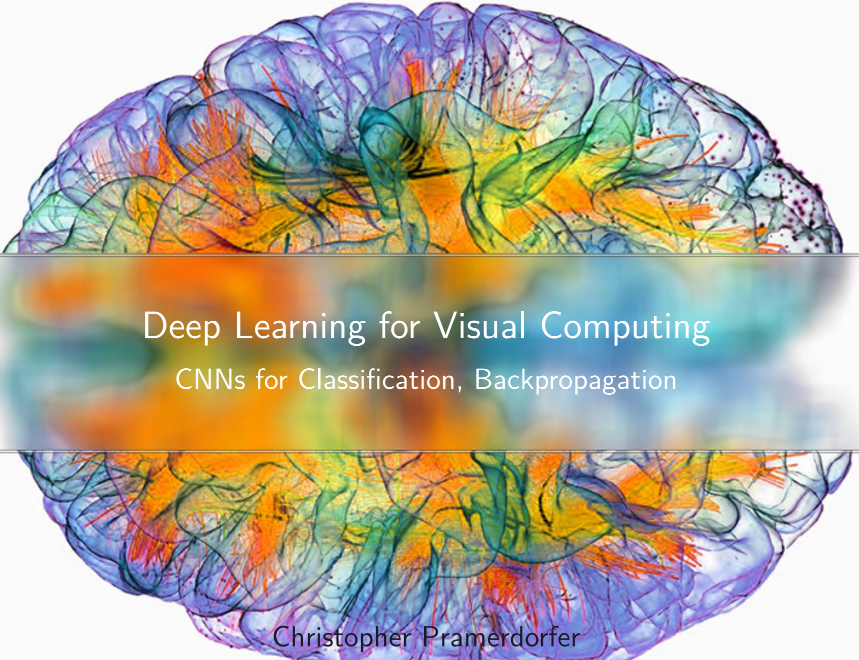


Deep Learning for Visual Computing

CNNs for Classification, Backpropagation

Christopher Pramerdorfer
Computer Vision Lab, TU Wien

Image by Neuroscape Lab

Topics

CNNs for classification

- General considerations
- Modern architectures

Computing $\nabla L(\boldsymbol{\theta})$ with backpropagation

CNNs for Classification

Three main layer types (previous lecture)

- Convolutional (conv), pooling (pool), fully-connected (fc)

Architectures always include two stages

- Frontend for feature extraction (conv and pool layers)
- Backend for classification (fc layers)

CNNs for Classification
General Considerations

Many hyperparameters

- Layer composition and individual layer properties
- In addition to learning rate, weight decay, momentum

Unable to test sufficient number of combinations

- Testing one combination can take days

Following considerations help with architecture design

Input image size

- Affects runtime quadratically
- Does not affect number of conv layer weights

Minimum input size depends on task and dataset

- Sizes larger than 224 pixels are uncommon
- Start small and see if increasing size helps

Input size should be divisible by 2 often

- Spatial resolution reduced by 2 multiple times
- Avoids issues with non-integral output resolution

Accomplished by scaling/cropping input images

Last layer is always fc layer with T neurons (T classes)

- Want to predict vector $\mathbf{w} \in \mathbb{R}^T$ of class scores

CNNs thus always end with a linear classifier

CNNs for Classification

General Considerations – Input Size Reduction

Spatial resolution (width W and height H) reduced to < 10

- Using multiple pooling or conv layers with stride
- Common final sizes : 1×1, 3×3, 5×5, 7×7

Motivation

- Reduce computational complexity
- Reduce dimensionality of final feature vector
- Increase final receptive field size
- Gain robustness to small translations in image

Reduction layers distributed evenly in network

Number of layers depends on input image size

Common reduction layers (both halve W and H)

- Max-pooling with stride 2 and extent 2×2 or 3×3
- Convolution with stride 2

Recall that depth equals number of layers with parameters

- Mainly determined by number of conv layers

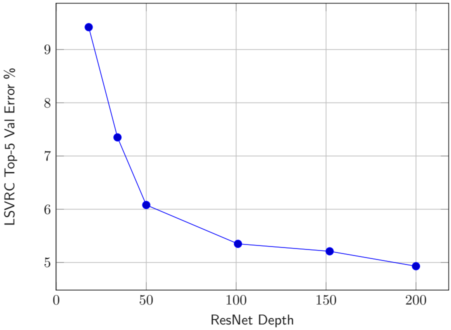
Suitable number depends on overall architecture and task

