

STA2311: Advanced Computational Methods for Statistics I

Class 4: Stochastic Optimization

Radu Craiu Robert Zimmerman

University of Toronto

October 3, 2023

- 1 Introduction
- 2 Stochastic Gradient Descent
- 3 Simulated Annealing
- 4 Genetic Algorithms

Section 1

Introduction

Assumed Knowledge

- In this class and the next, assume we know how to sample simple random variables
- Essentially what was discussed in Class 1
- Usually the multivariate normal distribution will suffice
 - ▶ To sample $\mathbf{X} \sim \mathcal{N}_d(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ in R, use

```
d <- 3
mu <- c(2,3,1)
Sigma <- matrix(c(1.3, -1.6, -0.6,
                 -1.6, 4.5, 0.9,
                 -0.6, 0.9, 3.5), nrow=3, ncol=3, byrow=T)
Z <- rnorm(n=d)
X <- mu + t(chol(Sigma))%*%Z
```

- We will discuss sampling much more thoroughly in Class 6

Deterministic vs. Stochastic Algorithms

- In the last two classes, we learned about classical optimization techniques and the EM algorithm
- These algorithms were all *deterministic*
- That is, each algorithm, run with the same inputs (known parameters, initializations, etc.) would invariably produce the same outputs
- It turns out that introducing randomness into certain optimization procedures can dramatically improve their performance
- Such algorithms are generally called *stochastic optimization procedures*

A Simple Stochastic Algorithm

- To illustrate, suppose we want to minimize a function $g : \mathbb{R}^d \rightarrow \mathbb{R}$
- Given a current best guess $\boldsymbol{\theta}_t$ of the minimizer, we can *propose* a candidate $\boldsymbol{\theta}' = \boldsymbol{\theta}_t + \mathbf{Z}$, where $\mathbf{Z} \sim \mathcal{N}_d(\mathbf{0}, \sigma^2 \mathbf{I})$ for some $\sigma^2 > 0$
- If $g(\boldsymbol{\theta}') < g(\boldsymbol{\theta}_t)$, then accept the candidate and set $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}'$; otherwise, set $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t$
- This is called *random optimization*, and works well when d is small
 - ▶ But performance becomes much worse as d increases due to the curse of dimensionality
- We will discuss several more refined stochastic optimization methods

Section 2

Stochastic Gradient Descent

Gradient Descent

- Recall the gradient descent (GD) algorithm from Class 2
- We aim to minimize a differentiable function $g : \mathbb{R}^d \mapsto \mathbb{R}$
- Starting with an initial value θ_0 , the basic GD procedure selects

$$\theta_{t+1} = \theta_t - h\nabla g(\theta_t)$$

for some pre-chosen $h > 0$

- Once a stopping criterion has been met (say at iteration T), we could output θ_T or $\frac{1}{T} \sum_{t=1}^T \theta_t$ or θ_s where $s = \operatorname{argmin}_{t \leq T} g(\theta_t)$

The Problem with GD

- In statistical applications we are interested in minimizing $g(\theta) = -l(\theta|\mathbf{y}) = -\sum_{i=1}^n \log(f(y_i|\theta))$ which we often do by solving $0 = \nabla g(\theta) = -\frac{1}{n} \sum_{i=1}^n \frac{\partial}{\partial \theta} \log(f(y_i|\theta))$
- For example, in multiple linear regression, the function to be minimized is

$$g(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (y_i - \boldsymbol{\theta}^\top \mathbf{x}_i)^2$$

and its gradient is

$$\nabla g(\boldsymbol{\theta}) = \frac{2}{n} \sum_{i=1}^n \left((\boldsymbol{\theta}^\top \mathbf{x}_i - y_i), x_{i,1}(\boldsymbol{\theta}^\top \mathbf{x}_i - y_i), \dots, x_{i,p}(\boldsymbol{\theta}^\top \mathbf{x}_i - y_i) \right) \quad (1)$$

