Visual AI

CPSC 532R/533R – 2019/2020 Term 2

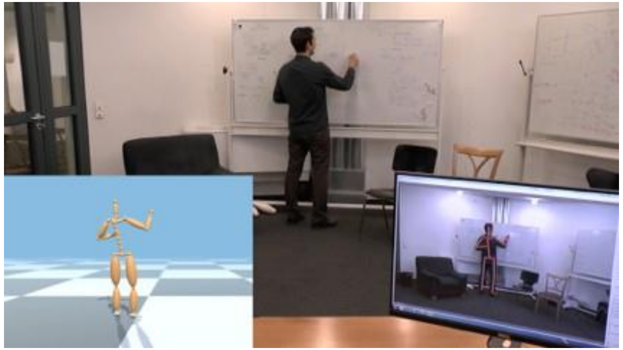
Lecture 3. Network architectures for image processing (and their optimization)

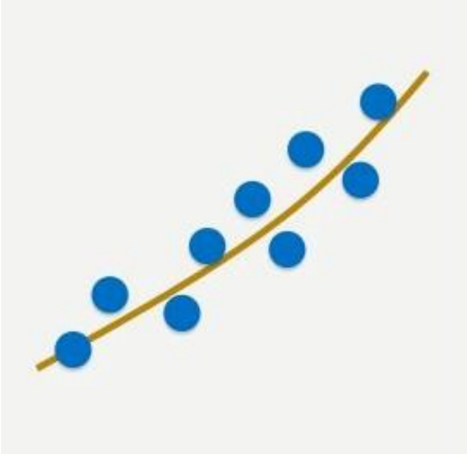
Helge Rhodin

Classification vs. regression

Classification

Regression

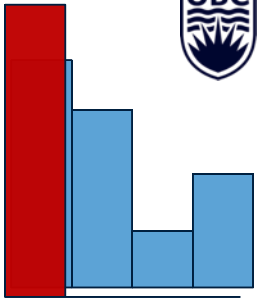




Cross-entropy loss / Cross-entropy criterion

Negative Log Likelihood (NLL) formulation for a *one-hot vector,* target class c

$$l_{\mathrm{NLL}}(x, c) = -\log(f_{[c]}(x))$$



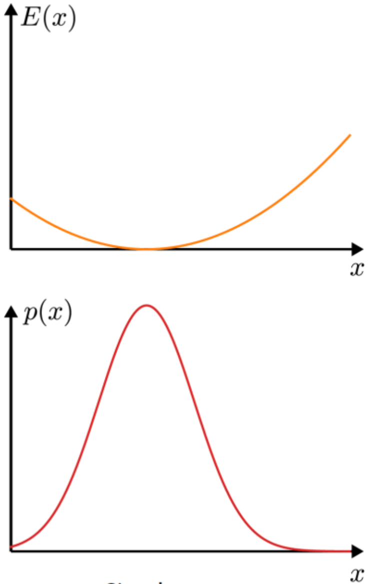
one-hot vector case

Regression revisited

Many loss functions are –log of probability distributions

Error functions
$E(x)$

Distributions
$p(x) = \exp(-E(x))$



Simple case

$$x^2$$

Mean squared error (MSE)

$$|x|$$

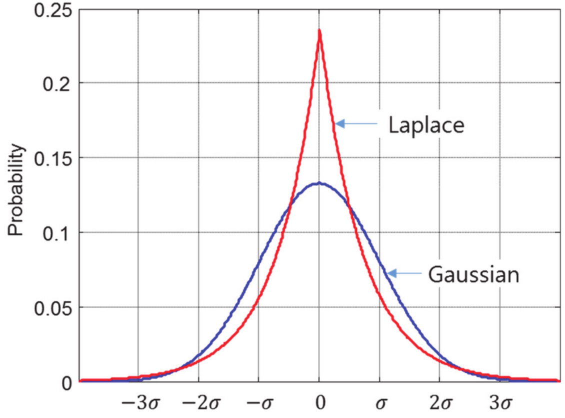
Mean absolute error (MAE)

$$\exp(-x^2)$$

Gaussian distribution

$$\exp(-|x|)$$

Laplace distribution



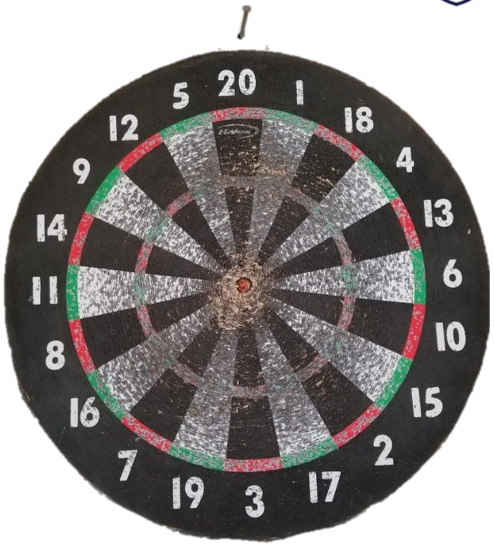
Recap done

Density networks

Assumption: Data is distributed according to a random process

$$y \sim N(F(x), \sigma), \text{ with mean } F(x) \text{ and standard deviation } \sigma$$

