Stochastic Optimization and Training Skills in Deep Learning

Wenjun Zeng

Reading Group of Ye Lab, UM

Sept. 16, 2018

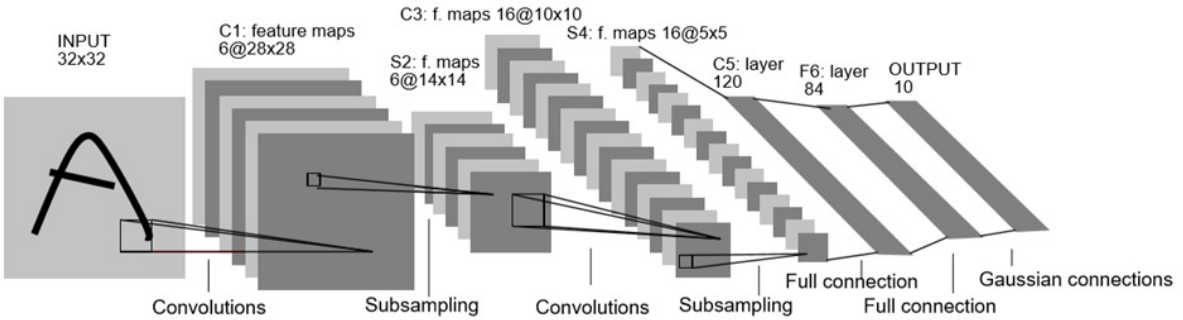
Contents

- Architecture of Deep Neural Networks
- Back Propagation
- Stochastic Optimization Algorithms
  - Stochastic Gradient/Subgradient Descent
  - Accelerated Methods
  - Variance Reduction
- Training Skills
  - Batch Normalization
  - Dropout
  - Parameter Tuning (Alchemy?)...

I. Architecture of Deep Neural Networks (DNN)

Take convolutional neural networks (CNN) as example.

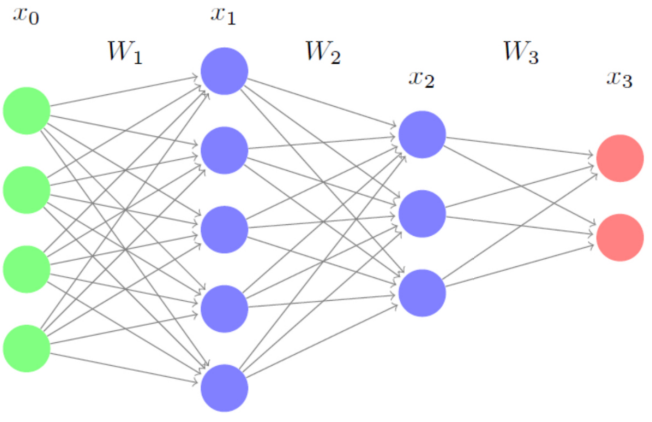
LeNet [LeCun *et al.*'1998]

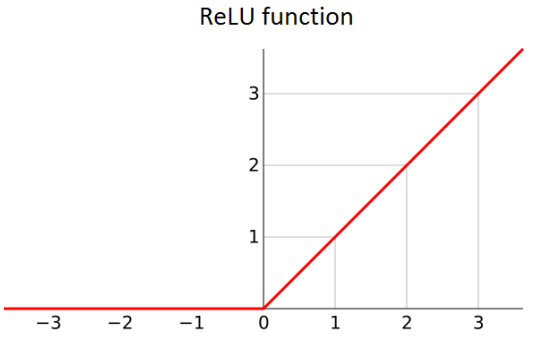


- Convolutional layer
- Pooling layer (subsampling)
- Full connection layer
- Nonlinear mapping is sometimes also called as a layer (ReLU layer)

DNN: many (from several tens to nearly 200) layers.

1.1 Full Connection Layer





- $\boldsymbol{x}_l \in \mathbb{R}^{n_l}$ $(l = 0, 1, \cdots, L)$: vector of l-th layer; L: number of layers
- $\boldsymbol{x}_0$ is input; $\boldsymbol{x}_L$ is output; other $\boldsymbol{x}_l$ are hidden layer variables.

