Deep Learning

LM Computer Science, Data Science, Cybersecurity

2nd semester - 6 CFU

References:

Deep Learning Book (main book)

Mitchell (machine learining concepts)

Bishop (machine learning)

Optimization for training deep models (chapter 8)

Optimization for neural networks

- Optimization of neural networks is difficult
  - Specialized set of techniques
- Optimization: goal is to optimize a performance measure
- ML: we care about true error, that is not computable

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- **Empirical risk minimization**: We optimize it **indirectly** through the Loss function on the training set

$$J\ \ (\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim \hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

  - Prone to overfitting (memorizing training data)
  - Many loss functions (e.g. **0-1 Loss**) are not differentiable
    - → no gradient descent

Surrogate Loss

- The loss function we care about (e.g. classification error) cannot be optimized efficiently
- Surrogate loss: a proxy, but easier to optimize
  - E.g. negative log likelihood as surrogate for 0-1 loss
- We don’t want to find a local minimum!
  - We don’t really care about the error on training
  - E.g. early stopping monitoring any loss on the validation set

Batch/Minibatch Algorithms

- Each update is based on the expected loss function over training data
- E.g. Gradient of NLL

$$J(\theta) = -\mathbb{E}_{(x,y)\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}\,|\boldsymbol{x})$$

  - Computing the expectation on the entire dataset is expensive
  - The standard error for the estimation of the mean scales less than linearly: $\frac{\sigma}{\sqrt{n}}$
  - Most optimization algorithms converge faster (in time, not number of updates) using less accurate gradient estimations w.r.t. the exact gradient
  - Small batch size have a regularization effect (**noise** in the gradient)
  - **Online** algorithms: consider one example at a time

Mini-batches for Gradient Descent

(Full, batch) gradient descent computes the gradient of the Loss on the WHOLE training set and THEN updates the weights

*Pro: Loss depends on ALL examples, it computes the **right** gradient*

Con: if training set contains 1 million examples, only 1 weights update per epoch (i.e. 1 update after having computed the output for ALL examples)

(Online) Stochastic gradient descent computes the gradient of the Loss on a SINGLE example of the training set and then updates the weights

Pro: if training set contains 1 million examples, 1 weights update per example (i.e. 1 million updates per epochs)

Con: gradient descent is less accurate since it computes the gradient of Loss ONLY with respect to a single example

COMPROMISE

Mini-batch Stochastic gradient descent computes the gradient of the Loss on a subset of examples (mini-batch) of the training set and then updates the weights

trade-off: more accurate computation of the gradient vs reduced number of updates

moreover: - allows for distribution of computation on several computational devices

- can leverage matrix/matrix operations, which are more efficient (e.g., GPUs)

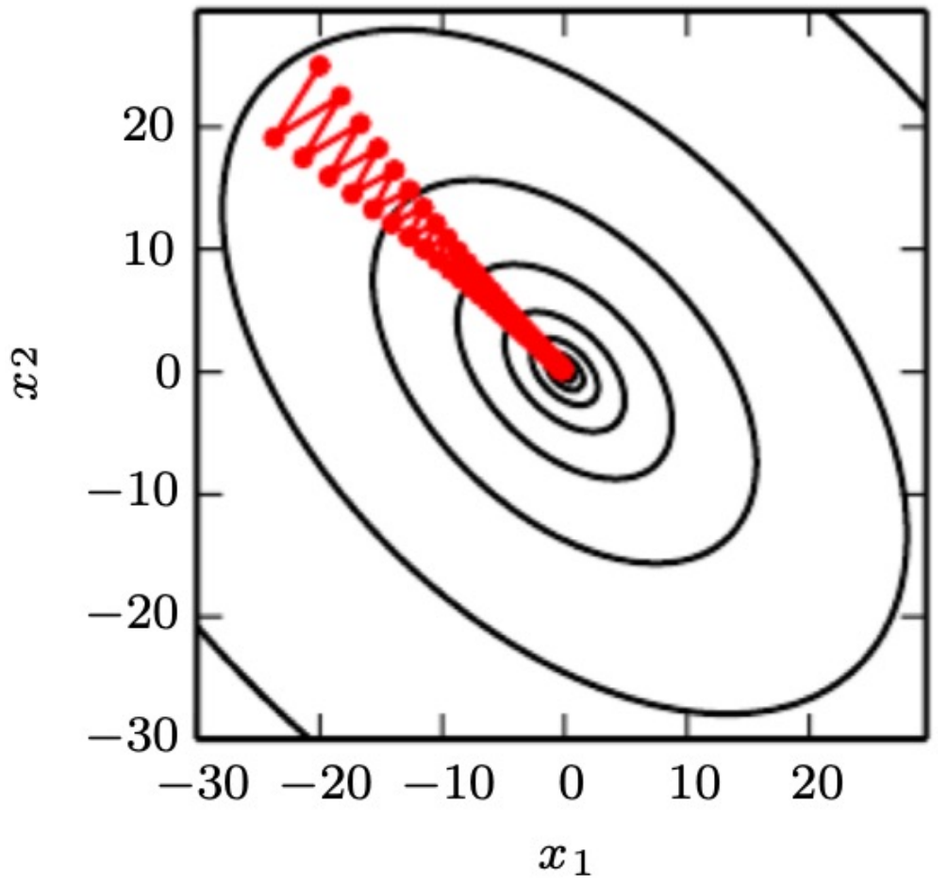
Challenges in Neural network Optimization

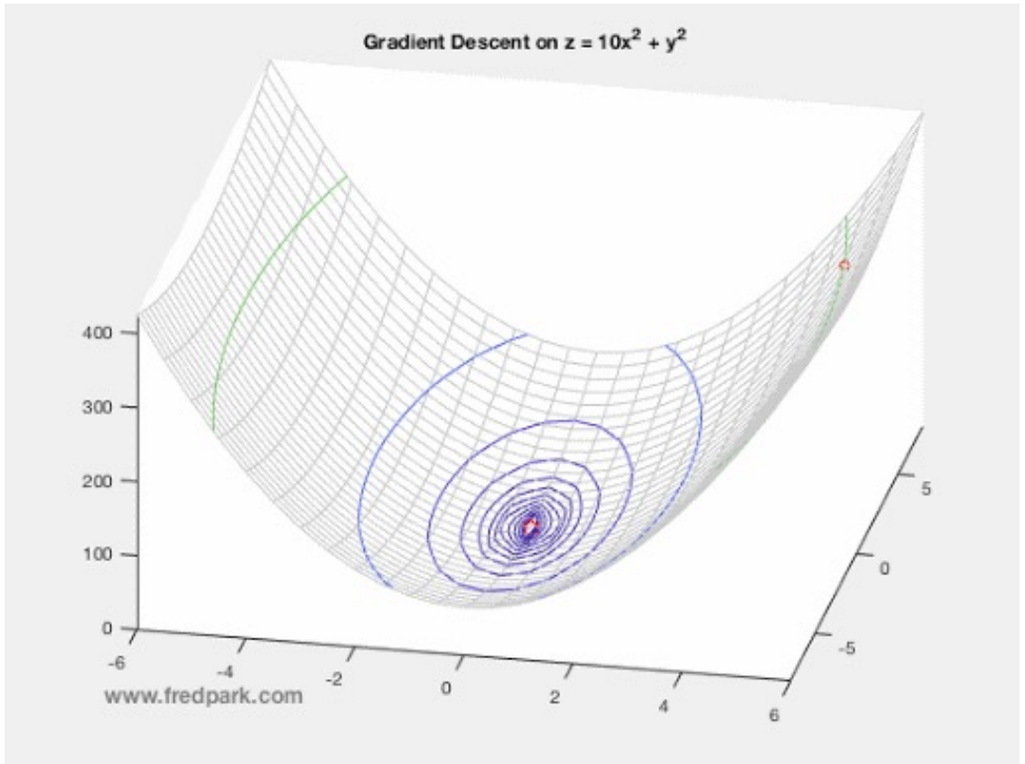
Ill-conditioning of Hessian

- Ill-conditioning of the Hessian Matrix
- When the matrix of second derivatives have a large variance
  - First-order methods (like SGD) may get stuck because gradient does not carry enough information about the curvature of the loss function
  - Even small steps in the gradient direction may result in an increase of the cost function
- Second order methods may solve this problem
  - But they’re not widespread in NN training

