Gradient Descent

Minimizing loss functions to find "optimal" model parameters

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Minimizing the loss function: How we train (many) models

- We need a way to find β such that: $\underset{\beta}{\arg\min}\, \mathscr{L}(\beta)$

- Could try grid search for linear models to find slope, y-intercept:

```
for m in np.linspace(…,…,num=100):
    for b in np.linspace(…,…,num=100):
        y_ = m * X + b
        loss = np.mean((y_ - y)**2) # MSE
        if loss < best[0]:
            best = (loss,m,b)
```

- Or, could try random β vectors and choose the β with lowest loss

Minimizing the loss: using loss information

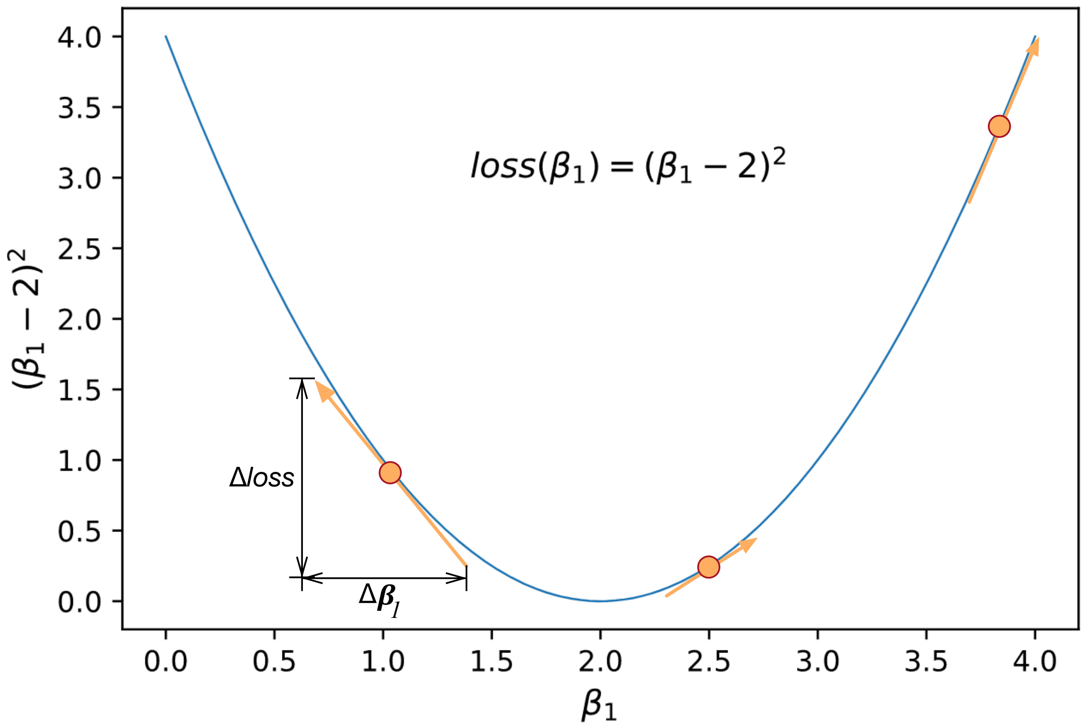
- Let's choose a random β and then tweak β with some $\Delta\beta$ in the downhill loss direction until any tweak would increase loss

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

- We use information about the loss function in the neighborhood of current β to decide which direction shifts towards smaller loss
- When loss goes up or doesn't change, we're done

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 β_1 in direction of increased loss (up)
- What is derivative of loss at β_1=1? β_1=3? β_1=2?
- Direction of lower loss is opposite/negative of derivative
- Derivative also has magnitude, which is bigger when slope is steeper
- **How to move**: $\beta_1 = \beta_1 - slope$

UNIVERSITY OF SAN FRANCISCO

Taking steps in right direction (1D β case)

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

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

```
b = random value
while not_converged:
    b = b - rate * gradient(b)
```

- β always converges on min loss if learning rate is small enough

Python gradient descent implementation

- First define a simple 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 pick a learning rate

```
b = np.random.uniform(0,4)
rate = .2
```

