Applied Machine Learning

Gradient Descent Methods

Siamak Ravanbakhsh

COMP 551 (winter 2020)

Learning objectives

Basic idea of

- gradient descent
- stochastic gradient descent
- method of momentum
- using adaptive learning rate
- sub-gradient

Application to

- linear regression and classification

Optimization in ML

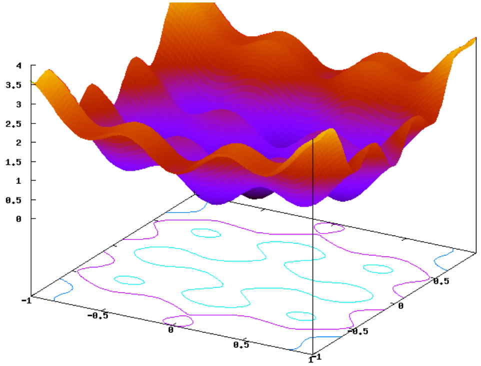
Inference and learning of a model often involves optimization:

optimization is a huge field

Optimization in ML

Inference and learning of a model often involves optimization: optimization is a huge field

bold: the setting considered in this class



- discrete (combinatorial) vs **continuous variables**
- constrained vs **unconstrained**
- for continuous optimization in ML:
  - **convex** vs **non-convex**
  - looking for **local** vs global optima?
  - **analytic gradient**?
  - analytic Hessian?
  - **stochastic** vs **batch**
  - **smooth** vs non-smooth

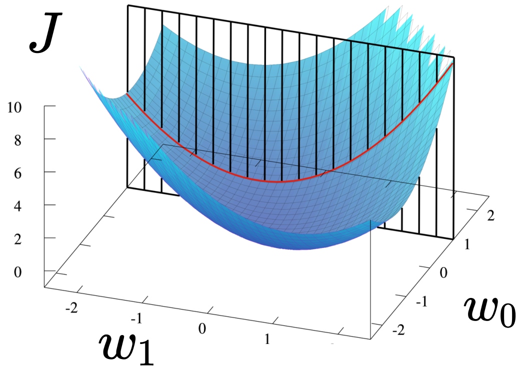
Gradient

for a multivariate function $J(w_0, w_1)$

partial derivatives instead of derivative

= derivative when other vars. are fixed

$$\frac{\partial}{\partial w_1} J(w_0, w_1) \triangleq \lim_{\epsilon \to 0} \frac{J(w_0, w_1 + \epsilon) - J(w_0, w_1)}{\epsilon}$$



we can estimate this numerically if needed

(use small epsilon in the the formula above)

Gradient

for a multivariate function $J(w_0, w_1)$

partial derivatives instead of derivative

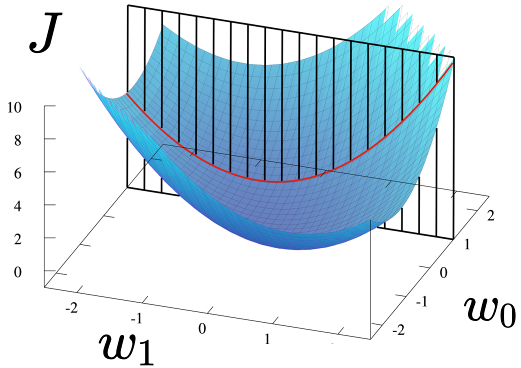
= derivative when other vars. are fixed

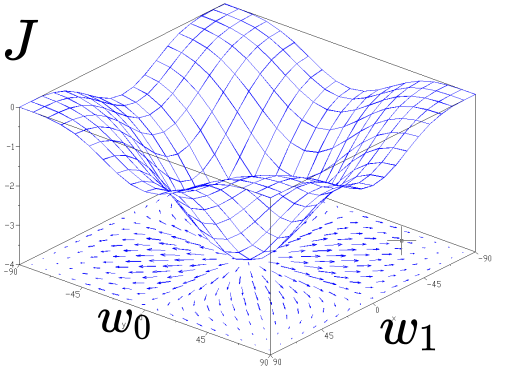
$$\frac{\partial}{\partial w_1} J(w_0, w_1) \triangleq \lim_{\epsilon \to 0} \frac{J(w_0, w_1 + \epsilon) - J(w_0, w_1)}{\epsilon}$$

we can estimate this numerically if needed
(use small epsilon in the the formula above)

gradient: vector of all partial derivatives

$$\nabla J(w) = [\frac{\partial}{\partial w_1} J(w), \cdots \frac{\partial}{\partial w_D} J(w)]^T$$





Gradient descent

an iterative algorithm for optimization

- starts from some $w^{\{0\}}$
- update using gradient $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla \mathcal{J}(w^{\{t\}})$

steepest descent direction

converges to a local minima

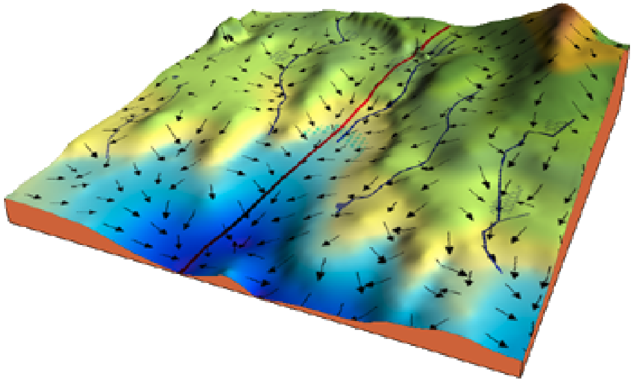


image: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Gradient descent

an iterative algorithm for optimization

- starts from some $w^{\{0\}}$
- update using gradient $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla \mathcal{J}(w^{\{t\}})$

steepest descent direction

learning rate

converges to a local minima

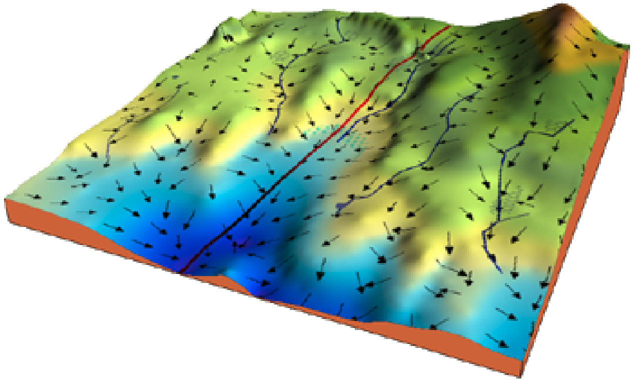


image: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Gradient descent

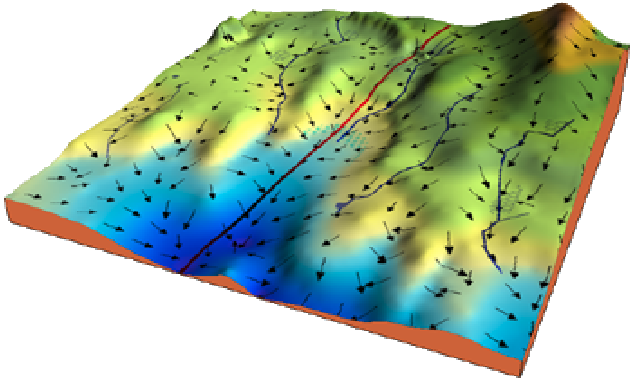
an iterative algorithm for optimization

- starts from some $w^{\{0\}}$
- update using gradient $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla \mathcal{J}(w^{\{t\}})$

steepest descent direction

learning rate

converges to a local minima



$$\nabla \mathcal{J}(w) = [\frac{\partial}{\partial w_1}\mathcal{J}(w), \cdots \frac{\partial}{\partial w_D}\mathcal{J}(w)]^T$$

image: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Gradient descent

an iterative algorithm for optimization

- starts from some $w^{\{0\}}$
- update using gradient $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla \mathcal{J}(w^{\{t\}})$

steepest descent direction

learning rate

cost function

(for maximization : objective function)

$$\nabla \mathcal{J}(w) = [\frac{\partial}{\partial w_1} \mathcal{J}(w), \cdots \frac{\partial}{\partial w_D} \mathcal{J}(w)]^T$$

converges to a local minima

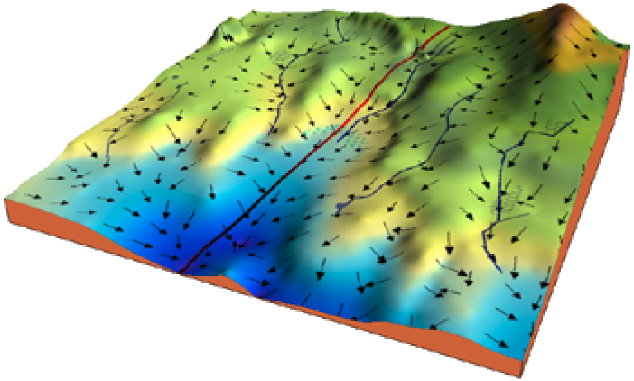
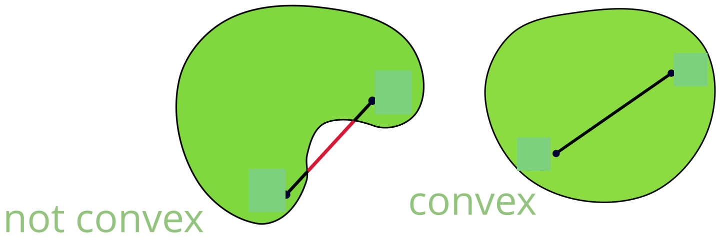


image: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

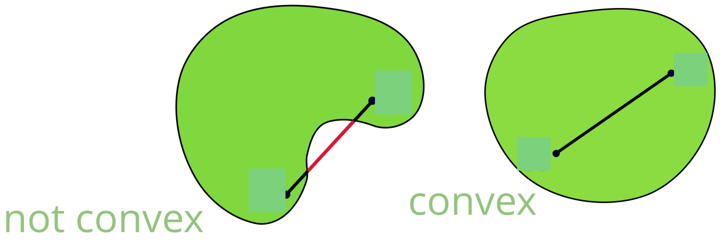
Convex function

a **convex** subset of $\mathbb{R}^N$ intersects any line in at most one line segment

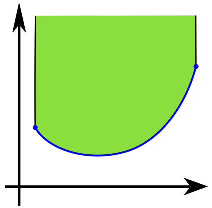


Convex function

a **convex** subset of $\mathbb{R}^N$ intersects any line in at most one line segment



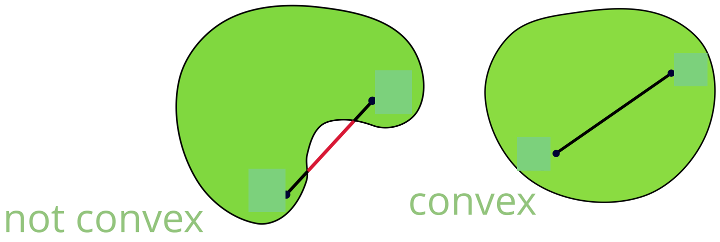
a **convex function** is a function for which the *epigraph* is a **convex set**



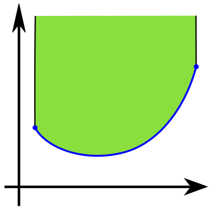
epigraph: set of all points above the graph

Convex function

a **convex** subset of $\mathbb{R}^N$ intersects any line in at most one line segment

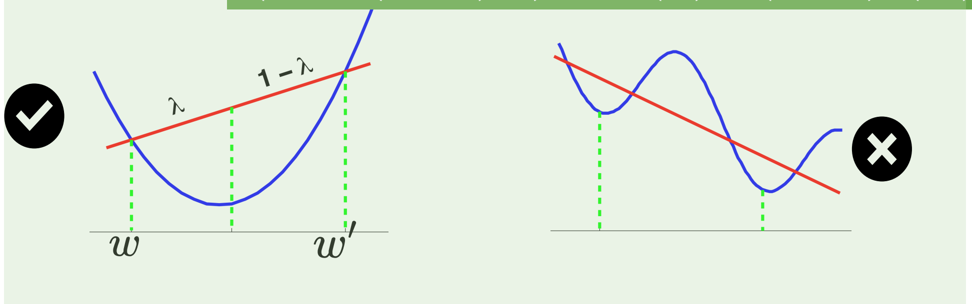


a **convex function** is a function for which the *epigraph* is a **convex set**



epigraph: set of all points above the graph

$$f(\lambda w + (1-\lambda)w') \leq \lambda f(w) + (1-\lambda) f(w') \quad 0 < \lambda < 1$$

