $\mathsf{twin}(h) = h \setminus \{x\}$; this may or may not belong to $\mathcal{H}[S]$. Let $\mathcal{T} = \{h \in \mathcal{H}[S] : x \in h \text{ and } \mathsf{twin}(h) \in \mathcal{H}[S]\}$. Notice $|\mathcal{H}[S]| - |\mathcal{H}[S \setminus \{x\}]| = |\mathcal{T}|$.

Now, what is the VC-dimension of $\mathcal{T}$? If $d' = \text{VCdim}(\mathcal{T})$, this means there is some set R of d' points in $S \setminus \{x\}$ that are shattered by $\mathcal{T}$. By definition of $\mathcal{T}$, all $2^{d'}$ subsets of R can be extended to either include x, or not include x and still be a set in $\mathcal{H}[S]$. In other words, $R \cup \{x\}$ is shattered by $\mathcal{H}$. This means, $d' + 1 \leq d$. Since $\text{VCdim}(\mathcal{T}) \leq d - 1$, by induction we have $|\mathcal{T}| \leq \binom{n-1}{\leq d-1}$ as desired. ∎

5.11.4 VC-Dimension of Combinations of Concepts

Often one wants to create concepts out of other concepts. For example, given several linear separators, one could take their intersection to create a convex polytope. Or given several disjunctions, one might want to take their majority vote. We can use Sauer's lemma to show that such combinations do not increase the VC-dimension of the class by too much.

Specifically, given k concepts $h_1, h_2, \ldots, h_k$ and a Booelan function f define the set $comb_f(h_1, \ldots, h_k) = \{x \in \mathcal{X} : f(h_1(x), \ldots, h_k(x)) = 1\}$, where here we are using $h_i(x)$ to denote the indicator for whether or not $x \in h_i$. For example, f might be the AND function to take the intersection of the sets h_i, or f might be the majority-vote function. This can be viewed as a *depth-two neural network*. Given a concept class $\mathcal{H}$, a Boolean function f, and an integer k, define the new concept class $COMB_{f,k}(\mathcal{H}) = \{comb_f(h_1, \ldots, h_k) : h_i \in \mathcal{H}\}$. We can now use Sauer's lemma to produce the following corollary.

Corollary 5.20 *If the concept class $\mathcal{H}$ has VC-dimension d, then for any combination function f, the class $COMB_{f,k}(\mathcal{H})$ has VC-dimension $O\big(kd\log(kd)\big)$.*

Proof: Let n be the VC-dimension of $\text{COMB}_{f,k}(\mathcal{H})$, so by definition, there must exist a set S of n points shattered by $\text{COMB}_{f,k}(\mathcal{H})$. We know by Sauer's lemma that there are at most n^d ways of partitioning the points in S using sets in $\mathcal{H}$. Since each set in $\text{COMB}_{f,k}(\mathcal{H})$ is determined by k sets in $\mathcal{H}$, and there are at most $(n^d)^k = n^{kd}$ different k-tuples of such sets, this means there are at most n^{kd} ways of partitioning the points using sets in $\text{COMB}_{f,k}(\mathcal{H})$. Since S is shattered, we must have $2^n \leq n^{kd}$, or equivalently $n \leq kd\log_2(n)$. We solve this as follows. First, assuming $n \geq 16$ we have $\log_2(n) \leq \sqrt{n}$ so $kd\log_2(n) \leq kd\sqrt{n}$ which implies that $n \leq (kd)^2$. To get the better bound, plug back into the original inequality. Since $n \leq (kd)^2$, it must be that $\log_2(n) \leq 2\log_2(kd)$. Substituting $\log n \leq 2\log_2(kd)$ into $n \leq kd\log_2 n$ gives $n \leq 2kd\log_2(kd)$. ∎

This result will be useful for our discussion of Boosting in Section 5.12.

5.11.5 Other Measures of Complexity

VC-dimension and number of bits needed to describe a set are not the only measures of complexity one can use to derive generalization guarantees. There has been significant work on a variety of measures. One measure called Rademacher complexity measures the extent to which a given concept class $\mathcal{H}$ can fit random noise. Given a set of n examples $S = \{x_1, \ldots, x_n\}$, the *empirical Rademacher complexity* of $\mathcal{H}$ is defined as $R_S(\mathcal{H}) = \mathbf{E}_{\sigma_1,\ldots,\sigma_n} \max_{h\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i h(x_i)$, where $\sigma_i \in \{-1, 1\}$ are independent random labels with $\text{Prob}[\sigma_i = 1] = \frac{1}{2}$. E.g., if you assign random ±1 labels to the points in S and the best classifier in $\mathcal{H}$ on average gets error 0.45 then $R_S(\mathcal{H}) = 0.55 - 0.45 = 0.1$. One can prove that with probability greater than or equal to $1-\delta$, every $h \in \mathcal{H}$ satisfies true error less than or equal to training error plus $R_S(\mathcal{H}) + 3\sqrt{\frac{\ln(2/\delta)}{2n}}$. For more on results such as this, see, e.g., [BM02].

5.12 Strong and Weak Learning - Boosting

We now describe *boosting*, which is important both as a theoretical result and as a practical and easy-to-use learning method.

A *strong learner* for a problem is an algorithm that with high probability is able to achieve any desired error rate ϵ using a number of samples that may depend polynomially on $1/\epsilon$. A *weak learner* for a problem is an algorithm that does just a little bit better than random guessing. It is only required to get with high probability an error rate less than or equal to $\frac{1}{2} - \gamma$ for some $0 < \gamma \leq \frac{1}{2}$. We show here that a weak-learner for a problem that achieves the weak-learning guarantee for any distribution of data can be boosted to a strong learner, using the technique of boosting. At the high level, the idea will be to take our training sample S, and then to run the weak-learner on different data distributions produced by weighting the points in the training sample in different ways. Running the weak learner on these different weightings of the training sample will produce a series of hypotheses $h_1, h_2, \ldots$, and the idea of our reweighting procedure will be to focus attention on the parts of the sample that previous hypotheses have performed poorly on. At the end we will combine the hypotheses together by a majority vote.

Assume the weak learning algorithm A outputs hypotheses from some class $\mathcal{H}$. Our boosting algorithm will produce hypotheses that will be majority votes over t_0 hypotheses from $\mathcal{H}$, for t_0 defined below. This means that we can apply Corollary 5.20 to bound the VC-dimension of the class of hypotheses our boosting algorithm can produce in terms of the VC-dimension of $\mathcal{H}$. In particular, the class of rules that can be produced by the booster running for t_0 rounds has VC-dimension $O(t_0\text{VCdim}(\mathcal{H})\log(t_0\text{VCdim}(\mathcal{H})))$. This in turn gives a bound on the number of samples needed, via Corollary 5.17, to ensure that high accuracy on the sample will translate to high accuracy on new data.

To make the discussion simpler, we will assume that the weak learning algorithm A, when presented with a weighting of the points in our training sample, always (rather than with high probability) produces a hypothesis that performs slightly better than random guessing with respect to the distribution induced by weighting. Specifically:

```
Boosting Algorithm

    Given a sample $S$ of $n$ labeled examples $\mathbf{x}_1, \ldots, \mathbf{x}_n$, initialize each
    example $\mathbf{x}_i$ to have a weight $w_i = 1$. Let $\mathbf{w} = (w_1, \ldots, w_n)$.

    For $t = 1, 2, \ldots, t_0$ do
        Call the weak learner on the weighted sample $(S, \mathbf{w})$, receiving
        hypothesis $h_t$.
        Multiply the weight of each example that was misclassified by
        $h_t$ by $\alpha = \frac{\frac{1}{2}+\gamma}{\frac{1}{2}-\gamma}$. Leave the other weights as they are.

    End

    Output the classifier MAJ$(h_1, \ldots, h_{t_0})$ which takes the majority vote
    of the hypotheses returned by the weak learner. Assume $t_0$ is odd so
    there is no tie.
```

Figure 5.6: The boosting algorithm

Definition 5.4 (γ-Weak learner on sample) *A γ-weak learner is an algorithm that given examples, their labels, and a nonnegative real weight w_i on each example $\mathbf{x}_i$, produces a classifier that correctly labels a subset of examples with total weight at least $(\frac{1}{2}+\gamma)\sum\limits_{i=1}^{n} w_i$.*

At the high level, boosting makes use of the intuitive notion that if an example was misclassified, one needs to pay more attention to it. The boosting procedure is in Figure 5.6.

