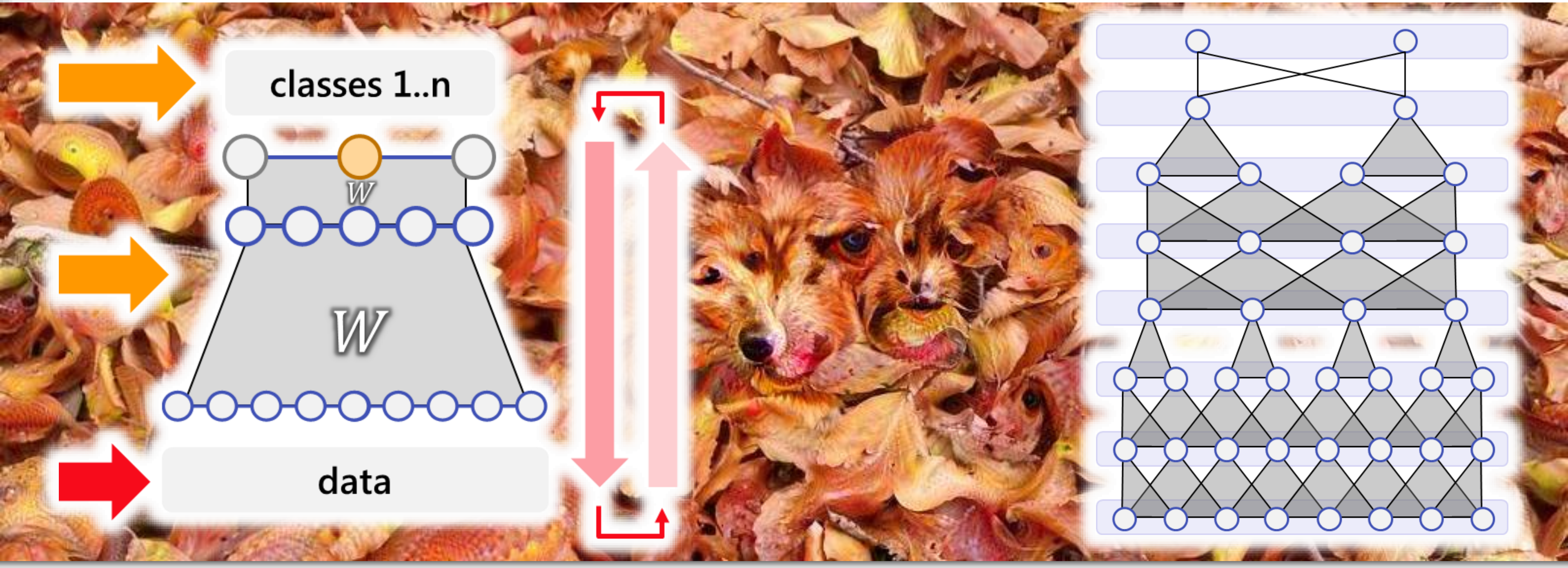
Modelling 2

STATISTICAL DATA MODELLING



[Deep Dream Image: Daniel Strecker]

Chapter 9

Deep Neural Networks

Michael Wand · Institut für Informatik, JGU Mainz · michael.wand@uni-mainz.de

Video #09

Down the Deep End

- **Back to the Future:** Neural Networks
- **Common Architectures**
- **Generative Models**

Artificial Neural Networks

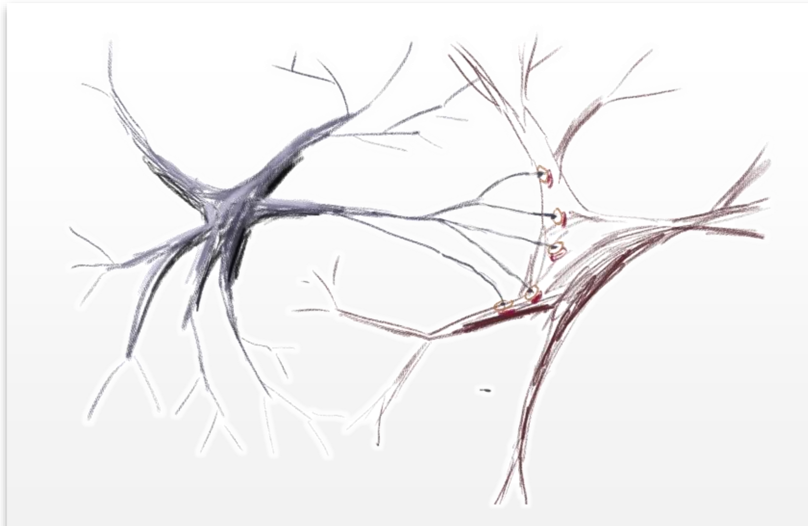
Crude Imitation of Nature

Laypersons (my) impression of neural circuits

Motivation: Biological Neural Networks

- Networks of computations
- Graph structure
- Neurons accumulate inputs until threshold
- Then “fire” output signals

Crude Imitation of Nature



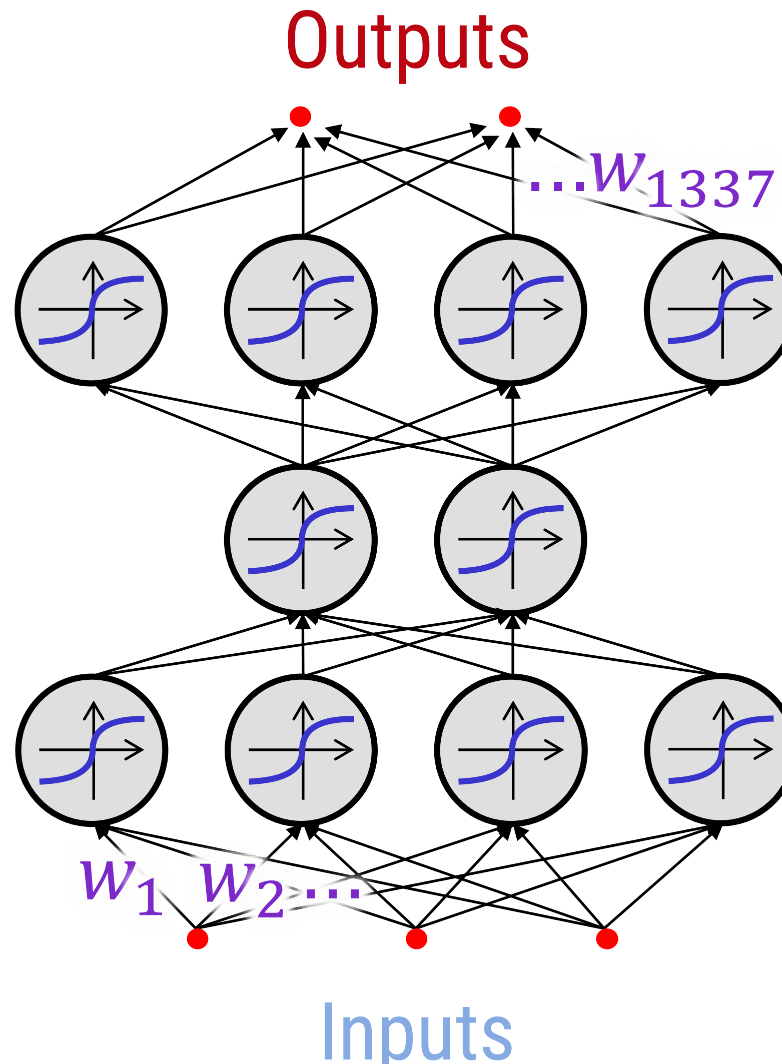
Dissimilar: Biological Neural Networks

- Complex computations
- Complex graph structure (including cycles)
- Sending, transmitting and gathering data non-trivial
- Spiking coding (not real numbers)

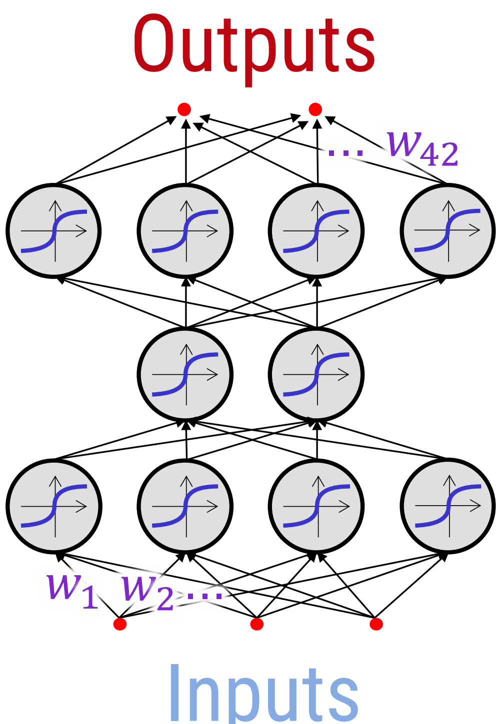
Artificial Neural Network

Simplified Model

- **Connections**
  → linear weights
- **Neurons**
  → Summation, activation
- **Activation**
  → simple non-linearity
- **Graph structure**
  → Simple pattern,
  often "feed-forward"

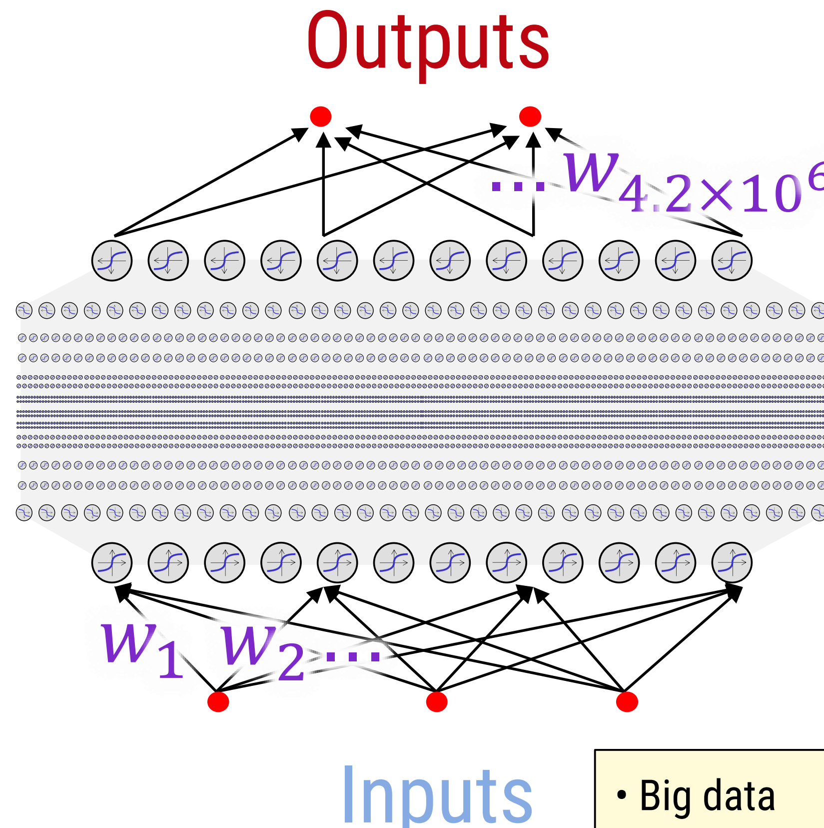


Neural Networks vs. Neural Networks



1980s / 1990s

- typ. 100s of "neurons"
- Bottleneck architecture



- Big data
- GPUs (TFlops)
- "Dirty tricks"

2010s

- 10^5 – 10^7 weights
- Overcomplete

Problems with NNs

Hard to train

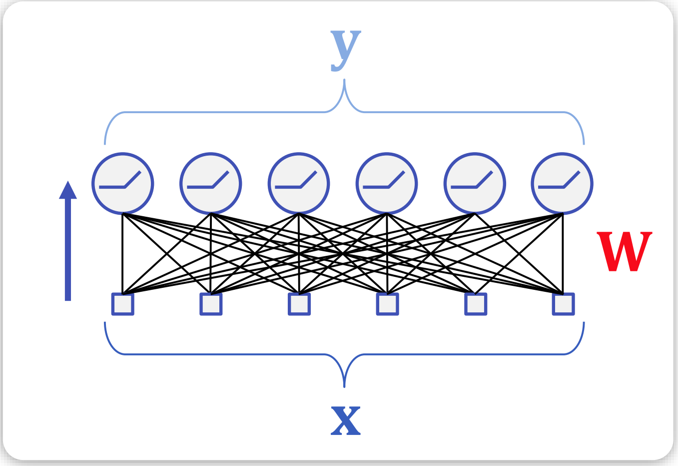
- Local minima
  - Overcomplete representations seem to find reasonable local minima
  - We do not fully understand why
- Numerical issues
  - Determining some of the weights ill-posed
  - “Dirty tricks” help a lot
  - Ongoing research

Inductive bias (NFL, BVT, etc.)

- Seems to work, no idea why

Deep Neural Networks

Architectural Building Blocks



Fully connected network layer

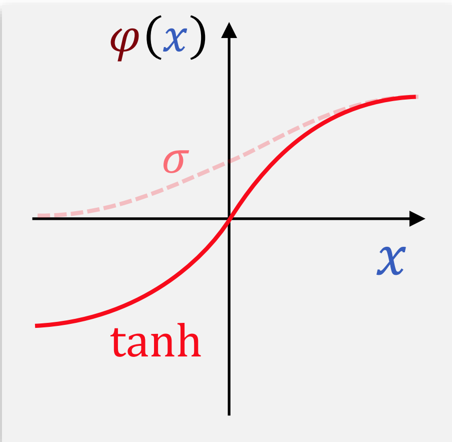
$$layer: \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$\mathbf{y} = layer(\mathbf{x}) = nonLinearity(\mathbf{W}\mathbf{x})$$

$$y_j = nonLinearity\left(\sum_{i=1}^{n} w_{ij} x_i\right)$$

$$nonLinearity(y) = \begin{cases} \max(y, 0) \;\; (\text{"relu"}) \\ \tanh(y) \\ \quad \dots \end{cases}$$

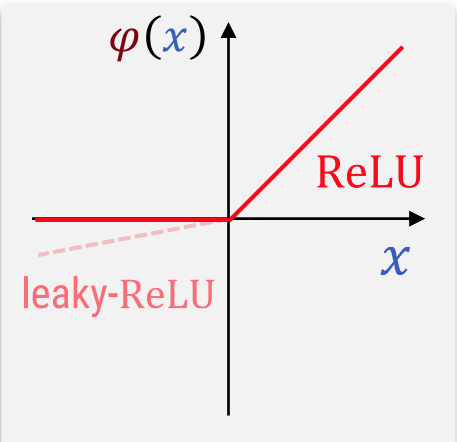
Non-Linearities



tangent hyperbolicus, sigmoid

$$\sigma(x) = \frac{e^x}{1-e^x}$$

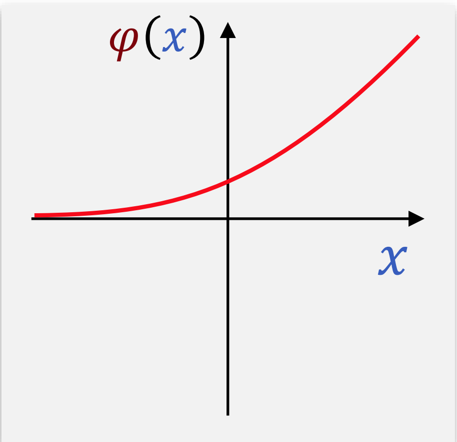
$$(\tanh(x) = 2\sigma(2x) - 1)$$



rectified linear unit

$$\mathrm{ReLU}(x) = \max(x, 0)$$

$$\text{leaky-ReLU}(x) = \max(x, \lambda x)$$
$$0 < \lambda < 1$$

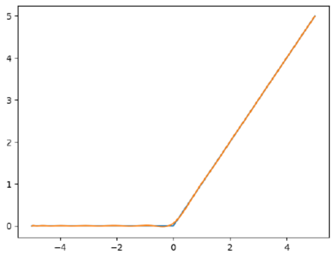


softplus

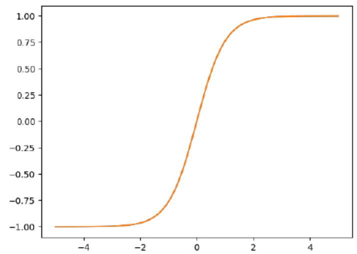
$$\mathrm{softplus}(x) = \ln(1 + e^x)$$

$$\mathrm{softplus}_\beta(x) = \frac{1}{\beta}\ln(1 + \beta e^x)$$

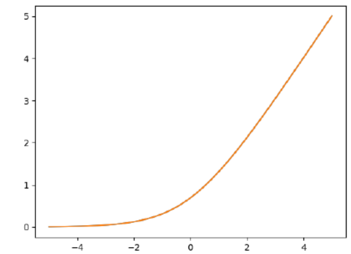
Millions more...



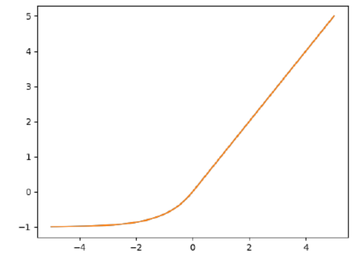
relu



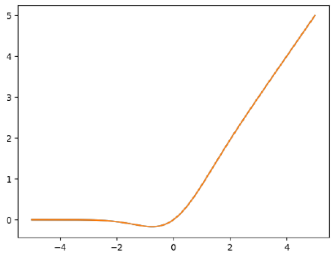
tanh



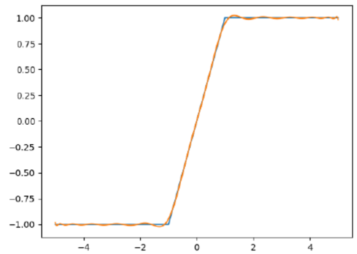
softplus



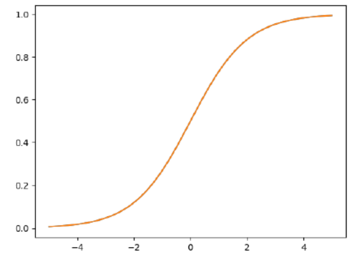
elu



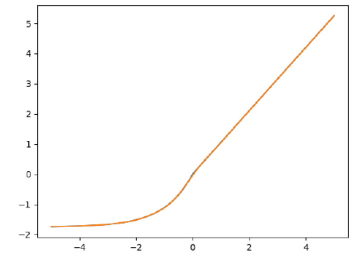
gelu



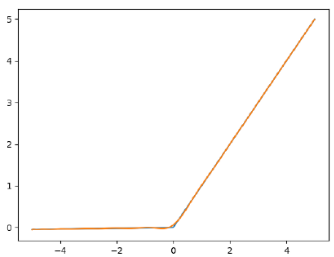
htanh



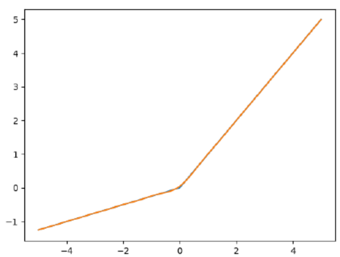
sigmoid



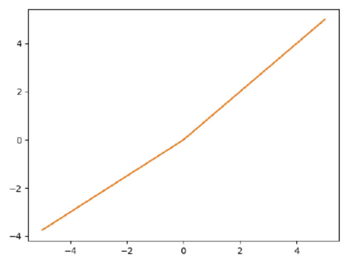
selu



lrelu



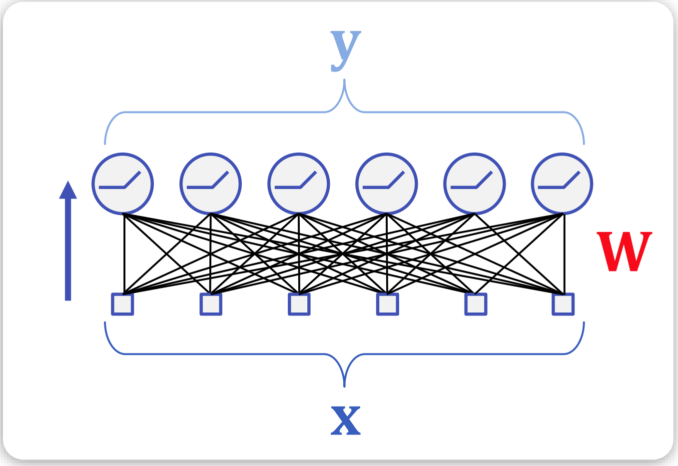
lrelu ($\alpha = 0.25$)



lrelu ($\alpha = 0.75$)

• • •

ReLU is Popular

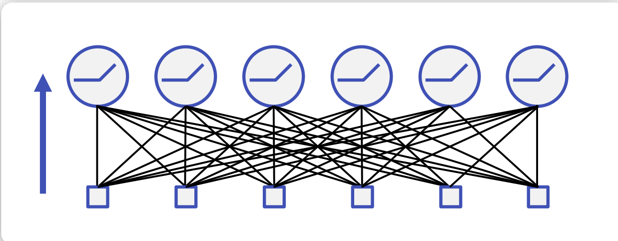


$$layer: \mathbb{R}^n \rightarrow \mathbb{R}^m$$

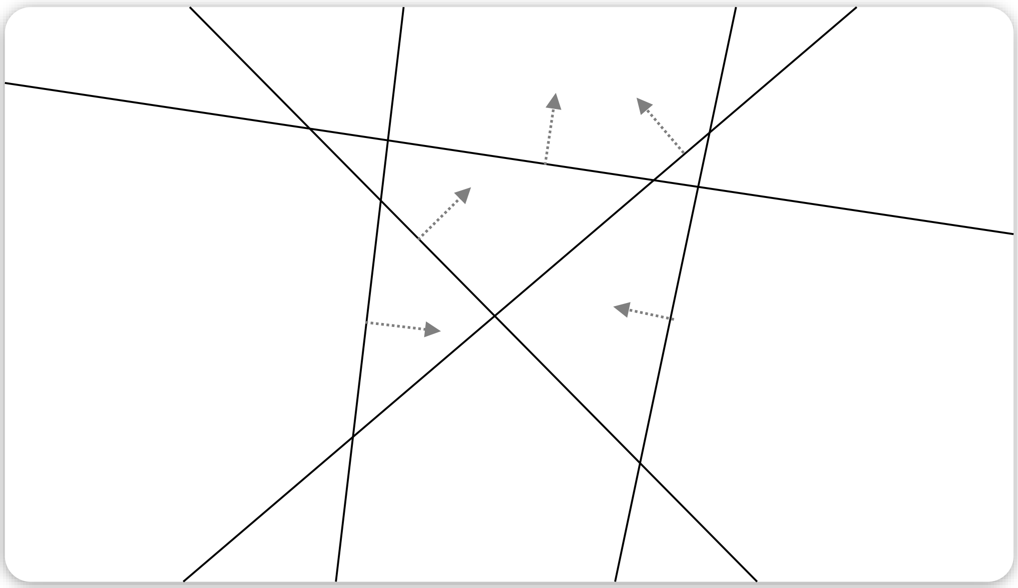
$$\mathbf{y} = layer(\mathbf{x}) = \varphi(\mathbf{W}\mathbf{x})$$

$$y_j = \max\left(0, \sum_{i=1}^{n} w_{ij} x_i\right)$$

Architectural Building Blocks



(Fully connected) network layer

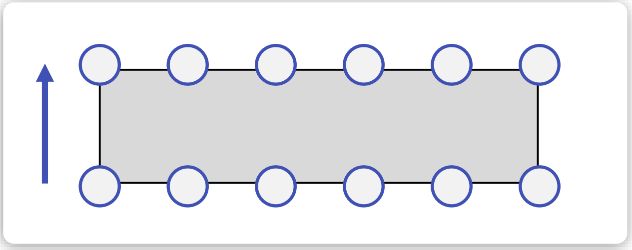


Interpretation: ReLu-Layer = Arrangement of Hyperplanes

Different linear map in each region

inactive halve has zero output in corr. coordiate

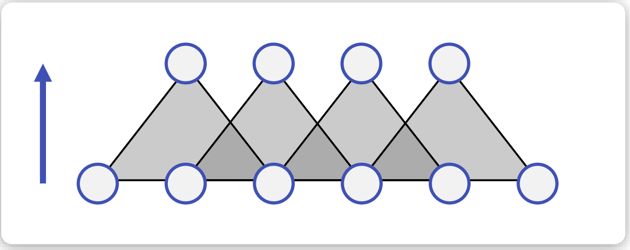
Architectural Building Blocks



Fully connected network layer

global connection / global dependencies

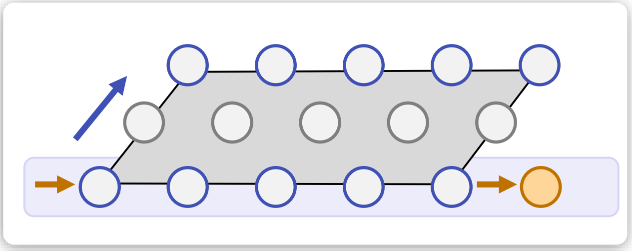
e.g: feature classification



Convolutional neural network

local connection / local correlations

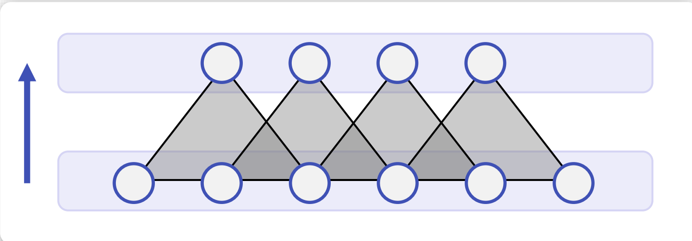
e.g.: image/audio/text data



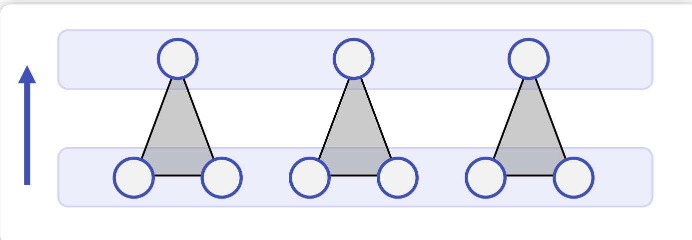
Recurrent neural networks

Markov-chain models with memory

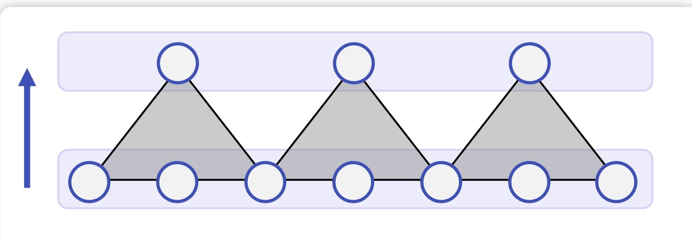
Convolutional Building Blocks



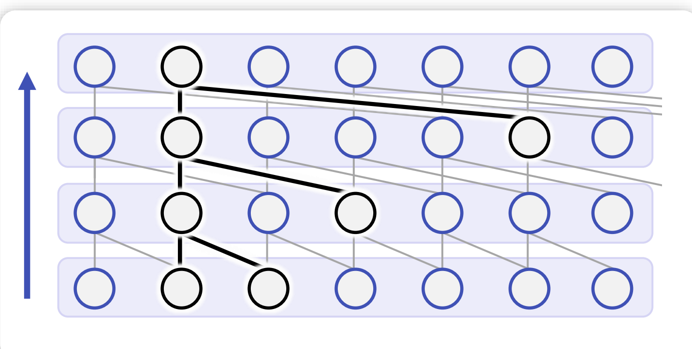
Convolutional neural network
local connection / local correlations



Pooling layer
reduce resolution (half, third, ...)

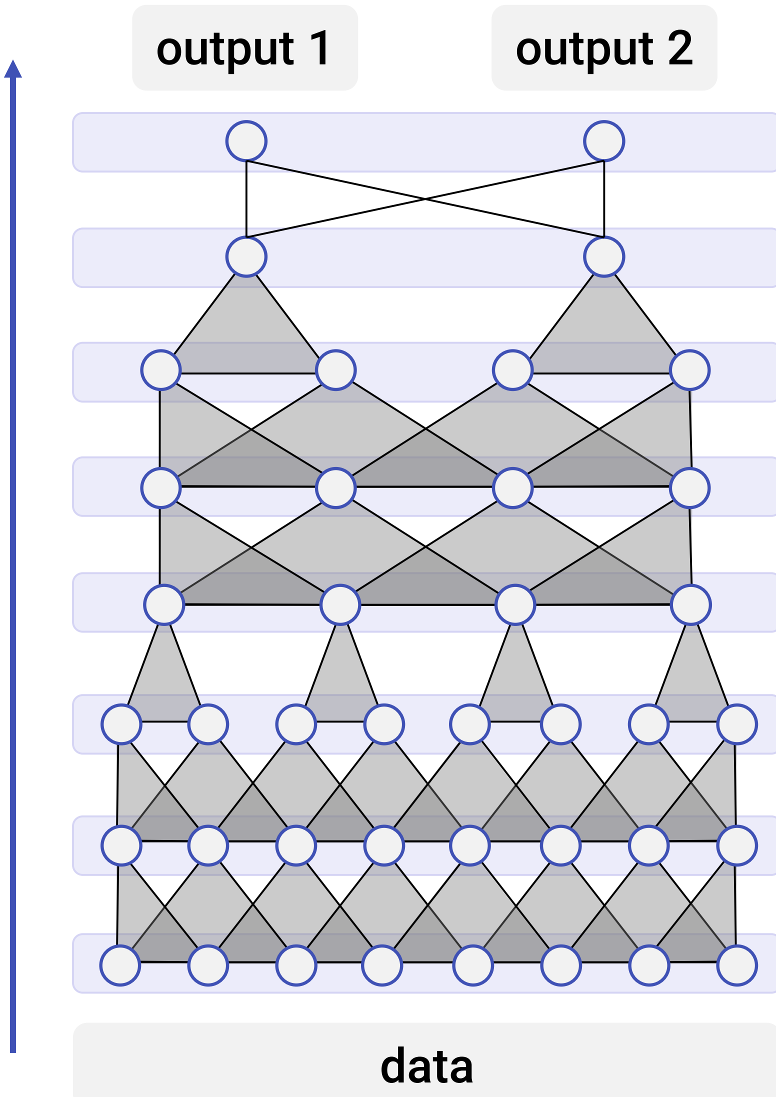


Convolution with stride
reduce resolution, learned filters



Dilated networks
aggregate context, same resolution

Image Classification



Fully Connected

Pooling / striding (typ. 2x2)

Convolution (typ. 3x3, residual)
Convolution (typ. 3x3, residual)
} **many layers**

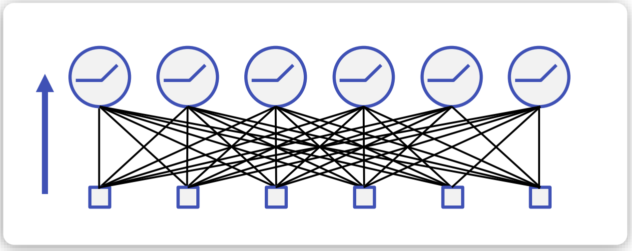
Pooling / striding (typ. 2x2)

Convolution (typ. 3x3, residual)
Convolution (typ. 3x3, residual)
} **many layers**

What does ReLU do?

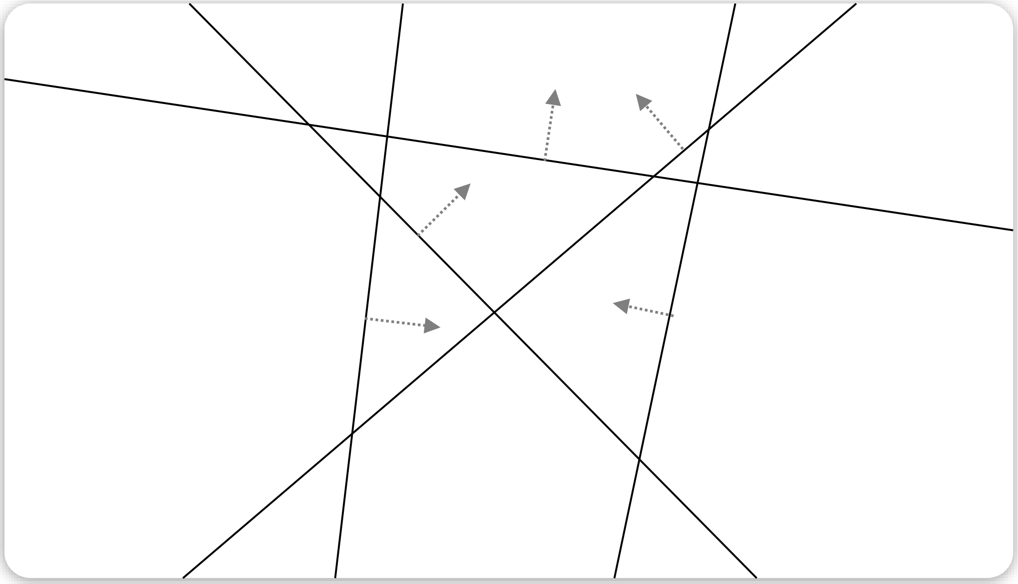
Architectural Building Blocks

network layer



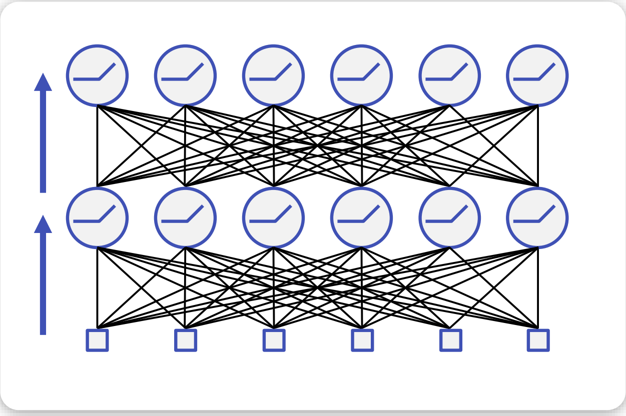
Interpretation

ReLU-layer = arrangement of hyperplanes



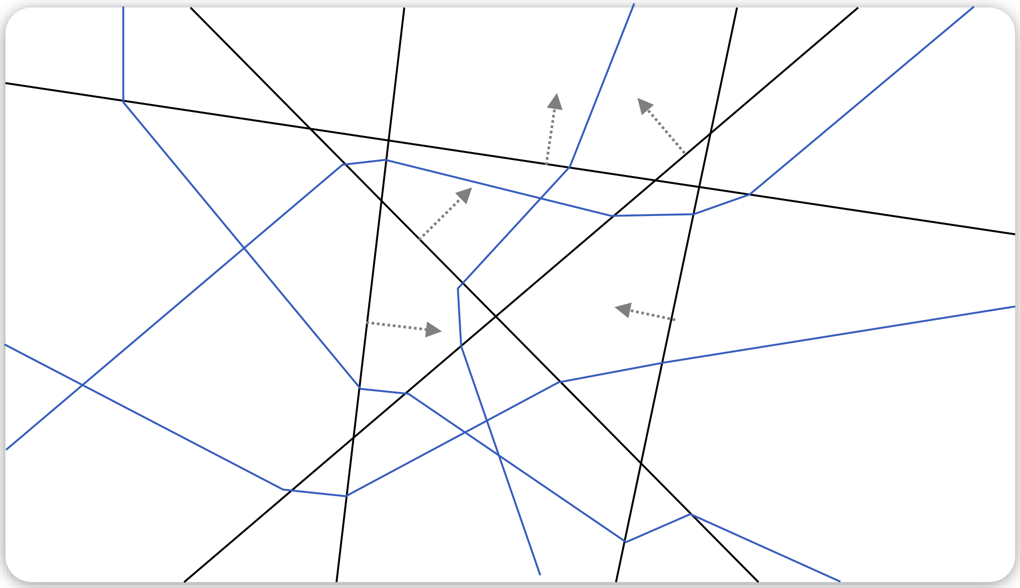
Architectural Building Blocks

two network layers



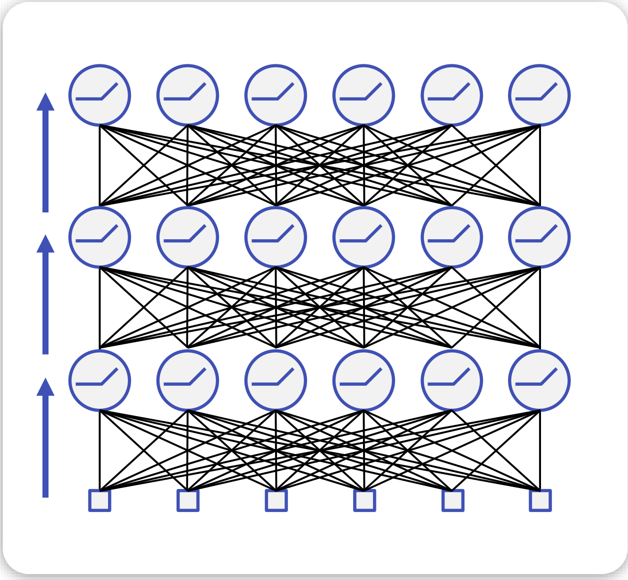
Interpretation

Nested ReLU-layer = nested convex cells



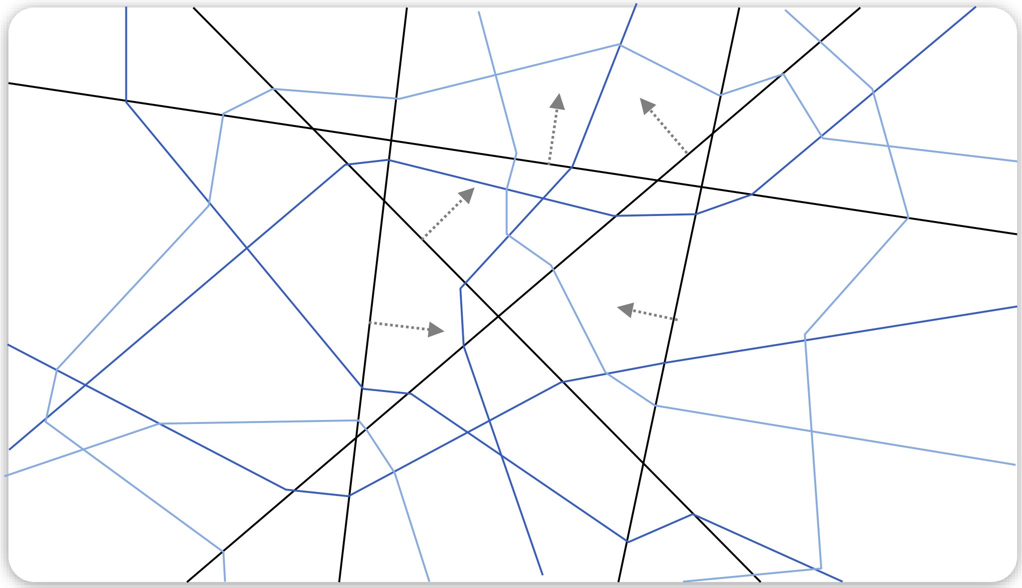
Architectural Building Blocks

two network layers



Interpretation

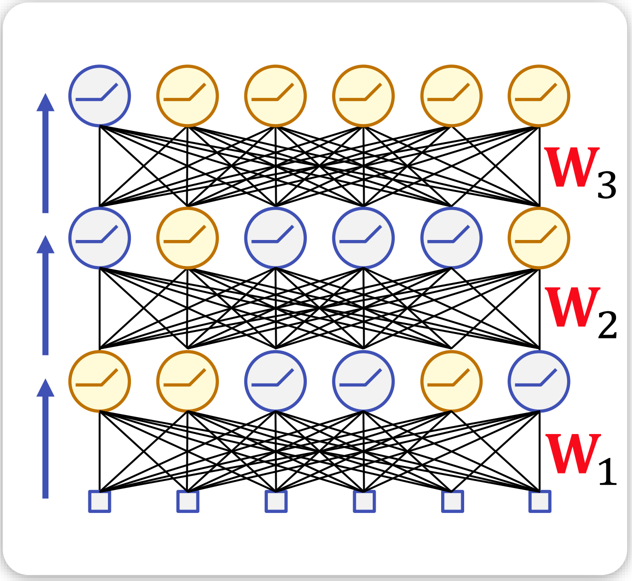
Nested ReLU-layer = nested convex cells



- Each cell has its own linear map
- applied to input to create output
- C^0-continuous

