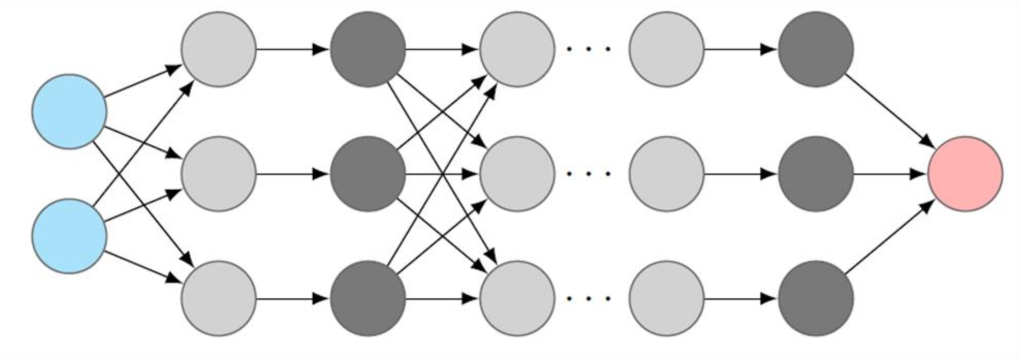
Deep Learning

Introductory course for Master students in computer science and mechatronics

Lecturer: Michael Möller – michael.moeller@uni-siegen.de
Exercises: Hartmut Bauermeister – hartmut.bauermeister@uni-siegen.de



Deep Learning

My perspective on what (supervised) "Deep Learning" is: A fancy word for function approximation

Assume there is an unknown function G that maps some kind of input data x to some kind of desired output y.

Assume we are given some evaluations of this (unknown) function G. This is what we will call ***training data***!

1. Choose a parameterized function $\mathcal{N}(x;\theta)$ in the hope that for the right choice of parameters θ it approximates the unknown function G well. We call $\mathcal{N}$ the ***network***, and sometimes refer to θ as the ***weights***.
2. Try to determine suitable weights θ in such a way that $\mathcal{N}(x_i;\theta) \approx y_i$ holds for all examples (x_i, y_i) from your training data set. This is referred to as ***training the network***.
3. Make try to ensure that both, the architecture as well as the training are chosen in such as way that the network makes good predictions during inference, i.e. on previously unseen data x: $\mathcal{N}(x;\theta) \approx G(x)$. We refer to this property as ***generalization***.

Linear regression

Training at the example of using a linear network and quadratic *loss function*

$$\mathcal{N}(x_j;\theta) = \theta \begin{pmatrix} x_j \\ 1 \end{pmatrix}$$

$$\mathcal{L}(\mathcal{N}(x_j,\theta), y_j) = \|\mathcal{N}(x_j,\theta) - y_j\|^2$$

The overall quality of the current parameters θ is then measured by summing the loss function over all training examples:

$$E(\theta) = \sum_j \|\mathcal{N}(x_j,\theta) - y_j\|^2$$

Finally, one tries to determine the optimal parameters θ as the *argument that minimizes the training costs*:

$$\hat{\theta} = \arg\min_\theta E(\theta)$$

For (affine) linear networks this results in a linear equation! Why?

During *inference*, one uses $\mathcal{N}(x;\hat{\theta})$ to make predictions.

Optimality conditions

If all partial derivatives of a function $E : \mathbb{R}^n \to \mathbb{R}$ exist and are continuous, we call E *continuously differentiable*, and call

$$\nabla E := \begin{pmatrix} \frac{\partial E}{\partial \theta_1} \\ \vdots \\ \frac{\partial E}{\partial \theta_n} \end{pmatrix} : \mathbb{R}^n \to \mathbb{R}^n$$

the *gradient* of E. Evaluating the gradient at a point θ yields a vector in $\mathbb{R}^n$.

Necessary condition for local minima: If $E : \mathbb{R}^n \to \mathbb{R}$ has a local minimum at some point θ, then it holds that

$$\nabla E(\theta) = 0$$

Fully connected networks

Deep Learning = Using deeply nested functions!

$$\mathcal{N}(x;\theta) = \ell^L(\ell^{L-1}(\dots(\ell^1(x;\theta^1)\dots);\theta^{L-1});\theta^L)$$

Architecture of fully connected networks

If the index i is odd:

$$\ell^i : \mathbb{R}^{n_i} \times \mathbb{R}^{n_{i+1}\times(n_i+1)} \to \mathbb{R}^{n_{i+1}}$$

$$\ell^i(z;\theta^i) = \theta^i \begin{pmatrix} z \\ 1 \end{pmatrix}$$

Output vector of size n^{i+1}

Parameter matrix of size $n^{i+1} \times (n^i + 1)$

Input vector concatenated with 1

This is called a *fully connected layer*!

If the index i is even:

Use a simple, componentwise nonlinear function!

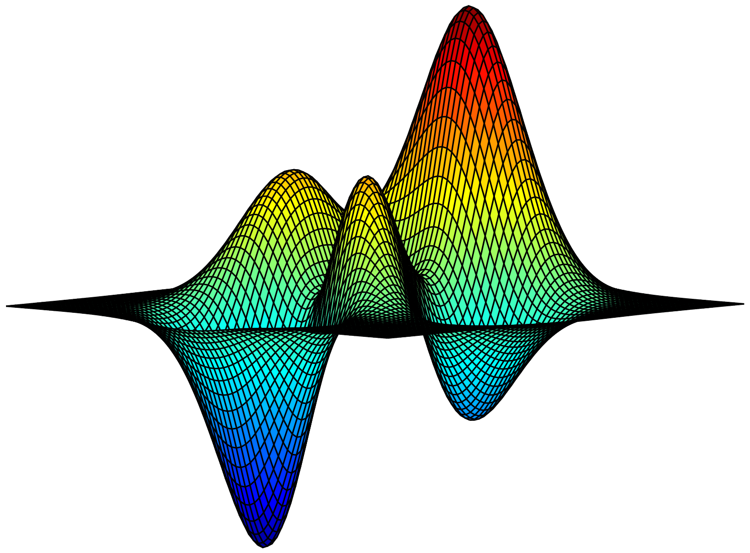
Example: *rectified linear unit*

$$\ell^i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$$

$$(\ell^i(z))_j = \max(z_j, 0)$$

This is called an activation function!

Training algorithms



Unfortunately, as soon as the network has at least one hidden layer, the optimization becomes much more difficult!

Reason: High dimensional nonconvex functions can rarely be optimized to global optimality!

Common deep learning approach: Resign the desire to compute global minimizers! Iteratively reduce the training costs – at most until you reach $\nabla E(\hat{\theta}) = 0$. One never even checks sufficient conditions for *local* minima.

Most commonly used algorithms are variants of

gradient descent!

