

Lecture 2: Optimization for Deep Learning

Supervised learning

$$\{x_1, x_2, x_3, \dots, x_M\} \subset \mathbb{R}^N$$

$$\{y_1, y_2, y_3, \dots, y_M\} \subset \mathcal{L}$$

$$f: \mathbb{R}^N \rightarrow \mathcal{L}$$

“Textbook” instance: binary linear classifier

$$\mathcal{L} = \{-1, 1\}$$

$$f(x) = \text{sgn } w^\top x$$

In practice, f would almost always be from some parameterized space

Minimizing empirical error

Quantity we may care about:

$$E(f) = \int [f(x) \neq y(x)] dx$$
$$x \sim P(x)$$

In practice, we can assess only *empirical* estimate:

$$E(f) = \sum_{i=1}^N [f(x_i) \neq y_i]$$

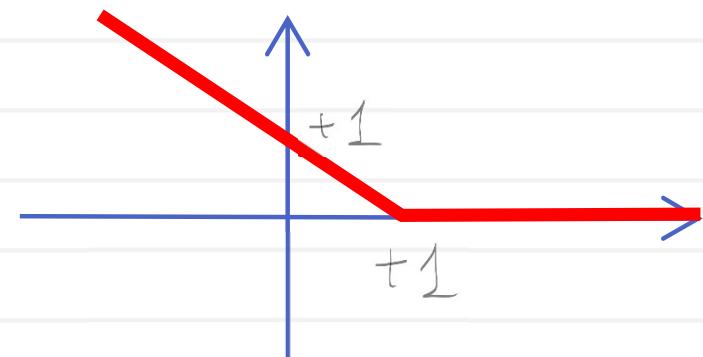
E.g. for binary linear classifier:

$$E(\omega) = \sum_{i=1}^N [\text{sgn } y_i \cdot \omega^\top x_i = -1]$$

Classification problem: hinge loss

Zero-one loss is “non-optimizable”.

Most popular relaxation:



$$E(\omega) = \sum_{i=1}^N [\operatorname{sgn} y_i \omega^\top x_i = -1]$$



$$E(\omega) = \sum_{i=1}^N \max(0, 1 - y_i \omega^\top x_i)$$

$$\frac{d E}{d \omega} = \sum_{i=1}^N [y_i \omega^\top x_i < 1] y_i x_i$$

Classification problem: logistic loss

Logistic function maps R to [0;1]: $\sigma(t) = \frac{1}{1+e^{-t}}$

Can treat logistic as probability:

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

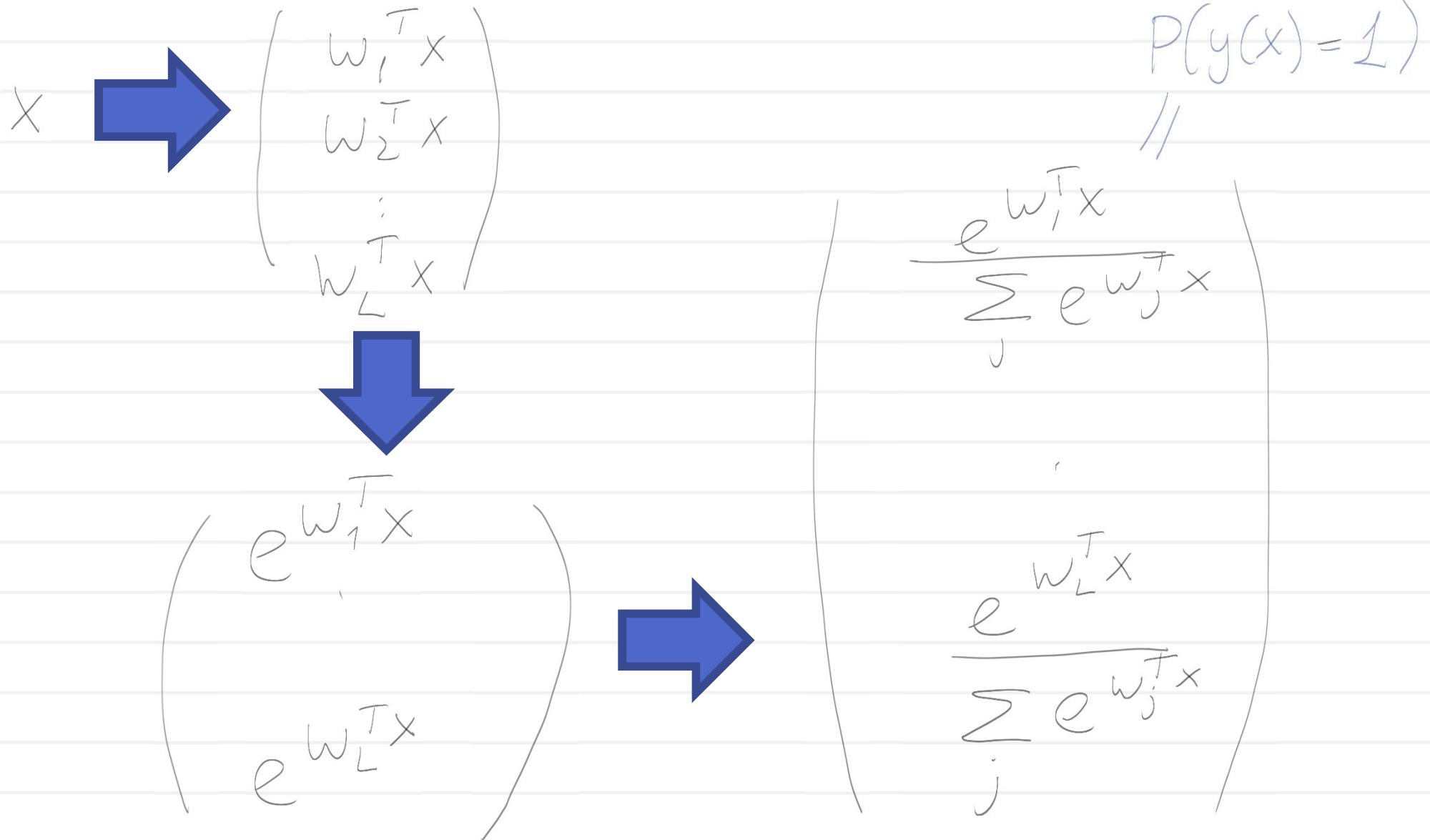
Define loss as -log likelihood (*ML estimation*):

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

$$\frac{dE}{d\omega} = \sum_{i=1}^N (\sigma(y_i \omega^T x_i) - 1) y_i x_i$$

Multinomial logistic regression

Softmax (generalizes logistic):



Multinomial logistic regression

Multinomial log loss (generalizes logistic loss):

$$E(\omega) = - \sum_{i=1}^N \log P(y(x)=y_i | \omega) =$$

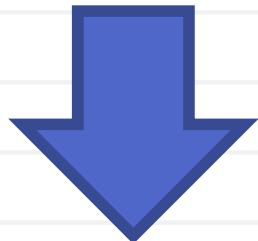
$$- \sum_{i=1}^N w_{y_i}^T x_i + \log \sum_{j=1}^L e^{w_j^T x_i}$$

(Part of the) gradient over w_j :

$$\frac{dE}{dw_j} = - \sum_{i=1}^N x_i ([y_i = j] - P(y(x_i) = j | \omega))$$

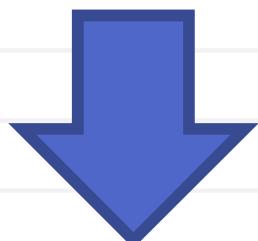
Supervised learning approach

Problem + Loss



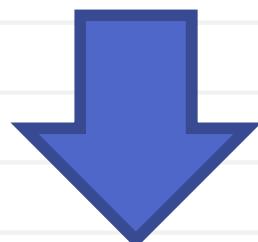
Approximation error

Parameterized predictor $p(w)$



Estimation error

$E(w) = (\text{loss on the training set}) + \text{regularizer}$



Optimization error

Result

Approximation error

Bayes-optimal classifier/predictor over all predictors:

$$f_0 = \arg \min_{f \in \mathcal{F}_0} \sum \ell(x, y, f)$$
$$(x, y) \sim P(x, y)$$

The class of predictors is usually restricted (e.g. to some parametric family):

$$\tilde{f} = \arg \min_{f \in \mathcal{F}} \sum \ell(x, y, f)$$
$$(x, y) \sim P(x, y)$$

$$\|f_0 - \tilde{f}\| = \text{approximation error}$$

Estimation error

$$\tilde{f} = \arg \min_{f \in \mathcal{F}} \int l(x, y, f) \\ (x, y) \sim P(x, y)$$

