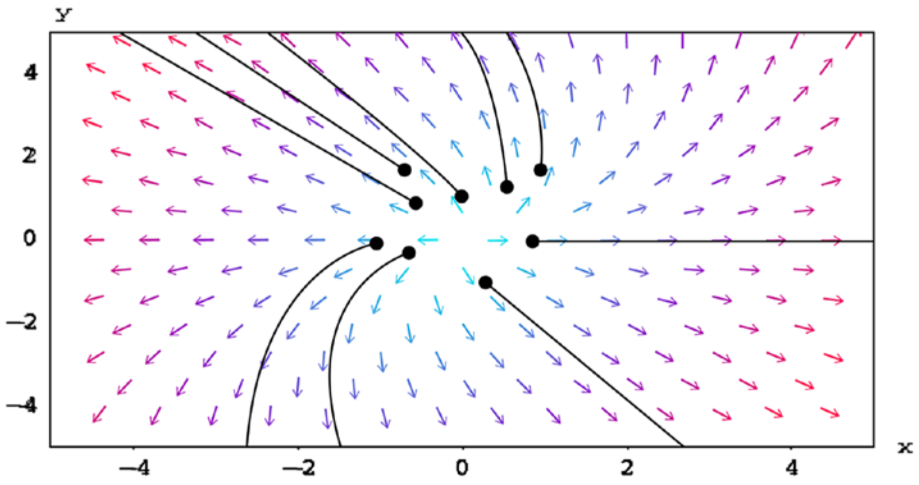


Figure 2-6. *Visulaization of eigenvectors*

Eigenvectors and eigenvalues become an integral part of understanding a technique discussed later in our discussion regarding a variable selection technique called principal components analysis (PCA). The eigendecomposition of a symmetric positive semi-definite matrix yields an orthogonal basis of eigenvectors, each of which has a non-negative eigenvalue. PCA studies linear relations among variables and is performed on the covariance matrix, or the correlation matrix, of the input data set. For the covariance or correlation matrix, the eigenvectors correspond to principal components and the eigenvalues to the variance explained by the principal components. Principal component analysis of the correlation matrix provides an orthonormal eigenbasis for the space of the observed data: in this basis, the largest eigenvalues correspond to the principal components that are associated with containing the most covariability of the observed dataset.

Linear Transformations

A linear transformation is a mapping $V \rightarrow W$ between two modules that preserves the operations of addition and scalar multiplication. When V = W, we call this a linear operator, or endomorphism, of V. Linear transformations always map linear subspaces onto linear subspaces, and sometimes this can be in a lower dimension. These linear maps can be represented as matrices, such as rotations and reflections. An example of where linear transformations are used is specifically PCA. Discussed in detail later, PCA is an orthogonal linear transformation of the features in a data set into uncorrelated principal components such that for K features, we have K principal components. I discuss orthogonality in detail in the following sections, but for now I focus on the broader aspects of PCA. Each principal component retains the variance from the original data set but gives us a representation of it such that we can infer the importance of a given principal component based on the contribution of the variance to the data set it provides. When translating this to the original data set, we then can remove features from the data set that we feel don’t exhibit significant amounts of variance.

A function $\mathcal{L}:\mathbb{R}^n \rightarrow \mathbb{R}^m$ is called a linear transformation if the following is true:

$$\mathcal{L}(ax) = a\mathcal{L}(x)\, \textit{for every } x \in \mathbb{R}^n \textit{and } a \in \mathbb{R}$$

$$\mathcal{L}(x+y) = \mathcal{L}(x) + \mathcal{L}(y)\, \textit{for every } x, y, \in \mathbb{R}^n$$

When we fix the bases for $\mathbb{R}^n$ and $\mathbb{R}^m$, the linear transformation $\mathcal{L}$ can be represented by a matrix A. Specifically, there exists $A \in \mathbb{R}^{mxn}$ such that the following representation holds. Suppose $x \in \mathbb{R}^n$ is a given vector and x' is the representative of *x* with respect to the given basis for $\mathbb{R}^m$. If $y = \mathcal{L}(x)$ and Y' is the representative of y with respect to the given basis for $\mathbb{R}^m$, then

$$y' = Ax'$$

We call A the matrix representation of $\mathcal{L}$ with respect to the given bases for $\mathbb{R}^n$ and $\mathbb{R}^m$.

Quadratic Forms

A *quadratic form* is a homogenous polynomial of the second degree in a number of variables and have applications in machine learning. Specifically, functions we seek to optimize that are twice differentiable can be optimized using Newton's method. The power in this is that if a function is twice differentiable, we know that we can reach an objective minimum.

A quadratic form $f:\mathbb{R}^n \rightarrow \mathbb{R}^m$ is a function such that the following holds true:

$$F(x) = x^T Q x$$

Where Q is an *n* x *n* real matrix. There is no loss of generality in assuming Q to be symmetric—that is, $Q = Q^T$.

Minors of a matrix Q are the determinants of the matrices obtained by successively removing rows and columns from Q. The principal minors are detQ itself and the determinants of matrices obtained by removing an ith row and an ith column.

Sylvester's Criterion

Sylvester's criterion is necessary and sufficient to determine whether a matrix is positive semi-definite. Simply, it states that for a matrix to be positive semi-definite, all the leading principal minors must be positive.

Proof: if real-symmetric matrix A has non-negative eigenvalues that are positive, it is called positive-definite. When the eigenvalues are just non-negative, A is said to be positive semi-definite.

A real-symmetric matrix A has non-negative eigenvalues if and only if A can be factored as $A = B^T B$, and all eigenvalues are positive if and only if B is non-singular.

Forward implication: if $A \in R^{nxn}$ is symmetric, then there is an orthogonal matrix P such that $A = PDP^T$, where $\mathrm{D} = \mathrm{diag}(\lambda_1 \lambda_2, \ldots, \lambda_\mathrm{n})$ is a real diagonal matrix with entries such that its columns are the eigenvectors of A. If $\lambda_i \geq 0$ for each I, $D^{\frac{1}{2}}$ exists.

