RWTH AACHEN UNIVERSITY

Advanced Machine Learning
Lecture 14

Tricks of the Trade

17.12.2015

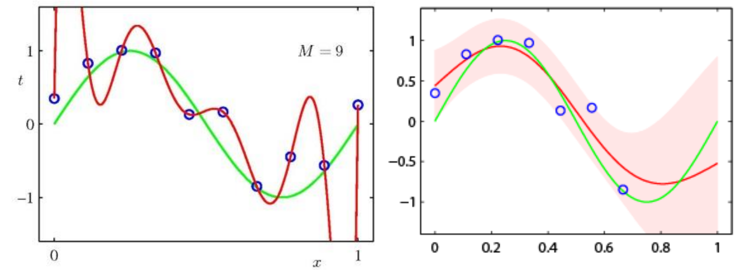
Bastian Leibe
RWTH Aachen
http://www.vision.rwth-aachen.de/

leibe@vision.rwth-aachen.de

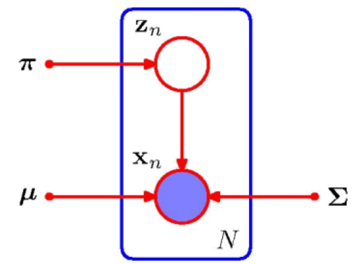
This Lecture: *Advanced Machine Learning*

- Regression Approaches
  - Linear Regression
  - Regularization (Ridge, Lasso)
  - Gaussian Processes

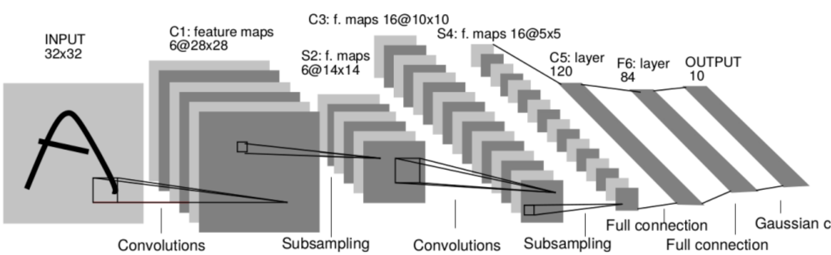
$$f : \mathcal{X} \rightarrow \mathbb{R}$$



- Learning with Latent Variables
  - Prob. Distributions & Approx. Inference
  - Mixture Models
  - EM and Generalizations



- Deep Learning
  - Linear Discriminants
  - Neural Networks
  - Backpropagation & Optimization
  - CNNs, RNNs, RBMs, etc.



Recap: Learning with Hidden Units

- **How can we train multi-layer networks efficiently?**
  - Need an efficient way of adapting all weights, not just the last layer.

- **Idea: Gradient Descent**
  - Set up an error function

$$E(\mathbf{W}) = \sum_n L(t_n, y(\mathbf{x}_n; \mathbf{W})) + \lambda\Omega(\mathbf{W})$$

  with a loss $L(\cdot)$ and a regularizer $\Omega(\cdot)$.

  - E.g., $L(t, y(\mathbf{x}; \mathbf{W})) = \sum_n \left(y(\mathbf{x}_n; \mathbf{W}) - t_n\right)^2$ — L_2 loss

$$\Omega(\mathbf{W}) = ||\mathbf{W}||_F^2$$

L_2 regularizer ("weight decay")

$\Rightarrow$ Update each weight $W_{ij}^{(k)}$ in the direction of the gradient $\frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(k)}}$

Gradient Descent

- Two main steps
  1. Computing the gradients for each weight — last lecture
  2. Adjusting the weights in the direction of the gradient — today

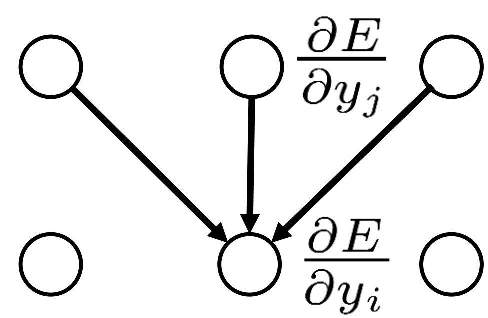
Recap: Backpropagation Algorithm

- Core steps
  1. Convert the discrepancy between each output and its target value into an error derivate.

$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$

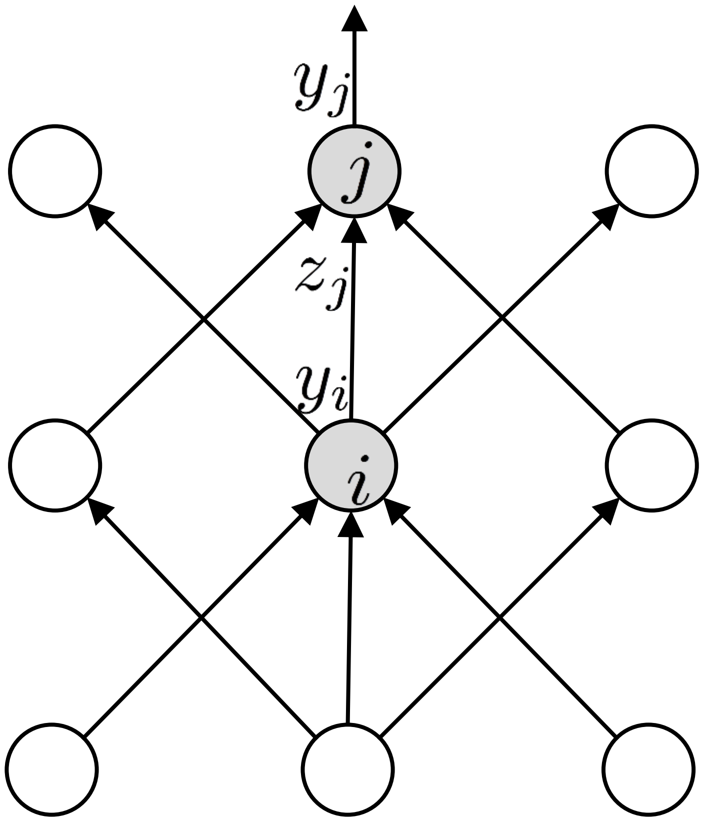
  2. Compute error derivatives in each hidden layer from error derivatives in the layer above.



  3. Use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights

$$\frac{\partial E}{\partial y_j} \longrightarrow \frac{\partial E}{\partial w_{ik}}$$

Recap: Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

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

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

- **Efficient propagation scheme**
  - y_i is already known from forward pass! (Dynamic Programming)

  $\Rightarrow$ Propagate back the gradient from layer j and multiply with y_i.

Recap: MLP Backpropagation Algorithm

- **Forward Pass**

$$\mathbf{y}^{(0)} = \mathbf{x}$$

for $k = 1, \ldots, l$ do

$$\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{y}^{(k-1)}$$

$$\mathbf{y}^{(k)} = g_k(\mathbf{z}^{(k)})$$

endfor

$$\mathbf{y} = \mathbf{y}^{(l)}$$

$$E = L(\mathbf{t}, \mathbf{y}) + \lambda\Omega(\mathbf{W})$$

- **Backward Pass**

$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}} L(\mathbf{t}, \mathbf{y}) + \lambda \frac{\partial}{\partial \mathbf{y}}\Omega$$

for $k = l, l\text{-}1, \ldots, 1$ do

$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{z}^{(k)}} = \mathbf{h} \odot g'(\mathbf{y}^{(k)})$$

