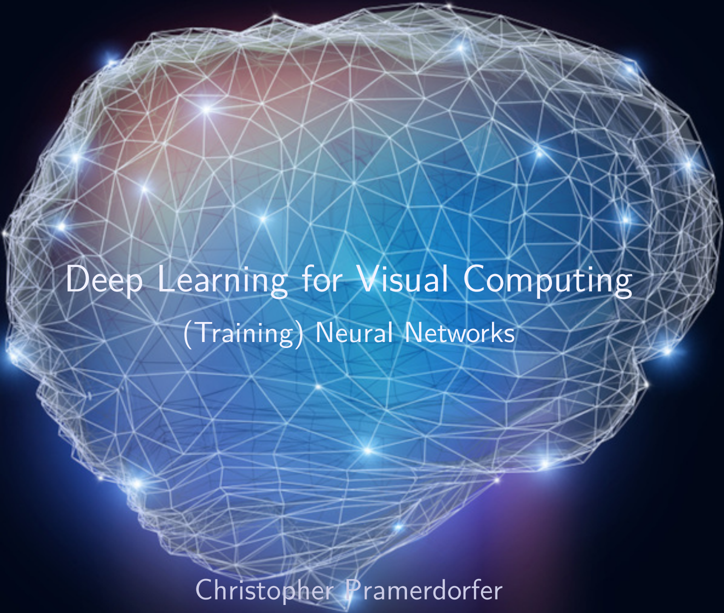


# Deep Learning for Visual Computing

## (Training) Neural Networks

Christopher Pramerdorfer

Computer Vision Lab, TU Wien

Image by Oracle

# Topics

Neural Networks

- Linear models
- Non-linear models

Training Neural Networks

- Derivatives in graphs
- Backpropagation

# Neural Networks
## Motivation

Powerful and flexible Machine Learning framework

- Describe $f : \mathbf{x} \mapsto \mathbf{w}$ via composition of simple sub-functions

Can be configured to be universal approximators

- Can represent arbitrary decision boundaries (classification)
- Can approximate any function to any degree (regression)

Deep Learning is virtually always implemented this way

An (Artificial) Neural Network (NN)

- Is a directed computational graph
- Vertices (neurons or units) are *scalar* functions of input
- Edges define data flow

And thus a function $f : \mathbf{x} \in \mathbb{R}^D \mapsto \mathbf{w} \in \mathbb{R}^T$

- That is composed of other functions (neurons)
- Neurons operate on (subsets of) $\mathbf{x}$ and/or neuron output

# Neural Networks
## Example

$$w = f(x) = f(g_1(h_1(x), h_2(x)), g_2(h_2(x), h_3(x)))$$

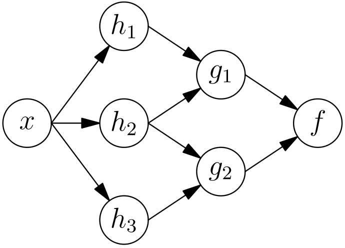



Image from wikipedia

# Feed-Forward Neural Networks
## Definition

Feed-Forward NNs are acyclic

- NNs with cycles are called Recurrent NNs

Can order neurons by distance from input(s)

- Neurons at same level in hierarchy form a layer
- Neurons in same layer usually perform same kind of operation

NNs with several layers are called deep (DNNs)

- Deep Learning is Machine Learning with DNNs
- Network below has depth of 3 (don't count input)

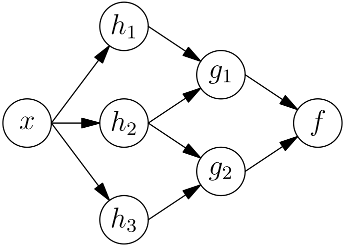



Image from wikipedia

$D$ input units $(x)$ and $T$ output units $(f)$

Flexible number of hidden units $(h, g)$

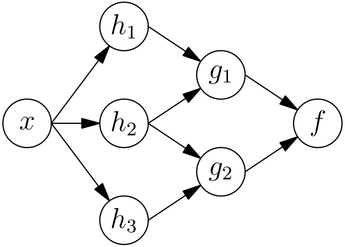



Image from wikipedia

# Linear Neural Networks

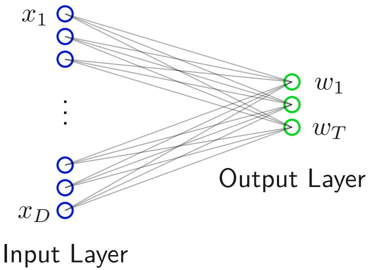
Recall that in linear models $\mathbf{w} = \mathbf{W}\mathbf{x} + \mathbf{b}$

