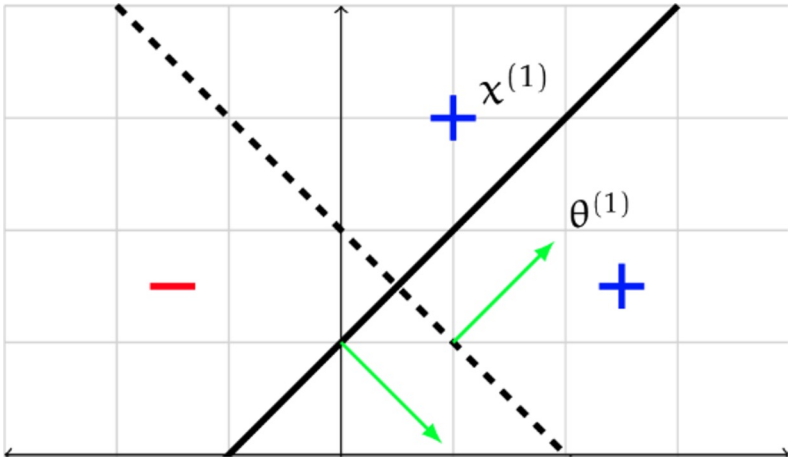
Spring 2023!

Introduction to Machine Learning



Lec-Rec 3: Gradient Descent

Review: Linear Regression Solution

$$\mathcal{E}_n(h) = J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 = \frac{1}{n}\underbrace{(\tilde{X}\theta - \tilde{Y})^T}_{1\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1}$$

$$\nabla_\theta J = \frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) = 0$$

$$\tilde{X}^T\tilde{X}\theta - \tilde{X}^T\tilde{Y} = 0$$

$$\tilde{X}^T\tilde{X}\theta = \tilde{X}^T\tilde{Y}$$

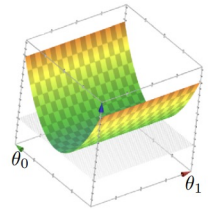
$$\theta = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T\tilde{Y}$$

Linear Regression Cost Function

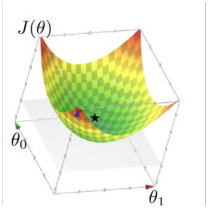
$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

We are incredibly fortunate that:

- The cost function is Convex (“curves up”)
  ⇒ local minimum is a global minimum



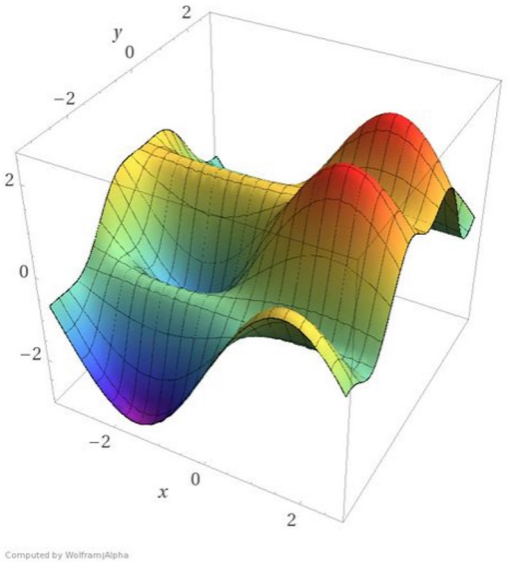
- It can be efficiently computed
  ⇒ there is a closed-form solution to $\nabla_\theta J = 0$



$$\theta = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T\tilde{Y}$$

Optimization is hard in general!

- Typical cost function



$$\nabla_\theta J = 0$$

Stationary points are not global (or even local) minimma

Gradient Descent

First-order, iterative algorithm for finding a local minimum of a differentiable function

- First-order: using only gradient information
- Iterative algorithm: iteratively update parameters
- Differentiable function: continuous optimization

Gradient Descent

Initialize $\theta^{(0)}$ (typically random)

Repeat:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

Until convergence

$$|J(\theta^{(t+1)}) - J(\theta^{(t)})| < \epsilon$$

Other stopping criteria: fixed # of iters o $J(\theta^{(t)})$ | close to 0

Gradient Descent

Initialize $\theta^{(0)}$ (typically random)

