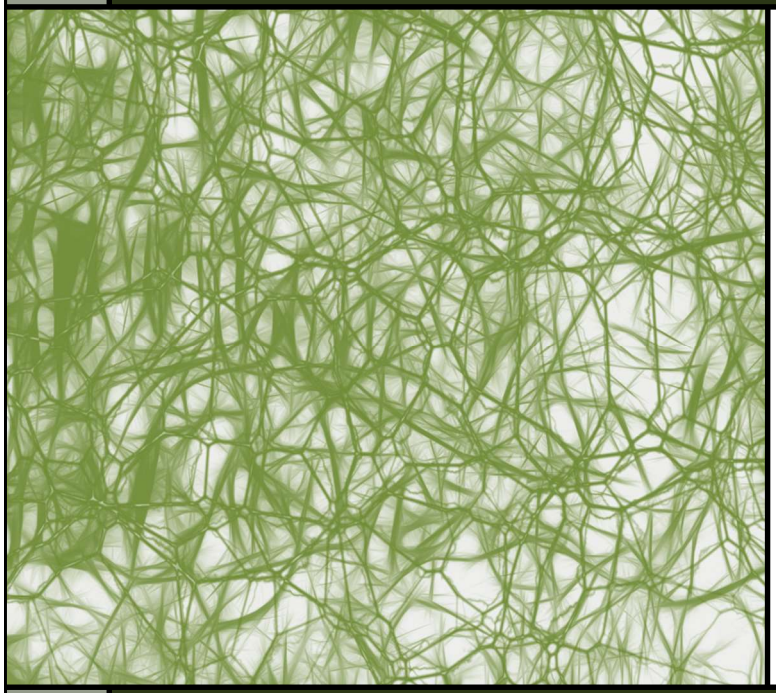
Department of Computer Science
University of Bristol

COMSM0045 – Applied Deep Learning 2020/21

comsm0045-applied-deep-learning.github.io



Lecture 04

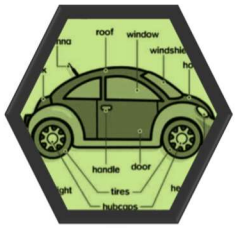
OPTIMISATION TECHNIQUES

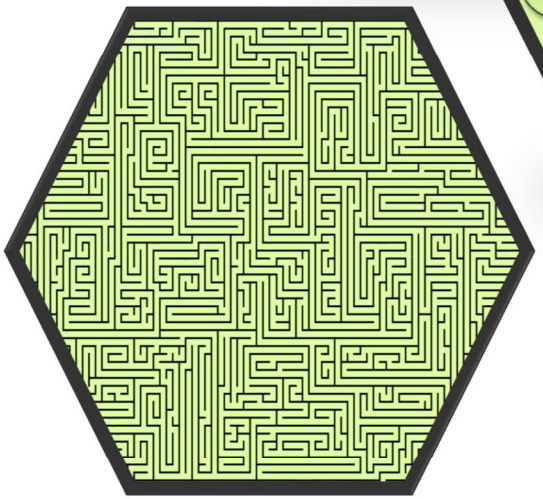
Tilo Burghardt | tilo@cs.bris.ac.uk

28 Slides

Agenda for Lecture 4

- Recap Backpropagation
- Optimisation Techniques





Recap: Backpropagation Algorithm

Recap: Backpropagation Algorithm

initialise all weights w_{ij}^l randomly

for $t=0,\ 1,\ 2,\ \ldots$ **do**

pick next training sample $([f_1^0, f_2^0, \ldots], [f_1^*, f_2^*, \ldots])$

FORWARD PASS: compute all $s_j^l = \sum_{i=1}^{d(l-1)} w_{ij}^l f_i^{l-1}$ and $f_j^l = g_j^l(s_j^l)$

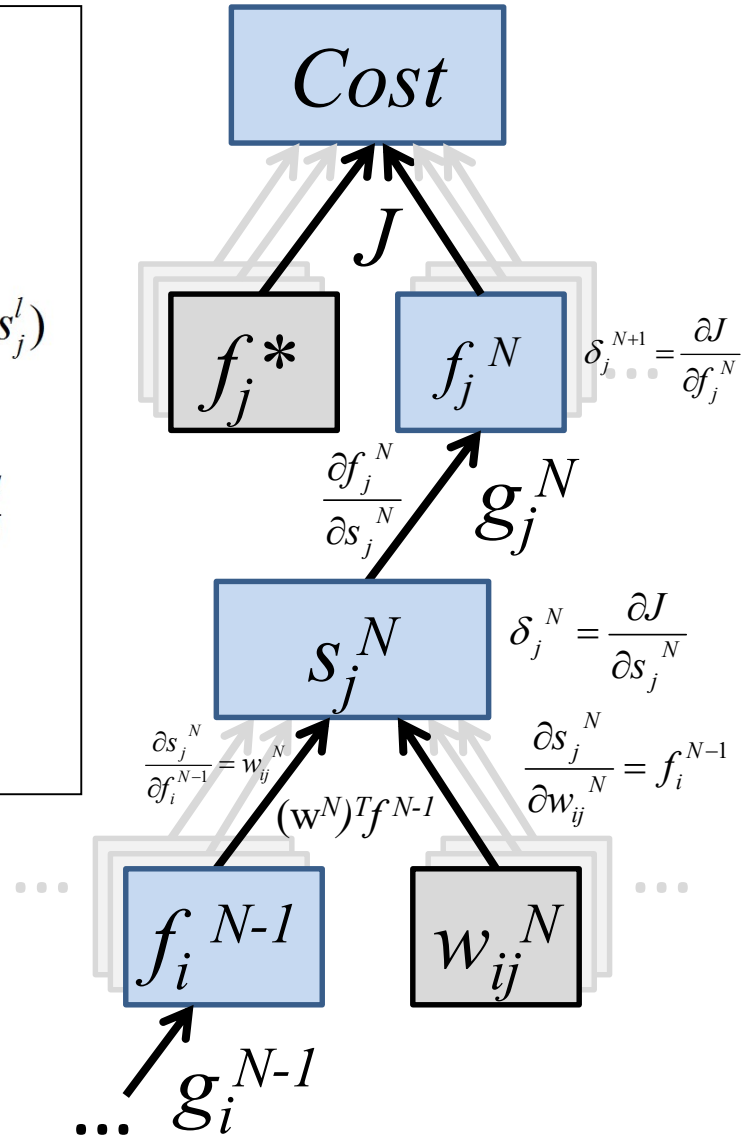
compute top deltas $\delta_j^N = g'^N_j(s_j^N) \cdot \partial J / \partial f_j^N$

