Lecture 2: Optimization for Deep Learning

Supervised learning

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

$$\{y_1\ y_2\ y_3\ \ldots\ y_M\} \subset \mathcal{Y}$$

$$f: \mathbb{R}^N \longrightarrow \mathcal{Y}$$

"Textbook" instance: binary linear classifier

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

$$f(x) = \mathrm{sgn}\ w^T x$$

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

Minimizing empirical error

Quantity we may care about:

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

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

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

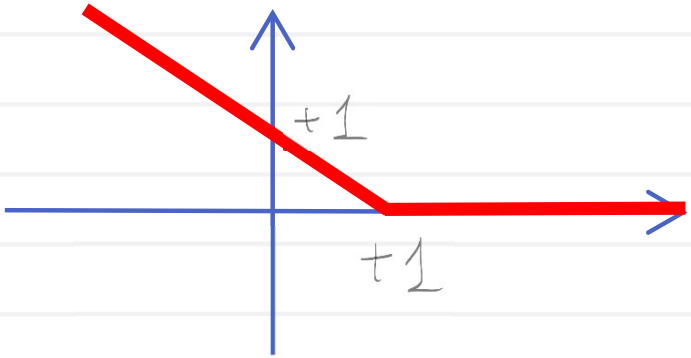
E.g. for binary linear classifier:

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

Classification problem: hinge loss

Zero-one loss is "non-optimizable".

Most popular relaxation:



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

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

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

Classification problem: logistic loss

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

Can treat logistic as probability:

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

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

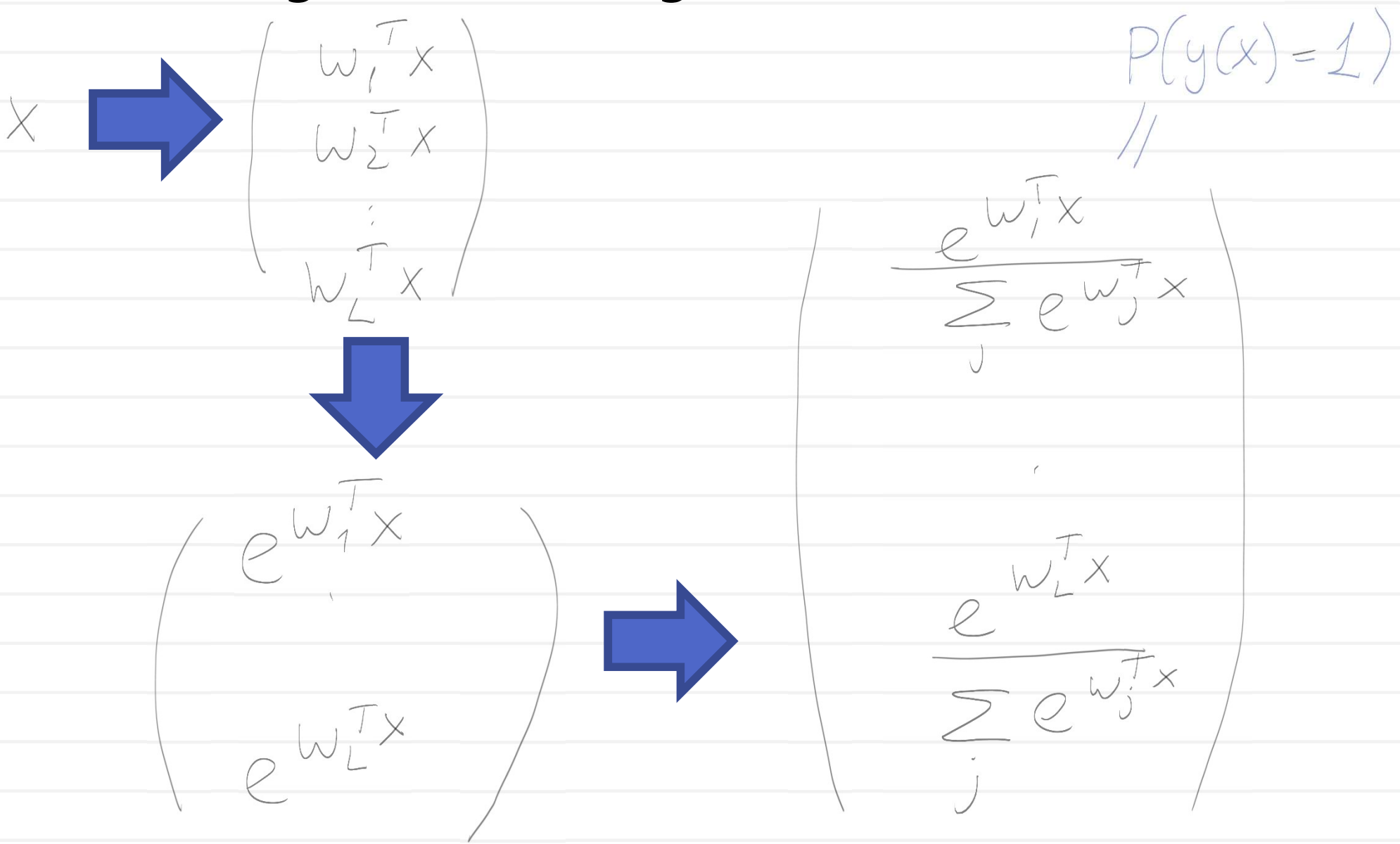
$$E(\omega) = -\sum_{i=1}^{N} \log P(y(x) = y_i \mid \omega) =$$

$$= \sum_{i=1}^{N} \log\left(1 + e^{-y_i \omega^T x_i}\right)$$

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

Multinomial logistic regression

Softmax (generalizes logistic):



Multinomial logistic regression

Multinomial log loss (generalizes logistic loss):

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

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

(Part of the) gradient over w_j:

$$\frac{dE}{dw_j} = -\sum_{i=1}^{N} x_i \left([y_i = j] - P(y(x_i)=j \mid w) \right)$$

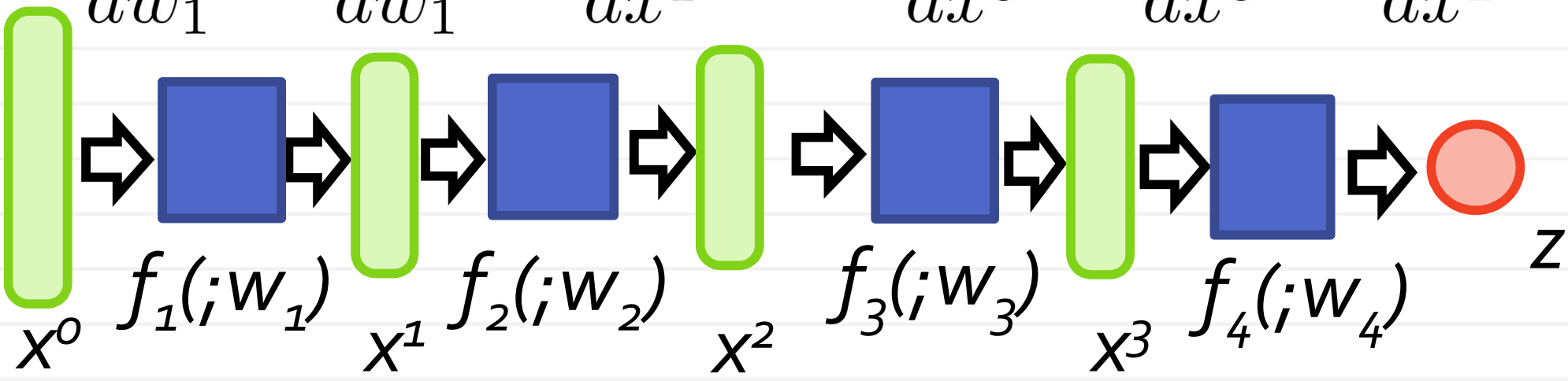
Sequential computation: *backpropagation*

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

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

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

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



Small scale setting: traditional optimization

$$\hat{\omega} = \arg\min_{\omega} \frac{1}{N} \sum_{i=1}^{N} l(x, y, \omega) + \lambda R(\omega)$$

- 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":

$$\min_{\omega, \xi} \frac{\lambda}{2} \|\omega\|^2 + \frac{1}{N} \sum_{i=1}^{N} \xi_i$$

$$\text{s.t.} \quad \xi_i \geq 1 \quad y_i \omega^T x_i$$

$$\xi_i \geq 0$$

Large-scale learning

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

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

- 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:

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

Stochastic gradient:

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

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

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 $\nabla(E, w[t]) = \left.\frac{dE^{c(t)}}{dw}\right|_{w[t]}$

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

Stochastic gradient descent (SGD)

SGD:

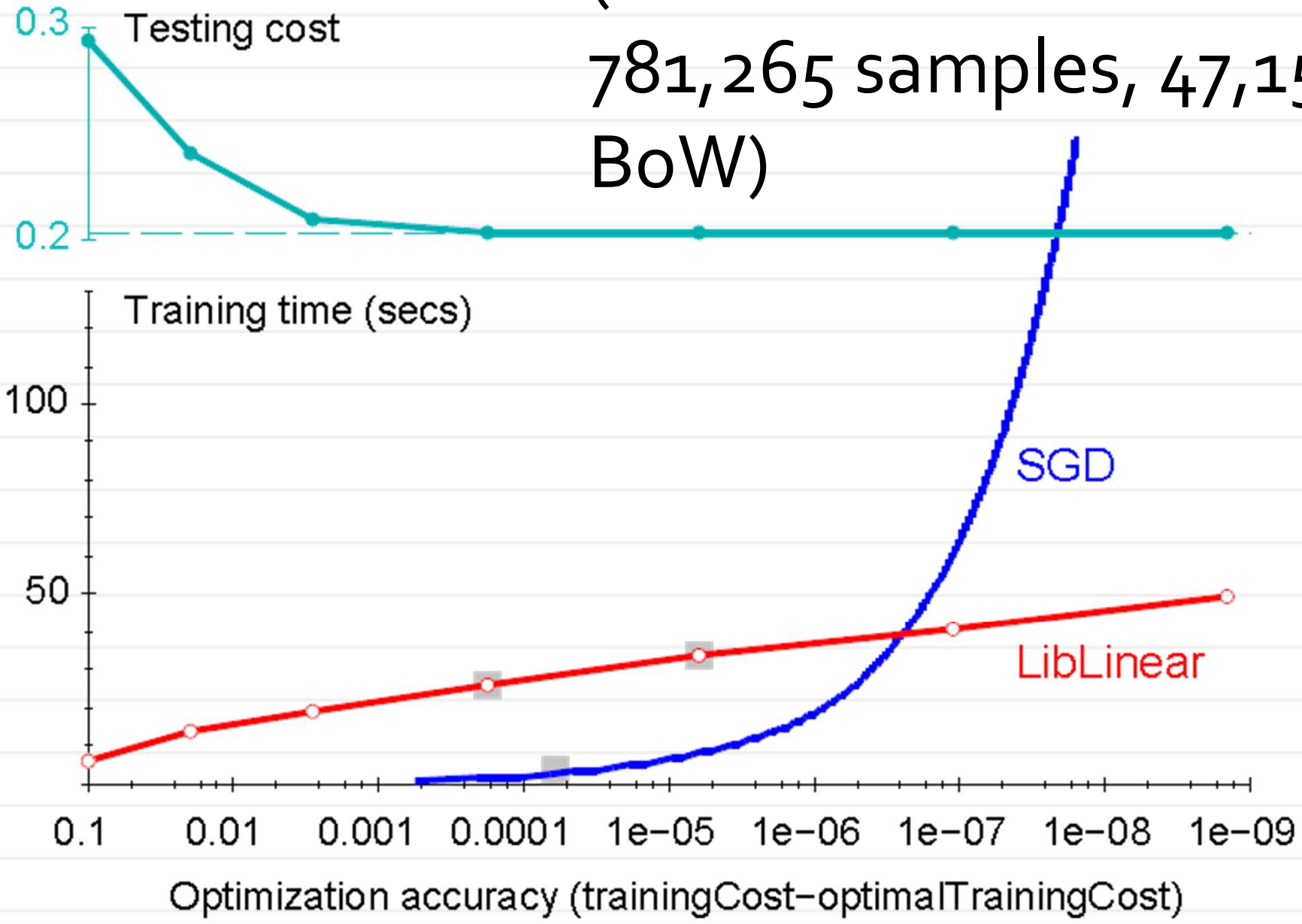
$$v[t] = -\alpha[t] \nabla(E, w[t])$$
$$w[t+1] = w[t] + v[t]$$

- 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:

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

Batch (aka mini-batch):

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

Batch stochastic gradient:

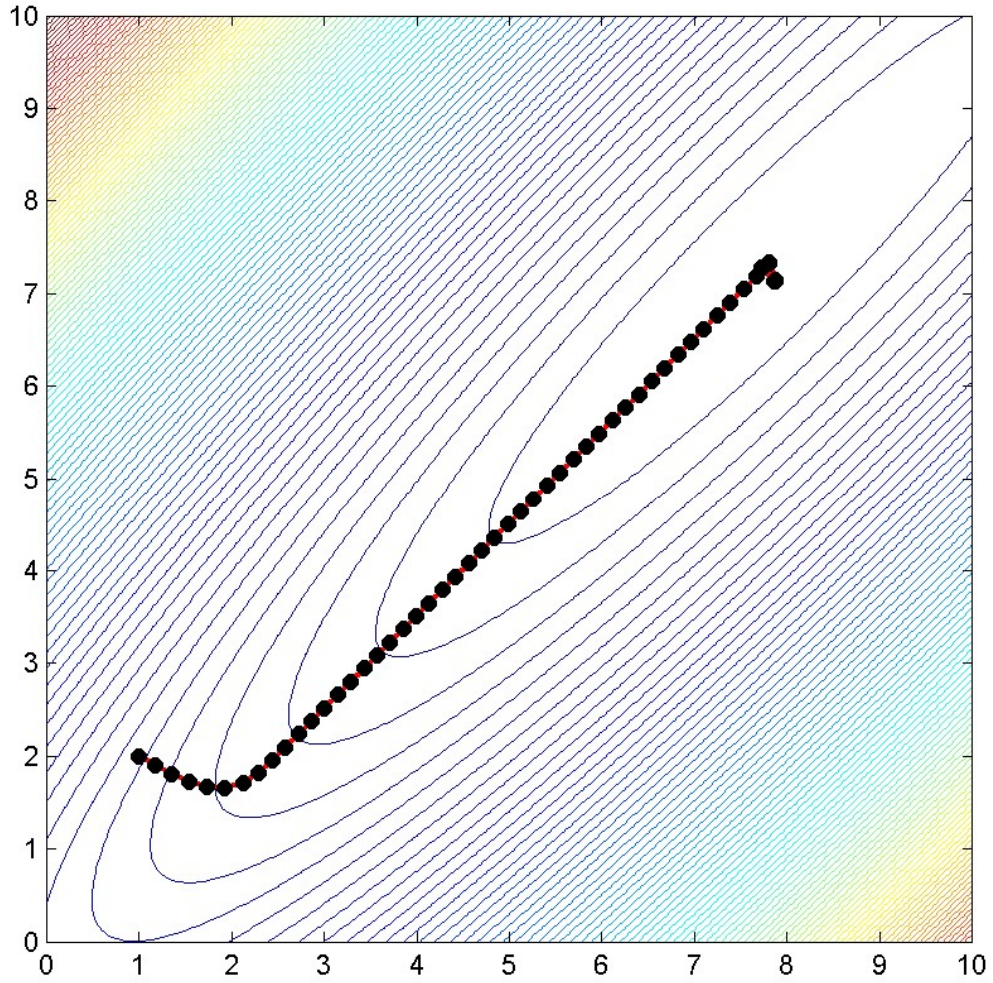
$$\frac{dE}{dw} = \frac{1}{N_b} \sum_{i=1}^{N_b} \frac{d\ell(x_{b(i)}, y_{b(i)}, w)}{dw} + \lambda \frac{dR}{dw}$$

Why batching?