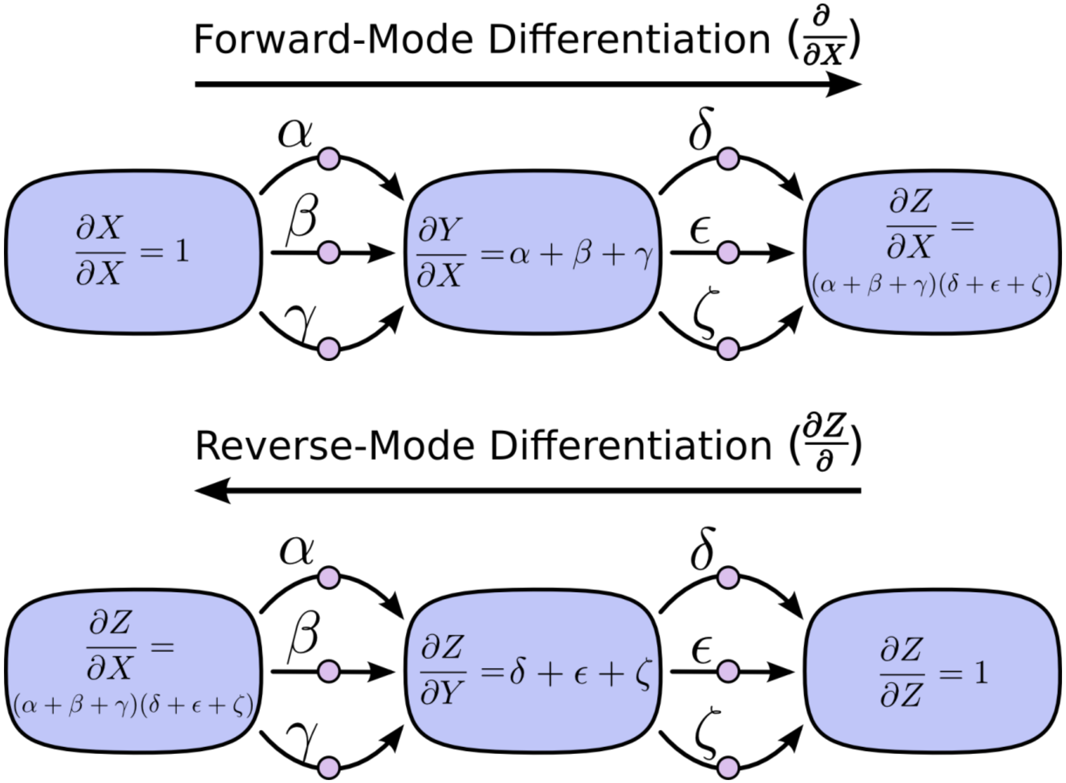
$$\frac{\partial E}{\partial \mathbf{W}^{(k)}} = \mathbf{h}\mathbf{y}^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial \mathbf{W}^{(k)}}$$

$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}^{(k-1)}} = \mathbf{W}^{(k)\top}\mathbf{h}$$

endfor

- **Notes**
  - For efficiency, an entire batch of data $\mathbf{X}$ is processed at once.
  - $\odot$ denotes the element-wise product

Recap: Computational Graphs



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

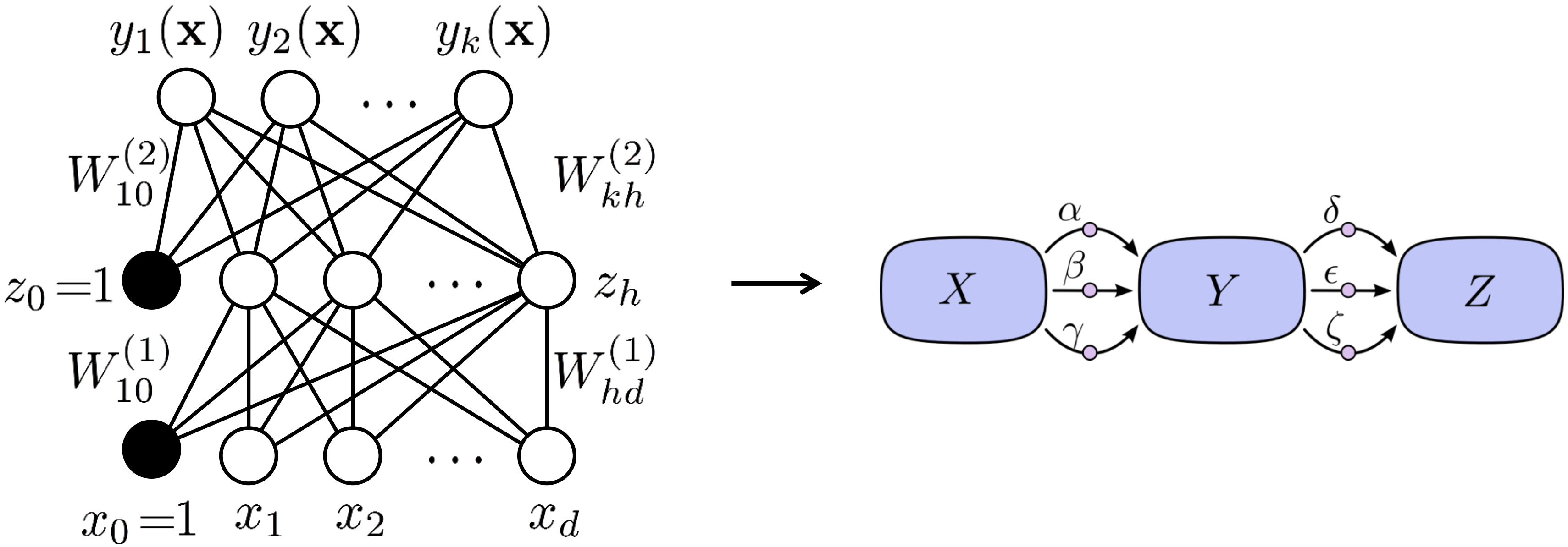
Apply operator $\frac{\partial Z}{\partial}$ to every node.

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

$\Rightarrow$ **Speed-up in $\mathcal{O}($#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

Image source: Christopher Olah, colah.github.io

Topics of This Lecture

- **Gradient Descent Revisited**
- **Data (Pre-)processing**
  - Stochastic Gradient Descent & Minibatches
  - Data Augmentation
  - Normalization
  - Initialization
- **Convergence of Gradient Descent**
  - Choosing Learning Rates
  - Momentum & Nesterov Momentum
  - RMS Prop
  - Other Optimizers
- **Other Tricks**
  - Batch Normalization
  - Dropout

Gradient Descent

- Two main steps
  1. Computing the gradients for each weight — last lecture
  2. Adjusting the weights in the direction of the gradient — today

- Recall: Basic update equation

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

- Main questions
  - On what data do we want to apply this?
  - How should we choose the step size η (the learning rate)?
  - In which direction should we update the weights?

Topics of This Lecture

- Gradient Descent
- Data (Pre-)processing
  - Stochastic Gradient Descent & Minibatches
  - Data Augmentation
  - Normalization
  - Initialization
- Convergence of Gradient Descent
  - Choosing Learning Rates
  - Momentum & Nesterov Momentum
  - RMS Prop
  - Other Optimizers
- Other Tricks
  - Batch Normalization
  - Dropout

Stochastic vs. Batch Learning

- Batch learning
  - Process the full dataset at once to compute the gradient.

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

- Stochastic learning
  - Choose a single example from the training set.
  - Compute the gradient only based on this example
  - This estimate will generally be noisy, which has some advantages.

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

Stochastiv vs. Batch Learning

- Batch learning advantages
  - Conditions of convergence are well understood.
  - Many acceleration techniques (e.g., conjugate gradients) only operate in batch learning.
  - Theoretical analysis of the weight dynamics and convergence rates are simpler.

- Stochastic learning advantages
  - Usually much faster than batch learning.
  - Often results in better solutions.
  - Can be used for tracking changes.

- Middle ground: Minibatches

Minibatches

- Idea
  - Process only a small batch of training examples together
  - Start with a small batch size & increase it as training proceeds.
- Advantages
  - Gradients will more stable than for stochastic gradient descent, but still faster to compute than with batch learning.
  - Take advantage of redundancies in the training set.
  - Matrix operations are more efficient than vector operations.
- Caveat
  - Error function should be normalized by the minibatch size, s.t. we can keep the same learning rate between minibatches

$$E(\mathbf{W}) = \frac{1}{N} \sum_n L(t_n, y(\mathbf{x}_n; \mathbf{W})) + \frac{\lambda}{N} \Omega(\mathbf{W})$$

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.
  - 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