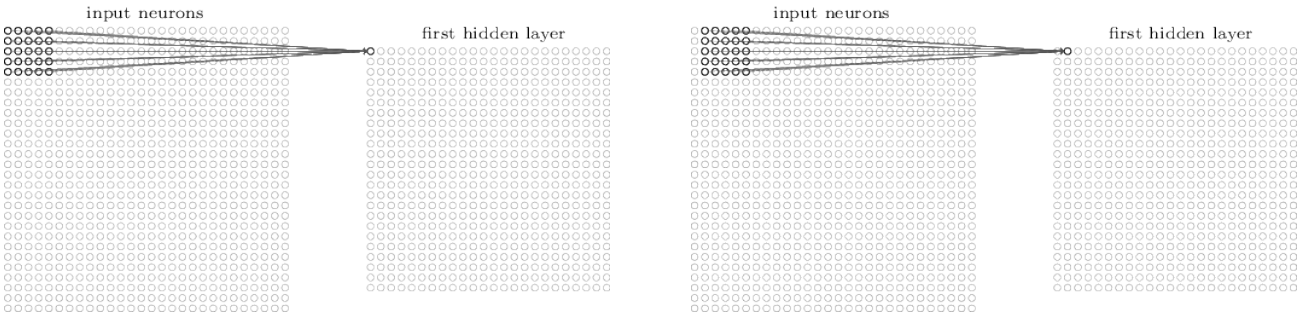
$$\boldsymbol{u}_l = \boldsymbol{W}_l \boldsymbol{x}_{l-1} + \boldsymbol{b}_l$$

$$\boldsymbol{x}_l = \sigma(\boldsymbol{u}_l)$$

- $\boldsymbol{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$ and $\boldsymbol{b}_l \in \mathbb{R}^{n_l}$ are weights and bias.
- $\sigma(\cdot)$ is nonlinear activation function, e.g., sigmoid or ReLU.

1.2 Convolutional Layer

Convolution layer in fact has similar structure as the full connection layer but a neuron of the next layer is merely connected with very few neurons of the last layer.



Local receptive fields: only make connections in small, localized regions of the input.

$$x_l(i,j) = \sigma\left(b + \sum_{p=0}^{FH}\sum_{q=0}^{FW} w_{p,q}x_{l-1}(i+p, j+q)\right)$$

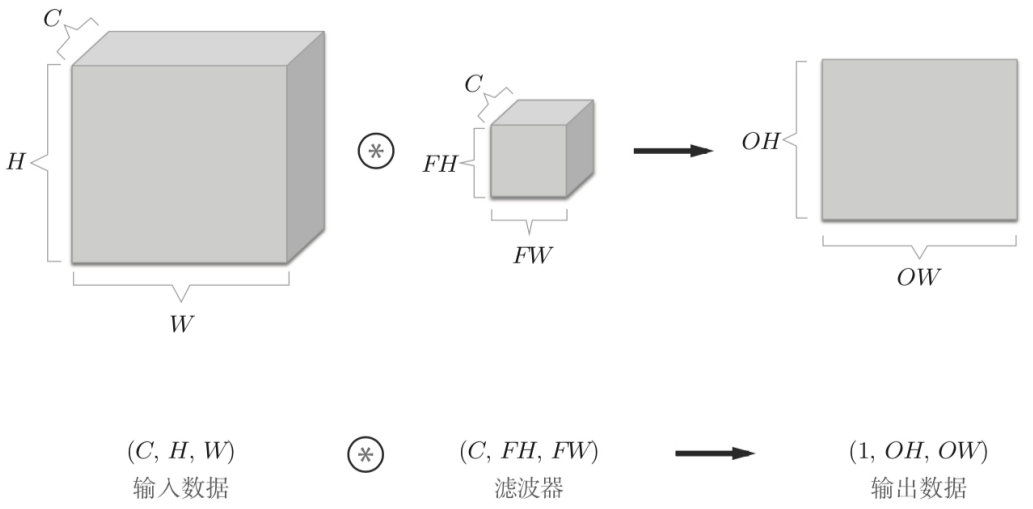
$\{w_{p,q}\}$ are coefficients of convolutional kernel (filter) with size $FH \times FW$.

Shared weights and biases: same convolution kernel applies to all pixels of the input.

Number of parameters significantly reduced compared with full connection case.