Given N samples $(x_i, y_i)_{i=1}^{N}$

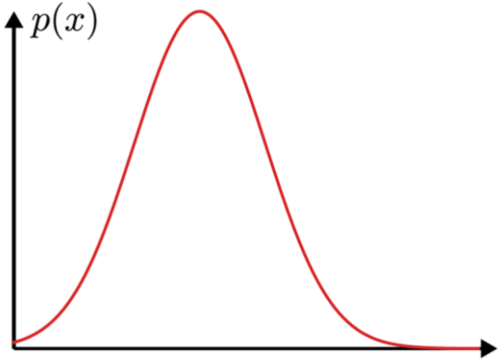


Goal: Find that function that maximizes the likelihood of the samples (x_i, y_i)

$$L = N(y|f_\theta(x), \sigma), \text{ with mean } f_\theta(x) \text{ a neural network with parameters } \theta$$

Solution: minimize the negative log-likelihood (here mean squared error)

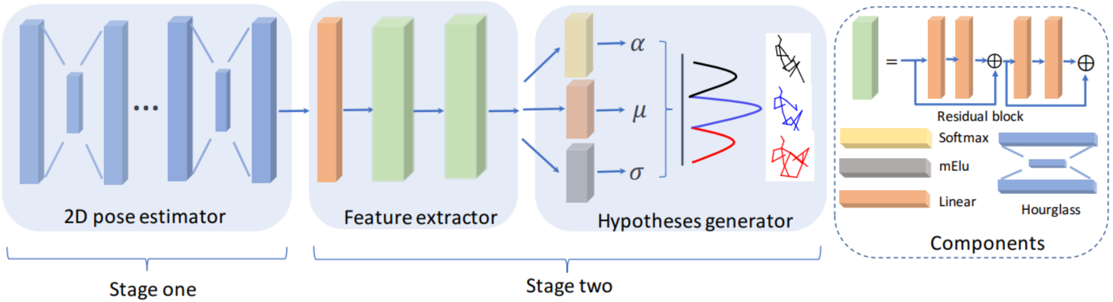
$$E = \frac{1}{2\sigma^2}(y - f_\theta(x))^2$$

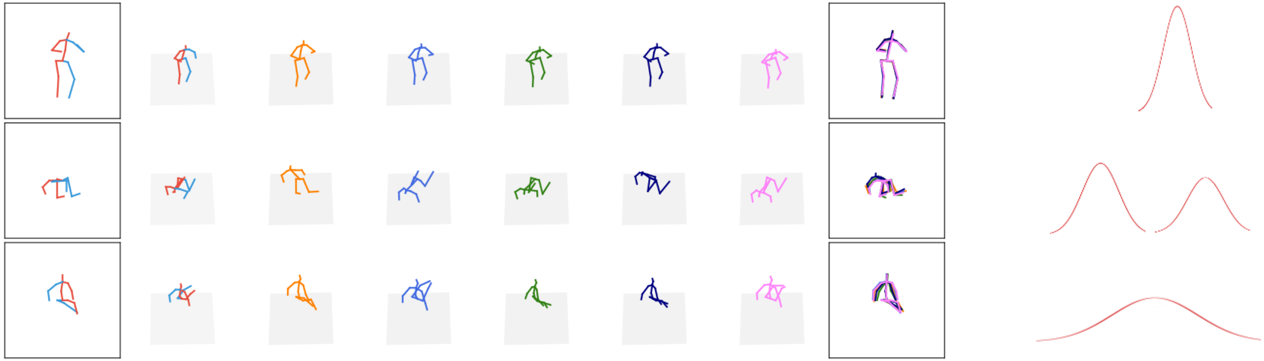


Density Networks: Predict the mean μ **and** standard deviation ϭ

$$\sigma, \mu = f_\theta(x) \qquad L = N(y|\mu, \sigma)$$

Generating Multiple Hypotheses for 3D Human Pose Estimation with Mixture Density Networks





Input 2D pose

3D proposals

Projected 3D proposals

Shape of distribution

Separable objective and mini batches

Separable objective over independent samples x,y

$$E(D,\theta) = \sum_{(\mathbf{x}^{(i)},y^{(i)})\in D} l(\quad (\mathbf{x}^{(i)},\theta), y^{(i)})$$

Evaluated over mini batches of size N

Stored as tensor, e.g.,

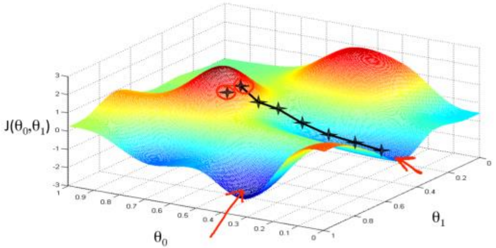
- dim 0: N, number of images in a batch
- dim 1: C, number of channels
- dim 2: H, height of the feature map
- dim 3: W, width of the feature map

Optimizers

Stochastic Gradient Descent

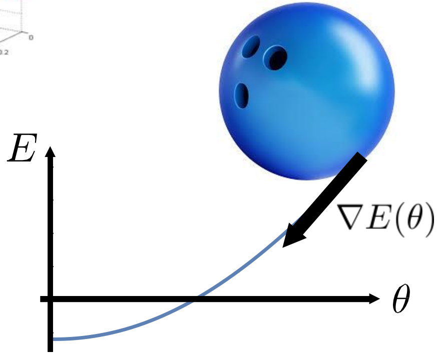
- Gradient descent on randomized mini batches (with learning rate alpha)

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



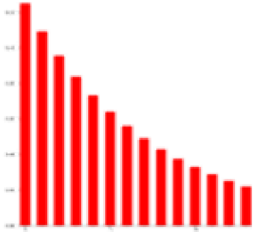
https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

momentum



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)



