

Applied Machine Learning

Gradient Descent Methods

Reihaneh Rabbany



COMP 551 (winter 2020)

Learning objectives

Basic idea of

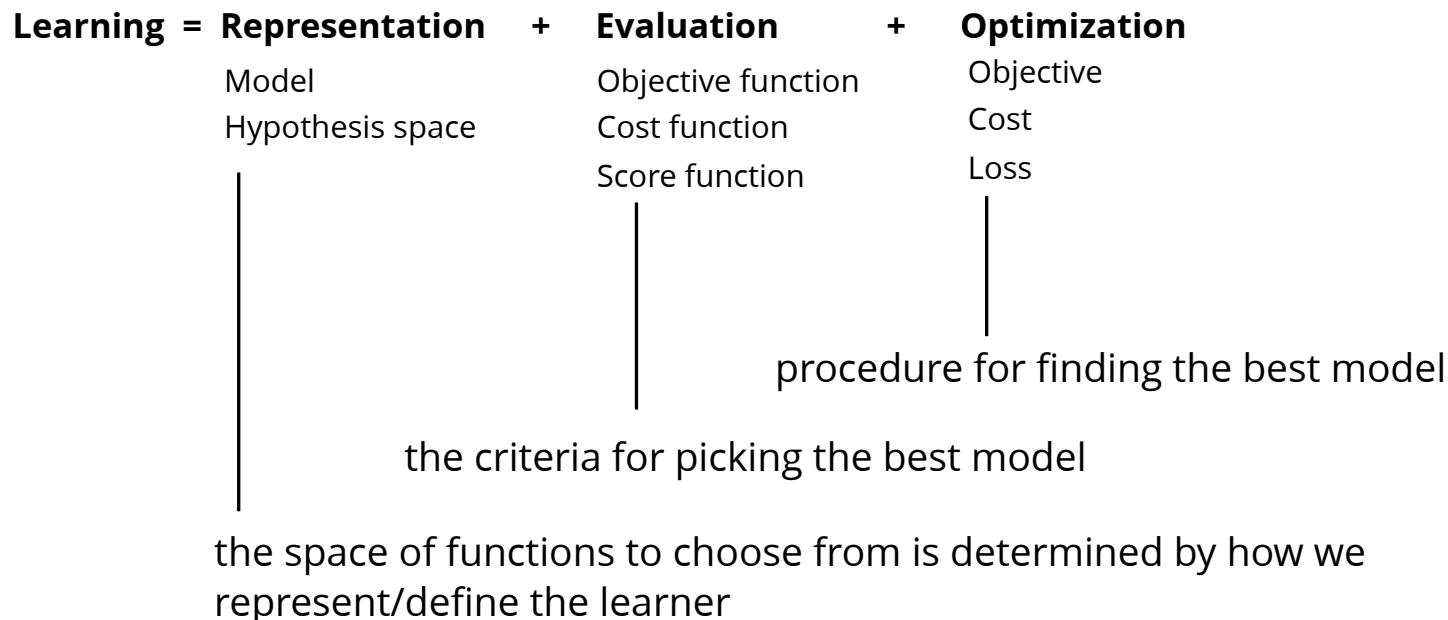
- gradient descent
- stochastic gradient descent
- method of momentum
- using adaptive learning rate
- sub-gradient

Application to

- linear regression and classification

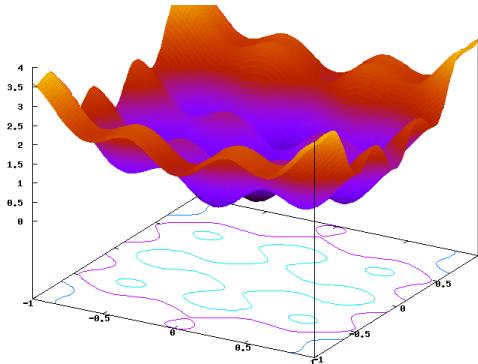
Optimization in ML

Inference and learning of a model often involves optimization:



Optimization 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

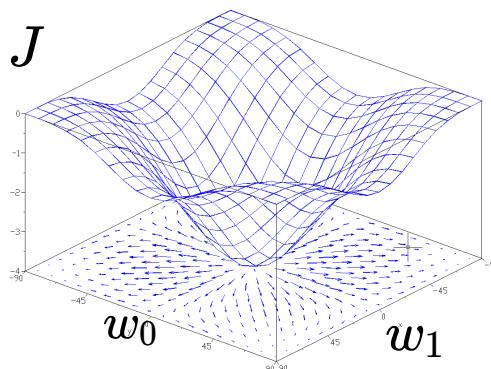
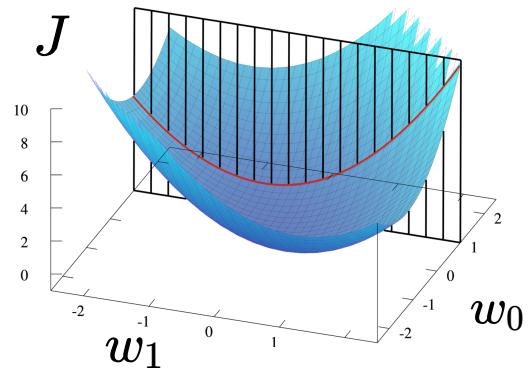
= derivative when other vars. are fixed

