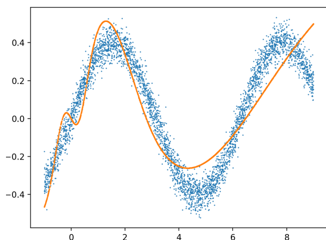
Training DNNs: Basic Methods

Ju Sun
Computer Science & Engineering
University of Minnesota, Twin Cities

March 3, 2020

Supervised learning as function approximation



- Underlying true function: f_0
- Training data: $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}$ with $\boldsymbol{y}_i \approx f_0(\boldsymbol{x}_i)$
- Choose a family of functions $\mathcal{H}$, so that $\exists f \in \mathcal{H}$ and f and f_0 are close
- Find f, i.e., optimization

$$\min_{f \in \mathcal{H}} \sum_i \ell(\boldsymbol{y}_i, f(\boldsymbol{x}_i)) + \Omega(f)$$

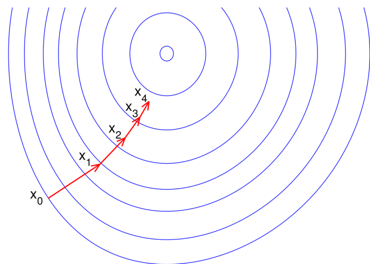
- **Approximation capacity: Univeral approximation theorems (UAT)** $\Longrightarrow$ replace $\mathcal{H}$ by $\mathrm{DNN}_{\boldsymbol{W}}$, i.e., a deep neural network with weights $\boldsymbol{W}$
- **Optimization:**

$$\min_{\boldsymbol{W}} \sum_i \ell(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}(\boldsymbol{x}_i)) + \Omega(\boldsymbol{W})$$

- **Generalization:** how to avoid over-complicated $\mathrm{DNN}_{\boldsymbol{W}}$ in view of UAT

Basics of numerical optimization

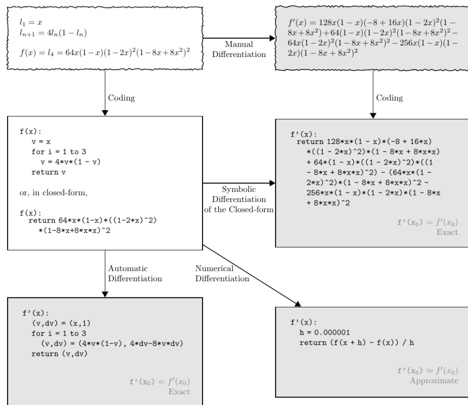
- 1st and 2nd optimality conditions
- iterative methods



Credit: aria42.com

- gradient descent
- Newton's method
- momentum methods
- quasi-Newton methods
- coordinate descent
- conjugate gradient methods
- trust-region methods
- etc

Computing derivatives



Credit: [Baydin et al., 2017]

- Analytic differentiation (by hand or by software)
- Finite difference approximation
- Automatic/Algorithmic differentiation (AD)

Ready to optimize DNNs!

Outline

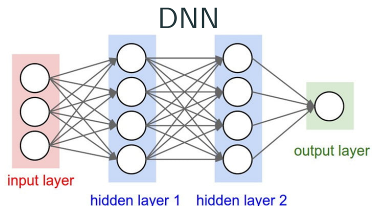
Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

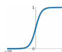
Suggested reading

Set up the problem

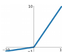


activation function

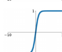
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

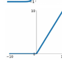


tanh
$\tanh(x)$

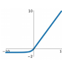


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

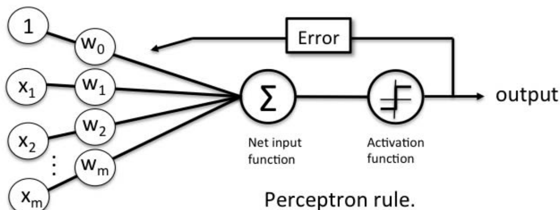


Credit: Stanford CS231N

$$\min_{\boldsymbol{W}} \sum_i \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

- Which activation at the hidden nodes?
- Which activation at the output node?
- Which ℓ?

Which activation at the hidden nodes?



Is the $\text{sign}(\cdot)$ activation good for derivative-based optimization?

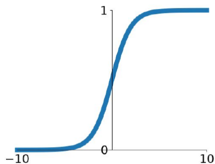
$$\nabla_{\boldsymbol{w}} \ell\left(\text{sign}\left(\boldsymbol{w}^\intercal \boldsymbol{x}\right), y\right) = \ell'\left(\text{sign}\left(\boldsymbol{w}^\intercal \boldsymbol{x}\right), y\right) \text{sign}'\left(\boldsymbol{w}^\intercal \boldsymbol{x}\right) \boldsymbol{x} = \boldsymbol{0}$$

almost everywhere (But why the classic Perceptron algorithm converges?)