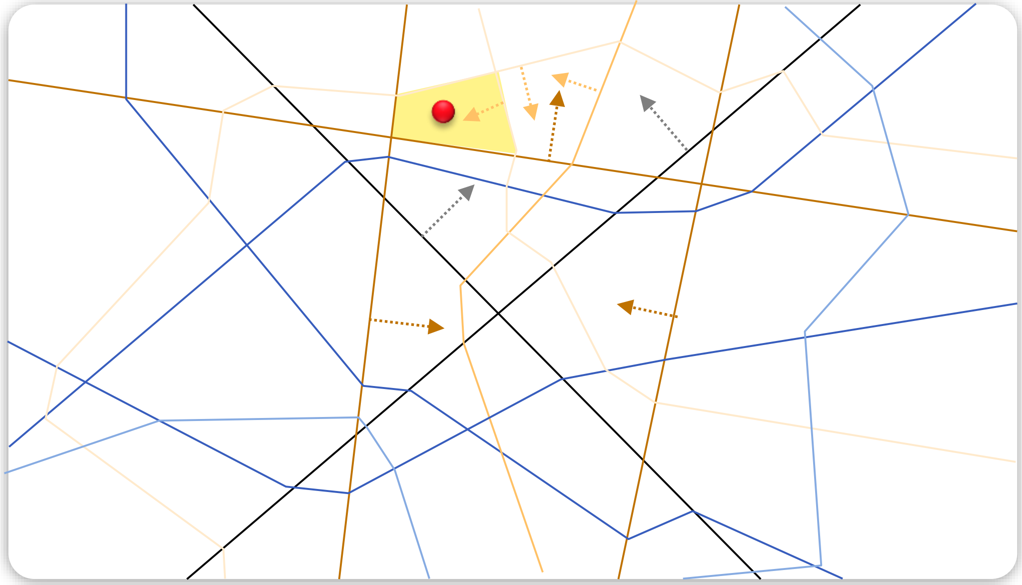
Architectural Building Blocks

two network layers



Interpretation

Nested ReLU-layer = nested convex cells



Activation Patterns

Encode combinatorial decisions
(which linear map to use)

Nomenclature

Language

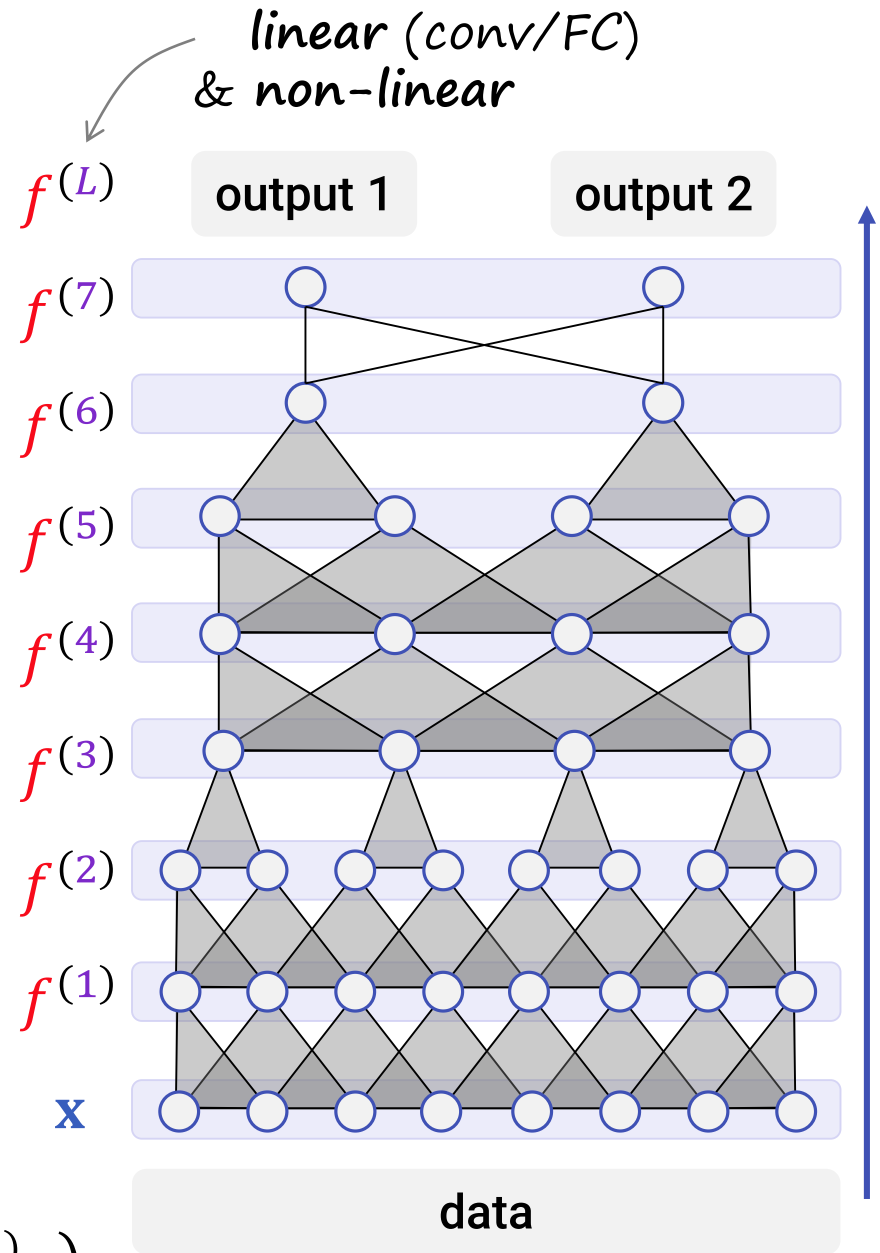
NN-Talk

- Input – what goes into the network
- Output – what comes out of the network
- “Features”, “hidden layers” – values at inner neurons
- Feed Forward Network – sequential processing
- Layer – one computation step in a ff network
- “preactivation” – number(s) going into the non-linearity
- “activation” – either
  - Numbers coming out of the non-linearity
  - Wether a ReLU has been switched “on”

Formalization

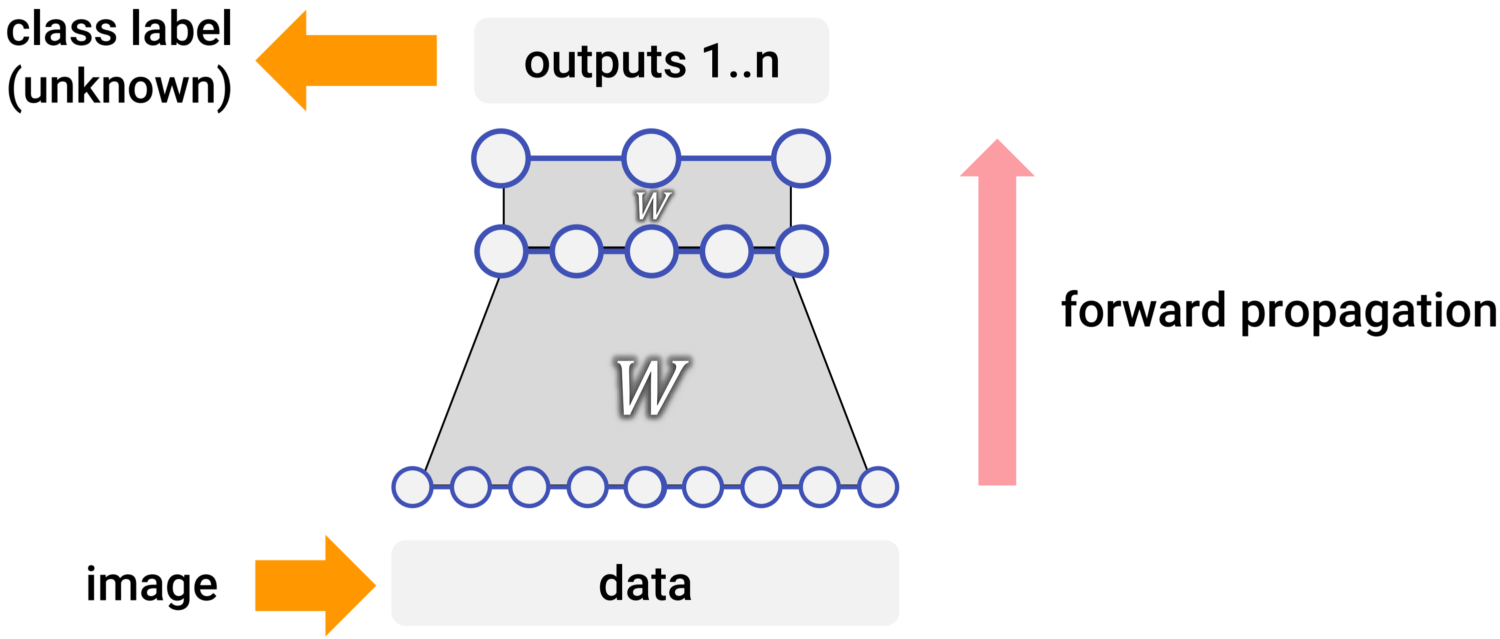
Network

- L layers $l = 1, \dots, L$
- Activations $y^{(l)} \in \mathbb{R}^{d_l}$
  - Input $f^{(0)} = \mathbf{x} \in \mathbb{R}^{d_0}$
  - Output $f^{(L)} \in \mathbb{R}^{d_L}$
  - Feed-forward
    $f^{(l)} = \mathrm{op}\left(f^{(l-1)}\right), f^{(0)} = \mathbf{x}$
- Layer function
  - Linear: $f^{(l)}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x}$
    (incl. conv., pooling, striding)
  - Non-linear: $f^{(l)}(\mathbf{x}) = \varphi(\mathbf{x})$
  - All-in-one: $f^{(l)}(\mathbf{x}) = \varphi\left(\mathbf{W}^{(l)}\mathbf{x}\right)$



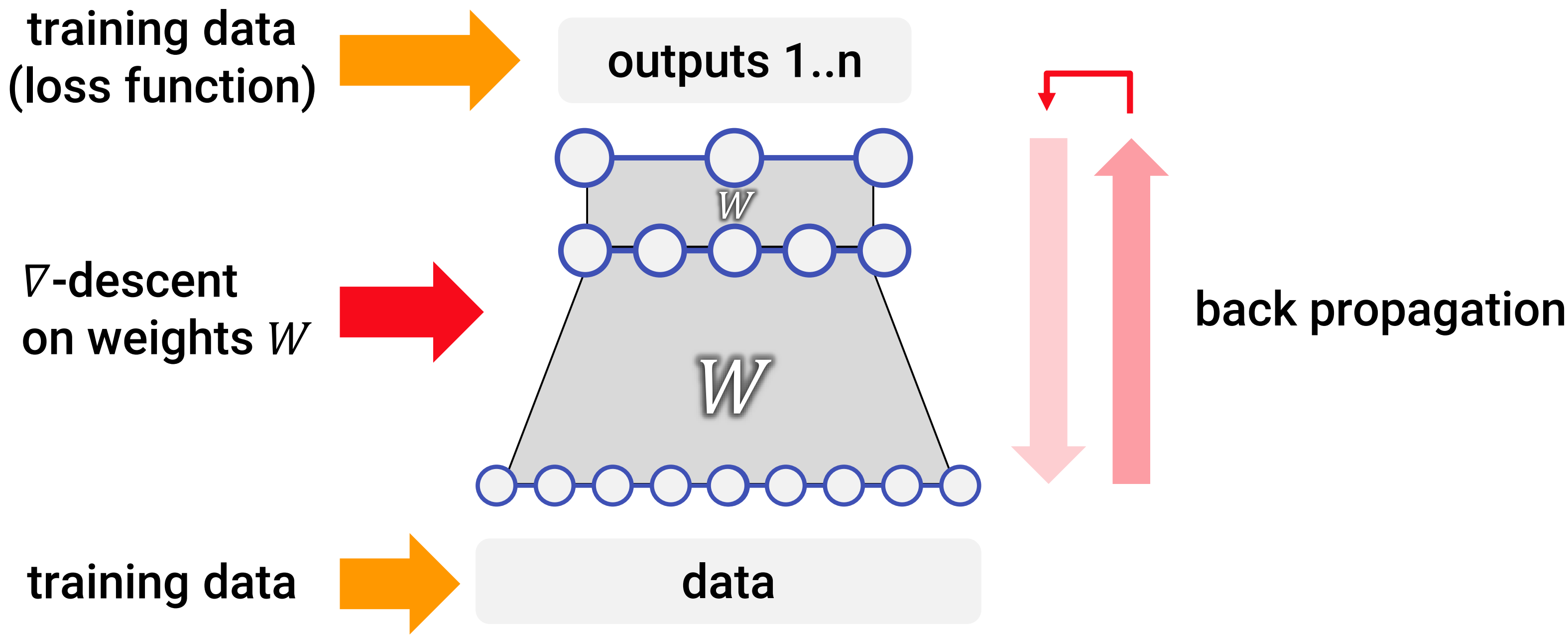
Training & Inference

Inference



class label
(unknown)
outputs 1..n
W
W
forward propagation
image
data

Discriminative Training



Some additional tricks…

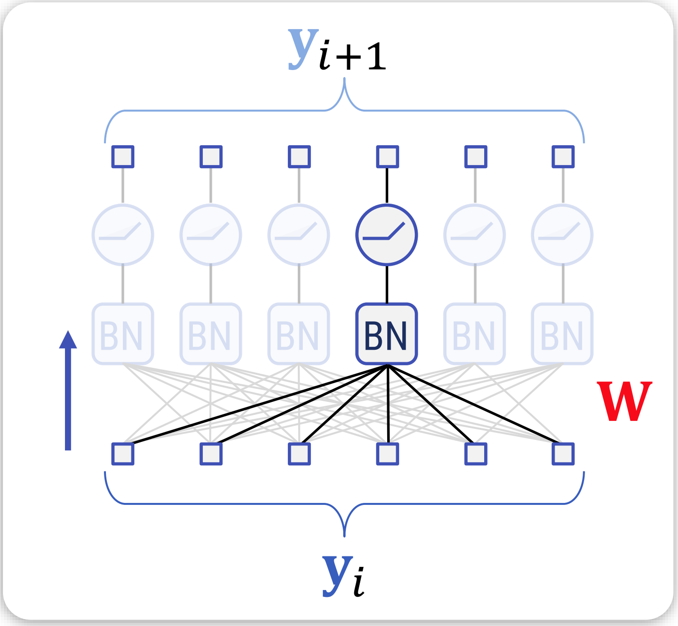
Batch Normalization

Batch-Norm Layer

- Normalize
  - mean $\mu = 0$
  - std. deviation $\sigma = 1$
- BN-Layer: per value

$$x \mapsto \alpha \frac{x - \mu}{\sigma} + \beta$$

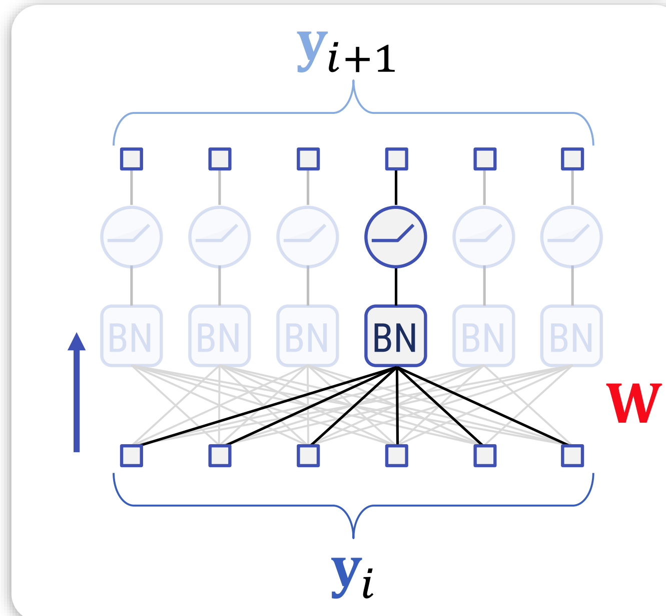
- Compute μ, σ from data
- Learn α, β



Batch Normalization

Training

- Est. μ, σ per batch
  - Empirical ML-estimators $\hat{\mu}$, $\hat{\sigma}$
  - Keep running means
    $\mu_{i+1} \to c\hat{\mu}_{i+1} + (1-c)\mu_i$
    $\sigma_{i+1} \to \cdots$
- Train α,β along with **W**
- Normalize



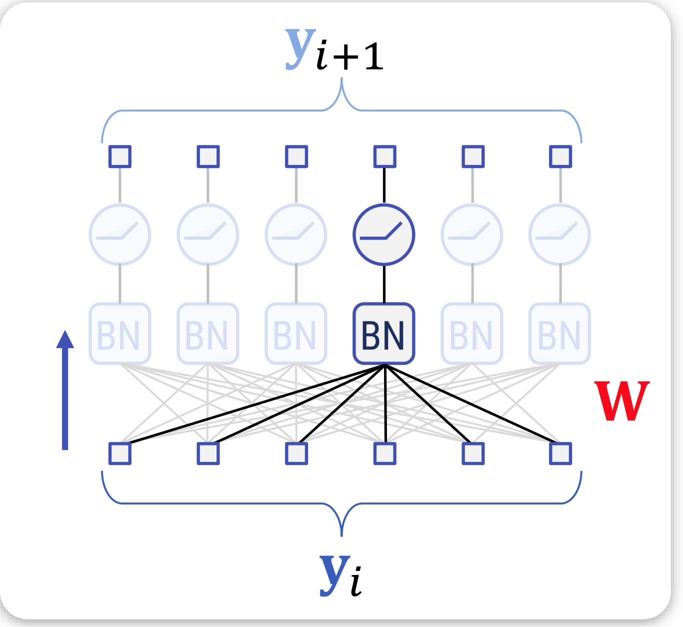
Testing: Normalize, too

- Use running averages
- μ_T, σ_T (T = last batch)

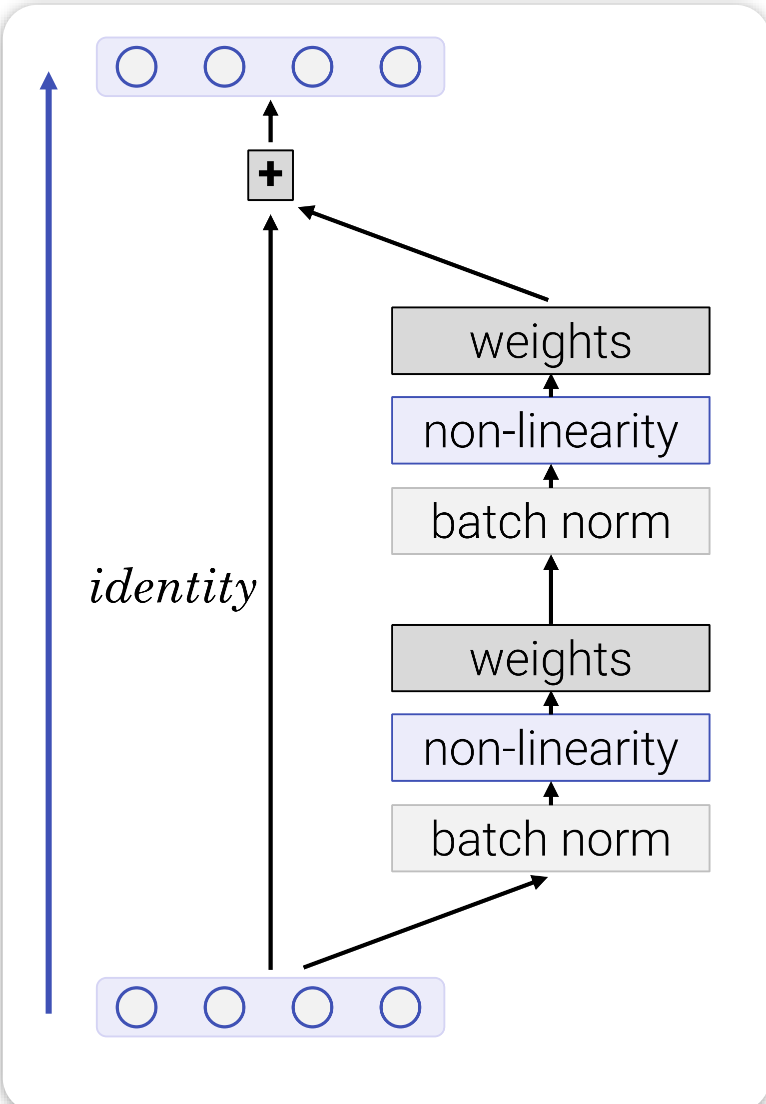
Batch Normalization

Some more alchemie

- Batch-Norm has problems
  - High-variance input data with small batches
  - Generative Networks (more later)
- Variants
  - Instance Norm
    - Only over one image
    - All convs/filters
  - Group Norm
  - Layer Norm



Residual Connections



Residual networks

Allows very deep networks

Identity mapping as default

Why all of this?

Batch-Norm

- "Covariate-Shift" – Data might hop around

ResNets (Batch-Norm?)

- "Vanishing gradients"
- Applying chain rule in network leads to dampening
- Some layers "do not move" anymore

What helps

- ResNet improves a lot
- BN causes "exploding grad.", the (maybe) converges

Summary

Deep Networks

Stack of

- Matrices
- Non-linearities
  - Simple ReLU “switches” do the trick (very well)

Optimization

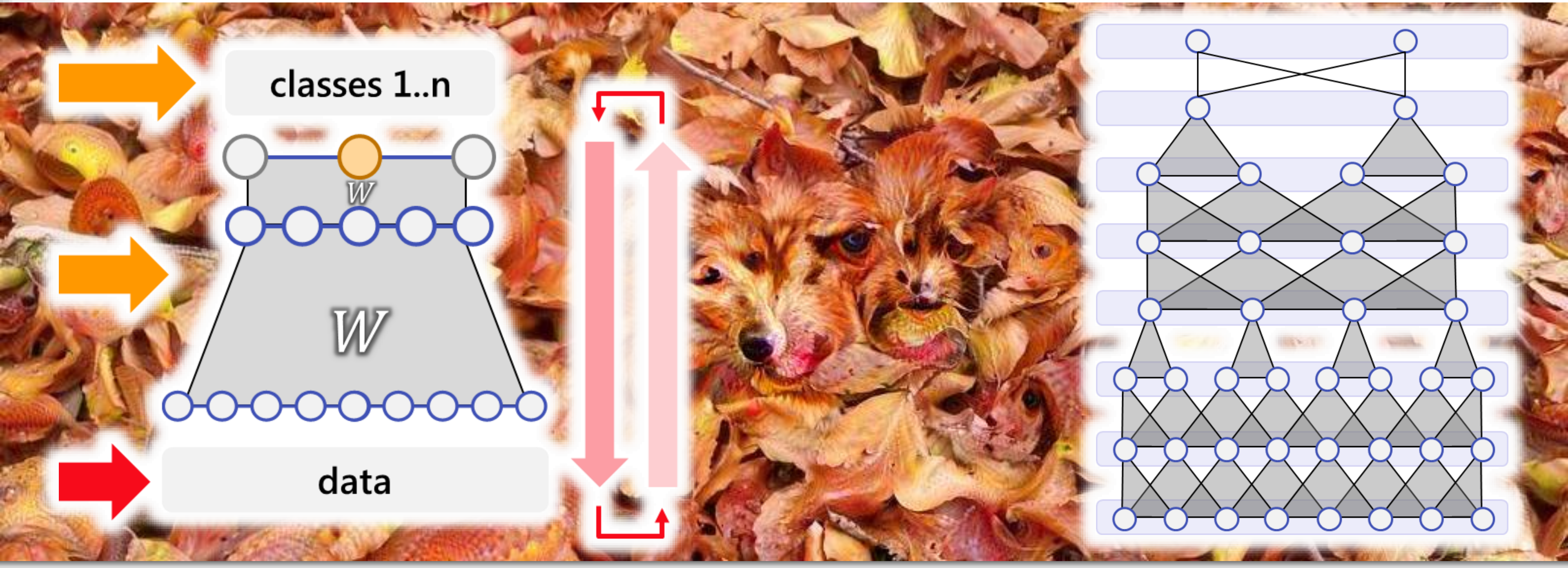
- Simple down-hill optimization
- Local minima do not seem to hurt

Numerics

- Some tricks to keep everything stable

Modelling 2

STATISTICAL DATA MODELLING



[Deep Dream Image: Daniel Strecker]

Chapter 9

Deep Neural Networks

Michael Wand · Institut für Informatik, JGU Mainz · michael.wand@uni-mainz.de

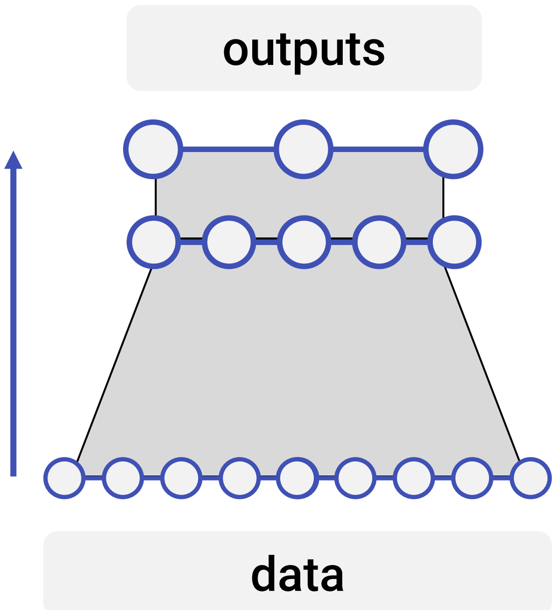
Video #09

Down the Deep End

- **Back to the Future:** Neural Networks
- **Common Architectures**
- **Generative Models**

Deep Regressor

Image Classification



Output values →

Fully connected
(typ. global av. pooling + 1 layer)

Convolution + pooling or striding
(typ. 20-100 layers)

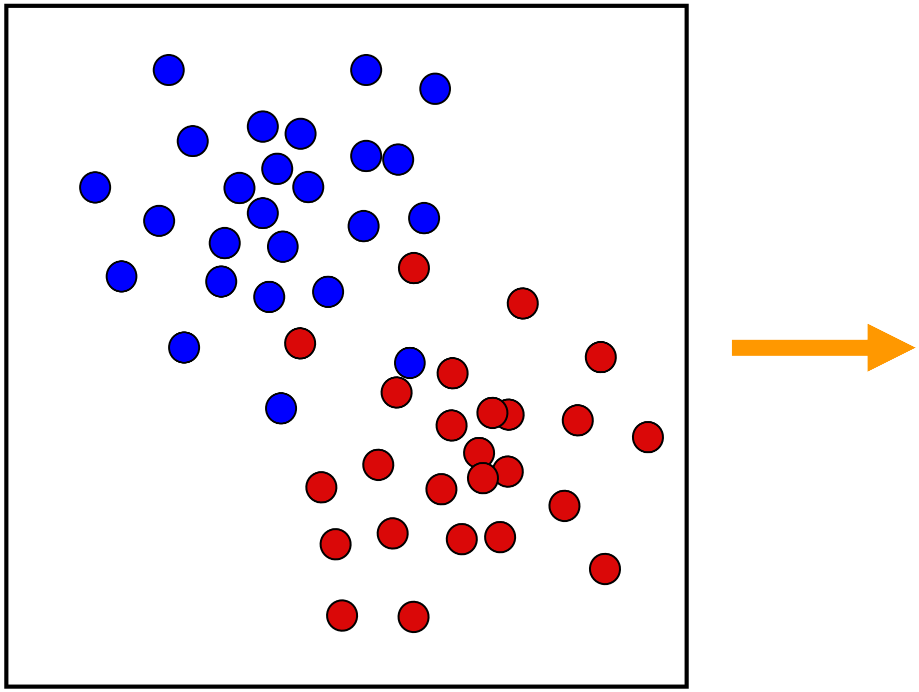
← Raw input images

Loss function

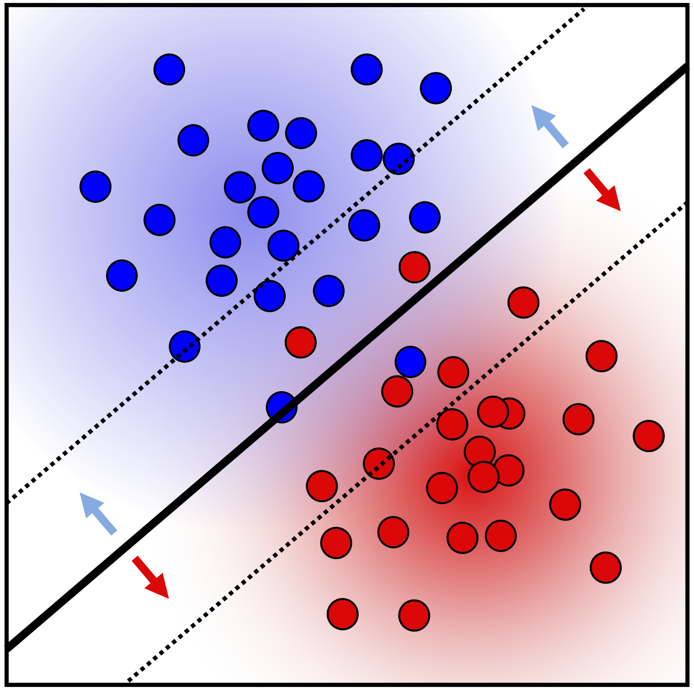
- Typ. Point-wise loss $\sum_{i=1}^{n} |[f_W(\mathbf{x})]_i - \mathbf{y}_i|^p$
- Often least-squares: $L(f_W(\mathbf{x}), \mathbf{y}) = ||f_W(\mathbf{x}) - \mathbf{y}||_2^2$

Deep Classifiers

SVM / Logistic, Softmax Regression



training set



linear separator
(now on the top layer)

Loss Function

Notation

- Neural network f, weights W, input $\mathbf{x}$, output $\mathbf{y}$
- Supervised, training data: $(\mathbf{x}_i, \mathbf{y}_i)_{i=1..n}$
- $f_W(\mathbf{x}) = \mathbf{y}$

Different Loss Functions

Regression:

- Least squares $(f_W(\mathbf{x}_i) - \mathbf{y}_i)^2$

Classification

- One-Hot-Vectors $\mathbf{y}_i$
- Cross Entropy:

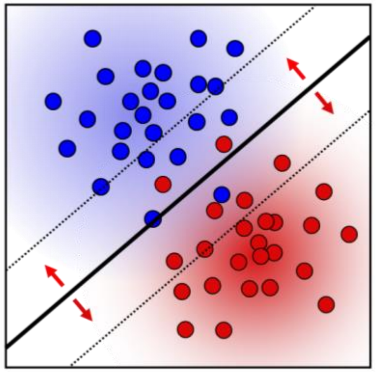
$$H\big(\mathrm{softmax}\big(f_W(\mathbf{x}_i)\big), \mathbf{y}_i\big)$$

- Max-Margin:

$$\mathrm{margin}\big(f_W(\mathbf{x}_i), \mathbf{y}_i\big)$$

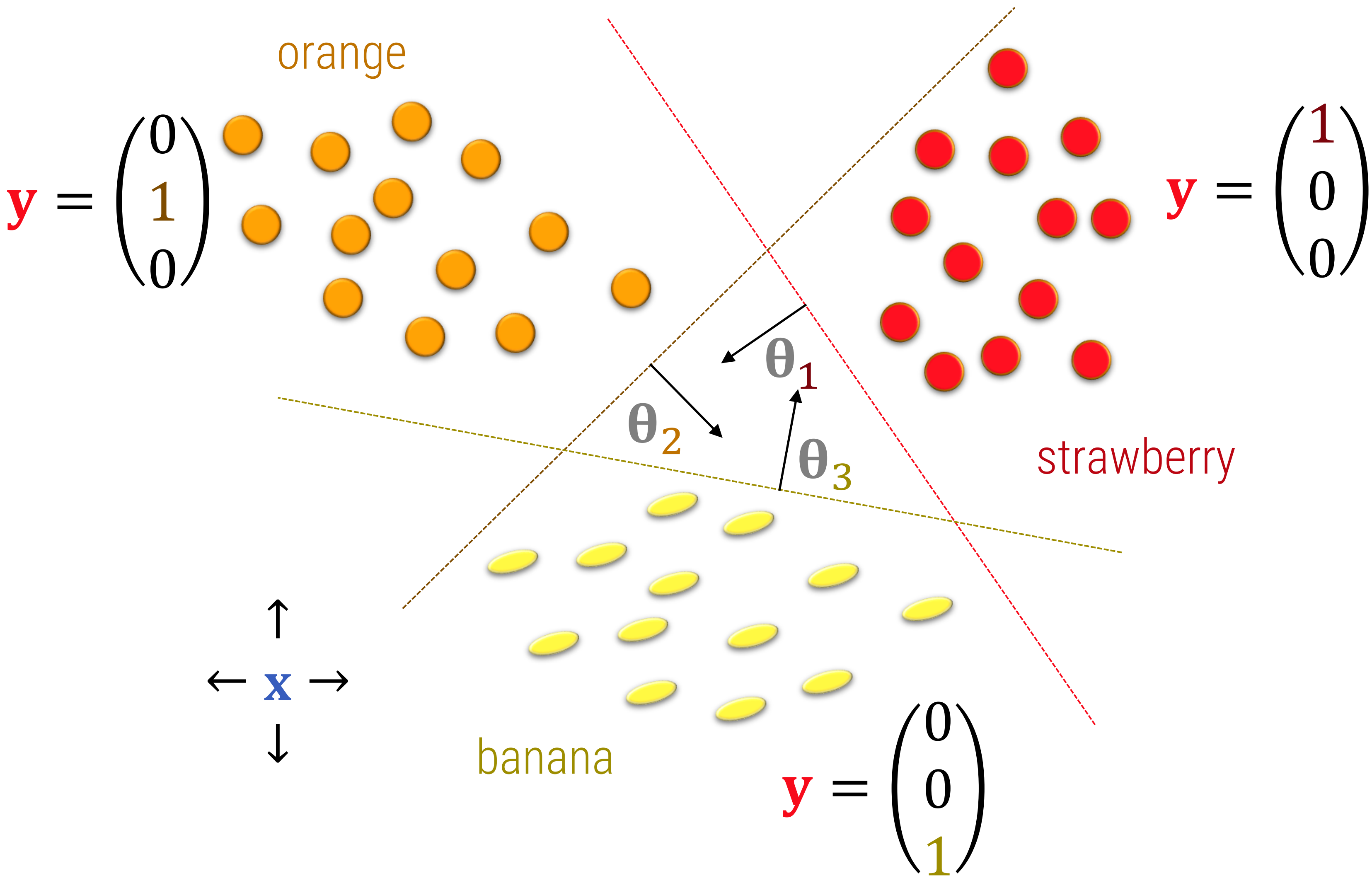
Softmax:

$$\mathrm{softmax}(\mathbf{y}) = \begin{pmatrix} \frac{e^{-y_1}}{\sum_{i=1}^{n} e^{-y_i}} \\ \vdots \\ \frac{e^{-y_n}}{\sum_{i=1}^{n} e^{-y_i}} \end{pmatrix}$$



CE-Loss

Geometry



orange
$\mathbf{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$
strawberry
$\mathbf{y} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$
θ_1
θ_2
θ_3
$\mathbf{x}$
banana
$\mathbf{y} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

