Machine Learning – Lecture 13

Neural Networks II

27.11.2019

Bastian Leibe

RWTH Aachen

http://www.vision.rwth-aachen.de

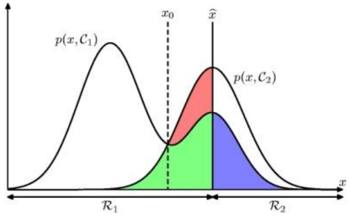
leibe@vision.rwth-aachen.de

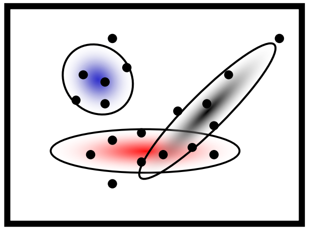
Course Outline

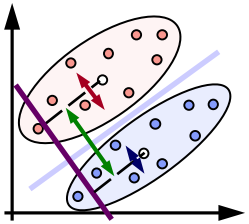
- Fundamentals
  - Bayes Decision Theory
  - Probability Density Estimation

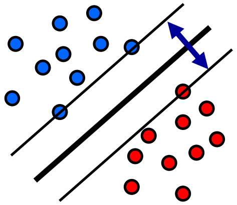
- Classification Approaches
  - Linear Discriminants
  - Support Vector Machines
  - Ensemble Methods & Boosting
  - Random Forests

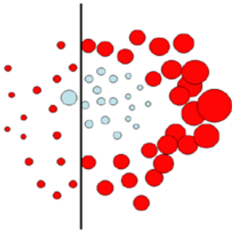
- Deep Learning
  - Foundations
  - Convolutional Neural Networks
  - Recurrent Neural Networks

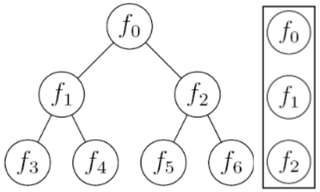


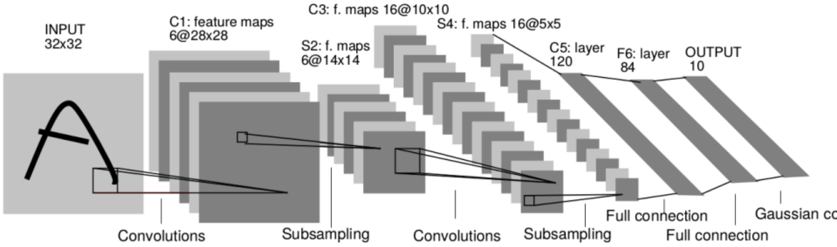










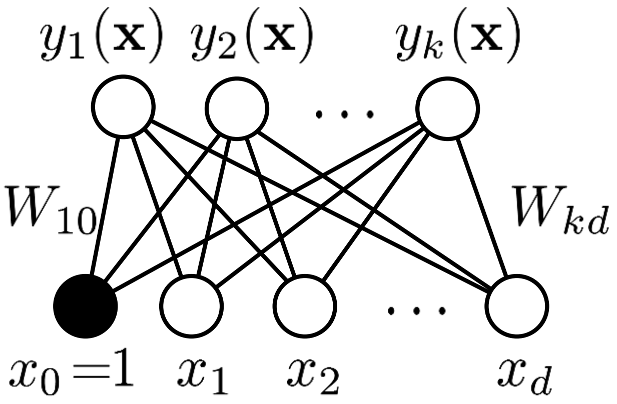


Topics of This Lecture

- Learning Multi-layer Networks
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Recap: Perceptrons

- One output node per class



Output layer

Weights

Input layer

- Outputs
  - Linear outputs

$$y_k(\mathbf{x}) = \sum_{i=0}^{d} W_{ki} x_i$$

With output nonlinearity

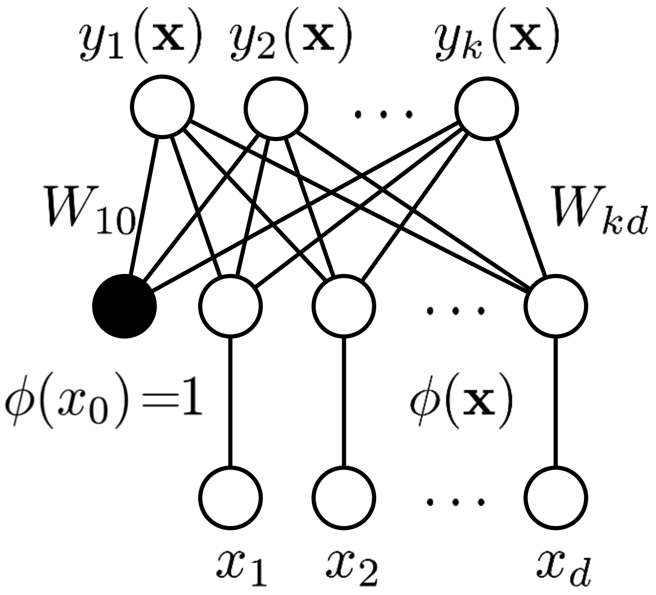
$$y_k(\mathbf{x}) = g\left(\sum_{i=0}^{d} W_{ki} x_i\right)$$

⇒ Can be used to do multidimensional linear regression or multiclass classification.

Slide adapted from Stefan Roth

Recap: Non-Linear Basis Functions

- Straightforward generalization



Output layer

Weights

Feature layer

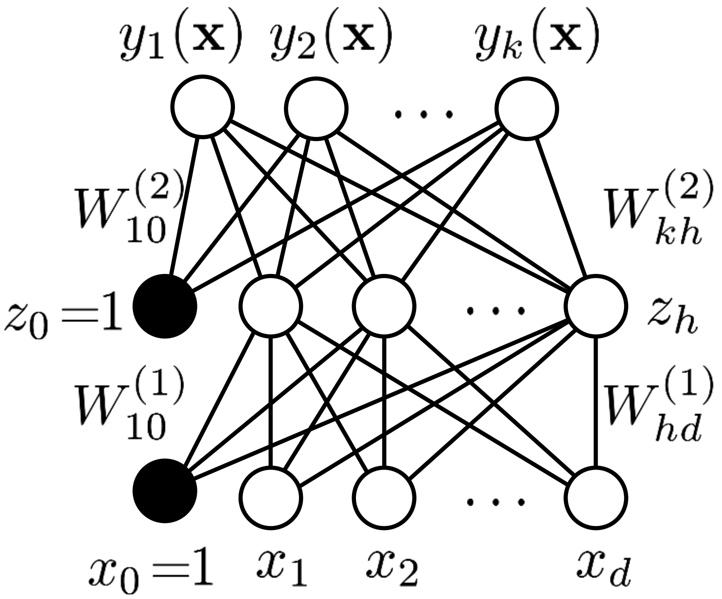
Mapping (fixed)

Input layer

- Remarks
  - Perceptrons are generalized linear discriminants!
  - Everything we know about the latter can also be applied here.
  - Note: feature functions $\phi(\mathbf{x})$ are kept fixed, not learned!

Recap: Multi-Layer Perceptrons

- Adding more layers



Output layer

Hidden layer

Mapping (learned!)

Input layer

- Output

$$y_k(\mathbf{x}) = g^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j\right)\right)$$

Slide adapted from Stefan Roth

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”)

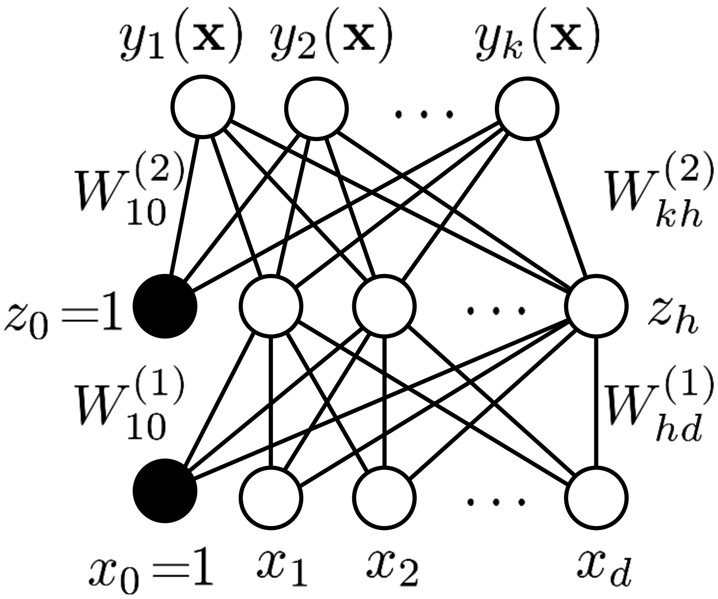
$\Rightarrow$ Update each weight $W_{ij}^{(k)}$ in the direction of the gradient $\frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(k)}}$

Gradient Descent

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

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



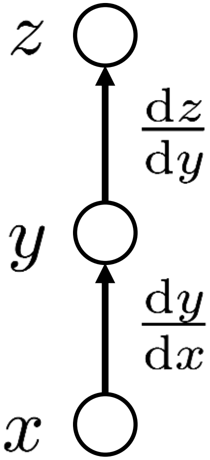
$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(1)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{hd}^{(1)}}$$

  - Compute the gradients for each variable analytically.
  - *What is the problem when doing this?*

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions


z
$\frac{\mathrm{d}z}{\mathrm{d}y}$
y
$\frac{\mathrm{d}y}{\mathrm{d}x}$
x

$$\Delta z = \frac{\mathrm{d}z}{\mathrm{d}y}\Delta y$$

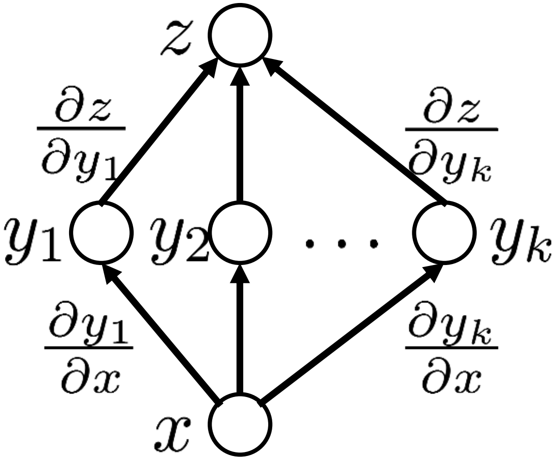
$$\Delta y = \frac{\mathrm{d}y}{\mathrm{d}x}\Delta x$$

$$\Delta z = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}\Delta x$$

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}$$

Excursion: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative


z
$\frac{\partial z}{\partial y_1}$
$\frac{\partial z}{\partial y_k}$
y_1
y_2
$\cdots$
y_k
$\frac{\partial y_1}{\partial x}$
$\frac{\partial y_k}{\partial x}$
x

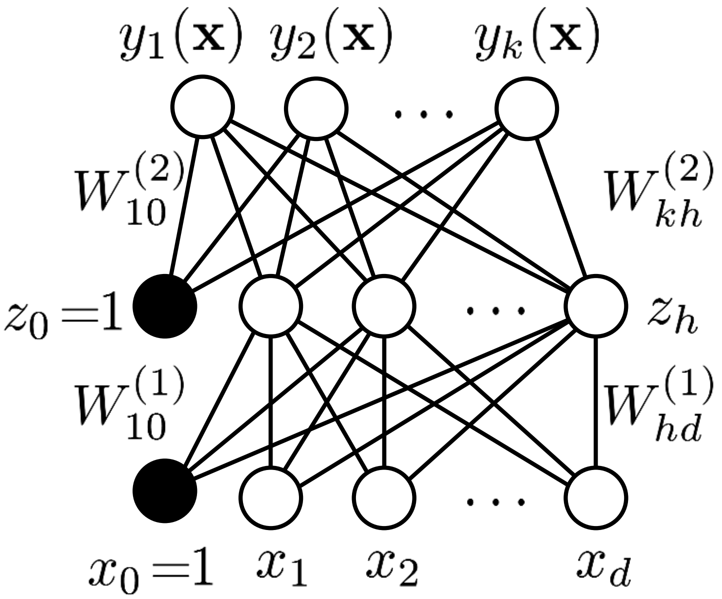
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \dots$$

$$= \sum_{i=1}^{k} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

$\Rightarrow$ Need to sum over all paths that lead to the target variable x.