Desiderata:

- Differentiable or almost everywhere differentiable
- Nonzero derivatives (almost) everywhere
- Cheap to compute

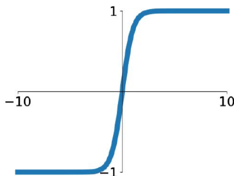
Sigmoid and hypertangent



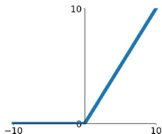
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- Differentiable? **Yes!**
- Nonzero derivatives? **Yes and No!** What happens for large positive and negative inputs?
- Cheap? $\exp(\cdot)$ is relatively expensive

What about $\tanh$?



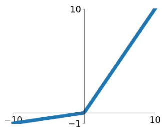
ReLU and friends



ReLU
(Rectified Linear Unit)

$$\sigma(x) = \max(0, x)$$

- Differentiable? **Yes!** (almost everywhere)
- Nonzero derivatives? **Yes and No!** What happens for $x < 0$?
- Cheap? **Yes!**

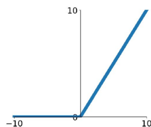


Leaky ReLU

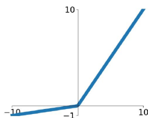
$$\sigma(x) = \max(\alpha x, x) \quad \text{(e.g., } \alpha = 0.01\text{)}$$

- Differentiable? **Yes!** (almost everywhere)
- Nonzero derivatives? **Yes!** (almost everywhere)
- Cheap? **Yes!**

ReLU and friends

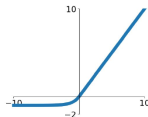


ReLU
(Rectified Linear Unit)



Leaky ReLU

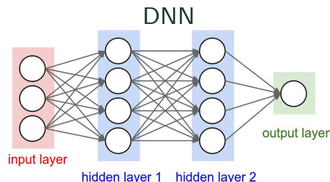
Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

- ReLU and Leaky ReLU are the most popular
- $\tanh$ less preferred but okay; sigmoid should be avoided
- Question: what do you think of $|\cdot|$ as activation?

Which activation at output node?



depending on the desired output

- unbounded scalar/vector output (e.g. , regression): identity activation
- binary classification with 0 or 1 output: e.g., sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$
- multiclass classification: labels into vectors via one-hot encoding

$$L_k \Longrightarrow [\underbrace{0, \ldots, 0,}_{k-1\,0's}, 1, \underbrace{0, \ldots, 0}_{n-k\,0's}]^\mathsf{T}$$

  Softmax activation:

$$\boldsymbol{z} \mapsto \left[\frac{e^{z_1}}{\sum_j e^{z_j}}, \ldots, \frac{e^{z_p}}{\sum_j e^{z_j}}\right]^\mathsf{T}.$$

- discrete probability distribution: softmax
- etc .

Which loss?

Which ℓ to choose? Make it differentiable, or almost so

- **regression**: $\|\cdot\|_2^2$ (common, `torch.nn.MSELoss`), $\|\cdot\|_1$ (for robustness, `torch.nn.L1Loss`), etc
- **binary classification**: encoder the classes as $\{0,1\}$, $\|\cdot\|_2^2$ or cross-entropy: $\ell(y,\hat{y}) = y\log\hat{y} - (1-y)\log(1-\hat{y})$ (min at $\hat{y}=y$, `torch.nn.BCELoss`)
- **multiclass classification** based on one-hot encoding and softmax activation: $\|\cdot\|_2^2$ or cross-entropy: $\ell(\boldsymbol{y},\widehat{\boldsymbol{y}}) = -\sum_i y_i \log\widehat{y_i}$ (min at $\boldsymbol{y}=\widehat{\boldsymbol{y}}$, `torch.nn.CrossEntropyLoss`)
- **multiclass classification label smoothing**, assuming m classes: one-hot encoding makes $n-1$ entropies in $\boldsymbol{y}$ 0's. When $y_i=0$, the derivative of $y_i\log\widehat{y_i}$ is $0 \implies$ no update due to y_i. Remedy: relax ... change $[\underbrace{0,\ldots,0,}_{k-1\,0's}1,\underbrace{0,\ldots,0}_{n-k\,0's}]^\intercal$ into $[\underbrace{\varepsilon,\ldots,\varepsilon,}_{k-1\,\varepsilon's}1-(m-1)\varepsilon,\underbrace{\varepsilon,\ldots,\varepsilon}_{n-k\,\varepsilon's}]^\intercal$ for a small ε
- **difference between distributions**: Kullback-Leibler divergence loss (`torch.nn.KLDivLoss`) or Wasserstein metric

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Suggested reading

Framework of line-search methods

A generic line search algorithm

Input: initialization $\boldsymbol{x}_0$, stopping criterion (SC), $k = 1$

1: **while** SC not satisfied **do**
2: choose a direction $\boldsymbol{d}_k$
3: decide a step size t_k
4: make a step: $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + t_k \boldsymbol{d}_k$
5: update counter: $k = k + 1$
6: **end while**

Four questions:

- How to choose direction $\boldsymbol{d}_k$?
- How to choose step size t_k?
- Where to initialize?
- When to stop?

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Suggested reading

From deterministic to stochastic optimization

Recall our optimization problem:

$$\min_{\boldsymbol{W}} \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

What happens when m is large, i.e., in the "big data" regime?