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

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

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

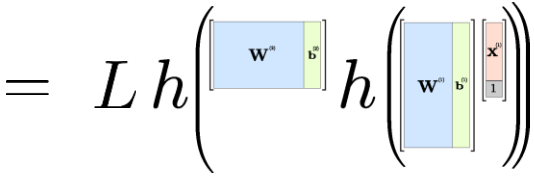
  Disadvantage: Properly tuned SGD can be more accurate
- Recent alternative
  - Learning with Random Learning Rates [Blier et al.,]
    - give each neuron a different learning rate
    - those with inappropriate rates will die (constant output for all feasible input values)
    - parameter free, more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

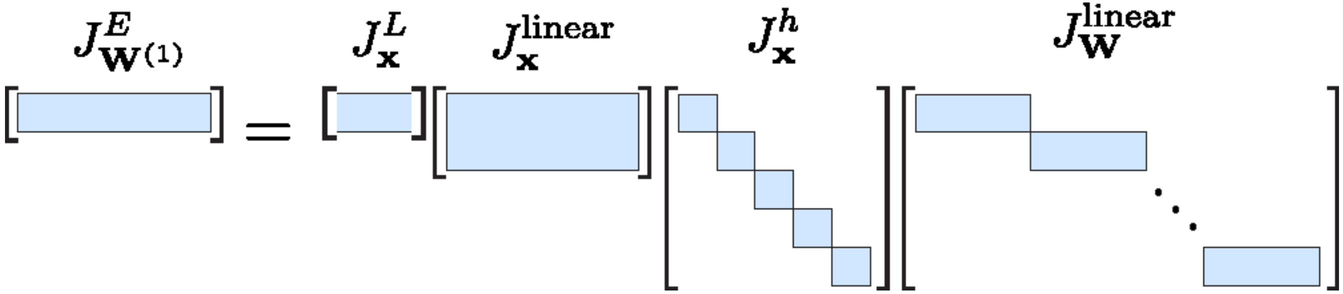
Forward pass $L(h(\text{linear}(h(\text{linear}(x, W^{(1)})), W^{(2)})))$



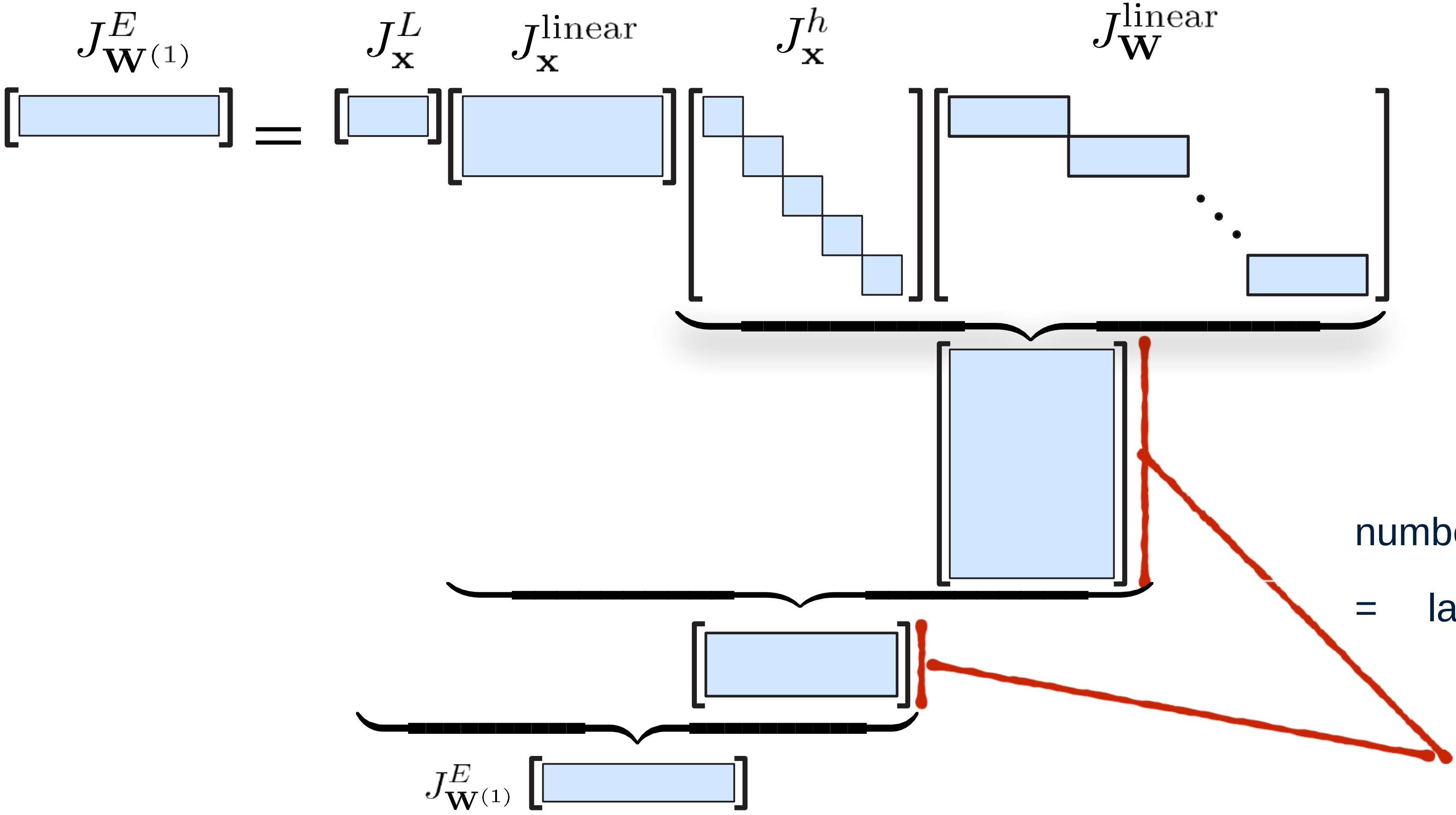
$$\mathbf{J}_x^f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jocobian matrix

