Gradient descent

Fraida Fund

Contents

Math prerequisites for this lecture: You should know about:

- derivatives and optimization (Appendix C in Boyd and Vandenberghe)
- complexity of algorithms (Big O notation)

In this lecture

- Runtime of OLS solution for multiple/LBF regression
- Solution using gradient descent
- Variations on main idea

Runtime of OLS solution

Limitations of OLS solution

- Specific to linear regression, L2 loss
- For extremely large datasets: runtime, memory

Background: Big O notation

Approximate the number of operations required, as a function of input size.

- Ignore constant terms, constant factors
- Ignore all but the dominant term

Example: $3n^3 + 100n^2 + 1000$?

Computing OLS solution

We had

$$\mathbf{w}^* = \left(\Phi^T\Phi\right)^{-1}\Phi^T\mathbf{y}$$

where Φ is an $n \times d$ matrix. If $n \geq d$ then it is (usually) full rank and a unique solution exists.

How long does it take to compute?

Runtime of a "naive" solution using "standard" matrix multiplication:

- $O(d^2n)$ to multiply $\Phi^T\Phi$
- $O(dn)$ to multiply $\Phi^T y$
- $O(d^3)$ to compute the inverse of $\Phi^T\Phi$

Since n is generally much larger than d, the first term dominates and the runtime is $O(d^2n)$.

(Note: in practice, we can do it a bit faster.)

Solution using gradient descent

Exhaustive search

Iterative solution

Suppose we would start with all-zero or random weights. Then iteratively (for t rounds):

- pick random weights
- if loss performance is better, keep those weights
- if loss performance is worse, discard them

For infinite t, we'd eventually find optimal weights - but clearly we could do better.

Background: Gradients and optimization

Gradient has *two* important properties for optimization:

At a minima (or maxima, or saddle point),

$$\nabla L(\mathbf{w}) = 0$$

At other points, $\nabla L(\mathbf{w})$ points towards direction of maximum (infinitesimal) rate of *increase*.

Gradient descent idea

To move towards minimum of a (smooth, convex) function, use first order approximation:

Start from some initial point, then iteratively

- compute gradient at current point, and
- add some fraction of the negative gradient to the current point

Gradient descent illustration

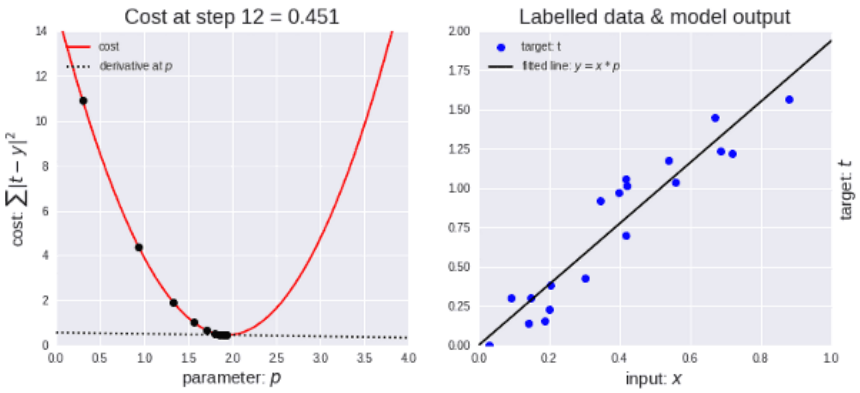


Figure 1: Link for animation. Image credit: Peter Roelants

Standard ("batch") gradient descent

For each step t along the error curve:

$$\begin{aligned}\mathbf{w}^{t+1} &= \mathbf{w}^t - \alpha \nabla L(\mathbf{w}^t) \\ &= \mathbf{w}^t - \frac{\alpha}{n} \sum_{i=1}^{n} \nabla L_i(\mathbf{w}^t, \mathbf{x}_i, y_i)\end{aligned}$$

Repeat until stopping criterion is met.

Example: gradient descent for linear regression (1)

With a mean squared error loss function

$$L(w, X, y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \langle w, x_i\rangle)^2$$
$$= \frac{1}{n}\|y - Xw\|^2$$

Example: gradient descent for linear regression (2)

we will compute the weights at each step as

$$w^{t+1} = w^t + \frac{\alpha^t}{n}\sum_{i=1}^{n}(y_i - \langle w^t, x_i\rangle)x_i$$
$$= w^t + \frac{\alpha^t}{n}X^T(y - Xw^t)$$

(dropping the constant 2 factor)

To update $\mathbf{w}$, must compute n loss functions and gradients - each iteration is $O(nd)$. We need multiple iterations, but in many cases it's more efficient than the previous approach.

