Applied Machine Learning

Gradient Descent Methods

Reihaneh Rabbany

McGill School of Computer Science

Learning objectives

Basic idea of

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

Application to

- linear regression and classification

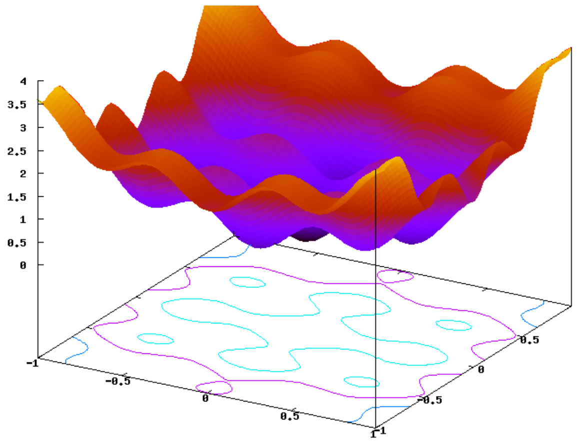
Optimization in ML

The core problem in ML is parameter estimation (aka model fitting), which requires solving an optimization problem of the loss/cost function

Optimization is a huge field

bold marks the settings we consider 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



Optimization in ML

x | input features → **ML algorithm** **with parameters w** $f(x; w)$ → y | output labels

The core problem in ML is parameter estimation (aka model fitting), which requires solving an optimization problem of the loss/cost function

$$J(w) = \frac{1}{N}\sum_{n=1}^{N} l(y^{(n)}, f(x^{(n)}; w))$$

$$w^* = \arg\min_w J(w)$$

Recall

Linear Regression:

model: $\hat{y} = f_w(x) = w^\top x \ : \mathbb{R}^D \to \mathbb{R}$

cost function: $J_w = \frac{1}{N}\sum_n \frac{1}{2}(y^{(n)} - \hat{y}^{(n)})^2$

Logistic Regression:

model: $\hat{y} = f_w(x) = \sigma(w^\top x) : \mathbb{R}^D \to \{0,1\}$

cost function: $J_w = \frac{1}{N}\sum_n -y\log(\hat{y}^{(n)}) - (1 - y^{(n)})\log(1 - \hat{y}^{(n)})$

partial derivatives: $\frac{\partial}{\partial w_d} J_w = \frac{1}{N}\sum_n (\hat{y}^{(n)} - y^{(n)})x_d^{(n)}$

gradient: vector of all partial derivatives: $\nabla J(w) = \frac{1}{N}\sum_n (\hat{y}^{(n)} - y^{(n)})x^{(n)}$

Dx1

how to find w^* given $\nabla J(w)$?

Gradient

Recall

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

partial derivatives instead of derivative

= derivative when other vars. are fixed

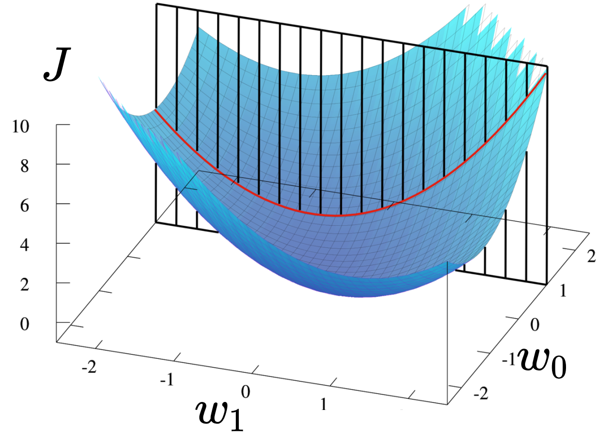
$$\frac{\partial}{\partial w_1} J(w_0, w_1) \triangleq \lim_{\epsilon \to 0} \frac{J(w_0, w_1 + \epsilon) - J(w_0, w_1)}{\epsilon}$$

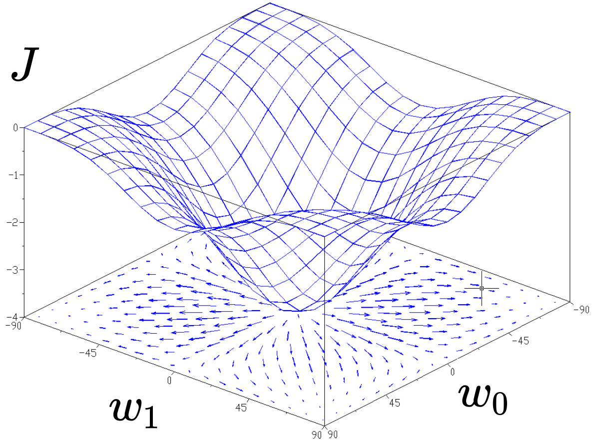
we can estimate this numerically if needed

(use small epsilon in the 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 $w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla J(w^{\{t\}})$

new notation!

steepest descent direction

learning rate

cost function

(for maximization : objective function)

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

converges to a local minima

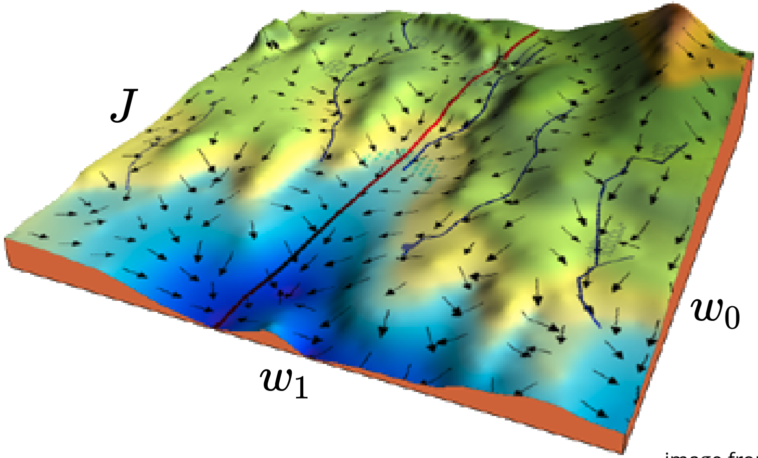
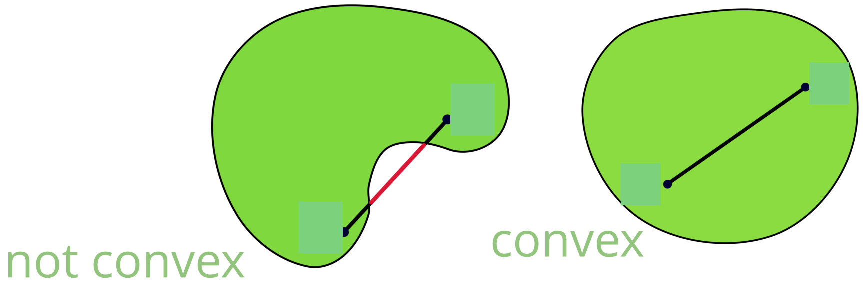


