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 and especially of vector and matrix operations (Appendix B in Boyd and Vandenberghe, also the complexity part of Section I, Chapter 1 and Section II, Chapter 5)

In this lecture

Addresses “How do we train the model efficiently?”

- 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: $3n^3 + 100n^2 + 1000$ would be $O(n^3)$.

Computing OLS solution

How long does it take to compute

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

where Φ is an $n \times d$ matrix?

Runtime of a “naive” solution using “standard” matrix multiplication:

- $O(d^2 n)$ to multiply $\Phi^T \Phi$
- $O(dn)$ to multiply $\Phi^T y$
- $O(d^3)$ to compute the inverse of $\Phi^T \Phi$ (Note: in practice, we can do it a bit faster.)

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

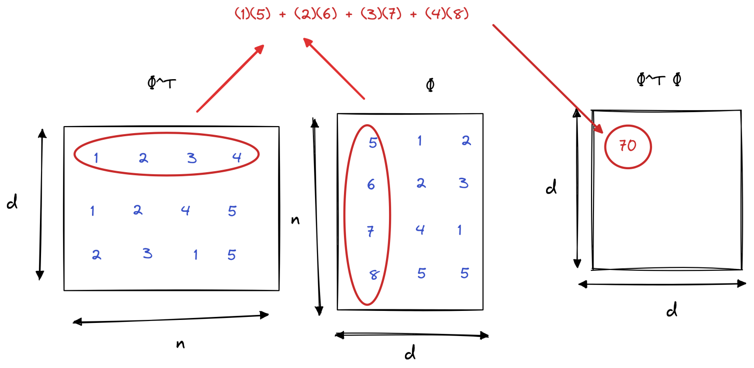

Figure 1: Computing $\Phi^T \Phi$. For each entry in the matrix, we need n multiplications, and then we need to fill in $d \times d$ entries to complete the matrix.

Solution using gradient descent

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

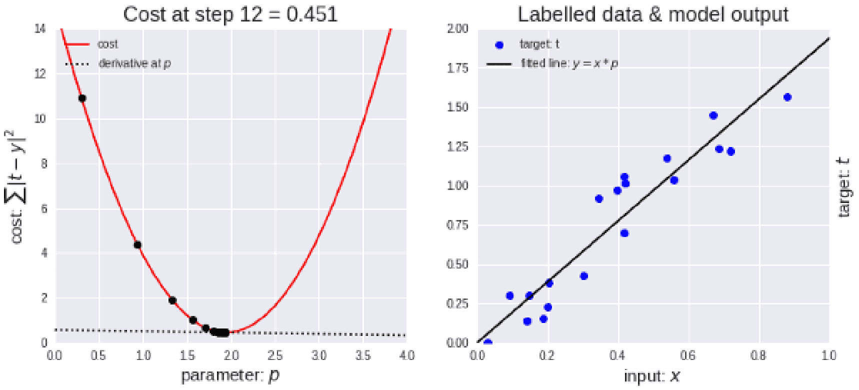

Figure 2: Link for animation. Image credit: Peter Roelants

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.

"Stopping criteria" may be: loss is sufficiently small, gradient is sufficiently close to zero, or a pre-set max number of iterations is reached.

Note: the superscript t tracks what iteration we are on. It's not an exponent!

Example: gradient descent for linear regression (1)

With a mean squared error loss function

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

Gradient of the loss function is:

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

we move in the *opposite* direction, so…

Example: gradient descent for linear regression (2)

We will compute the weights at each step as

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

(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.

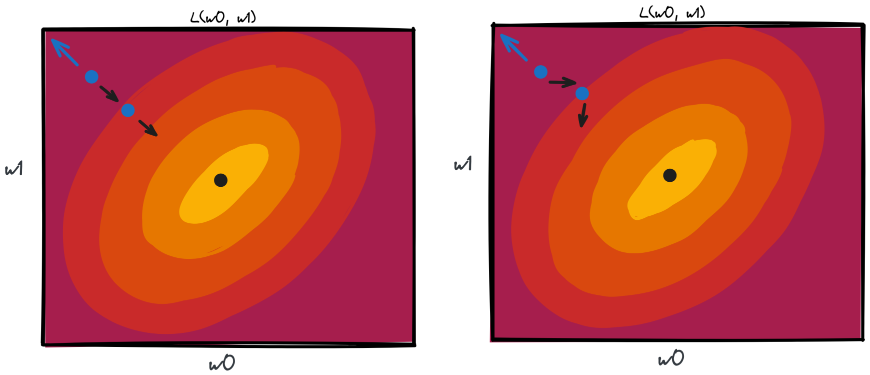

Figure 3: Full-batch gradient descent (left), SGD (right). Many of the steps will be in the wrong direction, but progress towards minimum occurs *on average*, as long as the steps are small.

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.
- But, we miss out on the benefits of vectorization. In practice, it takes longer to compute something over 1 sample 1024 times, than over 1024 samples 1 time.
- 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: $\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha^t g^t$

Now that each iteration is not equal to an iteration over *all* data, we need to introduce the idea of an "epoch":

- One epoch = one pass over *all* the data
- Mini-batch SGD is often used in practice because we get some benefit of vectorization, but also take advantage of redundancy in data.

