MASTER OF SCIENCE
IN ENGINEERING

Faster Optimisers

TSM_DeepLearn

Andreas Fischer
Klaus Zahn

We are grateful to J. Hennebert and M. Melchior, that they provided their slides

Plan for this week

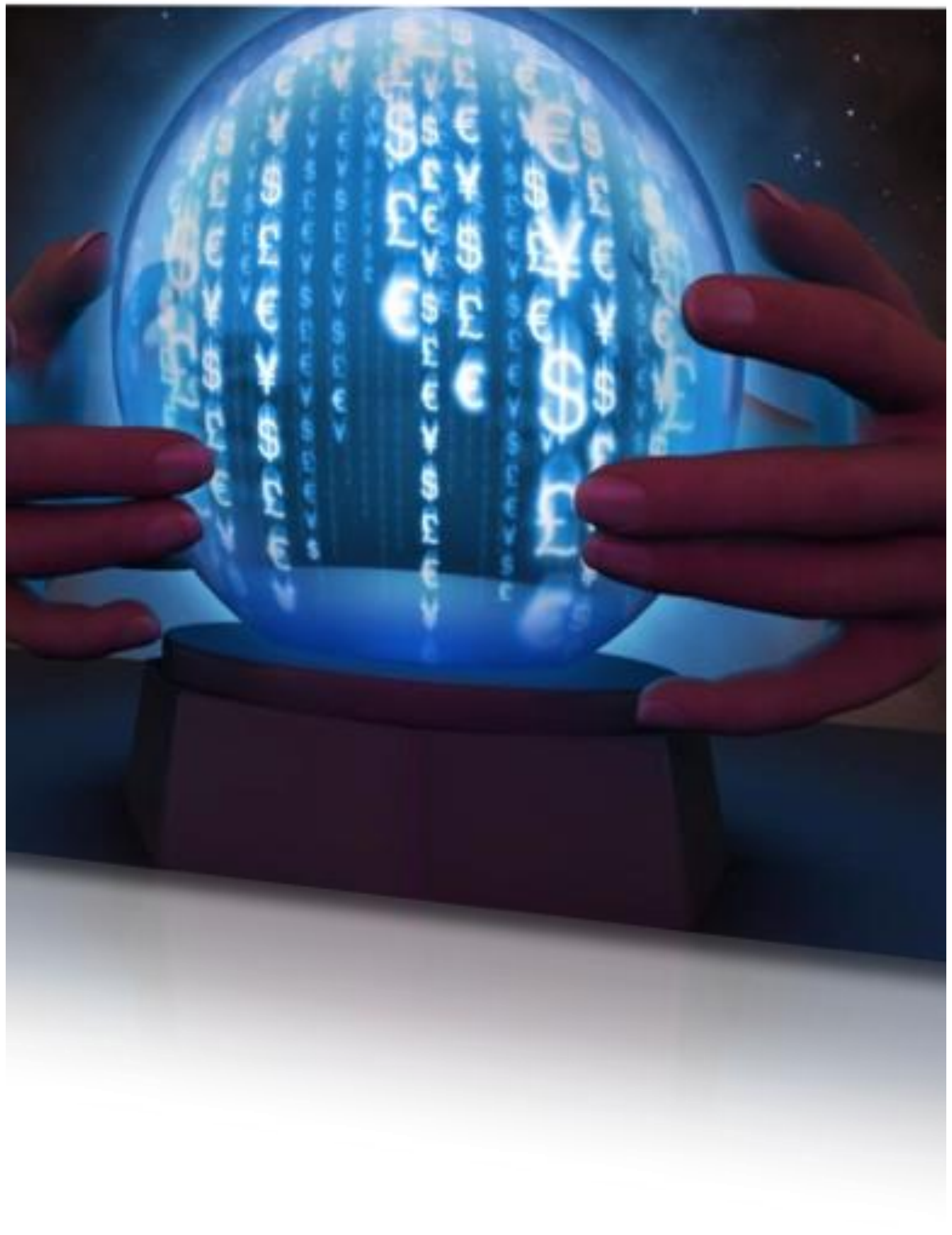
Explore and understand some of the problems observed while training deep neural nets:

- Finish understanding of Back Propgation
- Vanishing/exploding gradients
- Insufficient performance of the optimization schemes

Learn some of the tools to counter these problems

Recap from Last Week

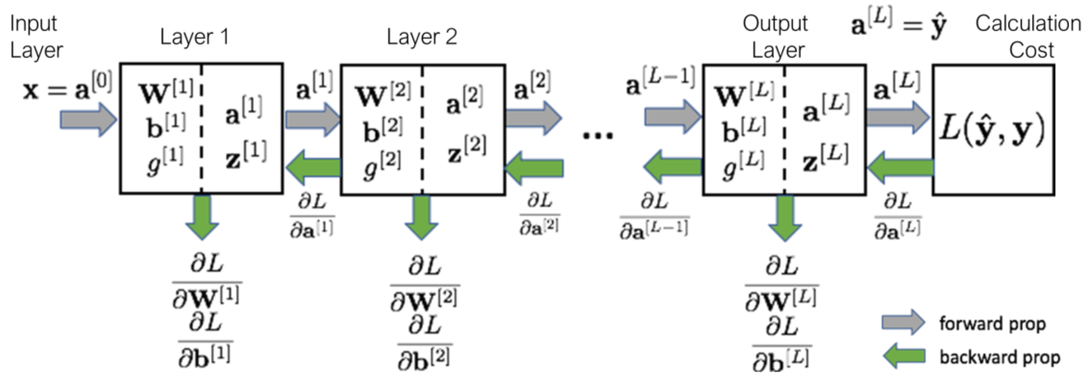
Backpropagation through
a single layer



Backprop through a Single Layer

(matrix notation)

normal
MLP layer
(not Softmax)


Input Layer
$\mathbf{x} = \mathbf{a}^{[0]}$
Layer 1
Layer 2
Output Layer
$\mathbf{a}^{[L]} = \hat{\mathbf{y}}$
Calculation Cost
$L(\hat{\mathbf{y}}, \mathbf{y})$
forward prop
backward prop

$$\frac{\partial L}{\partial \mathbf{z}^{[l]}} = \frac{\partial L}{\partial \mathbf{a}^{[l]}} * \frac{dg^{[l]}(\mathbf{z}^{[l]})}{dz}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \cdot \left(\mathbf{a}^{[l-1]}\right)^T$$

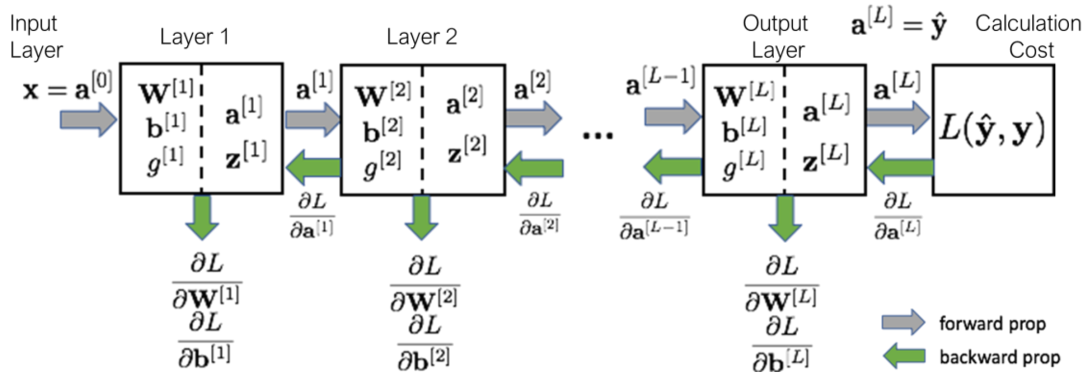
$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}}$$

$$\frac{\partial L}{\partial \mathbf{a}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^T \cdot \frac{\partial L}{\partial \mathbf{z}^{[l]}}$$

- $*$ is an *elementwise* multiplication of $\frac{\partial L}{\partial \mathbf{a}^{[l]}}$ and $\frac{\partial g^{[l]}(\mathbf{z}^{[l]})}{\partial z}$ both vectors of size n^l x 1
- Product between $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ and the transpose of $\mathbf{a}^{[l-1]}$ ($(..)^T$) is matrix product of $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ of size n^l x 1 and the transpose of $\mathbf{a}^{[l-1]}$ of size 1 x n^{l-1}
- The product of the transpose of $\mathbf{W}^{[l]}$ (dimension n^{l-1} x n^l) and $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ (dimension n^l x 1) is a matrix product.

Backprop through a Single Layer

(matrix notation)


Input Layer
$\mathbf{x} = \mathbf{a}^{[0]}$
Layer 1
$\mathbf{W}^{[1]}$ $\mathbf{b}^{[1]}$ $g^{[1]}$ $\mathbf{a}^{[1]}$ $\mathbf{z}^{[1]}$
Layer 2
$\mathbf{W}^{[2]}$ $\mathbf{b}^{[2]}$ $g^{[2]}$ $\mathbf{a}^{[2]}$ $\mathbf{z}^{[2]}$
Output Layer
$\mathbf{W}^{[L]}$ $\mathbf{b}^{[L]}$ $g^{[L]}$ $\mathbf{a}^{[L]}$ $\mathbf{z}^{[L]}$
$\mathbf{a}^{[L]} = \hat{\mathbf{y}}$
Calculation Cost
$L(\hat{\mathbf{y}}, \mathbf{y})$
$\frac{\partial L}{\partial \mathbf{a}^{[1]}}$ $\frac{\partial L}{\partial \mathbf{a}^{[2]}}$ $\frac{\partial L}{\partial \mathbf{a}^{[L-1]}}$ $\frac{\partial L}{\partial \mathbf{a}^{[L]}}$
$\frac{\partial L}{\partial \mathbf{W}^{[1]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[1]}}$ $\frac{\partial L}{\partial \mathbf{W}^{[2]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[2]}}$ $\frac{\partial L}{\partial \mathbf{W}^{[L]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[L]}}$
forward prop
backward prop

Softmax with
CE cost

$$\frac{\partial L}{\partial \mathbf{z}^{[l]}} = \hat{\mathbf{y}} - \mathbf{y}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \cdot \left(\mathbf{a}^{[l-1]}\right)^T$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}}$$

$$\frac{\partial L}{\partial \mathbf{a}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^T \cdot \frac{\partial L}{\partial \mathbf{z}^{[l]}}$$

- prediction $\hat{\mathbf{y}}$ and the true label $\mathbf{y}$ are of size n^L x 1. True label $\mathbf{y}$ is a one hot vector.
- Product between $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ and the transpose of $\mathbf{a}^{[l-1]}$ $((..)^T)$ is matrix product of $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ of size n^l x 1 and the transpose of $\mathbf{a}^{[l-1]}$ of size 1 x n^{l-1}
- The product of the transpose of $\mathbf{W}^{[l]}$ (dimension n^{l-1} x n^l) and $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ (dimension n^l x 1) is a matrix product.

Backprop through a Single Layer – for Batch

(matrix notation)

Cost now average over m samples of the mini-batch:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\right)$$

Equations for the forward propagation simply parallel processing of matrices for activations and logits where each column is associated with the corresponding input data sample:

$$\mathbf{A}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{a}^{[l](1)} & \mathbf{a}^{[l](2)} & \dots & \mathbf{a}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad \mathbf{Z}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)} & \mathbf{z}^{[l](2)} & \dots & \mathbf{z}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

($n_l \times m$ matrices)

$$\begin{aligned} \mathbf{Z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{A}^{[l]} &= g^{[l]}(\mathbf{Z}^{[l]}) \end{aligned}$$

The backpropagation equations are expressed similarly only average over samples must be done explicitly for $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ and $\frac{\partial L}{\partial \mathbf{b}^{[l]}}$.