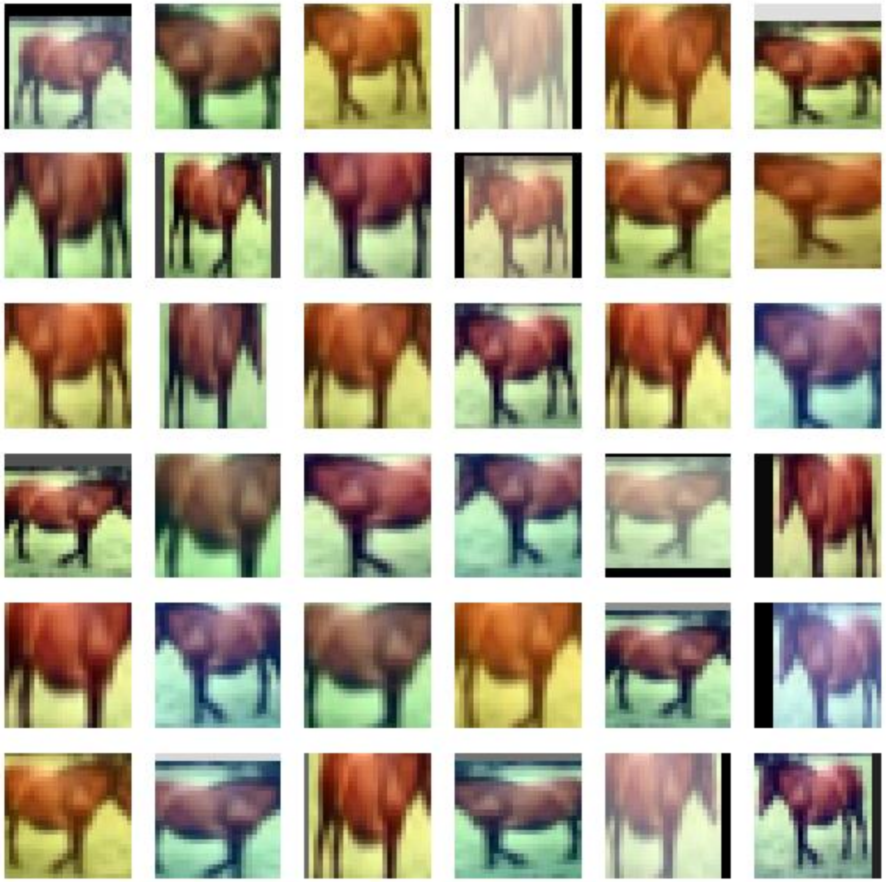
  - Flipping

  - Color PCA

Image source: Lucas Beyer

Data Augmentation

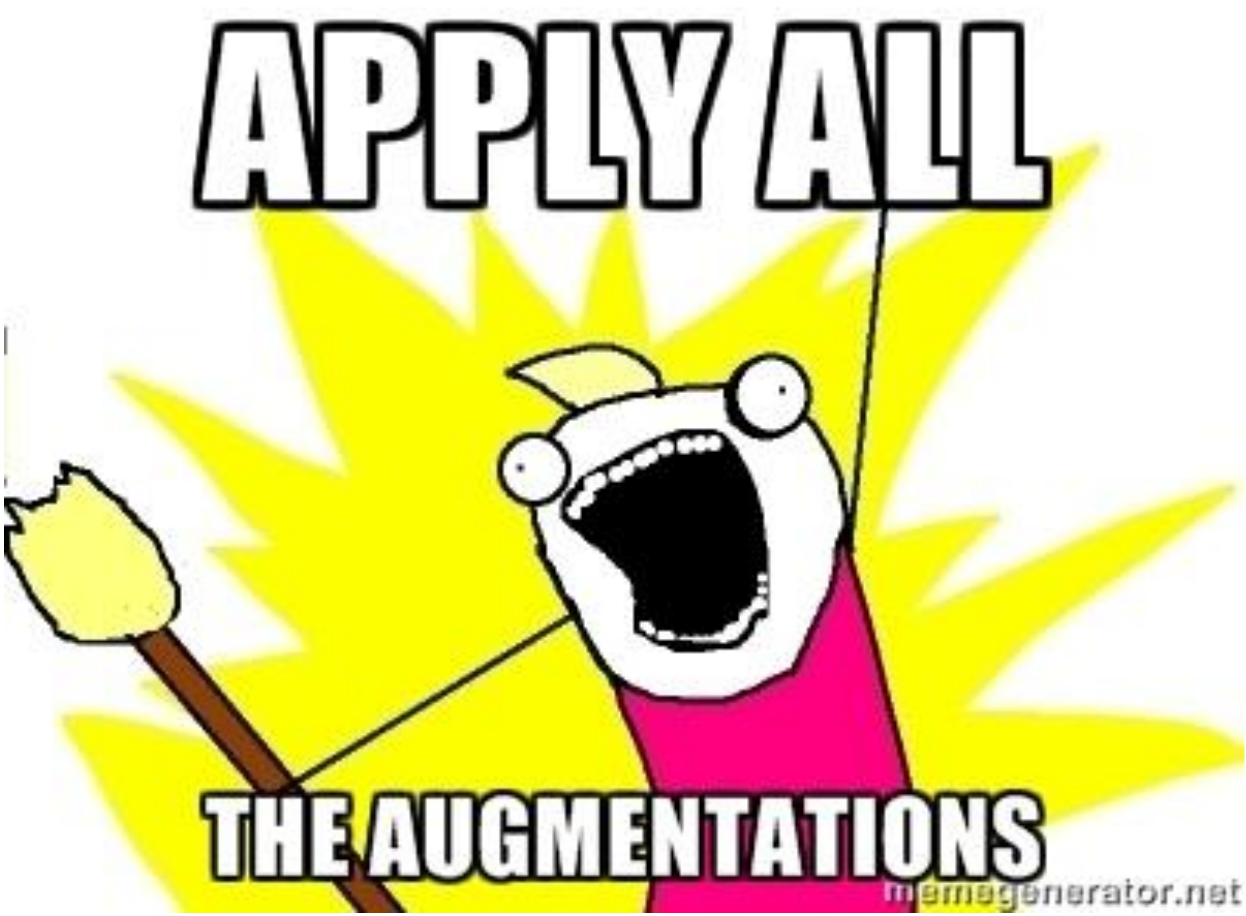
- Effect
  - Much larger training set
  - Robustness against expected variations
- During testing
  - When cropping was used during training, need to again apply crops to get same image size.
  - Beneficial to also apply flipping during test.
  - Applying several ColorPCA variations can bring another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



Augmented training data (from one original image)

Image source: Lucas Beyer

General Guideline



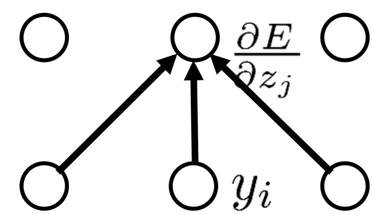
Normalization

- **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

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



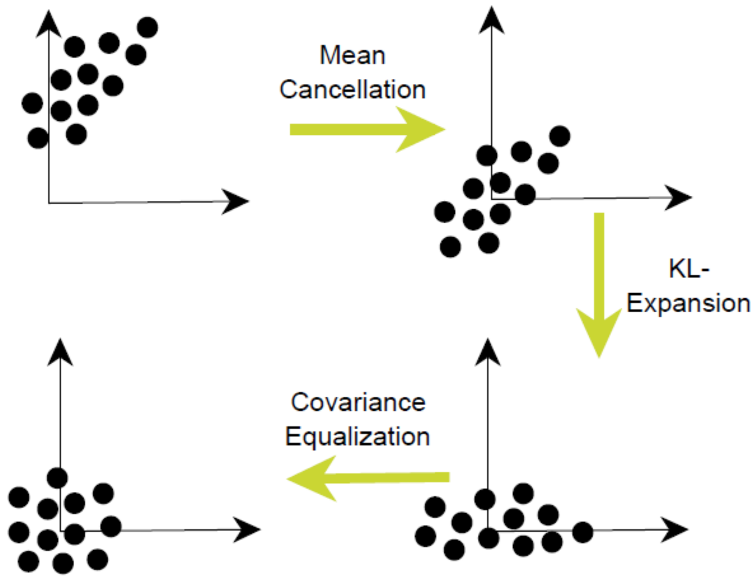
  - When all of the components of the input vector y_i are positive, all of the updates of weights that feed into a node will be of the same sign.

$\Rightarrow$ Weights can only all increase or decrease together.

$\Rightarrow$ Slow convergence

Normalizing the Inputs

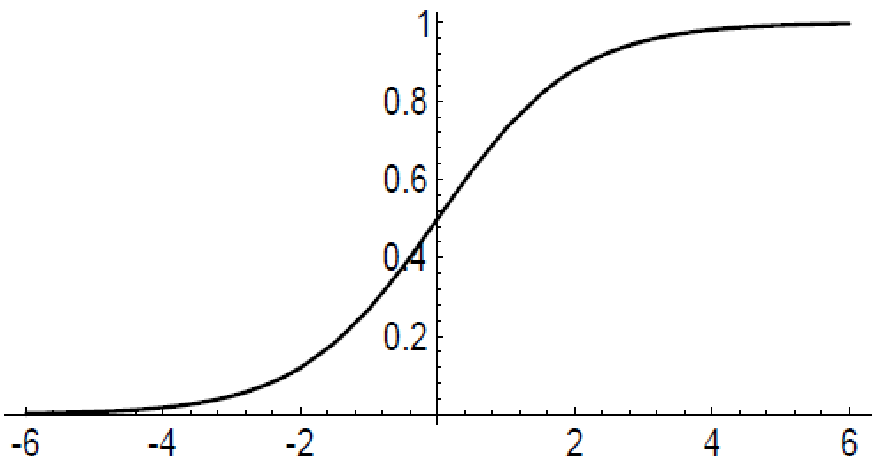
- **Convergence is fastest if**
  - The mean of each input variable over the training set is zero.
  - The inputs are scaled such that all have the same covariance.
  - Input variables are uncorrelated if possible.