Theorem 5.21 *Let A be a γ-weak learner for sample S. Then $t_0 = O(\frac{1}{\gamma^2}\log n)$ is sufficient so that the classifier $MAJ(h_1, \ldots, h_{t_0})$ produced by the boosting procedure has training error zero.*

Proof: Suppose m is the number of examples the final classifier gets wrong. Each of these m examples was misclassified at least $t_0/2$ times so each has weight at least $\alpha^{t_0/2}$. Thus the total weight is at least $m\alpha^{t_0/2}$. On the other hand, at time $t+1$, only the weights of examples misclassified at time t were increased. By the property of weak learning, the total weight of misclassified examples is at most $(\frac{1}{2}-\gamma)$ of the total weight at time t. Let weight(t) be the total weight at time t. Then

$$\begin{aligned}\text{weight}(t+1) &\leq \Big(\alpha\left(\tfrac{1}{2}-\gamma\right) + \left(\tfrac{1}{2}+\gamma\right)\Big) \times \text{weight}(t)\\ &= (1+2\gamma) \times \text{weight}(t).\end{aligned}$$

Since weight$(0) = n$, the total weight at the end is at most $n(1+2\gamma)^{t_0}$. Thus

$$m\alpha^{t_0/2} \leq \text{total weight at end} \leq n(1+2\gamma)^{t_0}.$$

Substituting $\alpha = \frac{1/2+\gamma}{1/2-\gamma} = \frac{1+2\gamma}{1-2\gamma}$ and rearranging terms

$$m \ \leq \ n(1-2\gamma)^{t_0/2}(1+2\gamma)^{t_0/2} \ = \ n[1-4\gamma^2]^{t_0/2}.$$

Using $1-x \leq e^{-x}$, $m \ \leq \ ne^{-2t_0\gamma^2}$. For $t_0 > \frac{\ln n}{2\gamma^2}$, $m < 1$, so the number of misclassified items must be zero. ∎

Having completed the proof of the boosting result, here are two interesting observations:

Connection to Hoeffding bounds: The boosting result applies even if our weak learning algorithm is "adversarial", giving us the least helpful classifier possible subject to Definition 5.4. This is why we don't want the α in the boosting algorithm to be too large, otherwise the weak learner could return the negation of the classifier it gave the last time. Suppose that the weak learning algorithm gave a classifier each time that for each example, flipped a coin and produced the correct answer with probability $\frac{1}{2}+\gamma$ and the wrong answer with probability $\frac{1}{2}-\gamma$, so it is a γ-weak learner in expectation. In that case, if we called the weak learner t_0 times, for any fixed $\mathbf{x}_i$, Hoeffding bounds imply the chance the majority vote of those classifiers is incorrect on $\mathbf{x}_i$ is at most $e^{-2t_0\gamma^2}$. So, the expected total number of mistakes m is at most $ne^{-2t_0\gamma^2}$. What is interesting is that this is the exact bound we get from boosting without the expectation for an adversarial weak-learner.

A minimax view: Consider a 2-player zero-sum game [24] with one row for each example $\mathbf{x}_i$ and one column for each hypothesis h_j that the weak-learning algorithm might output. If the row player chooses row i and the column player chooses column j, then the column player gets a payoff of one if $h_j(\mathbf{x}_i)$ is correct and gets a payoff of zero if $h_j(\mathbf{x}_i)$ is incorrect. The γ-weak learning assumption implies that for any randomized strategy for the row player (any "mixed strategy" in the language of game theory), there exists a response h_j that gives the column player an expected payoff of at least $\frac{1}{2}+\gamma$. The von Neumann minimax theorem [25] states that this implies there exists a probability distribution on the columns (a mixed strategy for the column player) such that for any $\mathbf{x}_i$, at least a $\frac{1}{2}+\gamma$ probability mass of the

[24] A two person zero sum game consists of a matrix whose columns correspond to moves for Player 1 and whose rows correspond to moves for Player 2. The ij^{th} entry of the matrix is the payoff for Player 1 if Player 1 choose the j^{th} column and Player 2 choose the i^{th} row. Player 2's payoff is the negative of Player1's.

[25] The von Neumann minimax theorem states that there exists a mixed strategy for each player so that given Player 2's strategy the best payoff possible for Player 1 is the negative of given Player 1's strategy the best possible payoff for Player 2. A mixed strategy is one in which a probability is assigned to every possible move for each situation a player could be in.

columns under this distribution is correct on $\mathbf{x}_i$. We can think of boosting as a fast way of finding a very simple probability distribution on the columns (just an average over $O(\log n)$ columns, possibly with repetitions) that is nearly as good (for any $\mathbf{x}_i$, more than half are correct) that moreover works even if our only access to the columns is by running the weak learner and observing its outputs.

We argued above that $t_0 = O(\frac{1}{\gamma^2} \log n)$ rounds of boosting are sufficient to produce a majority-vote rule h that will classify all of S correctly. Using our VC-dimension bounds, this implies that if the weak learner is choosing its hypotheses from concept class $\mathcal{H}$, then a sample size

$$n = \tilde{O}\left(\frac{1}{\epsilon}\left(\frac{\mathrm{VCdim}(\mathcal{H})}{\gamma^2}\right)\right)$$

is sufficient to conclude that with probability $1 - \delta$ the error is less than or equal to ϵ, where we are using the $\tilde{O}$ notation to hide logarithmic factors. It turns out that running the boosting procedure for larger values of t_0 i.e., continuing past the point where S is classified correctly by the final majority vote, does not actually lead to greater overfitting. The reason is that using the same type of analysis used to prove Theorem 5.21, one can show that as t_0 increases, not only will the majority vote be correct on each $\mathbf{x} \in S$, but in fact each example will be correctly classified by a $\frac{1}{2} + \gamma'$ fraction of the classifiers, where $\gamma' \to \gamma$ as $t_0 \to \infty$. I.e., the vote is approaching the minimax optimal strategy for the column player in the minimax view given above. This in turn implies that h can be well-approximated over S by a vote of a random sample of $O(1/\gamma^2)$ of its component weak hypotheses h_j. Since these small random majority votes are not overfitting by much, our generalization theorems imply that h cannot be overfitting by much either.

5.13 Stochastic Gradient Descent

We now describe a widely-used algorithm in machine learning, called *stochastic gradient descent* (SGD). The Perceptron algorithm we examined in Section 5.8.3 can be viewed as a special case of this algorithm, as can methods for deep learning.

Let $\mathcal{F}$ be a class of real-valued functions $f_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}$ where $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ is a vector of parameters. For example, we could think of the class of linear functions where $n = d$ and $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T x$, or we could have more complicated functions where $n > d$. For each such function $f_{\mathbf{w}}$ we can define an associated set $h_{\mathbf{w}} = \{\mathbf{x} : f_{\mathbf{w}}(\mathbf{x}) \geq 0\}$, and let $\mathcal{H}_{\mathcal{F}} = \{h_{\mathbf{w}} : f_{\mathbf{w}} \in \mathcal{F}\}$. For example, if $\mathcal{F}$ is the class of linear functions then $\mathcal{H}_{\mathcal{F}}$ is the class of linear separators.