Repeat:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

η is called a learning rate or step size

Until convergence

$$|J(\theta^{(t+1)}) - J(\theta^{(t)})| < \epsilon$$

Other stopping criteria: fixed # of iters o $J(\theta^{(t)})$ | close to 0

Gradient Descent

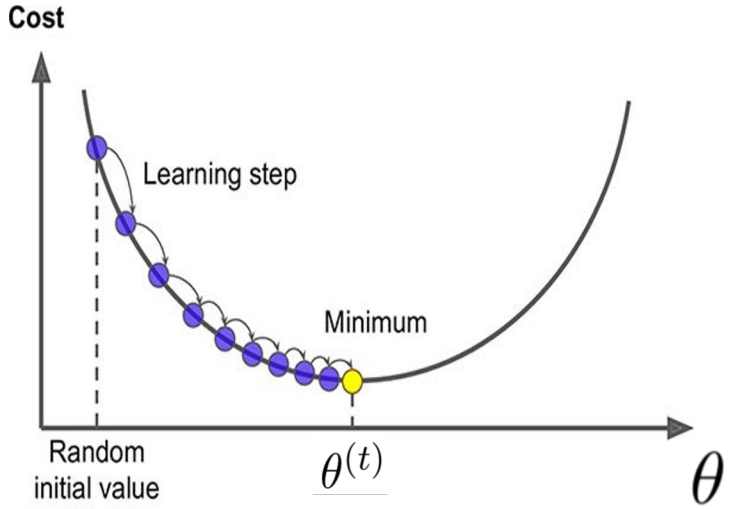
Initialize $\theta^{(0)}$ (typically random)

Repeat:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

Until convergence

$$|J(\theta^{(t+1)}) - J(\theta^{(t)})| < \epsilon$$



Other stopping criteria: fixed # of iters o $J(\theta^{(t)})$ | close to 0

Gradient Descent

Initialize $\theta^{(0)}$ (typically random)

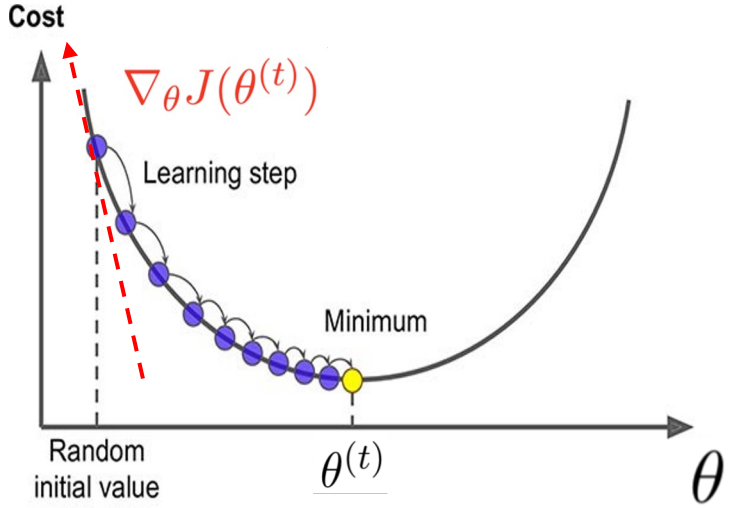
Repeat:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