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(1)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{hd}^{(1)}}$$

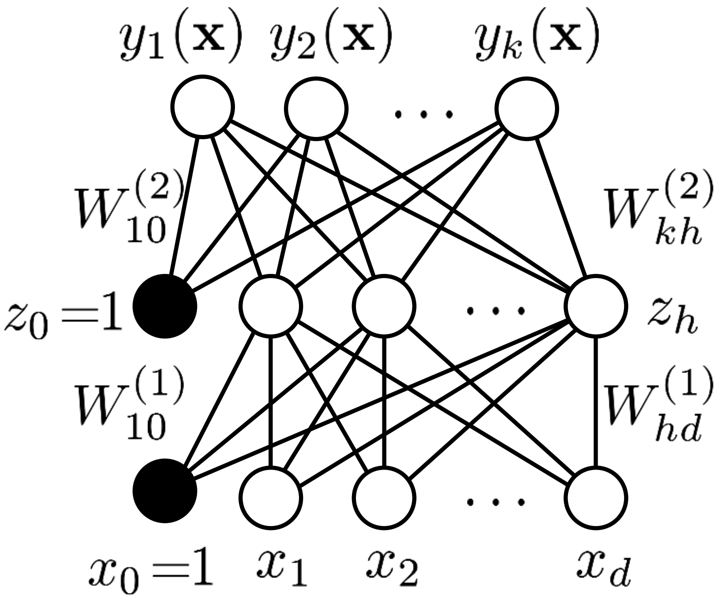
  - Compute the gradients for each variable analytically.
  - *What is the problem when doing this?*

⇒ With increasing depth, there will be exponentially many paths!

⇒ Infeasible to compute this way.

Obtaining the Gradients

- Approach 2: Numerical Differentiation

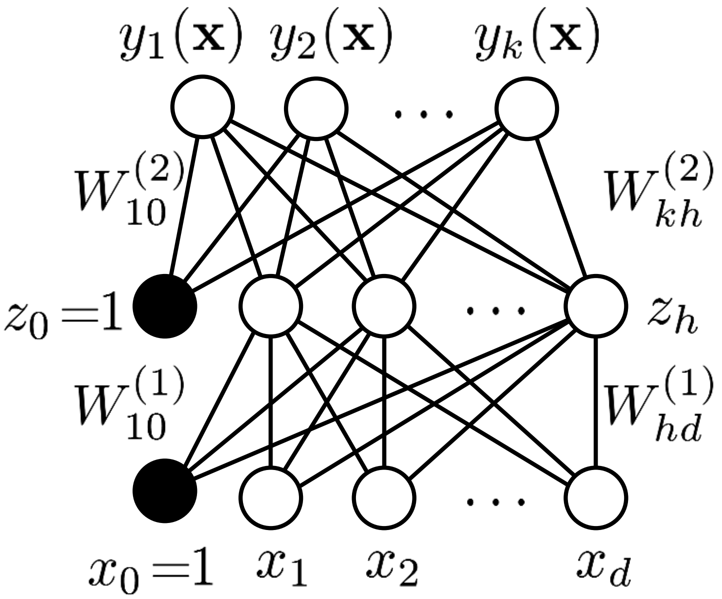


  - Given the current state $\mathbf{W}^{(\tau)}$, we can evaluate $E(\mathbf{W}^{(\tau)})$.
  - Idea: Make small changes to $\mathbf{W}^{(\tau)}$ and accept those that improve $E(\mathbf{W}^{(\tau)})$.

$\Rightarrow$ Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

Obtaining the Gradients

- Approach 3: Incremental Analytical Differentiation



$$\begin{tikzcd}
\frac{\partial E(\mathbf{W})}{\partial y_j} \arrow[dd] \arrow[dr] & \\
& \frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(2)}} \\
\frac{\partial E(\mathbf{W})}{\partial z_i} \arrow[dd] \arrow[dr] & \\
& \frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(1)}} \\
\frac{\partial E(\mathbf{W})}{\partial x_i} &
\end{tikzcd}$$

  - Idea: Compute the gradients layer by layer.
  - Each layer below builds upon the results of the layer above.

$\Rightarrow$ The gradient is propagated backwards through the layers.

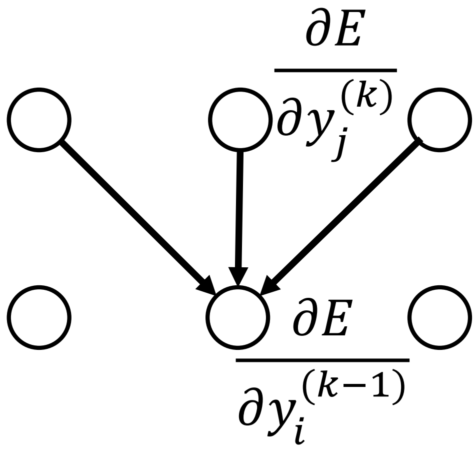
$\Rightarrow$ Backpropagation algorithm

Backpropagation Algorithm

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

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

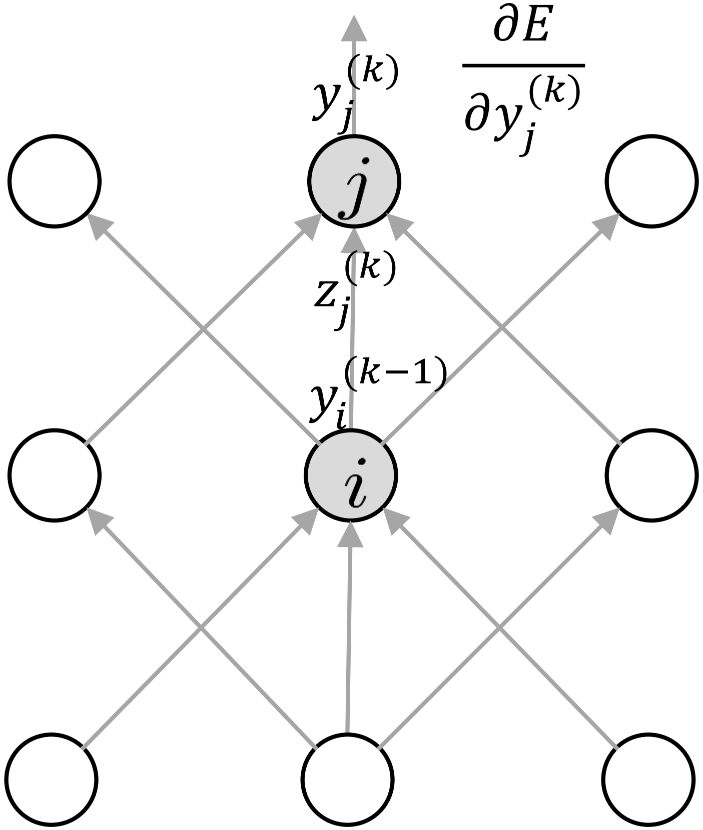
$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$


$\frac{\partial E}{\partial y_j^{(k)}}$
$\frac{\partial E}{\partial y_i^{(k-1)}}$

$$\frac{\partial E}{\partial y_j^{(k)}} \longrightarrow \frac{\partial E}{\partial w_{ji}^{(k-1)}}$$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\color{blue}{\frac{\partial E}{\partial z_j^{(k)}}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

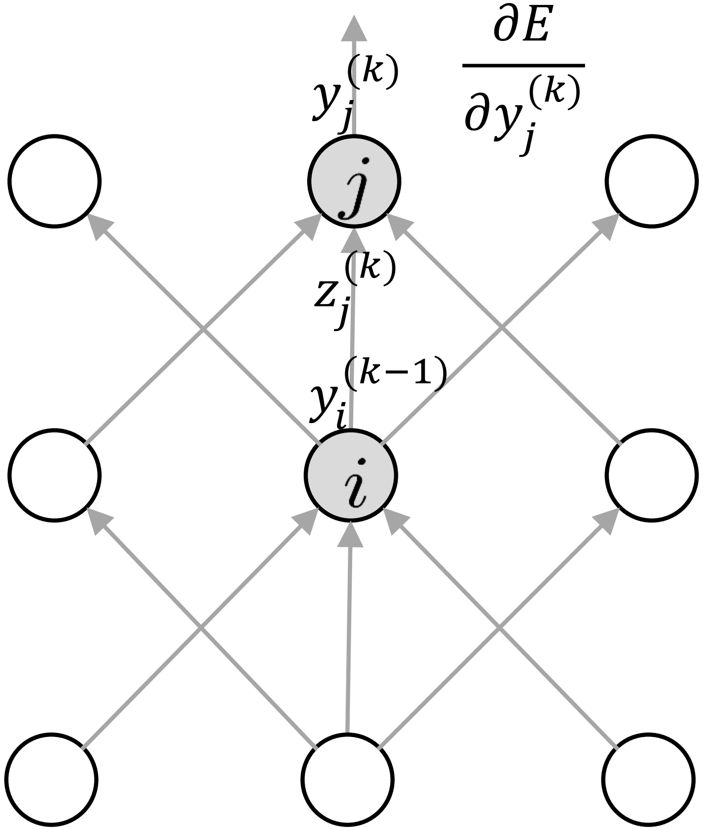
- Notation
  - $y_j^{(k)}$ Output of layer k
  - $z_j^{(k)}$ Input of layer k

Connections: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

$y_j^{(k)} = g\left(z_j^{(k)}\right)$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