To apply stochastic gradient descent, we also need a *loss function* $L(f_{\mathbf{w}}(\mathbf{x}), c^*(\mathbf{x}))$ that describes the real-valued penalty we will associate with function $f_{\mathbf{w}}$ for its prediction on an example $\mathbf{x}$ whose true label is $c^*(\mathbf{x})$. The algorithm is then the following:

Stochastic Gradient Descent:

Given: starting point $\mathbf{w} = \mathbf{w}_{init}$ and learning rates $\lambda_1, \lambda_2, \lambda_3, \ldots$
(e.g., $\mathbf{w}_{init} = \mathbf{0}$ and $\lambda_t = 1$ for all t, or $\lambda_t = 1/\sqrt{t}$).

Consider a sequence of random examples $(\mathbf{x}_1, c^*(\mathbf{x}_1)), (\mathbf{x}_2, c^*(\mathbf{x}_2)), \ldots$.

1. Given example $(\mathbf{x}_t, c^*(\mathbf{x}_t))$, compute the gradient $\nabla L(f_{\mathbf{w}}(\mathbf{x}_t), c^*(\mathbf{x}_t))$ of the loss of $f_{\mathbf{w}}(\mathbf{x}_t)$ with respect to the weights $\mathbf{w}$. This is a vector in $\mathbb{R}^n$ whose ith component is $\frac{\partial L(f_{\mathbf{w}}(\mathbf{x}_t), c^*(\mathbf{x}_t))}{\partial w_i}$.
2. Update: $\mathbf{w} \leftarrow \mathbf{w} - \lambda_t \nabla L(f_{\mathbf{w}}(\mathbf{x}_t), c^*(\mathbf{x}_t))$.

Let's now try to understand the algorithm better by seeing a few examples of instantiating the class of functions $\mathcal{F}$ and loss function L.

First, consider $n = d$ and $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, so $\mathcal{F}$ is the class of linear predictors. Consider the loss function $L(f_{\mathbf{w}}(\mathbf{x}), c^*(\mathbf{x})) = \max(0, -c^*(\mathbf{x})f_{\mathbf{w}}(\mathbf{x}))$, and recall that $c^*(\mathbf{x}) \in \{-1, 1\}$. In other words, if $f_{\mathbf{w}}(\mathbf{x})$ has the correct sign, then we have a loss of 0, otherwise we have a loss equal to the magnitude of $f_{\mathbf{w}}(\mathbf{x})$. In this case, if $f_{\mathbf{w}}(\mathbf{x})$ has the correct sign and is non-zero, then the gradient will be zero since an infinitesimal change in any of the weights will not change the sign. So, when $h_{\mathbf{w}}(\mathbf{x})$ is correct, the algorithm will leave $\mathbf{w}$ alone. On the other hand, if $f_{\mathbf{w}}(\mathbf{x})$ has the wrong sign, then $\frac{\partial L}{\partial w_i} = -c^*(\mathbf{x})\frac{\partial \mathbf{w}\cdot\mathbf{x}}{\partial w_i} = -c^*(\mathbf{x})x_i$. So, using $\lambda_t = 1$, the algorithm will update $w \leftarrow w + c^*(\mathbf{x})\mathbf{x}$. Note that this is exactly the Perceptron algorithm. (Technically we must address the case that $f_{\mathbf{w}}(\mathbf{x}) = 0$; in this case, we should view $f_{\mathbf{w}}$ as having the wrong sign just barely.)

As a small modification to the above example, consider the same class of linear predictors $\mathcal{F}$ but now modify the loss function to the hinge-loss $L(f_{\mathbf{w}}(\mathbf{x}), c^*(\mathbf{x})) = \max(0, 1 - c^*(\mathbf{x})f_{\mathbf{w}}(\mathbf{x}))$. This loss function now requires $f_{\mathbf{w}}(\mathbf{x})$ to have the correct sign *and* have magnitude at least 1 in order to be zero. Hinge loss has the useful property that it is an upper bound on error rate: for any sample S, the training error is at most $\sum_{\mathbf{x}\in S} L(f_{\mathbf{w}}(\mathbf{x}), c^*(\mathbf{x}))$. With this loss function, stochastic gradient descent is called the *margin perceptron* algorithm.

More generally, we could have a much more complex class $\mathcal{F}$. For example, consider a layered circuit of soft threshold gates. Each node in the circuit computes a linear function of its inputs and then passes this value through an "activation function" such as $a(z) = \tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$. This circuit could have multiple layers with the output of layer i being used as the input to layer $i + 1$. The vector $\mathbf{w}$ would be the concatenation of all the weight vectors in the network. This is the idea of *deep neural networks* discussed further in Section 5.15.

While it is difficult to give general guarantees on when stochastic gradient descent will succeed in finding a hypothesis of low error on its training set S, Theorems 5.7 and 5.5

imply that if it does and if S is sufficiently large, we can be confident that its true error will be low as well. Suppose that stochastic gradient descent is run on a machine where each weight is a 64-bit floating point number. This means that its hypotheses can each be described using $64n$ bits. If S has size at least $\frac{1}{\epsilon}[64n\ln(2)+\ln(1/\delta)]$, by Theorem 5.7 it is unlikely any such hypothesis of true error greater than ϵ will be consistent with the sample, and so if it finds a hypothesis consistent with S, we can be confident its true error is at most ϵ. Or, by Theorem 5.5, if $|S| \geq \frac{1}{2\epsilon^2}\big(64n\ln(2)+\ln(2/\delta)\big)$ then almost surely the final hypothesis h produced by stochastic gradient descent satisfies true error leas than or equal to training error plus ϵ.

5.14 Combining (Sleeping) Expert Advice

Imagine you have access to a large collection of rules-of-thumb that specify what to predict in different situations. For example, in classifying news articles, you might have one that says "if the article has the word 'football', then classify it as sports" and another that says "if the article contains a dollar figure, then classify it as business". In predicting the stock market, these could be different economic indicators. These predictors might at times contradict each other, e.g., a news article that has both the word "football" and a dollar figure, or a day in which two economic indicators are pointing in different directions. It also may be that no predictor is perfectly accurate with some much better than others. We present here an algorithm for combining a large number of such predictors with the guarantee that if any of them are good, the algorithm will perform nearly as well as each good predictor on the examples on which that predictor fires.

Formally, define a "sleeping expert" to be a predictor h that on any given example $\mathbf{x}$ either makes a prediction on its label or chooses to stay silent (asleep). We will think of them as black boxes. Now, suppose we have access to n such sleeping experts $h_1, \ldots, h_n$, and let S_i denote the subset of examples on which h_i makes a prediction (e.g., this could be articles with the word "football" in them). We consider the online learning model, and let $mistakes(A, S)$ denote the number of mistakes of an algorithm A on a sequence of examples S. Then the guarantee of our algorithm A will be that for all i

$$E\big(mistakes(A, S_i)\big) \leq (1+\epsilon)\cdot mistakes(h_i, S_i) + O\left(\tfrac{\log n}{\epsilon}\right)$$

where ϵ is a parameter of the algorithm and the expectation is over internal randomness in the randomized algorithm A.

As a special case, if $h_1, \ldots, h_n$ are concepts from a concept class $\mathcal{H}$, and so they all make predictions on every example, then A performs nearly as well as the best concept in $\mathcal{H}$. This can be viewed as a noise-tolerant version of the Halving Algorithm of Section 5.8.2 for the case that no concept in $\mathcal{H}$ is perfect. The case of predictors that make predictions on every example is called the problem of *combining expert advice*, and the more general case of predictors that sometimes fire and sometimes are silent is called the *sleeping experts* problem.

