Gradient Descent

Minimizing loss functions

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Minimizing the loss (error)

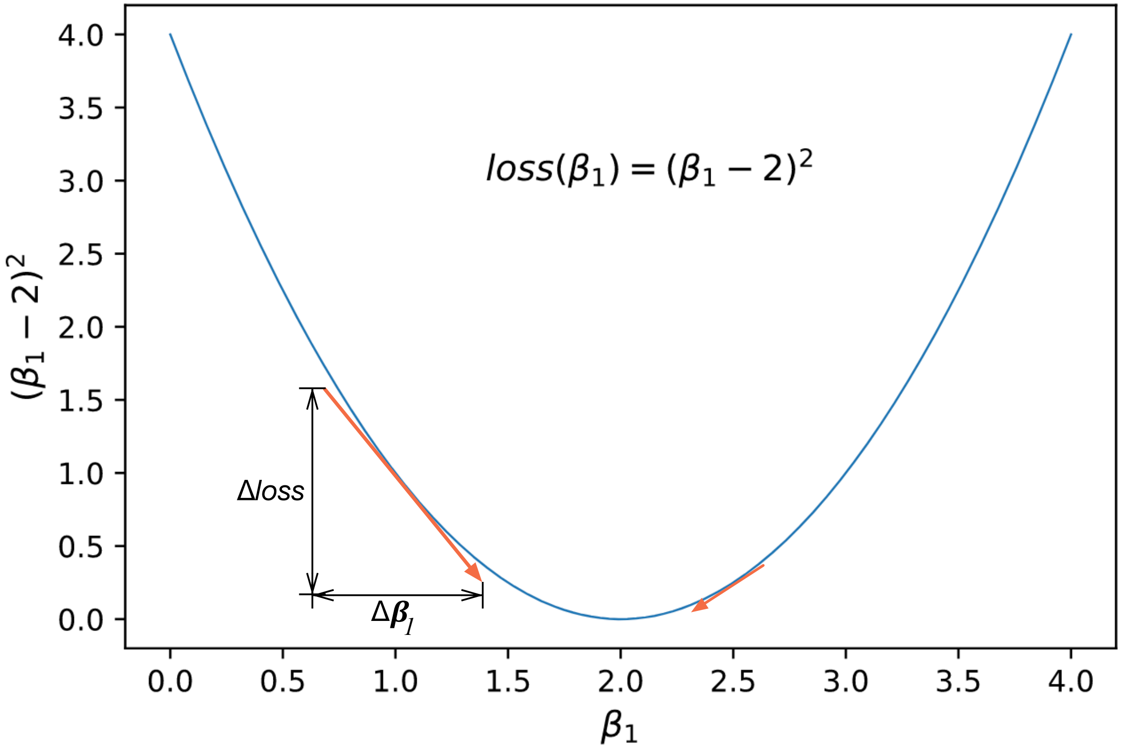
- We need a way to find β such that: $\underset{\beta}{\arg\min}\, \mathscr{L}(\beta)$
- Could try random β vectors and choose the β with lowest loss
- Or, better yet, choose a random β and then tweak with some $\Delta\beta$ in the downhill loss direction until any tweak would increase loss

$$\beta^{(t+1)} = \beta^{(t)} + \Delta\beta^{(t)}$$

- That must mean we are at some kind of bottom and cannot go further down in any direction, only up
- We could not minimize the loss further

How do we pick a direction to move?

- Use information (*gradient*) from loss function in vicinity of current β_1



- Derivative/slope of loss(β_1) is 2(β_1-2), which points in direction of increased loss
- Derivative of loss for β_1 < 2 is negative and derivative > 2 is positive
- What is derivative of loss at β_1=2?
- Direction of min loss is opposite of derivative
- Derivative also has magnitude

UNIVERSITY OF SAN FRANCISCO

Taking steps in right direction

- Direction of min loss is <u>opposite</u> of derivative so let's step in negative of derivative and scale it with a learning rate η:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \frac{d}{d\beta} \mathscr{L}(\beta^{(t)})$$

```
while True:
    b = b - rate * gradient(b)
```

Python gradient descent implementation

- First define loss function and its gradient:

```
def f(b) : return (b-2)**2
def gradient(b): return 2*(b-2)
```

- Then, pick a random starting point and learning rate

```
b = np.random.uniform()
rate = .2
```

- Loop until we've made some progress or until gradient(b)==0

```
for t in range(10): # for awhile
    b = b - rate * gradient(b)
```