Until convergence

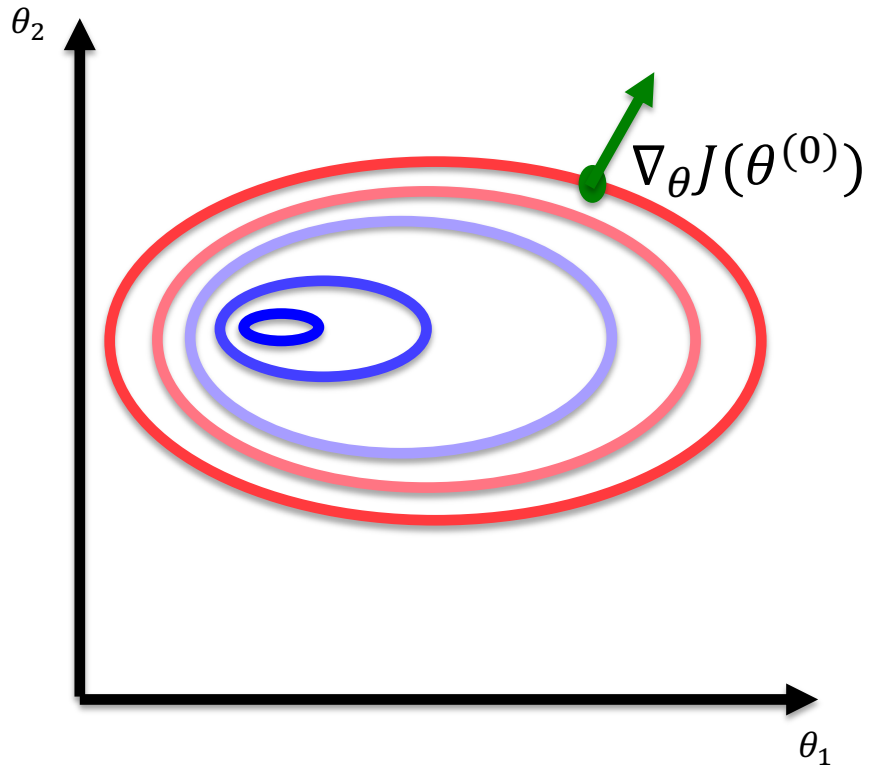
$$|J(\theta^{(t+1)}) - J(\theta^{(t)})| < \epsilon$$

Gradients point in the steepest direction => negative gradient takes a step opposite this direction

