Visual AI

CPSC 533R – 2020/2021 Term 1

Lecture 3. Deep Nets and their optimization

Helge Rhodin

Lecture overview

- Convolutions

- Optimizer

- Automatic differentiation and backprop

- Input and output normalization
- Vanishing gradient

- Deep network architectures

Changes due to high number of students

- Course projects in groups of 2-3 students
- The reading sessions will have 2-3 presentations
  - one student moderator per presentation
- Submit paper review the day before at 2pm at 11:59 pm
  - these will be forwarded to the moderator

Recap: Presentation and Project

- 1x Paper presentation (Weeks 3 – 12)
  - Presentation, once per student (25% of points) (15 min + ~15 min discussion)
    - **Arrange for a meeting with TA**
    - Pre-recorded or live, it is your choice
  - Read and review one out of the two papers presented per session (10% of points)

- 1x Project (40 % of points)
  - Project pitch (3 min, week 6&7)
  - Project presentation (10 min, week 13&14)
  - Project report (6 pages, Dec 14)

Detailed info on the website!

*The **presentation slides must be handed in and be discussed with the tutor latest by two working days before the presentation.** It is **your responsibility** to set up a meeting (~30 min duration) with the TA three days in advance. This session is to your own benefit and will not be graded. Submit your final slides on Canvas, the Slide upload/Presentation Assignment.*

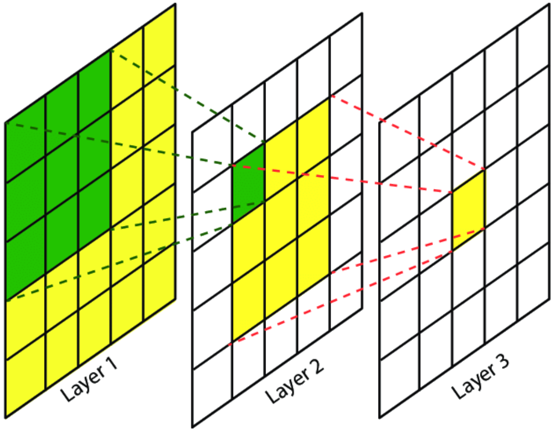
What is the benefit of **deep** neural networks?

In principle a fully-connected network is sufficient

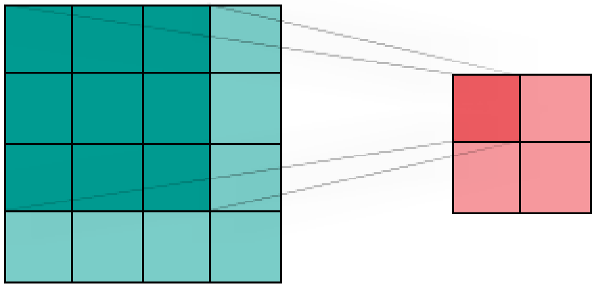
- universal approximator
- but hard to train!

Empirical observation (for image processing)

- convolution and pooling operations act as a strong prior
  - locality
  - translational invariance
- a deep network increases the receptive field
  - such large context helps
- many simple operations work better than a monolithic one
  - separable conv., group conv., 3x3 instead of 5x5, …
    (this lecture)



Convolutional neural network layer



Convolution

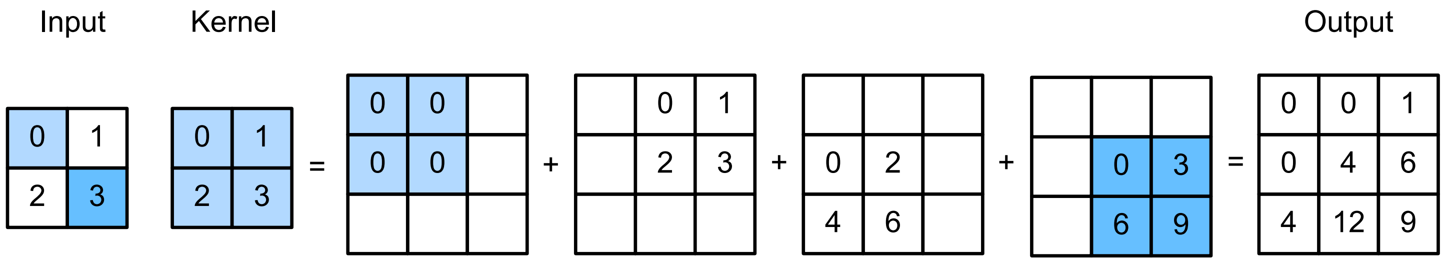
- **Local** linear transformation + activation function
  - sliding window, kernel size=size of window
    - the same kernel is applied repeatedly
      - stride=1: at every possible location
      - stride=S: slide kernel with step length S (jump every other pixel)
    - padding=P: add default values at the boundary, with width P
- Multiple layers form a convolutional neural network (CNN)

Transformation of input and output by convolutions

- output size = (input size + 2*padding – kernel size + stride)/stride
  - e.g., a 3x3 kernel that preserves size: W + 2*1 – 3 + 1 = W
  - e.g., a 4x4 kernel that reduces size by factor two: (W + 2*1 – 4 + 2)/2 = W/2
- holds per dimension, i.e., 1D, 2D and 3D convolutions

Transposed convolution

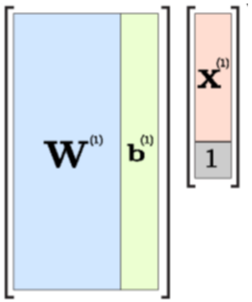
Example: 2D transposed convolution with 3x3 kernel



[https://d2l.ai/chapter_computer-vision/tranposed-conv.html]

Transposed what?