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

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

gradient: vector of all partial derivatives

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

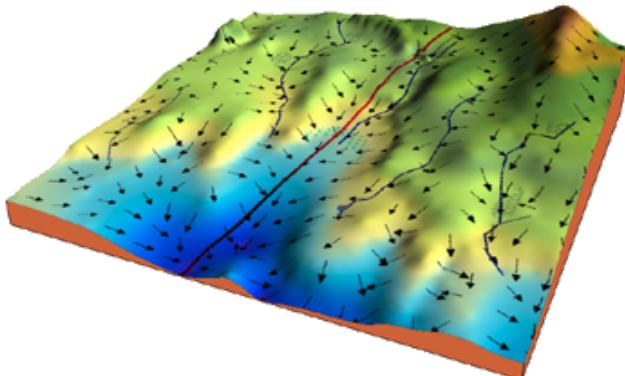


Gradient descent

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



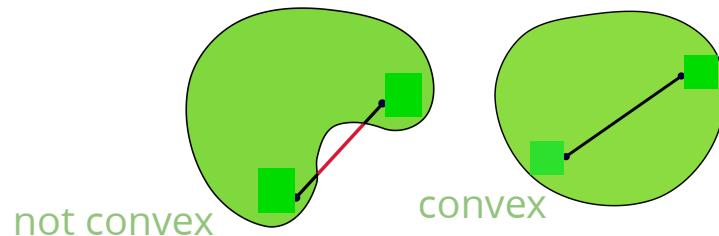
α learning rate
 $\nabla \mathcal{J}(w)$ cost function
(for maximization : objective function)

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

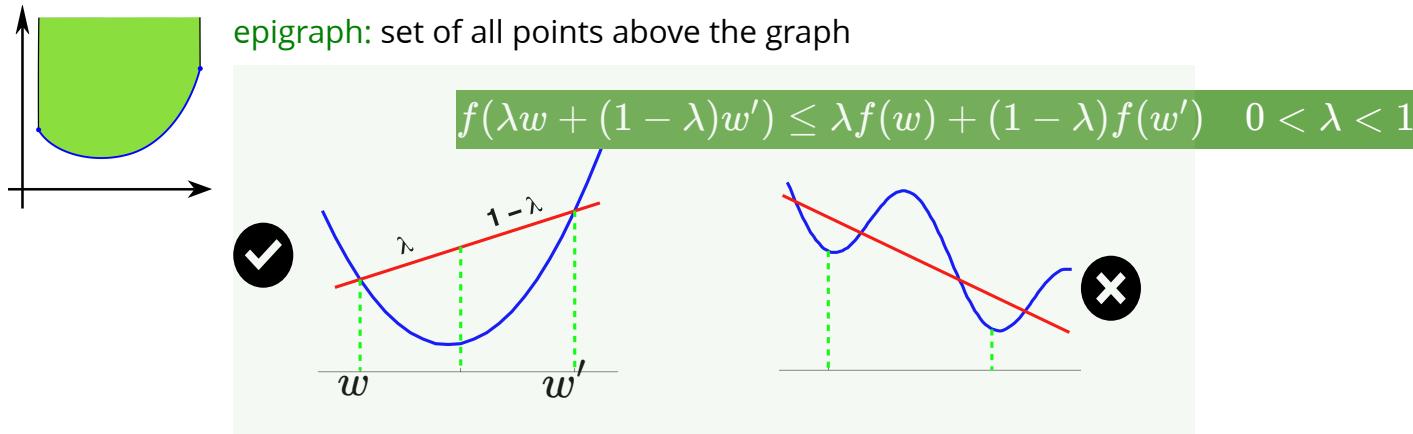
image: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Convex function

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



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



Convex function

Convex functions are easier to minimize:

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

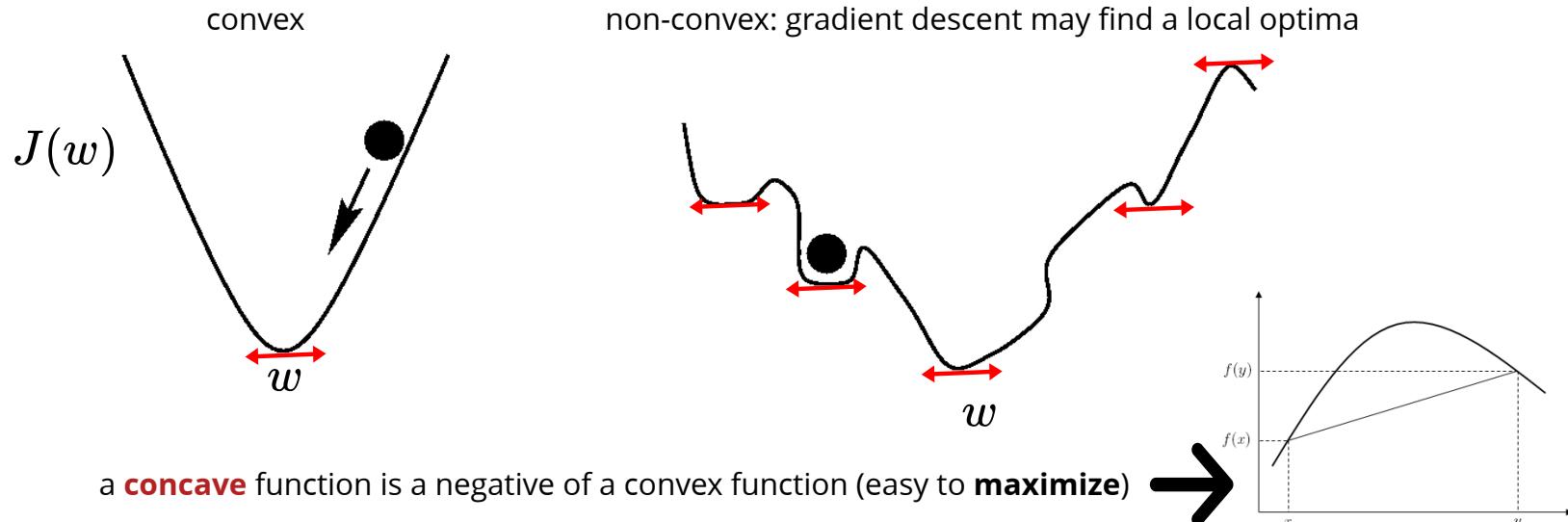


image: <https://www.willamette.edu/~gorr/classes/cs449/momrate.html>

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

sum of convex functions is convex

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

maximum of convex functions is convex

example $f(y) = \max_x \sqrt{xy}^4$

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: $\nabla J(w) = X^T(\hat{y} - y)$