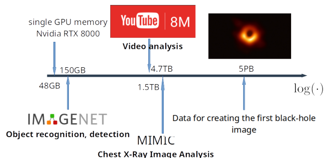
Blessing: assume $(\boldsymbol{x}_i, \boldsymbol{y}_i)$'s are iid, then

$$\frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) \to \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \ell\left(\boldsymbol{y}, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}\right)\right)$$

by the law of large numbers. Large $m \approx$ good generalization!

Curse: storage and computation

- **storage**: the dataset $\{(\boldsymbol{x}_i, \boldsymbol{y})\}$ typically stored on GPU/TPU for parallel computing—loading whole datasets into GPU often infeasible



- **computation**: each iteration costs at least $O(mn)$, where n is #(opt variables)—both can be large for training DNNs!

From deterministic to stochastic optimization

How to get around the storage and computation bottleneck when m is large?

stochastic optimization (stochastic = random)

Idea: use a small batch of data samples to approximate quantities of interest

- gradient: $\frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{W}} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) \to \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \nabla_{\boldsymbol{W}} \ell\left(\boldsymbol{y}, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}\right)\right)$

  approximated by **stochastic gradient**:

  $$\frac{1}{|J|}\sum_{j\in J} \nabla_{\boldsymbol{W}} \ell\left(\boldsymbol{y}_j, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_j\right)\right)$$

  for a random subset $J \subset \{1, \dots, m\}$, where $|J| \ll m$

- Hessian: $\frac{1}{m}\sum_{i=1}^{m} \nabla^2_{\boldsymbol{W}} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) \to \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \nabla^2_{\boldsymbol{W}} \ell\left(\boldsymbol{y}, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}\right)\right)$

  approximated by **stochastic Hessian**:

  $$\frac{1}{|J|}\sum_{j\in J} \nabla^2_{\boldsymbol{W}} \ell\left(\boldsymbol{y}_j, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_j\right)\right)$$

  for a random subset $J \subset \{1, \dots, m\}$, where $|J| \ll m$

... justified by the law of large numbers

Stochastic gradient descent (SGD)

In general (i.e., not only for DNNs), suppose we want to solve

$$\min_{\boldsymbol{w}} \; F(\boldsymbol{w}) \doteq \frac{1}{m} \sum_{i=1}^{m} f(\boldsymbol{w}; \boldsymbol{\xi}_i) \qquad \xi_i\text{'s are data samples}$$

idea: replace gradient with a stochastic gradient in each step of GD

Stochastic gradient descent (SGD)

Input: initialization $\boldsymbol{x}_0$, stopping criterion (SC), $k = 1$

1: **while** SC not satisfied **do**
2: sample a random subset $J_k \subset \{0, \ldots, m-1\}$
3: calculate the stochastic gradient $\widehat{\boldsymbol{g}_k} \doteq \frac{1}{|J_k|} \sum_{j \in J_k} \nabla_{\boldsymbol{w}} f(\boldsymbol{w}; \boldsymbol{\xi}_i)$
4: decide a step size t_k
5: make a step: $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} - t_k \widehat{\boldsymbol{g}_k}$
6: update counter: $k = k + 1$
7: **end while**

- J_k is redrawn in each iteration
- Traditional SGD: $|J_k| = 1$. The version presented is also called **mini-batch gradient descent**

What's an epoch?

- Canonical SGD: sample a random subset $J_k \subset \{1, \dots, m\}$ each iteration—sampling with replacement
- Practical SGD: shuffle the training set, and take a consecutive batch of size B (called **batch size**) each iteration—sampling without replacement

one pass of the shuffled training set is called one **epoch**.

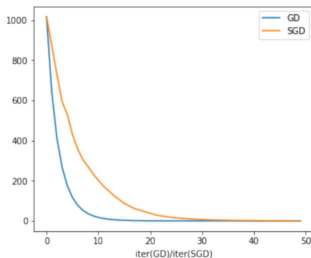
Practical stochastic gradient descent (SGD)

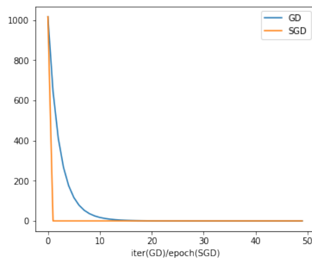
Input: init. $\boldsymbol{x}_0$, SC, batch size B, iteration counter $k = 1$, epoch counter $\ell = 1$

1: **while** SC not satisfied **do**
2: permute the index set $\{0, \cdots, m\}$ and divide it into batches of size B
3: **for** $i \in \{1, \dots, \#\text{batches}\}$ **do**
4: calculate the stochastic gradient $\widehat{\boldsymbol{g}_k}$ based on the ith batch
5: decide a step size t_k
6: make a step: $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} - t_k \widehat{\boldsymbol{g}_k}$
7: update iteration counter: $k = k + 1$
8: **end for**
9: update epoch counter: $\ell = \ell + 1$
10: **end while**

GD vs. SGD

Consider $\min_{\boldsymbol{w}} \ \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$, where $\boldsymbol{X} \in \mathbb{R}^{10000\times 500}$, $\boldsymbol{y} \in \mathbb{R}^{10000}$, $\boldsymbol{w} \in \mathbb{R}^{500}$





- By iteration: GD is faster
- By iter(GD)/epoch(SGD): SGD is faster
- Remember, cost of one epoch of SGD $\approx$ cost of one iteration of GD!

