Optimization for Neural Networks

Aditya Ramesh

Abstract

Top-scoring submissions for any benchmark involving image recognition or acoustic modeling invariably use neural networks. The intuition underlying neural networks is simple. Many sources attempt to convey this intuition, but lack clarity due to the use of imprecise notation. This makes it difficult for a practitioner to read one of these sources and subsequently implement neural networks to solve nontrivial tasks. One of the purposes of this report is to establish a solid foundation based on current practices, so that the reader can easily incorporate the results of current research.

Neural networks are prototypical of the large-scale optimization problems often encountered when applying machine learning in practice. None of the sources makes the connection between training neural networks and statistical learning theory. But this connection is necessary in order to explain the differences between small-scale problems and large-scale problems from the perspective of optimization. We develop guidelines based on this investigation, and use them to characterize the most successful optimization algorithms used to train neural networks today. Finally, we implement several of these algorithms, and discuss the results based on our current understanding.

1 Introduction to Neural Networks

1.1 Representation

Neural networks take their inspiration from information processing in biological systems. In the human brain, information is transmitted between neurons in the form of electrical and chemical signals sent across synapses. Each neuron receives signals from a set of input neurons, and, under certain conditions, will broadcast a signal to a set of output neurons. This response can be thought of as the result of a local computation involving the input signals. We can model the flow of information in this network of neurons using a directed graph G, in which the neurons are nodes and the synapses are edges. This rough biological conceptualization of neural communication leads to a mathematical structure that we will fashion into a model of computation.

Our goal will now be to derive a mathematical description for G, so that the resulting neural network can be used for function approximation. To emphasize that our notion of neural network has little to do with neuroscience, we will refer to the "neurons" in the network as nodes in G. We limit our discussion to *feed-forward* neural networks, which

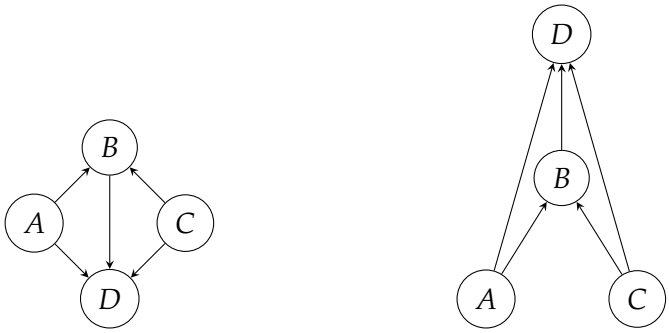


Figure 1.1: In 1.1a, we see a DAG, and in 1.1b, the representation of the DAG as a layered graph. Notice that the edges $A \to D$ and $C \to D$ are manifested as skip connections in the layered graph.

do not contain cycles. Consequently, G must be a directed acyclic graph (DAG). Now suppose that we wish to use G to approximate a function $f : \mathbb{R}^m \to \mathbb{R}^n$. If $f(x) = y$ for some $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, then our task will be to use G to "learn" what f does to transform x into y.

Let us first impose some organizational structure on G. We delegate to a set I of m input nodes the task of broadcasting the components of x to other nodes in G. In order for the neural network to be useful, the transmission of information must eventually cease at a set O of n output nodes. The information computed by each node in O will correspond to a component of $\hat{y}$, the network's approximation to y. As things stand, I and O can consist of arbitrary nodes of G. It would help if we could hide the entangled mass of nodes of edges (aka *connections*) involved in the communication between I and O, and simply think of I as x and O as $\hat{y}$. Fortunately, our existing definitions allow us to do far more than this.

Any DAG can be rendered as a layered graph. In a layered graph, nodes are arranged in horizontal rows, and only vertical connections in one direction, between nodes in different layers, are allowed. The process by which this is accomplished is called *layered graph drawing* (see Figure 1.1). Since G is a DAG, we can partition its nodes into an array of successive layers $L_0, \ldots, L_d$, where d is the *depth* of the neural network. The first layer, L_0, consists of the input nodes ordered from left to right based on the components of x to which they correspond. It is called the *input layer*. Not all output nodes necessarily belong to the last layer, so an *output layer* need not exist in general. In our case, we assume it does, so L_d must be the output layer. Layers in between L_0 and L_d are called *hidden layers*. Figure 1.2 depicts a small, two-layer neural network.

Computation in a neural network proceeds from layer to layer. Nodes in a layer are called *units*, and edges between units (which must necessarily be from different layers) are called *connections*. As in Figure 1.1b, connections between units in nonconsecutive layers can occur; these are called *skip connections*. Consequently, in order for information to propagate from L_k to L_{k+1}, we may require the outputs of all units from L_0 to L_{k-1}.

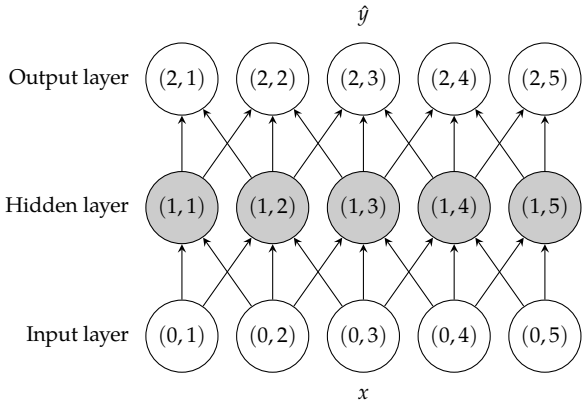


Figure 1.2: A small, two-layer neural network. The pair (k, j) is used to denote the jth unit in layer k.

Let $w := |L_k|$ denote the *width* of L_k. Rather than thinking about L_k as a subgraph of G, we associate L_k with a vector $z_k \in \mathbb{R}^w$. The value of component z_{ki} is given by the output of the of the ith unit from the left end of L_k. It is now evident that the vector x is transformed into $\hat{y}$ by a series of successive functions $\sigma_1, \ldots, \sigma_d$ given by

$$\begin{aligned} x =: z_0 &\overset{\sigma_1}{\longmapsto} z_1 \\ (z_0, z_1) &\overset{\sigma_2}{\longmapsto} z_2 \\ (z_0, z_1, z_2) &\overset{\sigma_3}{\longmapsto} z_3 \\ \vdots \qquad &\qquad \vdots \\ (z_0, \ldots, z_{d-1}) &\overset{\sigma_d}{\longmapsto} z_d := \hat{y}, \end{aligned} \tag{1.1}$$

where σ_k is realized by the units in L_k.

The jth unit in the input layer of a neural network simply returns the corresponding component x_j of the input vector x. On the other hand, each unit in the hidden and output layers forms a linear combination of the inputs from previous layers to which it is connected, and adds a bias parameter to the result. Suppose that L_k is not the input layer, so that $k > 0$. The jth unit in L_k is denoted by (k, j), and its activation a_{kj} is defined as

$$a_{kj} := \sum_{(l,i) \in \mathrm{Pa}(k,j)} w_{li \to kj} z_{li} + b_{kj}. \tag{1.2}$$

Here, $\mathrm{Pa}(k, j)$ denotes the parents of (k, j), which are the units that have edges directed toward (k, j). This notation allows us to easily generalize our discussion to layered networks that incorporate skip connections. The number $w_{li \to kj}$ is the *weight* associated with the connection $(l, i) \to (k, j)$, and b_{kj} the *bias* associated with (k, j). The quantity z_{li} is the output of unit (l, i), and is obtained by applying the unit's *activation function* $u_{li} : \mathbb{R} \to \mathbb{R}$ to the unit's activation a_{li}:

$$z_{li} = u_{li}(a_{li}). \tag{1.3}$$

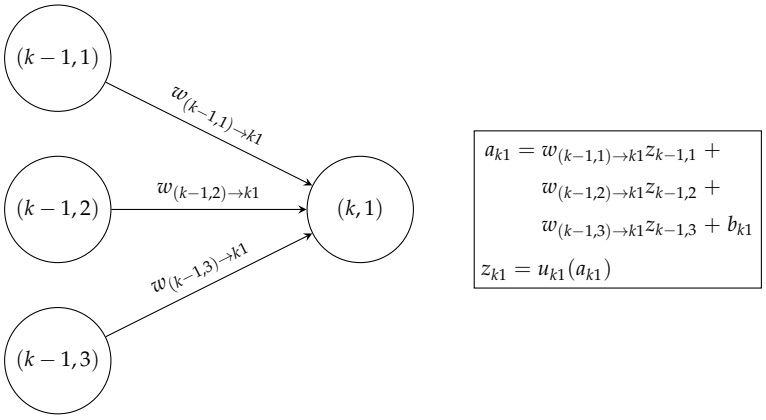


Figure 1.3: An illustration how the output of $(k, 1)$, a unit with three inputs from L_{k-1}, is computed. In this case, $\mathrm{Pa}(k, 1) = \{(k-1, 1), (k-1, 2), (k-1, 3)\}$. First, the activation a_{k1} is computed by forming a linear combination of the outputs of the units in $\mathrm{Pa}(k, 1)$ with the corresponding weights to (k, j), and adding the bias b_{k1}. Then, the output z_{k1} is determined by applying the activation function u_{k1} to a_{k1}.

Figure 1.3 shows how the activation of a unit with three parents is computed.

The process of computing the output vector z_k corresponds to applying the function σ_k. If $w := |L_k|$, then writing

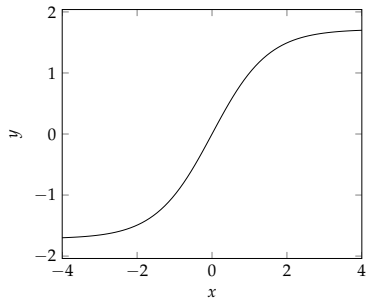
$$(z_0, \ldots, z_{k-1}) \overset{\sigma_k}{\longmapsto} z_k$$

is shorthand for saying that we compute z_k through w simultaneous applications of (1.2), followed by w simultaneous applications of (1.3). In order to compute the network's prediction $\hat{y} \in \mathbb{R}^m$ corresponding to an input $x \in \mathbb{R}^n$, we apply the functions $\sigma_1, \ldots, \sigma_d$ as described by (1.1). This process is called *forward propagation*.

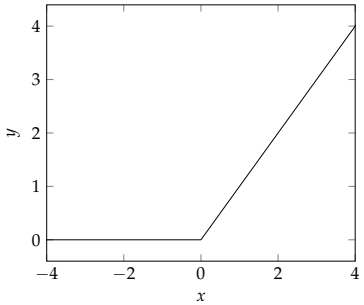
1.2 Activation Functions and Feature Learning

So far in our discussion, we have avoided making specific choices for the activation function $u_{kj} : \mathbb{R} \to \mathbb{R}$ associated with unit (k, j). Three of the most commonly-used activation functions in the literature are the identity, scaled tanh, and linear threshold functions (see Figure 1.4). (In the neural network literature, linear threshold functions are called "rectified linear units" (ReLU), an instance of specialized terminology that Mehryar Mohri despises.) Typically, all units in the same layer are associated with the same activation function. We therefore drop the redundant second index in the subscript of u_{kj}, and simply refer to the activation function as u_k.

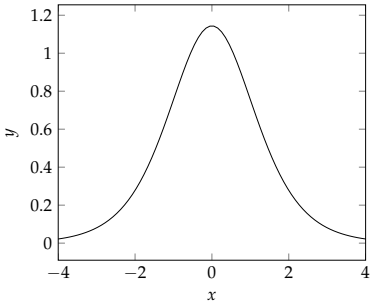
The choices for the constants a and b of the scaled tanh function are motivated by several reasons [LeCun et al., 1998b]. Firstly, the scaled tanh function is symmetric and approximately linear about the origin. If the inputs are decorrelated, our choices for a and b cause the variance of the scaled tanh function to be close to unity. This accelerates the convergence of optimization algorithms (see Section 1.5).