Ill-conditioning of Hessian

- Quadratic function with condition number 5 (poorly conditioned)
- The direction of most curvature has 5 times more curvature than the direction of least curvature
- The function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature



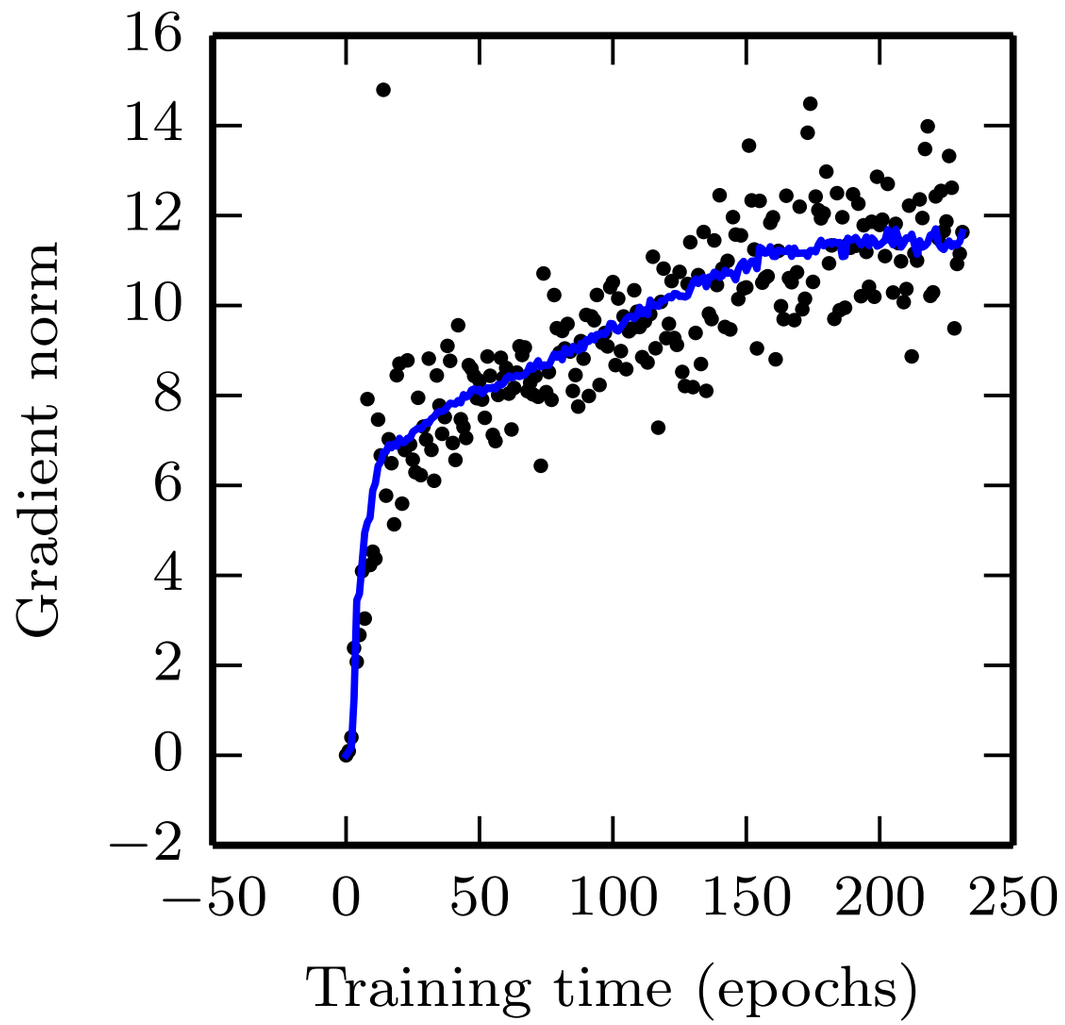


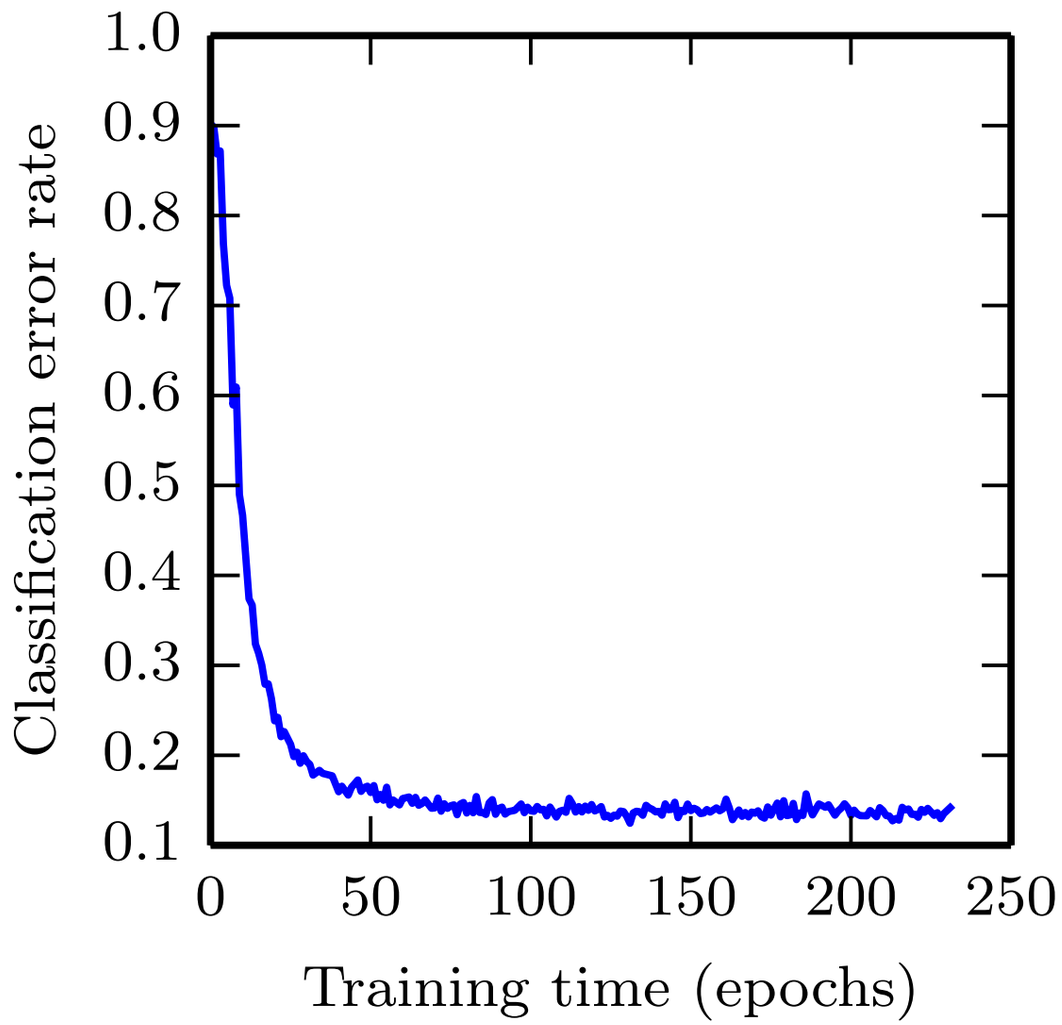
Local Minima

- **Convex** optimization: all local minima are global minima (flat region)
  - Any critical point is a good solution
- **Nonconvex** optimization: many non-optimal local minima
- A neural network may have an extremely large, or infinite number of local minima (due to symmetries and weight scaling)
  - Local minima of high cost are problematic
  - For many years this was believed to be a problem for NN optimization
  - Recently, this is not believed to be a problem anymore
    - It is sufficient to find a point with low cost