Backprop through a Single Layer – for Batch

(matrix notation)

We now introduce the derivative $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ of the cost function L with respect to the activation matrix $\mathbf{A}^{[l]}$.

$$\frac{\partial L}{\partial \mathbf{A}^{[l]}} = \begin{pmatrix} \vdots & & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{a}^{[l](1)}} & \cdots & \boxed{\frac{\partial L}{\partial \mathbf{a}^{[l](2)}}} & \cdots & \frac{\partial L}{\partial \mathbf{a}^{[l](m)}} \\ \vdots & & \vdots & & \vdots \end{pmatrix} \qquad (n_l \times m \text{ natrices})$$

partial derivatives of L w.r.t. to the activations in the l-th layer, evaluated for the i-th sample

Backprop through a Single Layer – for Batch

(matrix notation)

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{a}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{a}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{a}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} * \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{dg^{[l]}\left(\mathbf{z}^{[l](1)}\right)}{dz} & \frac{dg^{[l]}\left(\mathbf{z}^{[l](2)}\right)}{dz} & \cdots & \frac{dg^{[l]}\left(\mathbf{z}^{[l](m)}\right)}{dz} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \cdot \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} \cdot \begin{pmatrix} \cdots & \mathbf{a}^{[l-1](1)} & \cdots \\ \cdots & \mathbf{a}^{[l-1](2)} & \cdots \\ \vdots & & \\ \cdots & \mathbf{a}^{[l-1](m)} & \cdots \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \cdot \text{sum}\left(\frac{\partial L}{\partial \mathbf{Z}^{[l]}}, \text{axis} = 1\right) = \frac{1}{m} \cdot \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} \cdot \begin{pmatrix} \vdots \\ 1 \\ \vdots \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \begin{pmatrix} \cdots & \mathbf{w}^{[l](1)} & \cdots \\ \cdots & \mathbf{w}^{[l](2)} & \cdots \\ \vdots & & \\ \cdots & \mathbf{w}^{[l](m)} & \cdots \end{pmatrix} \cdot \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

Backprop through a Single Layer

(matrix notation)

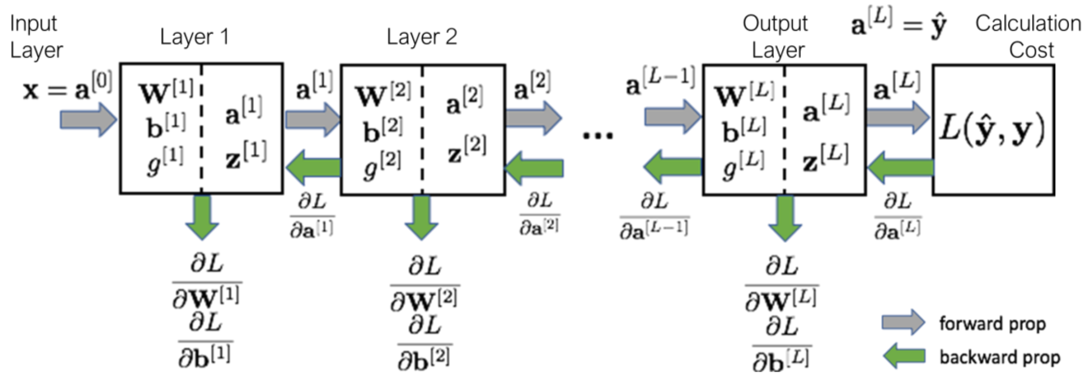
normal
MLP layer
(not Softmax)

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}} = \frac{\partial L}{\partial \mathbf{A}^{[l]}} * \frac{dg^{[l]}(\mathbf{Z}^{[l]})}{dz}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \left(\mathbf{A}^{[l-1]}\right)^{T}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^{T} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}}$$


Input Layer
$\mathbf{x} = \mathbf{a}^{[0]}$
Layer 1
$\mathbf{W}^{[1]}$ $\mathbf{b}^{[1]}$ $g^{[1]}$ $\mathbf{a}^{[1]}$ $\mathbf{z}^{[1]}$
$\mathbf{a}^{[1]}$
$\frac{\partial L}{\partial \mathbf{a}^{[1]}}$
$\frac{\partial L}{\partial \mathbf{W}^{[1]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[1]}}$
Layer 2
$\mathbf{W}^{[2]}$ $\mathbf{b}^{[2]}$ $g^{[2]}$ $\mathbf{a}^{[2]}$ $\mathbf{z}^{[2]}$
$\mathbf{a}^{[2]}$
$\frac{\partial L}{\partial \mathbf{a}^{[2]}}$
$\frac{\partial L}{\partial \mathbf{W}^{[2]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[2]}}$
...
$\mathbf{a}^{[L-1]}$
$\frac{\partial L}{\partial \mathbf{a}^{[L-1]}}$
Output Layer
$\mathbf{W}^{[L]}$ $\mathbf{b}^{[L]}$ $g^{[L]}$ $\mathbf{a}^{[L]}$ $\mathbf{z}^{[L]}$
$\frac{\partial L}{\partial \mathbf{W}^{[L]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[L]}}$
$\mathbf{a}^{[L]} = \hat{\mathbf{y}}$
$\mathbf{a}^{[L]}$
$\frac{\partial L}{\partial \mathbf{a}^{[L]}}$
Calculation Cost
$L(\hat{\mathbf{y}}, \mathbf{y})$
forward prop
backward prop

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}}, \frac{\partial L}{\partial \mathbf{A}^{[l]}}, \frac{dg^{[l]}(\mathbf{Z}^{[l]})}{dz}: \; n_l \text{ x } m$$

$$\mathbf{A}^{[l-1]}, \frac{\partial L}{\partial \mathbf{A}^{[l-1]}}: \; n_{l-1} \text{ x } m$$

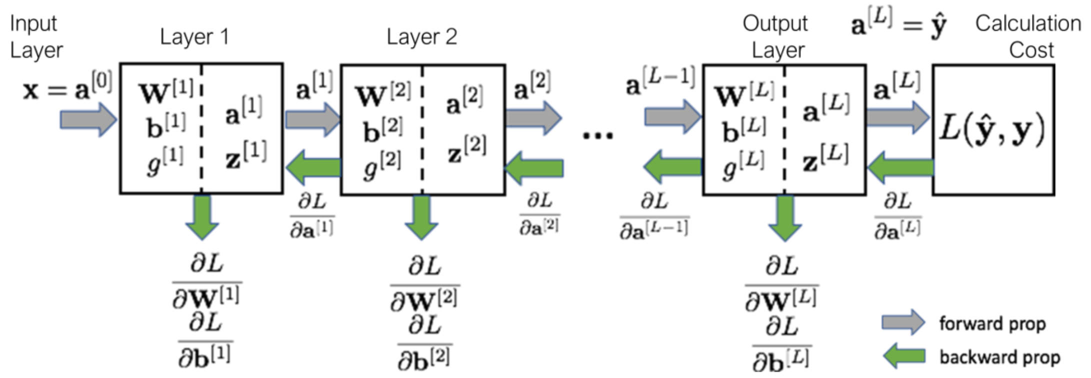
$$\frac{\partial L}{\partial \mathbf{W}^{[l]}}: \; n_l \text{ x } n_{l-1}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}}: \; n_l \text{ x } 1$$

$$\hat{\mathbf{Y}}, \mathbf{Y}: \; n_L \text{ x } m$$

Backprop through a Single Layer

(matrix notation)


Input Layer
$\mathbf{x} = \mathbf{a}^{[0]}$
Layer 1
$\mathbf{W}^{[1]}$ $\mathbf{b}^{[1]}$ $g^{[1]}$ $\mathbf{a}^{[1]}$ $\mathbf{z}^{[1]}$
$\mathbf{a}^{[1]}$
$\frac{\partial L}{\partial \mathbf{a}^{[1]}}$
$\frac{\partial L}{\partial \mathbf{W}^{[1]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[1]}}$
Layer 2
$\mathbf{W}^{[2]}$ $\mathbf{b}^{[2]}$ $g^{[2]}$ $\mathbf{a}^{[2]}$ $\mathbf{z}^{[2]}$
$\mathbf{a}^{[2]}$
$\frac{\partial L}{\partial \mathbf{a}^{[2]}}$
$\frac{\partial L}{\partial \mathbf{W}^{[2]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[2]}}$
...
$\mathbf{a}^{[L-1]}$
$\frac{\partial L}{\partial \mathbf{a}^{[L-1]}}$
Output Layer
$\mathbf{W}^{[L]}$ $\mathbf{b}^{[L]}$ $g^{[L]}$ $\mathbf{a}^{[L]}$ $\mathbf{z}^{[L]}$
$\frac{\partial L}{\partial \mathbf{W}^{[L]}}$ $\frac{\partial L}{\partial \mathbf{b}^{[L]}}$
$\mathbf{a}^{[L]} = \hat{\mathbf{y}}$
$\mathbf{a}^{[L]}$
$\frac{\partial L}{\partial \mathbf{a}^{[L]}}$
Calculation Cost
$L(\hat{\mathbf{y}}, \mathbf{y})$
forward prop
backward prop

Softmax with CE cost

$$\frac{\partial L}{\partial \mathbf{Z}^{[L]}} = \widehat{\mathbf{Y}} - \mathbf{Y}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \left(\mathbf{A}^{[l-1]}\right)^{T}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^{T} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}}$$

$\widehat{\mathbf{Y}}\,, \mathbf{Y}\text{:}\;\; n_L \text{ x } m$

$\mathbf{A}^{[l-1]}\,, \frac{\partial L}{\partial \mathbf{A}^{[l-1]}}\text{:}\;\; n_{l-1} \text{ x } m$

$\frac{\partial L}{\partial \mathbf{W}^{[l]}}\text{:}\;\; n_l \text{ x } n_{l-1}$

$\frac{\partial L}{\partial \mathbf{b}^{[l]}}\text{:}\;\; n_l \text{ x } 1$

Implementation in `numpy`

See PW for week 5

```
class DenseLayer:

    def __init__(self, num_in, num_out):

        self.num_in = num_in
        self.num_out = num_out

        # initialize weights and bias (zero or random)
        self.initialise_weights()

    def propagate(self, a_in):

        self.num_samples = a_in.shape[0] #may change in different steps
        self.a_in = a_in # required for back_prop
        self.Z = ...
        a_out = self.sigmoid(self.Z)

        return a_out

    def back_propagate(self, dL_da_out):

        dL_dz = ...
        dL_da_in = ...
        self.dL_dW = ...
        self.dL_dB = ...

        return dL_da_in
```

cache logits (a^{L-1})

compute gradient w.r.t. logits (z)

compute gradient w.r.t. to weights and bias by using gradient w.r.t. logits (z)

softmax layer needs to be handled specially

```
class MultiLayerPerceptron:

    def __init__(self, list_num_neurons):

        self.layers = []
        #first construct dense layers (if any)
        for i0 in range(len(list_num_neurons)-2):
            ...

    def propagate(self, x):

        for layers in self.layers:
            x = layers.propagate(x)

        return x

    def back_propagate(self, y):

        dL_da_out = self.layers[-1].back_propagate(y)
        for i0 in reversed(range(len(self.layers)-1)):
            dL_da_out = self.layers[i0].back_propagate(dL_da_out)

        return dL_da_out

    def gradient_descend(self, alpha):

        for layers in self.layers:
            layers.gradient_descend(alpha)
```

(mini)-batches of images

call forward pass of individual layes

call backprop of individual layers

parameter update using gradient descent

```
    def optimise(self, data, epochs, alpha, batch_size, debug=False):
        ...

        for i0 in range(0, epochs):
            #create batches for each epoch
            batches = MiniBatches(data['x_train'], data['y_train']
                                                  , batch_size)

            for ib in range(batches.number_of_batches()):
                self.batch = batches.next()
                x = self.batch['x_batch']
                y = self.batch['y_batch']
                self.propagate(x)
                self.back_propagate(y)
                self.gradient_descend(alpha)
```

main loop of gradient descent optimisation of MLP

Numerical Gradient Checking

For debugging, it is helpful to check numerically whether the calculation of the gradient of the loss function w.r.t. the weights and the biases is correct.

$$L = L(\ldots, W_{kl}^{[l]}, \ldots)$$

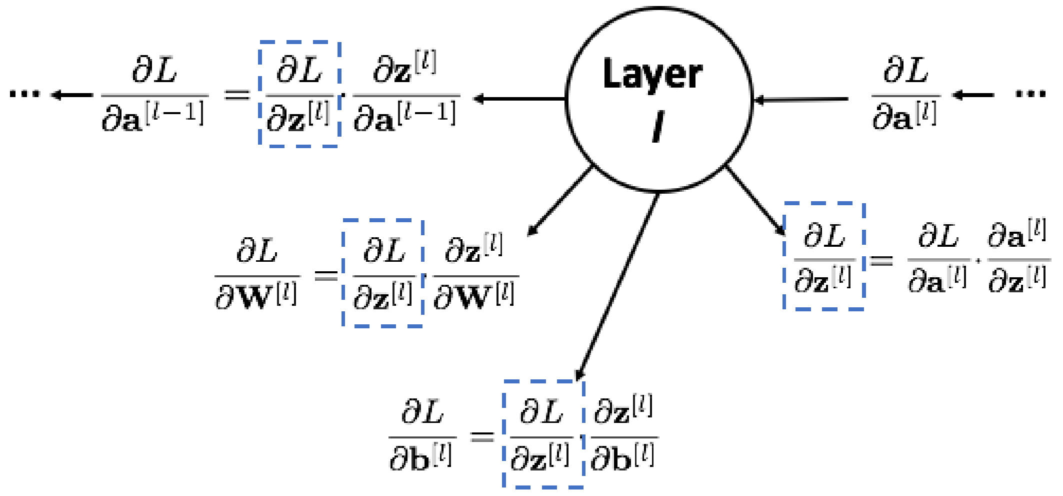
$$\frac{\partial L}{\partial W_{kl}^{[l]}} = \lim_{\epsilon \to \infty} \frac{L(\ldots, W_{kl}^{[l]} + \epsilon, \ldots) - L(\ldots, W_{kl}^{[l]}, \ldots)}{\epsilon}$$

$$\approx \frac{L(\ldots, W_{kl}^{[l]} + \epsilon_0, \ldots) - L(\ldots, W_{kl}^{[l]}, \ldots)}{\epsilon_0}$$

all other variables equal

Backprop for Other Layer Types

- So far, we have learned the formulas needed to implement backprop for MLP - incl. two layer types, fully-connected and softmax.