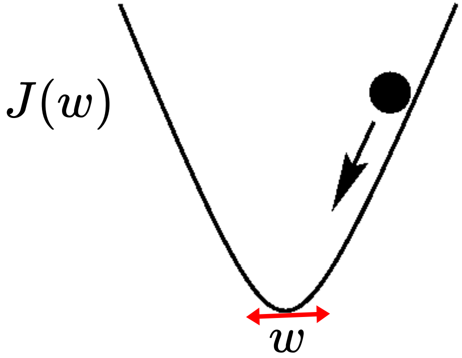


Convex function

Convex functions are easier to minimize:

- critical points are global minimum
- gradient descent can find it $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla \mathcal{J}(w^{\{t\}})$

convex



non-convex: gradient descent may find a local optima

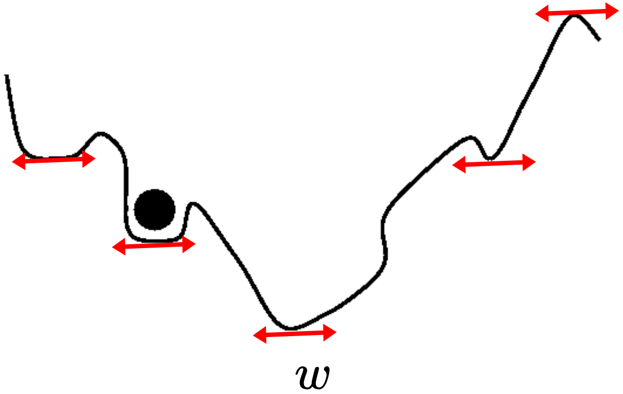


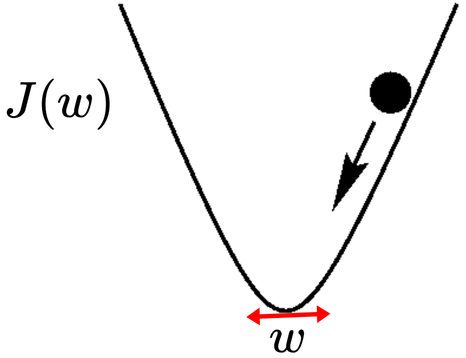
image: https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Convex function

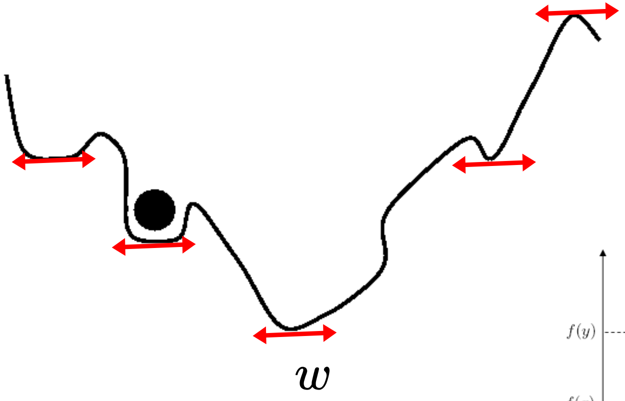
Convex functions are easier to minimize:

- critical points are global minimum
- gradient descent can find it $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla \mathcal{J}(w^{\{t\}})$

convex



non-convex: gradient descent may find a local optima



a **concave** function is a negative of a convex function (easy to **maximize**)

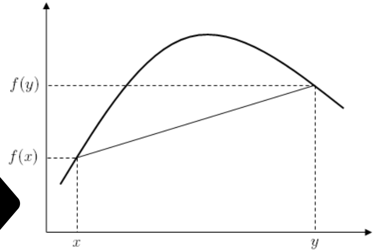


image: https://www.willamette.edu/~gorr/classes/cs449/momrate.html

Recognizing convex functions

a linear function is convex $w^T x$

Recognizing convex functions

a linear function is convex $w^T x$

convex if second derivative is positive everywhere $\frac{d^2}{x^2} f \geq 0$

example $x^{2d}, e^x, -\log(x), -\sqrt{x}$

Recognizing convex functions

a linear function is convex $w^T x$

convex if second derivative is positive everywhere $\frac{d^2}{x^2} f \geq 0$

example $x^{2d}, e^x, -\log(x), -\sqrt{x}$

sum of convex functions is convex

example $||WX - Y||_2^2 + \lambda ||w||_2^2$

Recognizing convex functions

a linear function is convex $w^T x$

convex if second derivative is positive everywhere $\frac{d^2}{x^2} f \geq 0$

example $x^{2d}, e^x, -\log(x), -\sqrt{x}$

sum of convex functions is convex

example $||WX - Y||_2^2 + \lambda ||w||_2^2$

maximum of convex functions is convex

example $f(y) = \max_{x \in [1,5]} \sqrt{x} y^4$

note this is not convex in x

Recognizing convex functions

a linear function is convex $w^T x$

convex if second derivative is positive everywhere $\frac{d^2}{x^2} f \geq 0$

example $x^{2d}, e^x, -\log(x), -\sqrt{x}$

sum of convex functions is convex

example $||WX - Y||_2^2 + \lambda ||w||_2^2$

maximum of convex functions is convex

example $f(y) = \max_{x \in [1,5]} \sqrt{x} y^4$

note this is not convex in x

composition of convex functions is generally **not** convex

example $(-\log(x))^2$

Recognizing convex functions

a linear function is convex $w^T x$

convex if second derivative is positive everywhere $\frac{d^2}{x^2} f \geq 0$

example $x^{2d}, e^x, -\log(x), -\sqrt{x}$

sum of convex functions is convex example $||WX - Y||_2^2 + \lambda ||w||_2^2$

maximum of convex functions is convex example $f(y) = \max_{x \in [1,5]} \sqrt{x} y^4$

note this is not convex in x

composition of convex functions is generally **not** convex example $(-\log(x))^2$

however, if ***f*,*g*** are convex, and ***g*** is non-decreasing $g(f(x))$ is convex

example $e^{f(x)}$

for convex ***f***

Gradient for linear and logistic regression

in both cases: $\nabla J(w) = \overset{D \times N}{X^T} \overset{N \times 1}{(\hat{y} - y)}$

linear regression: $\hat{y} = \overset{N \times D}{X} \overset{D \times 1}{w}$

logistic regression: $\hat{y} = \sigma(Xw)$

Gradient for linear and logistic regression

in both cases: $\nabla J(w) = \overset{D \times N}{X^T} \overset{N \times 1}{(\hat{y} - y)}$

linear regression: $\hat{y} = \overset{N \times D}{X} \overset{D \times 1}{w}$

logistic regression: $\hat{y} = \sigma(Xw)$

time complexity: $\mathcal{O}(ND)$

(two matrix multiplications)
compared to the direct solution for linear regression: $\mathcal{O}(ND^2 + D^3)$
gradient descent can be much faster for large D

Gradient for linear and logistic regression

in both cases: $\nabla J(w) = \overset{D \times N}{X^T} \overset{N \times 1}{(\hat{y} - y)}$

linear regression: $\hat{y} = \overset{N \times D}{X} \overset{D \times 1}{w}$

logistic regression: $\hat{y} = \sigma(Xw)$

```python
def gradient(X, y, w):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    return grad
```

time complexity: $\mathcal{O}(ND)$

(two matrix multiplications)
compared to the direct solution for linear regression: $\mathcal{O}(ND^2 + D^3)$
gradient descent can be much faster for large D

Gradient Descent

implementing gradient descent is easy!

```python
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w

```

code on the previous page

Gradient Descent

implementing gradient descent is easy!

```
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w
```

code on the previous page

Some **termination conditions**:

- some max #iterations
- small gradient
- a small change in the objective
- increasing error on validation set

early stopping (one way to avoid overfitting)

Example: GD for Linear Regression

applying this to to fit toy data

```python
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w

```

```python
def gradient(X, y, w):
    N,D = X.shape
    yh = np.dot(X, w)
    grad = np.dot(X.T, yh - y) / N
    return grad
```

Example: GD for Linear Regression

applying this to to fit toy data ←

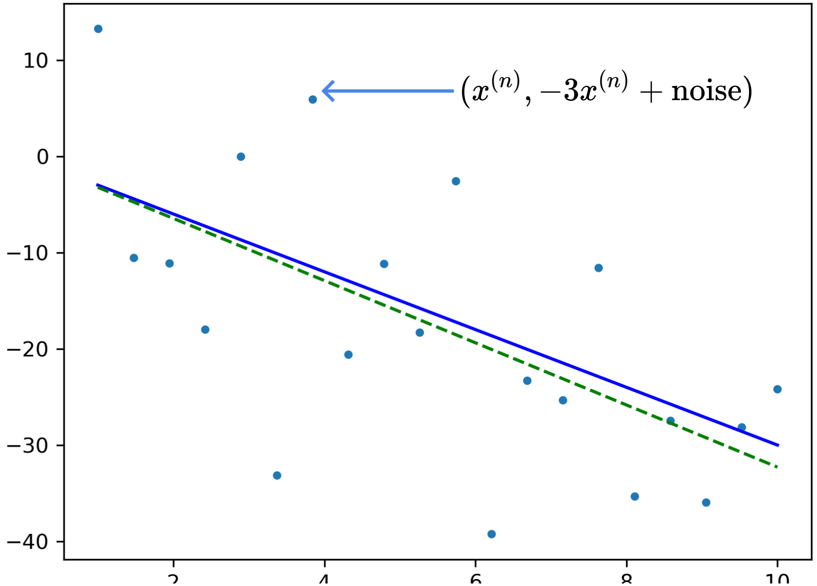
single feature (intercept is zero)

```
#D = 1
N = 20
X = np.linspace(1,10, N)[:,None]
y_truth = np.dot(x, np.array([-3.]))
y = y_truth + 10*np.random.randn(N)
```

Example: GD for Linear Regression

applying this to to fit toy data ←

single feature (intercept is zero)



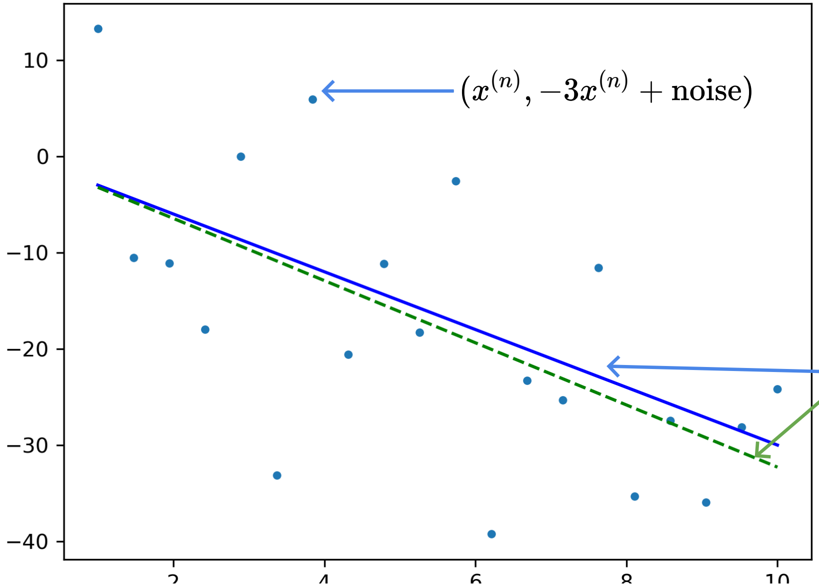

```
#D = 1
N = 20
X = np.linspace(1,10, N)[:,None]
y_truth = np.dot(x, np.array([-3.]))
y = y_truth + 10*np.random.randn(N)
```

Example: GD for Linear Regression

applying this to to fit toy data

single feature (intercept is zero)

```
#D = 1
N = 20
X = np.linspace(1,10, N)[:,None]
y_truth = np.dot(x, np.array([-3.]))
y = y_truth + 10*np.random.randn(N)
```