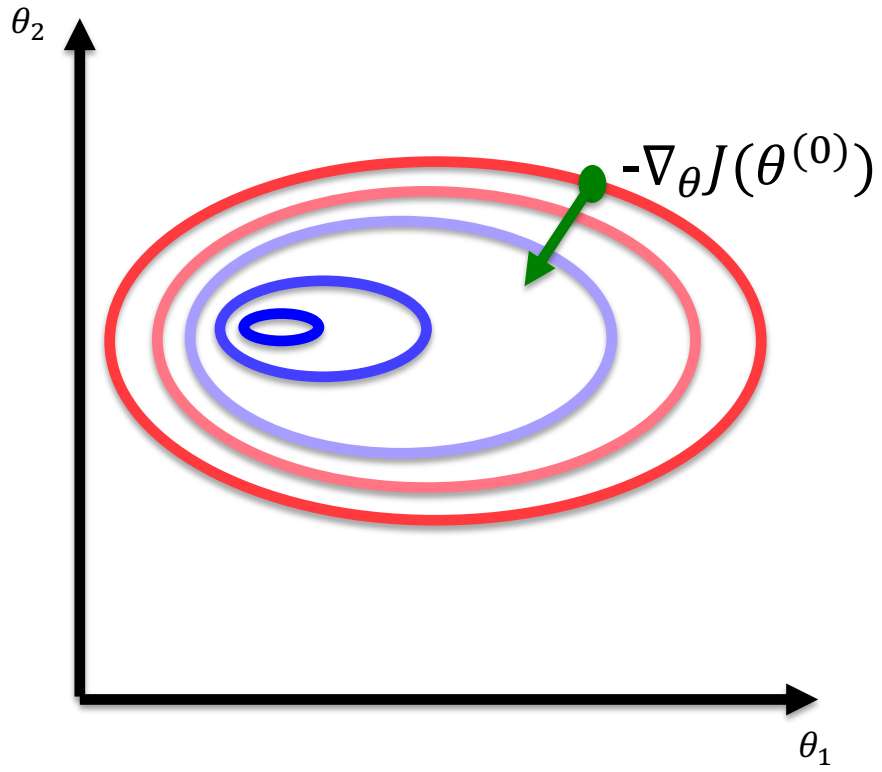


Other stopping criteria: fixed # of iters o $J(\theta^{(t)})$ | close to 0

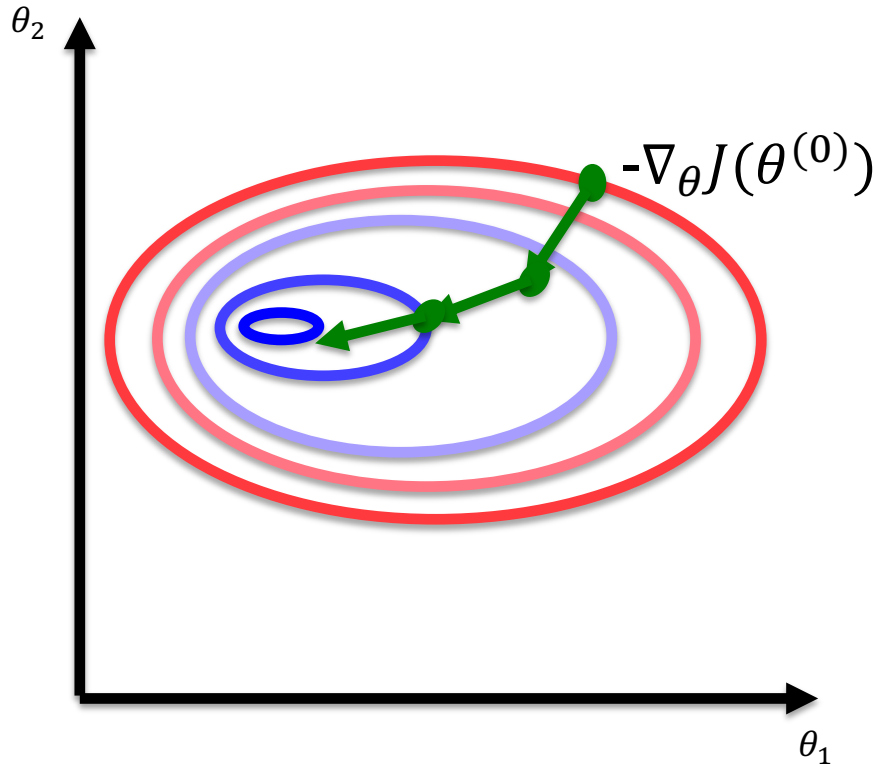
Gradient Descent in 2D



Gradient Descent in 2D



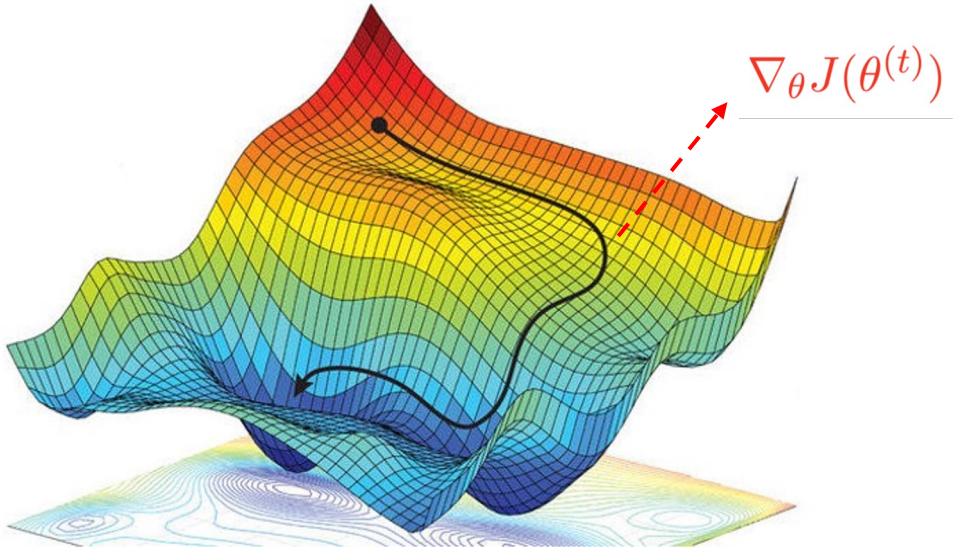
Gradient Descent in 2D



Gradient Descent for Non-Convex Functions

Cost function does not need to be convex

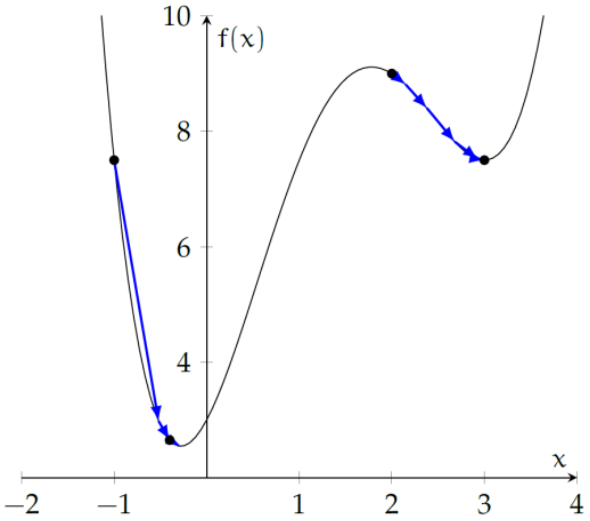
But for non-convex functions we often get stuck in a local minimum



Starting point matters

Starting point determines in which local minimum we end up

We can run gradient descent with different starting points and take the best solution



How to computer gradient?