Gradient norm

- To check if we reached a critical point, plot the norm of the gradient
  - Many times, the network does not reach a minimum at all





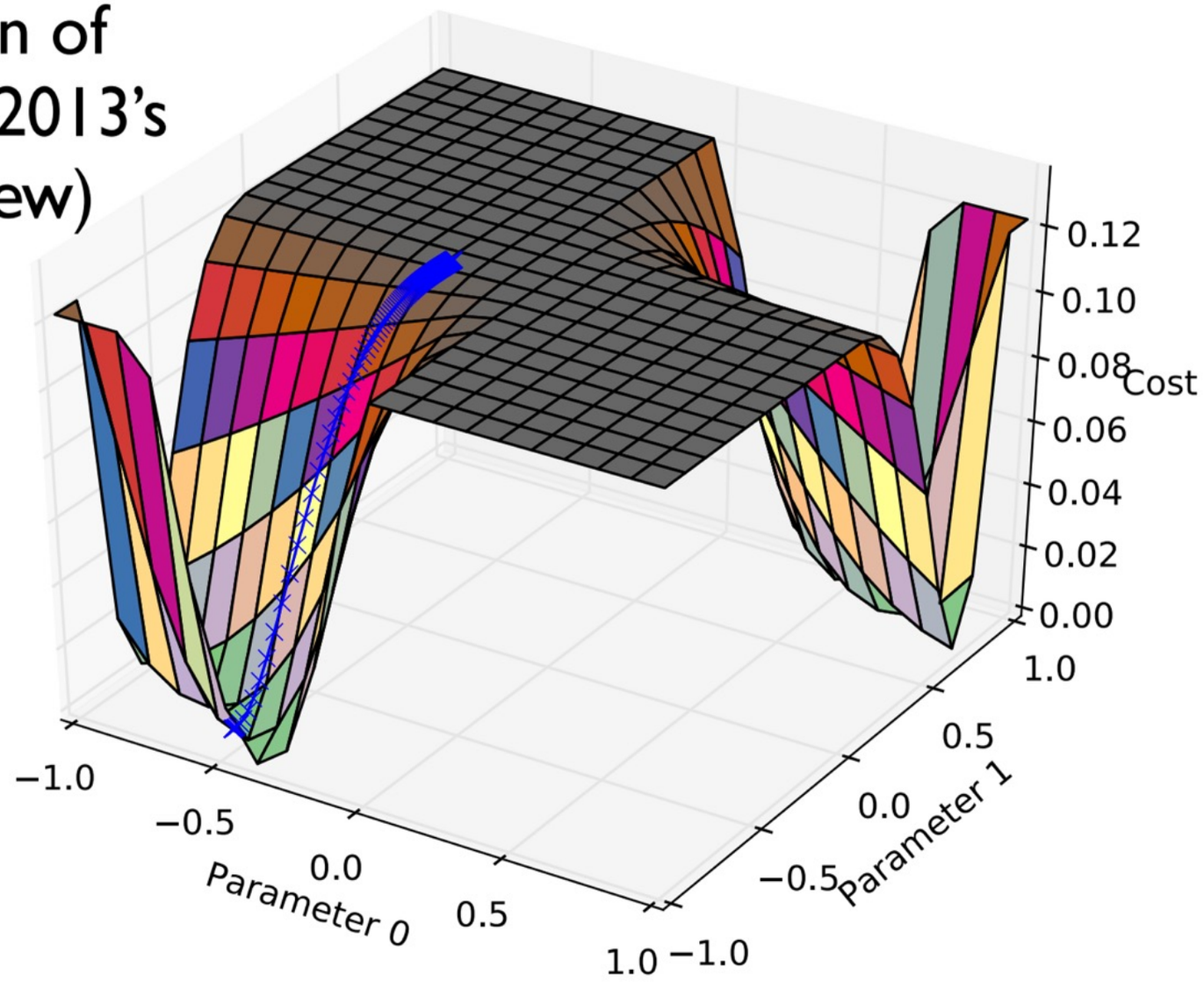
Flat regions

- In high-dimensional spaces Saddle points are more common than local minima. Intuition:
  - Hessian matrix at local minimum has only positive eigenvalues
  - Hessian at saddle point has a mix of positive and negative eigenvalues
    - In n-dimensional space, it is **exponentially unlikely** that all eigenvalues are positive
- Eigenvalues of Hessian are more likely to be positive on low-cost regions
  - Local minima are likely to have low cost

Implications of Saddle points

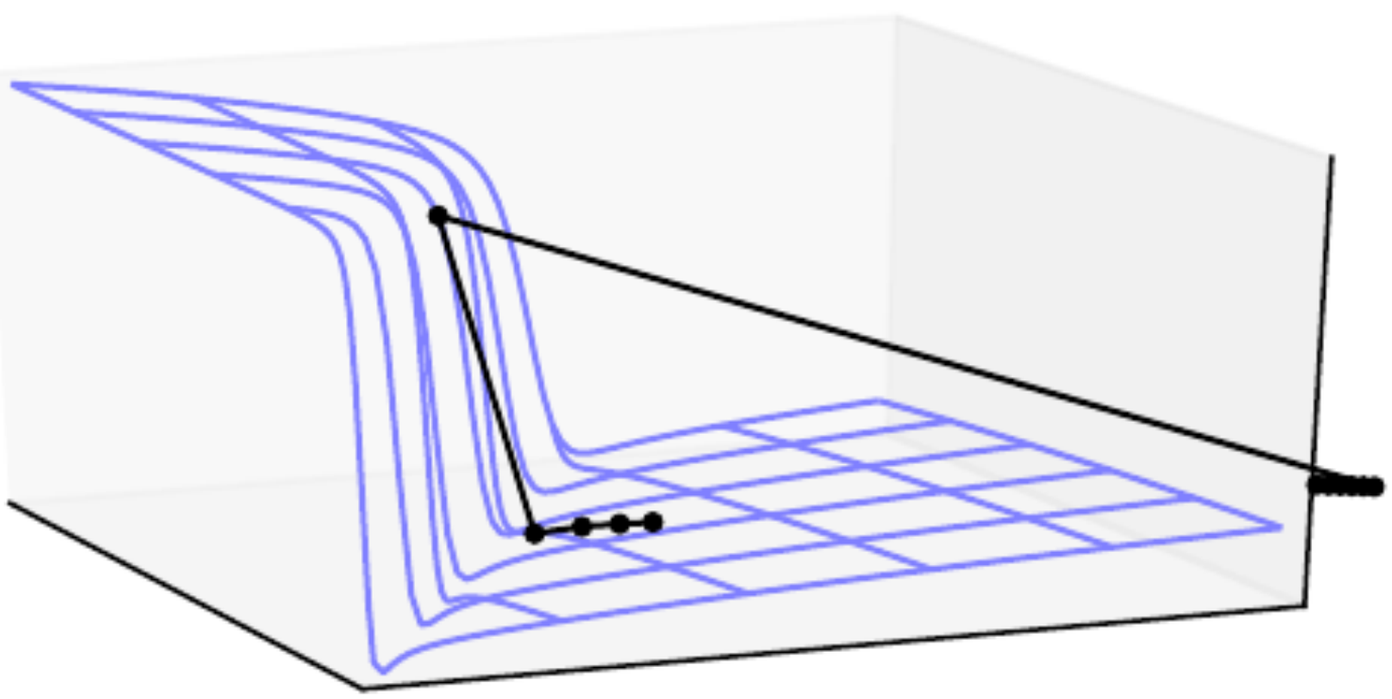
- First-order optimization algorithms
  - The gradient is small near a saddle point
  - Empirically, SGD seems able to escape saddle points quickly
- Second-order methods (e.g. Newton) find points with zero gradient
  - It can jump to a saddle point
- This is one reason why first-order methods are preferred in training deep NNs
- Flat regions are also a problem:
  - Both gradient and Hessian are zero