Tensor in Convolutional Layer

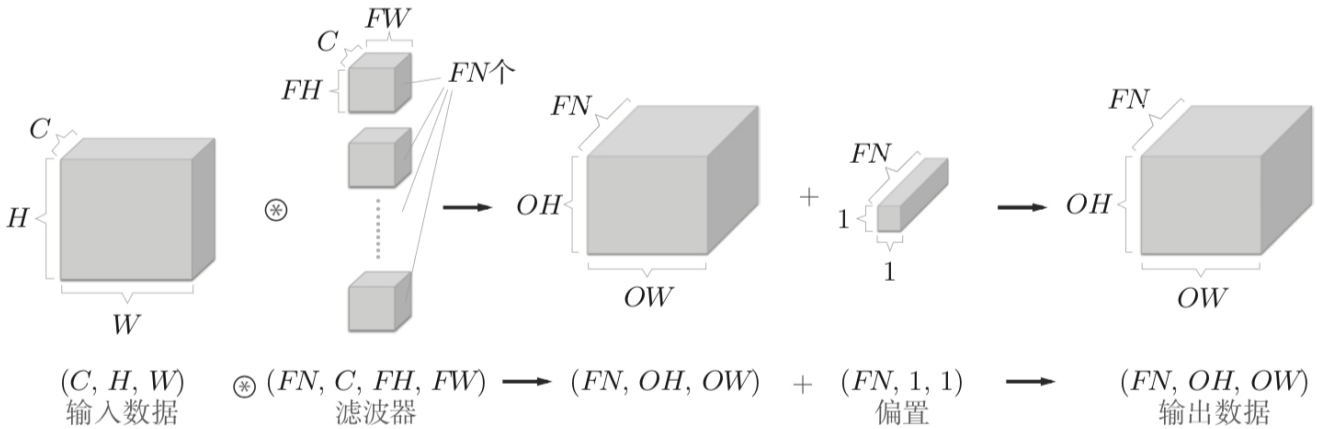
Using multiple filters, data of each convolutional layer are 3rd-order tensor.



C: number of channels

- At the input layer, $C = 3$ for colored images (RGB) and $C = 1$ for grayed images.
- At other convolutional layers, C equals the number of convolutional kernels.
- The output corresponding to one filter is a matrix (feature map), not a tensor.

For FN convolutional kernels



Number of weights is $FH \cdot FW \cdot C \cdot FN$ while number of bias is FN.

Total number of parameters $(FH \cdot FW \cdot C + 1) \cdot FN \ll (n_l + 1)n_{l-1}$

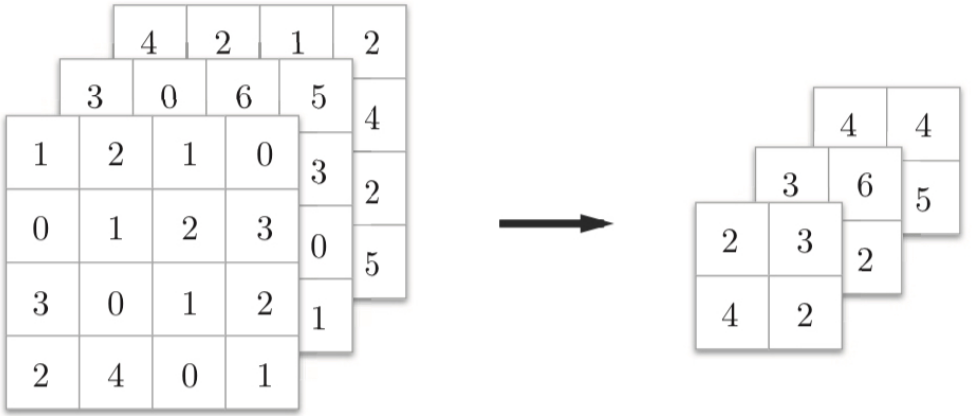
$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FW}{S} + 1$$

P is number of padding and S is stride.

1.3 Pooling Layer

Pooling is nothing but subsampling. We prefer max pooling.



- No parameters need to learn in pooling layers (pay attention when back propagation).
- Number of channels remain unchanged.

Pooling is important because it makes DNN robust to slight variation of the data.

II. Back Propagation (BP)

D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning representations by back-propagating errors," ***Nature****, 323, 533–536.*

- I think BP is the most important issue for NN and DNN. Without it, we cannot efficiently train an NN.
- It is based on the chain rule of multivariate derivative.
- Many courses, books, and papers present BP in a very long, tedious, puzzling, and even unclear way...