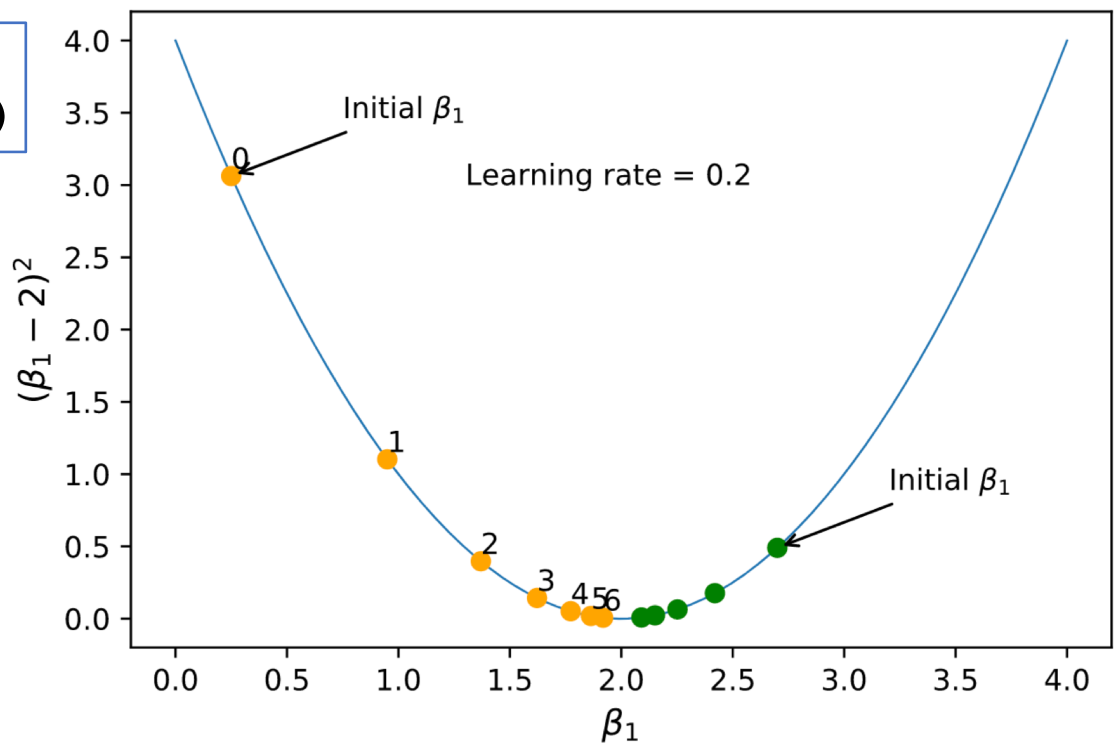
See https://github.com/parrt/msds621/blob/master/notebooks/linear-models/viz-gradient-descent.ipynb

Sample 1D gradient descent run

```
for t in range(7):
    b = b - rate * gradient(b)
```

b^(0)=0.25, f(beta_1)=3.06, gradient -3.50
b^(1)=0.95, f(beta_1)=1.10, gradient -2.10
b^(2)=1.37, f(beta_1)=0.40, gradient -1.26
b^(3)=1.62, f(beta_1)=0.14, gradient -0.76
b^(4)=1.77, f(beta_1)=0.05, gradient -0.45
b^(5)=1.86, f(beta_1)=0.02, gradient -0.27
b^(6)=1.92, f(beta_1)=0.01, gradient -0.16

Notice that β_1 accelerates and then slows down. Why?