- $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{w} \in \mathbb{R}^T$, $\mathbf{W} = [\mathbf{w}_1; \ldots; \mathbf{w}_T]$

To obtain the corresponding NN we define

- One input layer with $D$ neurons
- One output layer with $T$ neurons
- Each output neuron as $n_t(\mathbf{x}) = \mathbf{w}_t\mathbf{x} + b_t$

# Linear Neural Networks



# Linear Neural Networks

Each neuron in output layer computes linear function

- Such neurons/layers are called linear

Output neurons are connected to all neurons in previous layer

- Such neurons/layers are called fully-connected or dense

(D)NNs for classification end with a linear layer

- Sometimes the softmax is also considered part of the NN

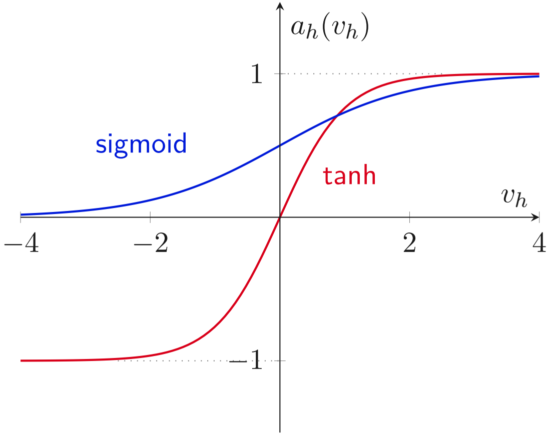
Linear NNs lack capacity ($f$ is linear)

To increase the capacity we add

- A linear hidden layer with $H$ neurons $v_h = n_h(\mathbf{x})$
- A layer with $H$ non-linear activation functions $a_h(v_h)$

Common activation functions for linear layers

- $a_h(v_h) = \tanh(v_h)$
- $a_h(v_h) = 1/(1 + \exp(-v_h))$ (logistic sigmoid)



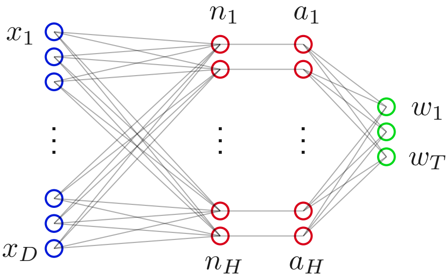
$a_h(v_h)$
1
sigmoid
tanh
$v_h$
$-4$
$-2$
2
4
$-1$

Such NNs are called Multi-Layer Perceptrons (MLPs)

- One or more of such pairs of hidden layers

Activation functions as layers

- Some papers/libraries consider activation functions own layers
- Others consider them part of the previous layer
- We will adopt both views depending on the context



$x_1$
$x_D$
$n_1$
$n_H$
$a_1$
$a_H$
$w_1$
$w_T$

Representational capacity depends on (hyperparameters)

- Number of hidden units $H$
- Type of activation functions
- Number of hidden layers (depth)

In practice a single pair of hidden layers is common

- Two pairs are used sometimes (also in DL)

## MLPs and Gradient Descent in action

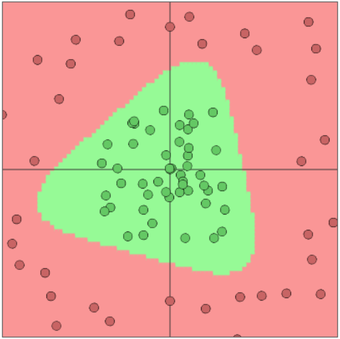


Image from cs.stanford.edu

# Training Neural Networks

We already know how to train (D)NN classifiers

- Cross-entropy loss $L(\boldsymbol{\theta})$
- Minibatch Gradient Descent with (Nesterov) momentum

In case of an MLP $\boldsymbol{\theta}$ consists of

- Multiplicative weights $\mathbf{w}_1 \cdots \mathbf{w}_H$ and $\mathbf{w}_1 \cdots \mathbf{w}_T$
- Additive biases $b_1 \cdots b_H$ and $b_1 \cdots b_T$

For Gradient Descent we must calculate $\nabla L(\boldsymbol{\theta})$

- We have not covered how to do this yet
- Can you think of an easy way?