However, if n is large, it may still be expensive!

Variations on main idea

Two main "knobs" to turn:

- "batch" size
- learning rate

Stochastic gradient descent

Idea:

At each step, compute estimate of gradient using only one randomly selected sample, and move in the direction it indicates.

Many of the steps will be in the wrong direction, but progress towards minimum occurs *on average*, as long as the steps are small.

Each iteration is now only $O(d)$, but we may need more iterations than for gradient descent. However, in many cases we still come out ahead (especially if n is large!).

See supplementary notes for an analysis of the number of iterations needed.

Also:

- SGD is often more efficient because of *redundancy* in the data - data points have some similarity.
- If the function we want to optimize does not have a global minimum, the noise can be helpful - we can "bounce" out of a local minimum.

Mini-batch (also "stochastic") gradient descent (1)

Idea:

At each step, select a small subset of training data ("mini-batch"), and evaluate gradient on that mini-batch.

Then move in the direction it indicates.

Mini-batch (also "stochastic") gradient descent (2)

For each step t along the error curve:

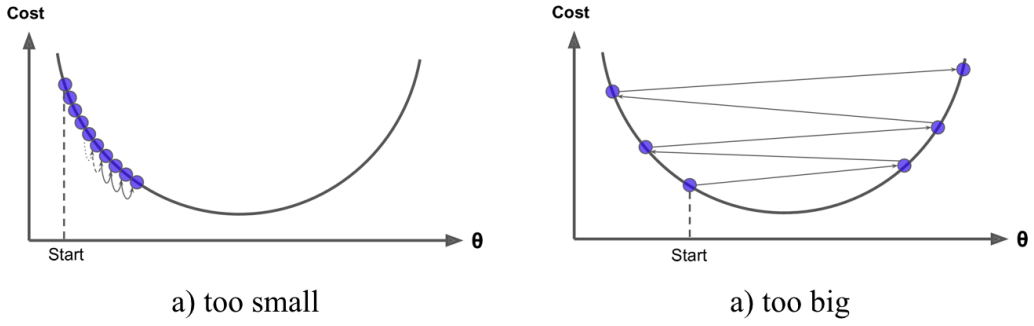
- Select random mini-batch $I_t \subset 1, \ldots, n$
- Compute gradient approximation:

$$g^t = \frac{1}{|I_t|} \sum_{i \in I_t} \nabla L(\mathbf{x}_i, y_i, \mathbf{w}^t)$$

- Update parameters: $\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha^t g^t$

This approach is often used in practice because we get some benefit of vectorization, but also take advantage of redundancy in data.

Selecting the learning rate



a) too small a) too big

Figure 2: Choice of learning rate α is critical

Also note: SGD "noise ball"

Annealing the learning rate

One approach: decay learning rate slowly over time, such as

- Exponential decay: $\alpha_t = \alpha_0 e^{-kt}$
- 1/t decay: $\alpha_t = \alpha_0/(1+kt)$

(where k is tuning parameter).

But: this is still sensitive, requires careful selection of gradient descent parameters for the specific learning problem.

Can we do this in a way that is somehow "tuned" to the shape of the loss function?

Gradient descent in a ravine (1)

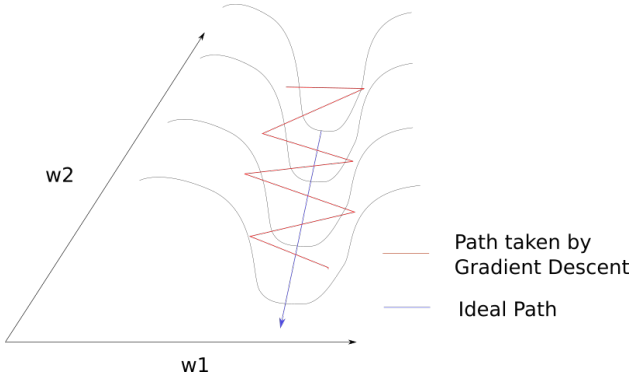


Figure 3: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of w_1.

Gradient descent in a ravine (2)

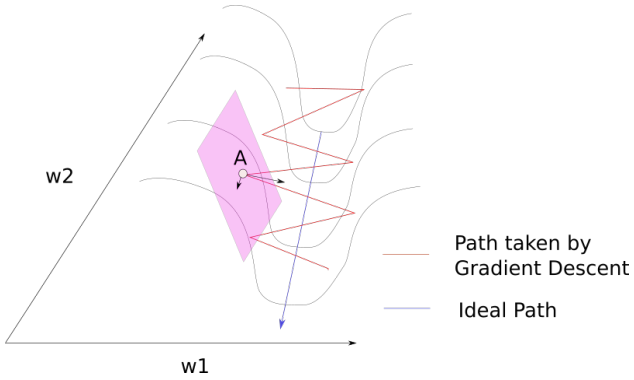


Figure 4: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of w_1.