Combining Sleeping Experts Algorithm:

Initialize each expert h_i with a weight $w_i = 1$. Let $\epsilon \in (0,1)$. For each example x, do the following:

1. [Make prediction] Let H_x denote the set of experts h_i that make a prediction on x, and let $w_x = \sum\limits_{h_j \in H_x} w_j$. Choose $h_i \in H_x$ with probability $p_{ix} = w_i/w_x$ and predict $h_i(x)$.

2. [Receive feedback] Given the correct label, for each $h_i \in H_x$ let $m_{ix} = 1$ if $h_i(x)$ was incorrect, else let $m_{ix} = 0$.

3. [Update weights] For each $h_i \in H_x$, update its weight as follows:

   - Let $r_{ix} = \left(\sum_{h_j \in H_x} p_{jx} m_{jx}\right)/(1+\epsilon) - m_{ix}$.
   - Update $w_i \leftarrow w_i(1+\epsilon)^{r_{ix}}$.
     Note that $\sum_{h_j \in H_x} p_{jx} m_{jx}$ represents the algorithm's probability of making a mistake on example x. So, h_i is rewarded for predicting correctly $(m_{ix} = 0)$ especially when the algorithm had a high probability of making a mistake, and h_i is penalized for predicting incorrectly $(m_{ix} = 1)$ especially when the algorithm had a low probability of making a mistake.

   For each $h_i \notin H_x$, leave w_i alone.

Theorem 5.22 *For any set of n sleeping experts $h_1, \ldots, h_n$, and for any sequence of examples S, the Combining Sleeping Experts Algorithm A satisfies for all i:*

$$E\big(mistakes(A, S_i)\big) \leq (1+\epsilon) \cdot mistakes(h_i, S_i) + O\left(\tfrac{\log n}{\epsilon}\right)$$

where $S_i = \{x \in S : h_i \in H_x\}$.

Proof: Consider sleeping expert h_i. The weight of h_i after the sequence of examples S is exactly:

$$\begin{aligned} w_i &= (1+\epsilon)^{\sum_{x \in S_i}\left[\left(\sum_{h_j \in H_x} p_{jx} m_{jx}\right)/(1+\epsilon) - m_{ix}\right]} \\ &= (1+\epsilon)^{E[mistakes(A,S_i)]/(1+\epsilon) - mistakes(h_i,S_i)}. \end{aligned}$$

Let $w = \sum_j w_j$. Clearly $w_i \leq w$. Therefore, taking logs, we have:

$$E\big(mistakes(A, S_i)\big)/(1+\epsilon) - mistakes(h_i, S_i) \leq \log_{1+\epsilon} w.$$

So, using the fact that $\log_{1+\epsilon} w = O(\frac{\log W}{\epsilon})$,

$$E\big(mistakes(A, S_i)\big) \leq (1+\epsilon) \cdot mistakes(h_i, S_i) + O\left(\tfrac{\log w}{\epsilon}\right).$$

Initially, $w = n$. To prove the theorem, it is enough to prove that w never increases. To do so, we need to show that for each x, $\sum_{h_i \in H_x} w_i(1+\epsilon)^{r_{ix}} \leq \sum_{h_i \in H_x} w_i$, or equivalently dividing both sides by $\sum_{h_j \in H_x} w_j$ that $\sum_i p_{ix}(1+\epsilon)^{r_{ix}} \leq 1$, where for convenience we define $p_{ix} = 0$ for $h_i \notin H_x$.

For this we will use the inequalities that for $\beta, z \in [0,1]$, $\beta^z \leq 1-(1-\beta)z$ and $\beta^{-z} \leq 1+(1-\beta)z/\beta$. Specifically, we will use $\beta = (1+\epsilon)^{-1}$. We now have:

$$\begin{aligned}
\sum_i p_{ix}(1+\epsilon)^{r_{ix}} &= \sum_i p_{ix}\beta^{m_{ix}-(\sum_j p_{jx}m_{jx})\beta} \\
&\leq \sum_i p_{ix}\Big(1-(1-\beta)m_{ix}\Big)\left(1+(1-\beta)\left(\sum_j p_{jx}m_{jx}\right)\right) \\
&\leq \left(\sum_i p_{ix}\right) - (1-\beta)\sum_i p_{ix}m_{ix} + (1-\beta)\sum_i p_{ix}\sum_j p_{jx}m_{jx} \\
&= 1-(1-\beta)\sum_i p_{ix}m_{ix} + (1-\beta)\sum_j p_{jx}m_{jx} \\
&= 1,
\end{aligned}$$

where the second-to-last line follows from using $\sum_i p_{ix} = 1$ in two places. So w never increases and the bound follows as desired. ∎

5.15 Deep Learning

Deep learning, or *deep neural networks*, refers to training many-layered networks of nonlinear computational units.

Each computational unit or gate works as follows: there are a set of "wires" bringing inputs to the gate. Each wire has a "weight"; the gate's output is a real number obtained by applying a non-linear "activation function" $g : \mathbf{R} \to \mathbf{R}$ to the the weighted sum of the input values. The activation function g is generally the same for all gates in the network, though, the number of inputs to individual gates may differ.

The input to the network is an example $\mathbf{x} \in R^d$. The first layer of the network transforms the example into a new vector $f_1(\mathbf{x})$. Then the second layer transforms $f_1(\mathbf{x})$ into a new vector $f_2(f_1(\mathbf{x}))$, and so on. Finally, the k^{th} layer outputs the final prediction $f(\mathbf{x}) = f_k(f_{k-1}(\ldots(f_1(\mathbf{x}))))$.

In supervised learning, we are given training examples $\mathbf{x_1}, \mathbf{x_2}, \ldots$, and corresponding labels $c^*(\mathbf{x_1}), c^*(\mathbf{x_2}), \ldots$. The training process finds a set of weights of all wires so as to minimize the error: $(f_0(\mathbf{x_1} - c^*(\mathbf{x_1}))^2 + (f_0(\mathbf{x_2}) - c^*(\mathbf{x_2}))^2 + \cdots$. (One could alternatively aim to minimize other quantities besides the sum of squared errors of training examples.) Often training is carried out by running stochastic gradient descient, i.e., doing stochastic gradient descent in the weights space.

The motivation for deep learning is that often we are interested in data, such as images, that are given to us in terms of very low-level features, such as pixel intensity values. Our goal is to achieve some higher-level understanding of each image, such as what objects are in the image and what they are doing. To do so, it is natural to first convert the given low-level representation into one of higher-level features. That is what the layers of the network aim to do. Deep learning is also motivated by multi-task learning, with the idea that a good higher-level representation of data should be useful for a wide range of tasks. Indeed, a common use of deep learning for multi-task learning is to share initial levels of the network across tasks.

A typical architecture of a deep neural network consists of layers of logic units. In a fully connected layer, the output of each gate in the layer is connected to the input of every gate in the next layer. However, if the input is an image one might like to recognize features independent of where they are located in the image. To achieve this one often uses a number of convolution layers. In a convolution layer, each gate gets inputs from a small $k \times k$ grid where k may be 5 to 10. There is a gate for each $k \times k$ square array of the image. The weights on each gate are tied together so that each gate recognizes the same feature. There will be several such collections of gates, so several different features can be learned. Such a level is called a convolution level and the fully connected layers are called autoencoder levels. A technique called *pooling* is used to keep the number of gates reasonable. A small $k \times k$ grid with k typically set to two is used to scan a layer. The stride is set so the grid will provide a non overlapping cover of the layer. Each $k \times k$ input grid will be reduced to a single cell by selecting the maximum input value or the average of the inputs. For $k = 2$ this reduces the number of cells by a factor of four.