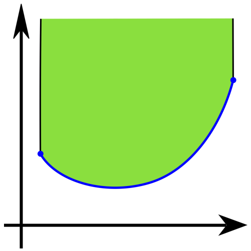
image from here

Convex function

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

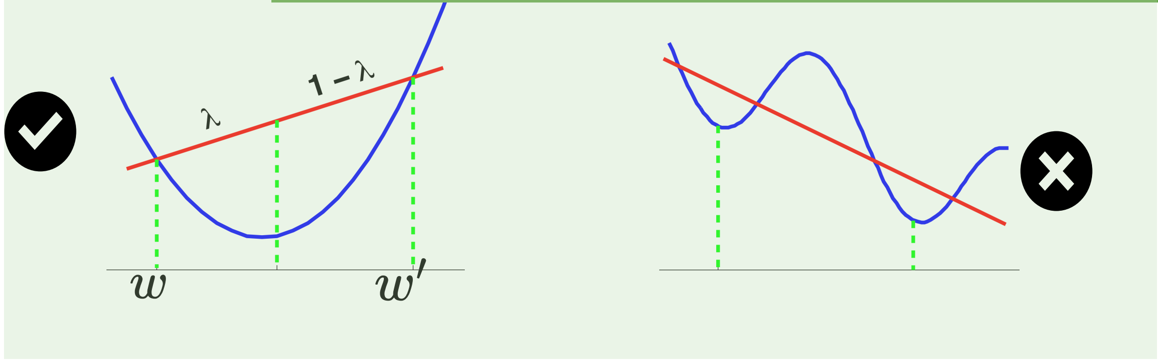


a **convex function** is a function for which the *epigraph* is a **convex set**



epigraph: set of all points above the graph

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



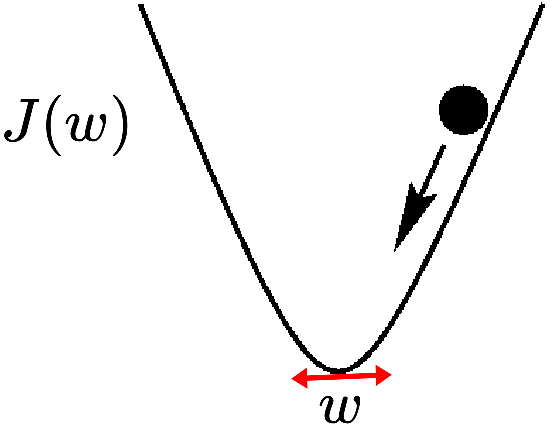
Minimum of a convex function

Convex functions are easier to minimize:

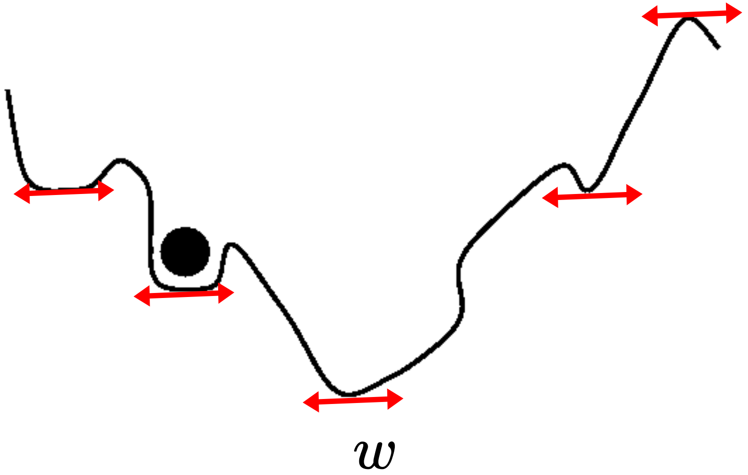
- critical points are global minimum
- gradient descent can find it

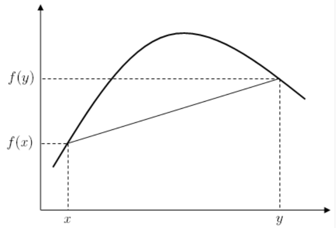
$$w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla J(w^{\{t\}})$$

convex



non-convex: gradient descent may find a local optima





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

image from here

Recognizing convex functions

a constant function is convex $f(x) = c$

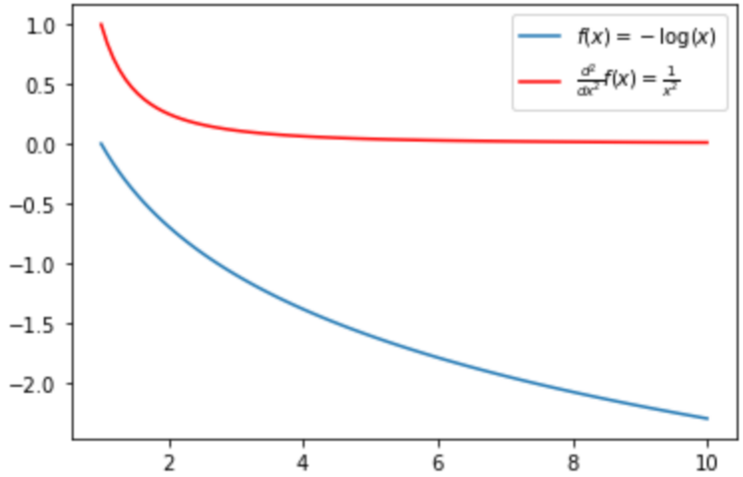
a linear function is convex $f(x) = w^\top x$