BACKWARD PASS: compute all $\delta_i^{l-1} = g'^{l-1}_i(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$

update weights $w_{ij}^l \leftarrow w_{ij}^l - \eta\ f_i^{l-1} \delta_j^l$

check if stopping criteria are met to break loop

return final weights w_{ij}^l



SGD

(Online) Backpropagation so far: Notational Compaction

initialise all weights w_{ij}^l randomly

for $t=0,\ 1,\ 2,\ \ldots$ **do**

pick next training sample $([f_1^0, f_2^0, \ldots], [f_1^*, f_2^*, \ldots])$

FORWARD PASS: compute all $s_j^l = \sum_{i=1}^{d(l-1)} w_{ij}^l f_i^{l-1}$ and $f_j^l = g_j^l(s_j^l)$

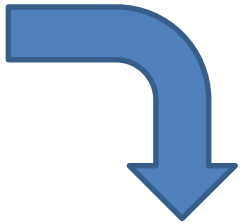
compute top deltas $\delta_j^N = g'^N_j(s_j^N) \cdot \partial J / \partial f_j^N$

BACKWARD PASS: compute all $\delta_i^{l-1} = g'^{l-1}_i(s_i^{l-1}) \sum_{j=1}^{d(l)} w_{ij}^l \delta_j^l$

update weights $w_{ij}^l \leftarrow w_{ij}^l - \eta\ f_i^{l-1} \delta_j^l$

check if stopping criteria are met to break loop

return final weights w_{ij}^l



initialise all weights W randomly

for $t=0,\ 1,\ 2,\ \ldots$ **do**

pick next training sample $(\mathrm{x}, \mathrm{f}^*)$

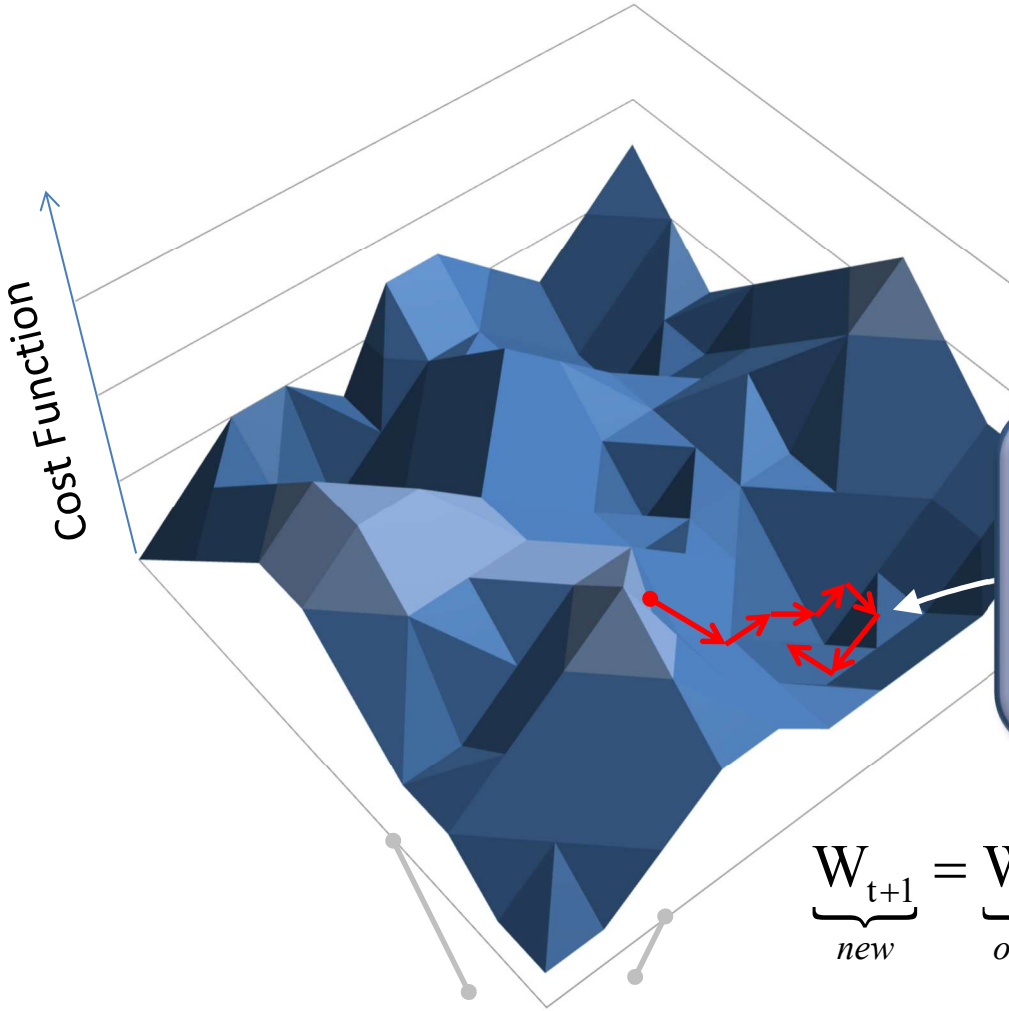
FORWARD-BACKWARD PASS: compute ∇J

update weights $\mathrm{W} \leftarrow \mathrm{W} - \eta \nabla J$

check if stopping criteria are met to break loop

return final weights W

Noisy Gradient Descent due to Online Sampling



Can we actually reach an exact minimum?

single samples will only very roughly approximate aspects of the cost function gradient in online mode, leading to a very noisy gradient descent which may not find the accurate minimum at all

$$\underbrace{W_{t+1}}_{new} = \underbrace{W_t}_{old} - \underbrace{\eta}_{learning\ rate} \underbrace{\nabla J(x; W_t)}_{steepest\ gradient\ given\ x}$$

Online, Deterministic and Stochastic Training

Can we actually reach an exact minimum?