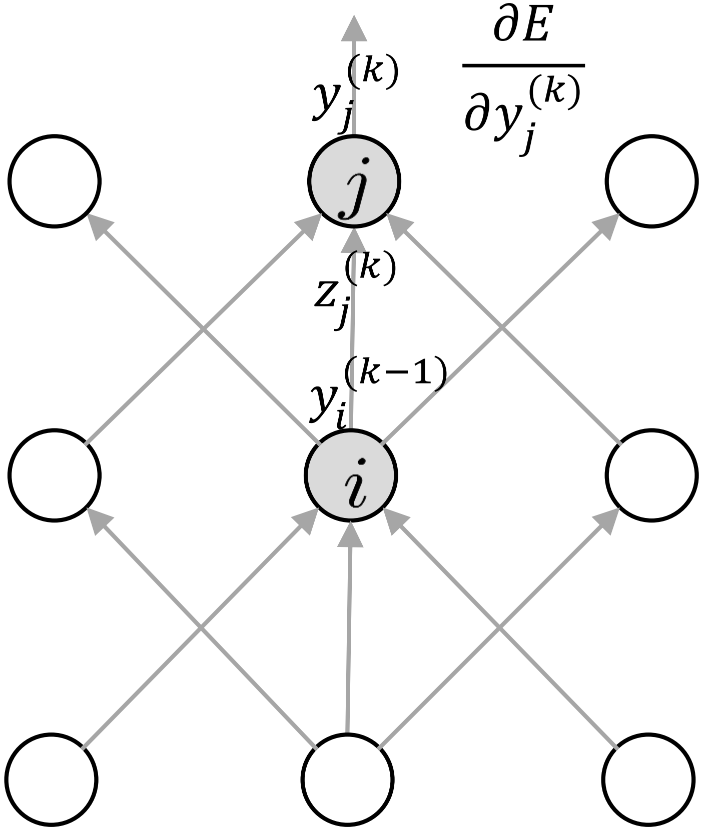
- Notation
  - $y_j^{(k)}$ Output of layer k Connections: $z_j^{(k)} = \sum w_{ji}^{(k-1)} y_i^{(k-1)}$
  - $z_j^{(k)}$ Input of layer k

$$\boxed{\frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} = w_{ji}^{(k-1)}}$$

Slide adapted from Geoff Hinton

Machine Learning Winter '19

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

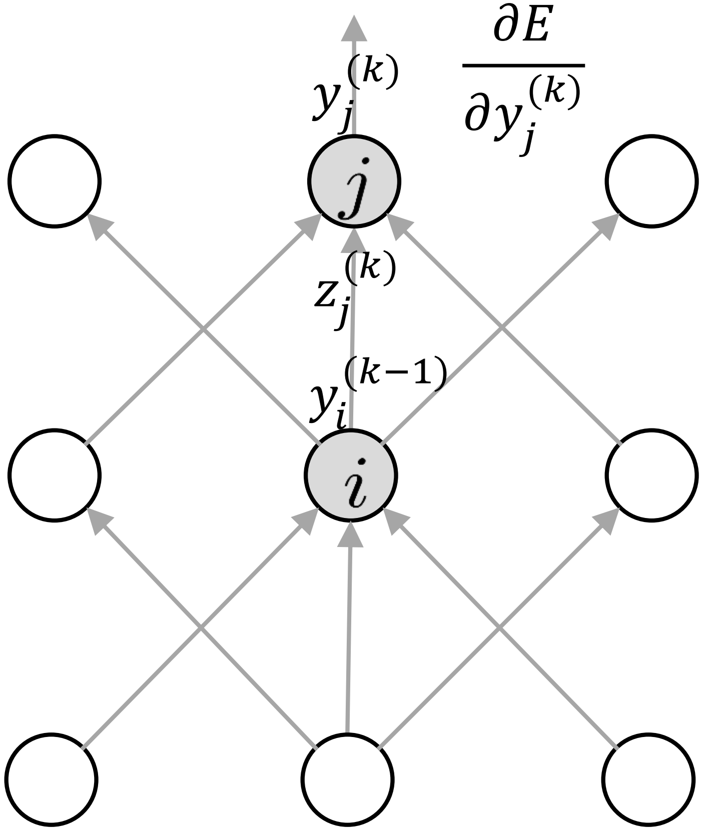
- Notation
  - $y_j^{(k)}$ Output of layer k
  - $z_j^{(k)}$ Input of layer k

Connections: $z_j^{(k)} = \sum w_{ji}^{(k-1)} y_i^{(k-1)}$

$$\boxed{\frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} = y_i^{(k-1)}}$$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

- Efficient propagation scheme
  - $y_i^{(k-1)}$ is already known from forward pass! (Dynamic Programming)

$\Rightarrow$ Propagate back the gradient from layer k and multiply with $y_i^{(k-1)}$.