Deep learning networks are trained by stochastic gradient descent (Section 5.13), sometimes called back propagation in the network context. An error function is constructed and the weights are adjusted using the derivative of the error function. This requires that the error function be differentiable. A smooth threshold is used such as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad \text{where} \quad \frac{\partial}{\partial x}\frac{e^e - e^{-e}}{e^x + e^{-x}} = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2$$

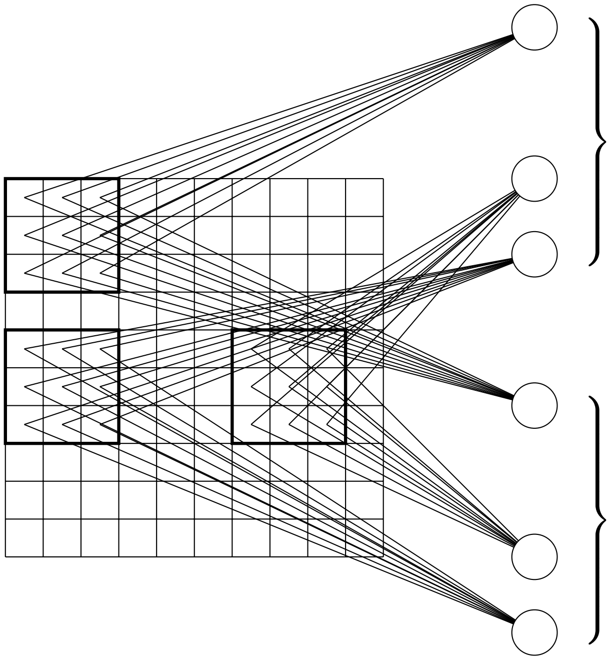
or $\text{sigmod}(x) = \frac{1}{1+e^{-x}}$ where

$$\frac{\partial\ \text{sigmod}(x)}{\partial x} = \frac{e^{-x}}{(1+e^{-x})^2} = sigmod(x)\frac{e^{-x}}{1+e^{-x}} = \text{sigmoid}(x)\big(1 - \text{sigmoid}(x)\big).$$

In fact the function

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{where} \quad \frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 1 & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

seems to work well even though its derivative at $x = 0$ is undefined. An advantage of ReLU over sigmoid is that ReLU does not saturate far from the origin.



Each gate is connected to a $k \times k$ grid. Weights are tied together.

Second set of gates each connected to a $k \times k$ grid. Weights are tied together.

Figure 5.7: Convolution layers

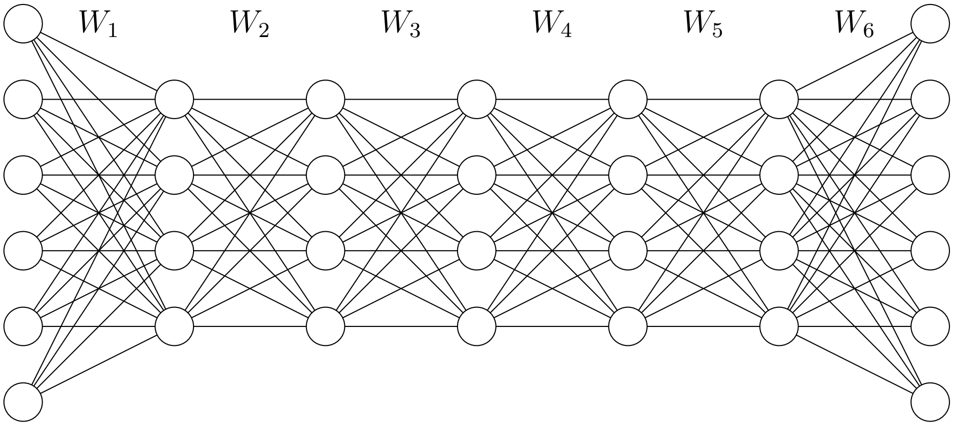


Figure 5.8: A deep learning fully connected network.

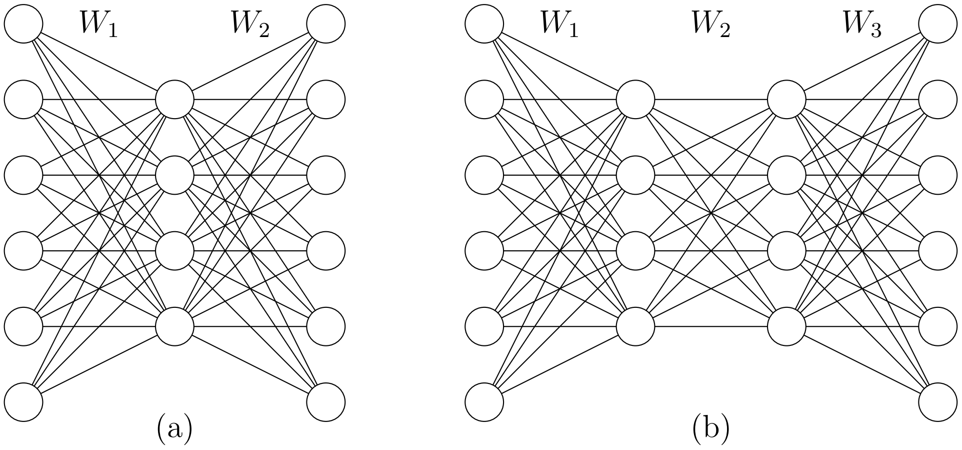


Figure 5.9: Autoencoder technique used to train one level at a time. In the Figure 5.9 (a) train W_1 and W_2. Then in Figure 5.9 (b), freeze W_1 and train W_2 and W_3. In this way one trains one set of weights at a time.

Training a deep learning network of 7 or 8 levels using gradient descent can be computationally expensive.[26] To address this issue one can train one level at a time on unlabeled data using an idea called autoencoding. There are three levels, the input, a middle level called the hidden level, and an output level as shown in Figure 5.9a. There are two sets of weights. W_1 is the weights of the hidden level gates and W_2 is W_1^T. Let $\mathbf{x}$ be the input pattern and $\mathbf{y}$ be the output. The error is $|\mathbf{x}-\mathbf{y}|^2$. One uses gradient descent to reduce the error. Once the weights W_1 are determined they are frozen and a second hidden level of gates is added as in Figure 5.9 b. In this network $W_3 = W_2^T$ and stochastic gradient descent is again used this time to determine W_2. In this way one level of weights is trained at a time.

The output of the hidden gates is an encoding of the input. An image might be a 10^8 dimensional input and there may only be 10^5 hidden gates. However, the number of images might be 10^7 so even though the dimension of the hidden layer is smaller than the dimension of the input, the number of possible codes far exceeds the number of inputs and thus the hidden layer is a compressed representation of the input. If the hidden layer were the same dimension as the input layer one might get the identity mapping. This does not happen for gradient descent starting with random weights.

The output layer of a deep network typically uses a softmax procedure. Softmax is a generalization of logistic regression where given a set of vectors $\{\mathbf{x_1}, \mathbf{x_2}, \ldots \mathbf{x_n}\}$ with labels $l_1, l_2, \ldots l_n$, $l_i \in \{0, 1\}$ and with a weight vector $\mathbf{w}$ we define the probability that

[26] In the image recognition community, researchers work with networks of 150 levels. The levels tend to be convolution rather than fully connected.

the label l given x equals 0 or 1 by

$$\text{Prob}(l = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^{\mathbf{T}}\mathbf{x}}} = \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})$$

and

$$\text{Prob}(l = 0|\mathbf{x}) = 1 - \text{Prob}(l = 1/\mathbf{x})$$

where σ is the sigmoid function.