- linear regression: $\hat{y} = Xw$
- 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



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



Gradient Descent

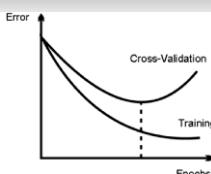
implementing gradient descent is easy!

```
1 def GradientDescent(X, # N x D  
2                      y, # N  
3                      lr=.01, # learning rate  
4                      eps=1e-2, # termination condition  
5                      ):  
6     N,D = X.shape  
7     w = np.zeros(D)  
8     g = np.inf  
9     while np.linalg.norm(g) > eps:  
10        g = gradient(X, y, w) code on the previous page  
11        w = w - lr*g  
12    return w  
13
```

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 fit toy data

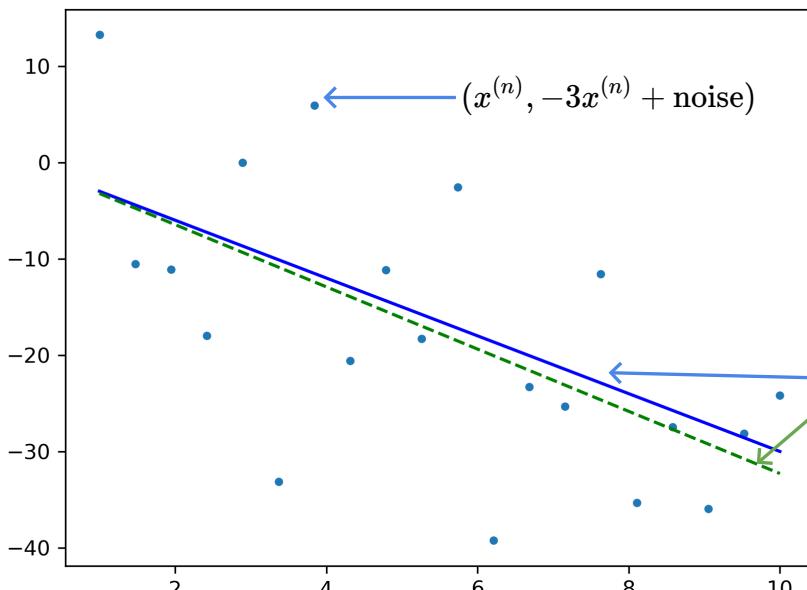
```
•••  
1 def GradientDescent(X, # N x D  
2                 y, # N  
3                 lr=.01, # learning rate  
4                 eps=1e-2, # termination condition  
5                 ):  
6     N,D = X.shape  
7     w = np.zeros(D)  
8     g = np.inf  
9     while np.linalg.norm(g) > eps:  
10         g = gradient(X, y, w) →  
11         w = w - lr*g  
12     return w  
13
```

```
•••  
1 def gradient(X, y, w):  
2     N,D = X.shape  
3     yh = np.dot(X, w)  
4     grad = np.dot(X.T, yh - y) / N  
5     return grad
```

Example: GD for Linear Regression

applying this to fit toy data ←

single feature (intercept is zero)



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

using direct solution method

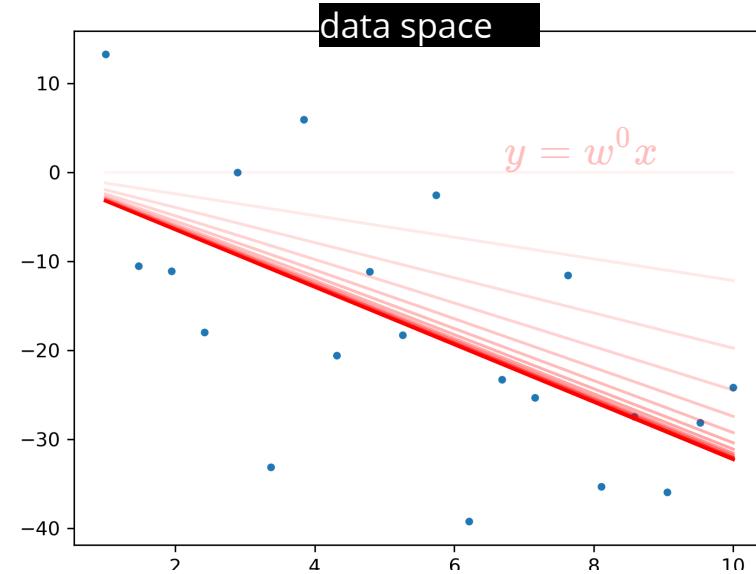
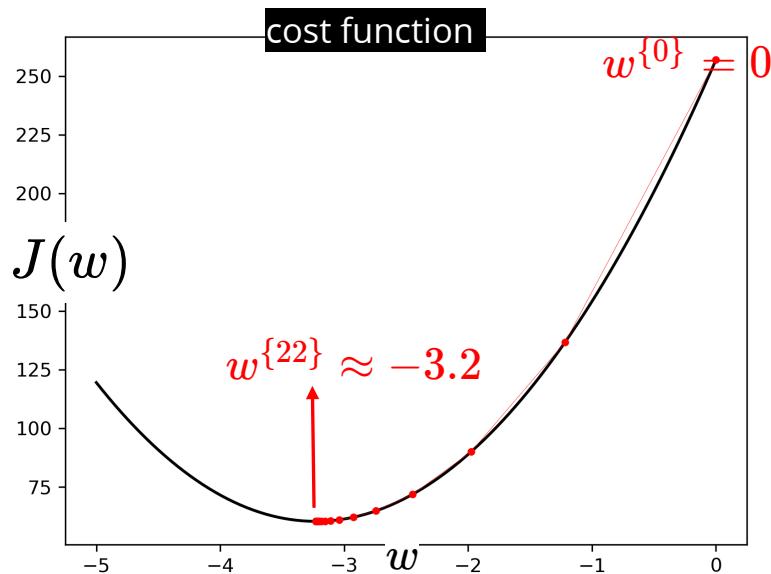
$$w = (X^T X)^{-1} X^T y \approx -3.2$$