- Different tasks require different levels of abstraction
- Hundreds of layers possible (few parameters)

CNNs for Classification
General Considerations – Network Depth

At least 6 conv layers required in most cases

Diminishing returns in terms of depth vs. performance



Distributed evenly in network

- One or several conv layers $\Rightarrow$ pooling layer $\Rightarrow$ repeat

Always use stride 1 and zero-padding to preserve W and H

- Or stride 2 to replace pooling layers

Always use ReLU activation function

- Faster gradient descent convergence

Prefer more layers with $c = 3$ to fewer with $c > 3$

- Fewer parameters, more expressive features (non-linearities)

Example (consistent depth D, 7×7 receptive field)

- One layer with $c = 7:\ 7 \cdot 7 \cdot D \cdot D = 49D^2$ weights
- Three layers with $c = 3:\ 3 \cdot 3 \cdot D \cdot D \cdot 3 = 27D^2$ weights

Exception : Initial conv layer with $c = 7$ and stride 2

- Common with large input sizes (see below)
- To reduce computational and memory requirements

Number of feature maps D should increase with network depth

- Initial number of 64 is common
- Final number often 256 or 512

New architectures usually designed for LSVRC

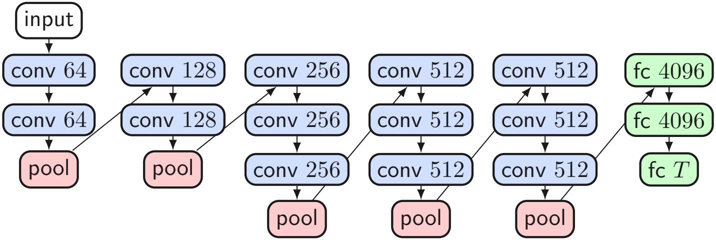
- Large dataset : 1000 classes, 1.4m samples
- Large color images : usually 224 by 224 pixels

Image from umich.edu

CNNs for Classification
Modern Architectures – VGGNet (VGG-16 for LSVRC)

Homogeneous architecture

- 3×3 convolutions, 2×2 max-pooling
- Depth usually doubled after pooling



Good template for conventional frontends

Changes based on dataset properties

- Number of pooling layers
- Initial number of feature maps
- Number of conv layers per pooling layer

Many parameters due to complex classifier stage

- MLP with two large hidden layers
- VGG-16 for LSVRC : about 140m parameters in total

Architecture composed of Inception modules

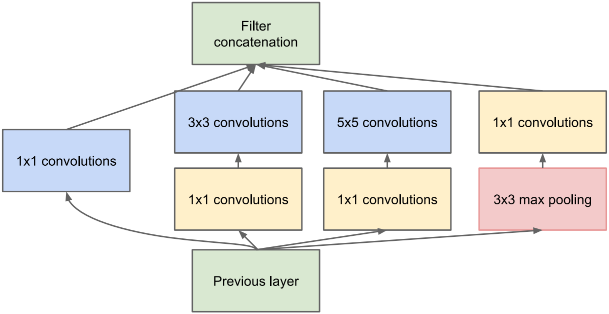


Image from [1]

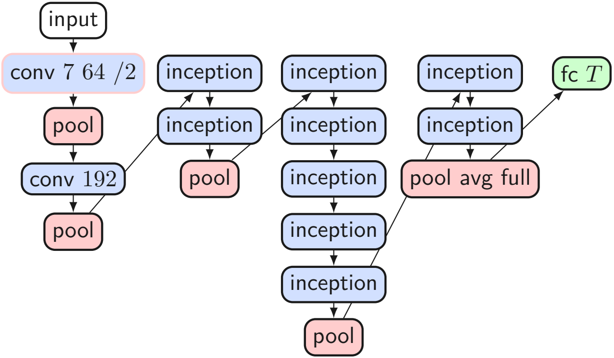
Extract features at four different scales