- Loop for a while or until L2 norm of gradient(b) == 0

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

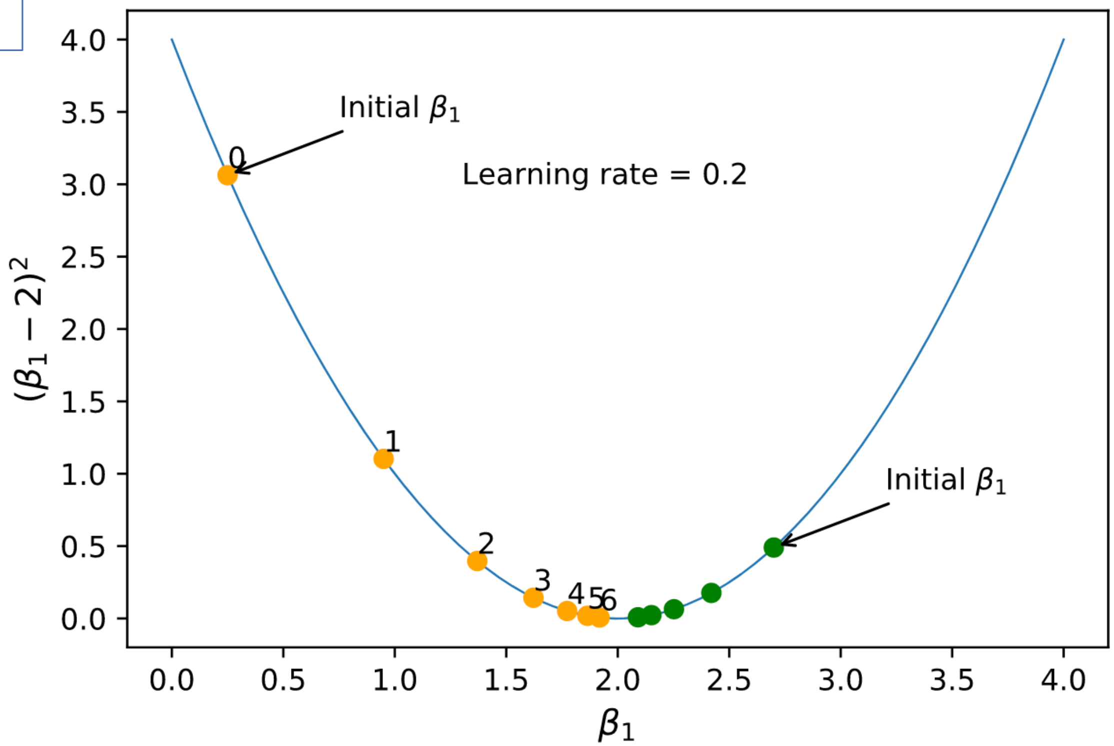
See https://github.com/parrt/msds621/blob/master/notebooks/linear-models/viz-gradient-descent.ipynb

Sample 1D gradient descent run

```
for t in range(7):
    b = b - 0.2 * gradient(b)
```

	beta_1	loss
0	0.055312	3.781813
1	0.833187	1.361453
2	1.299912	0.490123
3	1.579947	0.176444
4	1.747968	0.063520
5	1.848781	0.022867
6	1.909269	0.008232
7	1.945561	0.002964