Overall, SGD could be quicker to find a medium-accuracy solution with lower cost, which suffices for most purposes in machine learning [Bottou and Bousquet, 2008].

Step size (learning rate) for SGD

Recall the recommended step size rule for GD: **back-tracking line search**

key idea: $F(\boldsymbol{x} - t\nabla F(\boldsymbol{x})) - F(\boldsymbol{x}) \approx -ct \|\nabla F(\boldsymbol{x})\|^2$ for a certain $c \in (0, 1)$

Shall we do it for SGD? No, but why?

- SGD tries to avoid the m factor in computing the full gradient $\nabla_{\boldsymbol{w}} F(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{w}} f(\boldsymbol{w}; \boldsymbol{\xi}_i)$, i.e., reducing m to B (batch size)
- But computing $F(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} f(\boldsymbol{w}; \boldsymbol{\xi}_i)$ or $F(\boldsymbol{w} - t\widehat{\boldsymbol{g}}) = \frac{1}{m}\sum_{i=1}^{m} f(\boldsymbol{w} - t\widehat{\boldsymbol{g}}; \boldsymbol{\xi}_i)$ brings back the m factor; similarly for ∇F
- What about computing approximations to the objective values based on small batches also? Approximation errors for F and ∇F may ruin the stability of the Taylor criterion

Step size (learning rate, or LR) for SGD

Classical theory for SGD on convex problems requires

$$\sum_k t_k = \infty, \quad \sum_k t_k^2 < \infty.$$

Practical implementation: diminishing step size/LR, e.g.,

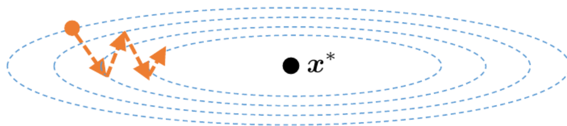
- $1/t$ **delay**: $t_k = \alpha/(1+\beta k)$, α, β: tunable parameters, k: iteration index
- **exponential delay**: $t_k = \alpha e^{-\beta k}$, α, β: tunable parameters, k: iteration index
- **staircase delay**: start from t_0, divide it by a factor (e.g., 5 or 10) every L (say, 10) epochs—popular in practice. Some heuristic variants:
  - watch the validation error and decrease the LR when it stagnates
  - watch the objective and decrease the LR when it stagnates

check out `torch.optim.lr_scheduler` in PyTorch! https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

Beyond the vanilla SGD

- Momentum/acceleration methods
- SGD with adaptive learning rates
- Stochastic 2nd order methods

Why momentum?



Credit: Princeton ELE522

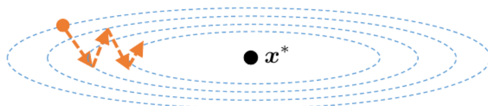
- GD is cheap ($O(n)$ per step) but overall convergence sensitive to conditioning
- Newton's convergence is not sensitive to conditioning but expensive ($O(n^3)$ per step)

A cheap way to achieve faster convergence? Answer: using historic information

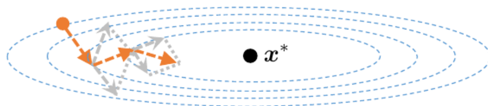
Heavy ball method

In physics, a heavy object has a large inertia/momentum — resistance to change velocity.

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k) + \beta_k \underbrace{(\boldsymbol{x}_k - \boldsymbol{x}_{k-1})}_{\textbf{momentum}} \quad \text{due to Polyak}$$



gradient descent



heavy-ball method

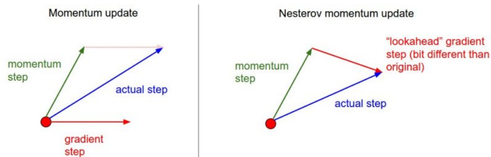
Credit: Princeton ELE522

History helps to smooth out the zig-zag path!

Nesterov's accelerated gradient methods

due to Y. Nesterov

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \beta_k \left(\boldsymbol{x}_k - \boldsymbol{x}_{k-1} \right) - \alpha_k \nabla f \left(\boldsymbol{x}_k + \beta_k \left(\boldsymbol{x}_k - \boldsymbol{x}_{k-1} \right) \right)$$



Credit: Stanford CS231N

$$\text{HB} \begin{cases} x_{\text{ahead}} = x + \beta(x - x_{\text{old}}), \\ x_{\text{new}} = x_{\text{ahead}} - \alpha \nabla f(x). \end{cases} \quad \text{Nesterov} \begin{cases} x_{\text{ahead}} = x + \beta(x - x_{\text{old}}), \\ x_{\text{new}} = x_{\text{ahead}} - \alpha \nabla f(x_{\text{ahead}}). \end{cases}$$

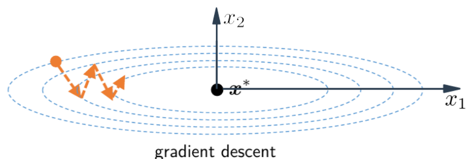
SGD with momentum/acceleration: replace the gradient term ∇f by the stochastic gradient $\widehat{g}$ based on small batches

check out `torch.optim.SGD` at (their convention slightly differs from here)

https://pytorch.org/docs/stable/optim.html#torch.optim.SGD

Why SGD with adaptive learning rate?