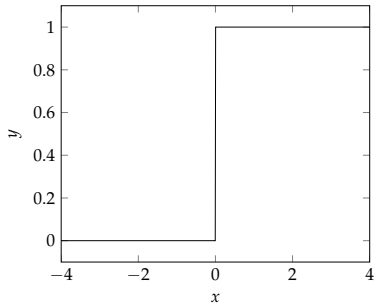
(a) Plot of $f(x) := a\tanh(bx)$, where $a := 1.7159$ and $b := 2/3$.



(b) Plot of $f(x) := \max(0, x)$.



(c) Plot of the derivative of the function in 1.4a.



(d) Plot of the derivative of the function in 1.4b.

Figure 1.4: Plots of two of the most common activation functions and their derivatives.

Secondly, the choices for these constant helps prevent the *vanishing gradient problem*. Using these constants causes the second derivative of the scaled tanh function to be maximized at ± 1. This is intentional: ± 1 are the binary target values that are typically used for classification in practice. Suppose that the asymptotes of the scaled tanh function had been at ± 1 instead of $\pm a$. Then the weights of the network would drastically increase, in order to produce the large activation values necessary to drive the outputs of tanh function to the target values at its asymptotes. The derivative of the scaled tanh function at these large activation values would be exponentially small (see Figure 1.4c). Thus, an optimization algorithm that used derivative information would be liable to "get stuck".

Finally, the choices for these constants allow us to interpret the output of the network as a measure of confidence towards its classification. When an input is near the decision boundary separating instances in two classes, we would like the network to output a small confidence value to reflect this uncertainty. If the asymptotes of the scaled tanh function were at ± 1, then the large activation values would force the outputs to one of the two extreme values, regardless of the certainty of the classification. Our choices for a and b avoid this problem.

Unlike the scaled tanh function, the identity function is typically only used for regression. One common configuration for regression involves alternating between layers using the scaled tanh and identity activation functions. For classification, we typically only

use the tanh or linear threshold functions. The adoption of the linear threshold function in the literature is relatively recent [Nair and Hinton, 2010], but many have found that it drastically reduces training time and improves generalization when compared to the scaled tanh function [Krizhevsky et al., 2012]. Despite these general guidelines, choosing the best activation functions for a particular problem can involve a certain degree of experimentation.

One of the principles underlying the effectiveness of neural networks is that of *hierarchical feature learning*, and is the focus of much of the deep learning research at NYU. A *feature* is a "simple" quantity derived from the input that is designed to identify one or more of the input's salient characteristics. For example, suppose that we wish to identify whether a given black and white image contains a human face. One potential feature is the difference between the sum of intensities of the pixels in the left half of the image and that of the right half of the image. In neural networks, a feature is a subset of activation values that become large when a given pattern is present in the input. This is a result of the weights and biases of the network being attuned to presence the pattern.

The success of neural networks in image recognition and acoustic modeling has been attributed to their ability to learn a *hierarchy* of features. In these applications, the location of a layer L_k in the network determines the relative scale of the patterns in the input captured by the features. In some sense, the lowest layers of the network "zoom in" to the input to capture low-level, local details, while the highest layers of the network "zoom out" to capture overarching, global patterns (see Figure 1.5). It is plausible that this phenomenon can occur, since each σ_k synthesizes the information from layers $L_0, \ldots, L_{k-1}$ to produce z_k.

The activation functions discussed so far are useful for learning the low- and mid-level features in the hierarchy. For learning high-level features, *radial basis functions* (RBFs) are sometimes more appropriate [LeCun et al., 1998a]. A radial basis function $\phi : \mathbb{R} \to \mathbb{R}$ is a continuous function whose value depends only on the *radius* from a prescribed center $c \in \mathbb{R}^n$. Suppose that we wish to approximate a function $f : \mathbb{R}^n \to \mathbb{R}$ whose values at the points $c_1, \ldots, c_m$ are known. By centering an RBF at each c_i, we can approximate f using the function $s : \mathbb{R}^n \to \mathbb{R}$ given by

$$s(x) := \sum_{i=1}^{m} \lambda_i \phi(\|x - c_i\|_2), \tag{1.4}$$

where $\lambda \in \mathbb{R}^m$ is an adjustable parameter vector. The most commonly-used RBFs are the Euclidean and Gaussian basis functions. These functions are given by

$$r \mapsto r \quad \text{and} \quad r \mapsto \exp(-kr^2/2),$$

respectively, where $k \in \mathbb{R}^+$.

Since RBFs are used for learning high-level features, they are usually embedded into one of the topmost layers of the network. As an example, suppose that we wish to classify 32×32 black and white images of handwritten digits into ten classes, where each class corresponds to a number from zero to nine. Suppose that L_k is a layer with 32^2

Figure 1.5: Visualization of the hierarchy of features in a fully-trained convolutional network for image classification, taken from Zeiler and Fergus [2014]. For each of the first five layers of the network, a selection of the highest activation values in the layer is shown alongside the corresponding input images. Note that in general, visualizing features in a meaningful way is very difficult.

units. By attaching a ten-unit layer L_{k+1} with RBF activation functions to L_k, we can generate ten scores from $z_k \in \mathbb{R}^{32\times 32}$. The ith score is given by $(z_{k+1})_i$, and measures the confidence that the input $x \in \mathbb{R}^n$ belongs to the ith class.

The idea is to get z_k to match a *prototype* for one of the classes as closely as possible. We can view the functions $\sigma_1, \ldots, \sigma_k$ as nonlinearly deconstructing and reassembling x to match one of these prototypes. The ten prototypes $c_1, \ldots, c_{10}$, are initialized by running a clustering algorithm, such as k-means, on the training sample. Each prototype is treated as the center of an RBF, and is subsumed into the weights of the network. The proximity of z_k to the ith prototype c_i is given by $\phi(\|x - c_i\|)$, and is a measure of confidence that x belongs to the ith class.

If ϕ is a *normalized RBF*, then the confidence scores are probabilities. A normalized RBF ϕ is the normalized form of another basis function ψ. Hence, the ith RBF is given by

$$\phi(x) := \frac{\psi(\|x - c_i\|)}{\sum_{j=1}^{10} \psi(\|x - c_j\|)}. \tag{1.5}$$

Now the ten outputs, $\phi(\|x - c_i\|)$, $i \in [1, 10]$, form a discrete probability distribution over the classes.

1.3 Classification and Regression

We have seen that a neural network is a biologically-inspired nonlinear function controlled by two adjustable vectors of parameters: the weights w and the biases b. Now consider a sample $S := \{(x_1, y_1), \ldots, (x_s, y_s)\}$, where each $x_k \in \mathbb{R}^n$ is an input vector, and each y_k is the target vector that we aim to predict when we are given x_k. We focus on two categories of tasks that we can perform using neural networks: classification and regression. For regression, we have $y_k \in \mathbb{R}^m$, so the output layer consists of a single unit.

For classification, two encoding schemes are possible. The first scheme can only be used for *unary classification*, in which we wish to classify x_k into one of several mutually-exclusive classes. In this scheme, each class is associated with an index, so $y_k, \hat{y}_k \in \mathbb{Z}$. These values are called a *place codes*. A special case of unary classification is *binary classification*, in which we seek to place x_k into one of two mutually-exclusive classes. When using place codes, the output layer consists of a single unit.

The second scheme can be used for unary classification as well as *multiclass classification*, in which the x_k can belong to one or more classes. This time, $y_k \in \{0, 1\}^m$ and $\hat{y}_k \in \mathbb{R}^m$. These values are called *distributed codes*. The component $(y_k)_i$ is one if x_k belongs to the ith class, and zero otherwise. On the other hand, $(\hat{y}_k)_i$ is a measure of confidence that x_k belongs to the ith class. Whether a larger value indicate increased or decreased confidence varies based on convention.

Distributed codes possess the advantage that they scale well to large numbers of classes, while place codes do not [LeCun et al., 1998b]. When using place codes, the single output unit of the network must assume one of finitely many values. If a large number of classes occur with nontrivial probabilities, then each unit in the penultimate

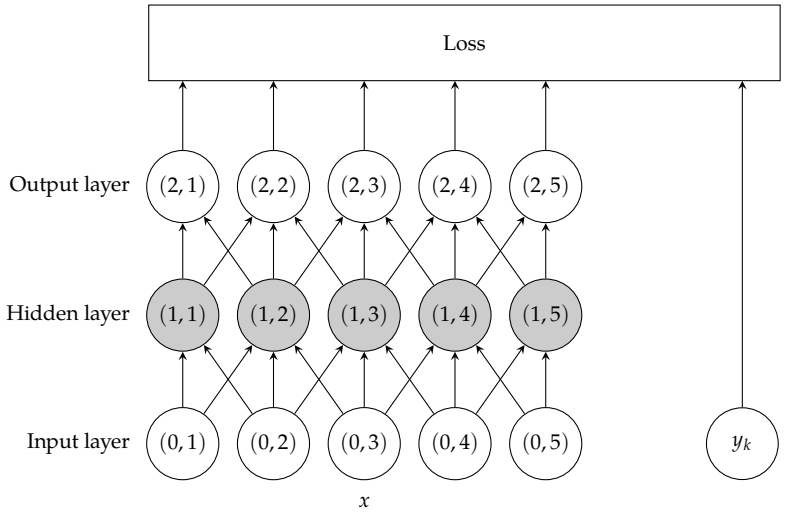


Figure 1.6: A visualization of how the kth instance (x_k, y_k) is fed into the two-layer neural network from Figure 1.2.

layer of the network must generate an output close to zero almost all of the time. This becomes increasingly difficult as the number of classes grows.

1.4 The Loss Function

Our goal is now to formulate the objective for the optimization algorithm used to calibrate the weights and biases. To do this, we need a function that measures how well the network's prediction $\hat{y}_k \in \mathbb{R}^m$ for $x_k \in \mathbb{R}^n$ matches the target value $y_k \in \mathbb{R}^m$. This function is called the *loss function* or *cost function* (see Figure 1.6). The choice of the loss function depends on the type of problem we would like to solve.

One way to arrive at the loss function is to assign a probabilistic interpretation to the network outputs [Bishop, 2006]. Let w and b be the vectors of weights and biases of the network. By choosing an explicit representation for the conditional distribution $p(y_k \mid x_k, w, b)$, we are led to a canonical loss function corresponding to this choice. Let $X = \{x_1, \ldots, x_s\}$ and $Y = \{y_1, \ldots, y_s\}$. If we assume that the instances $(x_k, y_k) \in S$ are iid, then we can apply the chain rule to write

$$p(Y \mid X, w, b) = \prod_{k=1}^{s} p(y_k \mid x_k, w, b).$$

The LHS of (1.4) is called the likelihood function. The canonical loss function is obtained by maximizing the likelihood function over the w and b. Maximizing the likelihood function is equivalent to minimizing its negative logarithm, which is given by

$$-\ln p(Y \mid X, w, b) = -\sum_{k=1}^{s} \ln p(y_k \mid x_k, w, b). \tag{1.6}$$

The LHS of (1.6) is called the negative log-likelihood function (NLL).