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



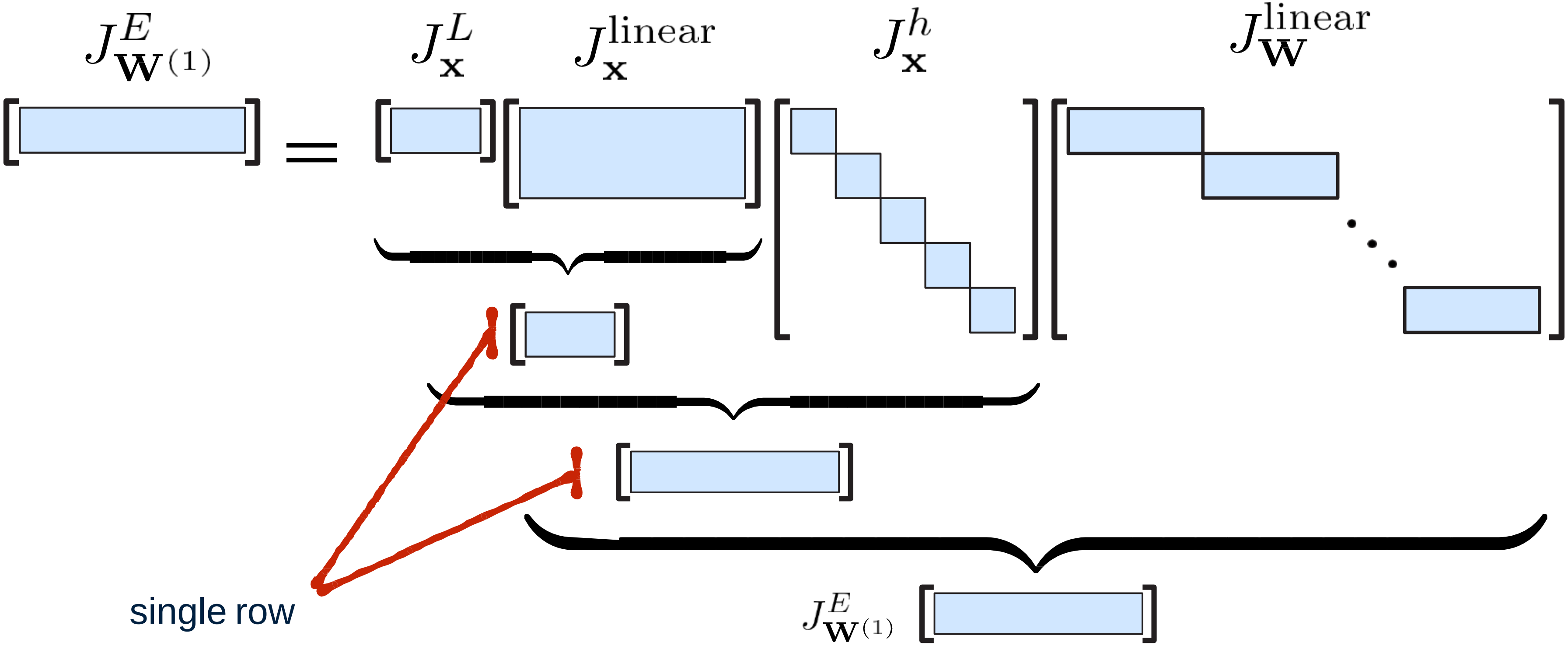
Forward propagation



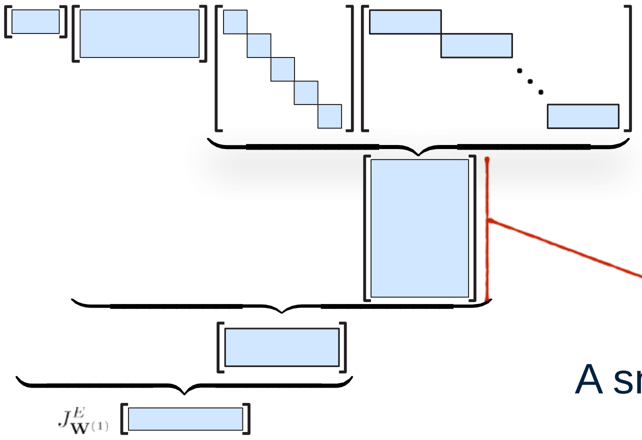
number of rows

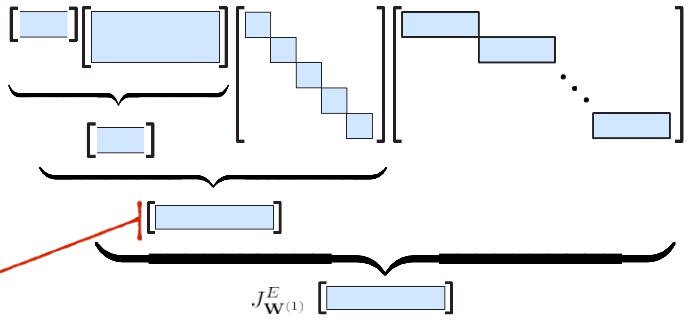
= layer width

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

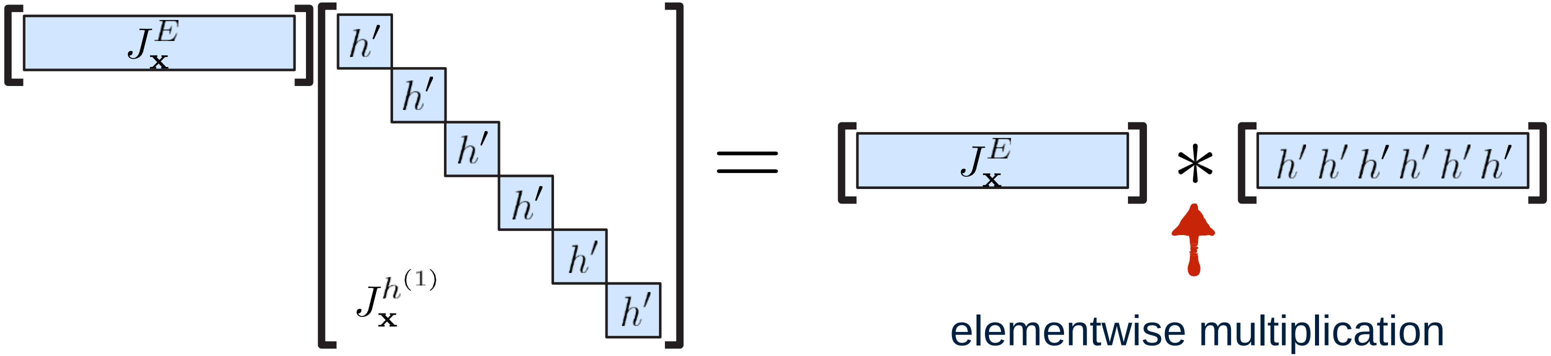
$$
\begin{bmatrix} \quad J^E_{\mathbf{x}} \quad \end{bmatrix}
\begin{bmatrix}
h' & & & & & \\
& h' & & & & \\
& & h' & & & \\
& & & h' & & \\
& & & & h' & \\
J^{h^{(1)}}_{\mathbf{x}} & & & & & h'
\end{bmatrix}
=
\begin{bmatrix} \quad J^E_{\mathbf{x}} \quad \end{bmatrix}
*
\begin{bmatrix} h' \; h' \;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