(Cartoon of Saxe et al 2013's worldview)



Cliffs and exploding gradients

- Deep networks have extremely steep regions (cliffs)
  - The gradient becomes big, and the weights can jump far away
  - Gradient clipping: reduces the step size



Exploding/Vanishing gradient

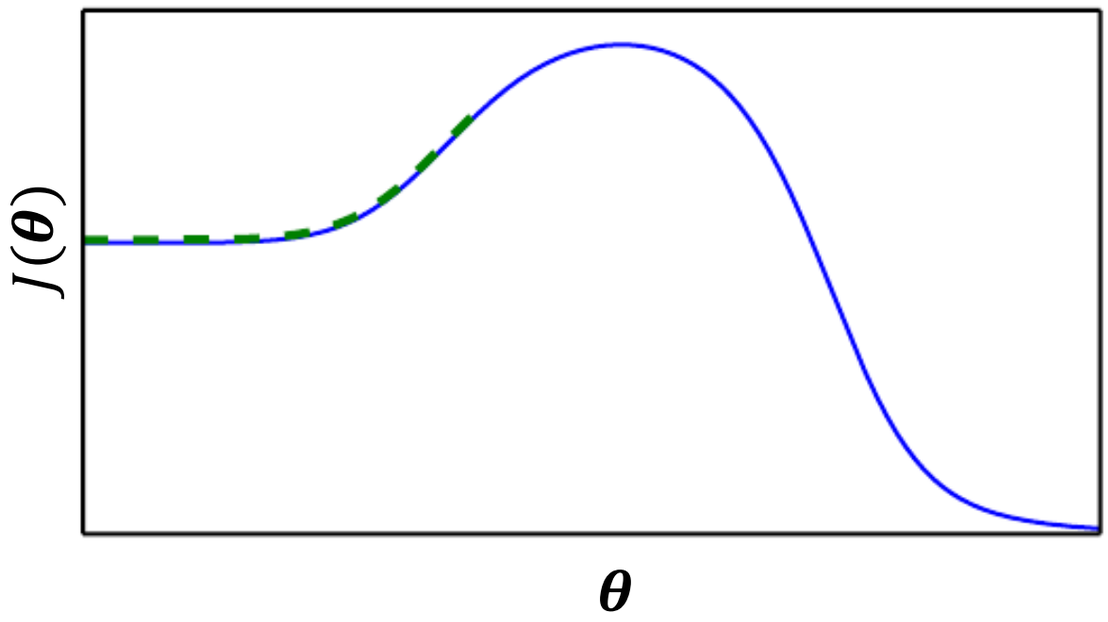
- Problematic for recurrent neural networks (second part of the course)
- Suppose we repeatedly multiply the input by a matrix $\boldsymbol{W} = \boldsymbol{V}\, diag(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$. After t steps, it's equivalent to multiply by $\boldsymbol{W}^t = \boldsymbol{V}\, diag(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}$(for linear activation)
  - If eigenvectors are not close to 1, they will **vanish** (become zero) or **explode**
  - Cliffs are an example of exploding gradient
- Same problem with activations that can saturate (e.g. sigmoid) providing gradient close to zero
- Feedforward networks use different matrices for each layer, and with non-saturating activations mainly avoid the problem

Inexact gradients

- Optimization algorithms assume access to true gradient or Hessian
  - In DL, we just have noisy/biased estimates

Local structure not representative of global structure

- Sometimes, optimization just don't reach any critical point
  - The problem in this case is bad initialization



Basic Optimization algorithm

- Stochastic gradient descent (**Minibatch**)
  - Main difference with Batch gradient descent: ϵ should decrease with time
    - The true gradient approaches 0 in a minimum, the stochastic estimate doesn't
  - E.g. linear decay (from ϵ_0 to ϵ_τ): $\epsilon_k = \left(1 - \frac{k}{\tau}\right)\epsilon_0 + \frac{k}{\tau}\,\epsilon_\tau$

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

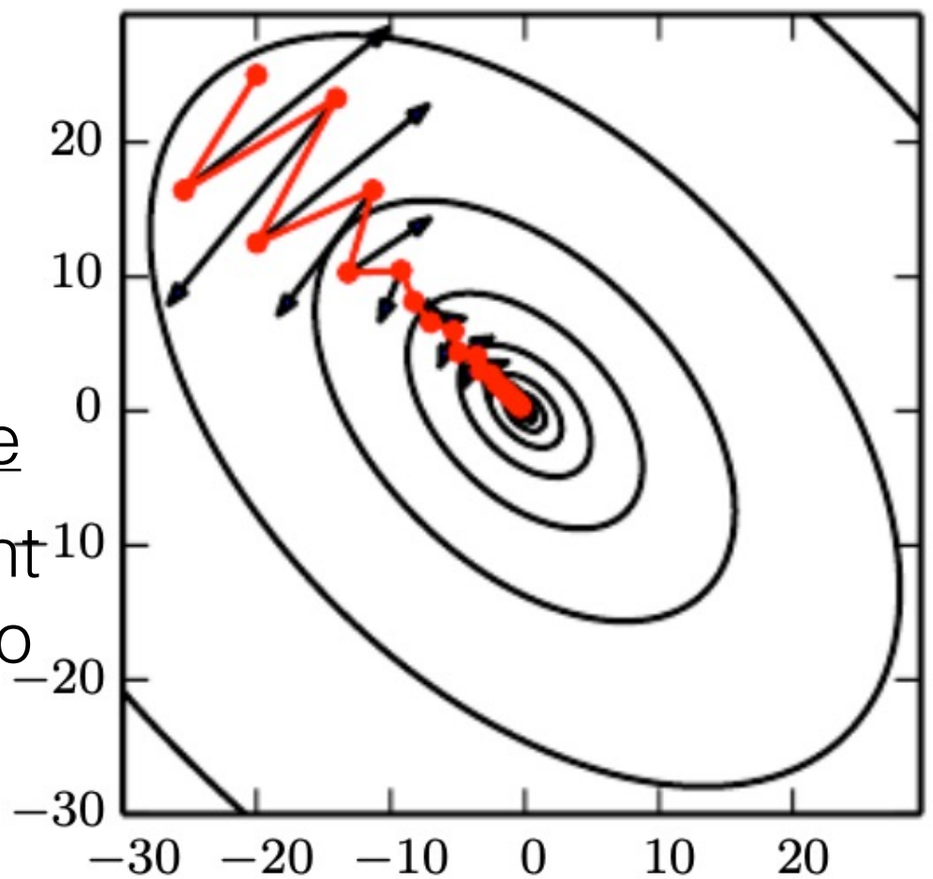
Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

