Least squares regression by gradient descent

James Chuang

December 22, 2016

Contents

In this post I explore the gradient descent algorithm as a solution to least squares minimization, inspired by section 4.5 of the Deep Learning Book.

Suppose we want to find the value of $\mathbf{x}$ that minimizes the squared error loss function $f(\mathbf{x})$.

$$f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$
$$f(\mathbf{x}) = (\mathbf{A}\mathbf{x} - \mathbf{b})^T (\mathbf{A}\mathbf{x} - \mathbf{b})$$

The toy motivation for this could be to perform simple (univariate) linear regression on some data set. In this case, the matrix $\mathbf{A}$ is an $N \times (p+1)$ matrix representing N training examples with p input variables, with a leading column of ones to allow for an intercept. The vector $\mathbf{b}$ is an N-vector of the training outputs, and $\mathbf{x}$ is a $(p+1)$ vector representing the linear model's slope and intercept which we want to solve for. The function $f(\mathbf{x})$ above is the squared error loss, which we will use to determine how poorly a line parameterized by the slope and intercept of a particular $\mathbf{x}$ fits the training data. If we can find the value of $\mathbf{x}$ which minimizes the squared error loss $f(\mathbf{x})$, then we will have found the line which best fits the training data (the best line is the 'least bad' line, i.e. it has the least error).

In order to minimize $f(\mathbf{x})$, we find the gradient:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b})$$
$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{A}^T\mathbf{b}$$

Ordinarily, we would now set $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$, and then solve for $\mathbf{x}$. However, since I'm solving this by gradient descent, here the gradient acts as a guide pointing in the direction which $f(\mathbf{x})$ increases most sharply. Therefore, we can find the minimum of $f(\mathbf{x})$ by walking in small steps in the opposite direction of the gradient (hence the name, 'gradient descent').

Minimization of $f(\mathbf{x})$ by gradient descent in pseudocode:

- Set the step size (ϵ) and tolerance (δ) to small, positive numbers.
  - **while** $\|\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{A}^T\mathbf{b}\|_2 > \delta$:
    - $\mathbf{x} \leftarrow \mathbf{x} - \epsilon(\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{A}^T\mathbf{b})$

First, I generate linearly related training data with Gaussian noise and find the linear fit by ordinary least squares (this is a positive control for this experiment).

```
library(tibble)
library(ggplot2)
library(viridis)

set.seed(1)

#function to generate linearly related training data with noise
generate.data = function(N, slope, intercept, mean, sd){
  # randomly pick N x-values uniformly distributed from 1 to 10
  X = matrix(runif(n=N, min=0, max=10))

  # generate y-values for each x, with Gaussian noise
  Y = matrix((intercept+slope*X) + rnorm(n=N, mean = mean, sd = sd))
```