where n is the number of observations in our training dataset

From GD to SGD

- If sample size n is large, the computational cost is too large so instead one could use a random sample from $\{y_1, y_2, \dots, y_n\}$ to produce an unbiased estimator of $\nabla g(\theta)$
- In *stochastic gradient descent (SGD)*, we replace the gradient $\nabla g(\theta^{(t)})$ with a random vector $\mathbf{W}^{(t)}$
- The random vector is chosen so that its *expected value* is $\nabla g(\theta^{(t)})$
- In other words, we seek an unbiased estimator of the gradient
- Canonical choices:
 - ▶ $\mathbf{W}^{(t)} = \nabla_{\theta} \log(f(y_I | \theta))$ where $I \sim \text{Uniform}\{1, \dots, n\}$
 - ▶ $\mathbf{W}^{(t)} = \frac{1}{K} \sum_{j=1}^K \nabla_{\theta} \log(f(y_{i_j} | \theta))$ where $\{i_1, \dots, i_K\}$ is a simple random sample (without replacement) from $\{1, \dots, n\}$.

Loss Functions

- In general, suppose that the function to be minimized is

$$g(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n L(\boldsymbol{\theta}, \mathbf{y}_i),$$

where \mathbf{y}_i is the i 'th observation in our dataset and L is a *loss function*

- Therefore

$$\nabla g(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \mathbf{y}_i) = \mathbb{E}_l \left[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{(t)}, \mathbf{Y}_l) \right]$$

- The random vector $\mathbf{W}^{(t)}$ we choose satisfies

$$\mathbb{E} \left[\mathbf{W}^{(t)} \mid \boldsymbol{\theta}^{(t)} \right] = \mathbb{E}_l \left[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{(t)}, \mathbf{Y}_l) \right]$$

where the left side expectation is with respect to whatever randomization procedure we have to determine \mathbf{W} .

Unbiased Estimators of the Gradient

- Use a single-observation-at-a-time design and cycle through all observations
- Partition data into batches of size $K \ll n$ which are sampled at random without replacement.

The Algorithm

- The *stochastic gradient descent* algorithm is
 - 1 Initialize the process at θ_0 and choose a pre-specified step size $h > 0$ and number of iterations T
 - 2 Make the updates $\theta_{t+1} = \theta_t - h_t \mathbf{W}^{(t)}$ for $0 \leq t \leq T$, where $\mathbf{W}^{(t)}$ is a random vector such that $\mathbb{E}[\mathbf{W}^{(t)} | \theta^{(t)}] = \nabla g(\theta^{(t)})$
 - 3 Step size h_t can be constant or can be decreased $h_t = \gamma^t h$ with $\gamma \in (0, 1)$.
- Clearly, the update is no longer guaranteed to decrease g at every update.
- Convergence is towards a ball centered at the stationary point θ^* so the method is often used when approximate rather than precise optimization is acceptable (e.g., ML applications with massive volume of training data)
- Theory...

Example: Logistic Regression

- Consider the same logistic regression example from Class 2

```
set.seed(7)
expit <- function(x) {1/(1+exp(-x))}
logit <- function(p) {log(p/(1-p))}
norm <- function(x) {sqrt(sum(x^2))}

n <- 1000

X1 <- rnorm(n=n)
X2 <- rbinom(n=n, size=1, prob=0.2)
X3 <- rpois(n=n, lambda=0.7)
X <- cbind(1, X1, X2, X3)

y <- rbinom(n=n, size=1, prob=expit(0.4 + 0.7*X1 + 3*X2 - X3))

grad.g <- function(theta, XX, yy) {
  t(XX) %*% (expit(apply(XX, 1, function(x) x*theta)) - yy)}
}
```

Example: Logistic Regression (Continued)

```
# use TT iterations
TT <- 1000
# size of minibatch
k <- 50

#want to go through all the data in rep1 updates
repeats1=n/k
#need to cycle through all the data rep2 times
repeats2=TT/repeats1

sub.sample=matrix(0,nrow=TT, ncol=k)
for(i in 1:repeats2){
  samp.inds<-sample(1:n)
  sub.sample[((i-1)*repeats1+1):(i*repeats1),]=
    matrix(samp.inds,ncol=k,nrow=repeats1,byrow=T) }

#stores the parameter values
ths <- cbind( rep(1, 4), matrix(0L, nrow=4, ncol=TT-1))
```

