Deep Learning

A course about theory & practice

Learning as Optimization

Marco Piastra

About why they did not use Deep Networks from the beginning

Problem: vanishing or exploding Gradients

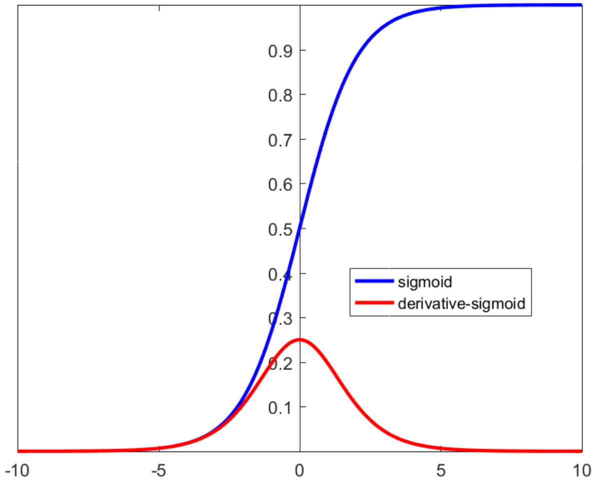
The gradient descent method implies updating the parameters at each step: making sure that the gradient does not either *vanish* or *explode* is not easy

For instance, in

$$\Delta \boldsymbol{W} = -\eta \, \frac{\partial L}{\partial \boldsymbol{W}} (\tilde{y}^{(i)}, y^{(i)})$$

the gradient contains a multiplicative term $\frac{\partial}{\partial x} g(x)$
which can be $\ll 1.0$

e.g. for the sigmoid function:



Problem: vanishing or exploding Gradients

The gradient descent method implies updating the parameters at each step: making sure that the gradient does not either *vanish* or *explode* is not easy

Consider a deep network

$$\tilde{y} = \boldsymbol{w} \cdot g(\boldsymbol{W}^{[k]} \cdots g(\boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}) \cdots + \boldsymbol{b}^{[k]}) + b$$

in which

- g *is the* *<u>identity function</u>* *and all* $\boldsymbol{b}^{[i]}$ *and* b *are* *<u>zero</u>;*
- *all hidden layers have the same size* d *of the input (i.e., al matrices are* *<u>square</u>);*
- *all* $\boldsymbol{W}^{[i]}$ *are identical and diagonalizable, with eigenbasis* $(\boldsymbol{e}_1, \cdots, \boldsymbol{e}_d)$*.*

This means that

i.e. first eigenvalue raised to the k-th power

$$\boldsymbol{W}^{[k]} \cdots \boldsymbol{W}^{[1]}\boldsymbol{x} = \boldsymbol{W}^k\boldsymbol{x} = \lambda_1^k(\boldsymbol{e}_1 \cdot \boldsymbol{x})\boldsymbol{e}_1 + \cdots \lambda_d^k(\boldsymbol{e}_d \cdot \boldsymbol{x})\boldsymbol{e}_d$$

$$= \lambda_1^k x_1 \boldsymbol{e}_1 + \cdots \lambda_d^k x_d \boldsymbol{e}_d$$

Moral: any $\lambda_i > 1$ *leads to explosion while any* $\lambda_i < 1$ *leads to vanishing*

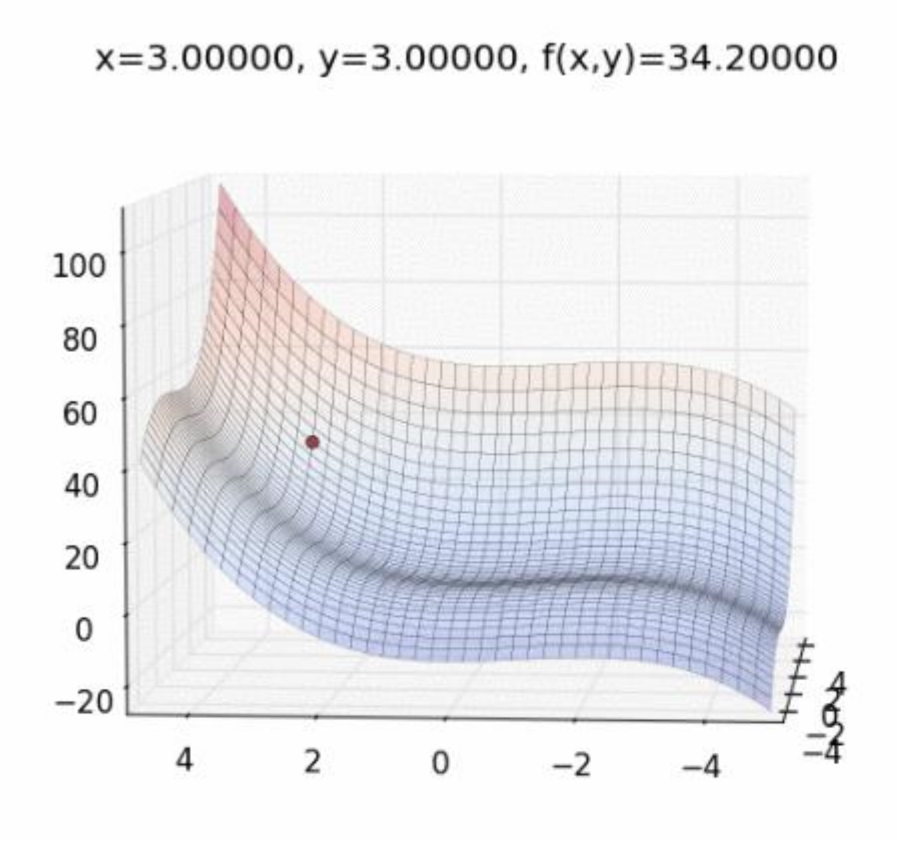
Problem: initial values of the parameters

However, the main problem of training is that of *initial values*…

Gradient Descent can only discover minima that are close to the initial values

Using deep networks
can only make this problem worse:

intuitively, with deeper networks,
the 'surface' can be even rougher…



[Image from http://cpmarkchang.logdown.com/posts/434534-optimization-method-momentum]

Improving optimization

Improving optimization

- ***SGD (or MBGD)***

  Standard, decaying learning rate
  Update step:

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

decaying
learning rate

mini-batch,
possibly a singleton

Improving optimization

- ***SGD (or MBGD)***

Standard, decaying learning rate
Update step:

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \underbrace{\eta}_{\text{decaying learning rate}} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(\underbrace{B}_{\text{mini-batch, possibly a singleton}}, \boldsymbol{\vartheta}^{(t-1)})$$

Many different ways to improve performance and speed rate:

- add some *momentum*
- take in account *2nd order derivatives*
- make the *learning rate adaptive*

Improving optimization

- ***SGD (or MBGD)***

Standard, decaying learning rate
Update step:

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

$$\eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

“force felt by the ball”

$$\boldsymbol{f} = -\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta})$$

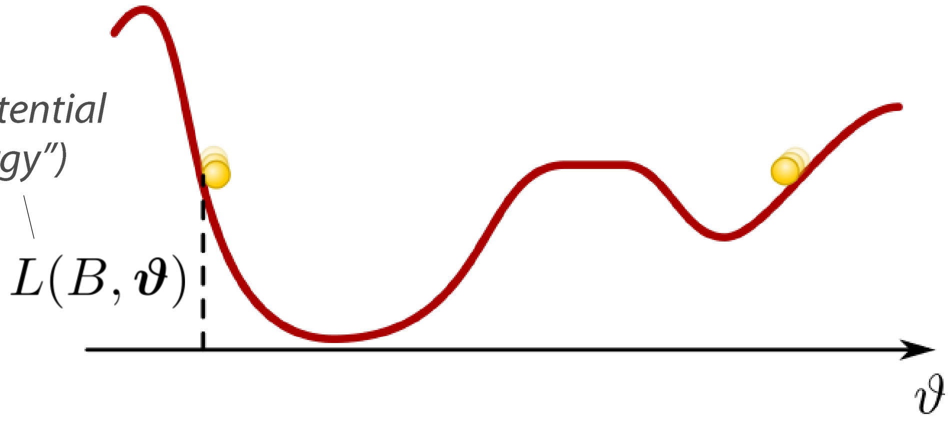
“acceleration”

$$\boldsymbol{f} = m\boldsymbol{a}$$

$$\boldsymbol{a} \propto -\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta})$$

... the gradient directly affects the <u>velocity</u>
(not the position)

Loss (“potential energy”)



Momentum

- ***Momentum***

  "Let the ball run“

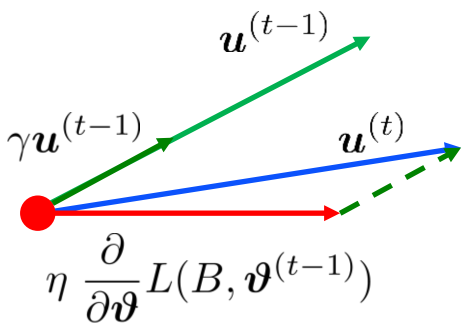
momentum term:
tendency to keep running at the same speed and direction

$$\boldsymbol{u}^{(t)} = \overbrace{\gamma \boldsymbol{u}^{(t-1)}} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}), \quad \boldsymbol{u}^{(0)} = \mathbf{0}$$

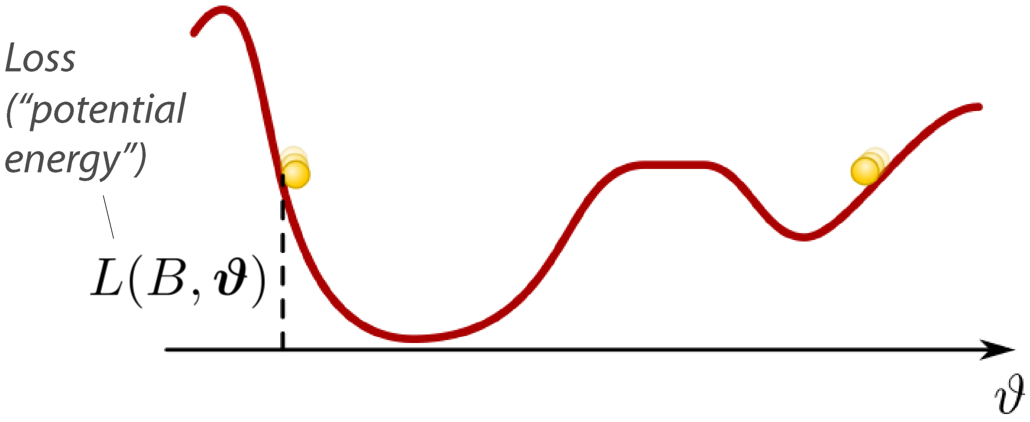
$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} + \boldsymbol{u}^{(t)}$$

$0 < \gamma < 1$

“coefficient of friction”



$\boldsymbol{u}^{(t-1)}$
$\gamma \boldsymbol{u}^{(t-1)}$
$\boldsymbol{u}^{(t)}$
$\eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$