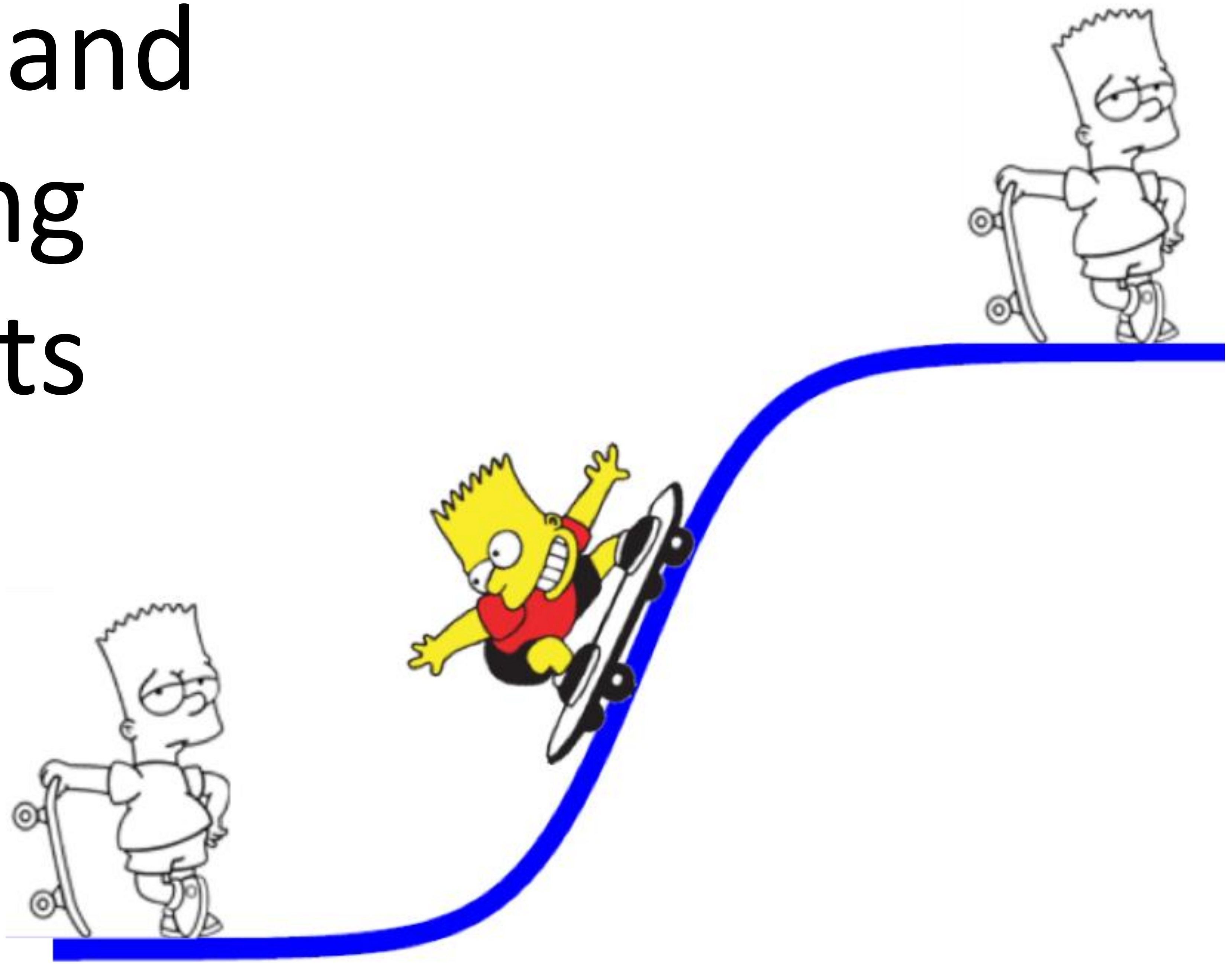

Layer
l
$\frac{\partial L}{\partial \mathbf{a}^{[l-1]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{a}^{[l-1]}}$
$\frac{\partial L}{\partial \mathbf{a}^{[l]}}$
$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{W}^{[l]}}$
$\frac{\partial L}{\partial \mathbf{z}^{[l]}} = \frac{\partial L}{\partial \mathbf{a}^{[l]}} \cdot \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}$
$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{b}^{[l]}}$

- For other layer types (e.g. CONV layers or RNN layers), the backprop formulas are derived very similarly - by back- propagating through the computational graph.
- For CNN, the computational structure turns out to be similar, for RNNs, we will also have terms resulting from back propagation through time.

Overview

- Vanishing and Exploding Gradients
- Faster Optimisation Schemes

Vanishing and Exploding Gradients



Difficulty in Training Deep Neural Nets

- Difficulties in training deep neural nets - given the techniques that were available up to ~2010.

- Various modifications have improved this situation. Important contributions from
  - X. Glorot, J. Bengio (2010)
  - S. Ioffe and C. Szegedy (2015)
  - etc.

- These aim at “avoiding” the saturation regions of the activation functions at early stages of the training by
  - stabilising variance when propagating forward and backward through the layers
  - choosing suitable activation function (different from sigmoid).

Multiplicative Structure of Backpropagation

Source: http://neuralnetworksanddeeplearning.com/chap5.html

- **Multiplicative Structure** of gradients w.r.t. weights and biases (chain rule!)

Example: MLP with n identical layers, each with single neuron and sigmoid activation function:

$$\begin{aligned}\frac{\partial L}{\partial w^{[l]}} &= a^{[l-1]} \cdot g'(z^{[l]}) \cdot \boxed{\left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right)} \cdot \frac{\partial L}{\partial a^{[L]}} \\ \frac{\partial L}{\partial b^{[l]}} &= g'(z^{[l]}) \cdot \boxed{\left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right)} \cdot \frac{\partial L}{\partial a^{[L]}}\end{aligned}$$

Product Term

- **Product Term** responsible for gradients to shrink to zero or to become large.

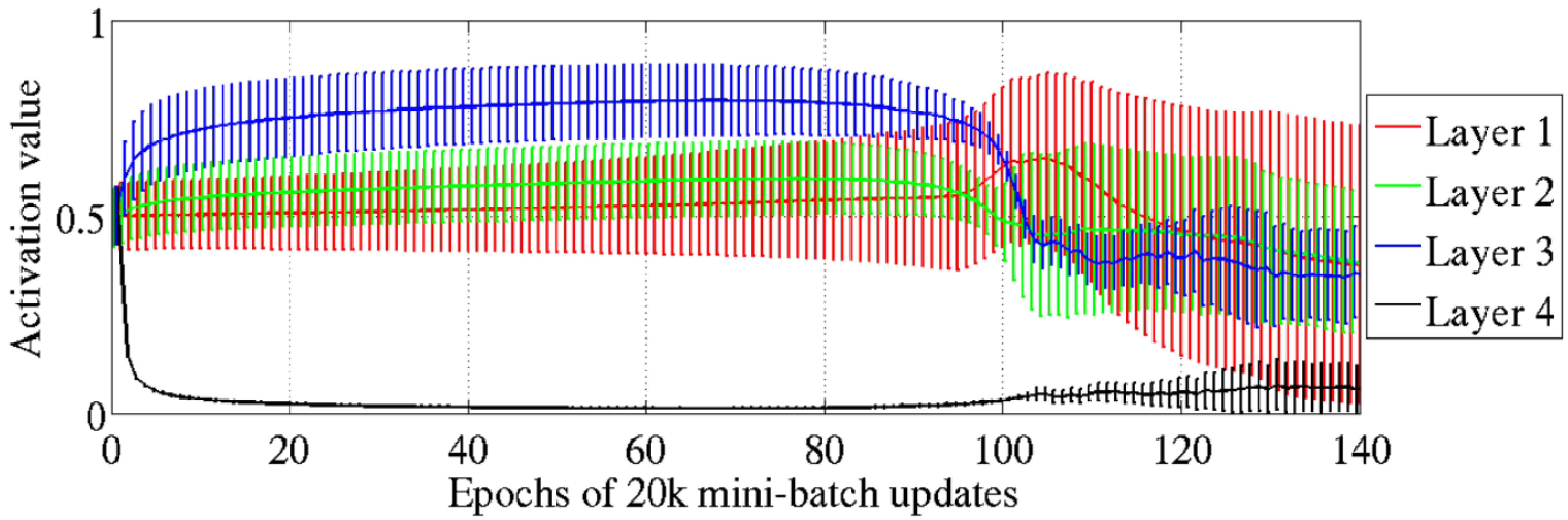
Sigmoid Function

$$\begin{aligned}\sigma'(z) &= \sigma(z)(1-\sigma(z)) \\ \|\sigma'(z)\| &\leq \sigma(0) = 1/4 \\ \sigma'(z) &\propto e^{-|z|} \text{ (for large } z)\end{aligned}$$

With random weights initialisation (standard normal) the variance of the logits is proportional to the number of inputs —> chance is high to reach saturation region with small gradient.

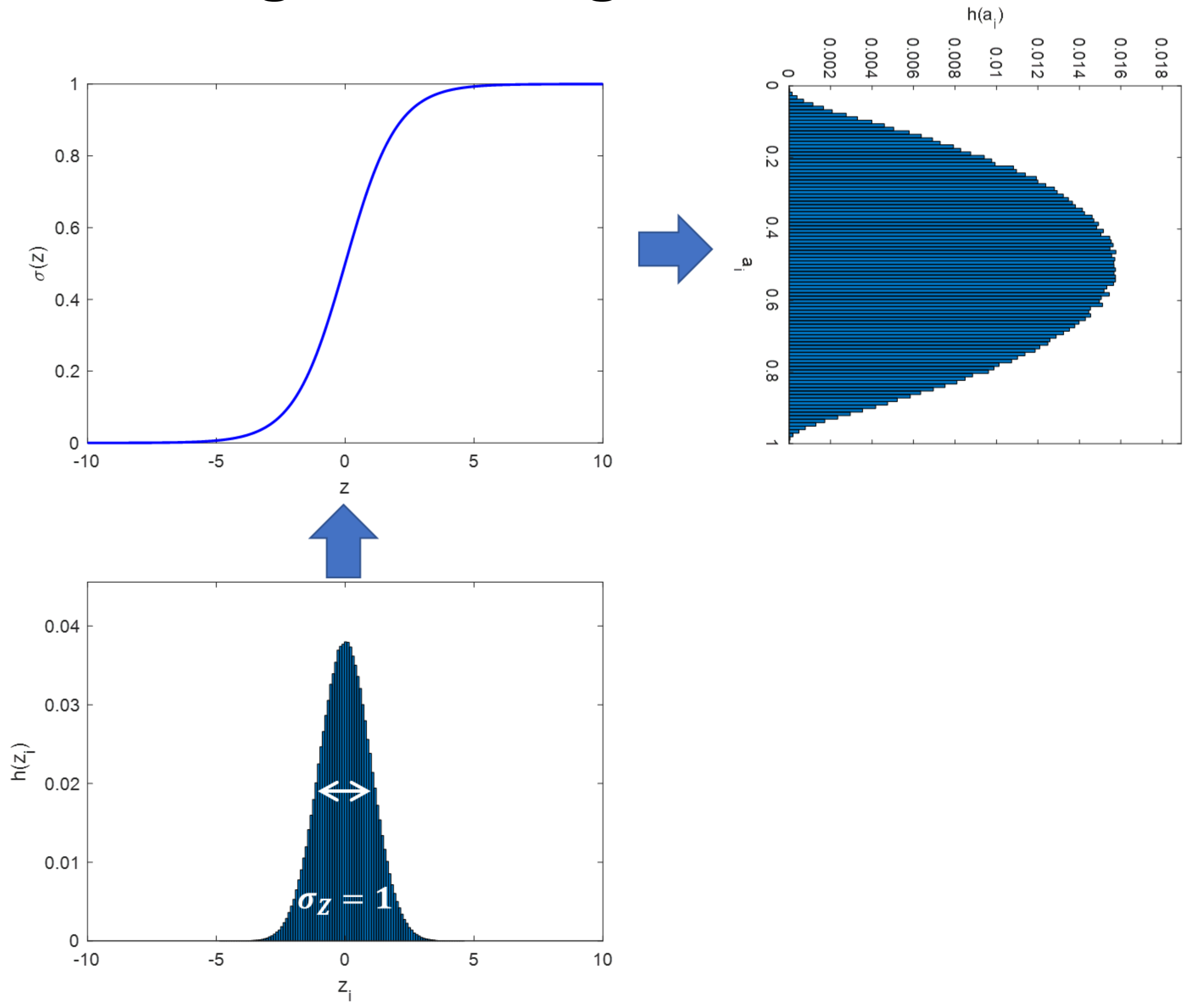
- **Coordinating Updates**: Product term (as a common factor) in several layers leads to coupled parameter updates. —> Instability in the training process.