Example: Logistic Regression (Continued)

```
for (t in 1:(TT-1)) {  
  # samp.indxs <- sample(1:n, size=k, replace=F)  
  X.k <- X[as.vector(sub.sample[t,]),]  
  y.k <- y[as.vector(sub.sample[t,])]  
  
  #fixed stepsize  
  alp <- 0.005  
  
  ths[,t+1] <- ths[,t] - alp*grad.g(ths[,t], X.k, y.k)  
}  
  
th.SGD <- ths[,TT]  
th.NR <- as.vector(glm(y ~ ., family = binomial(link="logit"),  
                      data=data.frame(y, X1, X2, X3))$coefficients)  
print(cbind(th.SGD, th.NR))  
par(mfrow=c(2,2))  
for (i in 1:4) ts.plot(ths[i,])
```


Section 3

Simulated Annealing

Basic Metallurgy

- Reference: [Goffe et al. \[1994\]](#)
- In metallurgy (or thermodynamics more generally), *annealing* means “slow cooling”
- When casting metallic objects from molten metal, the final goal is to bring the metal to a minimum-energy state (where it is very hard)
- However, hot metal is easier to mold
- One wants to shape the metal while slowly cooling it (cooling it too fast will not allow reaching the desired shape)
- ... but not *too* slowly, since we don't want to wait too long
- Simulated annealing seeks to minimize/maximize a function according to the same principle

The set up

- Interested in minimizing $g(\theta) : E \subset \mathbb{R}^d \rightarrow \mathbb{R}$ (or maximizing $-g(\theta)$) where g is not restricted (can be discrete, non-continuous, etc).
- We will assume here that g is bounded
- Note that max of $-g$ is also max of $\tilde{g} = \exp(-g)$ which can be thought of as an unnormalized density on E .
- Most of the algorithms studied so far have trouble escaping a local extrema point
- We “heat” $\tilde{g} = \exp(-g(\theta))$ into $\tilde{g}_t = \exp(-g(\theta)/t)$ where $t \geq 1$.
- Intuition: As $T \rightarrow \infty$ $\tilde{g}_T(\theta) = 1_E(\theta)$ so if \tilde{g} has extreme points separated by “big dips” its heated versions are more “level”

Example

```
f.T=function(nn=3,xx=c(1,1), TT=1){
  rez=(16*xx[1]*(1-xx[1])*xx[2]*(1-xx[2])*sin(nn*pi*xx[1])*sin(nn*pi*xx[2])
  return(exp(rez/TT))}

n.eval=100
co1=ppoints(n.eval)
co3=ppoints(n.eval)

TT=1
for(i in 1:n.eval)
co3[i]=f.T(n,c(co1[i],0.5),TT)
plot(co1,co3,col="black",type="l", xlab="x",ylab="f(x,0.5)")
```

Example

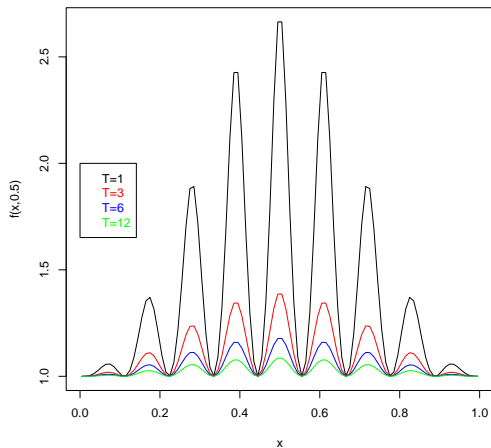


Figure 1: Illustration of annealing

Building Up the Algorithm

- Suppose at time t , our best guess of the optimum is $\theta^{(t)}$
- We construct a candidate θ' for $\theta^{(t+1)}$ by randomly perturbing one element of $\theta^{(t)}$
- If $g(\theta') < g(\theta_t)$, good — we've improved upon the old guess, so take $\theta_{t+1} = \theta'$
- Otherwise, θ_t could just be a *local* minimum that we'd like to escape from
- In order to escape, one must accept moves that seem counterproductive (e.g., increase the value of g occasionally)