element-wise multiplication

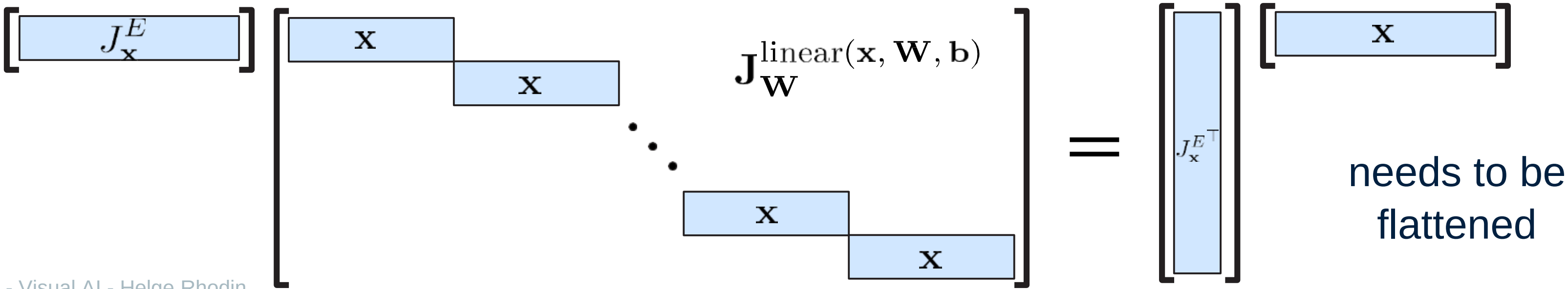
More optimizations

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

Backpropagation through activationfunction



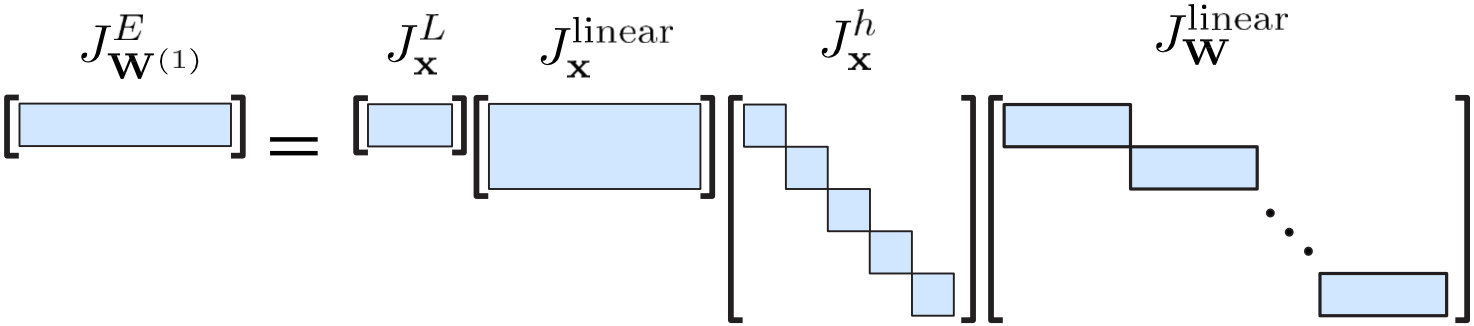
Backpropagation through linearlayer



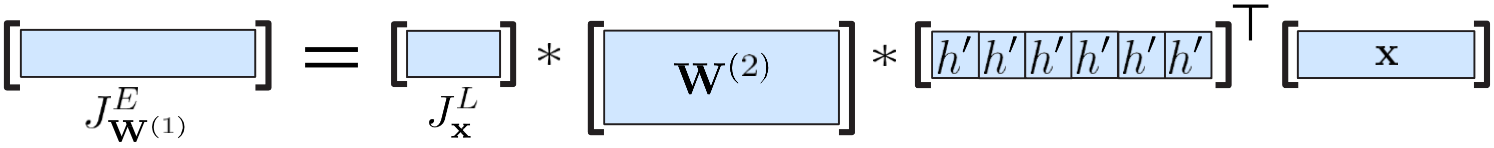
Advantage of backpropagation

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



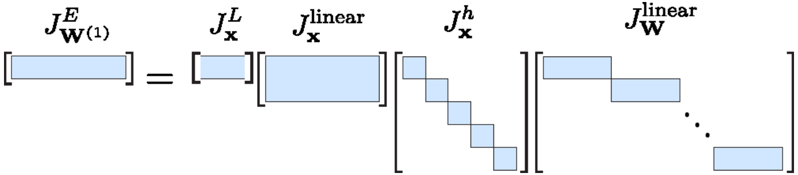
Compact backpropagation



Vanishing gradients problem

The objective function

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

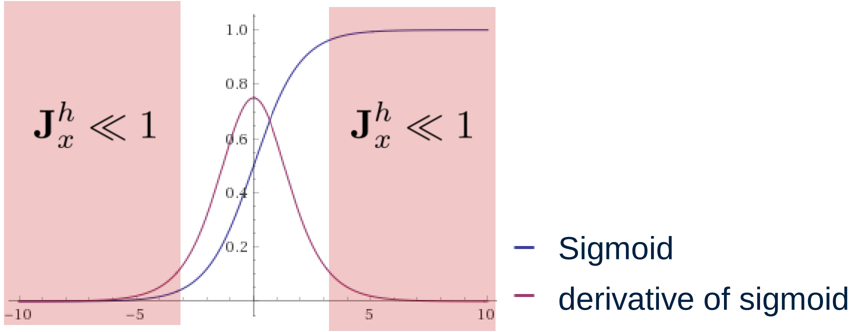


The gradient of O with respect to $W^{(2)}$

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

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!*

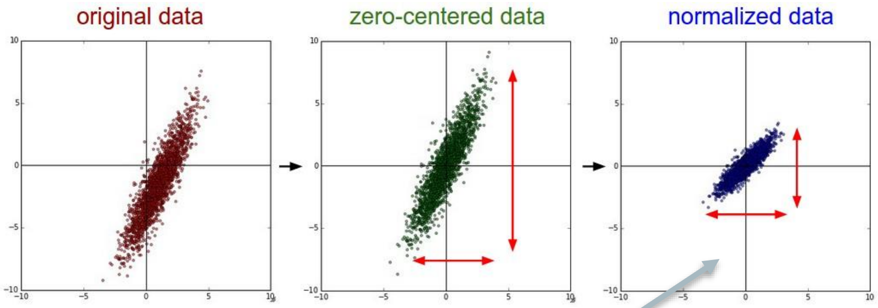
Mini break

Find a team partner for your course project

Input and output normalization

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

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



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

