

# Machine Learning

Tricks of the Trade - Lecture XII

# Today's topics

## Recap: Optimization

- Effect of optimisers

## Tricks of the Trade

- Shuffling
- Data Augmentation
- Normalization

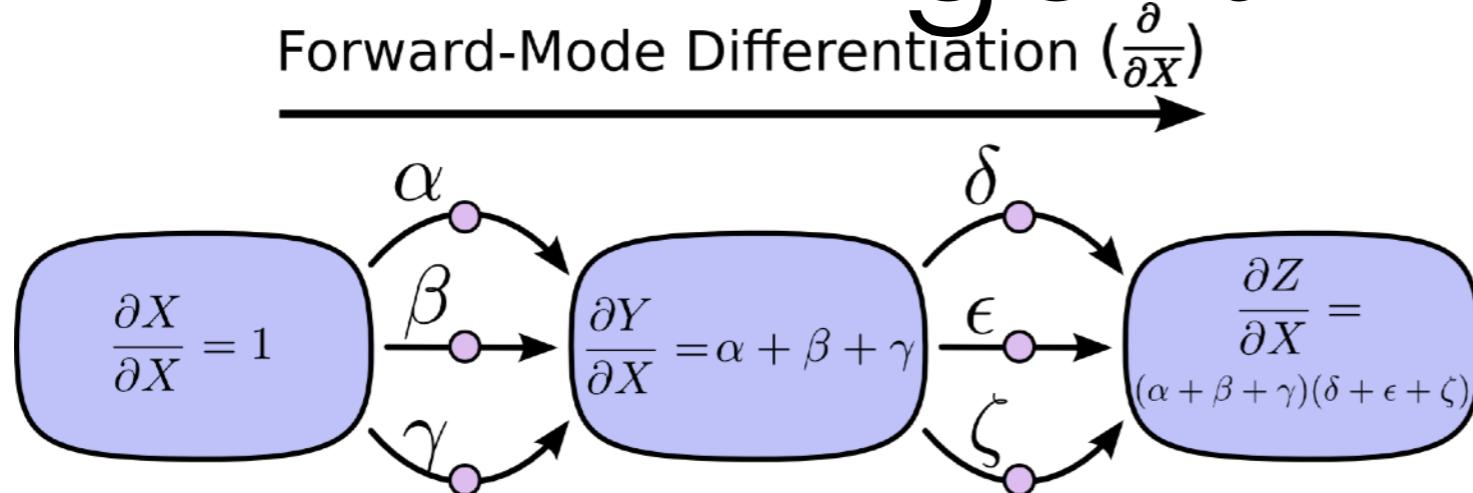
## Nonlinearities

## Initialization

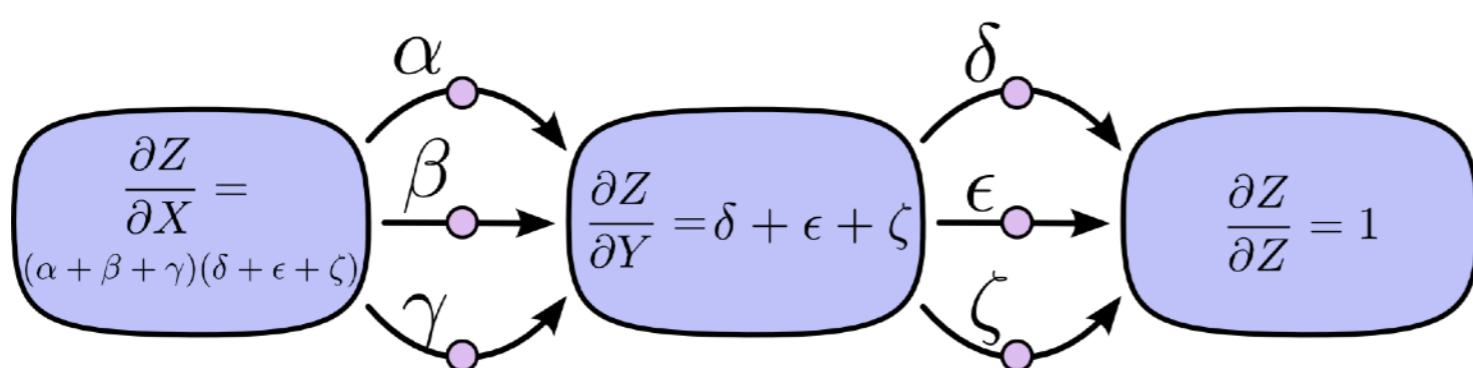
## Advanced Techniques

- Batch Normalization
- Dropout

# Recap: Efficient Factoring Algorithms



Apply operator  $\frac{\partial}{\partial X}$  to every node.



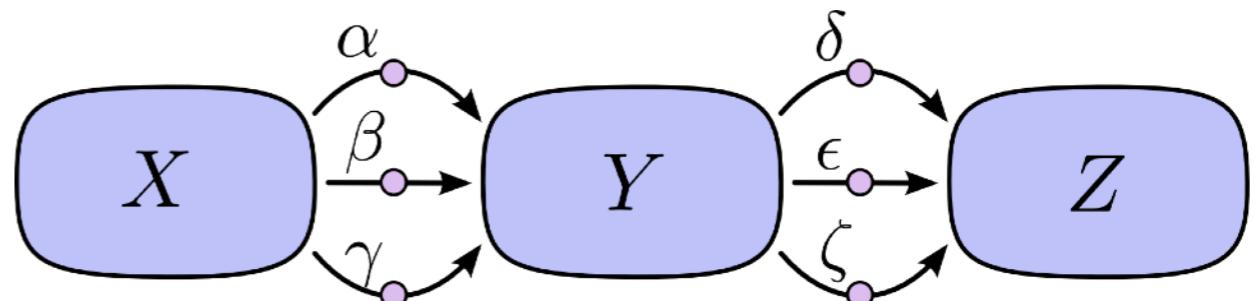
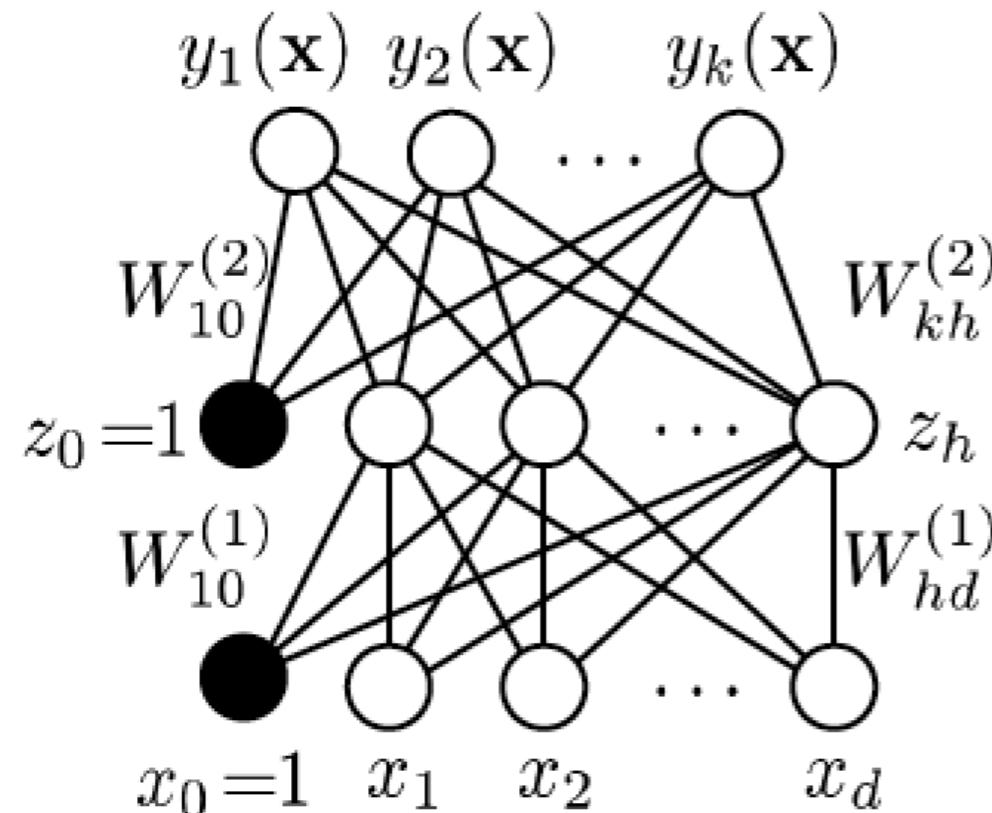
Apply operator  $\frac{\partial Z}{\partial}$  to every node.

- Forward differentiation needs one pass per node. Reverse-mode differentiation can compute all derivatives in one single pass.

=> Speed-up in  $O(\#inputs)$  compared to forward differentiation!

# Recap: Automatic Differentiation

## Approach for obtaining the gradients



- Convert the network into a computational graph.
  - Each new layer/module just needs to specify how it affects the forward and backward passes.
  - Apply reverse-mode differentiation.
- => Very general algorithm, used in today's Deep Learning packages

# Correction: Implementing Softmax Correctly

## Softmax output

- De-facto standard for multi-class outputs.

$$E(\mathbf{w}) = - \sum_{n=1}^N \sum_{k=1}^K \{\mathbb{I}(t_n = k) \ln \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{j=1}^K \exp(\mathbf{w}_j^T \mathbf{x})}\}$$

## Practical issue

- Exponentials get very big and can have vastly different magnitudes.
- **Trick 1:** Do not compute first softmax, then log, but instead directly evaluate log-exp in the denominator.
- **Trick 2:** Softmax has the property that for a fixed vector  $\mathbf{b}$

$$\text{softmax}(\mathbf{a} + \mathbf{b}) = \text{softmax}(\mathbf{a})$$

=> Subtract the largest weight vector  $\mathbf{w}_j$  from the others.

# Choosing the Right Learning Rate

## Analyzing the convergence of Gradient Descent

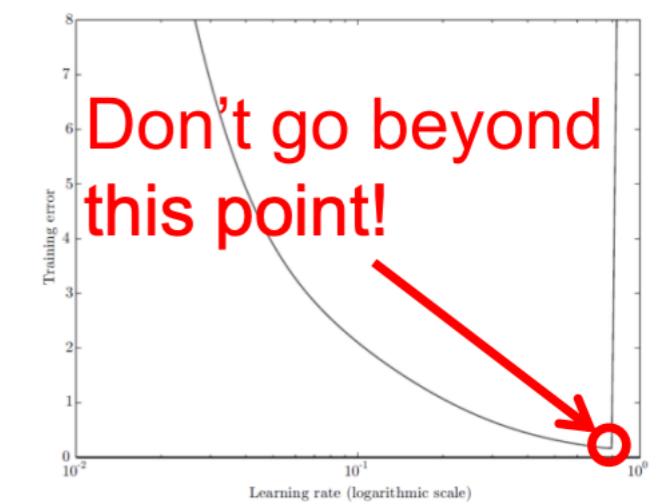
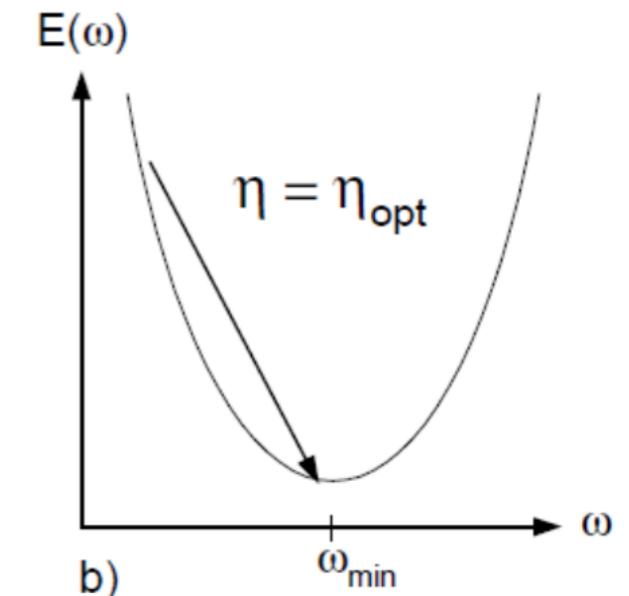
- Consider a simple 1D example first