ONLINE GRADIENT DESCENT

initialise all weights W randomly

for $t=0,\ 1,\ 2,\ \ldots$ **do**

pick next training sample $(\mathrm{x}, \mathrm{f}^*)$

FORWARD-BACKWARD PASS: compute ∇J

update weights $\mathrm{W} \leftarrow \mathrm{W} - \eta \nabla J$

check if stopping criteria are met to break loop

return final weights W

(MINIBATCH) STOCHASTIC GRADIENT DESCENT

initialise all weights W randomly

for $t=0,\ 1,\ 2,\ \ldots$ **do**

pick a small subset of training samples $(\mathrm{X}, \mathrm{F}^*)$

FORWARD-BACKWARD PASS: compute ∇J

update weights $\mathrm{W} \leftarrow \mathrm{W} - \eta \nabla J$

check if stopping criteria are met to break loop

return final weights W

DETERMINISTIC GRADIENT DESCENT

initialise all weights W randomly

for $t=0,\ 1,\ 2,\ \ldots$ **do**

use all training samples $(\mathrm{X}, \mathrm{F}^*)$

FORWARD-BACKWARD PASS: compute ∇J

update weights $\mathrm{W} \leftarrow \mathrm{W} - \eta \nabla J$

check if stopping criteria are met to break loop

return final weights W

given small enough learning rate DGD will make progress to true local minimum, but at high computational cost!

$$\nabla J = \frac{1}{|\mathrm{X}|} \nabla_{\mathrm{W}} \sum_j L(\mathrm{f}(\mathrm{x}_j, \mathrm{W}), \mathrm{f}^*)$$

Practical Solution: SGD with 'Simulated Annealing'

initialise all weights W randomly

for $k=0,\ 1,\ 2,\ \ldots\ \tau$ **do**

$$\eta_k = (1-\frac{k}{\tau})\eta_0 + \frac{k}{\tau}\eta_\tau$$

introduction of a changing learning rate η_k decreasing over $\tau+1$ steps by blending from a starting learning rate η_0 towards a final learning rate η_τ.

for $t=0,\ 1,\ 2,\ \ldots$ **do**

pick a small subset of training samples $(\mathrm{X}, \mathrm{F}^*)$

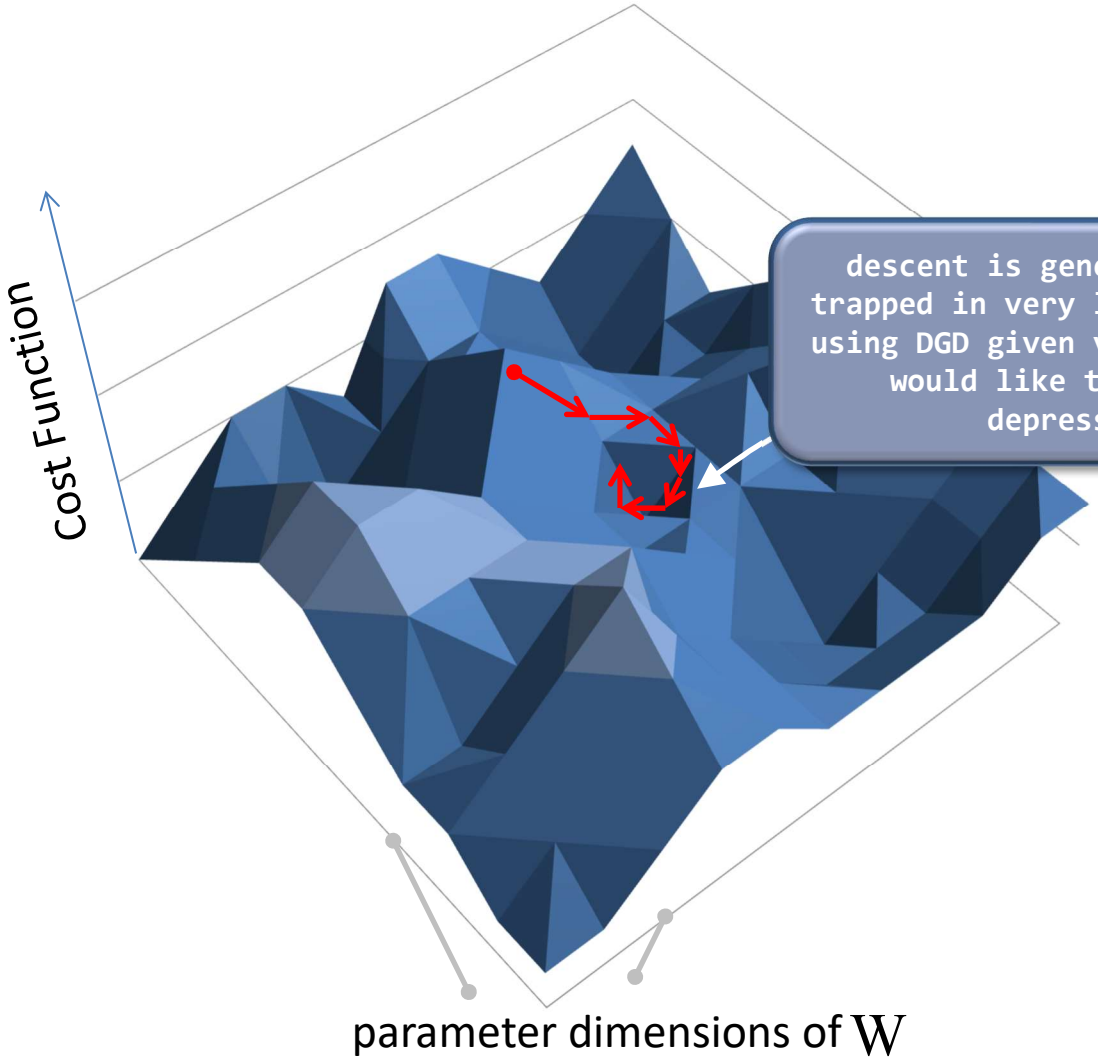
FORWARD-BACKWARD PASS: compute ∇J

update weights $\mathrm{W} \leftarrow \mathrm{W} - \eta_k \nabla J$

...

return final weights W

Slow Descent and Local 'Dips' of Cost Function



descent is generally very slow and may get trapped in very localised, shallow 'dips' when using DGD given very small learning rates – we would like to 'surf' over these local depressions on steep hills

Could we use physics principles to speed up descent?