$y = wx$

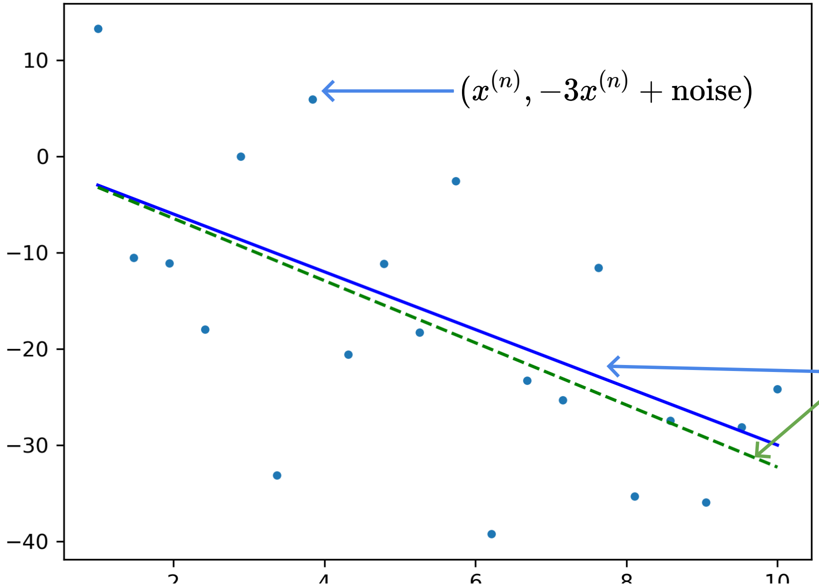
$y = -3x$

Example: GD for Linear Regression

applying this to to fit toy data ←

```
#D = 1
N = 20
X = np.linspace(1,10, N)[:,None]
y_truth = np.dot(x, np.array([-3.]))
y = y_truth + 10*np.random.randn(N)
```

single feature (intercept is zero)



using direct solution method

$$w = (X^T X)^{-1} X^T y \approx -3.2$$

$$y = wx$$

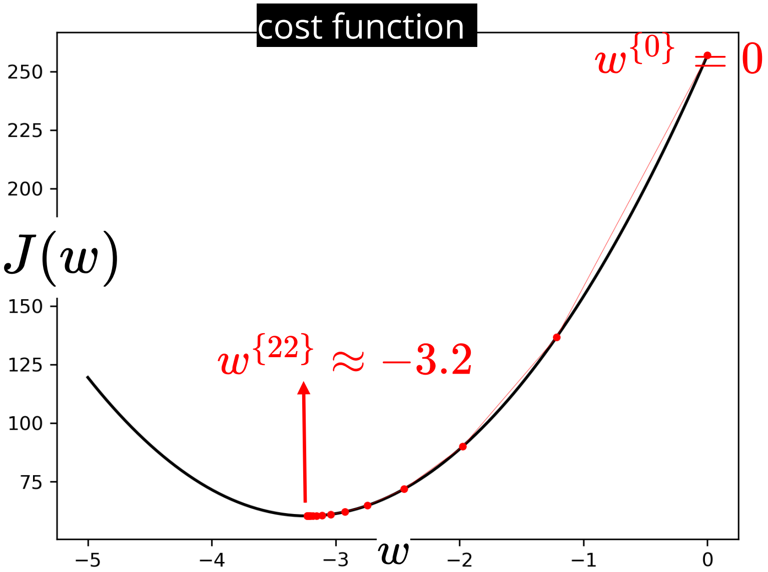
$$y = -3x$$

Example: GD for Linear Regression

After 22 iterations of Gradient Descent $w^{\{t+1\}} \leftarrow w^{\{t\}} - .01\nabla J(w^{\{t\}})$

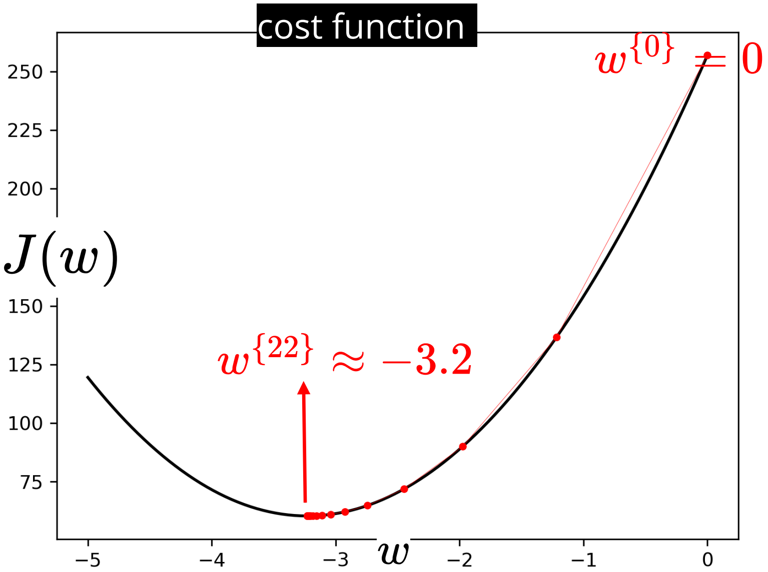
Example: GD for Linear Regression

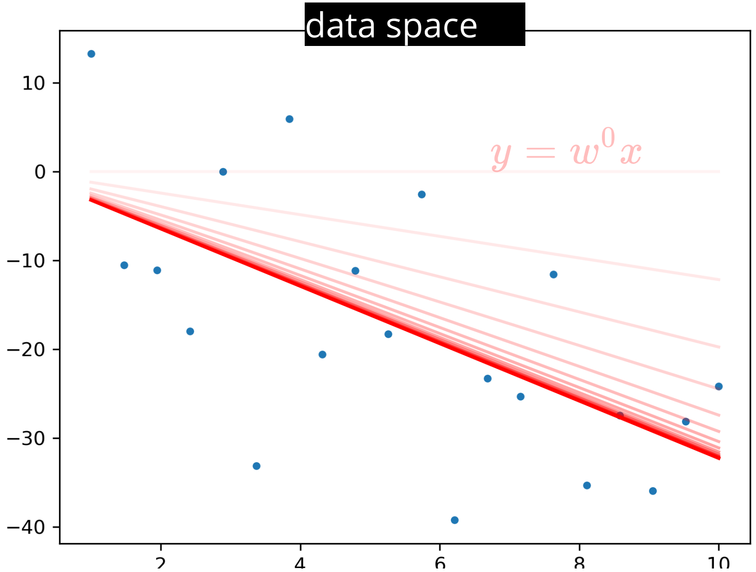
After 22 iterations of Gradient Descent $\quad w^{\{t+1\}} \leftarrow w^{\{t\}} - .01 \nabla J(w^{\{t\}})$



Example: GD for Linear Regression

After 22 iterations of Gradient Descent $w^{\{t+1\}} \leftarrow w^{\{t\}} - .01 \nabla J(w^{\{t\}})$



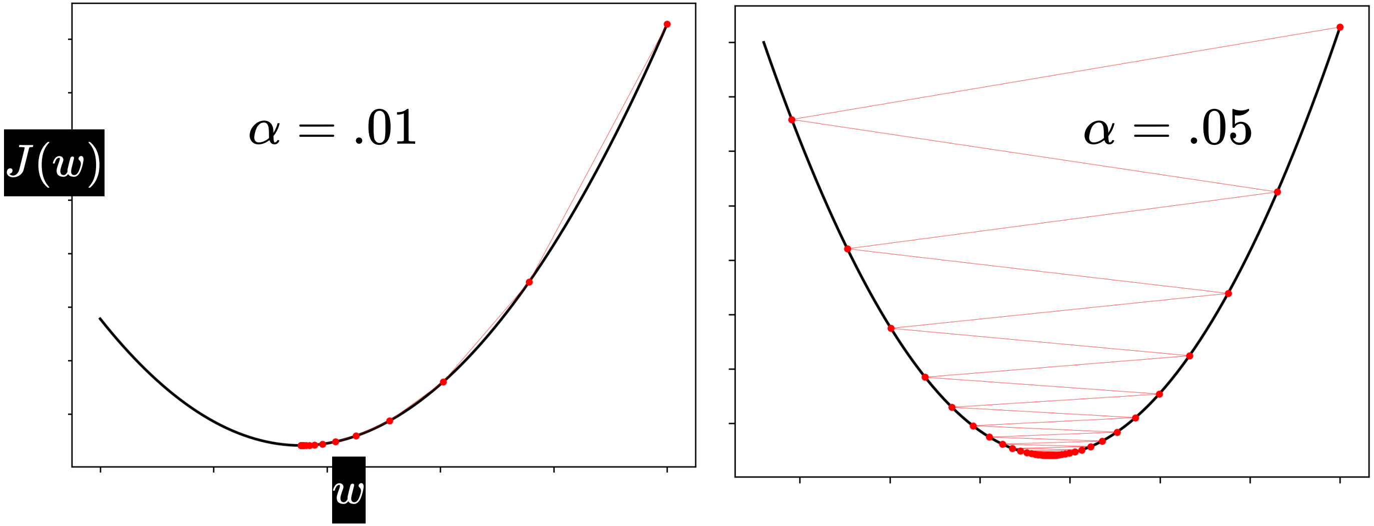


Learning rate α

Learning rate has a significant effect on GD

too small: may take a long time to converge

too large: it overshoots



GD for logistic Regression

example: *logistic regression for Iris dataset* (D=2, lr=.01)

```
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w

```

GD for logistic Regression

example: *logistic regression for Iris dataset* (D=2, lr=.01)

```
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w
```

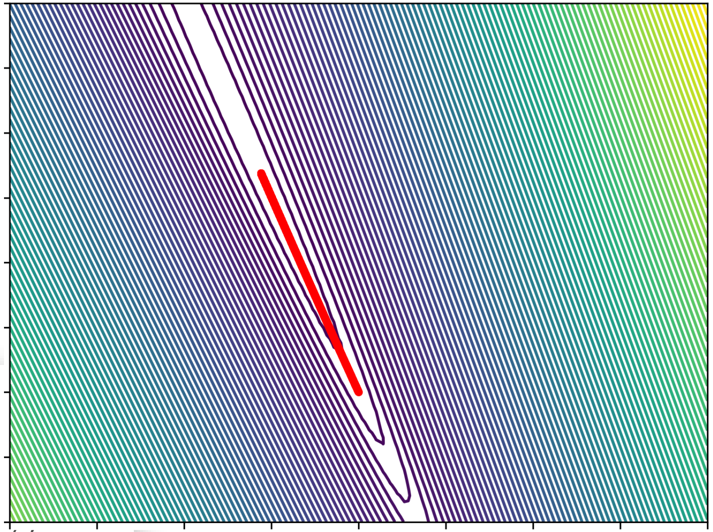
```
def gradient(X, y, w):
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y)
    return grad
```

GD for logistic Regression

example: *logistic regression for Iris dataset* (D=2, lr=.01)

```python
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w
```

```python
def gradient(X, y, w):
    yh = logistic(np.dot(X,
    grad = np.dot(X.T, yh -
    return grad
```



Stochastic Gradient Descent

we can write the cost function as a average over instances

$$J(w) = \frac{1}{N}\sum_{n=1}^{N} \boxed{J_n(w)}$$

cost for a single data-point

e.g. for linear regression $J_n(w) = \frac{1}{2}(w^T x^{(n)} - y^{(n)})^2$

Stochastic Gradient Descent

we can write the cost function as a average over instances

$$J(w) = \frac{1}{N} \sum_{n=1}^{N} J_n(w)$$

cost for a single data-point

e.g. for linear regression $J_n(w) = \frac{1}{2}(w^T x^{(n)} - y^{(n)})^2$

the same is true for the partial derivatives

$$\frac{\partial}{\partial w_j} J(w) = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial w_j} J_n(w)$$