$$W^{(\tau-1)} = W^{(\tau)} - \eta \frac{dE(W)}{dW}$$

- What is the optimal learning rate  $\eta_{opt}$ ?
- If  $E$  is quadratic, the optimal learning rate is given by the inverse of the Hessian

$$\eta_{opt} = \left( \frac{d^2 E(W^{(\tau)})}{dW^2} \right)^{-1}$$

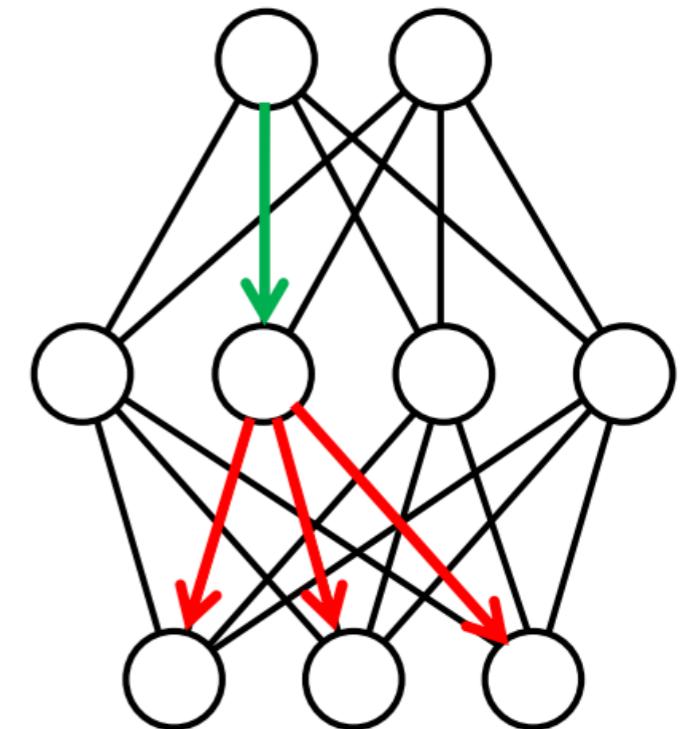
- Advanced optimization techniques try to approximate the Hessian by a simplified form.
- If we exceed the optimal learning rate, bad things happen!*



# Separate, Adaptive Learning Rates

## Problem

- In multilayer nets, the appropriate learning rates can vary widely between weights.
- The **magnitudes of the gradients** are often very different for the different layers, especially if the initial weights are small.
  - Gradients can get very small in the early layers of deep nets.



# Better Adaptation: RMSProp

## Motivation

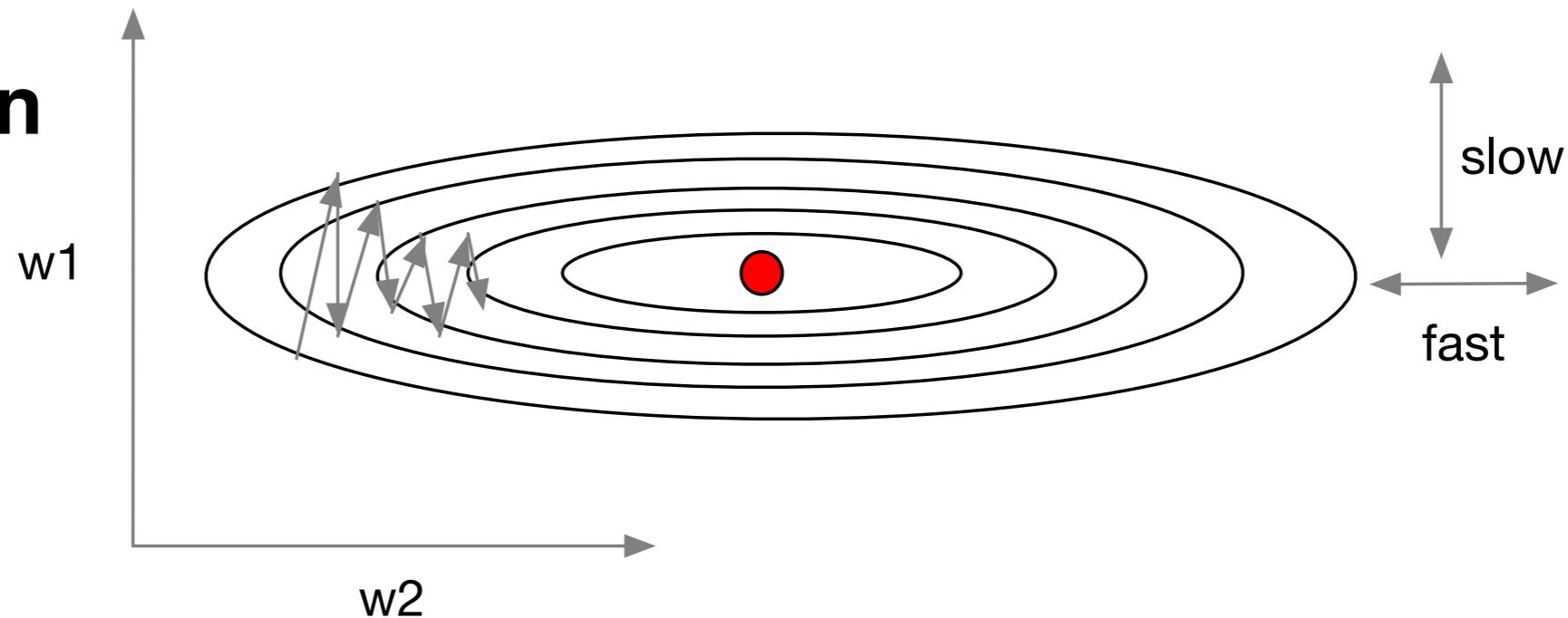
- The magnitude of the gradient can be very different for different weights and can change during learning.
- This makes it hard to choose a single global learning rate.
- For batch learning, we can deal with this by only using the sign of the gradient, but we need to generalize this for minibatches.

## Idea of RMSProp

- Divide the gradient by a running average of its recent magnitude
$$MeanSq(w_{ij}, t) = 0.9MeanSq(w_{ij}, t - 1) + 0.1(\frac{\partial E}{\partial w_{ij}}(t))^2$$
- Divide the gradient by  $sqrt(MeanSq(w_{ij}, t))$

# RMSProp

## Motivation



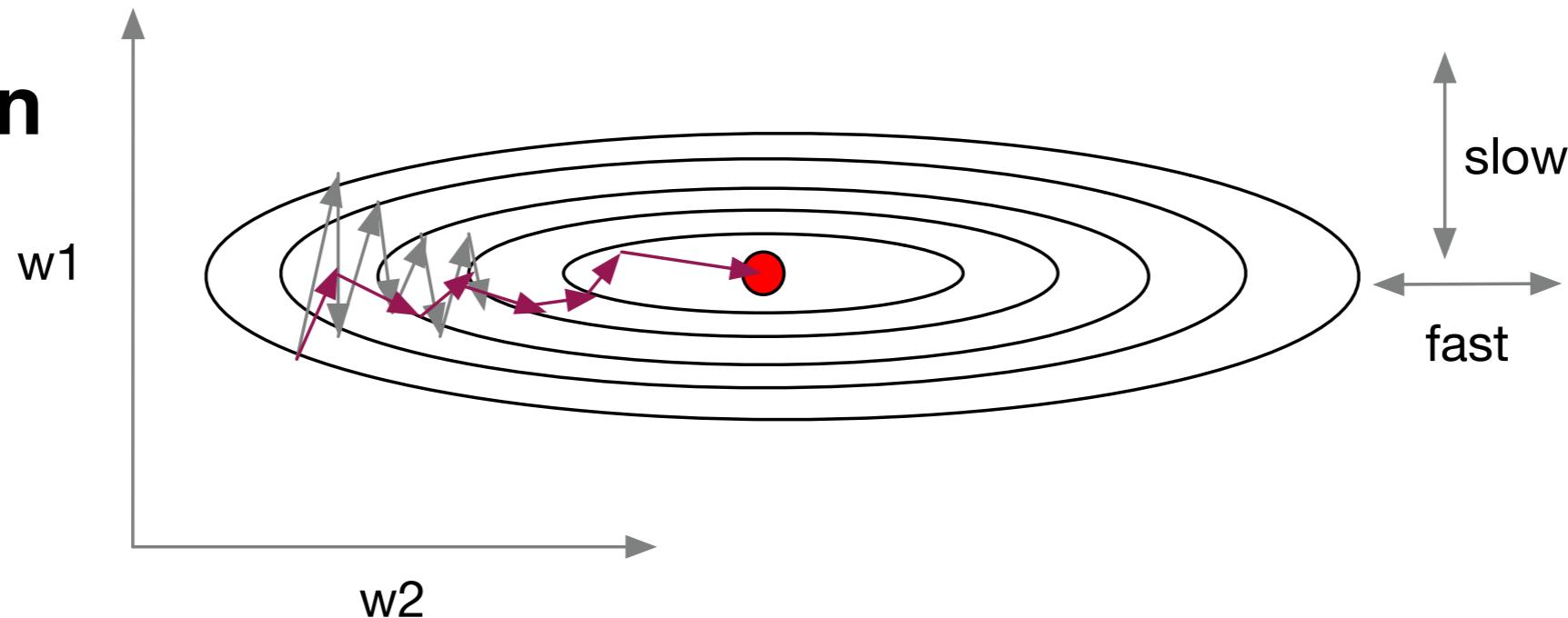
- Stands for Root mean square prop.
- This algorithm speeds up the gradient descent.

## Idea of RMSProp

- Divide the gradient by a running average of its recent magnitude  
$$MeanSq(w_{ij}, t) = 0.9MeanSq(w_{ij}, t - 1) + 0.1(\frac{\partial E}{\partial w_{ij}}(t))^2$$
- Divide the gradient by  $\sqrt{MeanSq(w_{ij}, t)}$

# RMSProp

## Motivation



- Stands for Root mean square prop.
- This algorithm speeds up the gradient descent.