Saturation of Activation Functions

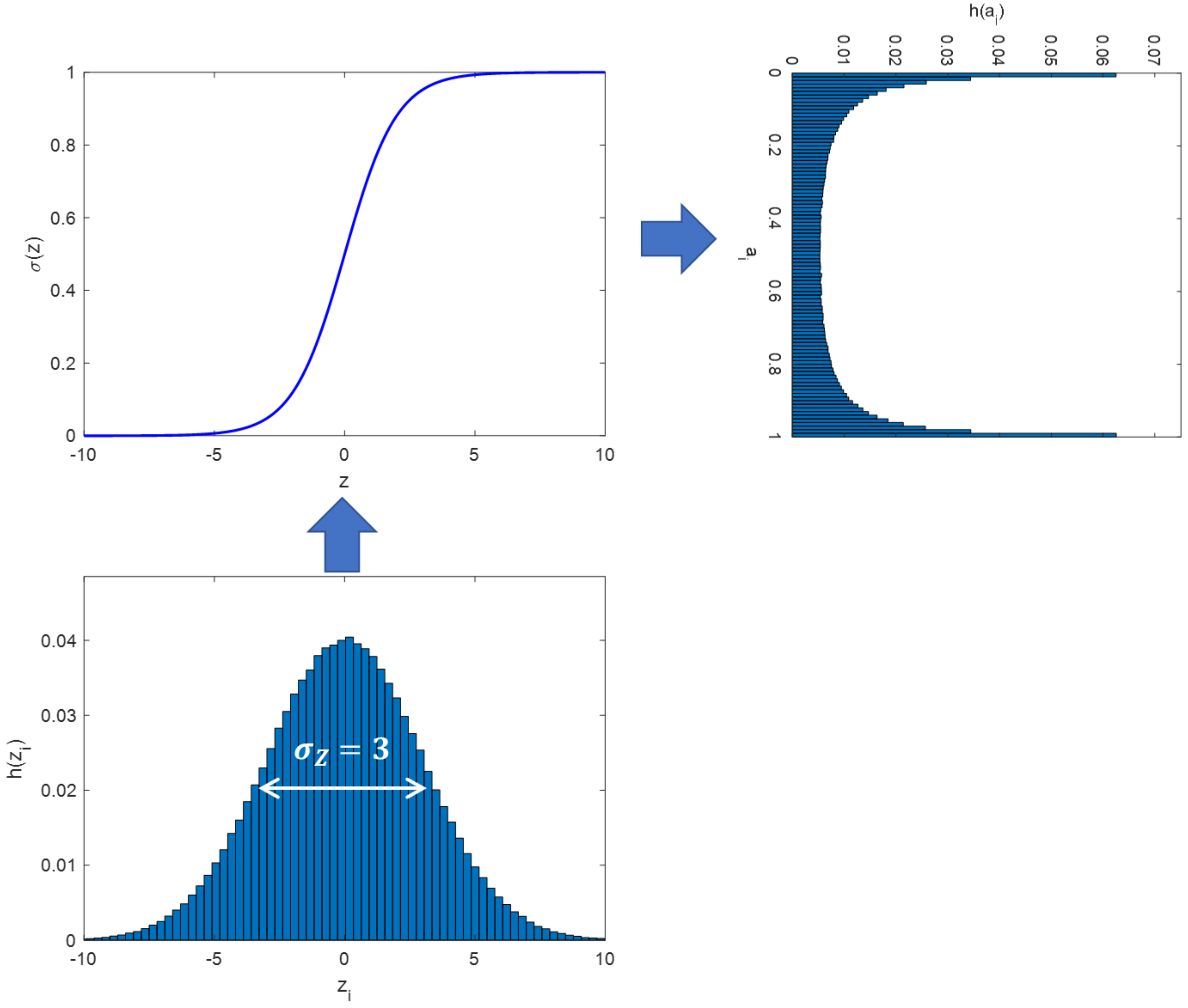


X. Glorot, J. Bengio (2010)

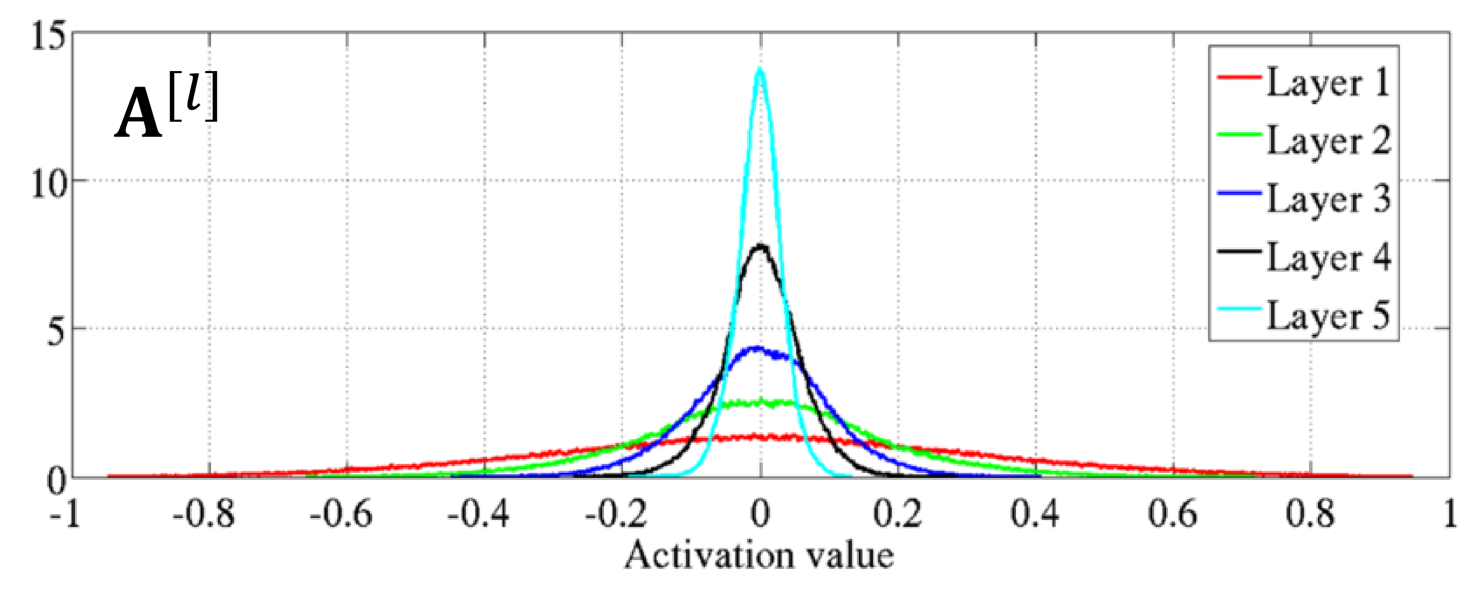
Saturation Regions in Sigmoid Function

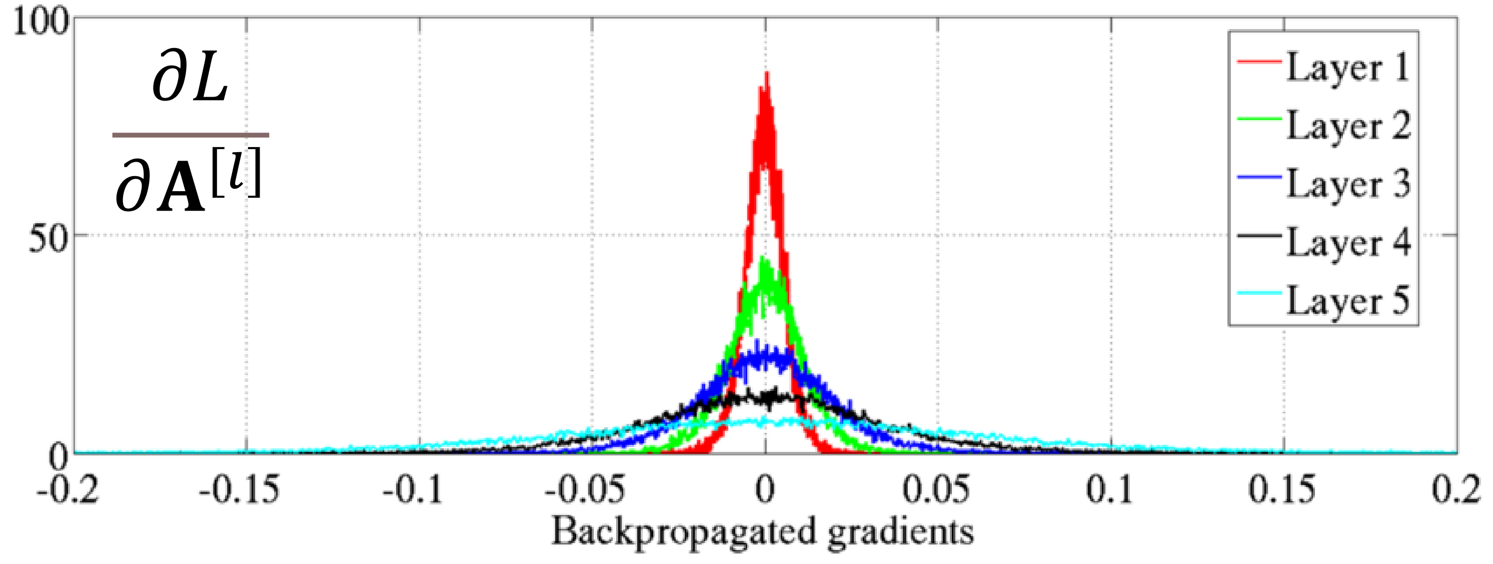


Saturation Regions in Sigmoid Function



Changing Variance of Activations and Gradients





State of the Art Initialisation in 2010

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\frac{1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}\right]$$

$$\mathbf{b}^{[l]} = 0$$

Solutions to these Problems

- Proper parameter initialisation:
  The change of variance and derivatives observed as a function of the layer index at the beginning of the training procedure can be handled with a proper initialization of the weights so that the logits do not grow too large in magnitude.

- Batch Normalisation:
  To further assure proper weight scaling also when training continues the so-called Batch Normalisation scheme will be introduced.

- Non-saturating Activation Functions:
  Finally the use of activation functions that do not saturate will also be beneficial for the train-ing of DL architectures.

- Gradient Clipping:
  Finally, application of clipping of the gradients i.e., manually limiting their absolute values, stabilises the training of deep NNs

Parameter Initialisation, Xavier/He Initialisation

Initialise weights at proper scale:

$$\mathrm{Var}\left(a_j^{[l]}\right) \sim \mathrm{Var}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]} + b_j^{[l]}\right) =^{(1)}$$

$$\mathrm{Var}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]}\right) =^{(2)} \sum_i \mathrm{Var}\left(W_{ji}^{[l]}\right) \cdot \mathrm{Var}\left(a_i^{[l-1]}\right)$$

$$\Rightarrow$$

$$\mathrm{Var}\left(a_j^{[l]}\right) \sim \sum_i \mathrm{Var}\left(W_{ji}^{[l]}\right) \cdot \mathrm{Var}\left(a_i^{[l-1]}\right) = \underbrace{n_{l-1} \cdot \mathrm{Var}\left(W^{[l]}\right)}_{(*)} \cdot \mathrm{Var}\left(a^{[l-1]}\right)$$

chose term $(*)$ of order one.

Parameter Initialisation, Xavier/He Initialisation

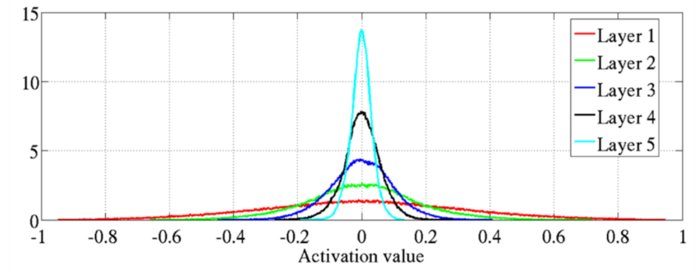
Initialise weights at proper scale:

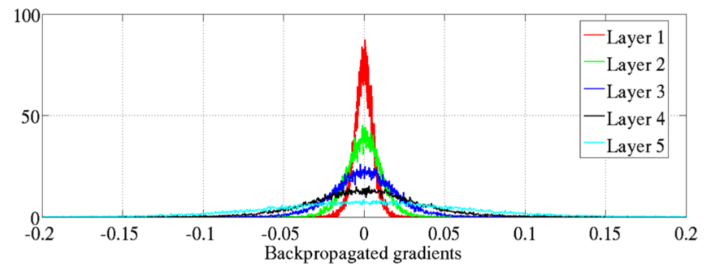
$$\mathrm{Var}\left(a_j^{[l]}\right) \sim \sum_i \mathrm{Var}\left(W_{ji}^{[l]}\right) \cdot \mathrm{Var}\left(a_i^{[l-1]}\right) = \underbrace{n_{l-1} \cdot \mathrm{Var}\left(W^{[l]}\right)}_{(*)} \cdot \mathrm{Var}\left(a^{[l-1]}\right)$$

chose term $(*)$ of order one.

