Optimization for Deep Learning

Slides by Viktor Lempitsky

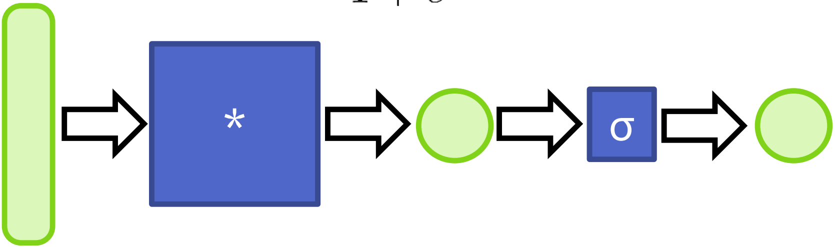
Deep learning: recap

End-to-end joint learning of all layers:

- multiple assembleable blocks
- each block is piecewise-differentiable
- gradient-based optimization
- gradients computed by backpropagation

Recap: training logistic regression

$$P(y(x) = y_i|w) = \frac{1}{1+e^{-y_i w^T x_i}} = \sigma(y_i w^T x)$$



$$E(w) = -\sum_{i=1}^{N} \log P(y(x) = y_i|w) = \sum_{i=1}^{N} \log(1+e^{-y_i w^T x_i})$$

Softmax (sigmoid generalization)

Softmax (generalizes logistic):

$$x \xrightarrow{W*} \begin{bmatrix} w_1^T x \\ w_2^T x \\ \ldots \\ w_L^T x \end{bmatrix}$$

$$\Big\downarrow \textit{exp}$$

$$\begin{bmatrix} \exp(w_1^T x) \\ \exp(w_2^T x) \\ \ldots \\ \exp(w_L^T x) \end{bmatrix} \xrightarrow{\textit{norm}} \begin{bmatrix} \dfrac{\exp(w_1^T x)}{\Sigma_i \exp(w_i^T x)} \\ \dfrac{\exp(w_2^T x)}{\Sigma_i \exp(w_i^T x)} \\ \ldots \\ \dfrac{\exp(w_L^T x)}{\Sigma_i \exp(w_i^T x)} \end{bmatrix}$$

$$P(y(x)=1) = \frac{\exp(w_1^T x)}{\Sigma_i \exp(w_i^T x)}$$

Multinomial logistic loss

$$P(y(x) = i) = \frac{\exp w_i^T x}{\sum_j \exp w_j^T x}$$

Multinomial log loss (generalizes logistic loss):

$$\begin{aligned} E(w) &= -\sum_i \log P(y(x_i) = y_i) = \\ &= -\sum_i \left[w_{y_i}^T x_i - \log \sum_j \exp w_j^T x_i \right] \end{aligned}$$

(Part of the) gradient over w_j:

$$\frac{dE}{dw_j} = -\sum_i x_i \left([y_i == j] - P(y(x_i) = j) \right)$$

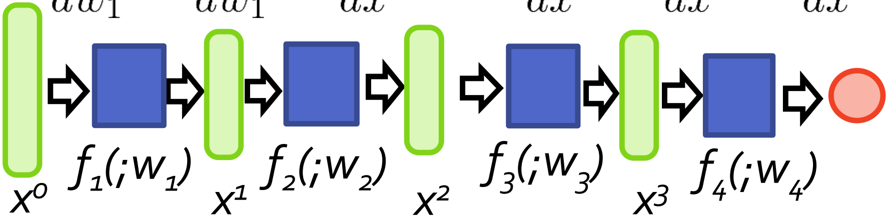
Sequential computation: *backpropagation*

$\frac{dz}{dx^3}, \frac{dz}{dw_4}$ can be computed

$$\frac{dz}{dw_3} = \frac{dx^3}{dw_3}^T \cdot \frac{dz}{dx^3} \qquad \frac{dz}{dx^2} = \frac{dx^3}{dx^2}^T \cdot \frac{dz}{dx^3}$$

$$\frac{dz}{dw_2} = \frac{dx^2}{dw_2}^T \cdot \frac{dz}{dx^2} \qquad \frac{dz}{dx^1} = \frac{dx^2}{dx^1}^T \cdot \frac{dz}{dx^2}$$

$$\frac{dz}{dw_1} = \frac{dx^1}{dw_1}^T \cdot \frac{dz}{dx^1} \qquad \frac{dz}{dx^0} = \frac{dx^1}{dx^0}^T \cdot \frac{dz}{dx^1}$$



Optimization for supervised ML

- *R(w)* denotes regularization e.g. $||w||^2$

- *$l(x_i, y_i, w)$* denotes loss for *i*-th example, e.g. – *$\log P(y(x_i) = y_i \mid w)$*

- The optimization objective is then:

$$E(w) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, w) + \lambda R(w)$$

Small scale setting: traditional optimization

$$E(w) = \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i, w) + \lambda R(w)$$

- Data are few, we can look through it at each optimization iteration
- Use adapted versions of standard optimization methods (gradient descent, quasi-Newton, quadratic programming,…)

Large-scale learning

$$E(w) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, y_i, w) + \lambda R(w)$$

$$\frac{dE}{dw} = \frac{1}{N} \sum_{i=1}^{N} \frac{dl(x_i, y_i, w)}{dw} + \lambda \frac{dR}{dw}$$

- Evaluating gradient is very expensive
- It will only be good for one (small) step

