M2177.0043 Introduction to Deep Learning

Lecture 5: Optimization[1]

Hyun Oh Song[1]

[1] Dept. of Computer Science and Engineering, Seoul National University

March 31, 2020

[1] Many slides and figures adapted Justin Johnson

Last time

- Constrained optimization
- Online method

Outline

Subgradient

Not all functions are differentiable. The absolute value $f(x) = |x|$, for example, is convex but not differentiable at 0.

Definition 1 (Subgradients)

Vector g is a **subgradient** of f at $x \in \operatorname{dom} f$ if

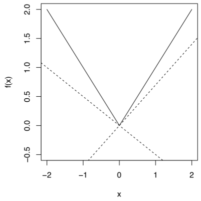
$$f(y) \geqslant f(x) + g^\top (y - x) \ \forall y \in \operatorname{dom} f$$

The set of all subgradients at x is called the **subdifferential** at x and is denoted as $\partial f(x)$.

The existence of subgradients is often sufficient for optimization.

Examples[2]

Consider $f : \mathbb{R} \to \mathbb{R}, f(x) = |x|$

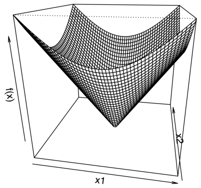


- For $x \neq 0, |y| \geqslant |x| + g(y - x), \forall y$. Or $|y| - gy \geqslant |x| - gx$. Satisfies if $|x| - gx = 0$. This means $g = sign(x)$.

- For $x = 0$, inequality simplifies to $|y| \geqslant gy$. This means $g \in [-1, 1]$.

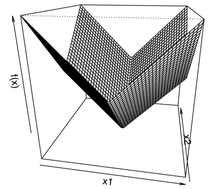
[2]taken from Tibshirani 10-725

Consider $f : \mathbb{R}^n \to \mathbb{R}, f(x) = \|x\|_2$



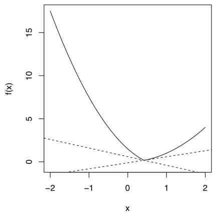
- For $x \neq 0$, differentiable $g = \frac{x}{\|x\|}$
- For $x = 0$, inequality simplifies to $\|y\|_2 \geqslant g^\top y$. So $\|y\|_2 \geqslant \|g\|_2 \|y\|_2 \cos\theta$. $g \in \{z \mid \|z\|_2 \leqslant 1\}$

Consider $f : \mathbb{R}^n \to \mathbb{R}, f(x) = \|x\|_1$



- For $x \neq 0$, differentiable $g_i = sign(x_i)$
- For $x = 0$, inequality simplifies to $\|y\|_1 \geqslant g^\top y$. So $g_i \in [-1, 1]$.

Let $f_1, f_2 : \mathbb{R}^n \to \mathbb{R}$, be convex differentiable and consider $f(x) = \max\{f_1(x), f_2(x)\}$



- For $f_1(x) > f_2(x)$, $g = \nabla f_1(x)$
- For $f_2(x) > f_1(x)$, $g = \nabla f_2(x)$
- For $f_1(x) = f_2(x)$, subgradient g is any point on the line segment between $\nabla f_1(x)$ and $\nabla f_2(x)$. Concretely, $g \in \{\theta_1 \nabla f_1(x) + \theta_2 \nabla f_2(x) \mid \theta_1 + \theta_2 = 1, \theta_1 \geqslant 0, \theta_2 \geqslant 0\}$

Outline

Stochastic gradient descent

- Consider minimizing an objective function that has the form of a sum of functions:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

- Each summand function $f_i(x)$ is typically associated with the i-th observation in a dataset. The standard (or "batch") gradient descent method would perform the following iterations:

$$x := x - \eta \nabla f(x) = x - \eta \sum_{i=1}^{n} \frac{1}{n} \nabla f_i(x)$$

- Computing the gradient can be very expensive if n is large. Stochastic gradient descent method *samples* the summand functions at every step for scalability.