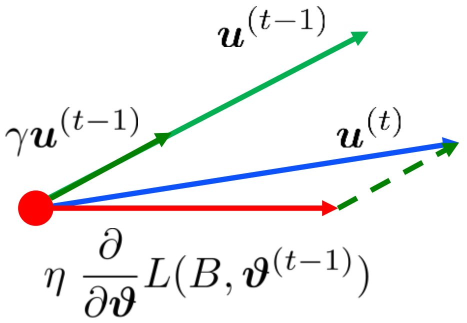
Momentum

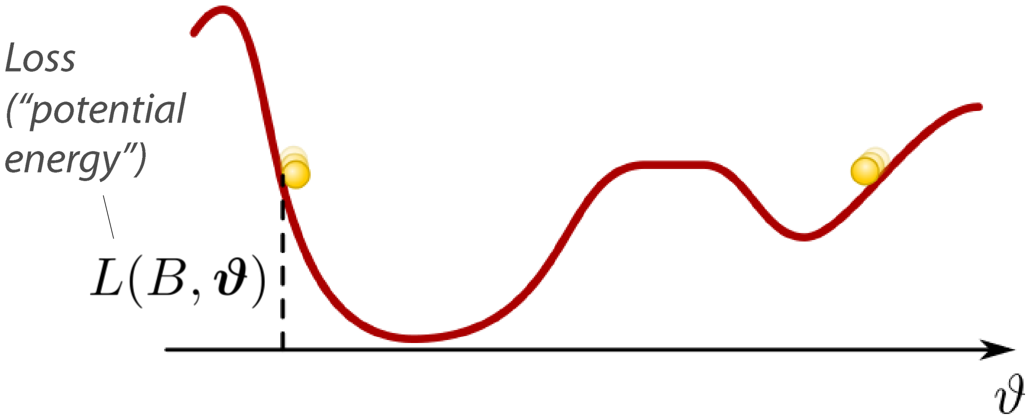
- ***Momentum***

  "Let the ball run“

$$\boldsymbol{u}^{(t)} = \gamma \boldsymbol{u}^{(t-1)} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}), \quad \boldsymbol{u}^{(0)} = \boldsymbol{0}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} + \boldsymbol{u}^{(t)}$$


$\boldsymbol{u}^{(t-1)}$
$\gamma \boldsymbol{u}^{(t-1)}$
$\boldsymbol{u}^{(t)}$
$\eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$



Consider $\boldsymbol{\vartheta}$ *as a position ...*

"velocity"

$$\boldsymbol{u} := \frac{\partial}{\partial t} \boldsymbol{\vartheta} \approx \boldsymbol{\vartheta}^{(t)} - \boldsymbol{\vartheta}^{(t-1)}$$

"acceleration"

$$\boldsymbol{a} \approx \boldsymbol{u}^{(t)} - \boldsymbol{u}^{(t-1)} \propto -\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta})$$

... the gradient directly affects the velocity (not the position)

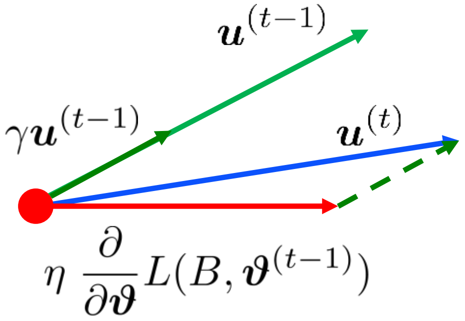
NAG

- ## *Momentum*

  "Let the ball run“

$$\boldsymbol{u}^{(t)} = \gamma \boldsymbol{u}^{(t-1)} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}), \quad \boldsymbol{u}^{(0)} = \mathbf{0}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} + \boldsymbol{u}^{(t)}$$

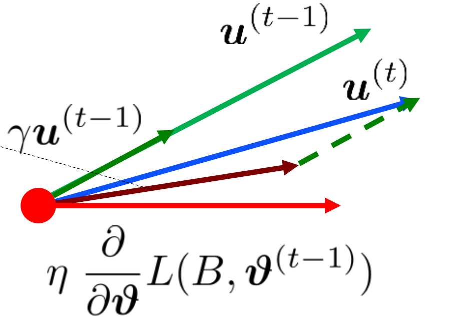

$\boldsymbol{u}^{(t-1)}$
$\gamma \boldsymbol{u}^{(t-1)}$
$\boldsymbol{u}^{(t)}$
$\eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$

- ## *Nesterov Accelerated Gradient (NAG)*

  "Let the ball run but be predictive"

$$\boldsymbol{u}^{(t)} = \gamma \boldsymbol{u}^{(t-1)} - \eta \, \underline{\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)} + \gamma \boldsymbol{u}^{(t-1)})}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} + \boldsymbol{u}^{(t)}$$

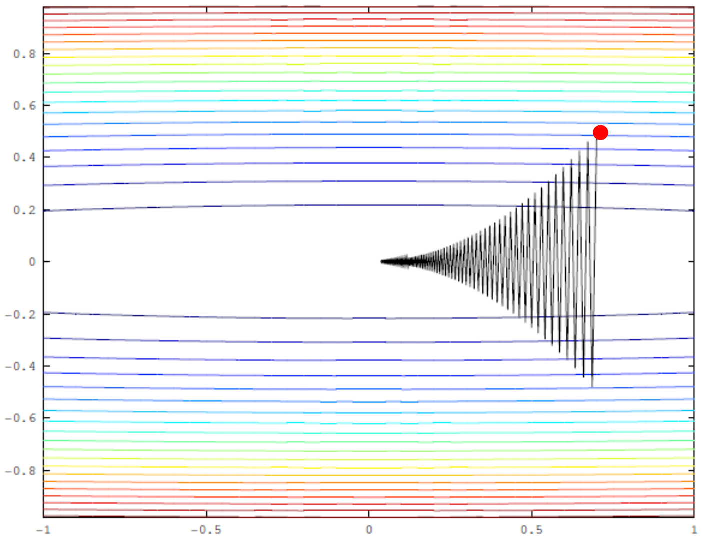

$\boldsymbol{u}^{(t-1)}$
$\boldsymbol{u}^{(t)}$
$\gamma \boldsymbol{u}^{(t-1)}$
$\eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$

2nd order methods

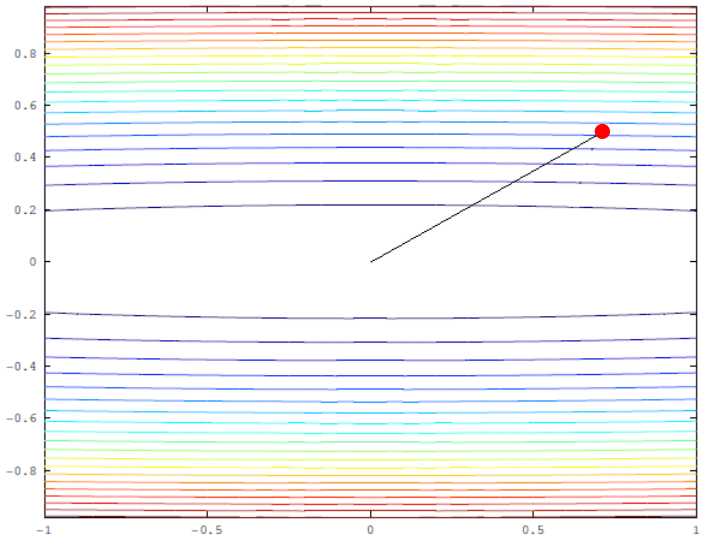
In this example (geometric view)

The level curves of a quadratic form in 2D are ellipses centered in the origin

Gradient Descent



Newton-Raphson



2nd order methods

- **Taylor's expansion**

$$L(B,\boldsymbol{\vartheta}) = L(B,\boldsymbol{\vartheta}^{(t-1)}) + \frac{\partial}{\partial\boldsymbol{\vartheta}}L(B,\boldsymbol{\vartheta}^{(t-1)})\cdot(\boldsymbol{\vartheta}-\boldsymbol{\vartheta}^{(t-1)})$$
$$+\frac{1}{2}(\boldsymbol{\vartheta}-\boldsymbol{\vartheta}^{(t-1)})\cdot\boldsymbol{H}\ (\boldsymbol{\vartheta}-\boldsymbol{\vartheta}^{(t-1)}) + \ \ldots$$

All terms in blue are constant

where:

$$\boldsymbol{H} := \frac{\partial}{\partial\boldsymbol{\vartheta}}\left(\frac{\partial}{\partial\boldsymbol{\vartheta}}L(B,\boldsymbol{\vartheta}^{(t-1)})\right)$$

The Hessian Matrix

- **Differentiate both sides and take** $\boldsymbol{\vartheta} = \boldsymbol{\vartheta}^{*}$

The argmin

$$\underbrace{\frac{\partial}{\partial\boldsymbol{\vartheta}}L(B,\boldsymbol{\vartheta}^{*})}_{\text{this must be 0}} = \frac{\partial}{\partial\boldsymbol{\vartheta}}L(B,\boldsymbol{\vartheta}^{(t-1)}) + \boldsymbol{H}(\boldsymbol{\vartheta}^{*}-\boldsymbol{\vartheta}^{(t-1)})$$

then:

$$\boldsymbol{\vartheta}^{*}-\boldsymbol{\vartheta}^{(t-1)} = -\boldsymbol{H}^{-1}\frac{\partial}{\partial\boldsymbol{\vartheta}}L(B,\boldsymbol{\vartheta}^{(t-1)})$$

2nd order methods

- ***Gradient Descent***

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

- ***Newton-Raphson's optimization method***

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \boldsymbol{H}^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

where: $$\boldsymbol{H} := \frac{\partial}{\partial \boldsymbol{\vartheta}} \left(\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \right)$$

Why is the Newton-Raphson's method better than GD?

2nd order methods

- ***Newton-Raphson's optimization method***

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\, \boldsymbol{H}^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \qquad \boldsymbol{H} := \frac{\partial}{\partial \boldsymbol{\vartheta}} \left(\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \right)$$

Example

a quadratic form, centered in the origin

$$L(B, \boldsymbol{\vartheta}) = \boldsymbol{\vartheta} \cdot \boldsymbol{A}\boldsymbol{\vartheta}$$

where;

a diagonal, positive definite matrix (therefore, L is convex)

$$\boldsymbol{A} := \begin{bmatrix} a_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & a_d \end{bmatrix}, \quad a_i > 0\ \forall i = 1, \dots, d$$

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}) = 2\boldsymbol{A}\boldsymbol{\vartheta}$$

$$\boldsymbol{H} = \frac{\partial}{\partial \boldsymbol{\vartheta}} \left(\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}) \right) = 2\boldsymbol{A} \qquad \boldsymbol{H}^{-1} = \frac{1}{2}\boldsymbol{A}^{-1} = \frac{1}{2} \begin{bmatrix} 1/a_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1/a_d \end{bmatrix}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\, \frac{1}{2} \boldsymbol{A}^{-1} 2\boldsymbol{A}\boldsymbol{\vartheta}^{(t-1)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \boldsymbol{\vartheta}^{(t-1)} = (1 - \eta)\boldsymbol{\vartheta}^{(t-1)}$$

What??