Softmax Regression

"Softmax" function $\boldsymbol{\sigma}\colon \mathbb{R}^K \to \mathbb{R}^K$

$$\boldsymbol{\sigma}(\mathbf{z}) \coloneqq \begin{pmatrix} \dfrac{e^{z_1}}{\sum_{j=1}^{K} e^{z_j}} \\ \vdots \\ \dfrac{e^{z_K}}{\sum_{j=1}^{K} e^{z_j}} \end{pmatrix}, \qquad \sigma_m(\mathbf{z}) \coloneqq \frac{e^{z_m}}{\sum_{j=1}^{K} e^{z_j}}$$

Classifier

Classifier

$$h_{\boldsymbol{\theta}}(\mathbf{x}) \coloneqq \boldsymbol{\sigma}\left(\underbrace{\begin{bmatrix} \boldsymbol{\theta}_1^T \cdot \mathbf{x} \\ \vdots \\ \boldsymbol{\theta}_K^T \cdot \mathbf{x} \end{bmatrix}}_{\mathbf{u}(\boldsymbol{\theta},\mathbf{x})}\right) = \boldsymbol{\sigma}(\mathbf{u}(\boldsymbol{\theta},\mathbf{x}))$$

$$\rightarrow\ h_{\boldsymbol{\theta}}(\mathbf{x}) \coloneqq \boldsymbol{\sigma}\left(\begin{bmatrix} f_{\mathbf{W}}(\mathbf{x})_1 \\ \vdots \\ f_{\mathbf{W}}(\mathbf{x})_K \end{bmatrix}\right) = \boldsymbol{\sigma}(f_{\mathbf{W}}(\mathbf{x}))$$

- Outputs class-probabilities
  - All output vector entries in [0,1]
  - Entries sum up to one

Classifier

Classifier

- MLE-Training via

$$\underset{\theta\in\mathbb{R}^{K\times d}}{\arg\min}\sum_{i=1}^{n}\left[\underbrace{\log\left(\sum_{j=1}^{K}e^{[f_{\mathbf{w}}(\mathbf{x})]_j}\right)}_{\text{normalization factor } Z}-\sum_{m=1}^{K}\underbrace{\mathbf{y}_{i,m}}_{\text{1 only for correct class}}\cdot\underbrace{\log\sigma_m(f_{\mathbf{w}}(\mathbf{x}))}_{\text{(neg)-log-likelihood of correct class}}\right]$$

$$=\underset{\theta\in\mathbb{R}^{K\times d}}{\arg\min}\sum_{i=1}^{n}\left[\underbrace{\log(Z)}_{\text{normalization}}-\underbrace{\log\sigma_{class_i}(f_{\mathbf{w}}(\mathbf{x}))}_{\text{(neg)-log-likelihood of correct class}}\right]$$

Cross Entropy as Maximum-Likelihood

$$\arg\min_{W} KL\left(\mathbf{y}_i \parallel f_W(\mathbf{x}_i)\right)$$

← KL-Divergence output ↔ labels

$$= \arg\min_{W} \sum_{k=1}^{n_l} [\mathbf{y}_i]_k \log_2 \frac{[\mathbf{y}_i]_k}{[f_W(\mathbf{x}_i)]_k}$$

$$= \arg\min_{W} \left(H\left(\mathbf{y}_i, f_W(\mathbf{x}_i)\right) - H(\mathbf{y}_i)\right)$$

$$= \arg\min_{W} \left(H\left(\mathbf{y}_i, f_W(\mathbf{x}_i)\right)\right)$$

← Cross-Entropy Loss

$$= \arg\min_{W} \sum_{k=1}^{n_l} [\mathbf{y}_i]_k \log_2 [f_W(\mathbf{x}_i)]_k$$

$$= \arg\min_{W} \log_2 [f_W(\mathbf{x}_i)]_{k=l_i}$$

← Class likelihood (maximization)

Geometry (on the top layer)

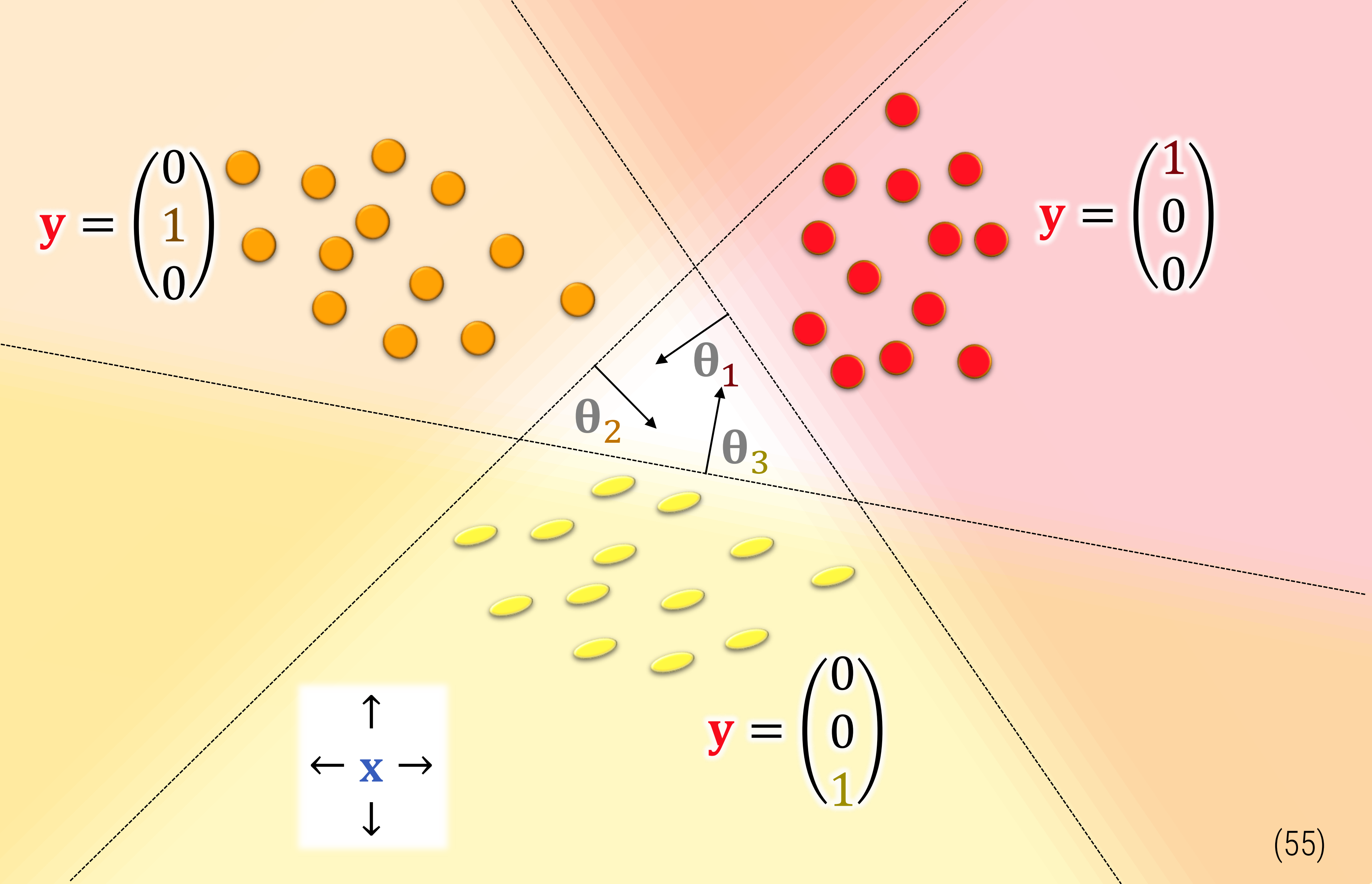
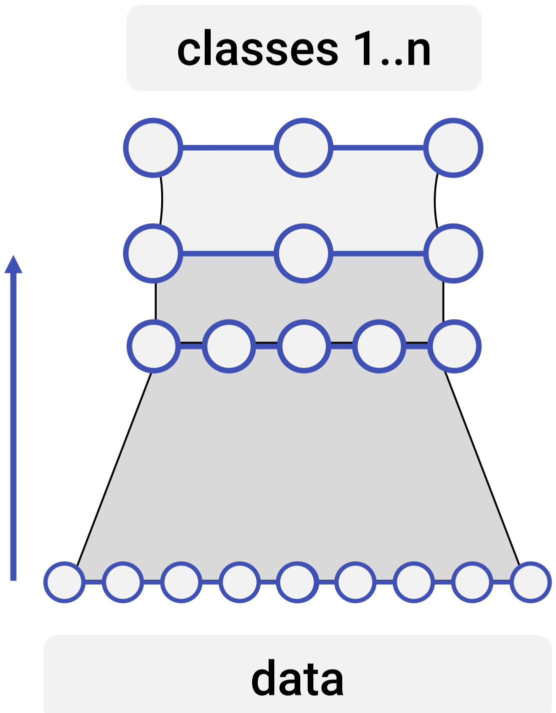


Image Classification



Class probabilities →

Softmax

Fully connected
(typ. global av. pooling + 1 layer)

Convolution + pooling or striding
(typ. 20-100 layers)

← Raw input images

Loss function

- Cross-Entropy loss
- (Hinge loss would work in principle, but uncommon)

How well does it work?

ImageNet

- 14 000 000 Color images (RGB)
  - Scraped from the web
  - Annotated via crowdsourcing
- 20 000 Classes

ImageNet Large Scale Visual Recognition Challenge

- 1 000 000 Training images
- 1 000 Non-overlapping categories

How well does it work?

ImageNet Large Scale Visual Recognition Challenge

- 1 000 000 Training images
- 1 000 Non-overlapping categories

Accuracy

- $\leq$ 2011: trad. methods — 25% top-5 error
- 2012: AlexNet — 16% top-5 error
- 2014: VGG-Net — 8% top-5 error
- 2015: ResNet / Inception — 5% top-5 error
- 2021: NFNet-v6 — 2.1% top5 error

  (frontrunner on papers with code 31/05/21)

Example
(Tales from Down the Deep End)

Object Detection

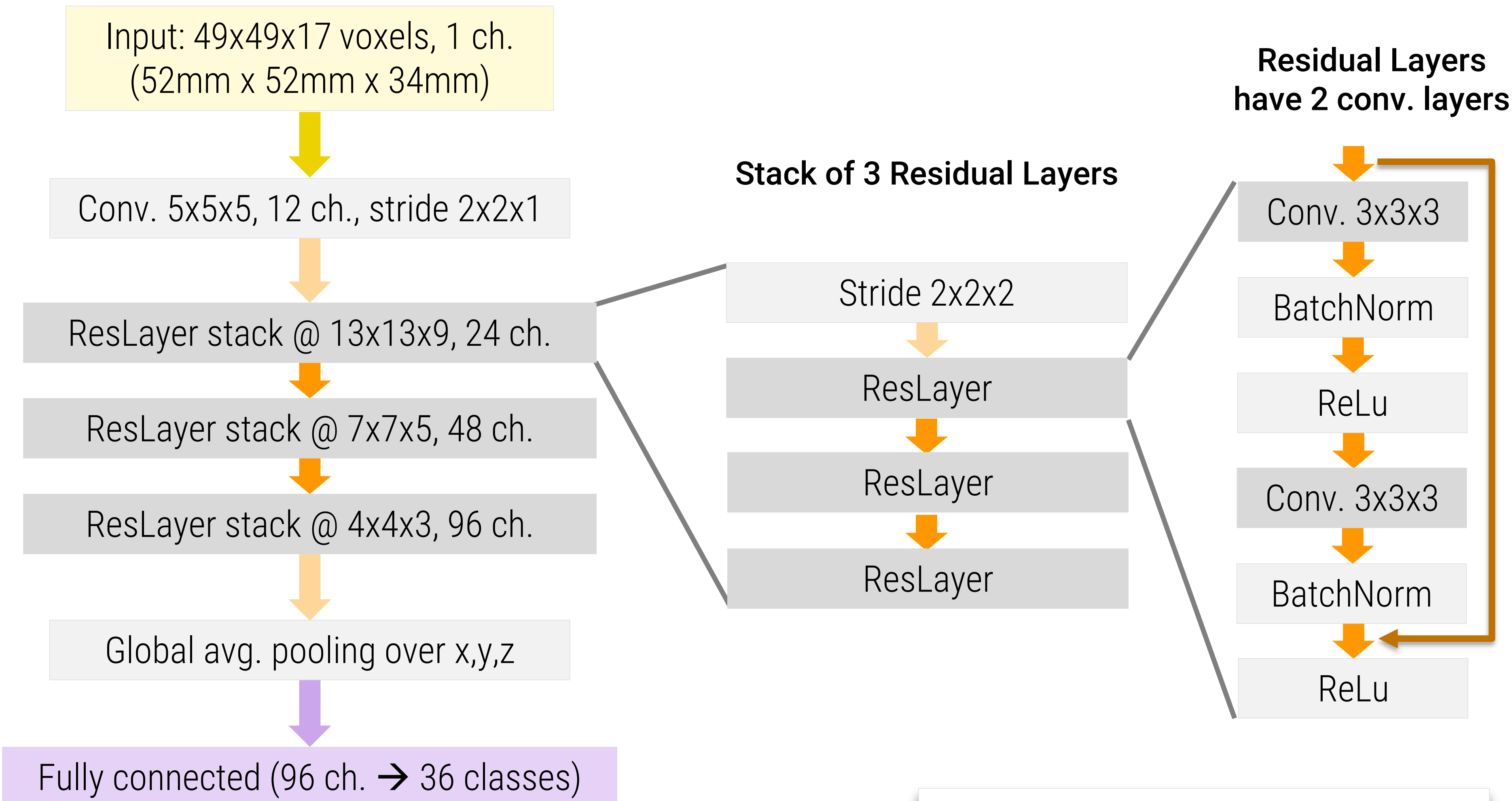
CT scans from University Hospital Mainz

- Centroids of vertebra annotated
- 36 Classes:
  - 18 different vertebra present in scans (C7, Th1-Th12, L1-L5)
  - 17 spaces between vertebra
  - 1 class for “not a vertebra”
- Training set: 152 CT scans
- Testing data: 66 CT scans

Microsoft Research Benchmark

- 150 examples (spine CTs)

Deep Residual Network



∑ 19 convolutional layers
1.809.336 trainable parameters

Sliding-Window Results

Task 2: 18 vertebrae types

- Identify vertebra *by index* (!)
- No context (bounding box)

Top-1-accuracy

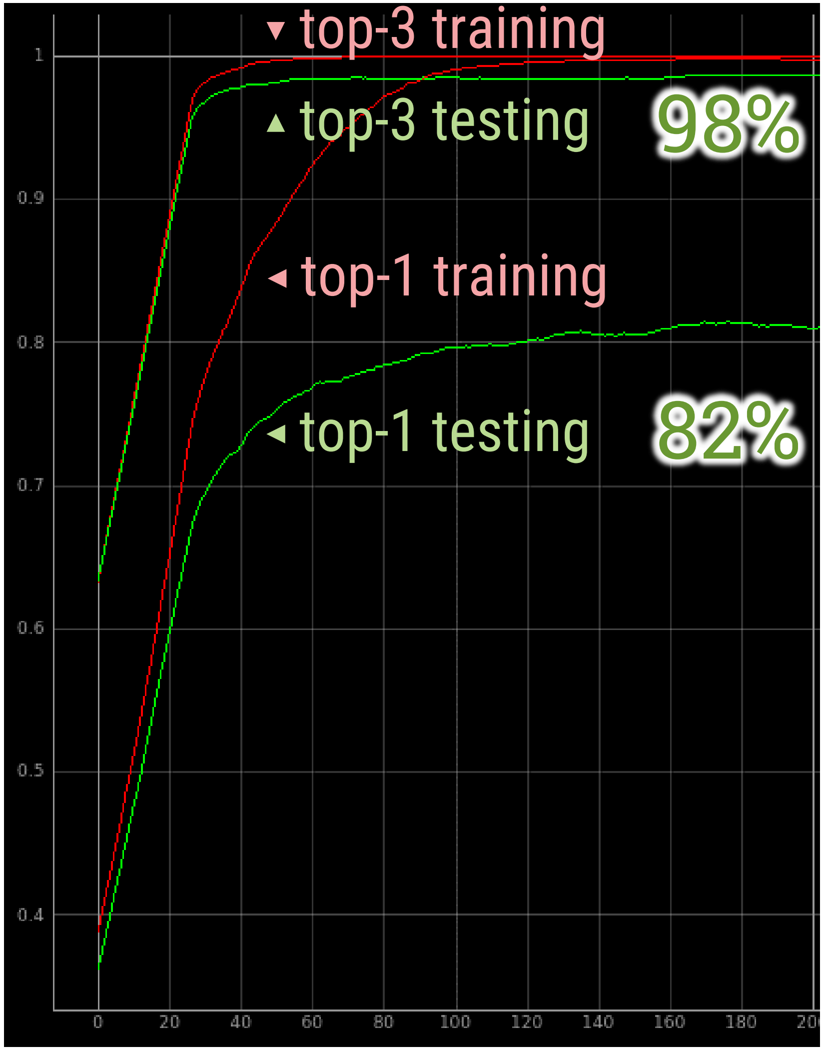
- Testing data ~ 82%

Top-3-accuracy

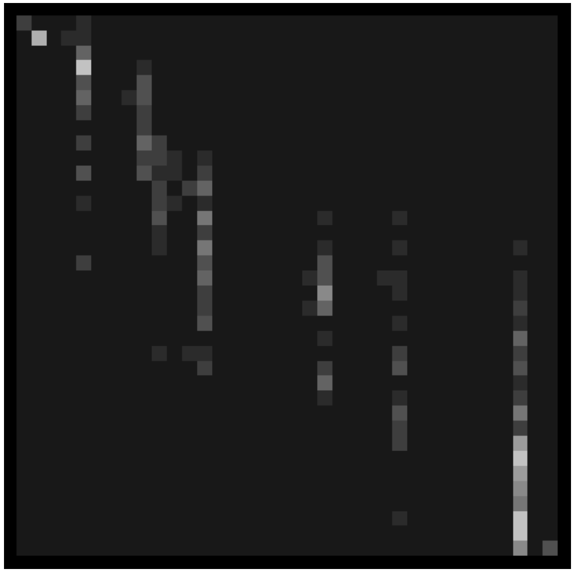
- Testing data ~ 98%

Training

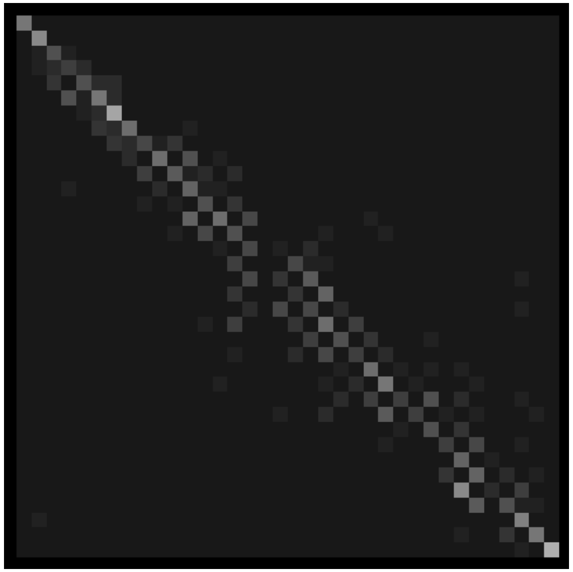
- Input: ~150 CT scans
- Training time: 10 min (dual Titan-X)



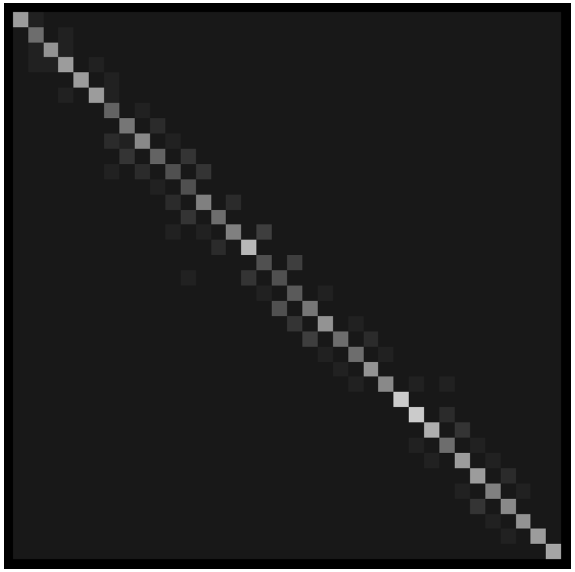
Confusion matrices



1 training step



5 training steps



200 training steps

Numerical Optimization

– Conventional Wisdom –

1^{st} Order: Gradient Descent

Gradient Descent:

- ∇E = direction of steepest ascent
  - Take small steps in direction $-\nabla E$
  - When $\nabla E = \mathbf{0}$, a critical point is found.
- Small enough steps guarantee convergence
  - In theory
  - In practice: usually slow, unstable
  - Does not work for ill-conditioned problems

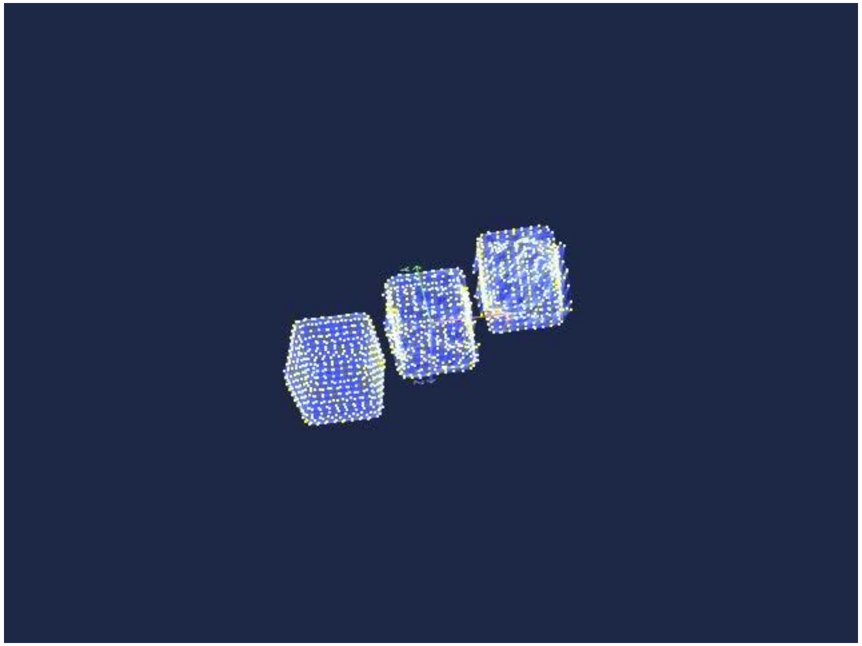
Line Search

Gradient descent line search

- Step size for gradient descent
  - Fit 1D parabola to *E* in gradient direction
  - Perform 1D Newton search
  - If E does not decrease at the new position
    - Try to half step width (say up to 10-20 times).
    - If this still does not decrease *E*, stop and output local minimum.

Gradient Descent goes Boomboomboom

Gradient Descent can be unstable



- Example:
  - Rigid object
  - Modeled by stiff springs
  - Bad conditioned problem
- Gradient descent cannot solve it in float32!
  - Either no progress or explosion
- Newton Method works fine
  - Converges in 5 steps, no boom

2^{nd} Order Non-Linear Solvers

Newton optimization

- Iteratively solve linear problems
- 2nd order Taylor expansions. Requires:
  - Function values
  - Gradient
  - Hessian matrix
- Typically, Hessian matrices are sparse.
  - Should be SPD (otherwise: trouble)
- Use conjugate gradients to solve for critical points

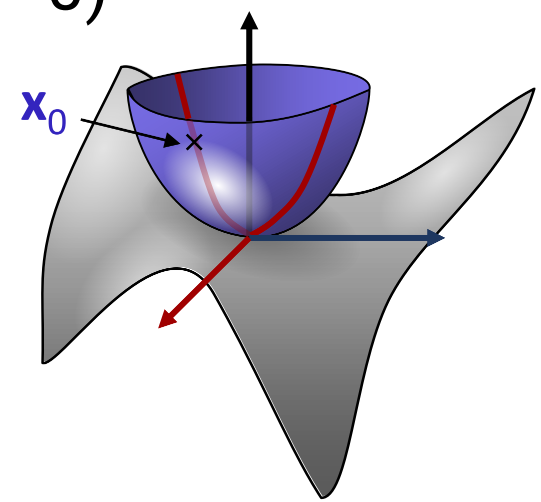
Newton Optimization

Newton Optimization

- Basic idea: Local quadratic approximation of E:

$$E(\mathbf{x}) \approx E(\mathbf{x}_0) + \nabla E(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^{\mathrm{T}} \cdot H_E(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0)$$

- Solve for vertex (critical point) of the fitted parabola
- Iterate until a minimum is found ($\nabla E = 0$)



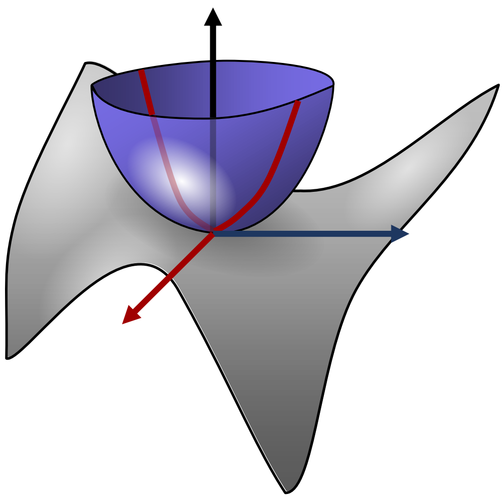
Properties:

- Typically much faster convergence, more stable
- No convergence guarantee

Newton Line Search

Line search for Newton-optimization:

- Following the quadratic fit might overshoot
- Line search:
  - Test value of E at new position
  - Half step width until error decreases (say 10-20 iterations)
  - Switch to gradient descent, if this does not work



Newton Optimization

Problem

- Steps might go uphill
- (Near-) zero or negative eigenvalues make problem ill-conditioned.

Simple solution

- Add $\lambda \mathbf{I}$ to the Hessian for a small λ.
- Sum of two quadrics: $\lambda \mathbf{I}$ keeps solution at $\mathbf{x}_0$.
- Comprehensive method: Levenberg-Marquant

What if I Hate Deriving the Hessian?

Gauss Newton

$$E(\mathbf{x}) = \sum_{i=1}^{n} f_i(\mathbf{x})^2 \Rightarrow \tilde{E}(\mathbf{x}) = \sum_{i=1}^{n} \left(\nabla f_i(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + f_i(\mathbf{x}_0)\right)^2$$

LBFGS

- "Quasi-Newton" method
- "Black box-solver"
  - Needs only gradient + function values

Non-linear conjugate gradients:

- With line search
- Usually faster than simple gradient decent

Numerical Optimization

– Big Data & Deep Learning –

“Big” Data

ImageNet LSVRC

- 1 000 000 Training images
- 1 000 Non-overlapping categories
- Resized to 224x224 pixels

Costs

- 147KB / image
- ca. 150GB/600GB image data (bytes / float32)

“Big” Data

Networks

AlexNet (2012):	62M	params	1.5 GFlops
VGG (2014):	138M	params	20 GFlops
Inception (2014):	6.5M	params	2 GFlops
ResNet-152 (2015):	60M	params	11 GFlops

(costs per forward-pass)

[https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96]

Training Algorithms

Gradient Descent

- Too expensive

Stochastic (Batch) Gradient Descent

- Sample only small batches (randomly)
- Small gradient descent steps
- No goodies
  - No 2^{nd} order information
  - Not even line search
- Fixed LR-schedule
  - “step decay”, typically $\lambda = 0.1, 0.001, 0.0001$
  - Fancy LR-schedules (e.g. “1-cycle”)

SGD Properties

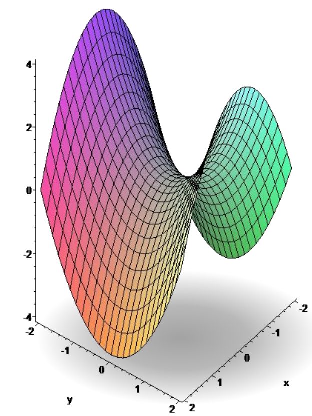
Interesting Properties

- Converges to GD for small enough steps
  - Accumulate gradients by small steps
- Noise from SGD “batching” *improves* learning
  - Better generalization for small batches / large LR
    - Only in the beginning
    - Always slow down later
    - Empirical result
  - Analytically
    - Cross-Entropy Loss + SGD increases margin
    - Similar to SVN (also: no need for hinge-loss)

More on SGD

Global minima?

- All of these only find local minima
- Problem seem to be saddle-points rather than local minima
  - Hand-wavy argument:
    "In high dim., hard to go up in all directions"
  - SGD is good at escaping saddle-points

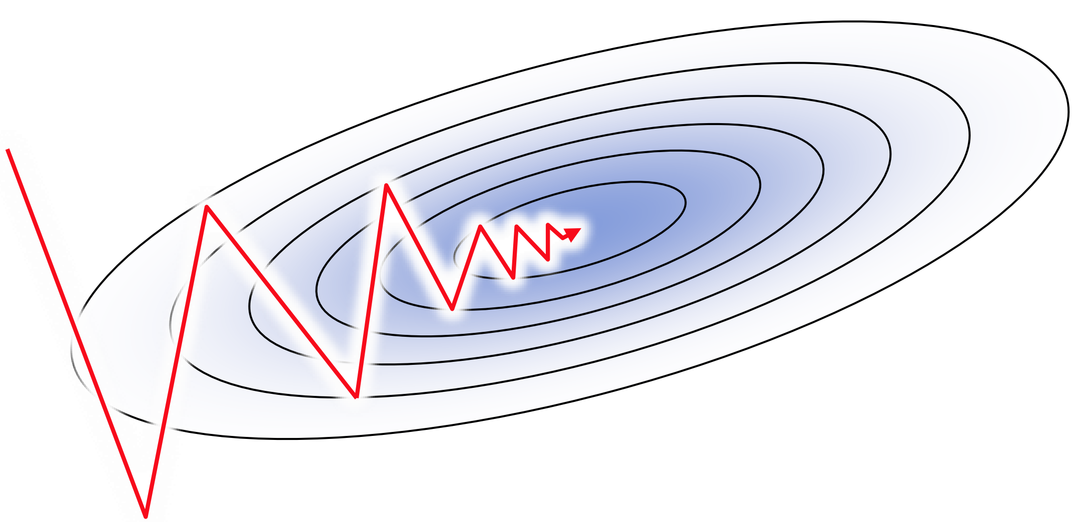


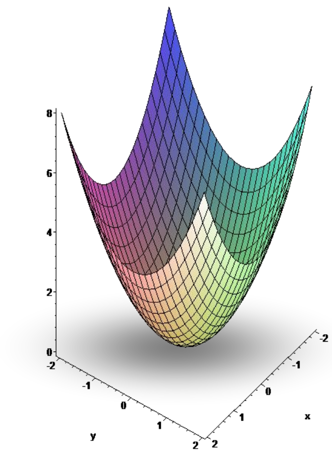
2nd-order Method

- Seems overall more expensive in a big-data setting
- Precaution needed wrt. saddle-points

Close to the minimum

Remember: GD does not work well





- Oscilatory behavior for anisotropic parabola
  - Fixed by conjugate gradients in numerics
- Simple trick for DL: “Momentum”

$$(\nabla_W f)^{i+1} = c(\nabla_W f)^i + (1-c)\widehat{\nabla_W f}$$

- Improves a bit, useful at the end of training

Many Other Methods

ADAM (popular)

- Adjust/normalize LR per layer
- 1st/2nd-order momentum-like terms

RMSProp, AdaGrad, etc.

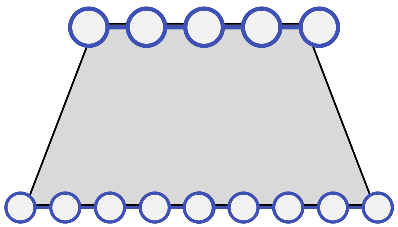
You can also use l-BFGS, if you like

How to solve general problems?

Central Building Block: Regression

Trained with Examples

Outputs

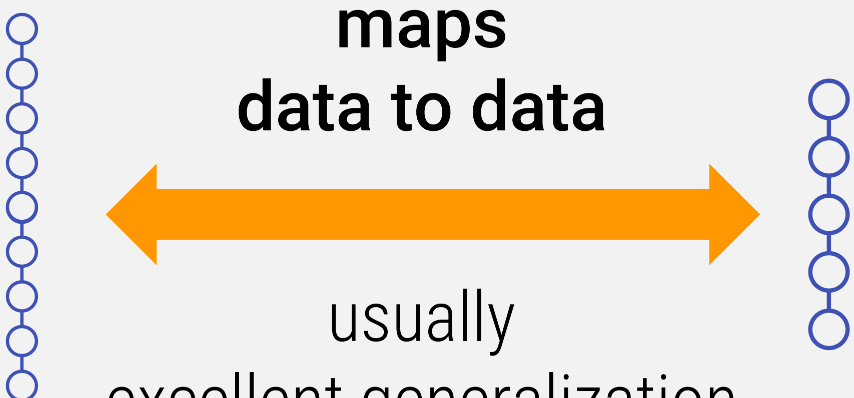


Inputs

Prediction $\in \mathbb{R}^m$

General Regressor

Data $\in \mathbb{R}^n$

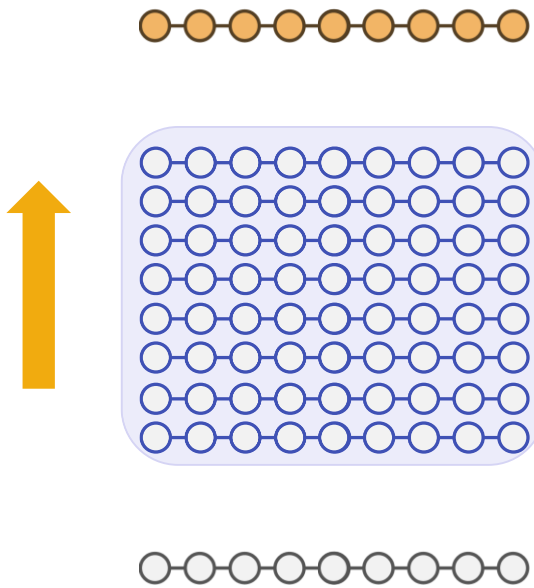


maps data to data

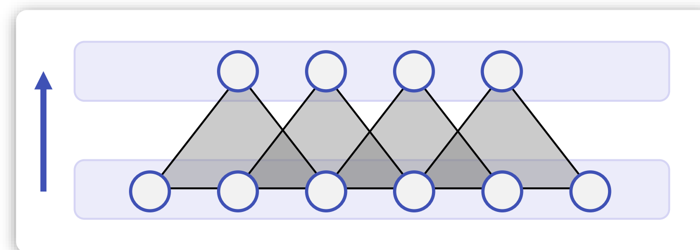
usually
excellent generalization
(*not clear why*)