Cannot integrate, have a finite sample of data:

$$\hat{f} = \arg \min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^N l(x_i, y_i, f) + \lambda R(f)$$

$$\|\tilde{f} - \hat{f}\| = \text{estimation error}$$

Optimization error

$$\hat{f} = \arg \min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^N l(x_i, y_i, f) + \lambda R(f)$$

Have a limited amount of time to optimize:

$$\hat{f}^{(t)} = \arg \min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^N l(x_i, y_i, f) + \lambda R(f)$$

$$\|\hat{f} - \hat{f}^{(t)}\| = \text{optimization error}$$

Optimization and supervised ML

$$f_0 = \arg \min_{f \in \mathcal{F}_0} \sum \ell(x, y, f)$$

$f_0(x, y) \sim P(x, y)$

Approximation error

$$\tilde{f} = \arg \min_{f \in \mathcal{F}} \sum \ell(x, y, f)$$

$\tilde{f}(x, y) \sim P(x, y)$

Estimation error

$$\hat{f} = \arg \min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^N \ell(x_i, y_i, f) + \lambda R(f)$$

Optimization error

$$\hat{f}^{(t)} = \arg \min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^N \ell(x_i, y_i, f) + \lambda R(f)$$

[Bottou & Bousquet NIPS 2007]

Overall error

$$\|f_0 - \hat{f}^{(\ell)}\| \leq$$

total error

$$\|f_0 - \tilde{f}\| +$$

approximation error

$$\|\tilde{f} - \hat{f}^{(\ell)}\| +$$

estimation error

$$\|\hat{f} - \hat{f}^{(\ell)}\|$$

optimization error

Various design
choices trade-off
different errors e.g.:

- Parameterization
- Time for data vs.
optimization

[Bottou & Bousquet NIPS 2007]

Small-scale learning vs Large-scale learning

$$\|f_0 - \hat{f}^{(\ell)}\| \leq$$

Small-scale learning:

we are data-bound,

estimation error

dominates

Large-scale learning:

we are time-bound,

optimization error is a

bigger issue

$$\|f_0 - \tilde{f}\| +$$

$$\|\tilde{f} - \hat{f}\| +$$

estimation error

$$\|\hat{f} - \hat{f}^{(\ell)}\|$$

optimization error

[Bottou & Bousquet NIPS 2007]

Small scale setting: traditional optimization

$$\hat{f} = \arg \min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^N l(x_i, y_i, f) + \lambda R(f)$$

- 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,...)

Example: from SVM to quadratic program

$$w = \arg \min_w \frac{1}{N} \sum_{i=1}^N h(x_i, y_i, w) + \frac{\lambda}{2} \|w\|_2^2$$

Hinge loss is often implemented using
“slack variables”:

$$\begin{aligned} & \min_{w, \gamma} \frac{\lambda}{2} \|w\|^2 + \frac{1}{N} \sum_{i=1}^N \gamma_i \\ \text{s.t. } & \gamma_i \geq 1 - y_i w^\top x_i \\ & \gamma_i \geq 0 \end{aligned}$$

Large-scale learning

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

$$\frac{dE}{d\omega} = \frac{1}{N} \sum_{i=1}^N \frac{d\ell(x_i, y_i, \omega)}{d\omega} + \lambda \frac{dR}{d\omega}$$

- 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:

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

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

Stochastic gradient is an unbiased estimate of gradient

Stochastic gradient descent (SGD)

SGD:

$$v[t] = -\alpha[t] \nabla(E, w[t])$$

$$w[t+1] = w[t] + v[t]$$

where

$$\nabla(E, w[t]) = \frac{d E^{i(t)}}{d w} \Big|_{w[t]}$$

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

Stochastic gradient descent (SGD)

SGD:

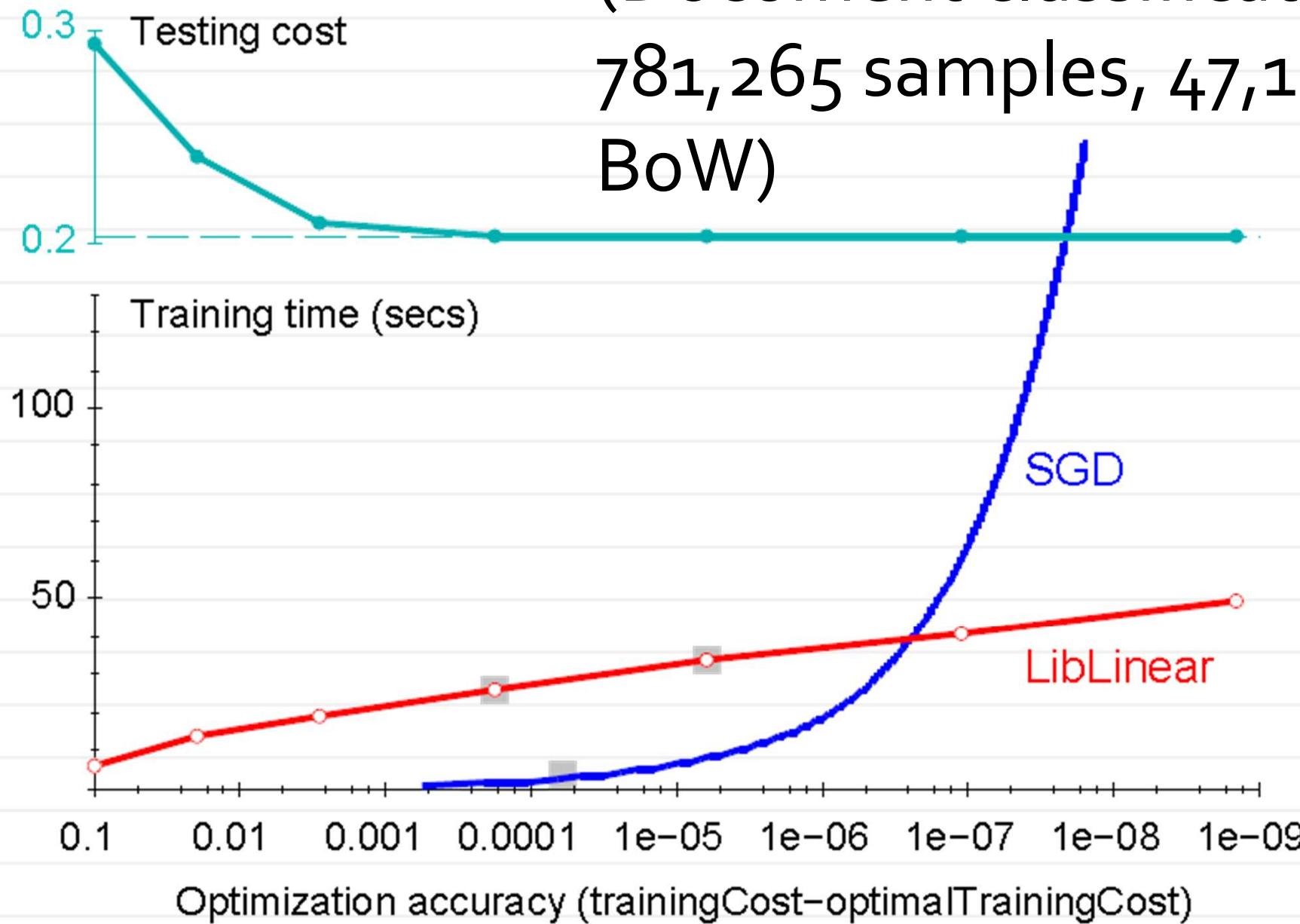
$$v[t] = -\alpha[t] \nabla(E, w[t])$$
$$w[t+1] = w[t] + v[t]$$

- 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

slide credit: 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):

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

Batch stochastic gradient:

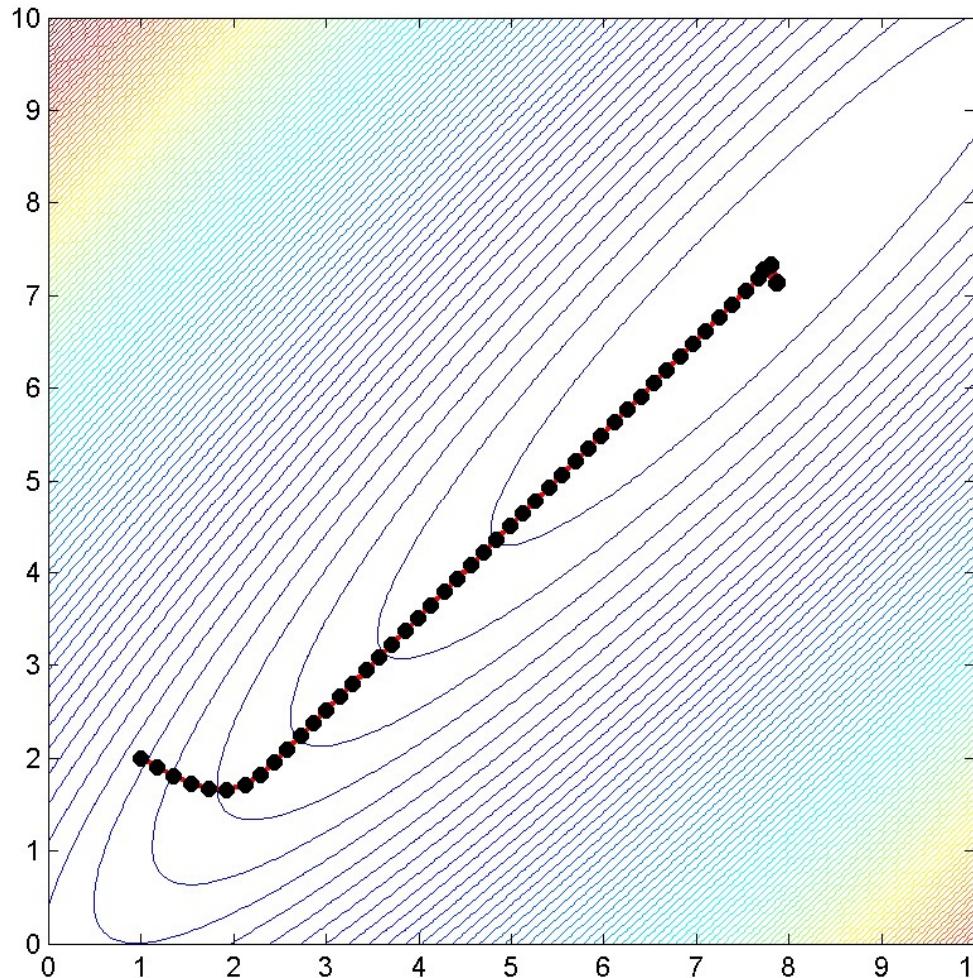
$$\frac{dE}{dw} = \frac{1}{N_b} \sum_{i=1}^{N_b} \frac{dl(x_{b(i)}, y_{b(i)}; w)}{dw} + \lambda \frac{dR}{dw}$$

Why batching?

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

- “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

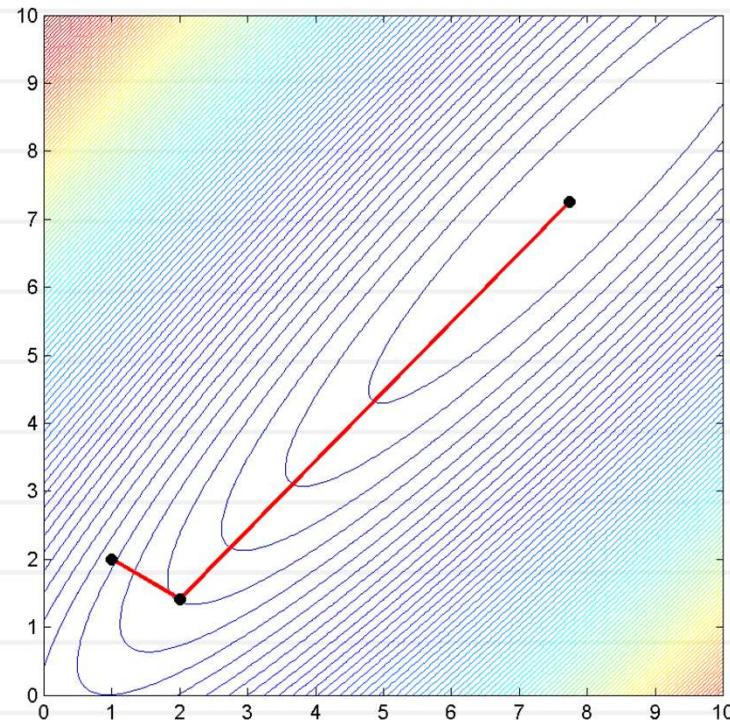
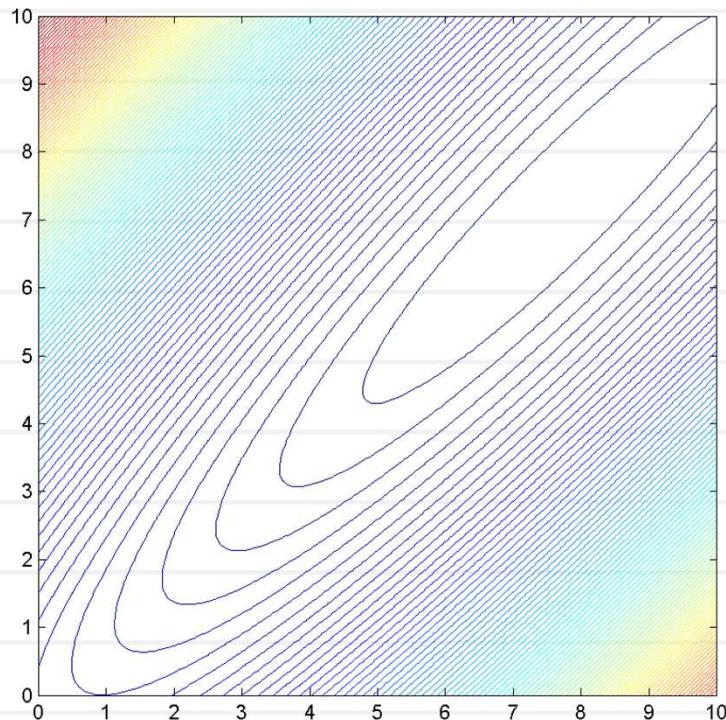
SGD inherits gradient descent problems



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

Better optimization methods

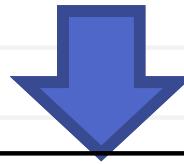
- Second order methods (Newton, Quasi-Newton)
- Krylov subspace methods, in particular *conjugate gradients*



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*):

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



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

Typical $\mu = 0.9$

Exponentially decaying running average

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

$$w[t+1] = w[t] + v[t+1]$$

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

$$= \mu^2 v[t-1] - \mu \alpha[t-1] \nabla (E, w[t-1])$$

$$- \alpha[t] \nabla (f, w[t]) =$$

$$= \mu^3 v[t-2] - \mu^2 \alpha[t-2] \nabla (E, w[t-2])$$

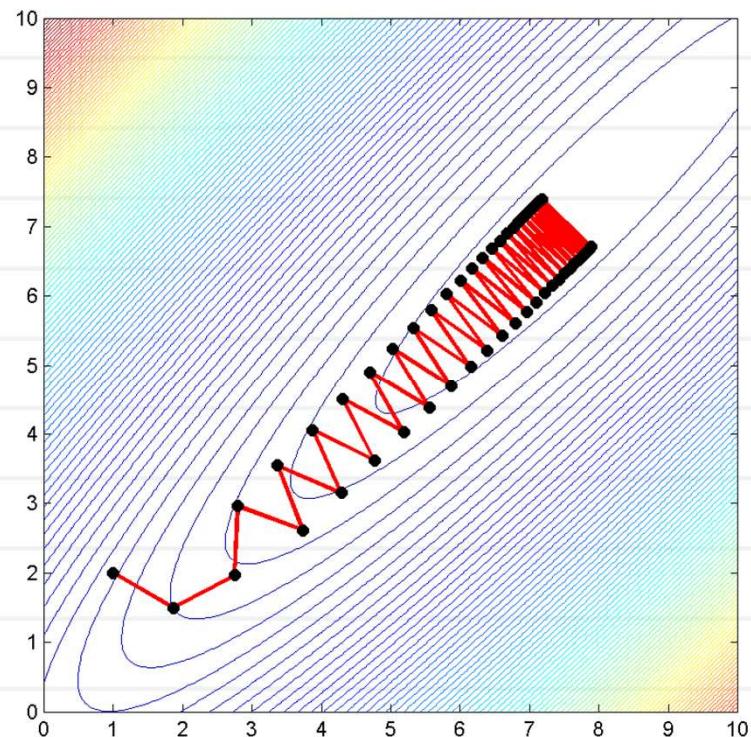
$$- \mu \alpha[t-1] \nabla (E, w[t-1]) - \alpha[t] \nabla (E, w[t]) =$$

$$= \mu^{k+1} v[t-k] + \sum_{i=0}^k \color{red}{\mu^i} \times [t-i] \nabla (E, w[t-i])$$