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{3}{n_{l-1}}}, \sqrt{\frac{3}{n_{l-1}}}\right]$$

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\frac{1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}\right]$$





Parameter Initialisation, Xavier/He Initialisation

For backpropagation:

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^{T} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}}$$

thus

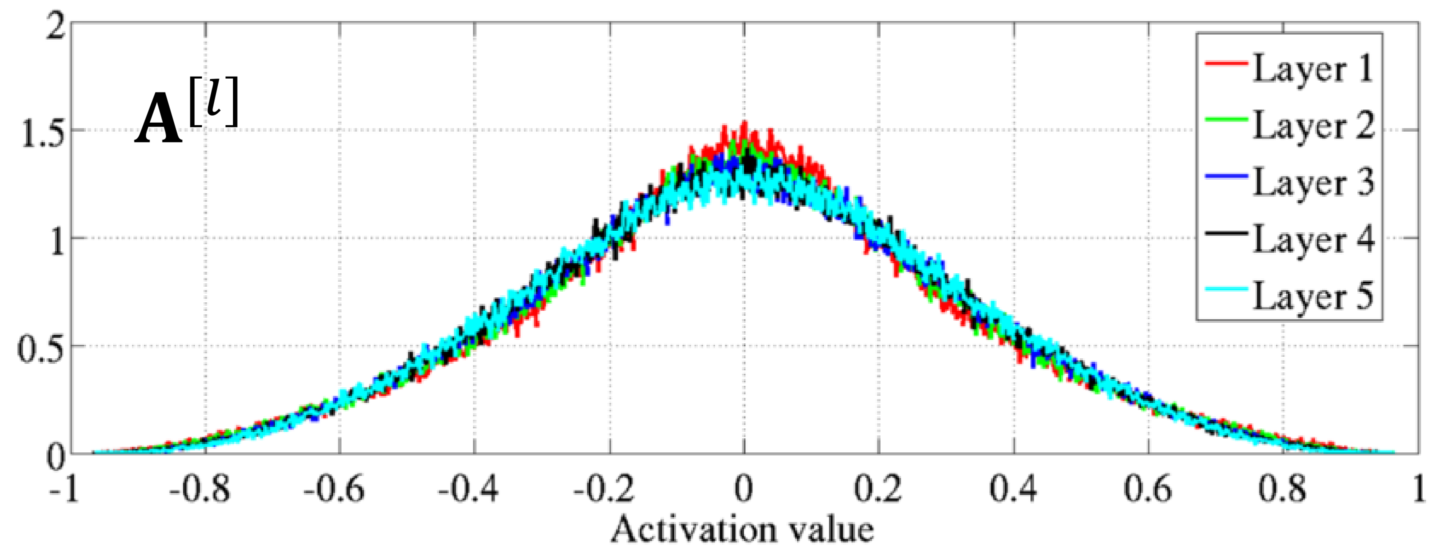
$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{3}{n_l}}, \sqrt{\frac{3}{n_l}}\right]$$

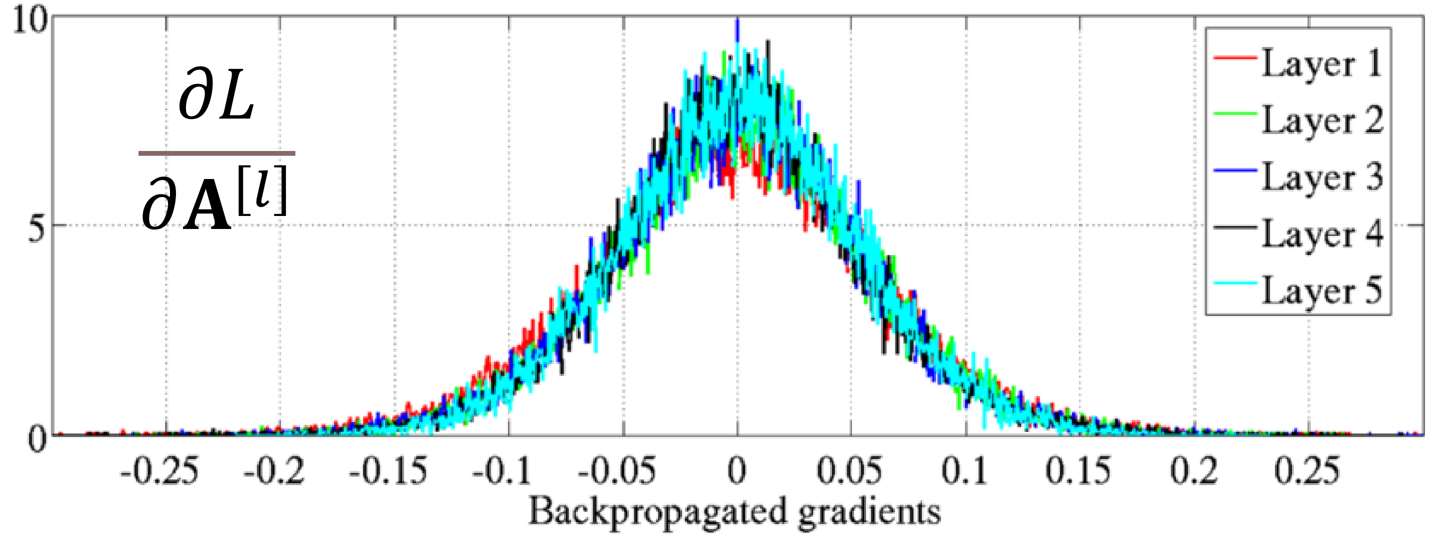
$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{3}{n_{l-1}}}, \sqrt{\frac{3}{n_{l-1}}}\right]$$

Parameter Initialisation, Xavier/He Initialisation

- **Compromise**:

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{6}{n_{l-1}+n_l}}, \sqrt{\frac{6}{n_{l-1}+n_l}}\right]$$

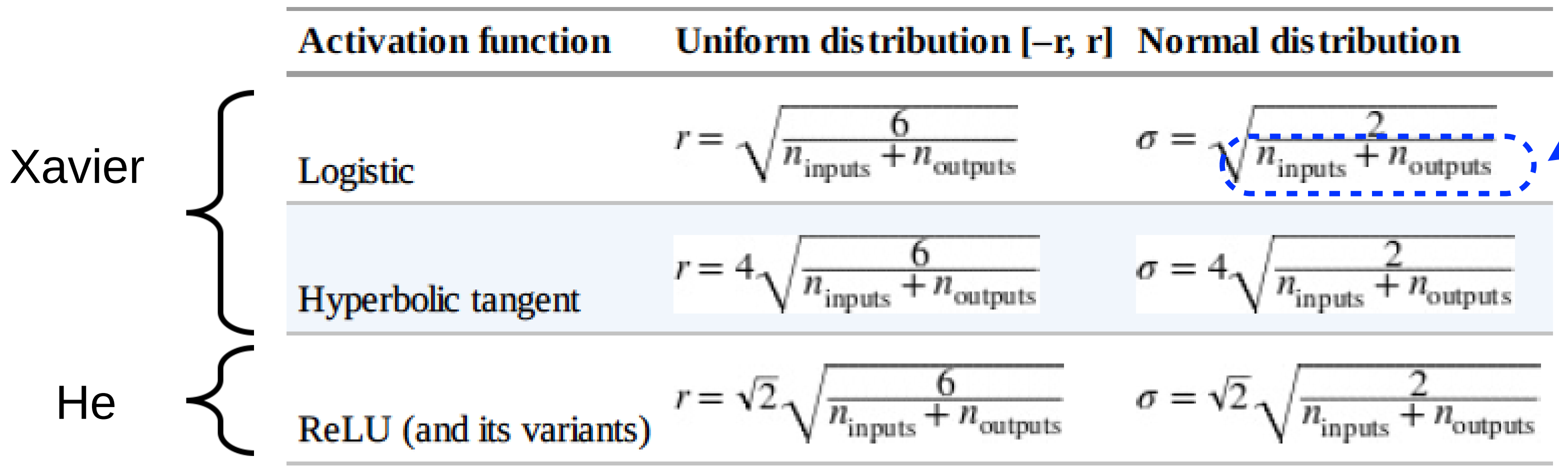




Parameter Initialisation, Xavier/He Initialisation

- **Extension to normal distribution and other activation functions**:

  - Properly scaling the width of the distribution (depends on the random distribution used to generate the weights and the activation function):



	Activation function	Uniform distribution [–r, r]	Normal distribution
Xavier	Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
	Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
He	ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

number of input and output connections **for a given layer**

  - Remark: Scheme does not guarantee to keep the input/output variance identical in both directions, but works well in practice.

Batch Normalisation

- Idea introduced by S. Ioffe and C. Szegedy(1):
  - Normalise the input to each layer per mini-batch just before applying the activation function - estimate mean and stdev used for the normalisation from the current mini-batch.
  - Then let the model learn the optimal scale and mean of the inputs for each layer.

- Very effective method to improve the stability of the training
  - According to Ioffe & Szegedy significantly reduces the problem of coordinat updates across many layers.
  - Limits the amount to which parameter updates in earlier layers can affect the distribution of values the later layers see.

In a more recent paper (S.Santurkar et al, NIPS 2018, (2)) this reason is questioned. They claim, that the cost surface is significantly smoothened by batch norm.

- Can be applied to any input or hidden layer in a network.

(1) "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," S. Ioffe and C. Szegedy (2015).
(2) "How Does Batch Normalization Help Optimization?", S.Santurkar et al, NIPS 2018.

Batch Normalisation Equations (1)

Logits matrix for mini-batch as input to activation function

$$\mathbf{Z}^{[l]}_{\{r\}} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)}_{\{r\}} & \mathbf{z}^{[l](2)}_{\{r\}} & \dots & \mathbf{z}^{[l](m)}_{\{r\}} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

Average over mini-batch (size m)

$$\boldsymbol{\mu}^{[l]}_{\{r\}} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{z}^{[l](i)}_{\{r\}} \qquad \boldsymbol{\sigma}^{[l]}_{\{r\}} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{z}^{[l](i)}_{\{r\}} - \boldsymbol{\mu}^{[l]}_{\{r\}}\right)^2}$$

Actual normalization

$$\hat{\mathbf{Z}}^{[l]}_{\{r\}} = \frac{\mathbf{Z}^{[l]}_{\{r\}} - \boldsymbol{\mu}^{[l]}_{\{r\}}}{\boldsymbol{\sigma}^{[l]}_{\{r\}} + \epsilon}$$

This would influence the reprsentational capacity of the model!

Batch Normalisation Equations (1)

Average over mini-batch (size m)

$$\boldsymbol{\mu}_{\{r\}}^{[l]} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{z}_{\{r\}}^{[l](i)} \qquad \boldsymbol{\sigma}_{\{r\}}^{[l]} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{z}_{\{r\}}^{[l](i)} - \boldsymbol{\mu}_{\{r\}}^{[l]}\right)^2}$$

Actual normalization

$$\hat{\mathbf{Z}}_{\{r\}}^{[l]} = \frac{\mathbf{Z}_{\{r\}}^{[l]} - \boldsymbol{\mu}_{\{r\}}^{[l]}}{\boldsymbol{\sigma}_{\{r\}}^{[l]} + \epsilon}$$

$\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ are model parameters and optimized during training

$$\tilde{\mathbf{Z}}_{\{r\}}^{[l]} = \boldsymbol{\gamma}^{[l]} \cdot \hat{\mathbf{Z}}_{\{r\}}^{[l]} + \boldsymbol{\beta}^{[l]}$$

Input to activation function

Batch Normalisation Equations (2)

- When using batch normalisation, the bias parameter $\mathbf{b}^{[l]}$ used in the definition of $\mathbf{Z}^{[l]}$ is eliminated since it is redundant:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \cancel{\mathbf{b}^{[l]}}$$

- Batch Normalisation at Test Time or in Production
  - The mean $\boldsymbol{\mu}^{[l]}$ and stdev $\boldsymbol{\sigma}^{[l]}$ computed per mini-batch during training are used to compute suitable averages and stdev's that can be used at test time or for production.