Fully-Convolutional Network

Regression target $\in \mathbb{R}^n$

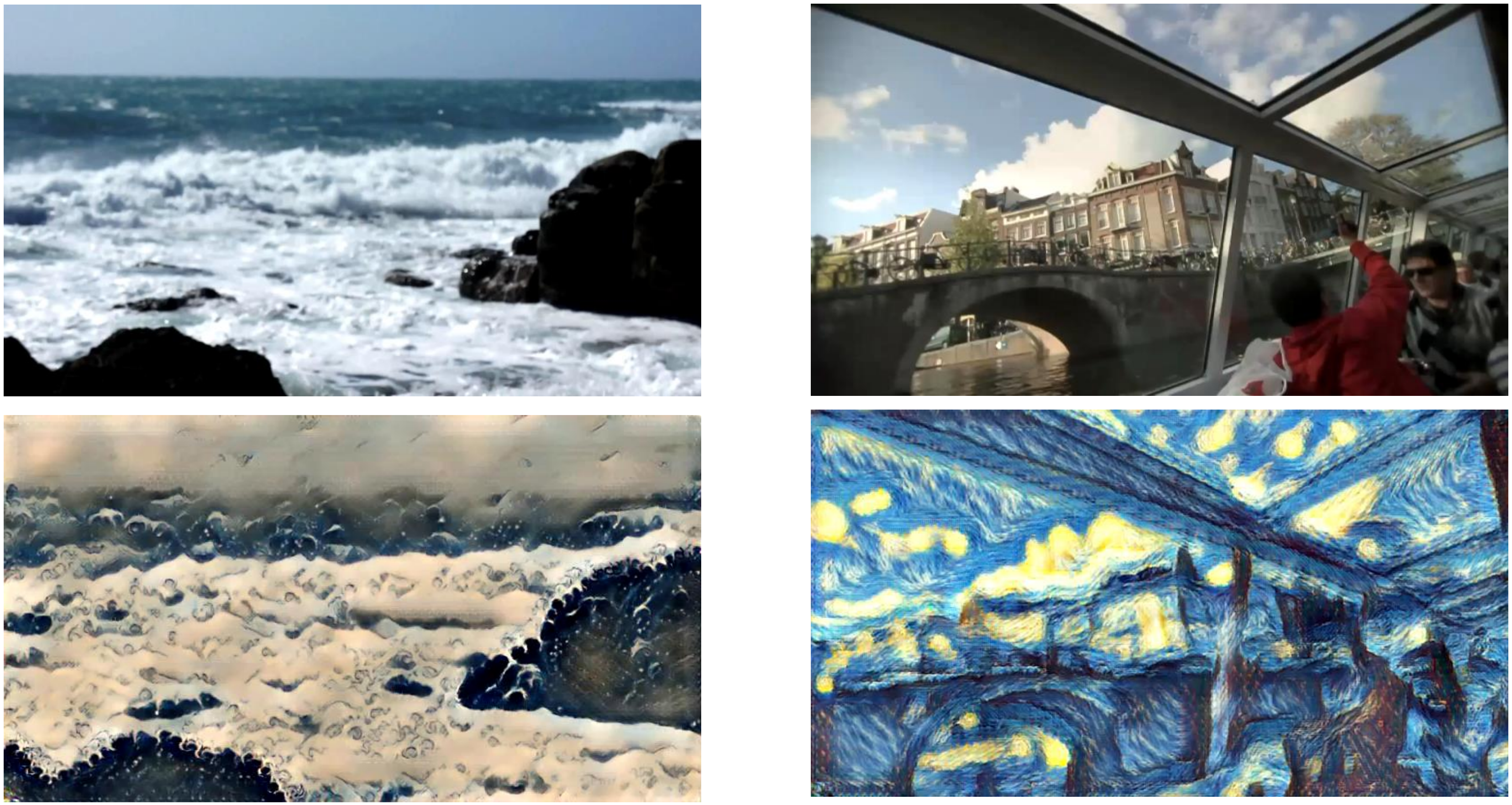


Input / source $\in \mathbb{R}^n$



convolutional layers

MGAN Style Transfer

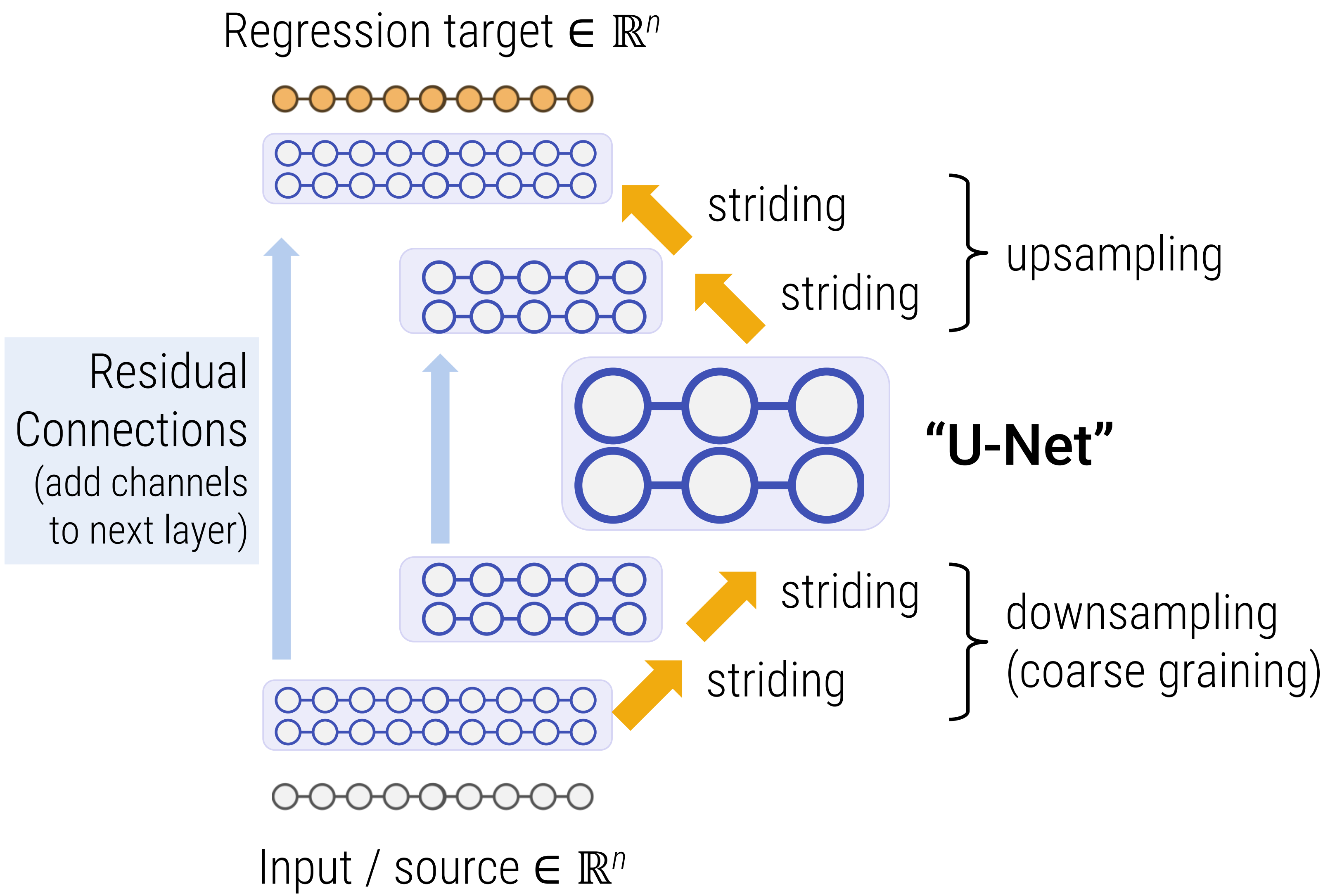


[joint work with Chuan Li, 2016]

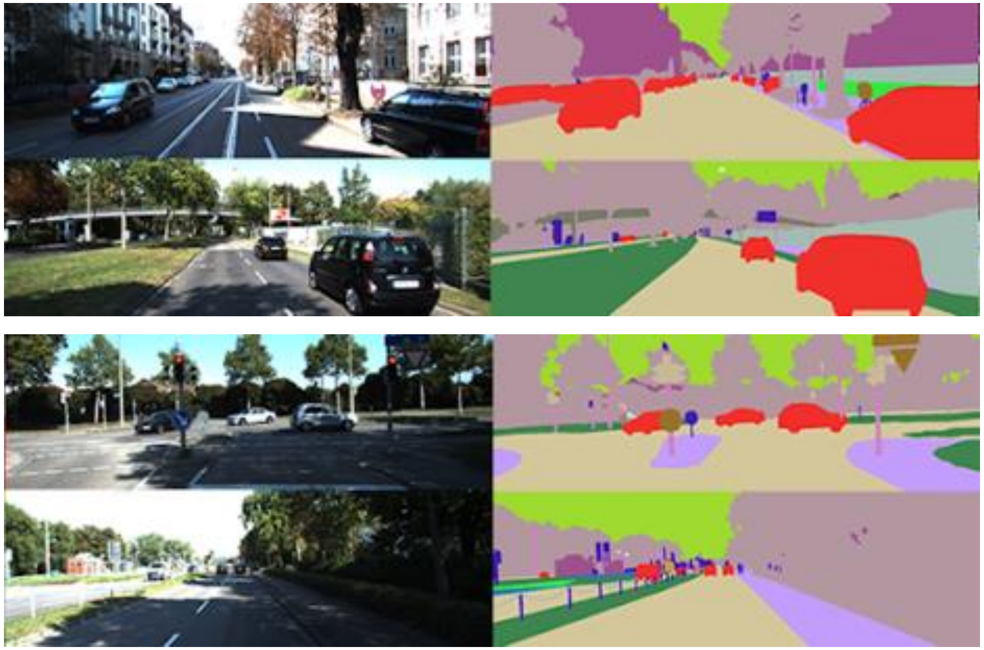
MGAN Style Transfer

[joint work with Chuan Li, 2016]

U-Net


Regression target ∈ $\mathbb{R}^n$
striding
striding
upsampling
Residual Connections (add channels to next layer)
"U-Net"
striding
striding
downsampling (coarse graining)
Input / source ∈ $\mathbb{R}^n$

Example: Segmentation



Fully-Convolutional Architectures

- Popular in image segmentation / annotation
- U-Net is the “Swiss-Army-Knife”

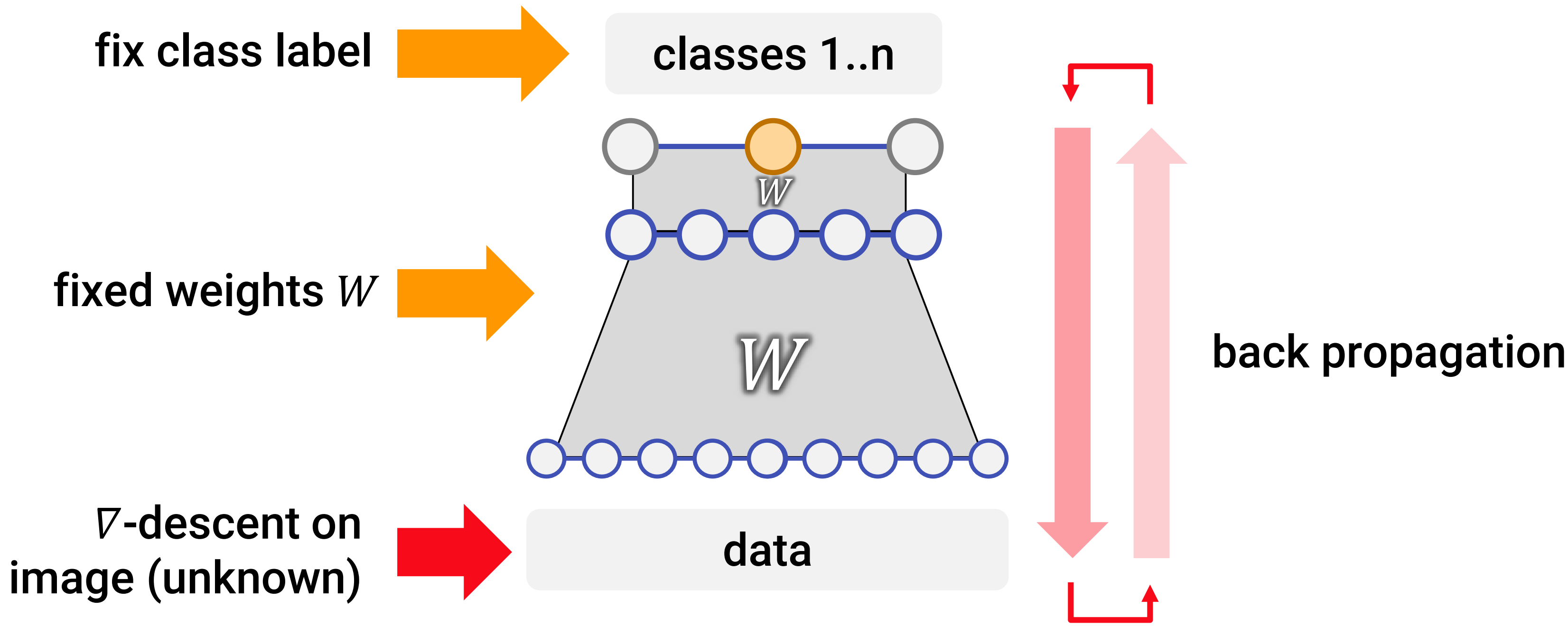
Example data from KITTI-Vision Benchmark Suite [Alhaija et al. 2018]
http://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015

What does it actually do?

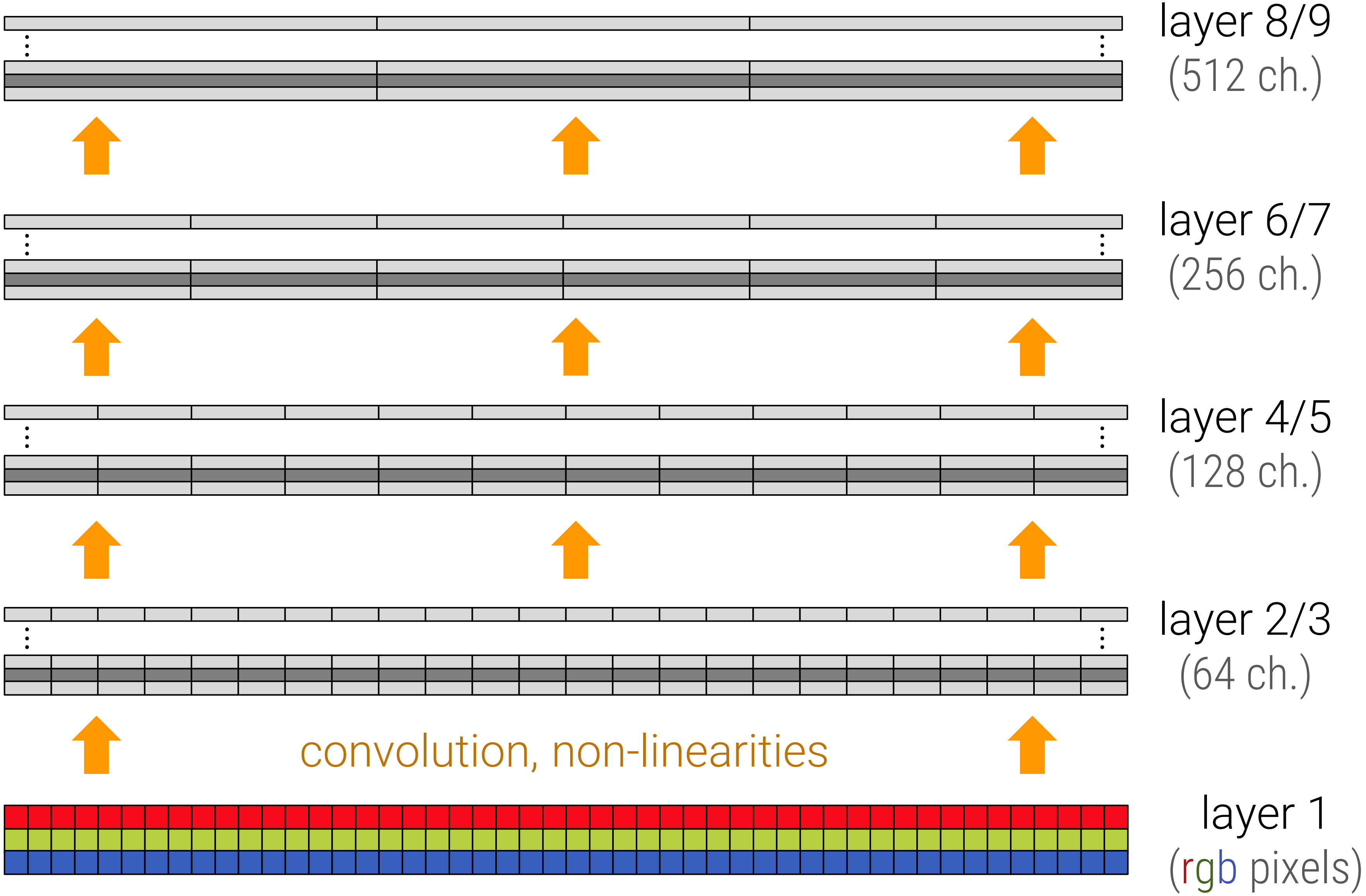
Variational Inversion

(also, we like pictures)

Variational Inversion



Deep-Network (Discriminative!)



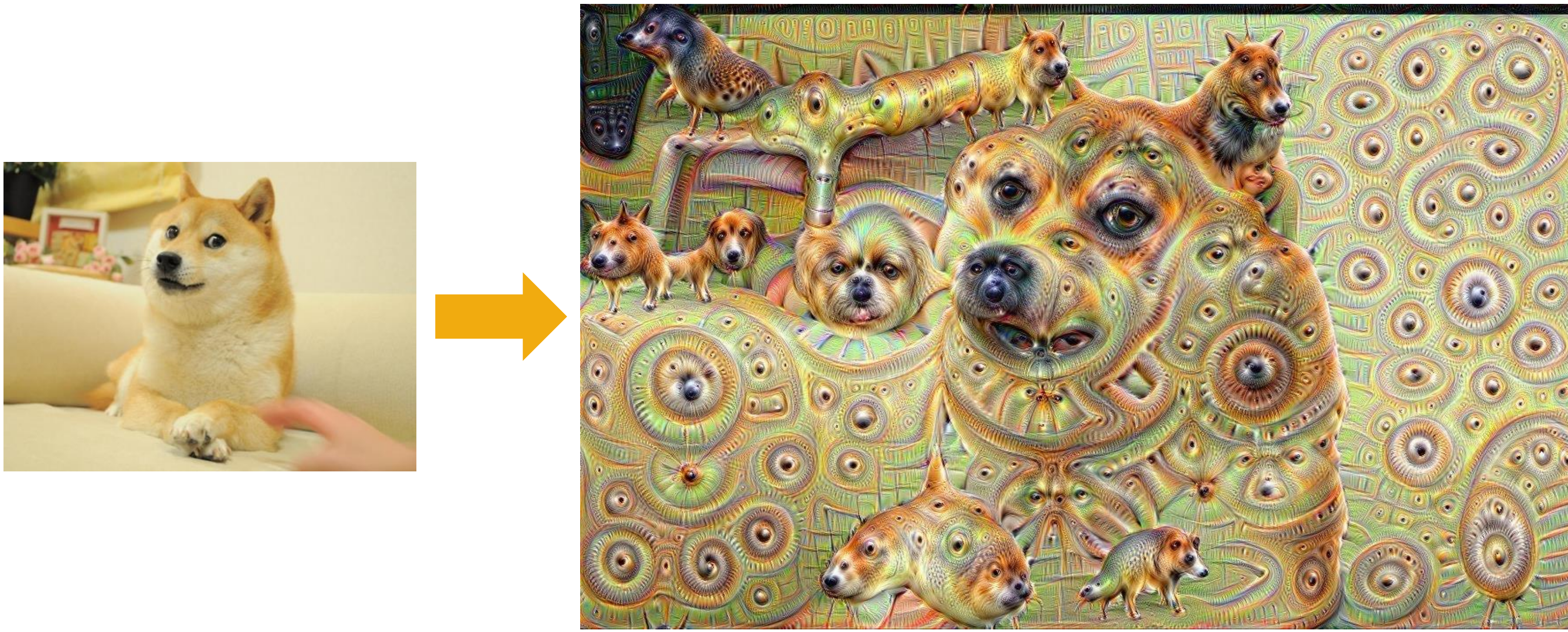
Google‘s „Deep Dream (Inceptionism)“ Algorithms

Image: Daniel Strecker

Google‘s „Deep Dream (Inceptionism)“ Algorithms

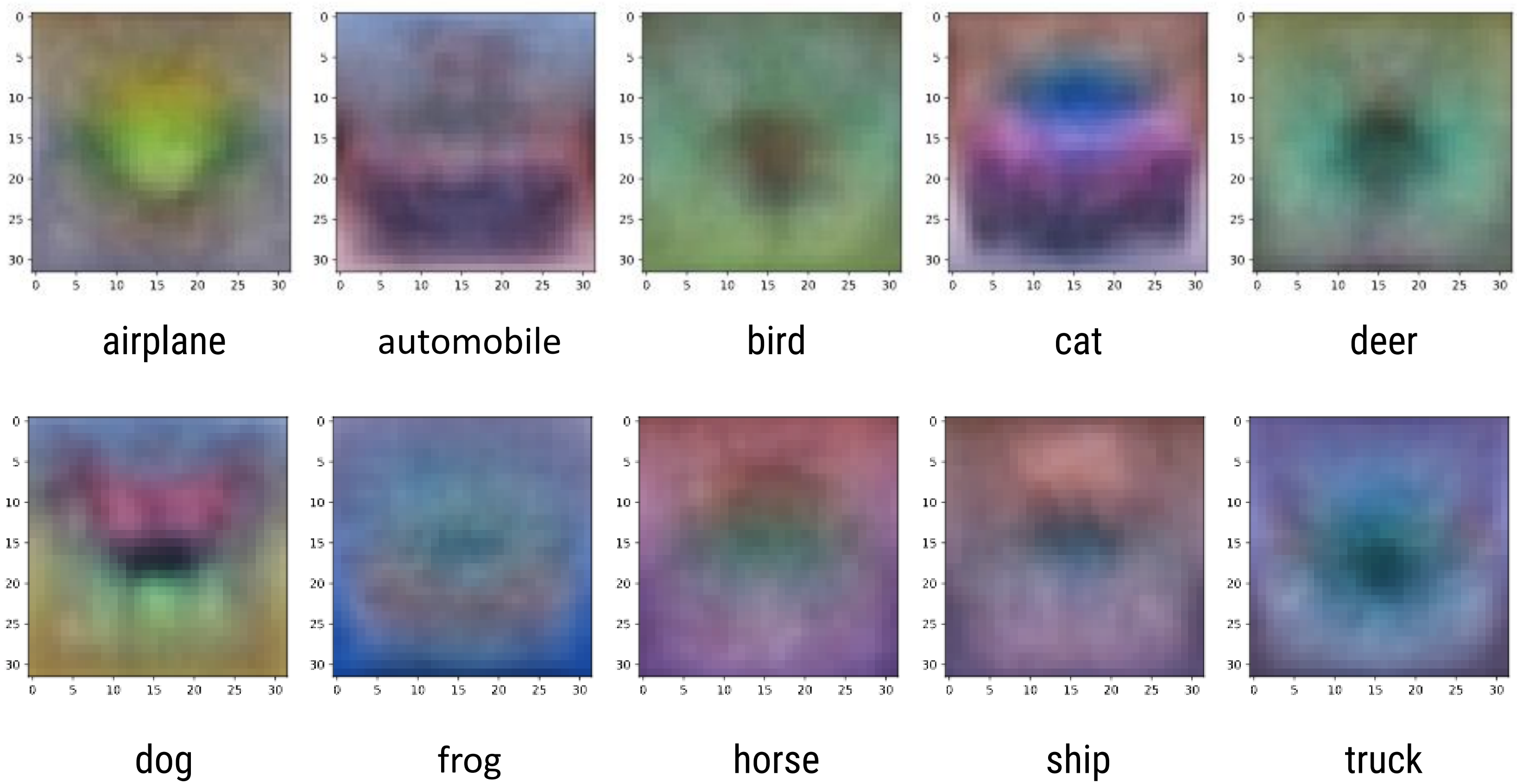
Image: Daniel Strecker

“Deep Dream”



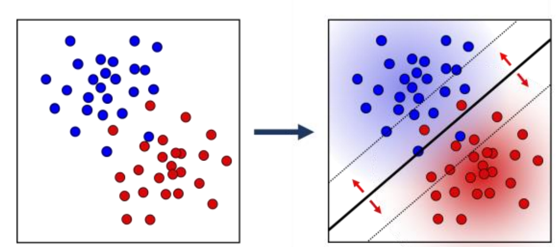
Source: Eric Wayne: “Google Deep Dream Getting Too Good”
https://artofericwayne.com/2015/07/08/google-deep-dream-getting-too-good

Linear SVM Dream (C=0.00001, L2/L2)

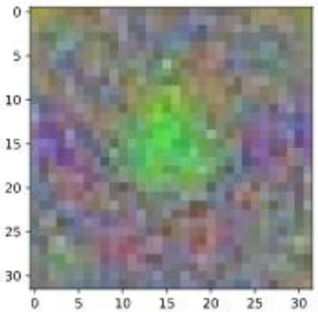


Accuracy

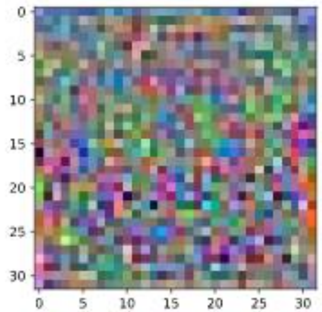
- Train: 37.2%, Test: 36.8%



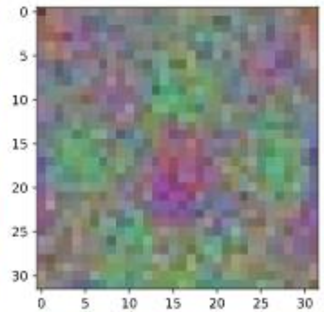
Linear SVM Dream (C=1.0, L2/L2)



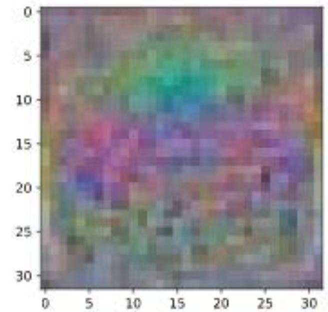
airplane



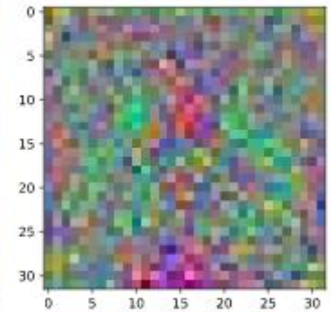
automobile



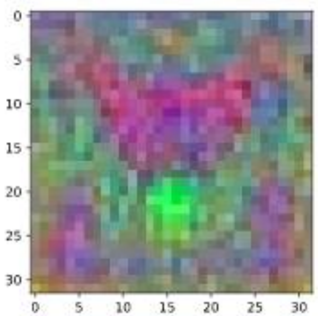
bird



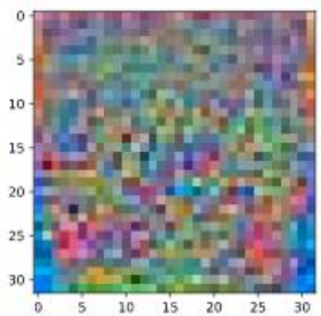
cat



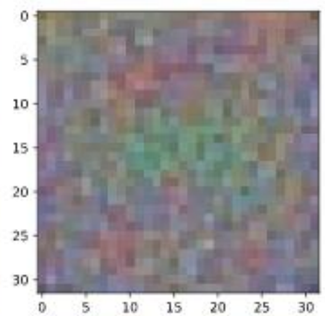
deer



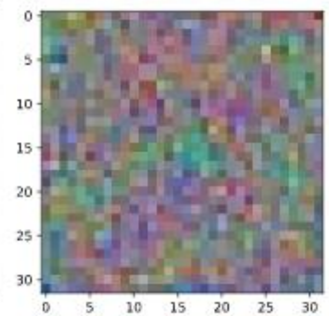
dog



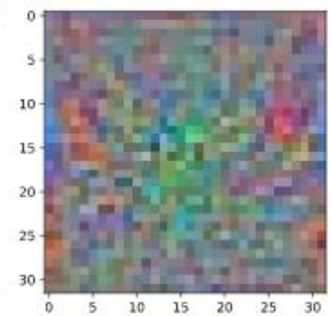
frog



horse



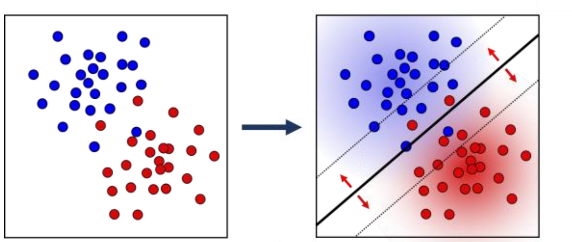
ship



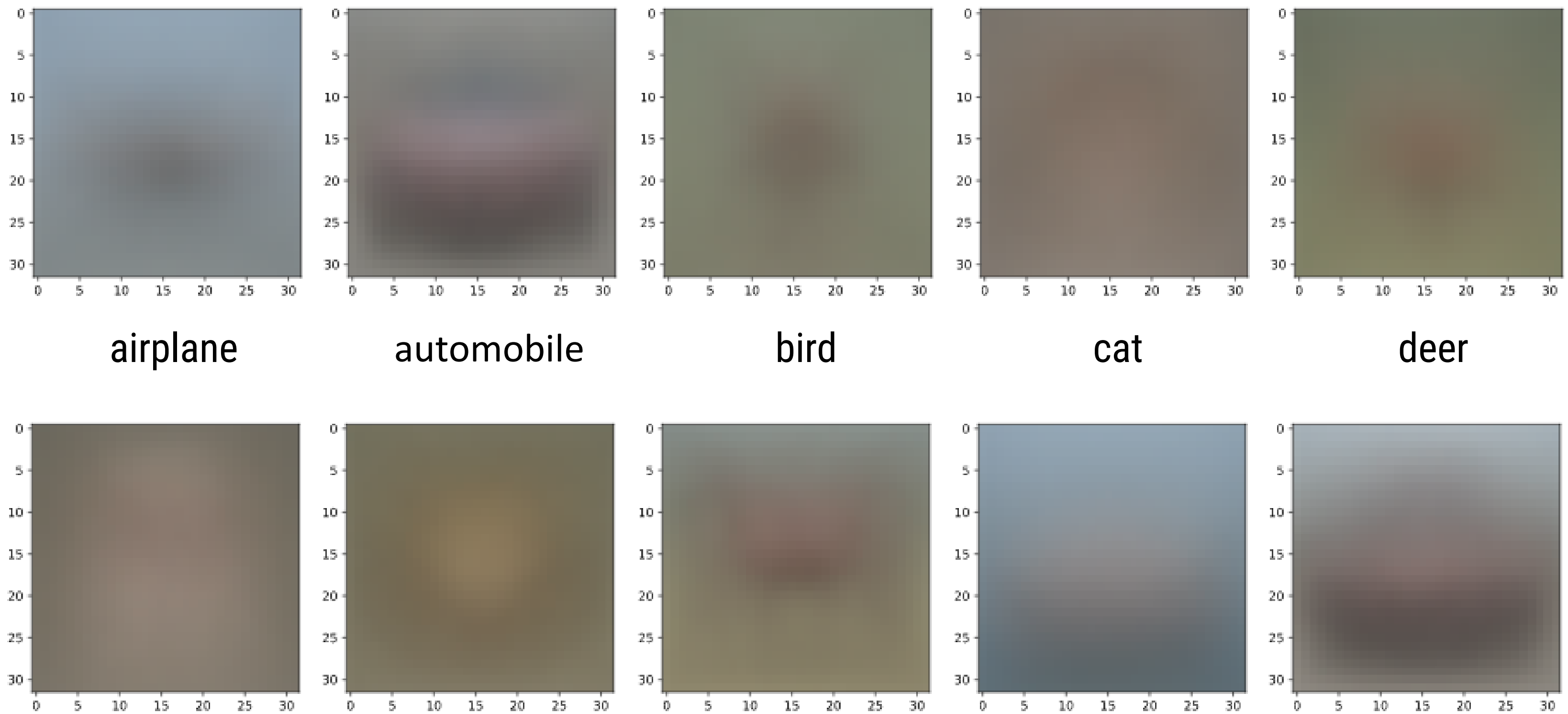
truck

Accuracy

- Train: 45.3%, Test: 39.8%



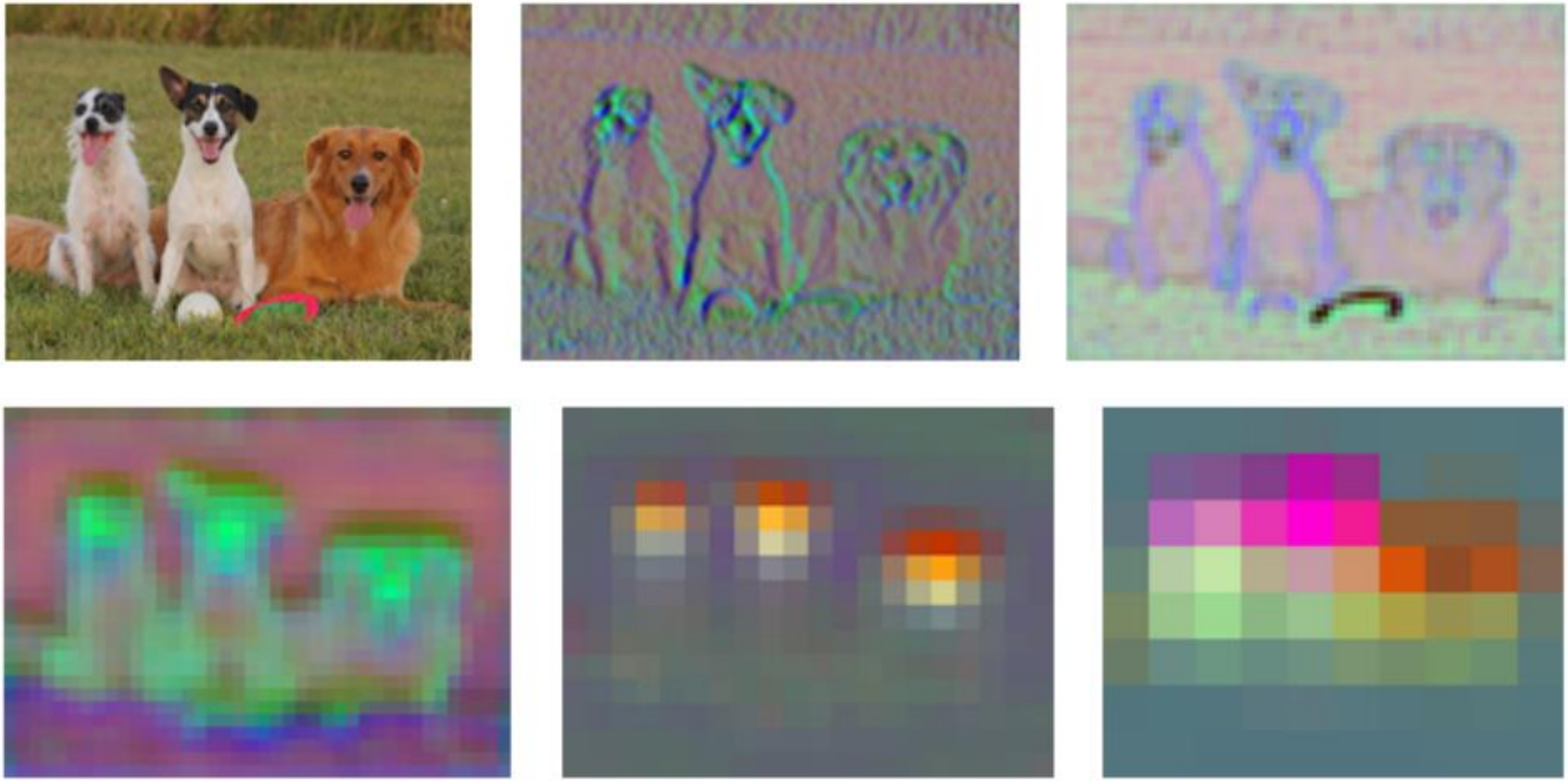
CIFAR-10 Class Averages



airplane automobile bird cat deer

dog frog horse ship truck

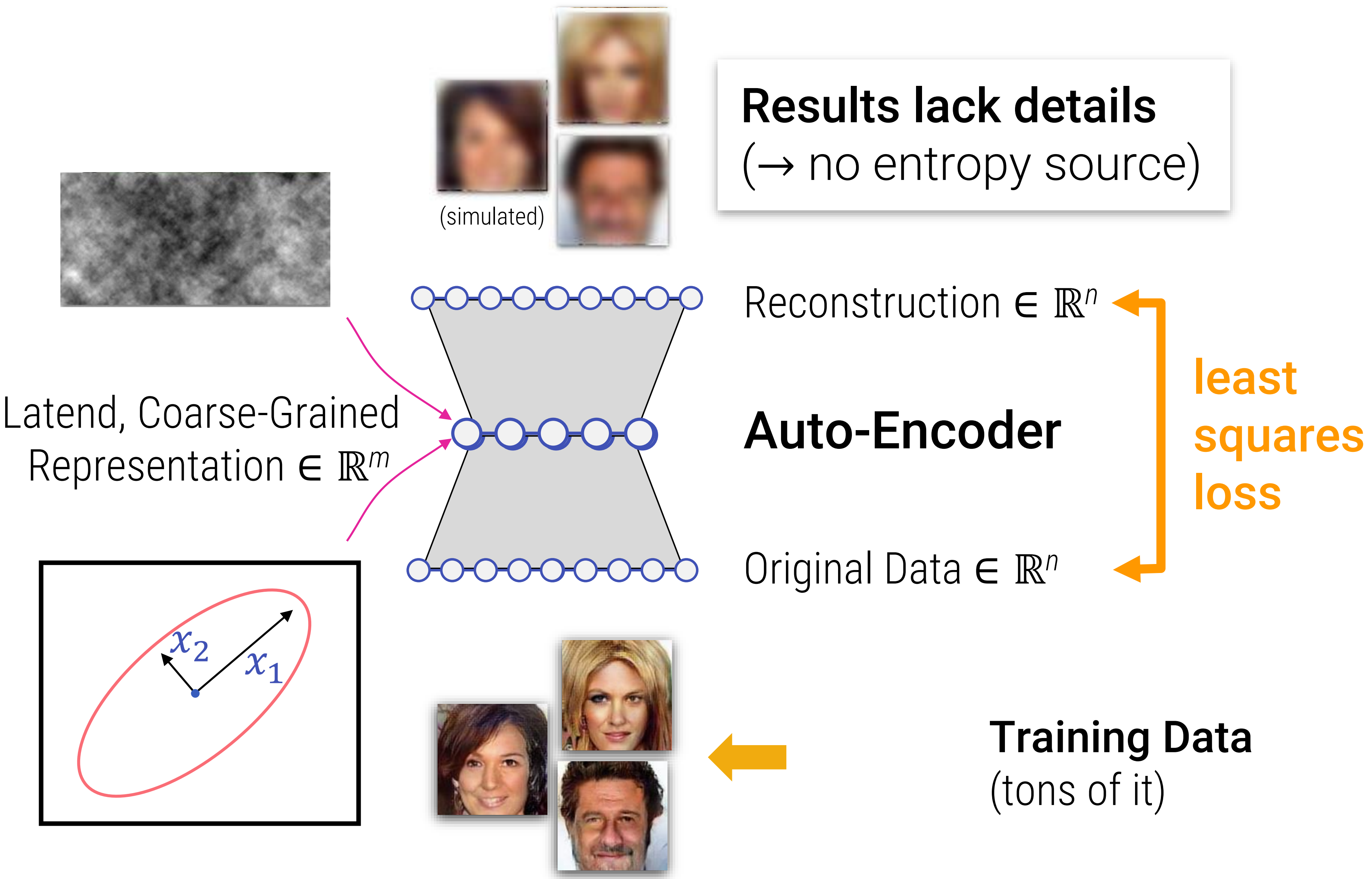
Dogs



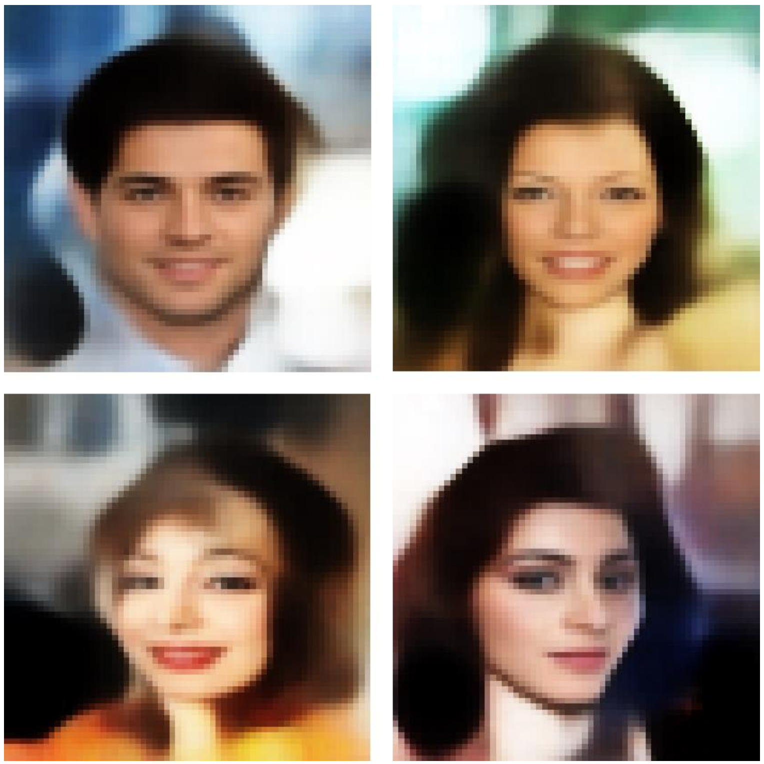
Autoencoders

Nonlinear Dimensionality Reduction

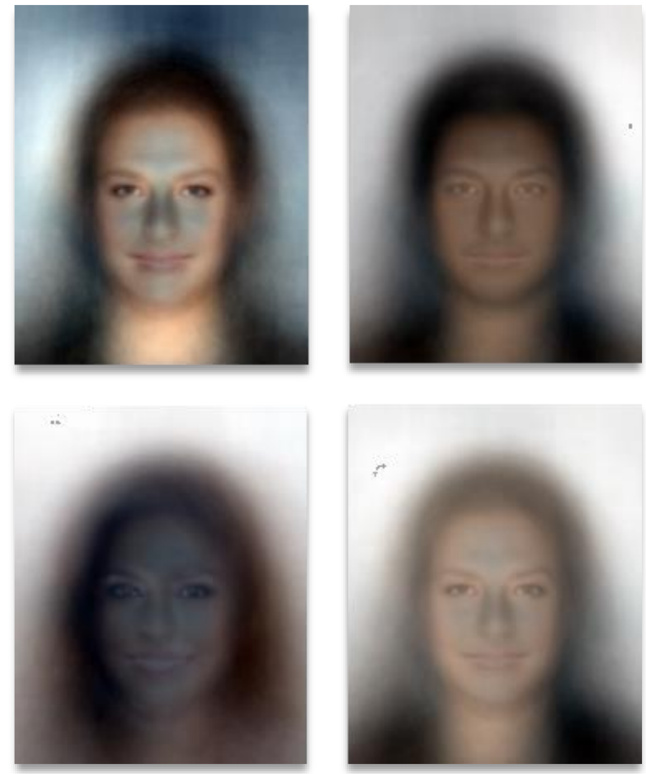
Auto-Encoder: Non-linear PCA



Example: Generative Models



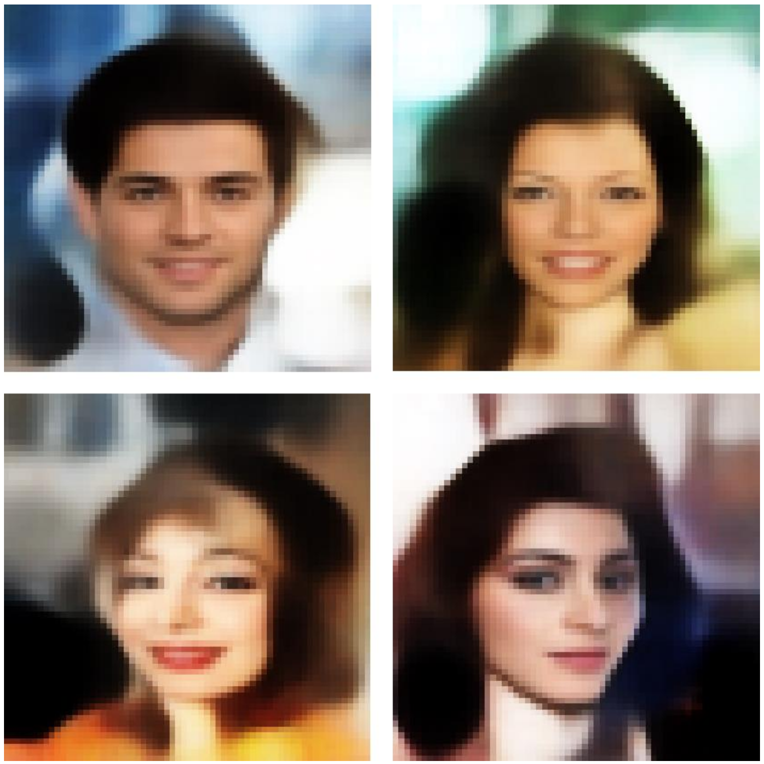
Autoencoder
(PCA in latent space)