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

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

$x \log(x), x > 0$

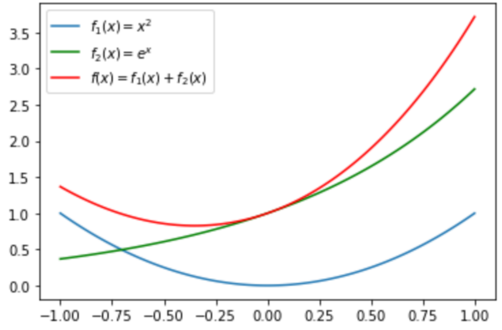
$x^a, x > 0, a > 1$



Recognizing convex functions

sum of convex functions is convex

example 1:



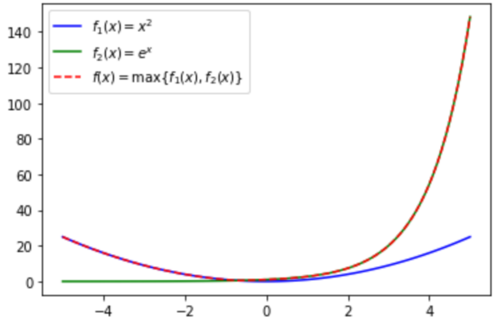
example 2:

sum of squared errors

$$J(w) = ||Xw - y||_2^2 = \sum_n (w^\top x^{(n)} - y)^2$$

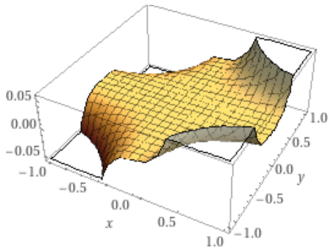
maximum of convex functions is convex

example 1:



example 2:

$$f(y) = \max_{x\in[0,2]} x^3y^4 = 9y^4$$

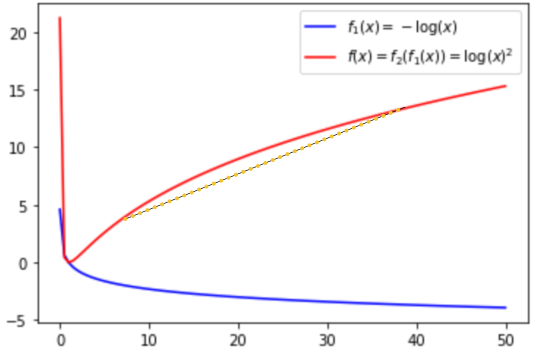


note this is not convex in x

Recognizing convex functions

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

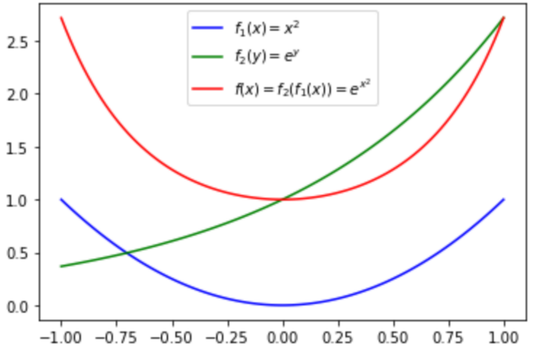
example $$(-\log(x))^2$$



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

example $$e^{f(x)}$$

for convex ***f***



Composition with affine map (linear function) is also convex, e.g. $f(w^\top x - y)$ if f is convex

Recognizing convex functions

is the logistic regression cost function convex in model parameters (w)?

$$J(w) = \frac{1}{N} \sum_{n=1}^{N} y^{(n)} \log\left(1 + e^{-w^\top x}\right) + \left(1 - y^{(n)}\right) \log\left(1 + e^{w^\top x}\right)$$

same argument

non-negative

linear

checking second derivative

$$\frac{\partial^2}{\partial z^2} \log(1 + e^{z}) = \frac{e^{-z}}{(1+e^{-z})^2} \geq 0$$

sum of convex functions

Gradient for linear and logistic regression

recall

in both cases: $\nabla J(w) = \frac{1}{N} \sum_n x^{(n)}(\hat{y}^{(n)} - y^{(n)}) = \frac{1}{N} \underset{D \times N}{X^\top} (\underset{N \times 1}{\hat{y}} - \underset{N \times 1}{y})$

linear regression: $\hat{y} = w^\top x$

logistic regression: $\hat{y} = \sigma(w^\top x)$

```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!