Reverse implication: if A can be factored as A = B^TB, then all eigenvalues of A are non-negative because for any eigenpair (x, λ)

$$\lambda = \left(\frac{x^T A x}{x^{Tx}}\right) = \left(\frac{x^T B^T B x}{x^T x}\right) = \left(\frac{\|Bx\|^2}{\|x\|^2}\right) \geq 0$$

Orthogonal Projections

A *projection* is linear transformation P from a vector space to itself such that $P^2 = P$. Intuitively, this means that whenever P is applied twice to any value, it gives the same result as it it were applied once. Its image is unchanged and this definition generalizes the idea of graphical projection moreover. $\mathcal{V}$ is a subspace of $\mathbb{R}^n$ if $\mathrm{x}_1, \mathrm{x}_2 \in \mathcal{V} \rightarrow \alpha x_1 + \beta x_2 \in \mathcal{V}$ for all $\alpha, \beta \in \mathbb{R}$. The dimension of this subspace is also equal o the maximum number of linearly independent vectors in $\mathcal{V}$. If $\mathcal{V}$ is a subspace of R^n, the orthogonal complement of $\mathcal{V}$, demoted $\mathcal{V}^\perp$, consists of all vectors that are orthogonal to every vector in $\mathcal{V}$. Thus, the following is true:

$$\mathcal{V}^\perp = \left\{x : v^T x = 0 \textit{ for all } v \in \mathcal{V}\right\}$$

The orthogonal complement of $\mathcal{V}$ is also a subspace. Together, $\mathcal{V}$ and $\mathcal{V}^\perp$ span R^n in the sense that every vector $x \in \mathbb{R}^n$ can be represented as

$$x = x_1 + x_2$$

where $x_1 \in \mathcal{V}$ and $x_2 \in \mathcal{V}^\perp$. We call the above representation the *orthogonal decomposition* of *x* with respect to $\mathcal{V}$. We say that x_1 and x_2 are orthogonal projections of *x* onto the subspaces $\mathcal{V}$ and $\mathcal{V}^\perp$ respectively. We write $\mathbb{R}^n = \mathcal{V} \otimes \mathcal{V}^\perp$, and say that $\mathbb{R}^n$ is a direct sum of $\mathcal{V}$ and $\mathcal{V}^\perp$. We say that a linear transformation of P is an orthogonal projector onto $\mathcal{V}$ for all $x \in \mathbb{R}^n$, we have $Px \in \mathcal{V}$ and $x - Px \in \mathcal{V}^\perp$.

Range of a Matrix

The *range* of a matrix defines the number of column vectors it contains.

Let $A \in \mathbb{R}^{mxn}$. The range, or image, of A, is written as the following:

$$\mathcal{R}(A) \triangleq \left\{Ax : x \in \mathbb{R}^n\right\}$$

Nullspace of a Matrix

The nullspace of a linear map $\mathcal{L}:\mathcal{V}\rightarrow\mathcal{W}$ between two vector spaces is the set of all elements of v of $\mathcal{V}$ for which $\mathcal{L}(v)=0,$ where zero denotes the zero vector in $\mathcal{W}$.

The nullspace, or kernel, of A is written as the following:

$$\mathcal{N}(A)\triangleq\left\{x\in\mathbb{R}^n : Ax=0\right\}$$

Hyperplanes

Earlier I mentioned the significance of the support vector machine and the hyperplane. In the context of regression problems, the observations within the hyperplane are acceptable as response variable solutions. In the context of classification problems, the hyperplanes form the boundaries between different classes of observations (shown in Figure 2-7).

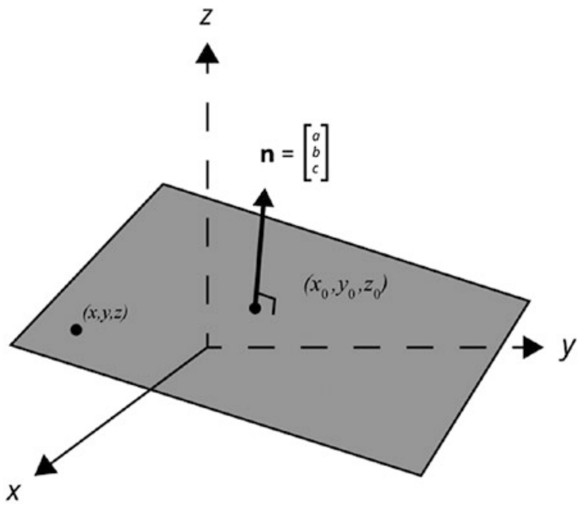


Figure 2-7. *Visualization of hyperplane*

We define a *hyperplane* as a subspace of one dimension less than its ambient space, otherwise known as the feature space surrounding the object.

Let $u=[u_1,u_2,\ldots,u_n]$, $\mathrm{u}\in\mathbb{R}$, where at least one of the u_i is non-zero. The set of all points $x=[x_1,x_2,\ldots,x_n]^T$ that satisfy the linear equation

$$u_1x_1+u_2x_2+\ldots+u_nx_n=v$$

is called a hyperplane of the space $\mathbb{R}^n$. We may describe the hyperplane with the following equation:

$$\{x \in \mathbb{R}^n : u^T x = v\}$$

A hyperplane is not necessarily a subspace of $\mathbb{R}^n$ because, in general, it does not contain the origin. For $n = 2$, the equation of the hyperplane has the form $u_1x_1 + u_2x_2 = v$, which is the equation of a straight line. Thus, straight lines are hyperplanes in $\mathbb{R}^2$. In $\mathbb{R}^3$, hyperplanes are ordinary planes. The hyperplane H divides $\mathbb{R}^n$ into two half spaces, denoted by the following:

$$H_+ = \{x \in \mathbb{R}^n : u^T x \geq 0\},$$

$$H_- = \{x \in \mathbb{R}^n : u^T x \leq 0\}.$$

Here H_+ is the positive half-space, and H_- is the negative half-space. The hyperplane H itself consists of the points for which $\langle u, x - a \rangle = 0$, where $a = [a_1, a_2, \ldots, a_n]^T$ is an arbitrary point of the hyperplane. Simply stated, the hyperplane H is all of the points x for which the vectors u and x – a are orthogonal to one another.

Sequences

A *sequence* of real numbers is a function whose domain is the set of natural numbers 1,2,…,k, and whose range is contained in $\mathbb{R}$. Thus, a sequence of real numbers can be viewed as a set of numbers $\{x_1, x_2, \ldots, x_k\}$, which is often also denoted as $\{x_k\}$.

Properties of Sequences

The *length* of a sequence is defined as the number of elements within it. A sequence of finite length n is also called an n-tuple. *Finite* sequences also include sequences that are empty or ones that have no elements. An *infinite* sequence refers to a sequence that is infinite in one direction. It is therefore described as having a first element, but not having a final element. A sequence with neither a first nor a final element is known as a *two-way infinite* sequence or *bi-infinite* sequence.