- Four columns in Inception module
- Resulting in $D^1 + D^2 + D^3 + D^4$ feature maps

$1 \times 1 \times D_l$ convolutions with $D_l \ll D_{l-1}$

- Perform depth dimensionality reduction
- Limit depth and increase efficiency

Example architecture for LSVRC (GoogLeNet)



Efficient architecture

- In terms of FLOPs and parameters (GoogLeNet : $\approx$ 7m)

Aggressive input size reduction at beginning

- Initial conv layer with $c = 7$ and stride 2

Simple and efficient backend

- Average pooling with extent $W_{l-1} \times H_{l-1}$ (output : 1×1)
- Followed by linear classifier

Current state-of-the-art architecture

Frontend consisting of multiple residual blocks

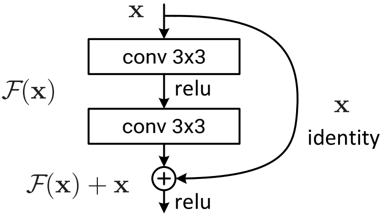


Image adapted from [2]

Motivation

- Very deep networks are hard to optimize
- Larger training error despite increased capacity

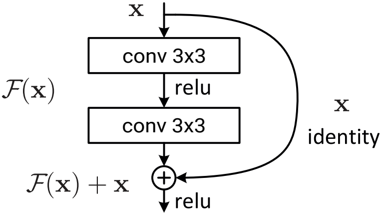


Image adapted from [2]

Residual blocks facilitate training of very deep networks

- Learn additive residual function $\mathcal{F}$ with respect to $\mathbf{x}$
- Provides guidance (learn what to add/remove from $\mathbf{x}$)

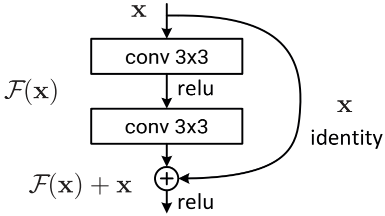


Image adapted from [2]

Residual blocks facilitate training of very deep networks

- Implemented using shortcut (skip) connections
- Residual networks currently reach depths up to 1000

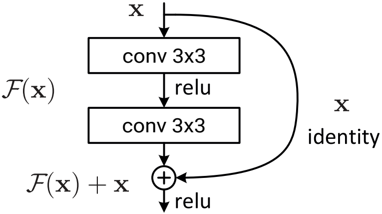
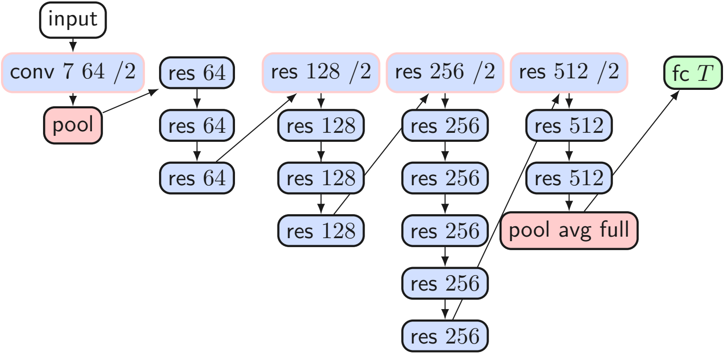


Image adapted from [2]

ResNet-34 for LSVRC



Aggressive input size reduction at beginning

Input size reduction using strided convolutions

Same efficient backend as GoogLeNet

Same input size and number of input size reductions

Number of feature maps increases with network depth

- Same initial (64) and final (512) counts
- Good guideline (but should tune on validation set)

State-of-the-art networks

- Are deep (many conv layers) and have simple linear backend
- Attempt to learn features that achieve linear separability

What works well on ImageNet works well in general

- Above architectures perform well on many datasets

Suggestions

- Use a deep ResNet for optimal performance
- Inception is good alternative if efficiency is relevant
- VGGNet (with simpler backend) is a decent baseline