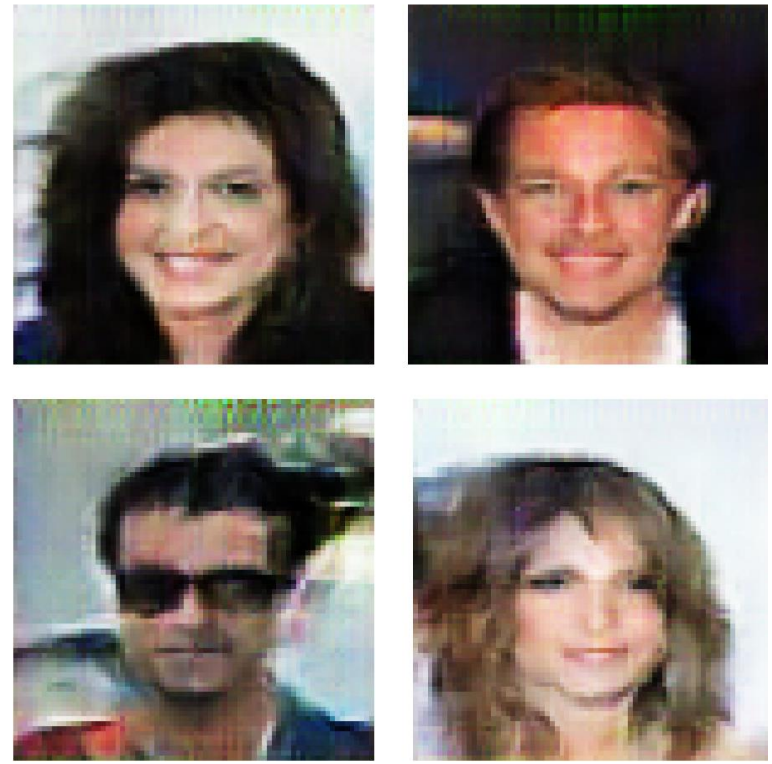
PCA
(linear dim. reduction)

[results courtesy of D. Schwarz, D. Klaus, A. Rübe]

Example: Generative Models



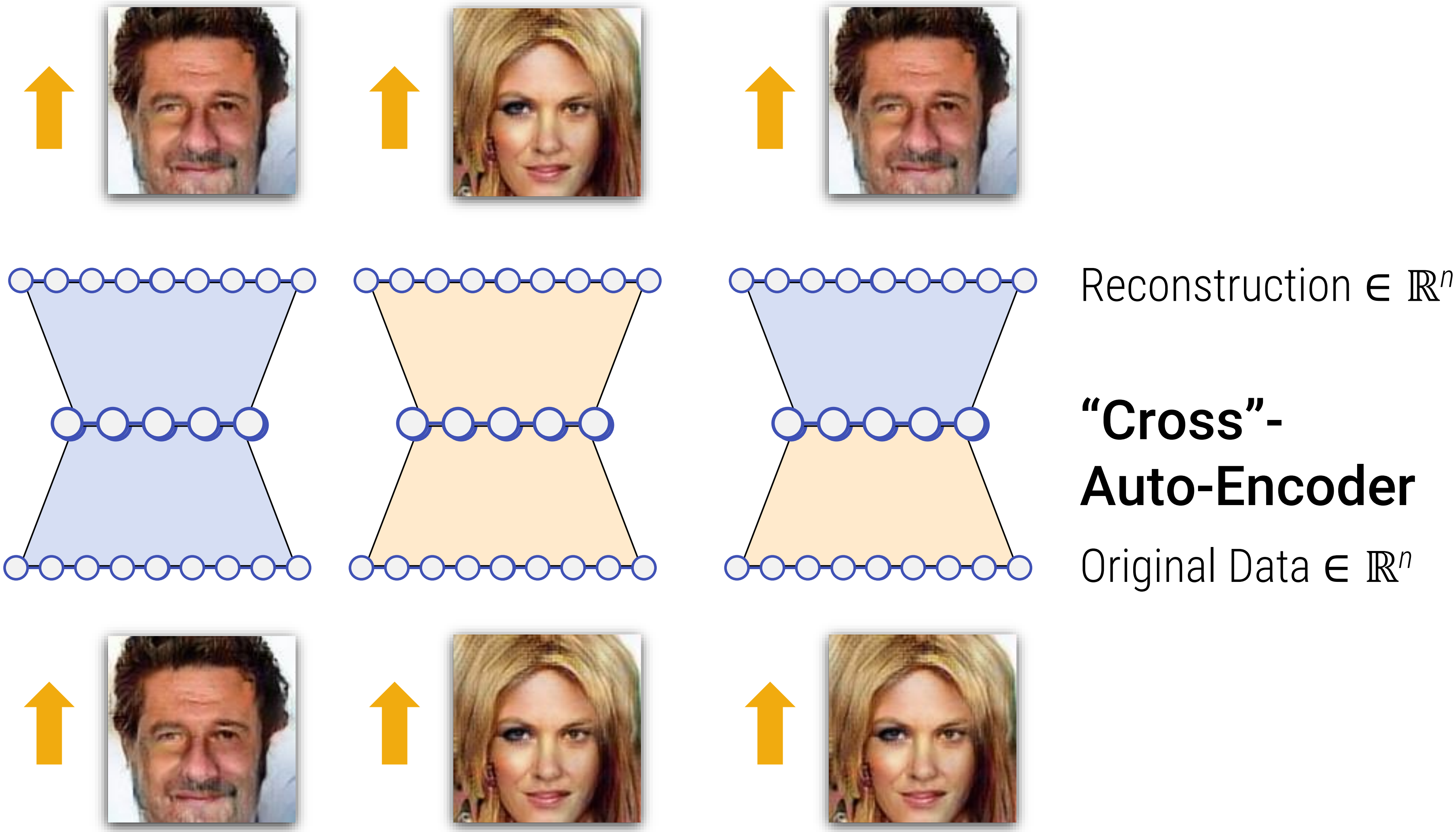
Autoencoder
(PCA in latent space)



WGAN-GP
(generative adversarial network)

[results courtesy of D. Schwarz, D. Klaus, A. Rübe]

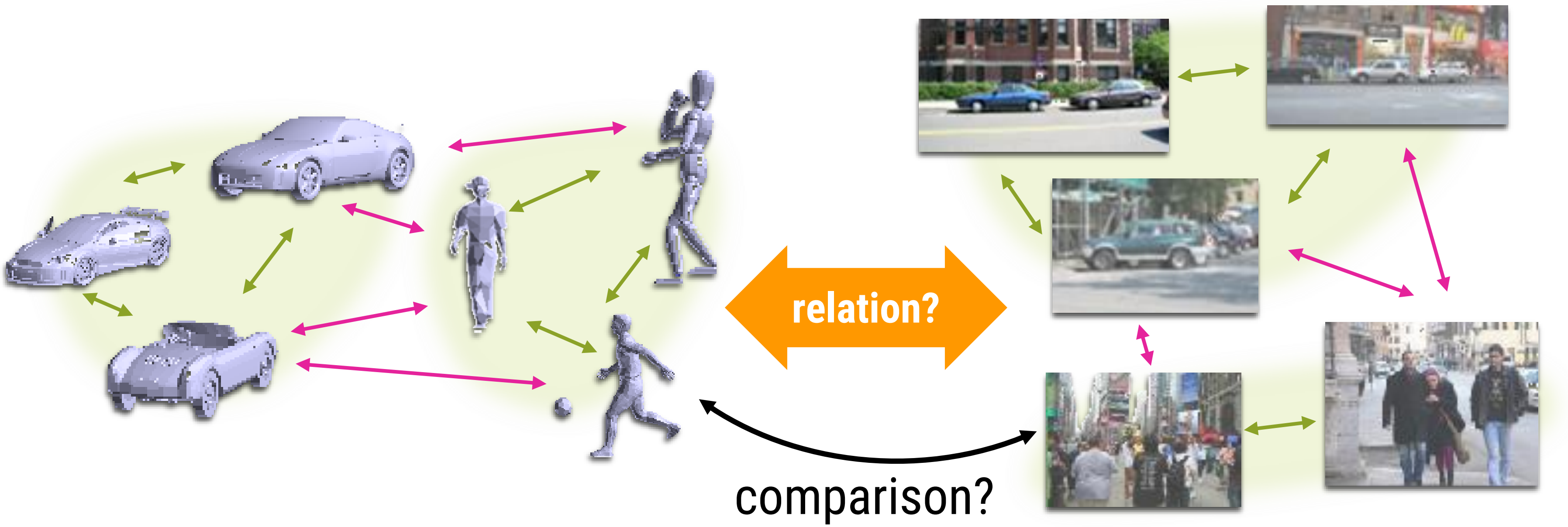
Cross Auto-Encoder



(Deep) Recommender Systems

(Siamese Network)

Relate Incomparable Data



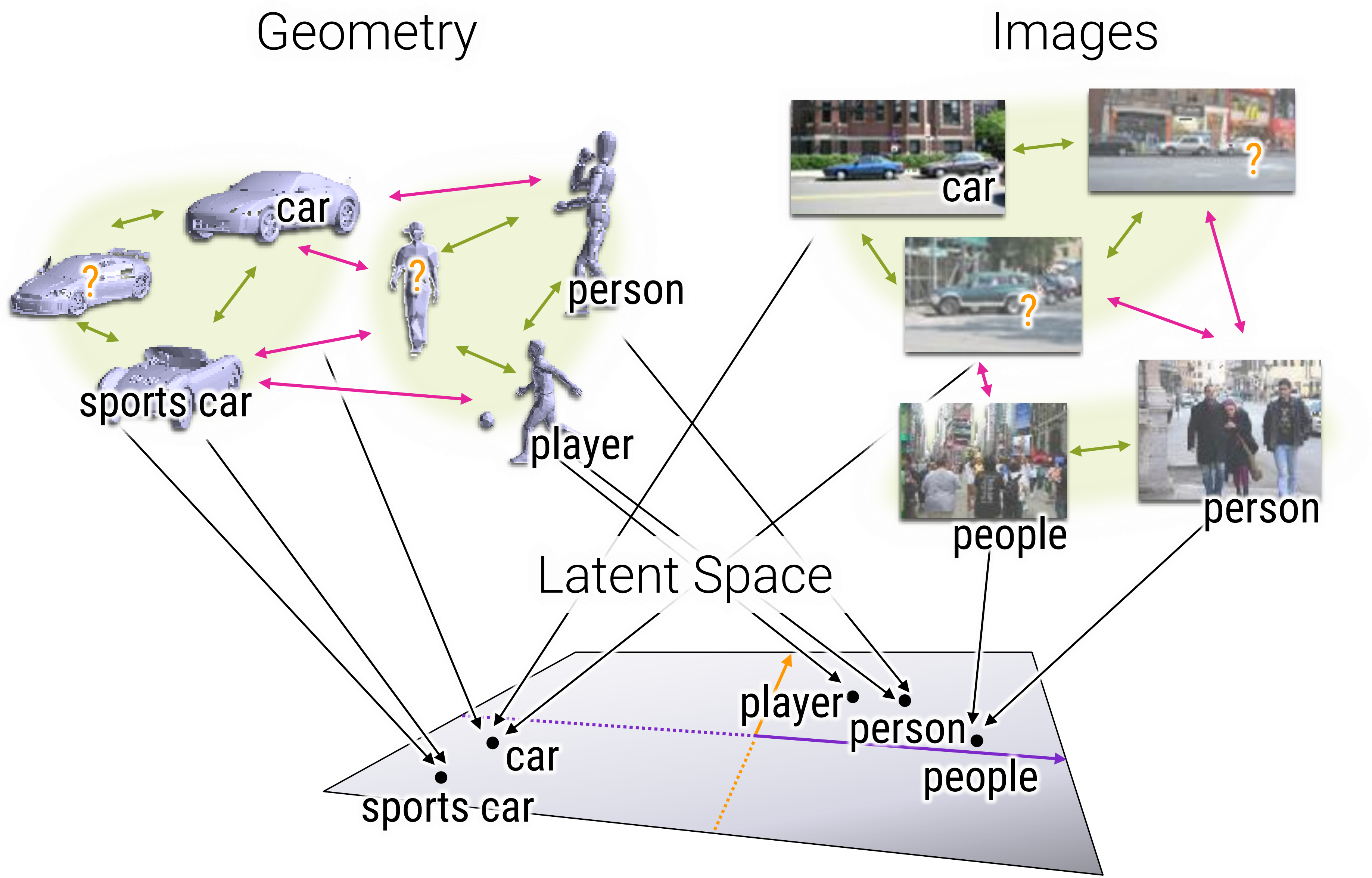
Relating Geometry

Relating Images

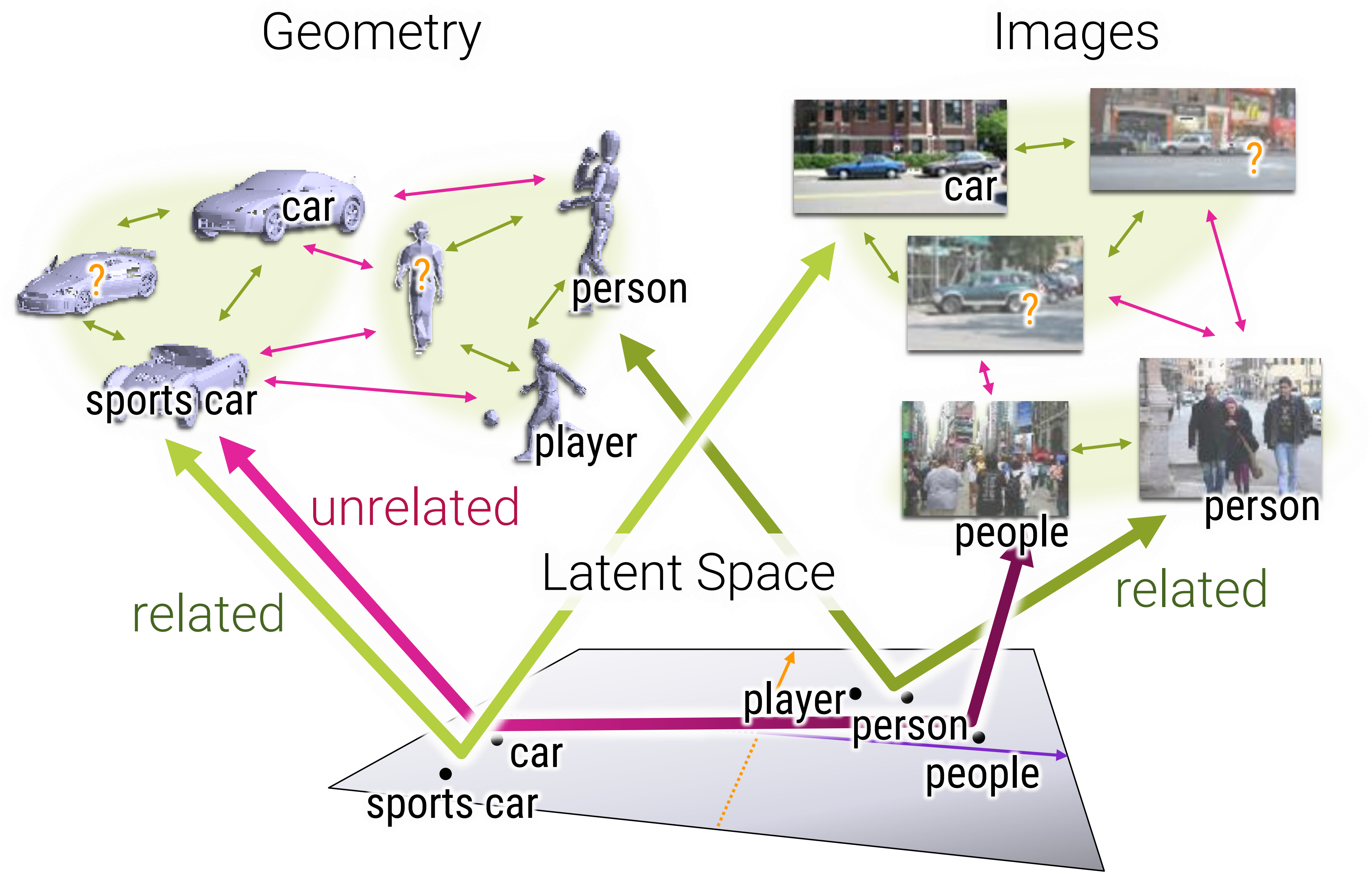
Problem

- Different modalities
- Direct comparison not meaningful

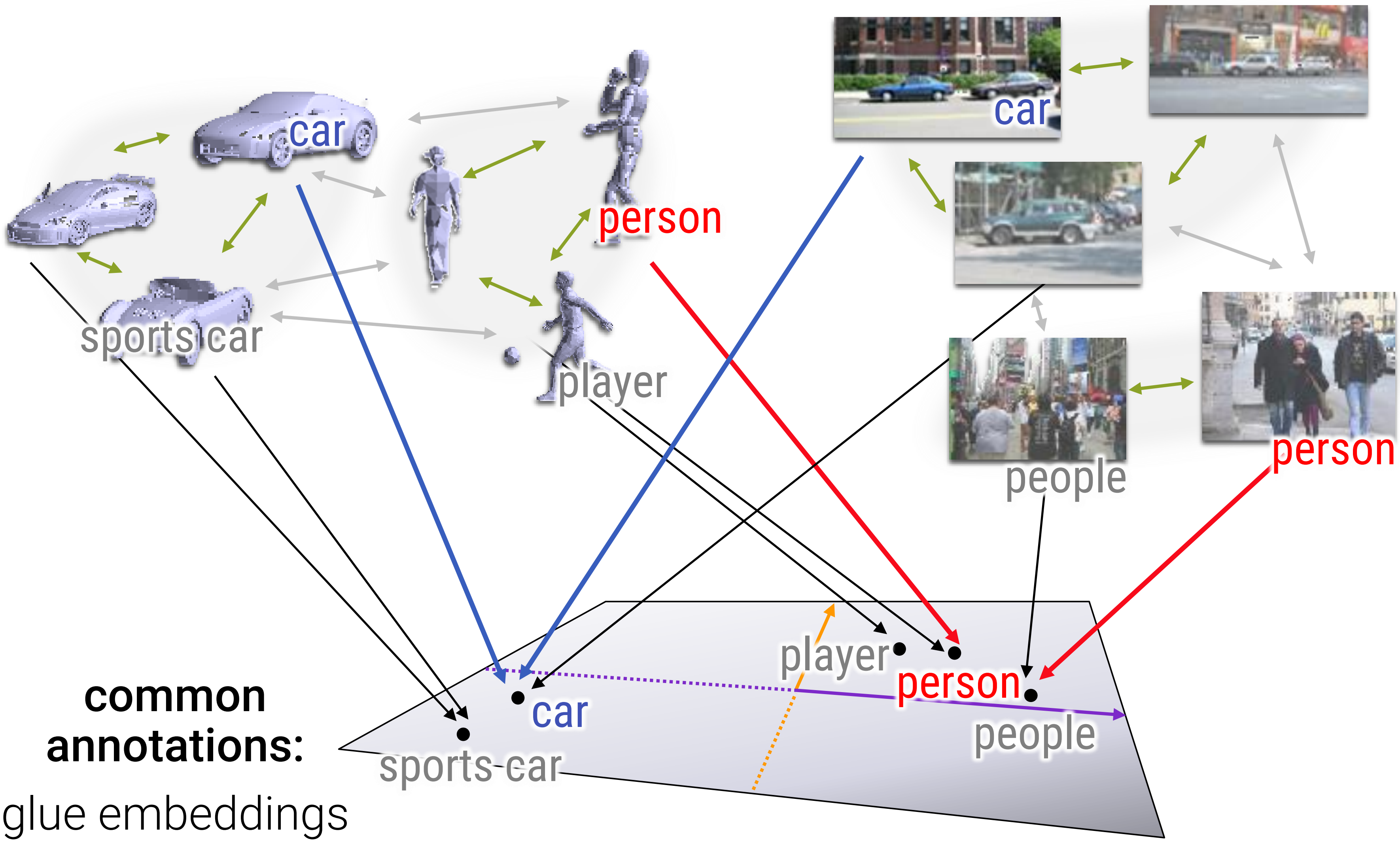
Latent Semantic Space



Latent Semantic Space



A Few Shared Annotations

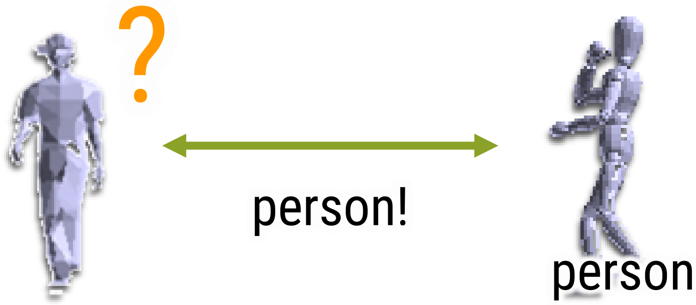


common annotations:

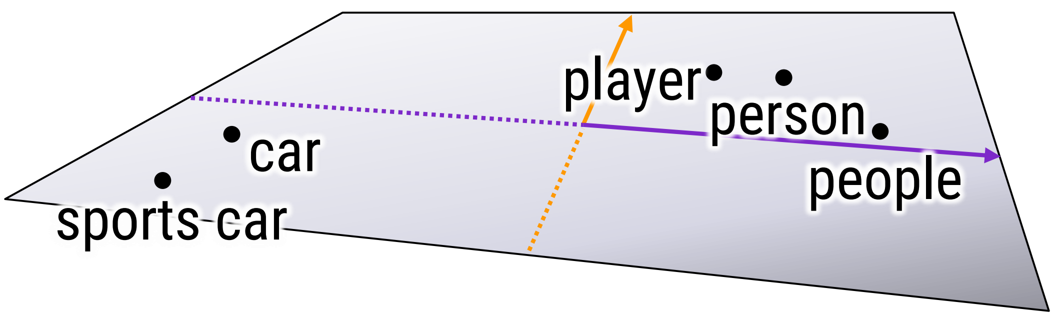
glue embeddings together

Information Gained

multi-modal correspondences

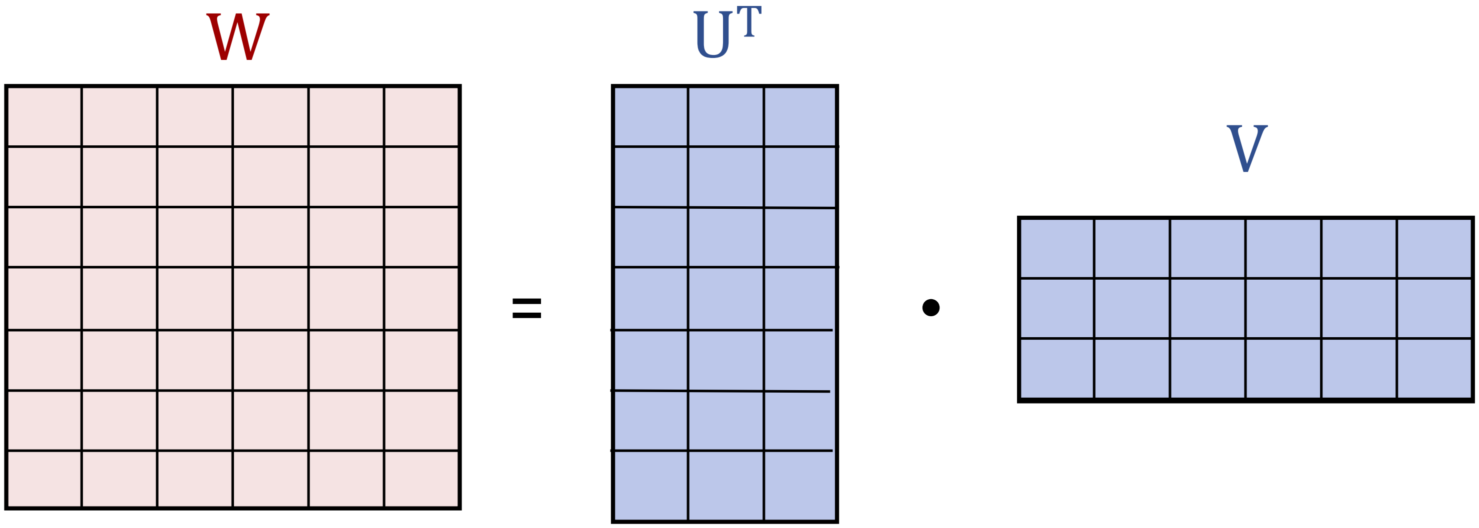


recognize unlabeled data



relate labels

Feature Sharing

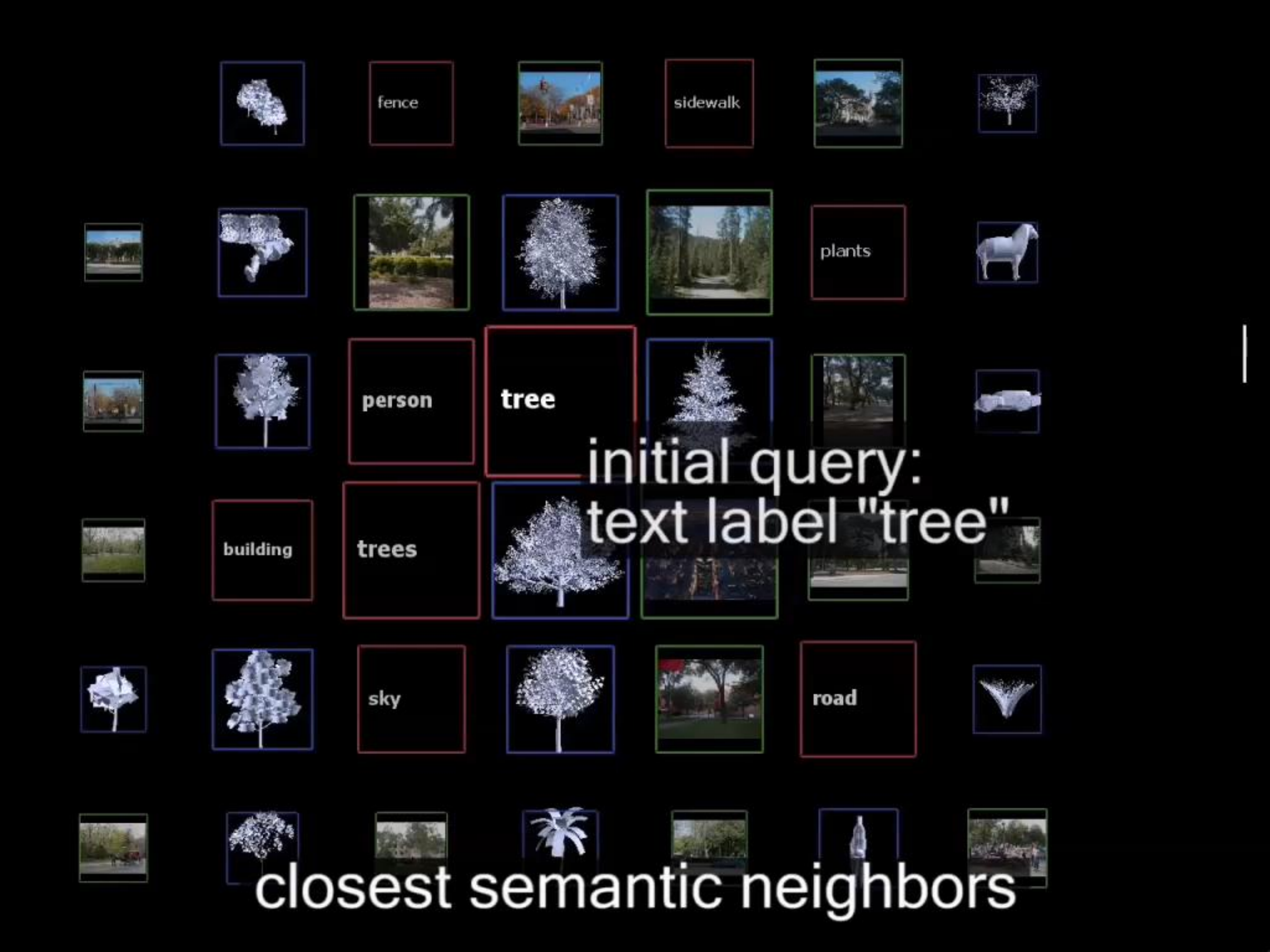


Two matrices

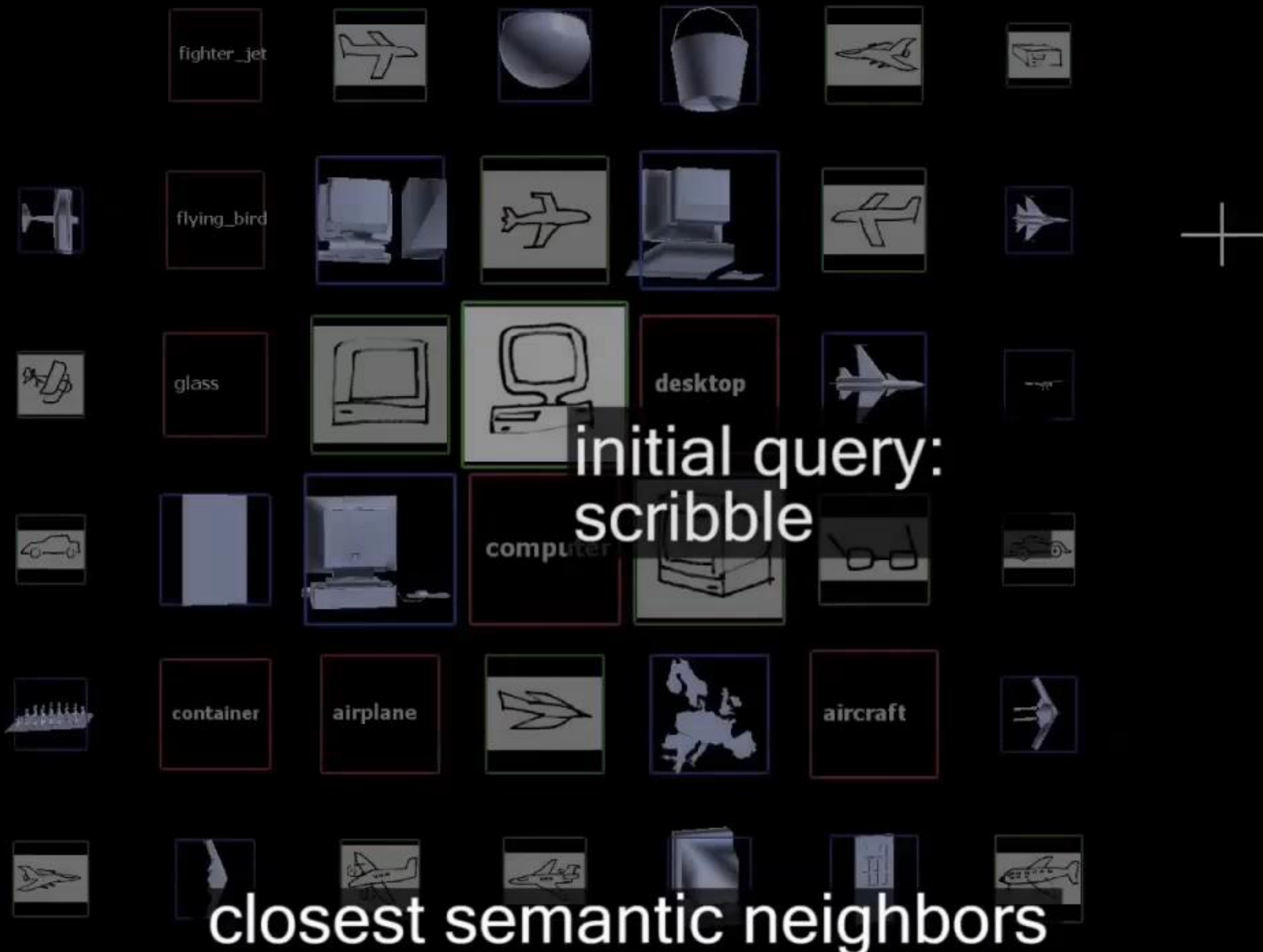
[Loeff & Farhadi 2008]

- **V** maps descriptors to latent space
- **U** maps labels $\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \ldots, \begin{pmatrix}0\\0\\1\end{pmatrix}$ to latent space