Stochastic gradient descent (SGD) idea:

- Evaluate a coarse approximation to grad
- Make "quick" steps

Stochastic gradient descent (SGD)

Gradient:

$$\frac{dE}{dw} = \frac{1}{N}\sum_{i=1}^{N}\frac{dl(x_i, y_i, w)}{dw} + \lambda\frac{dR}{dw}$$

Stochastic gradient:

$$\frac{dE^i}{dw} = \frac{dl(x_i, y_i, w)}{dw} + \lambda\frac{dR}{dw}$$

Stochastic gradient is an unbiased estimate of the gradient:

$$\frac{dE}{dw} = \frac{1}{N}\sum_{i=1}^{N}\frac{dE^i}{dw}$$

Stochastic gradient descent (SGD)

SGD: $$\boxed{\begin{array}{l} v[t] = -\alpha[t]\,\nabla(E, w[t]) \\ w[t+1] = w[t] + v[t] \end{array}}$$

where $\nabla(E, w[t]) = \dfrac{dE^{i(t)}}{dw}(w[t])$

- *i(t)* usually follow random permutations of training data
- One sweep over training data is called an **epoch**

Stochastic gradient descent (SGD)

SGD:

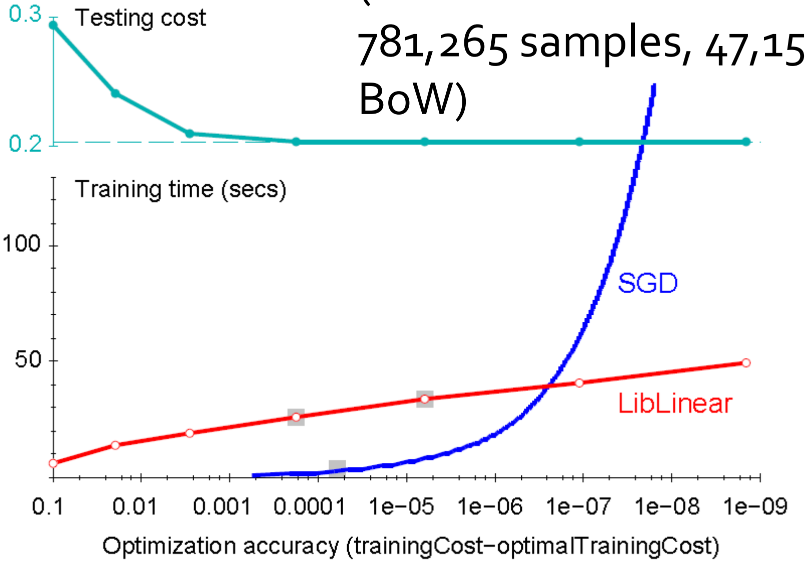
$$\boxed{\begin{aligned} v[t] &= -\alpha[t]\,\nabla(E, w[t]) \\ w[t+1] &= w[t] + v[t] \end{aligned}}$$

- One sweep over training data is called an **epoch**
- Popular choices for schedule $\alpha[t]$:
    - constant, e.g. $\alpha[t] = 0.0001$
    - piecewise constant, e.g. $\alpha[t]$ is decreased tenfold every N epochs
    - harmonic, e.g. $\alpha[t] = 0.001 / ([t/N]+10)$

The efficiency of SGD (“shallow” learning)

[L.Bottou]

(Document classification : 781,265 samples, 47,152-dim BoW)



Batch SGD

Gradient:

$$\frac{dE}{dw} = \frac{1}{N}\sum_{i=1}^{N}\frac{dl(x_i, y_i, w)}{dw} + \lambda\frac{dR}{dw}$$

Batch (aka mini-batch):

$$\{b_1, b_2, \ldots b_{N_b}\} \subset 1 \ldots N$$

Batch stochastic gradient:

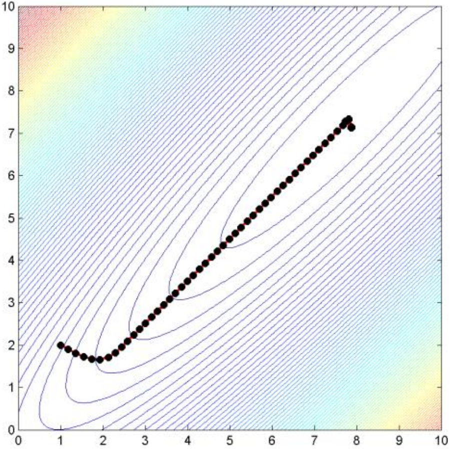
$$\frac{dE}{dw} = \frac{1}{N_b}\sum_{i=1}^{N_b}\frac{dl(x_{b(i)}, y_{b(i)}, w)}{dw} + \lambda\frac{dR}{dw}$$

Why do batching?

$$\frac{dE}{dw} = \frac{1}{N_b} \sum_{i=1}^{N_b} \frac{dl(x_{b(i)}, y_{b(i)}, w)}{dw} + \lambda \frac{dR}{dw}$$

- “Less stochastic” approximation, more stable convergence (questionable)
- **Main reason:** all modern architectures have parallelism, hence computing mini-batch grad is often as cheap as a single stochastic grad