Stochastic Gradient Descent

we can write the cost function as a average over instances

$$J(w) = \frac{1}{N} \sum_{n=1}^{N} J_n(w)$$

cost for a single data-point

e.g. for linear regression $J_n(w) = \frac{1}{2}(w^T x^{(n)} - y^{(n)})^2$

the same is true for the partial derivatives

$$\frac{\partial}{\partial w_j} J(w) = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial w_j} J_n(w)$$

therefore $\nabla J(w) = \mathbb{E}[\nabla J_n(w)]$

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

contour plot of the cost function + batch gradient update $w \leftarrow w - \alpha \nabla J(w)$

with small learning rate: guaranteed improvement at each step

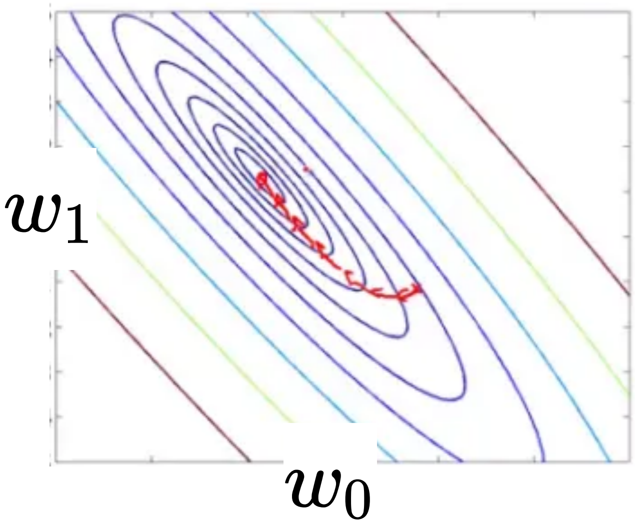


image:https://jaykanidan.wordpress.com

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

using stochastic gradient $w \leftarrow w - \alpha \nabla J_n(w)$

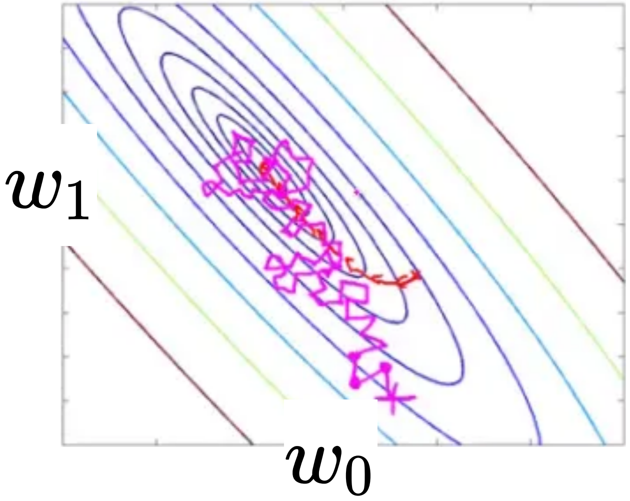


image:https://jaykanidan.wordpress.com

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

using stochastic gradient $w \leftarrow w - \alpha \nabla J_n(w)$

the steps are "on average" in the right direction

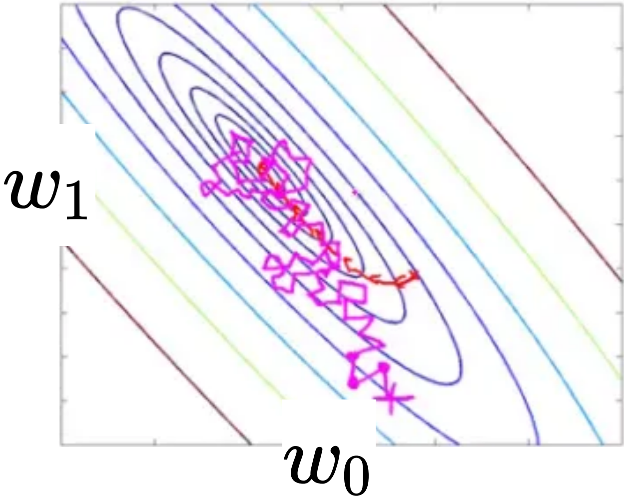


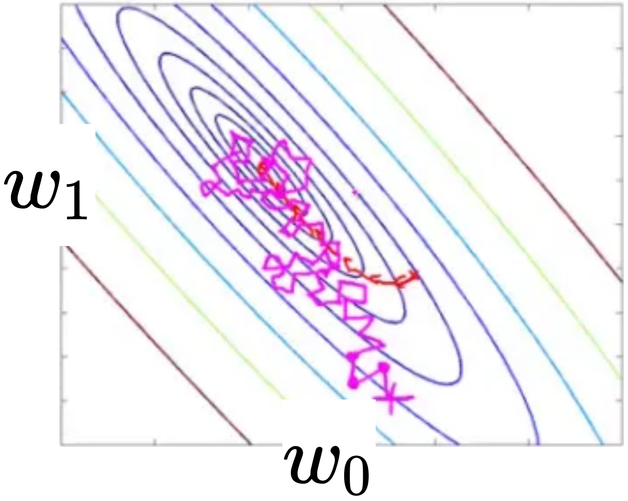
image:https://jaykanidan.wordpress.com

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

using stochastic gradient $w \leftarrow w - \alpha \nabla J_n(w)$

the steps are "on average" in the right direction



each step is using gradient of a different cost $J_n(w)$

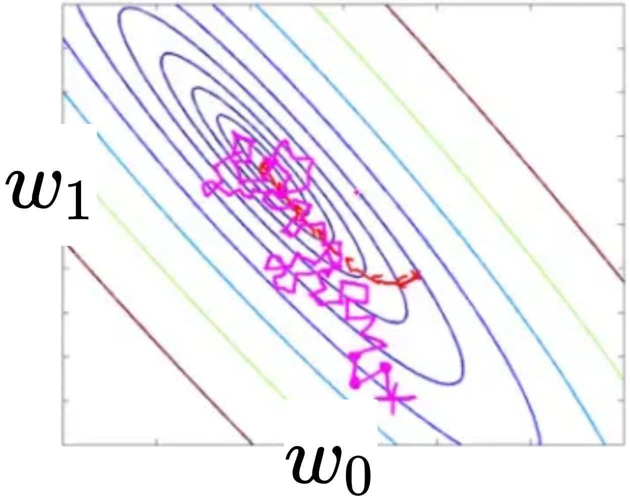
image:https://jaykanidan.wordpress.com

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

using stochastic gradient $w \leftarrow w - \alpha \nabla J_n(w)$

the steps are "on average" in the right direction



each step is using gradient of a different cost $J_n(w)$

each update is (1/N) of the cost of batch gradient

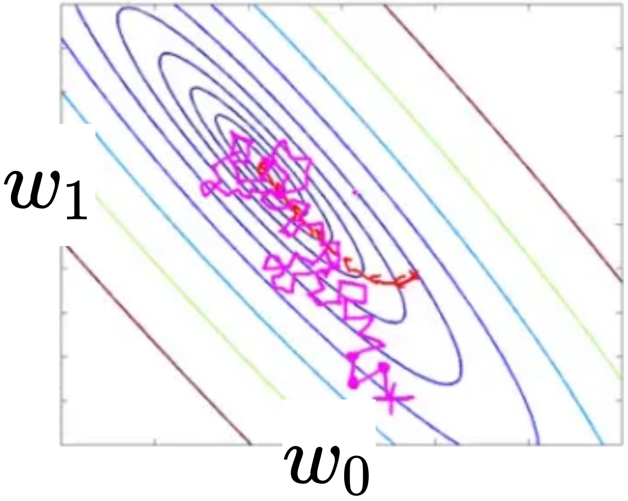
image:https://jaykanidan.wordpress.com

Stochastic Gradient Descent

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

using stochastic gradient $w \leftarrow w - \alpha \nabla J_n(w)$

the steps are "on average" in the right direction



each step is using gradient of a different cost $J_n(w)$

each update is (1/N) of the cost of batch gradient

e.g., for linear regression $\mathcal{O}(D)$

$$\nabla J_n(w) = x^{(n)}(w^T x^{(n)} - y^{(n)})$$

image:https://jaykanidan.wordpress.com

Example: SGD for logistic regression

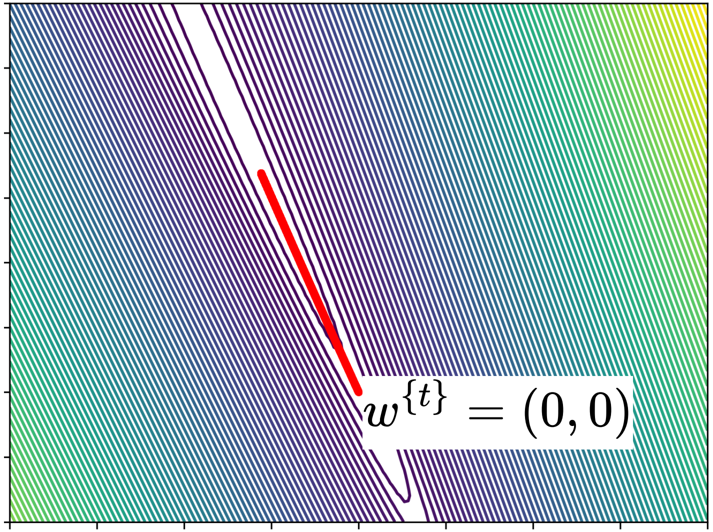
setting 1: using batch gradient

logistic regression for Iris dataset (D=2 , $\alpha = .1$)

```python
def GradientDescent(X, # N x D
                    y, # N
                    lr=.01, # learning rate
                    eps=1e-2, # termination codition
                   ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w

```

```python
def gradient(X, y, w):
    N, D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    return grad
```

after 8000 iterations

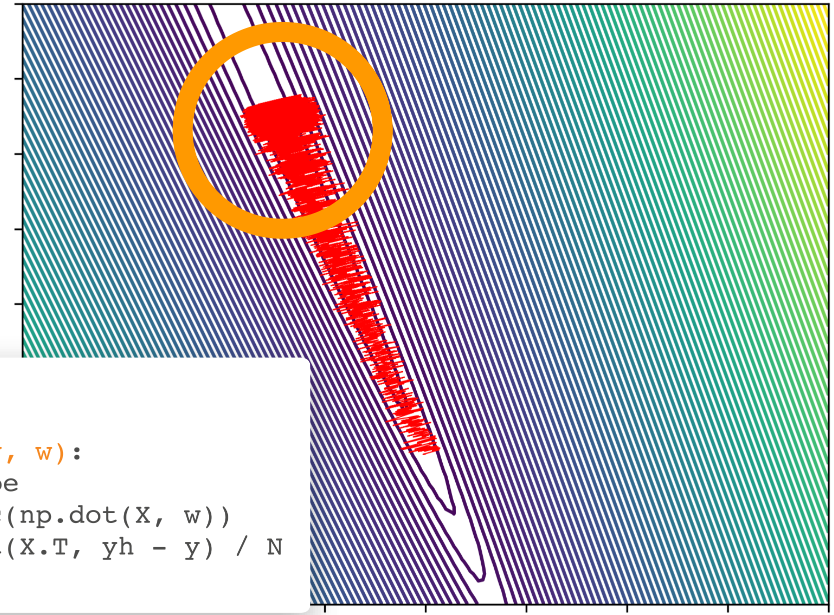


Example: SGD for logistic regression

setting 2: using stochastic gradient

logistic regression for Iris dataset (D=2, $\alpha = .1$)

```python
def StochasticGradientDescent(
                                        X, # N x D
                        y, # N
                        lr=.01, # learning rate
                        eps=1e-2, # termination codition
                       ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        n = np.random.randint(N)
        g = gradient(X[[n],:], y[[n]], w)
        w = w - lr*g
    return w

```

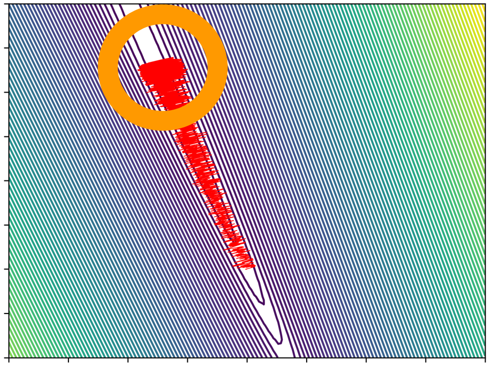


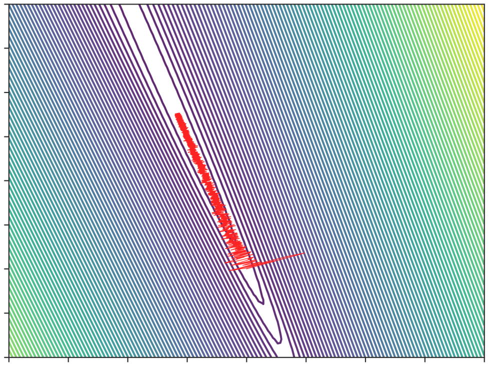
```python
def gradient(X, y, w):
    N, D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    return grad
```

Convergence of SGD

stochastic gradients are not zero at optimum

how to guarantee convergence?