- *Express convolution as linear matrix and transpose it*
    - *special case of a linear/fully-connected layer*



Convolution as matrix multiplication

$$\begin{bmatrix} 0, 1, 0, 2, 3, 0, 0, 0, 0 \\ 0, 0, 1, 0, 2, 3, 0, 0, 0 \\ 0, 0, 0, 0, 1, 0, 2, 3, 0 \\ 0, 0, 0, 0, 0, 1, 0, 2, 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 0\cdot1 + 1\cdot2 + 2\cdot4 + 3\cdot5 \\ 0\cdot2 + 1\cdot3 + 2\cdot5 + 3\cdot6 \\ 0\cdot4 + 1\cdot5 + 2\cdot7 + 3\cdot8 \\ 0\cdot5 + 1\cdot6 + 2\cdot8 + 3\cdot9 \end{bmatrix} = \begin{bmatrix} 25 \\ 31 \\ 43 \\ 49 \end{bmatrix}$$

Convolution with 2x2 kernel
as a linear mapping

Flattened 3x3 input image

$$\underset{\text{Input}}{\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}} * \underset{\text{Kernel}}{\begin{bmatrix} 0, 1 \\ 2, 3 \end{bmatrix}} = \underset{\text{Output}}{\begin{bmatrix} 25, 31 \\ 43, 49 \end{bmatrix}}$$

Convolution operator

Convolution as matrix multiplication (details)

What about horizontal striding?
Skip every second row

Larger kernel?
Add non-zeroes here
(5 values for 3x3 from 2x2)

Padding?
Insert row with some kernel elements missing

Larger input height?
Insert more rows here.

$$\begin{bmatrix} 0, 1, 0, 2, 3, 0, 0, 0, 0 \\ 0, 0, 1, 0, 2, 3, 0, 0, 0 \\ 0, 0, 0, 0, 1, 0, 2, 3, 0 \\ 0, 0, 0, 0, 0, 1, 0, 2, 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}$$

What about vertical striding?
Skip block of rows

Larger input width?
insert zeroes here.

Convolution

$$\underset{\text{Input}}{\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}} * \underset{\text{Kernel}}{\begin{bmatrix} 0, 1 \\ 2, 3 \end{bmatrix}}$$

Transposed convolution as matrix multiplication

Transposed convolution

$$\begin{bmatrix} 25, 31 \\ 43, 49 \end{bmatrix} *^{T} \begin{bmatrix} 0, 1 \\ 2, 3 \end{bmatrix}$$

Input Kernel

$$= \begin{bmatrix} 0, & 25, & 31 \\ 50, & 180, & 142 \\ 86, & 227, & 147 \end{bmatrix}$$

Output

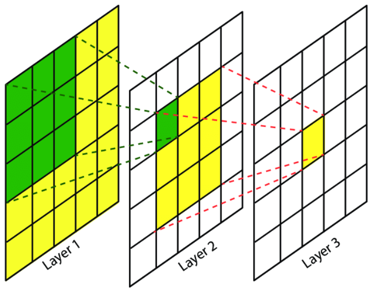
$$\begin{bmatrix} 0, & 0, & 0, & 0, \\ 1, & 0, & 0, & 0, \\ 0, & 1, & 0, & 0, \\ 2, & 0, & 0, & 0, \\ 3, & 2, & 1, & 0, \\ 0, & 3, & 0, & 1, \\ 0, & 0, & 2, & 0, \\ 0, & 0, & 3, & 2, \\ 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 25 \\ 31 \\ 43 \\ 49 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \cdot 25 \\ 1 \cdot 31 \\ 2 \cdot 25 \\ 3 \cdot 25 + 2 \cdot 31 + 1 \cdot 43 \\ 3 \cdot 31 + 1 \cdot 49 \\ 2 \cdot 43 \\ 3 \cdot 43 + 2 \cdot 49 \\ 3 \cdot 49 \end{bmatrix} = \begin{bmatrix} 0 \\ 25 \\ 31 \\ 50 \\ 180 \\ 142 \\ 86 \\ 227 \\ 147 \end{bmatrix}$$

Transposed weight matrix

Feature map size after convolutional kernels

Transformation of input and output by **convolutions**

- output size = (input size + 2*padding – kernel size + stride)/stride
  - e.g., a 3x3 kernel that preserves size: W + 2*1 – 3 + 1 = W
  - e.g., a 4x4 kernel that reduces size by factor two: (W + 2*1 – 4 + 2)/2 = W/2
- holds per dimension, i.e., 1D, 2D and 3D convolutions



Transformation of input and output by **transposed convolutions** (aka. deconvolution)

- output size = input size * stride - stride + kernel size - 2*padding
  - it has exactly the opposite effect of convolution
  - e.g., a 3x3 kernel that preserves size: W - 1 + 3 - 2*1 = W
  - e.g., a 4x4 kernel that **increases** size by **factor** two: W*2 + 2*1 – 4 + 2 = W*2
  - e.g., a 3x3 kernel that **increases** size by two elements: W - 1 + 3 - 2*0 = W + 2

Optimizers

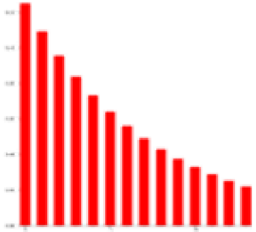
Stochastic Gradient Descent

- Gradient descent on randomized mini batches (with learning rate alpha)

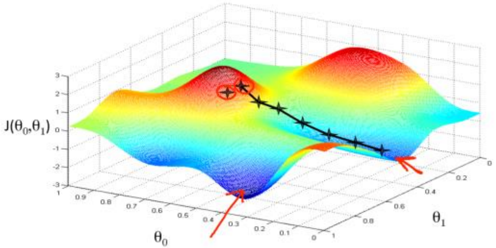
$$\theta_t = \theta_{t-1} - \alpha \sum_{i=1}^{n} \nabla E_i(\theta)/n,$$

Adam

- Momentum-based (continue with larger steps if the previous steps point in the same direction)
- Damp step-length if direction changes often (second moment is high)
- Uses exponential moving average (EMA)

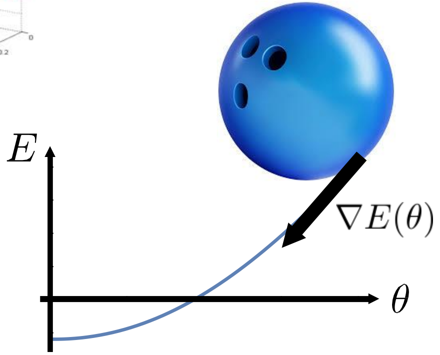


$$\bar{y}_t = \begin{cases} y_1, & t = 1 \\ \beta \cdot \bar{y}_{t-1} + (1-\beta) \cdot y_t, & t > 1 \end{cases}$$



https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

momentum



$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$$

Adam update rule

Smaller batch size can be better; it induces more noise!

Adam and co.