Gradient descent

Basic idea: For a continuously differentiable $E : \mathbb{R}^n \to \mathbb{R}$, the quantity $-\nabla E(\theta)$ points into the direction of steepest descent.

Move into this direction!

$$\underbrace{\theta(k+1)}_{\text{New parameters}} = \underbrace{\theta(k)}_{\text{Previous parameters}} \underbrace{- \; \tau \, \nabla E(\theta(k))}_{\text{Direction of steepest descent}}$$

The parameter τ is called *step-size* or *learning rate*.

Discussions on the board:

1. If the iteration converges, it converges to a point $\hat{\theta}$ with $\nabla E(\hat{\theta}) = 0$
2. For a sufficiently small τ it holds that $E(\theta(k+1)) \leq E(\theta(k))$,

   (even strict inequality if the algorithm has not yet converged)

UNIVERSITÄT SIEGEN

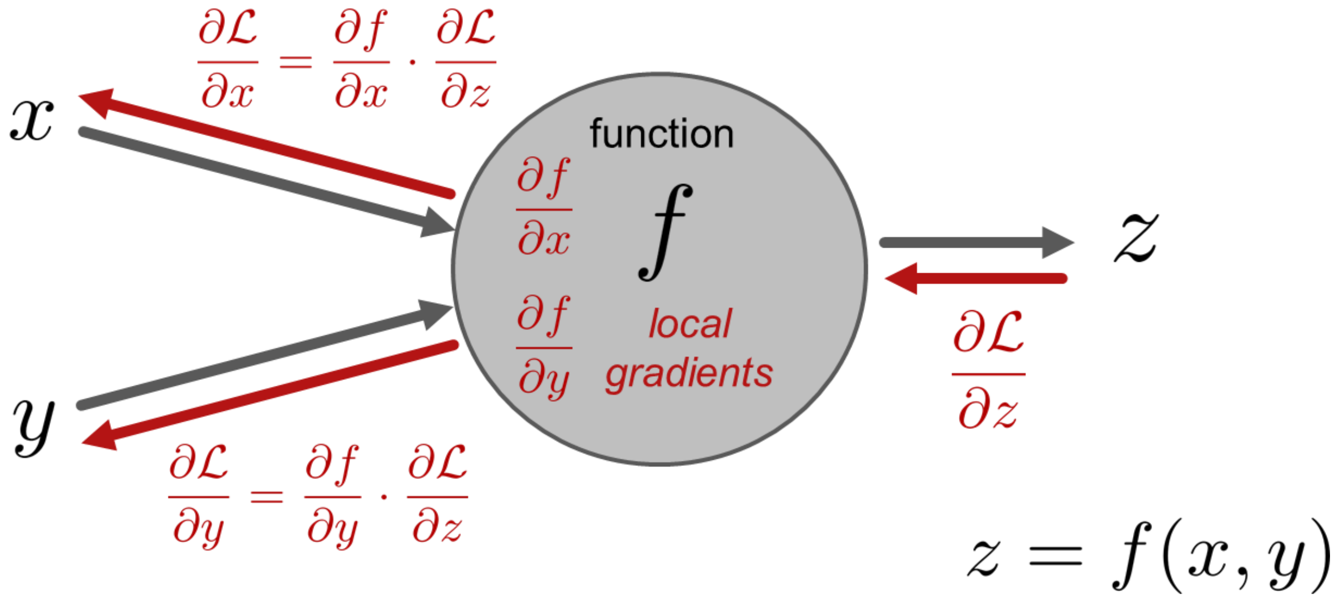
Gradient computation

How do we compute the gradient of a deeply nested function?

Remember: **Chain rule**

$$E = f \circ g \quad \Rightarrow \quad \nabla E(x) = \nabla g(x) \cdot \nabla f(g(x))$$

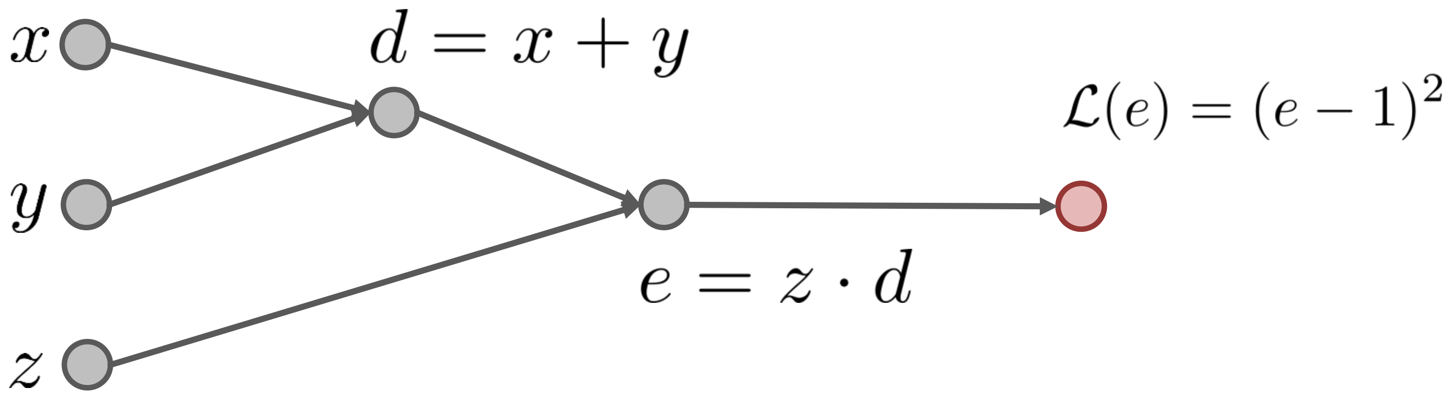
Essetially, gradient computations (known under the name of backpropagation) are just a repeated application of the chain rule!