Momentum (1)

- Idea: Update includes a *velocity* vector v, that accumulates gradient of past steps.
- Each update is a linear combination of the gradient and the previous updates.
- (Go faster if gradient keeps pointing in the same direction!)

Momentum (2)

Classical momentum: for some $0 \leq \gamma_t < 1$,

$$v_{t+1} = \gamma_t v_t - \alpha_t \nabla L\left(w_t\right)$$

so

$$w_{t+1} = w_t + v_{t+1} = w_t - \alpha_t \nabla L\left(w_t\right) + \gamma_t v_t$$

(γ_t is often around 0.9, or starts at 0.5 and anneals to 0.99 over many epochs.)

Note: $v_{t+1} = w_{t+1} - w_t$ is Δw.

Momentum: illustrated

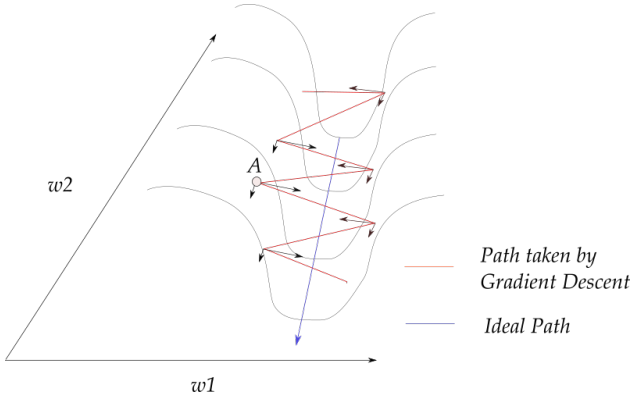


Figure 5: Momentum dampens oscillations by reinforcing the component along w_2 while canceling out the components along w_1.

RMSProp

Idea: Track per-parameter EWMA of square of gradient, to normalize parameter update step.

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_{t+1} + \epsilon}} \nabla L\left(w_t\right)$$

where

$$v_t = \gamma v_{t-1} + (1-\gamma) \nabla L\left(w_t\right)^2$$

Weights with recent gradients of large magnitude have smaller learning rate, weights with small recent gradients have larger learning rates.

Adam: Adaptive Moment Estimation

- Uses ideas from momentum (first moment) and RMSProp (second moment)!
- plus bias correction

Bias correction accounts for the fact that first and second moment estimates start at zero.

Illustration (Beale's function)

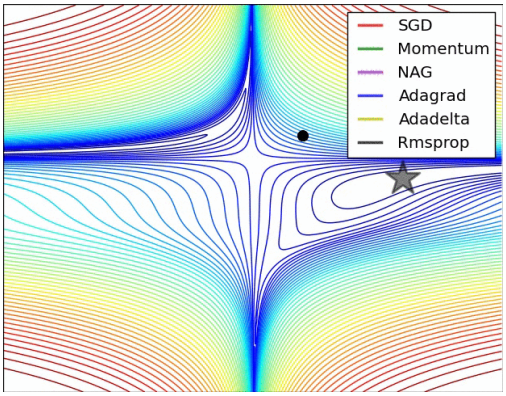


Figure 6: Animation credit: Alec Radford. Link for animation.

Due to the large initial gradient, velocity based techniques shoot off and bounce around, while those that scale gradients/step sizes like RMSProp proceed more like accelerated SGD.

Illustration (Long valley)

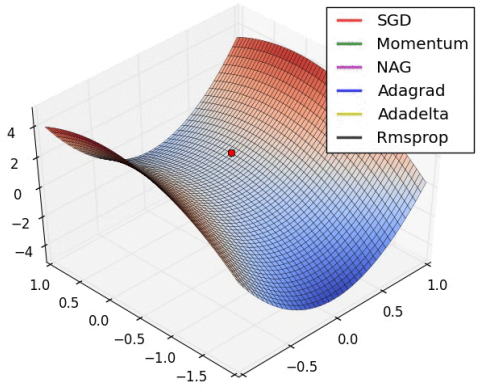


Figure 7: Animation credit: Alec Radford. Link for animation.

SGD stalls and momentum has oscillations until it builds up velocity in optimization direction. Algorithms that scale step size quickly break symmetry and descend in optimization direction.

Recap

- Gradient descent as a general approach to model training
- Variations