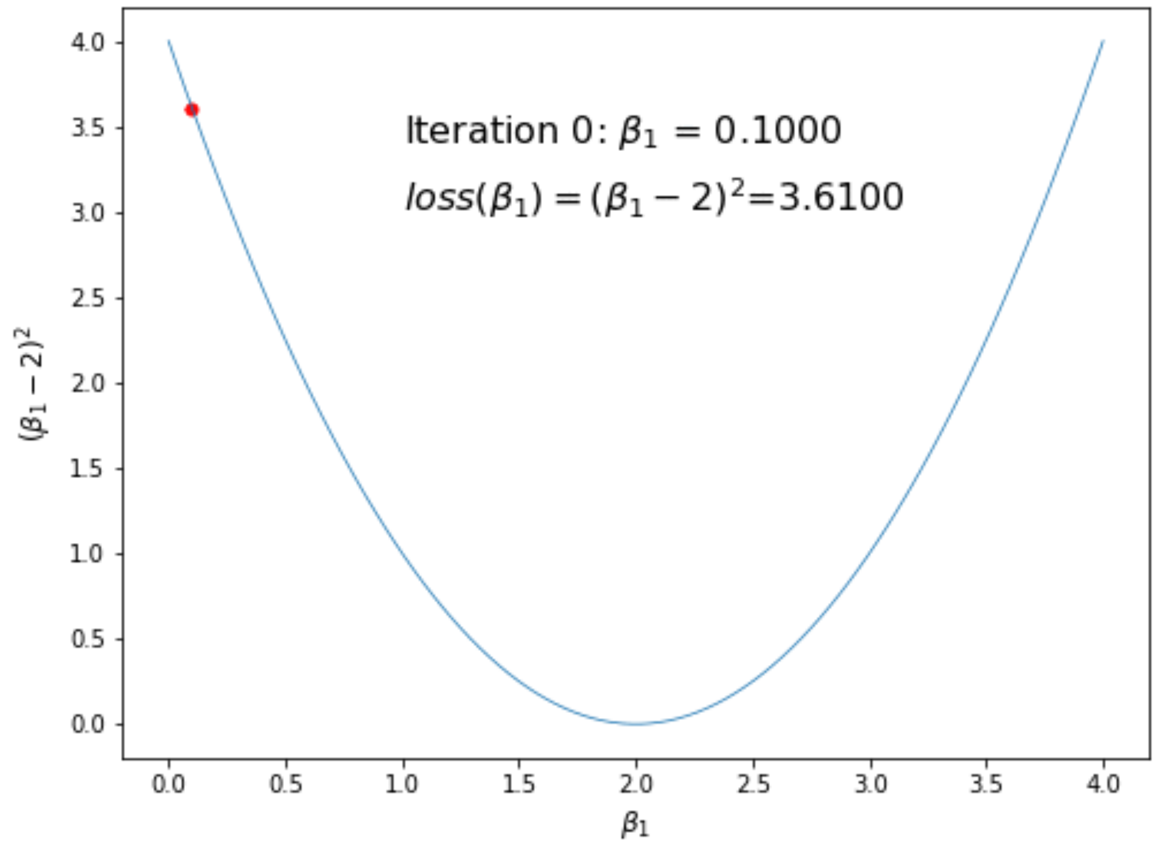
Notice β_1 accelerates and then slows down. Why?