Momentum: why it works

$$v[t+1] = \sum_{i=0}^k m^i \alpha[t-i] \nabla(E, w[t-i])$$

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



Nesterov accelerated gradient

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

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

Let us use this knowledge:

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

(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

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

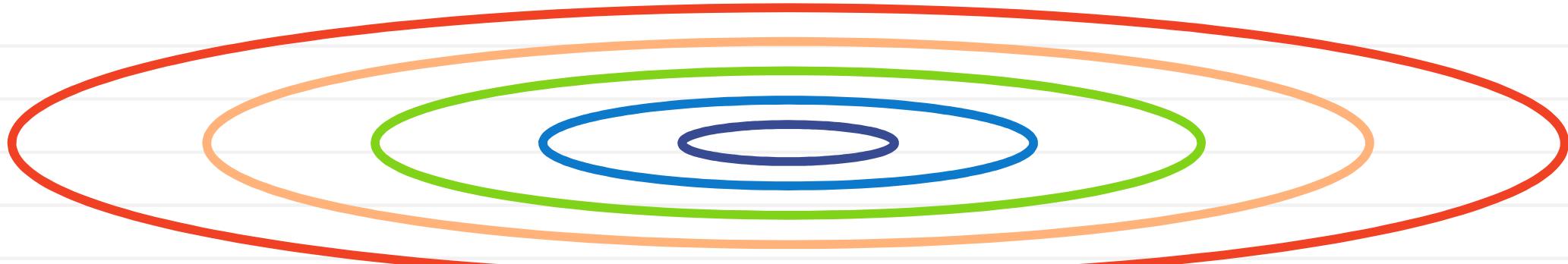
$$w[t+1] = w[t] - \frac{\alpha}{\sqrt{g[t]^2 + \epsilon}} \odot \nabla(E, w[t])$$

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

Adagrad method [Duchi et al. 2011]

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

$$w[t+1] = w[t] - \frac{\alpha}{\sqrt{g[t]^2 + \epsilon}} \odot \nabla(E, w[t])$$



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:

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

$$w[t+1] = w[t] - \frac{\alpha[t]}{\sqrt{g[t]^2 + \epsilon}} \odot \nabla(E, w[t])$$

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]