$$y = wx$$

$$y = -3x$$

Example: GD for Linear Regression

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

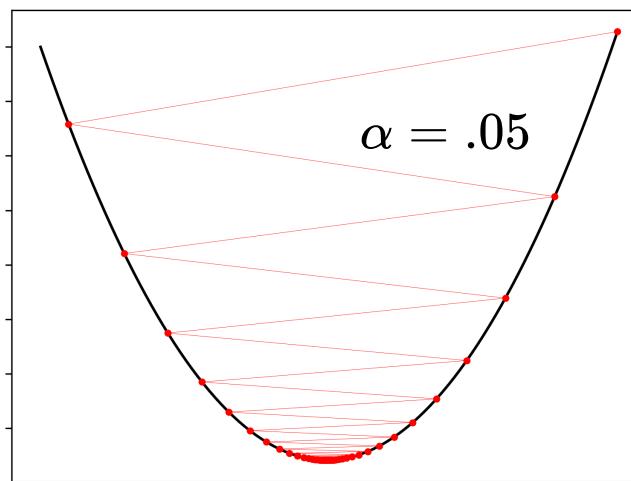
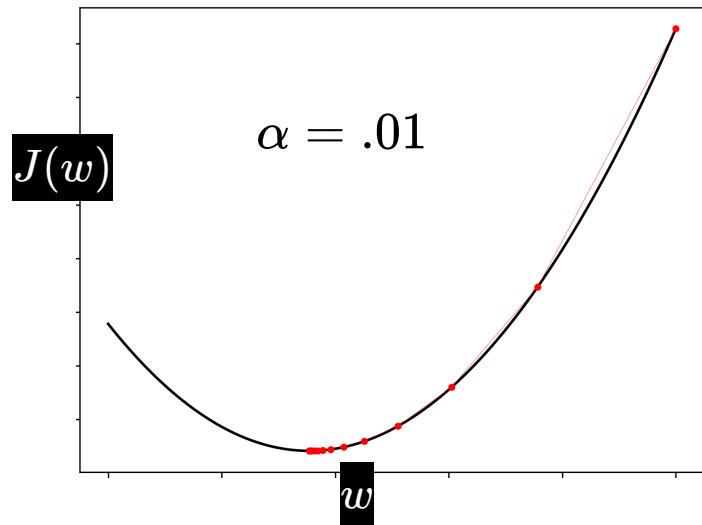


Learning rate α

Learning rate has a significant effect on GD

too small: may take a long time to converge

too large: it overshoots

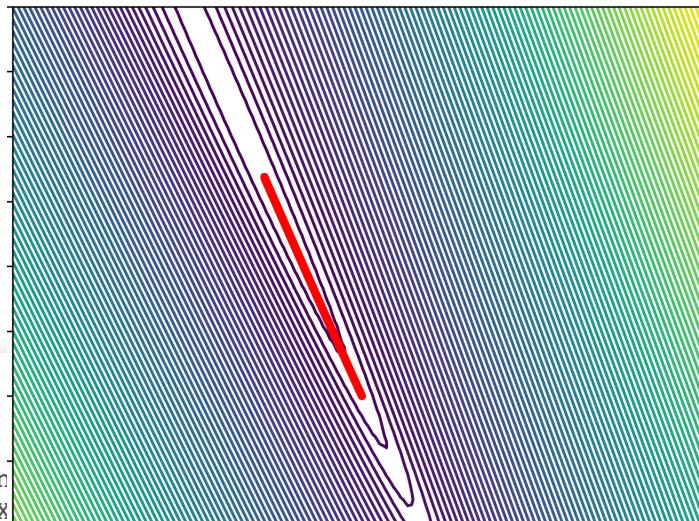


GD for logistic Regression

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

```
1 def GradientDescent(X, # N x D
2                     Y, # N
3                     lr=.01, # learning rate
4                     eps=1e-2, # termination condition
5                     ):
6     N,D = X.shape
7     w = np.zeros(D)
8     g = np.inf
9     while np.linalg.norm(g) > eps:
10         g = gradient(X, Y, w)
11         w = w - lr*g
12     return w
13
```

```
1 def gradient(X, Y,
2               yh = logistic(rn)
3               grad = np.dot(X
4               return grad
```



Stochastic Gradient Descent

we can write the cost function as a average over instances

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

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

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

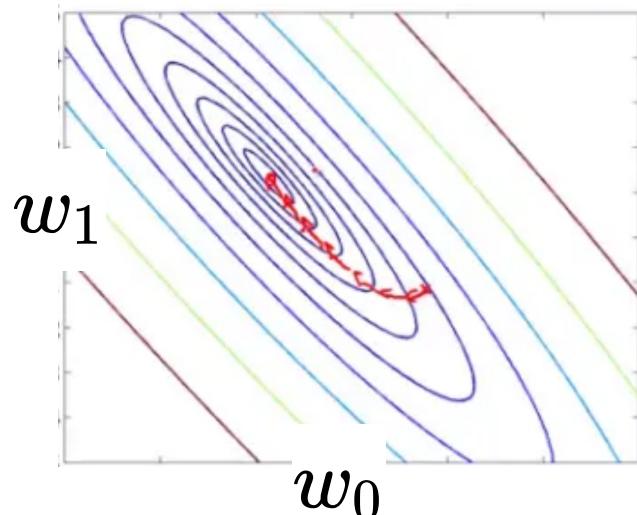


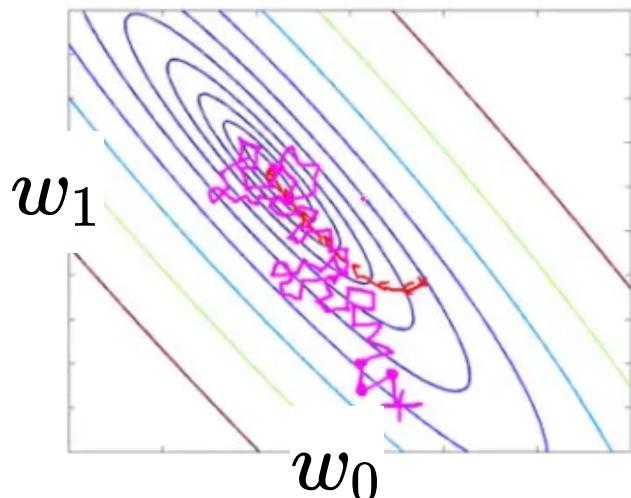
image:<https://jaykanidan.wordpress.com>