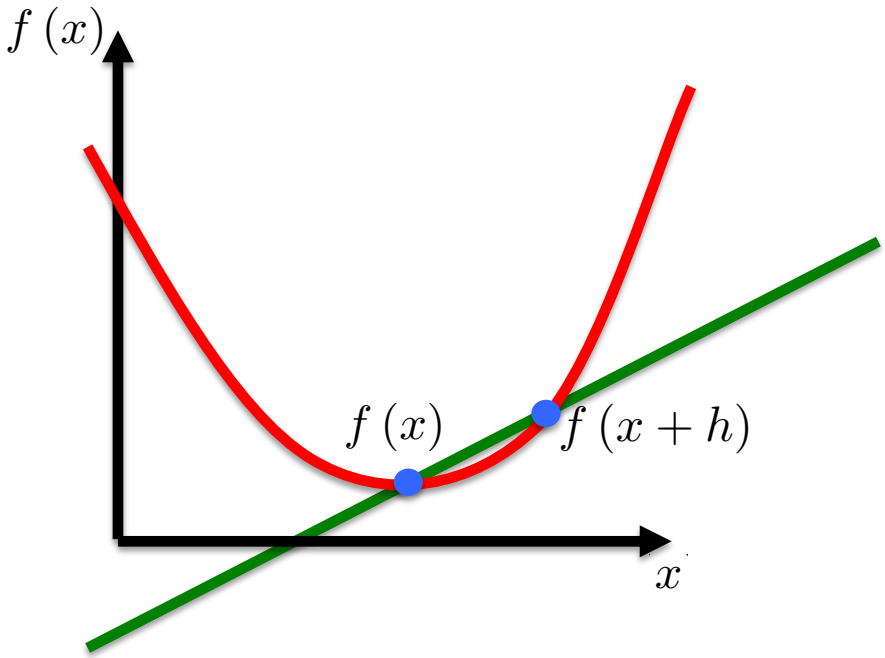
Computing gradient

Analytically (pencil/paper, Mathematica)

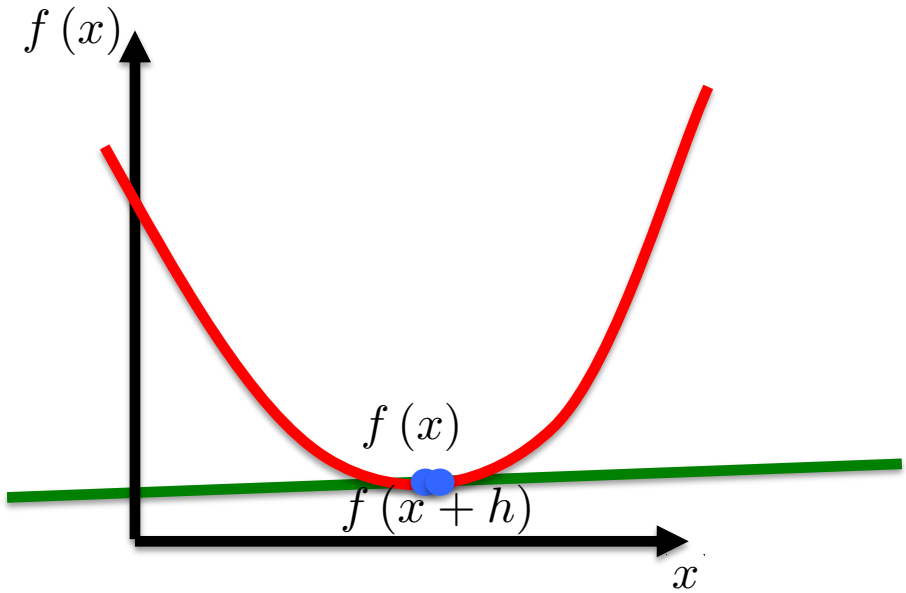
Finite differences

Automated differentiation

Computing gradient requires a limit



Computing gradient requires a limit



Finite differences

In finite differencing we choose h and evaluate $\frac{f(x+h)-f(x)}{h}$ numerically

Software like Matlab computes h automatically

Finite differencing is useful to verify correctness of analytical derivatives (yes, we often make mistakes in computing derivatives)

Expensive for high dimensional functions (e.g., how many function evaluation are required when working with d features?)