One advantage of NNs is that this calculation is very efficient

- Enables us to train complex (D)NNs

One way to obtain $\nabla L(\boldsymbol{\theta})$ is numerical differentiation

- $\nabla L_p(\boldsymbol{\theta}) = (L(\boldsymbol{\theta} + \mathbf{1}_p \epsilon) - L(\boldsymbol{\theta}))/\epsilon$ for $p \in [1, \dim(\boldsymbol{\theta})]$
- Vector $\mathbf{1}_p$ is $1$ at position $p$ and $0$ otherwise
- Follows directly from definition of the derivative

Practical considerations

- $\epsilon$ should be close to $0$ while avoiding numerical issues
- $\nabla L_p(\boldsymbol{\theta}) = (L(\boldsymbol{\theta} + \mathbf{1}_p \epsilon) - L(\boldsymbol{\theta} - \mathbf{1}_p \epsilon))/2\epsilon$ preferable

Trivial to implement

Only an approximation ($\epsilon$ cannot be arbitrarily small)

Too inefficient in practice

- Must evaluate $L \dim(\boldsymbol{\theta})$ times
- Complex (D)NNs have millions of parameters

We thus would prefer the analytic gradient

- Obtain $\nabla L$ analytically using calculus

Can compute $\nabla L(\boldsymbol{\theta})$ directly

- Accurate (no approximation)
- Potentially much more efficient (single evaluation)

Recall that a NN is a computational graph

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

Derivatives in such graphs can be computed iteratively

- Recursive application of the chain rule
- Recall that if $F(x) = f(g(x))$ then $F'(x) = f'(g(x))g'(x)$

To compute gradients in such graphs we

- Evaluate the graph and store local results (forward pass)
- Aggregate local gradients (backward pass)

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

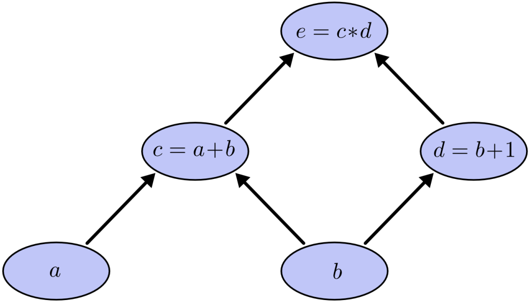



Image from colah.github.org

Forward pass with $a = 2$ and $b = 1$

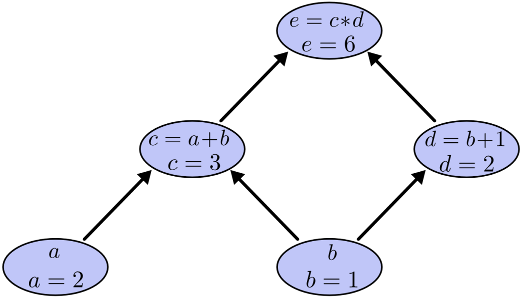



Image from colah.github.org

Every node can compute local gradients independently

- $\partial f / \partial x$ means partial derivative $f_x$

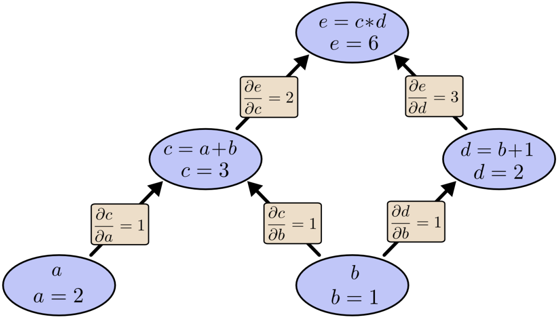



Image from colah.github.org

To obtain $\nabla e(2, 1)$ we use the multivariate chain rule

We calculate $\nabla e_a(2, 1)$ by

- Multiplying local gradients along every path from $a$ to $e$
- Summing over all resulting values