Stochastic Gradient Descent

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

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

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

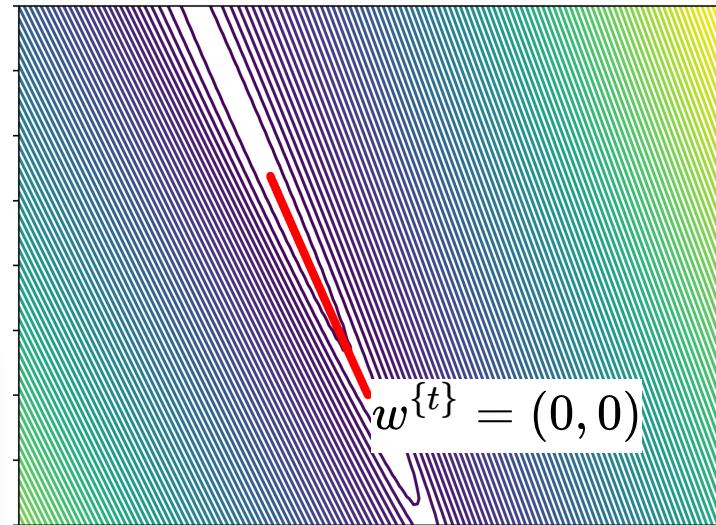
setting 1: using batch gradient

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

```
1 def GradientDescent(X, # N x D
2                     y, # N
3                     lr=.01, # learning rate
4                     eps=1e-2, # termination condition
5                     ):
6     N,D = X.shape
7     w = np.zeros(D)
8     g = np.inf
9     while np.linalg.norm(g) > eps:
10        g = gradient(X, y, w)
11        w = w - lr*g
12    return w
13
```

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

after 8000 iterations



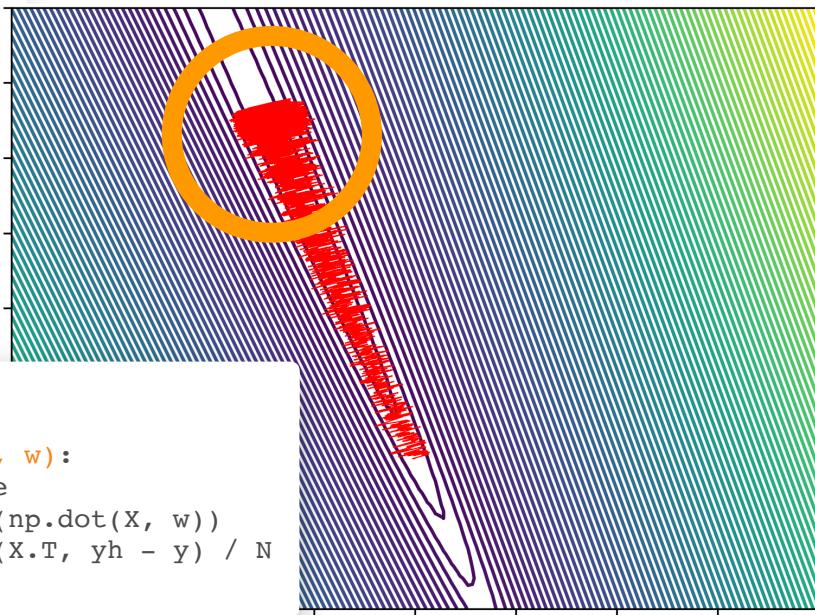
Example: SGD for logistic regression

setting 2: using **stochastic** gradient

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

```
● ● ●  
1 def StochasticGradientDescent(  
2                               X, # N x D  
3                               y, # N  
4                               lr=.01, # learning rate  
5                               eps=1e-2, # termination condition  
6 ):  
7     N,D = X.shape  
8     w = np.zeros(D)  
9     g = np.inf  
10    while np.linalg.norm(g) > eps:  
11        n = np.random.randint(N)  
12        g = gradient(X[[n],:], y[[n]], w)  
13        w = w - lr*g  
14    return w  
15
```

```
● ● ●  
1 def gradient(X, y, w):  
2     N, D = X.shape  
3     yh = logistic(np.dot(X, w))  
4     grad = np.dot(X.T, yh - y) / N  
5     return grad
```



Convergence of SGD

stochastic gradients are not zero at optimum
how to guarantee convergence?

schedule to have a smaller learning rate over time

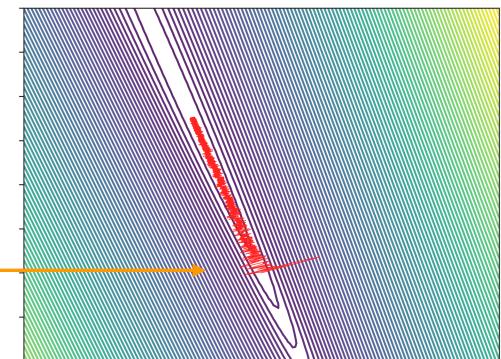
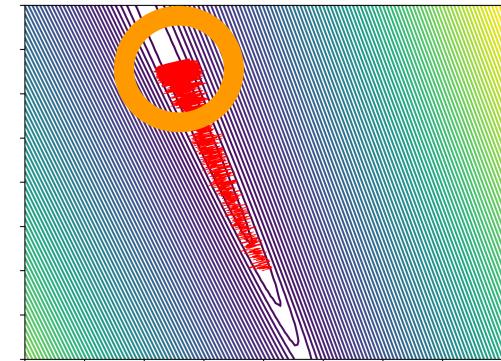
Robbins Monro

