Applied Machine Learning

Gradient Descent Methods

Siamak Ravanbakhsh

COMP 551 (winter 2020)
Learning objectives

Basic idea of

• gradient descent
• stochastic gradient descent
• method of momentum
• using adaptive learning rate
• sub-gradient

Application to

• linear regression and classification
Optimization in ML

Inference and learning of a model often involves optimization:
optimization is a huge field

**bold:** the setting considered in this class

- discrete (combinatorial) vs **continuous variables**
- constrained vs **unconstrained**
- for continuous optimization in ML:
  - **convex** vs non-convex
  - looking for **local** vs global optima?
  - **analytic gradient**?
  - analytic Hessian?
  - **stochastic** vs batch
  - **smooth** vs non-smooth
Gradient

for a multivariate function $J(w_0, w_1)$

partial derivatives instead of derivative

= derivative when other vars. are fixed

$$\frac{\partial}{\partial w_1} J(w_0, w_1) \triangleq \lim_{\epsilon \to 0} \frac{J(w_0, w_1 + \epsilon) - J(w_0, w_1)}{\epsilon}$$

we can estimate this numerically if needed
(use small epsilon in the formula above)

**gradient:** vector of all partial derivatives

$$\nabla J(w) = \left[ \frac{\partial}{\partial w_1} J(w), \cdots \frac{\partial}{\partial w_D} J(w) \right]^T$$
Gradient descent

an iterative algorithm for optimization

- starts from some $w^{0}$
- update using gradient $w^{t+1} \leftarrow w^{t} - \alpha \nabla \mathcal{J}(w^{t})$

converges to a local minima

\[ \nabla \mathcal{J}(w) = \left[ \frac{\partial}{\partial w_1} \mathcal{J}(w), \cdots, \frac{\partial}{\partial w_d} \mathcal{J}(w) \right]^T \]

Convex function

A convex subset of $\mathbb{R}^N$ intersects any line in at most one line segment.

A convex function is a function for which the epigraph is a convex set.

**Epigraph:** set of all points above the graph.

$$f(\lambda w + (1 - \lambda)w') \leq \lambda f(w) + (1 - \lambda)f(w') \quad 0 < \lambda < 1$$
Convex function

Convex functions are easier to minimize:

- critical points are global minimum
- gradient descent can find it

$$w^{t+1} \leftarrow w^t - \alpha \nabla J(w^t)$$

Convex

non-convex: gradient descent may find a local optima

a **concave** function is a negative of a convex function (easy to **maximize**)
Recognizing convex functions

A linear function is convex $w^T x$

Convex if second derivative is positive everywhere $\frac{d^2}{dx^2} f \geq 0$

Example $x^2, e^x, -\log(x), -\sqrt{x}$

Sum of convex functions is convex

Example $\|WX - Y\|_2^2 + \lambda \|w\|_2^2$

Maximum of convex functions is convex

Example $f(y) = \max_{x \in [1, 5]} \sqrt{xy^4}$

Note this is not convex in $x$

Composition of convex functions is generally not convex

Example $(-\log(x))^2$

However, if $f, g$ are convex, and $g$ is non-decreasing $g(f(x))$ is convex

Example $e^{f(x)}$ for convex $f$
Gradient for linear and logistic regression

in both cases: \[ \nabla J(w) = X^T(\hat{y} - y) \]

linear regression: \[ \hat{y} = Xw \]
logistic regression: \[ \hat{y} = \sigma(Xw) \]

time complexity: \[ \mathcal{O}(ND) \]
(two matrix multiplications)

compared to the direct solution for linear regression: \[ \mathcal{O}(ND^2 + D^3) \]

\begin{verbatim}
import numpy as np

def gradient(X, y, w):
    N, D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    return grad

1  def gradient(X, y, w):
2      N,D = X.shape
3      yh = logistic(np.dot(X, w))
4      grad = np.dot(X.T, yh - y) / N
5      return grad
\end{verbatim}
Gradient Descent

implementing gradient descent is easy!

```python
def GradientDescent(X, # N x D
y, # N
lr=.01, # learning rate
eps=1e-2, # termination condition
):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w
```

Some termination conditions:
- some max #iterations
- small gradient
- a small change in the objective
- increasing error on validation set