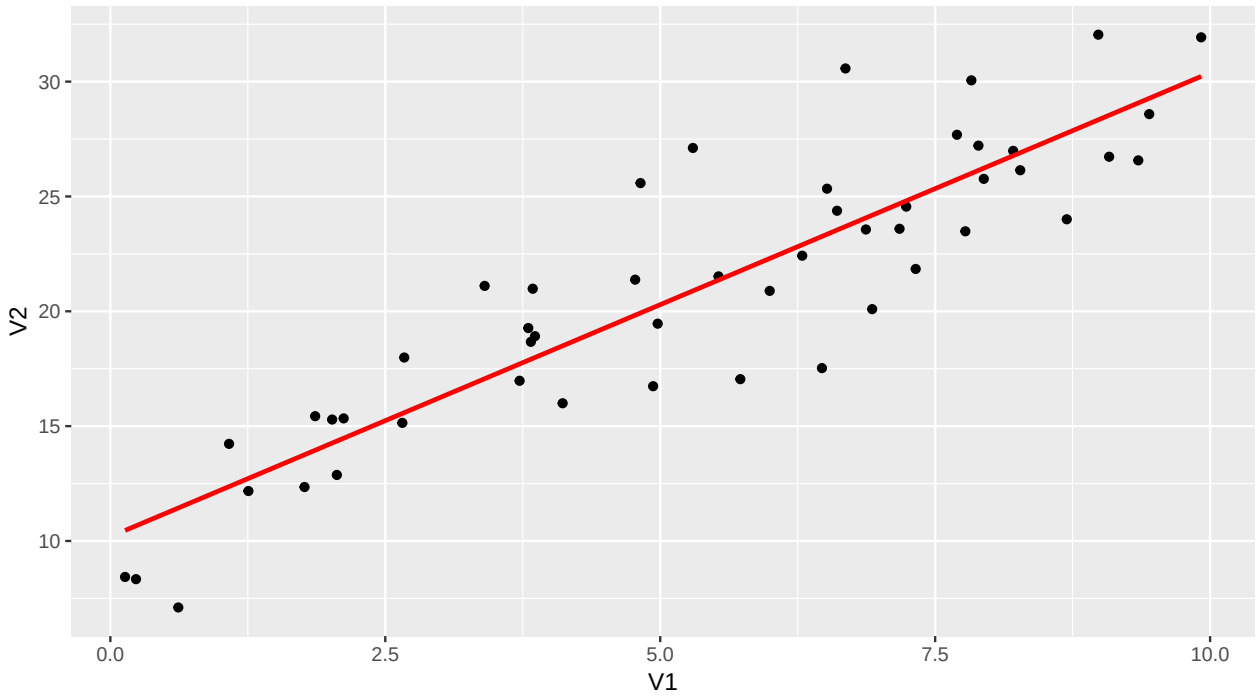


Figure 1: Linear fit by ordinary least squares. If gradient descent is any good, it should find a line close to the red line.

```
  return(cbind(X,Y))
}

data = generate.data(N=50, slope=2, intercept=10, mean=0, sd=3)
N = nrow(data)
A = cbind(rep(1,N), data[,1])
b = data[,2]

(ols.plot = ggplot(data = as_data_frame(data), aes(x=V1, y=V2)) +
            geom_point() +
            geom_smooth(method=lm, se=FALSE, color="red")
)
```

The above gradient descent pseudocode implemented in R:

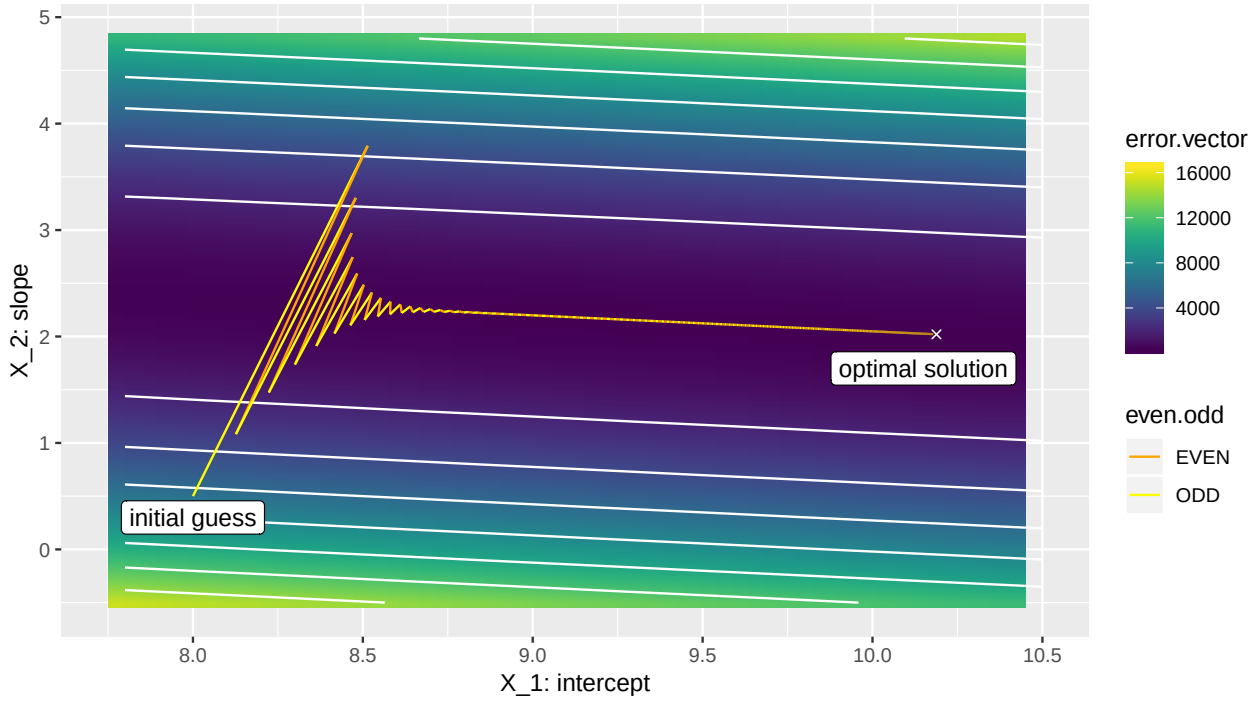
```
#returns L2 norm of error
error = function(A,x,b){
  return(norm(t(A) %*% A %*% x - t(A) %*% b, type="2"))
}

#gradient descent function
grad.desc = function(A, b, eps, delta, x){
  grad.desc.data = data_frame(epoch=numeric(),
                              x.old=double(),
                              y.old=double(),
                              x.new=double(),
                              y.new=double(),
                              error.old=double(),
                              error.new=double())
```

```
  epoch = 1
  err = error(A,x,b)

  while (err>delta){
    #uncomment below to get live output
    #print(paste("epoch: ", epoch,
                 # "; error: ", round(err, digits=6),
                 # "x: ", round(x[1], digits=3),
                 # ",",round(x[2], digits=3)))

    grad.desc.data[epoch,1] = epoch
    grad.desc.data[epoch,2] = x[1]
    grad.desc.data[epoch,3] = x[2]
    grad.desc.data[epoch,6] = err

    #the actual gradient descent occurs in this line:
    x = x-eps*(t(A) %*% A %*% x - t(A) %*% b)

    grad.desc.data[epoch,4] = x[1]
    grad.desc.data[epoch,5] = x[2]
    err = error(A,x,b)
    grad.desc.data[epoch,7] = err
    epoch = epoch+1
  }
  grad.desc.data$even.odd = ifelse(grad.desc.data$epoch%%2==0, "EVEN", "ODD")
  return(grad.desc.data)
}

grad.desc.data = grad.desc(A, b, eps=.001, delta=.1, x=c(8,0.5))
```

Now some plotting to look closer at what happened:

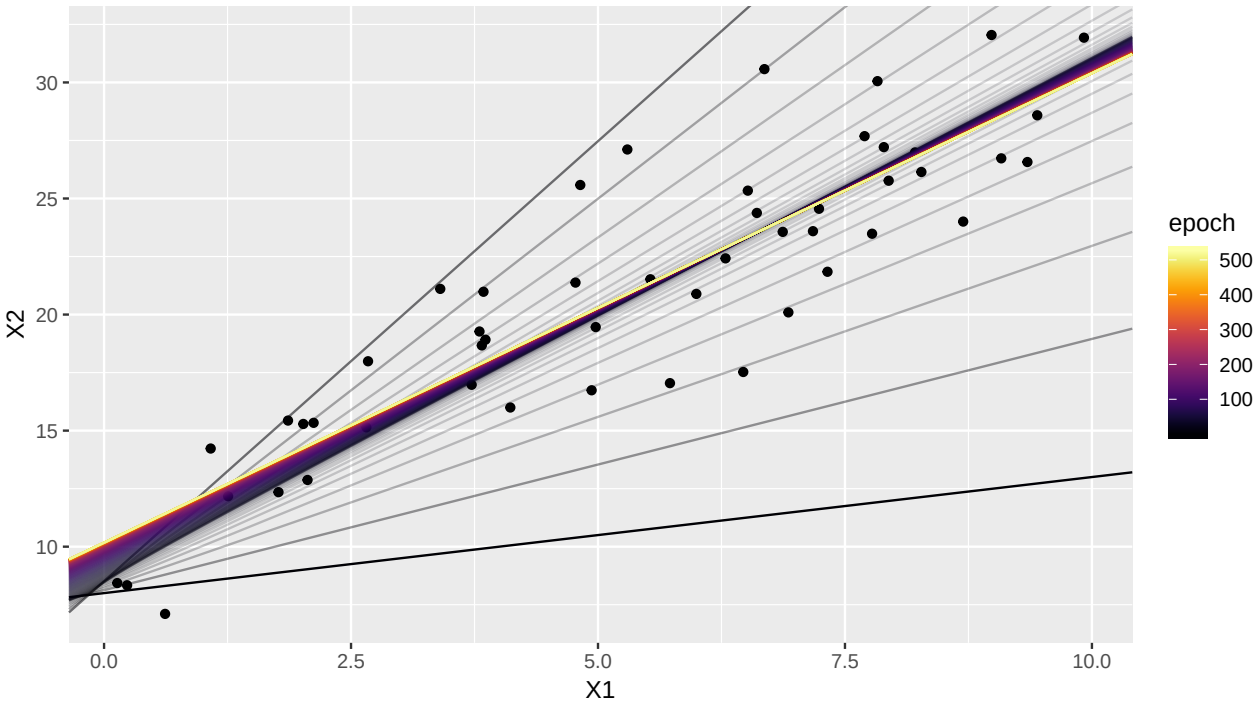
```
#Set data range for plotting
xmin = round(min(grad.desc.data$x.old), digits=1)-1
xmax = round(max(grad.desc.data$x.old), digits=1)+1
ymin = round(min(grad.desc.data$y.old), digits=1)-1
ymax = round(max(grad.desc.data$y.old), digits=1)+1

xrange = seq(from=xmin, to=xmax, by=0.1)
yrange = seq(from=ymin, to=ymax, by=0.1)

grid = as.matrix(expand.grid(xrange, yrange))
k = dim(grid)[1]

b.broadcast = b %*% t(rep(1, k))

error.vector = diag(t(b.broadcast-A%*%t(grid)) %*% (b.broadcast-A%*%t(grid)))
grid.error = as_data_frame(cbind(grid, error.vector))

lm.fit = lm(b~data[,1])
x.lm = as_data_frame(t(as.matrix(coef(lm.fit))))
names(x.lm) = c("V1","V2")

(error.plot = ggplot() +
```

```
        geom_raster(data= grid.error, aes(x=Var1, y=Var2, fill=error.vector), interpolate = TRUE) +
        geom_contour(data= grid.error, aes(x=Var1, y=Var2, z=error.vector), color="white") +
        geom_point(data= x.lm, aes(x=V1, y=V2), shape=4, color="white") +
        geom_segment(data= grad.desc.data,
                     aes(x=x.old, y=y.old, xend=x.new, yend=y.new, color=even.odd)) +
        scale_x_continuous(limits = c(7.75,10.5)) +
        scale_fill_viridis() +
        scale_color_manual(values = c("orange","yellow")) +
        xlab("X_1: intercept") + ylab("X_2: slope") +
        annotate("label",x=8,y=0.3, label="initial guess") +
        annotate("label",x=10.15,y=1.7, label="optimal solution")
)
```