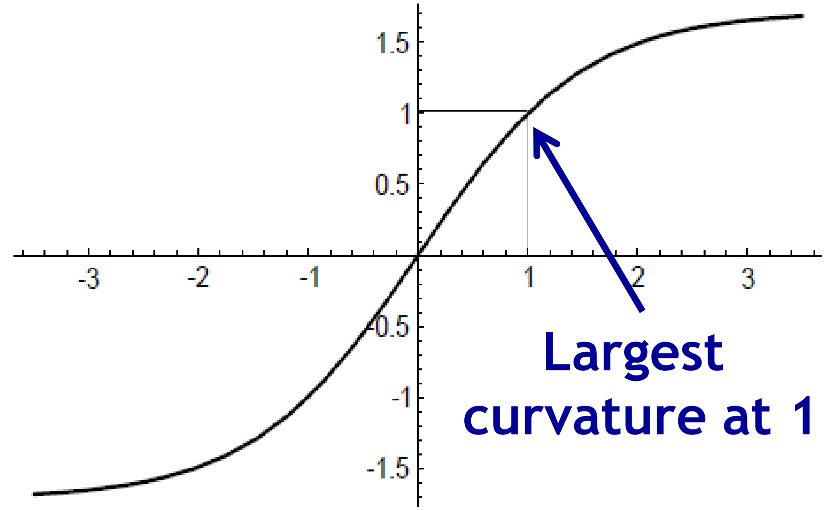


- **Advisable normalization steps (for MLPs)**
  - Normalize all inputs that an input unit sees to zero-mean, unit covariance.
  - If possible, try to decorrelate them using PCA (also known as Karhunen-Loeve expansion).

Image source: Yann LeCun et al., Efficient BackProp (1998)

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:**

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

$\Rightarrow$ **When used with transformed inputs, the variance of the outputs will be close to 1.**

Image source: Yann LeCun et al., Efficient BackProp (1998)

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**
  - Assuming that
    - The training set has been normalized
    - The sigmoid $f(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$ is used

  the initial weights should be randomly drawn from a distribution (e.g., uniform or Normal) with mean zero and standard deviation

$$\sigma_w = m^{-1/2}$$

  where m is the fan-in (#connections into the node).

Topics of This Lecture

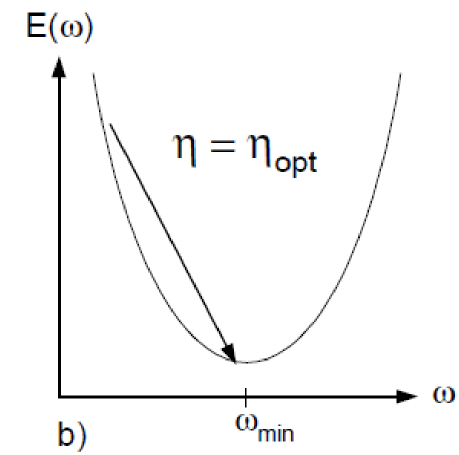
- Gradient Descent
- Data (Pre-)processing
  - Stochastic Gradient Descent & Minibatches
  - Data Augmentation
  - Normalization
  - Initialization
- Convergence of Gradient Descent
  - Choosing Learning Rates
  - Momentum & Nesterov Momentum
  - RMS Prop
  - Other Optimizers
- Other Tricks
  - Batch Normalization
  - Dropout

Choosing the Right Learning Rate

- **Analyzing the convergence of Gradient Descent**
  - **Consider a simple 1D example first**

$$W^{(\tau-1)} = W^{(\tau)} - \eta \frac{\mathrm{d}E(W)}{\mathrm{d}W}$$

  - **What is the optimal learning rate η_{opt}?**



  - **If E is quadratic, the optimal learning rate is given by the inverse of the Hessian**

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

  - ***What happens if we exceed this learning rate?***

Image source: Yann LeCun et al., Efficient BackProp (1998)

Choosing the Right Learning Rate

- **Behavior for different learning rates**

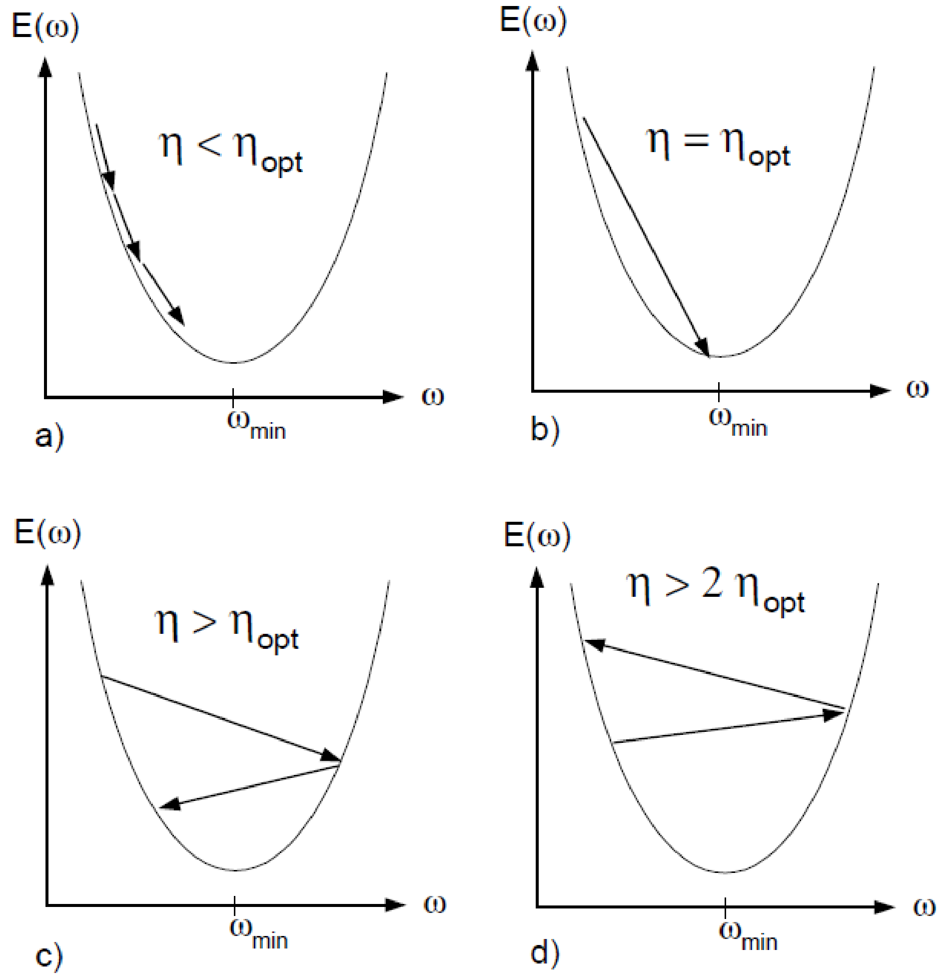


Image source: Yann LeCun et al., Efficient BackProp (1998)