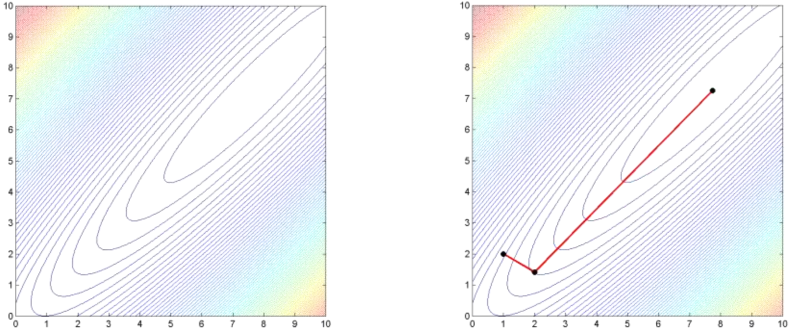
SGD inherits gradient descent problems



- Gradient descent is very poor “in ravines”
- SGD is no better

Better optimization methods

- Second order methods (Newton, Quasi-Newton)
- Krylov subspace methods, in particular *conjugate gradients*



Improving SGD using momentum

- Conjugate gradients use a combination of the current gradient and previous direction for the next step
- Similar idea for SGD (*momentum*):

$$\boxed{\begin{aligned} v[t] &= -\alpha[t]\,\nabla(E, w[t]) \\ w[t+1] &= w[t] + v[t] \end{aligned}}$$

$$\Downarrow$$

$$\boxed{\begin{aligned} v[t] &= \mu\, v[t-1] - \alpha[t]\,\nabla(E, w[t]) \\ w[t+1] &= w[t] + v[t] \end{aligned}}$$

Typical $\mu = 0.9$

Exponentially decaying running average

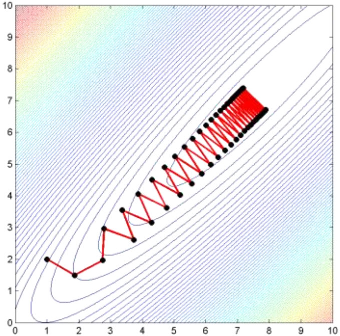
$$\boxed{\begin{aligned} &v[t] = \mu\, v[t\text{-}1] - \alpha[t]\, \nabla(E, w[t]) \\ &w[t\text{+}1] = w[t] + v[t] \end{aligned}}$$

$$\begin{aligned}
&v[t] = \mu\, v[t\text{-}1] - \alpha[t]\ \nabla(E, w[t]) = \\
&= \mu^2\, v[t\text{-}2] - \mu\alpha[t\text{-}1]\ \nabla(E, w[t\text{-}1]) \\
&\qquad\qquad - \alpha[t]\ \nabla(f, w[t]) = \\
&= \mu^3\, v[t\text{-}3] - \mu^2\, \alpha[t\text{-}2]\ \nabla(E, w[t\text{-}2]) \\
&\quad - \mu\alpha[t\text{-}1]\ \nabla(E, w[t\text{-}1]) - \alpha[t]\ \nabla(E, w[t]) = \\
&= \mu^{k+1}\, v[t\text{-}k\text{-}1] + \sum_{i=0}^{k} \mu^i \alpha[t-i] \nabla(E, w[t-i])
\end{aligned}$$

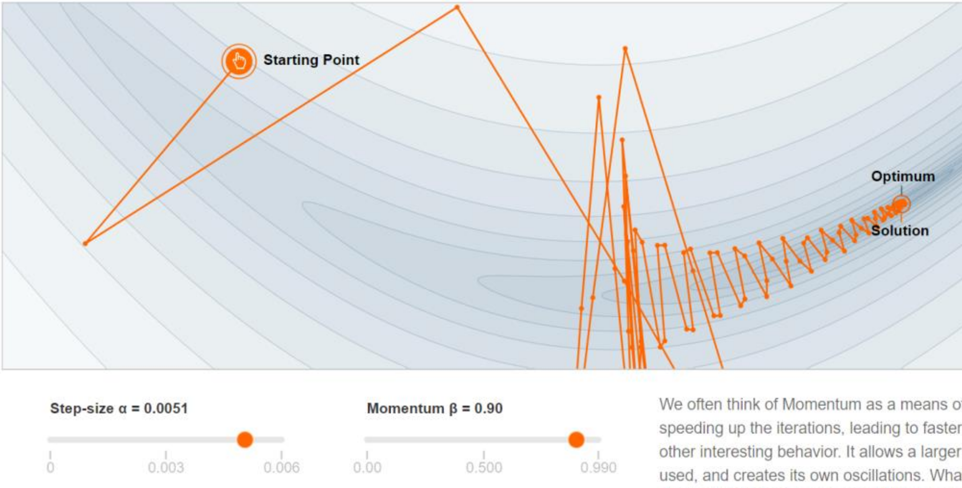
Momentum: why it works

$$v[t] \approx \sum_{i=0}^{k} \mu^i \alpha[t-i] \nabla(E, w[t-i])$$

- Smoothes out noise in SGD (~bigger batches)
- **Smoothes out oscilations inherent to gradient descent**
- Escapes local minima