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

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

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

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

Comparison: logistic regression

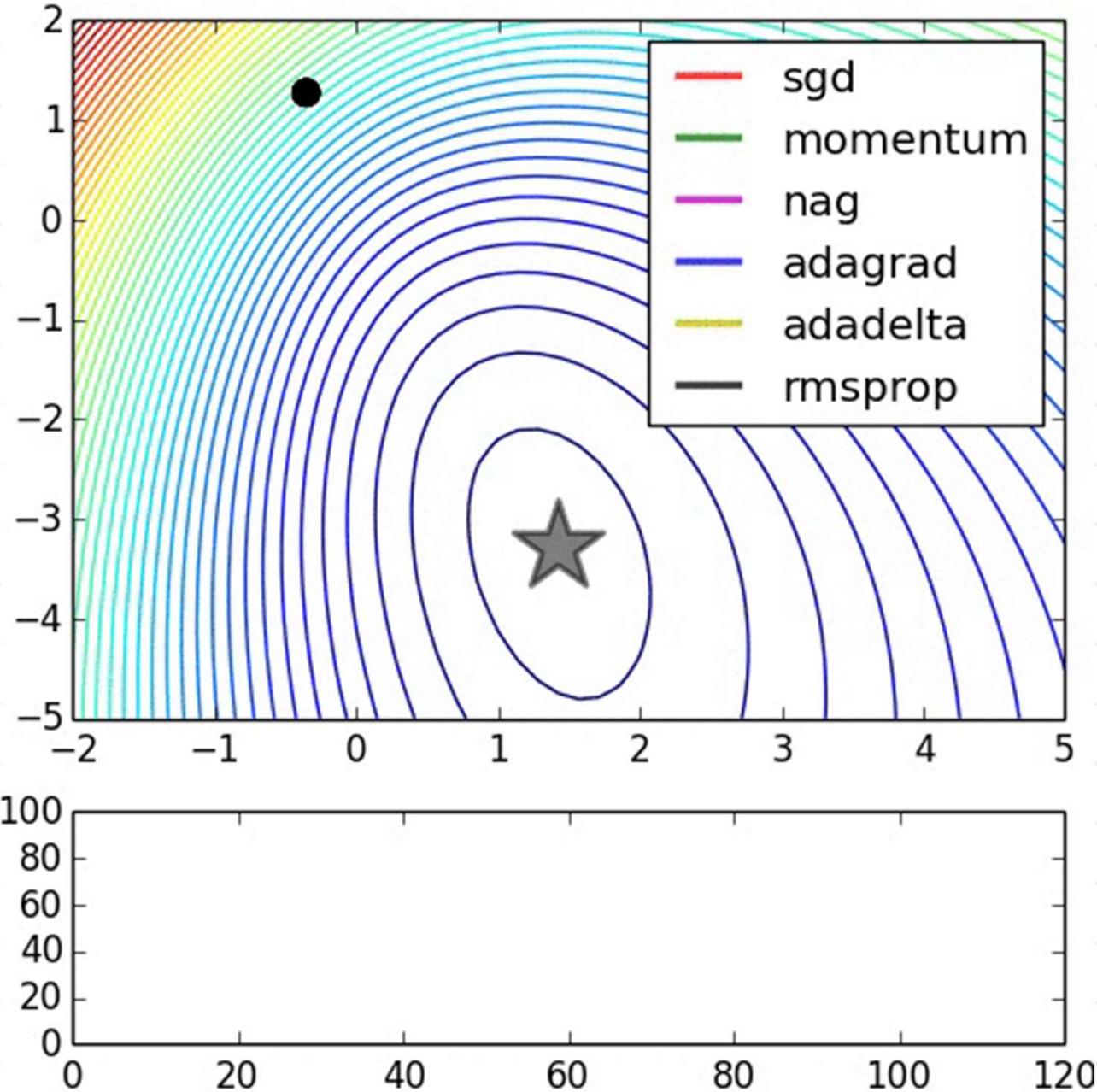


Image credit: Alec Redford

Further comparison

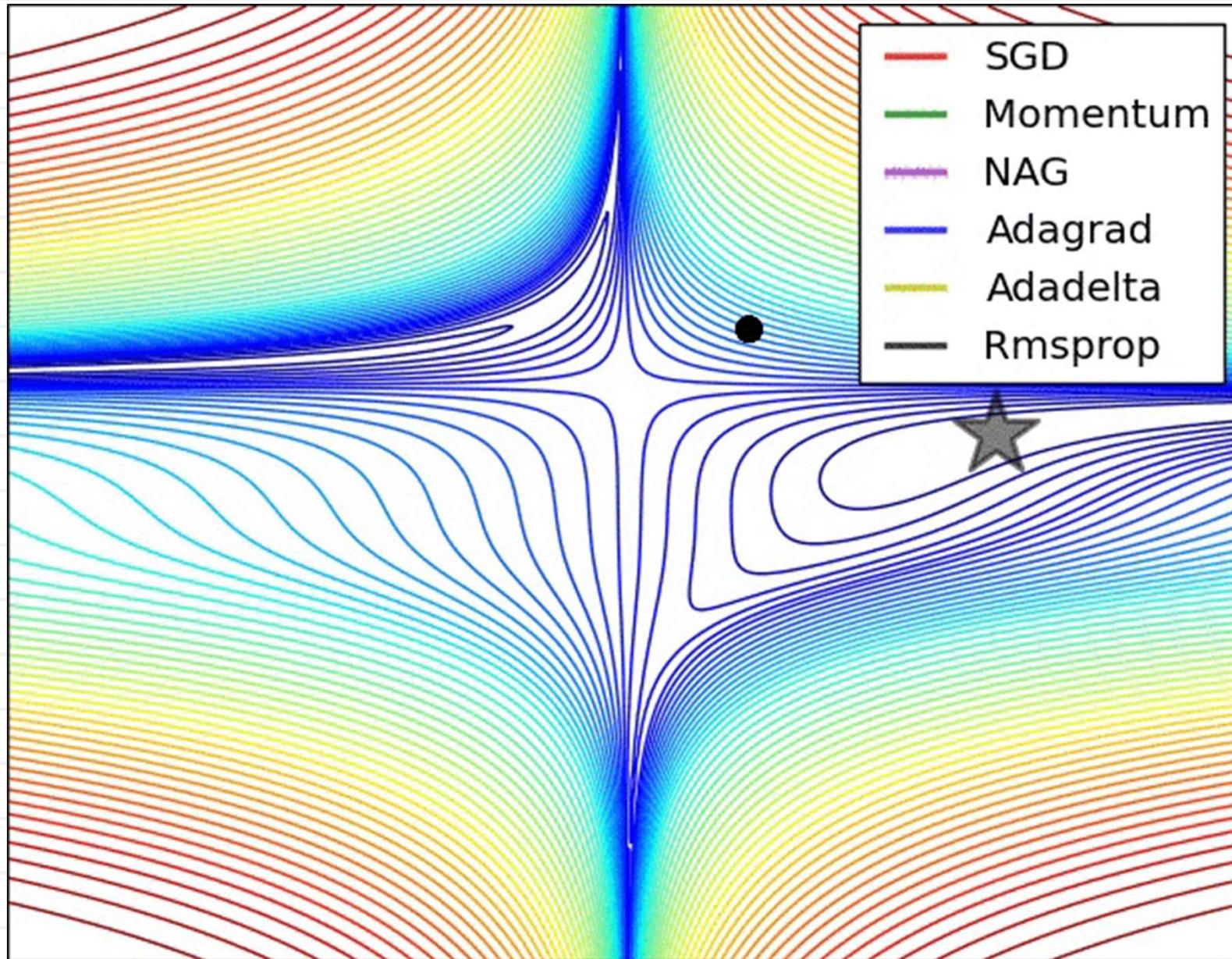


Image credit: Alec Redford

“Deep Learning”, Spring 2018: Lecture 2, “Optimization for DL”

Further comparison: escaping from a saddle

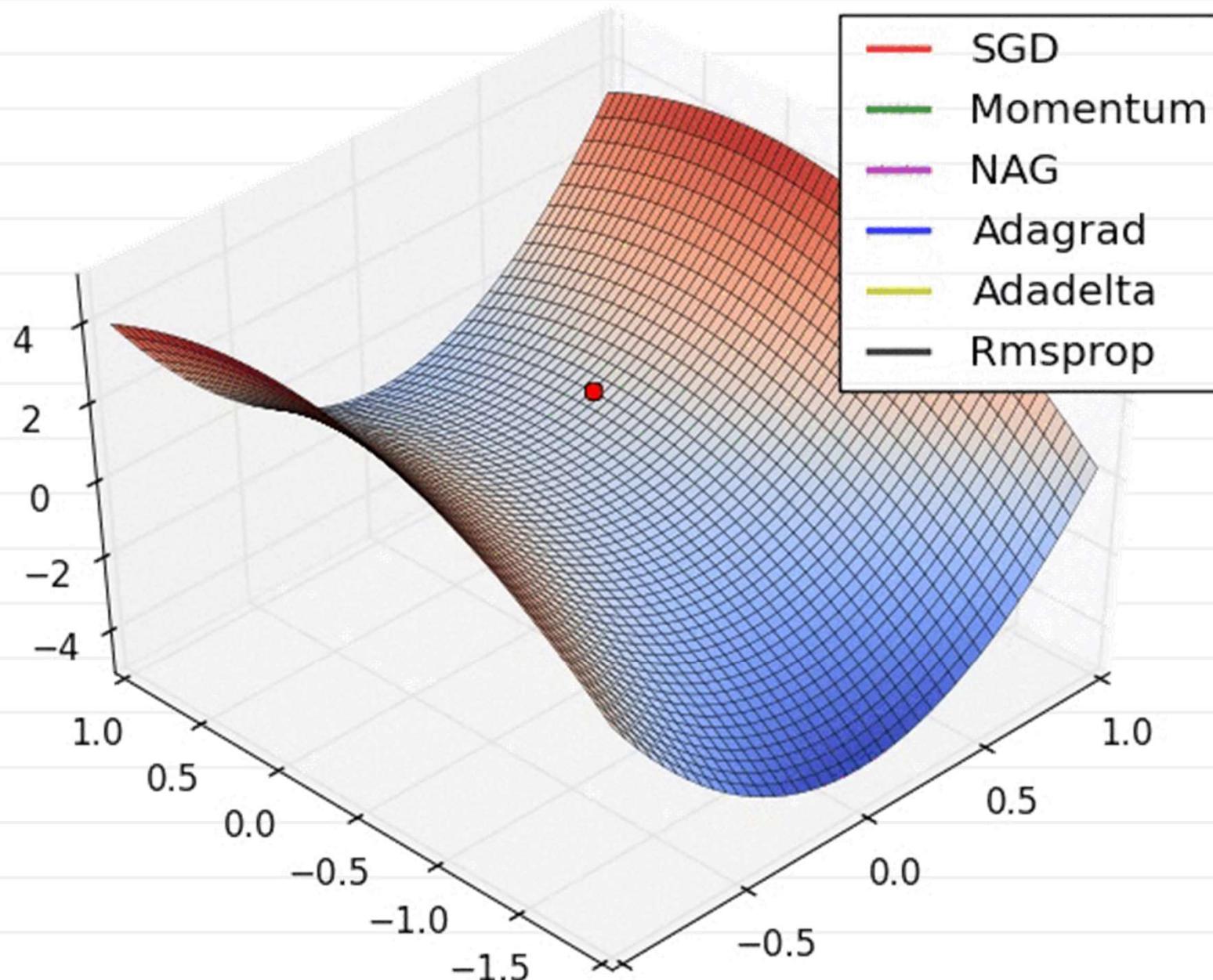


Image credit: Alec Redford

ADAM method [Kingma & Ba 2015]

ADAM = “ADAptive Moment Estimation”

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

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

$$w[t+1] = w[t] - \alpha \frac{\nabla(\mathcal{E}, w[t])}{\sqrt{g[t+1]^2 + \epsilon}} \odot v[t+1]$$