One derivation by LeCun [LeCun'1988] using the Lagrange multiplier method may be better. First, once $\{\boldsymbol{W}_l, \boldsymbol{b}_l\}$ is initialized or updated, do forward pass

$$
\begin{aligned}
&\text{for } l = 1, 2 \cdots, L \\
&\qquad \boldsymbol{u}_l = \boldsymbol{W}_l \boldsymbol{x}_{l-1} + \boldsymbol{b}_l \\
&\qquad \boldsymbol{x}_l = \sigma(\boldsymbol{u}_l) \\
&\text{end for}
\end{aligned}
$$

2.1 Training NN

- One (or mini-batch) training sample $\boldsymbol{x}_0$ and the desired target (e.g., label) $\boldsymbol{y} \in \mathbb{R}^n_L$
- Loss function, e.g., the ℓ_2-loss:

$$f(\{\boldsymbol{W}_l, \boldsymbol{b}_l\}) = \frac{1}{2}\|\boldsymbol{x}_L - \boldsymbol{y}\|^2$$

We use stochastic gradient descent (SGD) (discuss soon)

$$\boldsymbol{W}_l^{k+1} = \boldsymbol{W}_l^k - \alpha_k \frac{\partial f}{\partial \boldsymbol{W}_l^k}$$

$$\boldsymbol{b}_l^{k+1} = \boldsymbol{b}_l^k - \eta_k \frac{\partial f}{\partial \boldsymbol{b}_l^k}$$

where $\alpha_k > 0$ and $\eta_k > 0$ are learning rates.

Question: how to compute the gradients $\frac{\partial f}{\partial \boldsymbol{W}_l}$ and $\frac{\partial f}{\partial \boldsymbol{b}_l}$? BP solves it.

2.2 Summary of BP

1. For the last layer ($l = L$)

$$\frac{\partial f}{\partial \boldsymbol{u}_L} = (\boldsymbol{x}_L - \boldsymbol{y}) \odot \sigma'(\boldsymbol{u}_L)$$

2. For $l = L-1, L-2, \cdots, 1$ (backward pass)

$$\frac{\partial f}{\partial \boldsymbol{u}_l} = \boldsymbol{W}_l^{\mathrm{T}} \frac{\partial f}{\partial \boldsymbol{u}_{l+1}} \odot \sigma'(\boldsymbol{u}_l)$$

where $\odot$ is component-wise product and $\sigma'(\cdot)$ is the derivative of $\sigma(\cdot)$.

Then, for all $l = L, L-1, \cdots, 1$

$$\frac{\partial f}{\partial \boldsymbol{W}_l} = \frac{\partial f}{\partial \boldsymbol{u}_l} \boldsymbol{x}_l^{\mathrm{T}}$$

$$\frac{\partial f}{\partial \boldsymbol{b}_l} = \frac{\partial f}{\partial \boldsymbol{u}_l}$$

For convolutional layer, BP is slightly different from the full connection case. But the principle is the same.

III. Stochastic Optimization Algorithms

Many (almost all?) deep learning libraries use stochastic gradient methods to train DNN.

- Caffe: SGD, Adadelta,Adagrad, Adam, Nesterov, Rmsprop
- TensorFlow: SGD, Adadelta, AdagradDA, Adagrad, ProximalAdagrad, Ftrl, Momentum, Adam, CenteredRMSProp.
- Lasagne: SGD, Momentum, Nesterov, Adagrad, Rmsprop, Adadelta, Adam, Adamax...

Stochastic Optimization Algorithms is crucial for big DNN training.

3.1 Big-N Problem

Consider finite sum or empirical risk minimization (ERM)

$$\min_{\boldsymbol{x}\in\mathbb{R}^d} \left\{ f(\boldsymbol{x}) := \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{x}) \right\} \tag{1}$$

- $\boldsymbol{x} \in \mathbb{R}^d$ denotes parameters of neural networks, i.e., $\{\boldsymbol{W}_l, \boldsymbol{b}_l\}$; d can be very large.
- N: number of training samples; usually very large, e.g., million.
- $f_i(\boldsymbol{x})$: loss function due to sample i.

Sometimes regularization is used to suppress over-fitting

$$\min_{\boldsymbol{x}\in\mathbb{R}^d} \left\{ f(\boldsymbol{x}) := \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{x}) + \lambda r(\boldsymbol{x}) \right\} \tag{2}$$

Common regularizer used in Caffe and TensorFlow $r(\boldsymbol{x}) = \|\boldsymbol{x}\|_2^2$ and $r(\boldsymbol{x}) = \|\boldsymbol{x}\|_1$.

$\Rightarrow$ alternative to weight decay.