The effect of the momentum



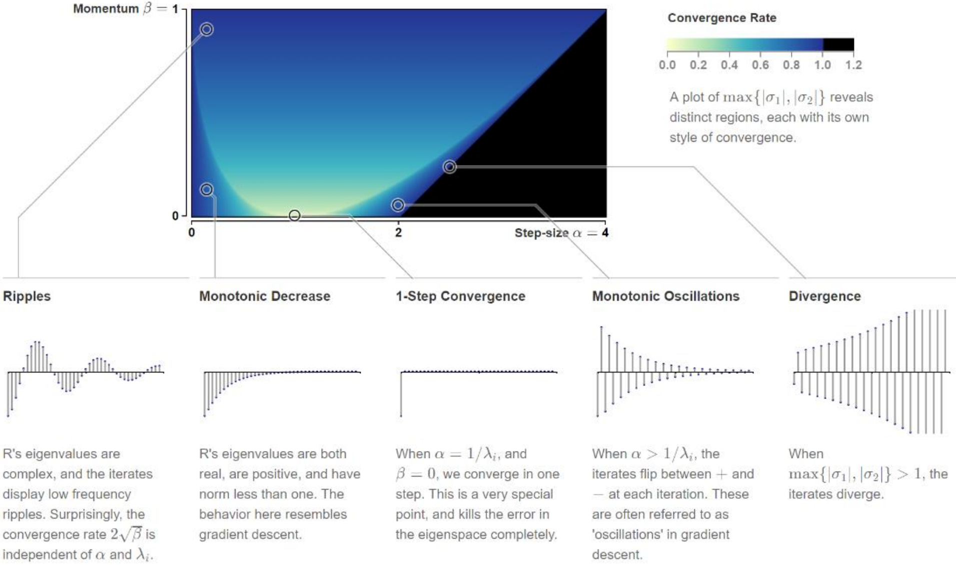
Step-size α = 0.0051

Momentum β = 0.90

We often think of Momentum as a means o
speeding up the iterations, leading to faster
other interesting behavior. It allows a larger
used, and creates its own oscillations. Wha

[Goh, Distill 2017]

Phase space along a single eigenvector



A plot of $\max\{|\sigma_1|, |\sigma_2|\}$ reveals distinct regions, each with its own style of convergence.

Ripples

R's eigenvalues are complex, and the iterates display low frequency ripples. Surprisingly, the convergence rate $2\sqrt{\beta}$ is independent of α and λ_i.

Monotonic Decrease

R's eigenvalues are both real, are positive, and have norm less than one. The behavior here resembles gradient descent.

1-Step Convergence

When $\alpha = 1/\lambda_i$, and $\beta = 0$, we converge in one step. This is a very special point, and kills the error in the eigenspace completely.

Monotonic Oscillations

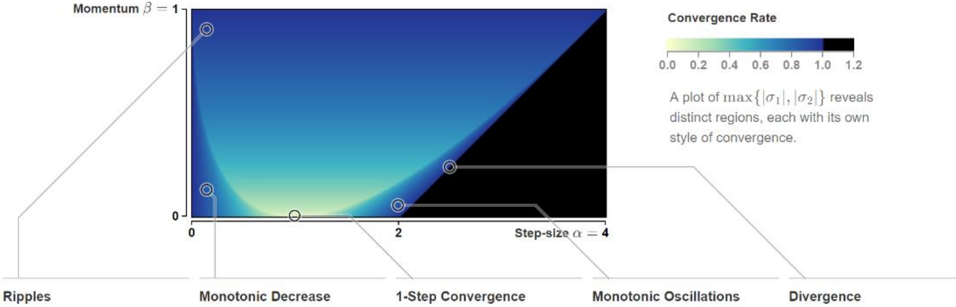
When $\alpha > 1/\lambda_i$, the iterates flip between $+$ and $-$ at each iteration. These are often referred to as 'oscillations' in gradient descent.

Divergence

When $\max\{|\sigma_1|, |\sigma_2|\} > 1$, the iterates diverge.

[Goh, Distill 2017]

Momentum: multiple eigenvalues



A plot of $\max\{|\sigma_1|, |\sigma_2|\}$ reveals distinct regions, each with its own style of convergence.

Optimal rate & momentum:

$$\alpha = \left(\frac{2}{\sqrt{\lambda_1} + \sqrt{\lambda_n}}\right)^2 \qquad \beta = \left(\frac{\sqrt{\lambda_n} - \sqrt{\lambda_1}}{\sqrt{\lambda_n} + \sqrt{\lambda_1}}\right)^2$$

Optimal speed:

$$\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$$

Convergence rate, **Momentum**

$$\frac{\kappa - 1}{\kappa + 1}$$

Convergence rate, **Gradient Descent**

[Goh, Distill 2017]

Momentum: multiple eigenvalues

Optimal rate & momentum:

$$\alpha = \left(\frac{2}{\sqrt{\lambda_1}+\sqrt{\lambda_n}}\right)^2 \quad \beta = \left(\frac{\sqrt{\lambda_n}-\sqrt{\lambda_1}}{\sqrt{\lambda_n}+\sqrt{\lambda_1}}\right)^2$$

In real network we do not know eigenvalues, so:

- we set the momentum high (e.g. 0.9)
- then we tune the learning rate

[Goh, Distill 2017]

Nesterov accelerated gradient

$$\boxed{\begin{aligned} v[t] &= \mu\, v[t-1] - \alpha[t]\, \nabla(E, w[t]) \\ w[t+1] &= w[t] + v[t] \end{aligned}}$$

Before we even compute the gradient, we have a good approximation where we will end up: $w[t+1] \approx w[t] + \mu\, v[t-1]$

Let us use this knowledge:

$$\boxed{\begin{aligned} v[t] &= \mu\, v[t-1] - \alpha[t]\, \nabla(E, w[t] + \mu\, v[t-1]) \\ w[t+1] &= w[t] + v[t] \end{aligned}}$$

(Computing the gradient at a more relevant spot)

Second-order methods

- Exponential smoothing helps, but still not optimal if large anisotropy exists
- Classic (Newton) solution: estimate the Hessian and make the update $v[t+1] = -H[t]^{-1} \nabla(E, w[t])$ (the lower the curvature the faster we go)
- Quasi-Newton methods: estimate some approximation to Hessian based on observed gradients
- Quasi-Newton can be used in batch mode, but same batch should be used over several iterations (why?)