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 $\alpha^{\{t\}} = \frac{10}{t}, \alpha^{\{t\}} = t^{-0.51}$



Minibatch SGD

use a minibatch to produce gradient estimates

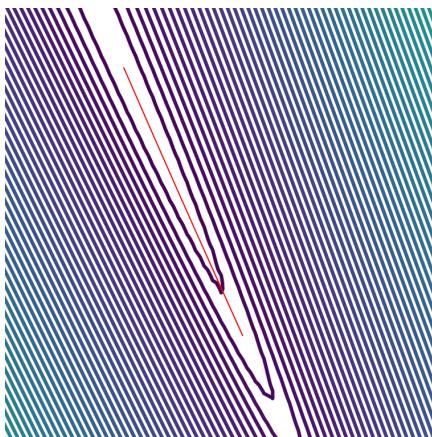
$$\nabla J_{\mathbb{B}} = \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

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

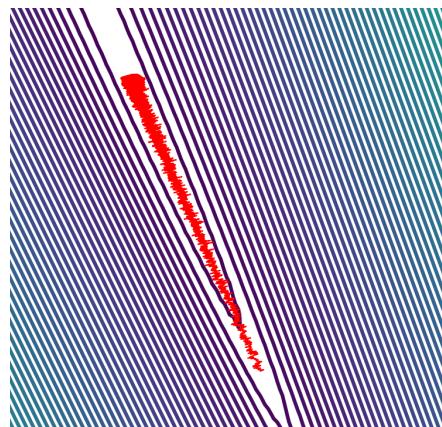


```
1 def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8):
2     N,D = X.shape
3     w = np.zeros(D)
4     g = np.inf
5     while np.linalg.norm(g) > eps:
6         minibatch = np.random.randint(N, size=(bsize))
7         g = gradient(X[minibatch,:], y[inibatch], w)
8         w = w - lr*g
9     return w
10
```

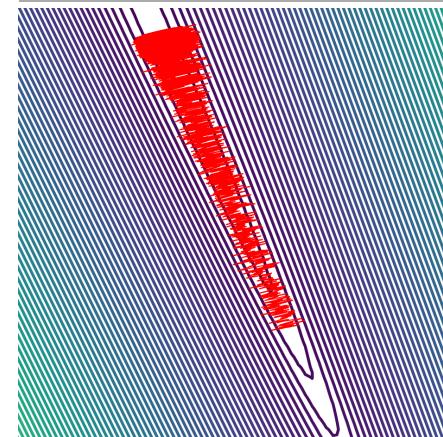
GD full batch



SGD minibatch-size=16



SGD minibatch-size=1



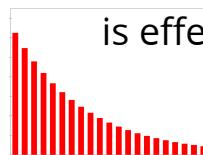
Momentum

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

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

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

|
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

• • •

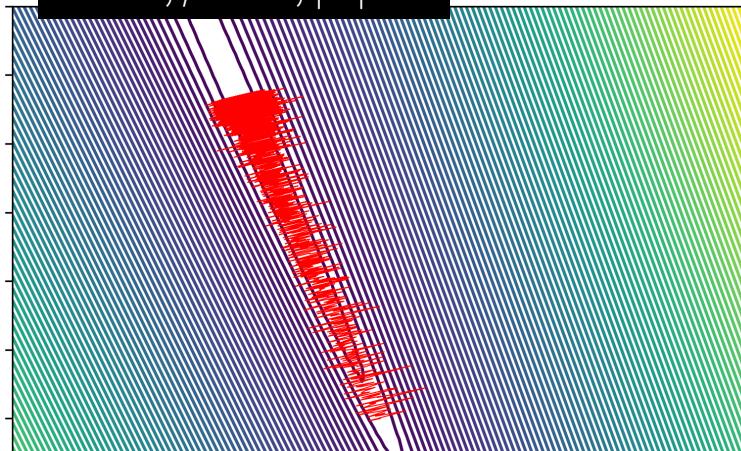
```
1 def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8, beta=.99):
2     N,D = X.shape
3     w = np.zeros(D)
4     g = np.inf
5     dw = 0
6     while np.linalg.norm(g) > eps:
7         minibatch = np.random.randint(N, size=(bsize))
8         g = gradient(X[minibatch,:], y[minibatch], w)
9         dw = (1-beta)*g + beta*dw
10        w = w - lr*dw
11    return w
12
```

Momentum

Example: logistic regression

no momentum

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

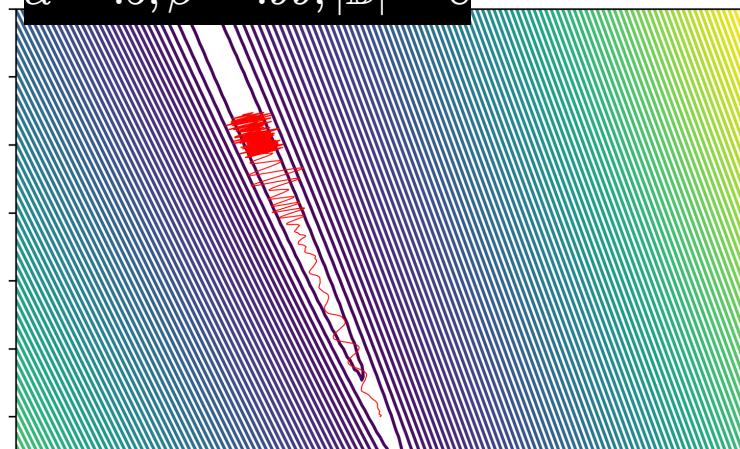


see the beautiful demo at Distill
<https://distill.pub/2017/momentum/>

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

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

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



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

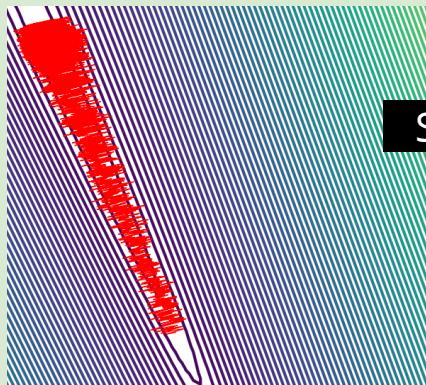
ϵ is to avoid numerical issues

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

Adagrad (**Adaptive gradient**)