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

- "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

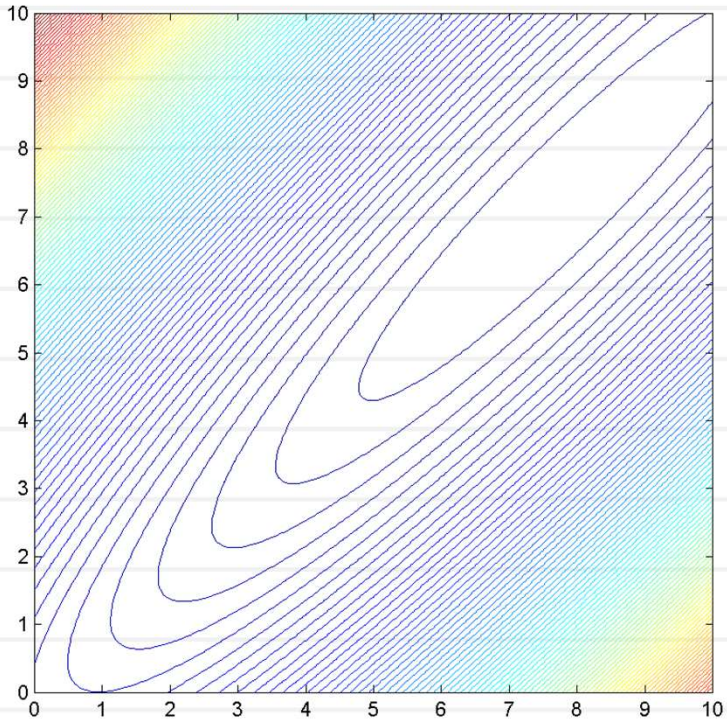
SGD inherits gradient descent problems

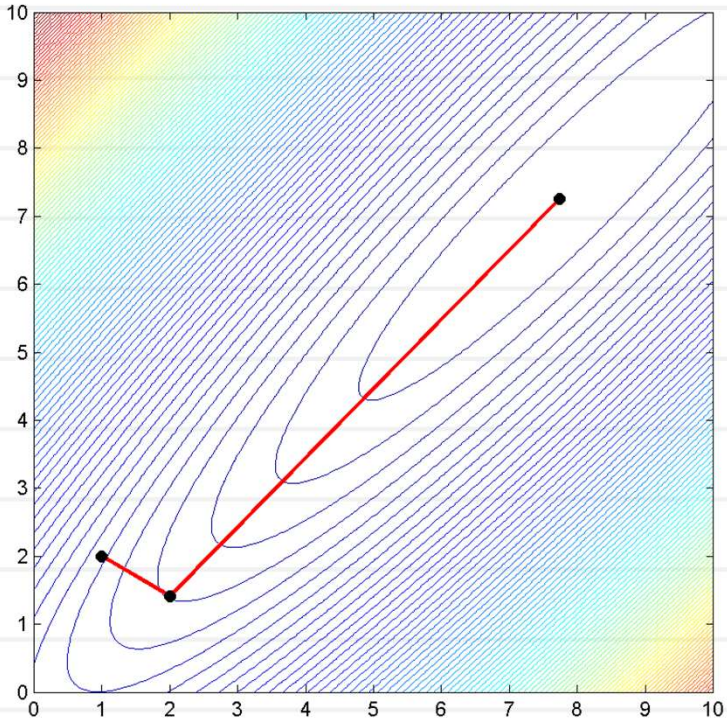


- Gradient descent is very poor "in ravines"
- SGD is no better

Better optimization methods

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

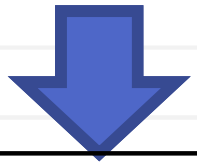




Improving SGD using momentum

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

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



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

Typical $\mu = 0.9$

Exponentially decaying running average

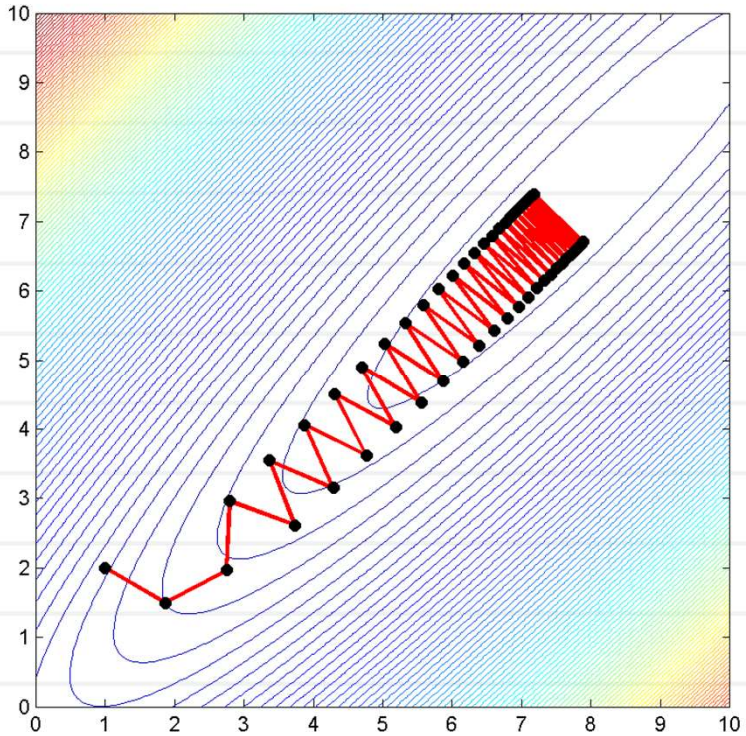
$$v[t] = \mu\, v[t-1] - \alpha[t]\, \nabla\,(E, w[t])$$
$$w[t+1] = w[t] + v[t]$$

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

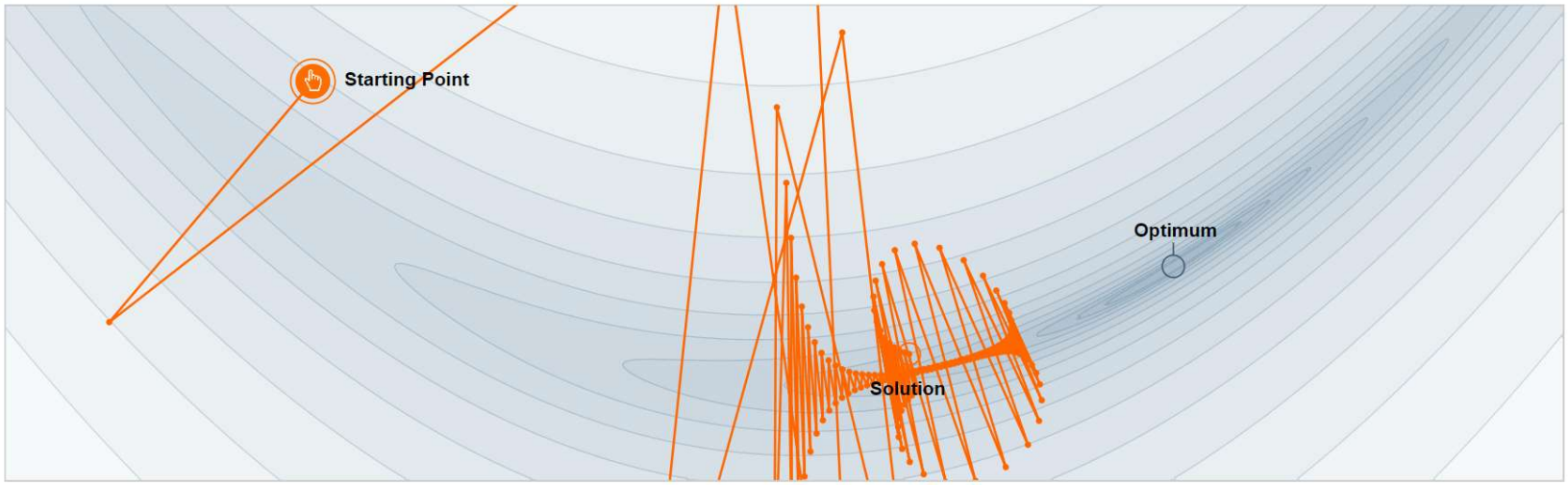
Momentum: why it works

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

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



The effect of the momentum



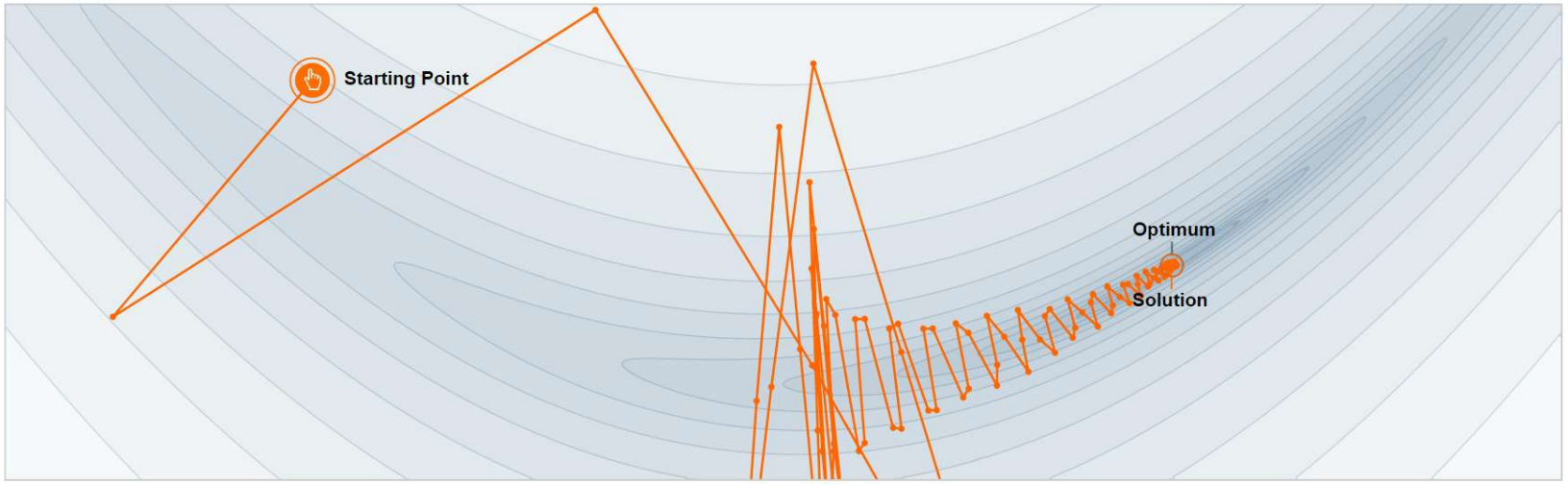
Step-size α = 0.0051

0 0.003 0.006

Momentum β = 0.0

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?