The Algorithm

- The basic *simulated annealing* algorithm is
 - 1 Choose a run length M , a cooling schedule $T : \{0, 1, \dots, M\} \rightarrow (0, \infty)$ and initialize the process at θ_0
 - 2 Evaluate $g(\theta_0)$
 - 3 For $1 \leq t \leq M$, propose a new θ' by perturbing a random coordinate:
 $\theta'_j = \theta_j + \xi_t$ for some random variable ξ_t
 - ★ If $g(\theta') < g(\theta_t)$, take $\theta_{t+1} = \theta'$
 - ★ If $g(\theta') \geq g(\theta_t)$, set $\theta_{t+1} = \theta_t$ with probability

$$e^{(g(\theta_t) - g(\theta')) / T(t)} = \frac{\tilde{g}_{T(t)}(\theta')}{\tilde{g}_{T(t)}(\theta_t)}$$

and $\theta_{t+1} = \theta_t$ otherwise

- Note that in our example the ratio $\frac{\tilde{g}_{T(t)}(\theta')}{\tilde{g}_{T(t)}(\theta_t)}$ will tend to be closer to 1 when $T(t) > 1$.

That Random Variable

- The random variable ξ_t used to perturb the coordinate at step t of the algorithm is chosen based on what we know about the objective function
- If g is continuous with domain \mathbb{R}^d , then we can take $\xi_t \stackrel{iid}{\sim} \mathcal{N}(0, \sigma^2)$ for some chosen variance σ^2
- Or the the algorithm can be made to be adaptive, with independent $\xi_t \sim \mathcal{N}(0, \sigma_t^2)$
- But the domain of g can be finite (like the configuration of atoms in a metal) too
 - ▶ In this case, the “perturbed” coordinate can be chosen randomly from the configuration space of g

The Cooling Schedule

- The “temperature” evolves according to the cooling schedule (or *annealing schedule*) and should generally decrease to 0 as $t \rightarrow \infty$
- There are many choices for the functional form of $T(t)$
- Linear: $T(t) = T(0) - dt$ for some $d > T(0)/M$
- Logarithmic: $T(t) = T(0)/\log(1 + t)$
- Geometric: $T(t) = r \cdot T(t - 1)$ for some $r \in (0, 1)$
- Exponential: $T(t + 1) = (T(1)/T(0))^t \cdot T(t)$

Example: Logistic Regression (Again)

```
set.seed(2311)
expit <- function(x) {1/(1+exp(-x))}
logit <- function(p) {log(p/(1-p))}
norm <- function(x) {sqrt(sum(x^2))}

n <- 1000

X1 <- rnorm(n=n)
X2 <- rbinom(n=n, size=1, prob=0.2)
X3 <- rpois(n=n, lambda=0.7)
X <- cbind(1, X1, X2, X3)

y <- rbinom(n=n, size=1, prob=expit(0.4 + 0.7*X1 + 3*X2 - X3))

g <- function(theta) {
  -sum(y*log(expit(apply(X, 1, function(x) x%%theta))) +
    (1-y)*log(1-expit(apply(X, 1, function(x) x%%theta))))}
}
```

Example: Logistic Regression (Again) (Continued)

```
M <- 5000
th <- rep(1, times=4)

for (t in 1:M) {
  TT <- 10/log(t+1)
  gth <- g(th)

  p.ind <- sample(1:4, size=1)
  th.p <- th
  th.p[p.ind] <- th.p[p.ind] + rnorm(n=1, sd=.5)
  gth.p <- g(th.p)

  if (gth.p < gth || runif(n=1) < min(exp(gth - gth.p)/TT,1)) {
    th <- th.p
  } else {
    th <- th
  }
}

th.SA <- th
```

Benefits

- Observe that there are essentially no restrictions on the objective function g
- It need not be continuous, which makes simulated annealing useful for discrete optimization problems
 - ▶ The *travelling salesman problem* is a classical example
- We will see later in the course that simulated annealing is a particular kind of Markov Chain Monte Carlo (MCMC) algorithm