## Idea of RMSProp

- Divide the gradient by a running average of its recent magnitude  
$$MeanSq(w_{ij}, t) = 0.9MeanSq(w_{ij}, t - 1) + 0.1(\frac{\partial E}{\partial w_{ij}}(t))^2$$
- Divide the gradient by  $\sqrt{MeanSq(w_{ij}, t)}$

# Other Optimizers

**AdaGrad**

[Duchi '10]

**AdaDelta**

[Zeiler '12]

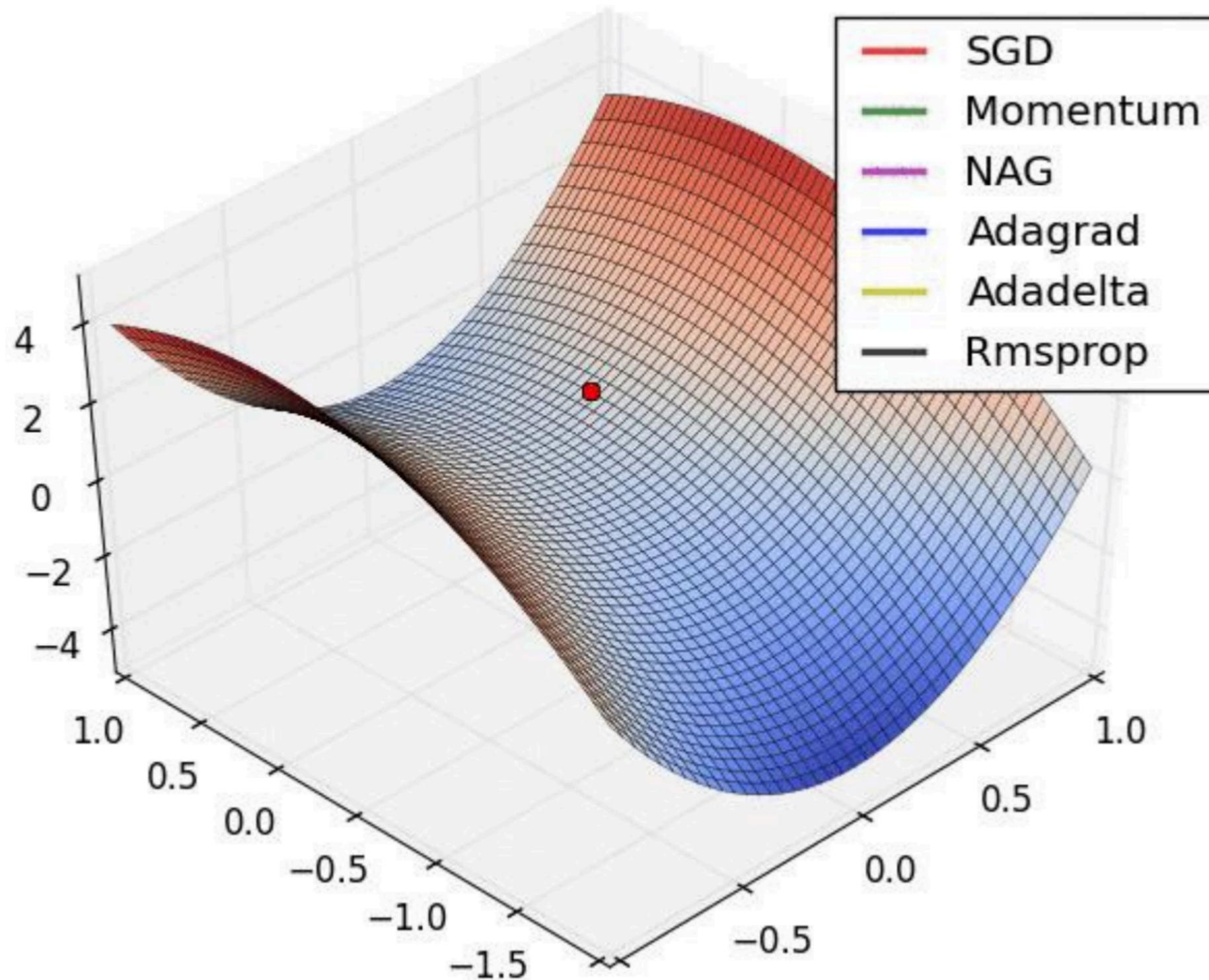
**Adam**

[Ba & Kingma '14]

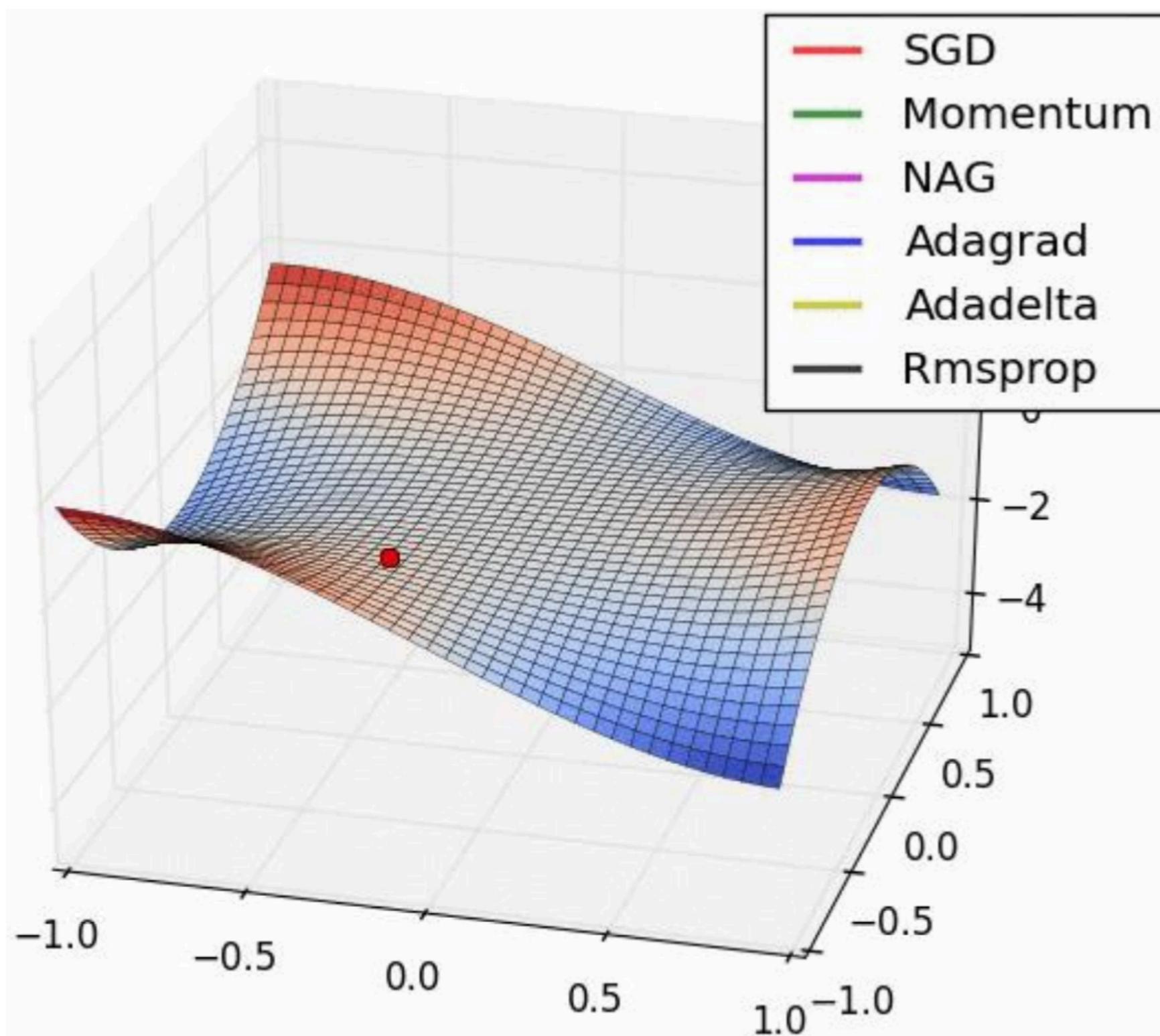
**Notes**

- All of those methods have the goal to make the optimization less sensitive to parameter settings.
- Adam is currently becoming the quasi-standard