- Adam is my current favorite
    - Not that sensitive to learning rate
    - No scheduler necessary
    - Intuitive motivation

  Disadvantage: Properly tuned SGD can be more accurate
- Recent alternative
    - Learning with Random Learning Rates [Blier et al.,]
        - give each neuron a different learning rate
        - those with inappropriate rates will die (constant output for all feasible input values)
        - parameter free, more stable training

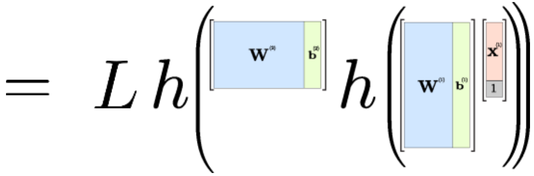
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

[Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization. ICLR 2015]

Automatic differentiation and backpropagation

Forward pass

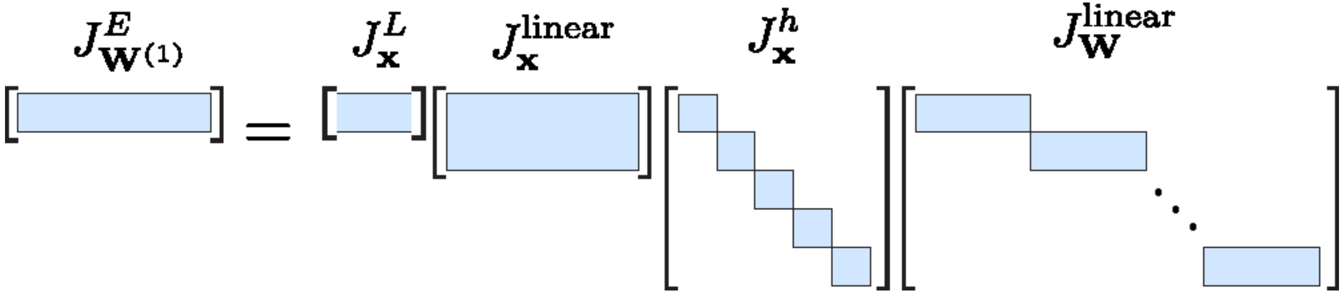
$$L(h(\text{linear}(h(\text{linear}(x, W^{(1)})), W^{(2)})))$$



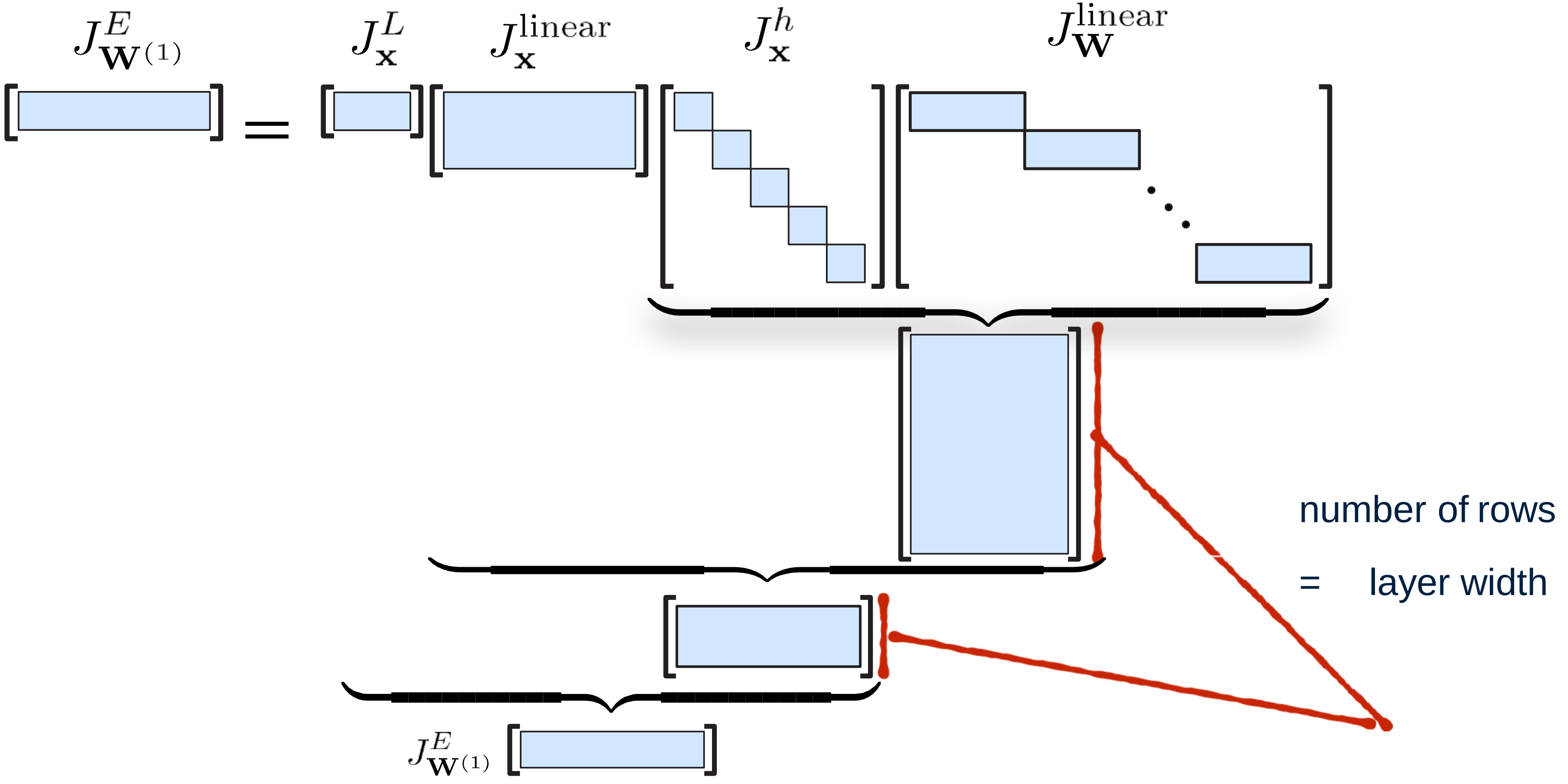
$$\mathbf{J}_x^f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jocobian matrix