Define a cost function

$$J(\mathbf{w}) = \sum_i \Big(l_i \log(\text{Prob}(l = 1|\mathbf{x})) + (1 - l_i) \log(1 - \text{Prob}(l = 1|\mathbf{x})) \Big)$$

and compute $\mathbf{w}$ to minimize $J(\mathbf{x})$. Then

$$J(\mathbf{w}) = \sum_i \Big(l_i \log(\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})) + (1 - l_i) \log(1 - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})) \Big)$$

Since $\frac{\partial \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})}{\partial w_j} = \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})(1 - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}))x_j$, it follows that $\frac{\partial \log(\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}))}{\partial w_j} = \frac{\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})(1-\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}))x_j}{\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})}$, Thus

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \sum_i \left(l_i \frac{\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})(1 - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}))}{\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})} x_j - (1 - l_i) \frac{(1 - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}))\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})}{1 - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})} x_j \right) \\
&= \sum_i \Big(l_i(1 - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}))x_j - (1 - l_i)\sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})x_j \Big) \\
&= \sum_i \Big((l_i x_j - l_i \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})x_j - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})x_j + l_i \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x})x_j \Big) \\
&= \sum_i \Big(l_i - \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}) \Big) x_j.
\end{aligned}$$

Softmax is a generalization of logistic regression to multiple classes. Thus, the labels l_i take on values $\{1, 2, \ldots, k\}$. For an input $\mathbf{x}$, softmax estimates the probability of each label. The hypothesis is of the form

$$h_w(x) = \begin{bmatrix} \text{Prob}(l = 1|\mathbf{x}, \mathbf{w_1}) \\ \text{Prob}(l = 2|\mathbf{x}, \mathbf{w_2}) \\ \vdots \\ \text{Prob}(l = k|\mathbf{x}, \mathbf{w_k}) \end{bmatrix} = \frac{1}{\sum_{i=1}^{k} e^{\mathbf{w_i^T x}}} \begin{bmatrix} e^{\mathbf{w_1^T x}} \\ e^{\mathbf{w_2^T x}} \\ \vdots \\ e^{\mathbf{w_k^T x}} \end{bmatrix}$$

where the matrix formed by the weight vectors is

$$W = (\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_k})^T$$

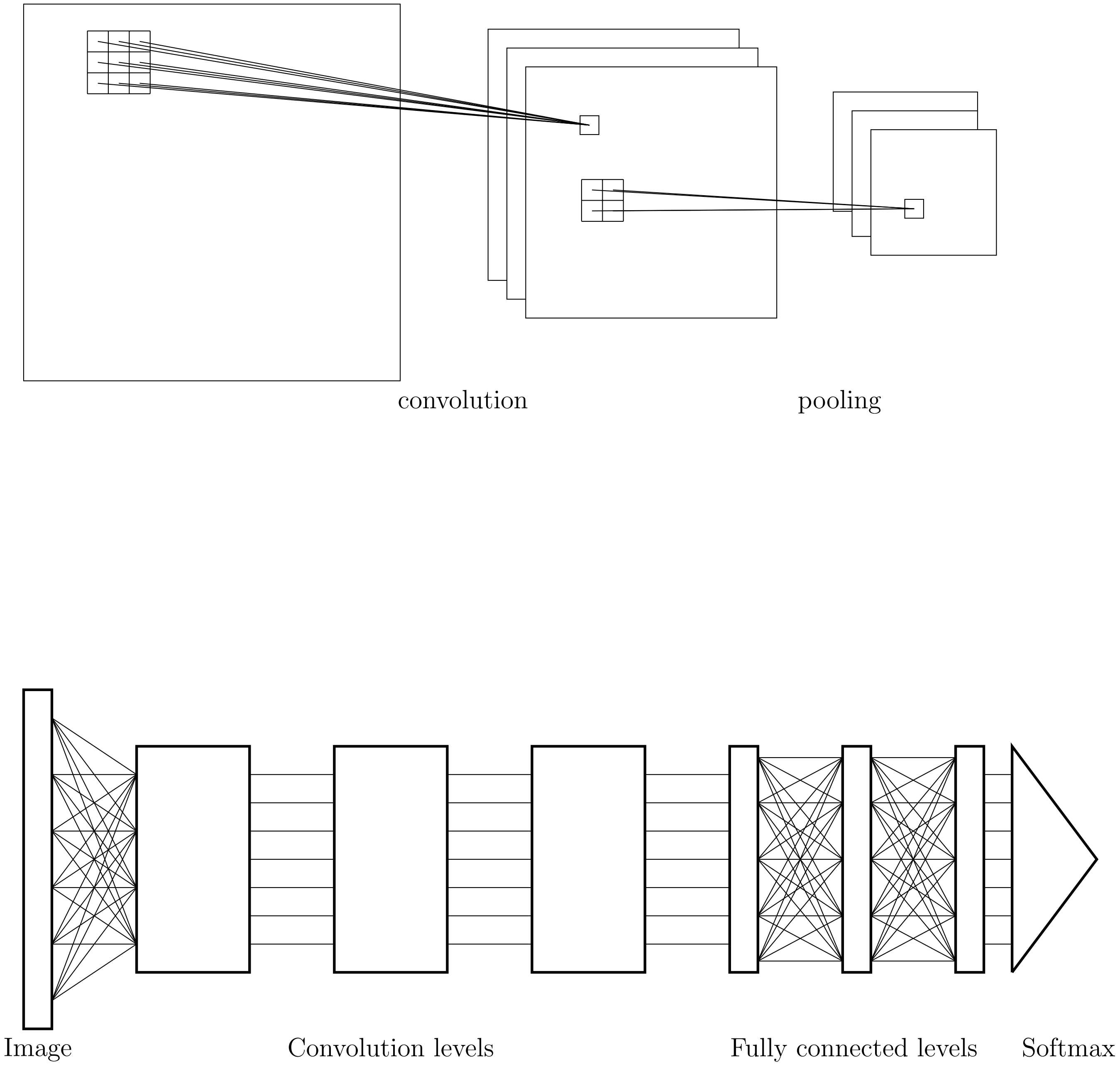


Figure 5.10: A convolution network

W is a matrix since for each label l_i, there is a vector $\mathbf{w_i}$ of weights.

Consider a set of n inputs $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$. Define

$$\delta(l = k) = \begin{cases} 1 & \text{if } l = k \\ 0 & \text{otherwise} \end{cases}$$

and

$$J(W) = \sum_{i=1}^{n} \sum_{j=1}^{k} \delta(l_i = j) \log \frac{e^{\mathbf{w_j^T} x_i}}{\sum_{h=1}^{k} e^{\mathbf{w_h^T} x_i}}.$$

The derivative of the cost function with respect to the weights is

$$\nabla_{\mathbf{w_i}} J(W) = -\sum_{j=1}^{n} \mathbf{x_j} \big(\delta(l_j = k) - \text{Prob}(l_j = k) | \mathbf{x_j}, W \big).$$

Note $\nabla_{\mathbf{w_i}} J(W)$ is a vector. Since $\mathbf{w_i}$ is a vector, each component of $\nabla_{\mathbf{w_i}} J(W)$ is the derivative with respect to one component of the vector $\mathbf{w_i}$.

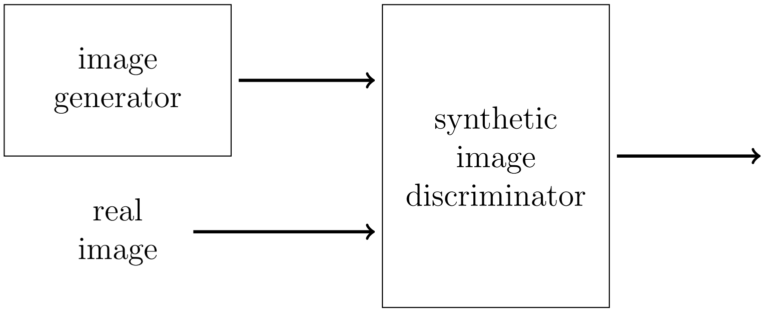
Over fitting is a major concern in deep learning since large networks can have hundreds of millions of weights. In image recognition, the number of training images can be significantly increased by random jittering of the images. Another technique called *dropout* randomly deletes a fraction of the weights at each training iteration. Regularization is used to assign a cost to the size of weights and many other ideas are being explored.