For regression, it is often appropriate to assume that

$$p(y_k \mid x_k, w, b) = N(y_k \mid \hat{y}_k, \beta^{-1}),$$

where N is the normal distribution, which for mean μ and variance σ^2 is defined as

$$N(x \mid \mu, \sigma^2) := \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

The value $\hat{y}_k$ is the network's prediction for x_k, and β is the precision (inverse variance) of the Gaussian noise. The corresponding NLL function is given by

$$\frac{\beta}{2}\sum_{k=1}^{s}(\hat{y}_k - y_k)^2 - \frac{s}{2}\ln\beta + \frac{s}{2}\ln(2\pi).$$

We discard the constant terms, which do not affect the minimization process. This leaves us with the sum-of-squares loss function, which is given by

$$E(w,b) := \frac{1}{2}\sum_{k=1}^{s}(\hat{y}_k - y_k)^2 = \sum_{k=1}^{s} e(\hat{y}_k, y_k), \tag{1.7}$$

where e is the per-instance loss function

$$e(\hat{y}_k, y_k) := \frac{1}{2}(\hat{y}_k - y_k)^2.$$

The sum-of-squares error function is generally inappropriate for classification. We adopt the use of distributed codes, so $y_k \in \{0,1\}^m$ and $\hat{y}_k \in \mathbb{R}^m$. We further suppose that the components of $\hat{y}_k$ determine a discrete probability distribution over the classes. For convenience, we define $y_{ki} := (y_k)_i$ and $\hat{y}_{ki} := (\hat{y}_k)_i$. Our assumption is that the conditional distribution is categorical, and given by

$$p(y_k \mid x_k, w, b) = \prod_{i=1}^{m}(\hat{y}_{ki})^{y_{ki}}.$$

The corresponding NLL function is the *multiclass cross-entropy function*, and does not have any constant terms that we can discard. So we define

$$E(w,b) := -\sum_{k=1}^{s}\sum_{i=1}^{m} y_{ki}\ln\hat{y}_{ki} = \sum_{k=1}^{s} e(\hat{y}_k, y_k),$$

where e is the per-instance loss function

$$e(\hat{y}_k, y_k) := -\sum_{i=1}^{m} y_{ki}\ln\hat{y}_{ki}.$$

In the special case where we are performing binary classification, with $y_k \in \{0, 1\}$ and $\hat{y}_k \in \mathbb{R}$, we can write

$$p(y_k \mid x_k, w, b) = (\hat{y}_k)^{y_k} \, (1 - \hat{y}_k)^{1 - y_k}$$

and

$$e(\hat{y}_k, y_k) := -\big(y_k \ln \hat{y}_k + (1 - y_k) \ln(1 - \hat{y}_k)\big).$$

The cross-entropy loss function is almost exclusively used with a special activation function for the output layer of the network. This activation function is the *softmax* function; for $z \in \mathbb{R}^m$, it is given by

$$\phi(z_i) := \frac{\exp(z_i)}{\sum_{j=1}^{m} \exp(z_j)}. \tag{1.8}$$

This is a normalized RBF of the form given by (1.5). It follows that using (1.8) as the output activation function will guarantee that the components of $\hat{y}_k$ determine a discrete probability distribution over the classes. For binary classification, (1.8) reduces to the *logistic sigmoid* function upon setting the component $\hat{y}_{k2}$, which corresponds to the second class, to zero:

$$\begin{aligned} \phi(\hat{y}_1) &= \frac{\exp(\hat{y}_1)}{\exp(\hat{y}_1) + \exp(\hat{y}_2)} \\ &= \frac{\exp(\hat{y}_1)}{\exp(\hat{y}_1) + 1} \\ &= \frac{1}{1 + \exp(-\hat{y}_1)}. \end{aligned}$$

Thus, the softmax function can be viewed as a generalization of the logistic sigmoid function to more than two classes.

1.5 Data Preprocessing and Initialization

Preprocessing the data can have drastic effects on convergence of the optimization algorithm used for learning. However, normalizing the features (the components of an input $x_k \in \mathbb{R}^n$) must be done with great care, so as not to destroy useful information encoded by the relative magnitudes of a feature. Suppose that it is equally important that we predict the target value for any input x_k from our training sample D. This is *not* the case, for instance, when we wish to predict whether the person corresponding to an input x_k has a rare disease. Then it can be beneficial to center and scale the features by subtracting the mean vector μ and scaling by the variance vector σ. Specifically, each input x_k is normalized by computing

$$\bar{x}_k := \frac{x_k - \mu}{\sigma}. \tag{1.9}$$

In general, it is recommended to *decorrelate* the features whenever possible [LeCun et al., 1998b]. Doing so decouples the updates to the individual weights of the network during the optimization process, which accelerates convergence. LeCun et al. [1998b] give an intuitive explanation for why decorrelating the inputs can be beneficial. Consider the extreme case where all components of the inputs are positive. Let $(1, j)$ be a unit in the second layer of the network, and let $\tilde{x}_k$ consist of the components of x_k corresponding to $\mathrm{Pa}\,(1, j)$ (see Figure 1.6). Let δ be the scalar error computed for $(1, j)$ using the techniques discussed in Section 1.6. Then a naïve first-order optimization algorithm will update the weights to $(1, j)$ by an amount proportional to $\delta \tilde{x}_k$. Thus, the weights to $(1, j)$ will either *all* increase or *all* decrease (depending on $\mathrm{sign}(\delta)$) after each update. This makes finding a good weight configuration difficult. A more sophisticated approach to decorrelating the inputs involves PCA [LeCun et al., 1998b, Krizhevsky et al., 2012].

The initial values of the weights can have a significant impact on both convergence speed and generalization error. Suppose that we use the scaled tanh activation function. In order to avoid vanishing gradient problem, we would like the activation values to remain in the range $[-1, 1]$, which is within the linear region of the scaled tanh function. One way of enforcing this is to require that the activation values have zero mean and unit variance [LeCun et al., 1998b].

Suppose that the inputs are centered and scaled according to (1.9), so that

$$\mathrm{E}(x_i) = 0 \quad \text{and} \quad \mathrm{Var}(x_i) = 1,$$

for each $i \in [1, n]$. From (1.2), we know that the activation of $(1, j)$ is given by

$$a_{1j} \coloneqq \sum_{(0,i) \in \mathrm{Pa}(1,j)} w_{0l \to 1j} x_{0l} + b_{1j}.$$

If the weights are chosen independently from the inputs, and both the weights and the biases have zero mean, then we have

$$\begin{aligned}
\mathrm{E}(a_{1j}) &= \mathrm{E}\left(\sum_{(0,i) \in \mathrm{Pa}(1,j)} w_{0l \to 1j} x_{0l} + b_{1j} \right) \\
&= \sum_{(0,i) \in \mathrm{Pa}(1,j)} \mathrm{E}(w_{0l \to 1j} x_{0l} + b_{1j}) \\
&= \sum_{(0,i) \in \mathrm{Pa}(1,j)} \big(\mathrm{E}(w_{0l \to 1j} x_{0l}) + \mathrm{E}(b_{1j}) \big) \\
&= \sum_{(0,i) \in \mathrm{Pa}(1,j)} \mathrm{E}(w_{0l \to 1j})\, \mathrm{E}(x_{0l}) = 0.
\end{aligned}$$

Thus, our requirement that the activation values have zero mean is satisfied.

We now describe how to enforce that $\mathrm{Var}(a_{1j}) = 1$. Let $f \coloneqq |\mathrm{Pa}(1, j)|$, and suppose that all weights have the same standard deviation σ. The value f is called the *fan-in*

of $(1, j)$. To solve for the value of σ that forces $\mathrm{Var}(a_{1j}) = 1$, we compute

$$\begin{aligned}
\mathrm{Var}(a_{1j}) &= \mathrm{Var}\left(\sum_{(0,i)\in \mathrm{Pa}(1,j)} w_{0l\to 1j} x_{0l} + b_{1j}\right) \\
&= \mathrm{Var}\left(\sum_{(0,i)\in \mathrm{Pa}(1,j)} w_{0l\to 1j} x_{0l} + \sum_{(0,i)\in \mathrm{Pa}(1,j)} b_{1j}\right) \\
&= \mathrm{Var}\left(\sum_{(0,i)\in \mathrm{Pa}(1,j)} w_{0l\to 1j} x_{0l}\right) \\
&= \sum_{(0,i)\in \mathrm{Pa}(1,j)} \left(\mathrm{Var}(w_{0l\to 1j})\,\mathrm{Var}(x_{0l}) - \mathrm{E}(w_{0l\to 1j})^2\,\mathrm{E}(x_{0l})^2\right) \\
&= \sum_{(0,i)\in \mathrm{Pa}(1,j)} \mathrm{Var}(w_{0l\to 1j})\,\mathrm{Var}(x_{0l}) \\
&= \sum_{(0,i)\in \mathrm{Pa}(1,j)} \sigma^2 = f\sigma^2.
\end{aligned}$$

So in order to enforce that $\mathrm{Var}(a_{1j}) = 1$, we should sample the weights from a distribution with zero mean and standard deviation given by

$$\sigma = \frac{1}{\sqrt{f}}. \tag{1.10}$$

The most common choices of distribution for weight initialization are the uniform and normal distributions [LeCun et al., 1998b, Krizhevsky et al., 2012].

The training dynamics of neural networks are vitally important but poorly understood. As the weights and biases of the network are calibrated using the optimization algorithm, the distributions of the activation values at each layer evolve. For deep networks with many layers, the vanishing gradient problem is greatly compounded. In particular, the greater the depth of a layer in the network, the greater the tendency of the activation values to cluster around zero or become very large [Glorot and Bengio, 2010]. This causes the vanishing gradient problem discussed in Section 1.2. The subtle numerical issues resulting from these complex interactions largely determines whether, when, and how a given optimization algorithm will converge.

The use of a good weight initialization scheme can help mitigate the undesirable saturation of the activation values [Glorot and Bengio, 2010]. Several authors have recently proposed new initialization schemes that they found were more effective than the one given here [Glorot and Bengio, 2010, Martens, 2010]. Another strategy is to incorporate a form of internal normalization into the network by adding *local response normalization* layers [Krizhevsky et al., 2012]. Part of the reason that training deep neural networks is so difficult is that these dynamics are so poorly understood. Development of a deeper understanding of what is happening in these situations is likely to have widespread practical consequences.

1.6 Computing Derivatives

All of the optimization algorithms that we will discuss require derivative information. We establish some useful notation before proceeding. Let e be the per-instance loss function, and let $\theta := (w, b) \in \mathbb{R}^p$ be the vector consisting of the weights and biases of the network. We now fix $(x_k, y_k) \in S$ and $\bar{\theta} \in \mathbb{R}^p$, and define $\varphi : \theta \mapsto e(\hat{y}_k, y_k)$, where $\hat{y}_k$ is the prediction of the network with parameters θ for x_k. The gradient vector and Hessian matrix of φ evaluated at $\bar{\theta}$ are given by

$$\bar{g} := (\nabla_\theta \varphi)(\bar{\theta}) \quad \text{and} \quad \overline{H} := (\nabla^2_\theta \varphi)(\bar{\theta}),$$

respectively. How can we compute $\bar{g}$ and $\overline{H}$ efficiently?

In Section 1.1, we introduced the forward propagation procedure. Given an input $x \in \mathbb{R}^n$, we evaluate the functions $\sigma_1, \ldots, \sigma_d$ corresponding to each layer in succession, culminating in the prediction $\hat{y} \in \mathbb{R}^m$ (see Figure 1.6). Each forward propagation requires $O(p)$ steps. An obvious way to compute the gradient is to use finite differences. For each $i \in [1, p]$, we calculate

$$(\bar{g})_i \approx \frac{\varphi(\bar{\theta} + \epsilon e_i) - \varphi(\bar{\theta})}{\epsilon},$$

where $\epsilon \ll 1$ is chosen to be small enough so that the difference is accurate, but not *so* small that the numerator only contains garbage after subtraction. This method is rather inefficient. Each component of $\bar{g}$ requires a finite difference, and each finite difference requires an additional forward propagation. This results in a total of $O(p^2)$ steps to compute the gradient. Using central differences for additional accuracy doubles the amount of work.