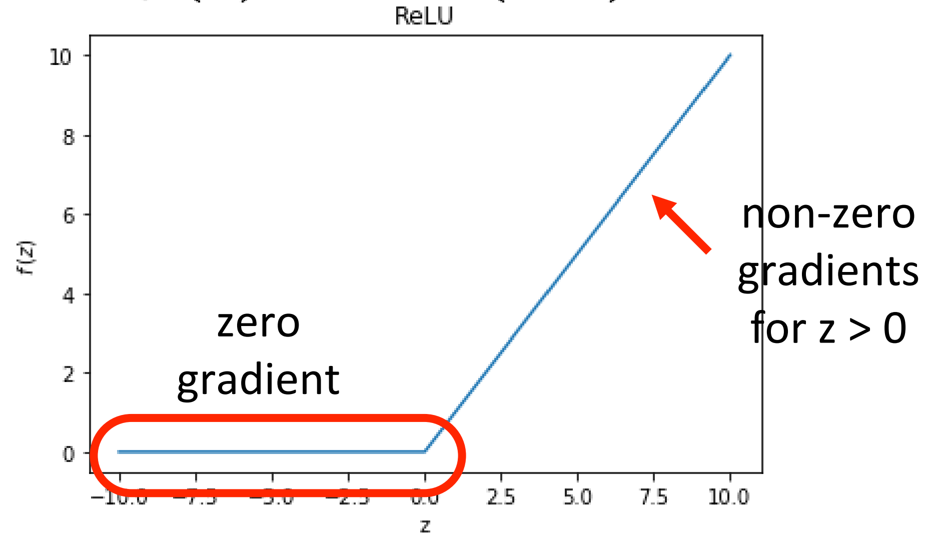
$$\boldsymbol{\mu}^{[l]} = (1-\beta) \cdot \boldsymbol{\mu}_{\{r\}}^{[l]} + \beta \cdot \boldsymbol{\mu}^{[l]}$$
$$\boldsymbol{\sigma}^{[l]} = (1-\beta) \cdot \boldsymbol{\sigma}_{\{r\}}^{[l]} + \beta \cdot \boldsymbol{\sigma}^{[l]}$$

- Batch normalisation changes backprop equations
  - The derivates w.r.t. to the new variables $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ need to be included.
  - The derivatives of the estimates $\boldsymbol{\mu}^{[l]}$ and stdev $\boldsymbol{\sigma}^{[l]}$ also need to be taken into account (see e.g. https://chrisyeh96.github.io/2017/08/28/deriving-batchnorm-backprop.html)

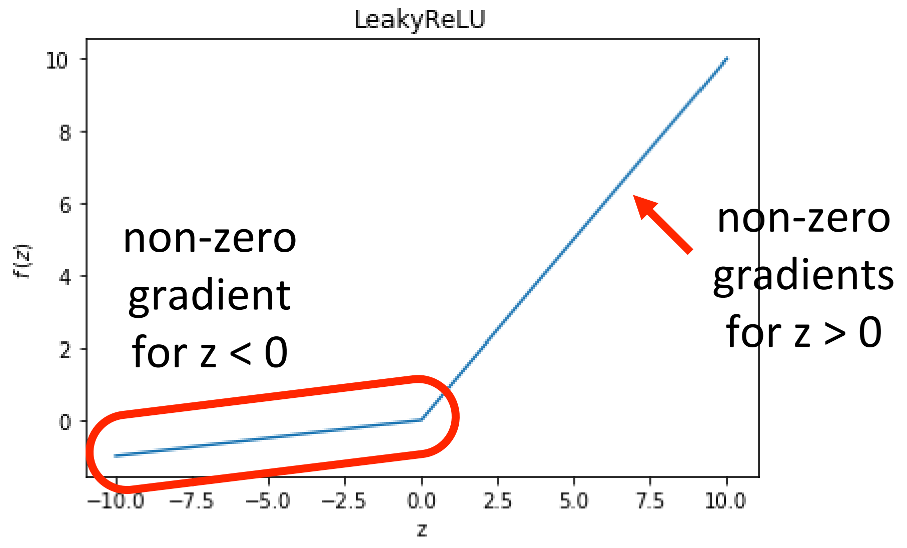
Non-Saturating Activation Functions

Avoid S-shaped activation functions that flatten out at larger z-magnitudes —> ReLU, LeakyReLU or ELU (see Week 3).

$$f(z) = \max(0, z)$$



$$f(z) = \max(\eta z, z)$$



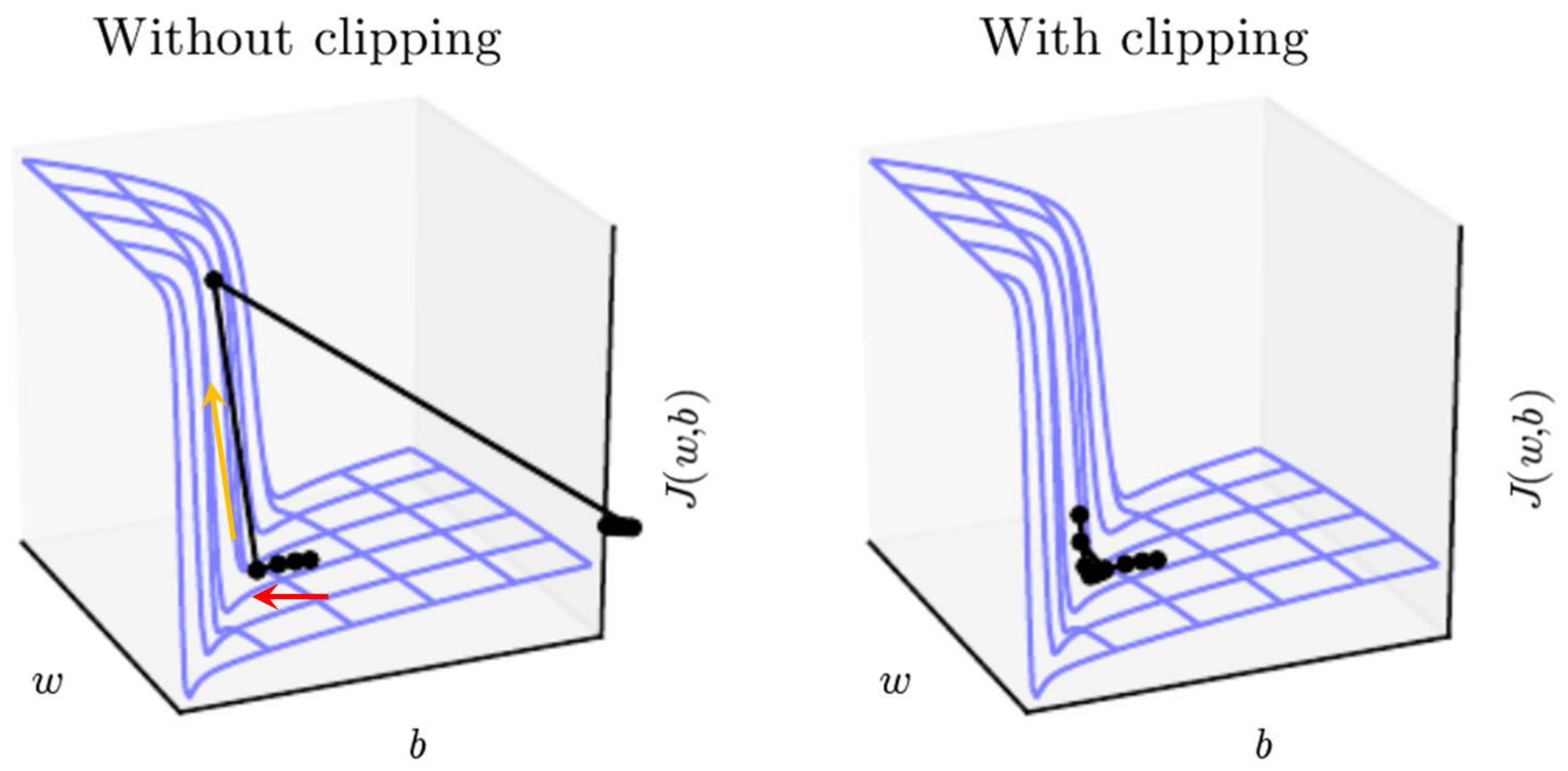
ReLU suffers the ***dying units problem***: During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it is outputting 0. Since the gradient at z<0 is 0, there is no weight update for this neuron and the neuron is likely to stay dead.

With the **leaky ReLU** (or its smooth version, the ELU) this problem cannot occur. The parameter η can be determined by hyper-parameter tuning or by setting a good default value ($\eta = 0.01$). See (*) for an evaluation of the different ReLUs.

(*) "Empirical Evaluation of Rectified Activations in Convolution Network," B. Xu et al. (2015).

Gradient Clipping

- Intuition provided in the “Deep Learning” book (I. Goodfellow): Deep networks often have extremely steep regions resembling cliffs - as a result of the composite structure of the operations where weights get multiplied. At the cliffs, the gradient can be become very large.

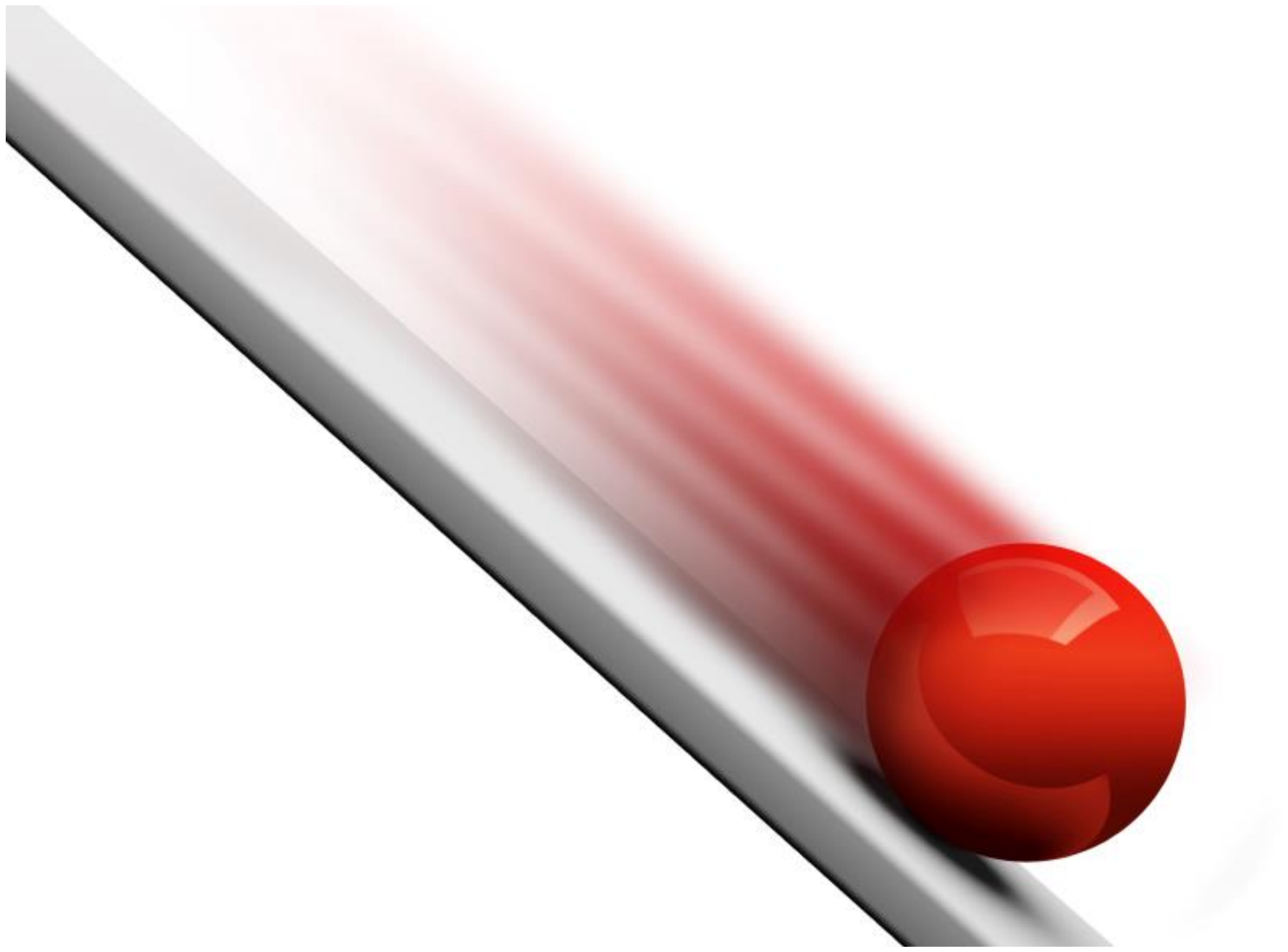


Advanced Optimisation Methods

Momentum

RMSprop

Adam



Gradient Descent in Deep Learning Models

- Vanilla Gradient Descent (BGD, MBGD, SGD) may lead to slow learning in flat regions.
- Cost functions in DNNs are non-convex: Multiple local minima, saddle points and flat regions (see previous slide) may occur.
- Non-global minima are *not* considered as a problem:
  - Typical network structures have symmetries (e.g. MLP w.r.t. permutations of the units in a layer). An according number of global minima must exist.
  - Other non-global minima are considered to be very rare: At given local minimum, cost function needs to grow in all directions, hard to meet in high dimensions).
  - BGD and SGD can help to escape from local minima.
- With saddle points or flat regions learning can get very slow.