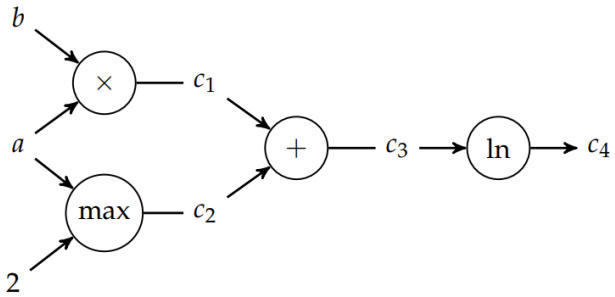
Automated differentiation

The chain rule:

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx}(f \circ g)(x) = \frac{df}{dg}\frac{dg}{dx}$$

Computational graph

- nodes are operations and edges are input relations
- forward or reverse accumulation



See Autograd

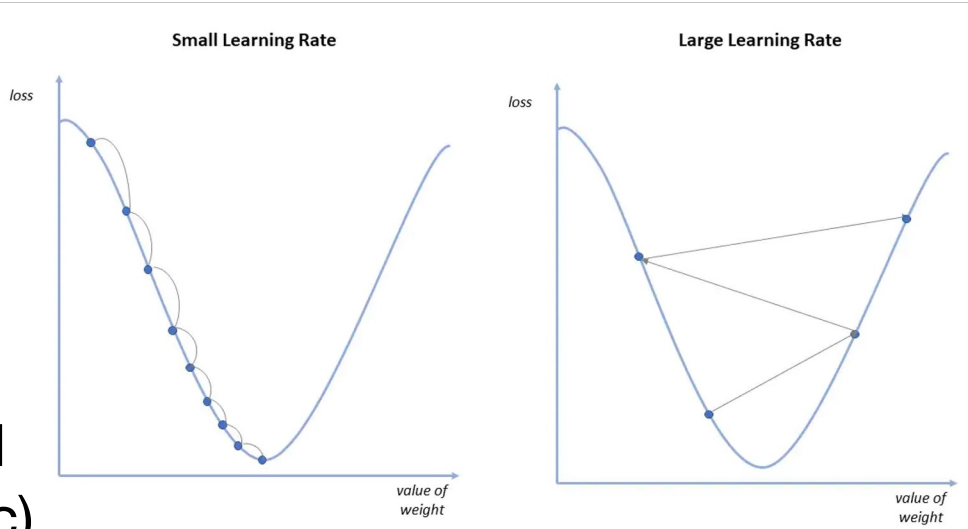
Gradient Descent

How to set the learning rate?

It can treated as another hyperparameter

$$\eta \in \{0.001, 0.01, 0.1\}$$

$$\eta_t = \frac{1}{t}$$



More advanced methods used in practice (soon in NN LecRec)

Gradient Descent for Linear Regression

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2$$

$$\nabla_\theta J = \frac{2}{n}\underbrace{\tilde{X}^T}_{d\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} = \frac{2}{n}\sum_{i=1}^{n}\left(\left[\theta^{(t-1)}\right]^T x^{(i)} - y^{(i)}\right)x^{(i)}$$

$$\theta^{(t)} = \theta^{(t-1)} - \eta\frac{2}{n}\sum_{i=1}^{n}\left(\left[\theta^{(t-1)}\right]^T x^{(i)} - y^{(i)}\right)x^{(i)}$$

But closed form solution is more efficient!