**early stopping** (one way to avoid overfitting)
Example: GD for Linear Regression

applying this to fit toy data
**Example: GD for Linear Regression**

applying this to fit toy data

single feature (intercept is zero)

\[ y = -3x + (x^{(n)} - 3x^{(n)} + \text{noise}) \]

using direct solution method

\[ w = (X^T X)^{-1} X^T y \approx -3.2 \]

\[ y = wx \]

\[ y = -3x \]
**Example: GD for Linear Regression**

After 22 iterations of Gradient Descent

\[ w^{(t+1)} \leftarrow w^{(t)} - 0.01 \nabla J(w^{(t)}) \]

**cost function**

\[ J(w) \]

\[ w^{(0)} \neq 0 \]

\[ w^{(22)} \approx -3.2 \]

**data space**

\[ y = w^0 x \]
Learning rate $\alpha$

Learning rate has a significant effect on GD

- too small: may take a long time to converge
- too large: it overshoots

$$J(w)$$

$\alpha = .01$

$\alpha = .05$
GD for logistic Regression

**example**: logistic regression for Iris dataset (D=2, lr=.01)

```python
def GradientDescent(X, # N x D
                     y, # N
                     lr=.01, # learning rate
def gradient(X, y, w):
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y)
    return grad
```
**Stochastic Gradient Descent**

We can write the cost function as an average over instances:

$$J(w) = \frac{1}{N} \sum_{n=1}^{N} J_n(w)$$

Cost for a single data-point, e.g. for linear regression:

$$J_n(w) = \frac{1}{2} (w^T x^{(n)} - y^{(n)})^2$$

The same is true for the partial derivatives:

$$\frac{\partial}{\partial w_j} J(w) = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial}{\partial w_j} J_n(w)$$

Therefore:

$$\nabla J(w) = \mathbb{E}[\nabla J_n(w)]$$
**Stochastic Gradient Descent**

Idea: use stochastic approximations $\nabla J_n(w)$ in gradient descent

Contour plot of the cost function + batch gradient update $w \leftarrow w - \alpha \nabla J(w)$

With small learning rate: **guaranteed** improvement at each step

![Image](https://jaykanidan.wordpress.com)
**Stochastic Gradient Descent**

Idea: use stochastic approximations \( \nabla J_n(w) \) in gradient descent using stochastic gradient \( w \leftarrow w - \alpha \nabla J_n(w) \)

the steps are "on average" in the right direction

\[
\nabla J_n(w) = x^{(n)}(w^T x^{(n)} - y^{(n)})
\]

each step is using gradient of a different cost \( J_n(w) \)
each update is (1/N) of the cost of batch gradient

e.g., for linear regression \( O(D) \)

\text{image:https://jaykanidan.wordpress.com}
Example: SGD for logistic regression

setting 1: using batch gradient

logistic regression for Iris dataset (D=2, $\alpha = .1$)

def GradientDescent(X, # N x D
                     y, # N
                     lr=.01, # learning rate
                     eps=1e-2, # termination condition
                     ):
    N,D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        g = gradient(X, y, w)
        w = w - lr*g
    return w

def gradient(X, y, w):
    N, D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    return grad

$w^t = (0, 0)$ after 8000 iterations
Example: SGD for logistic regression

setting 2: using stochastic gradient

logistic regression for Iris dataset (D=2, $\alpha = .1$)
Convergence of SGD

stochastic gradients are not zero at optimum
how to guarantee convergence?

schedule to have a smaller learning rate over time

Robbins Monro

the sequence we use should satisfy: \( \sum_{t=0}^{\infty} \alpha^t = \infty \)
otherwise for large \( \|w^{(0)} - w^*\| \) we can’t reach the minimum
the steps should go to zero \( \sum_{t=0}^{\infty} (\alpha^t)^2 < \infty \)

example \( \alpha^t = \frac{10}{t}, \alpha^t = t^{-0.51} \)
Minibatch SGD

use a minibatch to produce gradient estimates

$$\nabla J_B = \sum_{n \in B} \nabla J_n (w)$$

$B \subseteq \{1, \ldots, N\}$ a subset of the dataset

def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8):
    N, D = X.shape
    w = np.zeros(D)
    g = np.inf
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch, :], y[minibatch], w)
    return w
Momentum

to help with oscillations of SGD (or even full-batch GD):

- use a running average of gradients
- more recent gradients should have higher weights

\[
\Delta w^{\{t\}} \leftarrow \beta \Delta w^{\{t-1\}} + (1 - \beta) \nabla J_B(w^{\{t\}})
\]

\[
w^{\{t\}} \leftarrow w^{\{t-1\}} - \alpha \Delta w^{\{t\}}
\]

is effectively an exponential moving average

\[
\Delta w^{\{T\}} = \sum_{t=1}^{T} \beta^{T-t} (1 - \beta) \nabla J_B(w^{\{t\}})
\]

momentum of 0 reduces to SGD
common value > .9

there are other variations of momentum with similar idea
Momentum

to help with oscillations of SGD (or even full-batch GD):