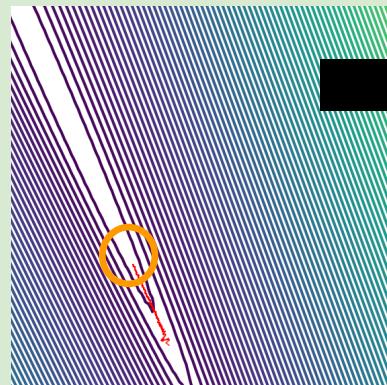
different learning rate for each parameter w_d
make the learning rate adaptive

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



SGD

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



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



```
1 def RMSprop(X, y, lr=.01, eps=1e-2, bsize=8, gamma=.9, epsilon=1e-8):
2     N,D = X.shape
3     w = np.zeros(D)
4     g = np.inf
5     S = 0
6     while np.linalg.norm(g) > eps:
7         minibatch = np.random.randint(N, size=(bsize))
8         g = gradient(X[minibatch,:], y[inibatch], w)
9         S = (1-gamma)*g**2 + gamma*S
10        w = w - lr*g/np.sqrt(S + epsilon)
11    return w
12
```

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**:

$$M^{\{t\}} \leftarrow \beta_1 M^{\{t-1\}} + (1 - \beta_1) \nabla J(w^{\{t-1\}}) \quad \begin{array}{l} \text{identical to method of momentum} \\ \text{(moving average of the first moment)} \end{array}$$

$$S^{\{t\}} \leftarrow \beta_2 S^{\{t-1\}} + (1 - \beta_2) \nabla J(w^{\{t-1\}})^2 \quad \begin{array}{l} \text{identical to RMSProp} \\ \text{(moving average of the second moment)} \end{array}$$

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

Adam (Adaptive Moment Estimation)

Adam combines thee two:

$$M^{\{t\}} \leftarrow \beta_1 M^{\{t-1\}} + (1 - \beta_1) \nabla J(w^{\{t-1\}}) \quad \begin{array}{l} \text{identical to method of momentum} \\ (\text{moving average of the first moment}) \end{array}$$

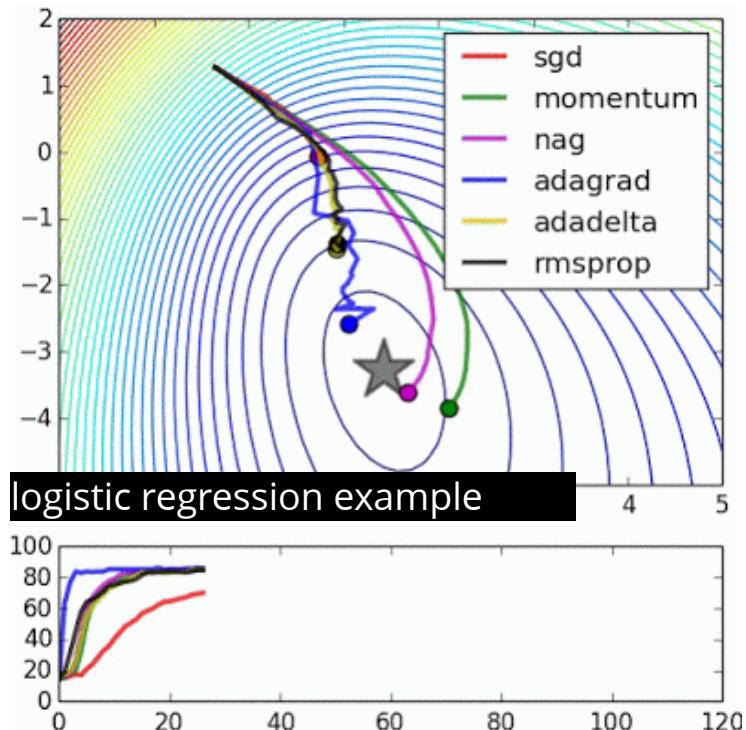
$$S^{\{t\}} \leftarrow \beta_2 S^{\{t-1\}} + (1 - \beta_2) \nabla J(w^{\{t-1\}})^2 \quad \begin{array}{l} \text{identical to RMSProp} \\ (\text{moving average of the second moment}) \end{array}$$

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

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

$$\hat{M}^{\{t\}} \leftarrow \frac{M^{\{t\}}}{1-\beta_1^t} \quad \hat{S}^{\{t\}} \leftarrow \frac{S^{\{t\}}}{1-\beta_2^t} \quad \begin{array}{l} \text{for large time-steps it has no effect} \\ \text{for small t, it scales up numerator} \end{array}$$