Backpropagation

CNNs are trained just like linear models

- Loss function $L(\boldsymbol{\theta})$ (cross-entropy loss)
- Minibatch gradient descent

We already know everything except how to compute $\nabla L(\boldsymbol{\theta})$

- Can be done efficiently using backpropagation

Backpropagation

Recall that neural networks are computational graphs

- Function $f : \mathbf{x} \mapsto \mathbf{w}$ composed of other functions
- Loss function of neural networks is again graph

Backpropagation algorithm computes derivatives in such graphs

- Via recursive application of the chain rule

Neural networks decompose to graphs of few basic expressions

- $f^1(x_1, x_2) = x_1 x_2$
- $f^2(x_1, x_2) = x_1 + x_2$
- $f^3(x_1, x_2) = \max(x_1, x_2)$

The corresponding derivatives are

- $f^1_{x_1}(x_1, x_2) = x_2$ and $f^1_{x_2}(x_1, x_2) = x_1$
- $f^2_{x_1}(x_1, x_2) = 1$ and $f^2_{x_2}(x_1, x_2) = 1$
- $f^3_{x_1}(x_1, x_2) = 1$ if $x_1 \geq x_2$ else 0
- $f^3_{x_2}(x_1, x_2) = 1$ if $x_2 \geq x_1$ else 0

$f^1_{x_1} = \partial f^1 / \partial x_1$ is partial derivative of f^1 with respect to x_1

Simple example : expression $e(a, b) = (a + b)(b + 1)$

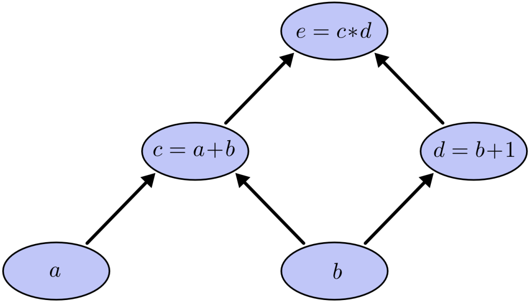


Image from colah.github.org

Evaluation (forward pass) with $a = 2$ and $b = 1$

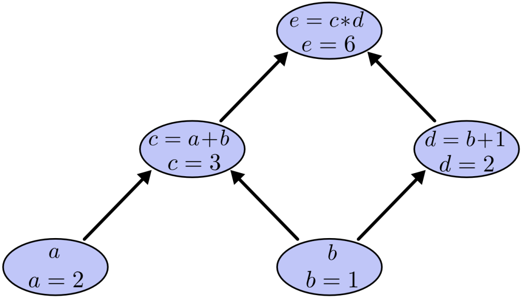


Image from colah.github.org

Every node can compute local gradients independently

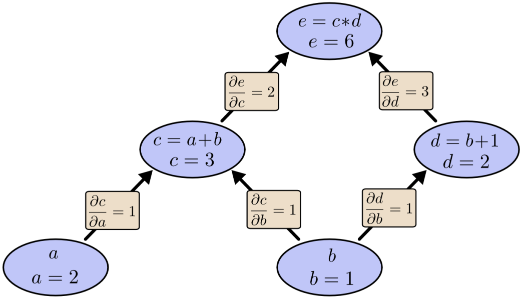


Image from colah.github.org

Can compute remaining gradients from local ones

- Using multivariate chain rule

To compute $f_{x_d}(\mathbf{x})$

- Multiply local gradients along every path from x_d to f
- Sum over all resulting values to obtain result

$$e_a(2,1) = c_a(2,1) \cdot e_c(2,1) = 1 \cdot 2 = 2$$
$$e_b(2,1) = \cdots = 1 \cdot 2 + 1 \cdot 3 = 5$$