Deep learning is an active research area. Some of the ideas being explored are what do individual gates or sets of gates learn. If one trains a network twice from starting with random sets of weights, do gates learn the same features? In image recognition, the early convolution layers seem to learn features of images rather than features of the specific set of images they are being trained with. Once a network is trained on say a set of images one of which is a cat one can freeze the weights and then find images that will map to the activation vector generated by the cat image. One can take an artwork image and separate the style from the content and then create an image using the content but a different style [GEB15]. This is done by taking the activation of the original image and moving it to the manifold of activation vectors of images of a given style. One can do many things of this type. For example one can change the age of a child in an image or change some other feature [GKL+15]. For more information about deep learning, see [Ben09].[27]

5.15.1 Generative Adversarial Networks (GANs)

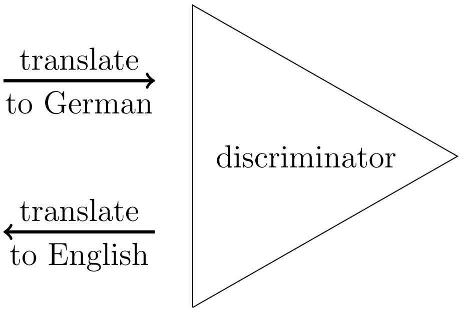
A method that is promising in trying to generate images that look real is to create code that tries to discern between real images and synthetic images.

[27] See also the tutorials: `http://deeplearning.net/tutorial/deeplearning.pdf` and `http://deeplearning.stanford.edu/tutorial/`.



One first trains the synthetic image discriminator to distinguish between real images and synthetic ones. Then one trains the image generator to generate images that the discriminator believes are real images. Alternating the training between the two units ends up forcing the image generator to produce real looking images. This is the idea of Generative Adversarial Networks.

There are many possible applications for this technique. Suppose you wanted to train a network to translate from English to German. First train a discriminator to determine if a sentence is a real sentence in German as opposed to a synthetic sentence. Then train a translator for English to German and a translator from German to English.



5.16 Further Current Directions

We now briefly discuss a few additional current directions in machine learning, focusing on *semi-supervised* learning, *active* learning, and *multi-task* learning.

5.16.1 Semi-Supervised Learning

Semi-supervised learning refers to the idea of trying to use a large unlabeled data set U to augment a given labeled data set L in order to produce more accurate rules than would have been achieved using just L alone. The motivation is that in many settings (e.g., document classification, image classification, speech recognition), unlabeled data is much more plentiful than labeled data, so one would like to make use of it if possible. Of course, unlabeled data is missing the labels! Nonetheless it often contains information that an algorithm can take advantage of.

As an example, suppose one believes the target function is a linear separator that separates most of the data by a large margin. By observing enough unlabeled data to estimate the probability mass near to any given linear separator, one could in principle then discard separators in advance that slice through dense regions and instead focus attention on just those that indeed separate most of the distribution by a large margin. This is the high level idea behind a technique known as Semi-Supervised SVMs. Alternatively, suppose data objects can be described by two different "kinds" of features (e.g., a webpage could be described using words on the page itself or using words on links pointing *to* the page), and one believes that each kind should be sufficient to produce an accurate classifier. Then one might want to train a *pair* of classifiers (one on each type of feature) and use unlabeled data for which one is confident but the other is not to bootstrap, labeling such examples with the confident classifier and then feeding them as training data to the less-confident one. This is the high-level idea behind a technique known as Co-Training. Or, if one believes "similar examples should generally have the same label", one might construct a graph with an edge between examples that are sufficiently similar, and aim for a classifier that is correct on the labeled data and has a small cut value on the unlabeled data; this is the high-level idea behind graph-based methods.

A formal model: The batch learning model introduced in Sections 5.1 and 5.6 in essence assumes that one's prior beliefs about the target function be described in terms of a class of functions $\mathcal{H}$. In order to capture the reasoning used in semi-supervised learning, we need to also describe beliefs about the *relation* between the target function and the data distribution. A clean way to do this is via a *notion of compatibility* χ between a hypothesis h and a distribution $\mathcal{D}$. Formally, χ maps pairs $(h, \mathcal{D})$ to $[0, 1]$ with $\chi(h, \mathcal{D}) = 1$ meaning that h is highly compatible with $\mathcal{D}$ and $\chi(h, \mathcal{D}) = 0$ meaning that h is very *in*compatible with $\mathcal{D}$. The quantity $1 - \chi(h, \mathcal{D})$ is called the *unlabeled error rate* of h, and denoted $err_{unl}(h)$. Note that for χ to be useful, it must be estimatable from a finite sample; to this end, let us further require that χ is an expectation over individual examples. That is, overloading notation for convenience, we require $\chi(h, \mathcal{D}) = \mathbf{E}_{x \sim \mathcal{D}}[\chi(h, x)]$, where $\chi : \mathcal{H} \times \mathcal{X} \to [0, 1]$.

For instance, suppose we believe the target should separate most data by margin γ. We can represent this belief by defining $\chi(h, x) = 0$ if x is within distance γ of the decision boundary of h, and $\chi(h, x) = 1$ otherwise. In this case, $err_{unl}(h)$ will denote the probability mass of $\mathcal{D}$ within distance γ of h's decision boundary. As a different example, in co-training, we assume each example can be described using two "views" that each are sufficient for classification; that is, there exist c_1^*, c_2^* such that for each example $x = \langle x_1, x_2 \rangle$ we have $c_1^*(x_1) = c_2^*(x_2)$. We can represent this belief by defining a hypothesis $h = \langle h_1, h_2 \rangle$ to be compatible with an example $\langle x_1, x_2 \rangle$ if $h_1(x_1) = h_2(x_2)$ and incompatible otherwise; $err_{unl}(h)$ is then the probability mass of examples on which h_1 and h_2 disagree.

As with the class $\mathcal{H}$, one can either assume that the target is fully compatible (i.e., $err_{unl}(c^*) = 0$) or instead aim to do well as a function of how compatible the target is. The case that we assume $c^* \in \mathcal{H}$ and $err_{unl}(c^*) = 0$ is termed the "doubly realizable case". The concept class $\mathcal{H}$ and compatibility notion χ are both viewed as *known.*

Intuition: In this framework, the way that unlabeled data helps in learning can be intuitively described as follows. Suppose one is given a concept class $\mathcal{H}$ (such as linear separators) and a compatibility notion χ (such as penalizing h for points within distance γ of the decision boundary). Suppose also that one believes $c^* \in \mathcal{H}$ (or at least is close) and that $err_{unl}(c^*) = 0$ (or at least is small). Then, unlabeled data can help by allowing one to estimate the *unlabeled error rate* of all $h \in \mathcal{H}$, thereby in principle reducing the search space from $\mathcal{H}$ (all linear separators) down to just the subset of $\mathcal{H}$ that is highly compatible with $\mathcal{D}$. The key challenge is how this can be done efficiently (in theory, in practice, or both) for natural notions of compatibility, as well as identifying types of compatibility that data in important problems can be expected to satisfy.

A theorem: The following is a semi-supervised analog of our basic sample complexity theorem, Theorem 5.3. First, fix some set of functions $\mathcal{H}$ and compatibility notion χ. Given a labeled sample L, define $\widehat{err}(h)$ to be the fraction of mistakes of h on L. Given an unlabeled sample U, define $\chi(h, U) = \mathbf{E}_{x\sim U}[\chi(h, x)]$ and define $\widehat{err}_{unl}(h) = 1 - \chi(h, U)$. That is, $\widehat{err}(h)$ and $\widehat{err}_{unl}(h)$ are the empirical error rate and unlabeled error rate of h, respectively. Finally, given $\alpha > 0$, define $\mathcal{H}_{\mathcal{D},\chi}(\alpha)$ to be the set of functions $f \in \mathcal{H}$ such that $err_{unl}(f) \leq \alpha$.