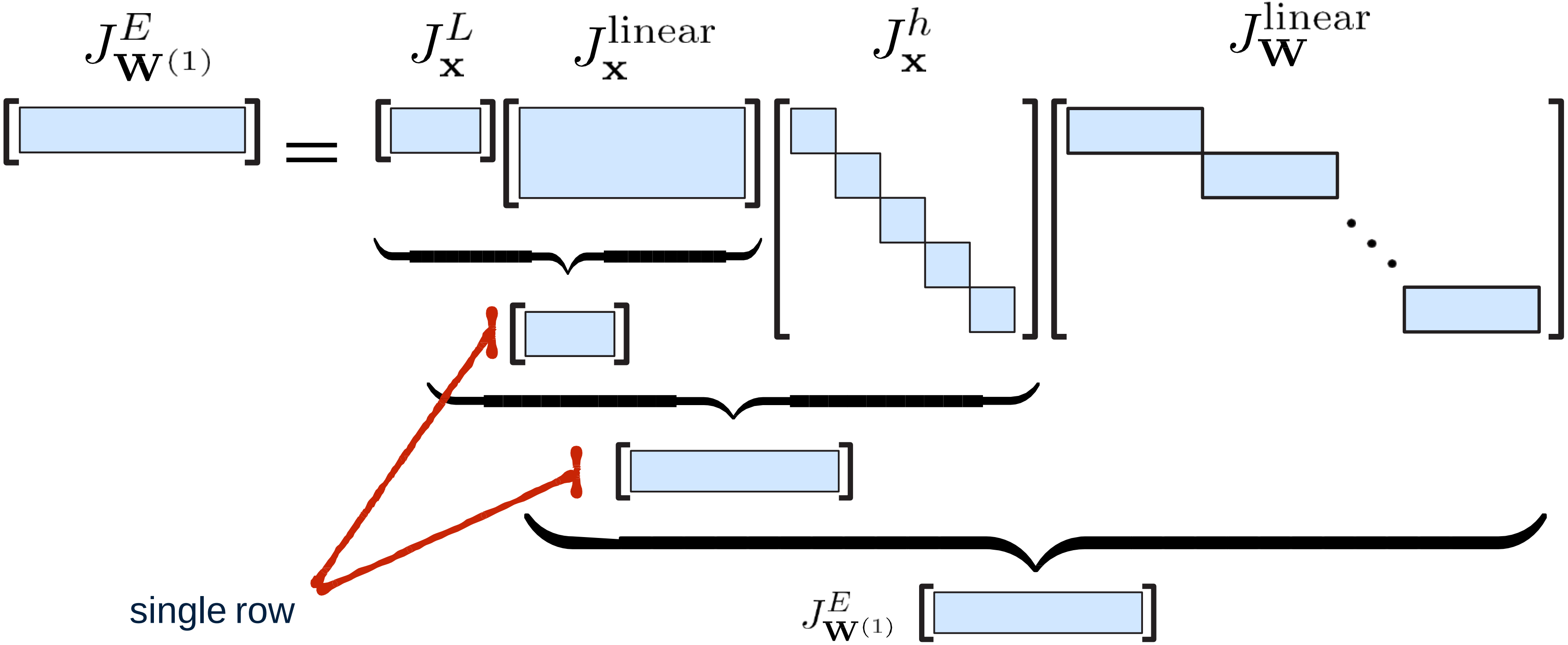
Backwards pass to $W^{(1)}$



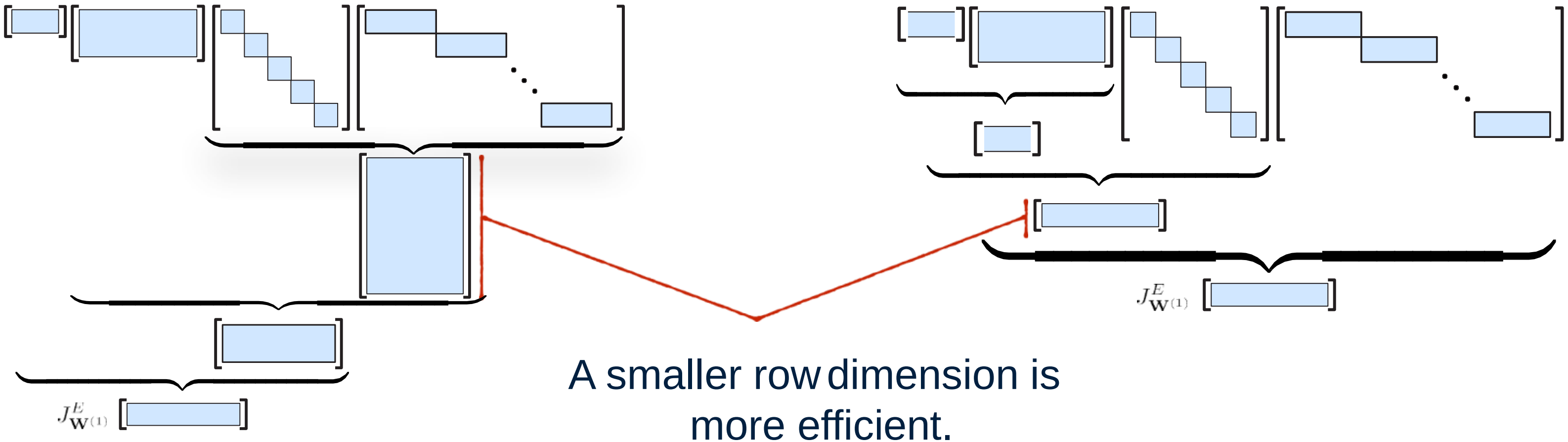
Forward propagation



Reverse mode - backpropagation



Forward vs. reverse mode



A smaller row dimension is more efficient.

Forward accumulation is more efficient for functions that have more outputs than inputs.

Reverse accumulation is more efficient for functions that have more inputs than outputs.

Backpropagation — a special case

Creating the Jacobian matrices is expensive. Instead, matrix products can be simplified.

Backpropagation through activationfunction

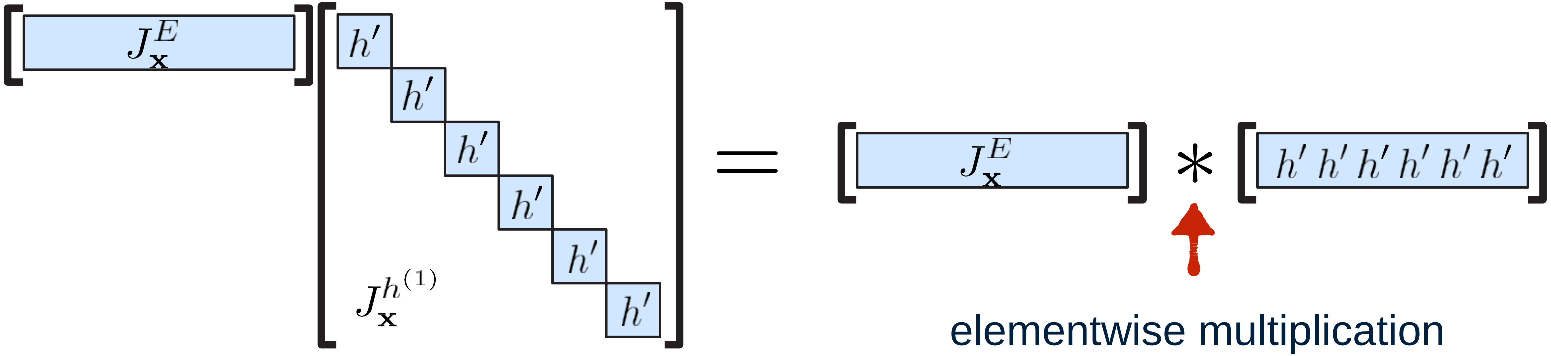
$$\begin{bmatrix} J^E_{\mathbf{x}} \end{bmatrix} \underbrace{\begin{bmatrix} h' & & & & & \\ & h' & & & & \\ & & h' & & & \\ & & & h' & & \\ & & & & h' & \\ & & & & & h' \end{bmatrix}}_{J^{h^{(1)}}_{\mathbf{x}}} = \begin{bmatrix} J^E_{\mathbf{x}} \end{bmatrix} * \begin{bmatrix} h' \; h' \; h' \; h' \; h' \; h' \end{bmatrix}$$