Fortunately, $\bar{g}$ can be computed in only $O(p)$ steps. We first compute the derivatives of u_d with respect to the parameters and activation values of the units in L_d. Using the chain rule, we can compute the derivatives of φ with respect to the parameters of L_d. The components of $\bar{g}$ corresponding to these parameters are now determined. To compute the same quantities for u_{d-1}, we require the derivatives of u_d. Proceeding with the computations now gives us the information necessary to compute the derivatives of φ with respect to the parameters of L_{d-1}. The components of $\bar{g}$ corresponding to these parameters are now determined. Repeating this process allows us to accumulate all of the components of $\bar{g}$ by visiting the layers of the network in *reverse order*. This procedure is hence called *backpropagation*; details can be found in Bishop [2006]. Backpropagation can be tricky to implement, so finite differences are often used to validate correctness.

Can we extend backpropagation to compute $\overline{H}$? Yes, but there is a problem. A small neural network has on the order of 10^6 parameters. Hence, $\bar{g}$ has 10^6 elements, and $\overline{H}$ has 10^{12} elements. So even for a small network, computing $\overline{H}$ is completely intractable. However, both $\operatorname{diag}(\overline{H})$ and $\overline{H}v$ for a vector $v \in \mathbb{R}^p$ can be computed efficiently using modified versions of backpropagation [Bishop, 2006]. Both of these quantities are exploited by various optimization algorithms.

1.7 Convolutional Networks

Convolutional networks are responsible for the success of models based on learning hierarchical representations (see Section 1.2). The area of machine learning that studies such models is called "deep learning". Of all of the methods commonly studied in deep learning, convolutional networks are the most practical: the top scoring methods for any benchmark involving image recognition or acoustic modeling invariably use them. Forward propagation's low computational cost (on the order of a few milliseconds per evaluation) allows convolutional networks to be used in low-power devices such as cameras and robots. Training convolutional networks using backpropagation has been studied extensively in the literature [LeCun et al., 1998b]. Due to space and time constraints, we do not discuss them here; see Ng et al. [2013] for an introduction.

2 Concepts from Statistical Learning Theory

2.1 Generalization and PAC-learning

Our initial goal when motivating the discussion for neural networks was to "learn" what a given function $f : \mathbb{R}^n \to \mathbb{R}^m$ does to transform $x \in \mathbb{R}^n$ to $y \in \mathbb{R}^m$. But what exactly does it mean to learn? So far, we have neglected to provide a definition for what it means for an algorithm "learn" a model to solve a given problem. A rigorous definition for learning is given in the field of statistical learning theory. This definition will have widespread consequences in shaping our criteria used to characterize optimization algorithms.

When performing binary classification, each training instance $(x_k, y_k) \in S$ is implicitly associated with a target *concept* that we wish to learn. This concept is a mapping c from the input space X to the output space $Y := \{0, 1\}$, such that $y_k = c(x_k)$. The set of all concepts associated with our classification task is called the *concept class*, and is denoted by C. A concept can be something simple, such as the set of axis-aligned rectangles in $\mathbb{R}^2$, or something exceedingly complex, such as the manifold of human faces in $\mathbb{R}^{256\times256}$, the space of 256×256, 1-bit, black and white images. We assume that all instances are drawn iid from some unknown distribution D over X.

Suppose that we are given a family of candidate functions F, such that $f : X \to \{0, 1\}$ for every $f \in F$. The task of our learning algorithm is to minimize the probability of a misclassification from an instance $x \sim D$. This leads to the following definition for *generalization error*.

Definition 1 (Generalization error [Mohri et al., 2012])**.** *Given a candidate function $f \in F$, a target concept $c \in C$, and an underlying distribution D, the* generalization error *or* risk *of f is defined as*

$$R(f) := \Pr_{x\sim D}(f(x) \neq c(x)) = \mathop{\mathrm{E}}_{x\sim D}(1\{f(x) \neq c(x)\}).$$

The generalization error is not a quantity that we can directly measure. Instead, we must work with the *empirical error*, the average number of misclassifications over our training sample $S \sim D$.

Definition 2 (Empirical error [Mohri et al., 2012])**.** *Given a candidate function $f \in F$, a target concept $c \in C$, and a sample $S \coloneqq \{(x_1, y_k), \ldots, (x_s, y_s)\} \sim D$, the* empirical error *or* empirical risk *of f is defined as*

$$\hat{R}(f) \coloneqq \frac{1}{s} \sum_{i=1}^{s} 1\{f(x) \neq c(x)\}.$$

There is a simple relationship between the empirical and generalization errors. Let $f \in F$. Using the linearity of expectations and the fact that S is drawn iid from D, it is easy to show that

$$\mathop{\mathrm{E}}_{S \sim D}(\hat{R}(f)) = R(f).$$

In other words, the generalization error of f is simply the average misclassification rate over samples drawn from D. We are now ready to present the definition of probably approximate correct (PAC) learning.

Definition 3 (PAC-learning [Mohri et al., 2012])**.** *Suppose that we are performing binary classification of inputs from an input space X, and let C be the associated concept class. Let $O(n)$ be an upper bound on the space complexity of an input $x \in X$. Given an algorithm A and a sample $S \sim D$, let $f_S \in F$ denote the candidate function found by A for S. Then C is PAC-learnable if there exists an algorithm A and a polynomial function $g : \mathbb{R}^4 \to \mathbb{R}$ such that for any $\epsilon, \delta \in (0, 1]$, for all distributions D over X, and for any target concept $c \in C$, the following holds for any sample $S \sim D$ of size $s \geq g(1/\epsilon, 1/\delta, n, \mathrm{size}(c))$:*

$$\Pr_{S \sim D}(R(f_S) \leq \epsilon) \geq 1 - \delta.$$

If A runs in $g(1/\epsilon, 1/\delta, n, \mathrm{size}(c))$, then C is said to be efficiently PAC-learnable. In this case, A is called a PAC-learning algorithm for C.

The definition for PAC-learning is verbose, but the underlying intuition is simple. Suppose that we have a target generalization error ϵ that we wish to attain with probability $1 - \delta$ on any sample $S \sim D$. Then C is PAC-learnable if the sample size required to meet our goals is bounded by a polynomial function of $1/\epsilon$ and $1/\delta$. Roughly speaking, A must find a function in F that is *approximately correct* with *high probability*. But it must be able to do this using a sample size that is not "too large". The analogous notion of space and time complexity for sample size requires this "high probability" bound.

One of the goals of statistical learning theory is to derive upper bounds on the generalization error in various situations. When such an upper bound applies to the particular situation at hand, we know that it is possible, at least in principle, to learn efficiently. Many of these upper bounds take the following form. Let $\delta \in (0, 1]$ be our target confidence, and let s be the size of the training samples drawn from D. Then with probability $1 - \delta$, it holds for any $f \in F$ that

$$R(f) \leq \hat{R}(f) + c\sqrt{\frac{\mathrm{capacity}(F)}{s}}, \tag{2.1}$$

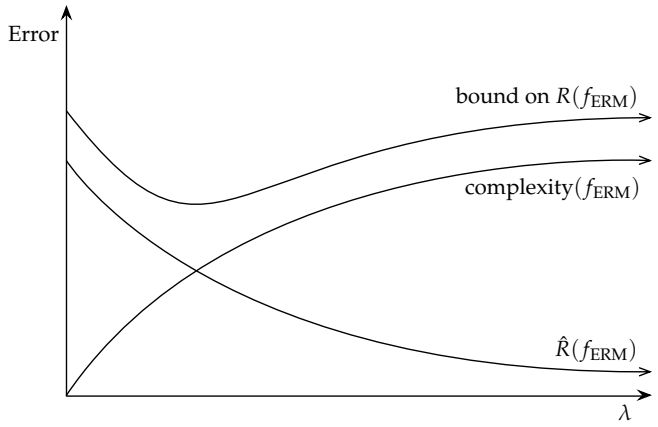


Figure 2.1: Visualization of the SRM procedure (adapted from Mohri et al. [2012]). The SRM procedure minimizes an upper bound of the generalization error that is a sum of the training error and the complexity term.

where $c \in \mathbb{R}^+$ and capacity(F) is a quantity that measures the complexity of our chosen family of functions. When we apply (2.1), we often subsume auxiliary constants into c. So two occurrences of c do *not* necessary represent the same constant value.

2.2 ERM vs SRM

The central idea of statistical learning theory is that learning is a *tradeoff* between minimizing the training error, and the complexity of the family of functions used to model C (see Figure 2.1). The former and latter quantities correspond to the first and second terms on the RHS of (2.1), respectively. A particularly illuminating way of expressing this tradeoff is as follows. Let

$$f^\star \coloneqq \arg\min_f R(f) \quad \text{and} \quad f_F^\star \coloneqq \arg\min_{f \in F} R(f).$$

In other words, $R(f^\star)$ is the best we can ever hope to do using an *arbitrary* function, while $R(f_F^\star)$ is the best we can do using a *fixed* function family F. Given a function $f \in F$, we define the *excess error* $\mathcal{E}$ as

$$\mathcal{E} \coloneqq R(f) - R(f^\star) = (R(f_F^\star) - R(f^\star)) + (R(f) - R(f_F^\star)) = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}}, \tag{2.2}$$

where

$$\mathcal{E}_{\text{app}} = R(f_F^\star) - R(f^\star) \quad \text{and} \quad \mathcal{E}_{\text{est}} = R(f) - R(f_F^\star)$$

The quantity $\mathcal{E}_{\text{app}}$ is called the *approximation error*, and measures how well functions in F can approximate functions in C. It is generally intractable, as we have no way of estimating it. The quantity $\mathcal{E}_{\text{est}}$ is called the *estimation error*, and measures the performance of f relative to that of the best-in-class hypothesis $f_F^\star$. It is easy to show that $\mathcal{E}_{\text{est}}$ is bounded

above by the second term on the RHS of (2.1). Equation 2.2 tells us that the quality of a candidate function f is determined by the sum of these errors.

Reducing the training error $\hat{R}(f)$ decreases $\mathcal{E}_{\text{est}}$, and *increasing* capacity(F) decreases $\mathcal{E}_{\text{app}}$. Suppose we are given a fixed training sample S, and wish to reduce $\mathcal{E}_{\text{app}}$. This requires increasing capacity(F), which enlarges the search space of functions for the optimization algorithm. Since the optimization algorithm must now look in a strictly larger space of functions, $\mathcal{E}_{\text{est}}$ is likely to increase. Let us try reduce $\mathcal{E}_{\text{est}}$. Since S is fixed, we cannot get more data; our only choice is to reduce capacity(F), which is likely to increase $\mathcal{E}_{\text{app}}$. We cannot continue to decrease one of the two errors without beginning to increasing the other (see Figure 2.1); this is the *approximation-estimation tradeoff*.

Minimizing only $\hat{R}(f)$ while ignoring capacity(F) is called *empirical risk minimization* (ERM), and is inadvisable on both theoretical and practical grounds. An alternative procedure that controls capacity(F) is called *structural risk minimization* (SRM). It works as follows. Let $c \in \mathbb{R}^+$, and define

$$F_c \coloneqq \{f \in F : \text{complexity}(f) \leq c\}, \tag{2.3}$$

where complexity(f) is a measure of complexity of f. Consequently, $F_a \subseteq F_b$ whenever $a, b \in \mathbb{R}^+$ such that $a < b$. ERM and SRM seek to find the functions given by

$$f_{\text{ERM}} \coloneqq \arg\min_{f \in F} \hat{R}(f) \quad \text{and} \quad f_{\text{SRM}} \coloneqq \arg\min_{\substack{c \in \mathbb{R}^+ \\ f \in F_c}} \hat{R}(f), \tag{2.4}$$

respectively. The SRM procedure finds a function in F that minimizes (2.1) (see Figure 2.1). For convenience, we can transform second equation in (2.4) into an unconstrained minimization problem by using the penalty method. This gives

$$f_{\text{SRM}} = \arg\min_{\substack{\lambda \in \mathbb{R}^+ \\ f \in F}} \left(\hat{R}(f) + \lambda \, \text{complexity}(f)\right). \tag{2.5}$$

The Lagrange multiplier λ called the *regularization parameter*, and penalizes more complex functions. We can now conduct our search over λ instead of over c. If H is a vector space, then we can define complexity$(f) \coloneqq \|f\|$ for some norm $\|\cdot\|$. Note that SRM involves finding the solution to several ERM subproblems, and is generally intractable. In practice, we find an *approximate* SRM solution by using cross-validation to choose $\lambda^\star$ from a finite sequence $\{\lambda_k\}_{k=1}^r$.