Slide adapted from Geoff Hinton

Summary: MLP Backpropagation

- Forward Pass

```
$\mathbf{y}^{(0)} = \mathbf{x}$
for $k = 1, ..., l$ 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

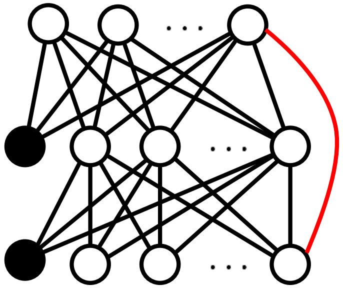
```
$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}} L(\mathbf{t}, \mathbf{y}) + \lambda \frac{\partial}{\partial \mathbf{y}} \Omega$
for $k = l, l\text{-}1, ..., 1$ do
    $\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{z}^{(k)}} = \mathbf{h} \odot g'(\mathbf{y}^{(k)})$
    $\frac{\partial E}{\partial \mathbf{W}^{(k)}} = \mathbf{h}\mathbf{y}^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial \mathbf{W}^{(k)}}$
    $\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}^{(k-1)}} = \mathbf{W}^{(k)\top}\mathbf{h}$
endfor
```

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

Analysis: Backpropagation

- Backpropagation is the key to making deep NNs tractable
  - However...
- The Backprop algorithm given here is specific to MLPs
  - It does not work with more complex architectures, e.g. skip connections or recurrent networks!
  - Whenever a new connection function induces a different functional form of the chain rule, you have to derive a new Backprop algorithm for it.

  ⇒ Tedious...



- Let's analyze Backprop in more detail
  - This will lead us to a more flexible algorithm formulation

Image source: colourbox.de, thinkstock

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

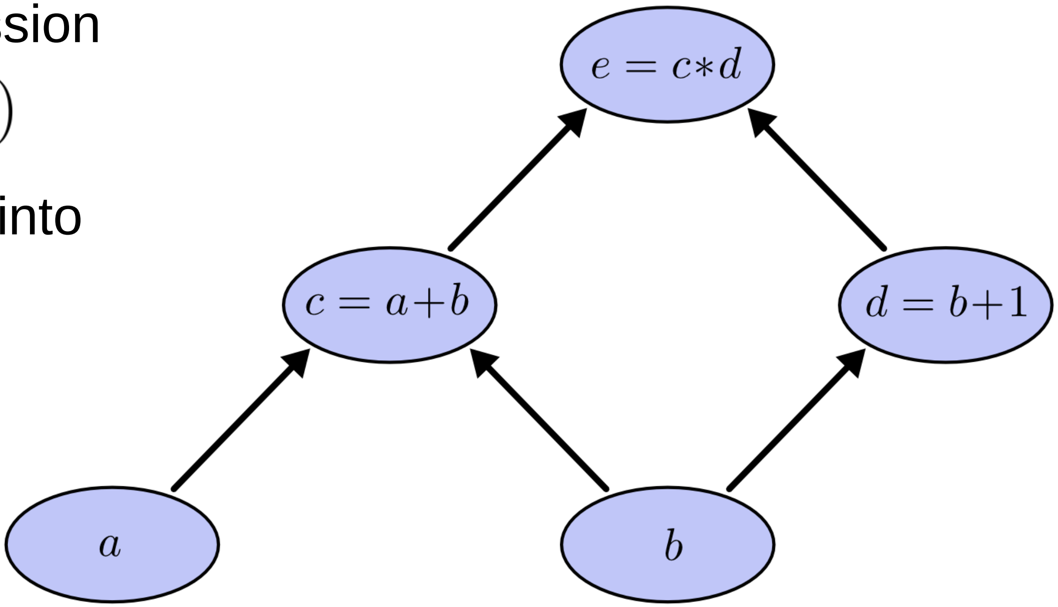
Computational Graphs

- We can think of mathematical expressions as graphs
  - E.g., consider the expression

$$e = (a+b)*(b+1)$$

  - We can decompose this into the operations

$$\begin{aligned} c &= a+b \\ d &= b+1 \\ e &= c*d \end{aligned}$$



and visualize this as a computational graph.

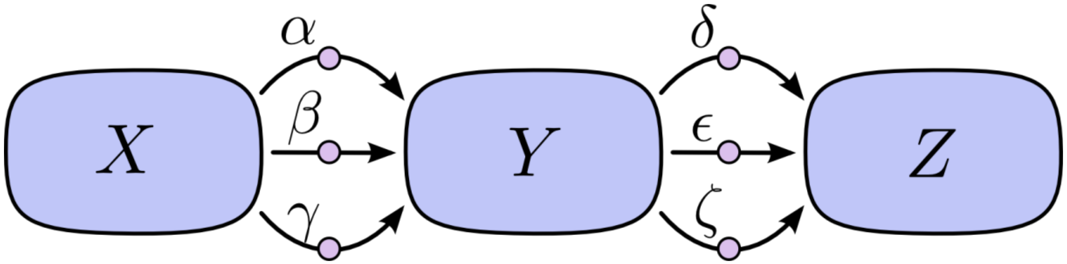
- Evaluating partial derivatives $\frac{\partial Y}{\partial X}$ in such a graph
  - General rule: sum over all possible paths from Y to X and multiply the derivatives on each edge of the path together.

Slide inspired by Christopher Olah

Image source: Christopher Olah, colah.github.io

Factoring Paths

- Problem: Combinatorial explosion
  - Example:



  - There are 3 paths from X to Y and 3 more from Y to Z.
  - If we want to compute $\frac{\partial Z}{\partial X}$, we need to sum over 3×3 paths:

$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