Theorem 5.23 *If $c^* \in \mathcal{H}$ then with probability at least $1 - \delta$, for labeled set L and unlabeled set U drawn from $\mathcal{D}$, the $h \in \mathcal{H}$ that optimizes $\widehat{err}_{unl}(h)$ subject to $\widehat{err}(h) = 0$ will have $err_{\mathcal{D}}(h) \leq \epsilon$ for*

$$|U| \geq \frac{2}{\epsilon^2}\left[\ln|\mathcal{H}| + \ln\frac{4}{\delta}\right], \text{ and } |L| \geq \frac{1}{\epsilon}\left[\ln|\mathcal{H}_{\mathcal{D},\chi}(err_{unl}(c^*) + 2\epsilon)| + \ln\frac{2}{\delta}\right].$$

Equivalently, for $|U|$ satisfying this bound, for any $|L|$, whp the $h \in \mathcal{H}$ that minimizes $\widehat{err}_{unl}(h)$ subject to $\widehat{err}(h) = 0$ has

$$err_{\mathcal{D}}(h) \leq \frac{1}{|L|}\left[\ln|\mathcal{H}_{\mathcal{D},\chi}(err_{unl}(c^*) + 2\epsilon)| + \ln\frac{2}{\delta}\right].$$

Proof: By Hoeffding bounds, $|U|$ is sufficiently large so that with probability at least $1 - \delta/2$, all $h \in \mathcal{H}$ have $|\widehat{err}_{unl}(h) - err_{unl}(h)| \leq \epsilon$. Thus we have:

$$\{f \in \mathcal{H} : \widehat{err}_{unl}(f) \leq err_{unl}(c^*) + \epsilon\} \subseteq \mathcal{H}_{\mathcal{D},\chi}(err_{unl}(c^*) + 2\epsilon).$$

The given bound on $|L|$ is sufficient so that with probability at least $1 - \delta$, all $h \in \mathcal{H}$ with $\widehat{err}(h) = 0$ and $\widehat{err}_{unl}(h) \leq err_{unl}(c^*) + \epsilon$ have $err_{\mathcal{D}}(h) \leq \epsilon$; furthermore, $\widehat{err}_{unl}(c^*) \leq err_{unl}(c^*) + \epsilon$, so such a function h exists. Therefore, with probability at least $1 - \delta$, the $h \in \mathcal{H}$ that optimizes $\widehat{err}_{unl}(h)$ subject to $\widehat{err}(h) = 0$ has $err_{\mathcal{D}}(h) \leq \epsilon$, as desired. ■

One can view Theorem 5.23 as bounding the number of labeled examples needed to learn well as a function of the "helpfulness" of the distribution $\mathcal{D}$ with respect to χ. Namely, a helpful distribution is one in which $\mathcal{H}_{\mathcal{D},\chi}(\alpha)$ is small for α slightly larger than the compatibility of the true target function, so we do not need much labeled data to identify a good function among those in $\mathcal{H}_{\mathcal{D},\chi}(\alpha)$. For more information on semi-supervised learning, see [BB10, BM98, CSZ06, Joa99, Zhu06, ZGL03].

5.16.2 Active Learning

Active learning refers to algorithms that take an active role in the selection of which examples are labeled. The algorithm is given an initial unlabeled set U of data points drawn from distribution $\mathcal{D}$ and then interactively requests for the labels of a small number of these examples. The aim is to reach a desired error rate ϵ using much fewer labels than would be needed by just labeling random examples (i.e., passive learning).

As a simple example, suppose that data consists of points on the real line and $\mathcal{H} = \{f_a : f_a(x) = 1 \text{ iff } x \geq a\}$ for $a \in R$. That is, $\mathcal{H}$ is the set of all threshold functions on the line. It is not hard to show (see Exercise 5.2) that a random labeled sample of size $O(\frac{1}{\epsilon}\log(\frac{1}{\delta}))$ is sufficient to ensure that with probability $\geq 1-\delta$, any consistent threshold a' has error at most ϵ. Moreover, it is not hard to show that $\Omega(\frac{1}{\epsilon})$ random examples are necessary for passive learning. However, with active learning we can achieve error ϵ using only $O(\log(\frac{1}{\epsilon}) + \log\log(\frac{1}{\delta}))$ labels. Specifically, first draw an unlabeled sample U of size $O(\frac{1}{\epsilon}\log(\frac{1}{\delta}))$. Then query the leftmost and rightmost points: if these are both negative then output $a' = \infty$, and if these are both positive then output $a' = -\infty$. Otherwise (the leftmost is negative and the rightmost is positive), perform binary search to find two adjacent examples x, x' such that x is negative and x' is positive, and output $a' = (x+x')/2$. This threshold a' is consistent with the labels on the entire set U, and so by the above argument, has error $\leq \epsilon$ with probability $\geq 1-\delta$.

The agnostic case, where the target need not belong in the given class $\mathcal{H}$ is quite a bit more subtle, and addressed in a quite general way in the "A^2" Agnostic Active learning algorithm [BBL09]. For more information on active learning, see [Das11, BU14].

5.16.3 Multi-Task Learning

In this chapter we have focused on scenarios where our goal is to learn a single target function c^*. However, there are also scenarios where one would like to learn *multiple* target functions $c_1^*, c_2^*, \ldots, c_n^*$. If these functions are related in some way, then one could hope to do so with less data per function than one would need to learn each function separately. This is the idea of *multi-task learning.*

One natural example is object recognition. Given an image $\mathbf{x}$, $c_1^*(\mathbf{x})$ might be 1 if $\mathbf{x}$ is a coffee cup and 0 otherwise; $c_2^*(\mathbf{x})$ might be 1 if $\mathbf{x}$ is a pencil and 0 otherwise; $c_3^*(\mathbf{x})$ might be 1 if $\mathbf{x}$ is a laptop and 0 otherwise. These recognition tasks are related in that image features that are good for one task are likely to be helpful for the others as well. Thus, one approach to multi-task learning is to try to learn a common representation under which each of the target functions can be described as a simple function. Another natural example is personalization. Consider a speech recognition system with n different users. In this case there are n target tasks (recognizing the speech of each user) that are clearly related to each other. Some good references for multi-task learning are [TM95, Thr96].

5.17 Bibliographic Notes

The basic theory underlying learning in the distributional setting was developed by Vapnik [Vap82], Vapnik and Chervonenkis [VC71], and Valiant [Val84]. The connection of this to the notion of Occam's razor is due to [BEHW87]. For more information on uniform convergence, regularization and complexity penalization, see [Vap98]. The Perceptron algorithm for online learning of linear separators was first analyzed by Block [Blo62] and Novikoff [Nov62]; the proof given here is from [MP69]. A formal description of the online learning model and its connections to learning in the distributional setting is given in [Lit87]. Support Vector Machines and their connections to kernel functions were first introduced by [BGV92], and extended by [CV95], with analysis in terms of margins given by [STBWA98]. For further reading on SVMs, learning with kernel functions, and regularization, see [SS01]. VC dimension is due to Vapnik and Chervonenkis [VC71] with the results presented here given in Blumer, Ehrenfeucht, Haussler and Warmuth [BEHW89]. Boosting was first introduced by Schapire [Sch90], and Adaboost and its guarantees are due to Freund and Schapire [FS97] . Analysis of the problem of combining expert advice was given by Littlestone and Warmuth [LW94] and Cesa-Bianchi et al. [CBFH^{+}97]; the analysis of the sleeping experts problem given here is from [BM07].