Adagrad method [Duchi et al. 2011]

Adagrad idea: scale updates along different dimensions according to accumulated gradient magnitude

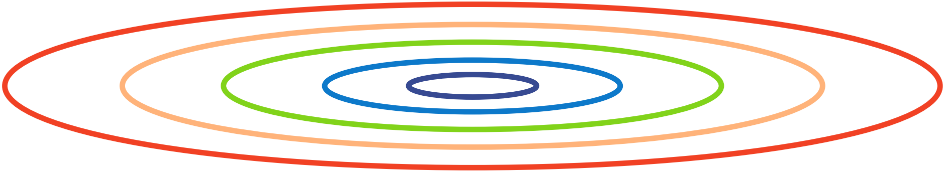
$$\boxed{\begin{aligned} g[t] &= g[t-1] + \nabla(E, w[t]) \odot \nabla(E, w[t]) \\ w[t+1] &= w[t] - \frac{\alpha}{\sqrt{g[t]+\epsilon}} \odot \nabla(E, w[t]) \end{aligned}}$$

Note: step lengths automatically decrease (perhaps too quickly).

Adagrad method [Duchi et al. 2011]

$$g[t] = g[t-1] + \nabla(E, w[t]) \odot \nabla(E, w[t])$$

$$w[t+1] = w[t] - \frac{\alpha}{\sqrt{g[t] + \epsilon}} \odot \nabla(E, w[t])$$



Adagrad in this case: find out that "vertical" derivatives are bigger, then make "vertical" steps smaller than "horizontal"

RMSPROP method [Hinton 2012]

Same as Adagrad, but replace accumulation of squared gradient with running averaging:

$$g[t] = \mu\, g[t-1] + (1-\mu)\, \nabla(E, w[t]) \odot \nabla(E, w[t])$$

$$w[t+1] = w[t] - \frac{\alpha[t]}{\sqrt{g[t] + \epsilon}} \odot \nabla(E, w[t])$$