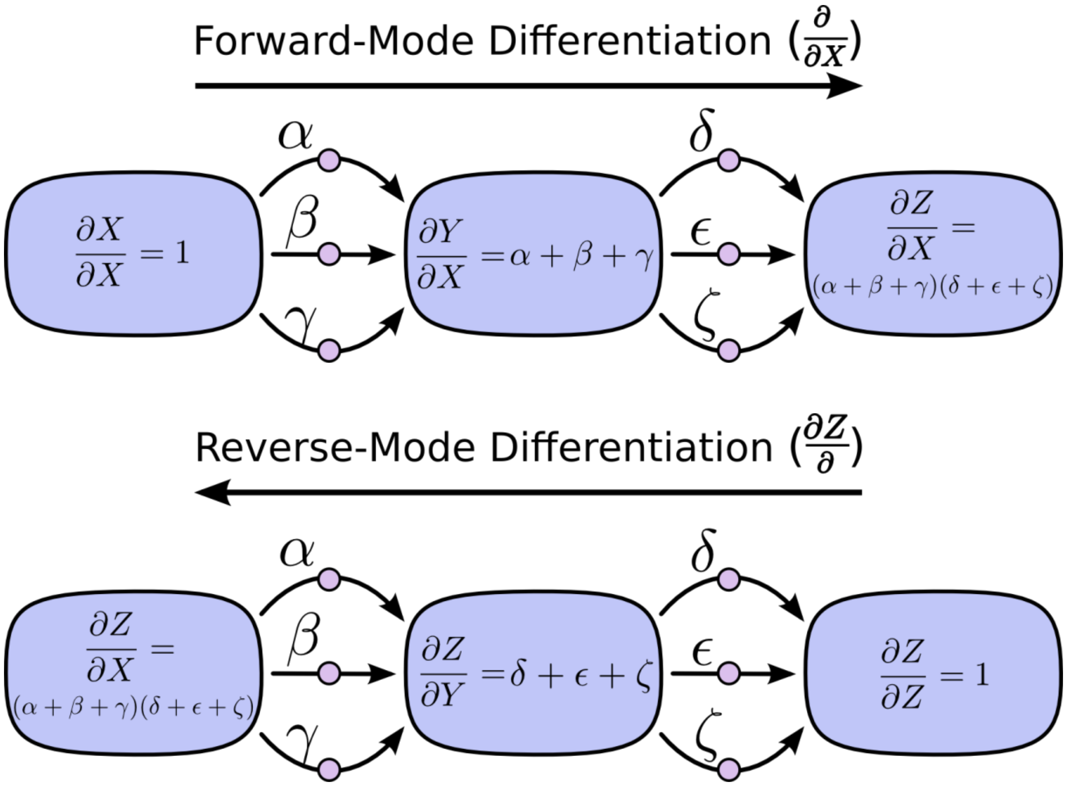
  - Instead of naively summing over paths, it's better to factor them

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma) * (\delta + \epsilon + \zeta)$$

Slide inspired by Christopher Olah

Image source: Christopher Olah, colah.github.io

Efficient Factored Algorithms


Forward-Mode Differentiation ($\frac{\partial}{\partial X}$)
α
β
γ
δ
ϵ
ζ
$\frac{\partial X}{\partial X} = 1$
$\frac{\partial Y}{\partial X} = \alpha + \beta + \gamma$
$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$
Reverse-Mode Differentiation ($\frac{\partial Z}{\partial}$)
$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$
$\frac{\partial Z}{\partial Y} = \delta + \epsilon + \zeta$
$\frac{\partial Z}{\partial Z} = 1$

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

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

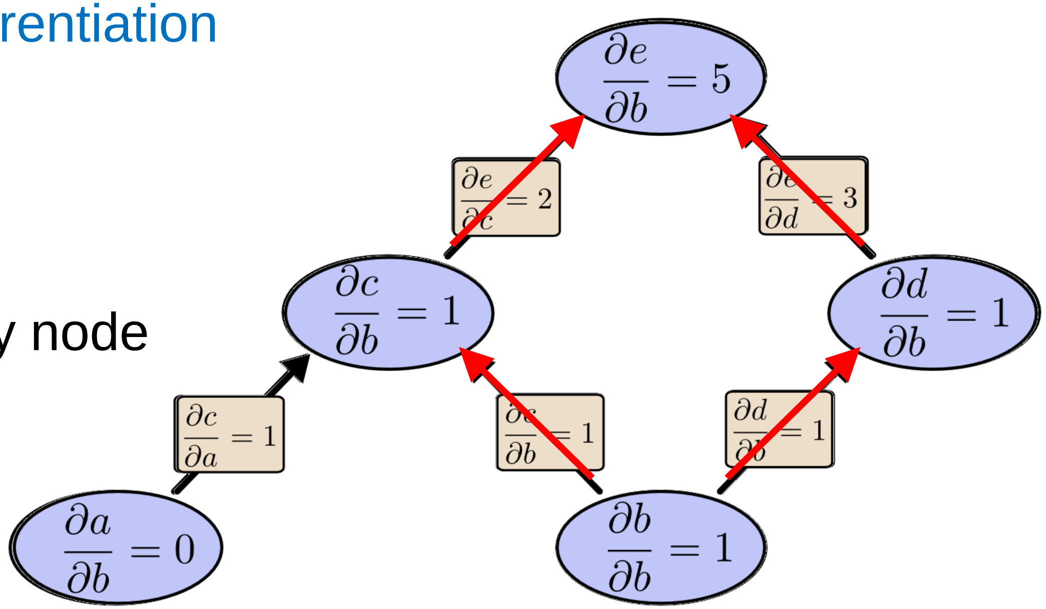
- Efficient algorithms for computing the sum
  - Instead of summing over all of the paths explicitly, compute the sum more efficiently by merging paths back together at every node.

Slide inspired by Christopher Olah

Image source: Christopher Olah, colah.github.io

Why Do We Care?

- Let’s consider the example again
  - Using forward-mode differentiation from b up...
  - Runtime: $\mathcal{O}(\#\text{edges})$
  - Result: derivative of every node with respect to b.



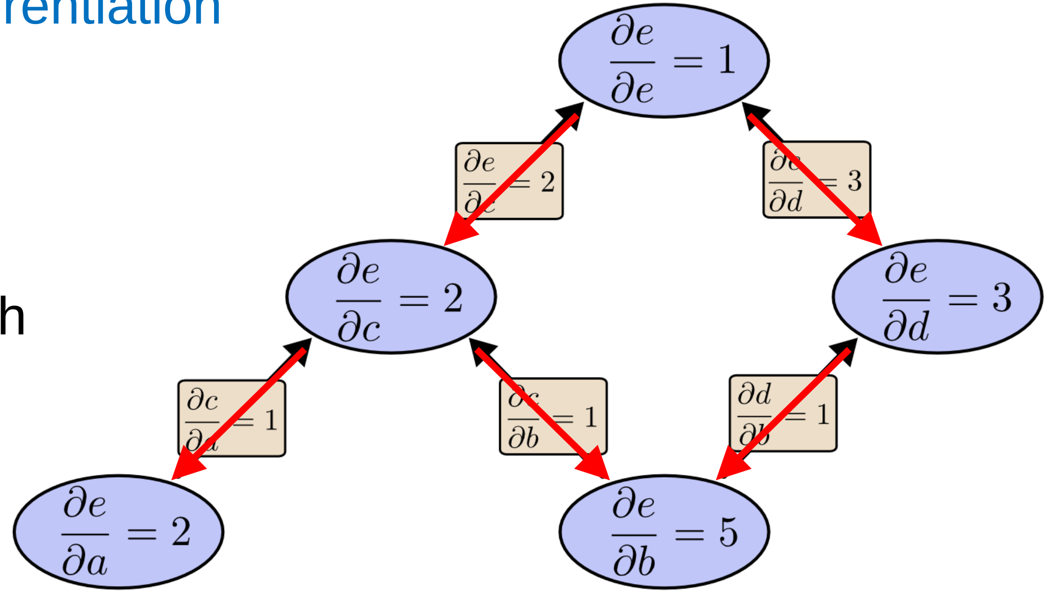
$\frac{\partial e}{\partial b} = 5$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial a}{\partial b} = 0$
$\frac{\partial b}{\partial b} = 1$

Image source: Christopher Olah, colah.github.io

Slide inspired by Christopher Olah

Why Do We Care?

- Let's consider the example again
  - Using reverse-mode differentiation from e down...
  - Runtime: $\mathcal{O}(\#\text{edges})$
  - Result: derivative of e with respect to every node.



$\frac{\partial e}{\partial e} = 1$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial e}{\partial a} = 2$
$\frac{\partial e}{\partial b} = 5$

$\Rightarrow$ *This is what we want to compute in Backpropagation!*

- Forward differentiation needs one pass per node. With backward differentiation we can compute all derivatives in one single pass.

$\Rightarrow$ Speed-up in $\mathcal{O}(\#\text{inputs})$ compared to forward differentiation!

Slide inspired by Christopher Olah

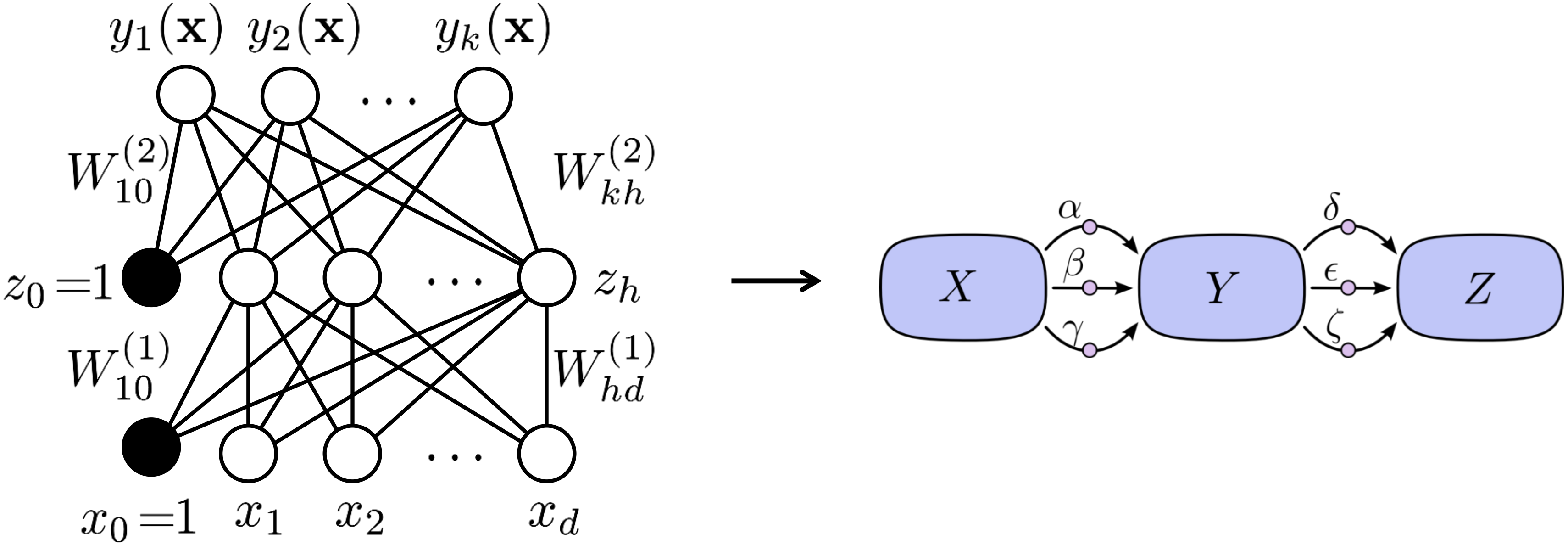