```
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

```

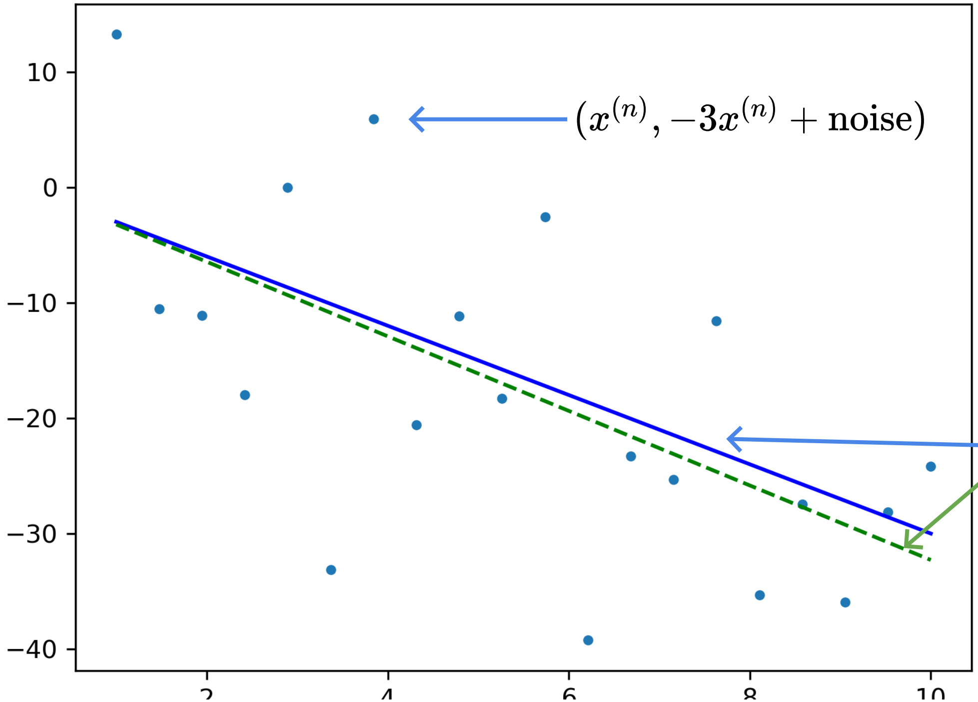
code on the previous page

Some **termination condition**:

- 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



using direct solution method

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

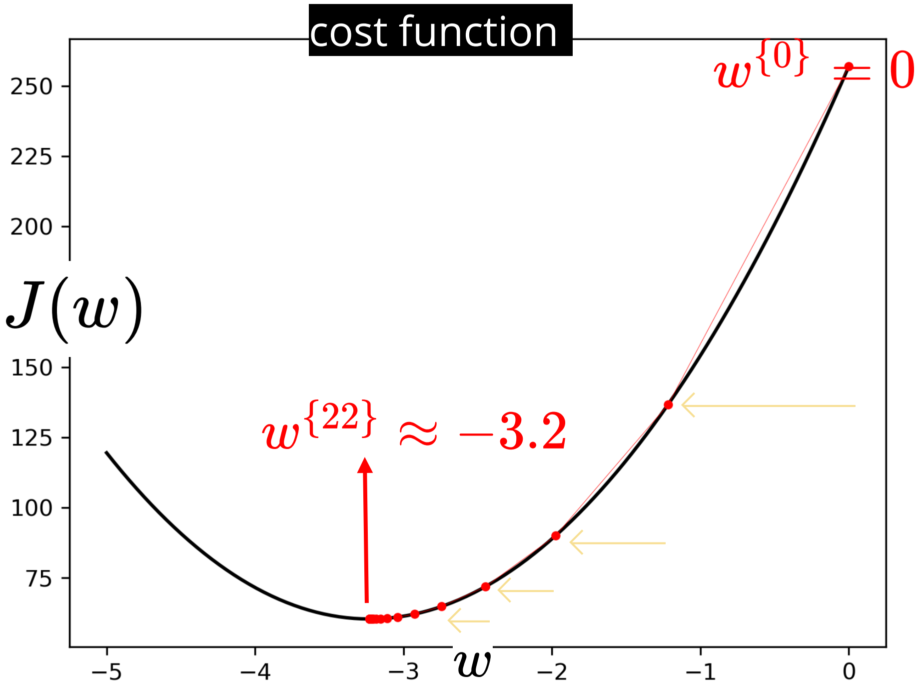
$$y = wx$$

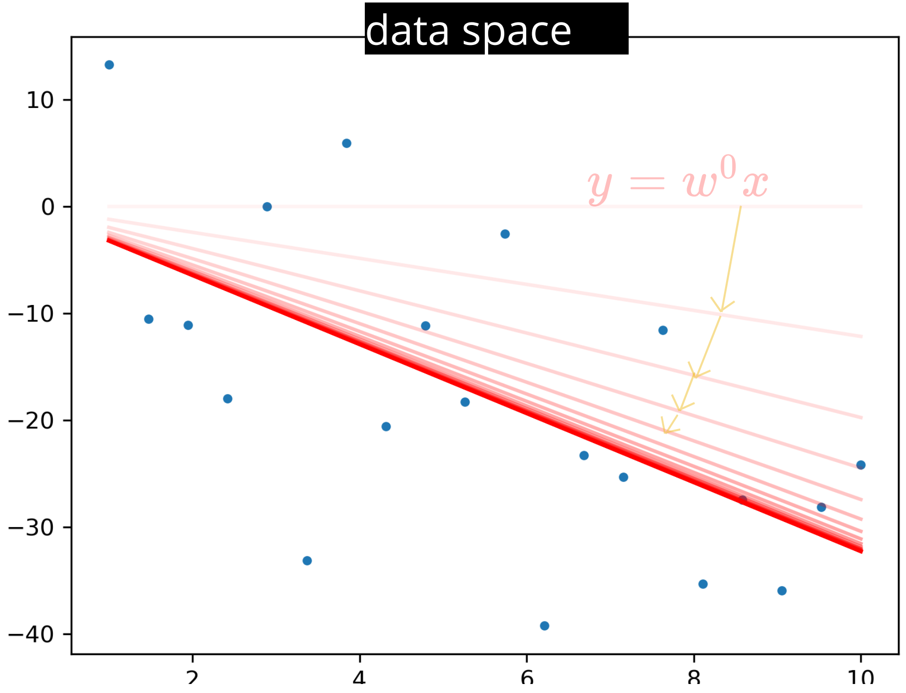
$$y = -3x$$

example

GD for linear regression

After 22 steps $w^{\{t+1\}} \leftarrow w^{\{t\}} - .01 \nabla J(w^{\{t\}})$

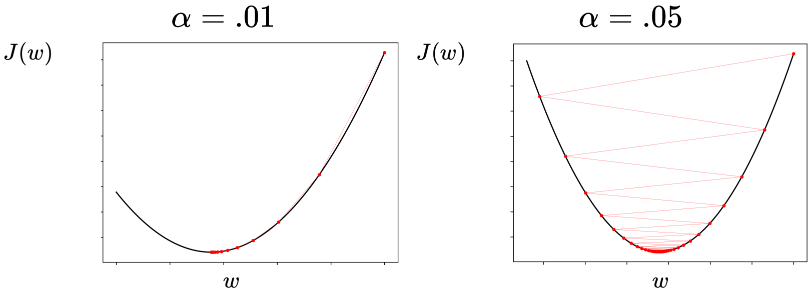




Learning rate α

Learning rate has a significant effect on GD

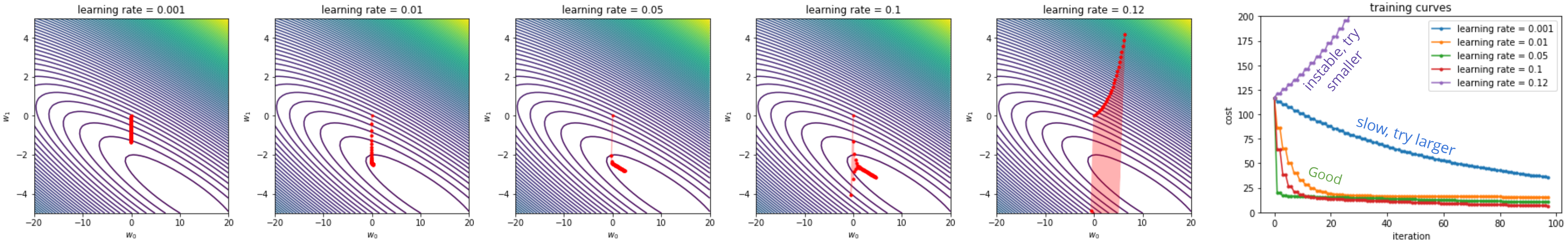
example, D=1
linear regression



too **small**: may take a long time to converge

too **large**: it overshoots or even diverges

example, D=2
linear regression
50 gradient steps



do a grid search usually between 0.001 to .1 to find the right value, look at the training curves

Stochastic Gradient Descent

we can write the cost function as an 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 $\nabla J(w) = \mathbb{E}_{\mathcal{D}}[\nabla J_n(w)]$