2nd order methods

In this example (geometric view)

$$L(B, \boldsymbol{\vartheta}) = \boldsymbol{\vartheta} \cdot \boldsymbol{A}\boldsymbol{\vartheta} \qquad \text{with} \qquad \boldsymbol{A} := \begin{bmatrix} a_1 & 0 \\ 0 & a_2 \end{bmatrix}, \; a_1 \ll a_2$$

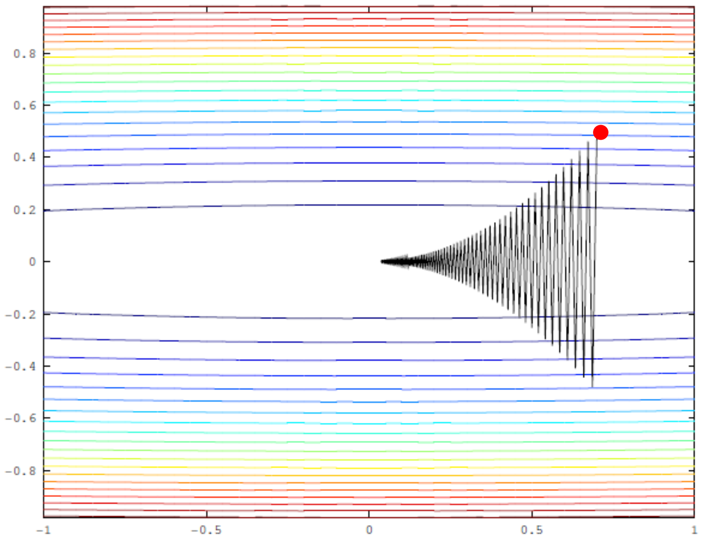
Gradient Descent

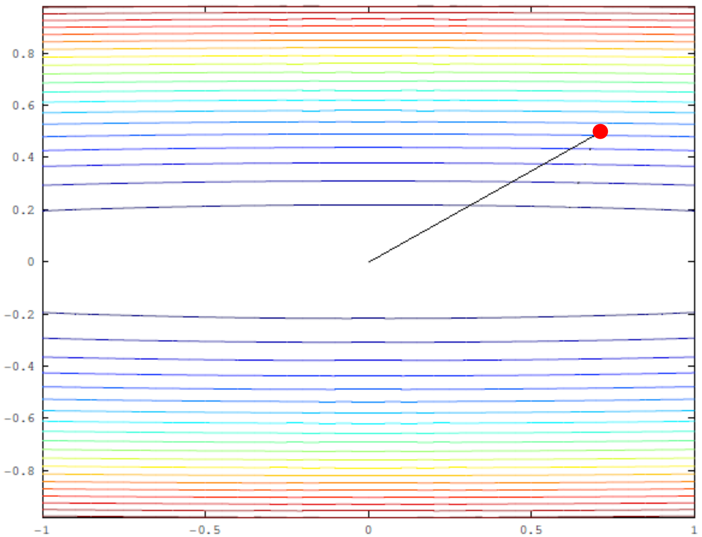
$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta 2\boldsymbol{A}\boldsymbol{\vartheta}^{(t-1)}$$

Newton-Raphson

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\boldsymbol{\vartheta}^{(t-1)}$$

The level curves of a quadratic form in 2D are ellipses centered in the origin



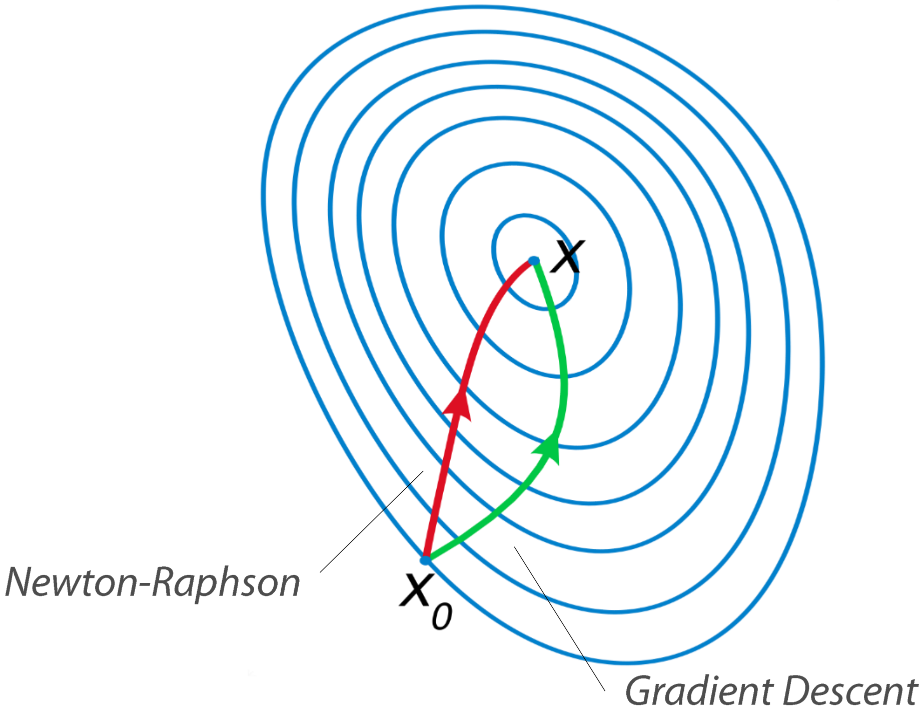


2nd order methods

- ***Newton-Raphson's optimization method***

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \boldsymbol{H}^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \qquad \boldsymbol{H} := \frac{\partial}{\partial \boldsymbol{\vartheta}} \left(\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \right)$$

The (inverse of the) Hessian Matrix takes into account also the curvature



AdaGrad

- ***Newton-Raphson's optimization method***

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \boldsymbol{H}^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \qquad \boldsymbol{H} := \frac{\partial}{\partial \boldsymbol{\vartheta}} \left(\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \right)$$

However

- Computing the inverse Hessian matrix is not easy, in general
- It requires $\mathcal{O}(d^3)$ time versus $\mathcal{O}(d)$ of the gradient

d is the number of parameters

AdaGrad

- ***Newton-Raphson's optimization method***

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\, \boldsymbol{H}^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \qquad \boldsymbol{H} := \frac{\partial}{\partial \boldsymbol{\vartheta}} \left(\frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)}) \right)$$

However

- Computing the inverse Hessian matrix is not easy, in general
- It requires $\mathcal{O}(d^3)$ time versus $\mathcal{O}(d)$ of the gradient

d is the number of parameters

- ***AdaGrad approximation***

$$G_i^{(t)} := \sqrt{\sum_{j=1}^{t} \left(\frac{\partial}{\partial \vartheta_i} L(B, \boldsymbol{\vartheta}^{(j)}) \right)^2} \qquad \boldsymbol{G}^{(t)} := \begin{bmatrix} G_1^{(t)} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & G_d^{(t)} \end{bmatrix}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\, (\boldsymbol{G}^{(t-1)})^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

AdaGrad

Gradient Descent

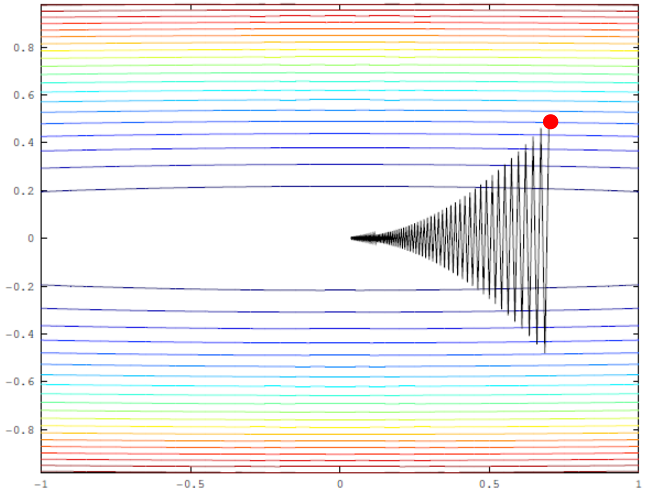
$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

Newton-Raphson

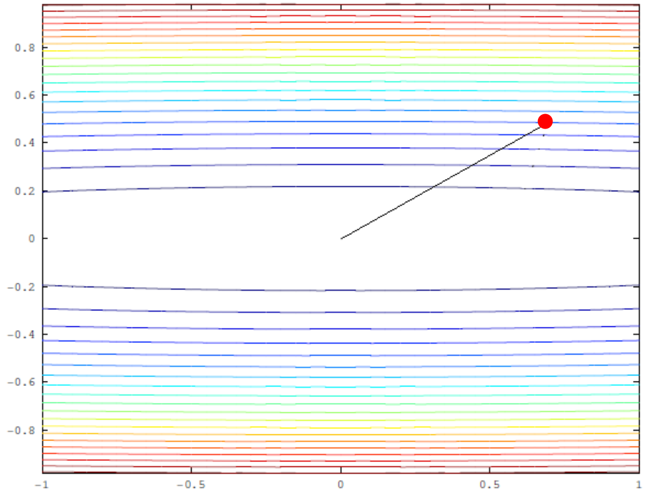
$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \boldsymbol{H}^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

AdaGrad

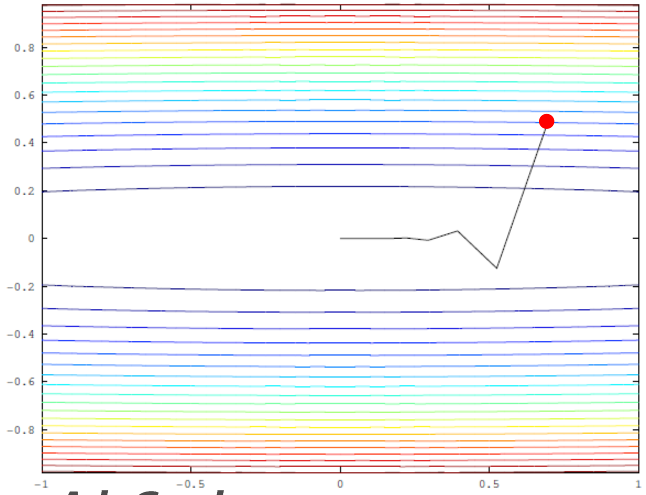
$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, (\boldsymbol{G}^{(t-1)})^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$



Gradient Descent



Newton-Raphson



AdaGrad

RMSprop

- ***AdaGrad approximation***

$$G_i^{(t)} := \sqrt{\sum_{j=1}^{t} \left(\frac{\partial}{\partial \vartheta_i} L(B, \boldsymbol{\vartheta}^{(j)}) \right)^2}$$