➡ **Some improvements beyond gradient descent desirable!**

Momentum

- Intuition (from A. Géron, 'Hands-On Machine Learning with Scikit-Learn and TensorFlow')

  "Imagine a ball rolling down a gentle slope on a smooth surface. It starts out slowly but quickly picks up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance).(…) In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom."

- Algorithm: Compute an exponentially decaying sum of past gradients and move in the direction of this sum.

$$\mathbf{m} \leftarrow \beta \cdot \mathbf{m} + \alpha \cdot \boldsymbol{\nabla}_\theta J$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{m}$$

$\beta = 0.9$ typically

How does the update scheme work?

Momentum

Update as function of training step t :

$t = 1$: $\quad \alpha \cdot \boldsymbol{\nabla}_\theta J$

$t = 2$: $\quad \beta \cdot \alpha \cdot \boldsymbol{\nabla}_\theta J + \alpha \cdot \boldsymbol{\nabla}_\theta J$

$t = 3$: $\quad \beta^2 \cdot \alpha \cdot \boldsymbol{\nabla}_\theta J + \beta \cdot \alpha \cdot \boldsymbol{\nabla}_\theta J + \alpha \cdot \boldsymbol{\nabla}_\theta J$

…

$t = n$: $\quad \beta^{n-1} \cdot \alpha \cdot \boldsymbol{\nabla}_\theta J + \beta^{n-2} \cdot \alpha \cdot \boldsymbol{\nabla}_\theta J + \cdots + \beta \cdot \alpha \cdot \boldsymbol{\nabla}_\theta J + \alpha \cdot \boldsymbol{\nabla}_\theta J = \frac{1-\beta^n}{1-\beta} \cdot \boldsymbol{\nabla}_\theta J$

Thus, for large t one obtains: $\quad \mathbf{m} = \frac{1}{1-\beta} \cdot \nabla_\theta J$

E.g., with $\beta = 0.9$: $\quad \mathbf{m} = 10 \cdot \nabla_\theta J$

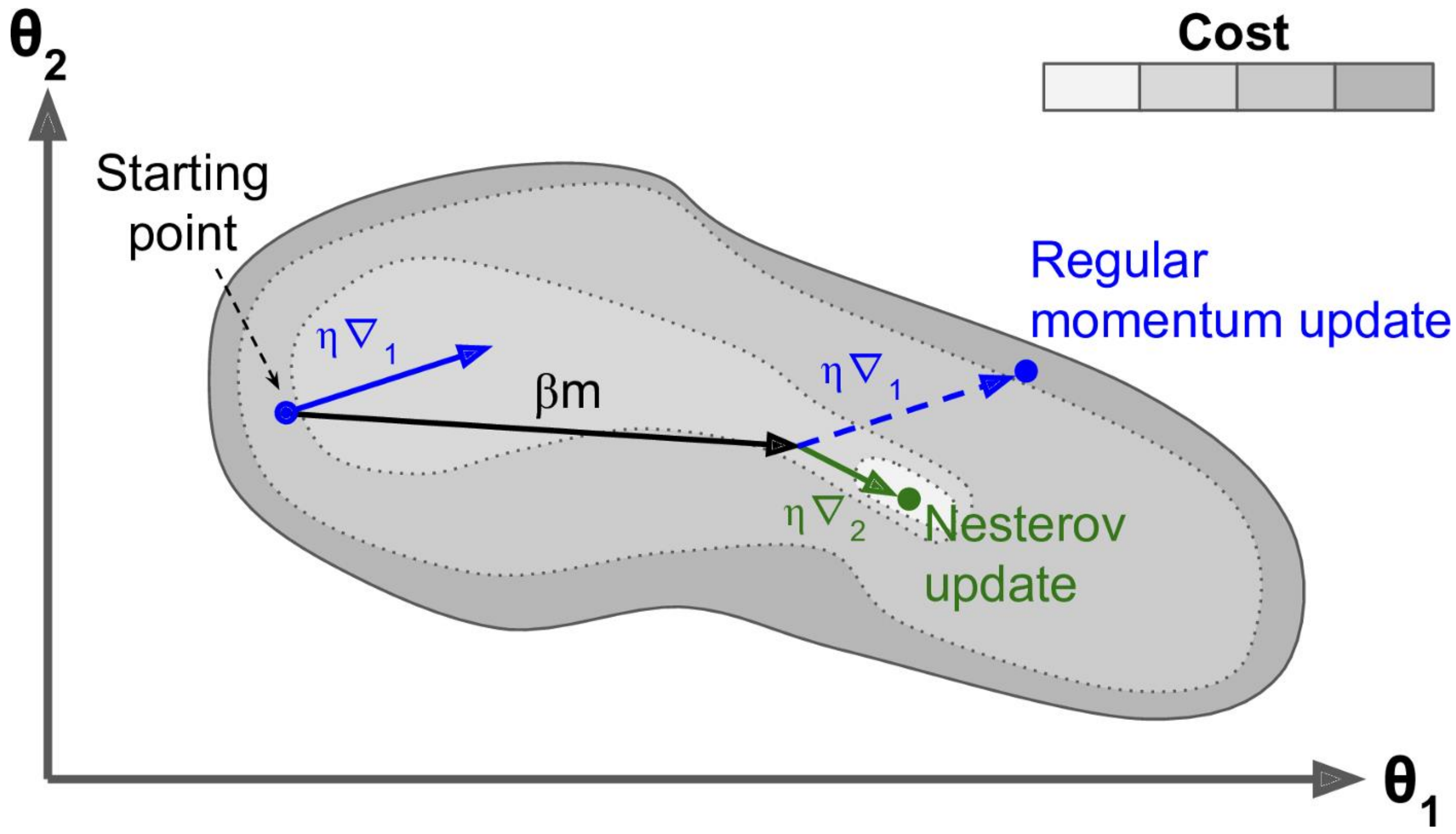
What physical model motivates the expression "**momentum**" optimizer?

Nesterov Accelerated Gradient Optimization

- Similar to Momentum only gradient avaluated ahead by $\beta \cdot \mathbf{m}$:

$$\mathbf{m} \leftarrow \beta \cdot \mathbf{m} + \alpha \cdot \nabla_\theta J(\boldsymbol{\theta} - \beta \cdot \mathbf{m})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{m}$$

$\beta = 0.9$ typically



AdaGrad Optimization

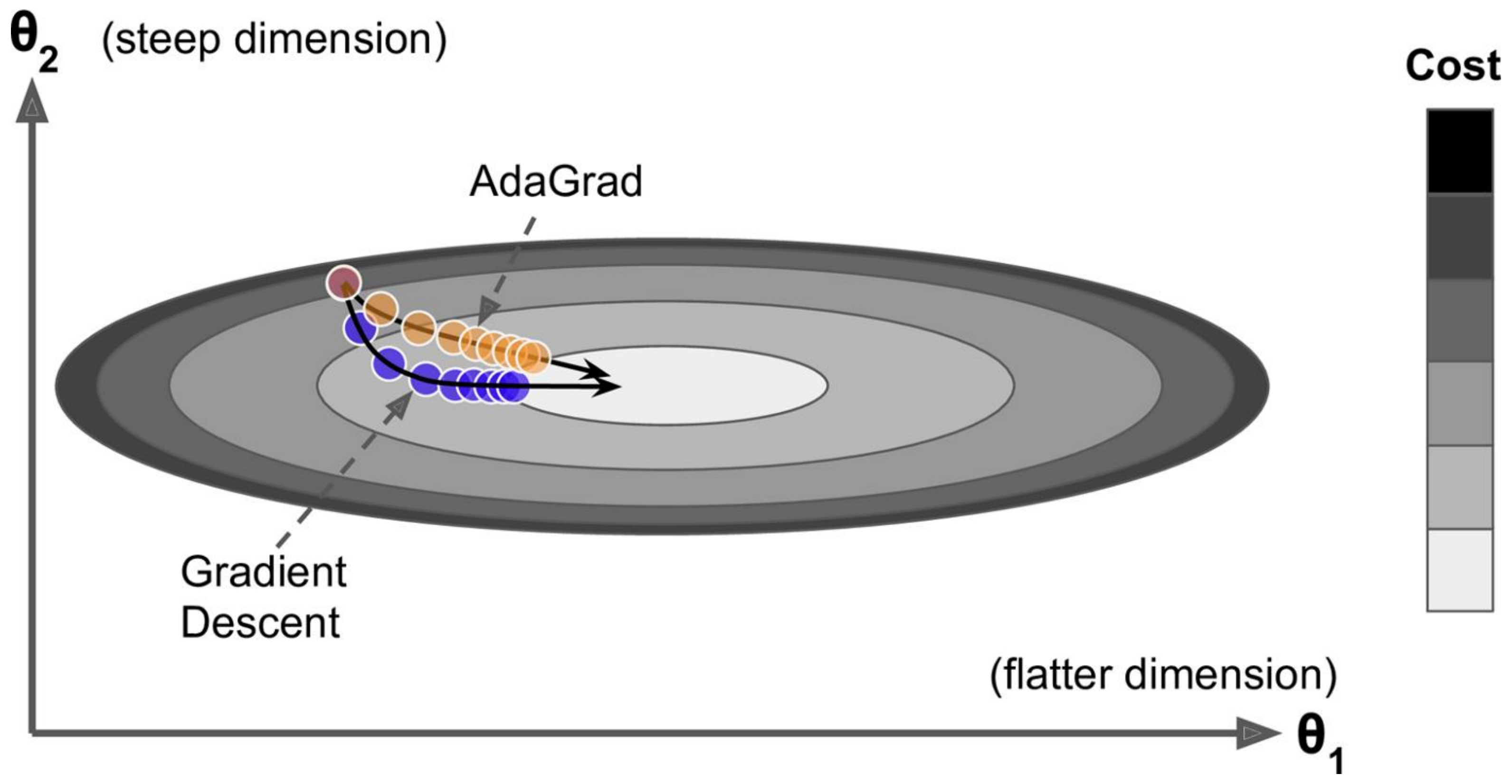
- Designed to avoid “elongated bowl” problem:

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J \cdot \nabla_\theta J$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s} + \epsilon}} \cdot \nabla_\theta J$$

all operations
component-wise

Scales learning rate



How does AdaGrad optimization scale a strong or week gradient direction respectively?

RMSProp Optimization

- extends AdaGrad:

$$\mathbf{s} \leftarrow \mathbf{s} + \boldsymbol{\nabla}_\theta J \cdot \boldsymbol{\nabla}_\theta J$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s} + \epsilon}} \cdot \boldsymbol{\nabla}_\theta J$$

- adds exponentially decaying average with decay parameter β

$$\mathbf{s} \leftarrow \beta \cdot \mathbf{s} + (1 - \beta) \cdot \boldsymbol{\nabla}_\theta J \cdot \boldsymbol{\nabla}_\theta J$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s} + \epsilon}} \cdot \boldsymbol{\nabla}_\theta J$$

typically $\beta = 0.9$

all operations
component-wise

What problem of AdaGrad optimization is solved by RMSProp optimization and how

Adam Optimization

- Combines
  - Momentum: line 1, except for exp. decaying average instead of sum
  - and RMSProp (line 2 + 5):

$$\mathbf{m} \leftarrow \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \nabla_\theta J$$

$$\mathbf{s} \leftarrow \beta_2 \cdot \mathbf{s} + (1 - \beta_2) \cdot \nabla_\theta J \cdot \nabla_\theta J$$

$$\widehat{\mathbf{m}} = \frac{\mathbf{m}}{1 - \beta_1}$$

$$\hat{\mathbf{s}} = \frac{\mathbf{s}}{1 - \beta_2}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\hat{\mathbf{s}} + \epsilon}} \cdot \widehat{\mathbf{m}}$$

typically:

$\beta_1 = 0.9$

$\beta_2 = 0.999$

What is the purpose of the so-called "bias corrections" of the estimates i.e., the division of m and s by 1-β_1 or 1-β_2 respectively?