Moreover, a sequence is said to be monotonically increasing if each term is greater than or equal to the one before it. For example, the sequence $an(n) = 1$ is monotonically increasing if an only if for all $a_{n+1} \geq a_n$. The terms *non-decreasing* and *non-increasing* are often used in place of increasing and decreasing in order to avoid any possible confusion with strictly increasing and strictly decreasing respectively.

If the sequence of real number is such that all the terms are less than some real numbers, then the sequence is said to be bounded from above. This means that there exists *M* such that for all *n*, $a_n \leq M$. Any such *M* is called an upper bound. Likewise, if, for some real m, $a_n \geq m$ for all *n* greater than some *N*, then the sequence is bounded from below, and any such *m* is called the lower bound.

Limits

A *limit* is the value that a function or sequence approaches as the input or index approaches some value. A number $x^* \in \mathbb{R}$ is called the limit of the sequence if for any positive ϵ there is a number *K* such that for all $k > K, |xk - x^*| < \epsilon$:

$$x^* = \lim_{k \to \infty} xk$$

A sequence that has a limit is called a *convergent* sequence. Informally speaking, a singly infinite sequence has a limit, if it approaches some value *L*, called the limit, as *n* becomes very large. If it converges towards some limit, then it is *convergent*. Otherwise it is *divergent*. Figure 2-8 shows a sequence converging upon a limit.

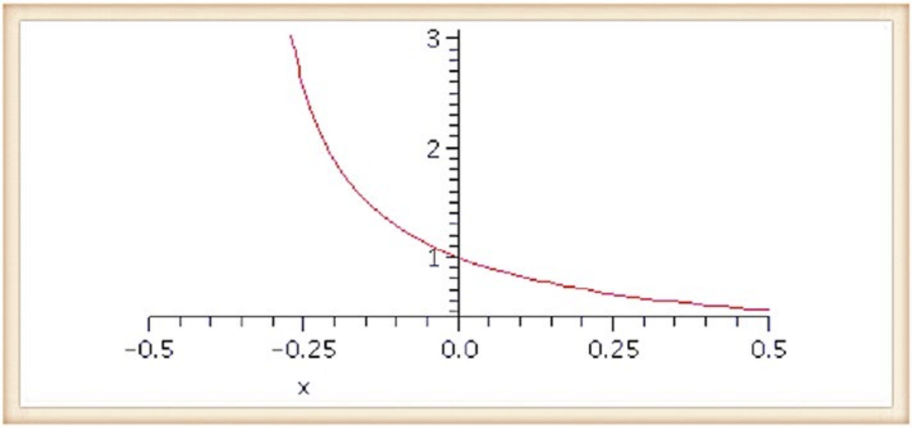


Figure 2-8. *A function converging upon 0 as x increases*

We typically speak of convergence within the context of machine learning and deep learning with reaching an *optimal solution*. This is ultimately the goal of all of our algorithms, but this becomes more ambiguous with the more difficult use cases readers encounter. Not every solution has a single global optimum—instead it could have local optima. Methods of avoiding these local optima are more specifically addressed in later chapters. Typically this requires parameter tuning of machine learning and deep learning algorithms, the most difficult part of the algorithm training process.

Derivatives and Differentiability

Differentiability becomes an important part of machine learning and deep learning, most specifically for the purpose of parameter updating. This can be seen via the back-propagation algorithm used to train multilayer perceptrons and the parameter updating of convolutional neural networks and recurrent neural networks. A derivative of a function measures the degree of change in one quantity to the degree of another. One of the most common examples of a derivative is a slope (change in y over x), or the return of a stock (price percentage change over time). This is a fundamental tool for calculus but is also the basis of many of the models we will study in the latter part of the book.

A function is considered to be *affine* if there exists a linear function $\mathcal{L}:\mathbb{R}^n \to \mathbb{R}^m$ and a vector $y \in \mathbb{R}^m$ such that

$$A(x) = \mathcal{L}(x) + y$$

for every $x \in \mathbb{R}^n$. Consider a function $f:\mathbb{R}^n \to \mathbb{R}^m$ and a point $x_0 \in \mathbb{R}^n$. We want to find an affine function A that approximates f near the point x0. First, it's natural to impose this condition:

$$A(x_0) = f(x_0)$$

We obtain $y = f(x_0) - \mathcal{L}(x_0)$. By the linearity of L,

$$\mathcal{L} + y = \mathcal{L}(x) - \mathcal{L}(x_0) + f(x_0) = \mathcal{L}(x - x_0) + f(x_0)$$

$$A(x) = \mathcal{L}(x - x_0) + f(x_0)$$

We also require that $A(x)$ approaches $f(x)$ faster than x approaches x_0.

Partial Derivatives and Gradients

Also utilized heavily in various machine learning derivations is the *partial derivative*. It is similar to a derivative, except we only take the derivative of one of the variables in the function and hold the others constant, whereas in a total derivative all the variables are evaluated. The gradient descent algorithm is discussed in Chapter 3, but we can discuss the broader concept of the gradient itself now. A *gradient* is the generalization of the concept of a derivative when applied to functions of several variables. The gradient represents the point of greatest rate of increase in the function, and its magnitude is the slope of the graph in that direction. It's a vector field whose components in a coordinate system will transform when going from one system to another:

$$\nabla f(x) = grad\ f(x) = \frac{df(x)}{dx}$$

Hessian Matrix

Functions can be differentiable more than once, which leads us to the concept of the Hessian matrix. The *Hessian* is a square matrix of second-order partial derivatives of a scalar values function, or scalar field:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

If the gradient of a function is zero at some point x, then f has a critical point at x. The determinant of the Hessian at x is then called the *discriminant*. If this determinant is zero, then x is called a degenerate critical point of *f*, or a non-Morse critical point of *f*. Otherwise, it is non-degenerate.

A Jacobian matrix is the matrix of first-order partial derivatives of a vector values function. When this is a square matrix, both the matrix and its determinant are referred to as the *Jacobian*:

$$\mathbf{J} = \frac{\mathrm{df}}{\mathrm{dx}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Summary

This brings us to the conclusion of the basic statistics and mathematical concepts that will be referenced in later chapters. Readers should feel encouraged to check back with this chapter when unsure about anything in later chapters. Moving forward, we'll address the more advanced optimization techniques that power machine learning algorithms, as well as those same machine learning algorithms that formed the inspiration of the deep learning methods we'll tackle afterwards.

