Lecture 12 Training Neural Networks

Shiwei Lan[1]

[1]School of Mathematical and Statistical Sciences
Arizona State University

STP598 Machine Learning and Deep Learning
Fall 2021

Table of Contents

Gradient Descent Optimization

- Most deep learning algorithms involve optimization of some sort.
- Training neural network often relies on minimizing some objective (cost/loss) function $f(x)$.
- To minimize f, we would like to find the direction u in which f decreases the fastest by using the directional derivative:

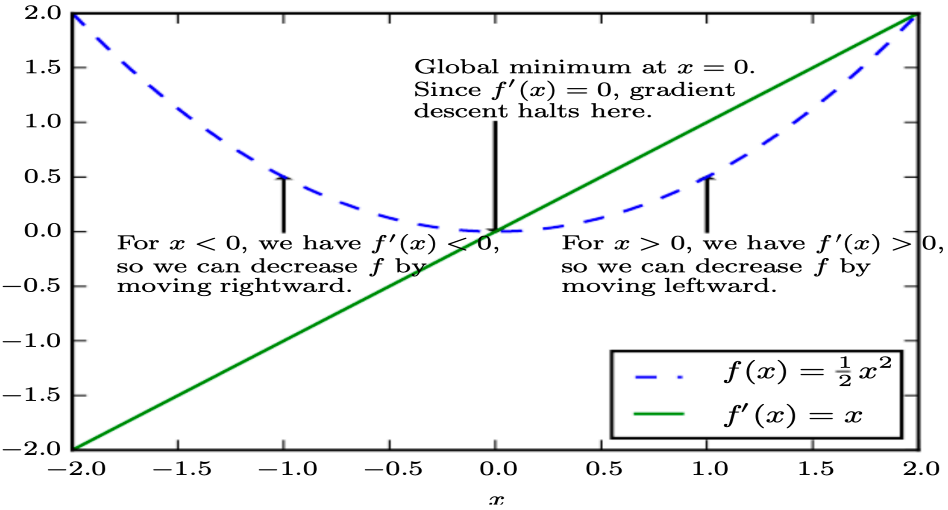
$$\min_{u,\|u\|=1} \frac{\partial}{\partial \alpha}\Big|_{\alpha=0} f(x+\alpha u) = \min_{u,\|u\|=1} u^T \nabla_x f(x) = \|\nabla_x f(x)\|_2 \min_{u,\|u\|=1} \cos\theta$$

- The minimal is achieved when $\theta = \pi$, i.e. the direction $u = -\nabla_x f(x)$ is the steepest descent or gradient descent.
- Then we update the state by

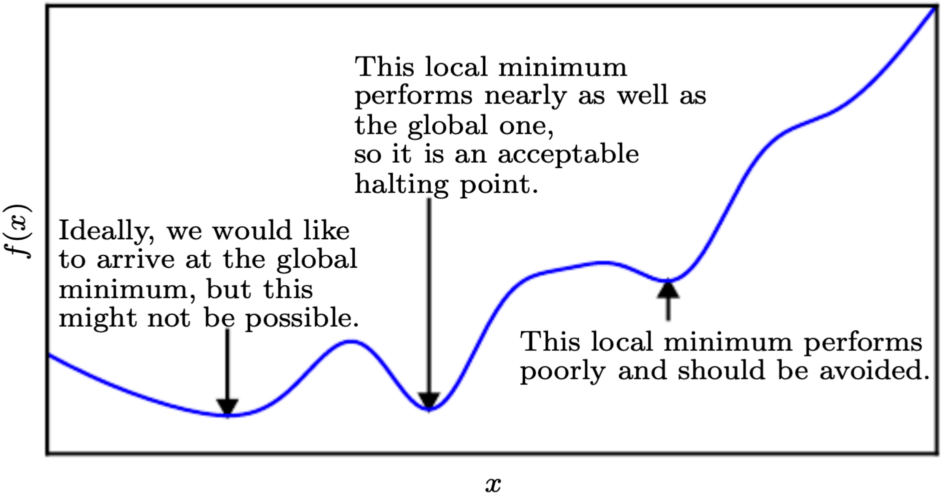
$$x' = x - \varepsilon \nabla_x f(x)$$

where ε is called *learning rate*.

Gradient Descent Optimization



Challenge: Multimodalities



Other Challenges

There are other challenges like:

- overflow/underflow, e.g. softmax function.
- ill-conditioning: $f(x) = A^{-1}x$ where $A \in \mathbb{R}^{n\times n}$ with eigenvalues $\{\lambda_i\}$, then condition number is $\max_{i,j} |\lambda_i/\lambda_j|$.
- complex landscape, e.g. plateaus, saddle points, cliffs...
- expensive gradients: large data volume.