In practice



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

image:Alec Radford

Adding L_2 regularization

do not penalize the bias w_0

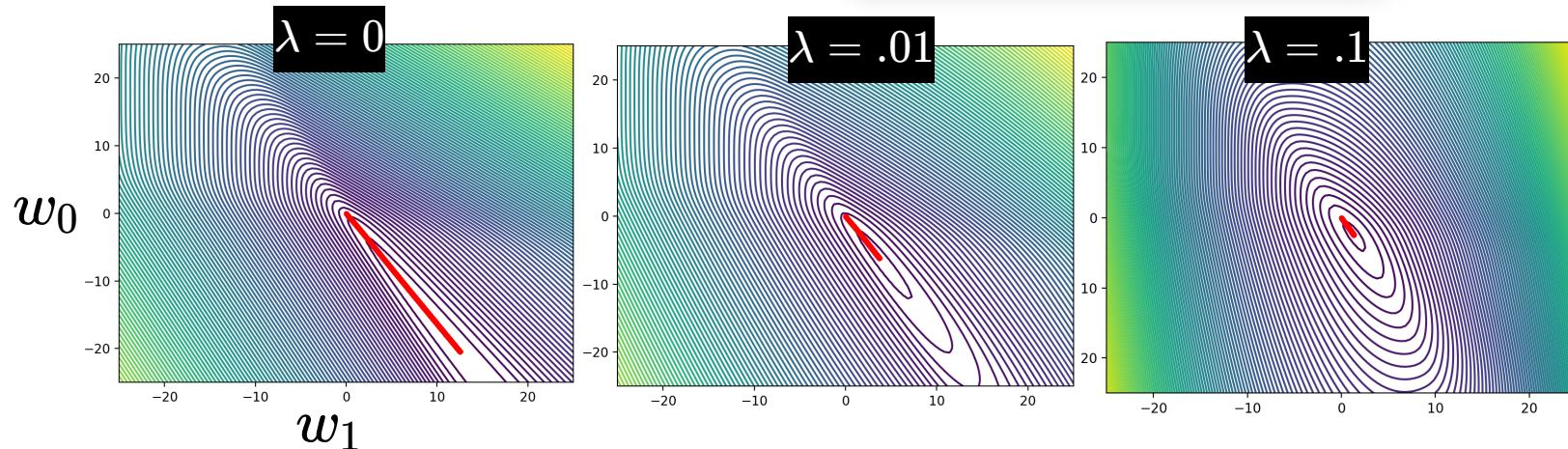
L2 penalty makes the optimization easier too!

note that the optimal w_1 shrinks

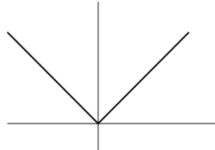


```
1 def gradient(X, y, w, lambdaa):  
2     N,D = X.shape  
3     yh = logistic(np.dot(X, w))  
4     grad = np.dot(X.T, yh - y) / N  
5     grad[1:] += lambdaa * w[1:]  
6     return grad
```

weight decay



Subderivatives

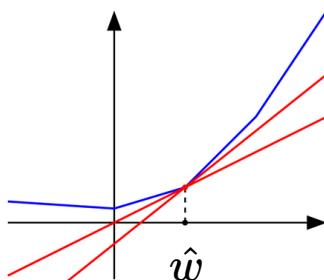


L1 penalty is no longer smooth or differentiable (at 0)
extend the notion of derivative to non-smooth functions

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

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

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

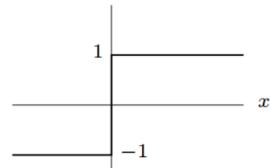
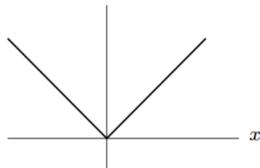


another expression for sub-differential
$$\partial f(\hat{w}) = \{g \in \mathbb{R} \mid f(w) > f(\hat{w}) + g(w - \hat{w})\}$$

Subgradient

example

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



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

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

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

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

subdifferential for functions of multiple variables

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

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

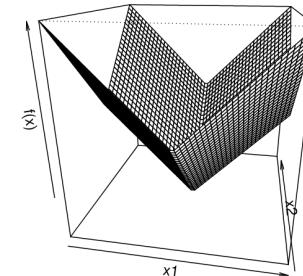


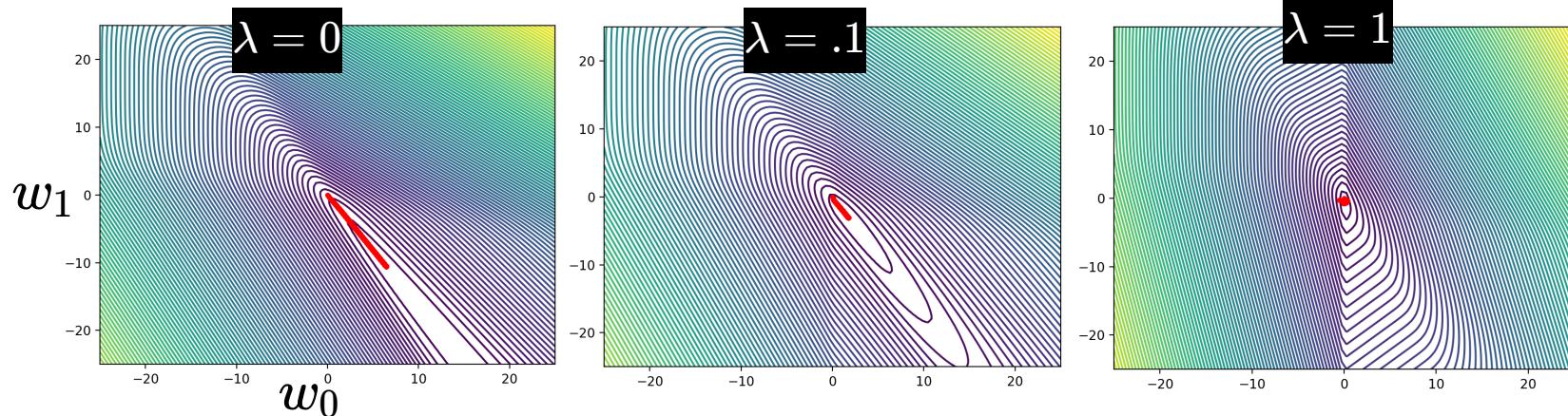
image credit: G. Gordon

Adding L_1 regularization

L1-regularized *linear regression* has efficient solvers
subgradient method for L1-regularized logistic regression
do not penalize the bias w_0
using **diminishing learning rate**
note that the optimal w_1 **becomes 0**



```
1 def gradient(X, y, w, Lambda):
2     N,D = X.shape
3     yh = logistic(np.dot(X, w))
4     grad = np.dot(X.T, yh - y) / N
5     grad[1:] += Lambda * np.sign(w[1:])
6     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

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