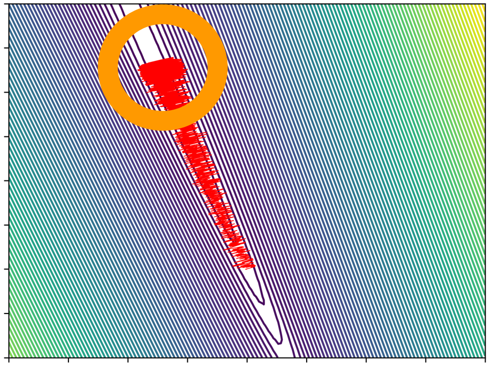


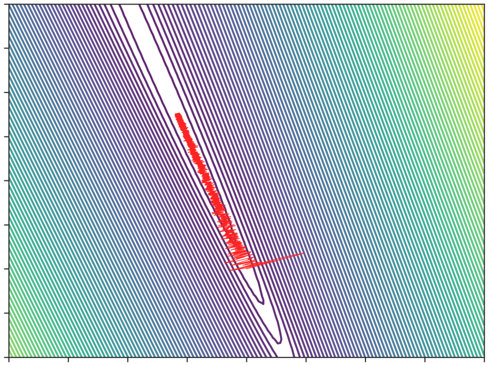
Convergence of SGD

stochastic gradients are not zero at optimum

how to guarantee convergence?

schedule to have a smaller learning rate over time



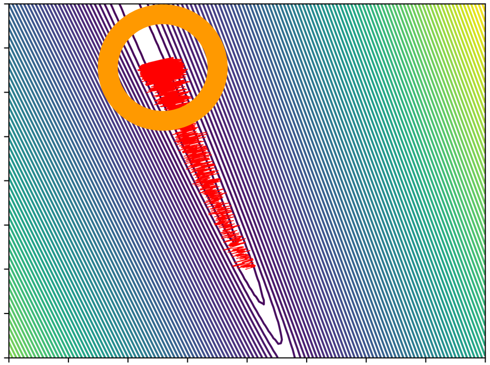


Convergence of SGD

stochastic gradients are not zero at optimum

how to guarantee convergence?

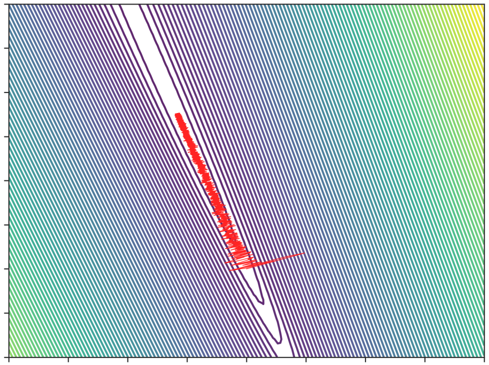
schedule to have a smaller learning rate over time



Robbins Monro

the sequence we use should satisfy: $\sum_{t=0}^{\infty} \alpha^{\{t\}} = \infty$

otherwise for large $||w^{\{0\}} - w^*||$ we can't reach the minimum

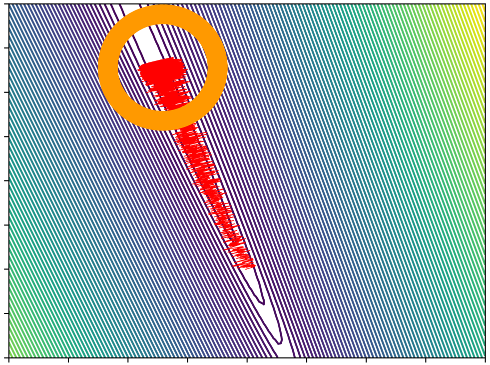


the steps should go to zero $\sum_{t=0}^{\infty} (\alpha^{\{t\}})^2 < \infty$

Convergence of SGD

stochastic gradients are not zero at optimum

how to guarantee convergence?



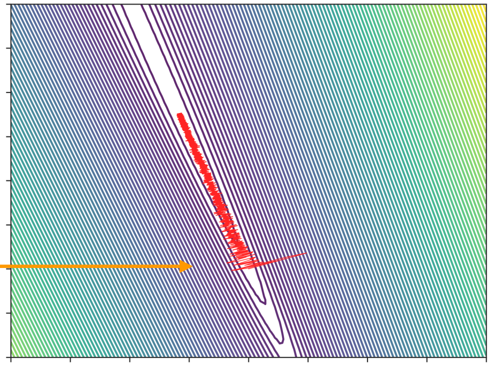
schedule to have a smaller learning rate over time

Robbins Monro

the sequence we use should satisfy: $\sum_{t=0}^{\infty} \alpha^{\{t\}} = \infty$

otherwise for large $||w^{\{0\}} - w^*||$ we can't reach the minimum

the steps should go to zero $\sum_{t=0}^{\infty} (\alpha^{\{t\}})^2 < \infty$



example $\alpha^{\{t\}} = \frac{10}{t}, \alpha^{\{t\}} = t^{-.51}$

Minibatch SGD

use a minibatch to produce gradient estimates

$$\nabla J_{\mathbb{B}} = \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

$\mathbb{B} \subseteq \{1, \ldots, N\}$ a subset of the dataset

Minibatch SGD

use a minibatch to produce gradient estimates

$$\nabla J_{\mathbb{B}} = \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

$\mathbb{B} \subseteq \{1, \ldots, N\}$ a subset of the dataset

```
def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[inibatch], w)
        w = w - lr*g
    return w

```

Minibatch SGD

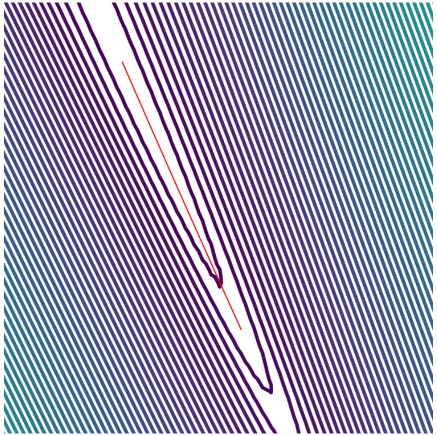
use a minibatch to produce gradient estimates

$$\nabla J_{\mathbb{B}} = \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

$\mathbb{B} \subseteq \{1, \ldots, N\}$ a subset of the dataset

```python
def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[inibatch], w)
        w = w - lr*g
    return w

```

GD full batch



Minibatch SGD

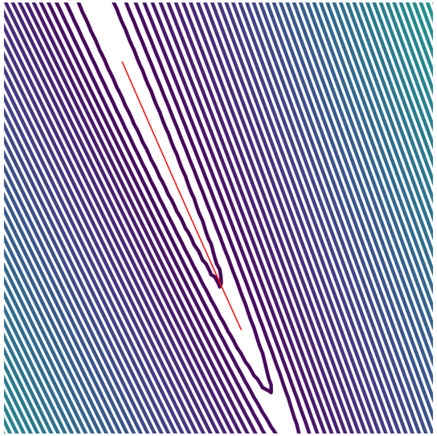
use a minibatch to produce gradient estimates

$$\nabla J_{\mathbb{B}} = \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

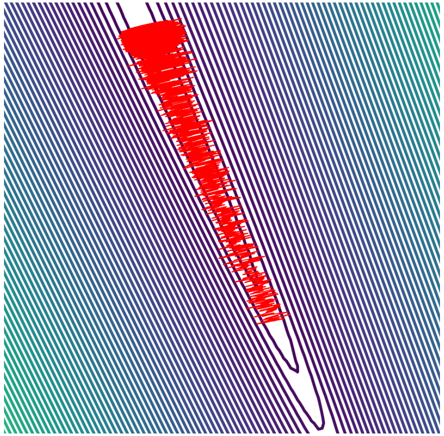
$\mathbb{B} \subseteq \{1, \ldots, N\}$ a subset of the dataset

```python
def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[inibatch], w)
        w = w - lr*g
    return w
```

GD full batch



SGD minibatch-size=1



Minibatch SGD

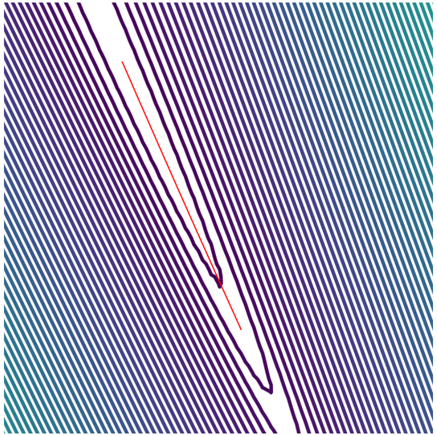
use a minibatch to produce gradient estimates

$$\nabla J_{\mathbb{B}} = \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

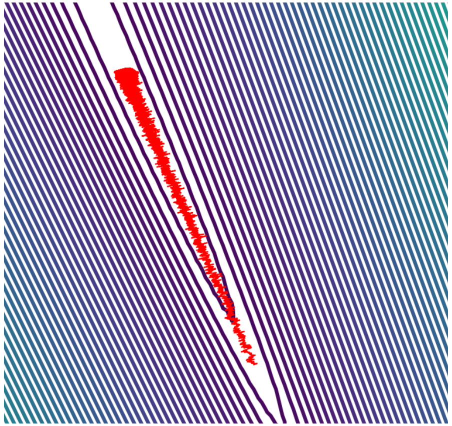
$\mathbb{B} \subseteq \{1, \ldots, N\}$ a subset of the dataset

```
def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[inibatch], w)
        w = w - lr*g
    return w

```

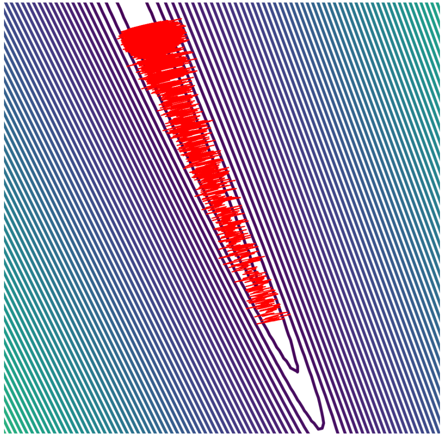
GD full batch



SGD minibatch-size=16



SGD minibatch-size=1



Momentum

to help with oscillations of SGD (or even full-batch GD):

- use a *running average* of gradients
- more recent gradients should have higher weights

Momentum

to help with oscillations of SGD (or even full-batch GD):

- use a *running average* of gradients
- more recent gradients should have higher weights

$$\Delta w^{\{t\}} \leftarrow \beta \Delta w^{\{t-1\}} + (1-\beta)\nabla J_{\mathbb{B}}(w^{\{t\}})$$

$$w^{\{t\}} \leftarrow w^{\{t-1\}} - \alpha \Delta w^{\{t\}}$$

Momentum

to help with oscillations of SGD (or even full-batch GD):

- use a *running average* of gradients
- more recent gradients should have higher weights

$$\Delta w^{\{t\}} \leftarrow \beta \Delta w^{\{t-1\}} + (1-\beta)\nabla J_{\mathbb{B}}(w^{\{t\}})$$

$$w^{\{t\}} \leftarrow w^{\{t-1\}} - \alpha \Delta w^{\{t\}}$$

momentum of 0 reduces to SGD
common value > .9

Momentum

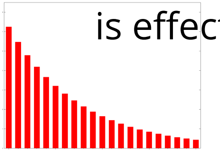
to help with oscillations of SGD (or even full-batch GD):

- use a *running average* of gradients
- more recent gradients should have higher weights

$$\Delta w^{\{t\}} \leftarrow \beta \Delta w^{\{t-1\}} + (1-\beta)\nabla J_{\mathbb{B}}(w^{\{t\}})$$

$$w^{\{t\}} \leftarrow w^{\{t-1\}} - \alpha \Delta w^{\{t\}}$$

momentum of 0 reduces to SGD
common value > .9



is effectively an exponential moving average

$$\Delta w^{\{T\}} = \sum_{t=1}^{T} \beta^{T-t}(1-\beta)\nabla J_{\mathbb{B}}(w^{\{t\}})$$

Momentum

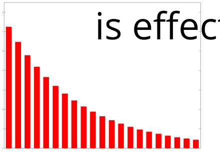
to help with oscillations of SGD (or even full-batch GD):

- use a *running average* of gradients
- more recent gradients should have higher weights

$$\Delta w^{\{t\}} \leftarrow \beta \Delta w^{\{t-1\}} + (1-\beta)\nabla J_{\mathbb{B}}(w^{\{t\}})$$

$$w^{\{t\}} \leftarrow w^{\{t-1\}} - \alpha \Delta w^{\{t\}}$$

momentum of 0 reduces to SGD
common value > .9



is effectively an exponential moving average

$$\Delta w^{\{T\}} = \sum_{t=1}^{T} \beta^{T-t}(1-\beta)\nabla J_{\mathbb{B}}(w^{\{t\}})$$

there are other variations of momentum with similar idea

Momentum

to help with oscillations of SGD (or even full-batch GD):

- use a running average of gradients
- more recent gradients should have higher weights