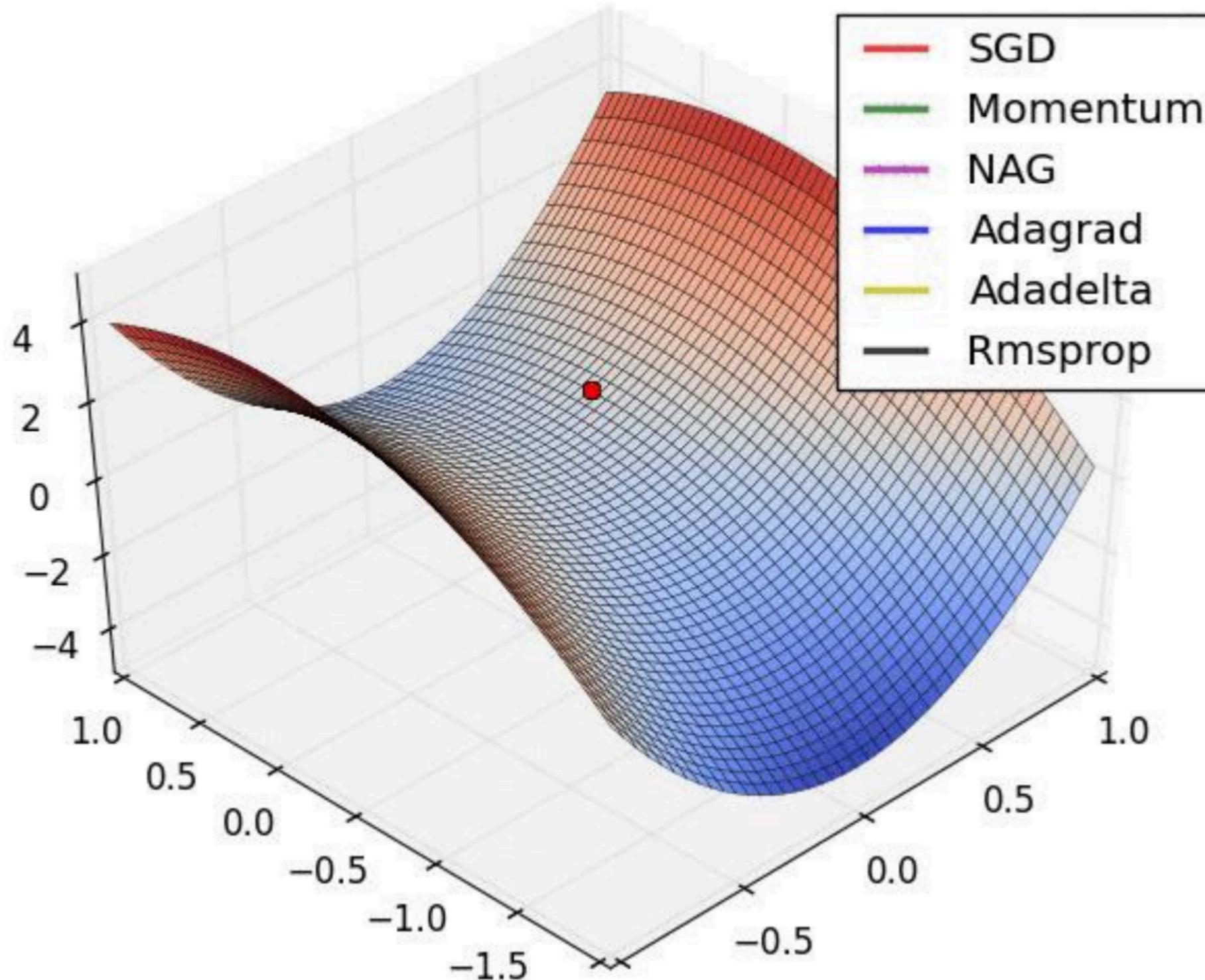
# Behavior in a Long Valley



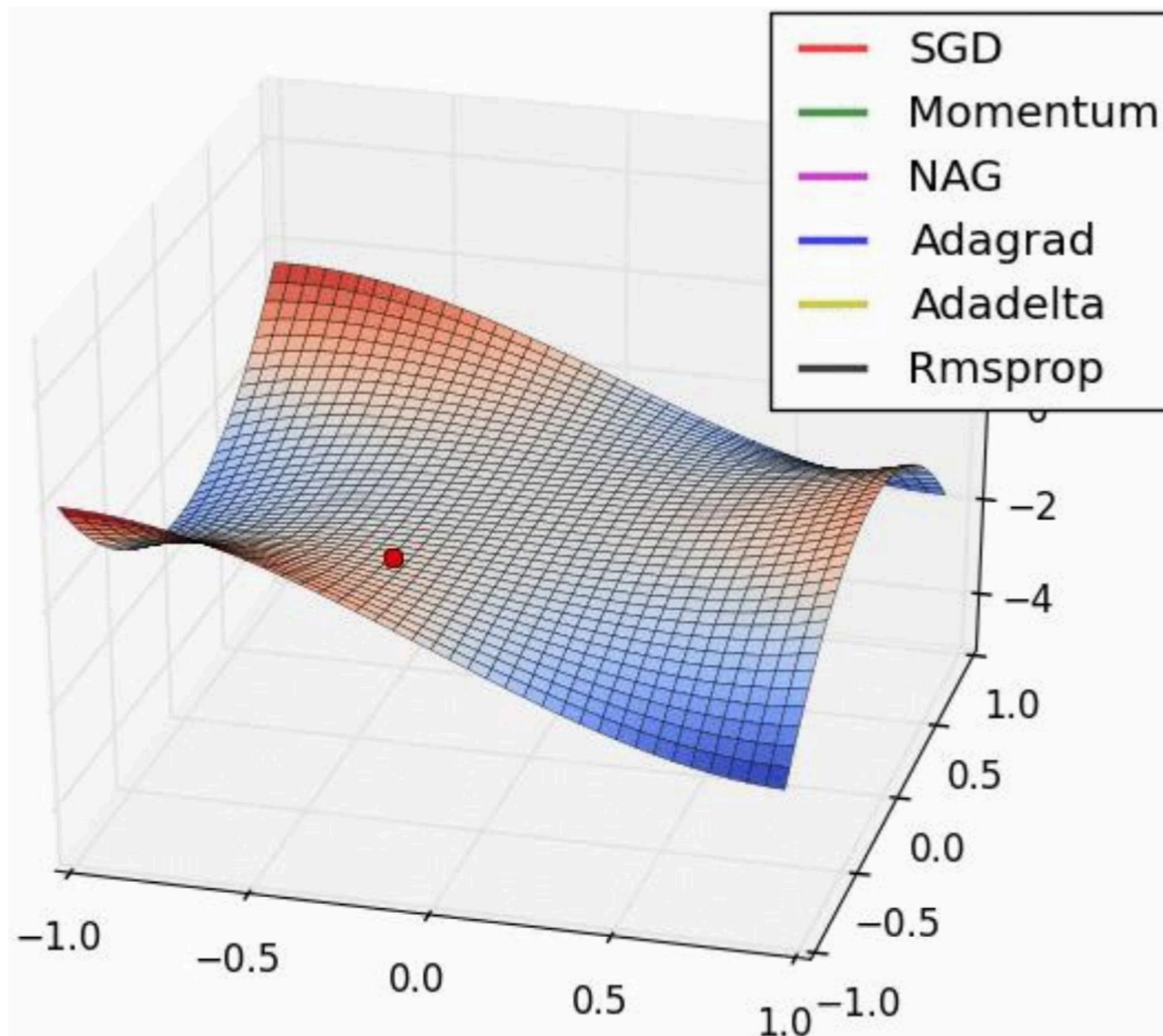
# Behavior around a Saddle Point



# Example: Behavior in a Long Valley



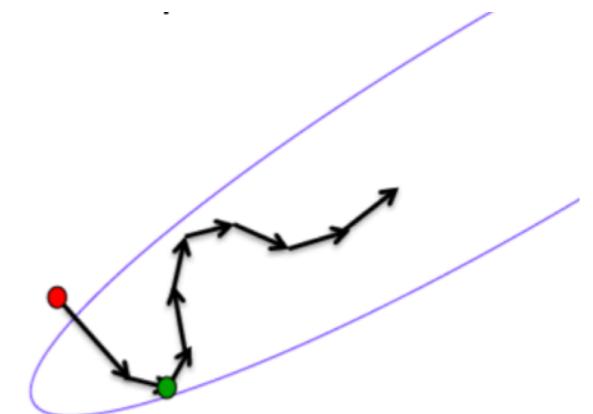
# Example: Behavior around a Saddle Point



# Recap: Advanced Optimization Techniques

## Momentum

- Instead of using the gradient to change the position of the weight “particle”, use it to change the velocity.
- Effect: dampen oscillations in directions of high curvature
- Nesterov-Momentum: Small variation in the implementation



## RMS-Prop

- Separate learning rate for each weight: Divide the gradient by a running average of its recent magnitude

## AdaGrad

## AdaDelta

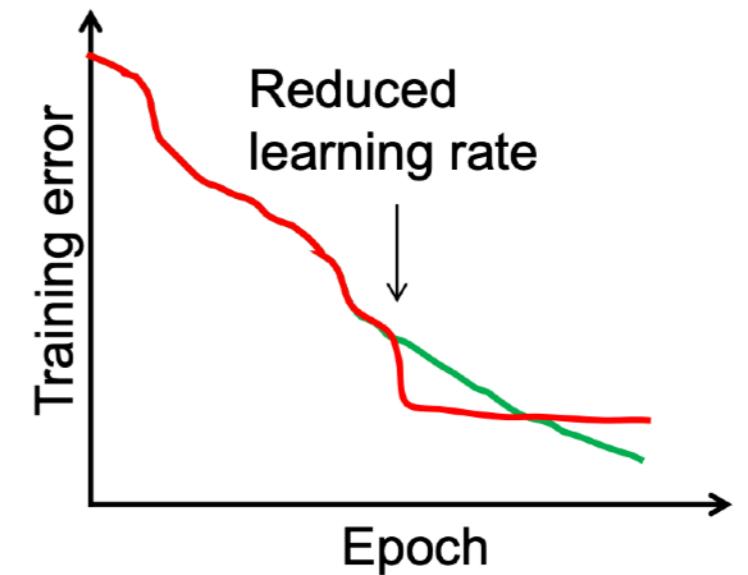
## Adam

Some more recent techniques, work better for some problems. Try them.

# Reducing the Learning Rate

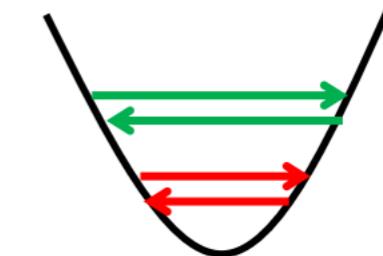
## Final improvement step after convergence is reached

- Reduce learning rate by a factor of 10.
- Continue training for a few epochs.
- Do this 1-3 times, then stop training.



## Effect

- Turning down the learning rate will reduce the random fluctuations in the error due to different gradients on different mini-batches.



*Be careful: Do not turn down the learning rate too soon!*

- Further progress will be much slower/impossible after that.

# Summary

**Deep multi-layer networks are very powerful.**

**But training them is hard!**

- Complex, non-convex learning problem
- Local optimization with stochastic gradient descent

**Main issue: getting good gradient updates for the lower layers of the network**

- Many seemingly small details matter!
- Weight initialization, normalization, data augmentation, choice of nonlinearities, choice of learning rate, choice of optimizer,...
- In the following, we will take a look at the most important factors (to be continued in the next lecture...)

# Today's topics

Recap: Optimization

- Effect of optimisers

Tricks of the Trade

- Shuffling
- Data Augmentation
- Normalization

Nonlinearities

Initialization

Advanced Techniques

- Batch Normalization
- Dropout

# Shuffling the Examples

## Ideas

- Networks learn fastest from the most unexpected sample.
- It is advisable to choose a sample at each iteration that is most unfamiliar to the system.
  - E.g. a sample from a different class than the previous one.
  - This means, do not present all samples of class A, then all of class B.
- A large relative error indicates that an input has not been learned by the network yet, so it contains a lot of information.
- It can make sense to present such inputs more frequently.
  - But: be careful, this can be disastrous when the data are outliers.

## Practical advice

- When working with stochastic gradient descent or minibatches, make use of shuffling.