Recall the struggle of GD on elongated functions, e.g., $f(x_1, x_2) = x_1^2 + 4x_2^2$



– (Quasi-)Newton's method: take the full curvature info, but expensive

– Momentum methods: use historic direction(s) to cancel out wiggles

Another heuristic remedy: balance out movements in all coordinate directions.
Suppose $\boldsymbol{g}$ is the (stochastic) gradient, for all i,

divide g_i by historic gradient magnitudes in the ith coordinate

Benefit: coordinate directions always with small (large) derivatives get sped up (slowed down). Think of the above $f(x_1, x_2)$ example!

Method 1: Adagrad

divide g_i by historic gradient magnitudes in the ith coordinate

At the $(k+1)^{th}$ iteration, for all i,

$$x_{i,k+1} = x_{i,k} - t_k \frac{g_{i,k}}{\sqrt{\sum_{j=1}^{k} g_{i,j}^2 + \varepsilon}}$$

or in elementwise notation

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \frac{\boldsymbol{g}_k}{\sqrt{\sum_{j=1}^{k} \boldsymbol{g}_j^2 + \varepsilon}}$$

Write $\boldsymbol{s}_k \doteq \sum_{j=1}^{k} \boldsymbol{g}_j^2$. Note that $\boldsymbol{s}_k = \boldsymbol{s}_{k-1} + \boldsymbol{g}_k^2$. So only need to incrementally update the $\boldsymbol{s}_k$ sequence, which is cheap

In PyTorch, `torch.optim.Adagrad`

https://pytorch.org/docs/stable/optim.html#torch.optim.Adagrad

Method 2: RMSprop

Adagrad:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \frac{\boldsymbol{g}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}} \quad \text{with } \boldsymbol{s}_k \doteq \sum_{j=1}^{k} \boldsymbol{g}_j^2.$$

update equation for $\boldsymbol{s}_k$: $\boldsymbol{s}_k = \boldsymbol{s}_{k-1} + \boldsymbol{g}_k^2$

Problems:

- Magnitudes in $\boldsymbol{s}_k$ becomes larger when k grows, and hence movements $t_k \frac{\boldsymbol{g}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}}$ become small when k is large.
- Remote history may not be relevant

Solution: **RMSprop**—gradually phase out the history. For some $\beta \in (0, 1)$

$$\boldsymbol{s}_k = \beta \boldsymbol{s}_{k-1} + (1-\beta)\, \boldsymbol{g}_k^2 \iff \boldsymbol{s}_k = (1-\beta)\left(\boldsymbol{g}_k^2 + \beta \boldsymbol{g}_{k-1}^2 + \beta^2 \boldsymbol{g}_{k-2}^2 + \dots\right)$$

Typical values for β: $0.9, 0.99$. In PyTorch, `torch.optim.RMSprop`

https://pytorch.org/docs/stable/optim.html#torch.optim.RMSprop

Method 3: Adam

Combine RMSprop with momentum methods

$$
\begin{aligned}
\boldsymbol{m}_k &= \beta_1 \boldsymbol{m}_{k-1} + (1-\beta_1)\,\boldsymbol{g}_k && \text{(combine momentum and stochastic gradient)} \\
\boldsymbol{s}_k &= \beta_2 \boldsymbol{s}_{k-1} + (1-\beta_2)\,\boldsymbol{g}_k^2 && \text{(scaling factor update as in RMSprop)} \\
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k - t_k \frac{\boldsymbol{m}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}}
\end{aligned}
$$

- Typical parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon =$ 1e-8.
- Recommended method to use!
- In PyTorch, `torch.optim.Adam`
  https://pytorch.org/docs/stable/optim.html#torch.optim.Adam
- Several recent variants: `torch.optim.AdamW`, `torch.optim.SparseAdam`, `torch.optim.Adamax`

Thoughts on adaptive LR methods

- adapting the LR or adapting the (stochastic) gradient? Two views of the same thing ($\odot$ denotes elementwise product)

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \frac{t_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}} \odot \boldsymbol{g}_k \quad \text{vs.} \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \frac{\boldsymbol{g}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}}$$

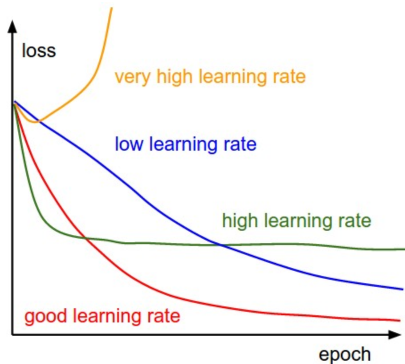
- adapting the gradient, familiar? What happens in Newton's method?