After a fixed number of epochs (passes over the entire data), * we may end up at a better minimum (lower loss) with a small batch size, * *but*, the time per epoch may be longer with a small batch size.

Selecting the learning rate

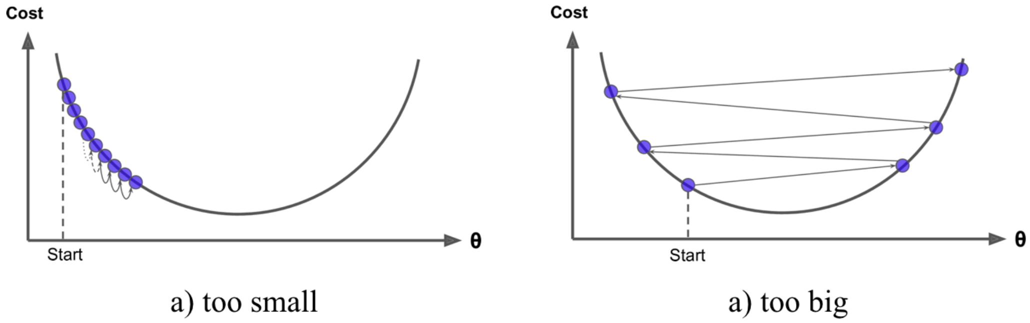

Figure 4: Choice of learning rate α is critical

Image credit: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition, by Aurélien Géron.

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)

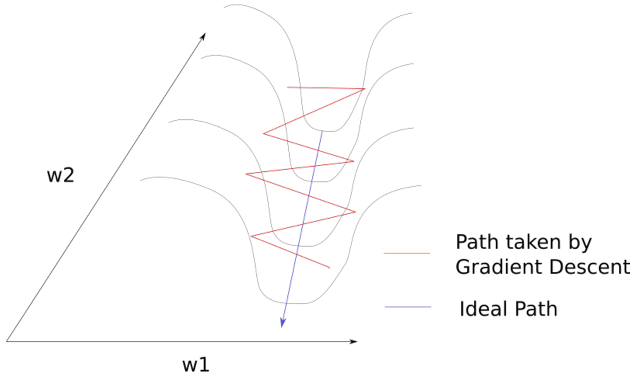

Figure 5: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of w_1.

Gradient descent in a ravine (2)

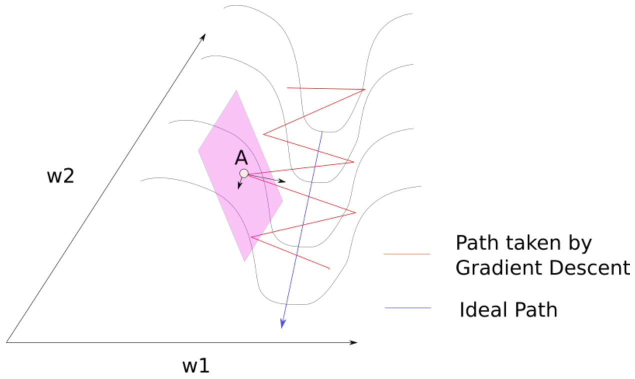

Figure 6: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of w_1.

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 $0 \leq \gamma_t < 1$,

$$v^{t+1} = \gamma^t v^t + \nabla L\left(w^t\right)$$

so

$$w^{t+1} = w^t - \alpha^t v^{t+1} = w^t - \alpha^t \left(\gamma^t v^t + \nabla L\left(w^t\right)\right)$$

(γ may be in range 0.9 - 0.99.)

Momentum: pseudocode

GD:

```
for t in range(num_steps):
  dw = compute_grad(w)
  w -= lr * dw
```

GD + Momentum:

```
for t in range(num_steps):
  dw = compute_grad(w)
  v = gamma * v + dw
  w -= lr * v
```

Momentum: illustrated

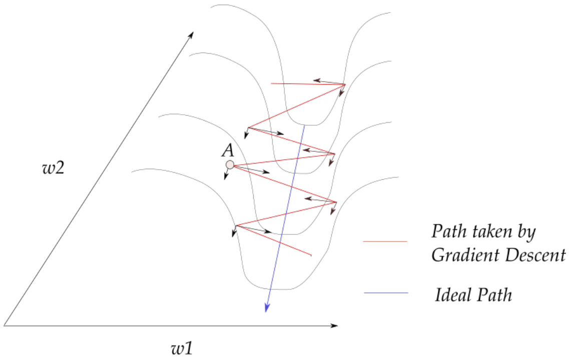

Figure 7: Momentum dampens oscillations by reinforcing the component along w_2 while canceling out the components along w_1.

AdaGrad (1)

Next idea: “per-parameter learning rates”!