$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial \mathcal{L}}{\partial z}$
x
function
$\frac{\partial f}{\partial x}$
f
$\frac{\partial f}{\partial y}$
local gradients
z
$\frac{\partial \mathcal{L}}{\partial z}$
y
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial f}{\partial y} \cdot \frac{\partial \mathcal{L}}{\partial z}$
$z = f(x, y)$

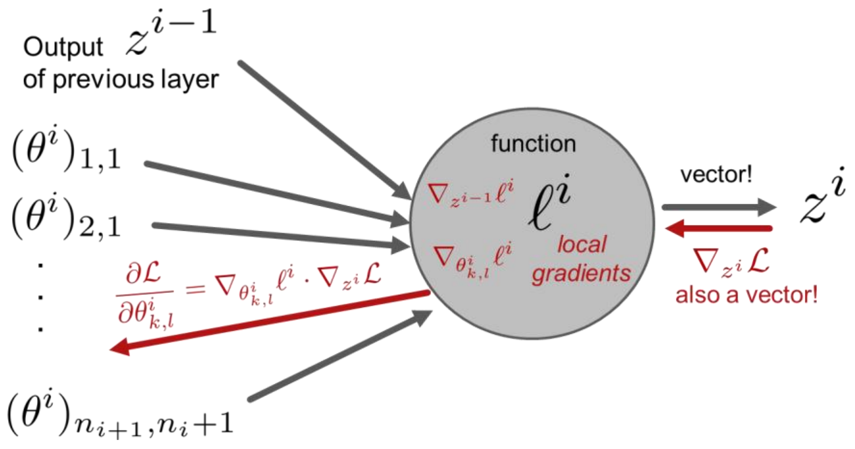
Backpropagation - examples

Scalar example:



Vector- and matrix-valued computations:

$$\mathcal{N}(x;\theta) \;=\; \theta_w^2\ \sigma(\theta_w^1 x \;+\; \theta_b^1) + \theta_b^2 \qquad E(\theta) = \|\mathcal{N}(x;\theta) - y\|^2$$



Output z^{i-1}
of previous layer
$(\theta^i)_{1,1}$
$(\theta^i)_{2,1}$
$(\theta^i)_{n_{i+1},n_i+1}$
$\frac{\partial \mathcal{L}}{\partial \theta^i_{k,l}} = \nabla_{\theta^i_{k,l}} \ell^i \cdot \nabla_{z^i}\mathcal{L}$
function
ℓ^i
$\nabla_{z^{i-1}}\ell^i$
$\nabla_{\theta^i_{k,l}}\ell^i$
local
gradients
vector!
z^i
$\nabla_{z^i}\mathcal{L}$
also a vector!

$$f(A,B) = A \cdot B \qquad A \in \mathbb{R}^{m \times n} \quad B \in \mathbb{R}^{n \times o}$$

"$\nabla_A f(A,B) =$ right multiplication with B^T"

"$\nabla_B f(A,B) =$ left multiplication with A^T"

Stochastic gradient descent (SGD)

Problem:

$$E(\theta) = \sum_{\text{training examples } j} \mathcal{L}(\mathcal{N}(x_j;\theta), y_j)$$

Can easily be a sum over 1,000,000 terms

Idea: Use only a few random summands to compute an approximate gradient:

$$E_k(\theta) = \sum_{j\in I(k)} \mathcal{L}(\mathcal{N}(x_j,\theta), y_j)$$

for a ***very small index set*** $I(k)$

Update the parameters using this approximation

$$\theta(k+1) = \theta(k) - \tau\nabla E_k(\theta(k)) \approx \theta^k - \tau\nabla E(\theta^k)$$

Randomly selecting entries in the index set $I(k)$ leads to the name ***stochastic gradient descent***. The training examples (x_j, y_j) with $j \in I(k)$ are called a ***mini-batch***.

Stochastic gradient descent (SGD)

Theoretical result: In order for

$$\lim_{k\to\infty} \mathbb{E}[\|\nabla E(\theta(k))\|^2] = 0$$

the step size $\tau(k)$ needs to satisfy

$$\sum_{k=1}^{\infty} \tau(k) = \infty, \qquad \sum_{k=1}^{\infty} \tau(k)^2 < \infty$$

Popular choice of stepsize: ***AdaGrad***

$$c(k+1) = c(k) + \|\nabla E_k(\theta(k))\|^2, \qquad \tau(k) = (c(k+1))^{-1/2}$$

$$\theta(k+1) = \theta(k) - \tau(k)\ \nabla E_k(\theta(k))$$

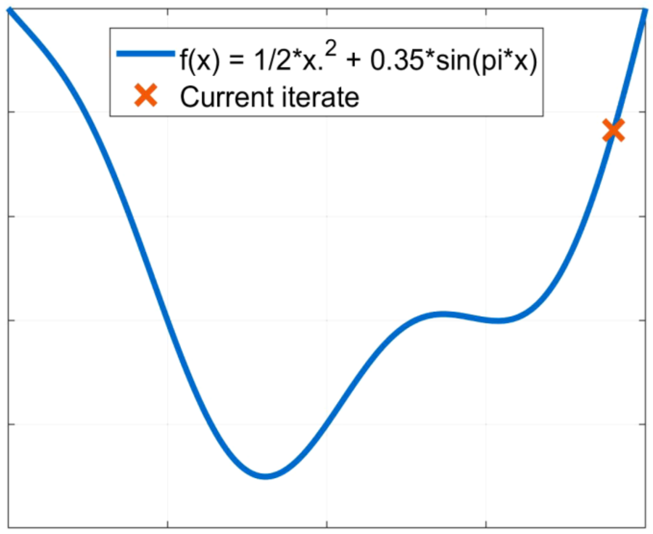
GD with Nesterov Momentum

$$\underbrace{\theta(k+1) = \theta(k) - \tau(k)\nabla E(\theta(k))}_{\text{Gradient descent}} + \underbrace{\alpha * v(k+1)}_{\text{+ additional velocity}}$$

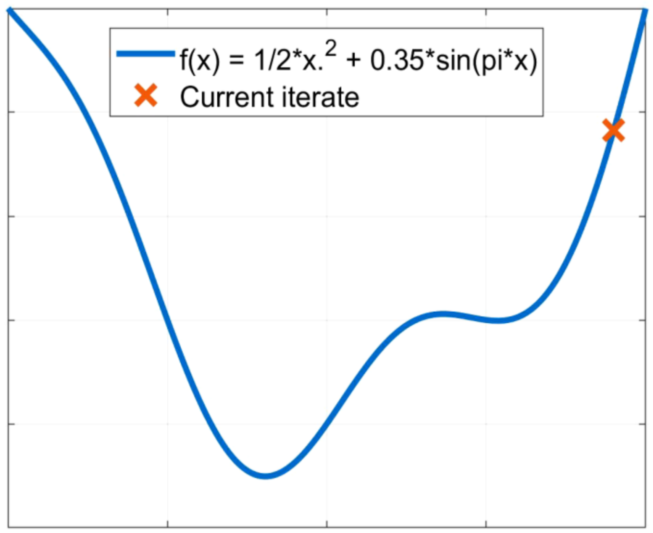
$$\underbrace{v(k+1)}_{\text{new velocity}} = \alpha \cdot \underbrace{v(k)}_{\text{old velocity}} - \underbrace{\tau(k)\nabla E(\theta(k))}_{\text{accumulating gradients, i.e. ``speeds + directions''}}$$

α ↓ damping with $\alpha < 1$, could be interpreted as *friction*

Gradient descent



Gradient descent with *Momentum*



Adam

SGD + Nesterov Momentum is one of the two most popular methods.

Possibly slightly more popular: Adam.

It combines some techniques we have seen before.