CHAPTER 3

A Review of Optimization and Machine Learning

Before we dive into the models and components of deep learning in depth, it's important to address the broader field it fits into, which is machine learning. But before that, I want to discuss, if only briefly, optimization. *Optimization* refers to the selection of a best element from some set of available alternatives. The objective of most machine learning algorithms is to find the optimal solution given a function with some set of inputs. As already mentioned, this often comes within the concept of a supervised learning problem or an unsupervised learning problem, though the procedures are roughly the same.

Unconstrained Optimization

Unconstrained optimization refers to a problem in which we much reach an optimal solution. In contrast to constrain optimization, there are constraints placed on what value of x we choose, allowing us to approach the solution from significantly more avenues. An example of an unconstrained optimization problem is the following toy problem:

$$\text{Minimize}\, f(x),\ \text{where}\, f(x) = x^2, x \in [-100, 100]$$

Figure 3-1 visualizes this function.

T. Beysolow II, *Introduction to Deep Learning Using R*, DOI 10.1007/978-1-4842-2734-3_3

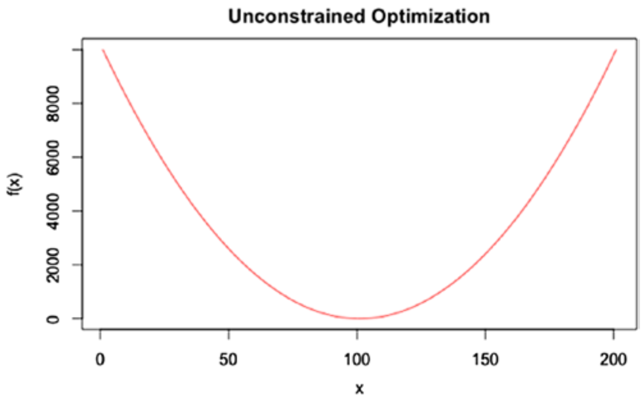


Figure 3-1. *Visualization of f(x)*

In this problem, because there are no constraints, we are allowed to choose whatever number for x is within the bounds defined. Given the equation we seek to minimize, the answer for x is 100. As we can see, we minimize the value of f(x) globally when we choose x. Therefore, we state that x = 100 = x*, which is a global minimizer of f(x). In contrast, here's a constrained optimization problem:

$$\text{Minimize } f(x), \text{ s.t.} (\text{subject to}) \, x \in \Omega$$

$$\text{where } f(x) = x^2 \text{ Subject to } x \in \Omega$$

The function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ that we want to minimize is a real-valued function and is called the objective/cost function. The vector x is a vector of length *n* consisting of independent variables where $x = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$. The variables within this vector commonly are referred to as *decision variables*. The set Ω is a subset of R called the constraint/feasible set. We say that the preceding optimization problem is a decision problem in which we must find the best vector of x that satisfies the objective subject to the constraint. Here, the best vector of x would result in a minimization of the objective function. In this function, because we have a constraint placed, we call this a constrained optimization problem. $x \in \Omega$ is known as the set constraint. Often, this takes the form of

$$\Omega = \{x : h(x) = 0, g(x) \leq 0\}$$

where *h* and *g* are some given functions. *h* and *g* are referred to as the *functional constraints*.

Imagine that we are still viewing the same function displayed in Figure 3-1, except that our feasible set is *Ω*. For simplicity's sake, let's say that *h*(*x*) and *g*(*x*) are equal to the following:

$$h(x) = g(x) = 10 - x$$

As such, the answer for the constrained optimization problem would be x = 10, because this is closest to the global minimizer of f(x), x = 100, while also satisfying the functional constraints listed in Ω. As we can see, the constraint set limits our ability to choose solutions, and therefore compromises must be made. We often encounter constrained optimization in a practical sense on a daily basis. For example, say a business owner is trying to minimize the cost of production in their factory. This would be a constrained optimization problem, due to the fact that should the business owner not want to adversely affect their business (and still continue production), there likely is a production output constraint that they will have placed on them, limiting the possible choices they have.

Most machine learning problems readers will encounter are framed in the scope of a constrained optimization problem, and that constraint is usually a function of the data set being analyzed. The reason for this is often because prior to the development of deep learning models, this was the closest method by which we could approach artificial intelligence. Broadly speaking, most machine learning algorithms focused on regression are constrained optimization problems, where the objective is to minimize the loss of accuracy within a given model. As we briefly discussed in the previous toy problem, there are two kinds of minimizers: local and global.

Local Minimizers

Assume that $f:\mathbb{R}^n \to \mathbb{R}^m$ is a real-values function defined on some set $\Omega \in \mathbb{R}^n$. A point *x** is a *local minimizer* of f over Ω $f(x) \geq f(x^*)$ for all $x \in \Omega$.

Global Minimizers

Assuming the same function *f* and its tertiary properties, a point x* is a global minimizer of *f* over Ω if $f(x) \geq f(x^*)$ for all $x \in \Omega$.

Broadly speaking, there can be multiple local minimizers in a given problem, but if there is a global minimum, there can only been one. In Figure 3-2, we can see this with respect to a mapping of a function.

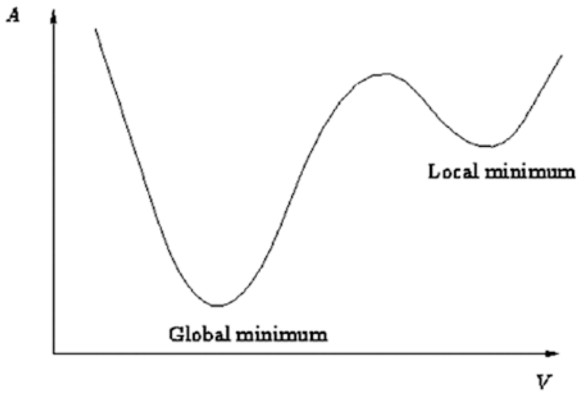


Figure 3-2. *Local versus global minima*

Depending on how much of the function we are evaluating at a given moment, we can choose a multitude of local minima. But if we're evaluating the full range of this function, we can see that there is only one global minimum. It is useful now to discuss how exactly we know that a solution we have reached, mathematically speaking, is optimal.