Table of Contents

Training Deep Models

- Learning $\neq$ pure optimization.
- In most machine learning scenarios, we care about some performance performance measure P, defined with respect to test set.
- We reduce a different cost function $J(\theta)$ in the hope that doing so will improve P. This is in contrast to pure optimization with J as the goal.
- Typically, the cost function is defined as an expectation of some loss function $L(\cdot,\cdot)$, namely, risk,

$$J(\theta) = \mathrm{E}_{(x,y)\sim p_{data}} L(f(x;\theta), y)$$

- In reality, we often minimize the an approximate version, empirical risk,

$$\tilde{J}(\theta) = \mathrm{E}_{(x,y)\sim \hat{p}_{data}} L(f(x;\theta), y) = \frac{1}{N}\sum_{i=1}^{N} L(f(x^{(i)};\theta), y^{(i)})$$

(Mini)-batch Algorithms

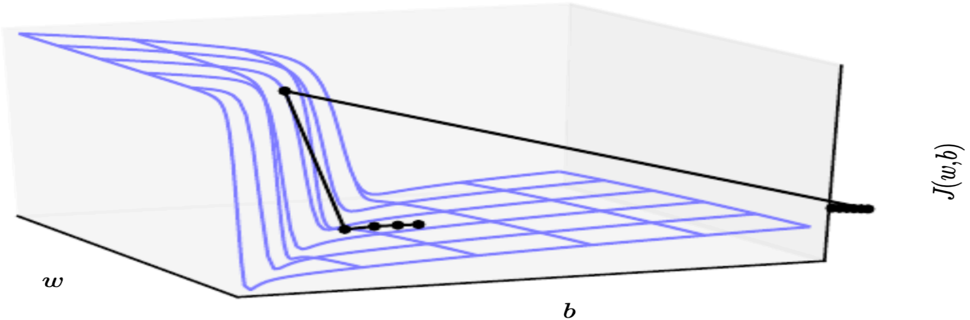
- Empirical risk minimization is prone to overfitting. In stead, we often consider a surrogate loss function, e.g. negative log-likelihood, i.e.

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{N} \log p_{model}(x^{(i)}, y^{(i)}; \theta)$$

- To combat the issue of expensive gradients when N is large, a small batch of data size m is (randomly) chosen to approximate the gradient in gradient descent algorithms:

$$\theta' = \theta - \frac{N\varepsilon}{m} \nabla_\theta \log p_{model}(x^{(i)}, y^{(i)}; \theta)$$

Challenge: cliff



- Like general gradient descent optimization, training neural network also faces the same challenges including cliffs, or exploding gradients.
- To alleviate such issue, gradient clipping is adopted when the norm of gradient $\|g\| > \mathrm{max\_norm}$ for some threshold $\mathrm{max\_norm}$:

$$g \leftarrow g \frac{\mathrm{max\_norm}}{\|g\|}$$

Challenge: compositions in deep models

- Very deep models involve the composition of several functions or layers.
- In practice, when we update all of the layers simultaneously, unexpected results can happen because many functions composed together are changed simultaneously, e.g. $\hat{y} = x w_1 w_2 \cdots w_l$ where $h_i = h_{i-1} w_i$, then the gradient in back-propagation could be either too small or too large.
- To solve this issue, batch normalization is adopted.
- Given a minibatch of activations $\mathbf{H}$, we normalize $\mathbf{H}$, we replace it with

$$\mathbf{H}' = \frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

where we have

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \mathbf{H}_i, \quad \boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\mathbf{H} - \boldsymbol{\mu})_i^2}, \ \delta \approx 10^{-8}$$

- At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ may be replaced by running averages that were collected during training time.

Stochastic Gradient Descent

- Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular.
- In practice, it is common to decay the learning rate ε linearly in the minibatch gradient descent until iteration τ:

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau, \quad \alpha = k/\tau.$$

such that the convergence condition, $\sum_{k=1}^{\infty} \varepsilon_k = \infty$, $\sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$, is met.

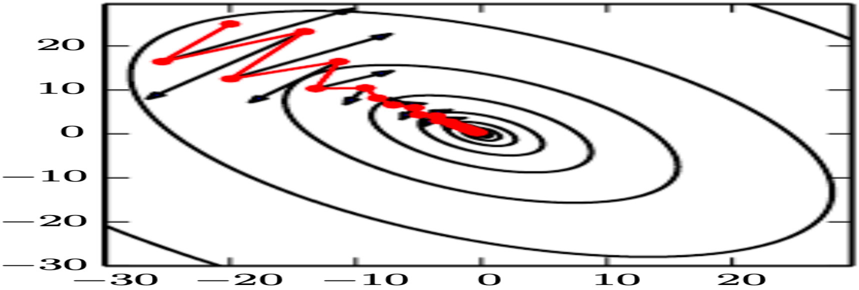
Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$
  end while

Momentum

- While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow.
- The method of momentum (Polyak, 1964) is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

$$v \leftarrow \alpha v - \varepsilon g(\theta), \quad \alpha \in [0, 1)$$
$$\theta \leftarrow \theta + v$$



Nesterov Momentum

- Sutskever et al. (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method (Nesterov, 1983, 2004).

$$v \leftarrow \alpha v - \varepsilon g(\theta + \alpha v), \quad \alpha \in [0, 1)$$
$$\theta \leftarrow \theta + v$$

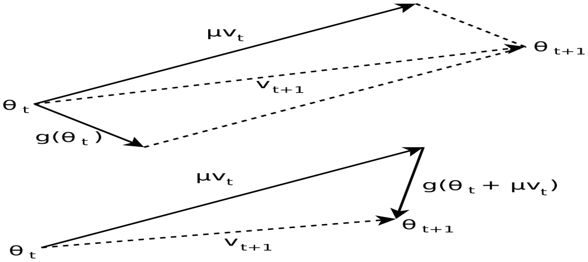


Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient

AdaGrad (Duchi et al., 2011)

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

RMSProp (Hinton, 2012)

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

Adam (Kingma and Ba, 2014)

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\epsilon\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

A nice list of optimizers implemented in PyTorch

Find more at `https://github.com/jettify/pytorch-optimizer`