A notion of velocity:

convex combination

$$v(k+1) = \cdot \qquad \cdot \quad \cdot \ (\beta_1 \cdot v(k) - (1-\beta_1) \cdot \nabla E_k(\theta(k)))$$

new velocity

old velocity

Stochastic approximate gradient

Adam

SGD + Nesterov Momentum is one of the two most popular methods.

Possibly slightly more popular: Adam.

It combines some techniques we have seen before.

A notion of velocity:

$$v(k+1) = \frac{1}{1-(\beta_1)^k} \cdot \quad (\beta_1 \cdot v(k) - (1-\beta_1) \cdot \nabla E_k(\theta(k)))$$

Keeping track of the norm of the stochastic gradients (similar to AdaGrad):

$$c(k+1) = \qquad \cdot \quad \left(\beta_2 c(k) + (1-\beta_2)\, (\nabla E_k(\theta^k) \odot \nabla E_k(\theta^k))\right)$$

convex combination

element-wise product

Adam

SGD + Nesterov Momentum is one of the two most popular methods.

Possibly slightly more popular: Adam.

It combines some techniques we have seen before.

A notion of velocity:

$$v(k+1) = \frac{1}{1-(\beta_1)^k} \cdot \quad (\beta_1 \cdot v(k) + (1-\beta_1)\cdot \nabla E_k(\theta(k)))$$

Keeping track of the norm of the stochastic gradients (similar to AdaGrad):

$$c(k+1) = \frac{1}{1-(\beta_2)^k} \cdot \quad \left(\beta_2 c(k) + (1-\beta_2)\ (\nabla E_k(\theta^k) \odot \nabla E_k(\theta^k))\right)$$

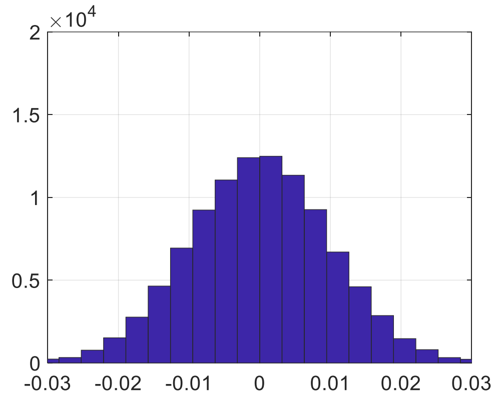
Move into the velocity direction with a vector-valued stepsize similar to AdaGrad:

$$\theta(k+1) = \theta(k) - \tau(k) \cdot d(k+1) \odot v(k+1), \qquad d(k+1) = (c(k+1)+\epsilon).^{\wedge(-1/2)}$$

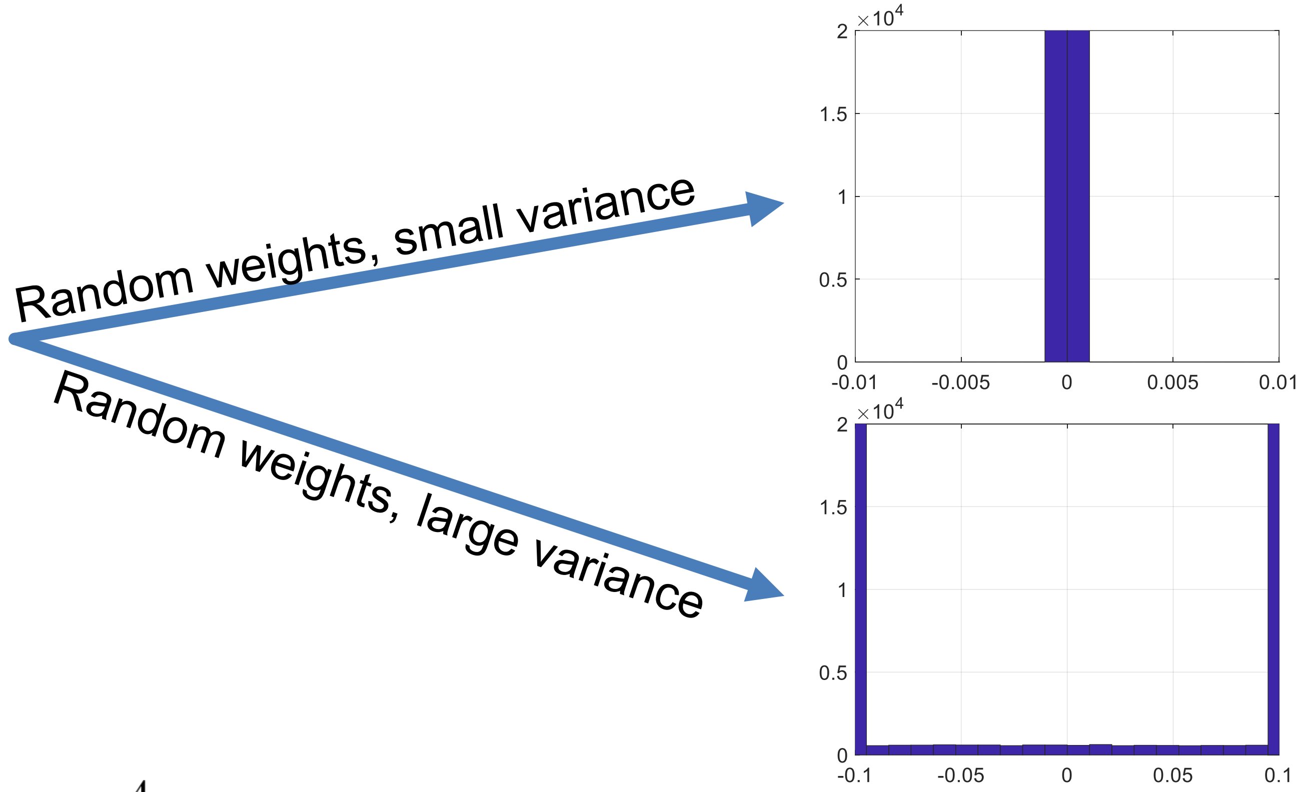
element-wise product

element-wise 1/sqrt

How to start?