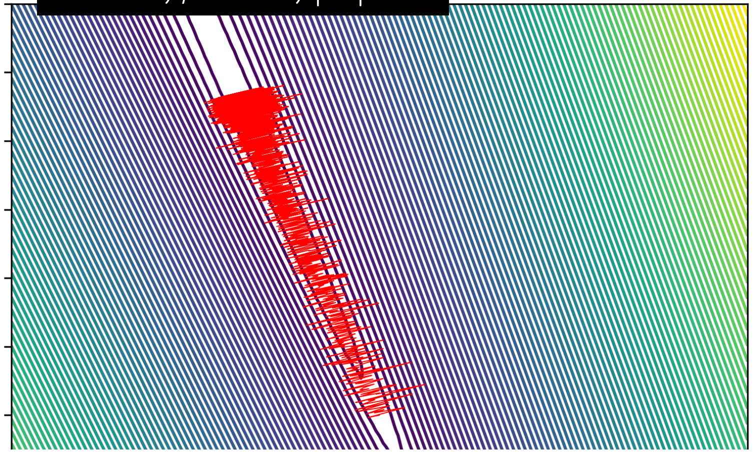
```
def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8, beta=.99):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    dw = 0
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[inibatch], w)
        dw = (1-beta)*g + beta*dw
        w = w - lr*dw
    return w

```

Momentum

Example: logistic regression

no momentum

$\alpha = .5, \beta = 0, |\mathbb{B}| = 8$



Momentum

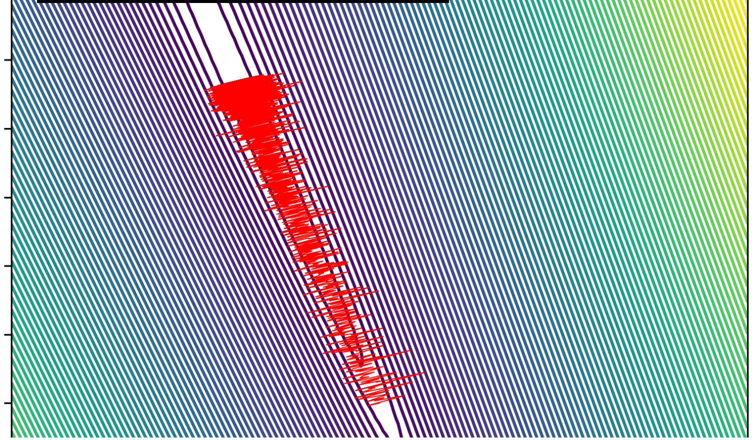
Example: logistic regression

$$\Delta w^t \leftarrow \beta \Delta w^{t-1} + (1-\beta)\nabla J_{\mathbb{B}}(w^{t-1})$$

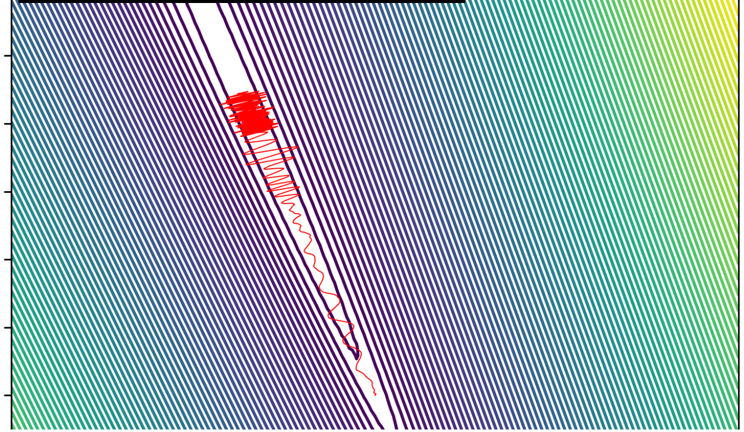
$$w^t \leftarrow w^{t-1} - \alpha \Delta w^t$$

no momentum

$\alpha = .5, \beta = 0, |\mathbb{B}| = 8$



$\alpha = .5, \beta = .99, |\mathbb{B}| = 8$



Momentum

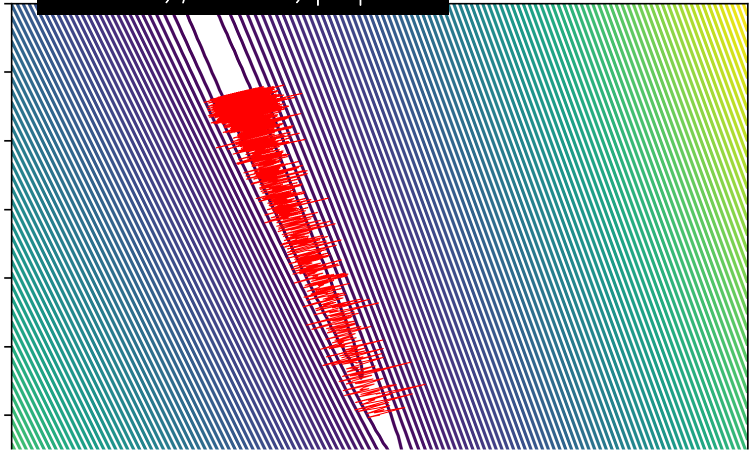
Example: logistic regression

$$\Delta w^t \leftarrow \beta \Delta w^{t-1} + (1-\beta)\nabla J_{\mathbb{B}}(w^{t-1})$$

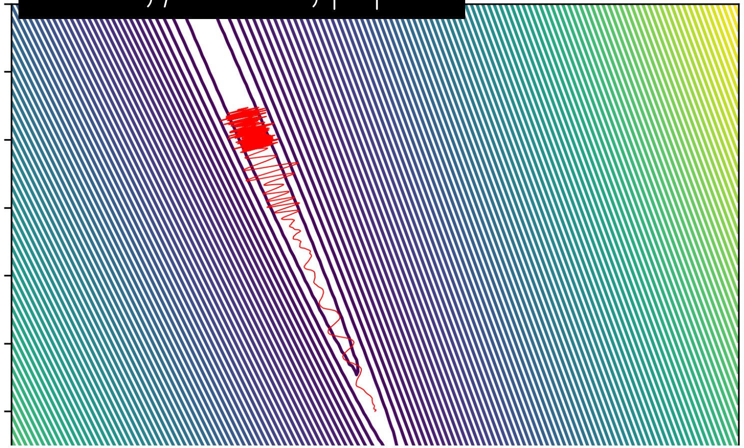
$$w^t \leftarrow w^{t-1} - \alpha \Delta w^t$$

no momentum

$\alpha = .5, \beta = 0, |\mathbb{B}| = 8$



$\alpha = .5, \beta = .99, |\mathbb{B}| = 8$



see the beautiful demo at Distill

https://distill.pub/2017/momentum/

Adagrad (Adaptive gradient)

use different learning rate for each parameter w_d

also make the learning rate adaptive

Adagrad (**Ada**ptive **grad**ient)

use different learning rate for each parameter w_d

also make the learning rate adaptive

$$S_d^{\{t\}} \leftarrow S_d^{\{t-1\}} + \frac{\partial}{\partial w_d} J(w^{\{t-1\}})^2$$

sum of squares of derivatives over all iterations so far (for individual parameter)

Adagrad (Adaptive gradient)

use different learning rate for each parameter w_d

also make the learning rate adaptive

$$S_d^{\{t\}} \leftarrow S_d^{\{t-1\}} + \frac{\partial}{\partial w_d} J(w^{\{t-1\}})^2$$

sum of squares of derivatives over all iterations so far (for individual parameter)

$$w_d^{\{t\}} \leftarrow w_d^{\{t-1\}} - \frac{\alpha}{\sqrt{S_d^{\{t-1\}} + \epsilon}} \frac{\partial}{\partial w_d} J(w^{\{t-1\}})$$

the learning rate is adapted to previous updates

ϵ is to avoid numerical issues

Adagrad (Adaptive gradient)

use different learning rate for each parameter w_d

also make the learning rate adaptive

$$S_d^{\{t\}} \leftarrow S_d^{\{t-1\}} + \frac{\partial}{\partial w_d} J(w^{\{t-1\}})^2$$

sum of squares of derivatives over all iterations so far (for individual parameter)

$$w_d^{\{t\}} \leftarrow w_d^{\{t-1\}} - \frac{\alpha}{\sqrt{S_d^{\{t-1\}} + \epsilon}} \frac{\partial}{\partial w_d} J(w^{\{t-1\}})$$

the learning rate is adapted to previous updates

ϵ is to avoid numerical issues

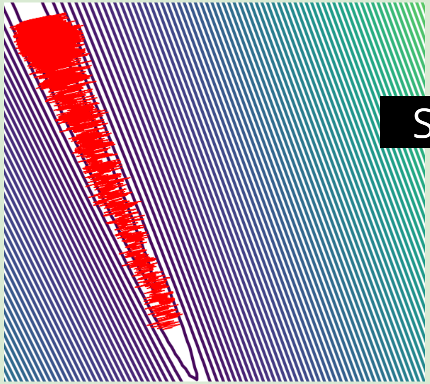
useful when parameters are updated at different rates (*e.g., NLP*)

Adagrad (Adaptive gradient)

different learning rate for each parameter w_d

make the learning rate adaptive

$\alpha = .1, |\mathbb{B}| = 1, T = 80,000$



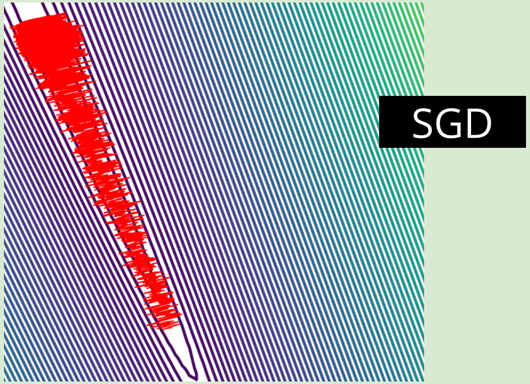
SGD

Adagrad (Adaptive gradient)

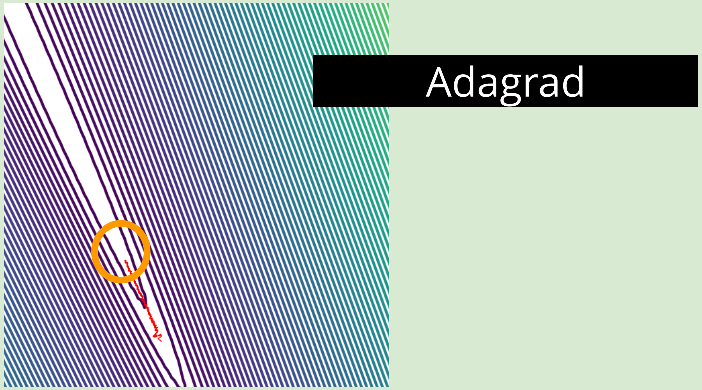
different learning rate for each parameter w_d

make the learning rate adaptive

$\alpha = .1, |\mathbb{B}| = 1, T = 80,000$



$\alpha = .1, |\mathbb{B}| = 1, T = 80,000, \epsilon = 1e-8$

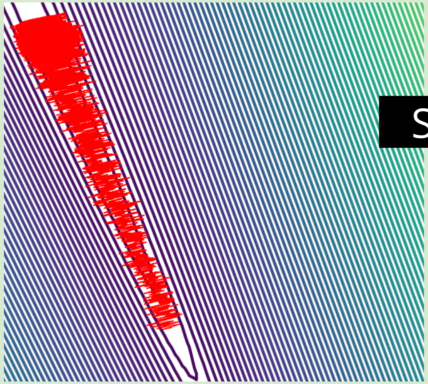


Adagrad (Adaptive gradient)

different learning rate for each parameter w_d

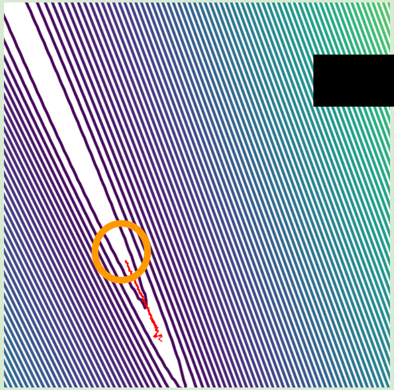
make the learning rate adaptive

$\alpha = .1, |\mathbb{B}| = 1, T = 80,000$



SGD

$\alpha = .1, |\mathbb{B}| = 1, T = 80,000, \epsilon = 1e-8$



Adagrad

problem: *the learning rate goes to zero too quickly*

RMSprop

(Root Mean Squared propagation)

solve the problem of diminishing step-size with Adagrad

- use exponential moving average instead of sum (similar to momentum)

$$S^{\{t\}} \leftarrow \gamma S^{\{t-1\}} + (1-\gamma)\nabla J(w^{\{t-1\}})^2$$

$$w^{\{t\}} \leftarrow w_d^{\{t-1\}} - \frac{\alpha}{\sqrt{S^{\{t-1\}}+\epsilon}}\nabla J(w^{\{t-1\}})$$

identical to Adagrad

```
def RMSprop(X, y, lr=.01, eps=1e-2, bsize=8, gamma=.9, epsilon=1e-8):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    S = 0
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[inibatch], w)
        S = (1-gamma)*g**2 + gamma*S
        w = w - lr*g/np.sqrt(S + epsilon)
    return w
```

