Optimization for Machine Learning

Nick Henderson, AJ Friend (Stanford University) Kevin Carlberg (Sandia National Laboratories)

August 13, 2019

Model fitting

Model fitting

Outline

Model fitting

Linear least squares: 1D case with linear data

Linear least squares: 1D case with non-linear data

Linear least squares: general formulation and matrix-vector form

Examples

Nonlinear least squares

Beyond least squares

Deep Feedforward Networks

Stochastic methods

Model fitting

Notation

The notation between these worlds is not consistent

- Optimization
 - ► *f*: optimization objective function
 - x: optimization variables
- Machine learning (this set of slides)
 - ϕ : optimization objective function (i.e., loss function)
 - β or θ : optimization variables (i.e., model parameters)
 - ▶ *f*: regression function mapping inputs to outputs
 - x: model inputs (i.e., independent variable)
 - y: model outputs (i.e., response variable)

Least-squares regression

- A type of model fitting with many applications
- **Goal**: find a model that best fits training data in the least-squares sense
- Illuminates the connection between unconstrained optimization and statistics/machine learning
- We will use the following iPython notebooks
 - least-squares.ipynb
 - polynomial-fit.ipynb
 - smooth.ipynb
 - huber.ipynb

Linear least squares: 1D case with linear data

Outline

Model fitting

Linear least squares: 1D case with linear data

Linear least squares: 1D case with non-linear data

Linear least squares: general formulation and matrix-vector form

Examples

Nonlinear least squares

Beyond least squares

Deep Feedforward Networks

Stochastic methods

Problem set up

Given: *m* training examples (i.e., training set)

x_i :	independent variable	y_i : response variable
	0.0	0.46
	0.11	0.31
	0.22	0.38
	0.33	0.39
	0.44	0.65
	0.56	0.40
	0.67	0.87
	0.78	0.69
	0.89	0.87
	1.0	0.88

Goal: construct a model that can predict y from x

Where might these data come from?

x: independent variable	y: response variable
height	weight
square feet	price of home
device property	failure rate
stock market return	individual asset return

Regression can be applied regardless of the origin of the data!

Regression: Approach

Goal: construct a model that can predict y from x

- In general, we do not know the mathematical model characterizing the underlying process that actually generated the data
- So, we assume that the data were generated from a model comprising the sum of a (deterministic) function and (stochastic) iid Gaussian noise:

$$y_i = f_{\text{true}}(x_i) + \sigma \cdot \epsilon_i, \quad i = 1, \dots, m$$

with $f_{\text{true}}(x_i)$ unknown and $\epsilon_i \sim N(0, 1)$

- ▶ We aim to construct f(x) such that $f(x) \approx f_{true}(x)$ in some sense
- > This is known as *regression* and is performed via optimization
 - objective function: residual sum of squares $\frac{1}{2}\sum_{i=1}^{m}(f(x_i) y_i)^2$
 - optimization variables: parameters within the assumed form of f(x)
- ▶ Then, we can make predictions $y \approx f(x)$ for new values of x.

Follow along in Python

- See least-squares.ipynb
- ▶ In this case, we have set $f_{\rm true}(x) = \theta_{\rm true} \cdot x_i + b_{\rm true}$ and $\sigma = 0.1$

```
\bullet \ \theta_{\rm true} = 0.6
```

```
\blacktriangleright b_{\rm true} = 0.3
```

- Run the first three cells of least-squares.ipynb
- Python code to generate data (in second cell):

```
np.random.seed(1)
theta = 0.6
b = 0.3
sigma = .1
x = np.linspace(0,1,10)
y = theta*x + b + sigma*np.random.standard_normal(x.shape)
```

Plot the data



The residuals

Any given data point will result in some error or residual

$$r_i = f(x_i) - y_i$$

▶ Due to the Gaussian noise, $y_i = f_{true}(x_i) + \sigma \cdot \epsilon_i \neq f_{true}(x_i)$. Thus, the true function $f_{true}(x)$ will yield residuals

$$r_{\mathsf{true},i} = f_{\mathsf{true}}(x_i) - y_i = -\sigma \cdot \epsilon_i$$

The residuals for $f_{true}(x)$



Linear regression in one dimension

ln linear regression, we enforce the regression function f(x) to be linear

 $f(x;\theta,b) = \theta \cdot x + b$

 \blacktriangleright regression function has two parameters: the slope θ and the y-intercept b

- semicolon separates model input from model parameters
- ► Note: the form of f(x) usually does not match the (generally unknown) form of f_{true}(x). We are lucky if this happens!