As an example, suppose that we are are performing least-squares regression, where $X = \mathbb{R}^n$ and $Y = \mathbb{R}$. Let S be a training sample consisting of s instances. We define

$$F \coloneqq \{x \mapsto w^t x + b : w \in \mathbb{R}^n, b \in \mathbb{R}\}, \tag{2.6}$$

and assume let E be the sum-of-squares error given by (1.7). According to the theory developed so far, choosing $f \coloneqq \arg\min_{f \in F} E(f)$ is highly inadvisable. Let complexity$(f) \coloneqq \|w\|_2$ for any $f \in F$. A better idea would be to try sequence of values for λ in (2.5), such as $\{\lambda_k\}_{k=-5}^5$, where $\lambda_k \coloneqq 2^k$. In this way, the results from statistical learning theory shape our perspective when solving practical problems.

3 Characterizing Optimization Algorithms

3.1 Goals of Optimization

We turn our focus to applying the results from Section 2 to characterize the quality of optimization algorithms for machine learning. To more closely model reality, we split $\mathcal{E}_{\text{est}}$ into two terms, yielding a threefold decomposition of $\mathcal{E}$. Let f_{ERM} be given by (2.4). During training, we may only require that the optimization algorithm produce a function $\hat{f} \in F$, whose empirical error is within a fixed tolerance $\rho \in \mathbb{R}^+$ of that of f_{ERM}:

$$\hat{R}(\hat{f}) - \hat{R}(f_{\text{ERM}}) \leq \rho. \tag{3.1}$$

The expectation of the LHS of (3.1) taken over a sample $S \sim D$ is called the *optimization error* [Bousquet and Bottou, 2008], and is defined as

$$\mathcal{E}_{\text{opt}} \coloneqq R(\hat{f}) - R(f_{\text{ERM}}).$$

We can now express the excess error of $\hat{f}$ as

$$\begin{aligned} \mathcal{E} &= (R(f_F^\star) - R(f^\star)) + (R(f_{\text{ERM}}) - R(f_F^\star)) + (R(\hat{f}) - R(f_{\text{ERM}})) && (3.2)\\ &= \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}. && (3.3) \end{aligned}$$

In practice, the quality of the solution to a machine learning problem is determined by monetary cost (which determines the maximum sample size $s_{\max}$) and human patience (which determines the limit on computation time $t_{\max}$). The definition for PAC-learning is one way of characterizing when one might be satisfied with the performance of a learning algorithm with respect to these constraints. Let $t(F, \rho, s)$ represent the time required by the optimization algorithm to find a function $\hat{f} \in F$ satisfying $\mathcal{E}_{\text{opt}} \leq \rho$, given a sample of size s. Minimizing the excess error in (3.3) subject to the constraints on $s_{\max}$ and $t_{\max}$ yields the following meta-optimization problem [Bousquet and Bottou, 2008]:

$$\begin{aligned} \underset{F,\rho,s}{\text{minimize}} \quad & \mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \\ \text{subject to} \quad & s \leq s_{\max} \\ & t(F, \rho, s) \leq t_{\max}. \end{aligned} \tag{3.4}$$

Table 3.1 shows how the errors and computing time are affected when capacity(F), ρ, and s are increased.

As stated in Bousquet and Bottou [2008], "the solution of the optimization program (3.4) depends critically on which budget constraint is active: [the] constraint $s < s_{\max}$ on the number of examples, or [the] constraint $t < t_{\max}$ on the training time". A *small-scale* problem is one that is constrained by the sample size. In this case, the computation time is not limited, so we can make $\mathcal{E}_{\text{opt}}$ negligible by choosing ρ sufficiently small. Using the fact that $\mathcal{E}_{\text{est}}$ is bounded above by the second term of (2.1), we have

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} \leq \mathcal{E}_{\text{app}} + c\sqrt{\frac{\text{capacity}(F)}{s}}. \tag{3.5}$$

	capacity(F)	s	ρ
$\mathcal{E}_{\text{app}}$	$\downarrow$	—	—
$\mathcal{E}_{\text{est}}$	$\updownarrow$	$\updownarrow$	—
$\mathcal{E}_{\text{opt}}$	$\uparrow$	$\uparrow$	$\downarrow$
t	$\uparrow$	$\uparrow$	$\downarrow$

Table 3.1: The effects of increasing capacity(F), s, and ρ on the three errors and computing time. The symbol $\updownarrow$ indicates that either an increase or a decrease is possible.

Hence minimizing $\mathcal{E}$ boils down to the discussion in Section 2.2. A *large-scale* problem is one that is constrained by the maximum computing time. In this case, performing approximate optimization by choosing $\rho > 0$ improves generalization, because we can process more training instances in the allotted time. Using (2.1), it is easy to show that

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \leq \mathcal{E}_{\text{app}} + \rho + c\sqrt{\frac{\text{capacity}(F)}{s}}. \tag{3.6}$$

Unfortunately, the bounds (3.5) and (3.6) are too pessimistic, and do not accurately reflect reality. In order to proceed with our analysis, we make some simplifying assumptions. First, we assume that F is the family of affine functions given by (2.6). In this special case, the generalization bound (2.1) simplifies to

$$R(f) \leq \hat{R}(f) + c\sqrt{\frac{n}{s}}.$$

A much sharper bound on the excess error can now be applied [Bousquet and Bottou, 2008]. It is given by

$$\mathcal{E} = \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \leq c\left(\mathcal{E}_{\text{app}} + \left(\frac{n}{s}\log\frac{s}{n}\right)^{\alpha} + \rho\right), \tag{3.7}$$

where $\alpha \in [1/2, 1]$ is a constant that depends on a variance bound on the loss function. Since the three components of the excess error should decrease at approximately the same rate [Bousquet and Bottou, 2008], we make the additional assumption that

$$\mathcal{E} \approx \mathcal{E}_{\text{app}} \approx \mathcal{E}_{\text{est}} \approx \mathcal{E}_{\text{opt}} \approx \left(\frac{n}{s}\log\frac{s}{n}\right)^{\alpha} \approx \rho. \tag{3.8}$$

3.2 Analysis of Optimization Algorithms

We now investigate the properties of four specific optimization algorithms. Assume that our empirical loss function E is convex and twice differentiable (e.g. the sum-of-squares loss function). For convenience, we let $\theta \in \mathbb{R}^{2n}$, where the first n components of θ refer to a weight vector $w \in \mathbb{R}^n$ and the last n components of θ refer to a bias vector $b \in \mathbb{R}^n$. Thus $\theta = (w, b)$, for some $w, b \in \mathbb{R}^n$. We define $E(\theta) := E(w, b)$. Since E is convex, it

has a unique minimum in terms of θ. Let $\theta_{\text{ERM}} \coloneqq (w_{\text{ERM}}, b_{\text{ERM}})$ be the parameter vector corresponding to f_{ERM}. For a given input $x_k \in \mathbb{R}^n$, we define $\hat{y}_k^{\text{ERM}} \coloneqq w_{\text{ERM}} x_k + b_{\text{ERM}}$.

The Hessian matrix H_{ERM} and gradient covariance matrix G_{ERM} evaluated at θ_{ERM} play an important role in our analyses [Bousquet and Bottou, 2008]. They are defined as

$$H_{\text{ERM}} \coloneqq (\nabla_\theta^2 E)(\theta_{\text{ERM}}) = \frac{1}{s} \sum_{i=1}^{s} (\nabla_\theta^2 e)(\hat{y}_k^{\text{ERM}}, y_k)$$

and

$$\begin{aligned} G_{\text{ERM}} &\coloneqq ((\nabla_\theta E)(\theta_{\text{ERM}}))\, ((\nabla_\theta E)(\theta_{\text{ERM}}))^t \\ &= \frac{1}{s} \sum_{i=1}^{s} \Big((\nabla_\theta e)(\hat{y}_k^{\text{ERM}}, y_k) \Big) \Big((\nabla_\theta e)(\hat{y}_k^{\text{ERM}}, y_k) \Big)^t, \end{aligned}$$

where e is the per-instance loss function associated with E. We assume that there exist constants $\lambda_{\max} \geq \lambda_{\min} > 0$ and $\nu > 0$ such that for any $\delta \in (0, 1]$, there exists a sample size $s \in \mathbb{N}$ for which

$$\text{tr}\left(G_{\text{ERM}} (H_{\text{ERM}})^{-1} \right) \leq \nu \quad \text{and} \quad \text{Sp}(H_{\text{ERM}}) \subset [\lambda_{\min}, \lambda_{\max}],$$

with probability at least $1 - \delta$. The condition number $\kappa \coloneqq \lambda_{\max} / \lambda_{\min}$ determines the difficulty of the optimization problem.

The optimization algorithms used in machine learning either operate on *full batches* or *mini-batches*. An full batch algorithm will only apply an update to θ after processing the *entire* training sample S. The derivative information computed by batch algorithms is averaged over the batch of instances. A mini-batch algorithm applies an update to θ every r iterations, where $r|s$. A *stochastic* or *online* algorithm uses a mini-batch size of one. The first two algorithms that we present are full batch algorithms, while the last two algorithms are stochastic.

Mini-batch algorithms are often several times faster than full batch algorithms. This is because mini-batch algorithms are better able to exploit redundancy in the training sample S. LeCun et al. [1998b] give an explanation for why this happens. Suppose that $|S| = 1000$, and consists of the same 100 instances replicated ten times in the same order. The average of the gradient over S is equal to the average of the gradient over the first 100 instances. A full batch algorithm will compute the same gradient ten times before applying a single update. In this time, an algorithm with a mini-batch size of 100 would apply ten updates. While duplicate instances are rare in practice, there are often clusters of instances that are similar in some sense.

Following the precedent of Stephen Wright, we refer to the algorithms below as *gradient update* algorithms rather than as *gradient descent* algorithms, because the search directions used by the update rules are not necessary descent directions.

- The gradient update (GU) algorithm uses the update rule

$$\theta_{k+1} \coloneqq \theta_k - \eta (\nabla_\theta E)(\theta_k) = \theta_k - \frac{\eta}{s} \sum_{i=1}^{s} (\nabla_\theta e)(\hat{y}_k^{\text{ERM}}, y_k)$$

Algorithm	Cost of one iteration	Iterations to reach ρ	Time to reach accuracy ρ	Time to reach $\mathcal{E} \leq c(\mathcal{E}_{\text{app}} + \epsilon)$
GU	$O(ns)$	$O\left(\kappa \log \frac{1}{\rho}\right)$	$O\left(ns\kappa \log \frac{1}{\rho}\right)$	$O\left(\frac{n^2\kappa}{\epsilon^{1/\alpha}} \log^2 \frac{1}{\epsilon}\right)$
2GU	$O(n^2 + ns)$	$O\left(\log\log \frac{1}{\rho}\right)$	$O\left((n^2 + ns) \log\log \frac{1}{\rho}\right)$	$O\left(\frac{n^2}{\epsilon^{1/\alpha}} \log \frac{1}{\epsilon} \log\log \frac{1}{\epsilon}\right)$
SGU	$O(n)$	$\frac{\nu\kappa^2}{\rho} + o\left(\frac{1}{\rho}\right)$	$O\left(\frac{n\nu\kappa^2}{\rho}\right)$	$O\left(\frac{n\nu\kappa^2}{\epsilon}\right)$
2SGU	$O(n^2)$	$\frac{\nu}{\rho} + o\left(\frac{1}{\rho}\right)$	$O\left(\frac{n\nu}{\rho}\right)$	$O\left(\frac{n\nu}{\epsilon}\right)$

Table 3.2: Asymptotic rates for the four algorithms discussed (adapted from Bousquet and Bottou [2008]).