Stochastic Gradient Descent

$$\theta^{(t)} = \theta^{(t-1)} - \eta \frac{2}{n} \sum_{i=1}^{n} \left(\left[\theta^{(t-1)} \right]^{T} x^{(i)} - y^{(i)} \right) x^{(i)}$$

This sum requires running the model for the entire dataset at each time step

Stochastic Gradient Descent: approximate the full gradient with a a single sample

Stochastic Gradient Descent

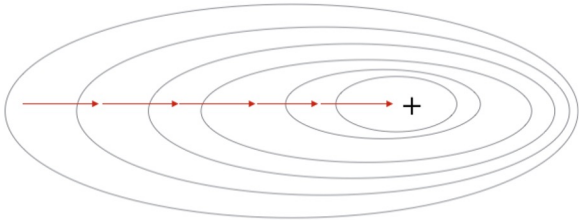
$$f(\Theta) = \sum_{i=1}^{n} f_i(\Theta)$$

$\Theta^{(0)} = \Theta_{init}$
for $t = 1$ **to** T
    randomly select $i \in \{1, 2, \ldots, n\}$
    $\Theta^{(t)} = \Theta^{(t-1)} - \eta(t) \nabla_{\Theta} f_i(\Theta^{(t-1)})$
return $\Theta^{(t)}$

With certain conditions on the learning rate, the model will get "close" to the optimum "fast enough".

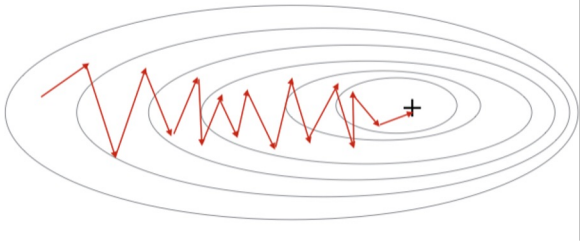
Stochastic Gradient Descent

Batch gradient descent



$$\frac{2}{n}\sum_{i=1}^{n}\left(\left[\theta^{(t-1)}\right]^{T}x^{(i)}-y^{(i)}\right)x^{(i)}$$

Stochastic gradient descent



$$2\left(\left[\theta^{(t-1)}\right]^{T}x^{(i)}-y^{(i)}\right)x^{(i)}$$

Intuition: SGD is a **noisy** estimate of the full batch gradient

In many cases, SGD can help escape local minima and lead to faster convergence.

SGD on Linear Regression

$$\theta^{(t)} = \theta^{(t-1)} - \eta \left(\left[\theta^{(t-1)} \right]^T x^{(i)} - y^{(i)} \right) x^{(i)}$$

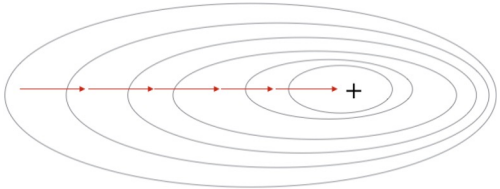
$$-\left(\left[\theta^{(t-1)} \right]^T x^{(i)} - y^{(i)} \right) = \left(y^{(i)} - \left[\theta^{(t-1)} \right]^T x^{(i)} \right) = \text{“residual”}$$

If residual ≈ 0, the current model’s guess is good and the model does not change much.

If residual ≠ 0, then we add $x^{(i)}$ to $\theta^{(t-1)}$, weighted by the residual and learning rate.

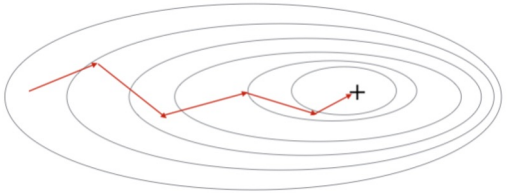
Mini-batch SGD

Batch GD



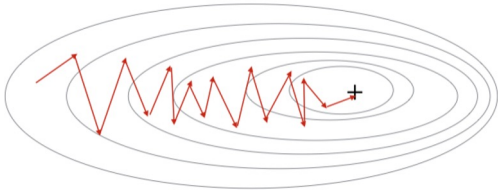
$$\frac{2}{n}\sum_{i=1}^{n}\left(\left[\theta^{(t-1)}\right]^{T}x^{(i)}-y^{(i)}\right)x^{(i)}$$

Mini-batch SGD



$$\frac{2}{b}\sum_{i=1}^{b}\left(\left[\theta^{(t-1)}\right]^{T}x^{(i)}-y^{(i)}\right)x^{(i)}$$

SGD



$$2\left(\left[\theta^{(t-1)}\right]^{T}x^{(i)}-y^{(i)}\right)x^{(i)}$$

Mini-batch SGD: randomly sample a mini-batch of size b