Fit the model via optimization

- \blacktriangleright Given training data $(x_i,y_i)_{i=1}^m$ with $x_i\in {\bf R}$ and $y_i\in {\bf R}$
- To fit the model, construct an optimization problem

minimize
$$\phi(\theta, b) = \frac{1}{2} \sum_{i=1}^{m} r_i(\theta, b)^2 = \frac{1}{2} \sum_{i=1}^{m} (f(x_i; \theta, b) - y_i)^2$$

- Optimization objective function: residual sum of squares (RSS)
 one contribution from each of the *m* training examples
- Optimization variables: parameters θ and b
- ▶ If the true underlying model actually is $y_i = \theta_{\text{true}} \cdot x_i + b_{\text{true}} + \sigma \cdot \epsilon_i$ with ϵ_i mean-zero Gaussian, then θ and b are the maximum-likelihood estimates of θ_{true} and b_{true}

Objective function



- ▶ The objective function $\phi(\theta, b)$ is appears to be convex (it is!)
- $\blacktriangleright\,$ The global minimum occurs around $\theta^{\star}\approx 0.6$ and $b^{\star}\approx 0.35$

Optimizing by hand

Recall the sufficient conditions for (unconstrained) optimality:

1. $\nabla \phi(\theta^{\star}, b^{\star}) = 0$

- 2. $\nabla^2 \phi(\theta^{\star}, b^{\star}) \succ 0$. This holds everywhere!
 - The objective function is strongly convex
 - ▶ This simplifies things: we only need to find a stationary point satisfying condition 1
 - This is one reason why convex optimization is so nice!

Let's compute θ^{\star} and b^{\star} such that that the first condition holds.

Compute gradient analytically and set to zero

Analytical gradient computation:

$$\frac{\partial \phi}{\partial \theta} = \frac{1}{2} \sum_{i=1}^{m} \frac{\partial}{\partial \theta} (\theta \cdot x_i + b - y_i)^2 = \theta \sum x_i^2 + b \sum x_i - \sum x_i y_i$$
$$\frac{\partial \phi}{\partial b} = \frac{1}{2} \sum_{i=1}^{m} \frac{\partial}{\partial b} (\theta \cdot x_i + b - y_i)^2 = \theta \sum x_i + nb - \sum y_i$$

Set analytical gradient to zero and obtain a system of equations:

$$\frac{\partial \phi}{\partial \theta} = 0$$
$$\frac{\partial \phi}{\partial b} = 0$$

Solution

$$\theta = \frac{\sum x_i y_i - \frac{1}{m} \sum x_i \sum y_i}{\sum x_i^2 - \frac{1}{m} (\sum x_i)^2}$$
$$b = \frac{\sum y_i - \theta \sum x_i}{m}$$

Let's look at $\boldsymbol{\theta}$

Something looks nice here:

$$\theta = \frac{\sum x_i y_i - \frac{1}{m} \sum x_i \sum y_i}{\sum x_i^2 - \frac{1}{m} (\sum x_i)^2}$$

Multiply both numerator and denominator by 1/m:

$$\theta = \frac{\frac{1}{m}\sum x_i y_i - \frac{1}{m}\sum x_i \frac{1}{m}\sum y_i}{\frac{1}{m}\sum x_i^2 - (\frac{1}{m}\sum x_i)^2}$$

We see sample covariance and variance here!

$$\theta = \frac{\operatorname{cov}(X,Y)}{\operatorname{var}(X)}$$

Let's solve in Python!