Momentum

- SGD can be slow
- Momentum designed to accelerate learning in case of noisy gradients, or small but consistent gradients
- Accumulate an exponential moving average of past gradients and keep moving in that direction
  - Additional parameter: velocity $\boldsymbol{v}$

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}),$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

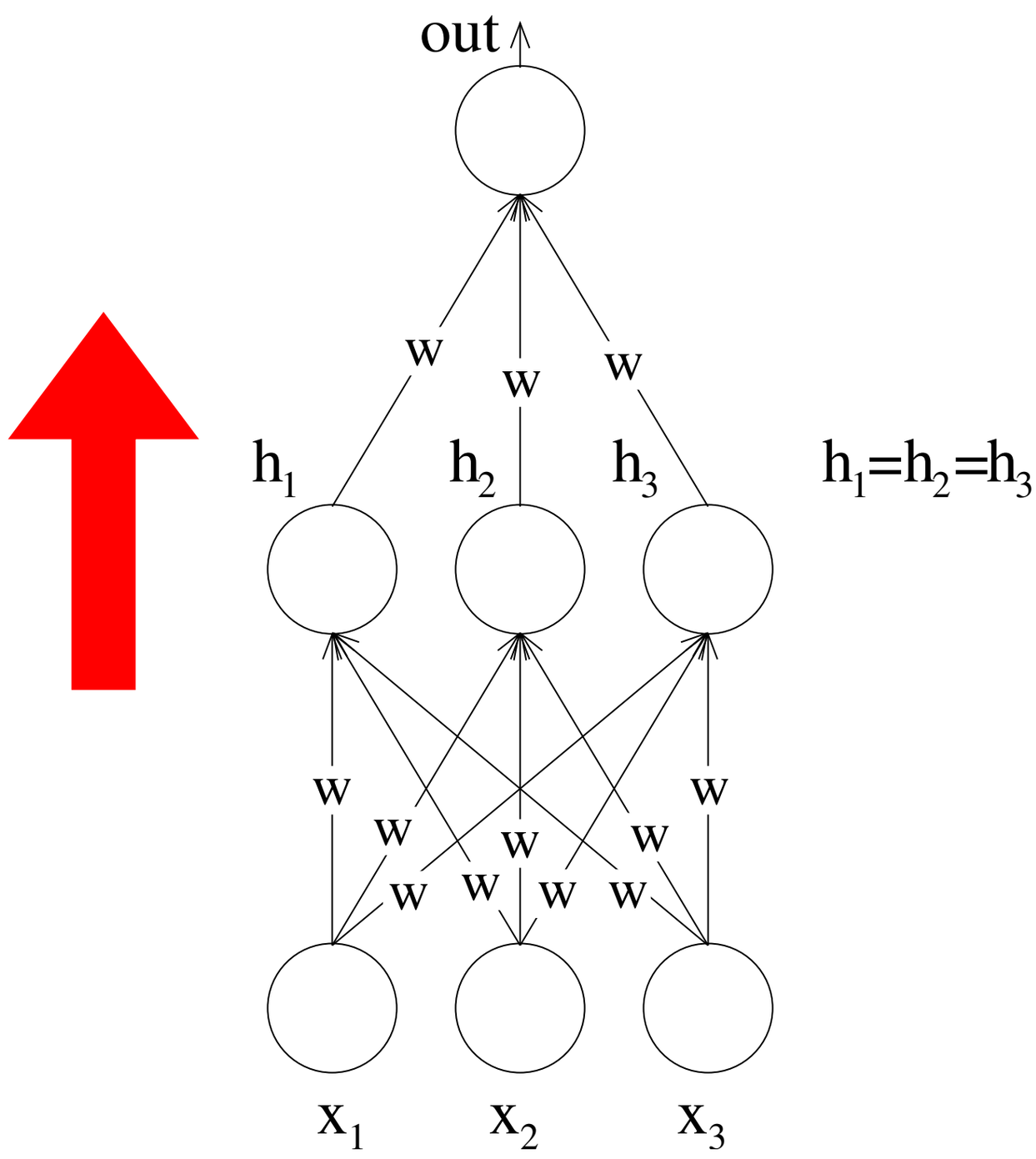
- The gradient step depends on how aligned the sequence of last gradients are
- **Nesterov** momentum: variant, with gradient evaluated after applying current velocity to $\boldsymbol{\theta}$ (i.e. at $\boldsymbol{\theta} + \alpha \boldsymbol{v}$)



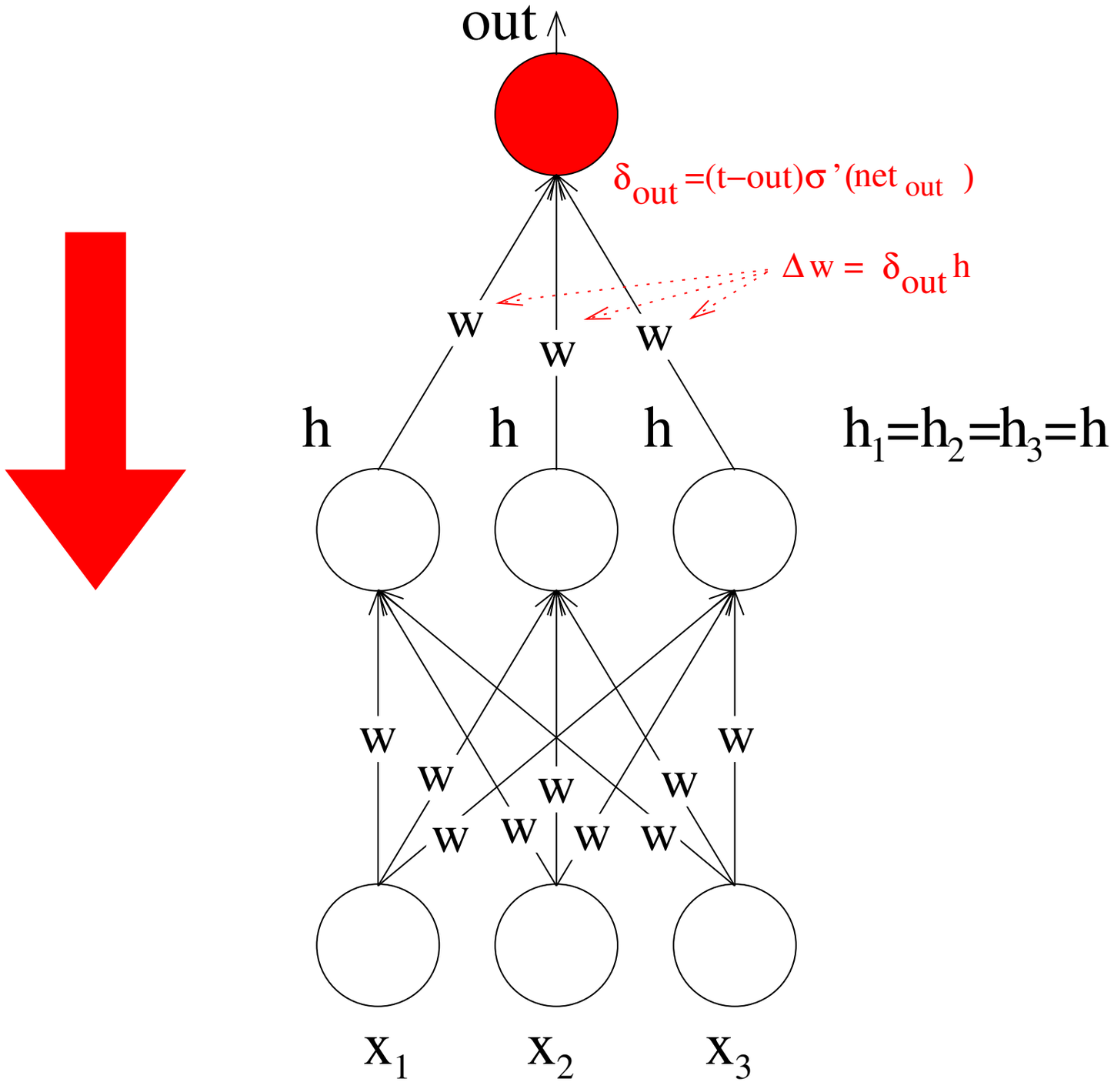
Parameter initialization

- DL training algorithms are iterative and depends on initialization
- Some bad initial points may let the algorithm fail due to numerical instabilities
- Starting points with similar training error may have different generalization error (different starting points -> different solutions)
- Current initialization strategies are simple and heuristic
- For sure: **break symmetry**