Distribution of data



Investigation suggests:

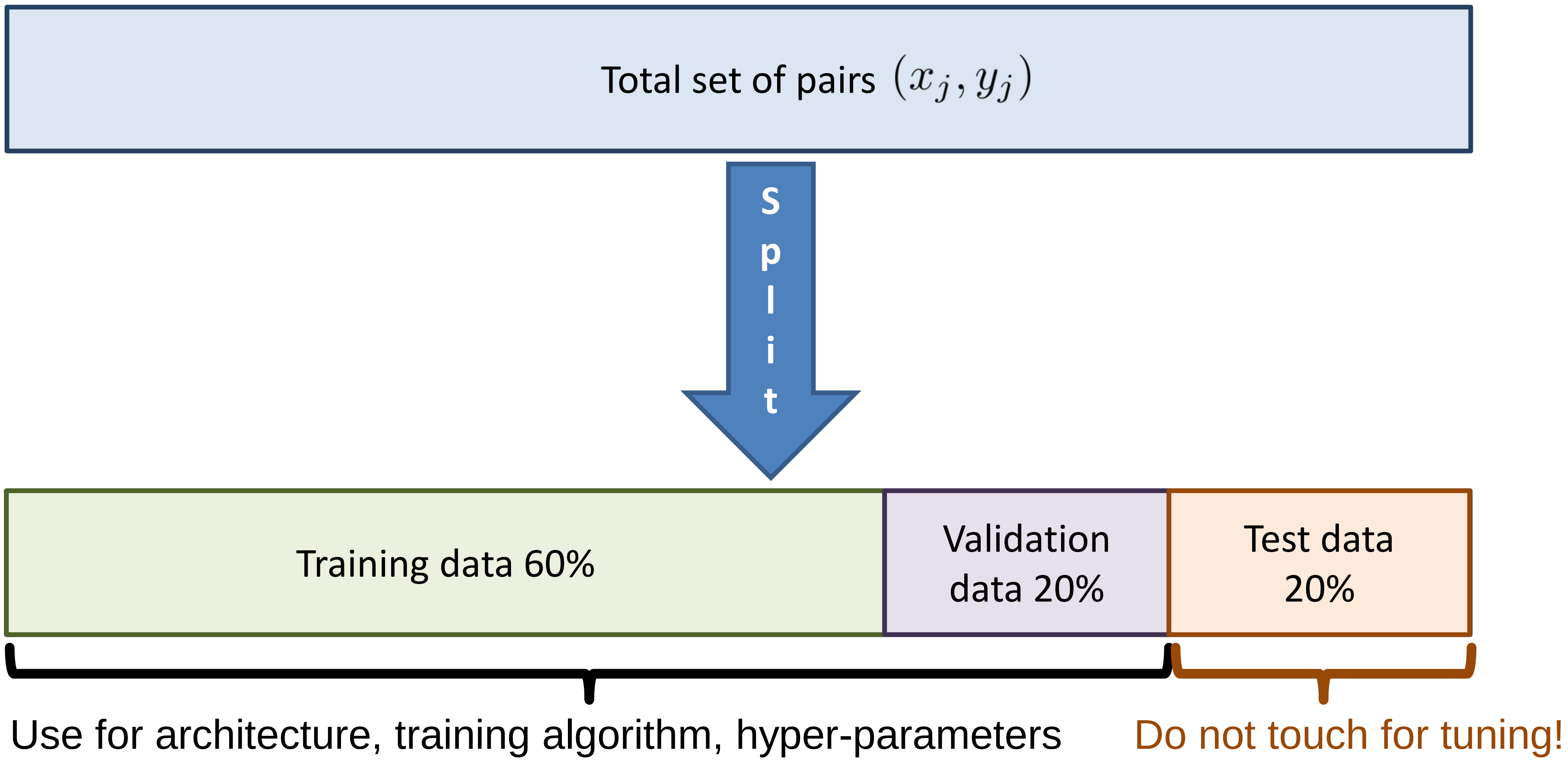
We choose $\mathrm{var}(\theta^i) = \dfrac{4}{n_i + n_{i+1}}$ for a fully connected layer $\theta^i \in \mathbb{R}^{n_{i+1} \times n_i}$

in a network with ReLUs.

Option: Use explicit normalization by batchnorm! It comes with parameters that allow you to undo the normalization.

Good practice: Validate!

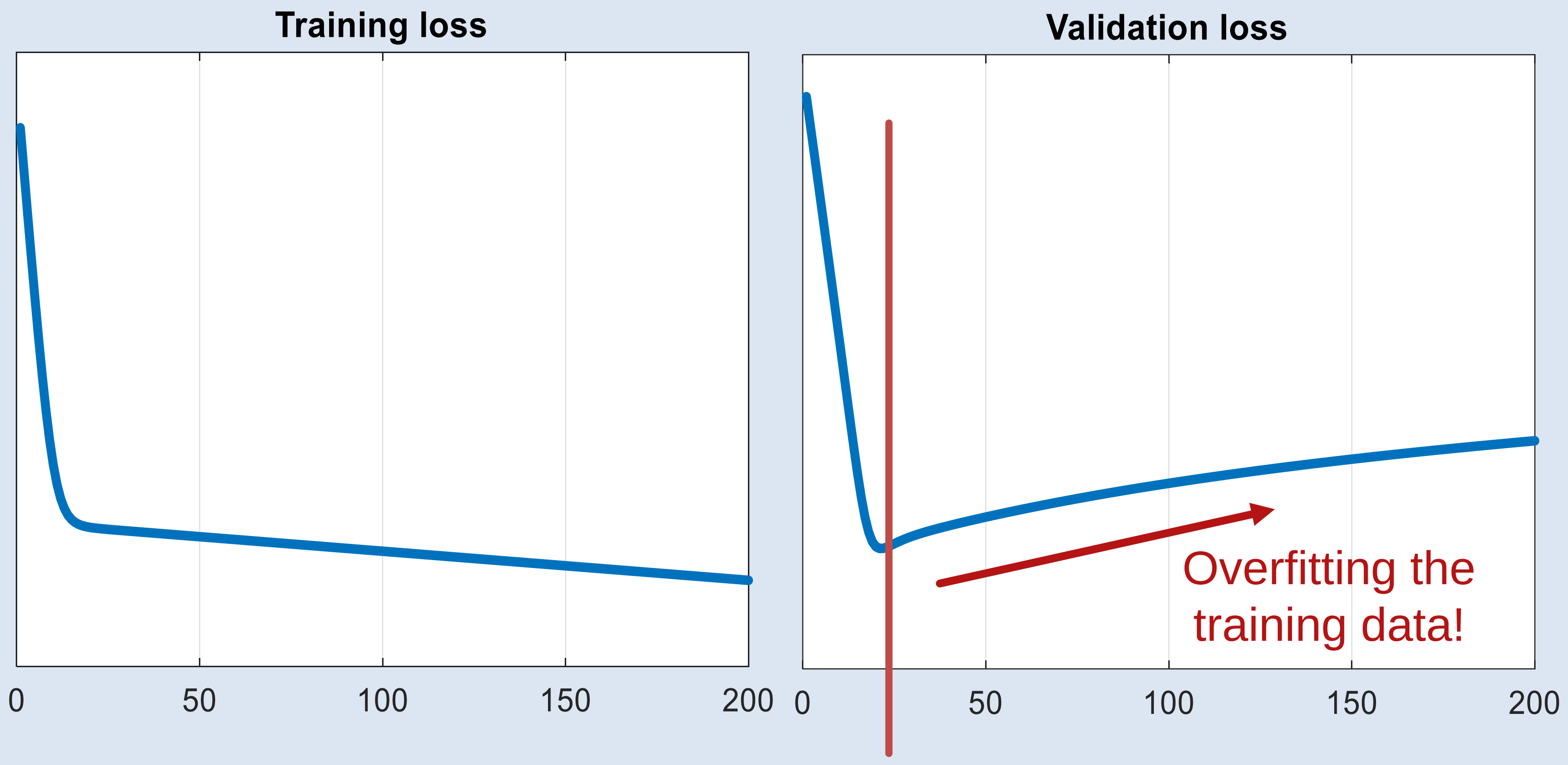
Without validation it is impossible to judge whether your model is reasonable or starts overfitting the data! Therefore:



Total set of pairs (x_j, y_j)
Split
Training data 60%
Validation data 20%
Test data 20%
Use for architecture, training algorithm, hyper-parameters
Do not touch for tuning!

Good practice: Validate!

How do you use the validation set?



This is where you want to stop your optimizer!

Ways to improve generalization

We have seen the following generalization techniques

Early stopping
Stop optimizing once the validation loss increases!

Data augmentation
If you know transformations that should not alter your prediction, use them to generate more training data

Regularization with a penalty
Bound or penalize your weights, e.g. by a quadratic penalty. Use any kind of prior information!

Dropout
Train redundant networks that can accomplish a task with many different features!

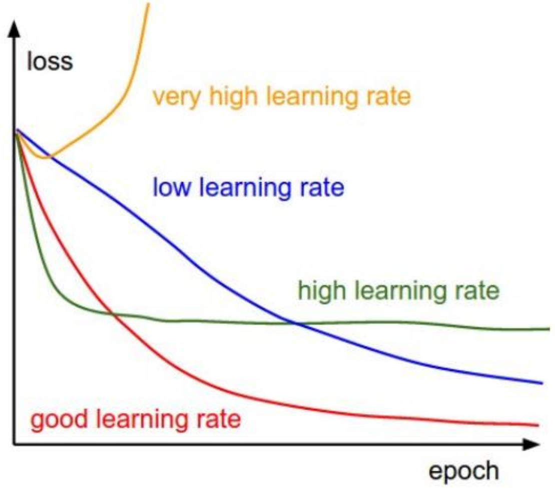
Deep Learning Recipe

Do you have a small training error?

No → Does the training error decay reasonably at the beginning of the training?

Yes → Increase the network's expressiveness.

No → Tune your training.



Yes ↓

Do you have a small validation error?

No → Use more data (if possible).

Regularization techniques.

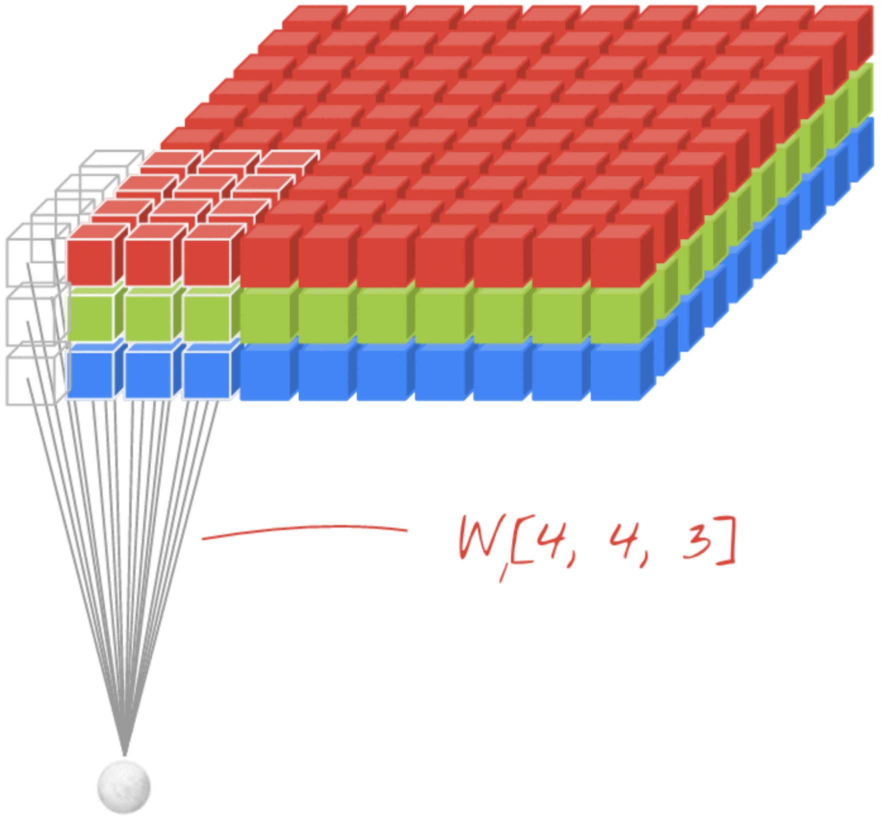
Use prior knowledge (modelling parts of the network)

Yes ↓

Perfect! You're done!

If nothing works, try a different network architecture.

Correlation/Convolution

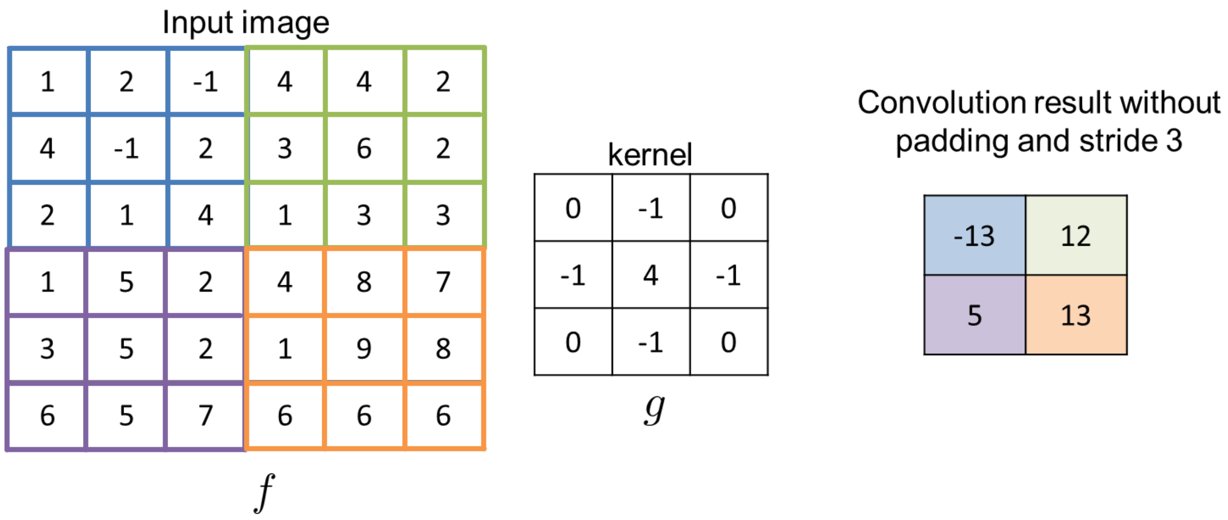


Animation taken from https://sites.google.com/site/nttrungmtwiki/home/it/data-science---python/tensorflow/tensorflow-and-deep-learning-part-3

Are you able to write down a formula for this operation?