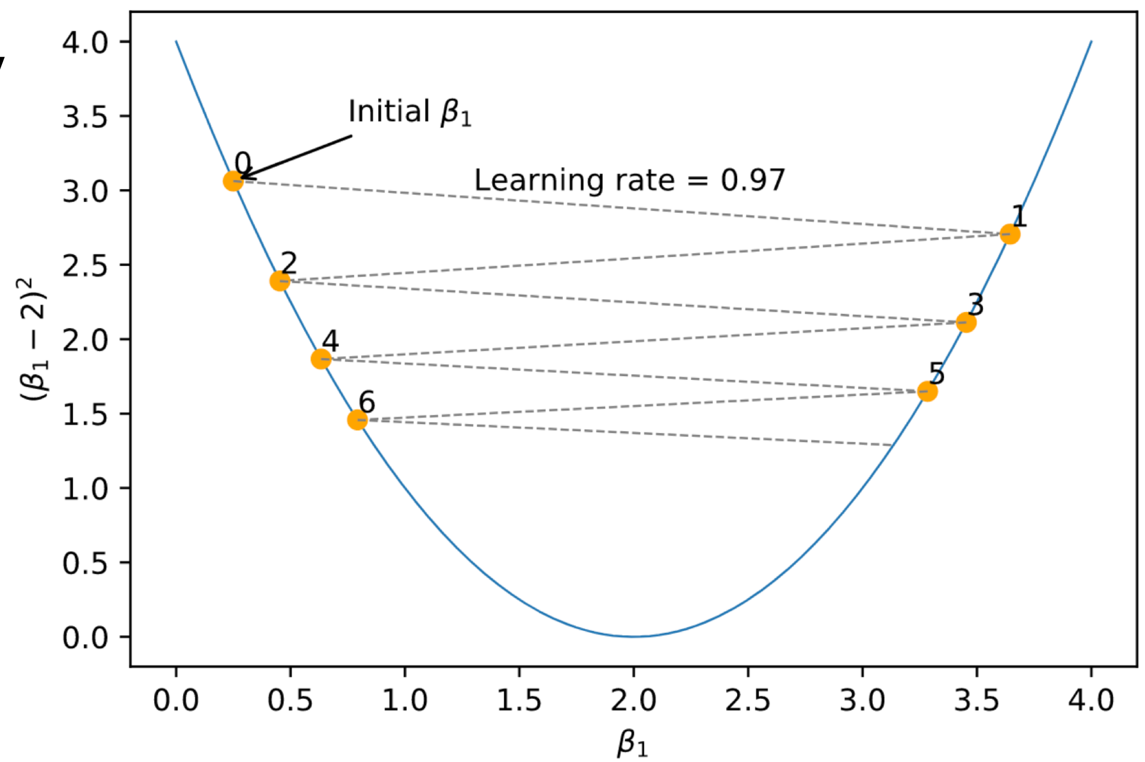


UNIVERSITY OF SAN FRANCISCO

See https://github.com/parrt/msds621/blob/master/notebooks/linear-models/viz-gradient-descent.ipynb

What if we crank up learning rate?

- β_1 oscillates across valley
- Picking learning rate is trial and error for our purposes but small like η=.00001 is a reasonable guess to start out
- If too small, we don't make progress to min

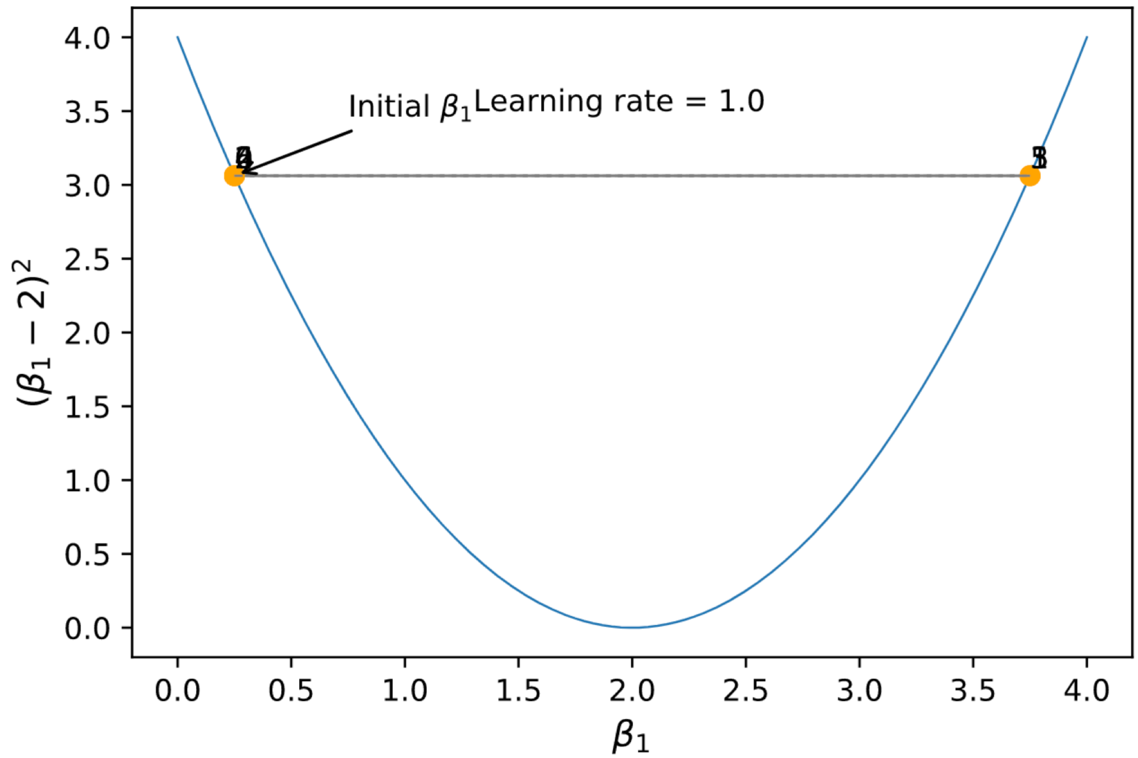


What if learning rate is really too high?