Momentum

Speeding up Learning via Momentum

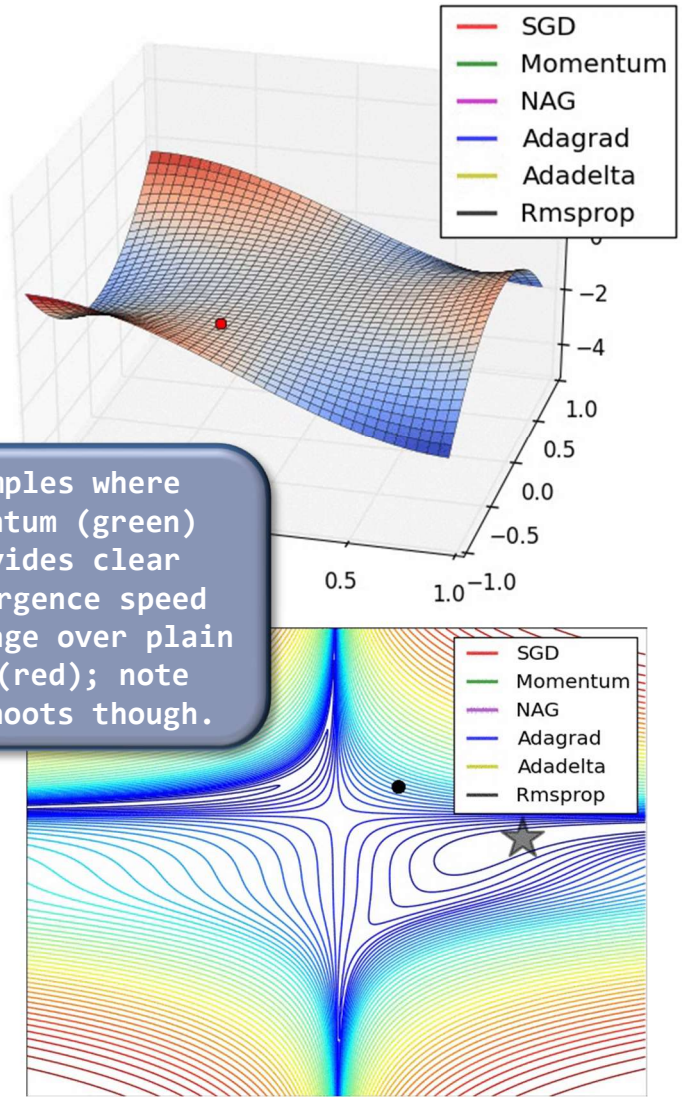
- **Idea:** introduce a velocity term v of 'current descent speed' and use current gradient to change this velocity rather than weights directly
- step sizes now depend on how large and how aligned a previous sequence of gradients has been
- formally, we change the update equations for weights from:

$$\mathrm{W}_{t+1} = \mathrm{W}_t - \eta \nabla J(\mathrm{X}; \mathrm{W}_t)$$

by introducing velocity accumulation:

$$v_{t+1} = \underbrace{\alpha}_{\text{momentum parameter}} v_t - \eta \nabla J(\mathrm{X}; \mathrm{W}_t)$$

$$\mathrm{W}_{t+1} = \mathrm{W}_t + \underbrace{v_{t+1}}_{\text{momentum}}$$



Examples where Momentum (green) provides clear convergence speed advantage over plain SGD (red); note overshoots though.

animation sources: Alec Radford

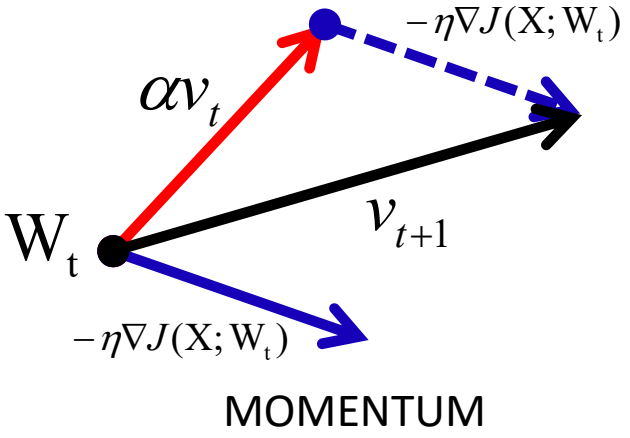
Nesterov Accelerated Gradient (NAG)

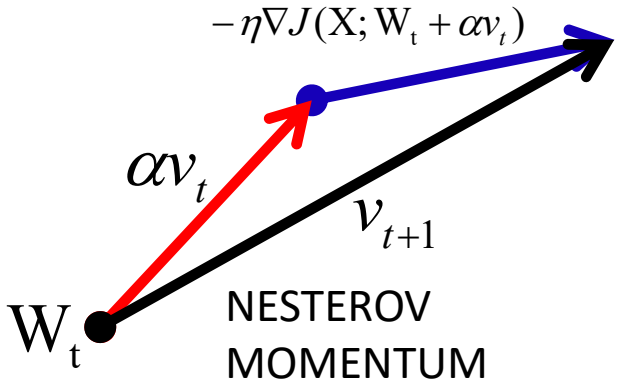
- **Idea:** don't calculate gradient at current position since momentum will carry us forward to another position anyway – take (lookahead) gradient at target
- can be seen as adding a 'correction term' to the standard method of momentum
- consistenly works slightly better than standard momentum in practice
- weights are now updated as follows:

$$v_{t+1} = \alpha v_t - \eta \nabla J(\mathrm{X};\ \underbrace{\mathrm{W_t} + \alpha v_t}_{\textit{preview location}}\ )$$

$$\mathrm{W_{t+1}} = \mathrm{W_t} + v_{t+1}$$

- however, still very slow progress on shallow plateau regions





Newton's Method

Newton's Method (2nd Order)

- **Idea:** let curvature rescale the gradient – multiplying the gradient by the inverse Hessian leads to an optimization that takes aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature
- great advantage: no extra learning rate or hyperparameters needed
- however, computing and inverting the Hessian is very expensive and space consuming (Hessian $\mathbf{H}$ has square size w.r.t. to number of weights!):

$$\mathrm{W}_{t+1} = \mathrm{W}_t - \mathbf{H}(J(\mathrm{X};\mathrm{W}_t))^{-1}\nabla J(\mathrm{X};\mathrm{W}_t)$$

- yet, *Newton's method* without modifications has a critical shortcoming: it is attracted to Saddle points...