Code:

```
# solve via numpy covariance function
A = np.vstack((x,y))
V = np.cov(A)
theta_est = V[0,1] / V[0,0]
b_est = (y.sum() - theta_est*x.sum()) / len(x)
print(theta_est)
print(b est)
Result:
theta est = 0.56604 (true value = 0.6)
b est = 0.30727 (true value = 0.3)
```

Look at the plot



Solve in CVXPY

Remember the optimization problem: minimize $\frac{1}{2}\sum_{i=1}^{m}(\theta \cdot x_i + b - y_i)^2$ We can write this directly in CVXPY:

```
from cvxpy import *
# Construct the problem.
theta_cvx = Variable()
b_cvx = Variable()
objective = Minimize(sum_squares(theta_cvx*x + b_cvx - y))
prob = Problem(objective)
# The optimal objective is returned by prob.solve().
result = prob.solve()
theta cvx.value = 0.56604, b cvx.value = 0.30727
```

Linear least squares: 1D case with non-linear data

Outline

Model fitting

Linear least squares: 1D case with linear data

Linear least squares: 1D case with non-linear data

Linear least squares: general formulation and matrix-vector form

Examples

Nonlinear least squares

Beyond least squares

Deep Feedforward Networks

Stochastic methods

What about these data?



▶ Here, $f_{true}(x) = \theta_{true} \exp(x) + b_{true}$, which we do not know

We just have access to the data!

We could fit a linear model

 \blacktriangleright Given our ignorance of $f_{\rm true}$, we could fit a linear model

 $f(x;\theta,b) = \theta \cdot x + b,$



▶ This yields an objective-function value of $\phi(\theta, b) = 283.63$ Linear least squares: 1D case with non-linear data

We can also fit an exponential model

If we think that the underlying model may be exponential, we can also try

 $f(x; \theta, b) = \theta \cdot \exp(x) + b$

- Model still linear in the parameters θ and b: "Linear least squares" (same optimization problem)
- But model nonlinear in the parameters: "Nonlinear regression"

CVXPY code:

```
theta = Variable()
b = Variable()
objective = Minimize(sum_squares(theta*np.exp(x) + b - y))
prob = Problem(objective)
result = prob.solve()
```

Result



- ▶ This yields a *smaller* objective-function value of $\phi(\theta, b) = 122.80$
 - better fit to training data
- Caution: can overfit training data
 - must assess generalization error on an independent test set

Linear least squares: general formulation and matrix-vector form

Linear least squares: general formulation and matrix-vector form

Outline

Model fitting

- Linear least squares: 1D case with linear data
- Linear least squares: 1D case with non-linear data

Linear least squares: general formulation and matrix-vector form

- Examples
- Nonlinear least squares
- Beyond least squares
- Deep Feedforward Networks
- Stochastic methods

General formulation for linear least squares

- ▶ $x \in \mathbf{R}^p$: p-dimensional model inputs (i.e., independent variables)
- ▶ $y \in \mathbf{R}$: model outputs (i.e., response variable)
- ▶ $f: \mathbf{R}^p \to \mathbf{R}$: model a linear combination of *n* functions $f_i: \mathbf{R}^n \to \mathbf{R}, i = 1, ..., n$:

$$f(x;\beta) = \sum_{i=1}^{n} f_i(x)\beta_i$$

- If f_i is nonlinear in x, then this is "nonlinear regression"
- Previous example: m = 2; $f_1(x) = 1$; $f_2(x) = x$ or $f_2(x) = \exp(x)$; $\beta_1 = \theta$, $\beta_2 = m$
- ▶ $\beta = (\beta_1, ..., \beta_n) \in \mathbf{R}^n$: optimization variables (i.e., model parameters)

Matrix-vector form

- Assume input-output data of the form $(x_j, y_j)_{j=1}^m$
- ▶ The residual for the *j*th data point is $r_j(\beta) = f(x_j; \beta) y_j$
- Residual sum of squares (RSS) objective function is

$$\phi(\beta) = \frac{1}{2} \sum_{j=1}^{m} r_j(\beta)^2 = \frac{1}{2} \sum_{j=1}^{m} (f(x_j; \beta) - y_j)^2 = \frac{1}{2} \sum_{j=1}^{m} (\sum_{i=1}^{n} f_i(x_j)\beta_i - y_j)^2$$