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

AdaGrad (2)

$$w^{t+1} = w^t - \frac{\alpha}{\sqrt{v^{t+1} + \epsilon}} \nabla L\left(w^t\right)$$

where

$$v^{t+1} = v^t + \nabla L\left(w^t\right)^2$$

Weights with large gradient have smaller learning rate, weights with small gradients have larger learning rates.

i.e.: take smaller steps in steep directions, take bigger steps where the gradient is flat.

AdaGrad: pseudocode

GD:

```
for t in range(num_steps):
  dw = compute_grad(w)
  w -= lr * dw
```

GD + AdaGrad:

```
grad_sq = 0
for t in range(num_steps):
  dw = compute_grad(w)
  grad_sq = grad_sq + dw * dw
  w -= lr * dw / sqrt(grad_sq + epsilon)
```

RMSProp: Leaky AdaGrad

Idea: Use EWMA to emphasize *recent* gradient magnitudes.

$$w^{t+1} = w^t - \frac{\alpha}{\sqrt{v^{t+1} + \epsilon}} \nabla L\left(w^t\right)$$

where

$$v^{t+1} = \gamma v^t + (1 - \gamma) \nabla L\left(w^t\right)^2$$

RMSProp: pseudocode

GD + AdaGrad:

```
grad_sq = 0
for t in range(num_steps):
  dw = compute_grad(w)
  grad_sq = grad_sq + dw * dw
  w -= lr * dw / sqrt(grad_sq + epsilon)
```

GD + RMSProp:

```
grad_sq = 0
for t in range(num_steps):
  dw = compute_grad(w)
  grad_sq = gamma * grad_sq + (1 - gamma) * dw * dw
  w -= lr * dw / sqrt(grad_sq + epsilon)
```

Adam: Adaptive Moment Estimation

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

Adam: pseudocode vs Momentum

GD + Momentum:

```
grad_sq = 0
for t in range(num_steps):
  dw = compute_grad(w)
  grad_sq = gamma * grad_sq + (1 - gamma) * dw * dw
  w -= lr * dw / sqrt(grad_sq + epsilon)
```

GD + Adam (without bias correction):

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_grad(w)
  moment1 = b1 * moment1 + (1 - b1) * dw
  moment2 = b2 * moment2 + (1 - b2) * dw * dw
  w -= lr * moment1 / sqrt(moment2 + epsilon)
```

Adam: pseudocode vs RMSProp

GD + RMSProp:

```
grad_sq = 0
for t in range(num_steps):
  dw = compute_grad(w)
  grad_sq = gamma * grad_sq + (1 - gamma) * dw * dw
  w -= lr * dw / sqrt(grad_sq + epsilon)
```

GD + Adam (without bias correction):

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_grad(w)
  moment1 = b1 * moment1 + (1 - b1) * dw
  moment2 = b2 * moment2 + (1 - b2) * dw * dw
  w -= lr * moment1 / sqrt(moment2 + epsilon)
```

Usually `b1` is smaller than `b2`, i.e. we update `moment1` more aggressively than `moment2`.

Adam: Pseudocode with bias correction

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_grad(w)
  moment1 = b1 * moment1 + (1 - b1) * dw
  moment2 = b2 * moment2 + (1 - b2) * dw * dw
  moment2_unbias = moment2 / (1 - b2 t)
  moment2_unbias = moment2 / (1 - b2 t)
  w -= lr * moment1_unbias / sqrt(moment2_unbias + epsilon)
```

When we initialize both moments to zero, they are initially “biased” to smaller values (since they update slowly!) This bias correction accounts for that.

Illustration (Beale's function)

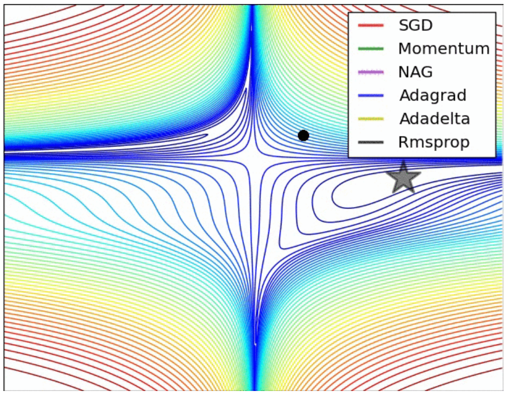

Figure 8: Animation credit: Alec Radford. Link for animation.

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)

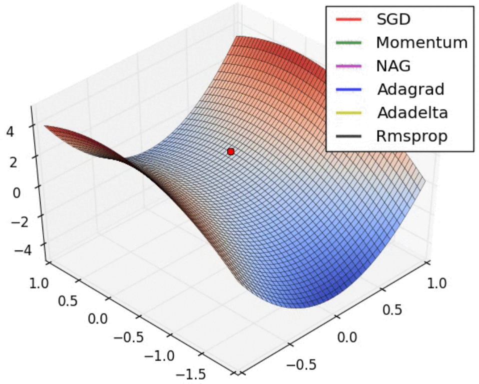

Figure 9: Animation credit: Alec Radford. Link for animation.

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 training
- Variations

Gradient descent is easy on linear regression! You won't get to apply any of these more advanced techniques until later in the semester, when we work with less friendly loss surfaces.