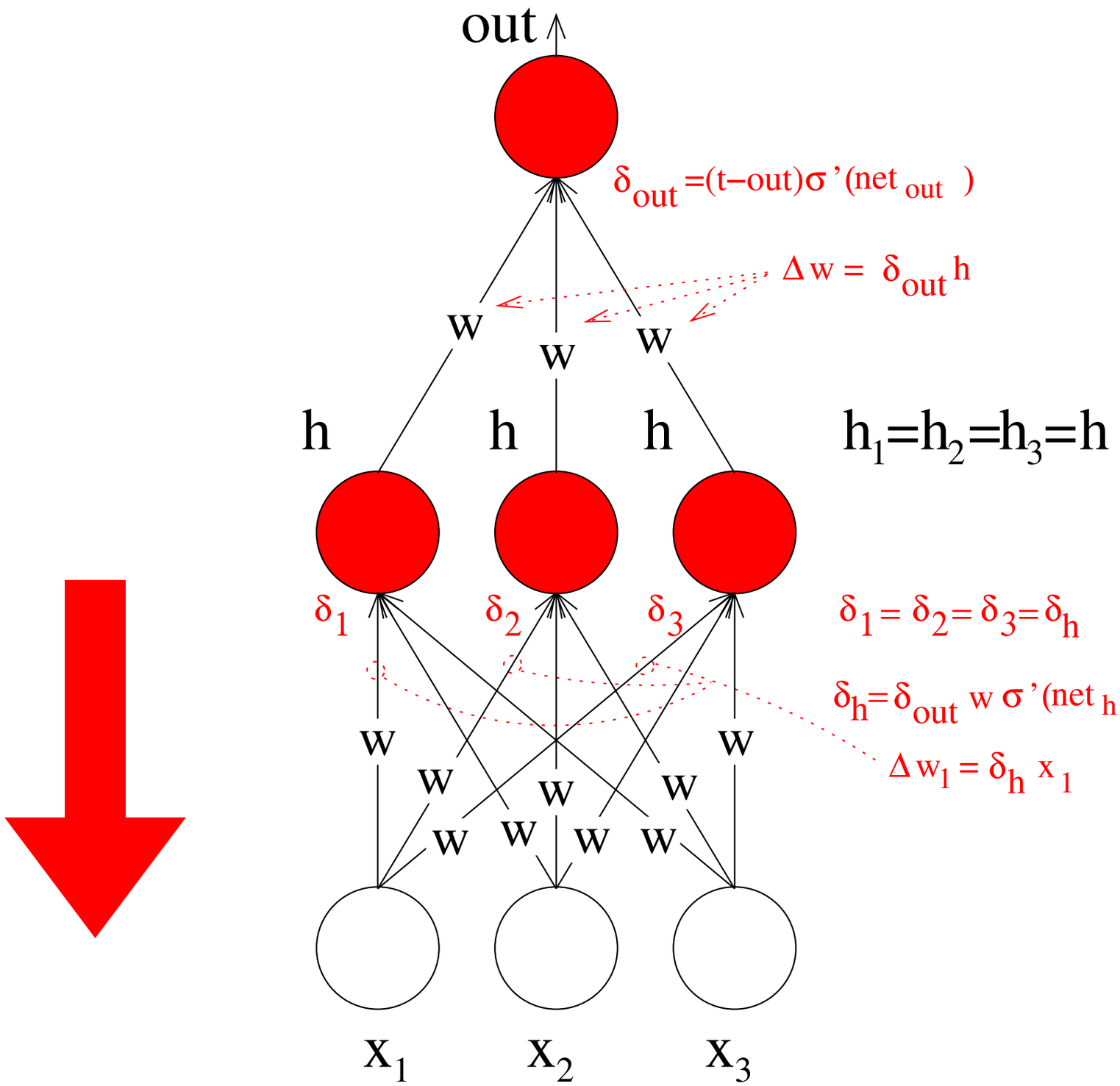
Symmetries



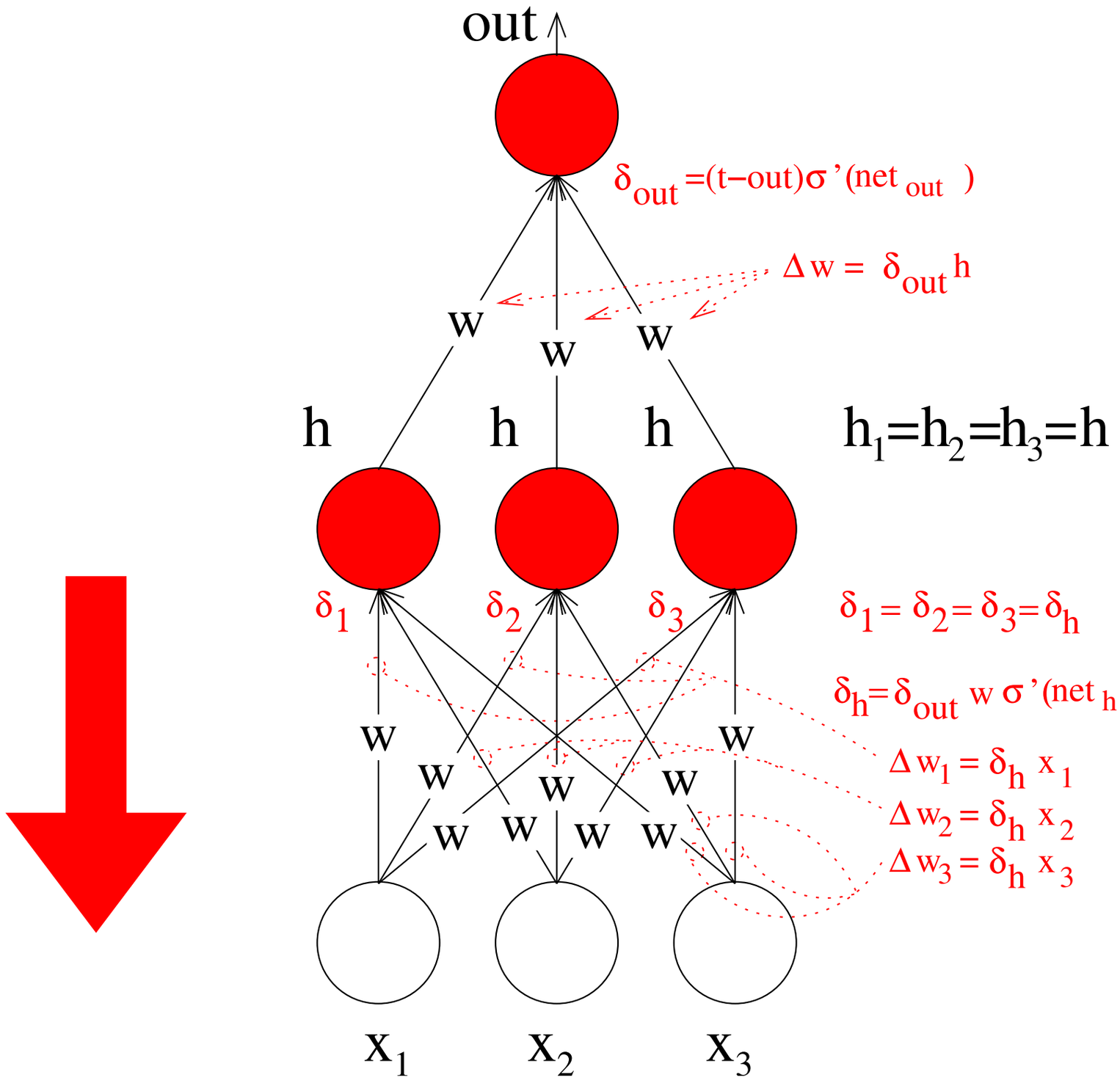
Symmetries


out
$\delta_{out}=(t-out)\sigma'(net_{out})$
$\Delta w = \delta_{out} h$
w
w
w
h
h
h
$h_1=h_2=h_3=h$
w
w
w
w
w
w
w
w
w
x_1
x_2
x_3

Symmetries


out
$\delta_{out}=(t-out)\sigma'(net_{out})$
$\Delta w = \delta_{out} h$
w
w
w
h
h
h
$h_1=h_2=h_3=h$
δ_1
δ_2
δ_3
$\delta_1=\delta_2=\delta_3=\delta_h$
$\delta_h=\delta_{out}\, w\, \sigma'(net_h)$
$\Delta w_1 = \delta_h\, x_1$
x_1
x_2
x_3

Symmetries


out
$\delta_{out}=(t-out)\sigma'(net_{out})$
$\Delta w = \delta_{out} h$
w
w
w
h
h
h
$h_1=h_2=h_3=h$
δ_1
δ_2
δ_3
$\delta_1=\delta_2=\delta_3=\delta_h$
$\delta_h=\delta_{out}\, w\, \sigma'(net_h)$
w
w
w
w
w
w
w
w
$\Delta w_1 = \delta_h\, x_1$
$\Delta w_2 = \delta_h\, x_2$
$\Delta w_3 = \delta_h\, x_3$
x_1
x_2
x_3

Parameter Initialization

- Random initialization for weights
  - **Gaussian** or **uniform** distribution (scale is important)
  - Large weights have a stronger symmetry breaking effect but..
  - Exploding values or gradient saturation
  - Gradient descent + early stopping is like imposing a gaussian prior around the initialization weights → init. close to $\mathbf{0}$
- Some heuristics for a FC layers with m inputs and n outputs:

$$W_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

  - Or Xavier **Glorot** (and Bengio) $W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$
  - If possible, the scale can be treated as a hyper-parameter
- Pre-defined constant for biases (usually 0 ot 0.1 for ReLU to avoid saturation)

Weights Initialization

- the variance of the outputs of each layer should be equal to the variance of its inputs
- the same should hold for the gradient
- trade-off: for each unit normalize the initial weights with respect to the number of incoming and outcoming connections

Examples of distributions (He inizialization) for inizialization according to the type of activation function (Normal distribution: mean = 0)

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

Pre-training

- Unsupervised pre-training:
  - Train an unsupervised model on the training data, use the resulting model as initialization
- Supervised pre-training (transfer learning):
  - Initialize the weights with a model trained on a related task
  - Especially effective with CNNs