- ***RMSprop approximation***

The overall sum is replaced by the exponential moving average (EMA)

$$g_i^{(t)} := \frac{\partial}{\partial \vartheta_i} L(B, \boldsymbol{\vartheta}^{(t)})$$

$$\mathrm{EMA}(g_i^2)^{(t)} := \gamma (g_i^{(t)})^2 + (1-\gamma)\mathrm{EMA}(g_i^2)^{(t-1)}$$

$$\boxed{G_i^{(t)} := \sqrt{\mathrm{EMA}(g_i^2)^{(t)}}}$$

$$\boldsymbol{G}^{(t)} := \begin{bmatrix} G_1^{(t)} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & G_d^{(t)} \end{bmatrix}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \left(\boldsymbol{G}^{(t-1)} \right)^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

AdaDelta

- ***RMSprop approximation***

$$g_i^{(t)} := \frac{\partial}{\partial \vartheta_i} L(B, \boldsymbol{\vartheta}^{(t)})$$

$$\mathrm{EMA}(g_i^2)^{(t)} := \gamma (g_i^{(t)})^2 + (1-\gamma)\mathrm{EMA}(g_i^2)^{(t-1)}$$

$$\boxed{G_i^{(t)} := \sqrt{\mathrm{EMA}(g_i^2)^{(t)}}}$$

Hessian approximation

$$\boldsymbol{G}^{(t)} := \begin{bmatrix} G_1^{(t)} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & G_d^{(t)} \end{bmatrix}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\, (\boldsymbol{G}^{(t-1)})^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

- ***AdaDelta approximation***

$$\boxed{D_i^{(t)} := \sqrt{\mathrm{EMA}(\Delta\vartheta_i^2)^{(t)}}}$$

'momentum' factor

$$\boldsymbol{D}^{(t)} := \begin{bmatrix} D_1^{(t)} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & D_d^{(t)} \end{bmatrix}$$

$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta\, \boldsymbol{D}^{(t-1)} (\boldsymbol{G}^{(t-1)})^{-1} \frac{\partial}{\partial \boldsymbol{\vartheta}} L(B, \boldsymbol{\vartheta}^{(t-1)})$$