Conditions for Local Minimizers

In this section, we derive conditions for a point x* to be a local minimizer. We use derivatives of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Recall that the first-order derivative of *f*, denoted *Df* is

$$Df \triangleq \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]$$

The gradient of *f* is just the transpose of *Df*. The second derivative, or the Hessian of *f*, is

$$F(x) \triangleq D^2 f(x) = \begin{pmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_1 x_n} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{pmatrix}$$

The first derivate/gradient gives us the direction of an approximation of the function, *f*, at a specific point. The second derivative, or Hessian, gives us a quadratic approximation of *f* at a point. Both the Hessian and the gradient can be used to find local solutions for optimization problems. The gradient is used, as discussed earlier, for parameter updating such as in linear regression via gradient descent. However, the Hessian also can be used for parameter updating in the context of deep learning. I talk about this more later, but recurrent neural networks typically are used in the case of modeling data that occurs in sequences such as time series or text segments. Specifically, recurrent neural networks often are difficult to train by a product of certain data sequences having long-term data dependencies. When training other deep learning architectures, we encounter training problems due to the very large number of weights. This creates a large Hessian matrix, virtually making Newton's method defunct.

Hessian-free optimization focuses on minimizing an objective function where instead of computing the Hessian, we compute the matrix-vector product. Provided that the Hessian matrix is positive-definite, we converge to a solution. By solving the following equation, we can effectively use Newton's method on the weight matrix to train a network

$$H_p = \lim_{\epsilon \rightarrow 0} \frac{\nabla f(\theta + \epsilon d) - \nabla f(\theta)}{\epsilon}$$

where H_p is the matrix-vector product, θ is some parameter (in this case, weights), and *d* is a user determined value.

In cases where the Hessian matrix is not positive-definite, convergence upon a solution is not guaranteed and leads to radically different results. We can, however, approximate the Hessian matrix using a Gauss-Newton approximant of the Hessian, whereupon the Hessian equals

$$H = J^T J$$

where J is the Jacobian matrix of the parameter.

This yields a guaranteed positive-definite matrix and therefore validates the assumptions necessary to guarantee convergence. Moving forward from regression, I want to discuss one of the mathematical underpinnings of classification algorithms: neighborhoods.

Neighborhoods

Neighborhoods are an important concept in the paradigm of classification algorithms. For example, the preeminent algorithm that uses this concept is K-nearest neighbors. One of the simpler algorithms, the user-defined K parameter determines the number of neighboring data points that are used to ultimately classify an object to a class of points. We define a *neighborhood* of a point as a set of points containing the aforementioned point without leaving the set. Consider a point $x \in \mathbb{R}^n$. A neighborhood of this set would be the equation

$$\left\{ y \in \mathbb{R}^n : \|y - x\| < \epsilon \right\}$$

where ϵ is some positive number defined in a given context. ϵ represents the bound that defines the size of a given neighborhood. Visually, we can consider a neighborhood as a sphere, or a space between two half spaces, with x as the center and ϵ as the radius, as in Figure 3-3.

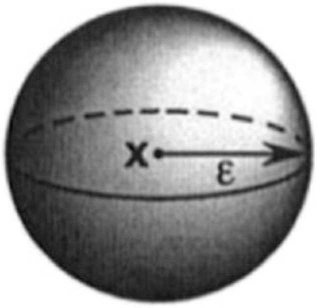


Figure 3-3. *Visualization of a neighborhood of a point x*

This will be an important to understanding any algorithms that use epsilon intensive loss to define separation of observations. Epsilon intensive loss is used particularly in the case of support vector machines, but K-nearest neighbors draws upon the concept of neighborhoods broadly to define observations within a given class.

Interior and Boundary Points

A point $x \in S$ is said to be an interior point of the set S if the set S contains some neighborhood of x. If all points within some neighborhood of x are also in S, the set of all the interior points of S is called the interior of S. A point x is said to be a boundary point of the set S if every neighborhood of x contains a point in S and a point not in S. Similarly, all the boundary points of S take the name boundary. A set is *open* if it contains a neighborhood of each of its points, or has no boundary points. A set is *closed* if it contains its boundary. A set is *compact* if it is both closed and bounded.

We've now reached the conclusion of our review of optimization. Now that we have addressed the prerequisite information necessary, we can discuss machine learning in depth and grasp the broader implications of the algorithms within this paradigm.

Machine Learning Methods: Supervised Learning

Machine learning can be segregated into two broad paradigms: supervised and unsupervised learning. *Supervised learning* is distinguished by the fact that prior to fitting a model, we know what the label/response variable Y is. As such, we can evaluate the efficacy of a model in an efficient manner. In *unsupervised learning*, we don't have this information, which doesn't allow us to determine the degree to which we are correct. Prior to discussing the challenges of both paradigms, it's reasonable to discuss the development of this field

History of Machine Learning

Machine learning was developed to create artificial intelligence in the mid-1950s. Its focus shifted towards creating programs that improved upon iteration, but were specifically made to accomplish one task and could generally be viewed as a method of function optimization. Artificial intelligence eventually began to become its own field, and as the end of the 20th century came, machine learning started to become a more developed and mature science. Machine learning takes contributions and inspiration from many fields, such as statistics and computer science, and the overlap is such that many statistics programs often include and encourage their students to become well versed in the techniques. The upcoming sections will address some of the most common machine learning algorithms, including some of those that serve as inspiration for the deep learning models described in the following chapters.

What Is an Algorithm?

Prior to this point, I've occasionally referred to algorithms. Simply stated, an *algorithm* is a process that we create for the purpose of accomplishing some task. In the following section, prior to tackling deep learning models in the next chapters, we will review important machine learning algorithms that will be utilized in deep learning models in addition to algorithms that are useful in the general practice of data science.

Regression Models

Regression refers to a set of problems in which we are trying to predict specific values. These could be prices of homes, the salary of an employee, or the length of a flower petal. More importantly, regression can also be used to measure the degree to which an explanatory variable(s), x, affect the response variable, Y.

Linear Regression

Imagine we're trying to predict the television ratings of a given show. We know, due to prior research, that the most popular demographic for this show is people aged 25–50 years old. We also see that there is a strong linear correlation between these two variables. As such, we decide to consider this our explanatory, or x, variable and the ratings our response, or Y, variable. How exactly would we proceed? Simple linear regression would be the most logical method. *Simple linear regression* utilizes relatively basic concepts for modeling explanatory variable(s), x, to a response variable, Y. Here we have the model

$$E(Y) = \beta_0 + \beta_1 x_1 + \ldots + \beta_k x_k,$$

where β_0 is the y-intercept and β_1 through β_k are the partial slopes corresponding to each explanatory variable x_1 through x_k, where k = 1, 2, ..., m, and m = the number of explanatory variables. This is known as a li*near probabilistic model*, because we're modeling the expectation of Y based on the assumption that it lies somewhere within a distribution of possible points from the ordinary least squares prediction of the point.