$\frac{1}{\sqrt{g[t+1]^2 + \epsilon}}$

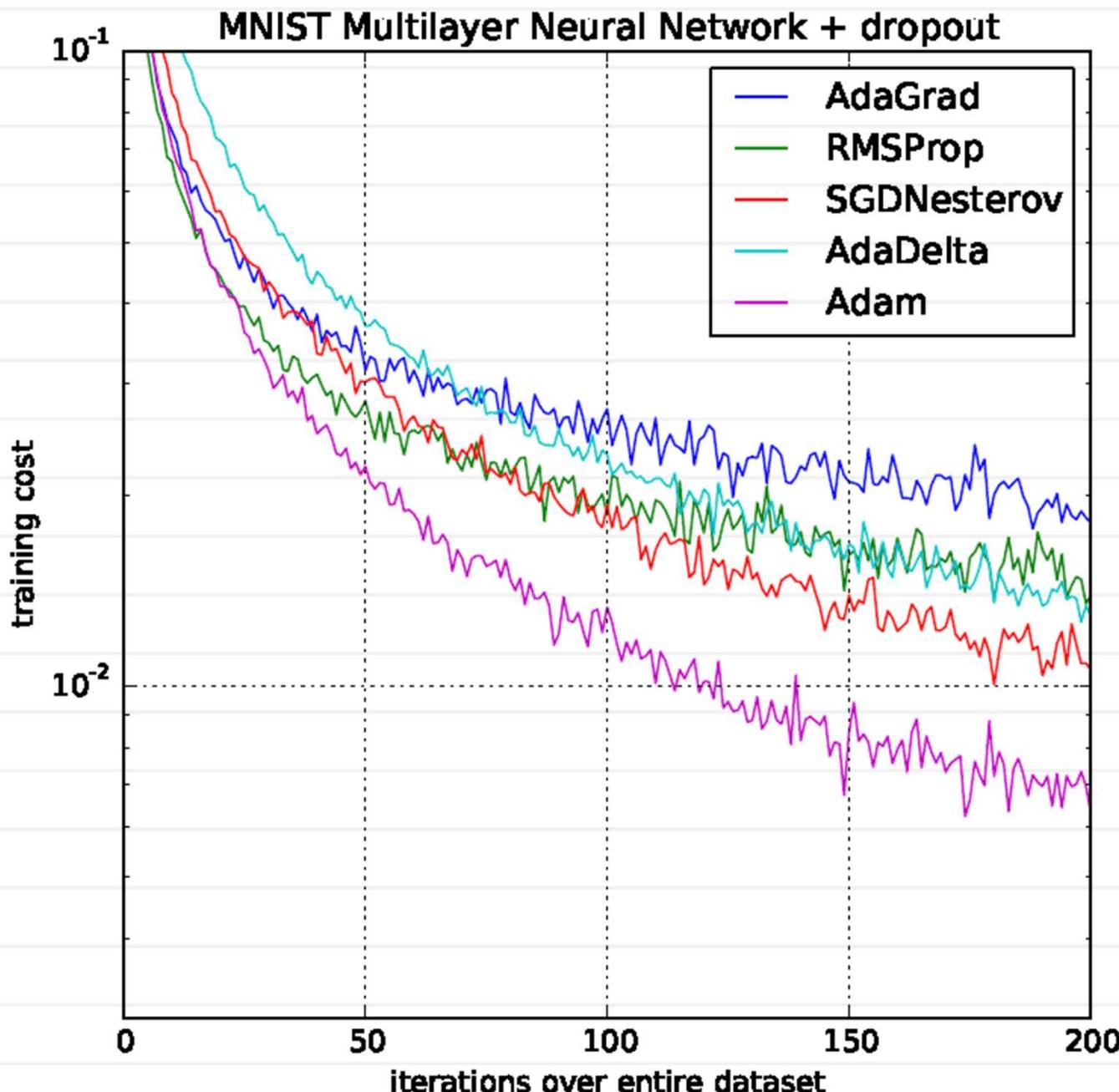
$\odot v[t+1]$

$1 - \beta^t$

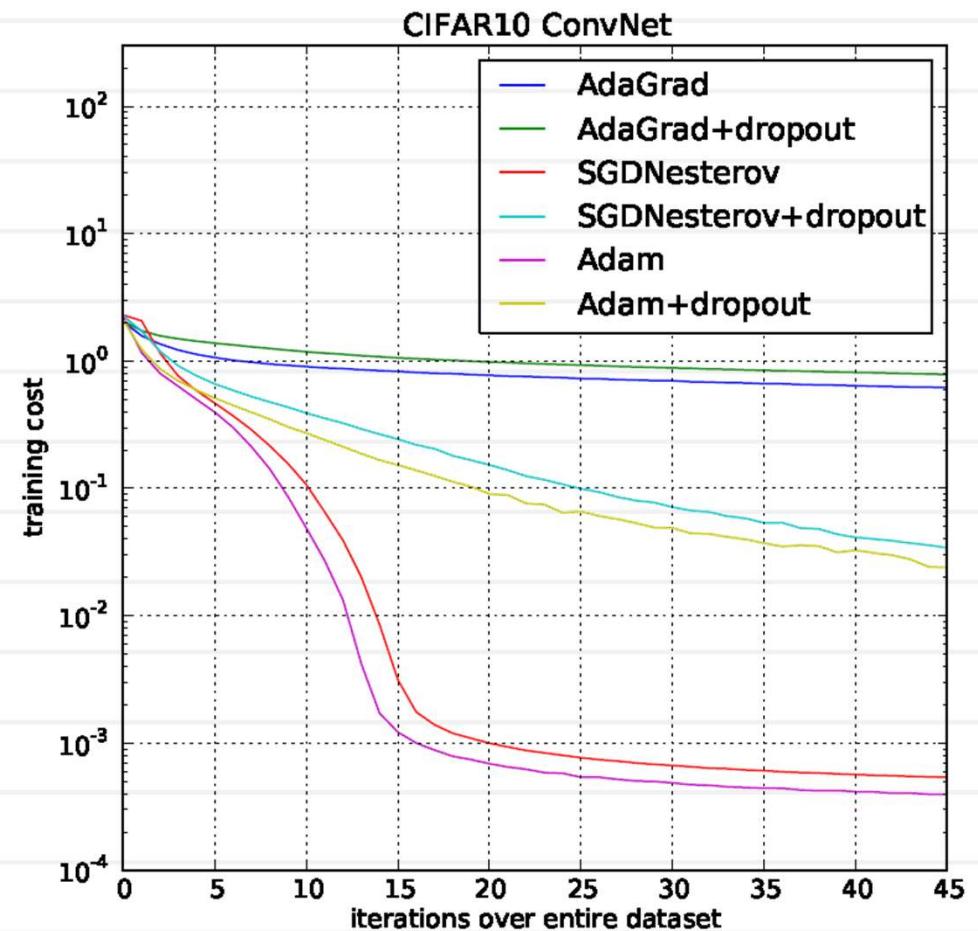
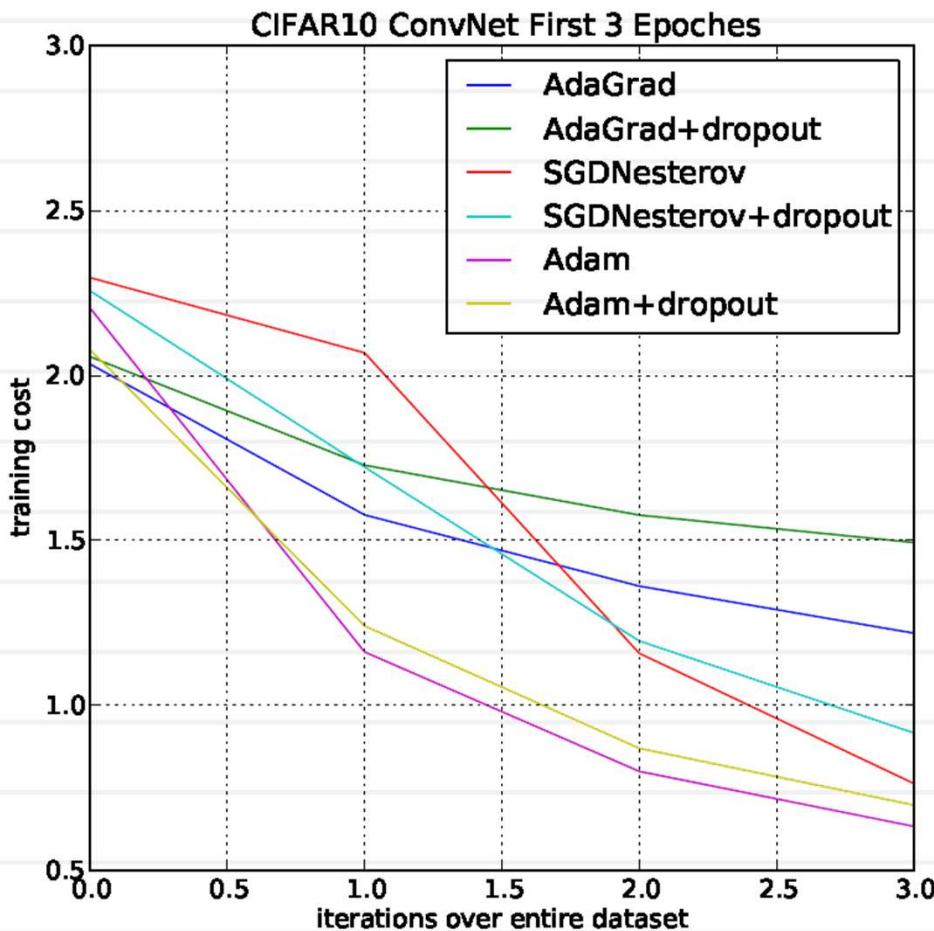
$1 - \mu^t$

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

ADAM method [Kingma & Ba 2015]



ADAM method [Kingma & Ba 2015]