Section 4

Genetic Algorithms

Inspired by Evolution

- Genetic Algorithms (GA) are iterative stochastic algorithms often used in discrete optimization
- The design is inspired by theory of evolution and adaptation through genetics
- Assumptions:
 - ▶ (A1) Fitness: the quality of a potential solution can be evaluated using a fitness function, e.g. if the problem of interest is finding $\arg \max_x g(x)$ then the fitness of a candidate solution x_0 is $g(x_0)$.
 - ▶ (A2) Representation: every candidate solution v to the optimization problem can be represented as a string of bits (vectors whose entries are 0 or 1) $v \in \{0, 1\}^M$

Implementation

- The algorithm cycles through the following steps:
 - ▶ From a population of candidate solutions at iteration t , $\mathcal{S}_t = \{v_i, g(v_i) : 1 \leq i \leq K\}$ we use *selection* to produce an intermediate population
 - ▶ The intermediate population can be created in different ways:
 - ★ It is \mathcal{S}_t
 - ★ Via sampling K times with replacement from \mathcal{S}_t using $p_i \propto g(v_i)$ as the sampling weights.
 - ★ Use the fittest L elements in \mathcal{S}_t and $K - L$ random elements from \mathcal{S}_t .
 - ▶ The population at time $t + 1$ is obtained from the intermediate population using crossover and mutation.

Crossover

- Consider two potential solutions/strings

$(101 \dots 1101)$ and $(yxy \dots xxy)$

- Randomly select the crossover point $c \in \{1, \dots, K - 1\}$

11010 \ / 01100101101

yxyyx / \ yxxyyyxyxxy

11010yxxyyyxyxxy and yxyyx01100101101

- There can be more than one crossover point

Mutation

- With certain probability ϵ a random element in the string is flipped
- Alternatively, can think of mutation of generating a random new bit for a randomly selected component of v_i
- In summary: for each i flip a coin to decide whether v_i will mutate. If it does select the component uniformly at random and with probability ϵ flip it.

Possible references: [Whitley \[1994\]](#) [Chatterjee et al. \[1996\]](#)

Example

- Find $\arg \max_{(x_1, x_2)} f(x_1, x_2)$ where

$$f(x_1, x_2) = [16x_1x_2(1 - x_1)(1 - x_2) \sin(9\pi x_1) \sin(9\pi x_2)]^2$$

```
library(rgl)
f=function(nn=3,xx=c(1,1)){
  rez=(16*xx[1]*(1-xx[1])*xx[2]*(1-xx[2])*
    sin(nn*pi*xx[1])*sin(nn*pi*xx[2]))^2
  return(rez)}

# we want to plot f

n.eval=100
co1=ppoints(n.eval)
co2=ppoints(n.eval)
s=matrix(0,n.eval,n.eval)
n=9
set.seed(17)
```

Example (cont'd)

```
for(i in 1:n.eval){for(j in 1:n.eval)
  {s[i,j]=f(n,c(co1[i],co2[j]))}}

persp3d(co1,co2,s,col = "white", package = "rgl")
contour(co1,co2,s)
S= 80 # size of each generation
itr= 100 # number of iterations to run
m=10 # the number of digits (determines the precision of solution)
m.rate=0.01 # mutation rate
rnk =c(1:S) # ranks the specimens in one generation
phi =c(1:S) # fitness for each specimen
best.fitness =c(1:(itr+1)) # best fitness found
fitness =matrix(0,nrow=(itr+1),ncol=S) # stores all the fitness-es
bin.current=array(0,c(S,2,m)) # current population
bin.next=array(0,c(S,2,m)) # current population
twos=(1/2)^c(1:m)
x.current=x.next=matrix(0,ncol=2,nrow=S)
```

Example (cont'd)

```
# initialize the population
```

```
for(i in 1:S){  
  bin.current[i,1,]=rbinom(m,1,0.5)  
  bin.current[i,2,]=rbinom(m,1,0.5)  
  x.current[i,1]=sum(twos*bin.current[i,1,])  
  x.current[i,2]=sum(twos*bin.current[i,2,])  
  phi[i]=f(n,c(x.current[i,1],x.current[i,2]))}  
  
fitness[1,]=phi  
rnk=order(phi)  
best.fitness[1]=max(phi)
```