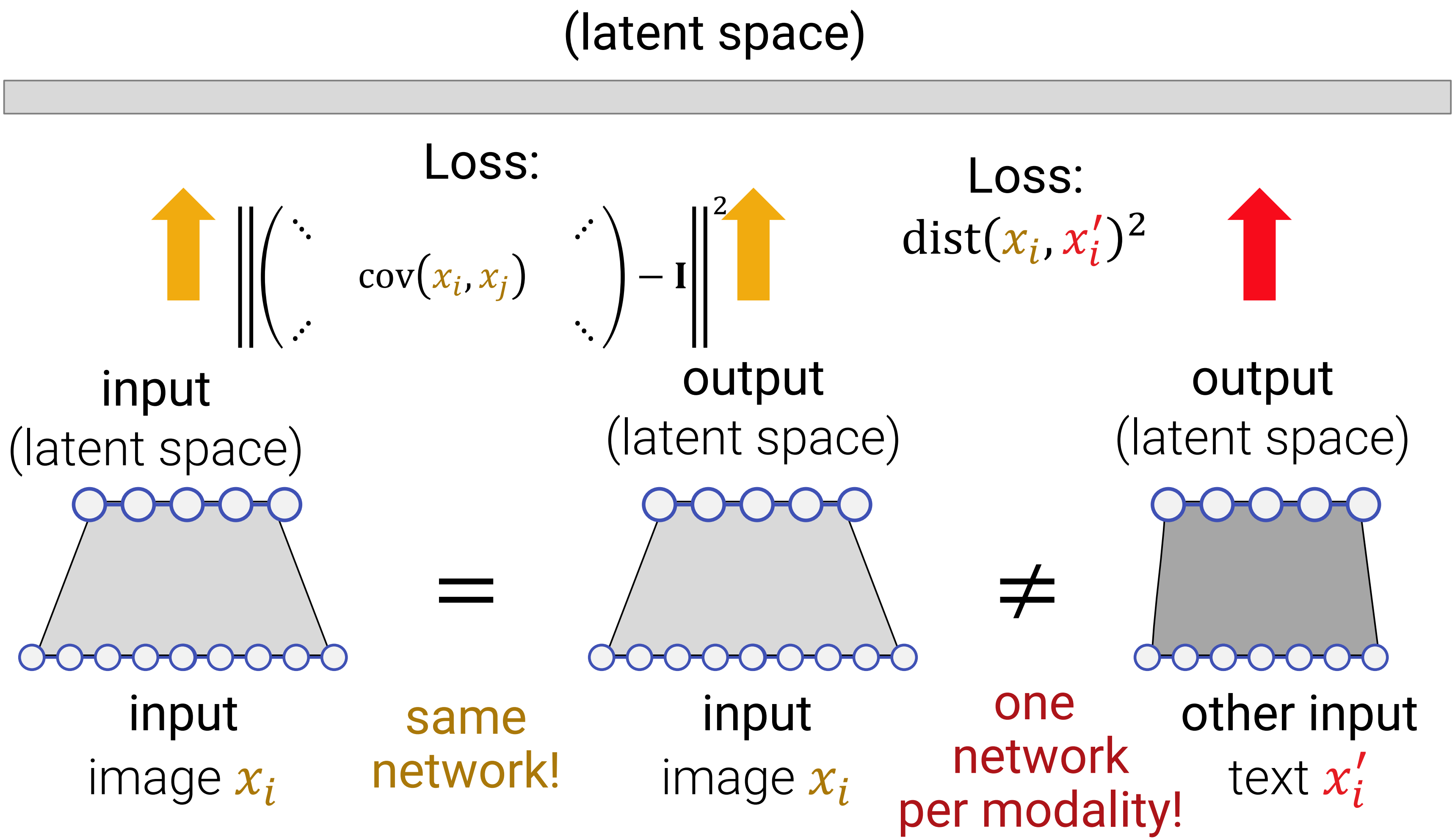

red
apple
strawberry
green
orange
banana
melon
lemon
yellow



fence
sidewalk
plants
person
tree
initial query:
text label "tree"
building
trees
sky
road
closest semantic neighbors


fighter_jet
flying_bird
glass
desktop
initial query:
scribble
container
airplane
aircraft
closest semantic neighbors

Siamese Network



Summary

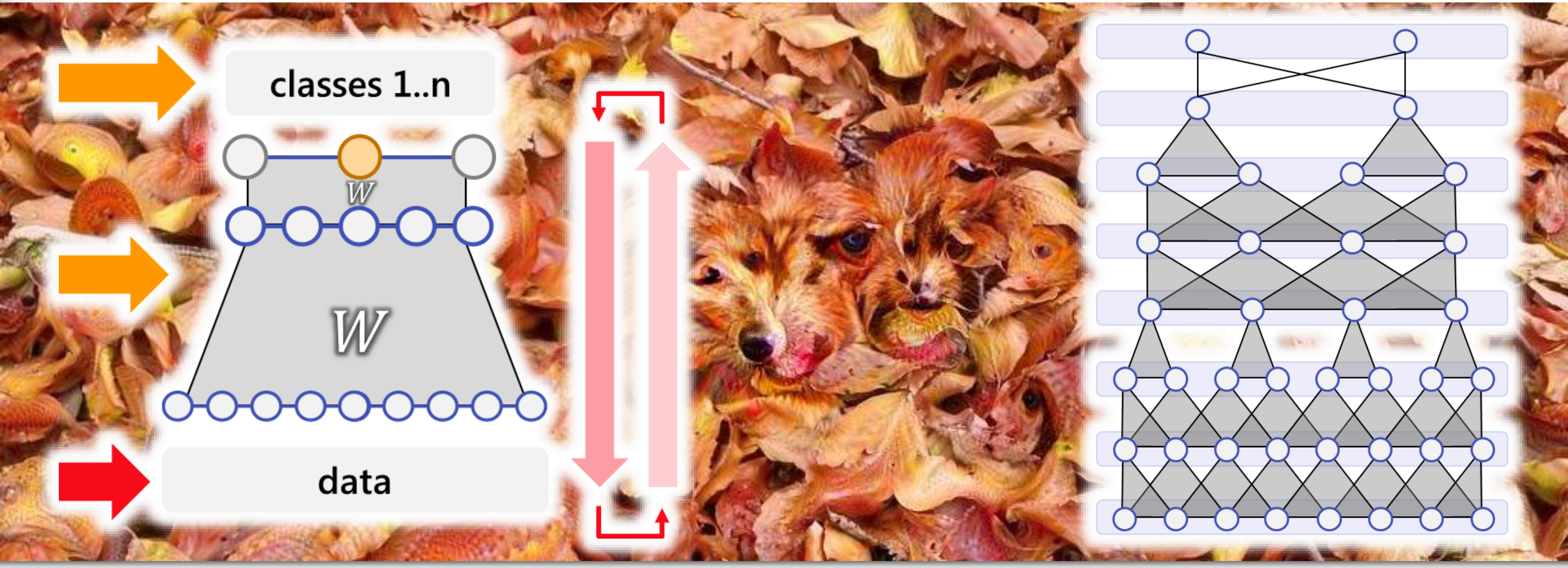
More on Deep Networks

Tasks

- Regression
  - Basic usage: Network encodes a function
  - Then add least-squares loss (or the similar)
- Classification
  - Typically soft-max regression with non-linear function
- Dimensionality reduction
  - Autoencoders
  - Better generative models soon!
- Embedding
  - Siamese networks

Modelling 2

STATISTICAL DATA MODELLING



[Deep Dream Image: Daniel Strecker]

Chapter 9

Deep Neural Networks

Michael Wand · Institut für Informatik, JGU Mainz · michael.wand@uni-mainz.de

Video #09

Down the Deep End

- **Back to the Future:** Neural Networks
- **Common Architectures**
- **Generative Models**

Generative Models

Overview

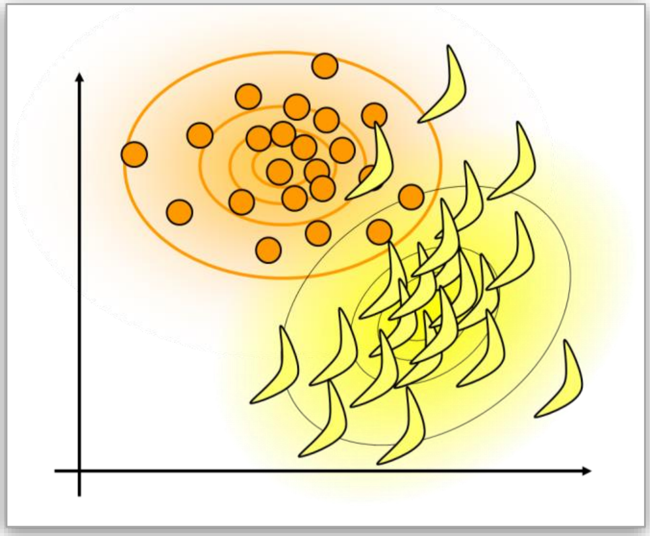
- Generative Models
- Generative networks

Methods

- Autoencoders revisited
- Problems with direct training
- Why not? – Normalizing flows
- Autoregressive models
- Generative adversarial networks

Generative Models

Generative Models



Given

- Samples (i.i.d.)

$$\mathbf{x}_i \in \mathbb{R}^d, i = 1, \ldots, n$$

Task

- Reconstruct probability density

$$p_{\boldsymbol{\theta}}: \mathbb{R}^d \to \mathbb{R}$$

such that

$$\mathbf{x}_i \sim p_{\boldsymbol{\theta}}$$

is likely/plausible.

- Need to find parameters $\boldsymbol{\theta} \in \mathbb{R}^k$.

How to do it?

You know the drill…

- Specify generator p_θ
  - Classically: E.g., a Gaussian
  - Deep: E.g., a generative network
- Maximum likelihood (ML)

$$\arg\max_{\boldsymbol{\theta}\in\mathbb{R}^k} \left[\prod_{i=1}^{n} p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right] = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^k} \left[\sum_{i=1}^{n} -\log p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right]$$

- Maximum a posteriori (MAP)

$$\arg\max_{\boldsymbol{\theta}\in\mathbb{R}^k} \left[P(\boldsymbol{\theta}) \prod_{i=1}^{n} p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right] = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^k} \left[-\log P(\boldsymbol{\theta}) + \sum_{i=1}^{n} -\log p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right]$$

How to do it?

You know the drill…

- Specify generator p_θ
  - Classically: E.g., a Gaussian
  - Deep: E.g., a generative network
- Maximum likelihood (ML)

$$\arg\max_{\theta\in\mathbb{R}^k}\left[\prod_{i=1}^{n} p_\theta(\mathbf{x}_i)\right] = \arg\min_{\theta\in\mathbb{R}^k}\left[\sum_{i=1}^{n} -\log p_\theta(\mathbf{x}_i)\right]$$

Typically, in deep nets, ML is the goal. (but even that is hard)

- Maximum a posteriori (MAP)

$$\arg\max_{\theta\in\mathbb{R}^k}\left[P(\theta)\prod_{i=1}^{n} p_\theta(\mathbf{x}_i)\right] = \arg\min_{\theta\in\mathbb{R}^k}\left[-\log P(\theta) + \sum_{i=1}^{n} -\log p_\theta(\mathbf{x}_i)\right]$$

Why Generative Models?

Applications for generative models

Creating samples – Example

- Input pretty pictures $\mathbf{x}_i \in \mathbb{R}^d, i = 1, \dots, n$
- Learn $p_{\boldsymbol{\theta}}$
- Output more pretty pictures $\mathbf{x} \sim p_{\boldsymbol{\theta}}$

Why Generative Models?

Applications for generative models

Data reconstruction – Example

- Again, learn $p_{\boldsymbol{\theta}}$ from examples first
- Now, collect noisy/incomplete data $\mathbf{d}$
  - E.g.: Noise, distortions
  - E.g.: Missing pixels
- Model noise/distortion as likelihood $P(\mathbf{d}|\mathbf{x})$
- Reconstruct $\mathbf{x}$ via

learned prior

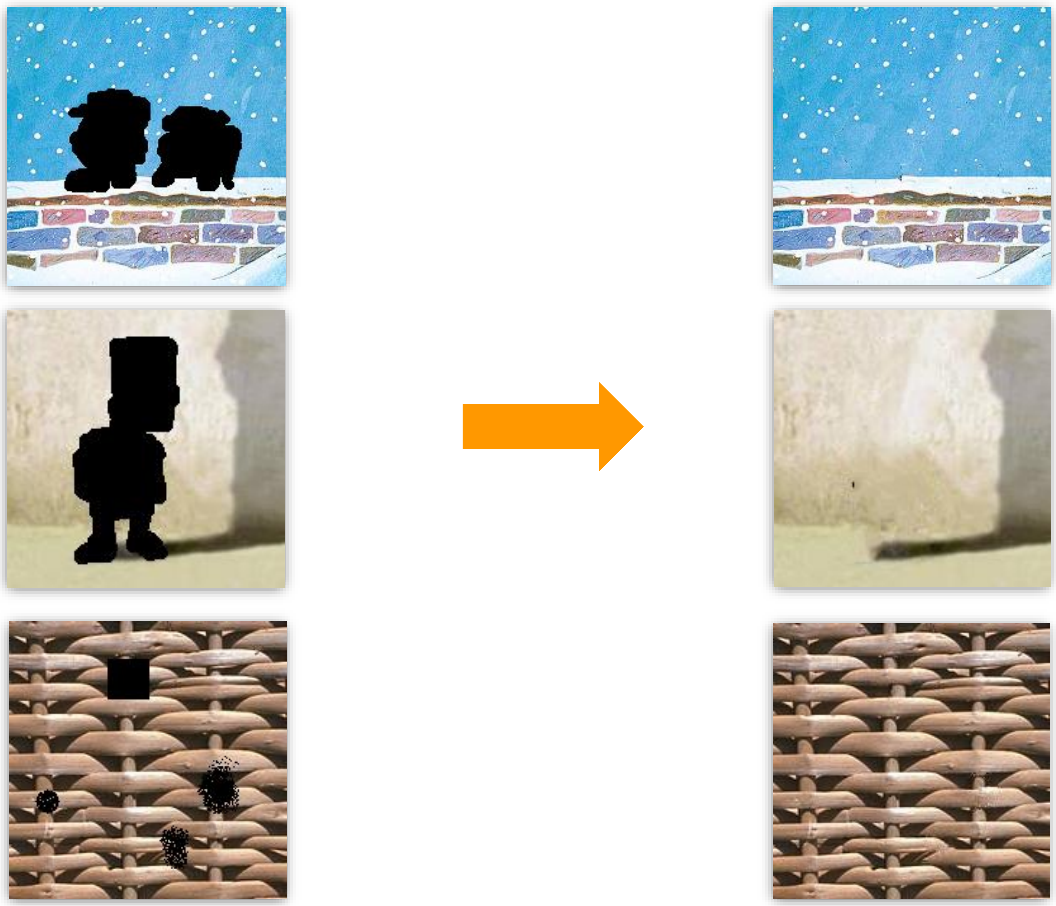
$$P(\mathbf{x}|\mathbf{d}) \sim P(\mathbf{d}|\mathbf{x})P(\mathbf{x}) = P(\mathbf{d}|\mathbf{x})\, p_\theta(\mathbf{x})$$

Reconstruction Applications

Image Denoising

Reconstruction Applications

Hole Filling

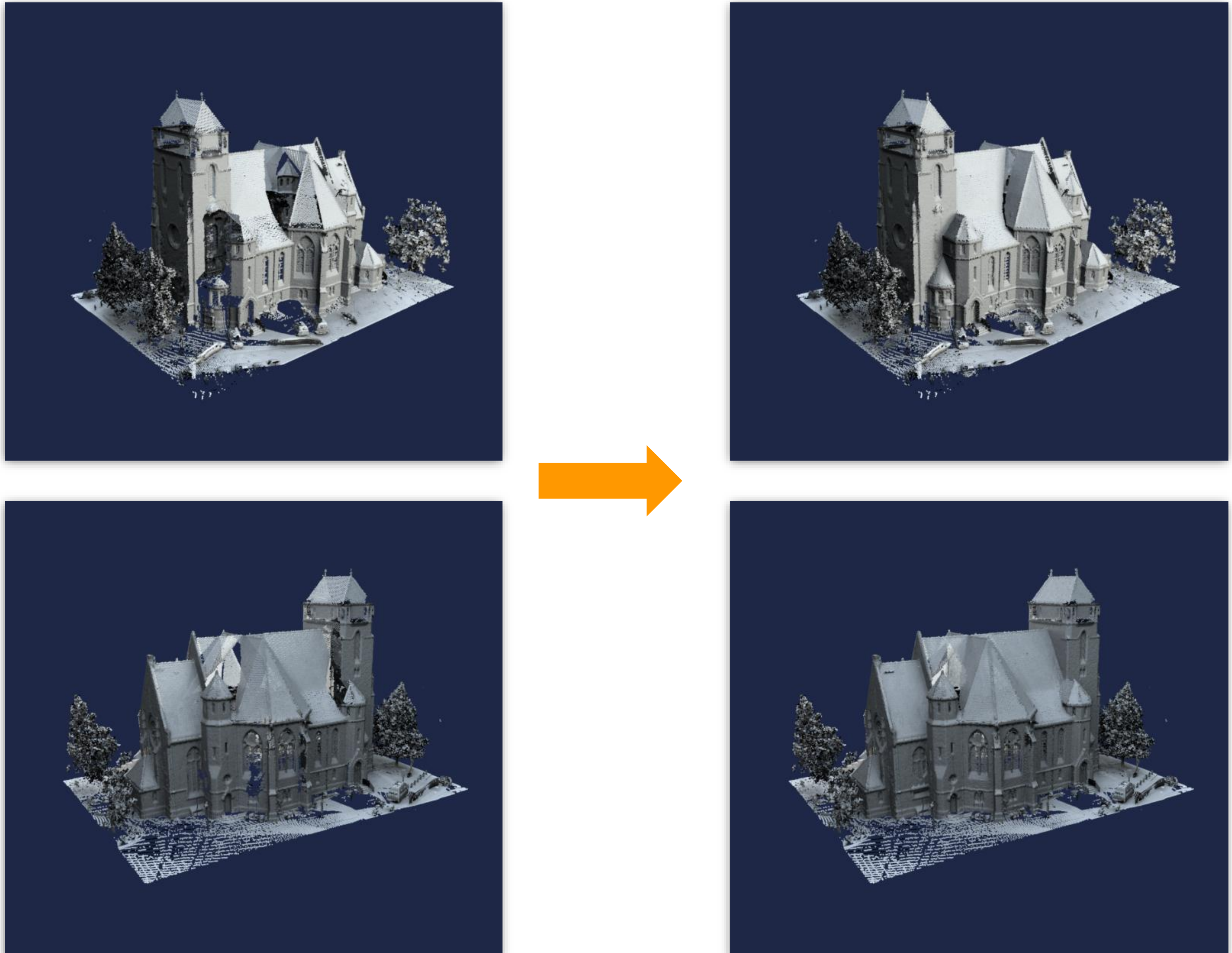


incomplete

statistical completion

[Results by Alexander Berner (MRF-Model)]

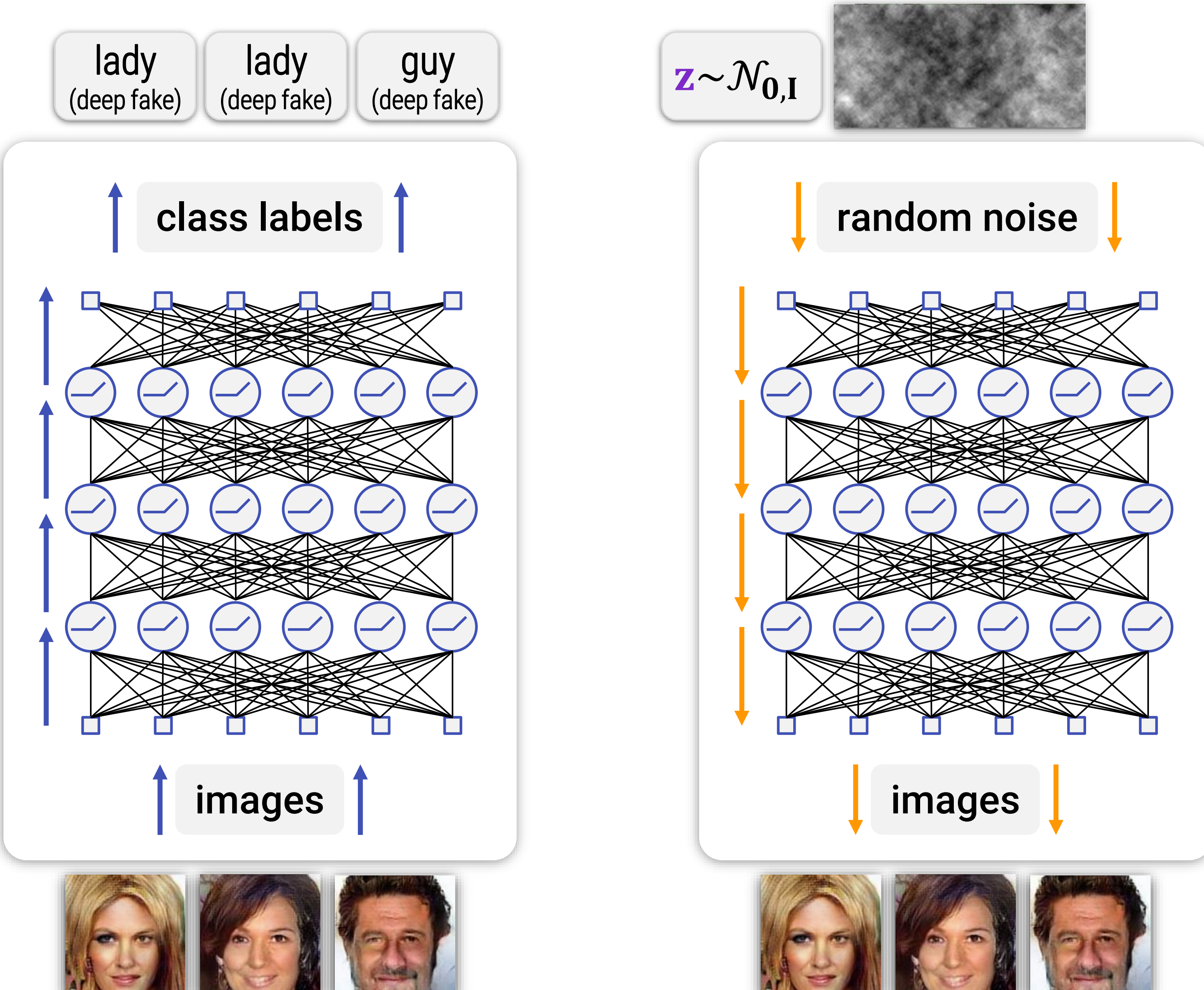
Hole Filling in 3D Scans



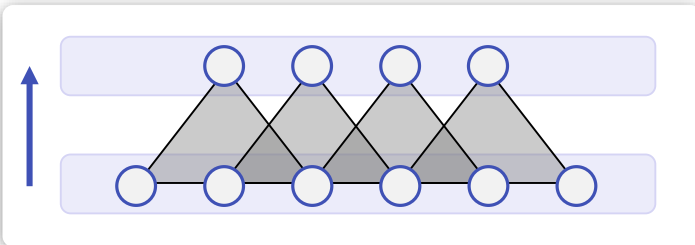
[Joint work with Martin Bokeloh, 2010 (unpublished)]

Generative Networks

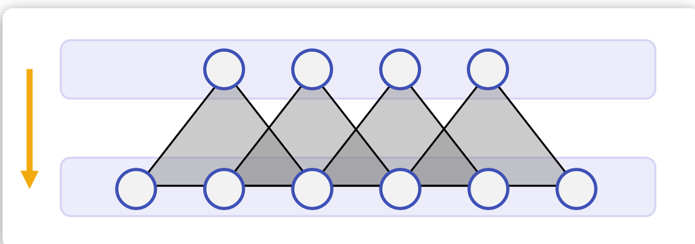
Generative Networks



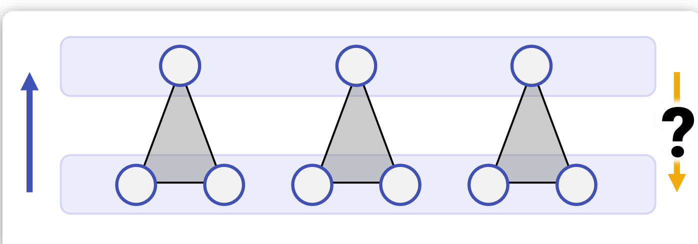
Convolutional Networks?



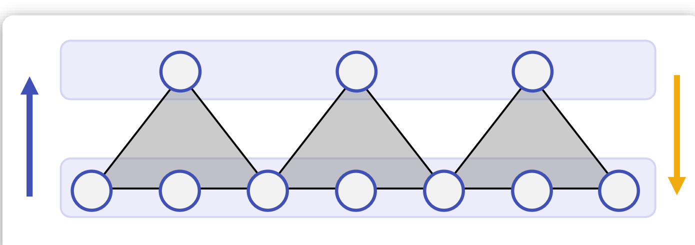
Convolutional network
Discriminative network



Convolutional network
Generative network

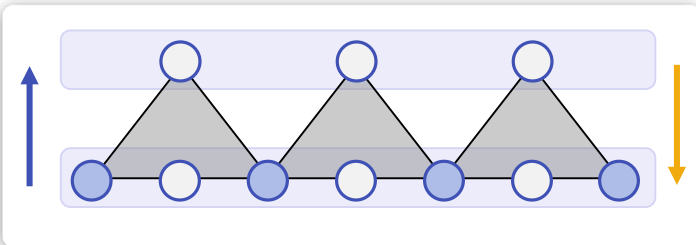


Max- / Average Pooling
Difficult to reverse



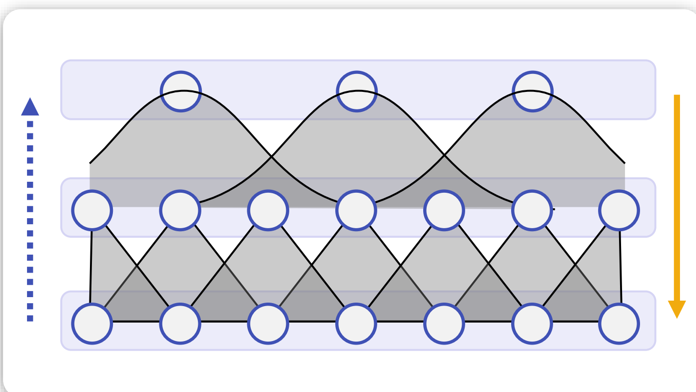
Striding
Just run in reverse

While we are at it...



Striding
Aliasing issues
(e.g.: visible grid
pattern in images)

(w/generative CNN)

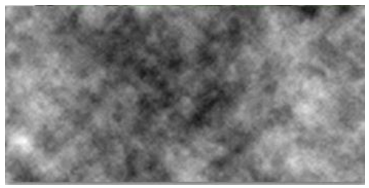


Resampled striding

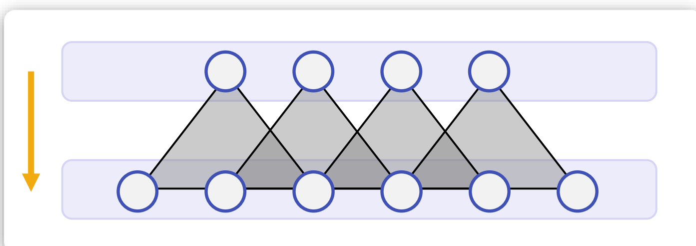
1) upsampling with
low-pass reconstruction filter
2) unstrided convolution

Anti-aliased (for hq-results)

How to Create the Output?



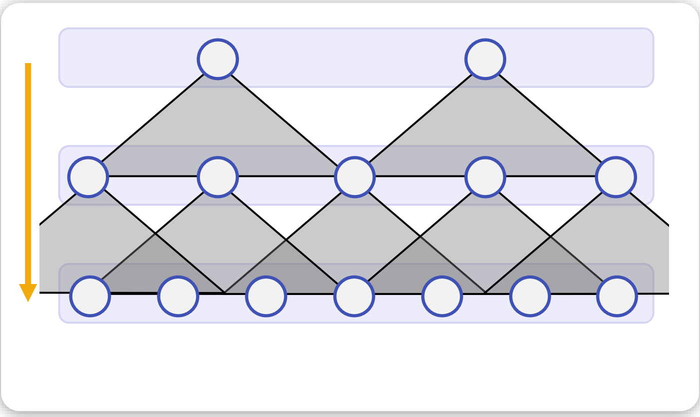
Input
(for example noise)



Convolutional network
Generative network

Synthesis
(after many layers...)

How to Create the Output?

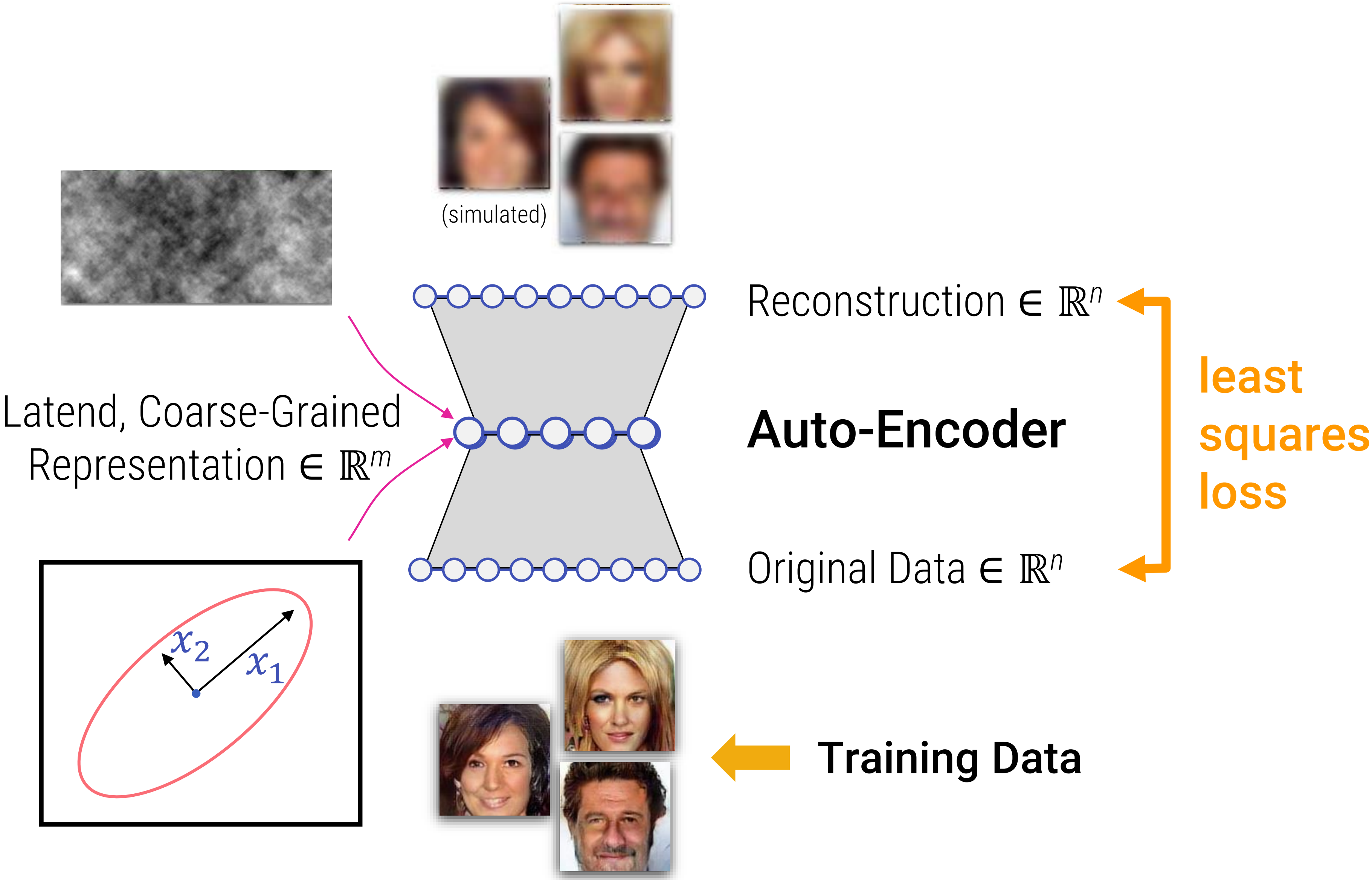


Fully convolution generator
many layers
some with stride
maybe upsampling layers, too

Great! How do we train it?

Autoencoder

Auto-Encoder: Non-linear PCA



Autoencoder Issues

Latent representation

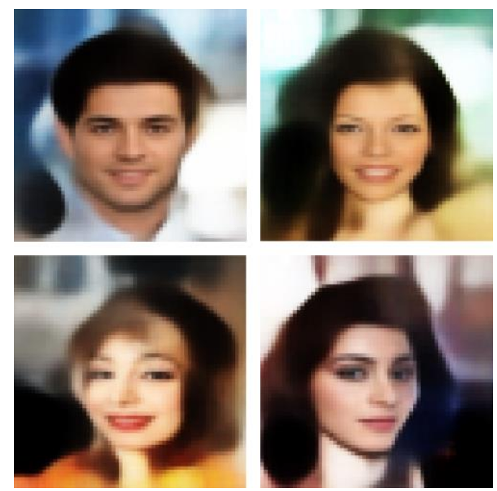
- Arbitrary representation
- Sampling might yield garbage

Fixes

- Fit Gaussian to latent space
  - Empirically, works well (YMMV)
- Variational Autoencoder
  - More principled solution

[results courtesy of D. Schwarz, D. Klaus, A. Rübe]

$\mathcal{N}_{\mathbf{0},\mathbf{I}}$ in latent space



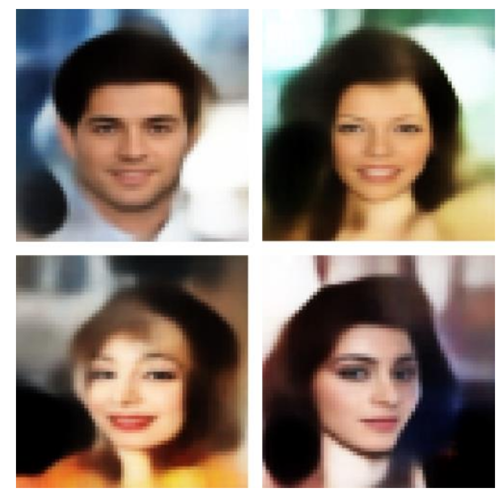
PCA in latent space

Autoencoder Issues

Lack of information

- Bottleneck reduces information content
  - Loss of entropy
  - Need new randomness
- L_2-loss enforces reproduction of original image
  - High-frequency details lost
  - Blurry results
- “Perceptual” metric difficult
  - In a vague sense, this is what GANs learn

$\mathcal{N}_{\mathbf{0},\mathbf{I}}$ in latent space



PCA in latent space

[results courtesy of D. Schwarz, D. Klaus, A. Rübe]

Autoencoder Issues

Autoencoders

- Dimensionality reduction
- Deterministic,
  not probabilistic

Fixes

- VAEs ff. introduce
  probabilistic model

$\mathcal{N}_{\mathbf{0},\mathbf{I}}$ in latent space

PCA in latent space

[results courtesy of D. Schwarz, D. Klaus, A. Rübe]