# Data Augmentation

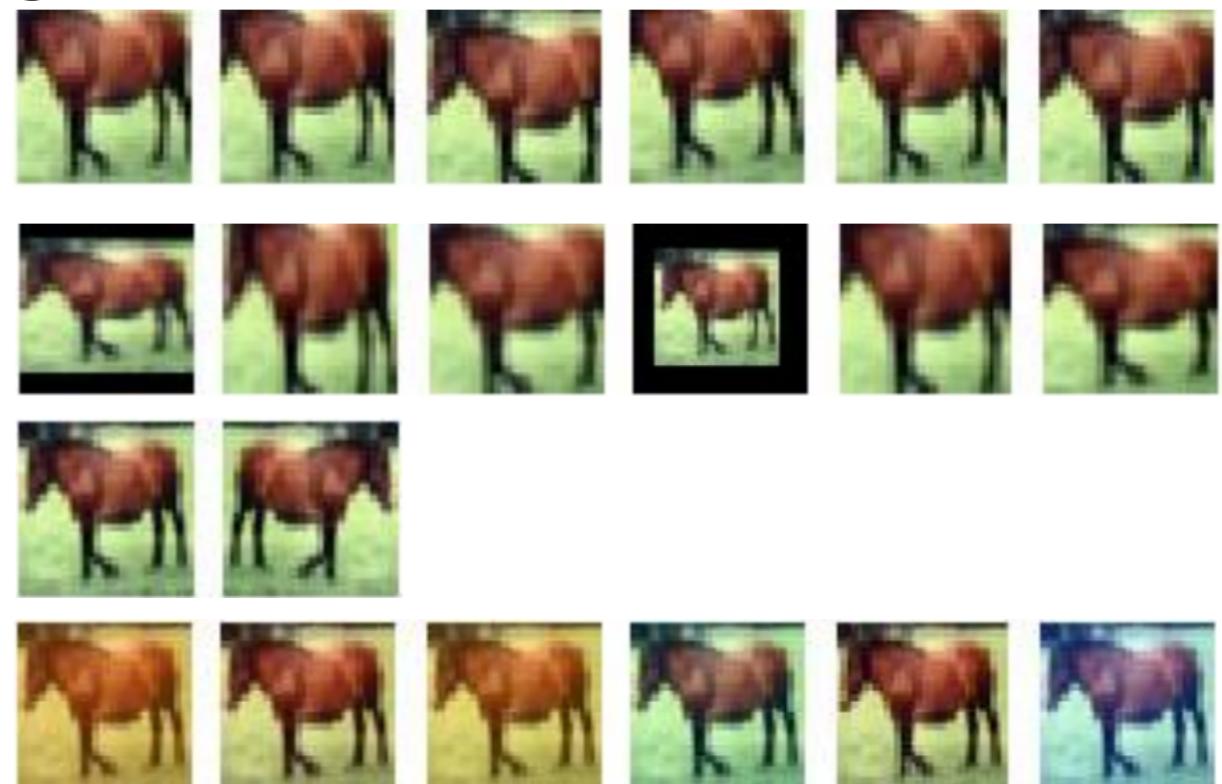
## Idea

- Augment original data with synthetic variations to reduce overfitting



## Example augmentations for images

- Cropping
- Zooming
- Flipping
- Color PCA



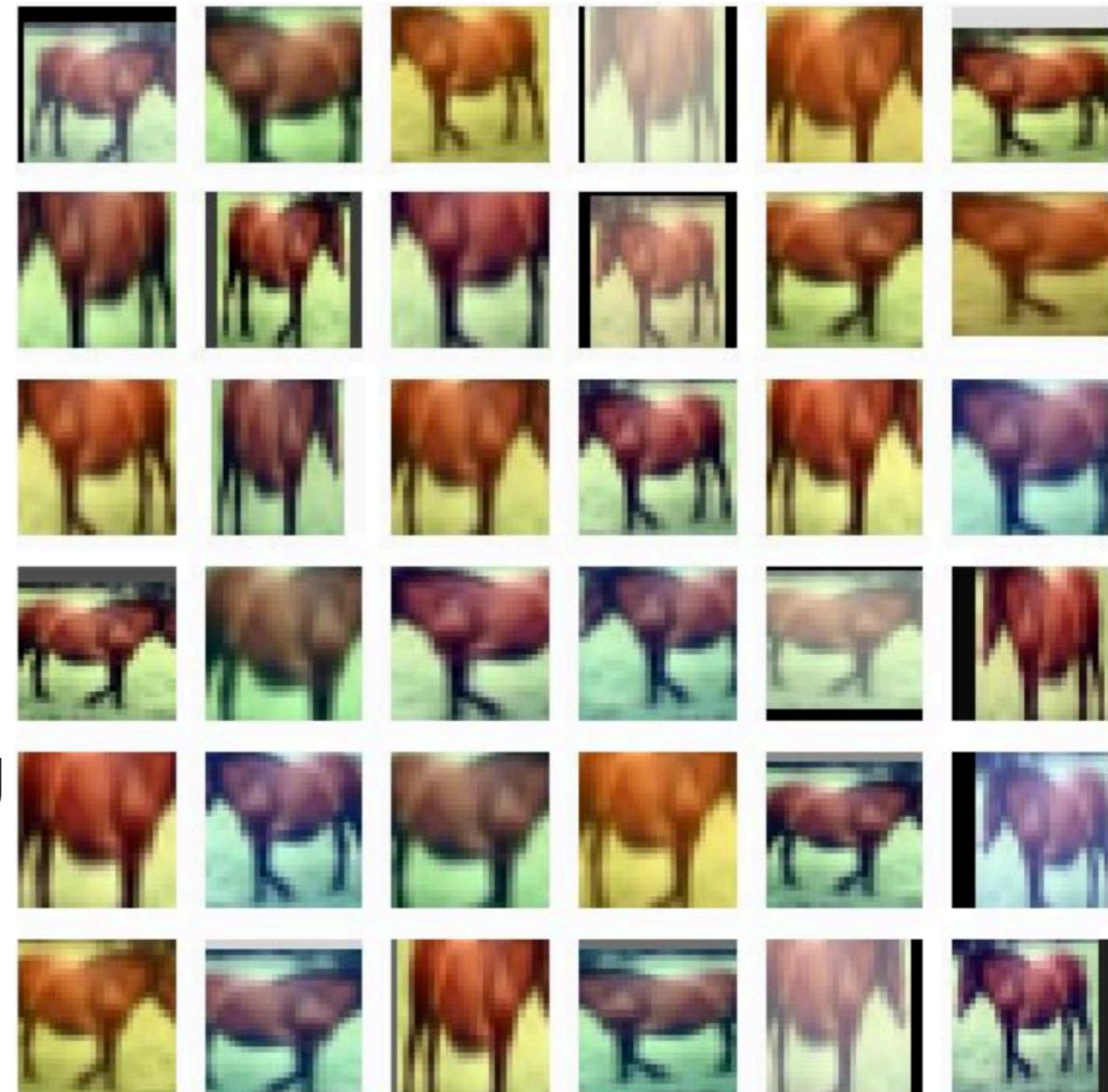
# Data Augmentation

## Effect

- Much larger training set
- Robustness against expected variations

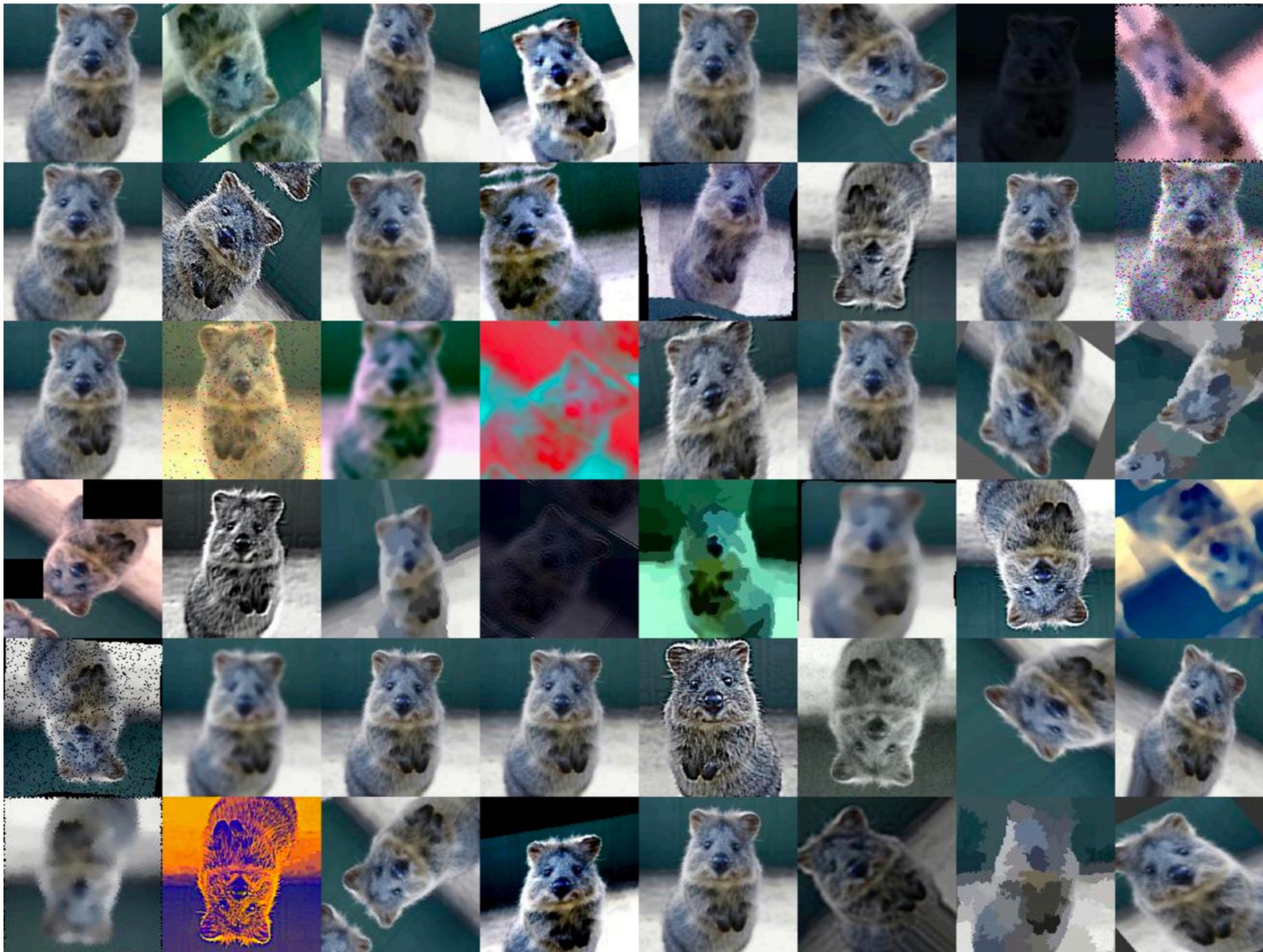
## During testing

- When cropping was used during training, need to again apply crops to get same image size.
- Beneficial to also apply flipping during test.
- Applying several ColorPCA variations can bring another ~1% improvement, but at a significantly increased runtime.



Augmented training data  
(from one original image)

# Practical Advice: Apply All the Augmentations

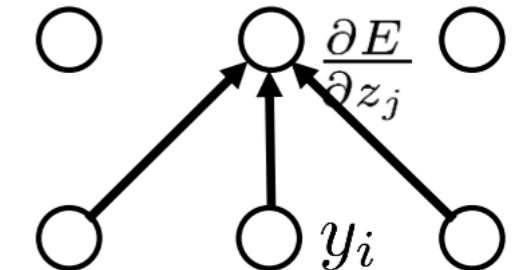


# Normalization

## Motivation

- Consider the Gradient Descent update steps

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \left. \frac{\partial E(\mathbf{w})}{\partial w_{kj}} \right|_{\mathbf{w}^{(\tau)}}$$



- From backpropagation, we know that

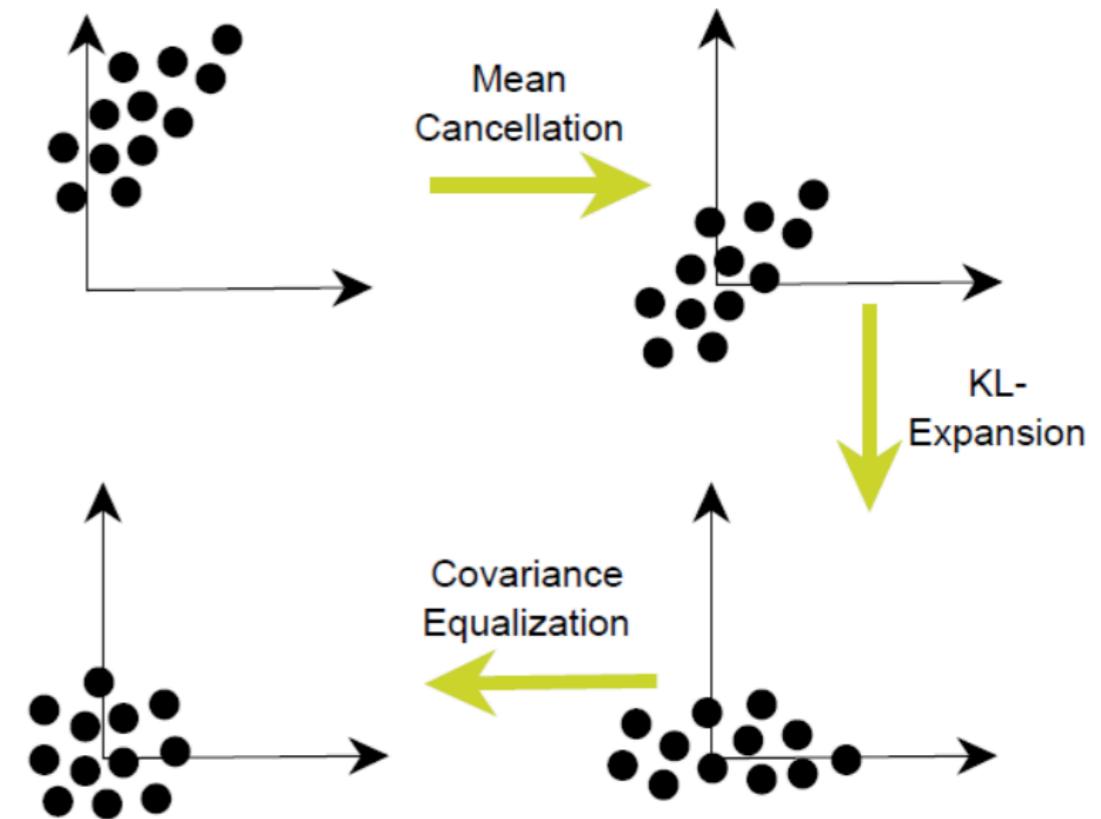
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = \textcolor{red}{y_i} \frac{\partial E}{\partial z_j}$$

- When all of the components of the input vector  $y_i$  are positive, all of the updates of weights that feed into a node will be of the same sign.
  - Weights can only all increase or decrease together.
  - Slow convergence

# Normalizing the Inputs

## Convergence is fastest if

- The mean of each input variable over the training set is zero.
- The inputs are scaled such that all have the same covariance.
- Input variables are uncorrelated if possible.