Comparison: logistic regression

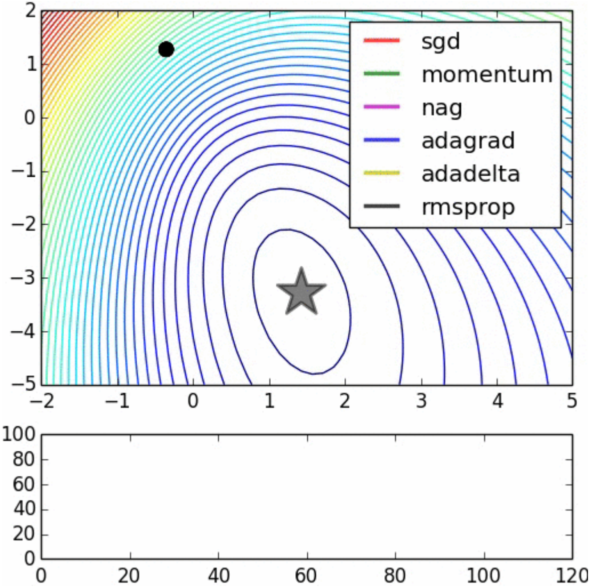


Image credit: Alec Redford

Further comparison

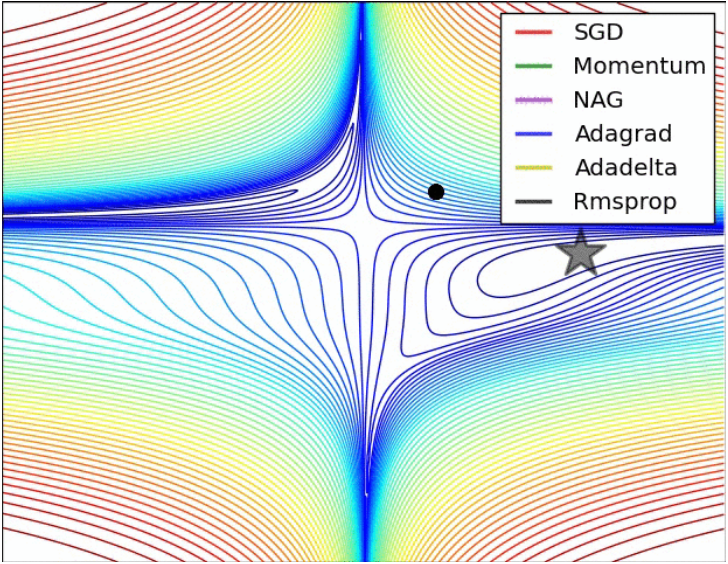


Image credit: Alec Redford

Further comparison: escaping from a saddle

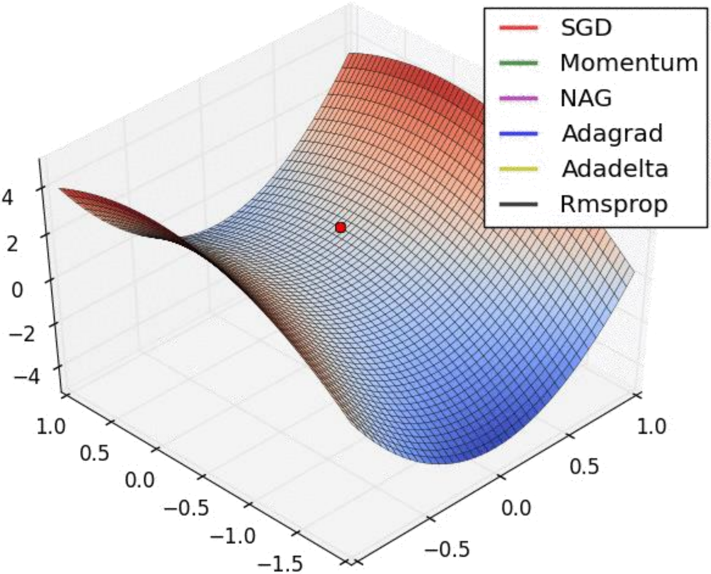


Image credit: Alec Redford

ADAM method [Kingma & Ba 2015]

ADAM = “ADAptive Moment Estimation”

$$v[t] = \beta\, v[t-1] + (1-\beta)\, \nabla(E, w[t])$$

$$g[t] = \mu\, g[t-1] + (1-\mu) \nabla(E, w[t]) \odot \nabla(E, w[t])$$

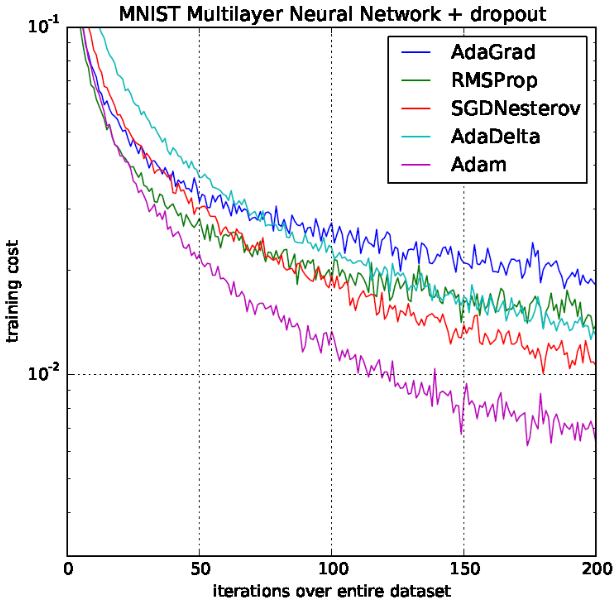
$$w[t+1] = w[t] - \alpha \frac{1}{\sqrt{g[t] + \epsilon}} \odot v[t]$$

$1 - \beta^t$

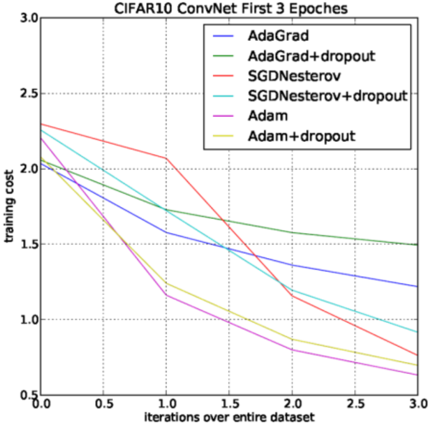
$1 - \mu^t$

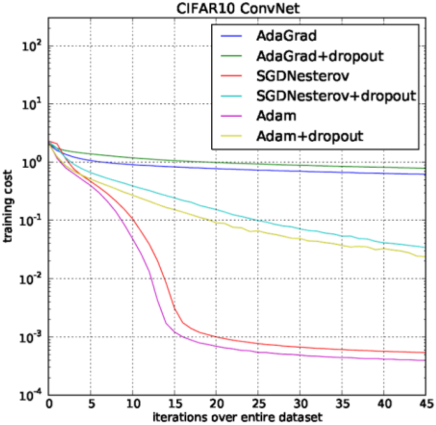
Recommended values: $\beta = 0.9$, $\mu = 0.999$, $\alpha = 0.001$, $\varepsilon = 10^{-8}$

ADAM method [Kingma & Ba 2015]



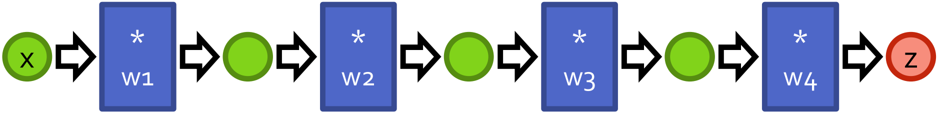
ADAM method [Kingma & Ba 2015]





Recap: optimization methods for DL

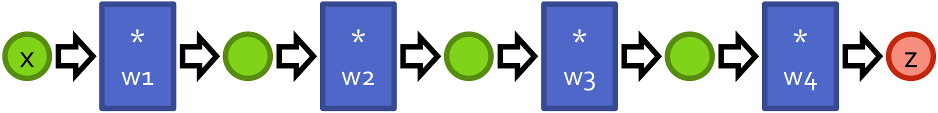
- Stochastic optimization is used always
- Optimization methods are not trying to estimate full Hessian (ignoring interaction between variables)