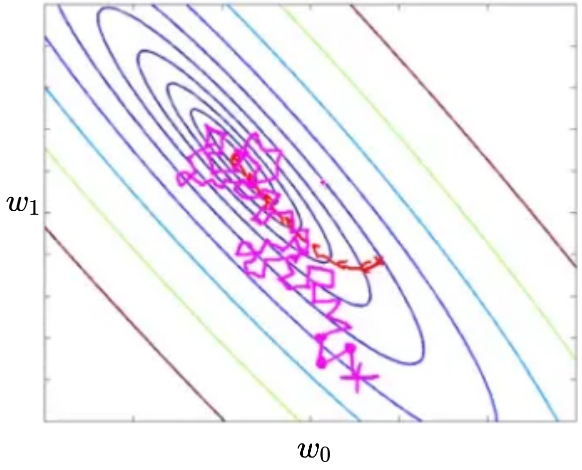
Stochastic Gradient Descent

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

stochastic gradient update

$$w \leftarrow w - \alpha \nabla J_n(w)$$

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

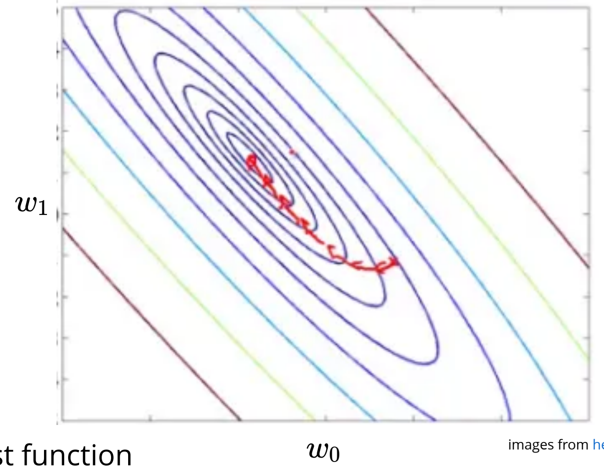
e.g., for linear regression $\mathcal{O}(D)$

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

batch gradient update

$$w \leftarrow w - \alpha \nabla J(w)$$

with small learning rate: guaranteed improvement at each step



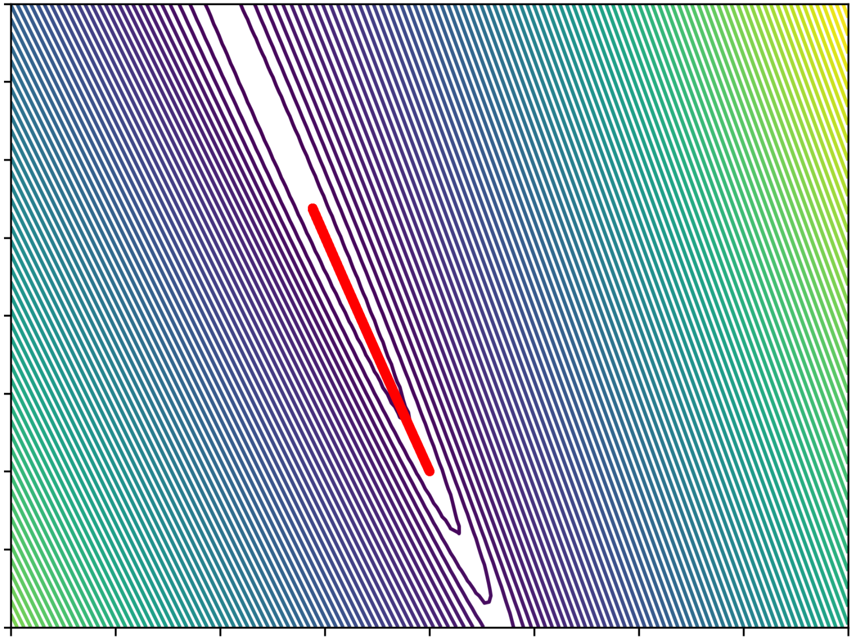
contour plot of the cost function

images from here

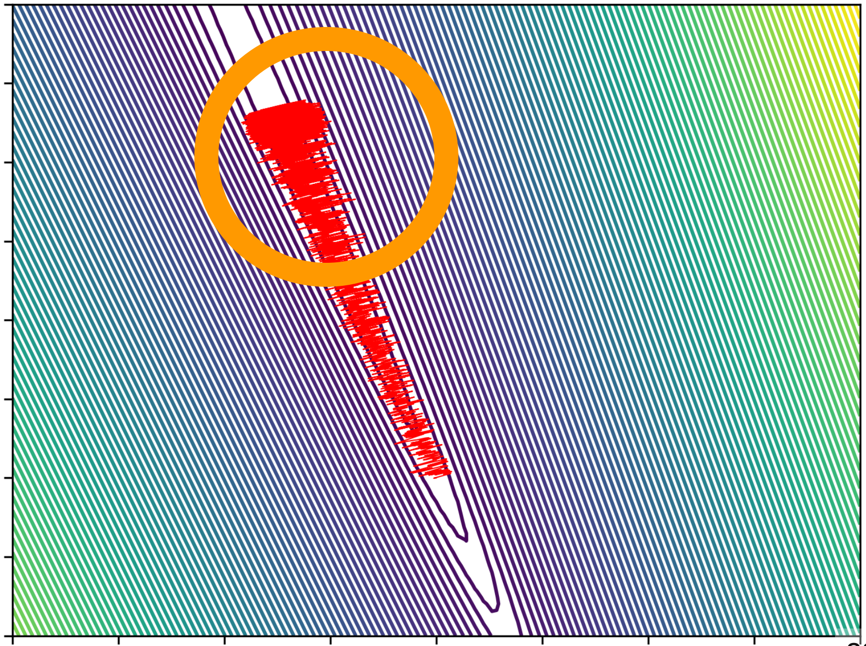
SGD for logistic regression example

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

batch gradient



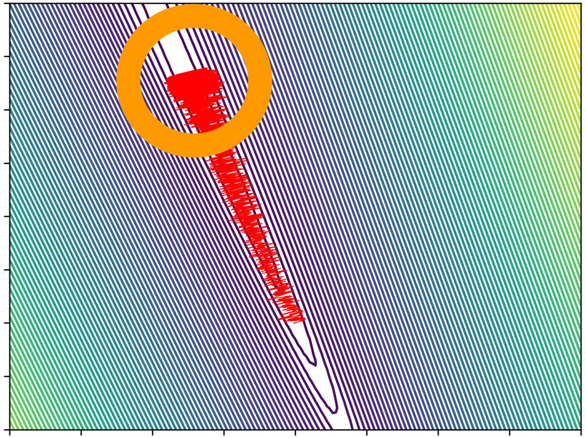
stochastic gradient



Convergence of SGD

stochastic gradients are not zero even at the optimum w

how to guarantee convergence?



idea: schedule to have a smaller learning rate over time