When $\eta = 1/\lambda_{\max}$, this algorithm requires $O(\kappa \log(1/\rho))$ updates to reach accuracy ρ.

- The second-order gradient update (2GU) algorithm uses the update rule

$$\theta_{k+1} := \theta_k - (H_{\text{ERM}})^{-1}(\nabla_\theta E)(\theta_k) = \theta_k - \frac{(H_{\text{ERM}})^{-1}}{s} \sum_{i=1}^{s} (\nabla_\theta e)(\hat{y}_k^{\text{ERM}}, y_k)$$

  Note that the Hessian is *not* evaluated at each θ_k: we assume that we know H_{ERM} in advance, and use the same matrix at each iteration. In general, this algorithm requires $O(\log\log(1/\rho))$ iterations to reach accuracy ρ.

- Given an instance $(x_k, y_k) \in S$, the stochastic gradient update (SGU) algorithm uses the update rule

$$\theta_{k+1} := \theta_k - \frac{\eta}{k}(\nabla_\theta e)(\hat{y}_k^{\text{ERM}}, y_k).$$

  If $\eta = 1/\lambda_{\min}$, this algorithm requires $\nu\kappa^2/\rho + o(1/\rho)$ iterations, on average, to reach accuracy ρ.

- Given an instance $(x_k, y_k) \in S$, the second-order stochastic gradient update algorithm (2SGU) uses the update rule

$$\theta_{k+1} := \theta_k - \frac{(H_{\text{ERM}})^{-1}}{k}(\nabla_\theta e)(\hat{y}_k^{\text{ERM}}, y_k).$$

  This algorithm requires $\eta/\rho + o(1/\rho)$ iterations, on average, to reach accuracy ρ. Note that, unlike 2GU, the use of second-order information does not weaken the dependence on ρ.

The asymptotic rates for the four algorithms that were presented are given in Table 3.2. The rates in the first column are based on the sizes of the vectors and matrices involved in the corresponding update rules. The rates in the second column are taken directly from our discussion. To obtain the rates in the third column, we multiply the corresponding rates in the first and second columns. One of our assumptions in (3.8) is that

$$\rho \approx \left(\frac{n}{s} \log \frac{s}{n}\right)^{\alpha}. \tag{3.9}$$

Substituting this value of ρ into the bound (3.7) and comparing the result to $c(\mathcal{E}_{\text{app}} + \epsilon)$ shows that

$$\rho \approx \epsilon. \tag{3.10}$$

Using (3.9) and (3.10), it can then be shown that

$$s \approx \frac{n}{\epsilon^{1/\alpha}} \log \frac{1}{\epsilon}. \tag{3.11}$$

Substituting (3.10) and (3.11) into the rates in the third column gives the rates in the fourth column.

Table 3.2 contains a wealth of information regarding the properties of the four optimization algorithms. This information leads to the following conclusions for large-scale problems [Bousquet and Bottou, 2008]:

- Stochastic algorithms are more appropriate when there is less need to optimize accurately. This is the case in large-scale problems, or when the estimation rate α is small. Note that the performance of the stochastic algorithms does not depend on the estimation rate α.
- Hastily incorporating second-order information into gradient update algorithms may only bring small improvements in $\mathcal{E}$. When minimizing $\mathcal{E}$, all four algorithms are dominated by the polynomial factor in $1/\epsilon$. The second-order methods shown here only improve the logarithmic factors in $1/\epsilon$.
- Stochastic algorithms have the best generalization performance, despite being the worst optimization algorithms when viewed in the traditional sense. When minimizing $\mathcal{E}$, the batch algorithms have a dependence on α as well as the logarithmic factor in $1/\epsilon$.

For small-scale problems, $\mathcal{E}_{\text{opt}}$ is negligible, so these conclusions do not necessarily apply.

One caveat is that stochastic algorithms are more vulnerable than batch algorithms to ill-conditioning. The results for both stochastic algorithms depend on the constant ν. The factor of κ in the results for GU becomes κ^2 in the results for SGU. As H_{ERM} becomes increasingly ill-conditioned, both κ and ν are liable to increase rapidly. One of the goals of decorrelating the inputs (Section 1.5) is to prevent H_{ERM} from having large eigenvalues [LeCun et al., 1998b]. In light of the results from Table 3.2, the importance of decorrelating the inputs becomes even more apparent.

4 Optimization Algorithms for Neural Networks

4.1 Challenges of Large-Scale Optimization

Training convolutional networks is prototypical of the large-scale problems discussed in the previous section. Our guidelines suggest that for such problems, optimization algorithms based on steepest descent should use mini-batches, and carefully evaluate the

tradeoffs of incorporating second-order information. Almost all optimization algorithms used to train convolutional networks, or deep neural networks of other kinds, abide by these constraints.

First-order algorithms must effectively deal with the vanishing gradient problem and the issue of "pathological curvature" [Martens, 2010]. In Section 1.5, we mentioned that activation values in deeper layers of a network have a greater tendency to cluster around zero or very large values. Since the gradients in one layer depend multiplicatively on those in the next (see Section 1.6), the vanishing gradient problem usually affects the lowest levels most severely. We also saw that when features of the input are correlated, the weight updates are no longer decoupled. Both of these phenomena suggest that *each* weight and bias in the network should be given its own learning rate [LeCun et al., 1998b], something that is usually not done for traditional optimization problems.

The asymptotic rates for the stochastic algorithms in Table 3.2 suggest vulnerability to ill-conditioning of the Hessian matrix, especially when second-order information is not used. The problem is in fact much more serious than this. Suppose that each weight and bias in the network is given its own learning rate. By viewing the full batch gradient update rule as a discrete time dynamical system, it can be shown that the optimal learning rate for the ith parameter θ_i scales the corresponding eigenvalue of the Hessian matrix to unity [LeCun et al., 1998b]. That is, the optimal learning rate for θ_i is given by

$$\eta_i^\star := \frac{r}{\lambda_i},$$

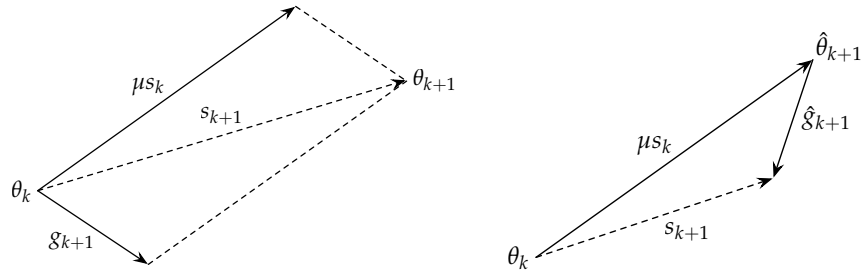
where r is the size of the mini-batch. Intuitively, this means that we would like to amplify η_i along directions of low curvature, and decay η_i along directions of high curvature.

Even for a small network, computing the spectrum of the Hessian matrix is completely intractable. LeCun et al. [1998b] give several techniques for estimating the maximum eigenvalue of the Hessian matrix, but dividing all of the learning rates by this quantity would unnecessarily retard progress. This means that first-order algorithms must somehow deal with pathological curvature without access to second-order information.

Unsurprisingly, the most successful ones do this by making innovative use of the information that is available. These algorithms enjoy widespread use, and can solve a variety of challenging tasks. However, there is a small but definite gap in the performance between first- and second-order algorithms [Martens, 2010, Ngiam et al., 2011, Sutskever et al., 2013]. Second-order algorithms seem to make effective use of the curvature information from Hessian matrix in order to explore regions of the parameter space that are otherwise inaccessible. This gain in performance comes at a cost: second-order algorithms are considerably more complex, and practitioners are less willing to implement them.

4.2 Momentum

Accelerated gradient methods have been the subject of much of the recent work in convex optimization theory [Sutskever et al., 2013]. Much of this work has involved the analysis



(a) The CM update rule. (b) The NAG update rule.

Figure 4.1: Illustration of the two momentum-based update rules (adapted from Sutskever et al. [2013]).

of classical momentum (CM) and Nesterov accelerated gradient (NAG). For the purposes of our discussion, let E denote the loss function, θ the parameters of the model, and k the current iteration. The value of θ at convergence is denoted by $\theta^\star$. Lastly, we define

$$g_k \coloneqq (\nabla_\theta E)(\theta_k) \quad \text{and} \quad H_k \coloneqq (\nabla^2_\theta E)(\theta_k).$$

The CM update rule is given by

$$\begin{aligned} s_{k+1} &\coloneqq \mu s_k - \eta g_k \\ \theta_{k+1} &\coloneqq \theta_k + s_{k+1}, \end{aligned}$$

where $\mu \in [0, 1]$ is the *momentum* parameter (see Figure 4.1a). The idea behind momentum is to "smooth out" the highly oscillatory path of stochastic updates using an exponentially-decaying average of past gradients. This dampens the wild oscillations that are characteristic of the unstable nature of SGU algorithms. It also amplifies the gradient along directions of low curvature. In a deterministic setting, CM requires $\sqrt{\kappa}$ times fewer iterations than steepest descent to reach a solution of accuracy ρ, where κ is defined as in Table 3.2 [Sutskever et al., 2013]. Hence, CM reduces the vulnerability of first-order algorithms to ill-conditioning of the Hessian matrix.

NAG differs from CM in that it *first* moves along the direction given by the momentum term, and *then* applies a correction based on the gradient at that point (see Figure 4.1b). This subtle change makes a large difference in practice. The NAG update rule can be written as [Sutskever et al., 2013]:

$$\begin{aligned} \hat{\theta}_{k+1} &\coloneqq \theta_k + \mu s_k \\ s_{k+1} &\coloneqq \mu s_k - \eta \hat{g}_{k+1} \\ \theta_{k+1} &\coloneqq \theta_k + s_{k+1}. \end{aligned}$$

To see why NAG can be more effective than CM, suppose that E briefly decreases along s_k, but soon begins to rapidly increase. Subtracting $-\eta \hat{g}_{k+1}$ from the momentum term helps us "steer away" from this poor update, whereas subtracting $-\eta g_k$ does not (see Figure 4.1).

The correction applied by CM will only occur at the next iteration, by which time we will have already moved along this bad update. In a deterministic setting for a smooth, convex function, NAG achieves a global convergence rate of $O(1/k^2)$, compared to convergence rate of $O(1/k)$ for steepest descent [Sutskever et al., 2013].

Figure 4.1 suggests that there is a relationship between CM and NAG. Suppose that s_k is an eigenvector of a quadratic function with positive-definite Hessian matrix H. Then the NAG update is equivalent to a CM update with momentum parameter

$$\mu_{\text{CM}} := \mu_{\text{NAG}}(1 - \eta\lambda),$$

where λ is the eigenvalue of H corresponding to s_k and $\eta \in [0, 1/\lambda]$ [Sutskever et al., 2013]. If $\eta \approx 0$, then $\eta\lambda \approx 0$ for all $\lambda \in \text{Sp}(H)$, and the NAG update is approximately the same as the CM update. On the other hand, if $\eta \approx 1/\lambda$, then $\mu \approx 0$, so NAG uses a smaller effective momentum than CM. This shows that NAG is able to automatically throttle the momentum parameter along directions of high curvature, adding a measure of stability that is absent in CM.