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

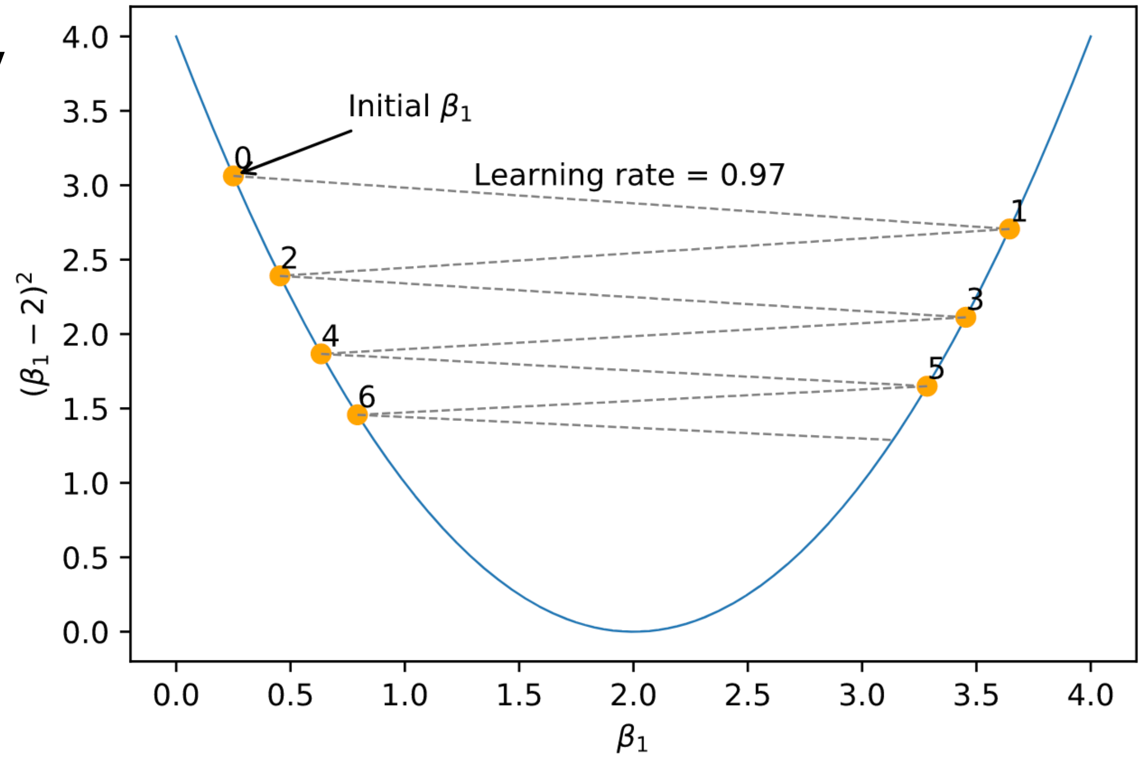
See https://github.com/parrt/msds621/blob/master/notebooks/linear-models/viz-gradient-descent.ipynb

1D function minimization in action



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 much progress towards min loss point



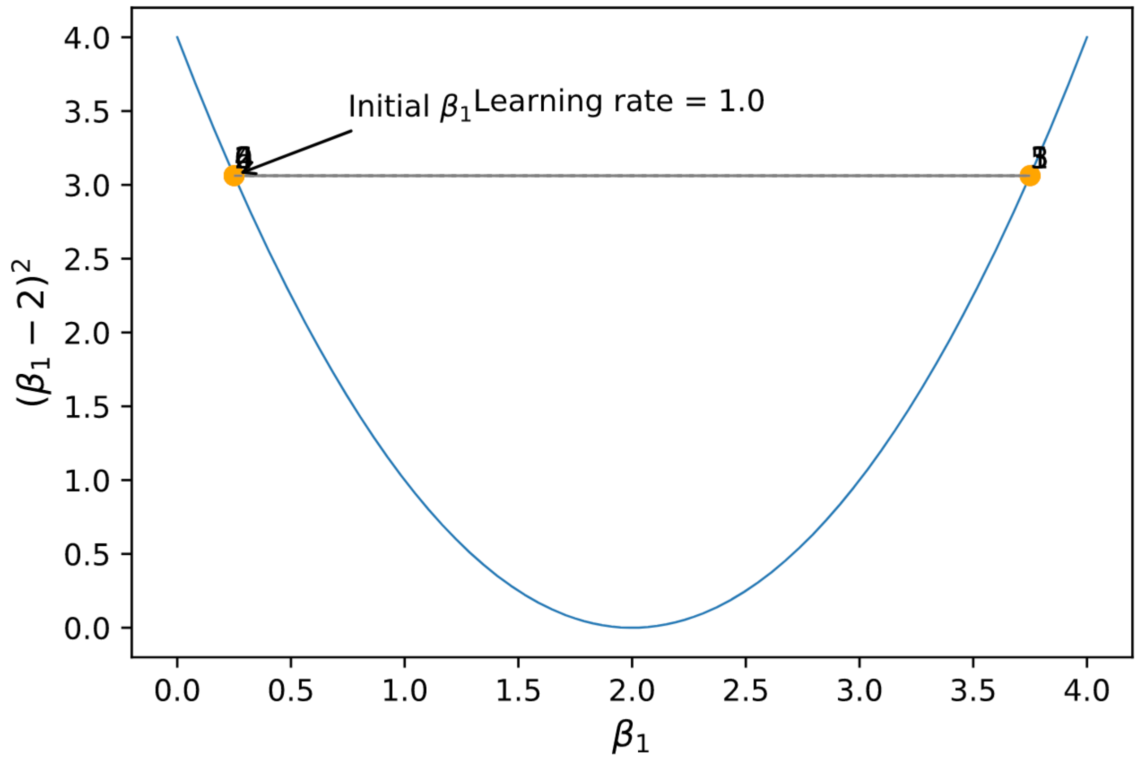
UNIVERSITY OF SAN FRANCISCO

What if learning rate is really too high?