Toy example: $z = w_4 \, w_3 \, w_2 \, w_1 \, x$

$$\frac{dz}{dw_2} = w_4 \, w_3 \, w_1 \, x$$

Problems with DL optimization

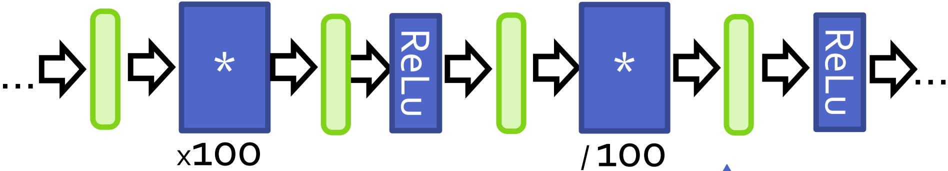


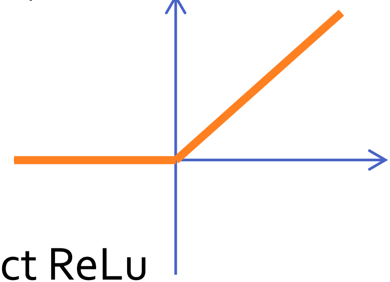
Toy example: $z = w_4\, w_3\, w_2\, w_1\, x$

$$\frac{dz}{dw_2} = w_4\, w_3\, w_1\, x \qquad \frac{dz}{dw_3} = w_4\, w_2\, w_1\, x$$

- *w = (1, 1, 1, 1)* and *w = (1,0.01,100,1)* define the same function ("gauge freedom"), but very different derivatives
- In the first case, derivatives and values are of order 1.
- In the second case, derivatives and values are wildly different

Gauge freedom in ReLu Networks



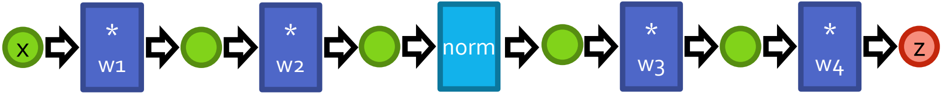


$\alpha > 0$

$$1/\alpha \, \mathrm{ReLu}(\alpha\, x) = \mathrm{ReLu}\,(x)$$

Thus: we can easily construct ReLu networks with **different** weights implementing the **same** function

Normalizing in the toy example



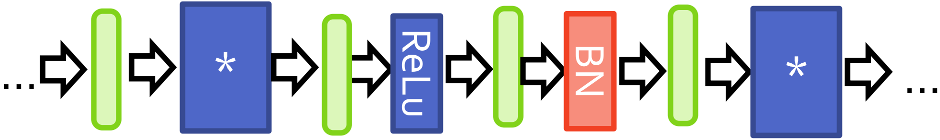
Toy example: $$z_i = w_4\, w_3\, \frac{w_2\, w_1\, x_i}{\frac{1}{N}\sum_{j=1}^{N} w_2\, w_1\, x_j}$$

$$\left[\frac{dz}{dw_3}\right]^i = w_4\, \frac{w_2\, w_1\, x_i}{\frac{1}{N}\sum_{j=1}^{N} w_2\, w_1\, x_j}$$

- Now, increasing w2 or w1 100x times will not change the partial derivative w.r.t. w_3 !
- The learning will become more stable

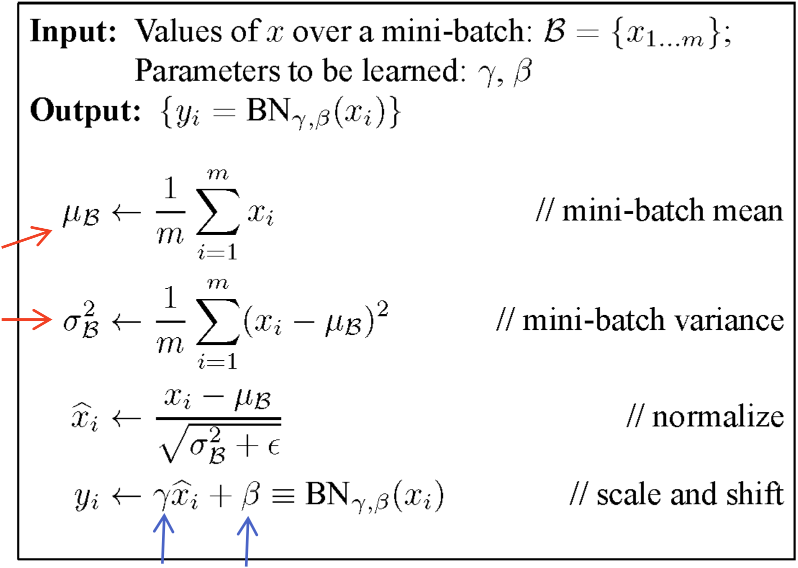
Batch normalization

[Szegedy and Ioffe 2015]



- Makes the training process invariant to some re-parameterizations
- Eliminates the bulk of cross-layer correlation between derivatives (off-diagonal Hessian vals)
- Use mini-batch statistics at training time to ensure that neuron activations are distributed “nicely” and the learning proceeds