Reducing the size

Stride:



Another frequently used way to reduce the size of the image are ***pooling layers***

All pooling variants use a sliding window over the image (similar to a convolution), but often in a non-overlapping fashion. Each window (of which one can specify the size), gets reduced to a single number, by

- Taking the maximum value among the entries within the window (*max.-pooling*)
- Taking the average value among the entries within the window (*avg.-pooling*)
- Less frequent: Taking the ℓ^p norm of each window (*fractional max-pooling*)

Transposed Convolution

- A convolution is a linear operator
- Any linear operator can be respresented as a matrix (for a given basis)
- Any matrix has a transpose
- The operation arising from transposing a convolution matrix is called transposed convolution or deconvolution or adjoint of a convolution.

1	0	-2	3	1	2	-1

$*$

0.5	1	0.5

$$\|$$

$$\begin{pmatrix} 0.5 & 1 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 1 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 1 & 0.5 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -2 \\ 3 \\ 1 \\ 2 \\ -1 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 2.5 \\ 2 \end{pmatrix}$$

Image classification

"Car"

"Balloon"

"Giraffe"

Representation of a classification result: $y \in \mathbb{R}^c$, where c is the number of classes. Identification: $y = e_i \Leftrightarrow \text{the object belongs to class } i$

Most common loss: Cross entropy

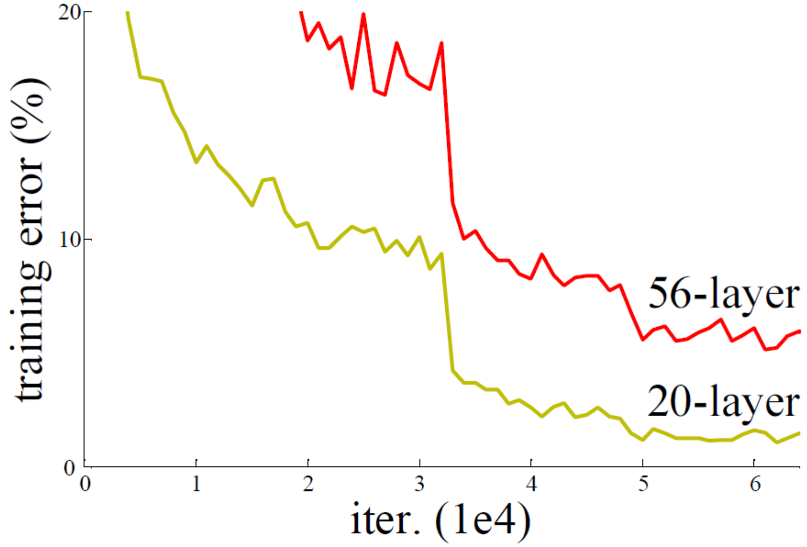
$$\begin{aligned}\mathcal{L}(z,y) &= -\sum_i y_i \log(z_i),\\ &= -\log(z_i) \quad \text{if } y = e_i\end{aligned}$$

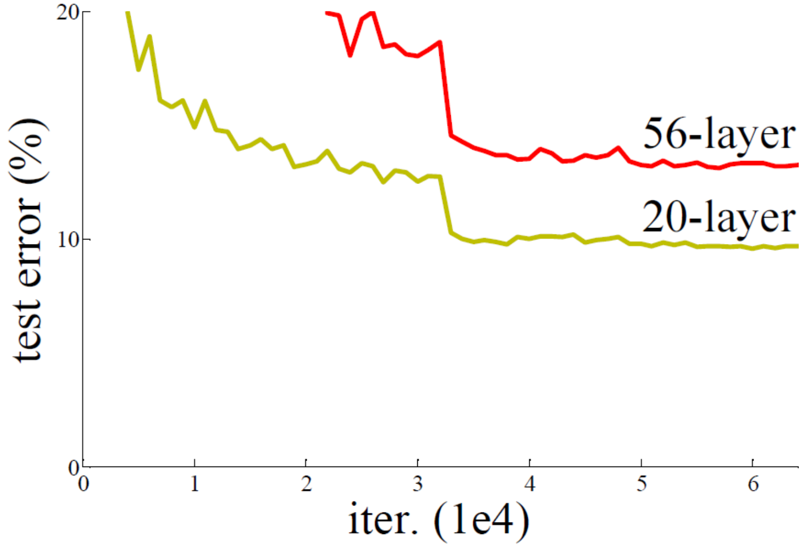
When combined with a soft-max

$$\begin{aligned}\mathcal{L}(sm(x),y) &= -\log\left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right),\\ &= -x_i + \log\left(\sum_j e^{x_j}\right),\end{aligned}$$

Going deeper?

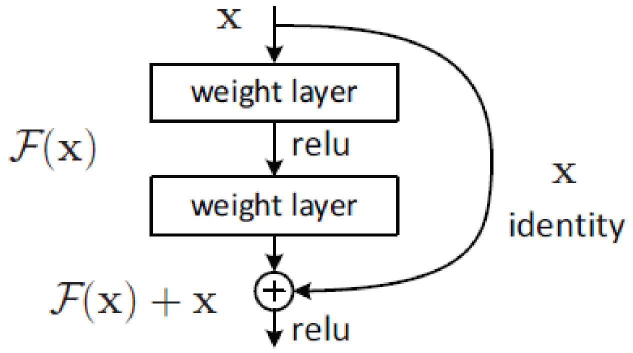
Strange behavior: At some point, going deeper does not improve the results DESPITE the network neither suffering from overfitting nor from an obvious problem in the training!