Example (cont'd)

```
for(j in 1:itr-1)
{
```

```
# BUILDS THE NEW GENERATION, SELECTING FIRST PARENT BASED ON
# FITNESS AND THE SECOND PARENT AT RANDOM
# THERE ARE S/2-2 BREEDINGS ALLOWED RESULTING IN S OFFSPRINGS
```

```
# we keep the top two specimens in the population
```

```
for(i in 1:S){
x.current[i,1]=sum(twos*bin.current[i,1,])
x.current[i,2]=sum(twos*bin.current[i,2,])
phi[i]=f(n,c(x.current[i,1],x.current[i,2]))
}
```

```
fitness[1,]=phi
rnk=order(phi)
```

```
bin.next[1,,]=bin.current[rnk[S],,]# this one has the highest rank in fitness
```

```
bin.next[S,,]=bin.current[rnk[S-1],,]# has the second highest rank in fitness
```

```
x.current[1,]=x.current[rnk[S],,]
```

Example (cont'd)

```
for(i in 2:(S/2)) {  
  # samples from {1,...,S} with probs prop to phi  
  parent1.index=sample(1:S,1,prob=phi)  
  parent2.index = sample(1:S,1)  
  #crossover position  
  pos = sample(1:(m-1),1)  
  #the first offspring is produced via crossover  
  # first coordinate  
  bin.next[i,1,1:pos]=bin.current[parent1.index,1,1:pos]  
  bin.next[i,1,(pos+1):m]=bin.current[parent2.index,1,(pos+1):m]  
  #second coordinate  
  bin.next[i,2,1:pos]=bin.current[parent1.index,2,1:pos]  
  bin.next[i,2,(pos+1):m]=bin.current[parent2.index,2,(pos+1):m]
```

Example (cont'd)

```
# THE MUTATION STEP IS PERFORMED
mutate = rbinom(m,1,m.rate)
#if a mutation has occurred, the coordinate is flipped
bin.next[i,1,] = (bin.next[i,1,]+mutate)%%2
# repeat the process for second offspring
pos = sample(1:(m-1),1)
mutate = rbinom(m,1,m.rate)
bin.next[S-i+1,1,1:pos]=bin.current[parent2.index,1,1:pos]
bin.next[S-i+1,1,(pos+1):m]=bin.current[parent1.index,1,(pos+1):m]
bin.next[S-i+1,2,1:pos]=bin.current[parent2.index,2,1:pos]
bin.next[S-i+1,2,(pos+1):m]=bin.current[parent1.index,2,(pos+1):m]
bin.next[i,1,] = (bin.next[i,1,]+mutate)%%2}
for(i in 2:(S-1)){
x.current[i,1]=sum(twos*bin.next[i,1,])
x.current[i,2]=sum(twos*bin.next[i,2,])
phi[i]=f(n,c(x.current[i,1],x.current[i,2]))
}
bin.current=bin.next
```

Example (cont'd)

```
# update the fitness values
for(k in 1:S){
  x.current[k,1]=sum(twos*bin.current[k,1,])
  x.current[k,2]=sum(twos*bin.current[k,2,])
  phi[k]=f(n,c(x.current[k,1],x.current[k,2]))  }

fitness[j+1,]=phi
best.fitness[j+1]=max(phi)}

print(x.current[rnk[S],])
plot(c(1,40),c(0,1.1),type="n", xlab="Generation", ylab="Best solution")
lines(c(1:itr),best.fitness[1:itr])
abline(h=f(n,c(0.5,0.5)),col="red")
```


References I

- Sangit Chatterjee, Matthew Laudato, and Lucy A Lynch. Genetic algorithms and their statistical applications: an introduction. *Computational Statistics & Data Analysis*, 22(6):633–651, 1996.
- William L Goffe, Gary D Ferrier, and John Rogers. Global optimization of statistical functions with simulated annealing. *Journal of econometrics*, 60(1-2):65–99, 1994.
- Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4: 65–85, 1994.