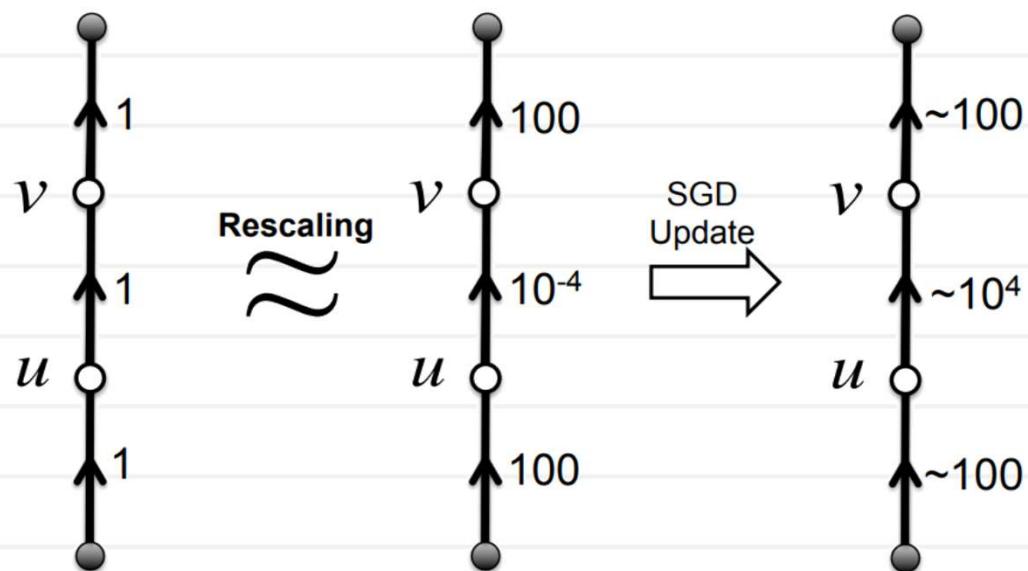


Gauge freedom, SGD, and explosions

“Gradient explosions” are a nasty phenomenon during SGD training of deep nets

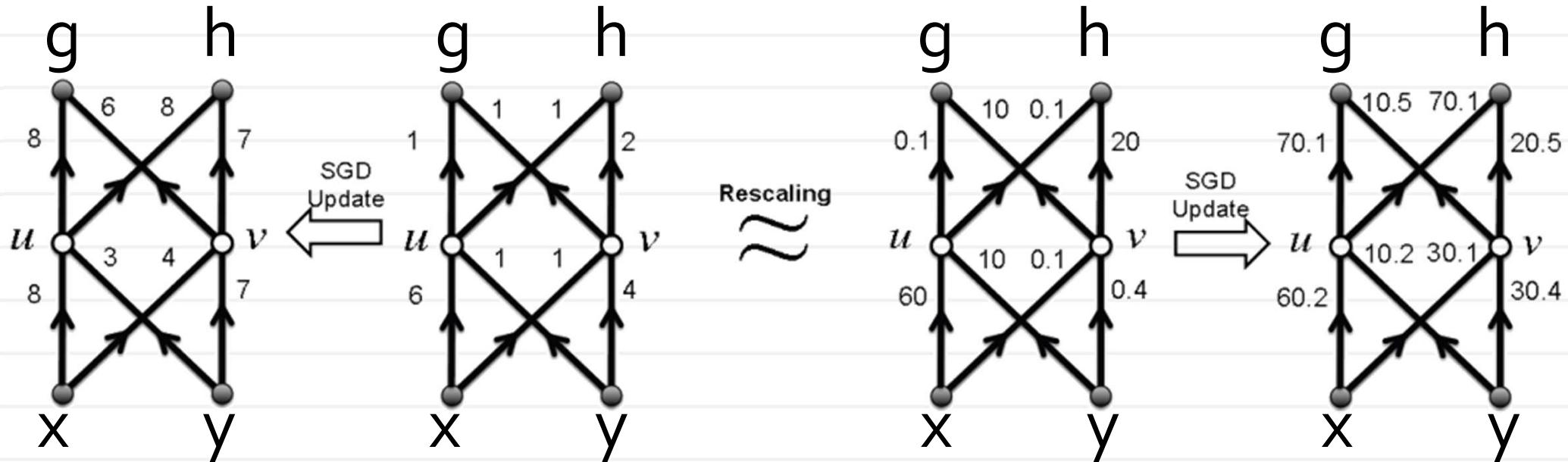


После этого взрыва у всех в радиусе
50 км
испортилось настроение



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

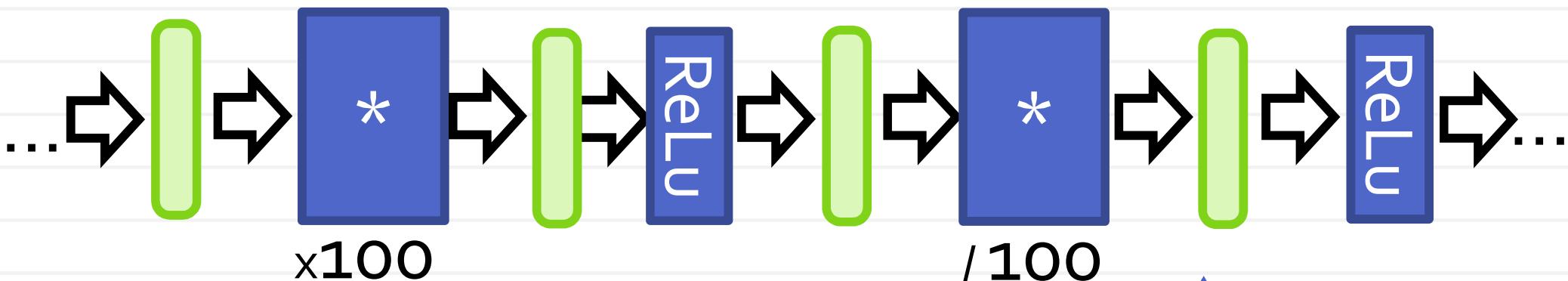
Gauge freedom, SGD, and explosions



1 SGD step for $(x, y = 1, 1)$ and $L = g + h$

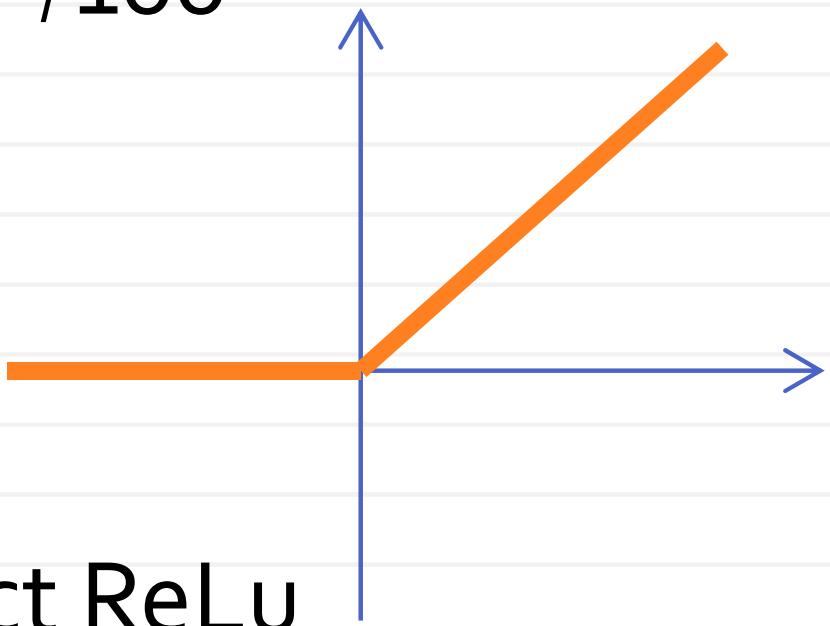
[Neyshabur, Salakhutdinov, Srebro, Path-SGD: Path-Normalized Optimization in Deep Neural Networks, NIPS2015]