Learning Rate vs. Training Error

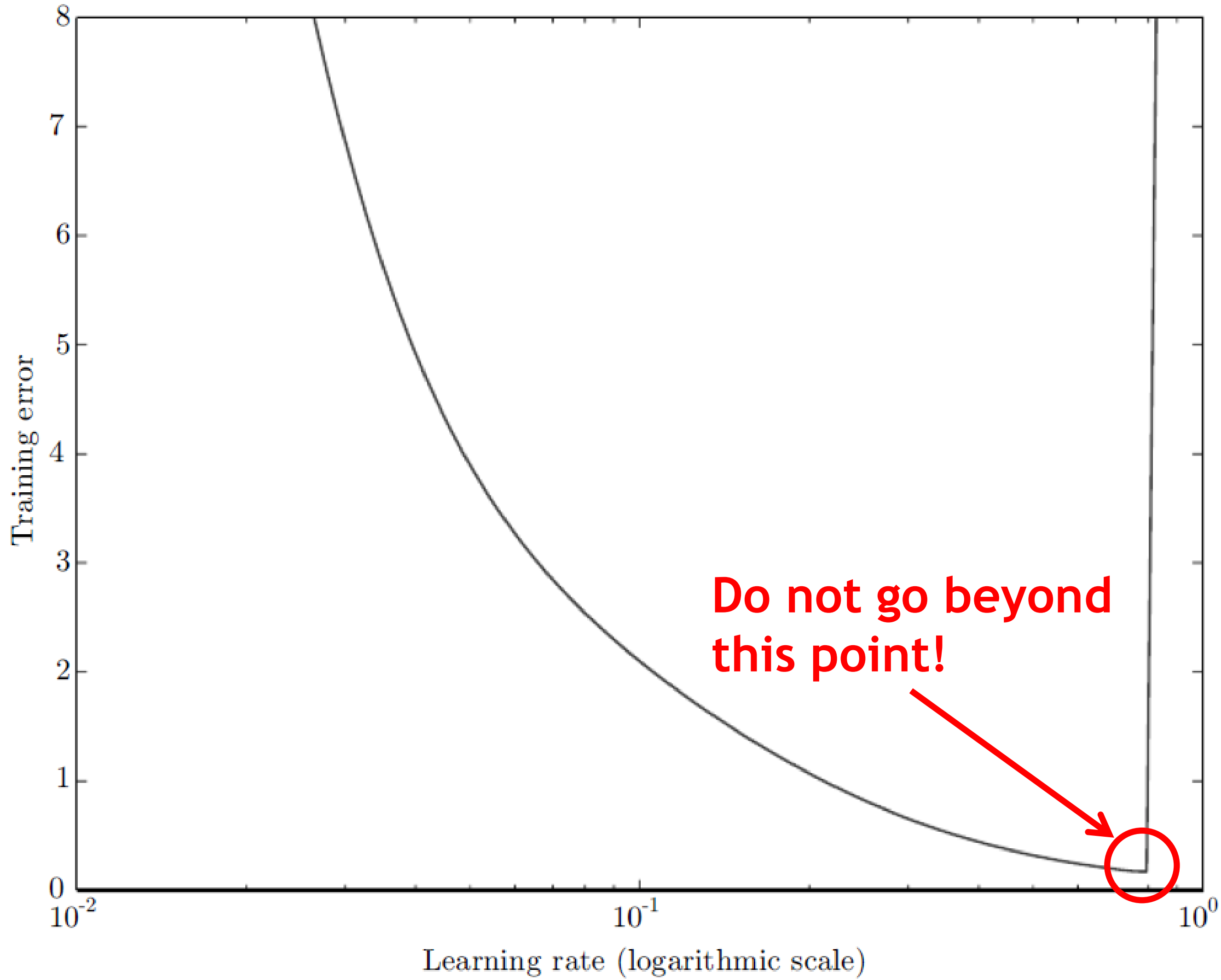
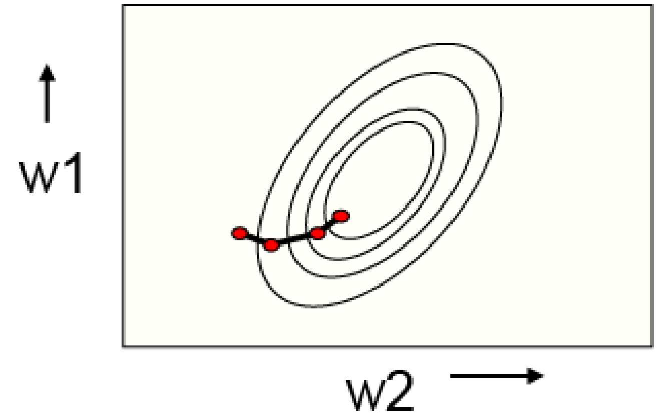


Image source: Goodfellow & Bengio book

Batch vs. Stochastic Learning

- **Batch Learning**
  - Simplest case: steepest decent on the error surface.

  ⇒ Updates perpendicular to contour lines



- **Stochastic Learning**
  - Simplest case: zig-zag around the direction of steepest descent.

  ⇒ Updates perpendicular to constraints from training examples.

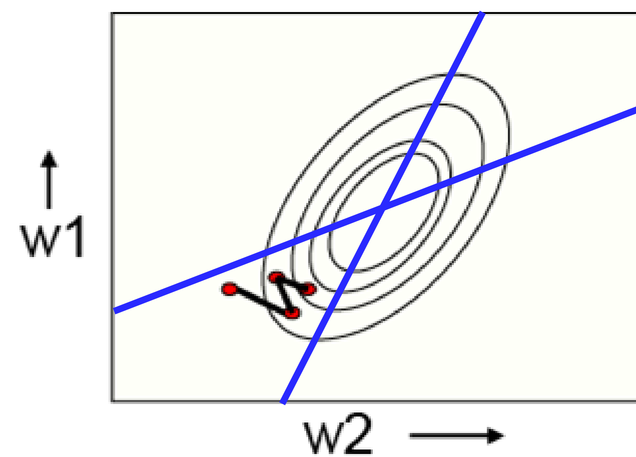
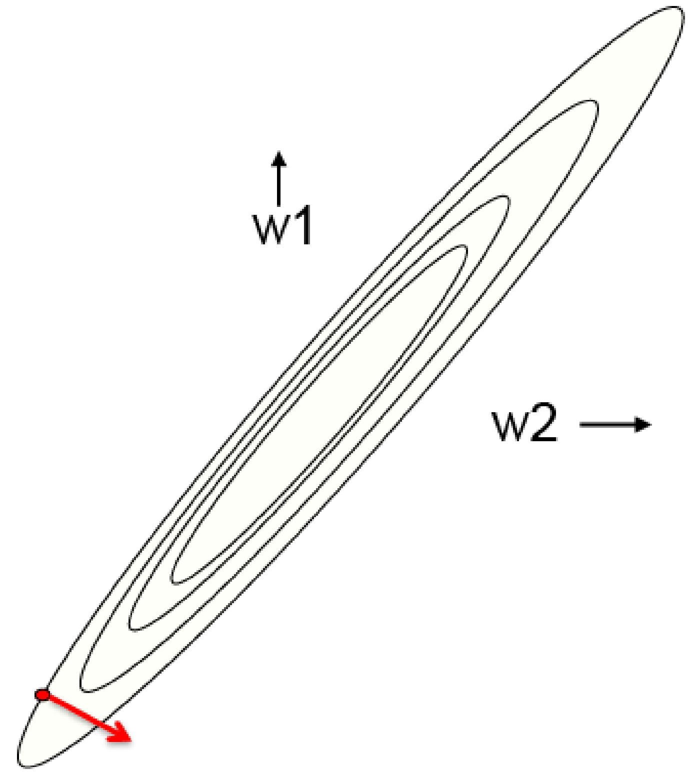


Image source: Geoff Hinton

Why Learning Can Be Slow

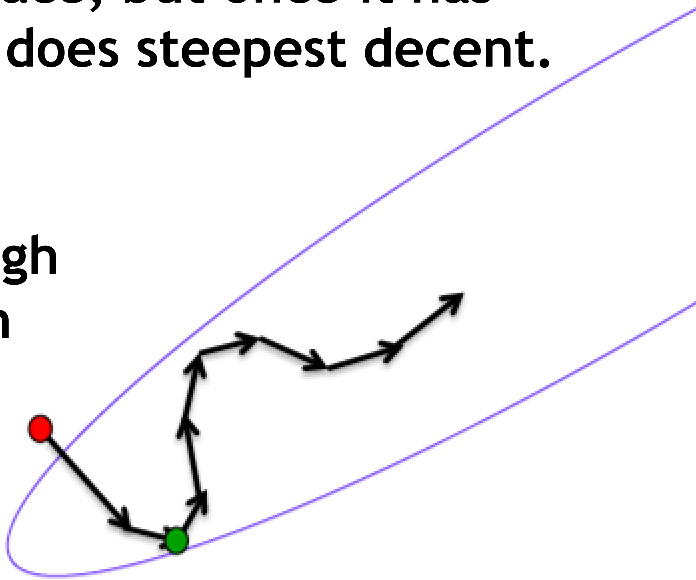
- If the inputs are correlated
  - The ellipse will be very elongated.
  - The direction of steepest descent is almost perpendicular to the direction towards the minimum!



This is just the opposite of what we want!

The Momentum Method

- **Idea**
  - Instead of using the gradient to change the position of the weight "particle", use it to change the velocity.
- **Intuition**
  - Example: Ball rolling on the error surface
  - It starts off by following the error surface, but once it has accumulated momentum, it no longer does steepest decent.
- **Effect**
  - Dampen oscillations in directions of high curvature by combining gradients with opposite signs.
  - Build up speed in directions with a gentle but consistent gradient.