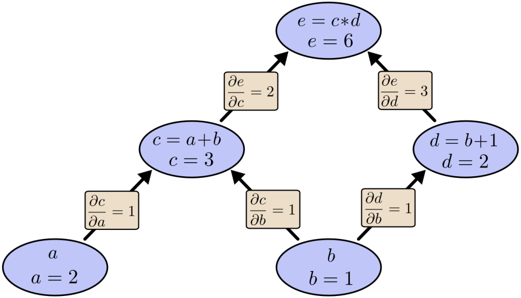


Image from colah.github.org

Backpropagation

Derivatives of $L(\boldsymbol{\theta})$

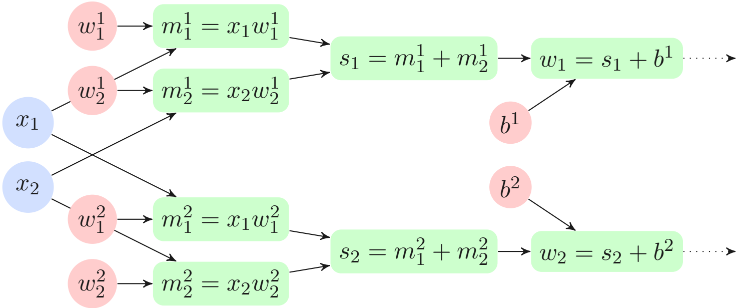
Recall that $L(\boldsymbol{\theta})$ is average over S per-sample losses $l_s(\boldsymbol{\theta})$

- Compute gradient $\nabla l_s(\boldsymbol{\theta})$ for all $s \in \{1, \dots, S\}$
- Average gradients to obtain $L(\boldsymbol{\theta})$

Above algorithm allows us to compute $\nabla l_s(\boldsymbol{\theta})$

- Can decompose any network to fundamental expressions
- Can do same for l_s (some additional expressions like $\exp(x)$)
- Composition of both is again graph

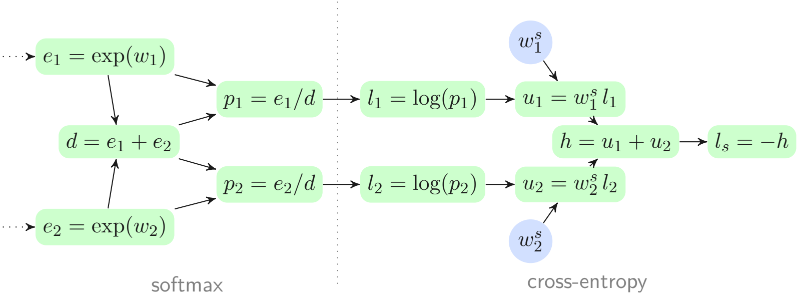
Example : linear classifier with $D = 2$ and $T = 2$



Backpropagation

Derivatives of $L(\boldsymbol{\theta})$

"Attached" decomposed cross-entropy loss l_s



Backpropagation

Derivatives of $L(\boldsymbol{\theta})$

Can compute $\nabla l_s(\boldsymbol{\theta})$

- Perform forward pass
- Compute all local gradients
- Compute $\partial l_s / \partial \theta_k$ for all parameters θ_k as above

Works but too inefficient

- Number of paths grows exponentially with graph complexity

Backpropagation
Path Factorization

Three paths from X to Y, three paths from Y to Z

$$\partial Z / \partial X = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

- 9 paths from X to Z

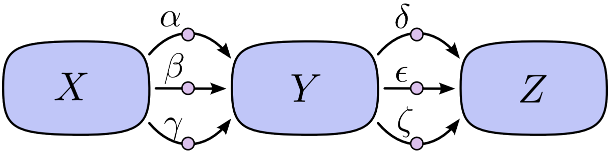


Image from colah.github.org

Much more efficient to factorize paths instead

- $\partial Z / \partial X = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$

Can use reverse-mode differentiation of Z to obtain factorization

- Computes derivative of Z with respect to *every* node
- Efficiently by touching every edge only once
- Called backpropagation in neural network community

Start at output node e and move towards inputs

At every node n

- For every child c, compute local gradient $l_c = \partial c / \partial n$
- For every child c, compute $m_c = l_c \cdot \partial e / \partial c$ ($\partial e / \partial e = 1$)
- Compute $\partial e / \partial n$ as sum over all m_c