Adaptive learning rates

- Idea: individual learning rate for each parameter (axis)
- Delta-bar-delta (1988), Batch learning, heuristic
  - Idea: if the partial derivative of the loss w.r.t. a parameter remains the same sign, the learning rate should increase
- More recently, Mini-batch algorithms

AdaGrad

- Scale learning rates of single parameters inversely proportional to the square root of the the sum of **all** historical squared values of the gradient
- Designed to converge rapidly when applied to convex problems

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSprop

- In AdaGrad, the accumulation from beginning of training results in **excessive decrease** in learning rate
- RMSprop adopts an **exponential moving average**
  - Discard history from ancient past
  - Idea: converge rapidly after finding a convex bowl

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Adam

- Momentum incorporated as an estimate of first and second order moments of the gradient (with exponential weighting)
- Bias correction to the moments to account for their initialization
- It is in general robust to the choice of hyper-parameters (learning rate may be tuned)

Second order methods

More efficient gradient descent optimization can reduce the vanishing gradient problem

- 1^{st} order gradient descent (black arrows) in RNN (i.e. Stochastic Gradient Descent) is not very effective because of regions of pathological curvature in the objective function $J(\mathbf{W})$

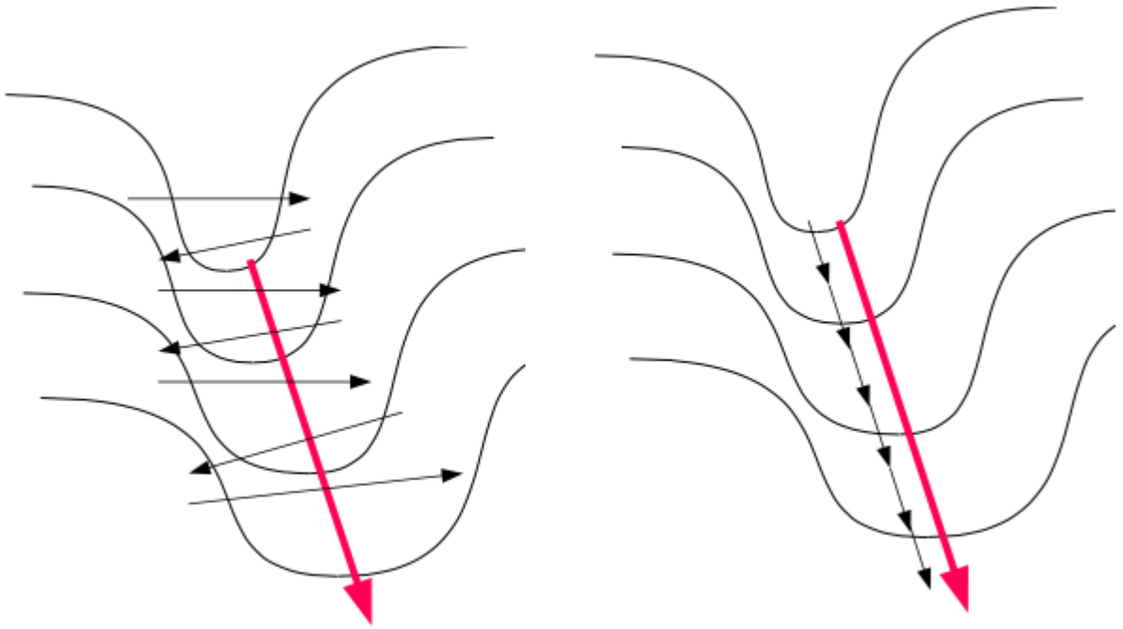


image taken from: Martens *ICML 2010*

- 2^{nd} order methods (red arrows), based on information about curvature, can do better, but are computationally expensive (computation of Hessian Matrix)

Newton's method

Taylor series: approximate $f(x)$ close to a point a

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2$$

- Consider the empirical risk

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{h}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

- Newton's method: second-order Taylor serie, starting from $\boldsymbol{\theta}_0$

$$J(\boldsymbol{\theta}+\boldsymbol{\delta}) = J(\boldsymbol{\theta}) + \boldsymbol{\delta}^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \frac{1}{2}\boldsymbol{\delta}^T \boldsymbol{H}(\boldsymbol{\theta})\boldsymbol{\delta}$$