element-wise
multiplication

More optimizations

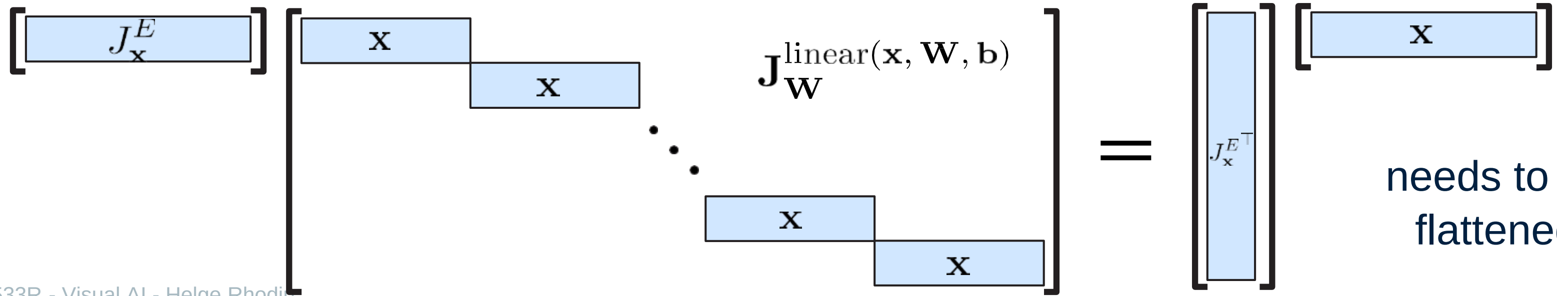
Creating the Jacobian matrices is expensive. Instead, matrix products can be simplified.

Backpropagation through activationfunction



elementwise multiplication

Backpropagation through linearlayer

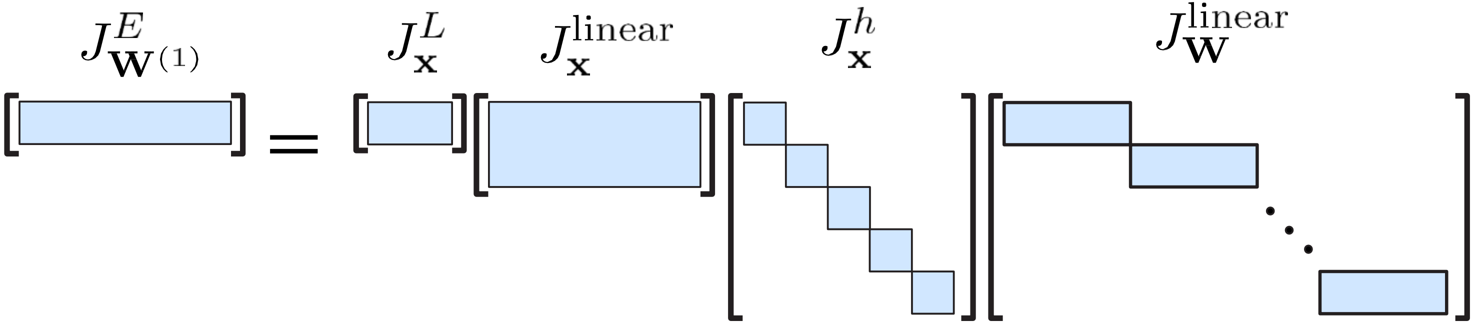


needs to be flattened

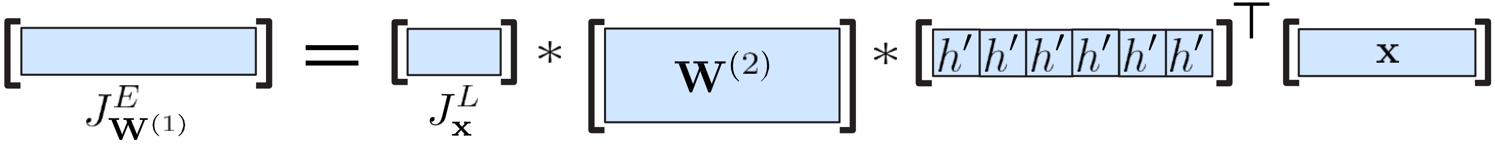
Advantage of backpropagation

Backpropagation is a form of reverse automatic differentiation, where the Jacobi matrix is not explicitly computed.
The gradient is propagated by simpler equivalent operations.

Jacobian formulation



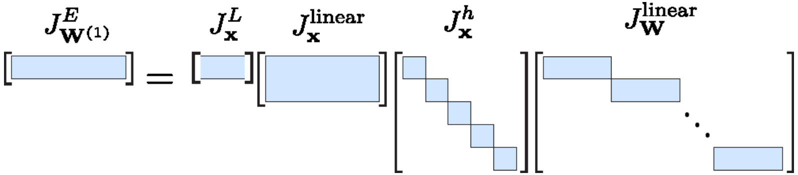
Compact backpropagation



Vanishing gradients problem

The objective function

$$O(x,y) = L(h(\mathrm{l}(h(\mathrm{l}(x, W^{(1)})), W^{(2)})), y)$$

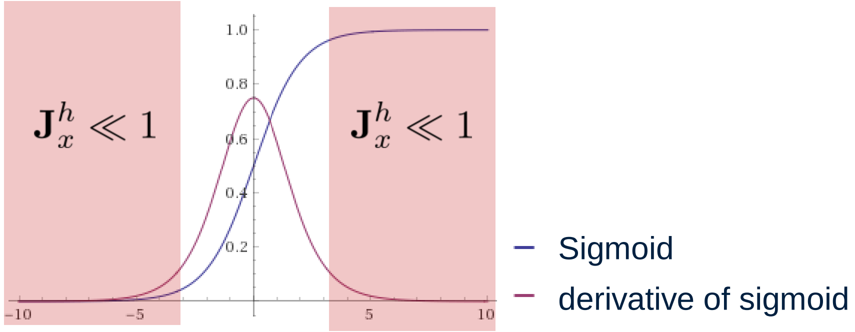


The gradient of O with respect to $W^{(2)}$

$$O(x,y) = \mathbf{J}^L_x \mathbf{J}^h_x \mathbf{J}^{\mathrm{l}}_{W^{(2)}}$$

The gradient of O with respect to $W^{(1)}$

$$O(x,y) = \mathbf{J}^L_x \mathbf{J}^h_x \mathbf{J}^{\mathrm{l}}_x \mathbf{J}^h_x \mathbf{J}^{\mathrm{l}}_{W^{(1)}}$$