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \,\mathrm{diag}\left(\frac{1}{\sqrt{\boldsymbol{s}_k + \varepsilon}}\right) \boldsymbol{g}_k \quad \text{vs.} \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \boldsymbol{H}_k^{-1} \boldsymbol{g}_k.$$

  ... approximate the Hessian (inverse) with a diagonal matrix. So adaptive methods are approximate 2nd order methods, and more faithful approximation possible.

- Learning rate t_k: similar to that for the vanilla SGD, but less sensitive and can be large

Diagnosis of LR



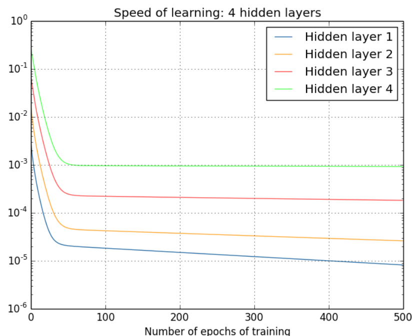
Credit: Stanford CS231N

- Low LR always leads to convergence, but takes forever
- Premature flattening is a sign of large LR; premature sloping is a sign of early stopping—increase the number of epochs!
- Remember the starecase LR schedule!

Why adaptive methods relevant for DL?

$$F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k) = \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \ldots (\boldsymbol{W}_1 \boldsymbol{x}_i))\right)\right)$$

Derivatives for early layers tend to be order of magnitude smaller than those for late layers, i.e., the **gradient vanishing/exploring phenomenon**



We'll explore more of this in HW3! See discussion in

http://neuralnetworksanddeeplearning.com/chap5.html

Why adaptive methods relevant for DL?

$$F(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k) = \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \ldots (\boldsymbol{W}_1 \boldsymbol{x}_i))\right)\right)$$

- Hypothesis: F has many saddle points and escaping saddle points causes the difficulty of training [Choromanska et al., 2015, Pascanu et al., 2014, Dauphin et al., 2014]
- Adaptive methods can escape saddle points efficiently; see, e.g., [Staib et al., 2020]

visualization comparison https://imgur.com/a/Hqolp

Stochastic 2nd order methods

Recall scalable 2nd order methods

- Quasi-Newton methods, esp. L-BFGS
- Trust-region methods

When #samples is large, we also want to use only mini batches to estimate any quantities of interest

- stochastic quasi-Newton methods: e.g., [Martens and Grosse, 2015] [Byrd et al., 2016] [Anil et al., 2020] [Roosta-Khorasani and Mahoney, 2018]
- stochastic trust-region methods: e.g., [Curtis and Shi, 2019], [Chauhan et al., 2018]

still active area of research. Hardware seems to be the main limiting factor

Outline

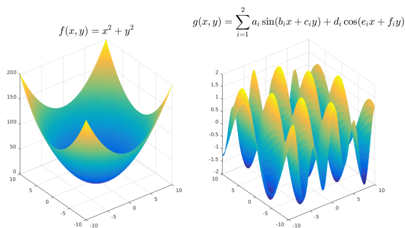
Three design choices

Training algorithms

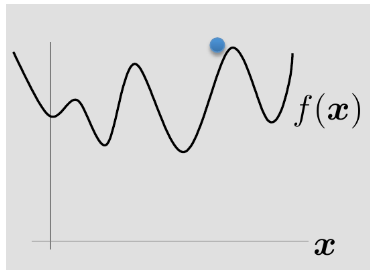
- Which method
- Where to start
- When to stop

Suggested reading

Where to initialize? the general picture



convex vs. nonconvex functions



- **Convex**: most iterative methods converge to the global min no matter the initialization
- **Nonconvex**: initialization matters a lot. Common heuristics: random initialization, multiple independent runs
- **Nonconvex**: clever initialization is possible with certain assumptions on the data:

https://sunju.org/research/nonconvex/

and sometimes random initialization works!

Where to initialize for DNNs?

$$F(\boldsymbol{W}_1, \dots, \boldsymbol{W}_k) = \frac{1}{m} \sum_{i=1}^m \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \dots (\boldsymbol{W}_1 \boldsymbol{x}_i))\right)\right)$$

- Are there bad initializations? Consider a simple case

$$F(\boldsymbol{W}_1, \boldsymbol{W}_2) = \frac{1}{m} \sum_{i=1}^m \left\| \boldsymbol{y}_i - \boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right) \right\|_2^2$$

$$\nabla_{\boldsymbol{W}_1} F(\boldsymbol{W}_1, \boldsymbol{W}_2) = -\frac{2}{m} \sum_{i=1}^m \left[\boldsymbol{W}_2^\intercal \left(\boldsymbol{y}_i - \boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right) \odot \sigma'\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right) \right] \boldsymbol{x}_i^\intercal$$

  * What about $\boldsymbol{W} = \mathbf{0}$? $\nabla_{\boldsymbol{W}_1} F = \mathbf{0}$—no movement on $\boldsymbol{W}_1$
  * What about very large (small) $\boldsymbol{W}$? Large (small) value & gradient—the problem becomes significant when there are more layers

- Are there principled ways of initialization?