3.2 Gradient Method

Deterministic gradient descent (GD) [Cauchy'1847]:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha_k \nabla f(\boldsymbol{x}^k) \quad \text{with } \nabla f(\boldsymbol{x}^k) = \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\boldsymbol{x}^k) \tag{3}$$

Well-established rules to determine learning rate α_k:

- $0 < \alpha_k < \frac{2}{L}$, e.g., $\alpha_k = \frac{1}{L}$, with L the Lipschitz constant of $\nabla f(\boldsymbol{x})$.
- α_k is determined by backtracking line search, e.g., Armijo's rule [Bertsekas'1999].

Convergence Guarantees [Nestrov'2004]

- Linear convergence rate of $\mathcal{O}\left((1 - \mu/L)\right)^k)$ for μ-strongly convex objective $f(\boldsymbol{x})$.
- Sublinear rate of $\mathcal{O}(1/k)$ without strong convexity.

Drawback: Too expensive to compute $\nabla f(\boldsymbol{x}^k) = \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\boldsymbol{x}^k)$ for big N.

3.3 Stochastic Gradient Method

Stochastic gradient descent (SGD) [Robbins-Monro'1951]:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha_k \nabla f_{i_k}(\boldsymbol{x}^k) \tag{4}$$

where i_k is selected uniformly at random from $\{1, 2, \cdots, N\}$.

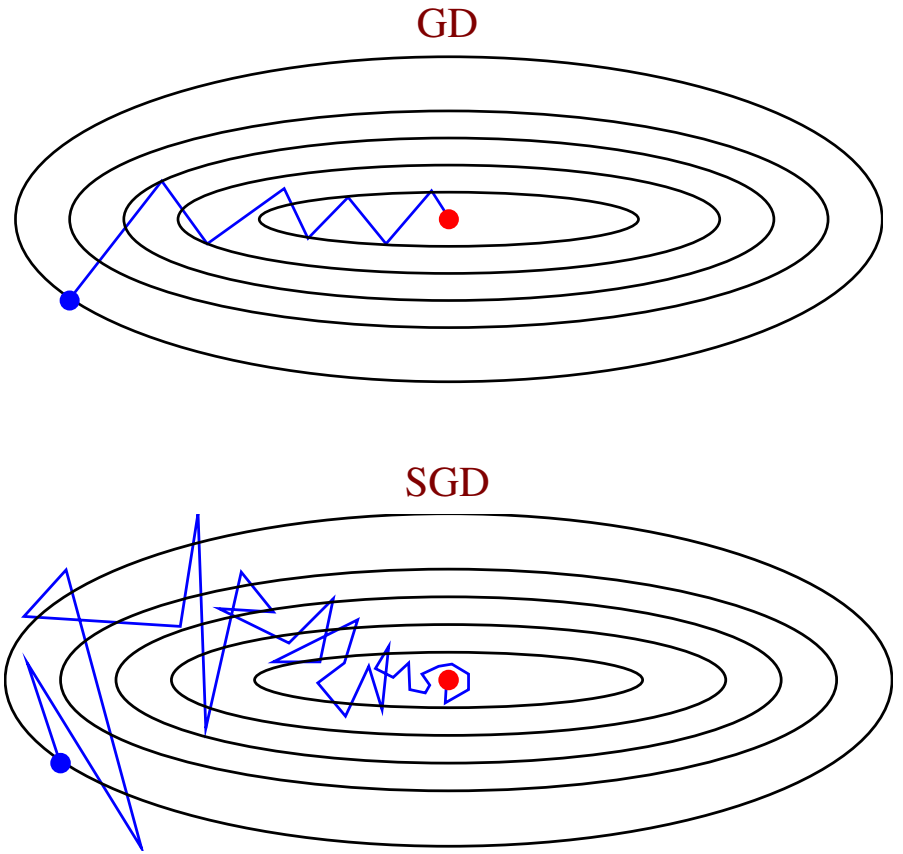
Motivation:

$$\mathbb{E}\left[\nabla f_{i_k}(\boldsymbol{x}^k)\right] = \sum_{i=1}^{N} \frac{1}{N} \nabla f_i(\boldsymbol{x}^k) = \nabla f(\boldsymbol{x}^k)$$

$\Rightarrow \nabla f_{i_k}(\boldsymbol{x}^k)$ is a cheap unbiased estimate of true gradient

- Per-iteration cost of SGD is N times lower than GD.
- Convergence requires $\alpha_k \to 0 \Rightarrow$ *line search not applicable—learning rate needs to tune!*
  Typical settings: $\alpha_k \sim \mathcal{O}(1/k)$ or $\alpha_k \sim \mathcal{O}(1/\sqrt{k})$.