Defining

$$A = \begin{bmatrix} f_1(x_1) & \cdots & f_n(x_1) \\ \vdots & \ddots & \vdots \\ f_1(x_m) & \cdots & f_n(x_m) \end{bmatrix}, \qquad \beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \qquad b = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

we can write the objective function as $\phi(\beta) = \frac{1}{2} \|A\beta - b\|_2^2$

Linear least squares: general formulation and matrix-vector form

Standard form for least squares

minimize
$$\frac{1}{2}||Ax - b||_2^2$$

In the context of model fitting:

- $A \in \mathbf{R}^{m \times n}$ is the matrix that contains data from independent variables
- ▶ $b \in \mathbf{R}^m$ is the vector containing response data (β on last slide)

r

- $x \in \mathbf{R}^n$ is the vector of model parameters
- \blacktriangleright For each of the m training examples, the residual is we have the equation

$$r_i = a_i^T x - b_i,$$

 $\blacktriangleright \ a_i^T \in \mathbf{R}^{1 \times n} \text{ is the } i \text{th row of } A$

Notation from statistics:

$$\displaystyle egin{array}{c} \mathsf{minimize}rac{1}{2}||\mathbf{X}eta-\mathbf{y}||_2^2 \ & eta & \mathbf{b} \ & \mathbf{b} \ & eta & \mathbf{b} \ &$$

CVXPY for least squares

```
# generate input and response data
 np.random.seed(1); n = 10 \# number of data points
 input data = np.linspace(0,1,n)
 response data = 0.6*input data + 0.3 + 0.1*np.random.standard normal(n)
 # least-squares matrix and vector
 A = np.vstack([input data,np.ones(n)]).T; b = response data
 # CVX problem
 x = Variable(A.shape[1])
 objective = Minimize(sum_squares(A*x - b))
 prob = Problem(objective); result = prob.solve()
 # get value & print
 x_star = np.array(x.value)
 print('slope = {:.4}, intercept = {:.4}'.format(x star[0,0],x star[1,0]))
 slope = 0.566, intercept = 0.3073
Linear least squares: general formulation and matrix-vector form
                                                                           36
```
Examples

Outline

Model fitting

- Linear least squares: 1D case with linear data
- Linear least squares: 1D case with non-linear data
- Linear least squares: general formulation and matrix-vector form

Examples

- Nonlinear least squares
- Beyond least squares
- Deep Feedforward Networks
- Stochastic methods

What about these data?



Polynomial regression

Polynomial model:

$$y \approx f(x;\beta) = \beta_1 + \beta_2 x + \beta_3 x^2 + \dots + \beta_n x^{n-1}$$

β_i, i = 1,..., n are the model parameters and optimization variables
 Linear least-squares framework: f(x) = Σⁿ_{i=1} f_i(x)β_i with monomials

$$f_i(x) = x^{i-1}, \ i = 1, \dots, n$$

Polynomial regression

> As before, define A, β , b to put in standard form for least squares

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ 1 & x_4 & x_4^2 & \dots & x_4^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix}, \qquad \beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \vdots \\ \beta_n \end{bmatrix}, \qquad b = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_m \end{bmatrix}$$

Solve the least-squares problem

$${f minimize} \quad rac{1}{2} \|Aeta-b\|_2^2$$

 \blacktriangleright This form for A is called the Vandermonde matrix $\mbox{Examples}$

Solve with CVXPY

```
See polynomial-fit.ipynb
def cvxpy_poly_fit(x,y,degree):
    # construct data matrix
    A = np.vander(x, degree+1)
    b = v
    beta cvx = Variable(degree+1)
    # set up optimization problem
    objective = Minimize(sum_squares(A*beta_cvx - b))
    constraints = []
    # solve the problem
    prob = Problem(objective, constraints)
    prob.solve()
    # return the polynomial coefficients
    return np.array(beta cvx.value)
```