Image source: Christopher Olah, colah.github.io

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Obtaining the Gradients

- Approach 4: Automatic Differentiation



  - 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

Modular Implementation

- Solution in many current Deep Learning libraries
  - Provide a limited form of automatic differentiation
  - Restricted to "programs" composed of "modules" with a predefined set of operations.

- Each module is defined by two main functions
  1. Computing the outputs $\mathbf{y}$ of the module given its inputs $\mathbf{x}$

     $$\mathbf{y} = \text{module}.\mathbf{fprop}(\mathbf{x})$$

     where $\mathbf{x}$, $\mathbf{y}$, and intermediate results are stored in the module.

  2. Computing the gradient $\partial E / \partial \mathbf{x}$ of a scalar cost w.r.t. the inputs $\mathbf{x}$ given the gradient $\partial E / \partial \mathbf{y}$ w.r.t. the outputs $\mathbf{y}$

     $$\frac{\partial E}{\partial \mathbf{x}} = \text{module}.\mathbf{bprop}\left(\frac{\partial E}{\partial \mathbf{y}}\right)$$

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Sidenote: Implementing Softmax Correctly

- Softmax output
  - De-facto standard for multi-class outputs

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

- 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 nominator and log-sum-exp in the denominator.
  - Trick 2: Softmax has the property that for a fixed vector $\mathbf{b}$

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

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

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Gradient Descent

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

- 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?

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.

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

Stochastic 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 be 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

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

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues

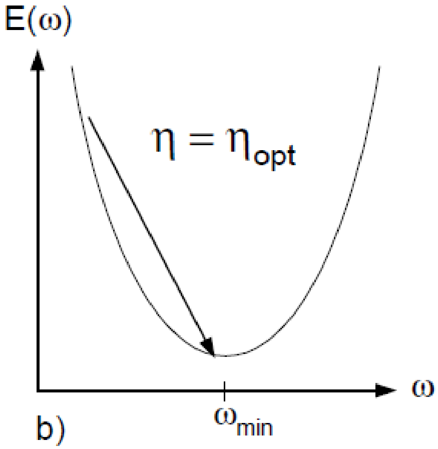
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers

- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

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

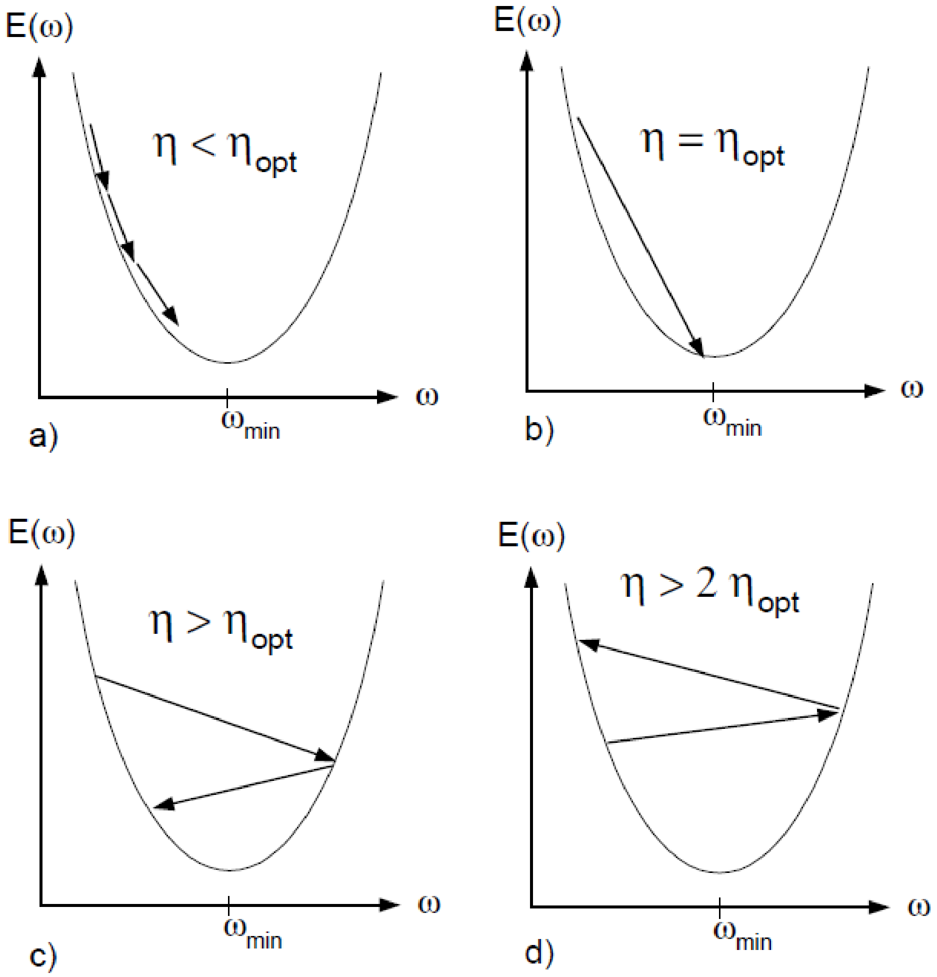