Step-size α = 0.0051

0 0.003 0.006

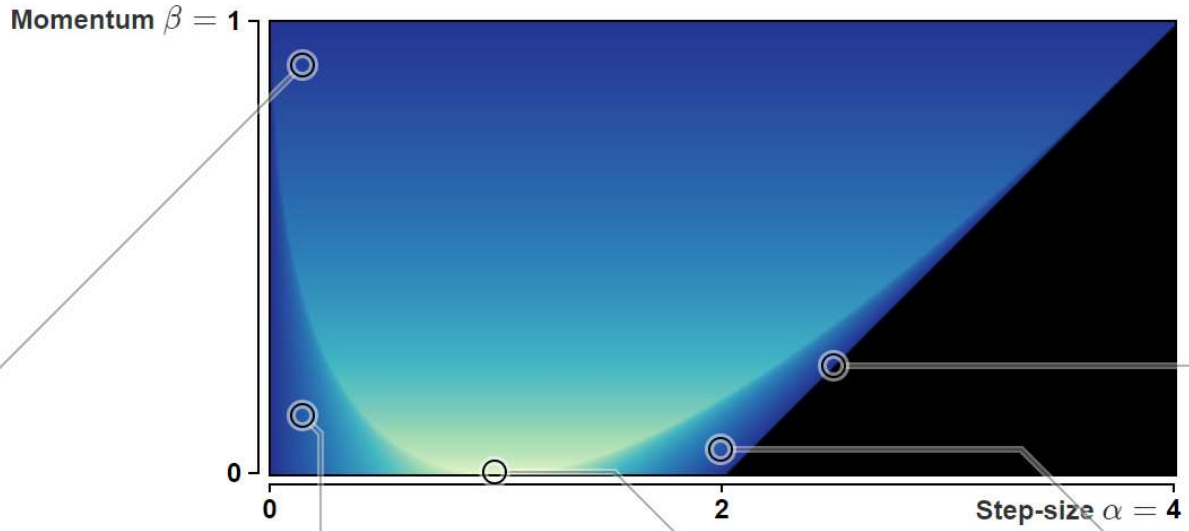
Momentum β = 0.90

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Gabriel Goh, Why momentum really works. Distill 2017

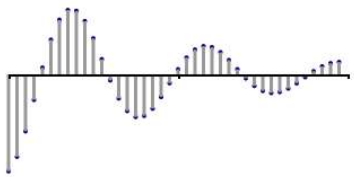
Phase space along a single eigenvector



Convergence Rate

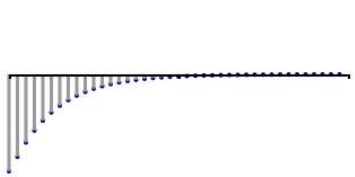
A plot of $\max\{|\sigma_1|, |\sigma_2|\}$ reveals distinct regions, each with its own style of convergence.

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

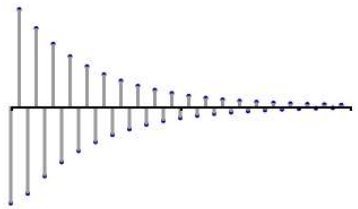


R's eigenvalues are both real, are positive, and have norm less than one. The behavior here resembles gradient descent.

1-Step Convergence

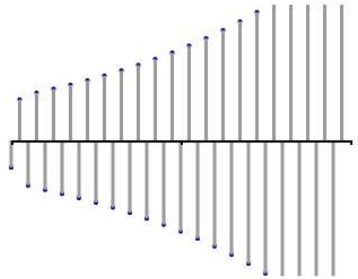
When $\alpha = 1/\lambda_i$, and $\beta = 0$, we converge in one step. This is a very special point, and kills the error in the eigenspace completely.

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.

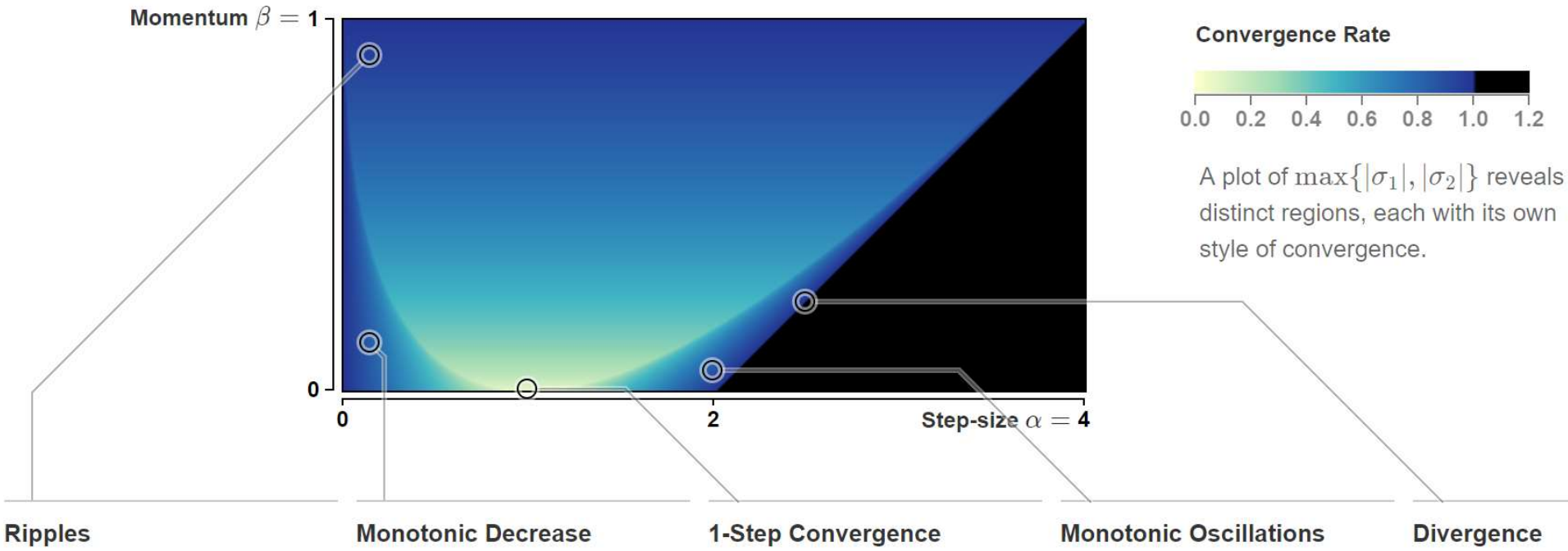
Divergence



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

Gabriel Goh, Why momentum really works. Distill 2017

The effect of the momentum



To get a global convergence rate, we must optimize over both α and β. This is a more complicated affair,[6] but they work out to be

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

Plug this into the convergence rate, and you get

$\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}$ Convergence rate, **Momentum**

$\frac{\kappa - 1}{\kappa + 1}$ Convergence rate, **Gradient Descent**

Gabriel Goh, Why momentum really works. Distill 2017

Nesterov accelerated gradient

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

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:

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

(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

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

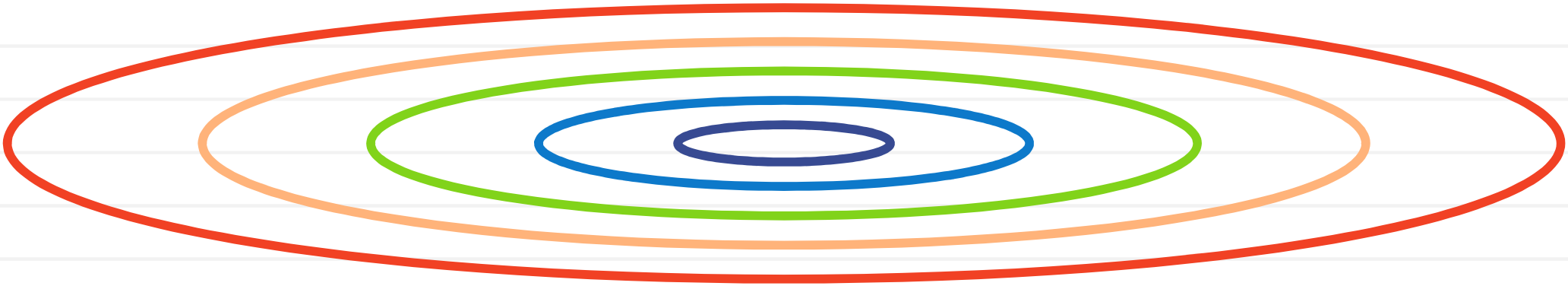
$$w[t+1] = w[t] - \frac{\alpha}{\sqrt{g[t] + \varepsilon}} \odot \nabla(E, w[t])$$