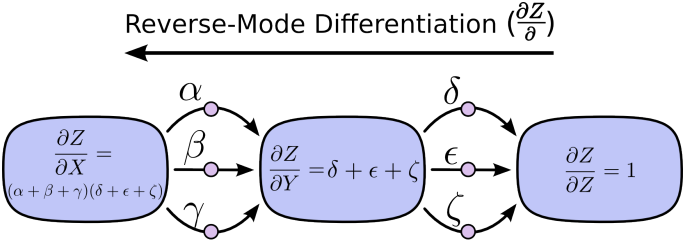


Image from colah.github.org

Backpropagation
Example 2

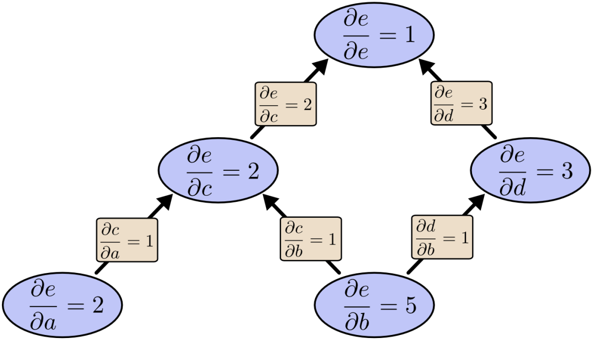
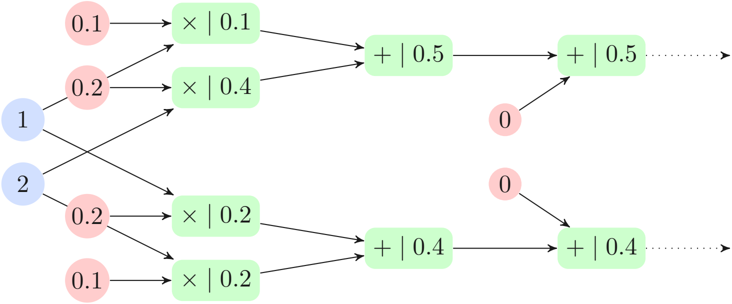