Adam (Adaptive Moment Estimation)

two ideas so far:

1. use momentum to smooth out the oscillations
2. adaptive per-parameter learning rate

both use exponential moving averages

Adam **combines the two**:

$$M^{\{t\}} \leftarrow \beta_1 M^{\{t-1\}} + (1-\beta_1)\nabla J(w^{\{t-1\}})$$

identical to method of momentum
(moving average of the first moment)

$$S^{\{t\}} \leftarrow \beta_2 S^{\{t-1\}} + (1-\beta_2)\nabla J(w^{\{t-1\}})^2$$

identical to RMSProp
(moving average of the second moment)

$$w^{\{t\}} \leftarrow w_d^{\{t-1\}} - \frac{\alpha \hat{M}^{\{t\}}}{\sqrt{\hat{S}^{\{t\}}} + \epsilon} \nabla J(w^{\{t-1\}})$$

Adam (Adaptive Moment Estimation)

Adam combines thee two:

$$M^{\{t\}} \leftarrow \beta_1 M^{\{t-1\}} + (1-\beta_1)\nabla J(w^{\{t-1\}})$$

identical to method of momentum
(moving average of the first moment)

$$S^{\{t\}} \leftarrow \beta_2 S^{\{t-1\}} + (1-\beta_2)\nabla J(w^{\{t-1\}})^2$$

identical to RMSProp
(moving average of the second moment)

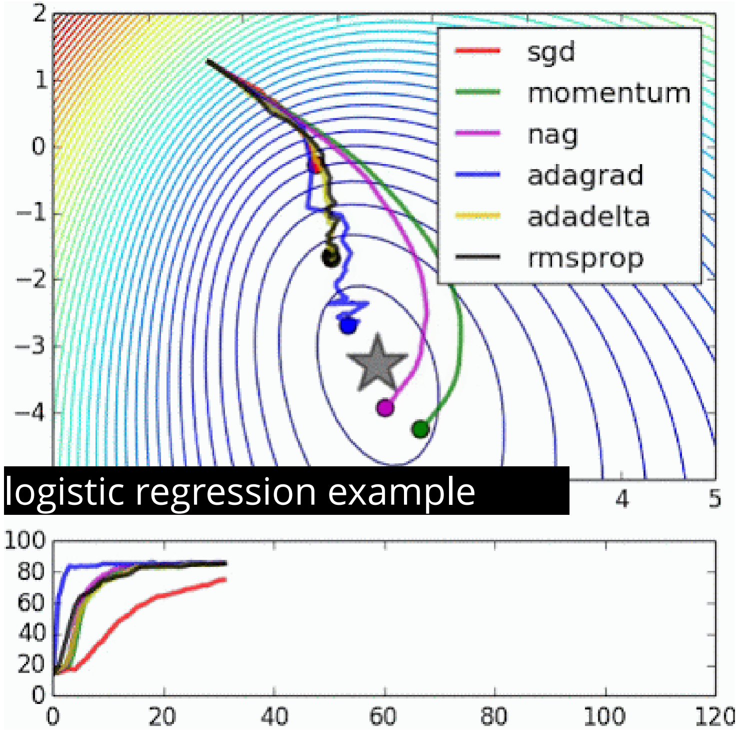
$$w^{\{t\}} \leftarrow w_d^{\{t-1\}} - \frac{\alpha \hat{M}^{\{t\}}}{\sqrt{\hat{S}^{\{t\}}+\epsilon}} \nabla J(w^{\{t-1\}})$$

since M and S are initialized to be zero, at early stages they are biased towards zero

$$\hat{M}^{\{t\}} \leftarrow \frac{M^{\{t\}}}{1-\beta_1^t} \quad \hat{S}^{\{t\}} \leftarrow \frac{S^{\{t\}}}{1-\beta_2^t}$$

for large time-steps it has no effect
for small t, it scales up numerator

In practice



logistic regression example

image:Alec Radford

the list of methods is growing ...

they have recommended range of parameters

- *learning rate, momentum etc.*

still may need some hyper-parameter tuning

these are all **first order methods**

- they only need the first derivative
- 2nd order methods can be much more effective, but also much more expensive

Adding L_2 regularization

do not penalize the bias w_0

Adding L_2 regularization

do not penalize the bias w_0

```
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay

Adding L_2 regularization

do not penalize the bias w_0

L2 penalty makes the optimization easier too!

```
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay

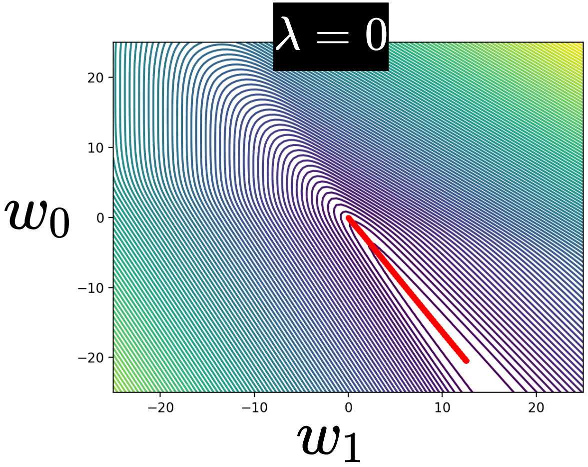
Adding L_2 regularization

do not penalize the bias w_0

L2 penalty makes the optimization easier too!

```
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay



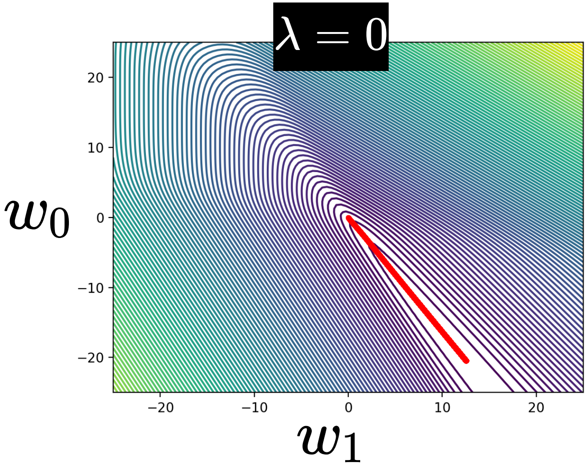
Adding L_2 regularization

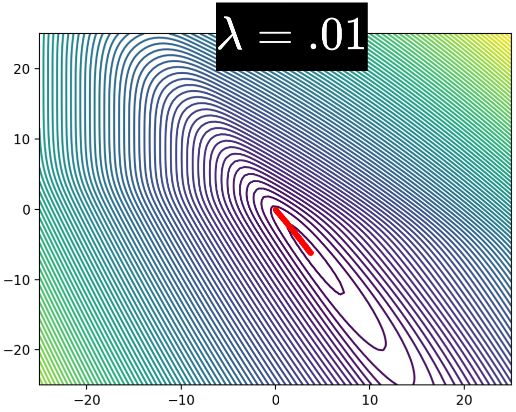
do not penalize the bias w_0

L2 penalty makes the optimization easier too!

```
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay





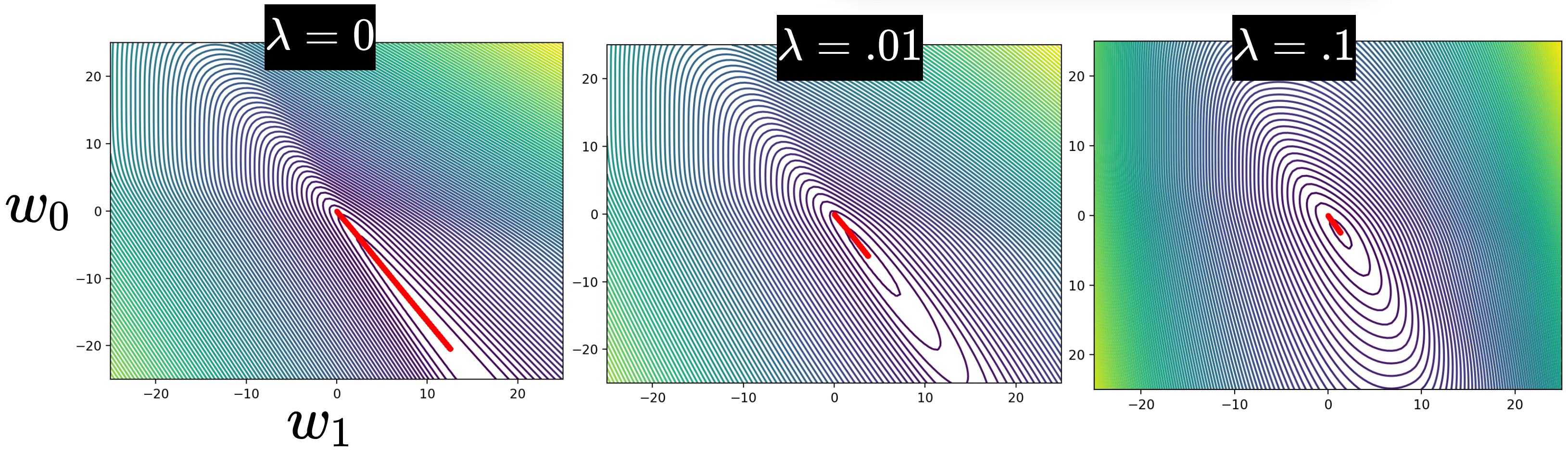
Adding L_2 regularization

do not penalize the bias w_0

L2 penalty makes the optimization easier too!

```python
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay



Adding L_2 regularization

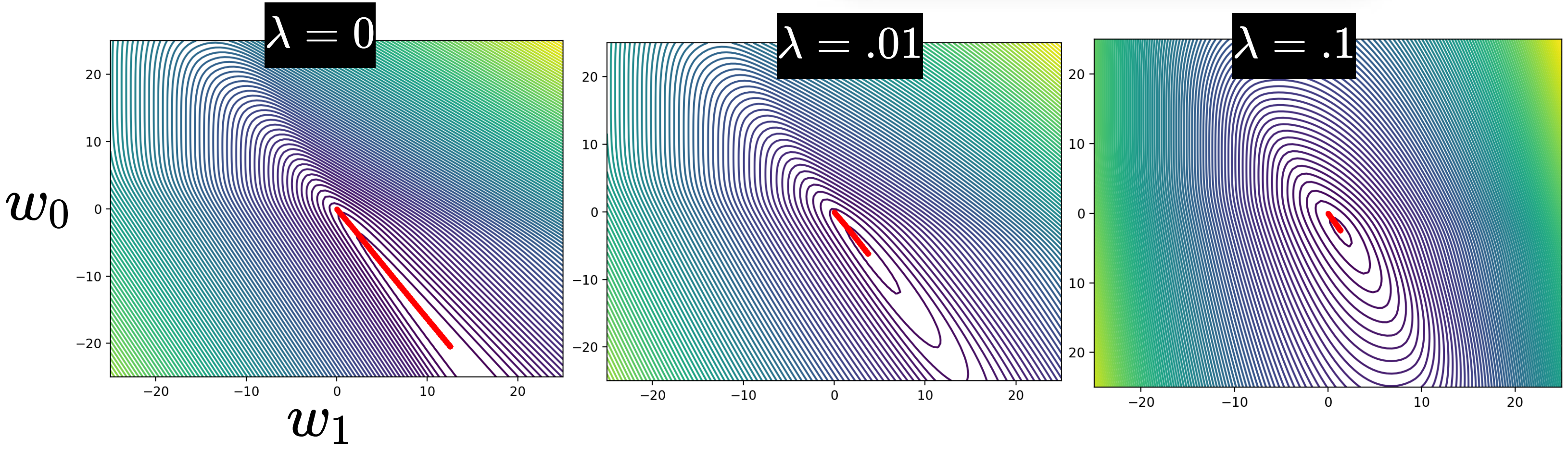
do not penalize the bias w_0

L2 penalty makes the optimization easier too!

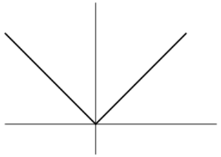
note that the optimal w_1 shrinks