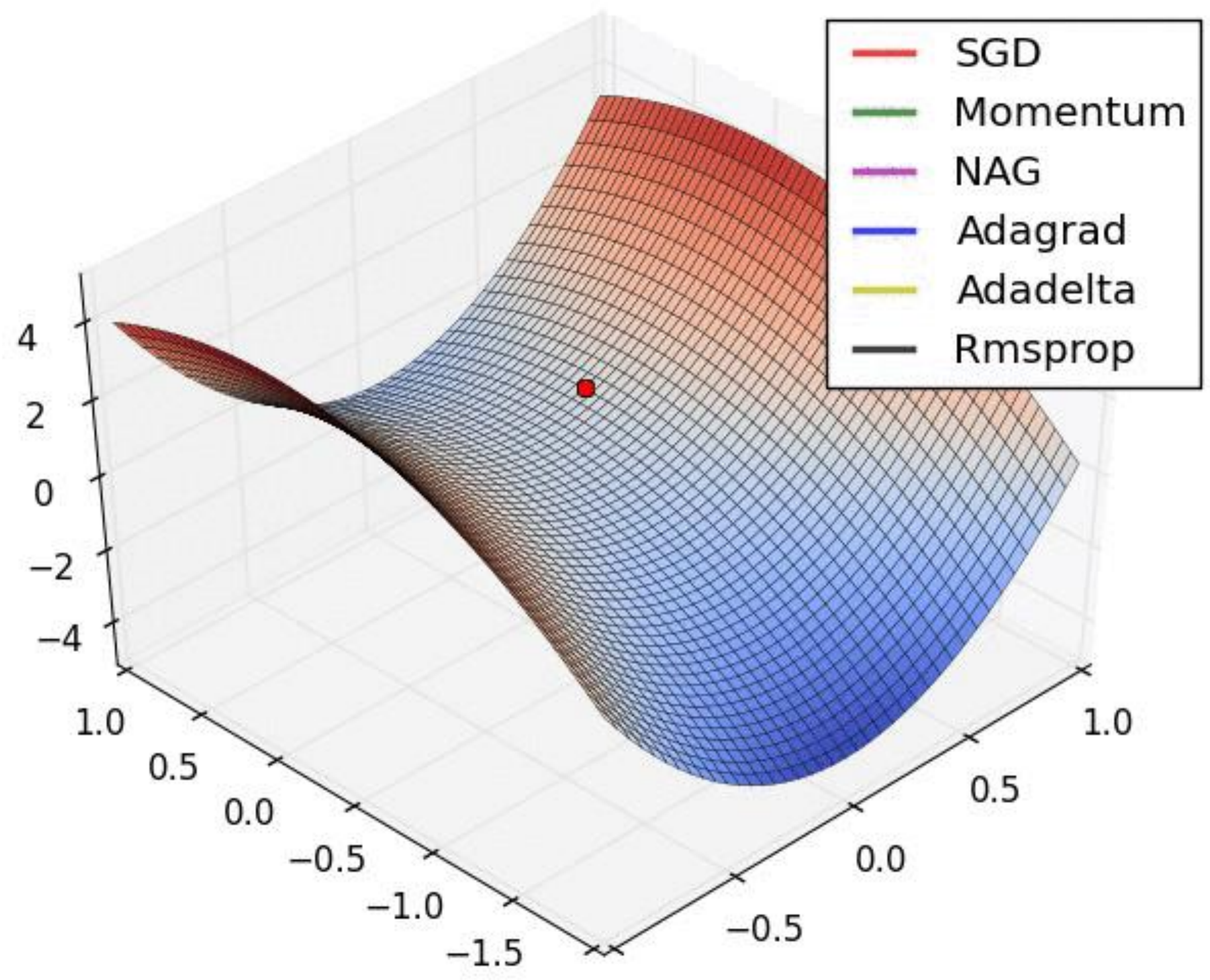
$$\mathbf{m} \leftarrow \beta_1 \cdot \mathbf{m} + (1-\beta_1) \cdot \nabla_\theta J$$

$$\mathbf{s} \leftarrow \beta_2 \cdot \mathbf{s} + (1-\beta_2) \cdot \nabla_\theta J \cdot \nabla_\theta J$$

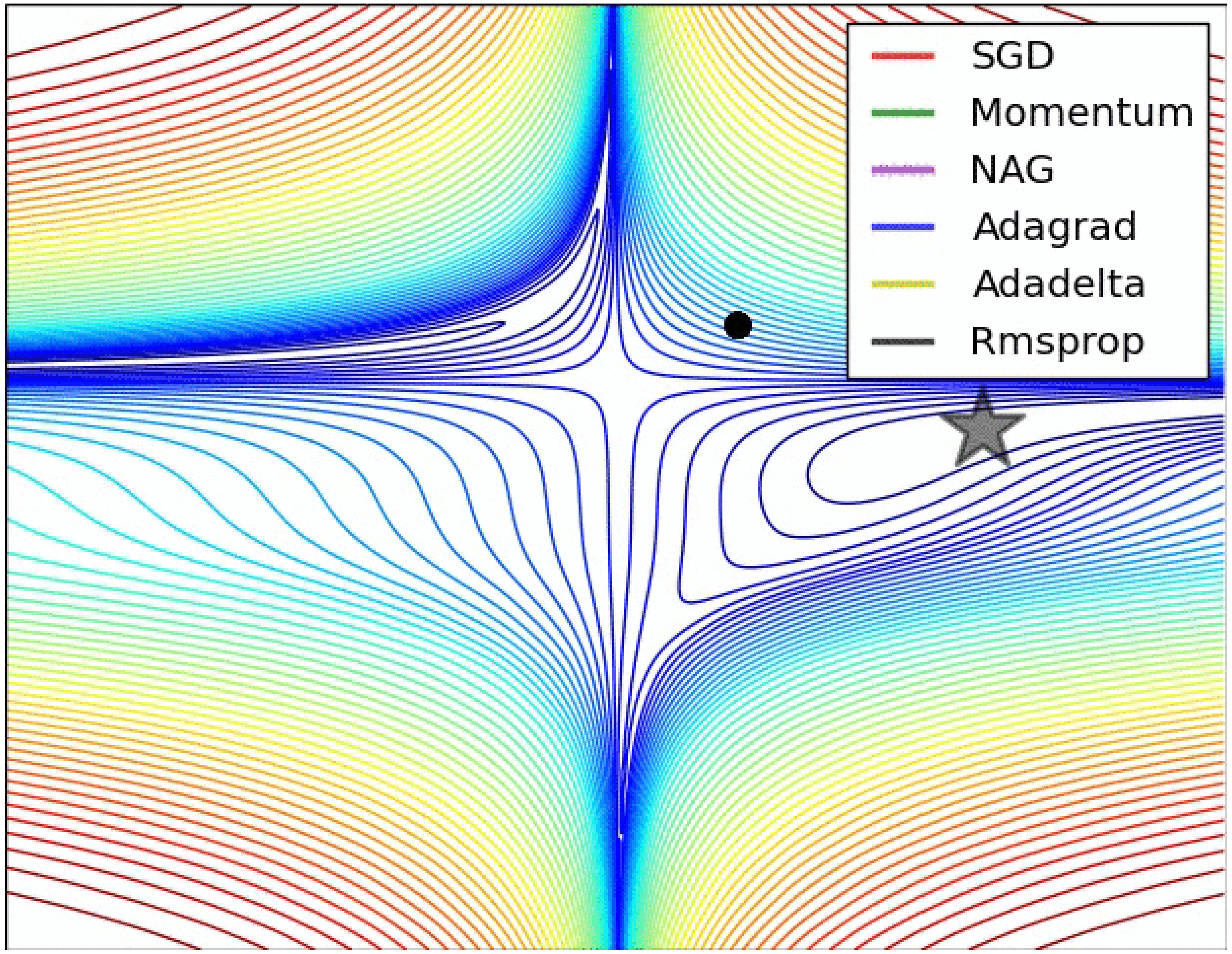
$$\widehat{\mathbf{m}} = \frac{\mathbf{m}}{1-\beta_1}$$

$$\hat{\mathbf{s}} = \frac{\mathbf{s}}{1-\beta_2}$$

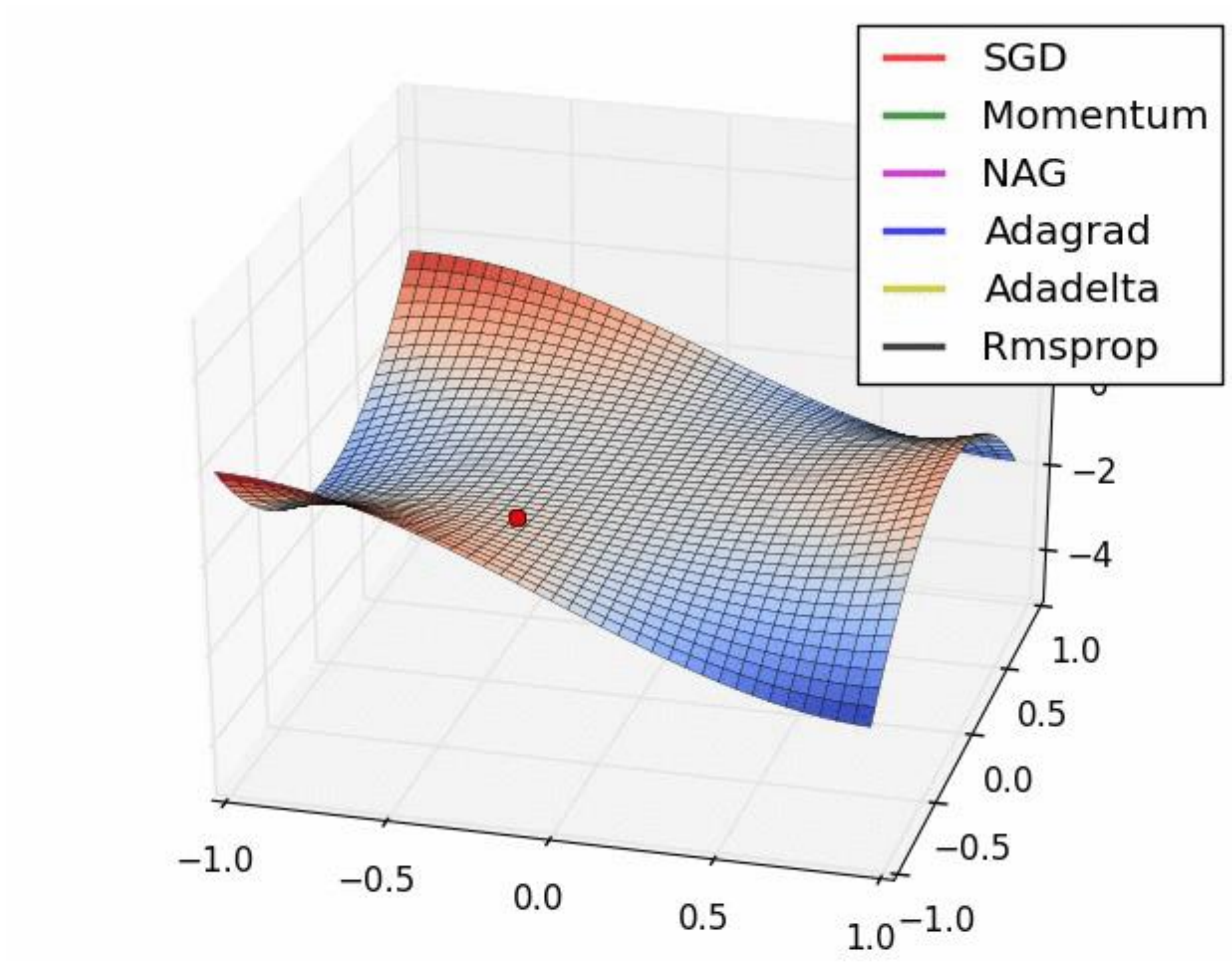
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\hat{\mathbf{s}} + \epsilon}} \cdot \widehat{\mathbf{m}}$$



- Optimizers that do not apply scaling based on gradient information struggle to break symmetry here:
  - GD just gets stuck on the ridge.
  - Both Momentum and Nesterov Optimization exhibits many oscillations along the ridge until they build up velocity in the perpendicular direction.
- All other optimizers that do apply scaling based on gradient information (AdaGrad, Adam and RMSProp) quickly break symmetry and begin to descent towards the optimum.



- GD behaves as expected i.e., it starts off rather fast along the steep part but then slows down and fails to converge when it gets stuck on the flat bottom of the bowl.
- Momentum based techniques (Momentum and Nesterov Optimization) acquire – due to the large initial gradient – a high velocity at the beginning which makes them shooting off and bouncing around.
- Optimizers that apply scaling based on gradient information (AdaGrad, Adam and RMSProp) proceed more like accelerated SGD and handle large gradients with more stability. Nevertheless, AdaGrad almost goes unstable – like momentum-based techniques – due to the high initial velocity.



- Due to the flat region GD has problems to start off and to converge.
- Momentum based techniques (Momentum and Nesterov Optimization) acquire a high velocity and explore around, almost taking a different path.
- Optimizers that apply scaling based on gradient information (AdaGrad, Adam and RMSProp) proceed like accelerated SGD and handle the situation well.

Learning Schedule

- Instead of adapting the learning rate in accordance with information collected during the optimisation process along the trajectory in parameter space we may manually adjust the learning rate in the different phases of learning (~ epochs or update steps).

- Typically, start with high learning rate, then reduce it:

Piecewise Constant Learning

Epochs	Rate
1-100	0.1
101-300	0.01
301-500	

Performance scheduling
Measure the validation error every *N* steps, reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling
Set the learning rate to a function of the iteration number *t*:

$$\alpha(t) = \alpha_0 \cdot e^{-t/T}$$

Power Scheduling

$$\alpha(t) = \alpha_0(1 + t/T)^{-c}$$

The hyperparameter c is typically set to 1.
Similar to exponential scheduling, but learning rate drops much more slowly.