$$x := x - \eta \boxed{\nabla f_i(x)}$$

Sanity check[3]

- Let us check that on a simple problem that the stochastic gradient descent yields the optimum. Let $p_1, \dots, p_m \in \mathbb{R}^n$, and define $f\colon \mathbb{R}^n \to \mathbb{R}_+$:

$$\forall x \in \mathbb{R}^n,\ f(x) = \frac{1}{2m} \sum_{i=1}^{m} \|x - p_i\|_2^2$$

- Note that here $f_i(x) = \frac{1}{2}\|x - p_i\|_2^2$ and $\nabla f_i(x) = x - p_i$. Moreover,

$$x^* = \operatorname*{argmin}_{x \in \mathbb{R}^d} f(x) = \frac{1}{m} \sum_{i=1}^{m} p_i$$

[3]Taken from Moritz Hardt's lecture

- Now, run SGM with $\eta_t = \frac{1}{t}$ in cyclic order i.e. $i_t = t$ and $x_0 = 0$:

$$\begin{aligned}
x_0 &= 0 \\
x_1 &= 0 - \frac{1}{1}(0 - p_1) = p_1 \\
x_2 &= p_1 - \frac{1}{2}(p_1 - p_2) = \frac{p_1 + p_2}{2} \\
&\vdots \\
x_m &= \frac{1}{m}\sum_{i=1}^{m} p_i = x^*
\end{aligned}$$

Minibatch SGD

- A compromise between computing the true gradient and the stochastic gradient of a single example
- Reduces the variance of the gradient estimate
- In practice, vector-process the gradient computation so the minibatch size fits in the memory

Minibatch SGD

Minibatch stochastic gradient descent

```
given a starting point $x \in \mathrm{dom} f$.
repeat
    1. Shuffle the data
    2. For every m sequence of data $i = 1, \ldots, \lceil n/m \rceil$
        a. Compute the minibatch gradient. $\Delta x = \frac{1}{m} \sum_{j=1}^{m} \nabla f_j(x)$
        b. Update the stepsize. $t := Update(t)$.
        c. Update. $x := x + t\Delta x$.
    3. Decay the stepsize
until stopping criterion is satisfied.
```

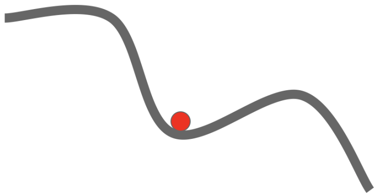
Distributed gradient descent for full batch gradient descent

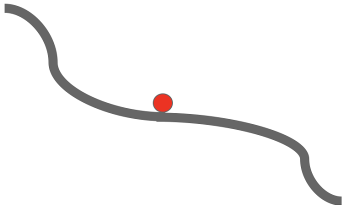
- **Map:** compute gradient on subblock and emit
  Reduce: aggregate parts of the gradients
-

```
given a starting point $x \in \mathrm{dom} f$.
repeat
    1. Compute the full gradient $\Delta x_{\mathsf{ds}} := \boxed{\sum_{i=1}^{n} \frac{1}{n} \nabla f_i(x).}$
    2. Line search. Choose a step size $t > 0$.
    3. Update. $x := x + t\Delta x_{\mathsf{ds}}$.
until stopping criterion is satisfied.
```

Problems in non-convex optimization

- What if the loss function has a local minima or saddle point?
- Zero gradient, gradient descent gets stuck





Momentum

- Think about roll a ball down a hill
- Build up "velocity" as a running mean of gradients

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha \boxed{v_{t+1}}$$

Typically set $\rho = 0.9$ or 0.99.