```
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay



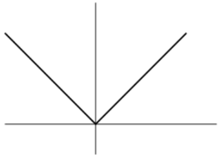
Subgderivatives



L1 penalty is no longer smooth or differentiable (at 0)

extend the notion of derivative to non-smooth functions

Subgderivatives



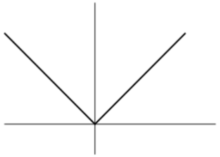
L1 penalty is no longer smooth or differentiable (at 0)

extend the notion of derivative to non-smooth functions

sub-differential is the set of all **sub-derivatives** at a point

$$\partial f(\hat{w}) = \left[\lim_{w \to \hat{w}^-} \frac{f(w) - f(\hat{w})}{w - \hat{w}}, \lim_{w \to \hat{w}^+} \frac{f(w) - f(\hat{w})}{w - \hat{w}}\right]$$

Subgderivatives



L1 penalty is no longer smooth or differentiable (at 0)

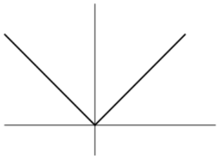
extend the notion of derivative to non-smooth functions

sub-differential is the set of all **sub-derivatives** at a point

$$\partial f(\hat{w}) = \left[\lim_{w \to \hat{w}^-} \frac{f(w)-f(\hat{w})}{w-\hat{w}}, \lim_{w \to \hat{w}^+} \frac{f(w)-f(\hat{w})}{w-\hat{w}}\right]$$

if ***f*** is differentiable at $\hat{w}$ then sub-differential has one member $\frac{d}{dw} f(\hat{w})$

Subgderivatives



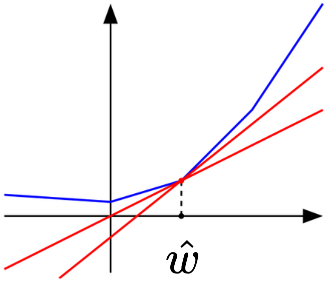
L1 penalty is no longer smooth or differentiable (at 0)

extend the notion of derivative to non-smooth functions

sub-differential is the set of all **sub-derivatives** at a point

$$\partial f(\hat{w}) = \left[\lim_{w \to \hat{w}^-} \frac{f(w) - f(\hat{w})}{w - \hat{w}}, \lim_{w \to \hat{w}^+} \frac{f(w) - f(\hat{w})}{w - \hat{w}}\right]$$

if f is differentiable at $\hat{w}$ then sub-differential has one member $\frac{d}{dw} f(\hat{w})$

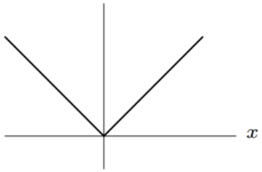


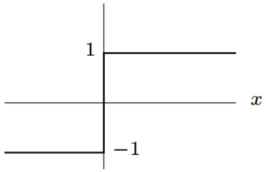
another expression for sub-differential

$$\partial f(\hat{w}) = \{g \in \mathbb{R} |\ f(w) > f(\hat{w}) + g(w - \hat{w})\}$$

Subgradient

example subdifferential absolute $f(w) = |w|$





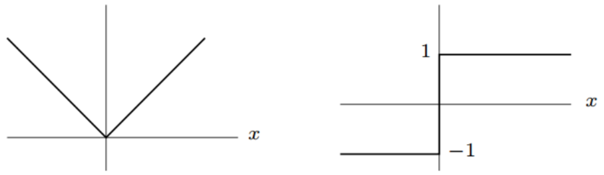
$$\partial f(0) = [-1, 1]$$

$$\partial f(w \neq 0) = \{\text{sign}(w)\}$$

image credit: G. Gordon

Subgradient

example subdifferential absolute $f(w) = |w|$



$$\partial f(0) = [-1, 1]$$

$$\partial f(w \neq 0) = \{\text{sign}(w)\}$$

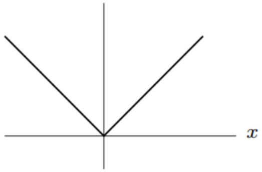
recall, **gradient** was the vector of **partial derivatives**

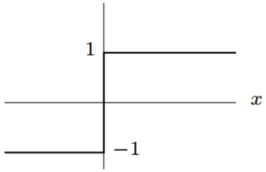
subgradient is a vector of **sub-derivatives**

image credit: G. Gordon

Subgradient

example subdifferential absolute $f(w) = |w|$





$$\partial f(0) = [-1, 1]$$

$$\partial f(w \neq 0) = \{\mathrm{sign}(w)\}$$

recall, **gradient** was the vector of **partial derivatives**

subgradient is a vector of **sub-derivatives**

subdifferential for functions of multiple variables

$$\partial f(\hat{w}) = \{g \in \mathbb{R}^D | f(w) > f(\hat{w}) + g^T(w - \hat{w})\}$$

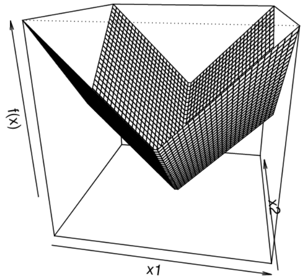
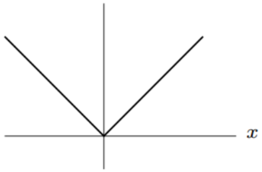
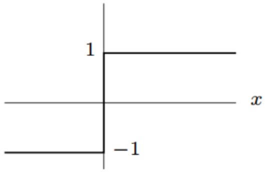


image credit: G. Gordon

Subgradient

example subdifferential absolute $f(w) = |w|$





$$\partial f(0) = [-1, 1]$$

$$\partial f(w \neq 0) = \{\text{sign}(w)\}$$

recall, **gradient** was the vector of **partial derivatives**

subgradient is a vector of **sub-derivatives**

subdifferential for functions of multiple variables

$$\partial f(\hat{w}) = \{g \in \mathbb{R}^D | f(w) > f(\hat{w}) + g^T(w - \hat{w})\}$$

we can use sub-gradient with diminishing step-size for optimization

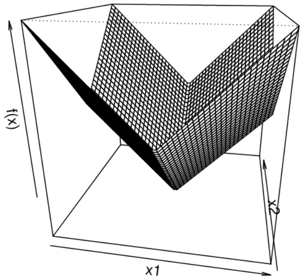


image credit: G. Gordon

Adding L_1 regularization

L1-regularized *linear regression* has efficient solvers

subgradient method for L1-regularized logistic regression

Adding L_1 regularization

L1-regularized *linear regression* has efficient solvers

subgradient method for L1-regularized logistic regression

do not penalize the bias w_0

using **diminishing learning rate**

Adding L_1 regularization

L1-regularized *linear regression* has efficient solvers

subgradient method for L1-regularized logistic regression

do not penalize the bias w_0

using **diminishing learning rate**

```python
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * np.sign(w[1:])
    return grad
```

Adding L_1 regularization

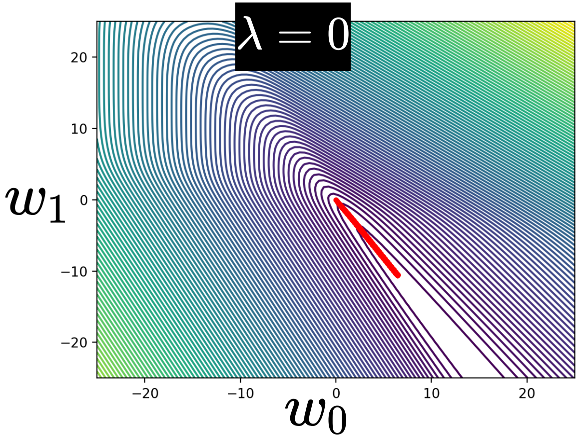
L1-regularized *linear regression* has efficient solvers

subgradient method for L1-regularized logistic regression

do not penalize the bias w_0

using **diminishing learning rate**

```python
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * np.sign(w[1:])
    return grad
```



Adding L_1 regularization

L1-regularized *linear regression* has efficient solvers

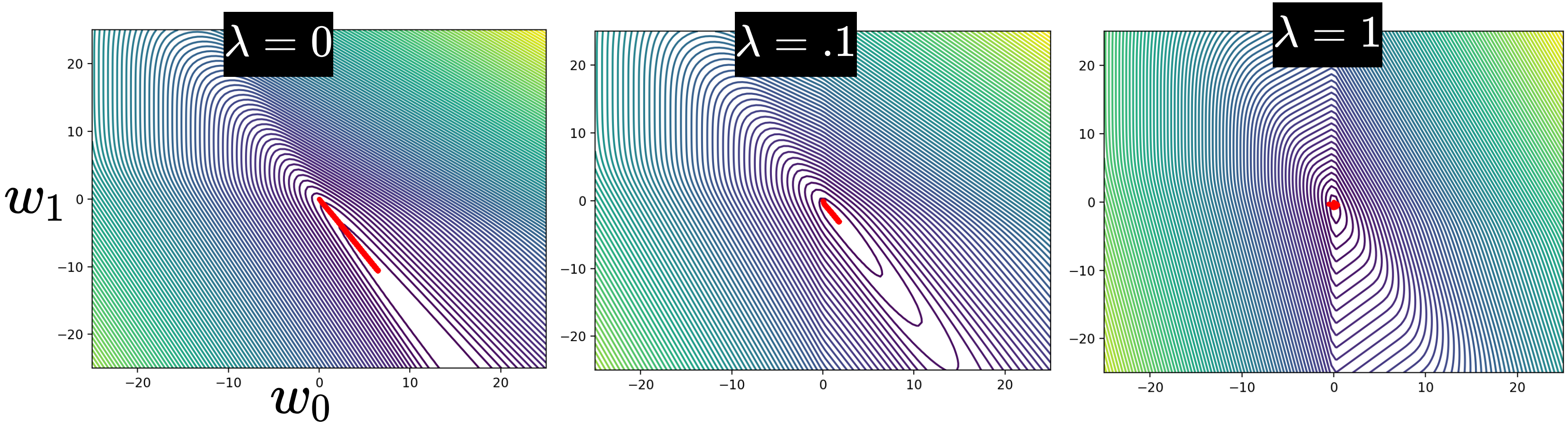
subgradient method for L1-regularized logistic regression

do not penalize the bias w_0

using **diminishing learning rate**

note that the optimal w_1 **becomes 0**

```python
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * np.sign(w[1:])
    return grad
```



Summary

learning: optimizing the model parameters (minimizing a cost function)

use **gradient descent** to find local minimum

- easy to implement (esp. using automated differentiation)
- for **convex functions** gives global minimum

Summary

learning: optimizing the model parameters (minimizing a cost function)

use **gradient descent** to find local minimum

- easy to implement (esp. using automated differentiation)
- for **convex functions** gives global minimum

Stochastic GD: for large data-sets use mini-batch for a noisy-fast estimate of gradient

- **Robbins Monro** condition: reduce the learning rate to help with the noise

better (stochastic) gradient optimization

- **Momentum:** exponential running average to help with the noise
- **Adagrad & RMSProp:** per parameter adaptive learning rate
- **Adam:** combining these two ideas

Summary

learning: optimizing the model parameters (minimizing a cost function)

use **gradient descent** to find local minimum

- easy to implement (esp. using automated differentiation)
- for **convex functions** gives global minimum

Stochastic GD: for large data-sets use mini-batch for a noisy-fast estimate of gradient

- **Robbins Monro** condition: reduce the learning rate to help with the noise

better (stochastic) gradient optimization

- **Momentum:** exponential running average to help with the noise
- **Adagrad & RMSProp:** per parameter adaptive learning rate
- **Adam:** combining these two ideas

Adding regularization can also help with optimization

Adadelta

solve the problem of diminishing step-size with Adagrad

- use exponential moving average instead of sum (similar to momentum)

also gets rid of a "learning rate" altogether

- use another moving average for that!

$$S^{\{t\}} \leftarrow \gamma S^{\{t-1\}} + (1-\gamma)\nabla J(w^{\{t-1\}})^2$$

moving average of the sq. gradient

$$U^{\{t\}} \leftarrow \gamma U^{\{t-1\}} + (1-\gamma)\Delta w^{\{t-1\}}$$

moving average of the sq. updates

$$\Delta w^{\{t\}} \leftarrow -\sqrt{\frac{U^{\{t-1\}}}{S^{\{t\}}+\epsilon}}\nabla J(w^{\{t-1\}})$$

square root of the ratio of the above is used as the adaptive learning rate

$$w^{\{t\}} \leftarrow w^{\{t-1\}} + \Delta w^{\{t\}}$$