Linear fit



Quadratic fit



Cubic fit



True model



Examples his was the true (but unknown) model that generated the data

Example: time series smoothing

- See smooth.ipynb
- ▶ Noisy observations (x_i, y_i) , i = 1, ..., m at regular intervals (discretized curve)
- New modeling approach
 - ▶ We assume we **don't** have a model for the curve (linear, polynomial, ...)
 - But we do believe that the curve should be smooth
- ▶ Idea: find β_i , i = 1, ..., m that are close to y_i , but are penalized for being nonsmooth
 - Linear least squares with $f_i(x_j) = \delta_{ij}(x_j)$, i = 1, ..., m (Kronecker delta)
 - \blacktriangleright The number of optimization variables n is equal to number of data points m

Time series data



Optimization problem

- Want $\beta_i \approx y_i$, $i = 1, \dots, m$
- Want $f(x_j) = \sum_{i=1}^n \delta_{ij}(x_j)\beta_i$ to be smooth on the grid x_j , $j = 1, \dots, m$
- Optimization problem

$$\underset{\beta}{\mathsf{minimize}} \ ||\beta - b||_2^2 + \rho \cdot \mathsf{penalty}(\beta)$$

- Introduce a penalty function to encourage smoothness
- Penalty parameter ρ enables trading off two competing objectives:
 - 1. ho small: $||eta-b||_2^2$ small and model is a better fit to training data
 - 2. ρ large: penalty(β) small and model is smoother

How to quantify smoothness?

- Smoothness: a curve whose *slope does not change much*
- The second derivative measures the rate of change of the slope
- > Approximate the second derivative via second-order finite differences as $D\beta$, where

$$D = \begin{pmatrix} 1 & -2 & 1 & 0 & \dots & & 0 \\ 0 & 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & -2 & -1 & 0 & \dots & 0 \\ \vdots & & & & & & & & \end{pmatrix}$$

assuming a uniform grid x_j , $j = 1, \ldots, m$.

Least squares model

Updated optimization problem:

$$\underset{\beta}{\text{minimize}} \quad \|\beta - b\|_2^2 + \rho \|D\beta\|_2^2$$

Standard form:

minimize
$$\left\| \begin{pmatrix} I \\ \rho D \end{pmatrix} \beta - \begin{pmatrix} b \\ 0 \end{pmatrix} \right\|_{2}^{2}$$

Solve the problem in $\ensuremath{\mathsf{CVXPY}}$

 $\rho = 1$



 $\rho = 10$



 $\rho = 1000$



Nonlinear least squares

Outline

Model fitting

- Linear least squares: 1D case with linear data
- Linear least squares: 1D case with non-linear data
- Linear least squares: general formulation and matrix-vector form
- Examples

Nonlinear least squares

- Beyond least squares
- Deep Feedforward Networks
- Stochastic methods

Nonlinear least squares

Linear least squares:

1. Model is *linear in the parameters*

$$f(x;\beta) = \sum_{i=1}^{n} \beta_i f_i(x)$$

- Linear regression: f_i is also linear in x
- ▶ Nonlinear regression: f_i is nonlinear in x (e.g., polynomials, exponential)
- 2. Minimize the residual sum of squares (RSS)

Nonlinear least squares:

- 1. Model $f(x;\beta)$ is nonlinear in the parameters β
- 2. Minimize the same objective function: residual sum of squares (RSS)
 - Again equivalent to maximum likelihood if additive Gaussian noise
 - Algorithms: line-search (Gauss-Newton) and trust-region (Levenberg-Marquardt)

Nonlinear least squares

Beyond least squares

Outline

Model fitting

- Linear least squares: 1D case with linear data
- Linear least squares: 1D case with non-linear data
- Linear least squares: general formulation and matrix-vector form
- Examples
- Nonlinear least squares

Beyond least squares

- Deep Feedforward Networks
- Stochastic methods

Quadratic loss function

- See huber.ipynb
- Least squares employs a quadratic loss function



This function imposes a severe penalty on large values

► As a result, the fit model is very sensitive to outliers (can overfit) Beyond Gant we use a different loss function?