When training deep neural networks, the *local* convergence properties of optimization algorithms are largely irrelevant. Since the parameter space for deep neural networks is so vast and complex, the "transitory period" [Sutskever et al., 2013] before local convergence sets in lasts for almost the entire duration of the optimization process. It is in this setting that the advantage of using momentum becomes clear.

Suppose that we are optimizing a smooth, convex function $f : \mathbb{R} \to \mathbb{R}$ using stochastic first-order algorithms. Without momentum (i.e. using vanilla SGU), we have a convergence rate of $O(\ell/k + \sigma/\sqrt{k})$, where ℓ is the Lipschitz constant and σ is the variance in the gradient estimate. With momentum, the convergence rate is $O(\ell/k^2 + \sigma/\sqrt{k})$ [Sutskever et al., 2013]. Since f is convex, the first term is dominant during the transitory period, and momentum gives us a clear advantage. As $\theta_k \to \theta^\star$, the second term takes effect, and both algorithms have approximately the same performance.

4.3 First-Order Optimization Algorithms

All of the successful first-order optimization algorithms for neural networks make use momentum in some form (see Figure 4.2). The first algorithm that we present is abbreviated DH (for "diagonal Hessian"), and is based on a diagonal approximation to the Levenberg-Marquardt algorithm [LeCun et al., 1998b]. In the spirit of CM, DH updates an exponentially decaying average $\bar{h}_k \in \mathbb{R}^n$ of the diagonal of the Hessian matrix. We define the division and absolute value operations for vectors so that they are carried out elementwise. The update rule for DH is given by

$$\begin{aligned}
\bar{h}_{k+1} &:= (1 - \tau)\bar{h}_k + \tau|\text{diag}(H_k)| + \epsilon \\
\theta_{k+1} &:= \theta_k - \frac{\eta}{\bar{h}_{k+1}}\, g_k,
\end{aligned}$$

where the *decay* parameter $\tau \in [0, 1]$ controls the "momentum" in the running average, and $\epsilon \in \mathbb{R}^n$ is a nonnegative constant vector chosen to prevent the effective learning

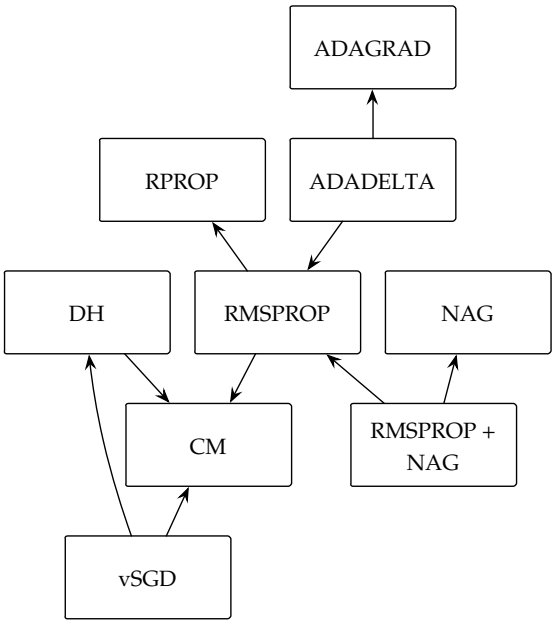


Figure 4.2: Hierarchy of the first-order optimization techniques and algorithms. An arrow from one block to another means that the algorithm represented by the first block is based on the one represented by the second. Here, CM stands for "classical momentum", NAG for "Nesterov accelerated momentum", and DH for "diagonal Hessian".

rate from growing too large. Using $\operatorname{diag}(H_k)$ to scale the individual learning rates can outperform carefully annealed versions of SGU that use global learning rates [LeCun et al., 1998b].

The "variance-based stochastic gradient descent" (vSGD) also uses $\operatorname{diag}(H_k)$, but incorporates several other features for added robustness. It is derived by optimizing the expected value of the per-sample loss based on a local quadratic model with diagonal Hessian matrix [Schaul et al., 2013]. Let $e \in \mathbb{R}^n$ be the constant vector of ones, and let $\odot$ denote the elementwise product operation. We define exponentiation for vectors so that the operation is carried out elementwise. The vSGD algorithm maintains exponentially decaying averages of the gradient, squared gradient, and Hessian diagonal vectors $\bar{g}, \bar{v}, \bar{h} \in \mathbb{R}^n$. These vectors are updated according to the rules

$$\begin{aligned}
\bar{g}_{k+1} &\coloneqq (e - \tau^{-1}) \odot \bar{g}_k + \tau^{-1} \odot g_k \\
\bar{v}_{k+1} &\coloneqq (e - \tau^{-1}) \odot \bar{v}_k + \tau^{-1} \odot (g_k)^2 \\
\bar{h}_{k+1} &\coloneqq (e - \tau^{-1}) \odot \bar{h}_k + \tau^{-1} \odot \operatorname{diag}(H_k) + \epsilon,
\end{aligned}$$

where $\epsilon \in \mathbb{R}^n$. The learning rate and decay parameters $\eta, \tau \in \mathbb{R}^n$ are updated according

to the rules

$$\eta_{k+1} \coloneqq \frac{(\bar{g}_{k+1})^2}{\bar{h}_{k+1} \odot \bar{v}_{k+1}} = (\bar{h}_{k+1})^{-1} \odot \frac{(\bar{g}_{k+1})^2}{\bar{v}_{k+1}}$$
$$\tau_{k+1} \coloneqq \left(e - \frac{(\bar{g}_{k+1})^2}{\bar{v}_{k+1}}\right) \odot \tau_k + 1.$$

The factor $(\bar{g}_{k+1})^2/\bar{v}_{k+1}$ in the update to η scales the effective learning rate to compensate for the vanishing gradient problem, while the factor $(\bar{h}_{k+1})^{-1}$ attempts to correct for pathological curvature. The update to τ is chosen such that the momentum increases when the step size decreases, and decreases when the step size increases. Lastly, the update to θ is given by

$$\theta_{k+1} \coloneqq \theta_k - \eta_{k+1} \odot g_k.$$

While the vSGD update rule is complex, it is designed to work effectively with a minimal amount of human effort. Previously proposed methods had on the order of ten or a hundred hyperparameters, and were often sensitive to the values chosen for them. Additionally, first-order methods often require careful annealing of the learning rate to yield acceptable results, or sometimes to converge at all [Krizhevsky et al., 2012]. The vSGD algorithm only has a single parameter that needs to be set (hence the name of the paper "No More Pesky Learning Rates"), and the learning rate automatically converges to zero as the value of the loss function becomes optimal. If the objective function is nonstationary and the data changes, vSGD will ramp up the learning rate in response [Schaul et al., 2013]. This makes vSGD suitable for online learning problems.

The vSGD algorithm works best when initialized with a *slow start* heuristic [Schaul et al., 2013]. By this, we mean that the parameter updates should be kept small artificially until the exponentially decaying averages become sufficiently accurate. We first seed the algorithm by computing the relevant averages over a "handful" of instances (the paper suggests $10^{-3} \cdot s$, where s is the size of the training sample). In order to keep the initial parameter updates small, the resulting value of $\bar{v}_1$ is "overestimated" by a factor C, which the paper recommends setting to $n/10$, where n is the dimension of the input space. This is the only parameter that needs to be set. Using this heuristic, the vSGD algorithm surpasses the performance of finely-tuned implementations of SGU and previously proposed adaptive learning algorithms [Schaul et al., 2013].

The "root mean square propagation" (RMSPROP) algorithm is based on a *full-batch* gradient update algorithm called "resilient backpropagation" (RPROP) [Hinton et al., 2014]. The RPROP algorithm maintains a separate learning rate for each parameter, and uses the update rule

$$\theta_{k+1} \coloneqq \theta_k - \eta_{k+1} \odot \operatorname{sign}(g_k),$$

where $\eta_1 = \eta_2 = e$, and

$$(\eta_{k+1})_i \coloneqq \begin{cases} \max(\eta_{\max}, \gamma_e(\eta_k)_i) & \text{if } \operatorname{sign}((g_{k-1})_i) = \operatorname{sign}((g_k)_i) \\ \min(\eta_{\min}, \gamma_c(\eta_k)_i) & \text{otherwise.} \end{cases}$$

The constants $\gamma_c, \gamma_e \in \mathbb{R}^+$ are the *contraction* and *expansion* factors, respectively. They must satisfy $\gamma_c < 1$ (e.g. 0.5) and $\gamma_e > 1$ (e.g. 1.2). The constants $\eta_{\min}, \eta_{\max} \in \mathbb{R}^+$ limit the growth of the learning rates so that they do not become "too large" or "too small"; typical settings are $\eta_{\min} = 10^{-6}$ and $\eta_{\max} = 50$.

The RPROP algorithm's strategy of only using the signs of the gradient vector is also its limitation: RPROP cannot be used with mini-batches. The signs of the gradient vector are only meaningful if the gradient over the current batch is a good approximation of the true gradient of the loss function E, which is evaluated over the entire training sample. To see how things can go wrong, suppose that RPROP were used as a stochastic algorithm. If the gradient of a parameter θ_i is $+0.1$ for nine instances and -0.9 for one instance, then we would like the update to θ_i to be close to zero. But RPROP will increase θ_i nine times, and only decrease θ_i once [Hinton et al., 2014].

The RMSPROP algorithm addresses this deficiency by scaling each component of the gradient by the *root mean square* of its exponentially decaying average. This leads to the update rule

$$\begin{aligned}
\bar{v}_{k+1} &\coloneqq (1-\tau)\bar{v}_k + \tau(g_k)^2 + \epsilon \\
\theta_{k+1} &\coloneqq \theta_k - \frac{\eta}{\sqrt{\bar{v}_{k+1}}}\, g_k,
\end{aligned}$$

where $\epsilon \in \mathbb{R}^n$ and $\tau, \eta \in \mathbb{R}^+$ such that $\tau \in [0, 1]$. The square root operation is performed elementwise. Incorporating NAG into RMSPROP can sometimes be beneficial [Hinton et al., 2014, climin developers, 2013]. This leads to the update rule

$$\begin{aligned}
\bar{v}_{k+1} &\coloneqq (1-\tau)\bar{v}_k + \tau(g_k)^2 + \epsilon \\
\hat{\theta}_{k+1} &\coloneqq \theta_k + \mu s_k \\
s_{k+1} &\coloneqq \frac{\eta}{\sqrt{\bar{v}_k}}\, \hat{g}_{k+1} \\
\theta_{k+1} &\coloneqq \theta_k + s_{k+1}.
\end{aligned}$$

For reasons that are currently unknown, the use of momentum with RMSPROP does not help as much as it usually does for other first-order algorithms [Hinton et al., 2014].

