Optimization for Neural Networks

Aditya Ramesh

Abstract

TODO: Update this!

We introduce the concept of a neural network as a biologically-inspired graphical template that allows us to construct nonlinear functions to perform classification and regression tasks. The nonlinear function represented by the neural network is controlled by a set of parameters that are calibrated during the training process. By viewing the update rule for the optimization algorithm used during training as a discrete dynamical system, we can gain some insight into the stability properties of a neural network. Using this insight, we derive an optimal value for the learning rate used during training.

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 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 strucuture 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

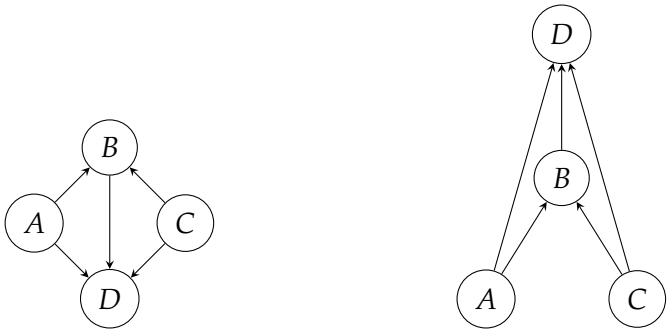


Figure 1.1: In 1.1a, we see a DAG, and in 1.1b, the representation of the DAG as a layered graph. Notice that the edges $A \to D$ and $C \to D$ are manifested as skip connections in the layered graph.

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_1, 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}. 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

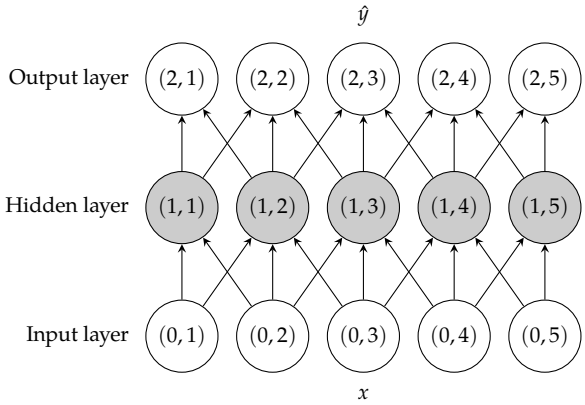


Figure 1.2: A small, two-layer neural network. The pair (k, j) is used to denote the jth unit in layer k.

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

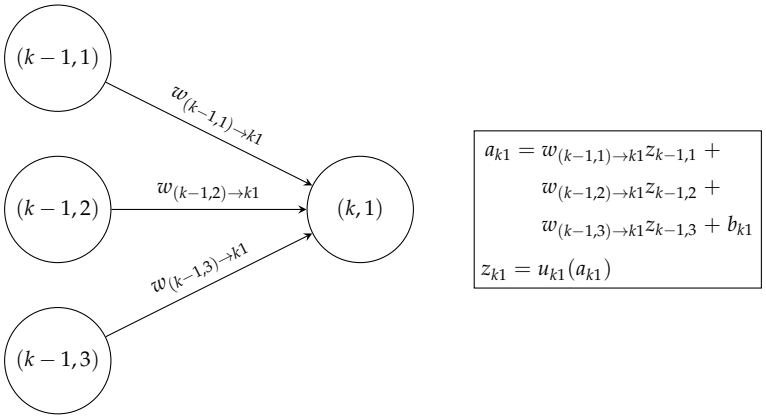


Figure 1.3: An illustration how the output of $(k,1)$, a unit with three inputs from L_{k-1}, is computed. In this case, $\mathrm{Pa}(k,1) = \{(k-1,1),(k-1,2),(k-1,3)\}$. First, the activation a_{k1} is computed by forming a linear combination of the outputs of the units in $\mathrm{Pa}(k,1)$ with the 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}.

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

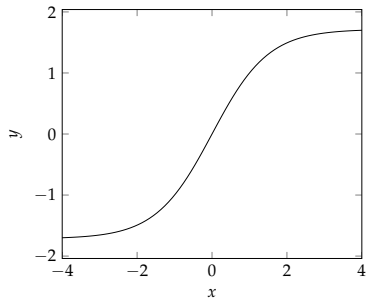
$$(z_0, \ldots, z_{k-1}) \overset{\sigma_k}{\longmapsto} z_k$$

is shorthand for saying that we compute z_k through w simultaneous applications of Equation 1.2, followed by w simultaneous applications of Equation 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 Equations 1.1. This process is called *forward propagation*.

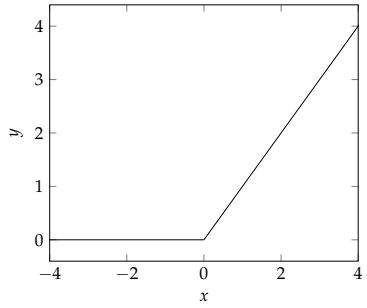
1.2 Activation Functions and Feature Learning

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.