Huber loss function

- ▶ The Huber function allows us to better handle outliers in data
 - Usual quadratic loss in interval [-M, M]
 - Linear loss for |x| > M

$$h_M(x) = \begin{cases} x^2 & |x| \le M\\ 2M|x| - M^2 & |x| > M \end{cases}$$



Beyond least squares

Huber loss function

- This function imposes a less severe penalty on large values
- Let's repeat the time-series example, but include extreme outliers
- \blacktriangleright Penalize closeness to data with Huber function h_M to reduce outlier influence:

minimize
$$\sum_{i=1}^{m} h_M(\beta_i - y_i) + \rho \|D\beta\|_2^2$$

- \blacktriangleright M parameter controls width of quadratic region, or "non-outlier" errors
- This is no longer least squares!
- CVXPY has implemented the Huber loss function

Huber data



```
Least-squares smoothing
```

```
# get second-order difference matrix
D = diff(n, 2)
rho = 20
```

```
beta = Variable(n)
obj = sum_squares(beta-b) + rho*sum_squares(D*beta)
Problem(Minimize(obj)).solve()
beta = np.array(beta.value).flatten()
```

Least-squares smoothing result



Model overfits the outliers

Beyond least squares

Huber smoothing

```
# get second-order difference matrix
D = diff(n, 2)
rho = 20
M = .15 # huber radius
```

```
beta = Variable(n)
obj = sum_entries(huber(beta-b, M)) + rho*sum_squares(D*beta)
Problem(Minimize(obj)).solve()
x = np.array(x.value).flatten()
```

Huber smoothing result



The model is less sensitive to outliers!

Beyond least squares

Deep Feedforward Networks

Outline

Model fitting

- Linear least squares: 1D case with linear data
- Linear least squares: 1D case with non-linear data
- Linear least squares: general formulation and matrix-vector form
- Examples
- Nonlinear least squares
- Beyond least squares
- Deep Feedforward Networks
- Stochastic methods

Deep Feedforward Networks

A deep feedforward network defineds a particular model $f(x;\beta)$

- f(x; β) = f⁽³⁾(f⁽²⁾(f⁽¹⁾(x; β₁); β₂); β₃) is a 'network' (function composition)
 f⁽ⁱ⁾(x; β_i): function charactering the *i*th layer with parameters β_i
 - parameters $\beta = (\beta_1, \beta_2, \beta_3) \in \mathbf{R}^n$
- Evaluating f is 'forward propagation': start at the beginning $(f^{(1)})$ and evaluate forward sequentially
- \blacktriangleright It is 'deep' if there are many composed functions, and thus β is high-dimensional
- f is genearly nonlinear in the parameters β
- ▶ if additive Gaussian noise, then MLE leads to nonlinear least squares
- ▶ other loss functions possible (e.g., non-Gaussian noise); then no longer least squares

Deep Feedforward Networks

Computing the gradient can be done by applying the chain rule, e.g.,

$$\frac{\partial \phi}{\partial \beta_2} = \frac{\partial \phi}{\partial f^{(3)}} \frac{\partial f^{(3)}}{\partial x} \frac{\partial f^{(2)}}{\partial \beta_2}, \quad \frac{\partial \phi}{\partial \beta_1} = \frac{\partial \phi}{\partial f^{(3)}} \frac{\partial f^{(3)}}{\partial x} \frac{\partial f^{(2)}}{\partial x} \frac{\partial f^{(1)}}{\partial \beta_1}$$

Computing the gradient is referred to as **back propagation**: the chain rule 'propagates' information from the end of the network (f⁽³⁾) upstream (e.g., to f⁽¹⁾)
Deep Feedforward Networks: optimization challenges in optimization

minimize
$$\phi(\beta) = \frac{1}{2} \sum_{i=1}^{m} (f(x_i; \beta) - y_i)^2$$

High-dimensional

- many β parameters n (due to many layers)
- \blacktriangleright many training samples m and (need lots of data to tune many parameters)
- **solution**: stochastic/minibatch methods (e.g., stochastic gradient descent)

Non-convex

- can get trapped in local minima
- **solution**: local minima seem to yield a "low-enough" cost-function value

III conditioning

solution: second-order methods (but hard for NNs)

Deep Feedforward Networks

Stochastic methods

Outline

Model fitting

- Linear least squares: 1D case with linear data
- Linear least squares: 1D case with non-linear data
- Linear least squares: general formulation and matrix-vector form
- Examples
- Nonlinear least squares
- Beyond least squares
- Deep Feedforward Networks

Stochastic methods

What does 'Big Data' mean for model fitting?