The gradient vanishes exponentially with respect to the number of layers if $\mathbf{J}^h_x < 1$

*Use ReLU rather than sigmoid in **deep** neural networks!*

Interactive session

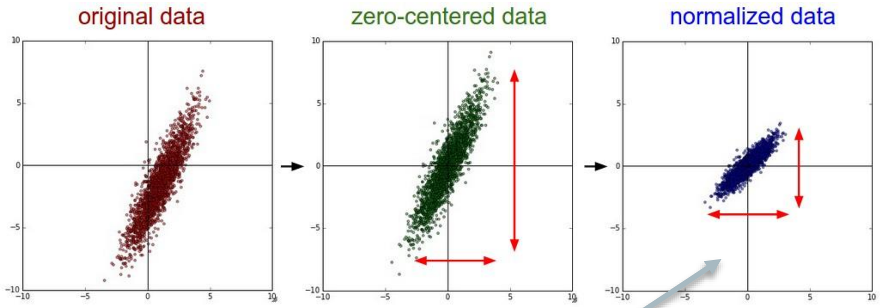
Monitoring feature activations

Input and output normalization

Goal: Normalize input and output variables to have μ=0 and σ=1

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sigma}$$

- For an image, normalize each pixel by the std and mean color (averaged over the **training** set)



http://cs231n.github.io/neural-networks-2/

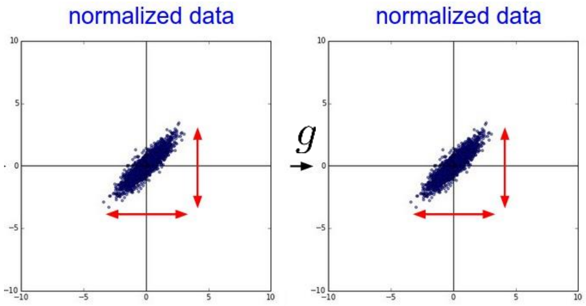
Related to data whitening

- whitening transforms a random vector to have zero mean and unit diagonal covariance
- by contrast, the default normalization for deep learning is element wise, neglecting dependency
  - the resulting covariance is not diagonal!

Neural network initialization

Goal: preserve mean and variance through the network

- Assume that the input is a random variable with var(x) = 1 and mean(x) = 0
- Derive the function g that describes the change of variance and mean between layers $\begin{pmatrix}\tilde{\mu}\\ \tilde{\nu}\end{pmatrix} = g\begin{pmatrix}\mu\\ \nu\end{pmatrix}$



Initialize the neural network weights (weights of linear layers) such that g is the identity function

- For the linear neuron with K incoming neurons

$$\begin{aligned}\mathrm{Var}(\mathbf{w}\cdot\mathbf{x}) &= \sum_{i=1}^{K}\mathrm{Var}(\mathbf{w}_i)\mathrm{Var}(\mathbf{x})\\ &= K\mathrm{Var}(\mathbf{w}_i)\mathrm{Var}(\mathbf{x}) \qquad \Rightarrow \quad \mathrm{Var}(\mathbf{w}) = \frac{1}{K}\\ &= K\mathrm{Var}(\mathbf{w}_i)\end{aligned}$$

$$\mathrm{Var}(x+y) = \mathrm{Var}(x) + \mathrm{Var}(y)$$

$$\mathrm{Var}(xy) = \mathrm{Var}(x)\mathrm{Var}(y)$$
$$\text{for variables with zero mean}$$

(simple) Xavier Initialization: Initialize with samples from a (Gaussian) distribution with std = $\sqrt{1/K}$

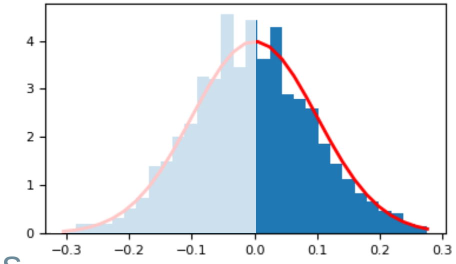
Neural network initialization II

The activation function changes the distribution

- the mean of ReLU(x) is nonzero
  - hence, the *variance of product* equation does not apply
  - instead, it holds

$$\mathrm{Var}(xy) = \mathrm{Var}(x)E(y^2)$$
for x zero mean and y arbitrary

$$\cancel{\mathrm{Var}(xy) = \mathrm{Var}(x)\mathrm{Var}(y)}$$
~~for variables with zero mean~~



  - and, assuming that y is from a symmetric distribution,

$$E(ReLU(\mathbf{y})^2) = \frac{1}{2}\mathrm{Var}(\mathbf{y})$$

  - the variance transformation of linear layer + activation becomes

$$\mathrm{Var}(\mathbf{w} \cdot \mathrm{ReLU}(\mathbf{x})) = \frac{K}{2}\mathrm{Var}(\mathbf{w}_i) \qquad \Rightarrow \qquad \mathrm{Var}(\mathbf{w}) = \frac{2}{K}$$

Kaiming He Initialization: Initialize with samples from a (Gaussian) distribution with std = $\sqrt{2/K}$

Other initializations

SIREN network

- sin(x) activation
  - requires different initialization as with ReLU
  - sin(x) works poorly when not adapting init

Xavier Initialization / ‘Normalized initialization’ by Glorot and Bengio

- for hyperbolic tangent and softsign
  - fan-in: n_i the dimension of the previous layer
  - fan-out: n_i+1 the output dimension of the layer
- motivation for fan-out: normalization of the gradients
  - same derivation as for the forward pass,
    but going backwards through the layers

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

Batch normalization

[Batch Normalization: Accelerating Deep Network Training ~~by Reducing Internal Covariate Shift~~]

- Normalize after each linear + activation function
  - normalize across minibatch, to have μ=0 and σ=1