## Advisable normalization steps (for MLPs only, not for CNNs)

- Normalize all inputs that an input unit sees to zero-mean, unit covariance.
- If possible, try to decorrelate them using PCA (also known as Karhunen-Loeve expansion).

# Today's topics

Recap: Optimization

- Effect of optimisers

Tricks of the Trade

- Shuffling
- Data Augmentation
- Normalization

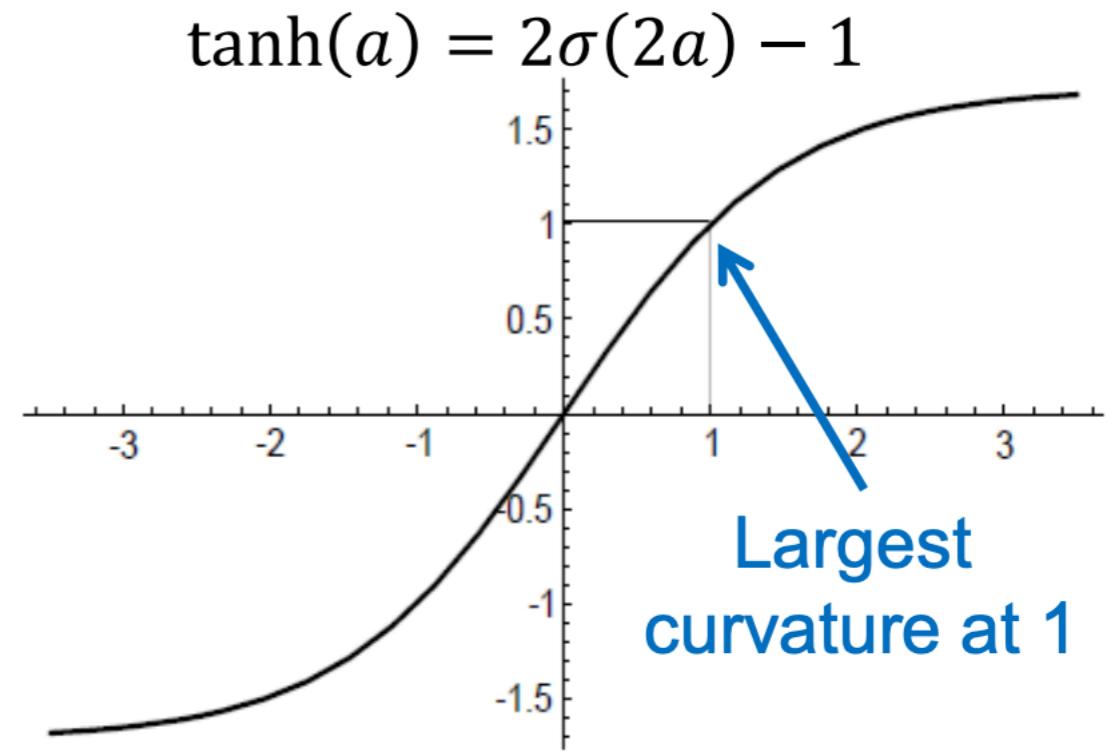
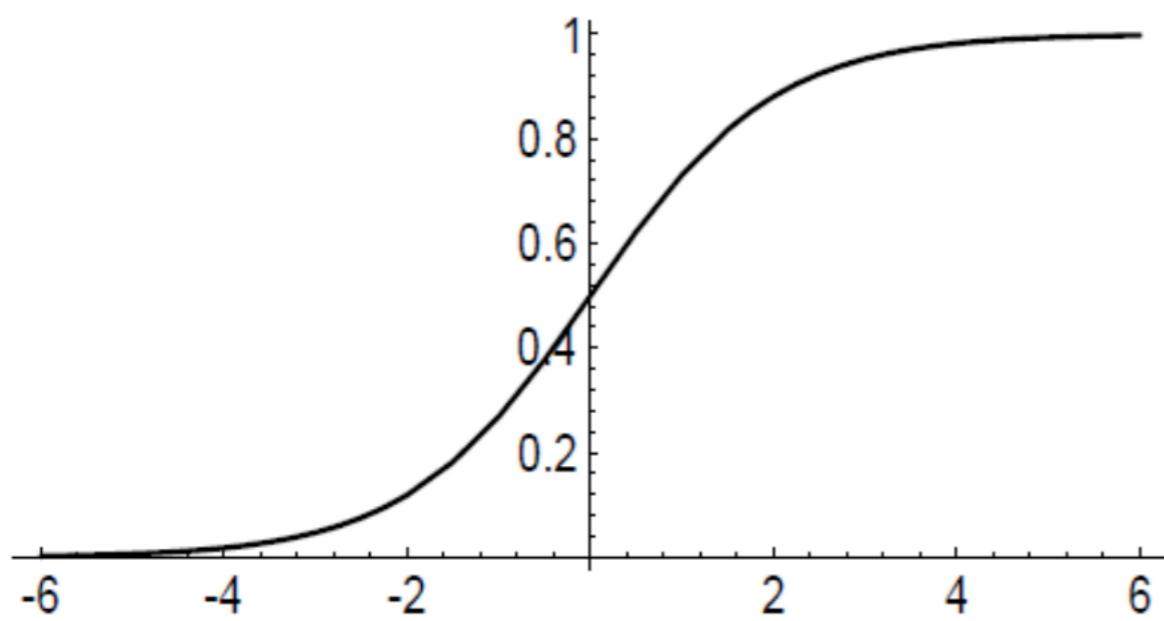
Nonlinearities

Initialization

Advanced Techniques

- Batch Normalization
- Dropout

# Choosing the Right Sigmoid



**Normalization is also important for intermediate layers**

- Symmetric sigmoids, such as  $\tanh$ , often converge faster than the standard logistic sigmoid.
- Recommended sigmoid:

$$f(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$$

- When used with transformed inputs, the variance of the outputs will be close to 1.

# Usage

## Output nodes

- Typically, a sigmoid or tanh function is used here.
  - Sigmoid for nice probabilistic interpretation (range [0,1]).
  - tanh for regression tasks

## Internal nodes

- Historically, tanh was most often used.
- tanh is better than sigmoid for internal nodes, since it is already centered.
- Internally, tanh is often implemented as piecewise linear function (similar to hard tanh and maxout).
- More recently: ReLU often used for classification tasks.

# Effect of Sigmoid Nonlinearities

## Effects of sigmoid/tanh function

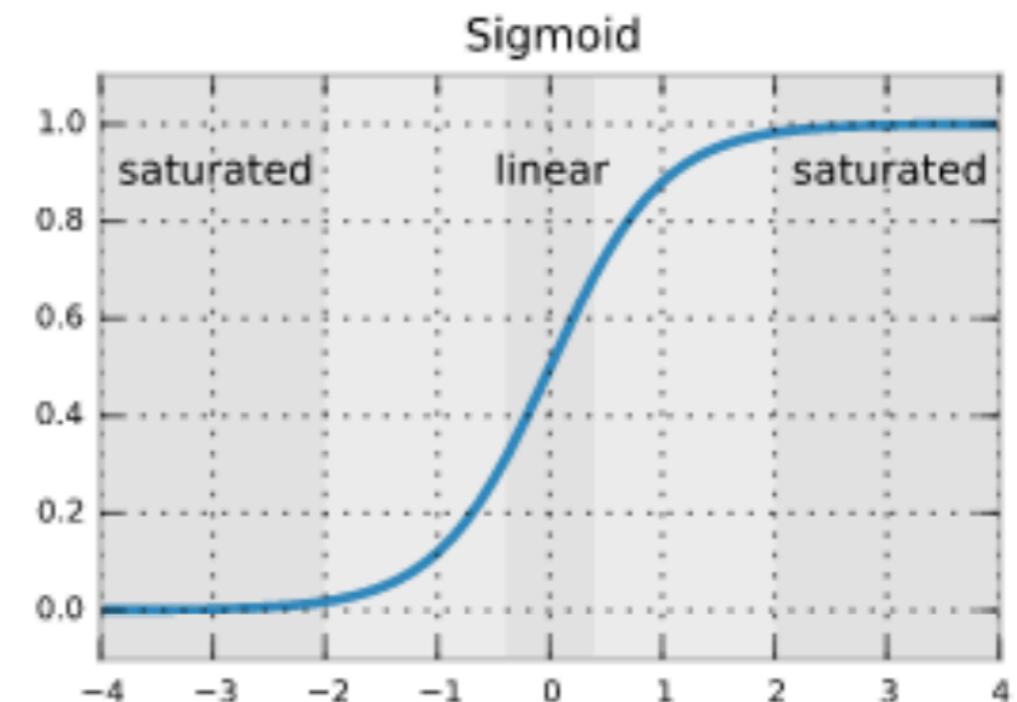
- Linear behavior around 0
- Saturation for large inputs

## If all parameters are too small

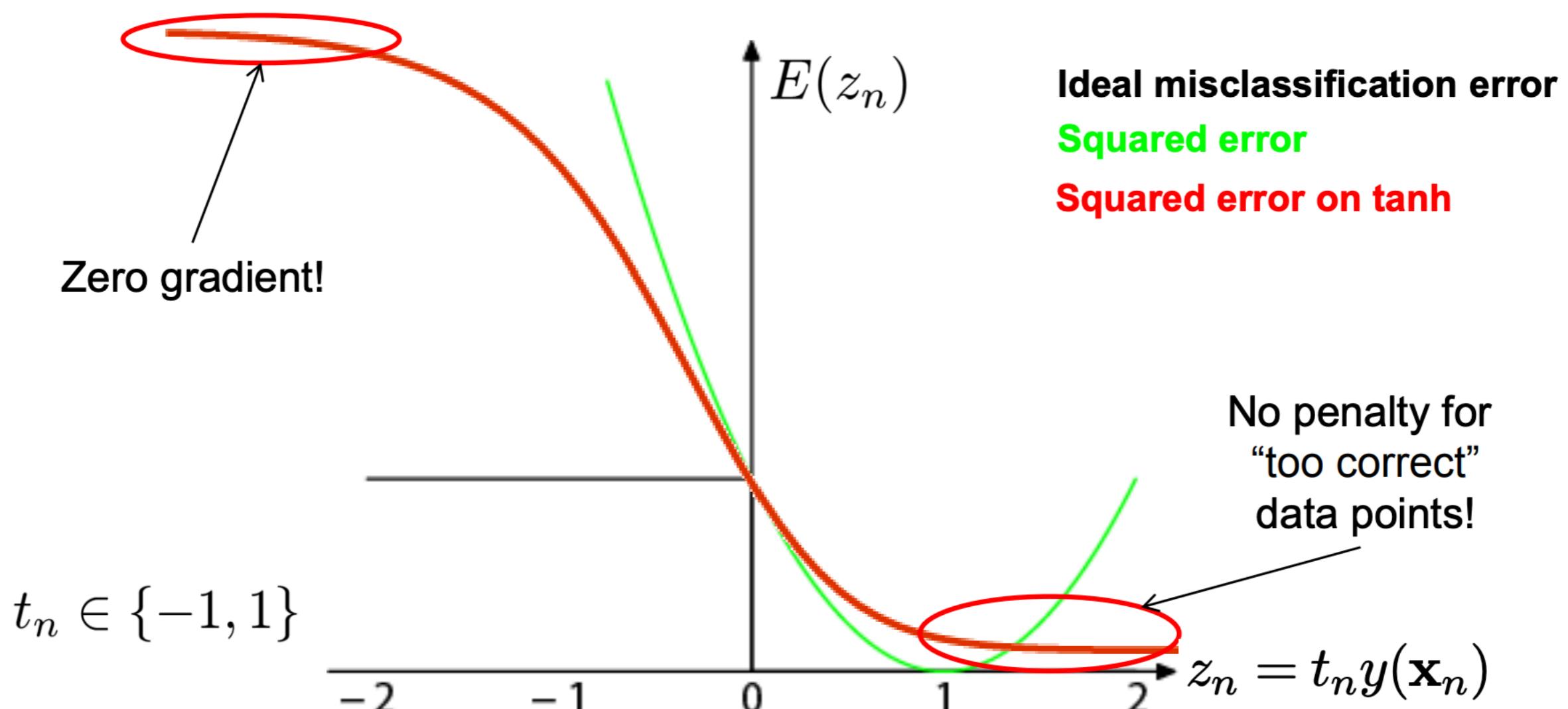
- Variance of activations will drop in each layer
- Sigmoids are approximately linear close to 0
- Good for passing gradients through, but...
- Gradual loss of the nonlinearity
- No benefit of having multiple layers