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

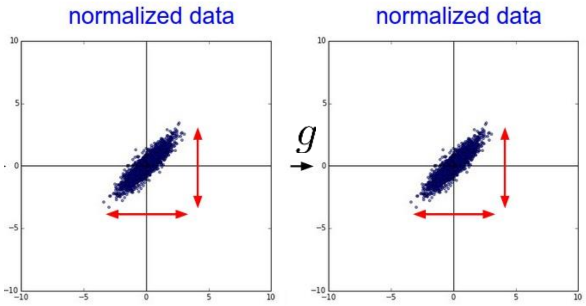
Related to data whitening

- 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

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

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

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

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

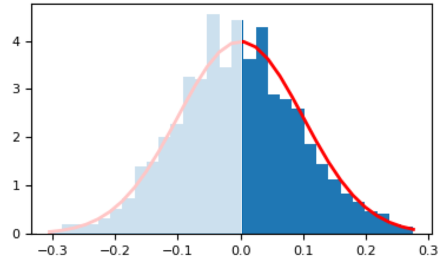
Neural network initialization II

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

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

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



  - 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

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

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

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

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

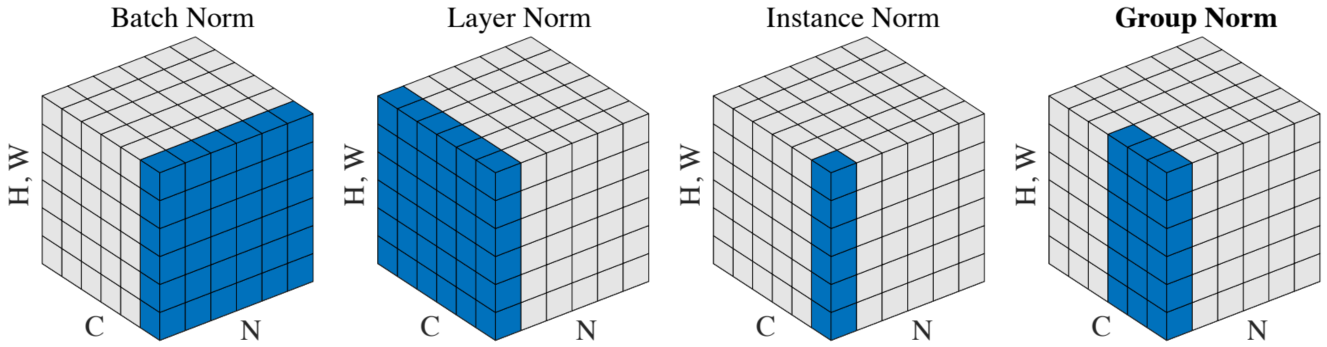
  - What if we only have a single image at inference time?
    - Re-apply mean and variance recorded during training (using exponential moving average)