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

  - Strict normalization reduces performance, hence, add back a learnable offset and scale

$$y_i \leftarrow \gamma \widehat{x}_i + \beta$$

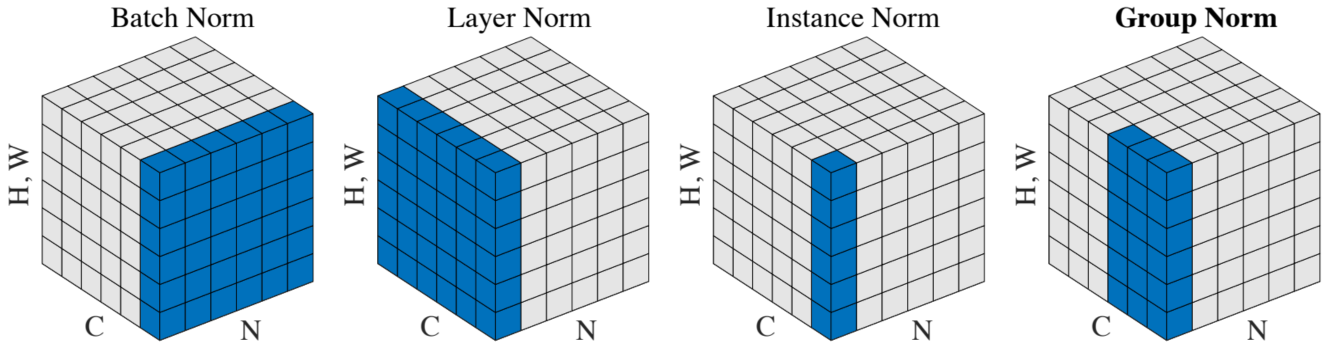
  - What if we only have a single image at inference time?
    - Re-apply mean and variance recorded during training (using exponential moving average)

Batch normalization effect and variants

What is the benefit of first normalizing and then ‘denormalizing’?

- noise from other images regularizes
- it separates learning of the variance (scale) and bias (offset) from the values itself
- Empirical: training deeper networks, with sigmoid activation, higher learning rate, and faster convergence

Variants normalize over different slices of the feature tensor:



[Wu and He. Group Normalization]

Regularization

Dropout

- randomly zero out activations
- re-weight the non-zero ones to maintain the distribution of the unmodified activations
    - induced noise reduces overfitting

Weight decay

$$\tilde{\mathbf{w}} = (1 - \tau)\mathbf{w} \text{ with } \tau \text{ small})$$

Weight decay and square prior are equivalent under certain conditions (vanilla SGD without momentum)

Prior on neural network weights

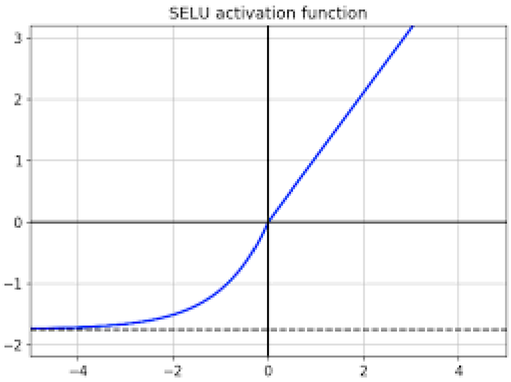
$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

Self-normalizing neural networks

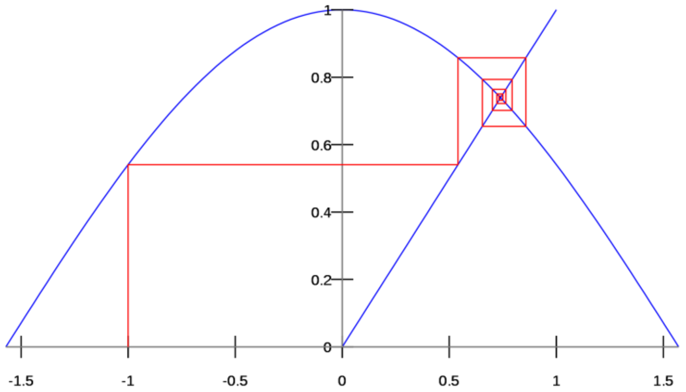
Self-normalizing Neural Networks

[Klambauer et al.]

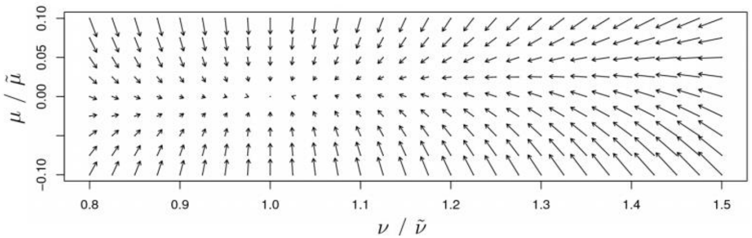
- fixed point enforced by choice of activation function (SELUs)
- stable and attracting fixed point for the function g that maps mean and variance from one layer to the next
- Possibility to train deep fully connected NNs



$$\mathrm{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leqslant 0 \end{cases}$$



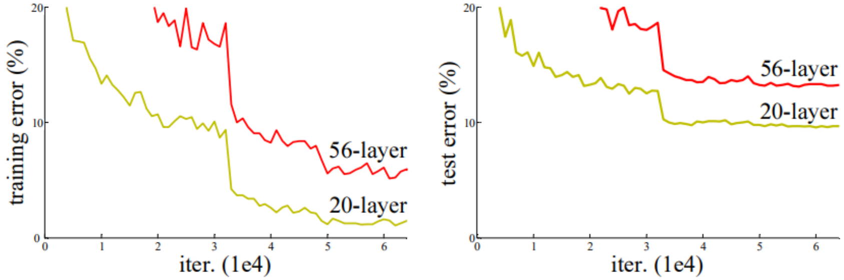
Fixed point iterations for cos(x)



Mapping of the function g towards mu=0 and v=1

Residual networks and skip connections

- Deep networks are hard to train



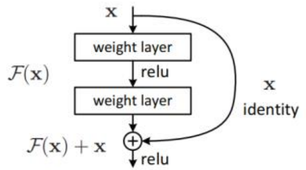


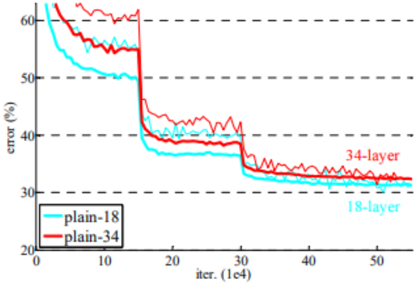
Figure 2. Residual learning: a building block.