Image source: Yann LeCun et al., Efficient BackProp (1998)

Learning Rate vs. Training Error

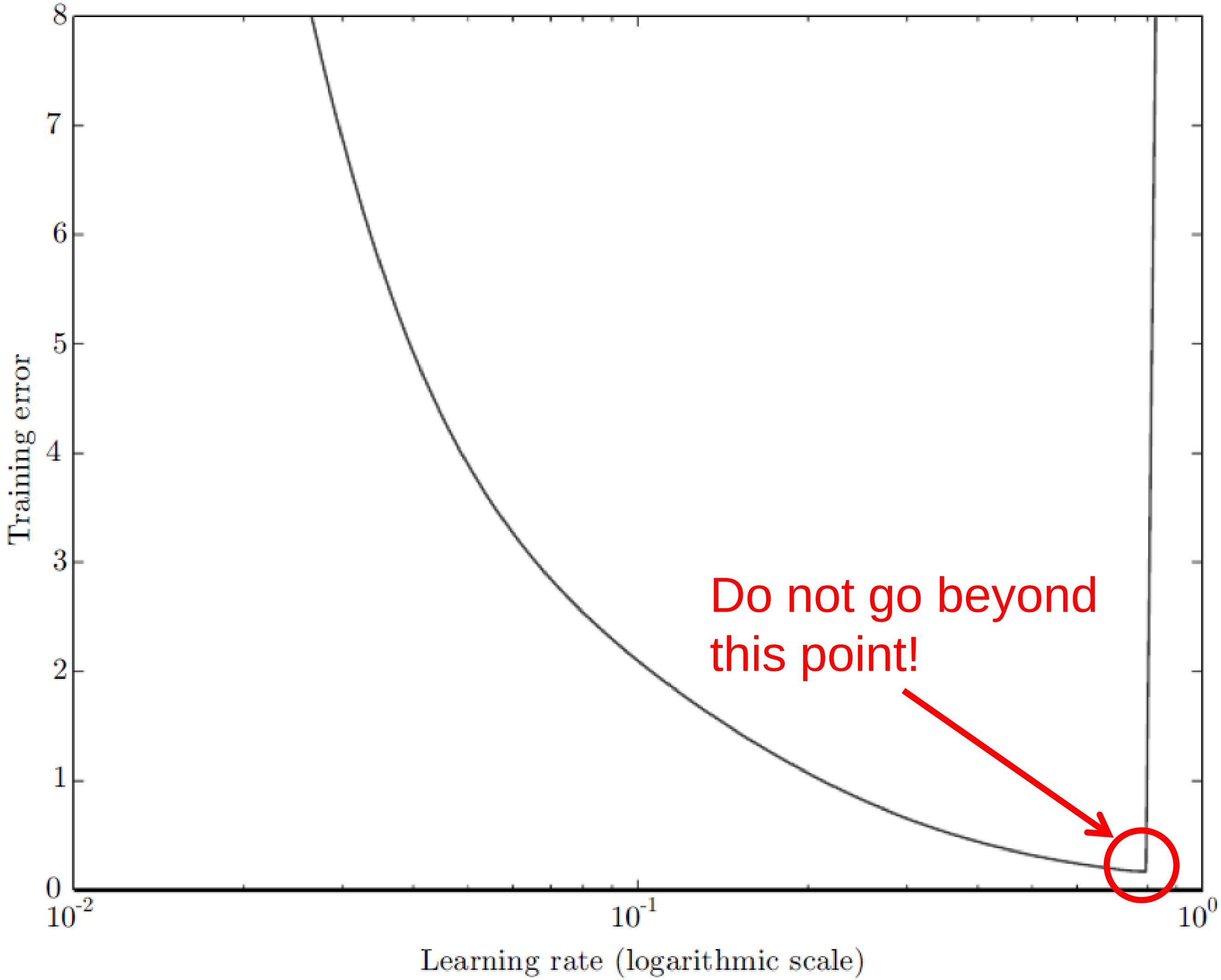


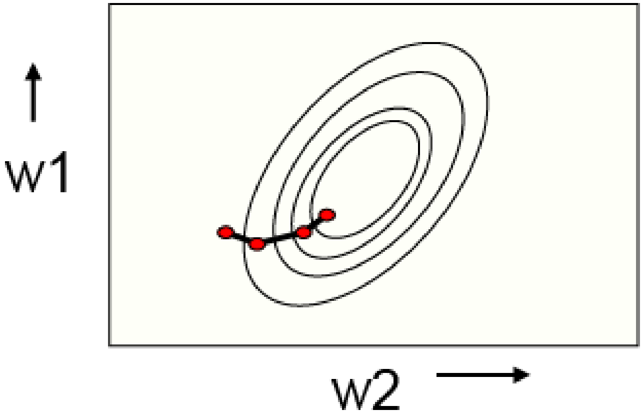
Image source: Goodfellow & Bengio book

Topics of This Lecture

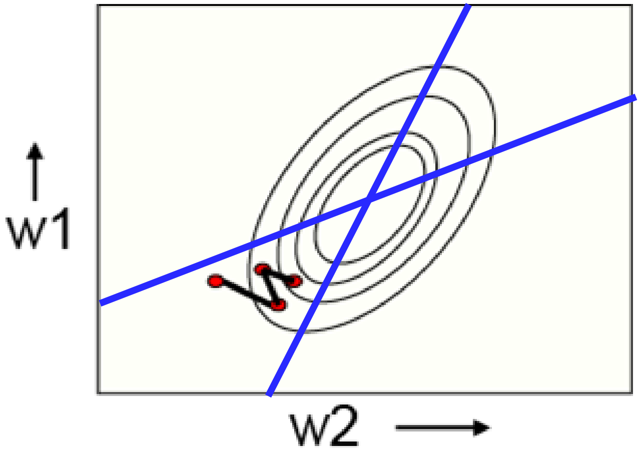
- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

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.

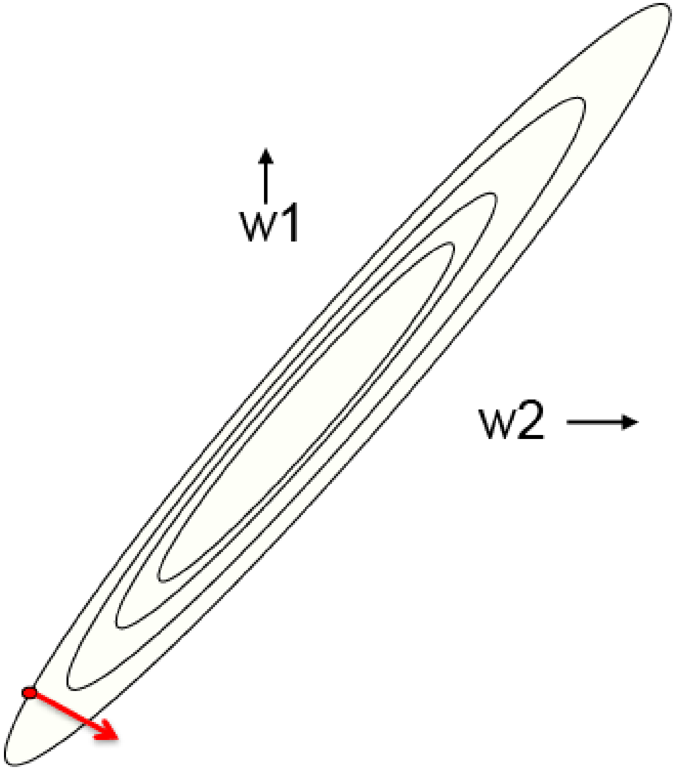


Slide adapted from Geoff Hinton

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!



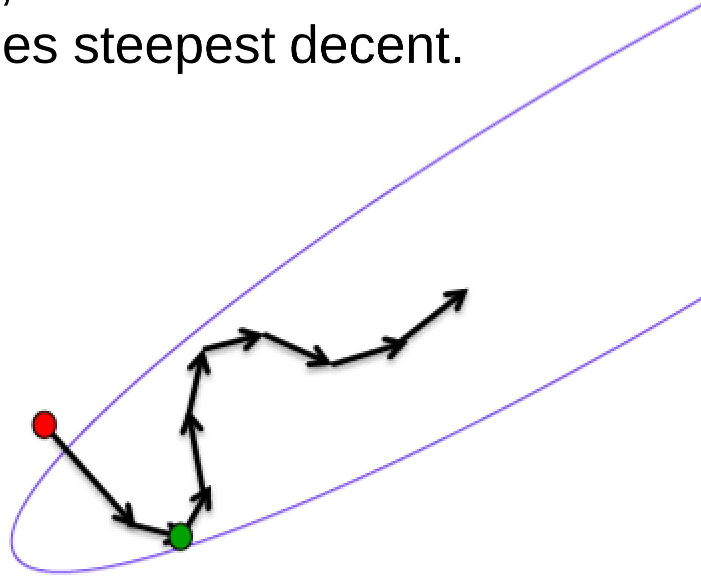
This is just the opposite of what we want!

Slide adapted from Geoff Hinton

Image source: Geoff Hinton

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.



Slide credit: Geoff Hinton

Image source: Geoff Hinton

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

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

Slide credit: Geoff Hinton

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 $\alpha \;= 0.99$).

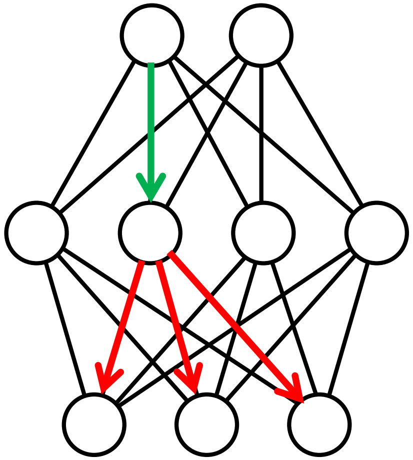
$\Rightarrow$ This allows us to learn at a rate that would cause divergent oscillations without the momentum.

Slide credit: Geoff Hinton

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.

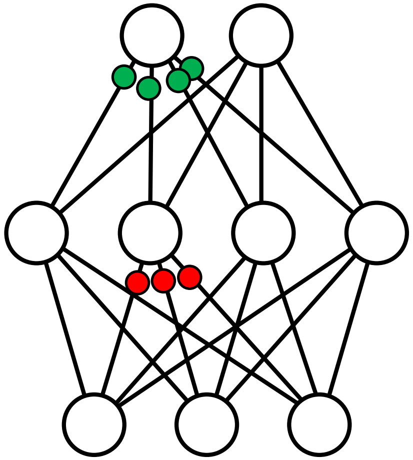


Slide adapted from Geoff Hinton

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)

Slide adapted from Geoff Hinton

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

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

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

Slide adapted from Geoff Hinton

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

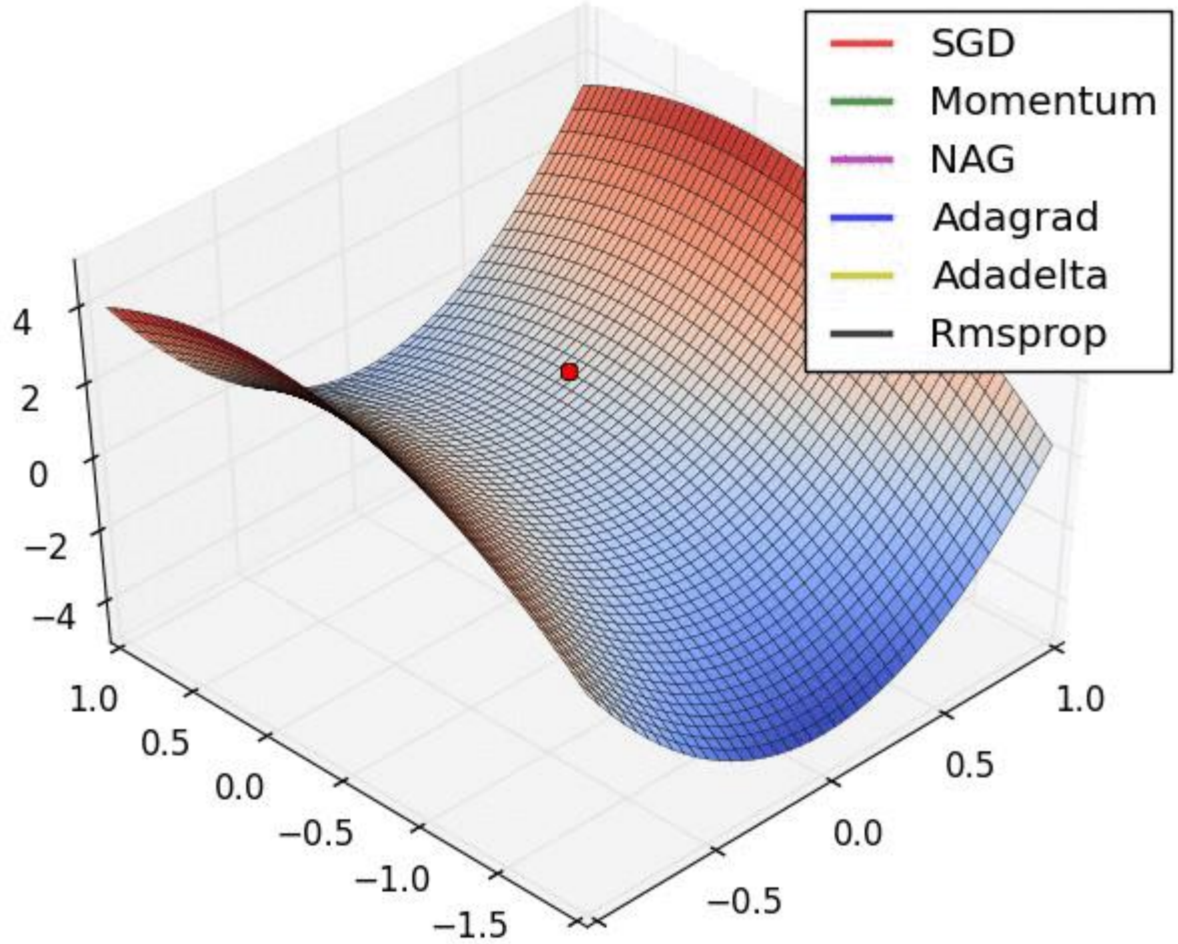


Image source: Alec Radford, http://imgur.com/a/Hqolp

Behavior around a Saddle Point

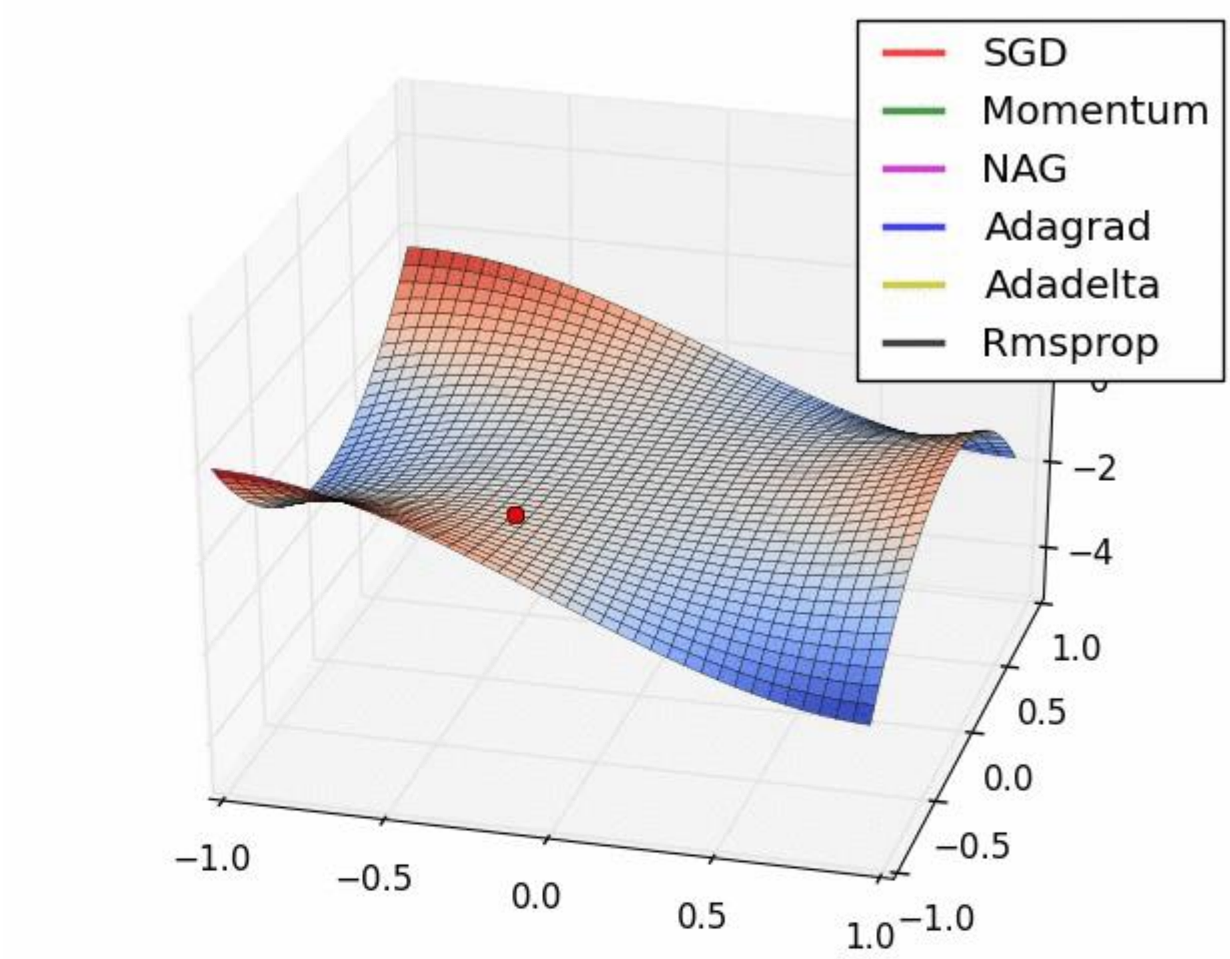


Image source: Alec Radford, http://imgur.com/a/Hqolp

Visualization of Convergence Behavior

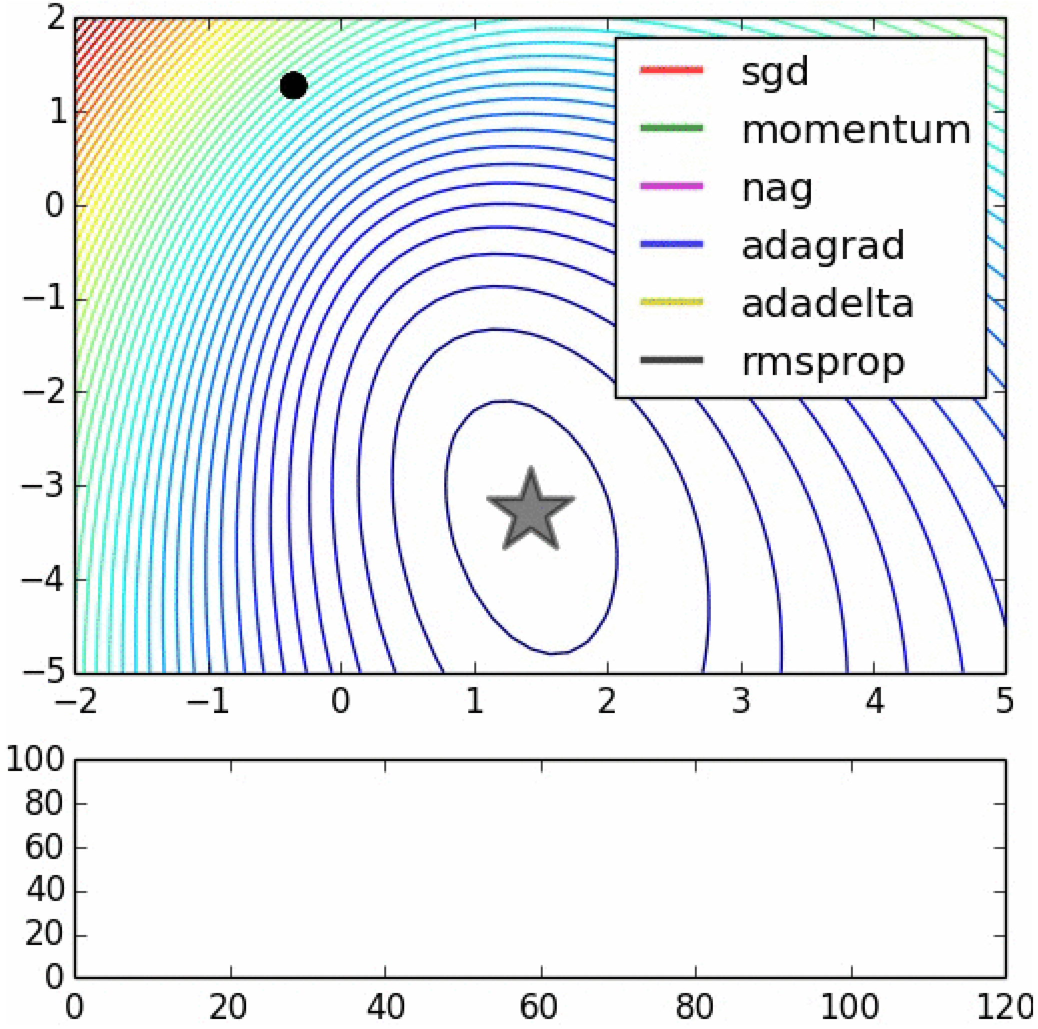
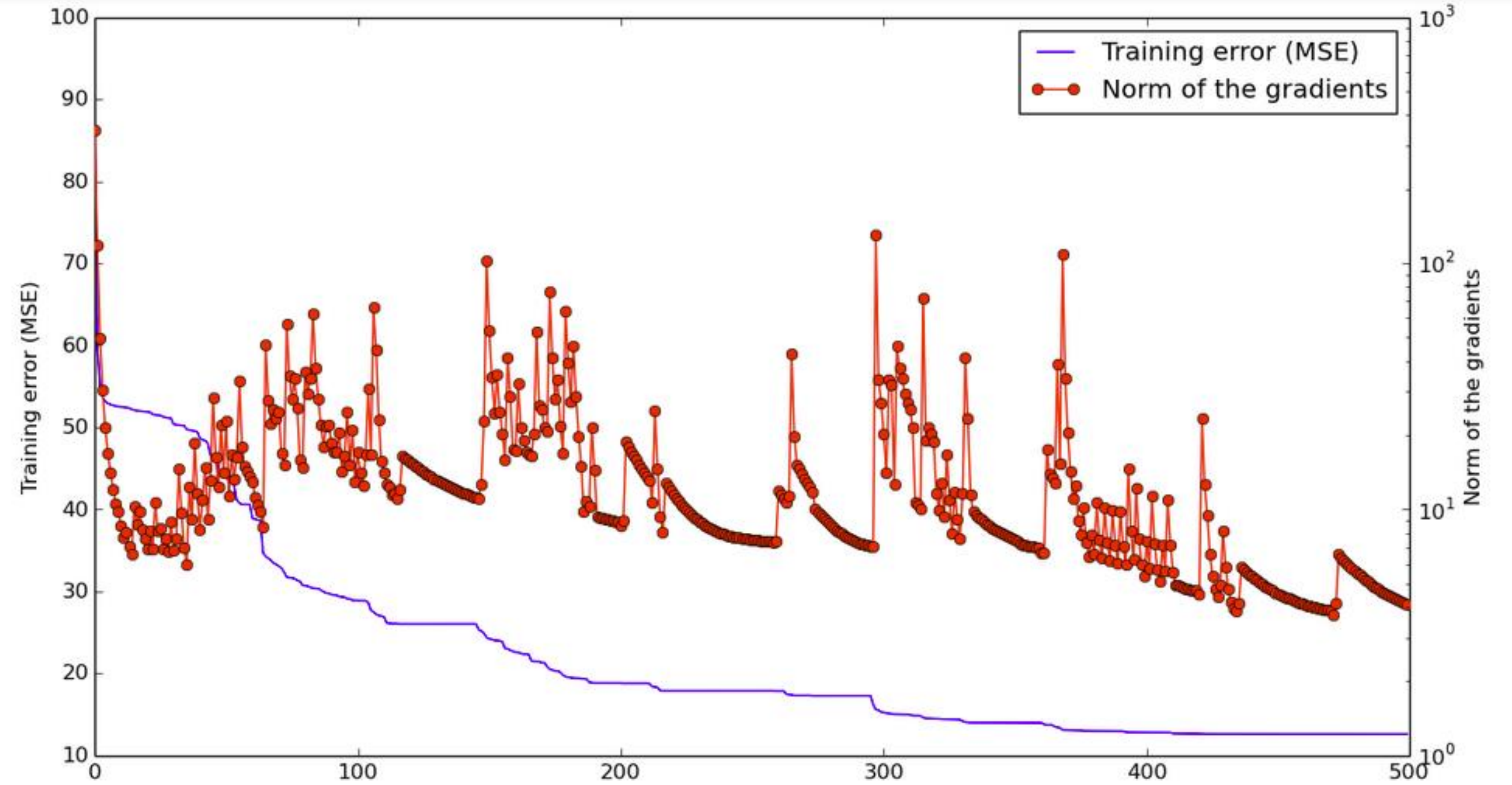


Image source: Aelc Radford, http://imgur.com/SmDARzn

Trick: Patience

- Saddle points dominate in high-dimensional spaces!

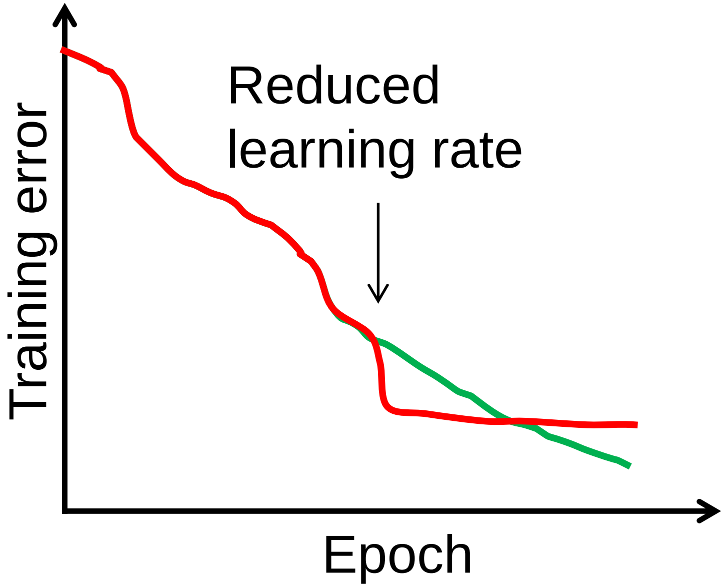


⇒ Learning often doesn’t get stuck, you just may have to wait...

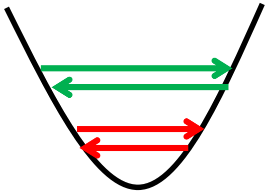
Image source: Yoshua Bengio

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/impossible after that.

Slide adapted from Geoff Hinton