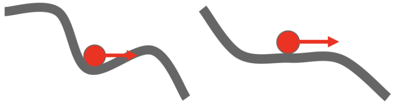


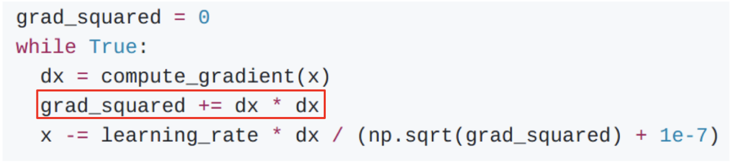
Figure: (Left) local minima, (Right) Saddle points

AdaGrad

- Adapt the learning rate to each parameter dimensions. Larger updates for infrequent and smaller updates for frequent parameters.
- Keep a diagonal matrix $G_t \in \mathbb{R}^{d \times d}$ where each diagonal element is the sum of squares of the gradients up to time step t.

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

- ϵ is a small constant ($\approx 1e-8$) to avoid divide by zero.

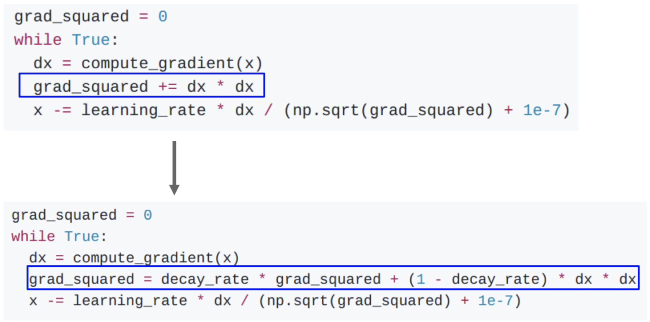


```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

- What happens to the denominator if you keep adding positive values?

RMSProp

- As opposed to accumulating all previous squared gradients, restrict the attention to recent gradients of some time window. Divide the learning rate by an exponentially decaying average of squared gradients
- Default $\gamma = 0.9, \eta = 0.001$ (by Geoff Hinton)



AdaGrad

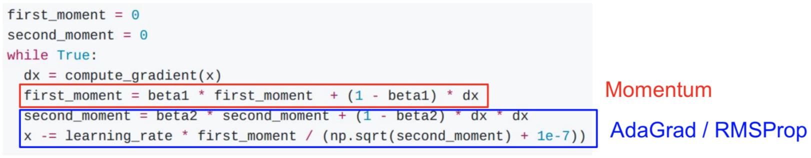
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam

- Combine momentum and AdaGrad/RMSProp

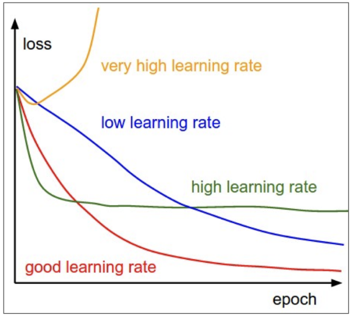


```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Sort of like RMSProp with momentum

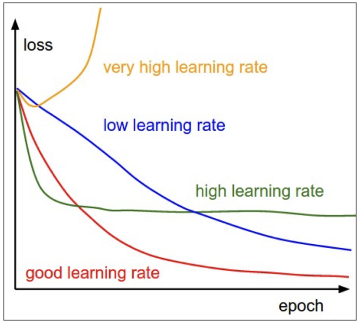
- What happens at first time step?

Learning rate (step size) as a hyperparameter



Q: Which one of these learning rates is best to use?

Learning rate (step size) as a hyperparameter



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Second-order optimization

- Quasi-Newton methods (BFGS most popular) Instead of inverting the Hessian ($O(n^3)$), approximate inverse Hessian with rank 1 updates over time ($O(n^2)$).
- L-BFGS (Limited memory BFGS) Does not form/store the full Hessian (would need $O(n^2)$ space). Instead, only retain recent m gradients (around 10 to 20). Hence the name *Limited memory*.
- Usually works very well in full batch. Does not transfer well to mini-batch setting. Gives bad results. Adapting second order optimization to large scale stochastic setting is an active area of research.

In practice

- **Adam** is a good default choice in most cases
- If you can afford to do full batch updates then try out **L-BFGS**.