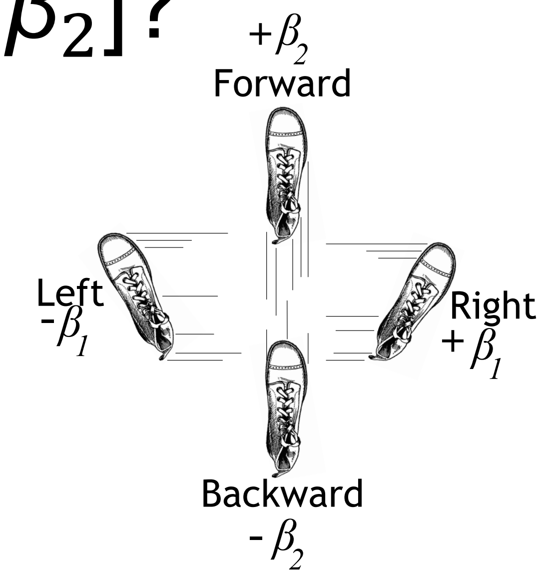
- We get nowhere:

	beta_1	loss
0	0.495633	2.263119
1	3.504367	2.263119
2	0.495633	2.263119
3	3.504367	2.263119
4	0.495633	2.263119

- It can even diverge, exploding β_1

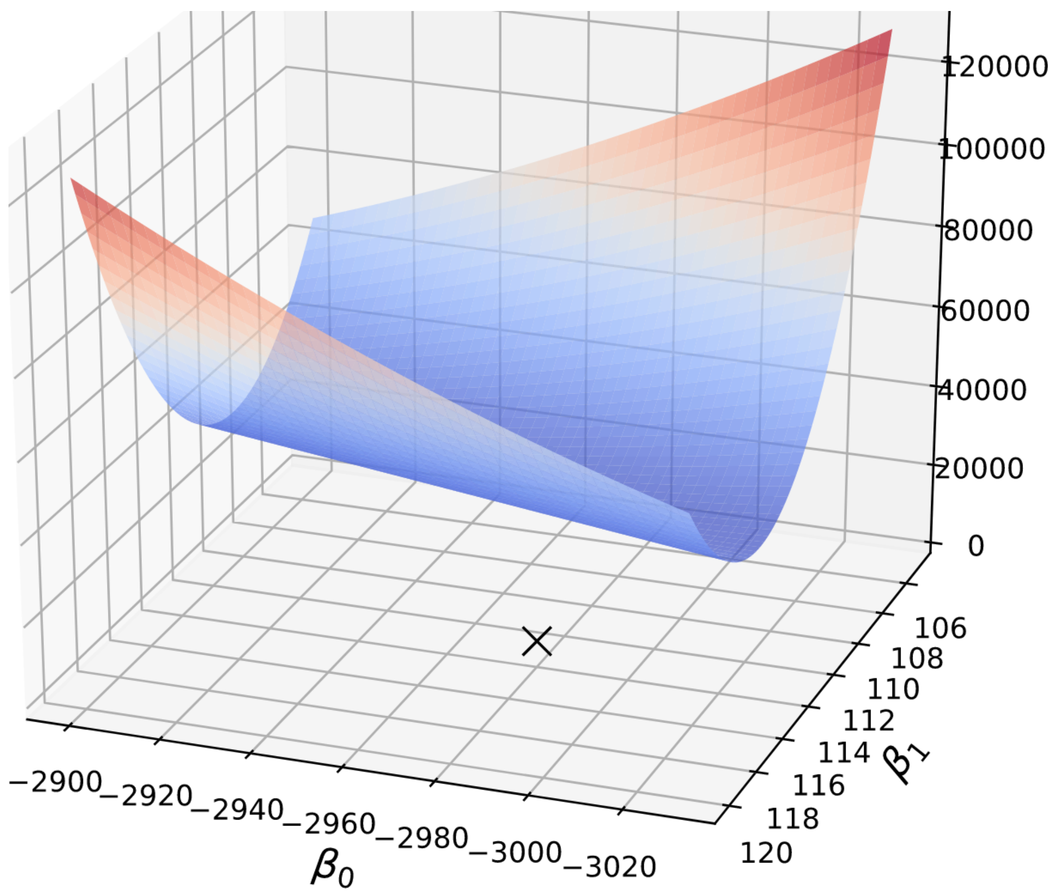


What happens in 2D for $\beta = [\beta_1, \beta_2]$?



- 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 into vector to obtain the best step direction
- Each direction's slope is a *partial derivative* and, combined, are the *gradient* vector

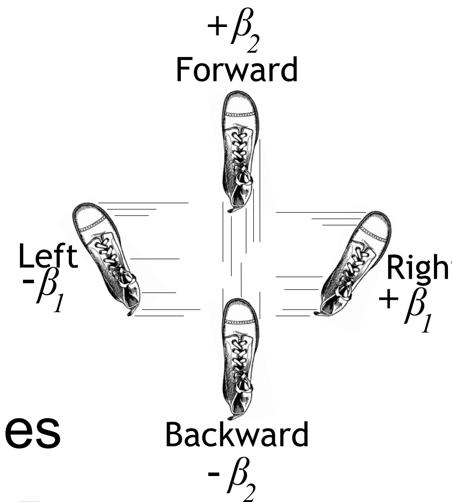
Loss function: 1-var regr. w/2 coeff (β_0, β_1)



- Shallow in β_0 dir
- Steep in β_1 dir
- This plot show loss for non-standardized variables so a unit change in β_0 doesn't change loss nearly as much for β_1

UNIVERSITY OF SAN FRANCISCO

Notation and finite difference approximation



- "Rise over run" is the derivative/slope of $f(x)$ at x:

$$\frac{d}{dx}f(x) = \frac{\partial}{\partial x}f(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Gradient of p-dim $\boldsymbol{x}$ vector has p partial derivative entries

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{x_1}f(\mathbf{x}) \\ \frac{\partial}{x_2}f(\mathbf{x}) \end{bmatrix} \approx \begin{bmatrix} \frac{f([x_1+h, x_2]) - f(\mathbf{x})}{h} \\ \frac{f([x_1, x_2+h]) - f(\mathbf{x})}{h} \end{bmatrix}$$

- The partial derivative is just the slope in 1 dir, holding others constant

See https://explained.ai/matrix-calculus/index.html