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 early 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…)*

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

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.

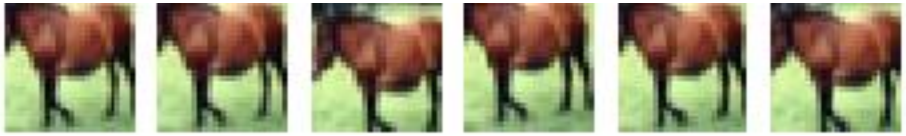
  - A large relative error indicates that an input has not been learned by the network yet, so it contains a lot of information.

  ⇒ It can make sense to present such inputs more frequently.
    - But: be careful, this can be disastrous when the data are outliers.

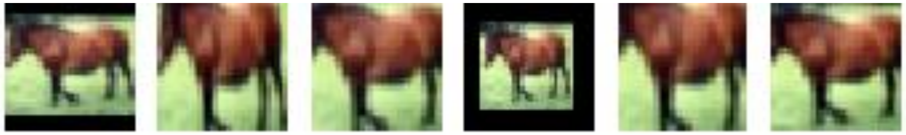
- Practical advice
  - When working with stochastic gradient descent or minibatches, make use of shuffling.

Data Augmentation

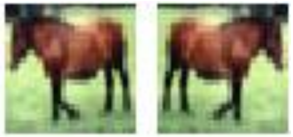
- Idea
  - Augment original data with synthetic variations to reduce overfitting
- Example augmentations for images
  - Cropping



  - Zooming



  - Flipping



  - Color PCA

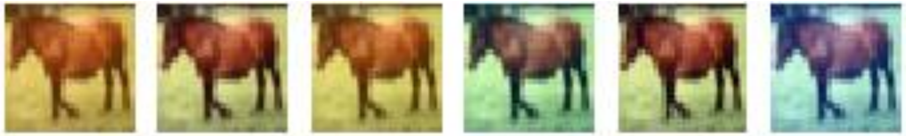
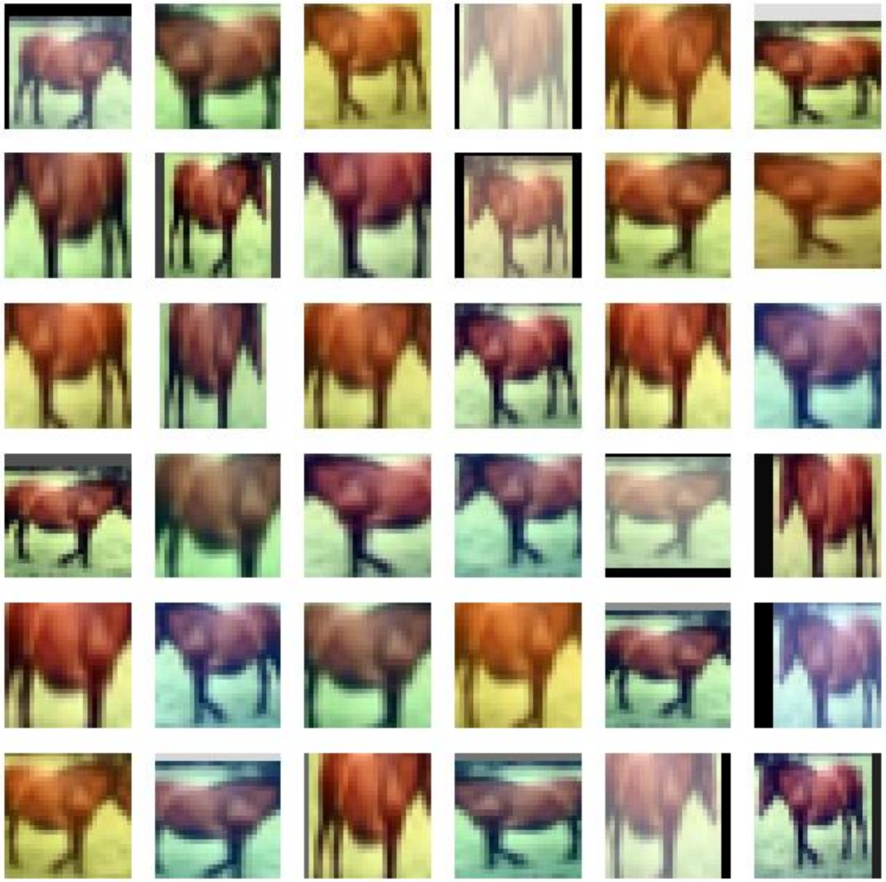


Image source: Lucas Beyer

Data Augmentation

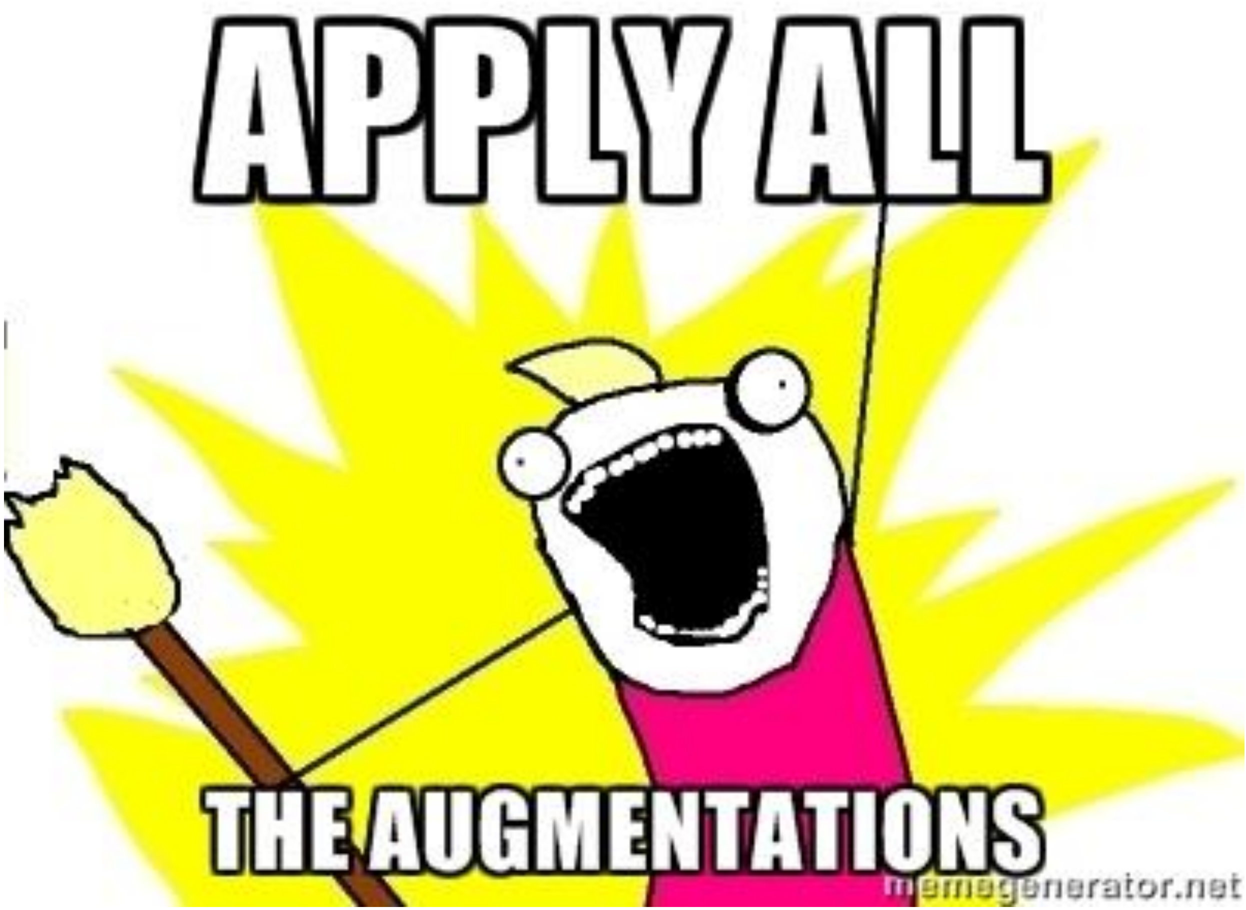
- Effect
  - Much larger training set
  - Robustness against expected variations
- During testing
  - When cropping was used during training, need to again apply crops to get same image size.
  - Beneficial to also apply flipping during test.
  - Applying several ColorPCA variations can bring another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



Augmented training data (from one original image)

Image source: Lucas Beyer

Practical Advice



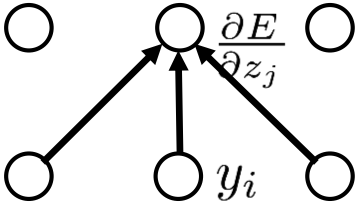
Normalization

- 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

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



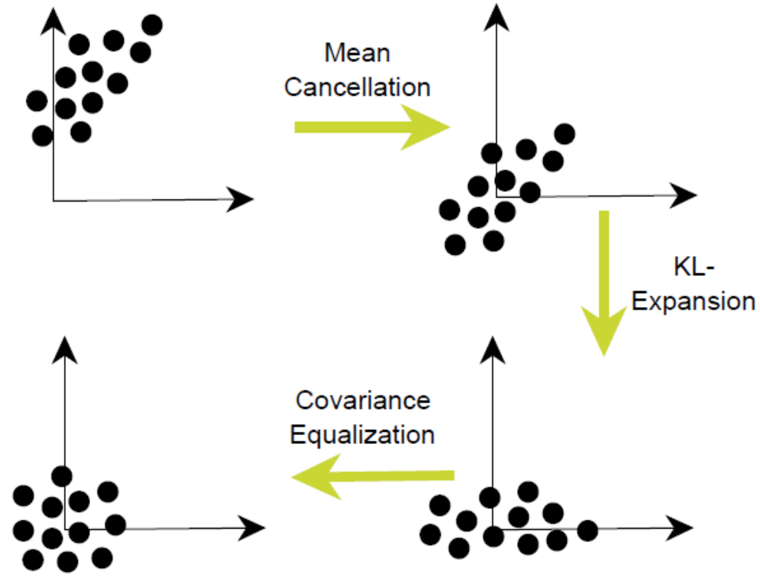
  - When all of the components of the input vector y_i are positive, all of the updates of weights that feed into a node will be of the same sign.

$\Rightarrow$ 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.



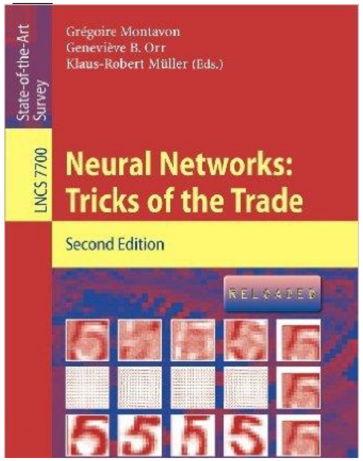
- Advisable normalization steps (for MLPs only, not for CNNs)
  - Normalize all inputs that an input unit sees to zero-mean, unit covariance.
  - If possible, try to decorrelate them using PCA (also known as Karhunen-Loeve expansion).

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

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.