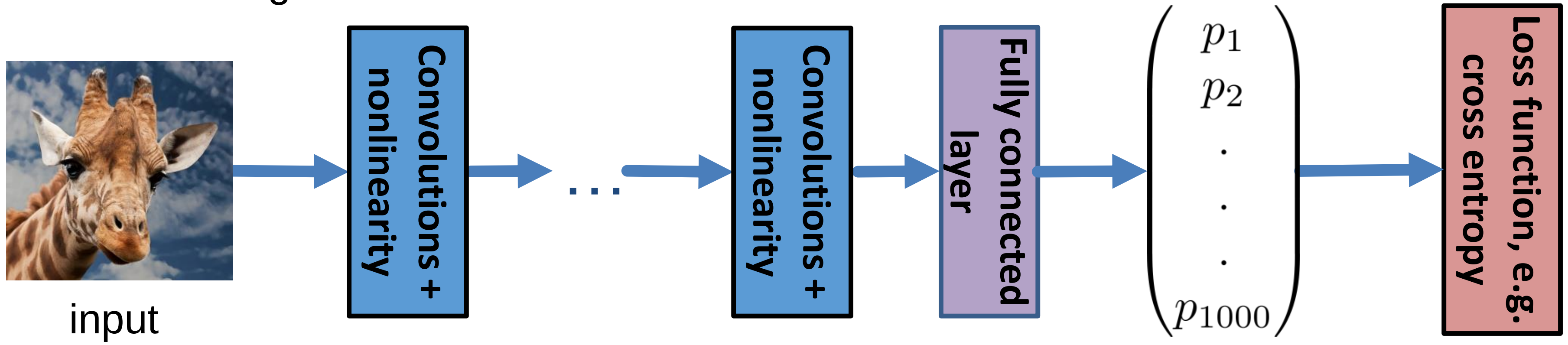
From He et al. “Deep Residual Learning for Image Recognition”

Possible solution: Use skip connections!



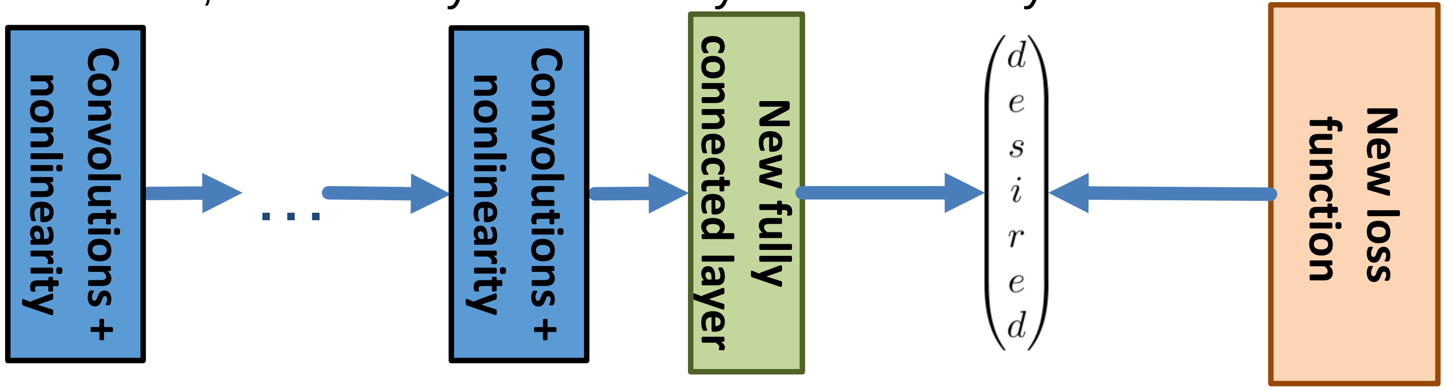
Transfer Learning

1. Train a network on a large annotated dataset, e.g. train for good classification results on imagenet



input
Convolutions + nonlinearity
...
Convolutions + nonlinearity
Fully connected layer
$\begin{pmatrix} p_1 \\ p_2 \\ \cdot \\ \cdot \\ \cdot \\ p_{1000} \end{pmatrix}$
Loss function, e.g. cross entropy

2. Copy the trained network, throw away the last fully connected layer



Convolutions + nonlinearity
...
Convolutions + nonlinearity
New fully connected layer
$\begin{pmatrix} d \\ e \\ s \\ i \\ r \\ e \\ d \end{pmatrix}$
New loss function

3. Assign a new layer, e.g. fully connected, and a new loss.

4. Train new network on little data, possibly freeze (or reduce stepsize for) convolutional weights.

Algorithm-based architectures

If you have an algorithm that works well on a certain problem, exploit its power and design network architectures along the algorithm's structure!

For example: If gradient descent on a model-based cost function works well, e.g.

$$x^{k+1} = x^k - \tau \nabla E(x^k)$$

You could replace it by

$$x^{k+1} = x^k - \tau \mathcal{N}(\nabla E(x^k); \theta)$$

for a small network $\mathcal{N}$ parameterized by θ.

Using 2 iterations of the modified gradient descent would result in an architecture

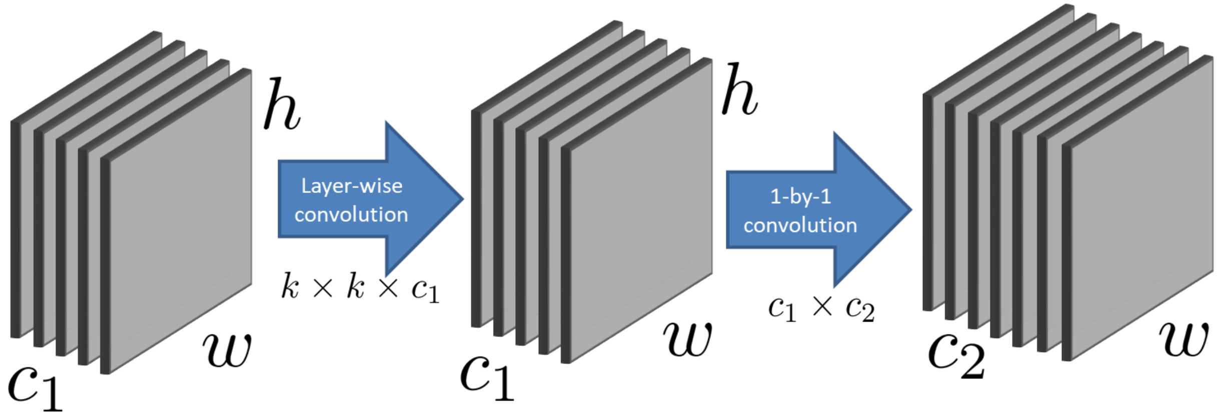
$$\tilde{\mathcal{N}}(x;\theta) = (x - \tau\mathcal{N}(\nabla E(x);\theta)) - \tau\mathcal{N}(\quad (x - \tau\mathcal{N}(\nabla E(x);\theta)) \quad ;\theta)$$

Common approaches would use at least 10 iterations, and possibly initialize the network to be identical. Note the ResNet-structure!

Network compression

For reducing the size and computation burdon of networks, we learned about

- Factorizing convolutions and group convolutions



- Reducing the number of channels
- Reducing the resolution of the input
- Quantizing weights

Any questions about anything?