Same for $e_b$ (and all other variables in general)

$$e_a(2,1) = c_a(2,1) \cdot e_c(2,1) = 1 \cdot 2 = 2$$

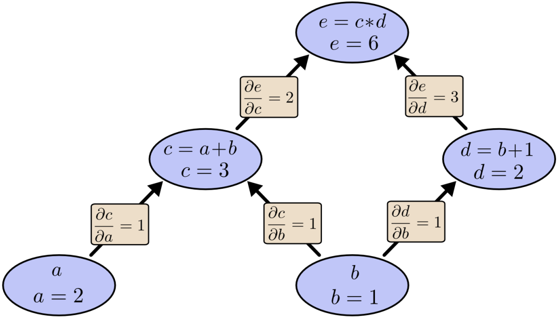



Image from colah.github.org

$$e_b(2,1) = c_b(2,1) \cdot e_c(2,1) + d_b(2,1) \cdot e_d(2,1) = 2 + 3 = 5$$

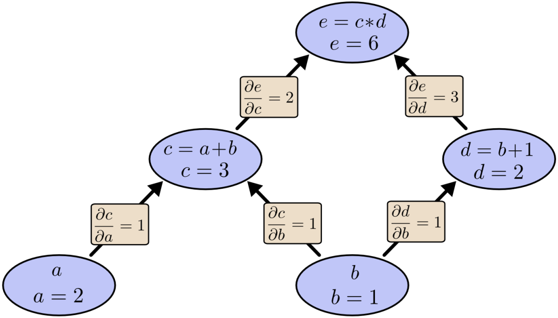



Image from colah.github.org

Can use the same algorithm to compute $L(\boldsymbol{\theta})$

Recall that the loss is an average over $S$ samples

- $L(\boldsymbol{\theta}) = 1/S \cdot \sum_s H(\mathbf{w}_s, \text{softmax}(f(\mathbf{x}_s; \boldsymbol{\theta})))$

So to compute $\nabla L(\boldsymbol{\theta})$ we

- Compute $\nabla H(\boldsymbol{\theta})$ for all $s$
- Average the results

To calculate $\nabla H(\boldsymbol{\theta})$ we

- Decompose the NN to simple functions
- Do the same for the softmax and cross-entropy
- Stack both to obtain a combined graph
- Use the same algorithm as above

MLPs are graphs of simple functions

- Can decompose the inner products

| $f$ | $f'$ |
|---|---|
| $x_1 + x_2$ | $1$ |
| $x_1 x_2$ | $x_2$ and $x_1$ |
| $\tanh(x)$ | $1 - \tanh^2(x)$ |

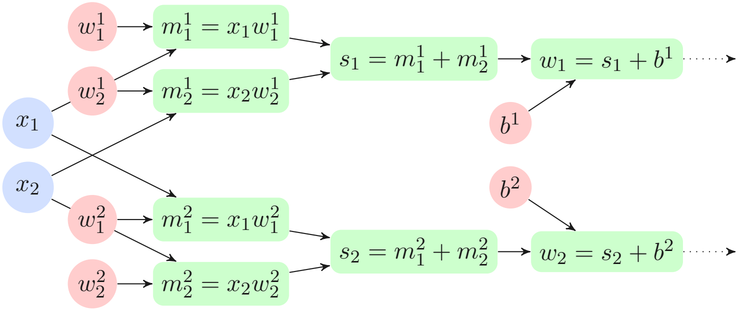
As are the cross-entropy and softmax functions

| $f$ | $f'$ |
|---|---|
| $\exp(x)$ | $\exp(x)$ |
| $\ln(x)$ | $1/x$ |
| $x_1/x_2$ | $1/x_2$ and $-x_1/x_2^2$ |

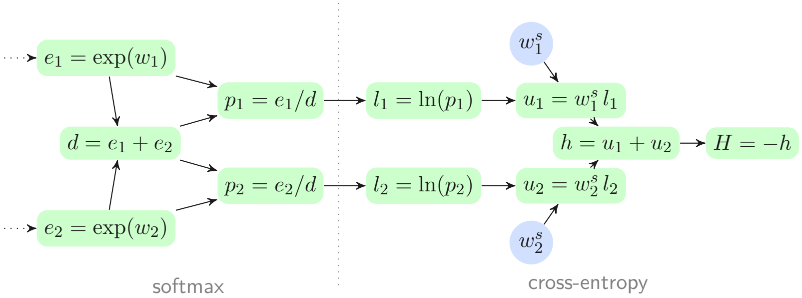
# Training Neural Networks

## Derivatives in NN classifiers

Linear classifier with $D = 2$ and $T = 2$



"Attached" softmax and cross-entropy



Graph is dense

- Must sum over many paths per partial derivative
- Number of paths grows exponentially with graph complexity
- Above algorithm not efficient enough for large NNs

Reverse-mode differentiation solves this problem

- Computes derivatives of output node wrt. all other nodes
- Efficiently by touching every edge only once
- Called backpropagation in neural network community

Achieved by

- Starting at the output (loss) node
- Propagating local gradients backwards to input nodes
- Storing intermediate results for efficiency