- Residual blocks with shortcut/skip connections

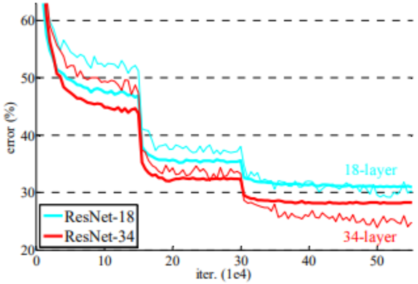
$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

  - no extra parameters
  - enables training of deep neural networks

Image net training

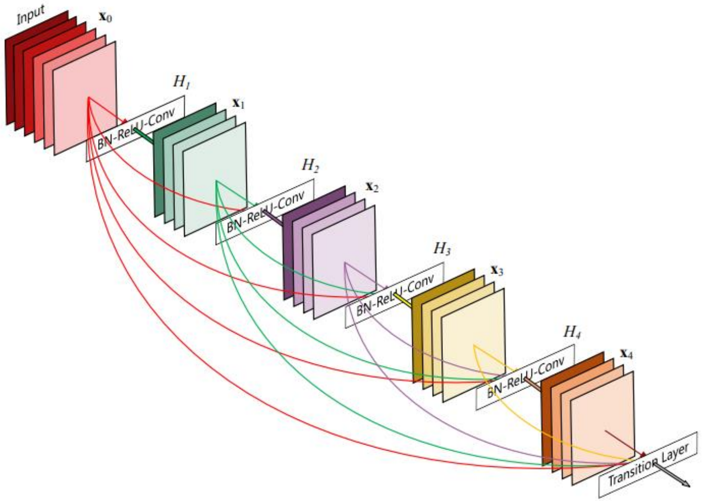


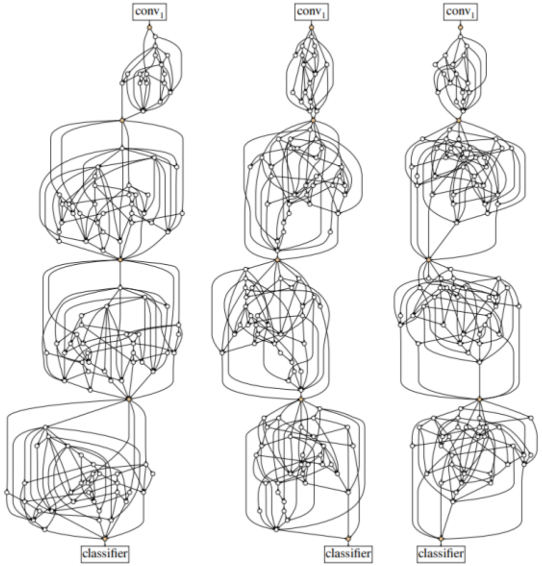
Baseline network



ResNet version

Other network architectures



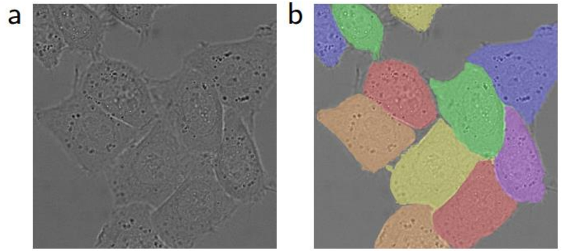


DenseNet
(skip connection to all future layers)

Randomly wired networks
(search for best wiring among candidates)

U-Net architecture

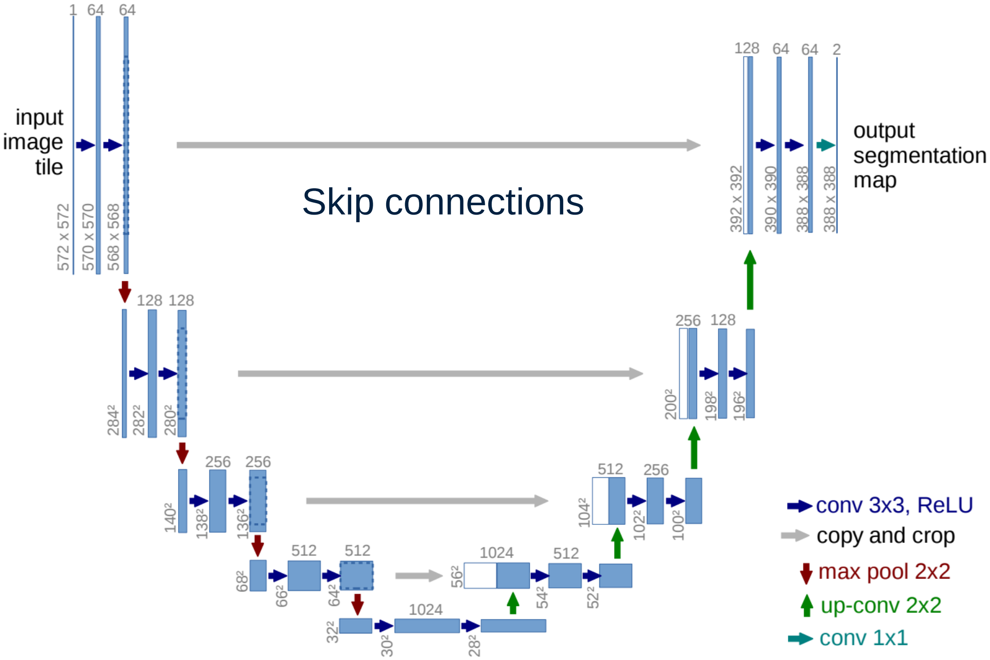
- Similar input and output resolution
- A global encoding is learned by down sampling (to 32 x 32 px)
- Progressive increase of channels maintains throughput / capacity
- Skip connections preserve details



Encoder
(spatial down sampling)

Decoder
(spatial up sampling)

Skip connections

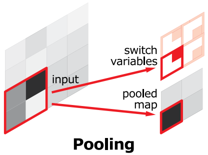


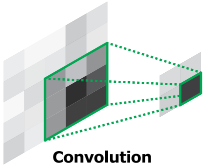
[U-Net: Convolutional Networks for Biomedical Image Segmentation]

Spatial down and up-sampling

Spatial down-sampling

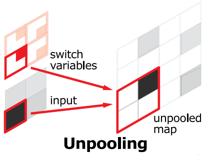
- max-pooling
- average pooling
- convolution with stride

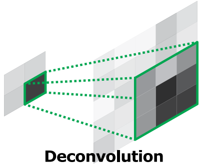




Spatial up-sampling

- max-unpooling
- (bilinear) interpolation
- deconvolution





[Learning Deconvolution Network for Semantic Segmentation]