Convergence Behavior



The objective of SGD does not monotonically decrease.

Theoretical Convergence of SGD

Assumptions:

- $\nabla f(\boldsymbol{x})$ is Lipschitz continuous with constant L (DNN loss function locally satisfies)
- Gradient is bounded: $\mathbb{E}_i \left[\|\nabla f_i(\boldsymbol{x})\|^2 \right] \leq M^2$.

Theorem *Define* $\Delta_k = \|\boldsymbol{x} - \boldsymbol{x}^*\|$. *For* μ*-strongly convex objective and a diminishing stepsize* $\alpha_k = \frac{\beta}{k+\gamma}$ *for some* $\beta > \frac{1}{2\mu}$ *and* $\gamma > 0$ *such that* $\alpha_1 \leq \frac{1}{2\mu}$. *Then, for any* $k \geq 1$

$$\mathbb{E}[f(k) - f(\boldsymbol{x}^*)] \leq \frac{L}{2}\mathbb{E}[\Delta_k^2] \leq \frac{L}{2}\frac{\nu}{k+\gamma}$$

where $\nu = \max\left(\frac{\beta^2 M^2}{2\beta\mu - 1}, (\gamma+1)\Delta_1^2\right)$.

- For fixed stepsize, we don't have convergence.
- For diminishing stepsize, convergence rate is $\mathcal{O}(1/k)$.
- For nonsmooth case (e.g., ReLU), convergence rate of stochastic subgradient is $\mathcal{O}(1/\sqrt{k})$.

3.4 Accelerated Methods

SGD with Momentum

$$\boldsymbol{v}^{k+1} = \mu_k \boldsymbol{v}^k - \alpha_k \nabla f_{i_k}(\boldsymbol{x}^k)$$
$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{v}^{k+1}$$

Nesterov Accelerated Gradient

Yurii Nesterov

Y. Nesterov, "A method for solving the convex programming problem with convergence rate $O(1/k^2)$," *Dokl. Acad. Nauk SSSR,* pp. 543–547, 1983.

Original Version of Nesterov

$$\boldsymbol{v}^{k+1} = (1+\mu_k)\boldsymbol{x}^k - \mu_k \boldsymbol{x}^{k-1}$$
$$\boldsymbol{x}^{k+1} = \boldsymbol{v}^{k+1} - \alpha_k \nabla f_{i_k}(\boldsymbol{v}^{k+1})$$

where $\mu_k = \frac{k+2}{k+5}$.

Momentum Version of Nesterov

$$\boldsymbol{v}^{k+1} = \mu_k \boldsymbol{v}^k - \alpha_k \nabla f_{i_k}(\boldsymbol{x}^k + \mu_k \boldsymbol{v}^k)$$
$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{v}^{k+1}$$

- For GD, convergence rate of Nesterov is improved from $\mathcal{O}(1/k)$ to $\mathcal{O}(1/k^2)$. $\mathcal{O}(1/k^2)$ is the optimal bound for non-strongly convex objective of all first-order methods! Recently still many researches on it. [Su-Boyd-Candès'2016], [Bubeck-Lee-Singh'2015]
- For stochastic optimization, is there any theoretical analysis on the improvement of convergence rate of Nesterov's acceleration? I don't find any at present...

Adaptive Subgradient (Adagrad) [Duchi-Hazan-Singer'2011]

Let $\boldsymbol{g}^k = \nabla f_{i_k}(\boldsymbol{x}^k)$

$$\left[\boldsymbol{x}^{k+1}\right]_n = \left[\boldsymbol{x}^k\right]_n - \frac{\alpha_k}{\sqrt{\sum_{j=1}^k \left[\boldsymbol{g}^j\right]_n^2 + \epsilon}} \left[\boldsymbol{g}^k\right]_n, \ n = 1, 2, \cdots, d$$

where $\epsilon > 0$ is small to avoid being divided by zero.

Drawback: denominator becomes larger and larger $\Rightarrow$ learning rate approaches 0 quickly.

Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google and for recognizing cats in Youtube videos. [Dean, Ng, *et al.* 2012]

RMSProp [Hinton]

Initialize $\mathbb{E}[(\boldsymbol{g}^0)^2] = \mathbf{0}$. At step $k = 1, 2, \cdots$

$$\mathbb{E}[(\boldsymbol{g}^k)^2] = \rho \mathbb{E}[(\boldsymbol{g}^{k-1})^2] + (1-\rho)(\boldsymbol{g}^k)^2$$
$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \frac{\alpha_k}{\sqrt{\mathbb{E}[(\boldsymbol{g}^k)^2] + \epsilon \mathbf{1}_d}} \nabla f_{i_k}(\boldsymbol{x}^k)$$

where $\rho = 0.9$ (like forgetting factor in adaptive filtering) and $\alpha_k = 10^{-3}$ is a good choice.

The root mean square (RMS) of $\boldsymbol{g}^k$ is denoted as $\mathrm{RMS}[\boldsymbol{g}^k] = \sqrt{\mathbb{E}[(\boldsymbol{g}^k)^2] + \epsilon \mathbf{1}_d}$.

Adadelta [Zeiler'2012]

The "delta" in Adadelta means the quantity $\Delta \boldsymbol{x}^k = \boldsymbol{x}^{k+1} - \boldsymbol{x}^k$.

Denote $\mathrm{RMS}[\boldsymbol{g}^k] = \sqrt{\mathbb{E}[(\boldsymbol{g}^k)^2] + \epsilon \mathbf{1}_d}$ and $\mathrm{RMS}[\Delta \boldsymbol{x}^k] = \sqrt{\mathbb{E}[(\Delta \boldsymbol{x}^k)^2] + \epsilon \mathbf{1}_d}$.

Initialize $\mathbb{E}[(\boldsymbol{g}^0)^2] = \mathbf{0}$ and $\mathbb{E}[(\Delta \boldsymbol{x}^0)^2] = \mathbf{0}$.

At step $k = 1, 2, \cdots$

$$\mathbb{E}[(\boldsymbol{g}^k)^2] = \rho \mathbb{E}[(\boldsymbol{g}^{k-1})^2] + (1-\rho)(\boldsymbol{g}^k)^2$$

$$\Delta \boldsymbol{x}^k = -\frac{\mathrm{RMS}[\Delta \boldsymbol{x}^{k-1}]}{\mathrm{RMS}[\boldsymbol{g}^k]} \boldsymbol{g}^k$$

$$\mathbb{E}[(\Delta \boldsymbol{x}^k)^2] = \rho \mathbb{E}[(\Delta \boldsymbol{x}^{k-1})^2] + (1-\rho)(\Delta \boldsymbol{x}^k)^2$$

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \Delta \boldsymbol{x}^k$$

where ρ is around 0.9.

Advantage: Adadelta does not need to set a learning rate.

Adaptive Moment Estimation (Adam) [Kingma-Ba'2015]

Idea: Combine Adadelta/RMSProp and momentum.

Moment: First and and second moments of (component-wise) gradient $\mathbb{E}[\boldsymbol{g}^k]$ and $\mathbb{E}[(\boldsymbol{g}^k)^2]$.

Initialize $\mathbb{E}[\boldsymbol{g}^0] = \mathbf{0}$ and $\mathbb{E}[(\boldsymbol{g}^0)^2] = \mathbf{0}$.

At step $k = 1, 2, \cdots$

$$
\begin{aligned}
\mathbb{E}[\boldsymbol{g}^k] &= \rho_1 \mathbb{E}[\boldsymbol{g}^{k-1}] + (1-\rho_1)\boldsymbol{g}^k \quad \text{(momentum term)} \\
\mathbb{E}[(\boldsymbol{g}^k)^2] &= \rho_2 \mathbb{E}[(\boldsymbol{g}^{k-1})^2] + (1-\rho_2)(\boldsymbol{g}^k)^2 \\
\widehat{\mathbb{E}}[\boldsymbol{g}^k] &= \frac{1}{1-\rho_1^k}\mathbb{E}[\boldsymbol{g}^k] \quad \text{(bias correct)} \\
\widehat{\mathbb{E}}[(\boldsymbol{g}^k)^2] &= \frac{1}{1-\rho_2^k}\mathbb{E}[(\boldsymbol{g}^k)^2] \\
\boldsymbol{x}^{k+1} &= \boldsymbol{x}^k - \frac{\alpha}{\sqrt{\mathbb{E}[(\boldsymbol{g}^k)^2] + \epsilon \mathbf{1}_d}}\widehat{\mathbb{E}}[\boldsymbol{g}^k]
\end{aligned}
$$

where ρ_1 (0.9) and ρ_2 (0.999) are decay rates while α is learning rate.