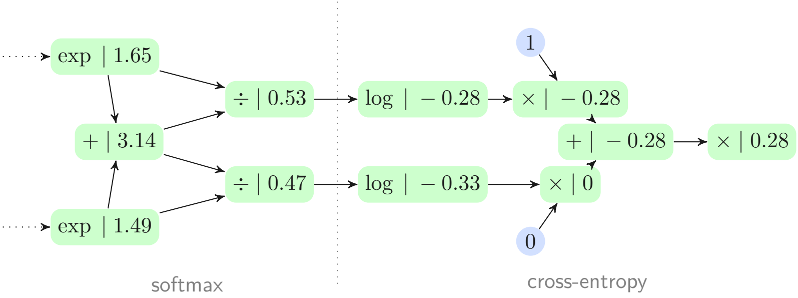
Image from colah.github.org

Backpropagation
Example 3

Forward pass using current parameters and training sample

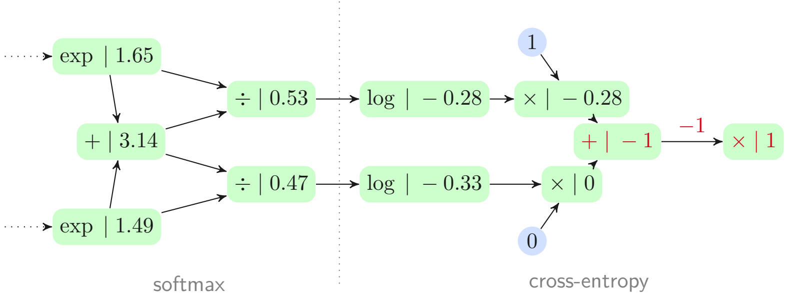


Forward pass using current parameters and training sample

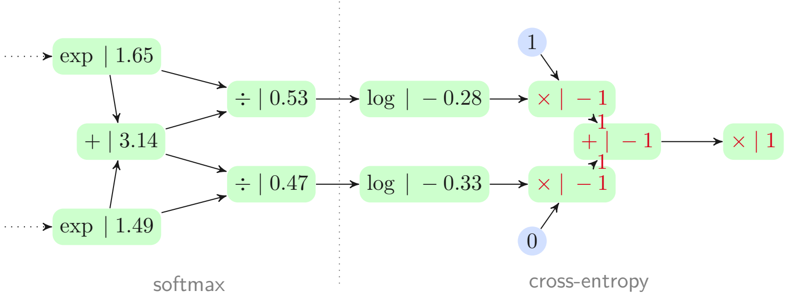


Backpropagation
Example 3

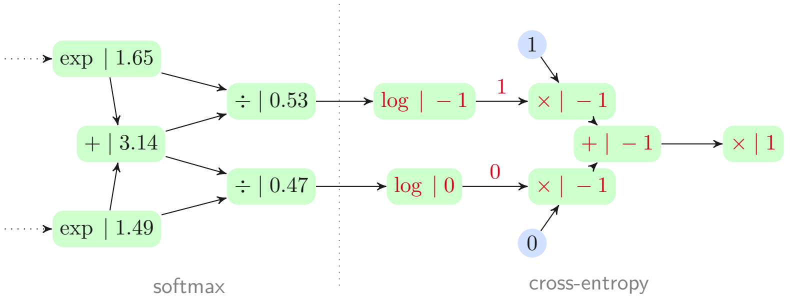
Backward pass step 1



Backward pass step 2

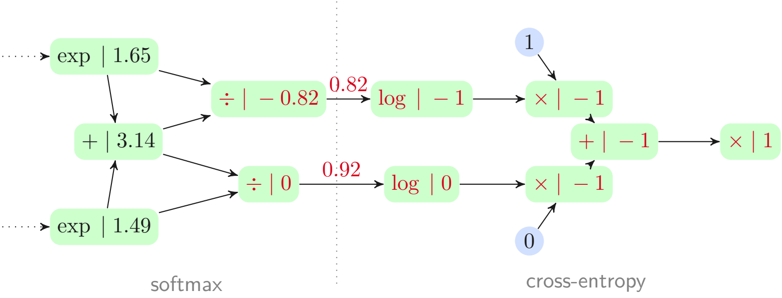


Backward pass step 3

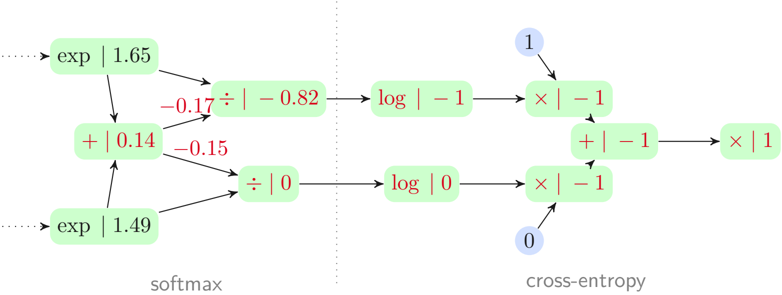


Backpropagation
Example 3

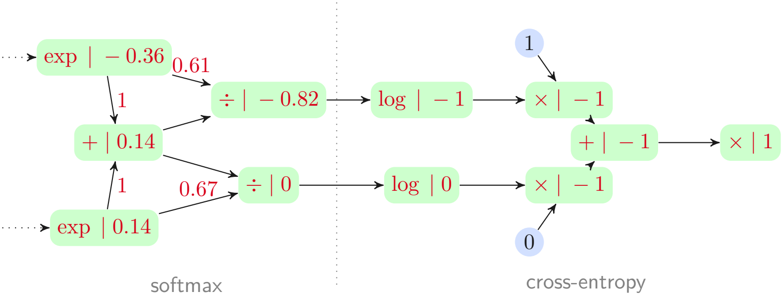
Backward pass step 4



Backward pass step 5



Backward pass step 6 (and so on ...)



Backpropagation

Backpropagation allows training of large networks

- Efficiency increase by factor of million and more

In practice decomposition is not as fine (vectorization)

- E.g. backward pass through conv layer is again convolution

Bibliography

[1] *Going deeper with convolutions*, CVPR, 2015.

[2] *Deep residual learning for image recognition*, CVPR, 2016.