Improving optimization

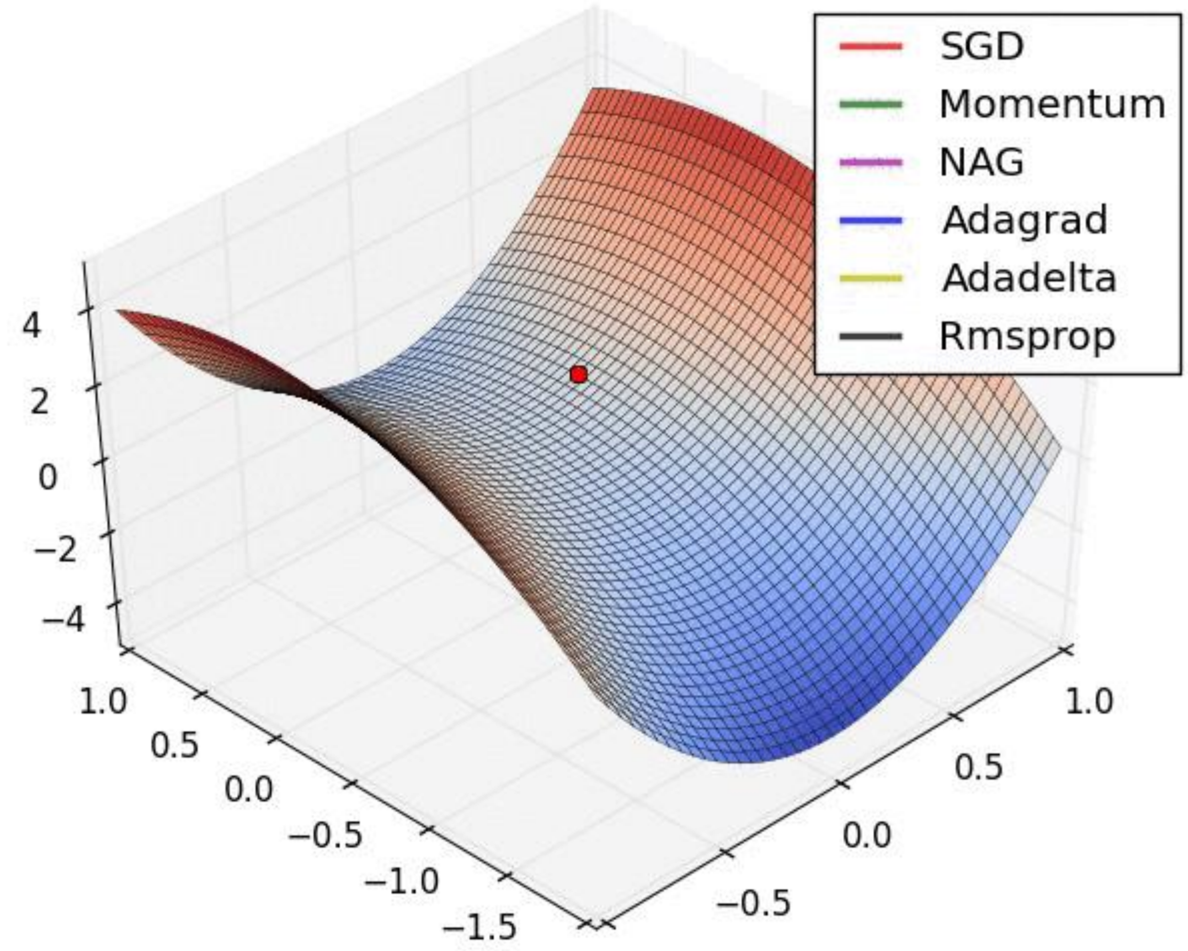


Image from https://imgur.com/a/Hqolp

Improving optimization

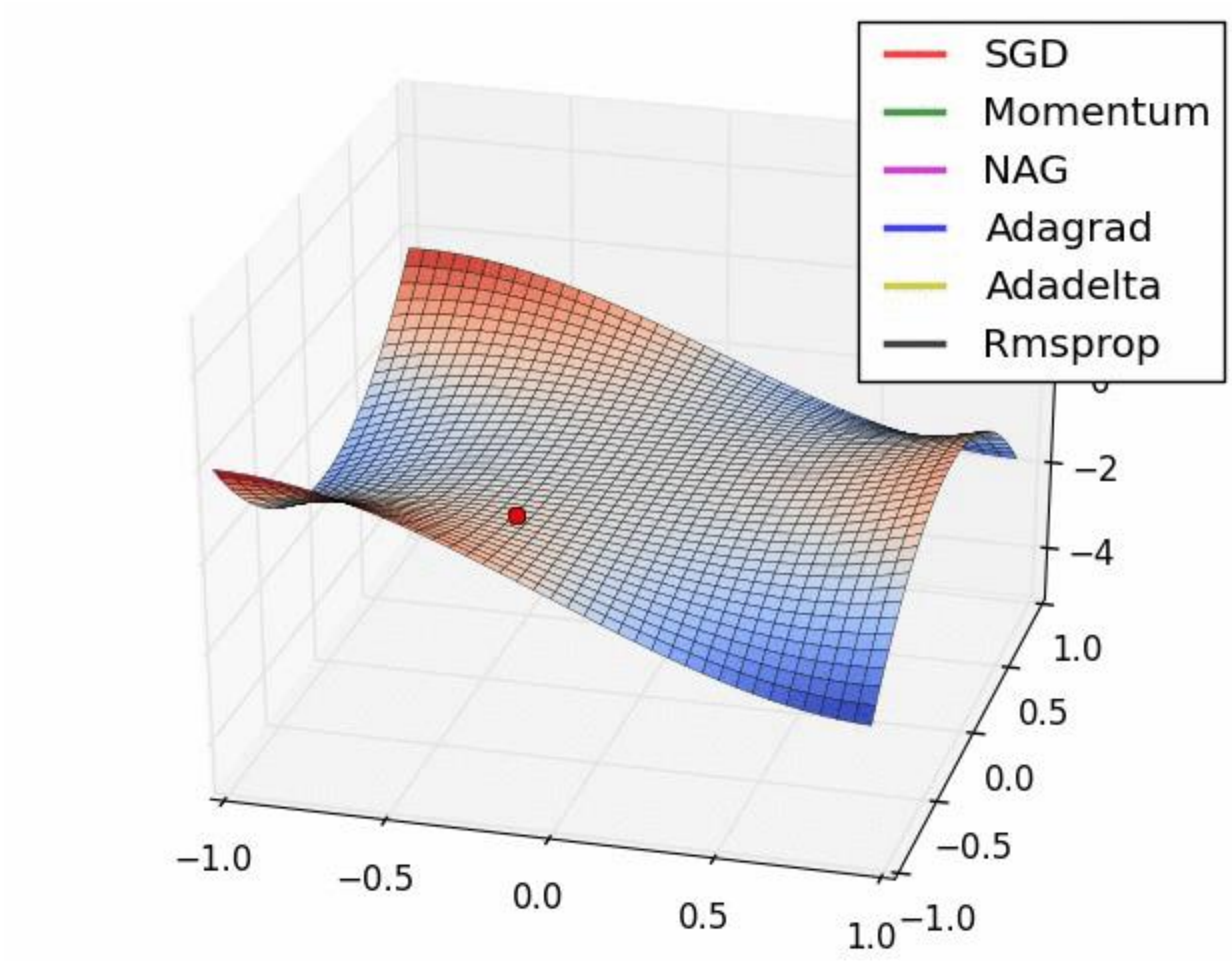


Image from https://imgur.com/a/Hqolp

Improving optimization

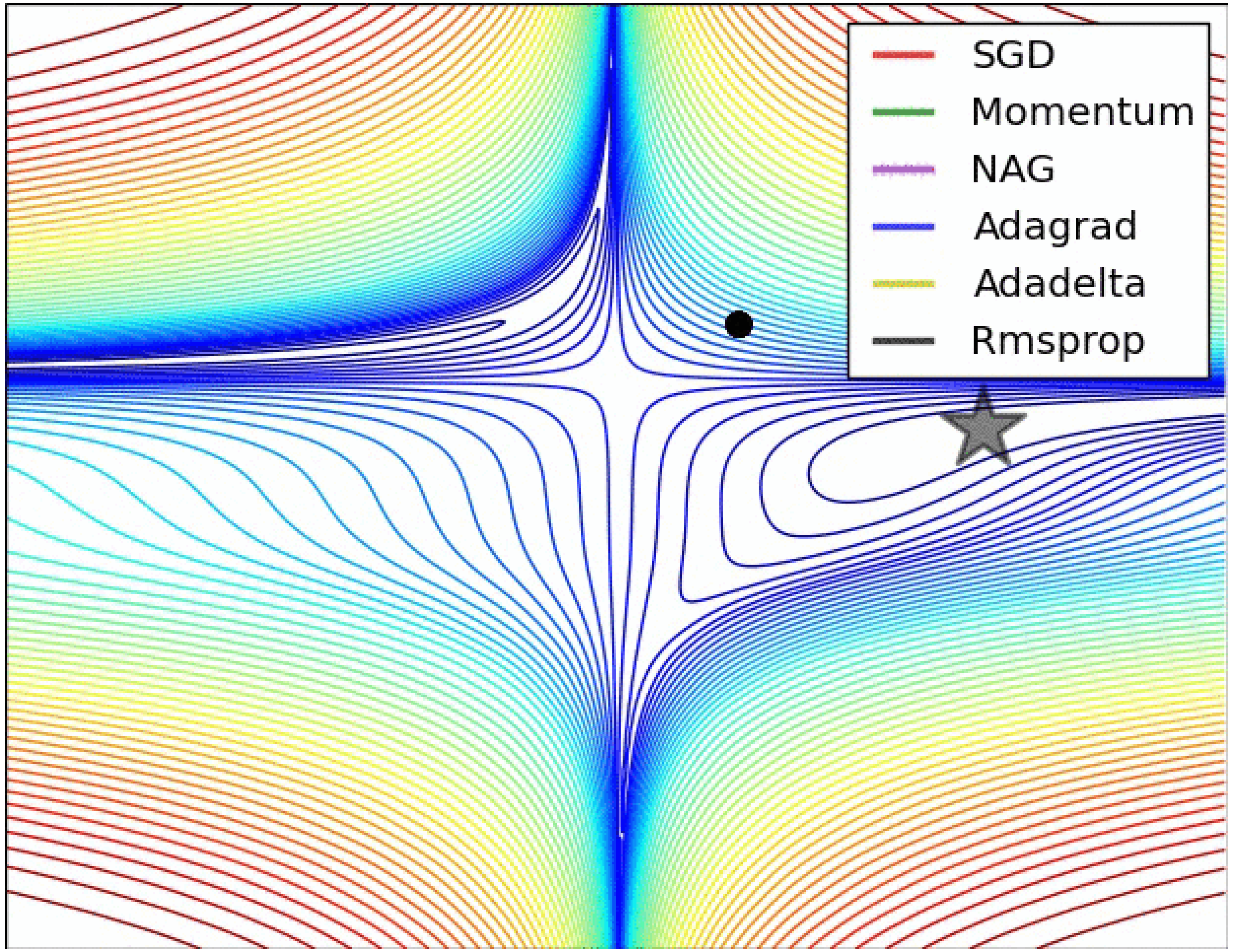


Image from https://imgur.com/a/Hqolp

Adam

- ***Replace components with their EMAs …***

$$m_i^{(t)} := \beta_1(g_i^{(t)}) + (1-\beta_1)m_i^{(t-1)}$$

$$\boldsymbol{m}^{(t)} := \begin{bmatrix} m_1^{(t)} \\ \vdots \\ m_d^{(t)} \end{bmatrix}$$

EMA of the gradient

$$r_i^{(t)} := \beta_2(g_i^{(t)})^2 + (1-\beta_2)r_i^{(t-1)}$$

$$\boldsymbol{r}^{(t)} := \begin{bmatrix} r_1^{(t)} \\ \vdots \\ r_d^{(t)} \end{bmatrix}$$

EMA of the Hessian approximation (vector form)

$$\hat{\boldsymbol{m}}^{(t)} := \frac{\boldsymbol{m}^{(t)}}{1-(1-\beta_1)^t}$$

$$\hat{\boldsymbol{r}}^{(t)} := \frac{\boldsymbol{r}^{(t)}}{1-(1-\beta_2)^t}$$

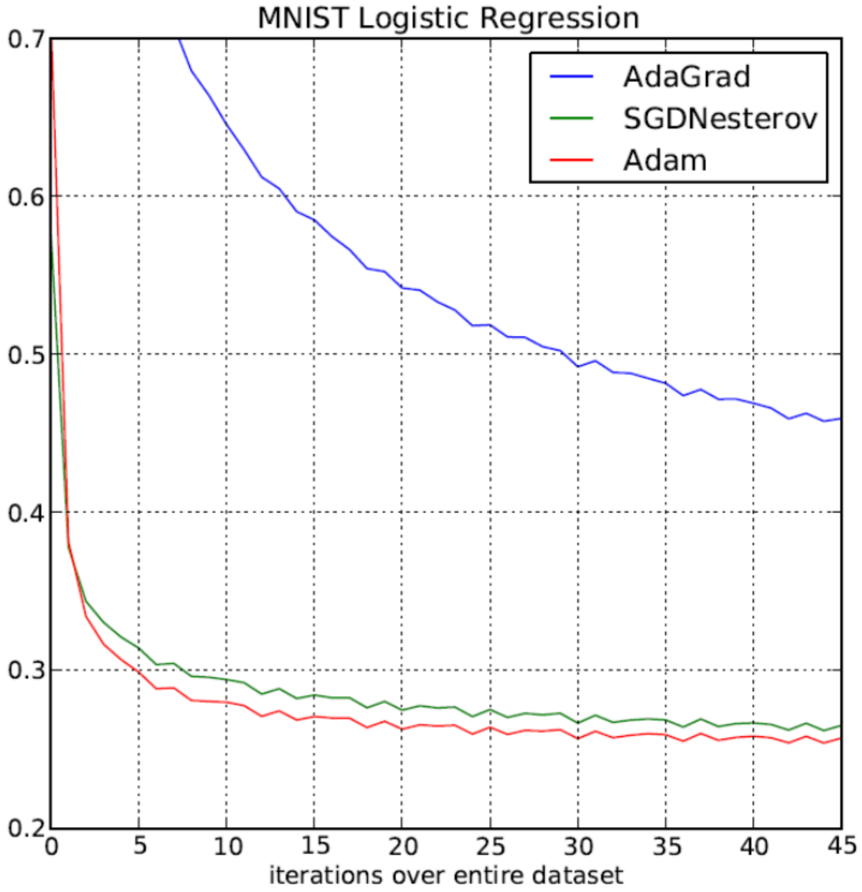
bias corrections (decay with time)

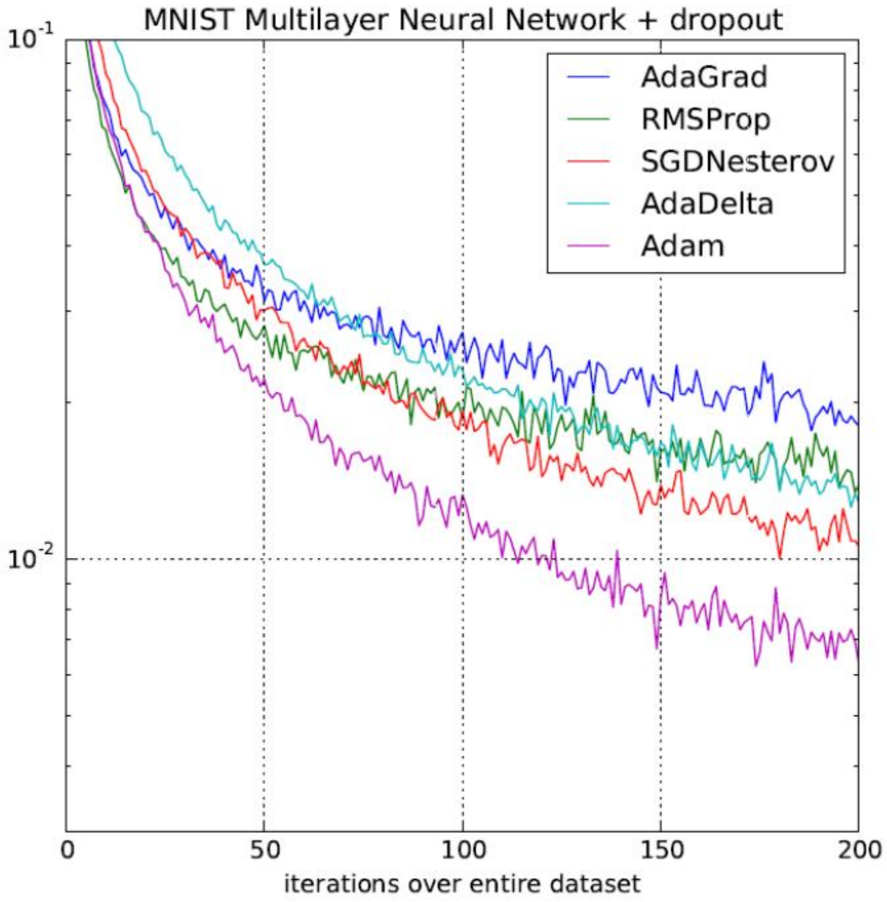
$$\boldsymbol{\vartheta}^{(t)} = \boldsymbol{\vartheta}^{(t-1)} - \eta \, \frac{\hat{\boldsymbol{m}}^{(t-1)}}{\sqrt{\hat{\boldsymbol{r}}^{(t-1)}}}$$

(elementwise)

Adam

- ***Experimentally***





Improving optimization

- ***Messages to take home***

  - Improved optimizers adopt a combination of intuition and mathematical modeling
  - In particular, some of them are *approximators* to 2nd order optimization methods
  - As such, there is no formal guarantee that they will be effective in *all* cases

Moral: in general, their effectiveness will depend on the optimization problem and the representation being used

A bag of wonderful <u>tricks</u>

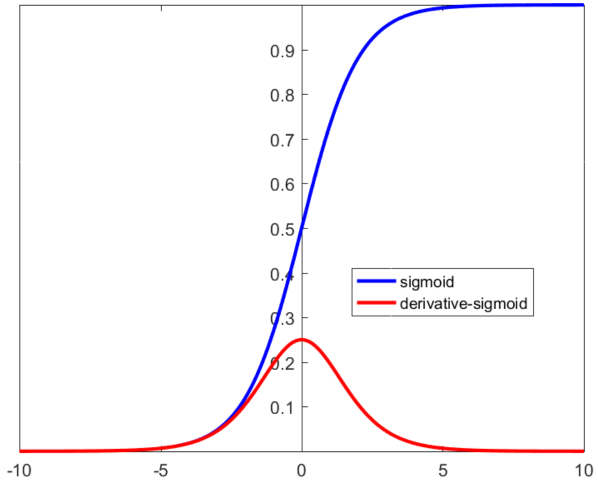
Why ReLU is better (sometimes)

The gradient descent method implies updating the parameters at each step: making sure that the gradient does not either *vanish* or *explode* is not easy

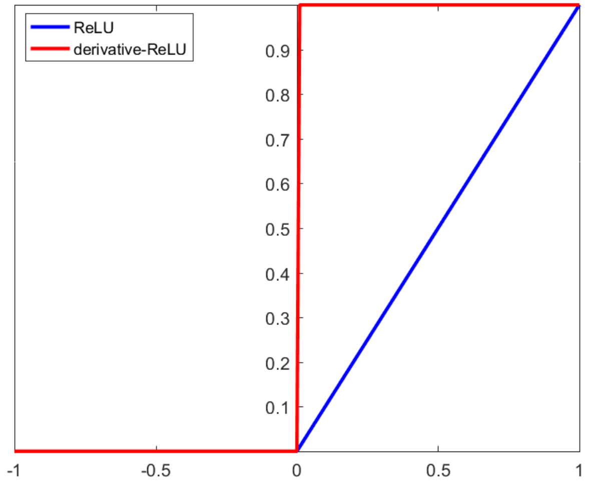
For instance, in

$$\Delta \boldsymbol{W} = -\eta \, \frac{\partial L}{\partial \boldsymbol{W}}(\tilde{y}^{(i)}, y^{(i)})$$

the gradient contains a multiplicative term $\frac{\partial}{\partial x} g(x)$ *which can be* $\ll 1.0$