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

Adagrad method [Duchi et al. 2011]

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

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



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:

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

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

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]

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

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

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

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

Comparison: logistic regression

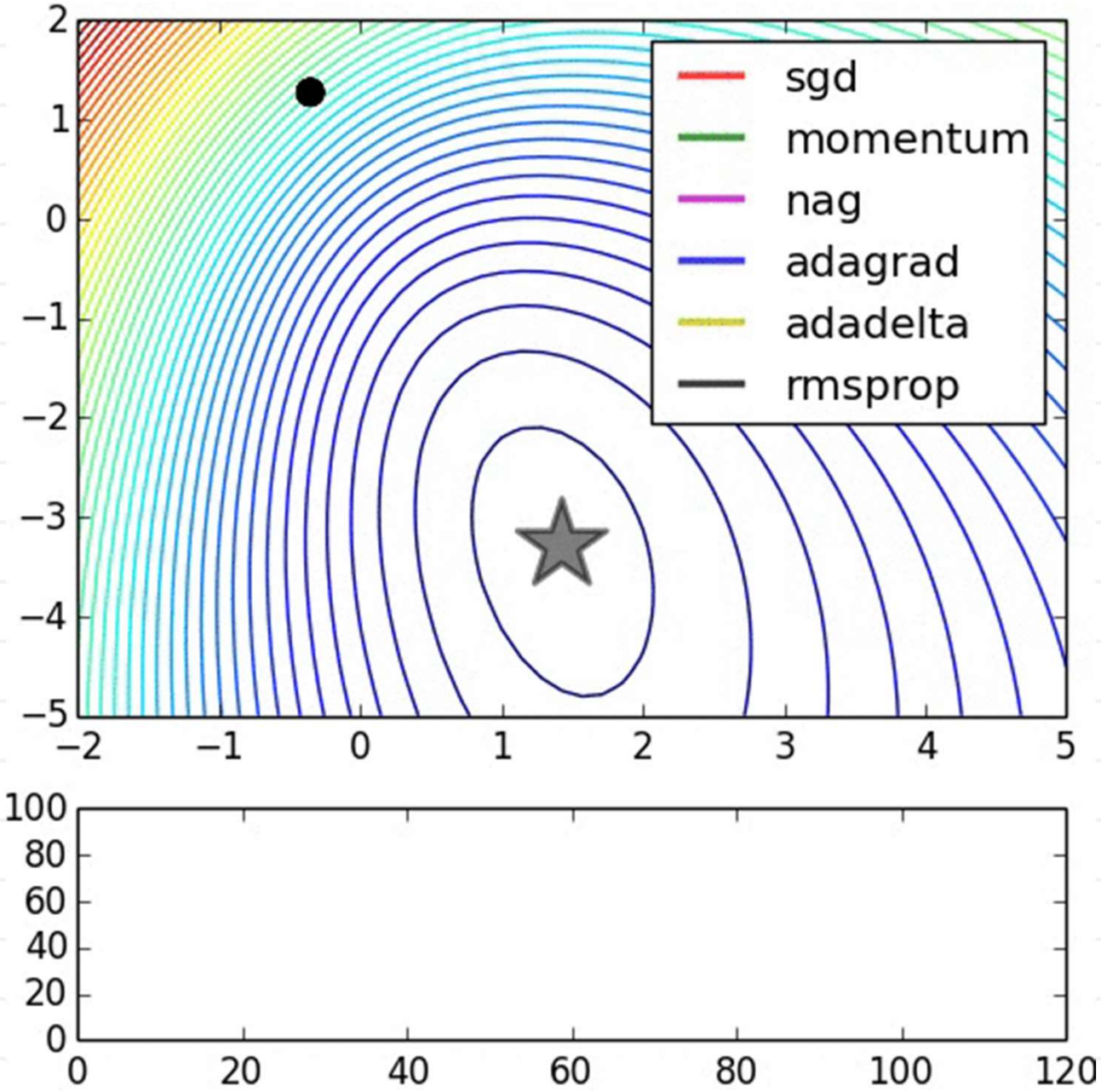


Image credit: Alec Redford

Further comparison

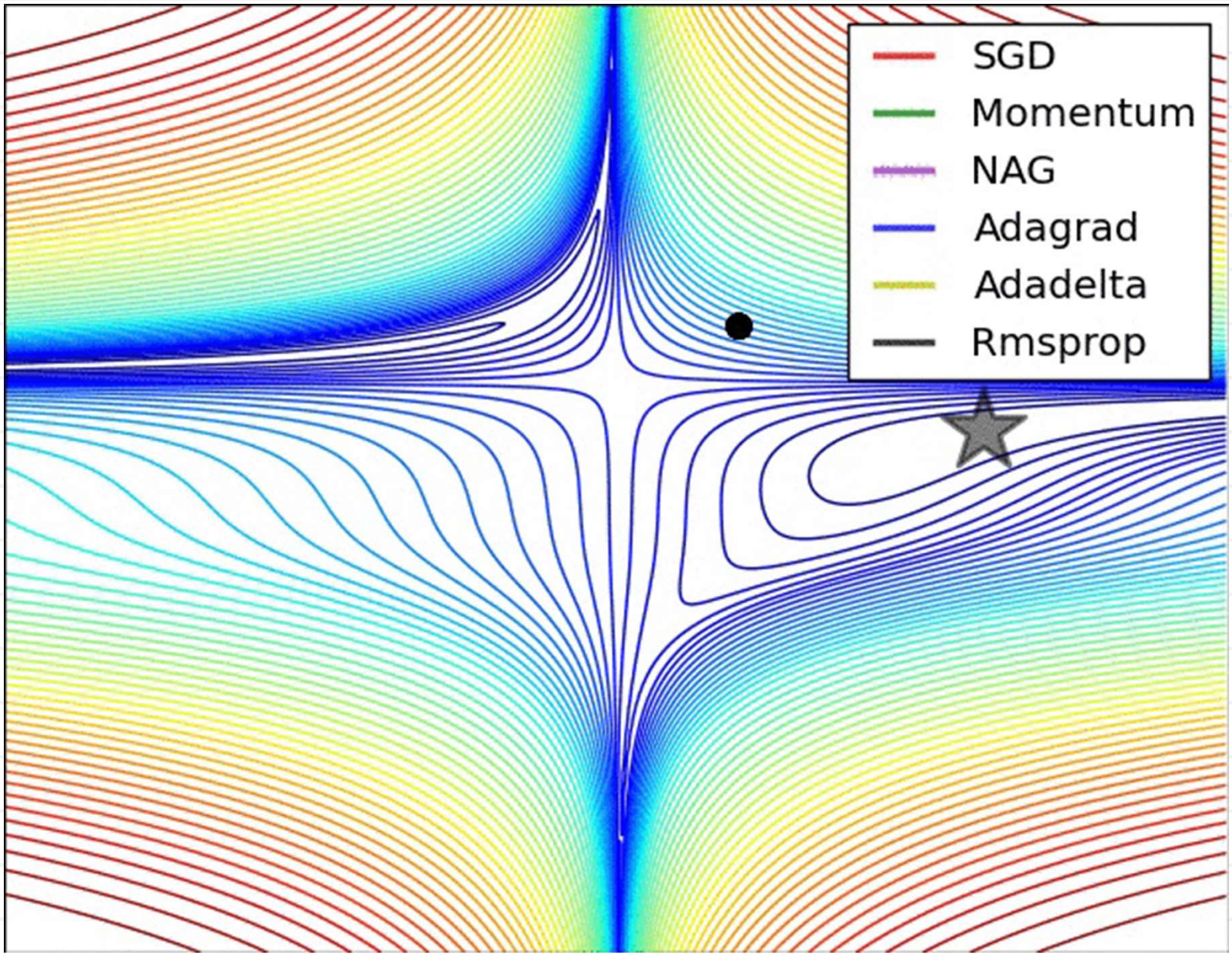


Image credit: Alec Redford

Further comparison: escaping from a saddle

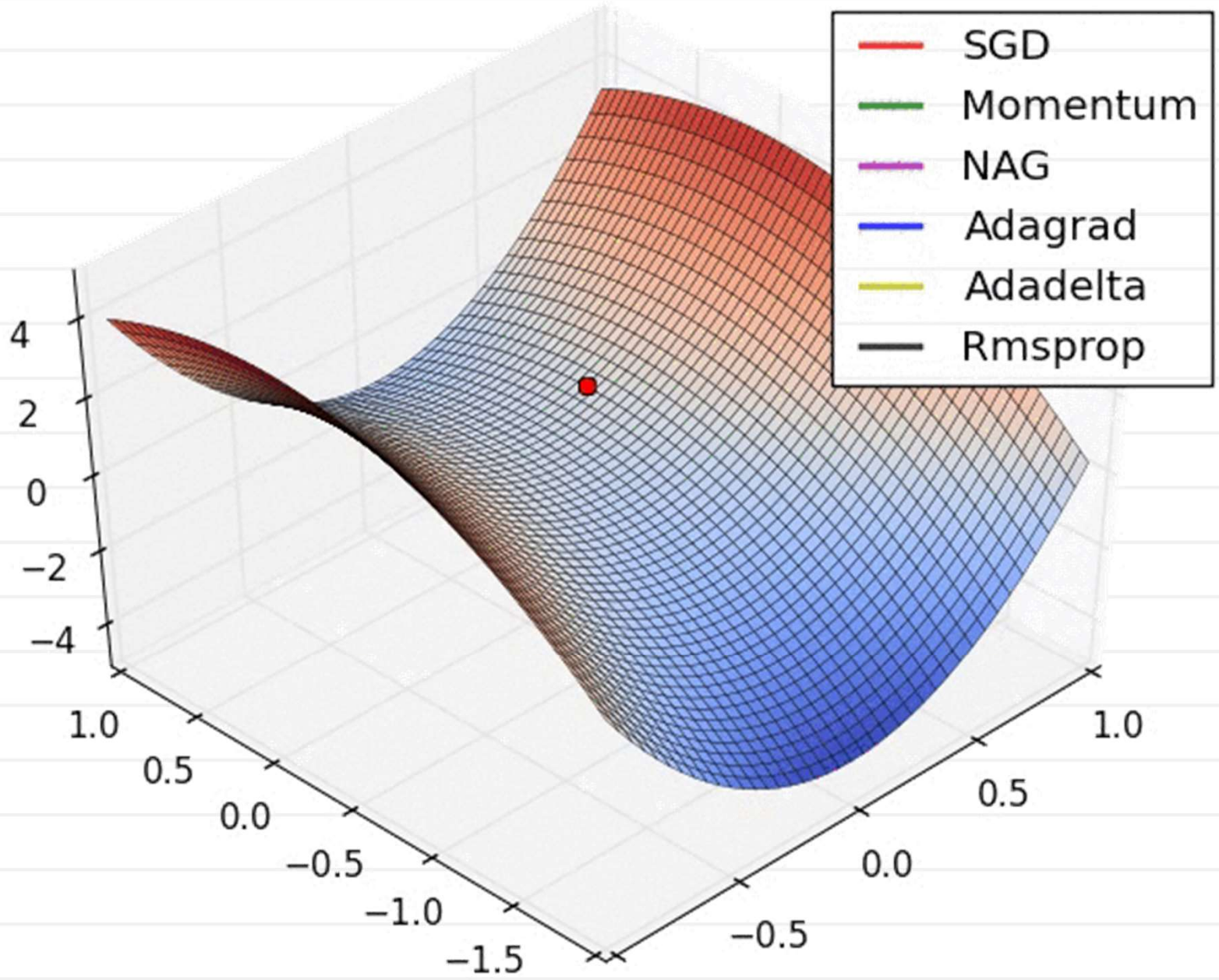


Image credit: Alec Redford

ADAM method [Kingma & Ba 2015]

ADAM = "ADAptive Moment Estimation"

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

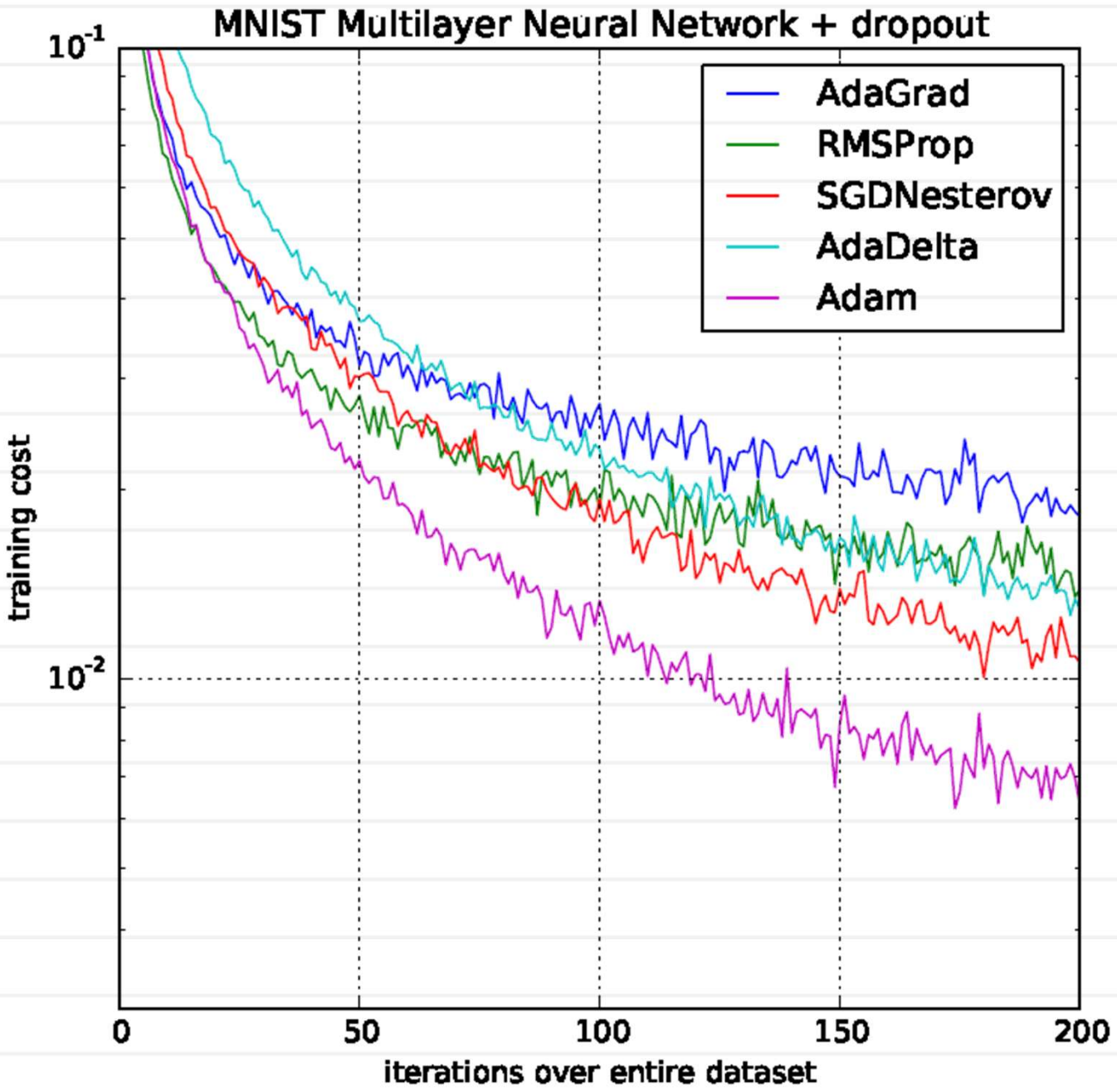
$$g[t] = \mu\, g[t-1] + (1-\mu)\,\nabla(E, w[t]) \odot \nabla(E, w[t])$$

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

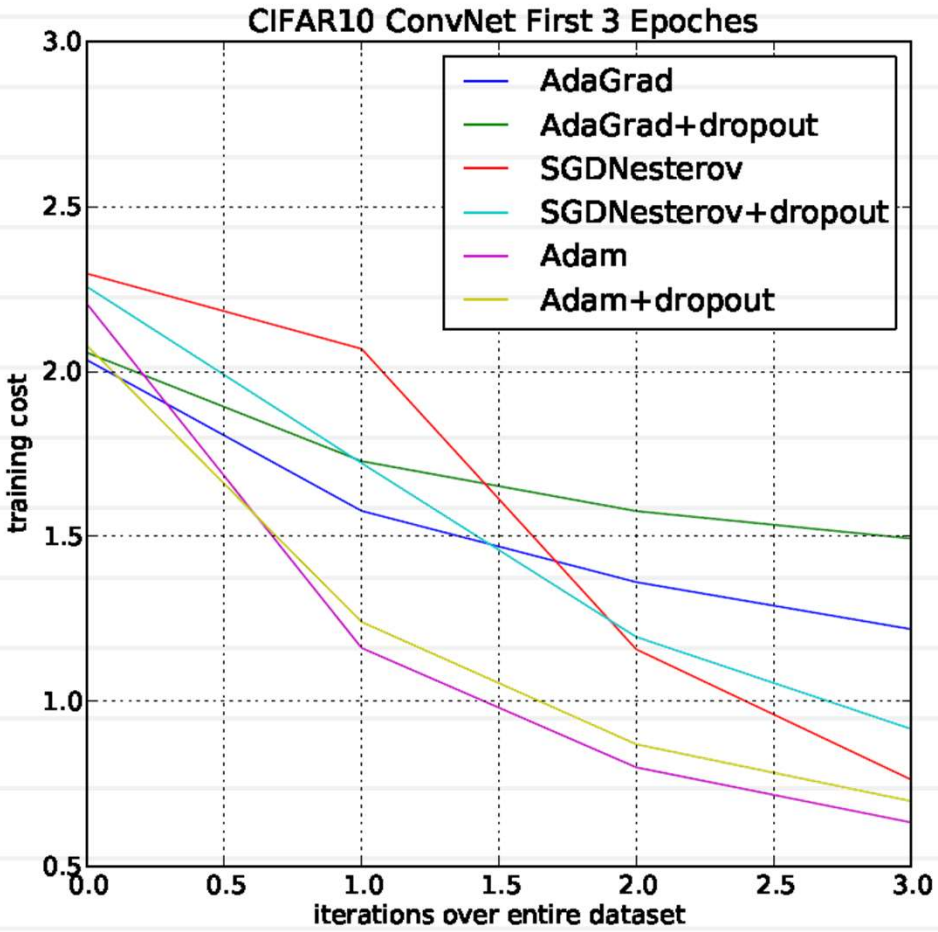
$$\frac{1}{1-\beta^t} \qquad \frac{1}{1-\mu^t}$$