The Momentum Method: Implementation

- **Change in the update equations**
  - **Effect of the gradient: increment the previous velocity, subject to a decay by $\alpha < 1$.**

$$\mathbf{v}(t) = \alpha \mathbf{v}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)$$

  - **Set the weight change to the current velocity**

$$\begin{aligned}
\Delta \mathbf{w} &= \mathbf{v}(t) \\
&= \alpha \mathbf{v}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t) \\
&= \alpha \Delta \mathbf{w}(t-1) - \varepsilon \frac{\partial E}{\partial \mathbf{w}}(t)
\end{aligned}$$

The Momentum Method: Behavior

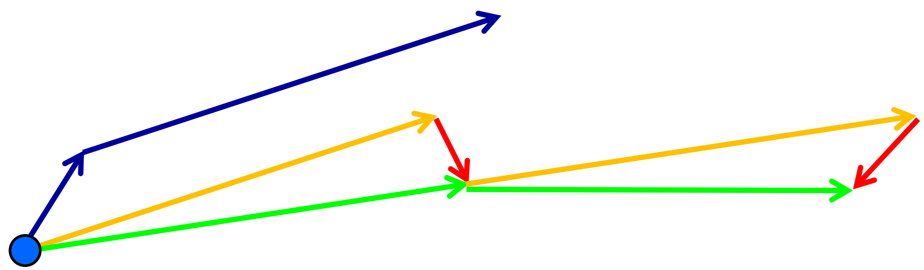
- Behavior
  - If the error surface is a tilted plane, the ball reaches a terminal velocity

$$\mathbf{v}(\infty) = \frac{1}{1-\alpha}\left(-\varepsilon \frac{\partial E}{\partial \mathbf{w}}\right)$$

    - If the momentum α is close to 1, this is much faster than simple gradient descent.
  - At the beginning of learning, there may be very large gradients.
    - Use a small momentum initially (e.g., $\alpha = 0.5$).
    - Once the large gradients have disappeared and the weights are stuck in a ravine, the momentum can be smoothly raised to its final value (e.g., $\alpha = 0.90$ or even $\alpha = 0.99$).

$\Rightarrow$ This allows us to learn at a rate that would cause divergent oscillations without the momentum.

Improvement: Nesterov-Momentum



Standard Momentum
Jump
Correction
Accumulated gradient

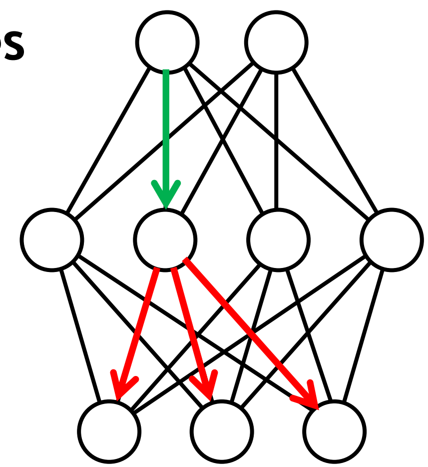
- **Standard Momentum method**
  - First compute the gradient at the current location
  - Then jump in the direction of the updated accumulated gradient
- **Improvement [Sutskever 2012]**
  - (Inspiration: Nesterov method for optimizing convex functions.)
  - First jump in the direction of the previous accumulated gradient
  - Then measure the gradient where you end up and make a correction.

$\Rightarrow$ *Intuition: It's better to correct a mistake after you've made it.*

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.

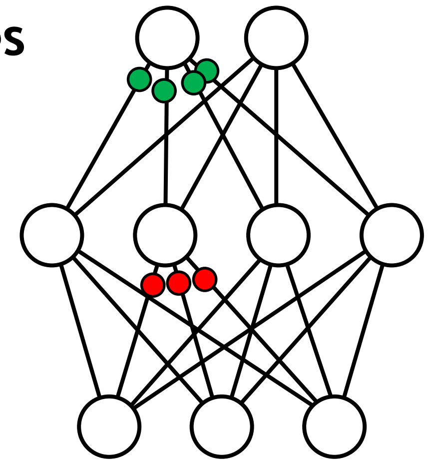
    ⇒ Gradients can get very small in the early layers of deep nets.



RWTH AACHEN UNIVERSITY

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.
  - The fan-in of a unit determines the size of the "overshoot" effect when changing multiple weights simultaneously to correct the same error.
    - The fan-in often varies widely between layers



- Solution
  - Use a global learning rate, multiplied by a local gain per weight (determined empirically)

Adaptive Learning Rates

- One possible strategy
  - Start with a local gain of 1 for every weight
  - Increase the local gain if the gradient for the weight does not change the sign.
  - Use small additive increases and multiplicative decreases (for mini-batch)

$$\Delta w_{ij} = -\varepsilon g_{ij} \frac{\partial E}{\partial w_{ij}}$$

$$\text{if } \left(\frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0$$

$$\text{then } g_{ij}(t) = g_{ij}(t-1) + 0.05$$

$$\text{else } g_{ij}(t) = g_{ij}(t-1) * 0.95$$

$\Rightarrow$ Big gains will decay rapidly once oscillation starts.

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

$$MeanSq(w_{ij}, t) = 0.9 MeanSq(w_{ij}, t-1) + 0.1 \left(\frac{\partial E}{\partial w_{ij}}(t) \right)^2$$

  - Divide the gradient by $\mathrm{sqrt}(MeanSq(w_{ij}, t))$.

Other Optimizers (Lucas)