In model fitting, the objective function is usually composed of a sum of m contributions:

$$\phi(\beta) = \frac{1}{m} \sum_{i=1}^{m} \phi_i(\beta)$$

- ϕ_i : is the loss associated with the *i*th training example
- ϕ : a sampling-based approximation of the expected loss
- 'Big Data' can refer to:
 - many training examples: m large
 - many parameters: n large
 - deep learning falls in this category!
- Specialized methods have been developed for these cases!
 - stochastic/minibatch methods (next)
 - distributed optimization (see 'Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers' by Boyd et al.)

Stochastic methods

Here, the gradient is also a sum of m contributions:

$$\nabla \phi(\beta) = \frac{1}{m} \sum_{i=1}^{m} \nabla \phi_i(\beta)$$

- Batch methods use this within gradient-based optimization
- **Benefit**: Preserves traditional convergence rates
- Drawbacks:
 - Requires accessing all m data points each iteration (costly)
 - Many data points are likely *redundant*
- Can we make this less expensive yet still maintain convergence?

Observations:

- 1. The objective is (usually) just the sample mean of the loss function
- 2. Expectations via Monte Carlo sampling converge slowly (rate $m^{-1/2}$)
- 3. Exact gradients aren't needed for convergence

Stochastic methods

Idea: inexpensively approximate the gradient with a *sample* of the data

Stochastic methods

Stochastic methods: compute approximate the gradient as

 $\nabla \phi(\beta) \approx \nabla \phi_i(\beta)$

- i is a randomly chosen training example
- Stochastic gradient descent (SGD): stochastic approximation to gradient descent:

$$x_{i+1} = x_k - \alpha_k \nabla \phi_i(\beta)$$

Benefits:

- each iteration is much cheaper
- often observe faster rate of convergence as a function of accessed data points
- ▶ a descent direction in expectation, i.e., $\mathbb{E}[\nabla \phi_i(\beta)] = \nabla \phi(\beta)$
- Drawbacks
 - slower rate of convergence as a function of iteration (sublinear for SGD)
 - observed slowdown as iterations progress due to noisy gradients

SGD performance in practice



Fig. 3.1 Empirical risk R_n as a function of the number of accessed data points (ADPs) for a batch L-BFGS method and the SG method (3.7) on a binary classification problem with a logistic loss objective and the RCV1 dataset. SG was run with a fixed stepsize of $\alpha = 4$.

Reference: Bottou, L., Curtis, F.E. and Nocedal, J., 2018. Optimization methods for large-scale machine learning. SIAM Review, 60(2), pp.223-311. Stochastic methods

Improving the convergence rate of stochastic methods

Noise reduction: reduce variance gradient estimate

Dynamic sampling: use minibatch estimates of the gradient at iteration k

$$\nabla \phi(\beta) \approx \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} \nabla \phi_i(\beta),$$

where the minibatch size $|S_k|$ increases with k.

- Gradient aggregation: reuse recently computed gradient information
 - Example: stochastic variance reduced gradient (SVRD):

$$\nabla \phi(\beta) \approx \nabla \phi_i(\beta) - (\nabla \phi_i(\bar{\beta}) - \nabla \phi(\bar{\beta}))$$

 \blacktriangleright $\bar{\beta}$: variables the last time the true batch gradient was computed

Improving the convergence rate of stochastic methods

Second-order methods: use sampled Hessian information

Subsampled Hessian-Free Newton Methods: minibatch estimate of the Hessian

$$\nabla^2 \phi(\beta) \approx \frac{1}{|\mathcal{S}_k^H|} \sum_{i \in \mathcal{S}_k^H} \nabla^2 \phi_i(\beta)$$

Can also enforce positive definiteness via subsampled Gauss-Newton approximations

Subsampled Quasi-Newton Methods:

typical quasi-Newton methods with stochastic estimates of the gradient