  * random initialization with proper scaling
  * orthogonal initialization

Random initialization

Idea: make all entries in $\boldsymbol{W}$ iid random, and also $\boldsymbol{W}_i$'s and $\boldsymbol{W}_i^\intercal$'s "well behaved"

> **A reasonable goal**: if all entries in $\boldsymbol{v} \in \mathbb{R}^d$ are independent and have zero mean, unit variance, the output $\sigma(\boldsymbol{w}^\intercal \boldsymbol{v}) \in \mathbb{R}$ (i.e., output of a single neuron) has a unit variance.

To seek a specific setting for $\boldsymbol{w} \in \mathbb{R}^d$, suppose $\boldsymbol{w}$ is iid with zero mean and σ is identity. Then:

$$\mathrm{Var}(\boldsymbol{w}^\intercal \boldsymbol{v}) = \mathrm{Var}\left(\sum_i w_i v_i\right) = \sum_i \mathrm{Var}(w_i v_i) = \sum_i \mathrm{Var}(w_i)\,\mathrm{Var}(v_i) = d\mathrm{Var}(w_i).$$

To make $\mathrm{Var}(\boldsymbol{w}^\intercal \boldsymbol{v}) = 1$, we will set $\mathrm{Var}(w_i) = 1/d$.

> For $\boldsymbol{W}_i$ with d inputs, set $\boldsymbol{W}_i$ iid zero-mean and $1/d$ variance

Random initialization

For $\boldsymbol{W}_i$ with d_{in} inputs, set $\boldsymbol{W}_i$ iid zero-mean and $1/d_{\text{in}}$ variance

A similar consideration of $\boldsymbol{W}_i^\intercal$ (due to its role in the gradient) also suggests that

For $\boldsymbol{W}_i$ with d_{out} outputs, set $\boldsymbol{W}_i$ iid zero-mean and $1/d_{\text{out}}$-variance

Xavier Initialization: set $\boldsymbol{W}_i \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ iid zero-mean and $\frac{2}{d_{\text{in}}+d_{\text{out}}}$-variance. For example:

- $\boldsymbol{W}_i \sim_{iid} \mathcal{N}\left(0, \frac{2}{d_{\text{in}}+d_{\text{out}}}\right)$ `torch.nn.init.xavier_normal_`
- $\boldsymbol{W}_i \sim_{iid} \text{uniform}\left(-\sqrt{\frac{6}{d_{\text{in}}+d_{\text{out}}}}, \sqrt{\frac{6}{d_{\text{in}}+d_{\text{out}}}}\right)$ `torch.nn.init.xavier_uniform_`

Random initialization

Recall our derivation assumed σ is identity, which may not be accurate.

For ReLU, based on the same assumptions on $\boldsymbol{v}$ and $\boldsymbol{w}$ as before:

$$\begin{aligned}\mathbb{E}\left[\mathrm{ReLU}\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right)\right] &= 0,\\ \mathrm{Var}\left(\mathrm{ReLU}\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right)\right) &= \mathbb{E}\left[\mathrm{ReLU}^2\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right)\right] = \frac{1}{2}\mathbb{E}\left[\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right)^2\right]\\ &= \frac{1}{2}\mathrm{Var}\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right) = \frac{1}{2} d\,\mathrm{Var}\left(w_i\right).\end{aligned}$$

Kaiming Initialization (for ReLU): set $\boldsymbol{W}_i \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ iid zero-mean and $\frac{2}{d_{\text{in}}}$-variance. For example:

- $\boldsymbol{W}_i \sim_{iid} \mathcal{N}\left(0, \frac{2}{d_{\text{in}}}\right)$ `torch.nn.init.kaiming_normal_`
- $\boldsymbol{W}_i \sim_{iid} \mathrm{uniform}\left(-\sqrt{\frac{6}{d_{\text{in}}}}, \sqrt{\frac{6}{d_{\text{in}}}}\right)$ `torch.nn.init.kaiming_uniform_`

But it only accounts for d_{in} or d_{out}; a proposed modification: set the variance to $\frac{c}{\sqrt{d_{\text{in}} d_{\text{out}}}}$ for some constant c [Defazio and Bottou, 2019]

Orthogonal initialization

Making all $\boldsymbol{W}_i$'s orthonormal is empirically shown to lead to competitive performance with fewer tricks (covered next lectures). See Sec 4.2 of [Sun, 2019] `torch.nn.init.orthogonal_`

There is a body of research proposing contraining/regularizing $\boldsymbol{W}_i$'s to be orthonormal, e.g., [Arjovsky et al., 2016, Bansal et al., 2018, Lezcano-Casado and Martínez-Rubio, 2019, Li et al., 2020]

See also the modified PyTorch package that allows manifold constraints

https://github.com/mctorch/mctorch

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

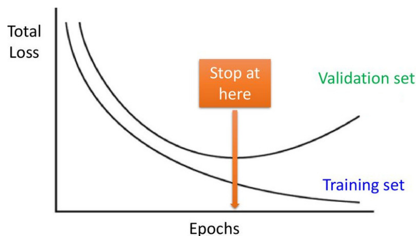
Suggested reading

When to stop in training DNNs?