Ordinary Least Squares (OLS)

Ordinary least squares is the most basic form of linear regression. The intuition behind why we pick that specific E(Y) value at a specific point x is that we want to find a value for E(Y) that minimizes the squared difference between the actual and predicted Y. When the preceding assumptions are met in a given experiment, we find that the OLS method yields minimum variance and unbiased estimator of Y and also is the maximum likelihood estimator for Y.

Assumptions underlying this model are the following:

- Error terms are normally distributed.
- There is constant variance when observing the error terms.
- Observations of data are independently and identically distributed.
- There is no multicollinearity across explanatory variables.

The intuition behind why we pick that specific E(Y) value at a specific point x is that we want to find a value for E(Y) that minimizes the squared difference between the actual and predicted Y. When the preceding assumptions are met in a given experiment, we find that the OLS method yields minimum variance and unbiased estimator of Y and also is the maximum likelihood estimator for Y. Imagine we have an xy plot, similar to the one show in Figure 3-4.

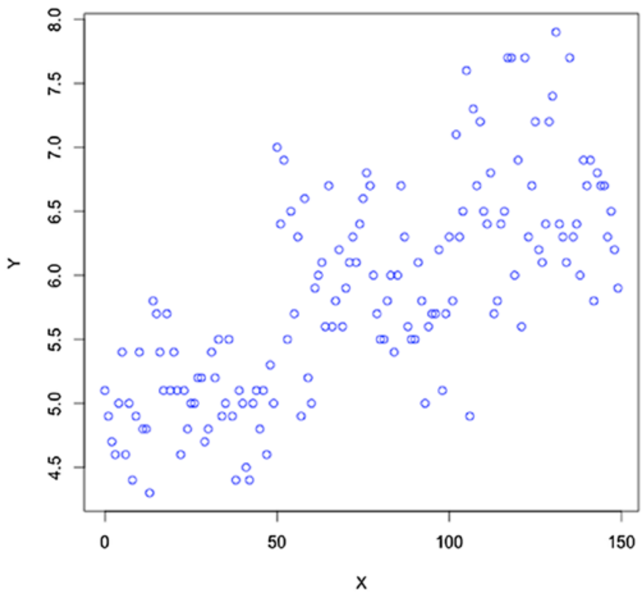


Figure 3-4. *Plotting of the response variable x*

There are in theory an infinite number of E(Y) line plots that we could make. However, only one solution yields an optimal solution that minimizes the error between E(Y) and y the most. Assuming there is only one explanatory variable, we derive the regression coefficient as the following:

$$\hat{\beta} = \left(\frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2} \right)$$

Alternatively, the regression coefficient equation can be written

$$\hat{\beta} = \arg\min_{\beta} \left\| y - x\beta \right\|$$

given by

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T y$$

The purpose of what we are doing here is to minimize the magnitude of the regression coefficient so that when we multiply it by x. Simply stated, we are trying to fine a line of best fit between the data and the regression line such that we minimize the average error between the predictions and actual data points. After we derive the regression coefficient, we can find the y-intercept, or the value of y when x = 0, as the following:

$$\beta_0 = \bar{y} - \bar{x}\beta$$

From this, we have all of the components of the E(Y) equation and can now model the data.

That said, using OLS to find a solution is not always the most optimal method. In cases of relatively small and simple data, utilizing OLS isn't particularly a problem. When data is complex and large, and we haven't satisfied the assumptions of OLS regression, it can be more effective to utilize the gradient descent method.

Gradient Descent Algorithm

As mentioned, the gradient of a function represents the point of greatest rate of increase in the function, and its magnitude is the slope of the graph in that direction. With that in mind, how can we apply the concept of a gradient to an algorithm in order to iteratively improve that? Gradient descent is an iterative algorithm in which you update a parameter by the negative of the gradient subject to some threshold you define or a certain number of iterations. The gradient is usually multiplied by a learning rate, which determines the speed of convergence toward an optimal solution for the function.

In the context of linear regression, our goal is to minimize the residual value between y^ and y, known as the *error function*, given by

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^i\right) - y^i\right)^2$$

where $h_\theta(x^i)$ is the predicted y value.

If our objective is to minimize the cost function as quick as possible, and the gradient is the vector that points in the steepest direction, we want to take the gradient of the cost function. The gradient is given by the following:

$$\frac{d}{d\theta_0}(\theta_0,\theta_1)=\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right)-y^i\right)$$

$$\frac{d}{d\theta_1}(\theta_0,\theta_1)=\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right)-y^i\right)\left(x_j\right)$$

To update the parameters, both the y-intercept and the regression coefficient, we calculate the following until the algorithm converges upon an optimal solution:

$$\theta_0:=\theta_0-\alpha\frac{d}{d\theta_0}(\theta_0,\theta_1)$$

$$\theta_1:=\theta_1-\alpha\frac{d}{d\theta_1}(\theta_0,\theta_1)$$

Multiple Linear Regression via Gradient Descent

The intuition behind multiple linear regression via gradient descent is the same as with simple linear regression—there is just a modification to accommodate for the multiple partial slopes being adjusted upon each iteration:

$$\theta_0:=\theta_0-\alpha\frac{d}{d\theta_0}(\theta_0,\theta_1,\ldots,\theta_n)$$

$$\theta_j:=\theta_j-\alpha\frac{d}{d\theta_j}(\theta_0,\theta_1,\ldots,\theta_n)$$

Learning Rates

One last aspect to discuss is the *learning rate*, denoted as α, which in fact is one of the most important aspects of the gradient descent algorithm. The learning rate determines the speed at which the gradient descent algorithm converges upon an optimal solution. Usually, the learning rate is initialized at a relatively small value—typically .01 or less. That said, choosing an optimal learning rate isn't necessarily always obvious, and not doing so can affect the "solution" yielded. Usually, gradient descent algorithms have two stopping conditions: 1) an optimal solution has been found, and 2) the maximum number of iterations allowed have been reached. The following problems associated with poor algorithm performance are due to the following situations:

- *The learning rate is too small*: In the instance that we choose a learning rate that's too small, the solution that the algorithm gives is in fact not the optimal solution, and we reach it due to the second stopping condition. Some might say that a way to avoid this is by choosing a learning rate is to increase the number of iterations, but that very well can defeat the purpose of this method, which is its computational efficiency.
- *The learning rate is too large*: If we are to choose a learning rate that is considerably larger than necessary, we may also never reach an optimal solution, though this one is due to a different reason. When the learning rate is too large, we find that the cost function upon each iteration may overcorrect and give us updated values for the coefficient that are far too small or far too large. As such, our reaching a solution would be by luck, and in most cases we would end up reaching the maximum solution.