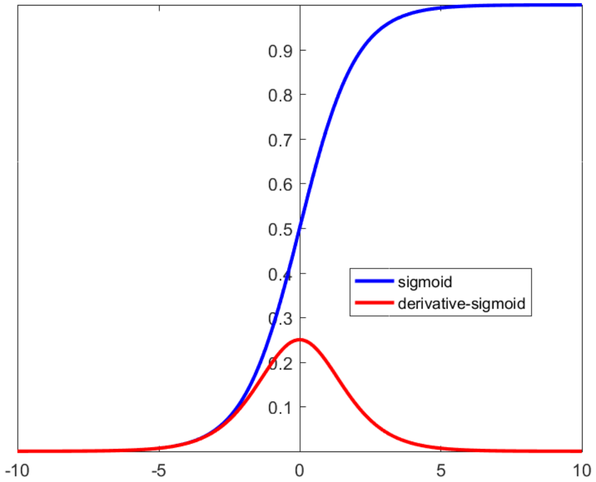
In general, the derivative of ReLU does not suffer from the same problem



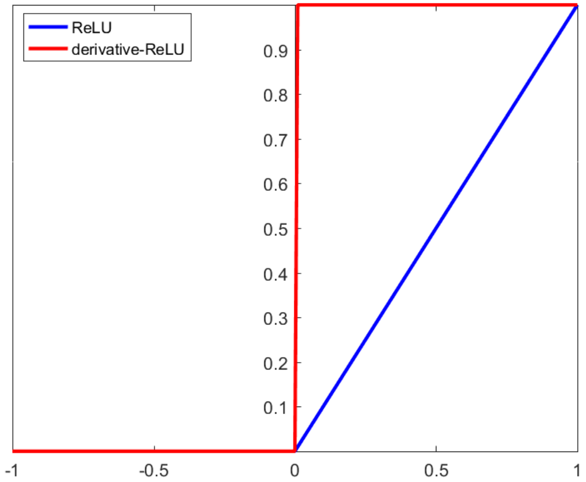
Why ReLU is better (sometimes)

In experimental practice (*sometimes*):

- ReLU alleviates the problem of initial values
  (i.e. when initial values are too far away and cause sigmoid or tanh to saturate)



In general,
the derivative of ReLU
does not suffer
from the same problem



Why ReLU is better (sometimes)

In experimental practice (*sometimes*):

- ReLU alleviates the problem of initial values
  (i.e. when initial values are too far away and cause sigmoid or tanh to saturate)
- ReLU may accelerate the training process

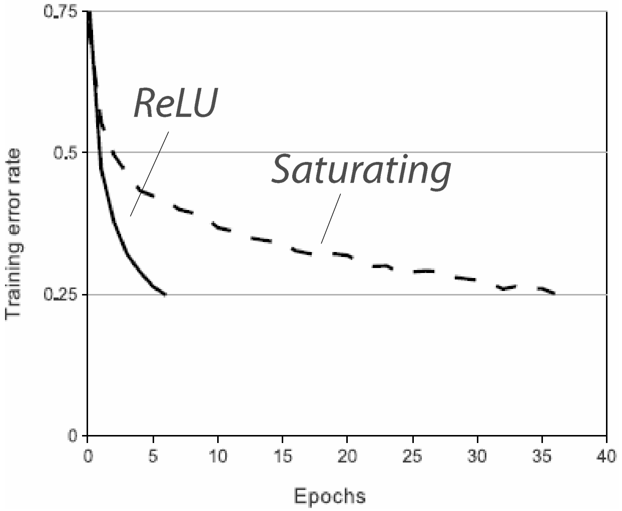
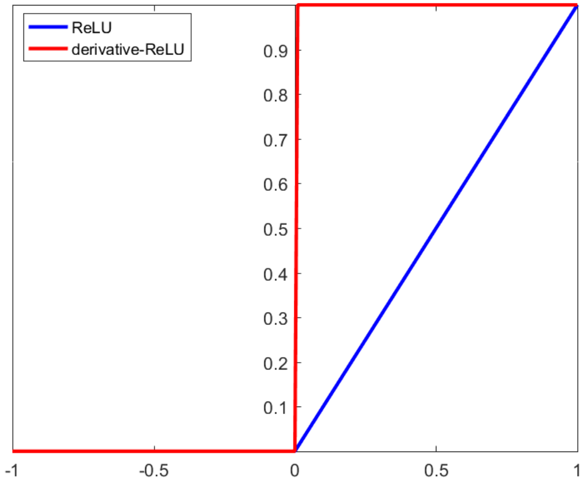


Image from [Krizhevsky, Sutskever & Hinton, 2012]



Input Normalization

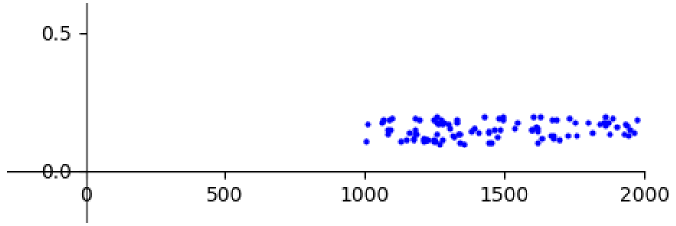
- ***Intuition***

Consider the (very simple) layer

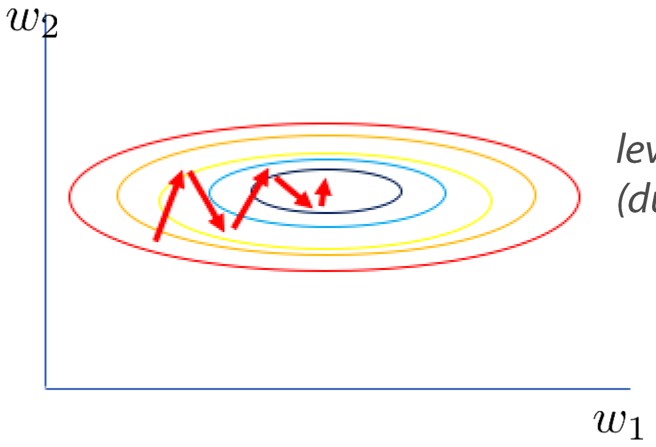
$$h(\boldsymbol{x}) := g(\boldsymbol{w}\boldsymbol{x} + b) = g(w_1 x_1 + w_2 x_2 + b)$$

and suppose $x_1 \in [1000, 2000],\ x_2 \in [0.1, 0.2]$

x_1 and x_2 are in completely different scales



- w_1 influences h a lot more than w_2
- training w_2 is challenging and slow



level curves of loss function (during training)

Image from https://https://www.jeremyjordan.me/batch-normalization/

Input Normalization

- ***Input normalization***

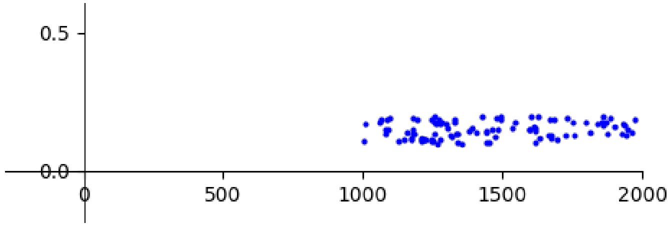
1) compute **mean $\boldsymbol{\mu}$** and (*component-wise*) **variance $\boldsymbol{\sigma}^2$** of inputs over dataset D

$$\boldsymbol{\mu} := \frac{1}{|D|}\sum_{\boldsymbol{x}\in D}\boldsymbol{x} \qquad \boldsymbol{\sigma}^2 := (\sigma_1^2,\ldots,\sigma_d^2,) \quad \text{with } \sigma_i^2 := \frac{1}{|D|}\sum_{\boldsymbol{x}\in D}(x_i-\mu_i)^2$$

2) normalize all inputs, component-wise

$$\hat{\boldsymbol{x}} := (\hat{x}_1,\ldots,\hat{x}_d), \quad \text{with } \hat{x}_i := \frac{x_i-\mu_i}{\sqrt{\sigma_i^2+\epsilon}}$$

to avoid division by zero



Input Normalization

- ***Input normalization***

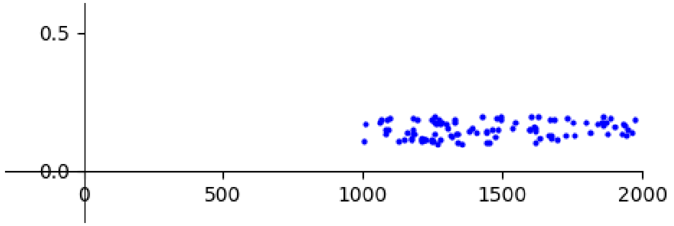
1) compute **mean** $\boldsymbol{\mu}$ and (*component-wise*) **variance** $\boldsymbol{\sigma}^2$ of inputs over dataset D

$$\boldsymbol{\mu} := \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x} \qquad \boldsymbol{\sigma}^2 := (\sigma_1^2, \ldots, \sigma_d^2,) \quad \text{with } \sigma_i^2 := \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} (x_i - \mu_i)^2$$

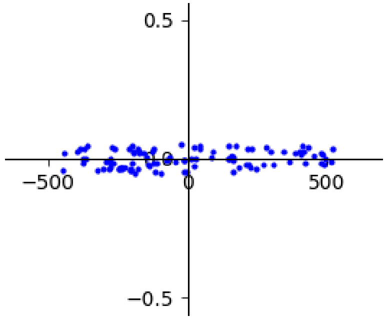
2) normalize all inputs, component-wise

$$\hat{\boldsymbol{x}} := (\hat{x}_1, \ldots, \hat{x}_d), \quad \text{with } \hat{x}_i := \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

to avoid division by zero

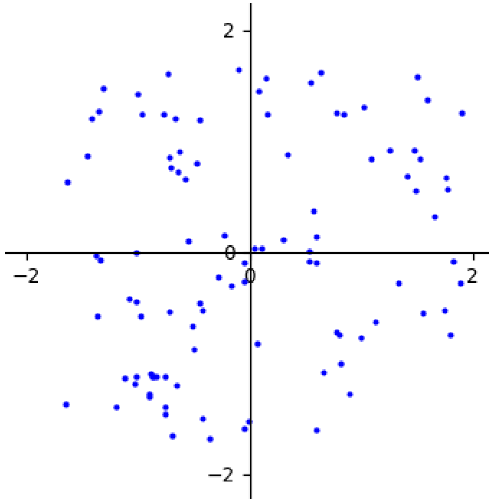


shift by $\boldsymbol{\mu}$



rescale each component by

$$\frac{1}{\sqrt{\sigma_i^2 + \epsilon}}$$



Input Normalization

- ***Input normalization***

1) compute **mean** $\boldsymbol{\mu}$ and (*component-wise*) **variance** $\boldsymbol{\sigma}^2$ of inputs over dataset D

$$\boldsymbol{\mu} := \frac{1}{|D|}\sum_{\boldsymbol{x}\in D}\boldsymbol{x} \qquad \boldsymbol{\sigma}^2 := (\sigma_1^2, \ldots, \sigma_d^2,) \quad \text{with } \sigma_i^2 := \frac{1}{|D|}\sum_{\boldsymbol{x}\in D}(x_i - \mu_i)^2$$