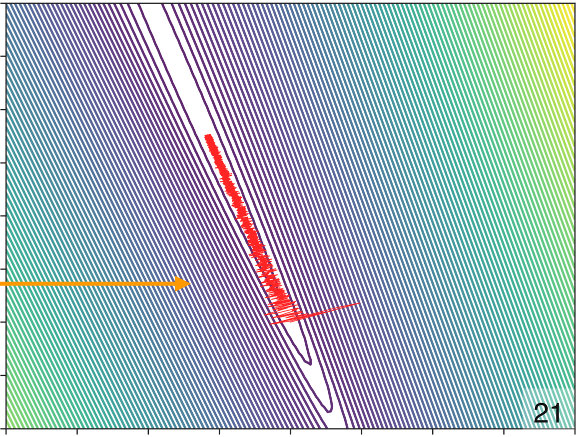
Robbins Monro

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

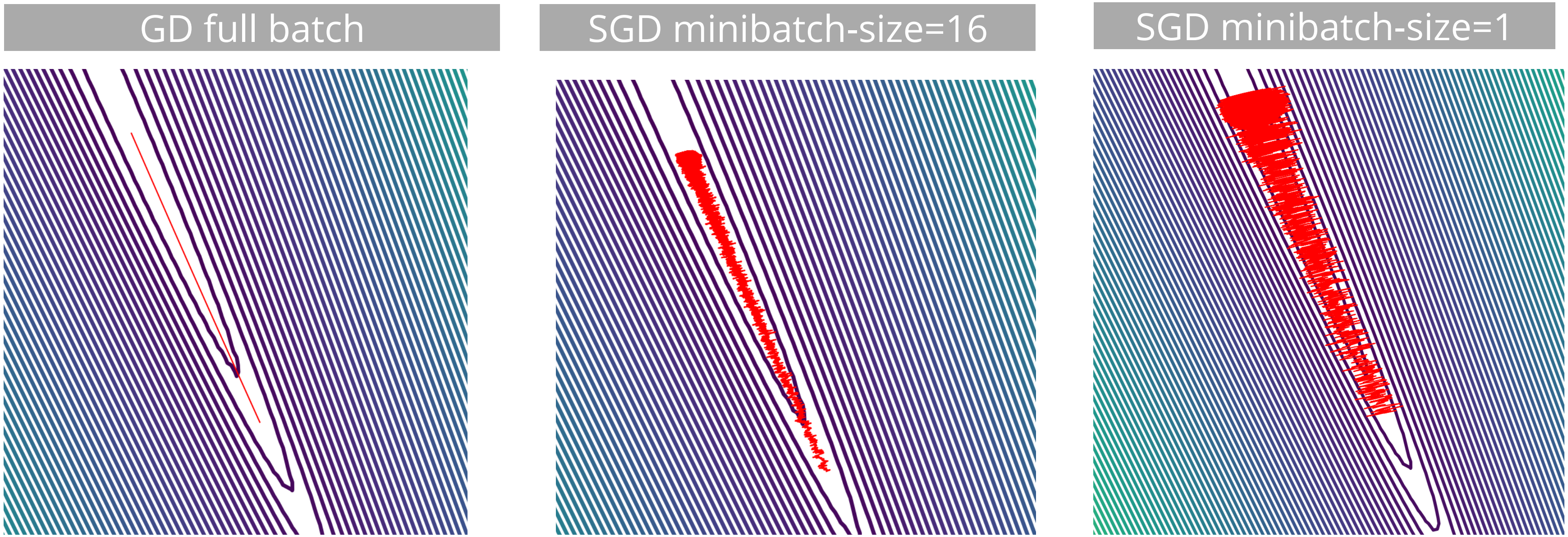


Minibatch SGD

use a minibatch to produce gradient estimates

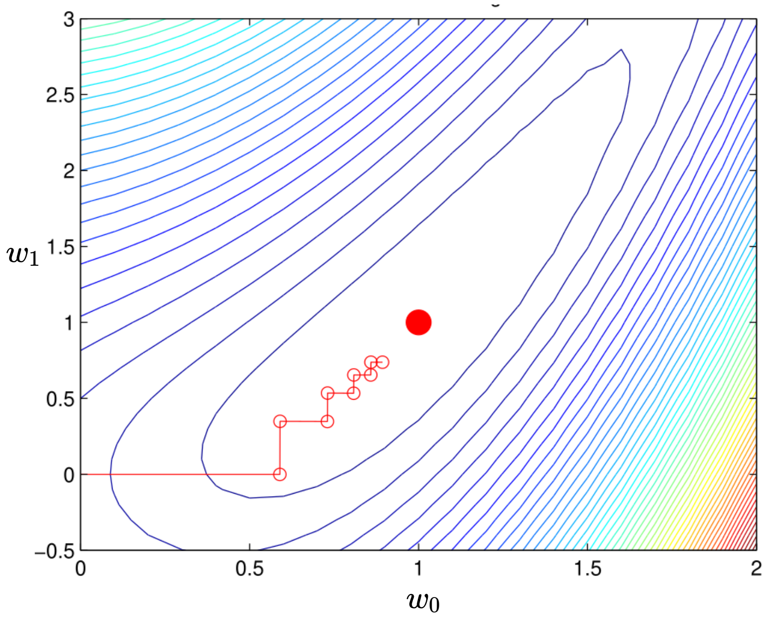
$$\nabla J_{\mathbb{B}} = \frac{1}{|\mathbb{B}|} \sum_{n \in \mathbb{B}} \nabla J_n(w)$$

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

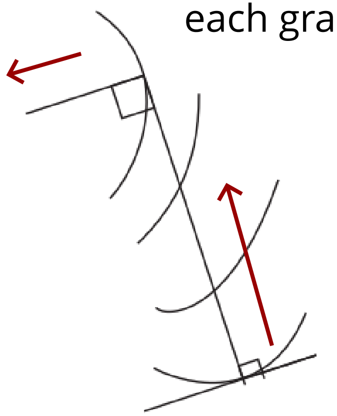


Oscillations

gradient descent can oscillate a lot!

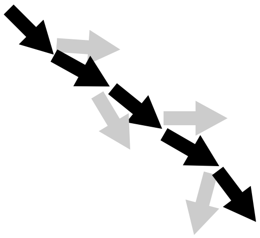


each gradient step is prependicular to isocontours



in SGD this is worsened due to *noisy* gradient estimate

Momentum



to help with oscillations:

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

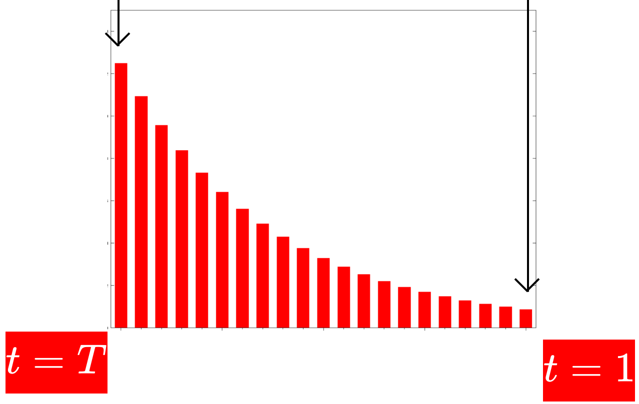
$$\Delta w^{\{t\}} \leftarrow \beta \Delta w^{\{t-1\}} + (1-\beta)\nabla J_{\mathbb{B}}(w^{\{t-1\}})$$

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

momentum of 0 reduces to SGD
common value > .9

weight for the **most recent** gradient $(1-\beta)$