General gradient descent

- Partial derivative is rate of change in one direction: $\frac{\partial}{\partial \beta_i} \mathscr{L}(\beta)$
- Combining 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

- Gradientvectors have magnitude and direction
- E.g., gradient of [-1,2] means take step to left, but bigger step forward
- Take that single step: $\beta = \beta - \eta$ * [-1, 2]
- In each direction, the partial derivative of loss function is 0 when flat
- When norm of gradient vector = 0, we're at min loss; choose that β

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 η*):

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

η scales the step we take each at each step (fold 2 into *η)*

Simplest gradient descent algorithm

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

Let $\vec{\beta} \sim 2N(0,1) - 1$ $\quad$ *(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 except for L1/L2 regression)*

repeat

$\quad \vec{\beta} = \vec{\beta} - \boxed{\eta\nabla\mathscr{L}(\vec{\beta})}$ new direction $\quad$ (*Recall* $\nabla_{\beta}\mathscr{L}(\beta) = -2\mathbf{X}'^{T}(\mathbf{y} - \mathbf{X}'\beta)$)

until $\|\nabla\mathscr{L}(\vec{\beta})\|_2 < precision$;

return $\vec{\beta}$

Adding momentum to particle update

- Reinforce movement in same direction; add fraction of previous dir

Algorithm: *momentum_minimize*$(\mathbf{X}, \mathbf{y}, \nabla\mathscr{L}, \eta, \gamma)$ **returns** coefficients $\vec{\beta}$