Adam is widely used in training DNN. More other methods such as Nestrov Adam (Nadam).

Mini-Batch Stochastic Gradient Descent

Just replace the stochastic gradient $\nabla f_{i_k}(\boldsymbol{x}^k)$ with $\frac{1}{m}\sum_{i_k=i_1}^{i_m}\nabla f_{i_k}(\boldsymbol{x}^k)$ where $\{i_1, \cdots, i_m\} \in \{1, \cdots, N\}$ is the randomly selected mini-batch index.

Typical mini-batch size $m = 100$.

IV. Training Skills

Batch Normalization

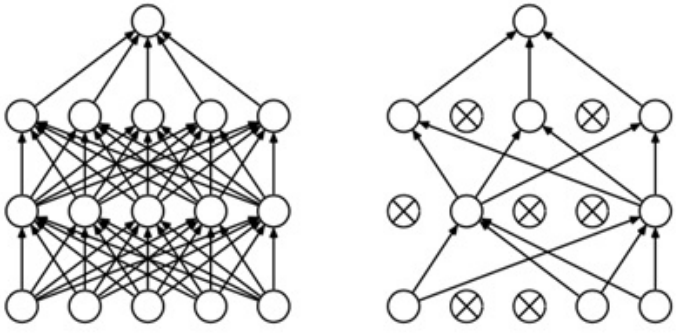
At each layer, shift min-batch data to zero-mean and unit variance.

$$\widehat{x}_i = \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\text{var}[x_i]}}$$

$$z_i = \gamma_i \widehat{x}_i + \beta_i$$

New technique: Group normalization [He'2018].

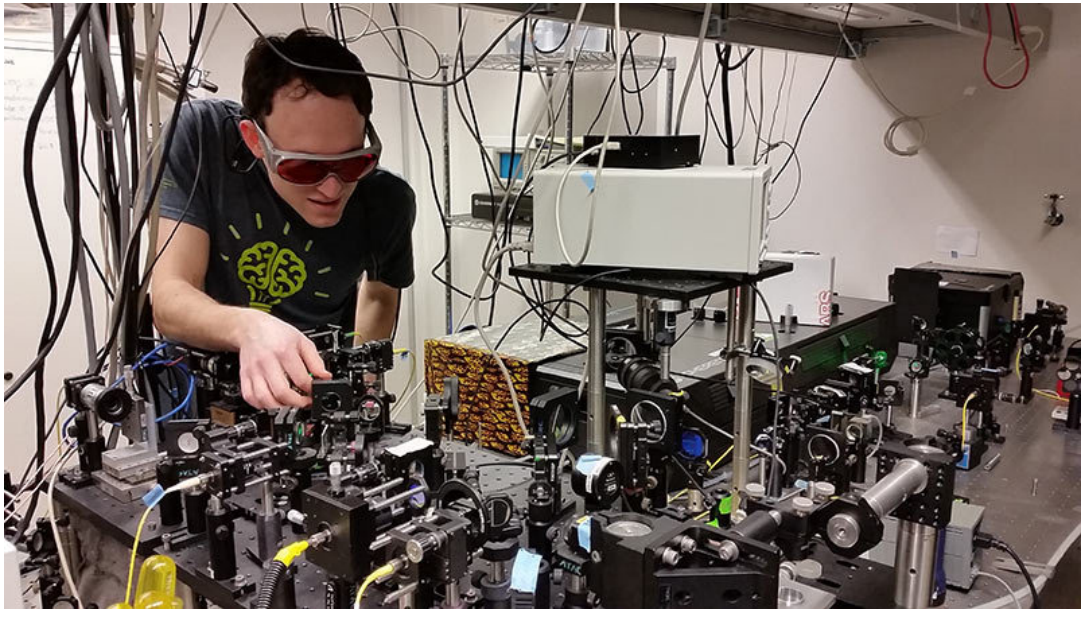
Dropout



Parameter Tuning (Alchemy?)

Parameter tuning is very important in deep learning.

Parameter tuning is somewhat like:



This picture has no relation with the context of deep learning.

Hyper-parameters need to tune in DNN:

- Number of layers (depth);
- Number of neurons of each layer (width)
- What is the structure of each layer? Linear layer or convolution layer?
- What is the activation function?
- Size of mini-batch
- Which optimization algorithm?
- Learning rate
- Any bias is needed for convolution layers?...

Thank you for your attention!