weight for the oldest gradient $(1-\beta)\beta^{T-1}$



is effectively an exponential moving average

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

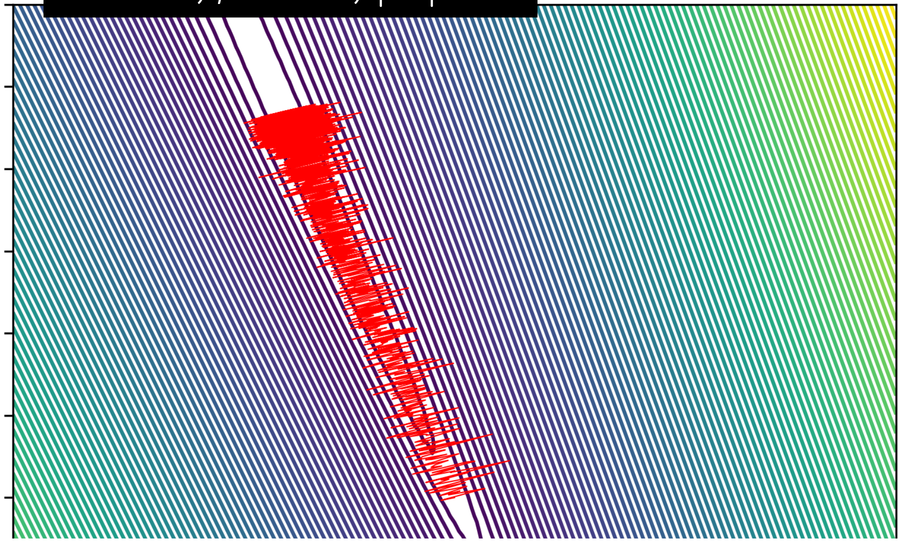
there are other variations of momentum with similar idea

Momentum

Example: logistic regression

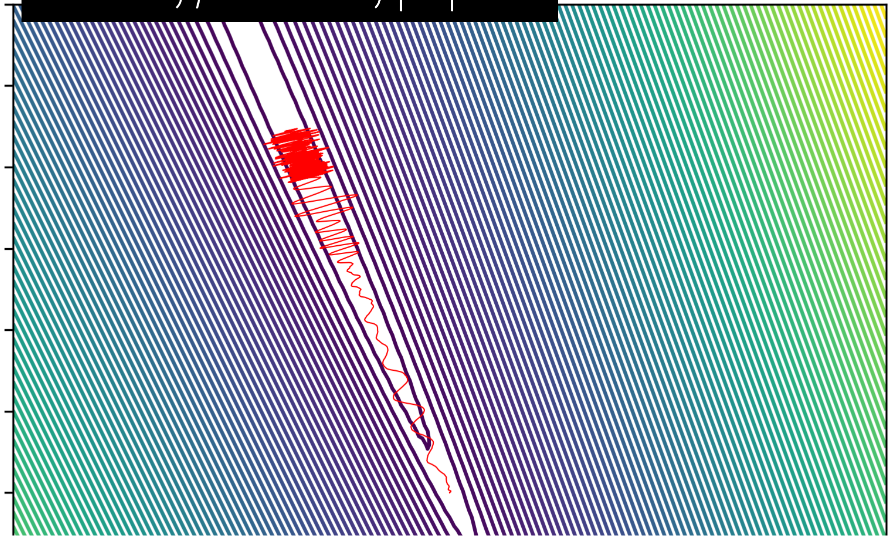
no momentum

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



with momentum

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



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**

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

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

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

the learning rate is adapted to previous updates

ϵ is to avoid numerical issues

useful when parameters are updated at different rates

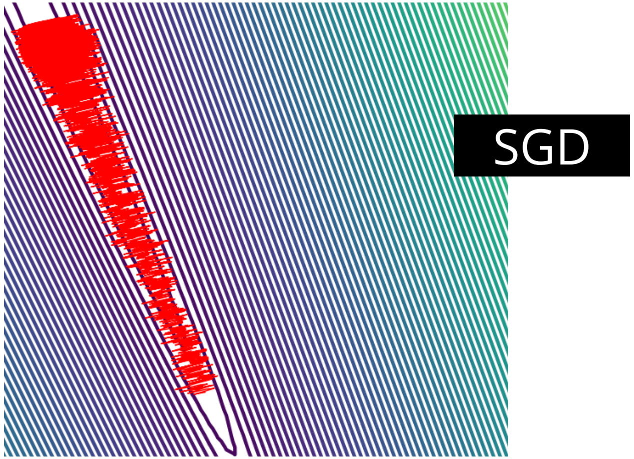
(e.g., sparse data when some features are often zero when using SGD)

Adagrad (Adaptive gradient)

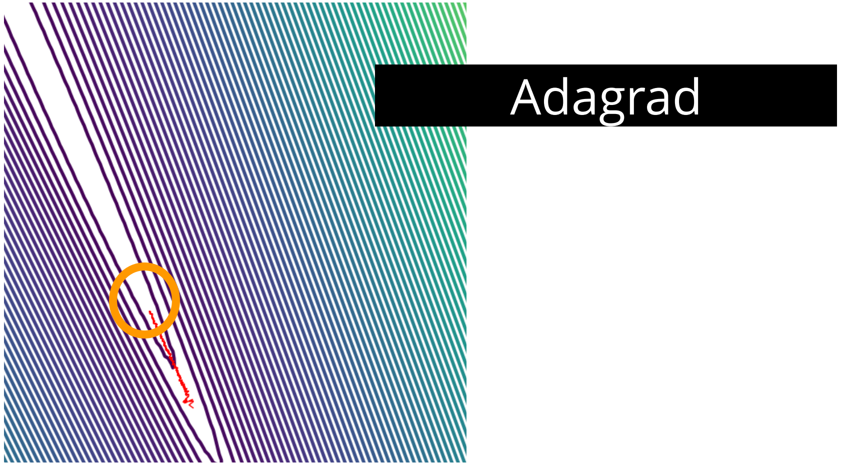
different learning rate for each parameter w_d