Gauge freedom in ReLu Networks



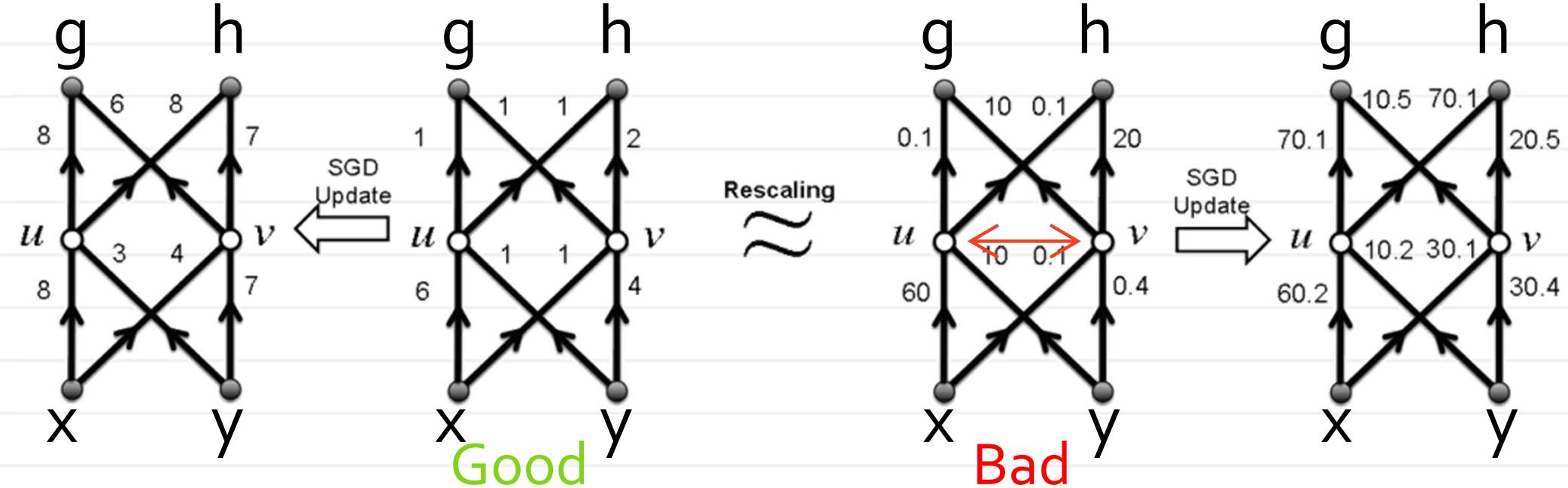
$$\alpha > 0$$

$$1/\alpha \operatorname{ReLU}(\alpha x) = \operatorname{ReLU}(x)$$



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

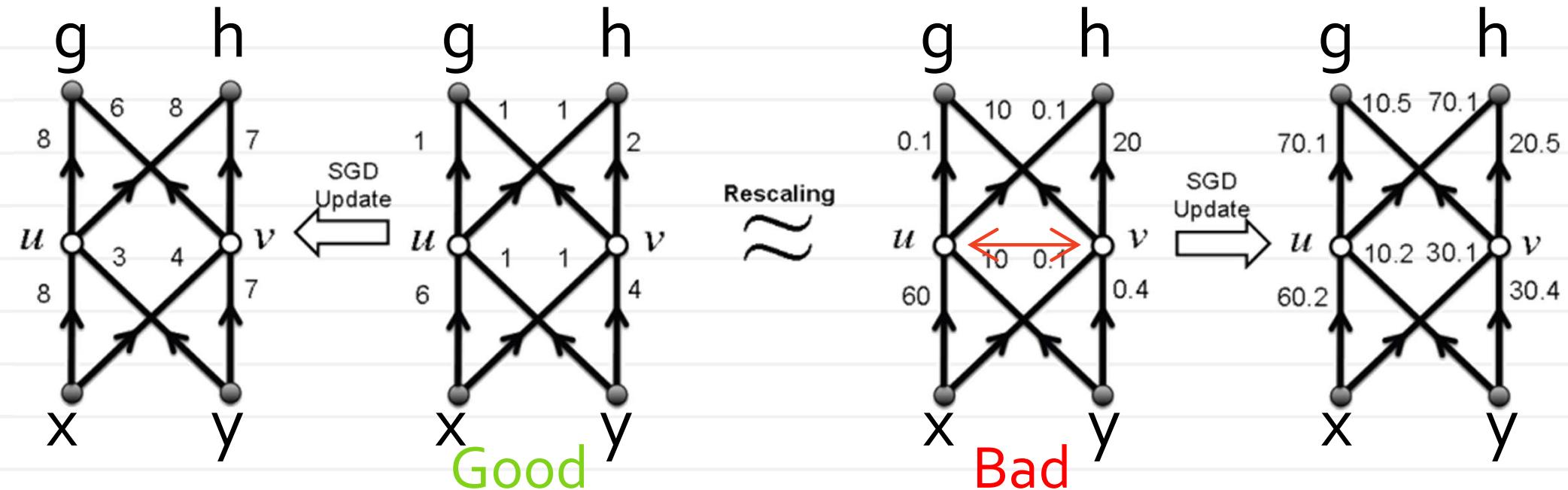
Initialization schemes



- **Basic idea 1:** units should be initialized to have comparable total input weights
- E.g. [Glorot&Bengio 2010] aka “Xavier-initialization”:
$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$
- E.g. [He et al, Arxiv15] for ReLu networks:

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

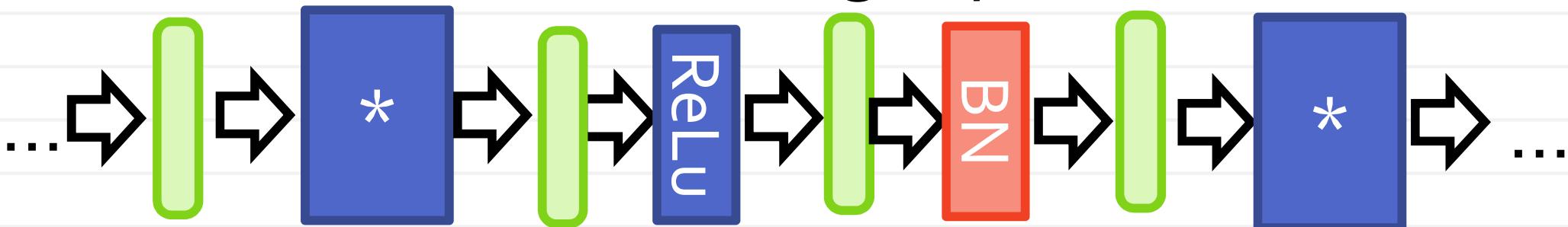
Initialization schemes



- **Basic idea 1:** units should be initialized to have comparable total input weights
- **Basic idea 2:** use layers which keep magnitude

Batch normalization

[Szegedy and Ioffe 2015]



- Makes the training process invariant to some re-parameterizations
- Use mini-batch statistics at training time
- Use population statistics at test time
- At test time can be “incorporated” into adjacent multiplicative layer

Batch normalization layer

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
Parameters to be learned: γ, β

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

covariant to reparameterization

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

learnable by SGD

[Szegedy and Ioffe 2015]

Batch re-normalization

Input: Values of x over a training mini-batch $\mathcal{B} = \{x_1 \dots m\}$; parameters γ, β ; current moving mean μ and standard deviation σ ; moving average update rate α ; maximum allowed correction r_{\max} , d_{\max} .

Output: $\{y_i = \text{BatchRenorm}(x_i)\}$; updated μ, σ .

$$\begin{aligned}\mu_{\mathcal{B}} &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_{\mathcal{B}} &\leftarrow \sqrt{\epsilon + \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2} \\ r &\leftarrow \text{stop_gradient} \left(\text{clip}_{[1/r_{\max}, r_{\max}]} \left(\frac{\sigma_{\mathcal{B}}}{\sigma} \right) \right) \\ d &\leftarrow \text{stop_gradient} \left(\text{clip}_{[-d_{\max}, d_{\max}]} \left(\frac{\mu_{\mathcal{B}} - \mu}{\sigma} \right) \right) \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} \cdot r + d \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \\ \\ \mu &:= \mu + \alpha(\mu_{\mathcal{B}} - \mu) \quad // \text{Update moving averages} \\ \sigma &:= \sigma + \alpha(\sigma_{\mathcal{B}} - \sigma)\end{aligned}$$

[Ioffe NIPS2017]

“All you need is a good init”

Algorithm 1 Layer-sequential unit-variance orthogonal initialization. L – convolution or full-connected layer, W_L - its weights, B_L - its output blob., Tol_{var} - variance tolerance, T_i – current trial, T_{max} – max number of trials.

Pre-initialize network with orthonormal matrices as in Saxe et al. (2014)

for each layer L **do**

while $|Var(B_L) - 1.0| \geq Tol_{var}$ and $(T_i < T_{max})$ **do**

 do Forward pass with a mini-batch

 calculate $Var(B_L)$

$W_L = W_L / \sqrt{Var(B_L)}$

end while

end for

- Initialize with orthonormal matrices (using SVD, take U or V matrices) – orthogonal columns/rows
- Re-normalize (as in batchnorm)
- Takes non-linearity into account

[Mishkin & Matas, ICLR16]

Recap

- Supervised learning: the most popular tool in machine learning
- Different types of error: Bayes, Approximation, Estimation, Optimization
- SGD optimization is used in large-scale setting
- Advanced SGD methods use running averages to smooth and rescale SGD steps
- Parameterizations (and reparameterizations) are very important

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 $O(1/k^2)$." Soviet Mathematics Doklady. Vol. 27. No. 2. 1983.

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. ICML 2015: 448-456

Dmytro Mishkin, Jiri Matas:
All you need is a good init. ICLR 2016

Sergey Ioffe, Batch Renormalization. NIPS 2017