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

$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$ $\quad$ (*Add first column of 1s except for L1/L2 regression*)

repeat

$\quad \vec{v} = \underbrace{\boxed{\gamma\vec{v}}}_{\text{old direction}} + \underbrace{\boxed{\eta\nabla\mathscr{L}(\vec{\beta})}}_{\text{new direction}}$ $\quad$ (*Add a bit of previous direction to next direction*)

$\quad \vec{\beta} = \vec{\beta} - \vec{v}$

until $\|\nabla\mathscr{L}(\vec{\beta})\|_2 < precision$;

return $\vec{\beta}$

γ is a new hyper parameter

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
- $\vec{h}$ grows and eventually slows down learning, possibly too early

Algorithm: $\mathit{adagrad\_minimize}(\mathbf{X}, \mathbf{y}, \nabla\mathscr{L}, \eta, \epsilon\text{=1e-5})$ **returns** coefficients $\vec{\beta}$

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

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

$\mathbf{X}' = (\vec{\mathbf{1}}, \mathbf{X})$ $\quad$ (*Add first column of 1s except for L1/L2 regression*)

repeat

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

$\quad \vec{\beta} = \vec{\beta} - \eta * \boxed{\dfrac{\nabla\mathscr{L}(\vec{\beta})}{(\sqrt{\vec{h}}+\epsilon)}}$ ← adjust w/update per β_i; low h(istory) for β_i increases its learning rate
(ϵ avoids divide by 0)

until $\|\nabla\mathscr{L}(\vec{\beta})\|_2 < \mathit{precision}$;

return $\vec{\beta}$

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

Loss, gradient functions for minimization

- Linear regression

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

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

(Can drop the 2, fold into learning rate)

- 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(**y**), find $\beta_{1..p}$ via:

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

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

(Can drop the 2, fold into learning rate)

- L1 (Lasso); 0-center x_i then β_0 = mean(**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

L1 Logistic gradient is tricky to get right

(See derivation of L1 gradients in appendix of project description)

Algorithm: $L1NegLogLikelihood(\mathbf{X}', \mathbf{y}, \beta')$

$err = \mathbf{y} - \sigma(\mathbf{X}' \cdot \beta')$ $\quad$ $(error\ vector$ is $n \times 1\ column\ vector)$

$\frac{\partial}{\partial \beta_0} = mean(err)$ $\quad$ $(usual\ log\text{-}likelihood\ gradient;\ use\ current\ \beta')$

$r = \lambda\,\text{sign}(\beta')$ $\quad$ $(regularization\ term\ p+1 \times 1\ column\ vector)$

$r[0] = 0$ $\quad$ $(kill\ \beta_0\ position\ but\ keep\ as\ p+1 \times 1\ vector)$

$\nabla = \frac{1}{n}\left\{\mathbf{X}'^{T} err - r\right\}$

$$\textbf{return} - \begin{bmatrix} \frac{\partial}{\partial \beta_0} \\ \nabla_1 \\ \vdots \\ \nabla_p \end{bmatrix}$$

gradients

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

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

Key takeaways

- Move β towards lower loss; consider each β_i direction separately
- Slope (change in loss/β_i) 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 β should therefore step by negative of gradient
- Gradient is the 0 vector at the minimum loss; i.e., flat
- Can stop optimizing when gradient norm is close to 0 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," $\beta^{(t+1)}$ will converge to a solution vector (maybe slowly)
- If too big, will bounce back and forth across valleys or diverge

- Adagrad

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

  - 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
- 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 optimizes β_0 differently than $\beta_{1..p}$

Lab time

- Exploring regularization for linear regression
  https://github.com/parrt/msds621/tree/master/labs/linear-models/gradient-descent.ipynb