First order — Second order

$$\boldsymbol{H}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2}{\partial \boldsymbol{\theta}_1^2} J(\boldsymbol{\theta}) & \cdots & \frac{\partial^2}{\partial \boldsymbol{\theta}_1 \partial \boldsymbol{\theta}_n} J(\boldsymbol{\theta}) \\ \cdots & & \\ \frac{\partial^2}{\partial \boldsymbol{\theta}_n \boldsymbol{\theta}_1} J(\boldsymbol{\theta}) & & \frac{\partial^2}{\partial \boldsymbol{\theta}_n^2} J(\boldsymbol{\theta}) \end{bmatrix}$$

Hessian (or curvature) matrix

- $\boldsymbol{H}_{ij}$ specifies how gradient in direction i changes as we move in direction j
- Steepest descent methods are not effective when off-diagonal elements are large (*disturbance* in simultaneous weight change)

Newton's method

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} J(\boldsymbol{\theta} + \boldsymbol{\delta})$$

The optimum step:

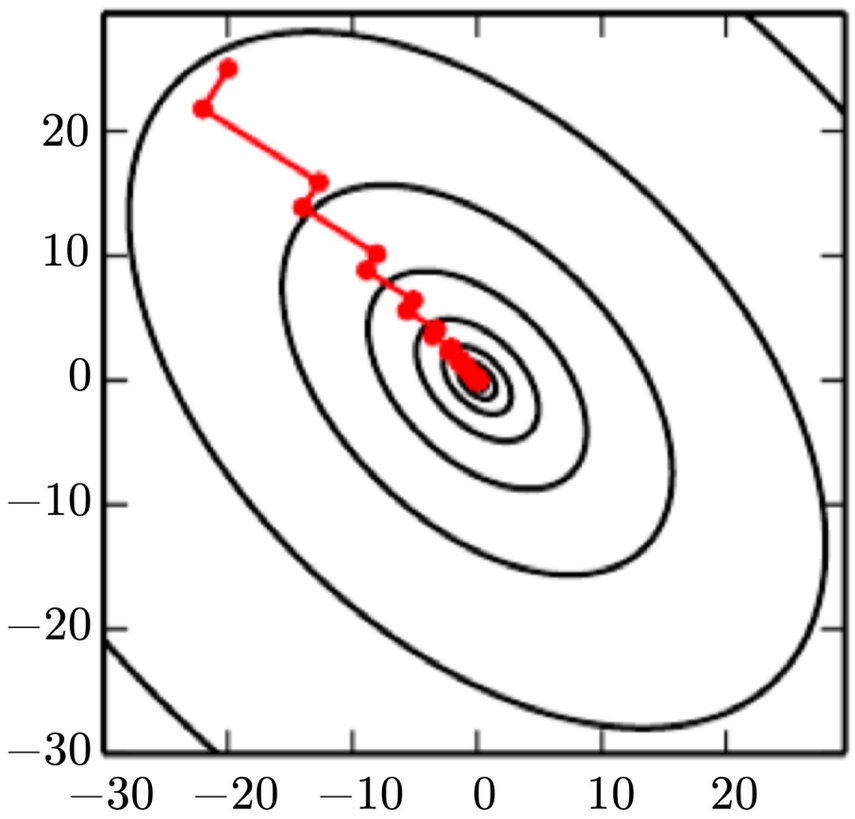
$$\frac{\partial J(\boldsymbol{\theta} + \boldsymbol{\delta})}{\partial \boldsymbol{\delta}} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}\ \ ) + \boldsymbol{H}(\boldsymbol{\theta})\boldsymbol{\delta} = \mathbf{0} \rightarrow \boldsymbol{\delta}^* = \boldsymbol{H}^{-1}(\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- If the function is quadratic, Newton's method finds directly the minimum
- If not, it can be used as update rule
- Computing $\boldsymbol{H}$ and $\boldsymbol{H}^{-1}$ is unfeasible for medium-sized networks

Hessian-free optimization

- Avoids the calculation of $\boldsymbol{H}^{-1}$
- Extension of the method of Steepest Descent
  - Variant of gradient descent with no fixed learning rate
  - Jumps to the point of lowest cost among the line defined by the gradient (line search, e.g. evaluating the function for different possible learning rates)

- Zig-Zag pattern: following gradient in one direction undoes progress in previously considered directions



Hessian-free optimization

- Conjugate directions: (for quadratic surfaces) ensures that the gradient along previous direction does not increase

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}$$

- Two directions are conjugate if

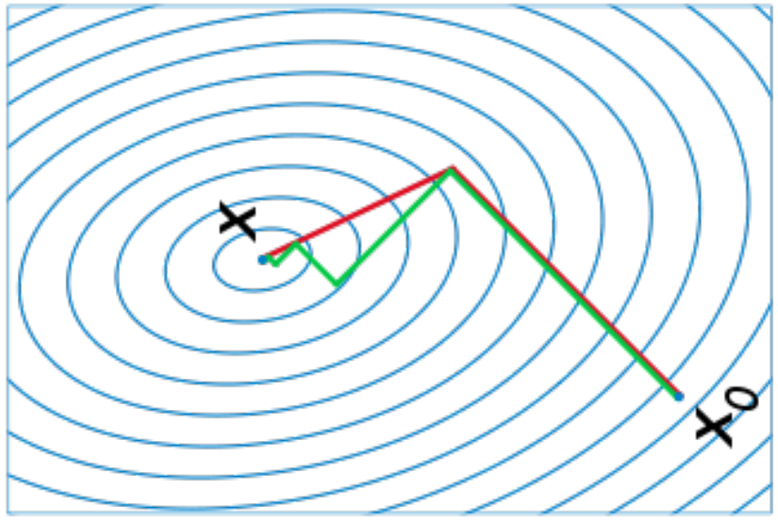
$$\boldsymbol{d}_t^T \boldsymbol{H} \boldsymbol{d}_{t-1} = 0$$

- However, we can compute the correct β_t without computing $\boldsymbol{H}$ (efficiency)
- (for quadratic surfaces in k dimensions) conjugate gradients method requires at most k steps (line searches) to achieve the minimum
- can be adapted for non-quadratic surfaces

Hessian Free Optimization

Intuition on how Conjugate Gradient works

- since $J(\boldsymbol{\theta} + \boldsymbol{\delta})$ is approximated, there is no hope to get the (local) minimum in one step
- an alternative is to perform multiple update steps, each of which finds the minimum along one direction
- however, avoid to undo previous work by choosing a new "conjugate" direction, i.e. which does not change the gradients in the previous directions



- the second direction in red has zero gradient with respect to the first direction in red (steepest descent is show in green)

Batch Normalization

- LeCun et al. (1998) showed that normalizing the inputs speeds up training
- Loffe and Szegedy (2015) proposed Batch Normalization to normalize hidden (pre-)activations
  - each unit's pre-activation is normalized (mean subtraction, standard deviation division)
  - during training, mean and standard deviation is computed for each mini-batch
  - backpropagation takes into account the normalization
  - at **test** time, the **global** mean and global standard deviation is used

Batch Normalization

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

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

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

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

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

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

stdev mean

Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

Batch Normalization

- Why normalize the pre-activation ?
  - can help to keep the pre-activation in a non-saturating regime

- At test time use the global mean and global standard deviation
  - removes the stochasticity of the mean and standard deviation
  - requires a final phase where, from the first to the last hidden layer:
    - propagate all training data to that layer
    - compute and store the global mean and global standard deviation for each unit
  - a running average can be used