## If activations become larger and larger

- They will saturate and gradient will become zero



# Another Note on Error Functions



## Squared error on sigmoid/tanh output function

- Avoids penalizing “too correct” data points.
- But: zero gradient for confidently incorrect classifications!
- Do not use L2 loss with sigmoid outputs (instead: cross-entropy)!

# Extension: ReLU

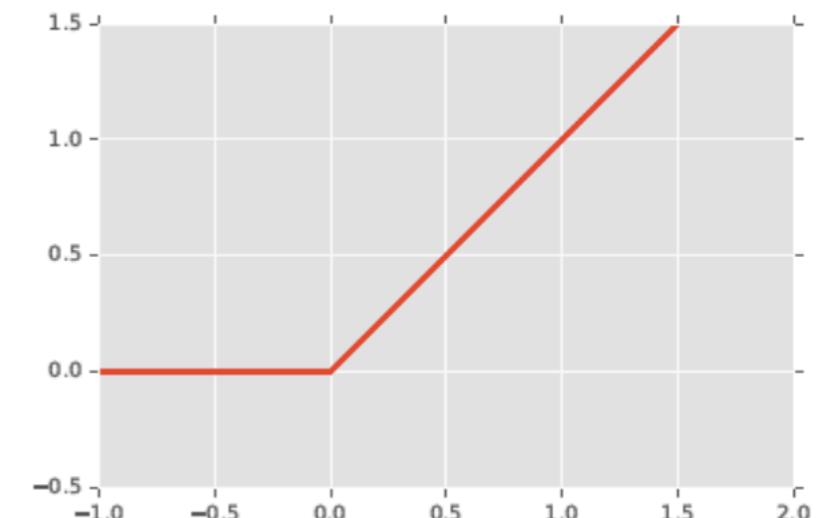
## Another improvement for learning deep models

- Use Rectified Linear Units (ReLU)

$$g(a) = \max\{0, a\}$$

**Effect: gradient is propagated with a constant factor**

$$\frac{\partial g(a)}{\partial a} = \begin{cases} 1, & a > 0 \\ 0, & \text{else} \end{cases}$$



## Advantages

- Much easier to propagate gradients through deep networks.
- We do not need to store the ReLU output separately
  - Reduction of the required memory by half compared to tanh!
- ReLU has become the de-facto standard for deep networks.

# Extension: ReLU

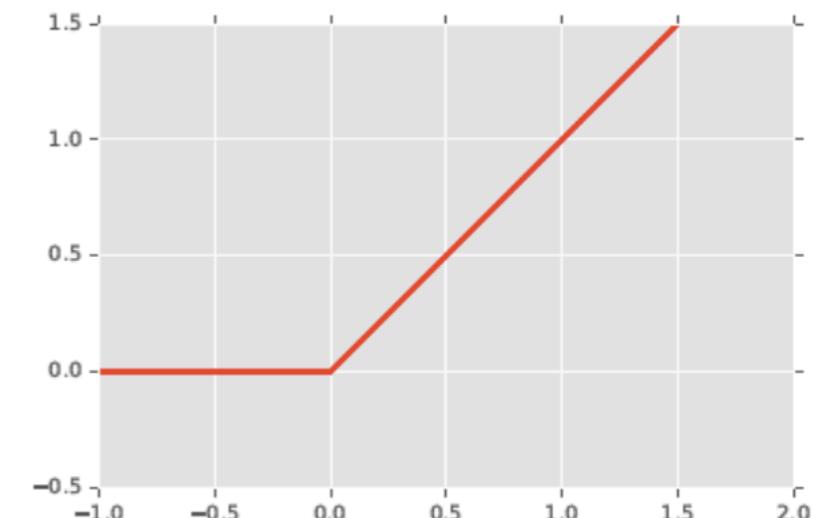
## Another improvement for learning deep models

- Use Rectified Linear Units (ReLU)

$$g(a) = \max\{0, a\}$$

**Effect: gradient is propagated with a constant factor**

$$\frac{\partial g(a)}{\partial a} = \begin{cases} 1, & a > 0 \\ 0, & \text{else} \end{cases}$$

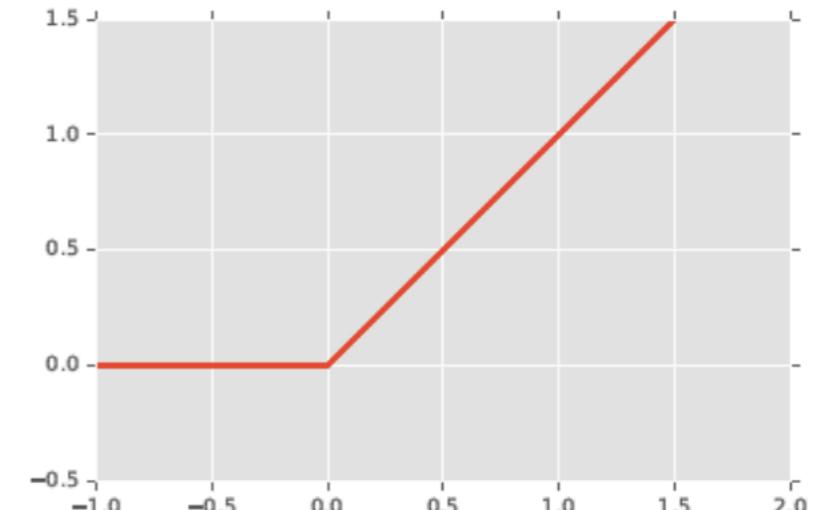


- **Disadvantages / Limitations**
  - A certain fraction of units will remain “stuck at zero”.
    - If the initial weights are chosen such that the ReLU output is 0 for the entire training set, the unit will never pass through a gradient to change those weights.
  - ReLU has an offset bias, since its outputs will always be positive

# Further Extensions

## Rectified linear unit (ReLU)

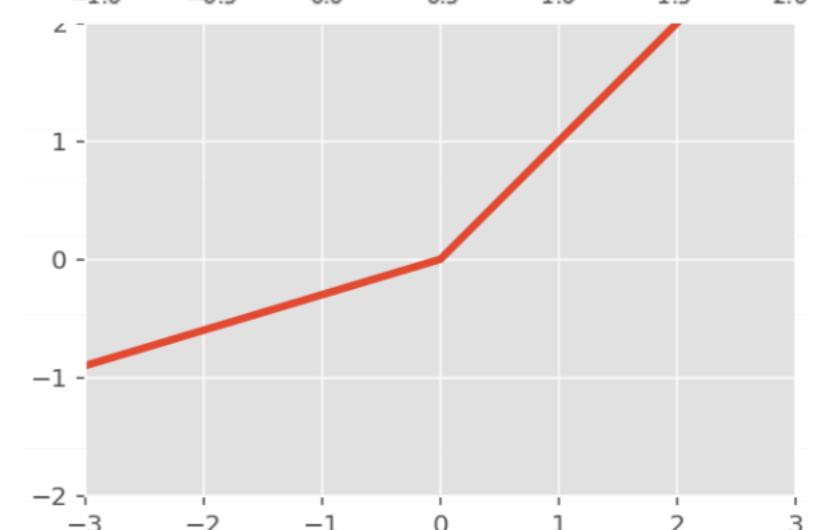
$$g(a) = \max\{0, a\}$$



## Leaky ReLU

$$g(a) = \max\{\beta a, a\}$$

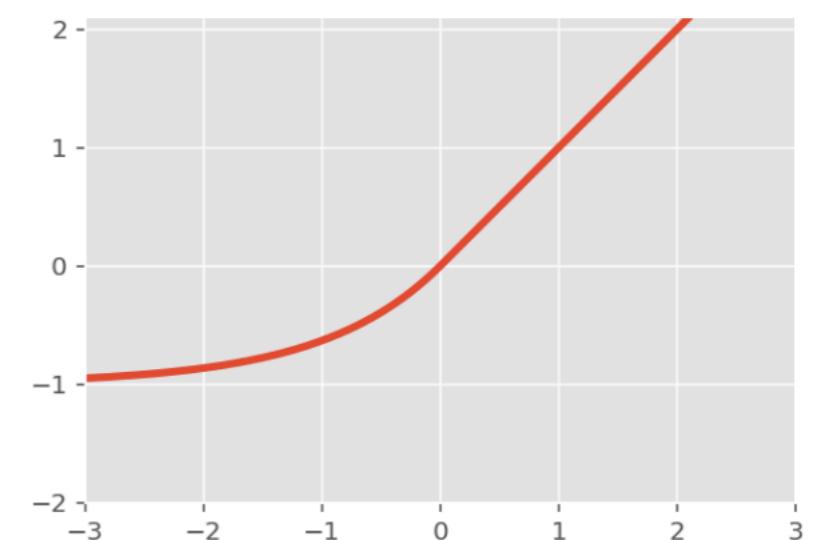
- Avoids stuck-at-zero units
- Weaker offset bias



## ELU

$$g(a) = \begin{cases} a, & x < 0 \\ e^a - 1, & x \geq 0 \end{cases}$$

- No offset bias anymore
- BUT: need to store activations



# Today's topics

Recap: Optimization

- Effect of optimisers

Tricks of the Trade

- Shuffling
- Data Augmentation
- Normalization

Nonlinearities

Initialization

Advanced Techniques

- Batch Normalization
- Dropout

# Initializing the Weights

## Motivation

- The starting values of the weights can have a significant effect on the training process.
- Weights should be chosen randomly, but in a way that the sigmoid is primarily activated in its linear region.

## Guideline (from [LeCun et al., 1998] book chapter)

- Assuming that
  - The training set has been normalized
  - The recommended sigmoid  $f(x) = 1.7159(\frac{2}{3}x)$  is used the initial weights should be randomly drawn from a distribution (e.g., uniform or Normal) with mean zero and variance

$$\sigma_w^2 = \frac{1}{n_{in}}$$

where  $n_{in}$  is the fan-in (#connections into the node).

# Historical Sidenote

**Apparently, this guideline was either little known or misunderstood for a long time**

- A popular heuristic (also the standard in Torch) was to use

$$W \sim U \left[ -\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}} \right]$$

- This looks almost like LeCun's rule. However...