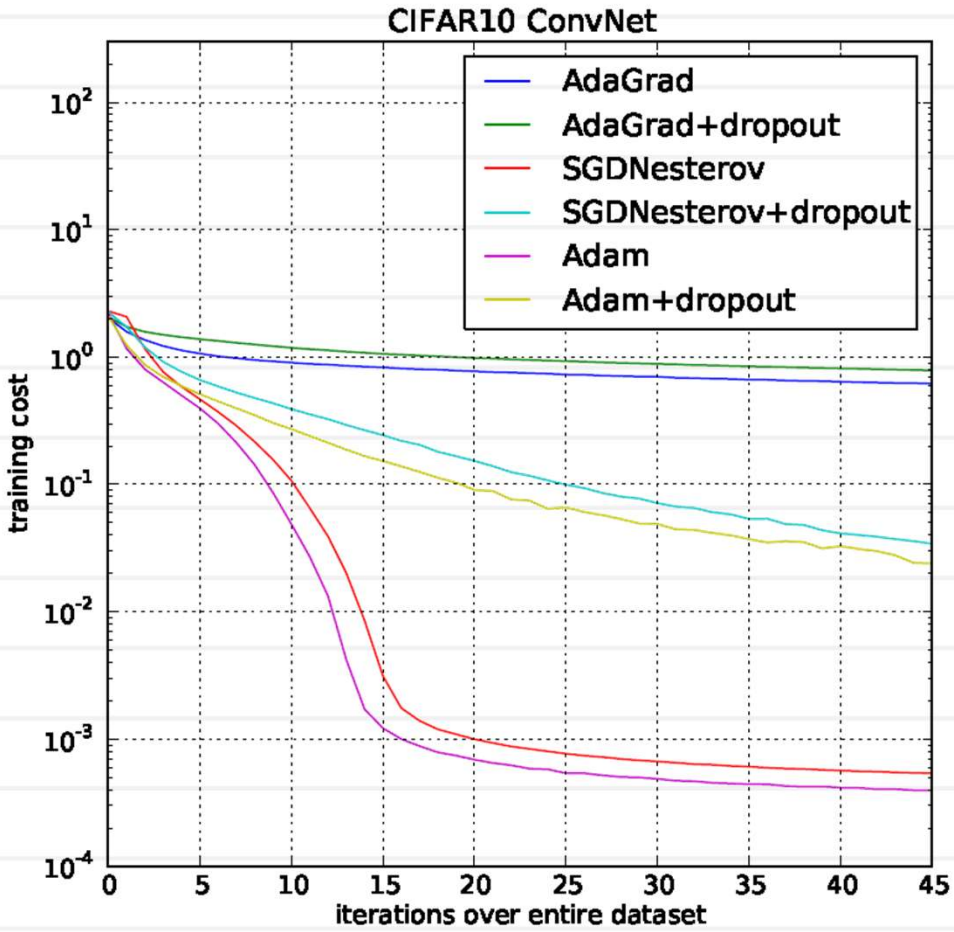
Recommended values: $\beta = 0.9$, $\mu = 0.999$, $\alpha = 0.001$, $\varepsilon = 10^{-8}$

ADAM method [Kingma & Ba 2015]

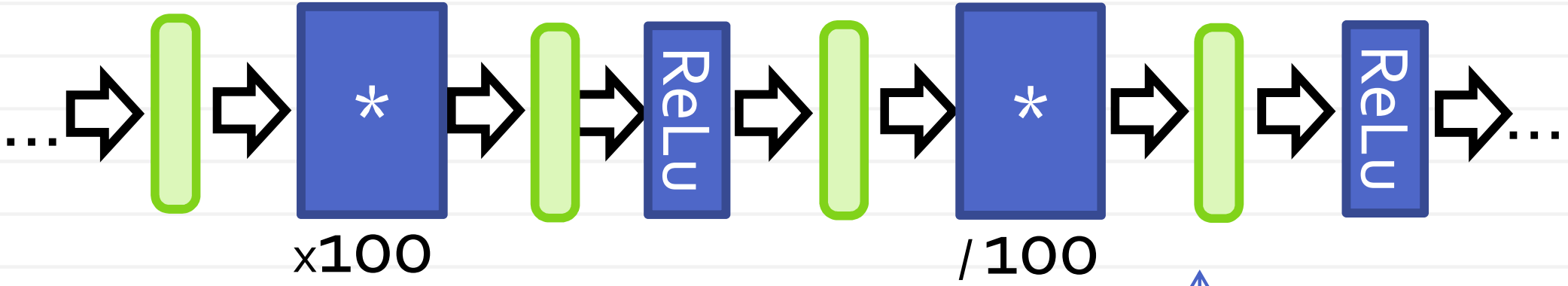


ADAM method [Kingma & Ba 2015]



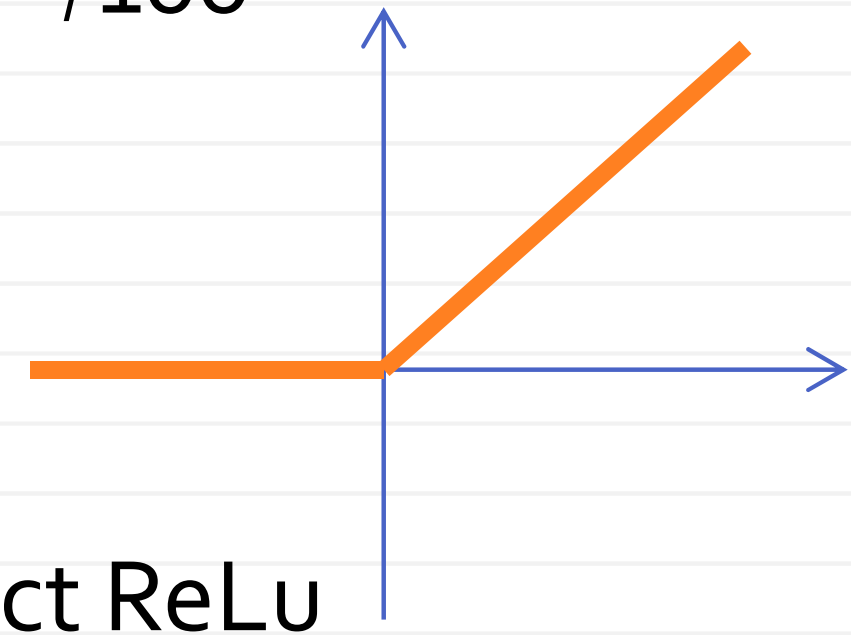


Gauge freedom in ReLu Networks



$\alpha > 0$

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



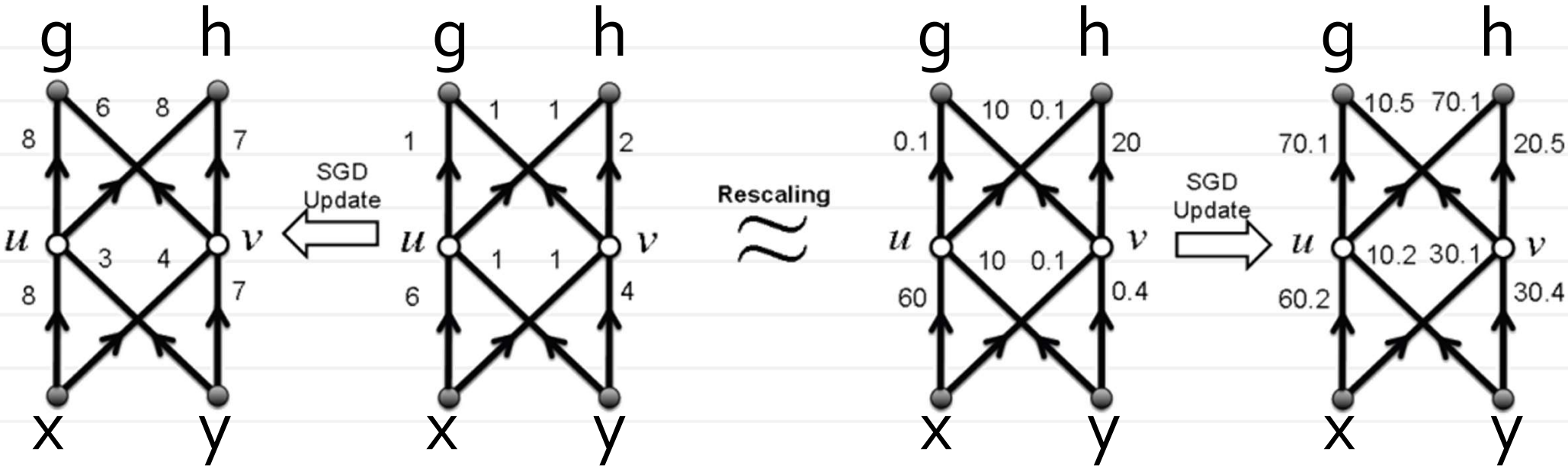
Thus: we can easily construct ReLu networks with **different** weights implementing the **same** function