- use a running average of gradients
- more recent gradients should have higher weights

```python
def MinibatchSGD(X, y, lr=.01, eps=1e-2, bsize=8, beta=.99):
    N, D = X.shape
    w = np.zeros(D)
    g = np.inf
    dw = 0
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch,:], y[minibatch], w)
        dw = (1-beta)*g + beta*dw
        w = w - lr*dw
    return w
```
Momentum

Example: logistic regression

\[ \alpha = 0.5, \beta = 0, |B| = 8 \]

\[ \Delta w^t \leftarrow \beta \Delta w^{t-1} + (1 - \beta) \nabla J_B(w^{t-1}) \]

\[ w^t \leftarrow w^{t-1} - \alpha \Delta w^t \]

see the beautiful demo at Distill

https://distill.pub/2017/momentum/
**Adagrad** *(Adaptive gradient)*

use different learning rate for each parameter $\mathbf{w}_d$
also make the learning rate adaptive

$$S_d^{(t)} \leftarrow S_d^{(t-1)} + \frac{\partial}{\partial w_d} J(w^{(t-1)})^2$$

sum of squares of derivatives over all iterations so far (for individual parameter)

$$\mathbf{w}_d^{(t)} \leftarrow \mathbf{w}_d^{(t-1)} - \frac{\alpha}{\sqrt{S_d^{(t-1)} + \epsilon}} \frac{\partial}{\partial w_d} J(w^{(t-1)})$$

the learning rate is adapted to previous updates
$\epsilon$ is to avoid numerical issues

useful when parameters are updated at different rates *(e.g., NLP)*
Adagrad (Adaptive gradient)

different learning rate for each parameter $\omega_d$
make the learning rate adaptive

$$\alpha = .1, |B| = 1, T = 80,000$$
$$\alpha = .1, |B| = 1, T = 80,000, \epsilon = 1e-8$$

**problem:** the learning rate goes to zero too quickly
RMSprop
(Root Mean Squared propagation)

solve the problem of diminishing step-size with Adagrad

- use **exponential moving average** instead of sum (similar to momentum)

\[
S^{(t)} \leftarrow \gamma S^{(t-1)} + (1 - \gamma) \nabla J(w^{(t-1)})^2
\]

\[
w^{(t)} \leftarrow w^{(t-1)} - \frac{\alpha}{\sqrt{S^{(t-1)} + \epsilon}} \nabla J(w^{(t-1)})
\]

identical to Adagrad

```python
def RMSprop(X, y, lr=.01, eps=1e-2, bsize=8, gamma=.9, epsilon=1e-8):
    N, D = X.shape
    w = np.zeros(D)
    g = np.inf
    S = 0
    while np.linalg.norm(g) > eps:
        minibatch = np.random.randint(N, size=(bsize))
        g = gradient(X[minibatch, :], y[minibatch], w)
        S = (1-gamma)*g**2 + gamma*S
        w = w - lr*g/np.sqrt(S + epsilon)
    return w
```
Adam (Adaptive Moment Estimation)

two ideas so far:
1. use momentum to smooth out the oscillations
2. adaptive per-parameter learning rate

both use exponential moving averages

Adam combines the two:

\[
\begin{align*}
M^{(t)} &\leftarrow \beta_1 M^{(t-1)} + (1 - \beta_1) \nabla J(w^{(t-1)}) \quad \text{(moving average of the first moment)} \\
S^{(t)} &\leftarrow \beta_2 S^{(t-1)} + (1 - \beta_2) \nabla J(w^{(t-1)})^2 \quad \text{(moving average of the second moment)} \\
w^{(t)} &\leftarrow w^{(t-1)} - \frac{\alpha M^{(t)}}{\sqrt{S^{(t)}} + \epsilon} \nabla J(w^{(t-1)})
\end{align*}
\]
Adam (Adaptive Moment Estimation)