  (see also "Hessian-free" 2nd-order methods)

RECAP: HESSIAN MATRIX

$$\mathbf{H}(J) = \begin{bmatrix} \cdots & \cdots & \cdots \\ \cdots & \dfrac{\partial^2 J}{\partial w_i \partial w_j} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}$$

Saddle Points as Critical Points

- There are various point categories of the objective function where the gradient is zero:
  - **Minima** (all Eigenvalues of Hessian positive),
  - **Maxima** (all Eigenvalues of Hessian negative),
  - **Saddle points** (both positive and negative Eigenvalues of Hessian)

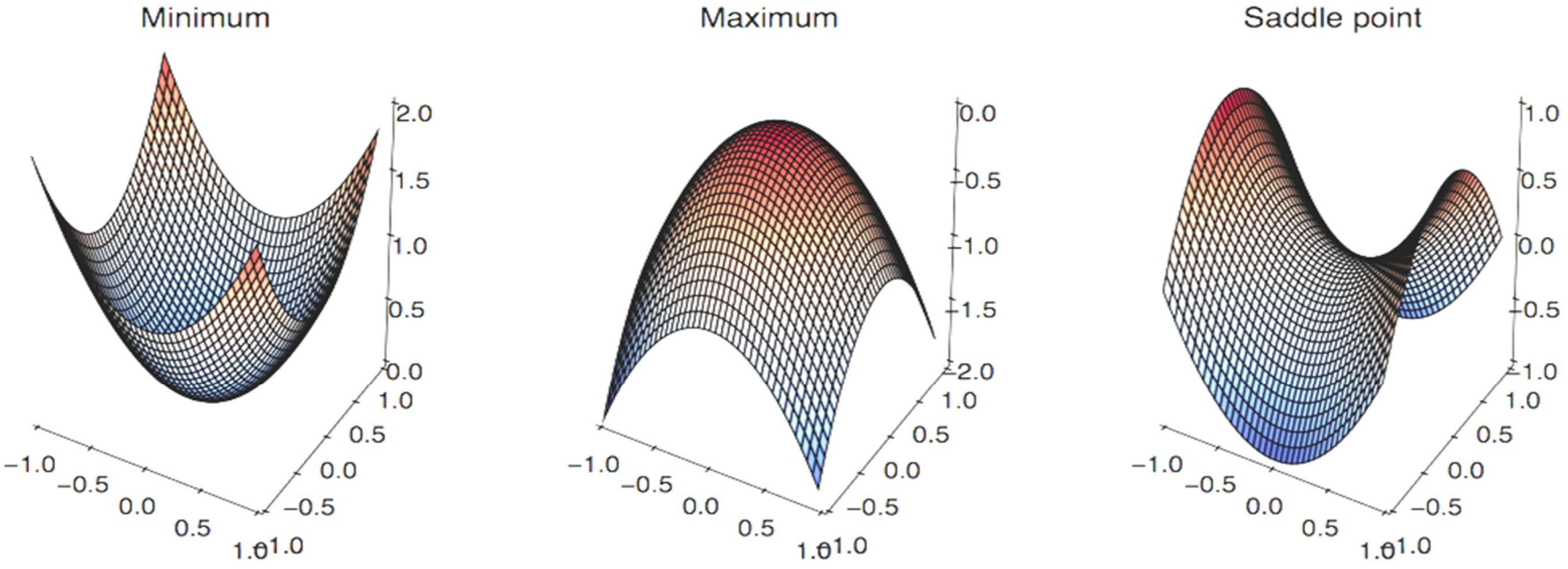
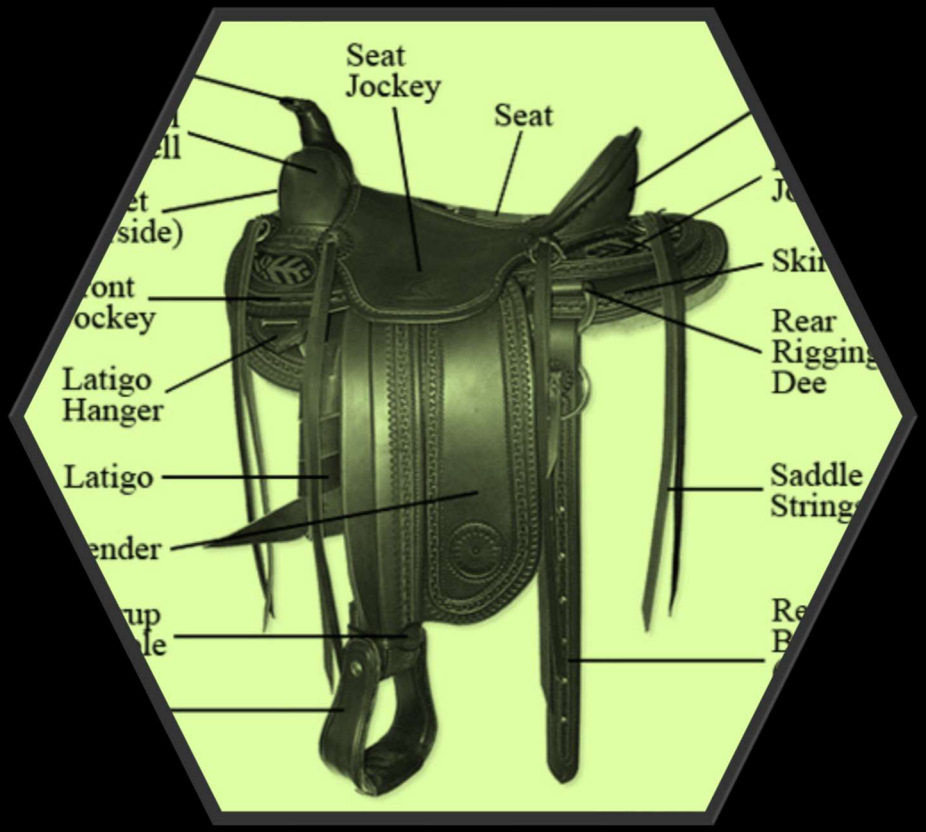


image adjusted from source: Ian Goodfellow 2015

Saddle Point Considerations



Are there more Saddle points or local Minima?