2) normalize all inputs, component-wise

$$\hat{\boldsymbol{x}} := (\hat{x}_1, \ldots, \hat{x}_d), \quad \text{with } \hat{x}_i := \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

3) apply $h(\hat{\boldsymbol{x}}) := g(\boldsymbol{w}\hat{\boldsymbol{x}} + b) = g(w_1\hat{x}_1 + w_2\hat{x}_2 + b)$

Input Normalization

- ***Input normalization***

1) compute **mean $\boldsymbol{\mu}$** and (*component-wise*) **variance $\boldsymbol{\sigma}^2$** of inputs over dataset D

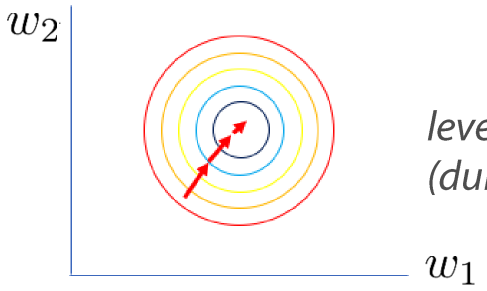
$$\boldsymbol{\mu} := \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x} \qquad \boldsymbol{\sigma}^2 := (\sigma_1^2, \ldots, \sigma_d^2,) \quad \text{with } \sigma_i^2 := \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} (x_i - \mu_i)^2$$

2) normalize all inputs, component-wise

$$\hat{\boldsymbol{x}} := (\hat{x}_1, \ldots, \hat{x}_d), \quad \text{with } \hat{x}_i := \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

3) apply $h(\hat{\boldsymbol{x}}) := g(\boldsymbol{w}\hat{\boldsymbol{x}} + b) = g(w_1 \hat{x}_1 + w_2 \hat{x}_2 + b)$

- training becomes
  <u>*faster*</u> and <u>*more stable*</u>
  (also allowing higher learning rates)



level curves of the loss function (during training)

Image from https://https://www.jeremyjordan.me/batch-normalization/

Batch Normalization

- ***Normalizing in between layers***

In a *DNN*

$$\tilde{\boldsymbol{y}} = \boldsymbol{h}^{[n]}(\boldsymbol{h}^{[n-1]}(\ldots(\boldsymbol{h}^{[2]}(\boldsymbol{h}^{[1]}(\boldsymbol{x})))\ldots))$$

<u>*each layer*</u> $\boldsymbol{h}^{[i]}$ has an input of its own, which should be *normalized*

How?

Batch Normalization

- ***Normalizing in between layers***

In a *DNN*

$$\tilde{\boldsymbol{y}} = \boldsymbol{h}^{[n]}(\boldsymbol{h}^{[n-1]}(\dots(\boldsymbol{h}^{[2]}(\boldsymbol{h}^{[1]}(\boldsymbol{x})))\dots))$$

each layer $\boldsymbol{h}^{[i]}$ has an input of its own, which should be *normalized*

Normalizing in between layers during training would require:

- pre-computing the input to each layer, for *each data item* in D
- applying normalization before proceeding further upwards
- doing it again after *each* updating the DNN parameters

Moral: *it's impossible*

Batch Normalization

- ***For each <u>mini-batch</u>:***

$$B = \left\{ \boldsymbol{x}^{(i)} \right\}_{i=1}^{m}$$

(all operations are performed element-wise)

$$\mathrm{BN}_{\boldsymbol{\beta},\boldsymbol{\gamma}}(\boldsymbol{x}^{(i)}) := \boldsymbol{\gamma}\hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}$$

trainable parameters

$$\hat{\boldsymbol{x}}^{(i)} = \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

avoid division by zero

$$\boldsymbol{\sigma}_B^2 = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_B \right)$$

$$\boldsymbol{\mu}_B = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}^{(i)}$$

Batch Normalization

▪ ***Training***

- at step t: $\boldsymbol{\mu}_{B^{(t)}}$ and $\boldsymbol{\sigma}^2_{B^{(t)}}$ are computed over the <u>*current*</u> mini-batch $B^{(t)}$
- parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ (for each BN-layer) are trained
  in the same way as the other parameters in the DNN
- *exponential moving averages* of mean and variance of the mini-batches $B^{(t)}$ are <u>*collected*</u>

$$\mathrm{MA}(\boldsymbol{\mu})^{(t)} := \delta \cdot \boldsymbol{\mu}_{B^{(t)}} + (1-\delta) \cdot \mathrm{MA}(\boldsymbol{\mu})^{(t-1)}, \qquad \mathrm{MA}(\boldsymbol{\mu})^{(1)} := \boldsymbol{\mu}_{B^{(1)}}$$
$$\mathrm{MA}(\boldsymbol{\sigma}^2)^{(t)} := \delta \cdot \boldsymbol{\sigma}^2_{B^{(t)}} + (1-\delta) \cdot \mathrm{MA}(\boldsymbol{\sigma}^2)^{(t-1)}, \qquad \mathrm{MA}(\boldsymbol{\sigma}^2)^{(1)} := \boldsymbol{\sigma}^2_{B^{(1)}}$$

▪ ***Inference***

Inference is typically performed for fewer inputs, possibly just one ...

Batch Normalization

- ***Training***
  - at step t: $\boldsymbol{\mu}_{B^{(t)}}$ and $\boldsymbol{\sigma}^2_{B^{(t)}}$ are computed over the *current* mini-batch $B^{(t)}$
  - parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ (for each BN-layer) are trained
    in the same way as the other parameters in the DNN
  - *exponential moving averages* of mean and variance of the mini-batches $B^{(t)}$ are *collected*

$$\begin{aligned} \mathrm{MA}(\boldsymbol{\mu})^{(t)} &:= \delta \cdot \boldsymbol{\mu}_{B^{(t)}} + (1-\delta) \cdot \mathrm{MA}(\boldsymbol{\mu})^{(t-1)}, \qquad & \mathrm{MA}(\boldsymbol{\mu})^{(1)} &:= \boldsymbol{\mu}_{B^{(1)}} \\ \mathrm{MA}(\boldsymbol{\sigma}^2)^{(t)} &:= \delta \cdot \boldsymbol{\sigma}^2_{B^{(t)}} + (1-\delta) \cdot \mathrm{MA}(\boldsymbol{\sigma}^2)^{(t-1)}, \qquad & \mathrm{MA}(\boldsymbol{\sigma}^2)^{(1)} &:= \boldsymbol{\sigma}^2_{B^{(1)}} \end{aligned}$$

- ***Inference***

  Normalize using the moving averages collected *during training*

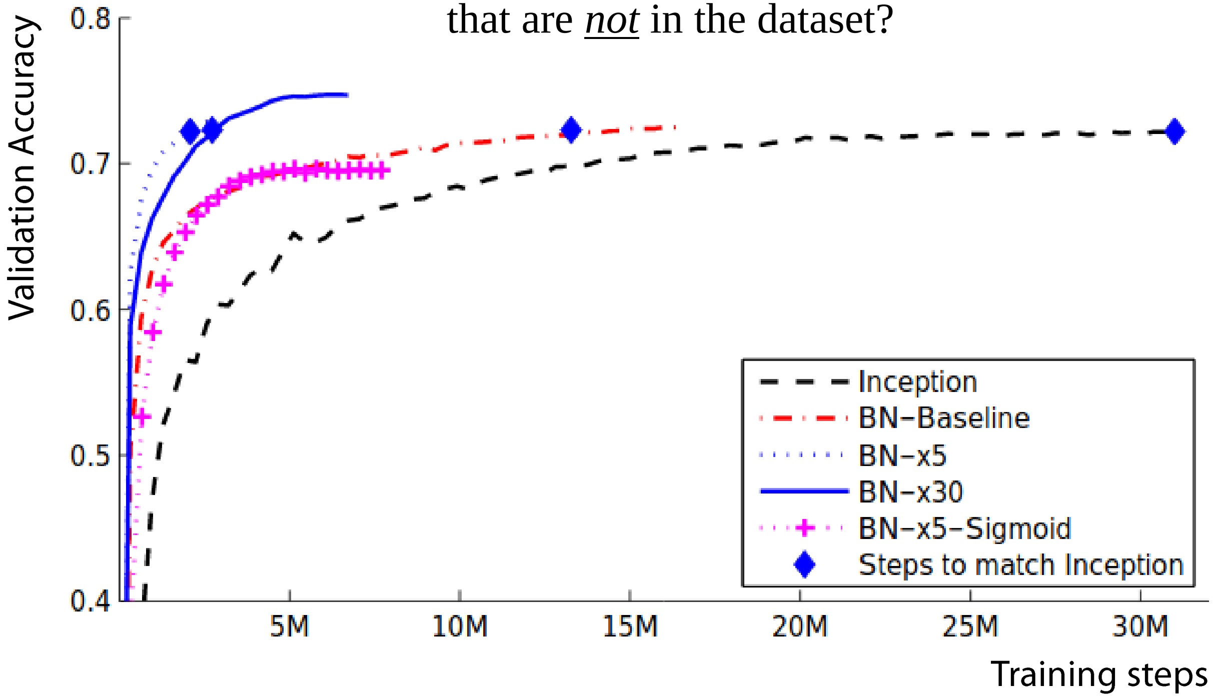
  - $\boldsymbol{\mu} := \mathrm{MA}(\boldsymbol{\mu})^{(T)}$
  - $\boldsymbol{\sigma}^2 := \mathrm{MA}(\boldsymbol{\sigma}^2)^{(T)}$

  as collected during the training process

Batch Normalization

- ***Does it work?***

How good is the approximator when applied to data items that are *not* in the dataset?