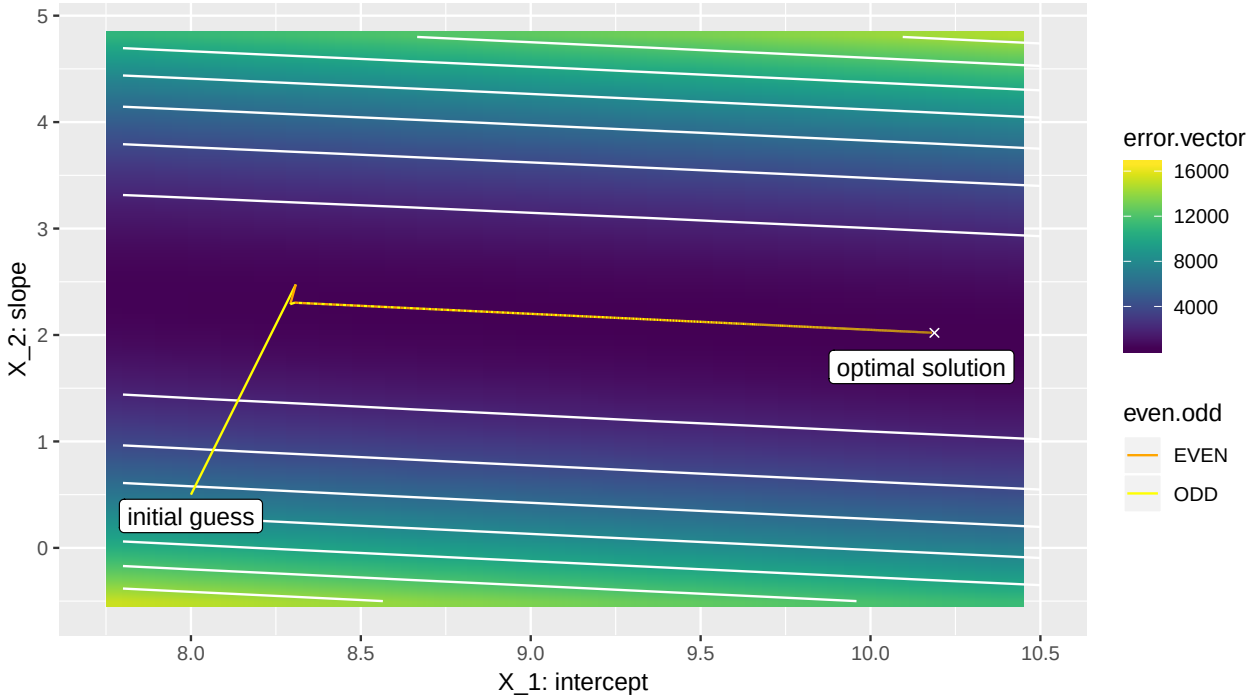
Above, we see gradient descent in action as plotted in parameter space, where the x-axis is x_1, the intercept, and the y-axis is x_2, the slope. The movement in each iteration of gradient descent is plotted in alternating yellow and orange. Gradient descent keeps taking small steps in the steepest direction downwards until the error decreases by less than the tolerance δ.

```
(lm.plot = ggplot() +
             geom_point(data=data.frame(data), aes(x=X1, y=X2)) +
             #geom_abline(slope=2, intercept=10) + #generating line
             #least squares analytical soln
             #geom_smooth(data=data.frame(data), aes(x=X1, y=X2), method = lm) +
             geom_abline(data=grad.desc.data,
                         aes(intercept=x.old, slope=y.old, color=epoch, alpha=1/epoch)) +
             scale_alpha(guide='none') +
             scale_color_viridis(option="inferno")
)
```



Above is what the line represented by the slope and intercept looks like through iterations of gradient descent. As the algorithm progresses (indicated by the progression of black to yellow), the line gets closer and closer to the optimal line.

In the above example, we see that gradient descent wastes time crossing the 'valley' in error space rather than going towards the optimal solution because going down the valley is the steepest direction. One way to improve this is to decrease the step size. Here's what gradient descent with a step size of $\epsilon = .0005$ (half the previous step size) looks like in parameter space:



Now the initial steps are not large enough to reach the other side of the 'valley', so less time is spent repeatedly crossing the valley. However, the initial steps are still limited by the fact that gradient descent only goes in the direction of steepest descent, which in this

case is perpendicular in parameter space to the optimal solution. This can be avoided by taking into account information about the curvature of the function, i.e. its second derivatives which are contained in the Hessian matrix. Algorithms which take the Hessian into account are known as second-order optimization algorithms, while the algorithm used here only uses the information in the gradient, making it a first-order optimization algorithm.