Batch normalization layer



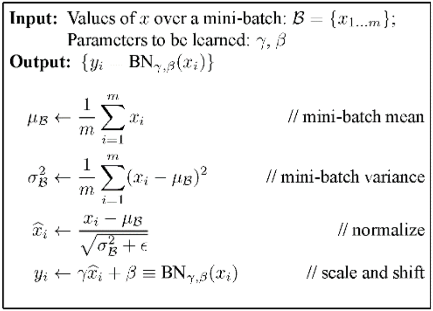
```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$    // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$    // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$    // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$    // scale and shift
```

covariant to reparameterization

learnable by SGD

[Szegedy and Ioffe 2015]

Batch normalization layer



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- At training time mean and variance are estimated per batch
- At test time, usually (running) averages over the dataset are used
- At test time, batch norm can be “merged in”
- For small batches, this is a big test<->train mismatch ☹

Solutions to train-test mismatch:

- Keep training time behavior
- Switch to test behavior and fine-tune

Alternatives to BatchNorm

- Layer Norm [Ba et al. NIPS’16], Instance Norm [Ulyanov et al.Arxiv16], Group renorm [Wu and He, ECCV18] – normalize over statistics of certain specific groups of variables **within** the same sample
- Batch Renorm [Ioffe NIPS’17]: gradually switch between train and test time behavior during training
- Weight norm [Salimans and Kingma NIPS’16]: decouple direction and magnitude of weight matrices

Initialization schemes

- **Basic idea 1:** units should be initialized to have comparable total input weights
- **Basic idea 2:** use layers which keep magnitude (otherwise both forwardprop and backprop will suffer from explosion/attenuation to zero; normalization layers solve this issue)
- E.g. [Glorot&Bengio 2010] aka "Xavier-initialization":

$$W \sim U\Big[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}\Big]$$

- E.g. [He et al, Arxiv15] for ReLu networks:

$$W \sim \mathcal{N}(0, \sqrt{2/n_i})$$

Recap

- Batch SGD optimization is used in large-scale setting
- Advanced SGD methods use running averages to smooth and rescale SGD steps
- Normalization layers are important and used in most modern deep architectures

Bibliography

Léon Bottou, Olivier Bousquet:
The Tradeoffs of Large Scale Learning. NIPS 2007: 161-168

Nesterov, Yurii. "A method of solving a convex programming problem with convergence rate O (1/k2)." Soviet Mathematics Doklady. Vol. 27. No. 2. 1983.

G. Goh, Why momentum really works? DISTILL 2017
https://distill.pub/2017/momentum/

John C. Duchi, Elad Hazan, Yoram Singer:
Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research 12: 2121-2159 (2011)

Matthew D. Zeiler:
ADADELTA: An Adaptive Learning Rate Method. CoRR abs/1212.5701

Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." ICLR 2015

Bibliography

Sergey Ioffe, Christian Szegedy:
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.
ICML2015: 448-456

Sergey Ioffe, Batch Renormalization. NIPS 2017

Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, Victor S. Lempitsky:
Texture Networks: Feed-forward Synthesis of Textures and Stylized Images. ICML 2016: 1349-1357

Lei Jimmy Ba, Jamie Ryan Kiros, Geoffrey E. Hinton:
Layer Normalization. CoRR abs/1607.06450 (2016)

Yuxin Wu, Kaiming He: Group Normalization. ECCV (13) 2018: 3-19

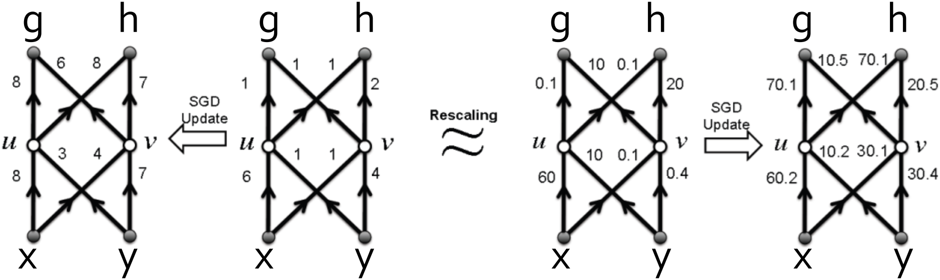
Tim Salimans, Diederik P. Kingma:
Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks.
NIPS 2016: 901

Xavier Glorot, Yoshua Bengio:
Understanding the difficulty of training deep feedforward neural networks. AISTATS 2010: 249-256

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun:
Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. ICCV 2015: 1026-1034

Initialization schemes

One more toy example: 1 SGD step for (x,y = 1,1) and L = g+h



[Neyshabur, Salakhutdinov, Srebro, Path-SGD: Path-Normalized Optimization in Deep Neural Networks, NIPS2015]

Units of measurements

- Let our coordinates be measured in meters. What is the unit of measurement for gradients? Assume unitless function…
- (Stochastic) gradient descent is inconsistent.
- Newton method is consistent.

Adadelta method [Zeiler 2012]

$$g[t] = \mu\, g[t-1] + (1-\mu)\, \nabla(E, w[t]) \odot \nabla(E, w[t])$$

$$w[t+1] = w[t] - \frac{\sqrt{d[t] + \epsilon}}{\sqrt{g[t] + \epsilon}} \odot \nabla(E, w[t])$$

$$d[t+1] = \mu\, d[t] + (1-\mu)\, (w[t+1]-w[t]) \odot (w[t+1]-w[t])$$

- No step length parameter (good!)
- Correct units within the updates