Recall that a natural stopping criterion for general GD is $\|\nabla f(\boldsymbol{w})\| \le \varepsilon$ for a small ε. Is this good when training DNNs?

- Computing $\nabla f(\boldsymbol{w})$ each iterate is expensive (recall why we moved from GD to SGD)
- Stochastic gradient is inherently **noisy**—the norm at a true critical point may be large

A practical/pragmatic stopping strategy: **early stopping**



... periodically check the validation error and stop when it doesn't improve

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Suggested reading

Suggested reading

- Sun, Ruoyu. "Optimization for deep learning: theory and algorithms." arXiv preprint arXiv:1912.08957 (2019).
- UIUC IE598-ODL Optimization Theory for Deep Learning https://wiki.illinois.edu/wiki/display/IE598ODLSP19/IE598-ODL++Optimization+Theory+for+Deep+Learning
- Stanford CS231n course notes: Neural Networks Part 1: Setting up the Architecture https://cs231n.github.io/neural-networks-1/
- Stanford CS231n course notes: Neural Networks Part 2: Setting up the Data and the Loss https://cs231n.github.io/neural-networks-2/
- Stanford CS231n course notes: Neural Networks Part 3: Learning and Evaluation https://cs231n.github.io/neural-networks-3/

References i

[Anil et al., 2020] Anil, R., Gupta, V., Koren, T., Regan, K., and Singer, Y. (2020). **Second order optimization made practical.** *arXiv:2002.09018*.

[Arjovsky et al., 2016] Arjovsky, M., Shah, A., and Bengio, Y. (2016). **Unitary evolution recurrent neural networks.** In *International Conference on Machine Learning*, pages 1120–1128.

[Bansal et al., 2018] Bansal, N., Chen, X., and Wang, Z. (2018). **Can we gain more from orthogonality regularizations in training deep cnns?** In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 4266–4276. Curran Associates Inc.

[Baydin et al., 2017] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2017). **Automatic differentiation in machine learning: a survey.** *The Journal of Machine Learning Research*, 18(1):5595–5637.

[Bottou and Bousquet, 2008] Bottou, L. and Bousquet, O. (2008). **The tradeoffs of large scale learning.** In *Advances in neural information processing systems*, pages 161–168.

References ii

[Byrd et al., 2016] Byrd, R. H., Hansen, S. L., Nocedal, J., and Singer, Y. (2016). **A stochastic quasi-newton method for large-scale optimization.** *SIAM Journal on Optimization*, 26(2):1008–1031.

[Chauhan et al., 2018] Chauhan, V. K., Sharma, A., and Dahiya, K. (2018). **Stochastic trust region inexact newton method for large-scale machine learning.** *arXiv:1812.10426*.

[Choromanska et al., 2015] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). **The loss surfaces of multilayer networks.** In *Artificial intelligence and statistics*, pages 192–204.

[Curtis and Shi, 2019] Curtis, F. E. and Shi, R. (2019). **A fully stochastic second-order trust region method.** *arXiv:1911.06920*.

[Dauphin et al., 2014] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). **Identifying and attacking the saddle point problem in high-dimensional non-convex optimization.** In *Advances in neural information processing systems*, pages 2933–2941.

[Defazio and Bottou, 2019] Defazio, A. and Bottou, L. (2019). **Scaling laws for the principled design, initialization and preconditioning of relu networks.** *arXiv:1906.04267*.

[Lezcano-Casado and Martínez-Rubio, 2019] Lezcano-Casado, M. and Martínez-Rubio, D. (2019). **Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group.** *arXiv1901.08428*.

[Li et al., 2020] Li, J., Fuxin, L., and Todorovic, S. (2020). **Efficient riemannian optimization on the stiefel manifold via the cayley transform.** *arXiv:2002.01113*.

[Martens and Grosse, 2015] Martens, J. and Grosse, R. (2015). **Optimizing neural networks with kronecker-factored approximate curvature.** In *International conference on machine learning*, pages 2408–2417.

[Pascanu et al., 2014] Pascanu, R., Dauphin, Y. N., Ganguli, S., and Bengio, Y. (2014). **On the saddle point problem for non-convex optimization.** *arXiv preprint arXiv:1405.4604*.

[Roosta-Khorasani and Mahoney, 2018] Roosta-Khorasani, F. and Mahoney, M. W. (2018). **Sub-sampled newton methods.** *Mathematical Programming*, 174(1-2):293–326.

References iv

[Staib et al., 2020] Staib, M., Reddi, S. J., Kale, S., Kumar, S., and Sra, S. (2020). **Escaping saddle points with adaptive gradient methods.** *arXiv:1901.09149*.

[Sun, 2019] Sun, R. (2019). **Optimization for deep learning: theory and algorithms.** *arXiv:1912.08957*.