make the learning rate adaptive

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



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



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)

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

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

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

identical to Adagrad

note that $S^{\{t\}}$ here is a vector and with the square root is element-wise

Adam (Adaptive Moment Estimation)

the default algorithm in practice

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)

$$w^{\{t\}} \leftarrow w^{\{t-1\}} - \frac{\alpha}{\sqrt{\hat{S}^{\{t\}} + \epsilon}} \hat{M}^{\{t\}}$$

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

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

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

In practice

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

logistic regression example

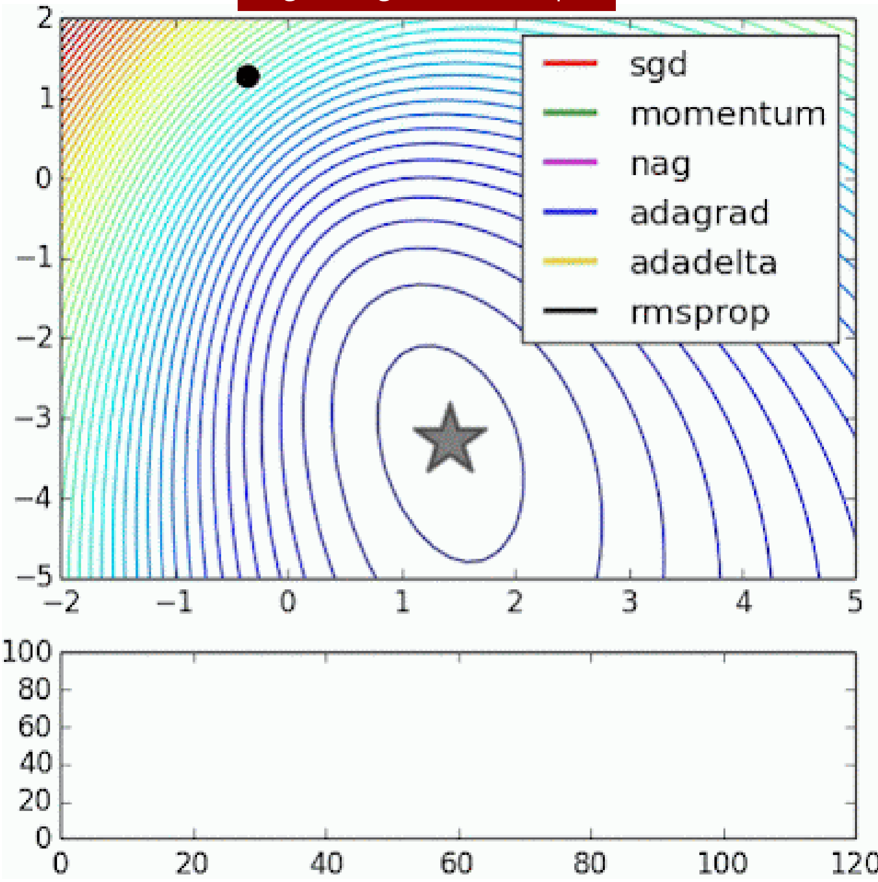


image:Alec Radford

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