Initialization schemes

$f(x_i, w_1) \rightarrow y_i$ $\qquad$ $f(x_i, w_2) \rightarrow y_i$

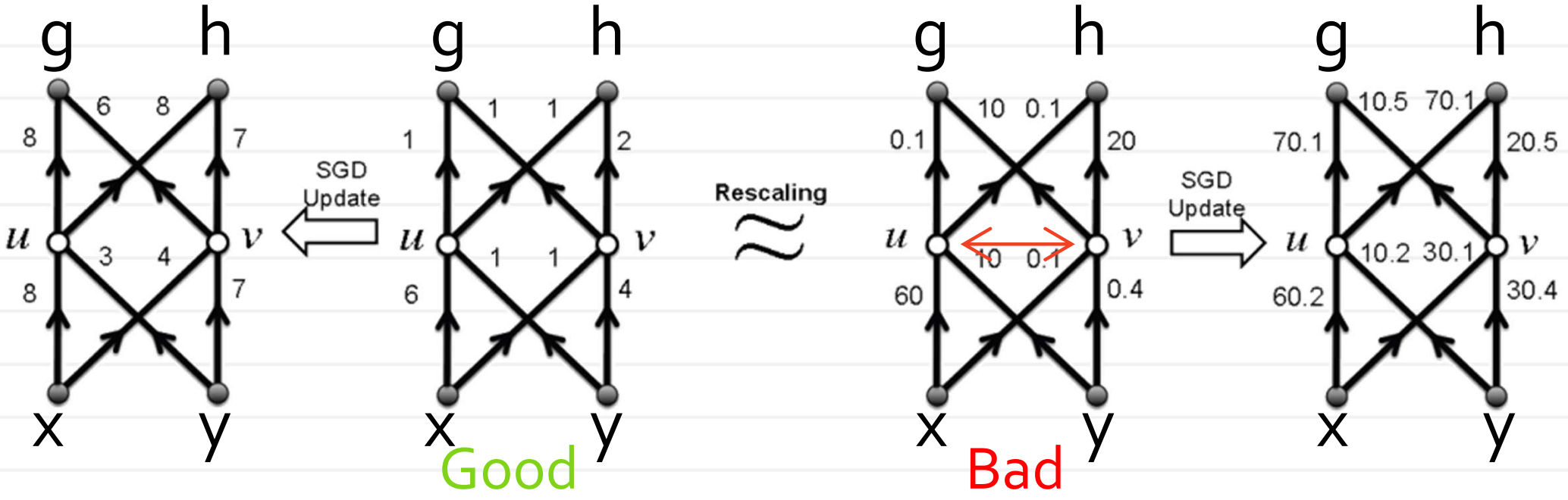
- The loss value is the same for all x_i
- The loss derivatives are not the same!

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]

Initialization schemes



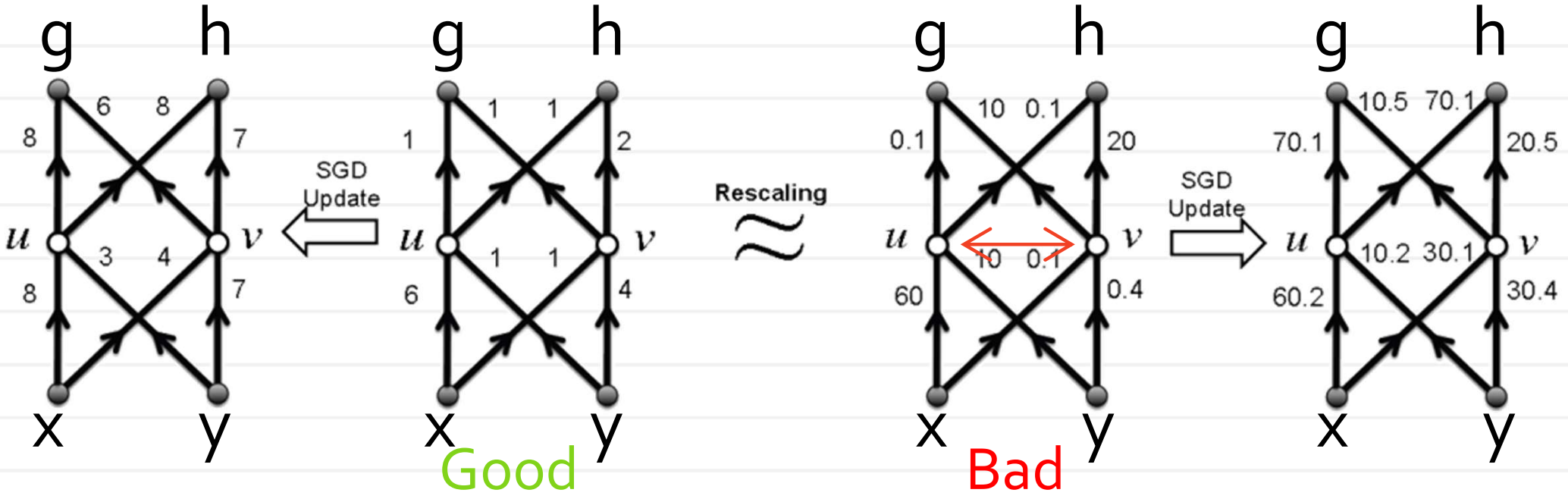
- **Basic idea 1:** units should be initialized to have comparable total input weights
- E.g. [Glorot&Bengio 2010] aka "Xavier-initialization":

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

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

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

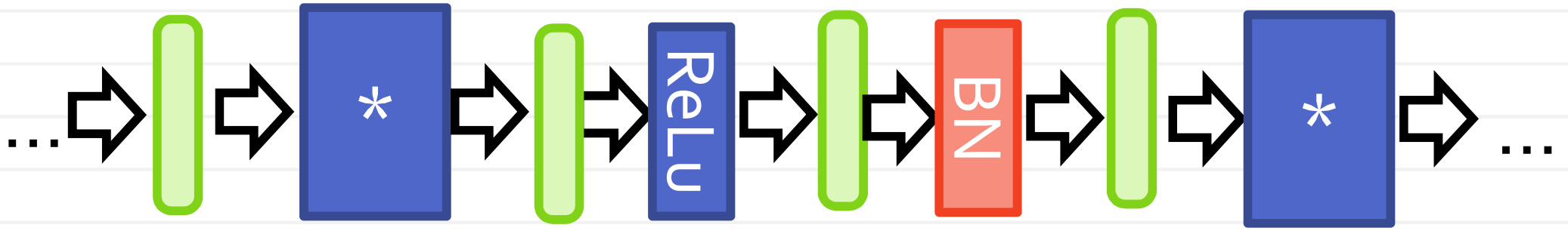
Initialization schemes



- **Basic idea 1:** units should be initialized to have comparable total input weights
- **Basic idea 2:** use layers which keep magnitude (otherwise both forwardprop and backprop will suffer from explosion/attenuation to zero)

Batch normalization

[Szegedy and Ioffe 2015]



- Makes the training process invariant to some re-parameterizations
- 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: γ, β

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

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

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

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

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

covariant to reparameterization

learnable by SGD

[Szegedy and Ioffe 2015]

Batch normalization layer

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

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

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

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

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

- At training time mean and variance are estimated per batch
- At test time, usually 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 ☹

[Szegedy and Ioffe 2015]

TODO: other normalizations

Solutions to train-test mismatch:

- Keep training time behavior
- Switch to test behavior and fine-tune
- Layer Norm [Ba et al. NIPS'16], Instance Norm [Ulyanov et al.Arxiv16], Group Norm [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

Recap

- Batch 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/k2)." 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. ICML2015: 448-456

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

Sergey Ioffe, Batch Renormalization. NIPS 2017