Start at output node $e$ and move towards inputs

At every node $n$

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

# Training Neural Networks
## Backpropagation

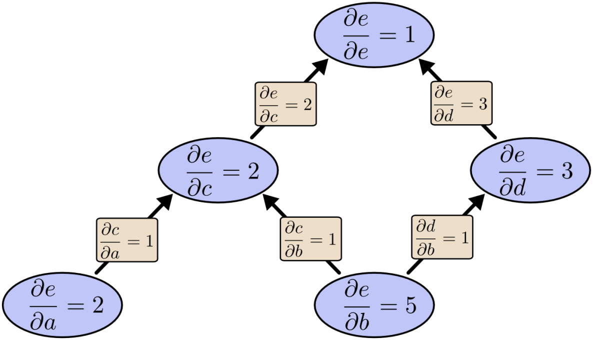



Image from colah.github.org

(D)NNs are always trained using this algorithm

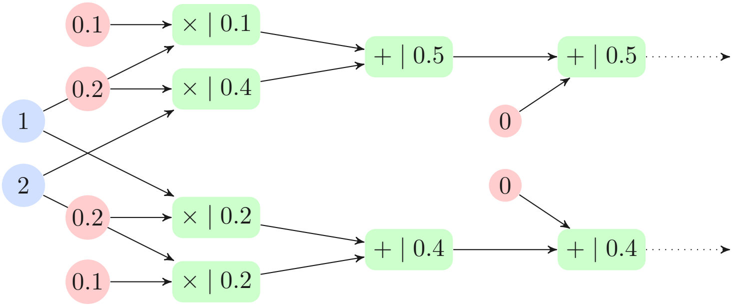
- Can increase efficiency by many magnitudes

In practice

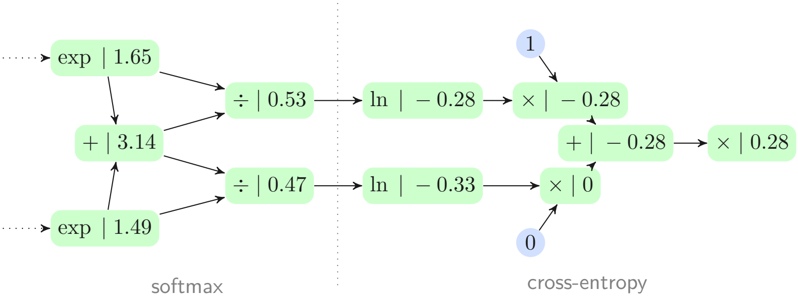
- Graph composition not as fine (vectorization)
- $\nabla H(\boldsymbol{\theta})$ computed in parallel for all $s$ (data parallelism)

# Training Neural Networks
## Backpropagation – Example

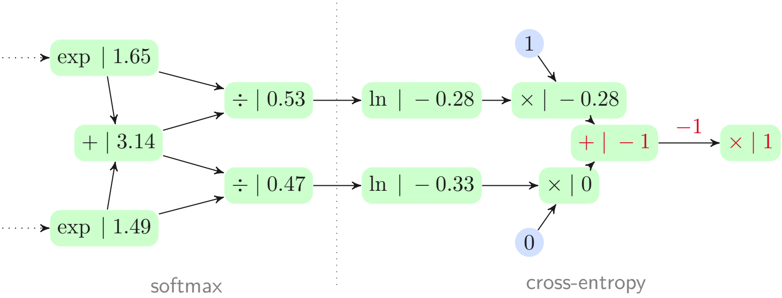
Forward pass using current parameters and training sample



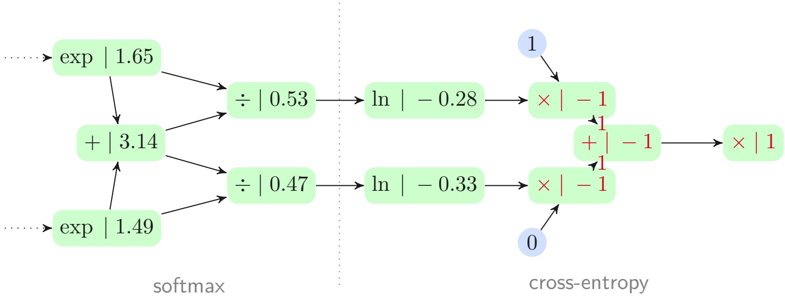
Forward pass using current parameters and training sample



Backward pass step $1$



Backward pass step $2$



Backward pass step $3$ (and so on …)