- for an arbitrary problem, assume sign of Hessian Eigenvalues is random:
  - exponentially less likely to get 'all positive' (i.e. being a Minimum) with higher and higher parameter dimensions
- Random Matrix Theory provides further insight:
  - the lower J is, the more likely to find positive Eigenvalues
- neural nets without non-linearities have global minima connected via a single manifold and many Saddle points (Saxe et al, 2013)

GOOD NEWS:

→ Most critical points with higher cost J should be Saddle points and they offer a chance to escape from them particularly via symmetry-breaking descent-methods!

→ Most local minima should therefore have a low cost J associated with them and *may* be reachable via descent!

High Number of Saddle points

- experiments and theoretical arguments (Dauphin et al 2014, Choromanska et al 2015) provide some support that neural nets have indeed as many Saddle points as Random Matrix Theory proposes
- in fact, the number of Saddle points may increase exponentially with the dimensionality of the function

MONKEY SADDLE

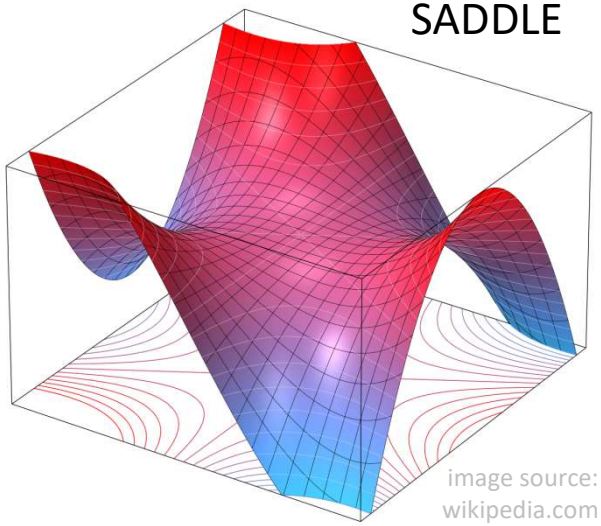


image source: wikipedia.com

NOT SO GREAT NEWS:

→ *Newton's method* will work poorly (since being attracted to Saddle points) with a high chance of getting stuck

→ however, idea of a function-adaptive learning rate seems valuable

SADDLE POINT EXAMPLES (white)

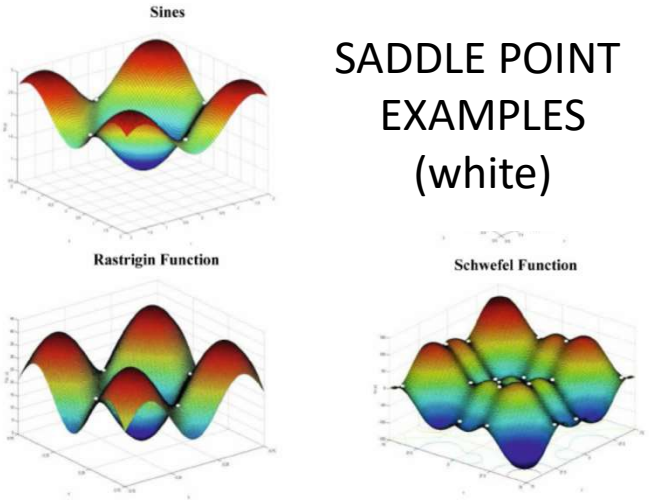


image source: Chen et al. 2012

Per-weight Adaptive Gradients

Adagrad$_{\text{(adaptive gradient)}}$ *(Duchi et al. 2011)*

- **Idea:** keep track of per-weight learning rates to force evenly spread learning speeds – weights that are associated with high gradients have their effective rate of learning decreased, whilst weights that have infrequent or particularly small updates have their rates increased.
- such `monotonic learning' may help with issues including breaking of symmetries and slow progress in particular dimensions
- update now uses a W_t-sized accumulator A:

$$\mathrm{A}_{t+1} = \mathrm{A}_t + (\nabla J(\mathrm{X}; \mathrm{W}_t))^2$$

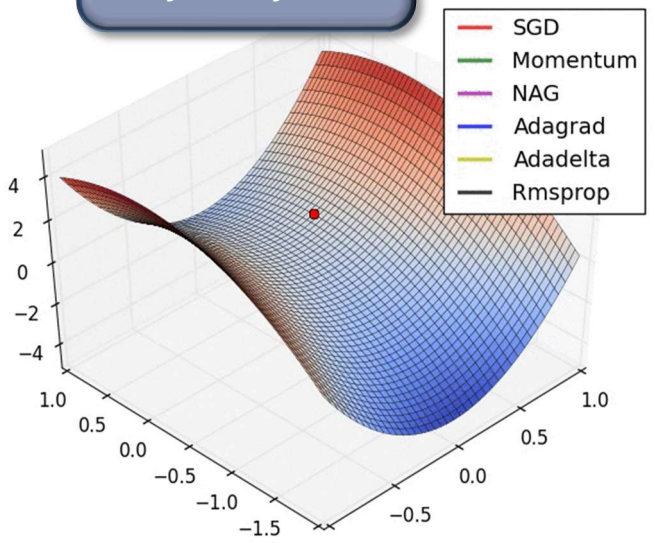
element-by-element squaring

$$\mathrm{W}_{t+1} = \mathrm{W}_t - \eta \frac{\nabla J(\mathrm{X}; \mathrm{W}_t)}{(\sqrt{\mathrm{A}_{t+1}} + \varepsilon)}$$

avoiding division by zero

- however, this `monotonic learning' is a very aggressive approach and lacks the possibility of late adjustments...learning usually stops too early...