**When sampling weights from a uniform distribution [a,b]**

- Keep in mind that the standard deviation is computed as

$$\sigma^2 = \frac{1}{12} (b - a)^2$$

- If we do that for the above formula, we obtain

$$\sigma^2 = \frac{1}{12} \left( \frac{2}{\sqrt{n_{in}}} \right)^2 = \frac{1}{3} \frac{1}{n_{in}}$$

→ Activations & gradients will be attenuated with each layer! (bad)

# Glorot Initialization

## Breakthrough results

- In 2010, Xavier Glorot published an analysis of what went wrong in the initialization and derived a more general method for automatic initialization.
- This new initialization massively improved results and made direct learning of deep networks possible overnight.
- Let's look at his analysis in more detail...

X. Glorot, Y. Bengio, Understanding the Difficulty of Training Deep Feedforward Neural Networks, AISTATS 2010.

# Analysis

## Variance of neuron activations

- Suppose we have an input  $X$  with  $n$  components and a linear neuron with random weights  $W$  that spits out a number  $Y$ .
- What is the variance of  $Y$  ?

$$Y = W_1 X_1 + W_2 X_2 + \dots + W_n X_n$$

- If inputs and outputs have both mean 0, the variance is

$$\begin{aligned} Var(W_i, X_i) &= E[X_i]^2 Var(W_i) + E[W_i]^2 Var(X_i) + Var(W_i)Var(i_i) \\ &= Var(W_i)Var(X_i) \end{aligned}$$

- If the  $X_i$  and  $W_i$  are all i.i.d, then

$$Var(Y) = Var(W_1 X_1 + W_2 X_2 + \dots + W_n X_n) = nVar(W_i)Var(X_i)$$

- The variance of the output is the variance of the input, but scaled by  $n$   $Var(W_i)$ .

# Analysis (cont'd)

## Variance of neuron activations

- if we want the variance of the input and output of a unit to be the same, then  $n \text{Var}(W_i)$  should be 1. This means

$$\text{Var}(W_i) = \frac{1}{n} = \frac{1}{n_{in}}$$

- If we do the same for the backpropagated gradient, we get

$$\text{Var}(W_i) = \frac{1}{n_{out}}$$

- As a compromise, Glorot & Bengio proposed to use

$$\text{Var}(W) = \frac{2}{n_{in} + n_{out}}$$

→ Randomly sample the weights with this variance. That's it.

# Sidenote

## When sampling weights from a uniform distribution [a,b]

- Again keep in mind that the standard deviation is computed as

$$\sigma^2 = \frac{1}{12}(b - a)^2$$

- Glorot initialization with uniform distribution

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right]$$

# Extension to ReLU

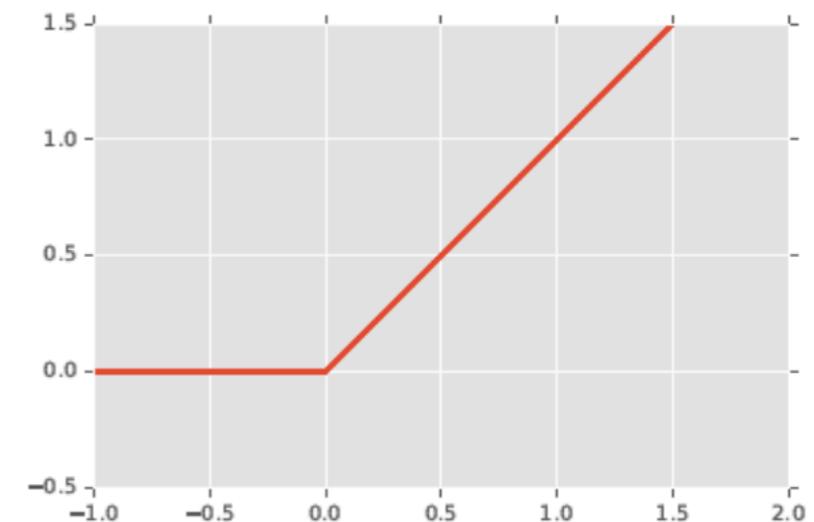
## Important for learning deep models

- Rectified Linear Units (ReLU)

$$g(a) = \max\{0, a\}$$

- Effect: gradient is propagated with a constant factor

$$\frac{\partial g(a)}{\partial a} = \begin{cases} 1, & a > 0 \\ 0, & \text{else} \end{cases}$$



## We can also improve them with proper initialization

- However, the Glorot derivation was based on tanh units, linearity assumption around zero does not hold for ReLU.
- He et al. made the derivations, derived to use instead

$$Var(W) = \frac{2}{n_{in}}$$

# Today's topics

Recap: Optimization

- Effect of optimisers

Tricks of the Trade

- Shuffling
- Data Augmentation
- Normalization

Nonlinearities

Initialization

Advanced Techniques

- Batch Normalization
- Dropout

# Batch Normalization [Ioffe & Szegedy '14]

## Motivation

- Optimization works best if all inputs of a layer are normalized.

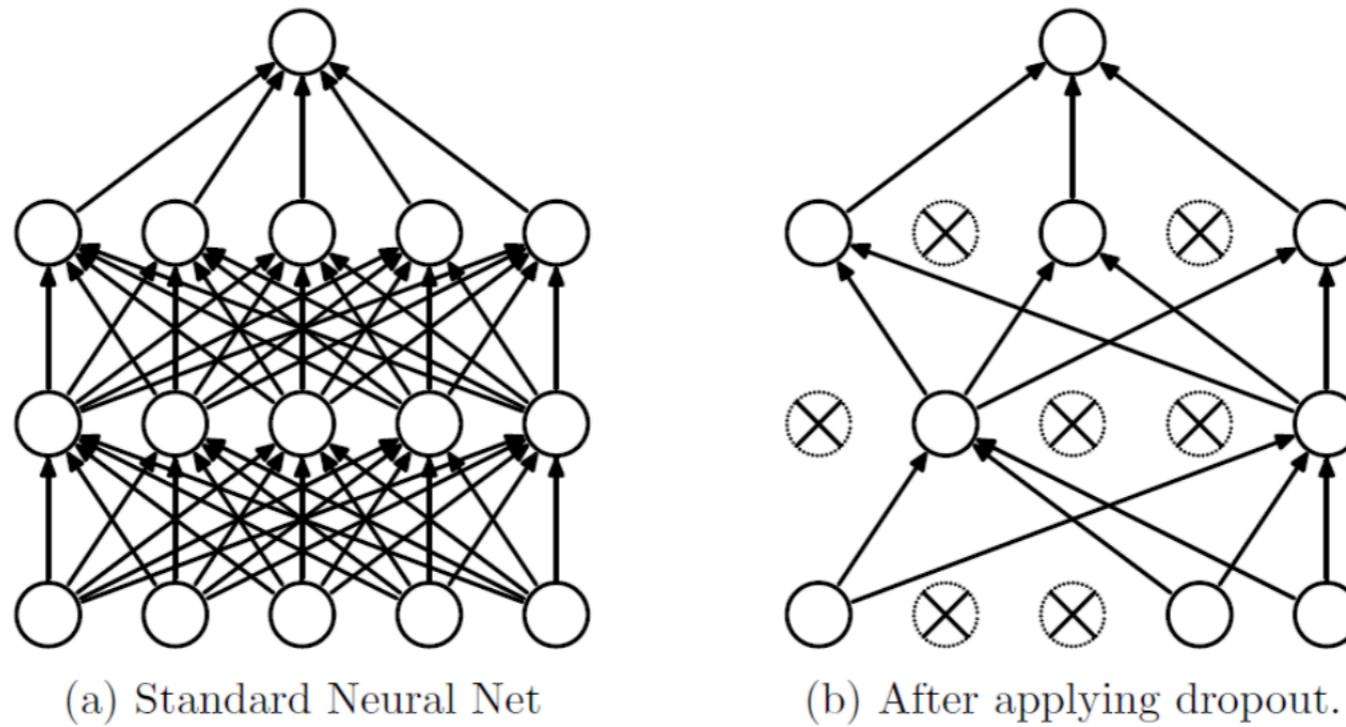
## Idea

- Introduce intermediate layer that centers the activations of the previous layer per minibatch.
- I.e., perform transformations on all activations and undo those transformations when backpropagating gradients
- Complication: centering + normalization also needs to be done at test time, but minibatches are no longer available at that point.
  - Learn the normalization parameters to compensate for the expected bias of the previous layer (usually a simple moving average)

## Effect

- Much improved convergence (but parameter values are important!)
- Widely used in practice

# Dropout [Srivastava, Hinton '12]



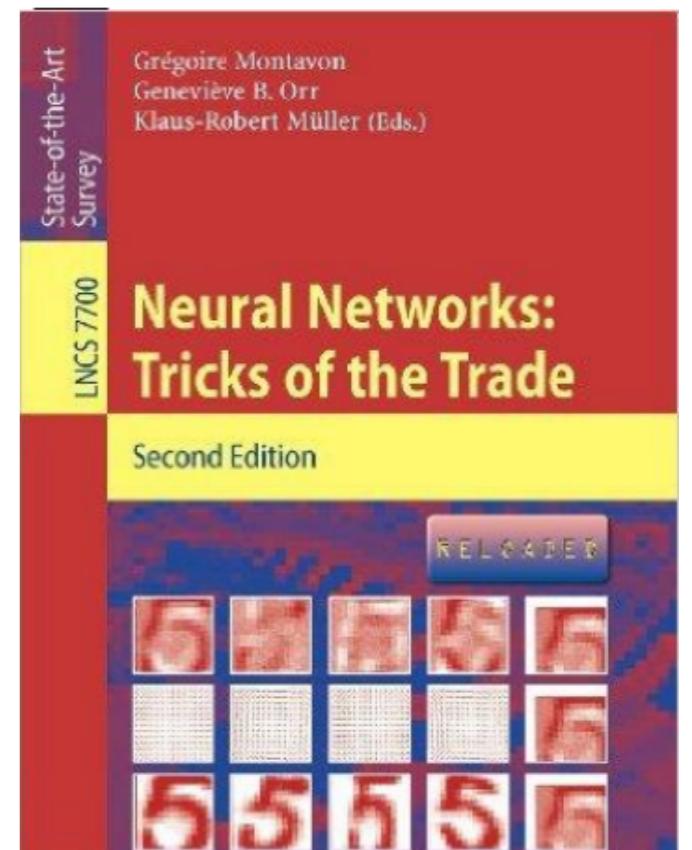
## Idea

- Randomly switch off units during training.
  - Change network architecture for each data point, effectively training many different variants of the network.
  - When applying the trained network, multiply activations with the probability that the unit was set to zero.
- Greatly improved performance

# References

More information on Neural Networks can be found in Chapter 1 of the book

G. Montavon, G. B. Orr, K-R Mueller  
(Eds.) Neural Networks: Tricks of the  
Trade Springer, 1998, 2012



Yann LeCun, Leon Bottou, Genevieve B. Orr, Klaus-Robert  
Mueller Efficient BackProp, Ch.1 of the above book., 1998

# References and Future Reading

## ReLU

- X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, AISTATS 2011.

## Initialization

- X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, AISTATS 2010.
- K. He, X.Y. Zhang, S.Q. Ren, J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, ArXiV 1502.01852v1, 2015.
- A.M. Saxe, J.L. McClelland, S. Ganguli, Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, ArXiV 1312.6120v3, 2014.

# References and Future Reading

- Batch Normalization
- S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ArXiV 1502.03167, 2015.

## Dropout

- N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR, Vol. 15:1929-1958, 2014.