- 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

RWTH AACHEN UNIVERSITY

Behavior in a Long Valley

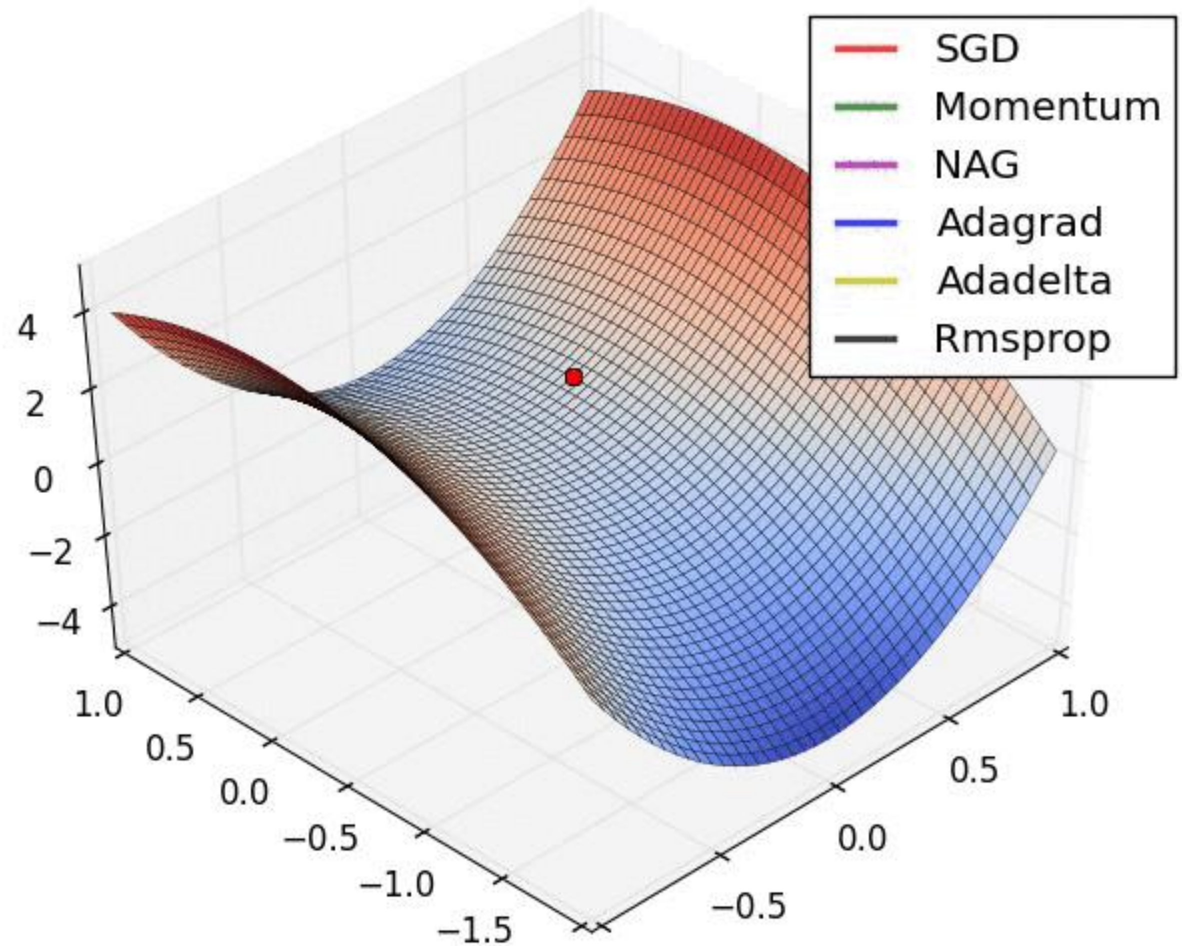


Image source: Aelc Radford, http://imgur.com/a/Hqolp

Behavior around a Saddle Point

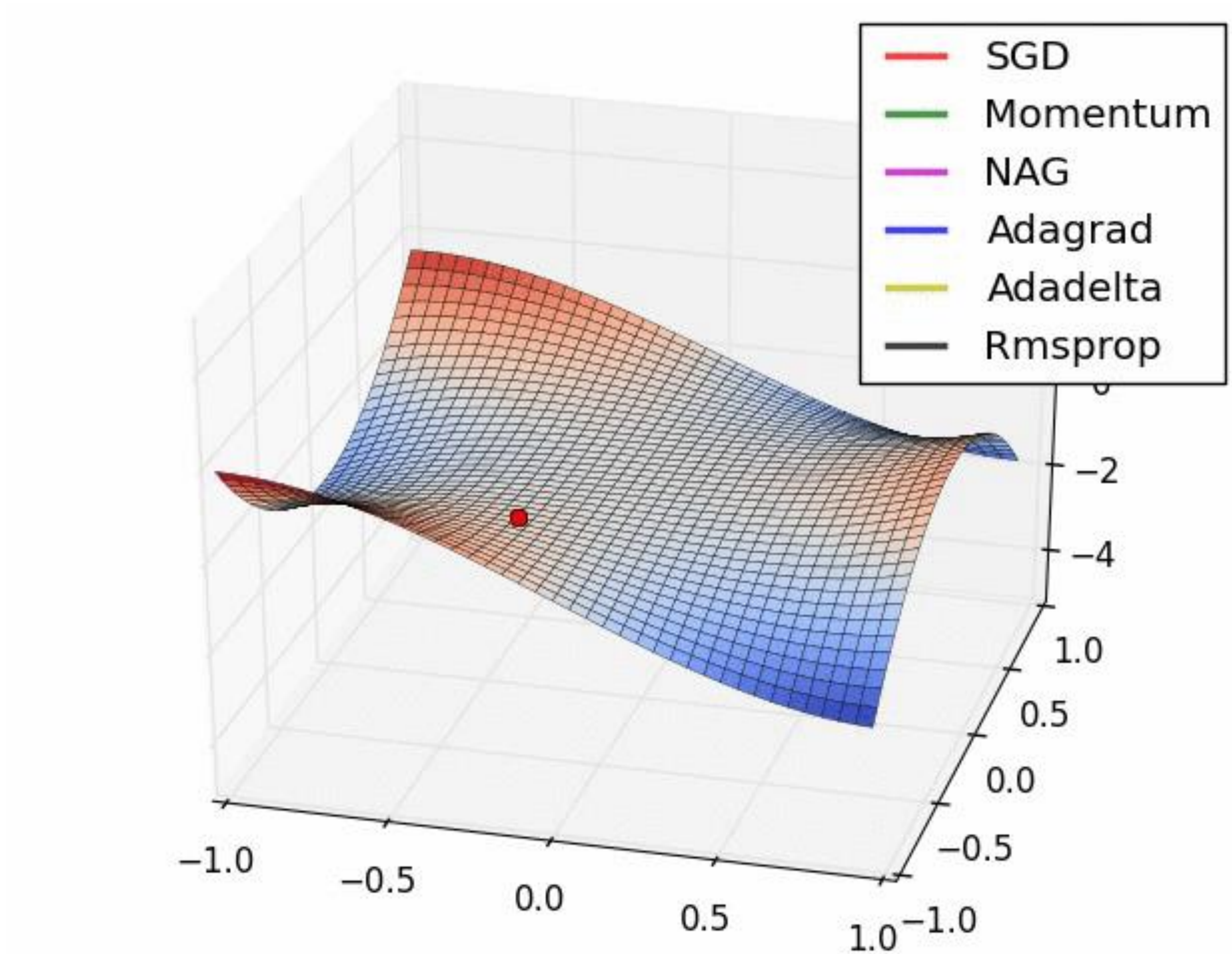


Image source: Aelc Radford, http://imgur.com/a/Hqolp

Visualization of Convergence Behavior

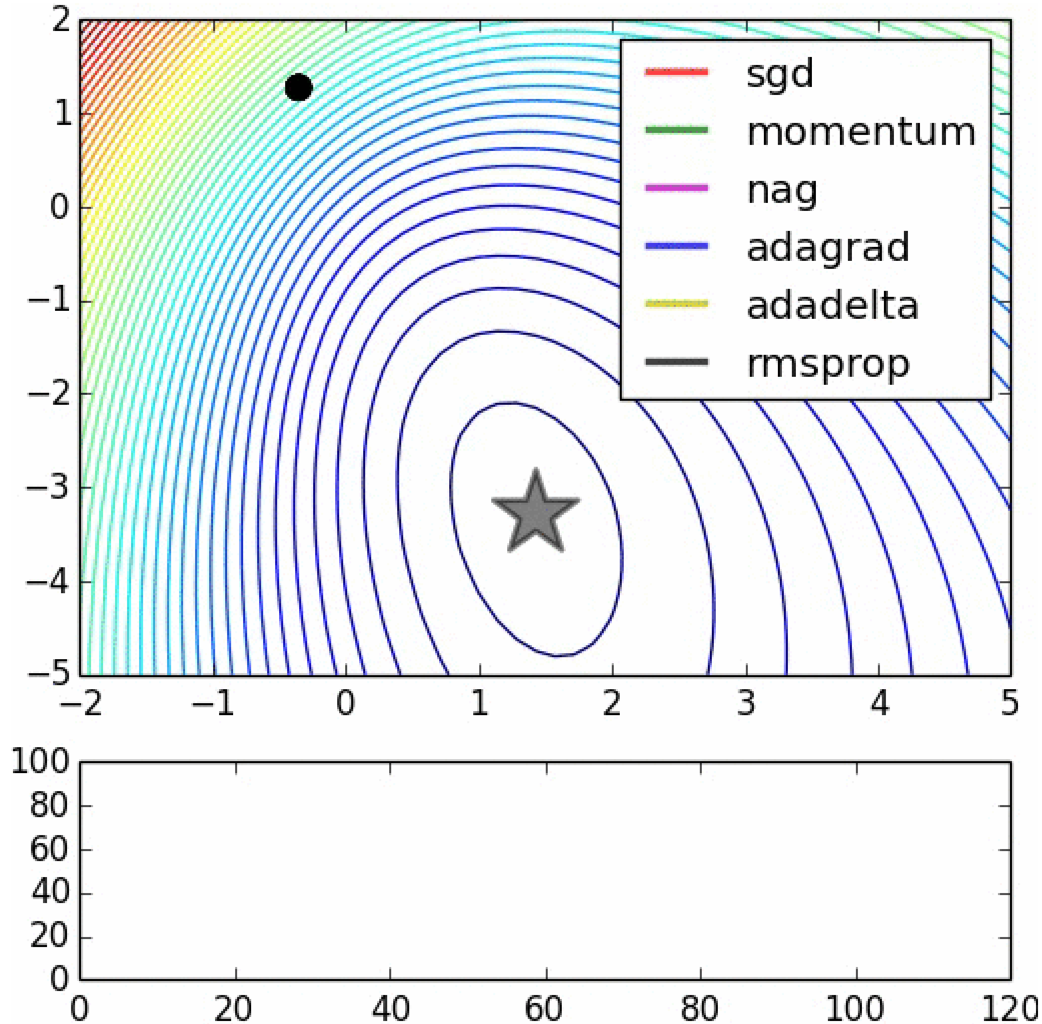
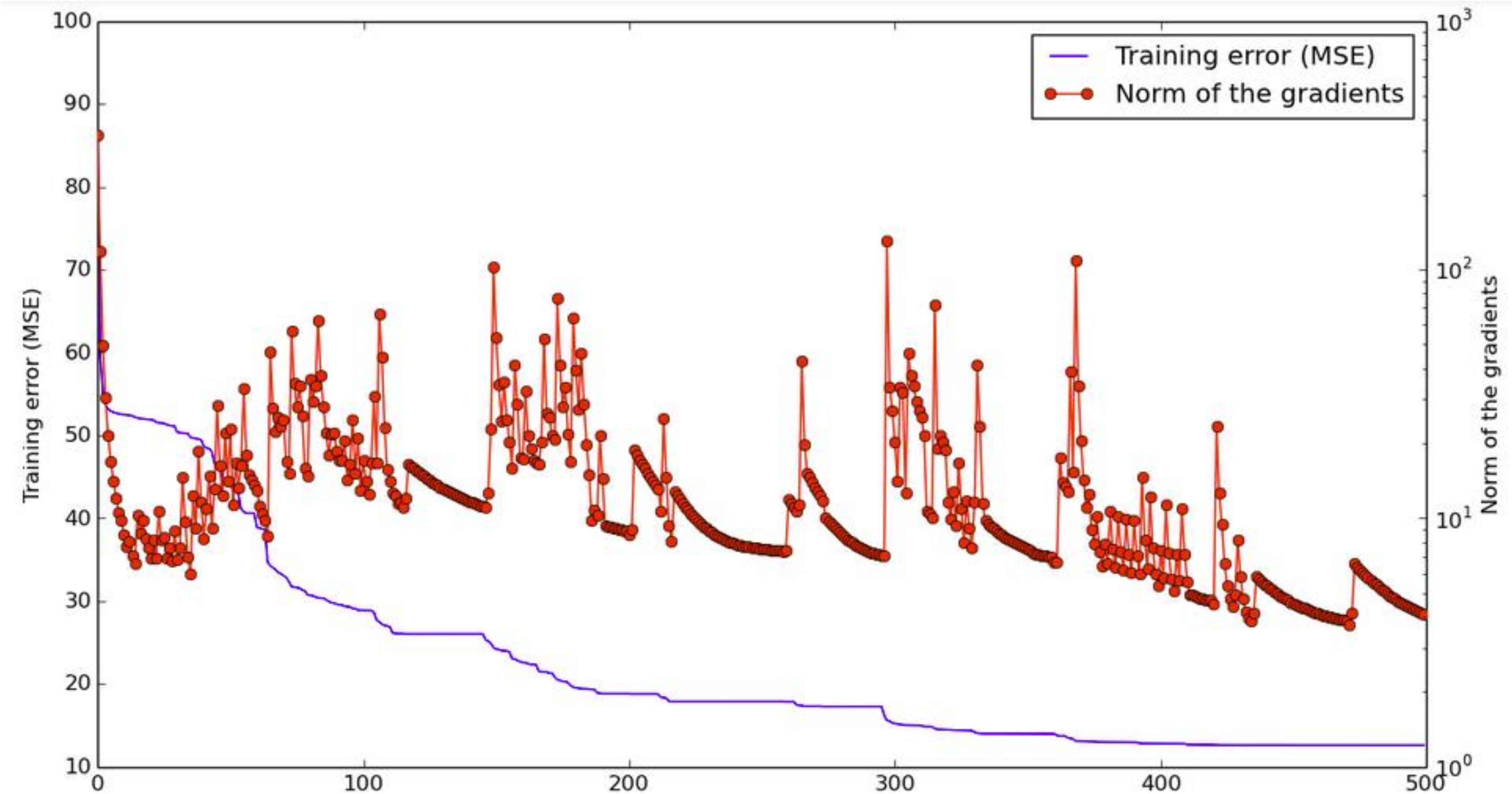