Example where Adagrad breaks symmetry...



animation sources: Alec Radford

Concept: element-wise dampening of historically highly active gradient components (A being large) and amplification of slowly changing gradient components (A being small)

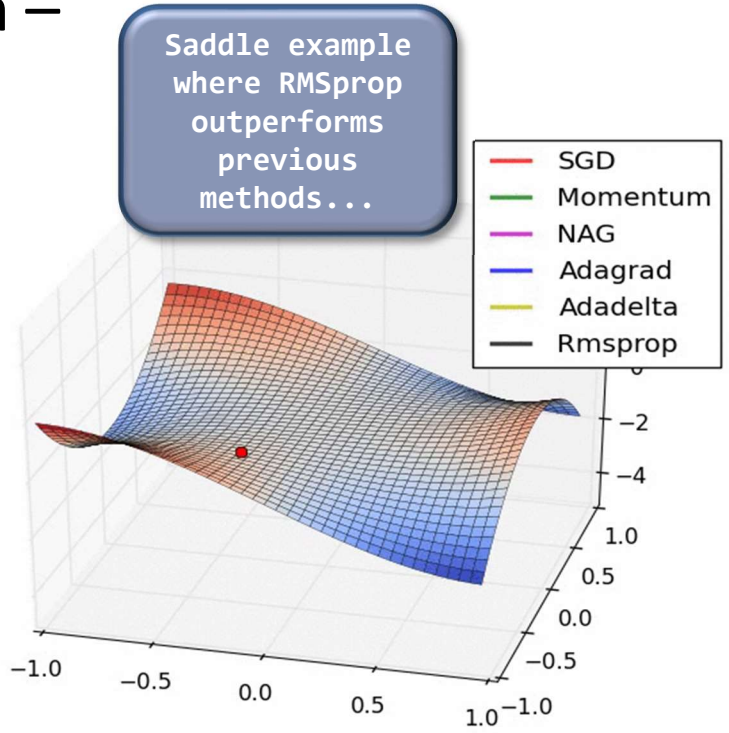
RMSprop *(Hinton "L.6 S.29")*

- **Idea:** root-mean-square propagation – combat the aggressive reduction in Adagrad's learning speed by propagation of a smooth running average
- update equations now introduce a smoothing parameter β:

$$A_{t+1} = \beta A_t + (1-\beta)(\nabla J(X; W_t))^2$$

$$W_{t+1} = W_t - \eta \frac{\nabla J(X; W_t)}{(\sqrt{A_{t+1}} + \varepsilon)}$$

- just adding standard momentum does not help much in improving performance further *(see Hinton)*
- however, further smoothing and correction operations can be applied...

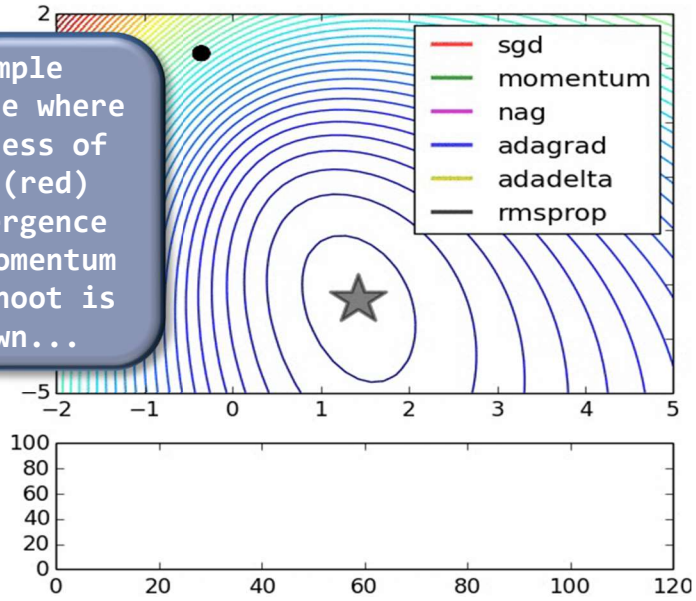


animation sources:
Alec Radford

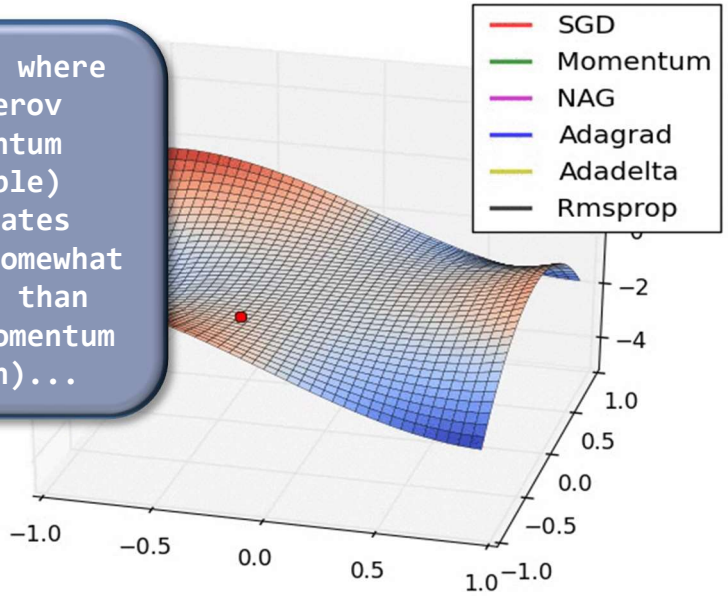
Concept: element-wise dampening of recently highly active gradient components (A being large) and amplification of slowly changing gradient components (A being small)

Some Observations in Convergence Visualisations

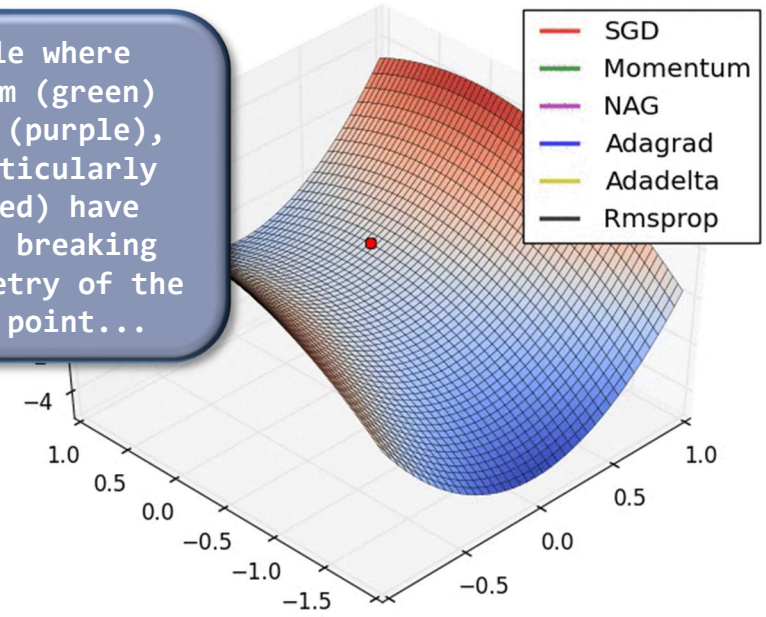
Simple example where slowness of SGD (red) convergence and Momentum overshoot is shown...