Adam combines the three:

\[
M^{(t)} \leftarrow \beta_1 M^{(t-1)} + (1 - \beta_1) \nabla J(w^{(t-1)}) \quad \text{identical to method of momentum}
\]

\[
S^{(t)} \leftarrow \beta_2 S^{(t-1)} + (1 - \beta_2) \nabla J(w^{(t-1)})^2 \quad \text{identical to RMSProp}
\]

\[
w^{(t)} \leftarrow w^{(t-1)} - \frac{\alpha M^{(t)}}{\sqrt{\hat{S}^{(t)}} + \epsilon} \nabla J(w^{(t-1)})
\]

since M and S are initialized to be zero, at early stages they are biased towards zero

\[
\hat{M}^{(t)} \leftarrow \frac{M^{(t)}}{1 - \beta_1^t} \quad \hat{S}^{(t)} \leftarrow \frac{S^{(t)}}{1 - \beta_2^t}
\]

for large time-steps it has no effect

for small t, it scales up numerator
In practice

the list of methods is growing ...

they have recommended range of parameters

- learning rate, momentum etc.

still may need some hyper-parameter tuning

these are all **first order methods**

- they only need the first derivative
- 2nd order methods can be much more effective, but also much more expensive

image:Alec Radford
Adding $L_2$ regularization

- do not penalize the bias $w_0$
- L2 penalty makes the optimization easier too!
- note that the optimal $w_1$ shrinks

```python
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * w[1:]
    return grad
```

weight decay
Subgderivatives

L1 penalty is no longer smooth or differentiable (at 0) extend the notion of derivative to non-smooth functions

**sub-differential** is the set of all **sub-derivatives** at a point

\[
\partial f(\hat{w}) = \left[ \lim_{w \to \hat{w}^-} \frac{f(w) - f(\hat{w})}{w - \hat{w}}, \lim_{w \to \hat{w}^+} \frac{f(w) - f(\hat{w})}{w - \hat{w}} \right]
\]

if \( f \) is differentiable at \( \hat{w} \) then sub-differential has one member \( \frac{d}{dw} f(\hat{w}) \)

another expression for sub-differential

\[
\partial f(\hat{w}) = \{ g \in \mathbb{R} \mid f(w) > f(\hat{w}) + g(w - \hat{w}) \}
\]
**Subgradient**

Subgradient is a vector of sub-derivatives. Recall, **gradient** was the vector of **partial derivatives**. **Subgradient** is a vector of **sub-derivatives**.

Subdifferential for functions of multiple variables:

\[ \partial f(\hat{w}) = \left\{ g \in \mathbb{R}^D \mid f(w) > f(\hat{w}) + g^T (w - \hat{w}) \right\} \]

We can use sub-gradient with diminishing step-size for optimization.
Adding $L_1$ regularization

L1-regularized *linear regression* has efficient solvers
subgradient method for L1-regularized logistic regression
do not penalize the bias $w_0$
using *diminishing learning rate*
note that the optimal $w_1$ becomes 0

```
def gradient(X, y, w, lambdaa):
    N,D = X.shape
    yh = logistic(np.dot(X, w))
    grad = np.dot(X.T, yh - y) / N
    grad[1:] += lambdaa * np.sign(w[1:]),
    return grad
```

$\lambda = 0$

$\lambda = .1$

$\lambda = 1$
Summary

learning: optimizing the model parameters (minimizing a cost function)
use gradient descent to find local minimum
  • easy to implement (esp. using automated differentiation)
  • for convex functions gives global minimum

Stochastic GD: for large data-sets use mini-batch for a noisy-fast estimate of gradient
  • Robbins Monro condition: reduce the learning rate to help with the noise
  • Momentum: exponential running average to help with the noise
  • Adagrad & RMSProp: per parameter adaptive learning rate
  • Adam: combining these two ideas

Adding regularization can also help with optimization
Adadelta

solve the problem of diminishing step-size with Adagrad

- use exponential moving average instead of sum (similar to momentum)
also gets rid of a "learning rate" altogether
- use another moving average for that!

\[
\begin{align*}
S^{\{t\}} & \leftarrow \gamma S^{\{t-1\}} + (1 - \gamma) \nabla J(w^{\{t-1\}})^2 & \text{moving average of the sq. gradient} \\
U^{\{t\}} & \leftarrow \gamma U^{\{t-1\}} + (1 - \gamma) \Delta w^{\{t-1\}} & \text{moving average of the sq. updates} \\
\Delta w^{\{t\}} & \leftarrow -\sqrt{\frac{U^{\{t-1\}}}{S^{\{t\}} + \epsilon}} \nabla J(w^{\{t-1\}}) & \text{square root of the ratio of the above is used as the adaptive learning rate} \\
w^{\{t\}} & \leftarrow w^{\{t-1\}} + \Delta w^{\{t\}}
\end{align*}
\]