Image source: Aelc Radford, http://imgur.com/SmDARzn

Trick: Patience

- Saddle points dominate in high-dimensional spaces!

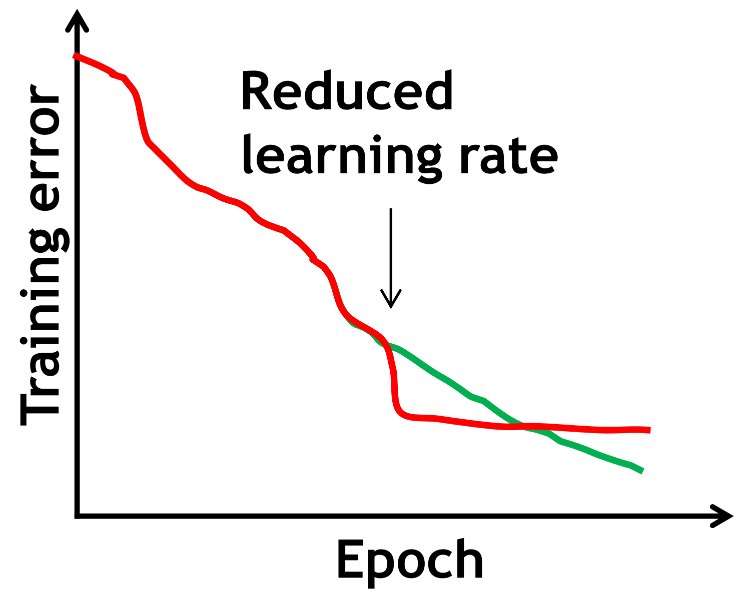


$\Rightarrow$ **Learning often doesn't get stuck, you just may have to wait...**

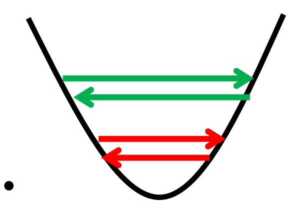
Image source: Yoshua Bengio

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 minibatches.**



- ***Be careful: Do not turn down the learning rate too soon!***
  - **Further progress will be much slower after that.**

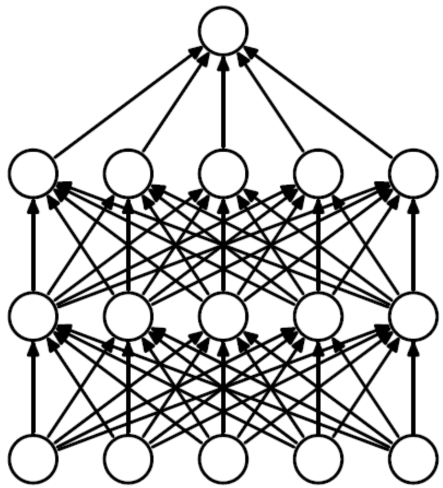
Topics of This Lecture

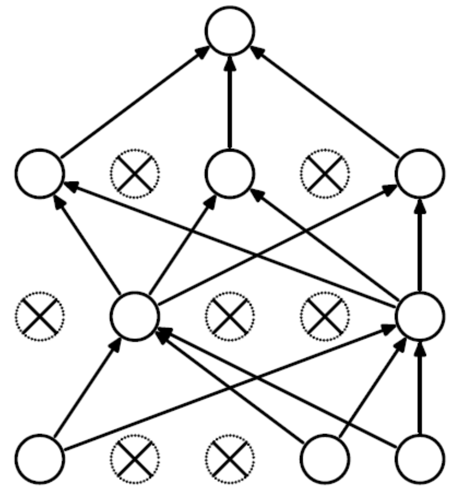
- Gradient Descent
- Data (Pre-)processing
  - Stochastic Gradient Descent & Minibatches
  - Data Augmentation
  - Normalization
  - Initialization
- Convergence of Gradient Descent
  - Choosing Learning Rates
  - Momentum & Nesterov Momentum
  - RMS Prop
  - Other Optimizers
- **Other Tricks**
  - **Batch 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
- **Effect**
  - Much improved convergence

Dropout [Srivastava, Hinton ’12]


(a) Standard Neural Net


(b) After applying dropout.

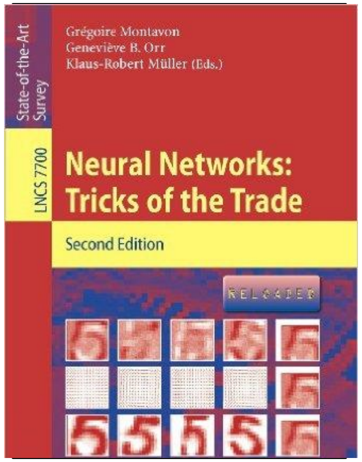
- **Idea**
  - **Randomly switch off units during training.**
  - **Change network architecture for each data point, effectively training many different variants of the network.**
  - **When applying the trained network, multiply activations with the probability that the unit was set to zero.**

  ⇒ **Greatly improved performance**

References and Further Reading

- More information on many practical tricks 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.