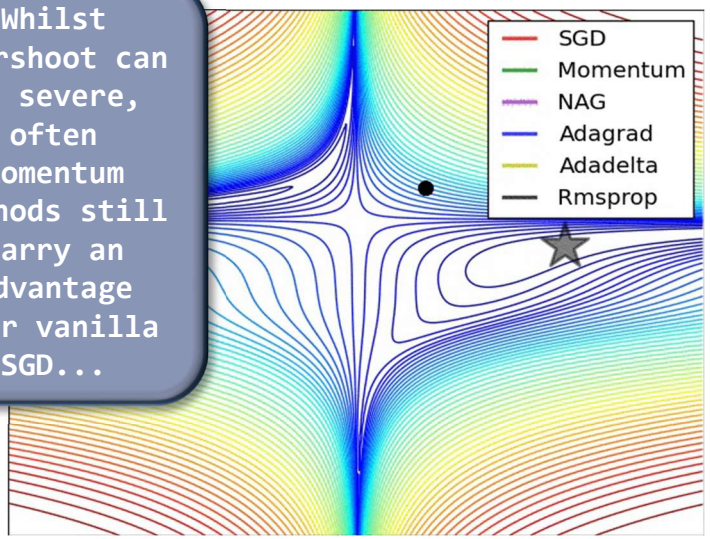
Example where Nesterov Momentum (purple) navigates saddle somewhat faster than plain Momentum (green)...



Example where Momentum (green) and NAG (purple), but particularly SGD (red) have trouble breaking the symmetry of the saddle point...



Whilst overshoot can be severe, often Momentum methods still carry an advantage over vanilla SGD...



animation sources:
Alec Radford

Adam

Adam(adaptive moment estimation) *(Kingma & Ba 2014)*

- **Idea 1:** smooth RMSprop's usually `noisy' incoming gradient (beyond the effect of mini-batching) using a new parameter α:

$$G_{t+1} = \alpha G_t + (1-\alpha)\nabla J(X; W_t))$$
$$A_{t+1} = \beta A_t + (1-\beta)(\nabla J(X; W_t))^2$$
$$W_{t+1} = W_t - \eta \frac{G_{t+1}}{(\sqrt{A_{t+1}} + \varepsilon)}$$

- **Idea 2:** correct for the impact of bias introduced by 'initialising' the two smoothed measures – i.e. starting with $t=1$ 'fade-in' the smoothing effect exponentially by introducing $\overline{G}$ and $\overline{A}$:

$$G_{t+1} = \alpha G_t + (1-\alpha)\nabla J(X; W_t))$$
$$\overline{G} = G_{t+1} / (1-\alpha^t)$$
$$A_{t+1} = \beta A_t + (1-\beta)(\nabla J(X; W_t))^2$$
$$\overline{A} = A_{t+1} / (1-\beta^t)$$
$$W_{t+1} = W_t - \eta \frac{\overline{G}}{(\sqrt{\overline{A}} + \varepsilon)}$$

Summary Adam (adaptive moment estimation)

fading of gradient → force fast start-up

$$G_{t+1} = \alpha G_t + (1-\alpha)\nabla J(X; W_t))$$

smoothing of gradients

$$\overline{G} = G_{t+1} / (1-\alpha^t)$$

fading of per-weight magnitudes

$$A_{t+1} = \beta A_t + (1-\beta)(\nabla J(X; W_t))^2$$

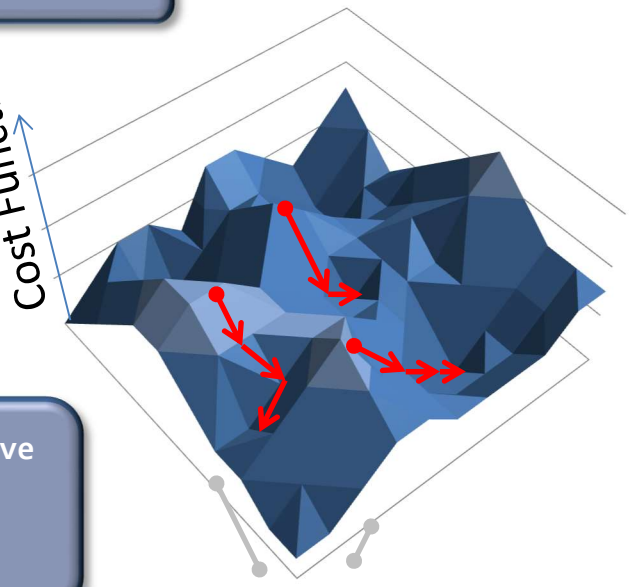
RMS propagation

element-wise squaring

$$\overline{A} = A_{t+1} / (1-\beta^t)$$

$$W_{t+1} = W_t - \eta \frac{\overline{G}}{(\sqrt{\overline{A}} + \varepsilon)}$$

element-wise dampening of recently highly active gradient components (A being large) and amplification of slowly changing gradient components (A being small)



Right, can we train deep networks now? – Maybe...

- Why is applying Adam to ReLU-based networks not a guarantee for successful deep learning then?
  - We have introduced new parameters α, β, ε... : how to set these so-called `hyper-parameters'?
  - Even our mini-batch size has not been discussed...
  - We have not talked about network initialisation – this matters a lot and can change results drastically if done wrong.
  - Overfitting is likely to occur in deep networks as in any learning system: regularisation techniques are critical to achieve good generalisation beyond the training data available!
  - Number of parameters explodes in deep networks; we may need to share them or reuse the entire net (e.g. CNNs/RNNs).
  - The simple loss functions discussed so far need extending to provide better results for common tasks such as classification.
  - The data we deal with is part of the training process – we have not talked about data at all so far...

  → Yet, applying deep learning and achieving top-end results often involves a lot of parameter tuning, testing and trial-and-error of various designs and techniques available, and performance is critically dependent on the quality of training data and also the GPU-sizes which limit network designs – it is still as much an `engineering process' as it is a science...

Next: COST FUNCTIONS, REGULARISATION AND DEPTH

- Key Loss Functions
- L1 and L2 Weight Decay
- Dropout and Noise
- Data Augmentation
- Why deep is advantageous...
- Scalability Considerations...