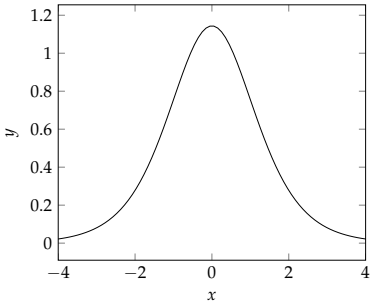
The choices for the constants a and b of the scaled tanh function are motivated by several reasons [LeCun et al., 1998]. Firstly, the scaled tanh function is symmetric and approximately linear about the origin. If the inputs are decorrelated, our choices for a and b cause the variance of the scaled tanh function to be close to unity. This has been shown to accelerate learning in neural networks.



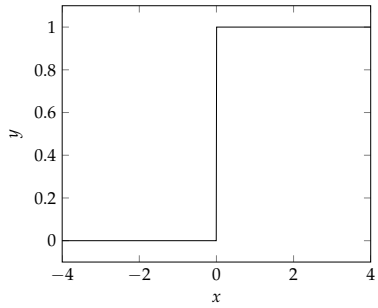
(a) Plot of $f(x) := a\tanh(bx)$, where $a := 1.7159$ and $b := 2/3$.



(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 the ± 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. Thus, a minimization algorithm using derivative information would be liable to "get stuck".

Finally, the choices for these constants allow us to intrepret 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., 2001]. 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 units. By

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.

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

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

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 $S \coloneqq \{(x_1, y_1), \ldots, (x_N, y_N)\}$, 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 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., 1998]. 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 layer of

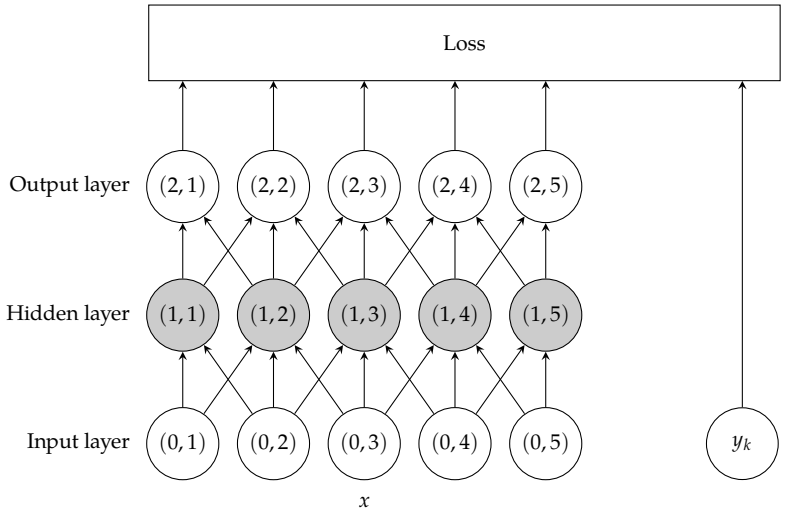


Figure 1.6: A visualization of how the kth instance (x_k, y_k) is fed into the two-layer neural network from Figure 1.2.

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 minimization algorithm used to adjust 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 $X = \{x_1, \ldots, x_N\}$ and $Y = \{y_1, \ldots, y_N\}$. If we assume that the instances $(x_k, y_k) \in S$ are iid, then we can apply the chain rule to write

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

The LHS of Equation 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

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

The LHS of Equation 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

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

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

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

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

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

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 Equation 1.5. It follows that using Equation 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, Equation 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

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., 1998].

When all of the instances in the training sample have the same importance, it is often beneficial to *decorrelate* the inputs. One way of doing this is as follows [LeCun et al., 1998]. Given a sample D, we compute the mean and variance for each of the n components of the inputs; this yields the vectors $\mu, \sigma \in \mathbb{R}^n$, respectively. Each input $x_k \in \mathbb{R}^n$ is normalized by computing

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

Now assume that the inputs are decorrelated, so that

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

for each $i \in [1, n]$. From Equation 1.2, we know that the activation of $(1, j)$, the jth unit in the first layer of the network, 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

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

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

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

The most common choices of distribution for weight initialization are the uniform and normal distributions [LeCun et al., 1998, 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 minimization 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]. The subtle numerical issues resulting from these complex interactions largely determines whether, when, and how a given minimization 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. Developing a deeper understanding of what is happening in these situations is likely to have widespread practical consequences.

1.6 Computing Derivatives

TODO

1.7 Convolutional Networks

TODO

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 determine when an optimization algorithm is good.

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 $Y \coloneqq \{0, 1\}$, 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 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*

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

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 a 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 $S \coloneqq \{(x_1, y_k), \ldots, (x_N, y_N)\} \sim D$, the* empirical error *or* empirical risk *of f is defined as*

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

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 the definition of probably approximate correct (PAC) learning.