Choosing An Appropriate Learning Rate

Now that we have an understanding of what the problems associated with choosing an incorrect learning rate are, we need to find out how to choose one. One possible solution is to hardcode various gradients and see how the algorithms perform across each iteration. In the following method, we update the step size upon each iteration of the gradient descent algorithm.

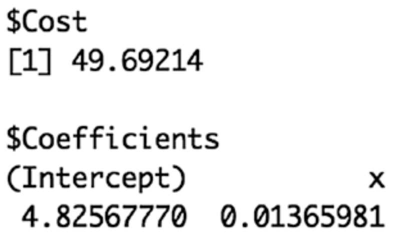
The bold driver approach compares the most recent gradient value to the gradient value derived upon the prior iteration. If the error has decreased, increase the learning rate by a moderate amount. If the error has increased, decrease the learning rate by 50%.

In the following code example, we are modifying the iris data set. This data set dates back to Ronald Fisher; he used it for an initial set of experiments. It is popular when displaying fundamental aspects of various statistical and machine learning algorithms. Here, we're taking the first column of the iris data set and modeling that against an X variable (which is merely length), such that the data displayed forms a linear pattern. This is just an example to display the mechanics of OLS linear regression. In the following code, we fit the data to the OLS regression via the `lm()` function. We then calculate the sum of squared residuals, which we denote as `Cost` in the output. We then extract the regression coefficients for this model from the `lm()` function and then output these two attributes in a data frame:

```
#Modifying Data From Iris Data Set
data(iris)
Y <-  matrix(iris[,1])
X <-  matrix(seq(0,149, 1))
```

```
olsExample <- function(y = Y, x = X){
 y_h <- lm(y ~ x)(1)
  y_hf <- y_h$fitted.values
  error <- sum((y_hf - y)^2) (2)
  coefs <- y_h$coefficients (3)
  output <- list("Cost" = error, "Coefficients" = coefs)
  return(output)
}
```

When we run the code, we observe the results shown in Figure 3-5.



```
$Cost
[1] 49.69214

$Coefficients
(Intercept)           x
 4.82567770  0.01365981
```

Figure 3-5. *Output of OLS regression function*

`Cost` is the sum of squares and the `Coefficients` accordingly are listed as the y intercept followed by the partial slope for the x variable. We will use this as a baseline for comparing the performance of linear regression via gradient descent. Again, the purpose of this explanation is to show the efficiency and ability of the gradient descent algorithm to replicate the results of a simple OLS regression:

```
#Gradient Descent Without Adaptive Step
gradientDescent <- function(y = Y, x = X, alpha = .0001, epsilon = .000001,
maxiter = 300000){
  #Intializing Parameters
  theta0 <-  0
  theta1 <-  0
  cost <- sum(((theta0 + theta1*x) - y)^2)
  converged <- FALSE
  iterations <- 1
```

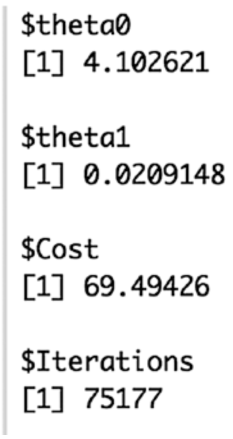
Moving forward, we define a function for the implementation of linear regression via gradient descent. This gradient descent algorithm has a constant learning rate, though you can alter this parameter, as well as the loss tolerance, should you choose to use this implementation on other data sets. We have defined the maximum amount of iterations as 300,000, which will force the algorithm to cease at the solution should it not reach an optimal one before that. When analyzing the code specifically, we begin by initializing the parameters `theta0` and `theta1` at 0. Users may feel free to alter the code and initialize the parameters with values randomly sampled from a normal distribution, but should divide these values by 10 to ensure that they are not overly large. We initialize the `cost` function as the SSR of 0 minus all y values, from which we will begin to alter the parameters:

```
#Gradient Descent Algorithm
while (converged == FALSE){
  gradient0 <- as.numeric((1/length(y))*sum((theta0 + theta1*x) - y))
  gradient1 <- as.numeric((1/length(y))*sum((((theta0 + theta1*x) - y)*x)))

  t0 <- as.numeric(theta0 - (alpha*gradient0))
  t1 <- as.numeric(theta1 - (alpha*gradient1))

  theta0 <- t0
  theta1 <- t1

  error <- as.numeric(sum(((theta0 + theta1*x) - y)^2))

  if (as.numeric(abs(cost - error)) <= epsilon){
    converged <- TRUE
  }
    cost <- error
    iterations <- iterations + 1
  if (iterations == maxiter){
    converged <- TRUE
  }
}
```

Although we haven't converged on a solution, or we have not reached the maximum amount of iterations allowed when executing the function, we create the `gradient0` and `gradient1` variables, which correspond to the parameters `theta0` and `theta1` respectively. We then update the `theta0` and `theta1` parameters using the gradient contained within the `gradient0` and `gradient1` variables. After this, we calculate the error, and continue looping from `while` (`converged == FALSE`) until the stopping condition has been reached:

```
  output <- list("theta0" = theta0, "theta1" = theta1, "Cost" = cost,
  "Iterations" = iterations)
  return(output)
}
```

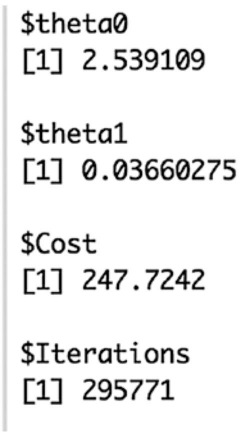
Here, we're running a simple linear regression where the `y` and `x` variables are initialized randomly. When we run the code as stated, we get the results shown in Figure 3-6.



```
$theta0
[1] 4.102621

$theta1
[1] 0.0209148

$Cost
[1] 69.49426

$Iterations
[1] 75177
```