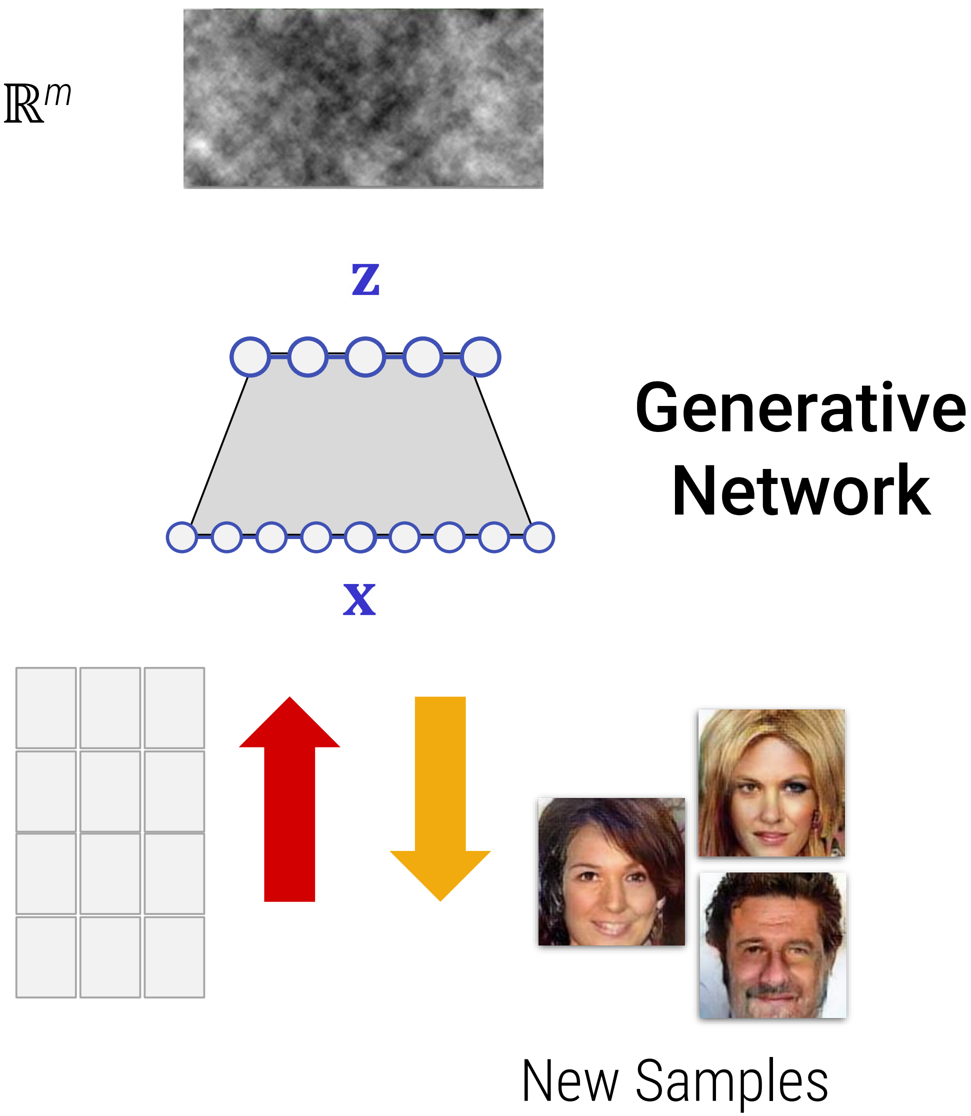
Training of
Generative Networks

Learning Schemes

Gaussian Noise $\in \mathbb{R}^m$

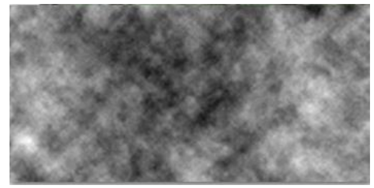
Original Data $\in \mathbb{R}^n$

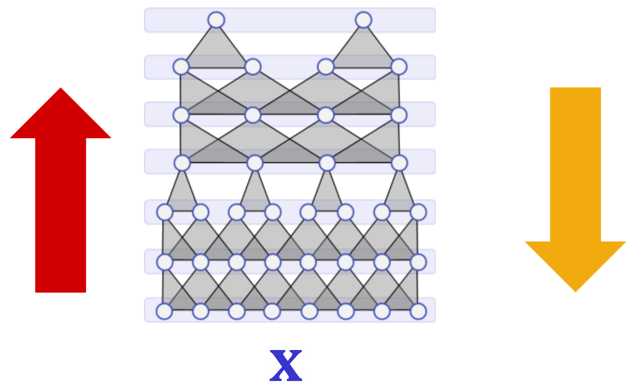
Training Data



Learning Schemes

Gaussian Noise ∈ $\mathbb{R}^m$

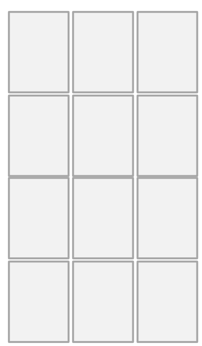




Original Data ∈ $\mathbb{R}^n$

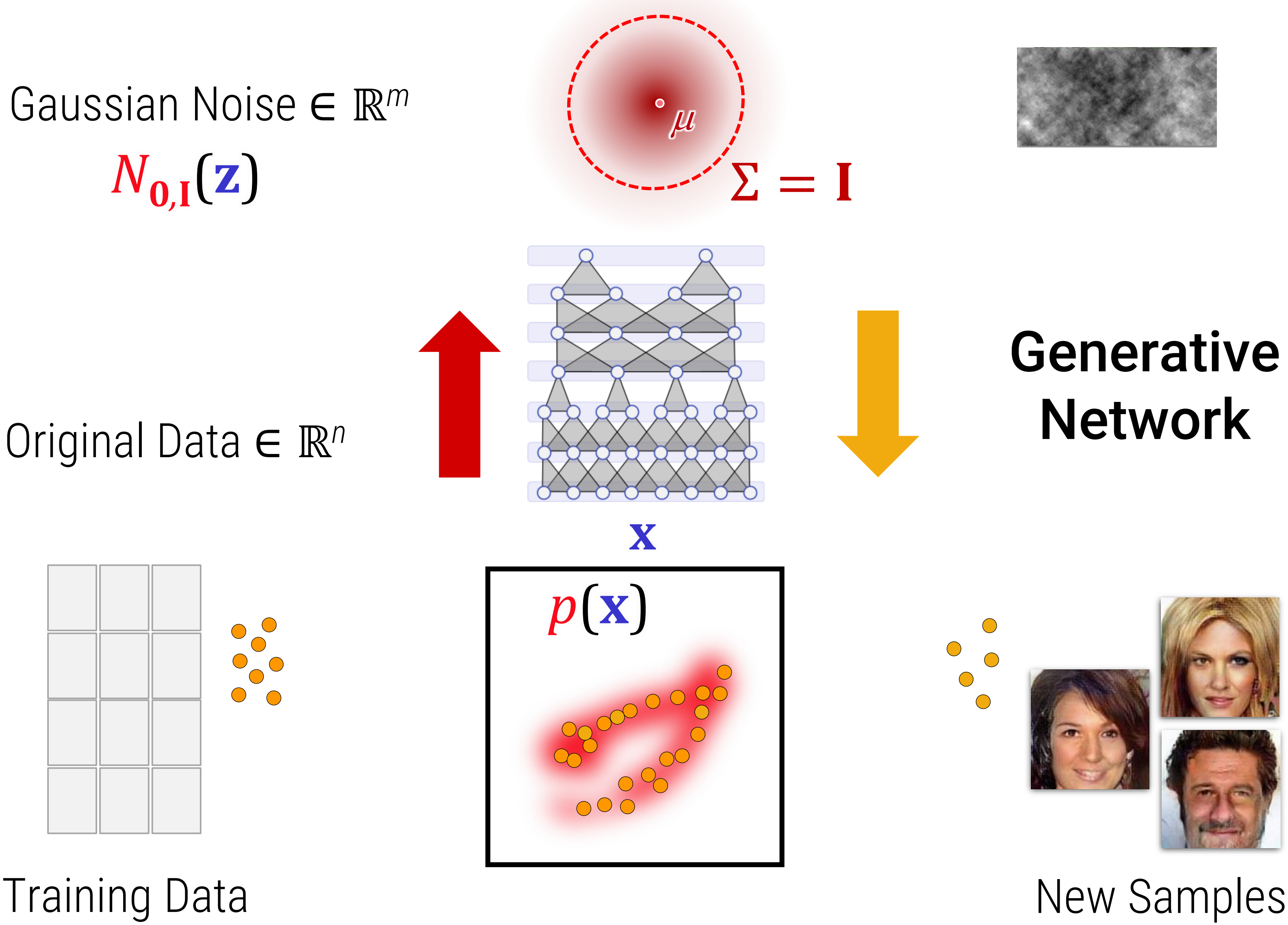
Generative Network

Training Data

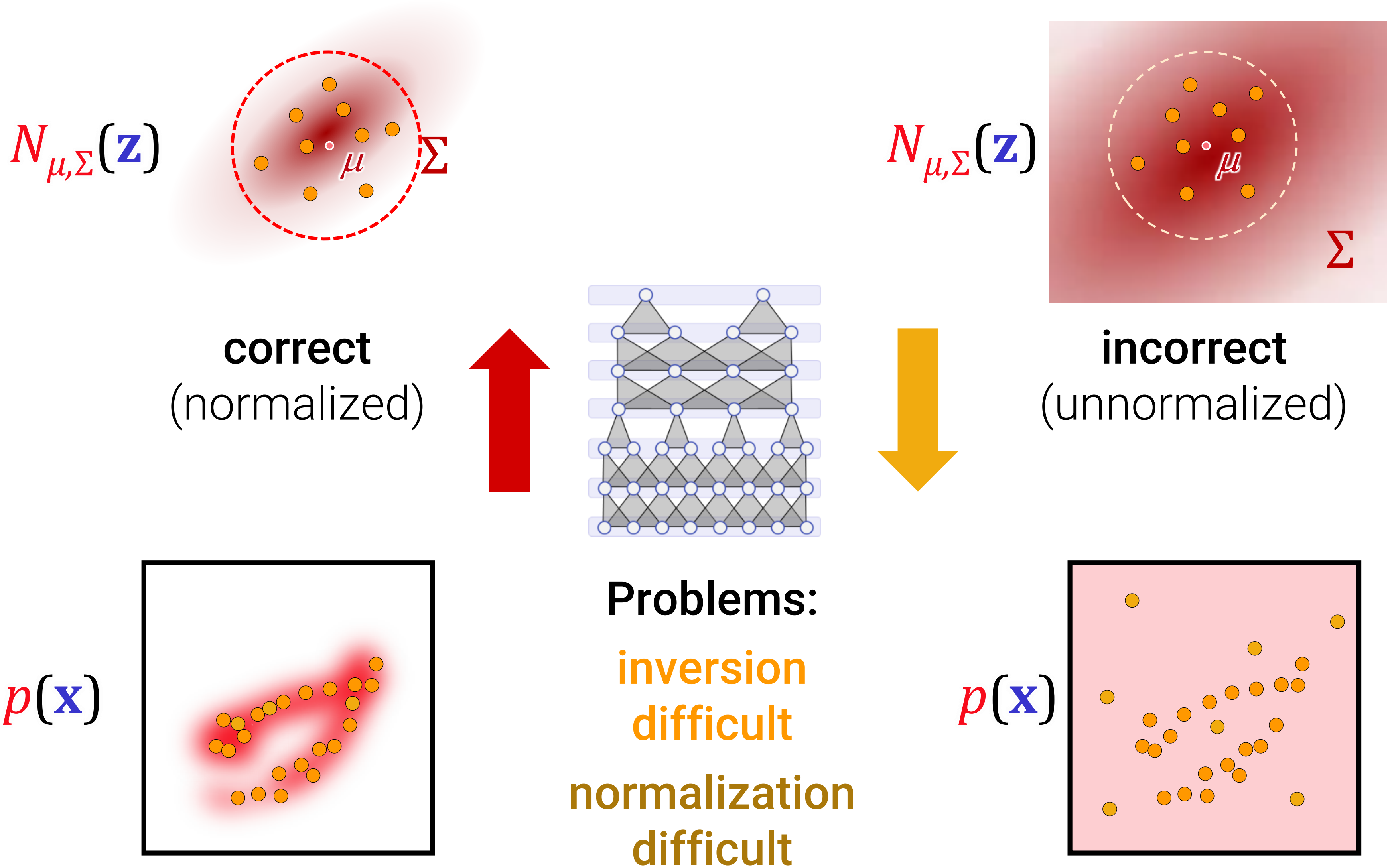


New Samples

Learning Schemes



Problem: Need Normalized Density!



Let's try…

We will have…

- Samples (i.i.d.)

$$\mathbf{x}_i \in \mathbb{R}^d, i = 1, \dots, n$$

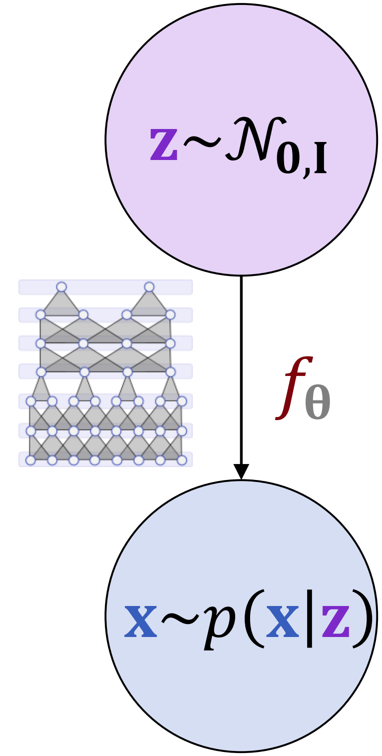
- Noise source: $\mathbf{z} \in \mathbb{R}^k, \mathbf{z} \sim \mathcal{N}_{\mathbf{0},\mathbf{I}}$
- Generative network

$$f_{\boldsymbol{\theta}}: \mathbb{R}^k \to \mathbb{R}^{d_{\mathbf{x}}}, \boldsymbol{\theta} \in \mathbb{R}^{d_{\boldsymbol{\theta}}}$$

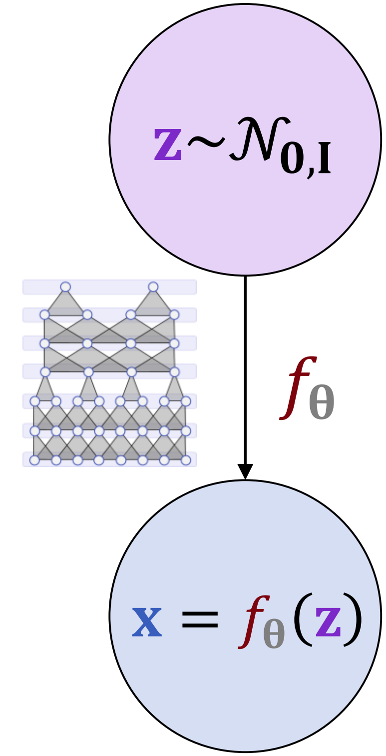
$$f_{\boldsymbol{\theta}}(\mathbf{z}) = \mathbf{x}$$

- ML-objective

$$\boldsymbol{\theta} = \arg\max_{\boldsymbol{\theta} \in \mathbb{R}^k} \left[\prod_{i=1}^{n} p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right]$$



Let's try…



We will have…

- In order to maximize

$$\boldsymbol{\theta} = \underset{\boldsymbol{\theta} \in \mathbb{R}^k}{\arg\max} \left[\prod_{i=1}^{n} p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right]$$

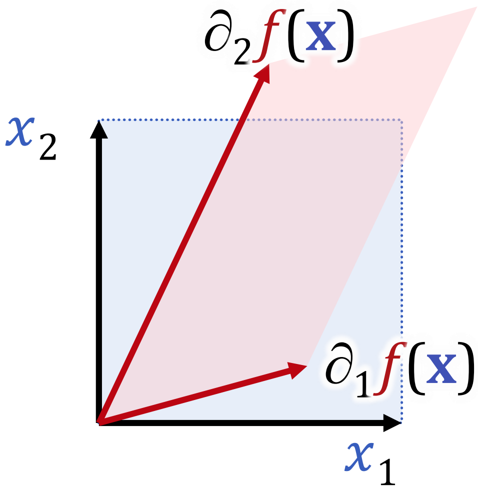
- We need to compute

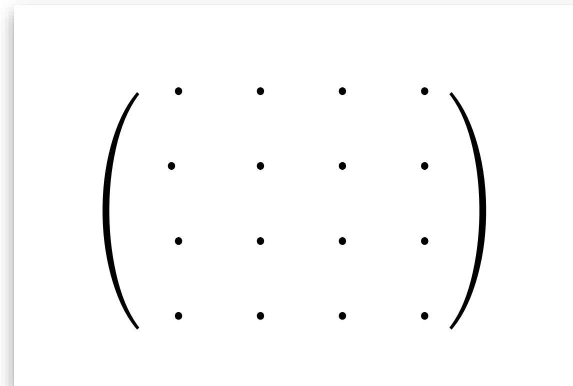
$$p_{\boldsymbol{\theta}}(\mathbf{x}_i) = \mathcal{N}_{\mathbf{0},\mathbf{I}} \left(f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}_i) \right) \cdot \left| \det \left(\nabla f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}_i) \right) \right|$$

Wait – why is that?

$$p_{\boldsymbol{\theta}}(\mathbf{x}_i) = \mathcal{N}_{\mathbf{0},\mathbf{I}}\left(f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}_i)\right) \cdot \left|\det\left(\nabla f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}_i)\right)\right|$$

Jacobian: Geometric Interpretation





Function

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

Jacobian matrix ("Gradient")

$$\nabla f = \begin{pmatrix} \partial_{x_1} f_1(\mathbf{x}) & \cdots & \partial_{x_n} f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \partial_{x_1} f_m(\mathbf{x}) & \cdots & \partial_{x_n} f_m(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} | & & | \\ \partial_1 f(\mathbf{x}) & \cdots & \partial_n f(\mathbf{x}) \\ | & & | \end{pmatrix} \in \mathbb{R}^{n \times m}$$

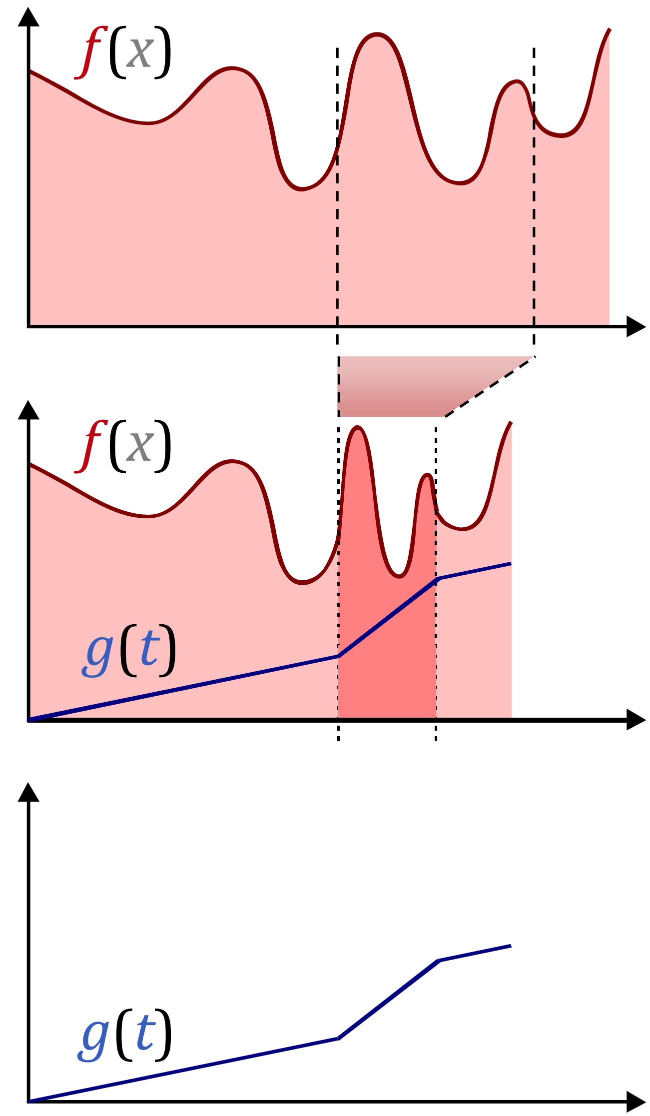
Integral Transformations

Integration by substitution:

$$\int_a^b f(x)dx = \int_{g^{-1}(a)}^{g^{-1}(b)} f(g(t)) \cdot g'(t)dt$$

Need to compensate

- Speed of movement affects measured area
  - **Faster:** shrinks measured area
  - **Slower:** inflates

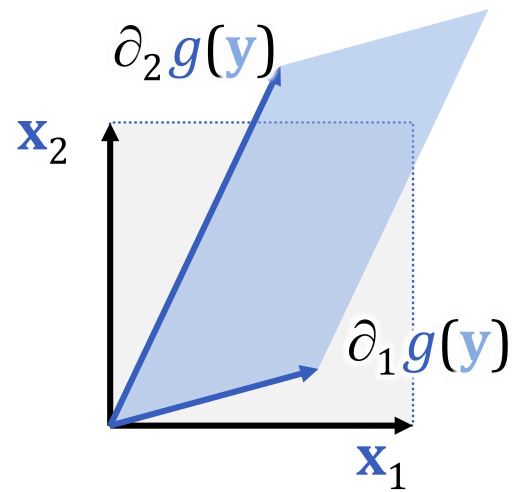


Multi-Dimensional Substitution

Transformation of Integrals:

$$\int_{\Omega} f(\mathbf{x}) d\mathbf{x} = \int_{g^{-1}(\Omega)} f\big(g(\mathbf{z})\big) \cdot |\det[\nabla g(\mathbf{z})]| d\mathbf{z}$$

- $g \in C^1$, invertible
- Jacobian approximates local behavior of $g(\cdot)$
- Determinant: local area/volume change



Probability Density

Probability of an Event A:

- Forward application

$$P(\mathrm{A}) = \int_{\mathbf{x}\in\mathrm{A}} p(\mathbf{x})d\mathbf{x}$$

$$= \int_{\mathbf{z}\in g^{-1}(\mathrm{A})} p\big(g(\mathbf{z})\big)|\det[\nabla g(\mathbf{z})]|d\mathbf{z}$$

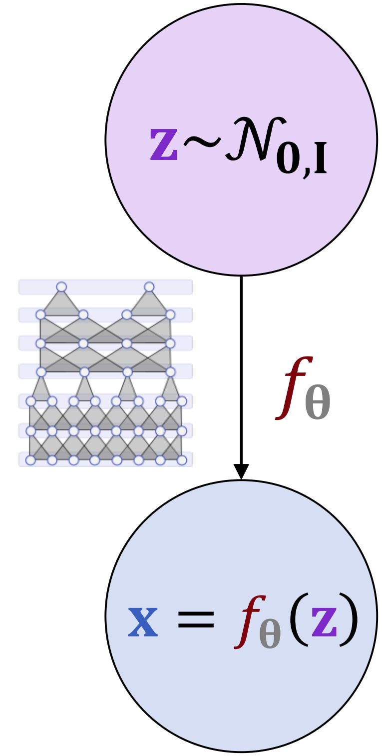
- Reverse problem

$$\mathbf{x} = f_{\boldsymbol{\theta}}(\mathbf{z}) \;\; \rightarrow \;\; p(\mathbf{x}) = p(\mathbf{z}|\mathbf{x}) = p_{\mathbf{z}}\left(f_{\boldsymbol{\theta}}^{-1}(\mathbf{x})\right)$$

- Thus

$$p(\mathbf{x}) = p\left(f_{\boldsymbol{\theta}}^{-1}(\mathbf{x})\right)\underbrace{\left|\det\left[\nabla f_{\boldsymbol{\theta}}^{-1}(\mathbf{x})\right]\right|}_{=(\det[\nabla f_{\boldsymbol{\theta}}(\mathbf{x})])^{-1}}$$

This is our life now



We will have…

- In order to maximize

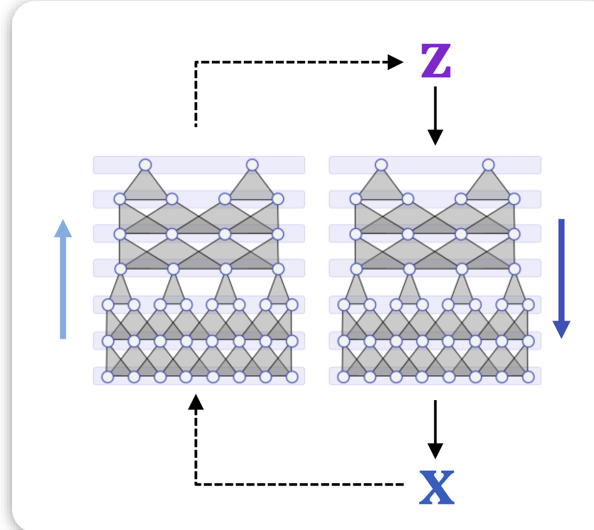
$$\boldsymbol{\theta} = \underset{\boldsymbol{\theta} \in \mathbb{R}^k}{\arg\max} \left[\prod_{i=1}^{n} p_{\boldsymbol{\theta}}(\mathbf{x}_i) \right]$$

- We need to compute

$$p_{\boldsymbol{\theta}}(\mathbf{x}_i) = \mathcal{N}_{\mathbf{0},\mathbf{I}}\left(f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}_i)\right) \cdot \left|\det\left(\nabla f_{\boldsymbol{\theta}}^{-1}(\mathbf{x}_i)\right)\right|$$

- Which is not so easy
  - Inverting the network $f_{\boldsymbol{\theta}}$ is difficult/costly (if possible)
  - Computing the Jacobian matrix is costly
  - Computing the determinant is costly

Vanilla-Version



First attempt

- Just use an arbitrary network

Compute inverse?

- E.g. fit an (approximate) inverse network to it
  - Takes minutes (all data points), each time

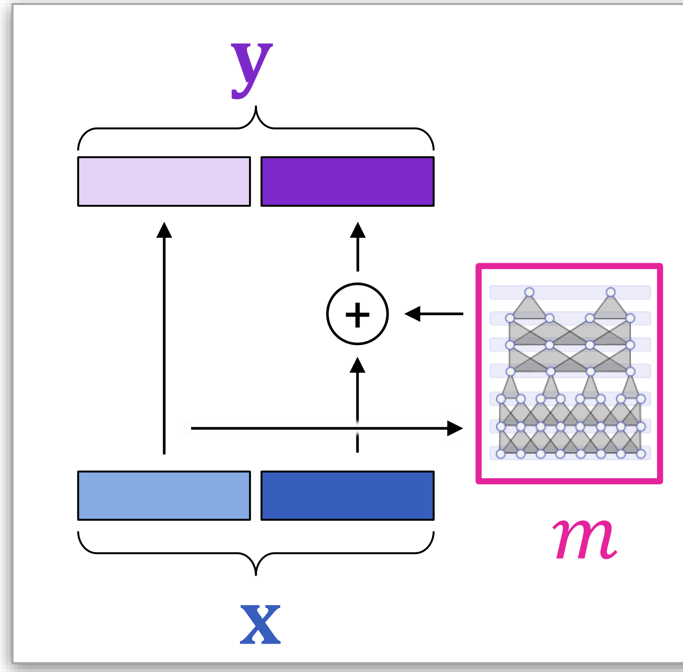
Compute determinant

- Backprop + linear algebra
  - Determinants of large matrices, per data point

Maybe not impossible, but very expensive

Why not?
Normalized Flows

Clever Architecture

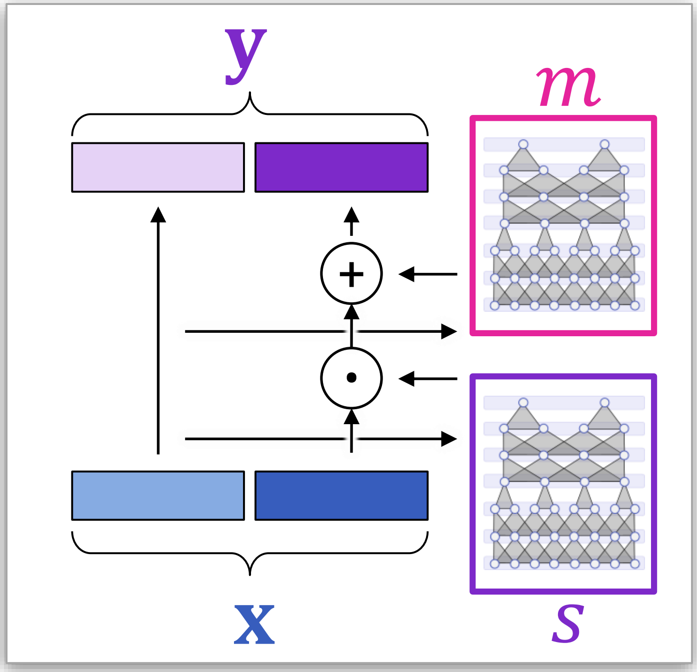


NICE – making our life easier

[Dinh et al. 2014]

- Input $\mathbf{x} \in \mathbb{R}^n$
- Output $f(\mathbf{x}) \in \mathbb{R}^n$ (and $d < n$)
  $$f(\mathbf{x}) = \left[\mathbf{x}_{[1:d]} \,\middle|\, \mathbf{x}_{[d+1:n]} + m\left(\mathbf{x}_{[1:d]}\right)\right]$$
- Inverse
  $$f^{-1}(\mathbf{y}) = \left[\mathbf{y}_{[1:d]} \,\middle|\, \mathbf{y}_{[d+1:n]} - m\left(\mathbf{y}_{[1:d]}\right)\right]$$
- $\det\left(\nabla f(\mathbf{x})\right) = \det\begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \nabla m & \mathbf{I} \end{pmatrix} = 1$
- Swap parts $\mathbf{x}_{[1:d]}, \mathbf{x}_{[d+1:n]}$ with every layer

Nicer



RealNVP [Dinh et al. 2017]

- Function

$$f(\mathbf{x}) = \left[\mathbf{x}_{[1:d]} \,\middle|\, \mathbf{x}_{[d+1:n]} \odot \exp\left(s\left(\mathbf{x}_{[1:d]}\right)\right) + m\left(\mathbf{x}_{[1:d]}\right)\right]$$

- Inverse

$$f^{-1}(\mathbf{y}) = \left[\mathbf{y}_{[1:d]} \,\middle|\, \left(\mathbf{y}_{[d+1:n]} - m\left(\mathbf{y}_{[1:d]}\right)\right) \odot \exp\left(s\left(\mathbf{x}_{[1:d]}\right)\right)^{-1}\right]$$

- $\det\left(\nabla f(\mathbf{x})\right) = \det\begin{pmatrix} \mathbf{I} & & \mathbf{0} & \\ & e^{s_1} & & \\ \square & & \ddots & \\ & & & e^{s_d} \end{pmatrix}$

Training

Maximum Likelihood Training

$$\arg\max_{\theta \in \mathbb{R}^k} \left(\prod_{i=1}^{n} p_{\theta}(\mathbf{x}_i) \right) = \arg\min_{\theta \in \mathbb{R}^k} \left(\sum_{i=1}^{n} -\log p_{\theta}(\mathbf{x}_i) \right)$$

Neg-log-likelihood

$$\begin{aligned} -\log p_{\theta}(\mathbf{x}) &= -\log\left(\mathcal{N}_{0,\mathbf{I}}\left(f_{\theta}^{-1}(\mathbf{x})\right)\right) - \log\left((\det[\nabla f_{\theta}(\mathbf{x})])^{-1}\right) \\ &= -\log\left(\mathcal{N}_{0,\mathbf{I}}\left(f_{\theta}^{-1}(\mathbf{x})\right)\right) + \log\left((\det[\nabla f_{\theta}(\mathbf{x})])\right) \end{aligned}$$

Training

Neg-log-likelihood

$$-\log p_{\theta}(\mathbf{x}) = -\log\left(\mathcal{N}_{0,\mathbf{I}}\left(f_{\theta}^{-1}(\mathbf{x})\right)\right) - \log\left((\det[\nabla f_{\theta}(\mathbf{x})])^{-1}\right)$$
$$= -\log\left(\mathcal{N}_{0,\mathbf{I}}\left(f_{\theta}^{-1}(\mathbf{x})\right)\right) + \log\left((\det[\nabla f_{\theta}(\mathbf{x})])\right)$$

Multi-layer network

$$-\log p_{\theta}(\mathbf{x})$$
$$= -\log\left(\mathcal{N}_{0,\mathbf{I}}\left(f_{\theta}^{-1}(\mathbf{x})\right)\right) + \sum_{l=1}^{L} \log\left(\left(\det\left[\nabla f_{\theta}^{(l)}(\mathbf{x})\right]\right)\right)$$

Results

Quality

- Good image quality, but optimized GANs are better
- Newer variants of related ideas perform better

Versality

- We have an explicit likelihood
- Can be used as prior for image completion, reconstruction etc.

Speed

- Evaluation fast and training, too.

Autoregressive Models

Autoregressive Models

Sequence

- Data

$$x_1, x_2, \dots, x_d \in \mathbb{R}$$

- Distribution

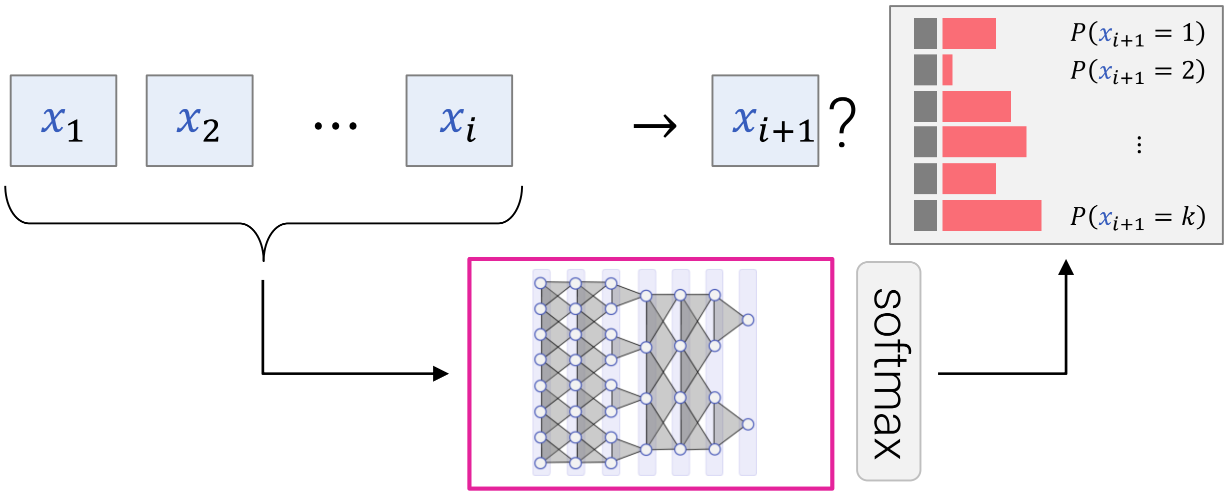
$$p(x_1, x_2, \dots, x_d)$$

- Chain rule (in general)

$$p(x_1, x_2, \dots, x_d)$$

$$= p(x_1) \cdot p(x_2|x_1) \cdots p(x_{d-1}|x_{d-2}, \dots, x_1) \cdot p(x_d|x_{d-1}, \dots, x_1)$$

Autoregressive Models



Idea

- Predict one value at a time

$$x_1, \text{then } x_2, \text{then } x_3, \dots, \text{then } x_d$$

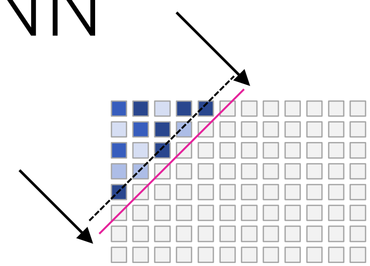
- Generative probabilistic model

predict *distribution* $p(x_d \mid x_{d-1}, \dots, x_1)$
based on *values* $x_{d-1}, \dots, x_1 \in \mathbb{R}$

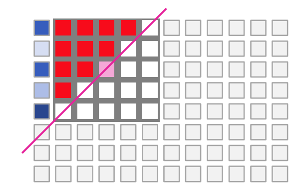
still intractable, but we just use a network

Concrete Examples

Image generation: PixelRNN / PixelCNN



- Images are created pixel-by-pixel
  - Along diagonals (left-top)
- PixelRNN: recurrent neural network (LSTM)
- PixelCNN: convolution kernel (faster)

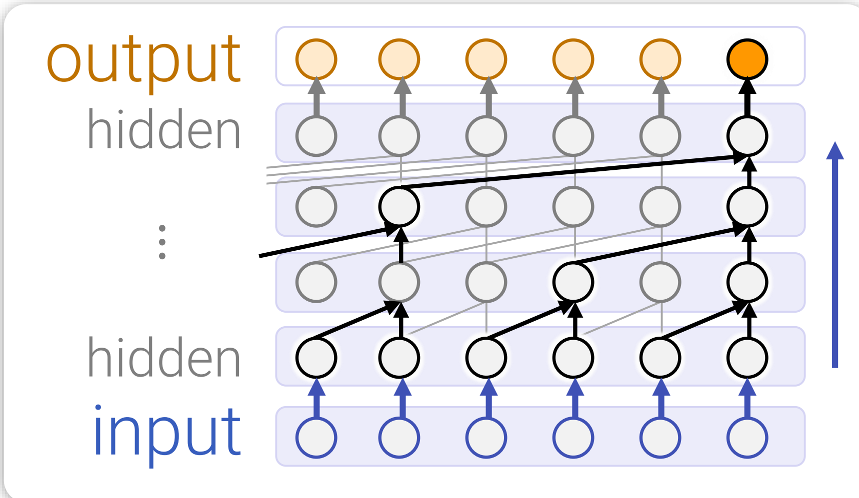


- Distribution for x_{i+1}
  - 256 proability values (entries) for 256 pixel grey-scales
  - RGB-values are predicted sequentially (!)