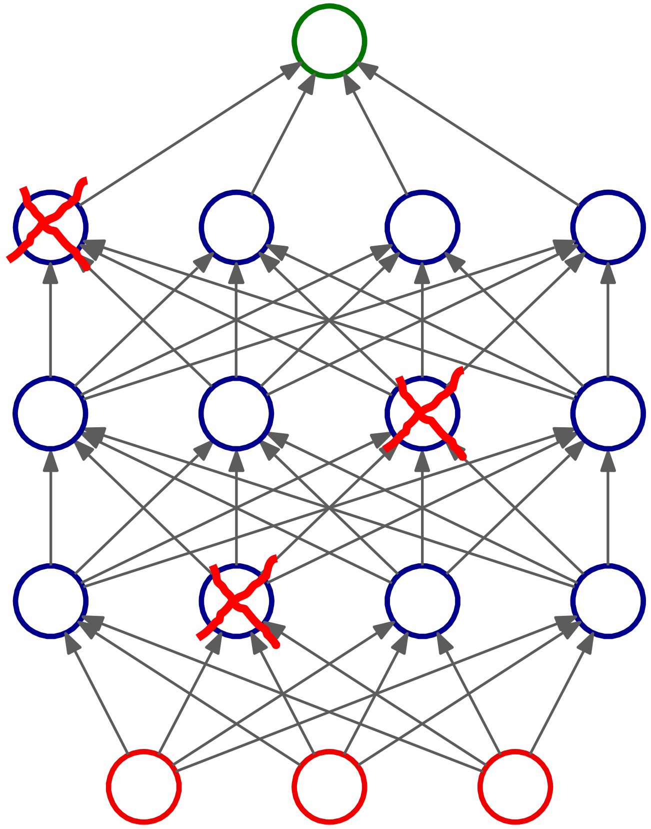
- Batch normalization acts as a *reparametrization* of the optimization process that
  1. makes the loss function smoother
  2. allows higher learning rates
  3. reduces chances to getting stuck into local minima

Image from [Ioffe and Szegedy 2015]

Dropout

- **Knocking-out at random**

  For each mini-batch, a small percentage of 'units' is de-activated

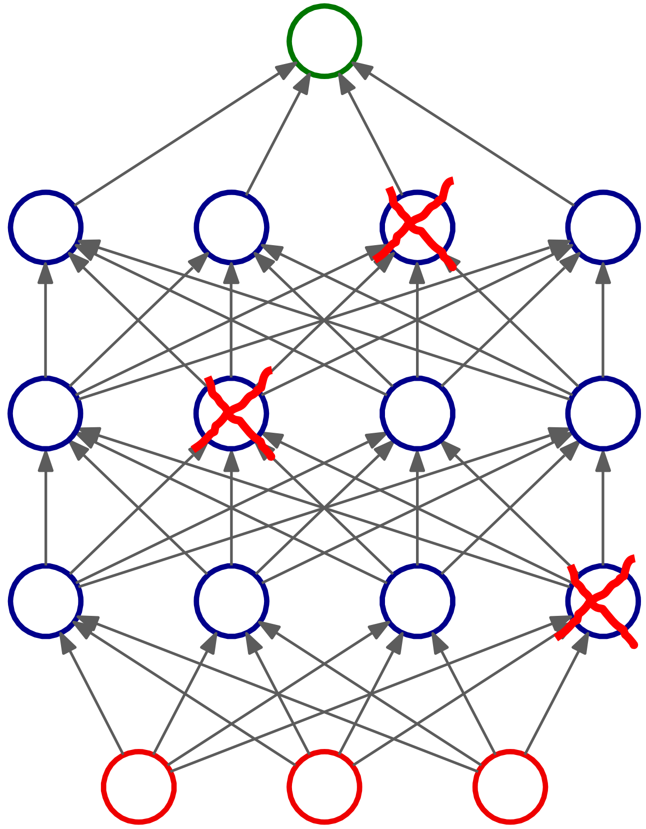


Training: mini-batch 1

Dropout

- **Knocking-out at random**

  For each mini-batch, a small percentage of 'units' is de-activated

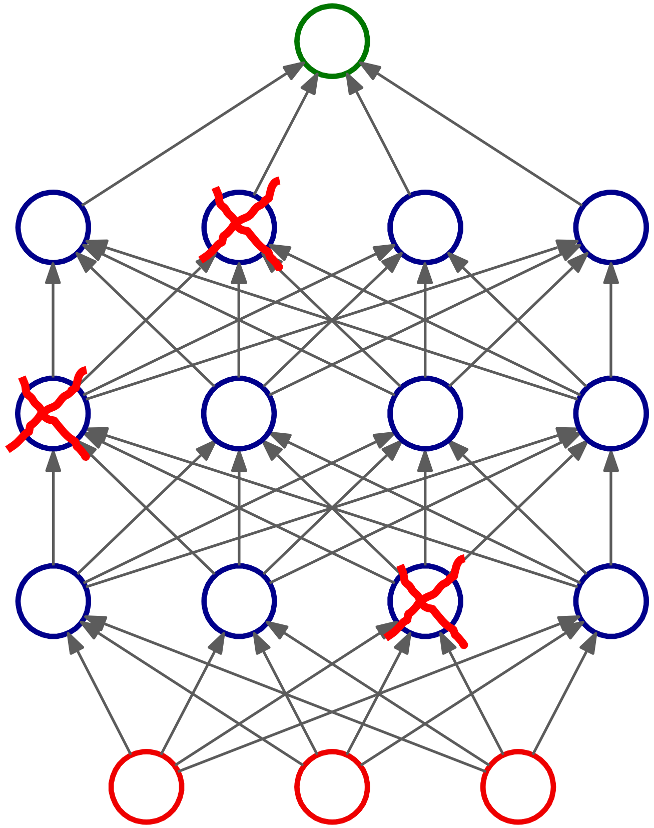


Training: mini-batch 2

Dropout

- **Knocking-out at random**

  For each mini-batch, a small percentage of 'units' is de-activated

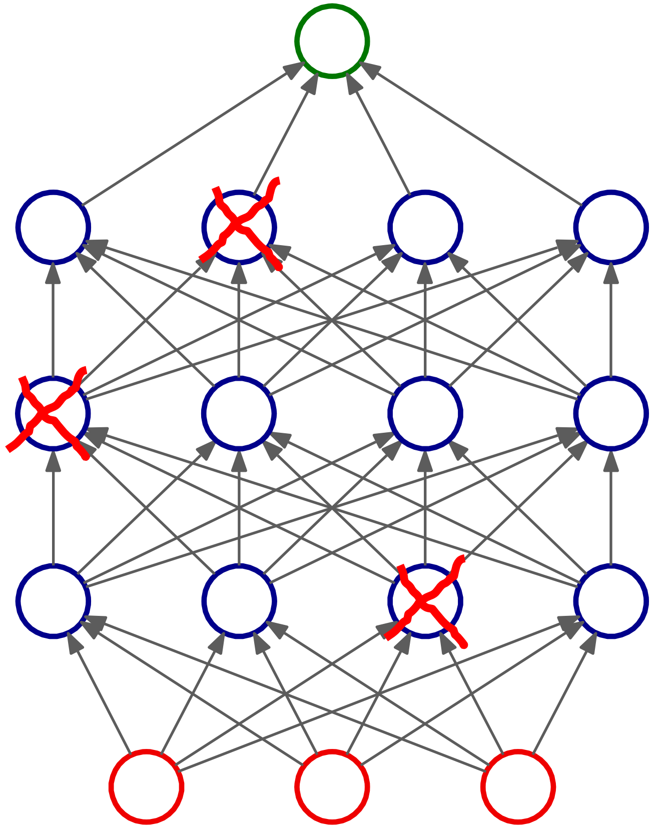


Training: mini-batch 3

Dropout

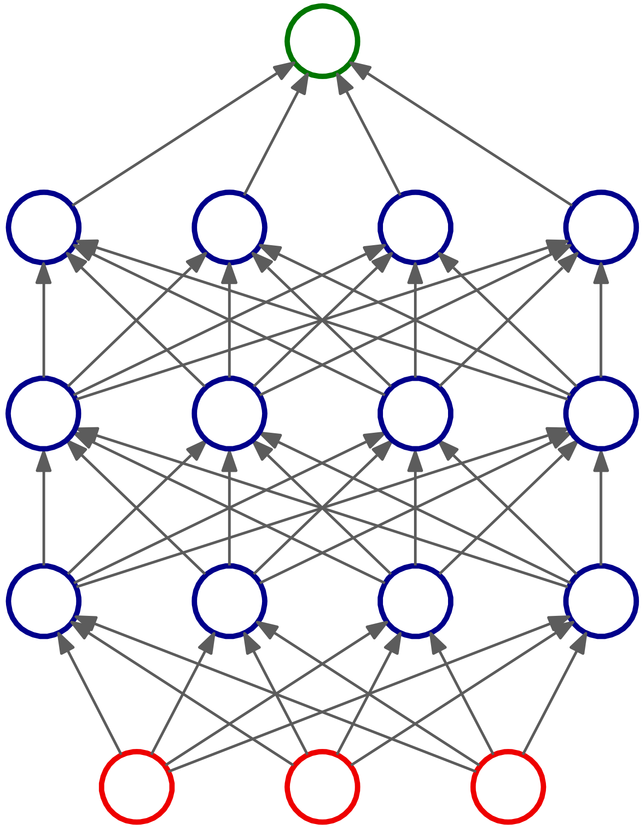
- **Knocking-out at random**

  For each mini-batch, a small percentage of 'units' is de-activated



Training

At runtime
(or validation time),
when making predictions,
dropout is not active



Prediction

Contrasting Overfitting

- ***Applying Dropout***

  In a typical experiment

  - initially, the performance on D_{val} improves slowly
  - then it becomes better and more resilient to *overfitting (to be explained next)*

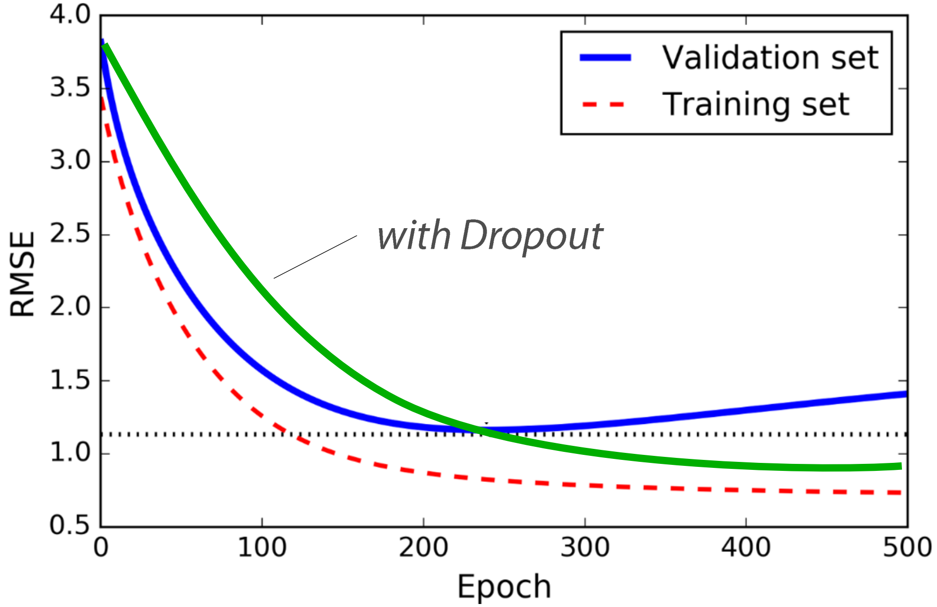


Image from https://www.safaribooksonline.com/library/view/hands-on-machine-learning/9781491962282/ch04.html