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

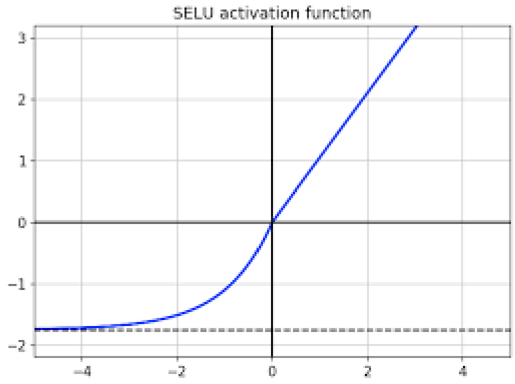
$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

Self-normalizing neural networks

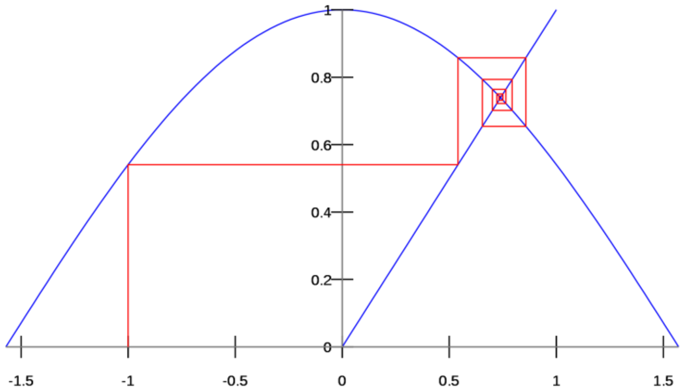
Self-normalizing Neural Networks

[Klambauer et al.]

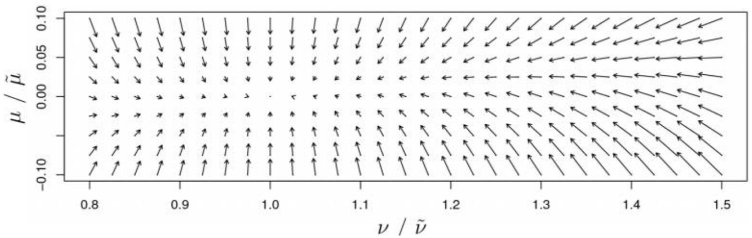
- fixed point enforced by choice of activation function (SELUs)
- stable and attracting fixed point for the function g that maps mean and variance from one layer to the next
- Possibility to train deep fully connected NNs



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



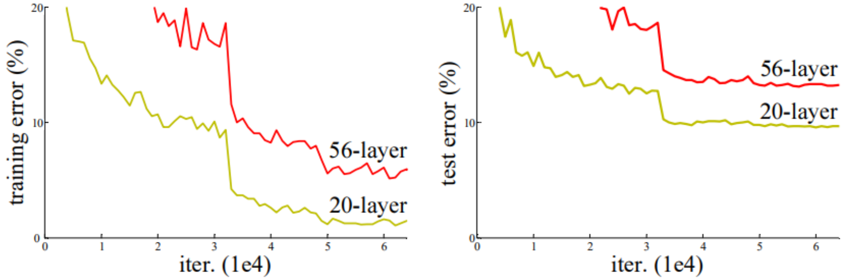
Fixed point iterations for cos(x)



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

Residual networks and skip connections

- Deep networks are hard to train



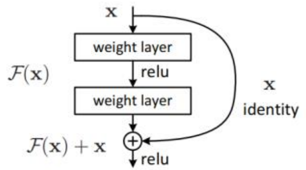


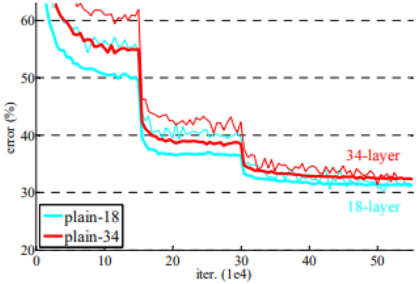
Figure 2. Residual learning: a building block.

- Residual blocks with shortcut/skip connections

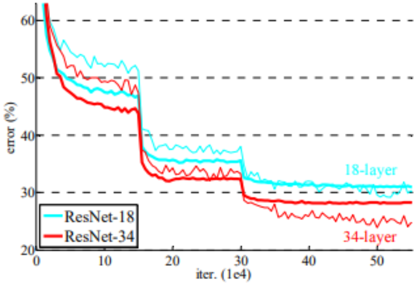
$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}$$