- We get nowhere

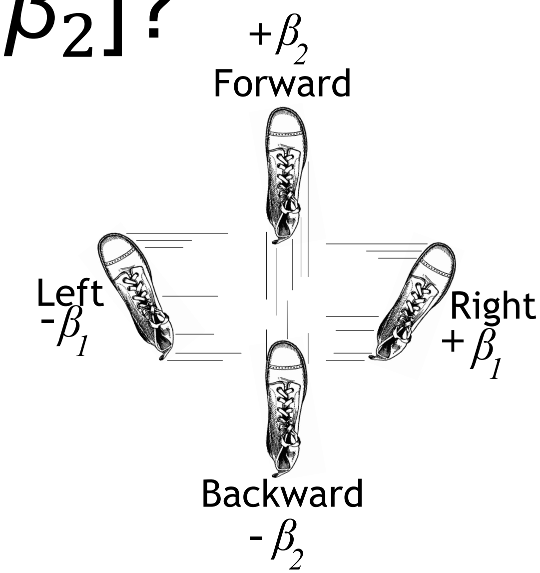
b^(0)=0.25, f(beta_1)=3.06, gradient -3.50
b^(1)=3.75, f(beta_1)=3.06, gradient 3.50
b^(2)=0.25, f(beta_1)=3.06, gradient -3.50
b^(3)=3.75, f(beta_1)=3.06, gradient 3.50
b^(4)=0.25, f(beta_1)=3.06, gradient -3.50
b^(5)=3.75, f(beta_1)=3.06, gradient 3.50
b^(6)=0.25, f(beta_1)=3.06, gradient -3.50

- It can even diverge, exploding β_1



UNIVERSITY OF SAN FRANCISCO

What happens in 2D for $\beta = [\beta_1, \beta_2]$?



- Imagine you're stuck on a mountain in the dark and need to get to the bottom
- Take steps to left, right, forward, backward or at an angle to minimize the "elevation function"
- Treat each direction separately, then combine them to obtain the best step direction
- Each direction's slope is a *partial derivative* and, combined, are the *gradient* vector

General gradient descent

- Partial derivative is rate of change in one direction: $\frac{\partial}{\partial \beta_i}\mathscr{L}(\beta)$
- Combining *p* partial derivatives into vector gives the *gradient:* ∇_β
- Gradient points in direction of increased loss, so must go in negative gradient direction to decrease loss as before:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \nabla_\beta \mathscr{L}(\beta^{(t)})$$ where *η* is a learning rate

- Gradients have magnitude and direction
- E.g., *β*=[-1,2] means take a step to left, bigger step forward
- Take a single step: *β* = *β* – *η x* [-1, 2]
- In any direction, the partial of the loss function is 0 when flat
- When gradient vector = 0 vector, we're at min loss

Update equation needs gradient:

$$\beta^{(t+1)} = \beta^{(t)} - \nabla_\beta \mathscr{L}(\beta^{(t)})$$

Gradient of $\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}'\beta) \cdot (\mathbf{y} - \mathbf{X}'\beta)$ for lin regression is

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\beta)$$

So update equation becomes (adding *learning rate* $\boldsymbol{\eta}$):

$$\beta^{(t+1)} = \beta^{(t)} - \eta\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\beta^{(t)})$$

η scales the step we take each update

Simplest gradient descent algorithm

Algorithm: $basic\_minimize(\mathbf{X},\ \mathbf{y},\ \mathscr{L},\ \eta)$ **returns** coefficients $\vec{b}$

Let $\vec{b} \sim 2N(0,1) - 1$ $\quad$ (*init b with random* $p+1$*-sized vector with elements in [-1,1)*)

$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$ $\quad$ (*Add first column of 1s to data*)

repeat

$\quad \nabla_{\vec{b}} = -\mathbf{X}'^T(\mathbf{y} - \mathbf{X}'\vec{b})$

$\quad \vec{b} = \vec{b} - \eta\nabla_{\vec{b}}$

until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;

return $\vec{b}$

For your projects, you must implement fancier Adagrad version, which has and adjusts a learning rate per dimension per next slide

General adagrad gradient descent

- Single learning rate for all dimensions is brutally slow
- Imagine long shallow valley with steep walls
- η small enough for steep walls is way too slow for other, shallow dimension