The ADADELTA algorithm [Zeiler, 2012] is a modification of the ADAGRAD algorithm based on RMSPROP, although Zeiler did not seem to know of RMSPROP by that name at the time of publication. The ADAGRAD algorithm uses the update rule

$$\begin{aligned}
\bar{v}_{k+1} &\coloneqq \bar{v}_k + (g_k)^2 + \epsilon \\
\theta_{k+1} &\coloneqq \theta_k - \frac{\eta}{\sqrt{\bar{v}_{k+1}}}\, g_k,
\end{aligned}$$

where $\epsilon \in \mathbb{R}^n$ and $\eta \in \mathbb{R}^+$. This update rule is the same as that of RMSPROP, except that the average of the gradient vector is not exponentially decaying. As a result, the effective learning rate can only decrease over time. If η is chosen too small, then progress will be painfully slow. ADADELTA makes the running average of $(g_k)^2$ exponentially

decreasing, and also multiplies the step by the root mean square of the average of the past updates (see Zeiler [2012] for the motivation behind why this is done). These modifications lead to the update rule

$$\begin{aligned}
\bar{v}_{k+1} &\coloneqq (1-\tau)\bar{v}_k + \tau(g_k)^2 + \epsilon \\
s_{k+1} &\coloneqq \sqrt{\frac{\bar{s}_k}{\bar{v}_{k+1}}}\, g_k \\
\bar{s}_{k+1} &\coloneqq (1-\tau)\bar{s}_k + \tau(s_{k+1})^2 + \epsilon \\
\theta_{k+1} &\coloneqq \theta_k - s_{k+1}.
\end{aligned}$$

The ADADELTA algorithm possesses many of the same properties as vSGD. The numerator s_k of the step has an effect similar to that of momentum, while the denominator $\bar{v}_{k+1}$ scales the learning rates to compensate for the vanishing gradient problem. Since the value s_k lags behind by an iteration, ADADELTA is robust to sudden changes in the gradient, much in the same way as NAG. As $\theta_k \to \theta^\star$, $\bar{s}_k, \bar{v}_{k+1} \to \epsilon$, and $\sqrt{\bar{s}_k / \bar{v}_{k+1}} \to e$. This property provides an intrinsic annealing schedule. Unfortunately, the updates that are generated as $\theta_k \to \theta^\star$ are still too aggressive, and local convergence is inferior to that of a carefully tuned version of SGU with momentum.

4.4 Second-Order Optimization Algorithms

The two most successful optimization algorithms for deep neural networks that can justifiably be called "second-order methods" are L-BFGS and an adaptation of the linear CG method called "Hessian-free optimization" (HF) [Martens, 2010]. Both of these algorithms are second-order methods from traditional optimization that have been adapted for use with mini-batches. The L-BFGS algorithm maintains a history of past values of θ_k and g_k in order to implicitly compute approximate Hessian-vector products. Since the Hessian matrix is only crudely approximated by past updates, accuracy can suffer compared to that of HF, which computes *exact* Hessian-vector products [Ngiam et al., 2011].

HF is the *only* known competitive optimization algorithm for deep neural networks that computes the Hessian-vector products exactly. At each outer iteration, HF computes the gradient over a relatively large mini-batch. It then iterates the linear CG method to minimize the local quadratic model determined by the gradient and Hessian. At each inner iteration of CG, a modified version of backpropagation (see Section 1.6) is used to compute the Hessian-vector products exactly. These inner iterations are terminated when the relative per-iteration progress becomes sufficiently small.

In the process of adapting linear CG to work for deep neural networks, Martens developed custom strategies for damping, choosing the mini-batch size, termination, "hot-starting" the inner iterations, preconditioning, initialization, and more [Martens, 2010]. [Martens and Sutskever, 2012] give a comprehensive tutorial on using HF in practice. While implementing HF is involved, its performance in several tasks is still unmatched [Martens, 2010, Ngiam et al., 2011, Sutskever et al., 2013]. Unfortunately,

practitioners are reluctant to implement it due to its underlying complexity, instead preferring to continue experimentation with first-order methods in order to tighten the performance gap [Sutskever et al., 2013].

5 Experiments

I benchmarked the first-order methods covered in this report on the MNIST database of handwritten digits [LeCun et al., 1998a]. This data set consists of 60,000, 28×28, 8-bit, black and white images of handwritten digits. The training set consists of 50,000 images, and the test set consists of the remaining 10,000 images. Classifying handwritten digits is now considered to be an easy task, since relatively small convolutional networks can achieve error rates considerably less than one percent without any data augmentation techniques (e.g. affine distortions). Nonetheless, the MNIST database is still a useful benchmark that is almost invariably used in publications that propose new techniques for convolutional networks.

The convolutional network that is used for this benchmark is the same one described by LeCun et al. [1998a], except some small changes are made for the sake of simplicity. Instead of using the given sparse connection pattern between layers S2 and C3, I connected each unit in layer C3 to the corresponding units in every feature map of layer S2. Additionally, the center of each Euclidean RBF unit was randomly initialized from the values ± 1 instead of being chosen to represent ASCII bitmaps of the corresponding digit. The images in the training sample were centered and scaled as described by (1.9), and the initial weights of each layer were sampled from a uniform distribution with mean zero and standard deviation given by (1.10).

I used four variants each of SGU: one with a global, static learning rate, one with a global learning rate and an annealing schedule, one with CM, and with NAG. The global learning rate was chosen by performing a grid search in $(0, 2]$, using increments of $\rho_1 := 0.1$. For each trial learning rate, the convolutional network was trained on a randomly chosen sample of 200 training instances to minimize the test error. I identified the learning rate $\eta_1^\star$ achieving the lowest mean test error computed from five initializations, each using a newly-chosen training sample. The grid search was refined twice: first on the interval $[\eta_1^\star - \rho_1, \eta_1^\star + \rho_1]$, with $\rho_2 := 0.01$, and next on the interval $[\eta_2^\star - \rho_2, \eta_2^\star + \rho_2]$ with $\rho_3 := 0.001$.

The annealing schedule I used is given by

$$\eta_k := \frac{\eta_0}{1 + \beta k},$$

as suggested by [Bottou, 2012]. Many other annealing schedules are used in practice; see LISA Lab [2011] for more examples. The parameters $\eta_0^\star, \beta^\star$ were found using a twice-refined grid search on $(0, 2]^2$, using the same values of ρ_1, ρ_2, ρ_3 given previously. The annealing schedule for the momentum is the same one used by Sutskever et al. [2013], and is given by

$$\mu_k := \min\left(1 - 2^{-1-\log_2(\lfloor sk/r \rfloor + 1)}, \mu_{\max}\right),$$

Algorithm	Parameters			Test Error (%)
SGU (global)	$\eta := 0.006$			1.14
SGU (annealed)	$\eta_0 := 0.013$	$\beta := 1.005$		0.83
SGU + CM	$\eta_0 := 0.004$	$\mu_{\max} := 0.982$		0.78
SGU + NAG	$\eta_0 := 0.006$	$\mu_{\max} := 0.995$		0.76
DH (global)	$\eta := 0.011$	$\tau := 0.634$		1.11
DH (annealed)	$\eta_0 := 0.014$	$\tau := 0.667$	$\beta := 1.012$	0.82
DH + CM	$\eta := 0.012$	$\tau := 0.661$	$\mu_{\max} := 0.988$	0.78
DH + NAG	$\eta := 0.013$	$\tau := 0.665$	$\mu_{\max} := 0.993$	0.75
vSGD	$C := 48$			0.74
RMSPROP	$\eta := 0.004$	$\tau := 0.792$		0.76
RMSPROP + NAG	$\eta := 0.004$	$\tau := 0.810$	$\mu := 0.664$	0.76
ADADELTA	$\eta := 0.006$			0.78

Table 5.1: Results for the first-order optimization algorithms on the MNIST database.

where s is the size of the training sample (50,000 in our case), r is the mini-batch size, and $\mu_{\max} \in \mathbb{R}^+$ limits the growth of the momentum parameter. For both CM and NAG, I kept the learning rate η constant, as was done in Sutskever et al. [2013]. Both $\mu_{\max}$ and η were selected using a twice-refined grid search on $(0, 2]$ using the same values for ρ_1, ρ_2, ρ_3 given previously. The parameters for the same four variants of DH were chosen using the same grid search procedures, and τ was selected using a twice-refined grid search on $(0, 1]$ with the same values of ρ_1, ρ_2, ρ_3.

While vSGD, RMSPROP, and ADADELTA are claimed to require little human effort, selecting the best values for the hyperparameters still requires a grid search. To find the best values for η and τ for RMSPROP, η, τ, and μ for RMSPROP + NAG, and τ for ADADELTA, I performed the usual twice-refined grid search. The initial search intervals for all parameters were $(0, 1]$. To find the best value of C for vSGD, I performed a grid search in $(0, 1000]$, with $\rho_1 := 100$, $\rho_2 := 10$, and $\rho_3 := 1$. All hyperparameters were kept constant during training. Due to time constraints, I set the mini-batch size r to 200 for all of the algorithms tested. The best results obtained are given in Table 5.1. Since computing time is not a determining factor for the quality of the solution found, I allowed all algorithms to run until convergence.

Several features of the results confirm our expectations. The adaptive algorithms (vSGD, RMSPROP, and ADADELTA), which use separate learning rates for each parameter, generally outperformed those that used global learning rates. However, the differences in performance were not as great as I would have expected. Each modification, such as the incorporation of momentum or $\operatorname{diag}(H)$, only yielded minuscule improvements in the test error at convergence. Moreover, these improvements were only apparent after exhaustive grid searches were used to find the best values for all of the hyperparameters. A practitioner is unlikely to have the time or patience to do this for each new problem. In this respect, the adaptive algorithms provide a clear advantage:

they are robust with respect to selection of the hyperparameters.

I was surprised to see that the heuristics used by the adaptive algorithms only provided marginal improvements over the best results for algorithms that used global learning rates. One explanation for this is that a finely-tuned annealing schedule ensures that the effective learning rates gently approach zero as $\theta_k \to \theta^\star$. Both vSGD and ADAGRAD have intrinsic annealing schedules, but the updates generated as $\theta_k \to \theta^\star$ are probably too aggressive. Further experimentation with custom annealing schedules is likely to provide improvements over the results for the adaptive algorithms. These results show that careful testing can provide valuable information about the relative performance of algorithms that would otherwise be missed.

6 Conclusion

Research suffers from a lack of theoretical understanding of the training dynamics of neural networks. Most of the techniques used by the recent adaptive optimization algorithms for neural networks were already mentioned over a decade earlier [LeCun et al., 1998b]. But because the experiments carried out at that time were not comprehensive enough, these techniques were quickly forgotten, and had to be rediscovered later. The availability of more general software that would allow practitioners to easily try out a variety of optimization algorithms for a given problem may help avoid this problem.

Due to space and time constraints, several interesting topics were not addressed by the experiments. The MNIST database was the only data set used for the experiments. The extent to which the relative performance of optimization algorithms depends on specific features of the data remains to be seen. There is also evidence that incorporating a line search into first-order optimization algorithms, even for small mini-batches, may improve generalization [Ngiam et al., 2011]. The interaction between line search procedures and the other techniques used by these algorithms remains unexplored. Lastly, none of the second-order algorithms mentioned in Section 4.4 were implemented. For a different type of convolutional network, Sutskever et al. [2013] report a test error of 0.69% on the MNIST database, using a modified version of HF. It would be interesting to incorporate the results for L-BFGS and HF into Table 5.1.

References

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

climin developers. rmsprop. `http://climin.readthedocs.org/en/latest/rmsprop.html#tieleman2012rmsprop`, 2013. Accessed: 2014-11-24.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Geoff Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning. 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998a.

Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998b.

LISA Lab. Training – pylearn2 dev documentation. `http://deeplearning.net/software/pylearn2/library/train.html`, 2011. Accessed: 2014-12-08.

James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

James Martens and Ilya Sutskever. Training deep and recurrent neural networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade*, 2012.

Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN 026201825X, 9780262018258.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas, Awni Hannun, Brody Huval, Tao Wang, and Sameep Tandon. Unsupervised feature learning and deep learning tutorial. `http://ufldl.stanford.edu/tutorial/`, 2013. Accessed: 2014-11-24.

Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272, 2011.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *Proc. International Conference on Machine learning (ICML'13)*, 2013.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer, 2014.