Definition 3 (PAC-learning [Mohri et al., 2012])**.** *Let $O(n)$ be an upper bound on the space complexity of an input $x \in X$. A concept class is said to be 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 $N \geq g(1/\epsilon, 1/\delta, n, \mathrm{size}(c))$:*

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

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."

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 N be the size of the training samples drawn from D. Then with probability $1 - \delta$, it holds for any $f \in F$ that

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

where $c \in \mathbb{R}^+$ and capacity(F) is a quantity that measures the complexity of our chosen family of functions. When we apply the bound (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. These quantities correspond to the first and second terms on the RHS of (2.1), respectively. This tradeoff can be expressed in another way that is particularly illuminating. Let

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

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

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

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 intractible, 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 causes the estimation error to decrease, while reducing the complexity term causes the approximation error to decrease. But reducing the approximation error often requires increasing capacity(F), and this increases the complexity

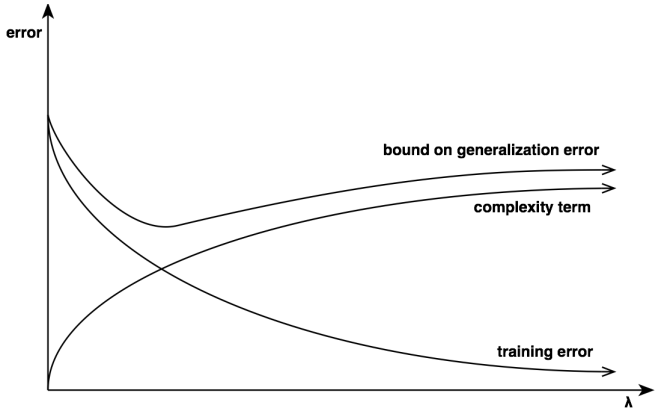


Figure 2.1: Visualization of the SRM procedure (adapted from Mohri et al. [2012]). The SRM procedure minimizes an upper bound of the generalization error that is a sum of the training error and the complexity term.

term. On the other hand, reducing the complexity term requires decreasing capacity(F), inhibiting our ability to approximate C. This increases the training error. The interplay between these two errors is called the *approximation-estimation tradeoff*.

Minimizing only the training error while ignoring the complexity term is called *empirical risk minimization* (ERM), and is inadvisable on both theoretical and practical grounds. An alternative procedure that contols capacity(F) is called *structural risk minimization* (SRM). Let $c \in \mathbb{R}^+$, and define

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

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

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

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

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

The Lagrange multiplier λ called the 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 intractible in general. 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 $Y \in \mathbb{R}$. Let S be a training sample consisting of N instances. We define

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

and assume let E be the sum-of-squares error given by Equation 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 Equation 2.5, such as $\{\lambda_k\}_{k=-5}^{5}$, where $\lambda_k \coloneqq 2^k$. The results from statistical learning theory shape our understanding of how we should solve problems in machine learning.

3 Characterizing Optimization Algorithms

We turn our focus to applying statistical learning theory to characterize the quality of optimization algorithms for machine learning. Let f_{ERM} be given by Equation 2.4. During training, we 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 $S \sim D$ 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 $N_{\max}$) 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 $T(F, \rho, N)$ 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 N. Minimizing the excess error in Equation 3.3 subject to the constraints on $N_{\max}$ and $T_{\max}$ yields the following meta-optimization problem [Bousquet and Bottou, 2008]:

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

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

Table 3.1 shows how the errors and computing time are affected when capacity(F), ρ, and N 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 $N < N_{\max}$ 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

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

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. It is easy to show that

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

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 simpliyfing assumptions. First, we assume that F is the family of affine functions given by Equation 2.2. In this special case, the generalization bound (2.1) simplifies to

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

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

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

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

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

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 $\theta_{\text{ERM}} := (w_{\text{ERM}}, b_{\text{ERM}})$ be the parameter vector corresponding to f_{ERM}. For a given input $x_k \in \mathbb{R}^n$, we define $\hat{y}_k^{\text{ERM}} := w_{\text{ERM}} x_k + b_{\text{ERM}}$.

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 given by

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

and

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

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 $N \in \mathbb{N}$ such that, with probability at least $1 - \delta$, we have

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

The condition number $\kappa := \lambda_{\max}/\lambda_{\min}$ determines the difficulty of the optimization problem.

The optimization algorithms used in machine learning can either use *batch updates* or *stochastic updates*. An algorithm that uses stochastic updates will apply an update to θ for each instance in S. On the other hand, an algorithm that uses batch updates will apply an update to θ every r iterations, where $r|N$. The derivative information computed by batch algorithms is averaged over the r iterations. For the purposes of our discussion, we assume that the batch size is N.

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 computed by the update rules are not necessary descent directions.

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

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

  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

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

  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 algorithm uses the update

References

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

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

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.

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

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*, pages 306–351. IEEE Press, 2001.

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

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.

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