Algorithm: *adagrad_minimize*($\mathbf{X}$, $\mathbf{y}$, $\mathscr{L}$, $\nabla\mathscr{L}$, η, ϵ=1e-5) **returns** coefficients $\vec{b}$

Let $\vec{b} \sim 2N(0,1) - 1$ (*random* $p+1$*-sized vector with elements in [-1,1)*)

$h = \vec{0}$ ($p+1$*-sized sum of squared gradient history*)

$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$ (*Add first column of 1s*)

repeat

$\quad \vec{h} \mathrel{+}= \nabla\mathscr{L} \otimes \nabla\mathscr{L}$ (*track sum of squared partials, use element-wise product*)

$\quad \vec{b} = \vec{b} - \eta * \frac{\nabla\mathscr{L}}{(\sqrt{\vec{h}}+\epsilon)}$ ← normalize update per β_i; low h for β_i increases its learning rate

until $|(\mathscr{L}(\vec{b}) - \mathscr{L}(\vec{b}_{prev}))| < precision$;

return $\vec{b}$

See http://cs231n.github.io/neural-networks-3/#ada

UNIVERSITY OF SAN FRANCISCO

Loss, gradient functions for minimization

- Linear regression

$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}'\beta) \cdot (\mathbf{y} - \mathbf{X}'\beta)$$

$$\nabla_{\beta}\, \mathscr{L}(\beta) = -2\mathbf{X}'^{T}(\mathbf{y} - \mathbf{X}'\beta)$$

- Logistic regression

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \left\{ y^{(i)} \mathbf{x}'^{(i)}\beta - log(1 + e^{\mathbf{x}'\beta}) \right\}$$

$$\nabla_{\beta}\mathscr{L}(\beta) = -\mathbf{X}'^{T}(\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

L1, L2 regression loss, gradient functions

- L2 (Ridge); 0-center x_i then β_0 = mean($\mathbf{y}$), find $\beta_{1..p}$ via:

$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}\beta) \cdot (\mathbf{y} - \mathbf{X}\beta) + \lambda\beta \cdot \beta$$

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + 2\lambda\beta$$

- L1 (Lasso); 0-center x_i then β_0 = mean($\mathbf{y}$), find $\beta_{1..p}$ via:

$$\mathscr{L}(\beta) = (\mathbf{y} - \mathbf{X}\beta) \cdot (\mathbf{y} - \mathbf{X}\beta) + \lambda \sum_{j=1}^{p} |\beta_j|$$

$$\nabla_\beta \mathscr{L}(\beta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\,\mathrm{sign}(\beta)$$

L1 logistic loss, gradient functions

- Must compute β_0 differently; partial β_0 is a function of β_0

$$\frac{\partial}{\partial \beta_0} \mathscr{L}(\beta, \lambda) = mean(\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta))$$

- Other β_i are functions of β_0 but not within the penalty term

$$\nabla_{\beta_{1..p}} \mathscr{L}(\beta, \lambda) = \frac{1}{n} \left\{ \mathbf{X}^T (\mathbf{y} - \sigma(\mathbf{X}' \cdot \beta')) - \lambda \, \mathrm{sign}(\beta) \right\}$$

- Combine to get full gradient vector

Key takeaways

- Move towards lower loss; consider each direction separately
- Slope in direction β_i is partial derivative: $\frac{\partial}{\partial \beta_i}\mathscr{L}(\beta)$
- Gradient is p or p+1 dimensional vector of partial derivatives
- Gradients point "upwards" towards higher cost/loss
- Coefficients should therefore move in opposite direction of gradient
- Gradient is the 0 vector at the minimum loss; i.e., flat
- Can stop optimizing when gradient is close to 0 vector or
  when $\beta_i^{(t+1)}$ is very close to $\beta_i^{(t)}$ or
  after fixed number of iterations

More key takeaways

- Coefficient update equation:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \nabla_{\beta} \mathscr{L}(\beta^{(t)})$$ where η is a learning rate

- If η is "small enough," $\beta_i^{(t+1)}$ will converge to a solution vector (maybe slowly)
- If too big, will bounce back and forth across valleys or diverge
- Adagrad
  - Single learning rate too slow; need a rate per dimension
  - Increases update step size for dimensions with shallow slopes historically
  - Slows down across all dimensions over time as history sum *h* gets bigger

$$\vec{b} = \vec{b} - \eta * \frac{\nabla \mathscr{L}}{(\sqrt{\vec{h}} + \epsilon)}$$

- L1, L2 linear regression doesn't optimize β_0, it's just mean(**y**), if we 0-center x_i
- L1, L2 logistic regression optimizes $\beta_{0..p}$ but β_0 differently than $\beta_{1..p}$