  - no extra parameters
  - enables training of deep neural networks

Image net training

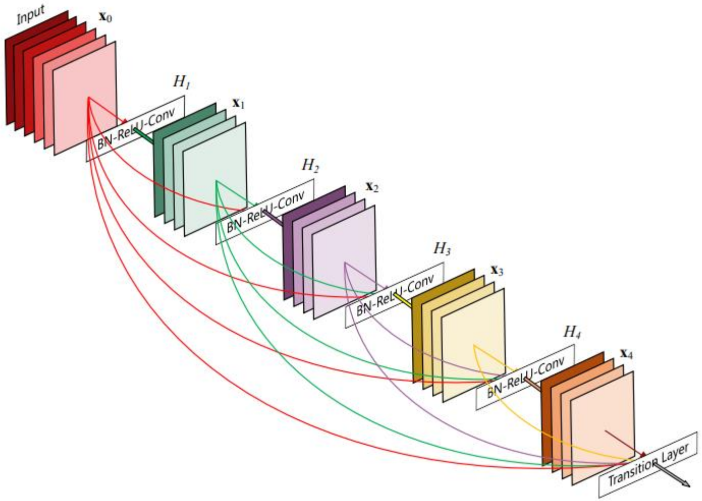


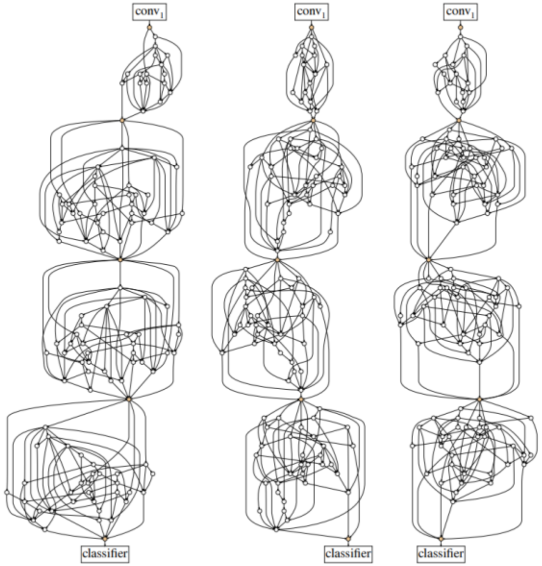
Baseline network



ResNet version

Other network architectures



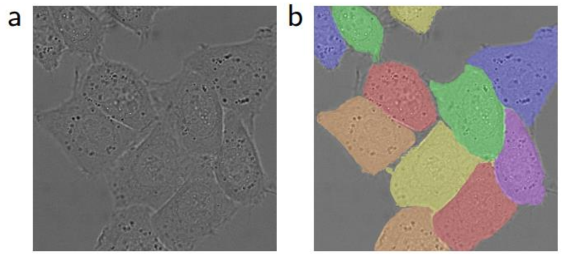


DenseNet
(skip connection to all future layers)

Randomly wired networks
(search for best wiring among candidates)

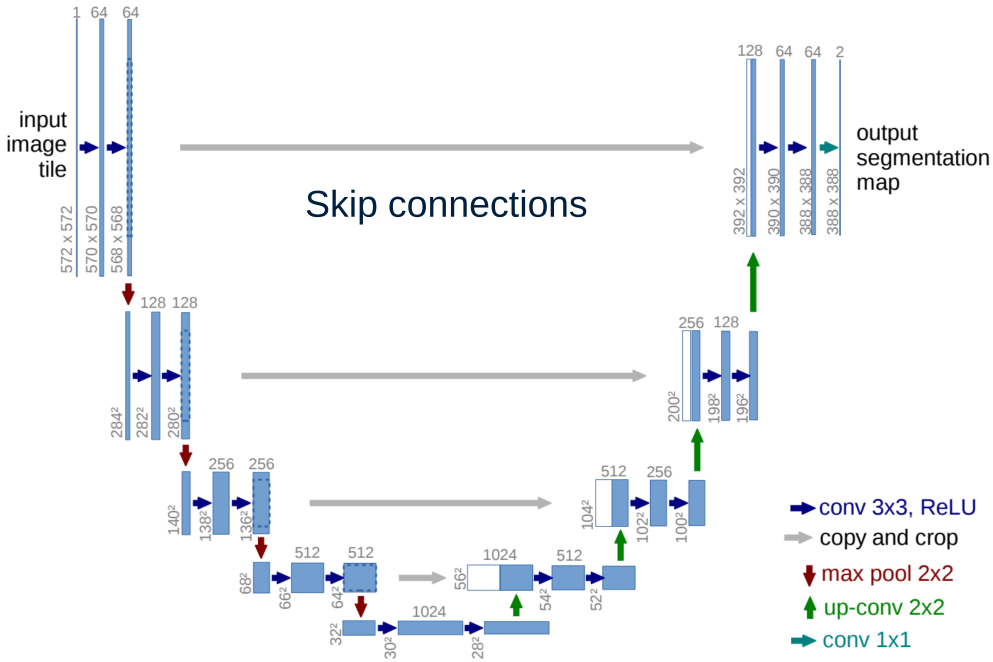
U-Net architecture

- 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)

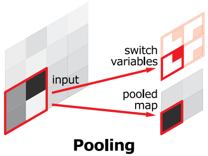


[U-Net: Convolutional Networks for Biomedical Image Segmentation]

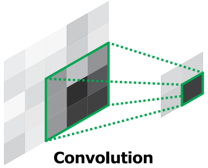
Spatial down and upsampling

Spatial downsampling

- max-pooling

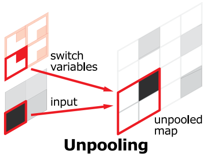


- average pooling
- convolution with stride

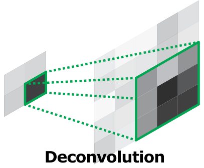


Spatial upsampling

- max-unpooling



- (bilinear) interpolation
- deconvolution



[Learning Deconvolution Network for Semantic Segmentation]

Hidden questions