Improvements possible

- Multi-resolution version (e.g. PixelCNN++, U-net like)

WaveNet



Dilated Convolutions

- Multi-scale structure
- Auto-regressive architecture
- Used for generating sound
- Expensive training (sequential processing)

Generative Adversarial Networks (GANs)

Never mind the likelihood…

Alternative idea

- We do not learn a distribution
- Instead, we (only) learn a sampler

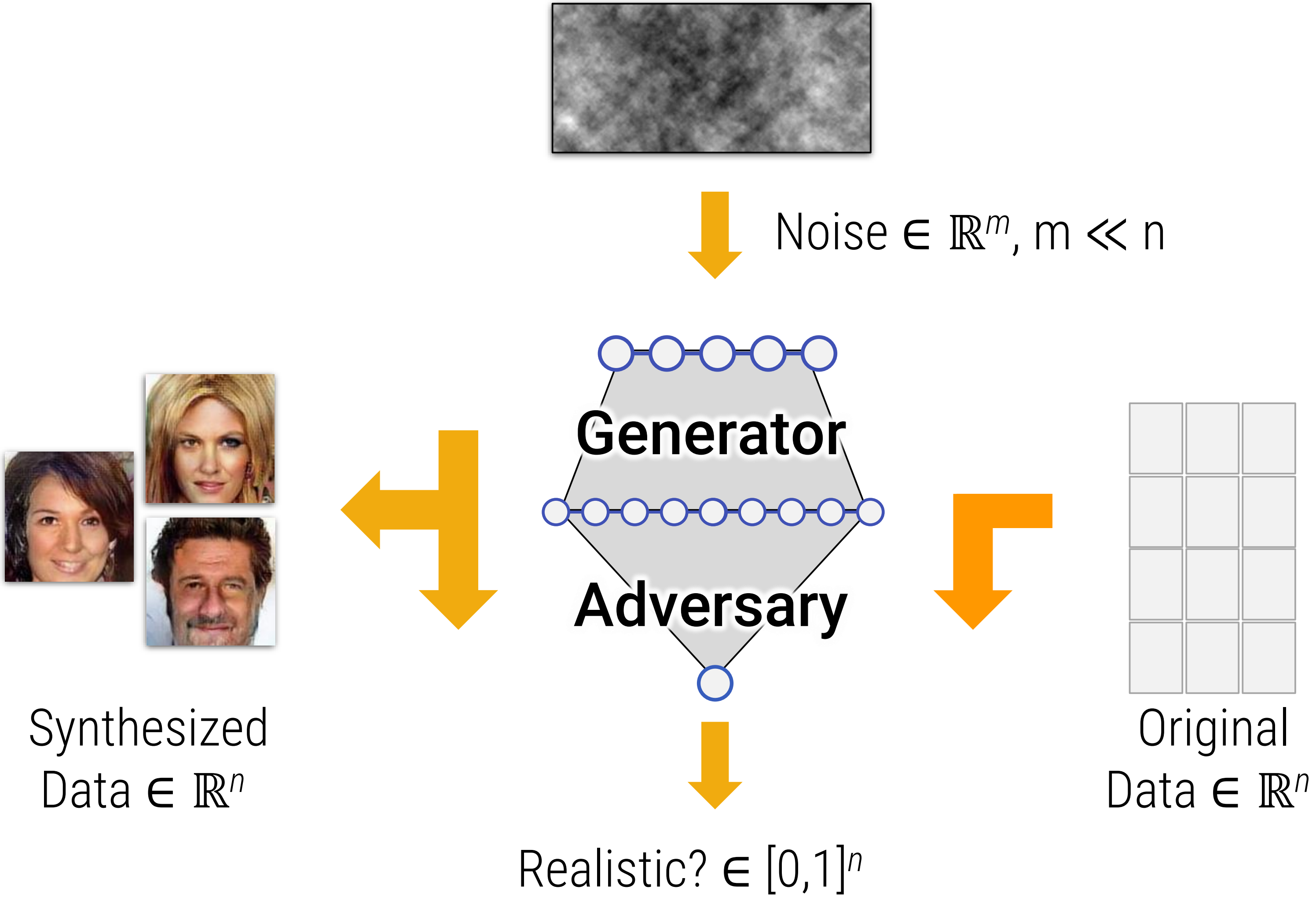
Sampling seems easier

- It is possible to learn "good" samplers without explicit representation of the likelihood

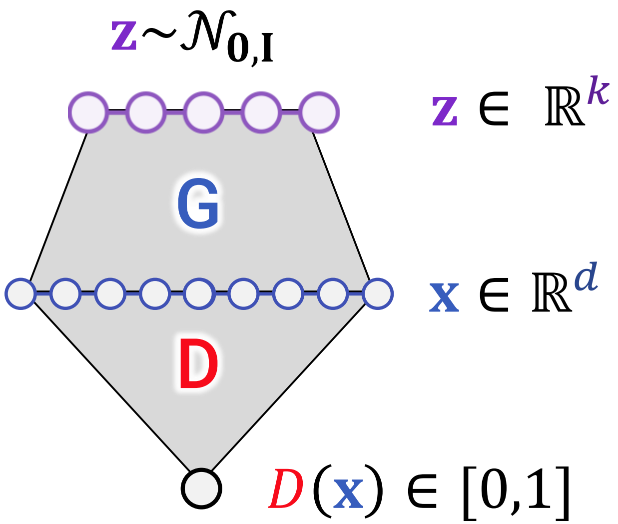
"Generative Adversarial Networks"

- Idea: Complaining is easier than doing
- Let the complainers teach the doers

Learning Scheme



Formalization



Data: Samples (i.i.d.)

$$\mathbf{x}_i \in \mathbb{R}^d, i = 1, \dots, n$$

Networks

- Generator $G_{\boldsymbol{\theta}}: \mathbb{R}^k \to \mathbb{R}^d$
  - Takes random noise

    $$\mathbf{z} \sim \mathcal{N}_{\mathbf{0},\mathbf{I}}, \quad \mathbf{z} \in \mathbb{R}^k$$

  - Outputs "fake" samples

    $$\mathbf{x} \in \mathbb{R}^d$$

- Discriminator $D_{\boldsymbol{\phi}}: \mathbb{R}^d \to [0,1]$
  - Learns to distinguish "real" from "fake" data
  - Output: likelihood of "real"

Loss Function

Distributions

- $p_{data}: \mathbb{R}^d \to \mathbb{R}$ actual data distribution
- $p_G: \mathbb{R}^d \to \mathbb{R}$ generator distribution
- $p(\mathbf{z}): \mathbb{R}^k \to \mathbb{R}$ latent noise distribution (typ. $\mathcal{N}_{\mathbf{0},\mathbf{I}}$)

Objective function

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} V\left(D_{\boldsymbol{\phi}}, G_{\boldsymbol{\theta}}\right)$$

$$V\left(D_{\boldsymbol{\phi}}, G_{\boldsymbol{\theta}}\right) = \mathbb{E}_{\mathbf{x} \sim p_{data}}\left[\log D_{\boldsymbol{\phi}}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}\left[\log\left(1 - D_{\boldsymbol{\phi}}\left(G_{\boldsymbol{\theta}}(\mathbf{z})\right)\right)\right]$$

Loss function

$$\min_{\theta} \max_{\phi} V(D_{\phi}, G_{\theta})$$

View of the discriminator

$$\underbrace{\mathbb{E}_{\mathbf{x}\sim p_{data}}\left[\log D_{\phi}(\mathbf{x})\right]}_{\text{large}} + \underbrace{\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - D_{\phi}(G_{\theta}(\mathbf{z}))\right)\right]}_{\text{large}}$$

large 🚀 — D recognizes true images

large 🚀 — low score for images of G

View of the generator

$$\underbrace{\mathbb{E}_{\mathbf{x}\sim p_{data}}\left[\log D_{\phi}(\mathbf{x})\right]}_{\text{indifferent}} + \underbrace{\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - D_{\phi}(G_{\theta}(\mathbf{z}))\right)\right]}_{\text{small}}$$

indifferent — No information for G

small 🚀 — G fools D

Optimization

Training

- Discriminator tries to distinguish real / fake
  - Maximize prediction accuracy
- Generator tries to fool discriminator
  - Minimizes prediction accuracy
- Minimax game
- Nash equilibrium at true distribution

Nash-Equilibrium

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} V(D_{\boldsymbol{\phi}}, G_{\boldsymbol{\theta}})$$

Optimal discriminator

$$D_G^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_G(\mathbf{x})}$$ (Bayes-optimal likelihood ratio)

Optimal generator?

short: $p_d = p_{data}$

$$\mathbb{E}_{\mathbf{x}\sim p_d}[\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - D_G^*(G_{\boldsymbol{\theta}}(\mathbf{z}))\right)\right]$$

$$= \mathbb{E}_{\mathbf{x}\sim p_d}\left[\log \frac{p_d(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})}\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - \frac{p_d(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})}\right)\right]$$

$$= \mathbb{E}_{\mathbf{x}\sim p_d}\left[\log \frac{p_d(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})}\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(\frac{p_G(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})}\right)\right]$$

Nash-Equilibrium

short: $p_d = p_{data}$

Optimal generator?

$$\mathbb{E}_{\mathbf{x}\sim p_d}[\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - D_G^*(G_{\boldsymbol{\theta}}(\mathbf{z}))\right)\right]$$

$$= \mathbb{E}_{\mathbf{x}\sim p_d}\left[\log \frac{p_d(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})}\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(\frac{p_G(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})}\right)\right]$$

For $p_{data} = p_G$, we obtain

$$\ldots = \mathbb{E}_{\mathbf{x}\sim p_d}\left[\log \frac{1}{2}\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(\frac{1}{2}\right)\right] = 2\log\frac{1}{2}$$

Next, we show that this is really optimal

Optimality

short: $p_d = p_{data}$

First term in objective

$$\mathbb{E}_{\mathbf{x}\sim p_d}[\log D_G^*(\mathbf{x})] = \int_{\mathbf{x}\in\mathbb{R}^d} p_d(\mathbf{x}) \log \frac{p_d(\mathbf{x})}{p_d(\mathbf{x}) + p_G(\mathbf{x})} d\mathbf{x}$$

$$= \log\frac{1}{2} + \int_{\mathbf{x}\in\mathbb{R}^d} p_d(\mathbf{x}) \log \frac{p_d(\mathbf{x})}{\frac{1}{2}\left(p_d(\mathbf{x}) + p_G(\mathbf{x})\right)} d\mathbf{x}$$

$$= \log\frac{1}{2} + KL\left(p_d(\mathbf{x}) \,\middle\|\, \frac{p_d(\mathbf{x}) + p_G(\mathbf{x})}{2}\right)$$

Optimality

short: $p_d = p_{data}$

Second term in objective

$$\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - D_G^*\left(G_{\boldsymbol{\theta}}(\mathbf{z})\right)\right)\right]$$

$$= \int_{\mathbf{z}\in\mathbb{R}^k} p(\mathbf{z}) \log\left(\frac{p_G\left(G_{\boldsymbol{\theta}}(\mathbf{z})\right)}{p_d\left(G_{\boldsymbol{\theta}}(\mathbf{z})\right) + p_G\left(G_{\boldsymbol{\theta}}(\mathbf{z})\right)}\right) d\mathbf{z}$$

$$= \log\frac{1}{2} + KL\left(p_G(\mathbf{x}) \,\middle\|\, \frac{p_d(\mathbf{x}) + p_G(\mathbf{x})}{2}\right)$$

Optimality

short: $p_d = p_{data}$

Sum of the two terms

$$V(D_G^*, G_{\boldsymbol{\theta}}) = 2\log\frac{1}{2} + KL\left(p_G(\mathbf{x}) \,\middle\|\, \frac{p_d(\mathbf{x}) + p_G(\mathbf{x})}{2}\right)$$

$$+ KL\left(p_d(\mathbf{x}) \,\middle\|\, \frac{p_d(\mathbf{x}) + p_G(\mathbf{x})}{2}\right)$$

$$= \underbrace{2\log\frac{1}{2}}_{\text{minimum value}} + 2\,JS\big(p_G(\mathbf{x}) \,\|\, p_d(\mathbf{x})\big)$$

minimum value

(as shown before)

so much for the theory, but now…

Practice

How to Build & Operate a GAN

Practical Training

- Min-max game is unrealistically hard to compute
- Thus: simultaneous gradient descent on $V\left(D_{\phi}, G_{\theta}\right)$
  - Alternate true/fake images every other iteration
- Significant problem
  - Theoretically, this scheme does not necessarily converge
  - Practically, it is highly unstable
  - Can be stabilized with a big bag of tricks
- Typical problems
  - Vanishing gradients: typically D wins, G stalls
  - Mode collapse: G learns a small set of deceiving examples

How to Build & Operate a GAN

Tips & Tricks (useful)

- **Images:** Using a convolutional generator
  - Strided convolutions for upsampling
  - Maybe resampling filters
  - Known as "DCGAN" – deep convolutional GAN [Radford et al. ICLR 2016]
- DCGAN approach has become standard

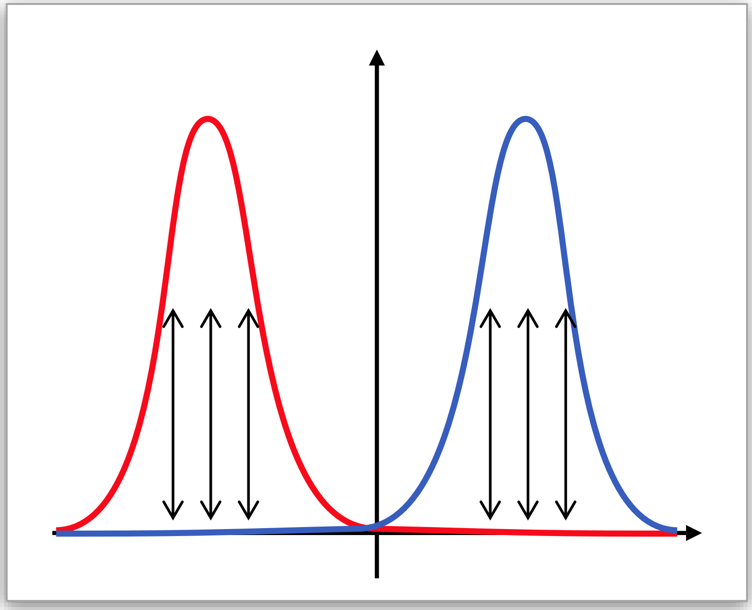
How to Build & Operate a GAN

Tips & Tricks (≈ Alchemy)

- Training
  - Discriminator might get too smart
    - Schedule updates
    - Modify objective for G slightly
      $\max \log D$ instead of $\min \log(1 - D)$
- BatchNorm is problematic
  - Use InstanceNorm instead
  - At least separate batches for "true" and "fake"
- Batch-Discrimination
  - Feed batches at once to D
  - Avoids (to some extend) "mode-collapse"

Wasserstein GANs

JS has its issues…



Reminder

$$KL(p \parallel q) = \sum_{i=1}^{n} p_i \log_2 \frac{p_i}{q_i}$$

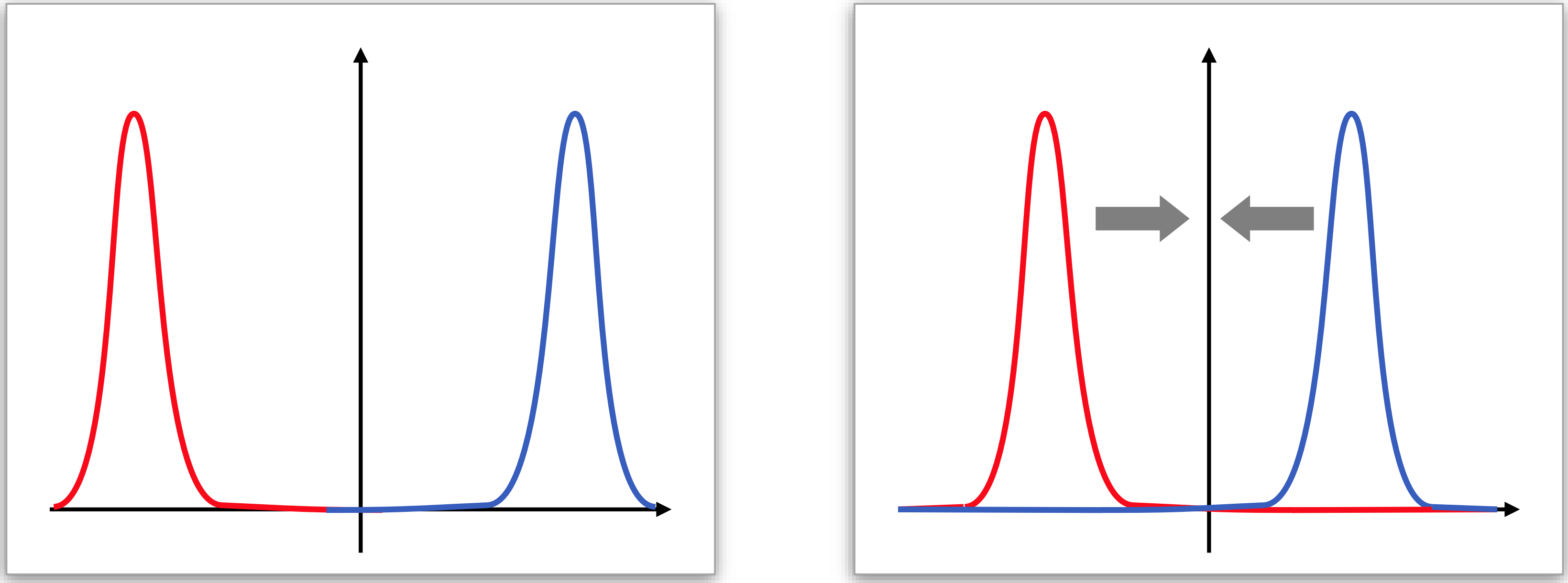
(discrete probabilities)

$$JS(p \parallel q)$$
$$= \frac{1}{2}\left(KL\left(p \parallel \frac{p+q}{2}\right) + KL\left(q \parallel \frac{p+q}{2}\right)\right)$$

Problems with KL/JS

- Point-wise comparison
  - Unaligned densities yield singularities in KL (not in JS)
  - Gradients of JS vanish
- GANs optimize JS → vanishing gradients

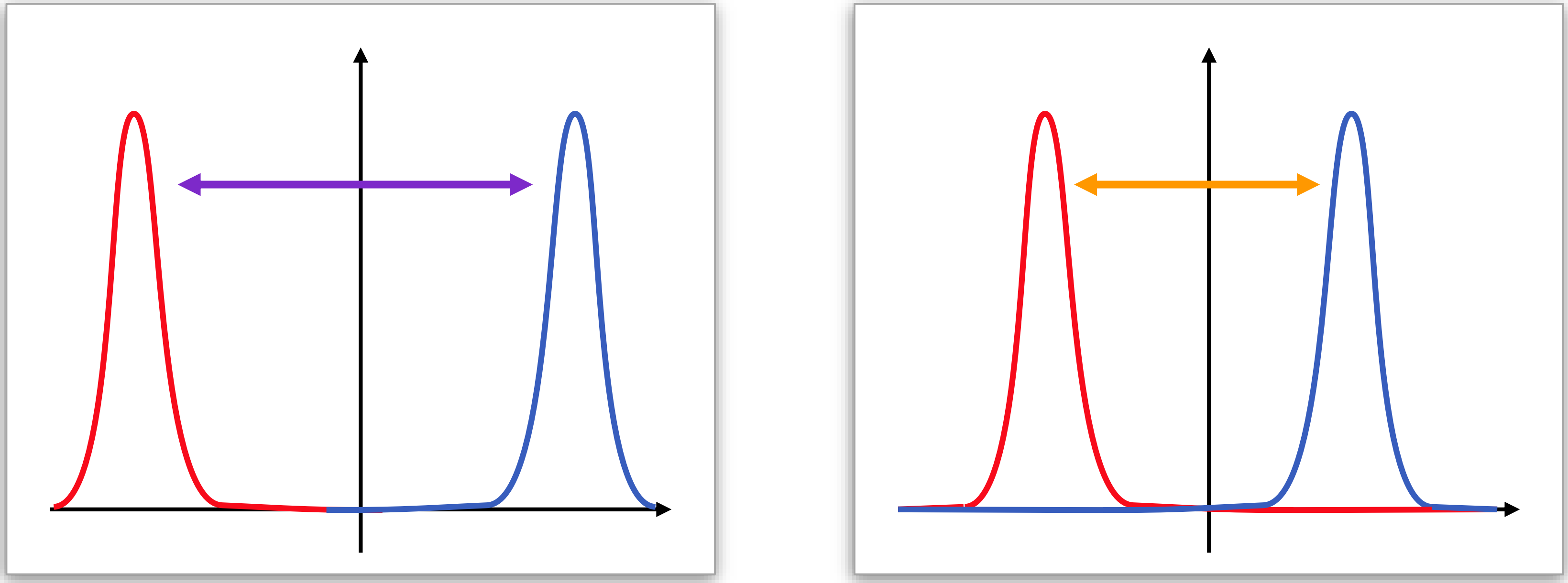
JS has its issues...



These two distributions

- Approximately the same distance
- How to get closer: JS not informative

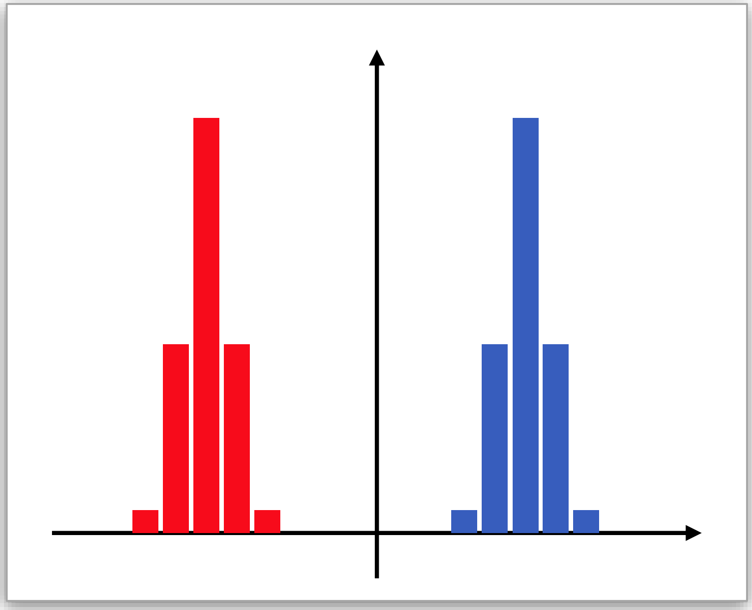
Earth-Mover's Distance – Wasserstein W_1

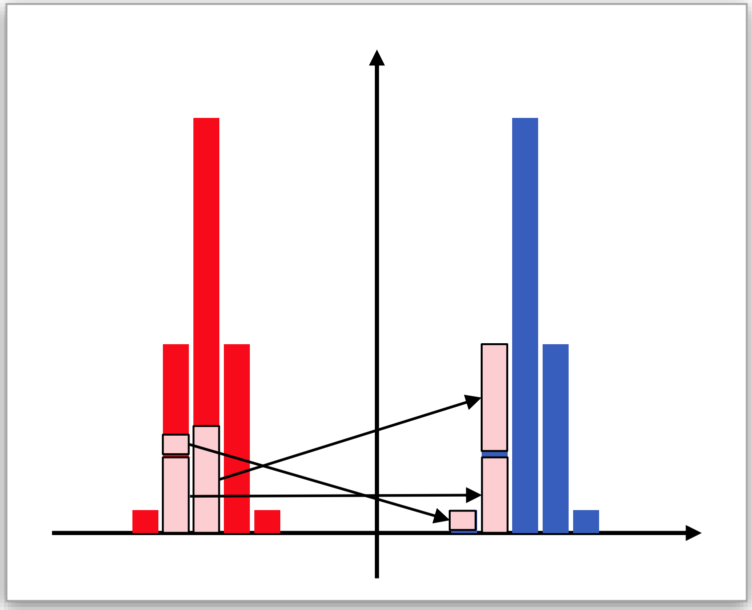


New Idea

- "Optimal transport"
  - Move probability density from p to q
  - Cost = mass x distance
  - Optimal transport = "earth-mover's distance" (Wasserstein W_1)

Definition (basic)

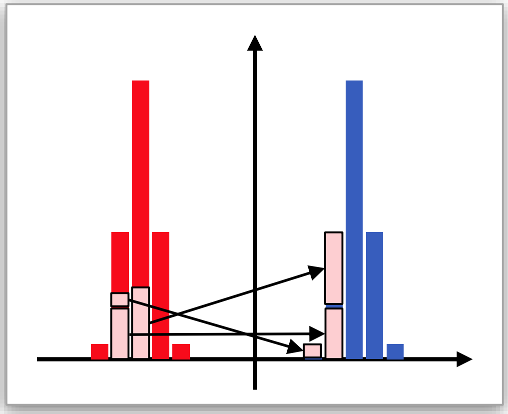




Transport Plan

- Shovel red to blue
- Amount of shoveled red must add to blue
- Cannot take more than available

Definition (basic)



Transport Plan

- Discrete model

  $p_1, \dots, p_n, q_1, \dots, q_n$

- Transport plan

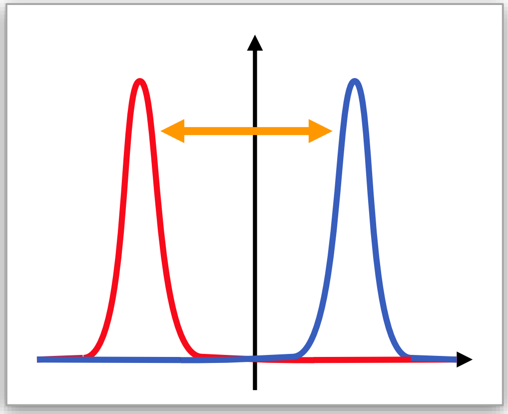
$$\pi_{p,q}(i,j) \geq 0, \qquad \sum_i \pi_{p,q}(i,j) = p_j, \qquad \sum_j \pi_{p,q}(i,j) = q_i$$

- “Shoveling-costs”

$$C(\pi_{p,q}) = \sum_{i,j} \pi_{p,q}(i,j)|i-j|$$

$$W_1(p,q) = \inf_{\text{valid } \pi} C(\pi_{p,q})$$

Definition (basic)



General case

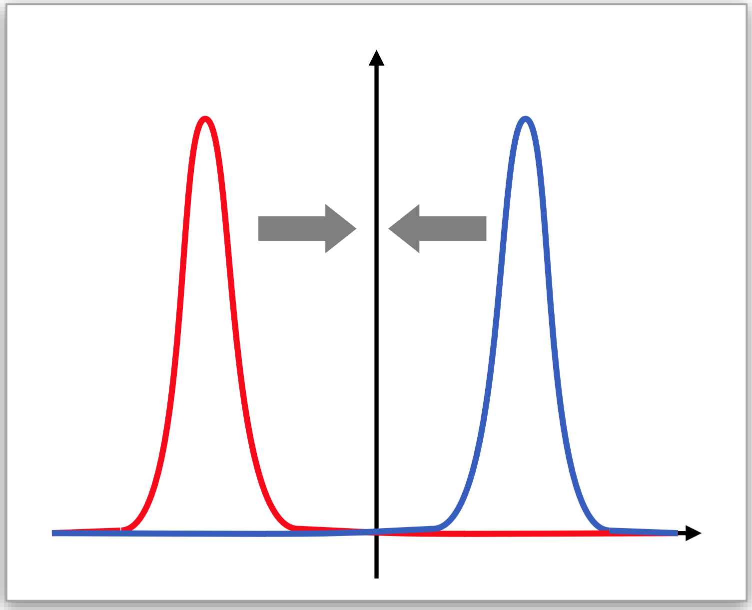
- Distributions $p, q: \mathbb{R}^d \rightarrow \mathbb{R}$
- Transport plan:
  Joint distribution $\pi(x, y)$ such that

$$\pi(x) = p(x), \quad \pi(y) = q(y)$$

- Wasserstein-distance

$$W_1(p, q) = \inf_{\substack{\text{distr. } \pi(x,y), \\ \pi(x)=p(x), \\ \pi(y)=q(y)}} \left(\mathbb{E}_{(x,y)\sim\pi}[||x - y||]\right)$$

Wasserstein GANs



Great idea

- Replace JS-distance in GAN-objective by Wasserstein-distance
  - No vanishing gradients
  - Fixes (*some*) convergence issues
- Problem:
  Looks very very highly totally unfortunately – intractable

Really great idea

- We can compute it indirectly

Kantorovich-Rubinstein Duality

Wasserstein distance

$$W_1(p,q) = \inf_{\substack{\text{distr. } \pi(x,y), \\ \pi(x)=p(x), \\ \pi(y)=q(y)}} \left(\mathbb{E}_{(x,y)\sim\pi}[\|x-y\|]\right)$$

Dual characterization

Lipschitz-constant bounded for f

$$W_1(p,q) = \sup_{\|f\|_L \le 1} \left(\mathbb{E}_{x\sim p}[f(x)] - \mathbb{E}_{y\sim q}[f(y)]\right)$$

What does it buy us?

- Still intractable (high-dim. f)
- But we can use a network to approximate f

Old Design

Old GAN

$$\min_{\theta} \max_{\phi} V\left(D_{\phi}, G_{\theta}\right)$$

$$V\left(D_{\phi}, G_{\theta}\right) = \mathbb{E}_{\mathbf{x} \sim p_{data}}\left[\log D_{\phi}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}\left[\log\left(1 - D_{\phi}\left(G_{\theta}(\mathbf{z})\right)\right)\right]$$

Gradients (downhill)

$$\nabla_{\phi} D = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\phi} \log D_{\phi}(\mathbf{x}_i) + \frac{1}{n}\sum_{i=1}^{n} \nabla_{\phi} \log\left(1 - D_{\phi}\left(G_{\theta}(\mathbf{z}_i)\right)\right)$$

$$\nabla_{\theta} = -\frac{1}{n}\sum_{i=1}^{n} \nabla_{\theta} \log\left(1 - D_{\phi}\left(G_{\theta}(\mathbf{z}_i)\right)\right)$$

Old Design

Old GAN – "improved" variant

$$\min_{\theta} \max_{\phi} V\left(D_{\phi}, G_{\theta}\right)$$

$$V\left(D_{\phi}, G_{\theta}\right) = \mathbb{E}_{\mathbf{x}\sim p_{data}}\left[\log D_{\phi}(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log\left(1 - D_{\phi}\left(G_{\theta}(\mathbf{z})\right)\right)\right]$$

Gradients (downhill)

$$\nabla_{\phi} D = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\phi} \log D_{\phi}(\mathbf{x}_i) + \frac{1}{n}\sum_{i=1}^{n} \nabla_{\phi} \log\left(1 - D_{\phi}\left(G_{\theta}(\mathbf{z}_i)\right)\right)$$

$$\nabla_{\theta} = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\theta} \log\left(D_{\phi}\left(G_{\theta}(\mathbf{z}_i)\right)\right)$$

New Design

Wassserstein GAN

$$\min_{\theta} \max_{\phi} W_{D_\phi}\left(D_\phi, G_\theta\right)$$

$$W_{D_\phi}(p_{data}, p_G) = \mathbb{E}_{\mathbf{x}\sim p_{data}}\left[D_\phi(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[D_\phi\left(G_\theta(\mathbf{z})\right)\right]$$

$$\text{with } \left\|\nabla_{\mathbf{x}} D_\phi(\mathbf{x})\right\| \leq 1$$

Gradients

$$\nabla_\phi D = \sum_{i=1}^{n} \nabla_\phi D_\phi(\mathbf{x}_i) + \sum_{i=1}^{n} \nabla_\phi D_\phi\left(G_\theta(\mathbf{z}_i)\right)$$

$$\nabla_\theta = -\sum_{i=1}^{n} \nabla_\theta \left(D_\phi\left(G_\theta(\mathbf{z}_i)\right)\right)$$

Modifications

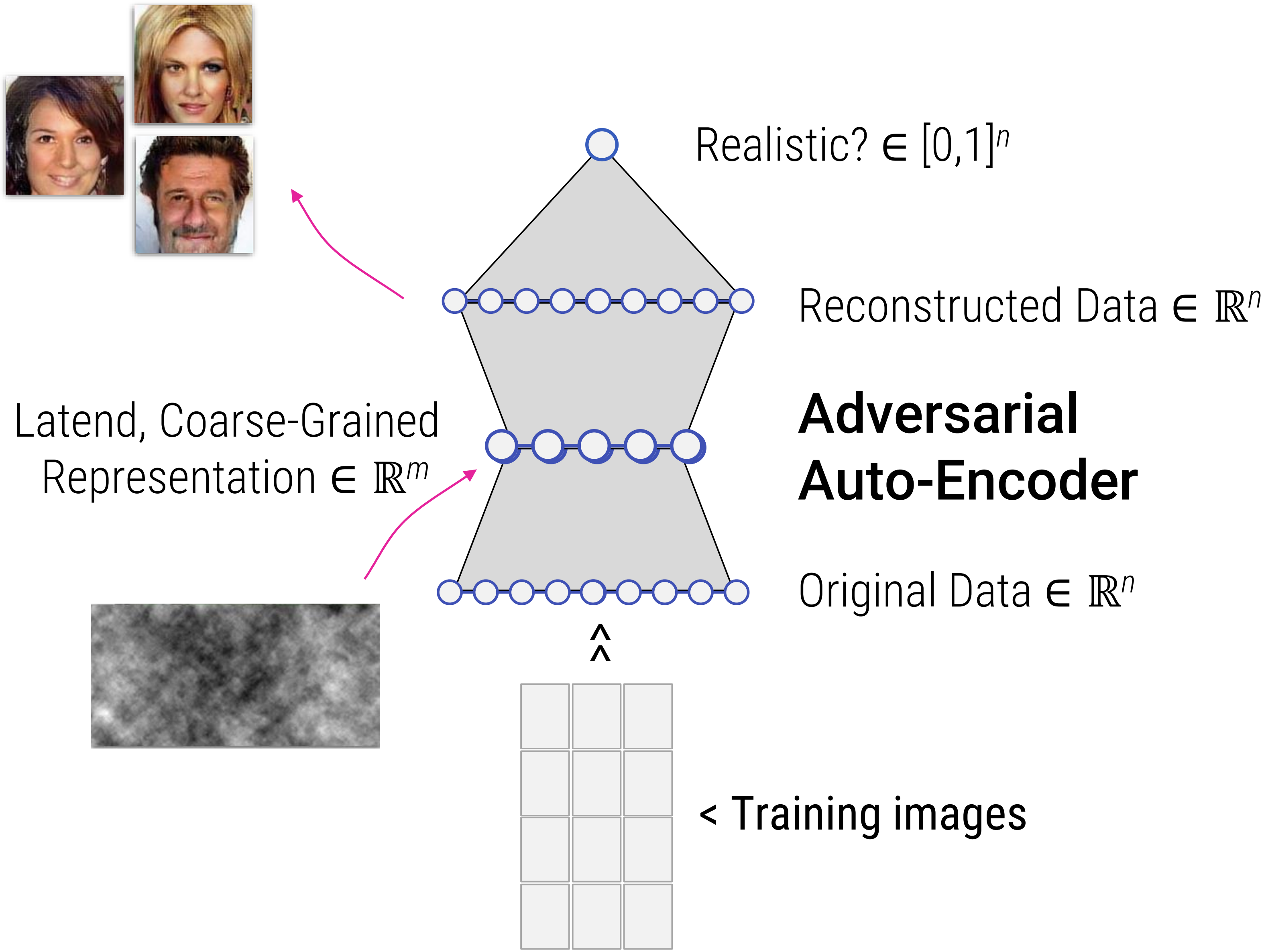
Discriminator D → "Critic" D

- Same architecture for D
- But no probabilistic output (no sigmoid)
- Needs Lipschitz-condition!
  - Option 1: Clipping of gradients
    - Original WGAN paper, does not work so well
  - Option 2: Gradient penalty
    - Penalty term $(||\nabla D|| - 1)^2$, works better
  - Option 3: Spectral normalization
    - Limit singular values of weight layers

Overall very similar to original GAN

Some Results

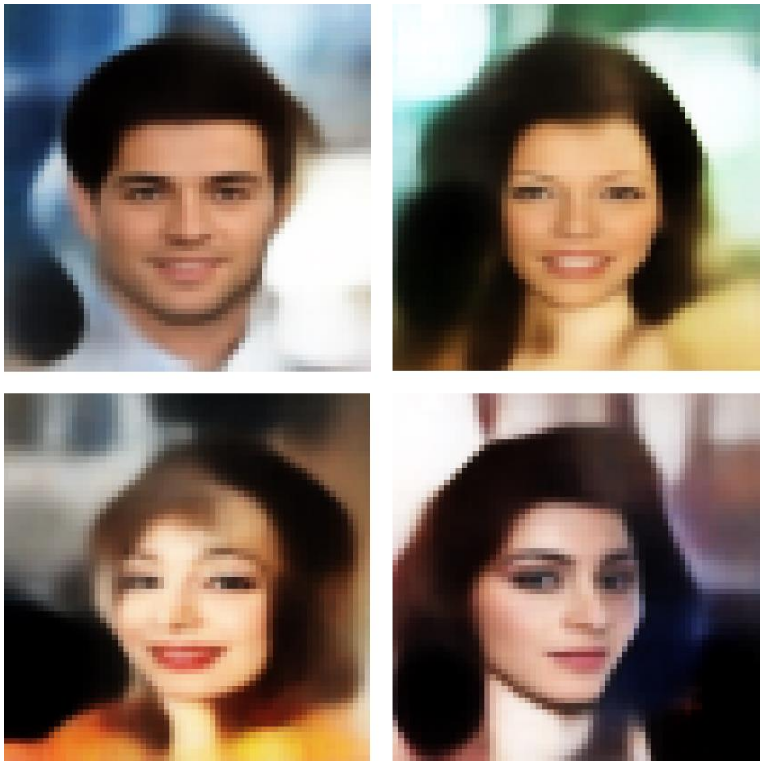
Simple Solution: AE+ GAN



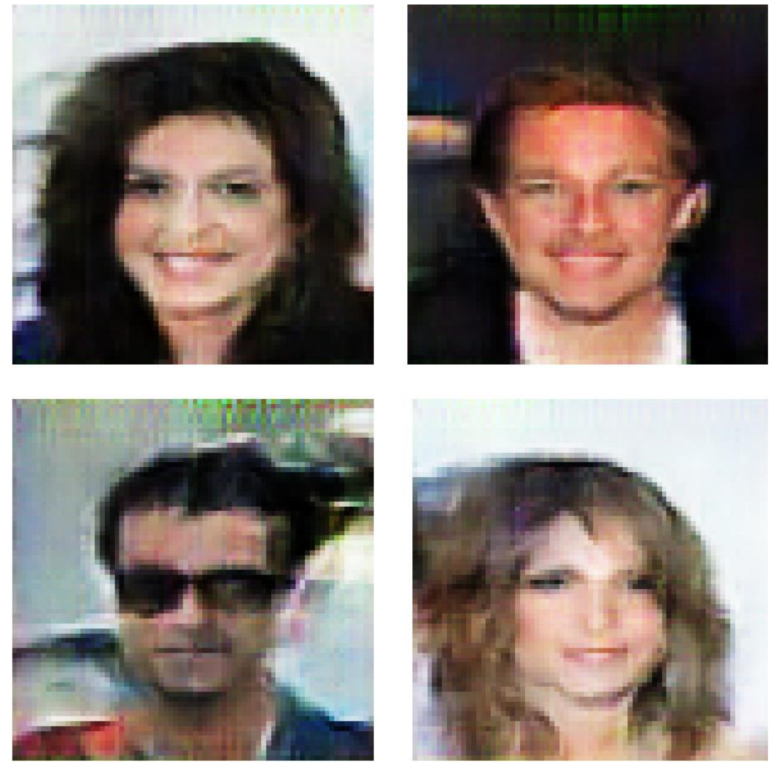
Noise → Images

MGANs [joint work with Chuan Li] trained on CelebA, GAN with AE conditioned on VGG-features

Wasserstein-GAN-GP (limited GPU)



Autoencoder
(PCA in latent space)



WGAN-GP
(generative adversarial network)

[results courtesy of D. Schwarz, D. Klaus, A. Rübe]

Style-Based GAN [Kerras et al. 2018]

Tero Karras, Samuli Laine, Timo Aila:
A Style-Based Generator Architecture for Generative Adversarial Networks, 2018
[image by Wikipedia user OwlsMcGee, https://en.wikipedia.org/wiki/File:Woman_1.jpg]

Summary

Generative Models

Generative deep networks

- Learning is a surprisingly difficult problem
  - Even if we assume/have “magic” regressors
- Difficulties
  - Relative Likelihood: Inverting networks
  - Absolute likelihood: proper normalization

Several tricks (we saw only an excerpt here)

- Autoencoders
- Flow-based models
- Autoregressive models
- Generative adversarial networks