Figure 3-6. *Output of gradient descent without adaptive step function*

`theta0` is the y-intercept, `theta1` is the partial slope for the `x` variable, `cost` is the sum of squares, and `iterations` is the number of iterations performed. Here, we observe lower regression coefficients that are roughly the same baseline sum of squares error. However, if we're to use a learning rate that's too large, we often will get an error because the regression coefficients have become infinitely large. When the learning rate is too small, we notice what's shown in Figure 3-7.



```
$theta0
[1] 2.539109

$theta1
[1] 0.03660275

$Cost
[1] 247.7242

$Iterations
[1] 295771
```

Figure 3-7. *Output of gradient descent with small learning rate*

We see that the algorithm doesn't converge upon the minimum, but reaches a feasible solution and is cut off by the loss tolerance we set. Incidentally, we're also near the maximum number of iterations allowed. The consequence of incorrectly choosing an algorithm is relative to the context in which a given algorithm is being applied. But all users should be careful to evaluate the results they find on any machine learning or deep learning algorithm. As such, it's important that we are as confident as humanly possible when choosing a solution.

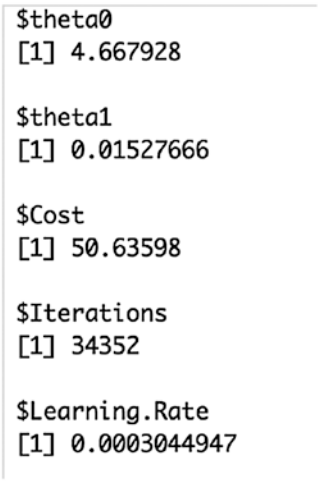
In the next example, we run the same algorithm using an adaptive step size to compare the performance:

```
#Gradient Descent with Adaptive Step
adaptiveGradient <- function(y = Y, x = X, alpha = .0001, epsilon = .000001,
maxiter = 300000){
  #Intializing Parameters
  theta0 <-  0
  theta1 <-  0
  cost <- sum(((theta0 + theta1*x) - y)^2)
  converged <- FALSE
  iterations <- 1

  #Gradient Descent Algorithm
  while (converged == FALSE){
    gradient0 <- as.numeric((1/length(y))*sum((theta0 + theta1*x) - y))
    gradient1 <- as.numeric((1/length(y))*sum((((theta0 + theta1*x) - y)*x)))

    t0 <- as.numeric(theta0 - (alpha*gradient0))
    t1 <- as.numeric(theta1 - (alpha*gradient1))

    delta_0 <- t0 - theta0
    delta_1  <- t1 - theta1
    if (delta_0 < theta0){
      alpha <- alpha*1.10
    } else {
      alpha <- alpha*.50
    }
```

Here, we apply the same gradient descent function, except now we apply the bold driver approach so that we have an adaptive learning rate. The bold driver approach alters the learning rate from one individual iteration to the next based on the prior result. Simply stated, if the gradient increases from one iteration to the next, the learning rate increases by 10%. If the gradient decreases, we decrease the learning rate by 50%. Readers can feel free to alter these parameters should they choose, to experiment on the results received:

```
    theta0 <- t0
    theta1 <- t1
    error <- as.numeric(sum(((theta0 + theta1*x) - y)^2))
    if (as.numeric(abs(cost - error)) <= epsilon){
      converged <- TRUE
    }
    cost <- error
    iterations <- iterations + 1
    if (iterations == maxiter){
      converged <- TRUE
    }
  }
  output <- list("theta0" = theta0, "theta1" = theta1, "Cost" = cost,
  "Iterations" = iterations, "Learning.Rate" = alpha)
  return(output)
}
```

Upon executing this function, we receive the results shown in Figure 3-8.



```
$theta0
[1] 4.667928

$theta1
[1] 0.01527666

$Cost
[1] 50.63598

$Iterations
[1] 34352

$Learning.Rate
[1] 0.0003044947
```

Figure 3-8. *Output of gradient descent with adaptive learning rate functions*

Of the algorithms we have tested, and given our objective to minimize the cost and the regression coefficient size, linear regression via gradient descent *with* adaptive step size or the OLS method would be acceptable. As an interesting observation, the algorithm converged upon this solution significantly faster than the gradient descent method with a static learning rate did.

Newton's Method

For instances in which we're looking to minimize a quadratic function, Newton's method often proves useful. Newton's method is a way to find the roots of a function, or where f(x) is equal to 0. It was developed by Isaac Newton and Joseph Raphson. To calculate an optimal point, we derive the equation

$$x^{k+1} = x^k - \left(\frac{f'\left(x^k\right)}{f''\left(x^k\right)}\right)$$

where f' is the first derivative of a given function and f" is the second derivative of a given function. This is known as the *secant method*. Newton's method works particularly well if the f'"(x) > 0, but if f"(x) < 0, it might not converge upon a global minimum. We know that Newton's function will always converge to a global optimum if the Hessian of the function is positive semi-definite. Another drawback to Newton's method is also that convergence is not guaranteed if the starting point is considerably far from the global minimum. In the instance that Newton's method doesn't converge upon the global minimum, there is a heuristic that can be used to overcome this, covered next.

Levenberg-Marquardt Heuristic

The Levenberg-Marquardt (LM) algorithm is most applicable when a function isn't twice differentiable or its Hessian matrix isn't positive-definite. The equation is given by the following:

$$x^{k+1} = x^k - \left(F\left(x^k\right) + \mu_k I\right)^{-1} g^k$$

Consider a square matrix F that isn't positive-definite. The eigenvalues of this matrix may not be positive but are all real numbers. Consider a matrix

$$G = F + \mu I$$

where $\mu \geq 0$. The eigenvectos of G are $\lambda + \mu$. Therefore, the following must be true:

$$Gv_i = (F + \mu I)v_i$$

$$= Fv_i + \mu I v_i$$

$$= \lambda_i v_i + \mu v_i$$

$$= (\lambda_i + \mu)v_i,$$

With this modification, all the eigenvalues of G are therefore positive, and then G would have to be positive definite. If μ is also sufficiently large enough, we can confirm that the direction that Newton's algorithm chooses will always be toward the direction of steepest descent. The final modification to the algorithm will be to add in a step size:

$$x^{k+1} = x^k - \alpha\left(F\left(x^k\right) + \mu_k I\right)^{-1} g^k$$