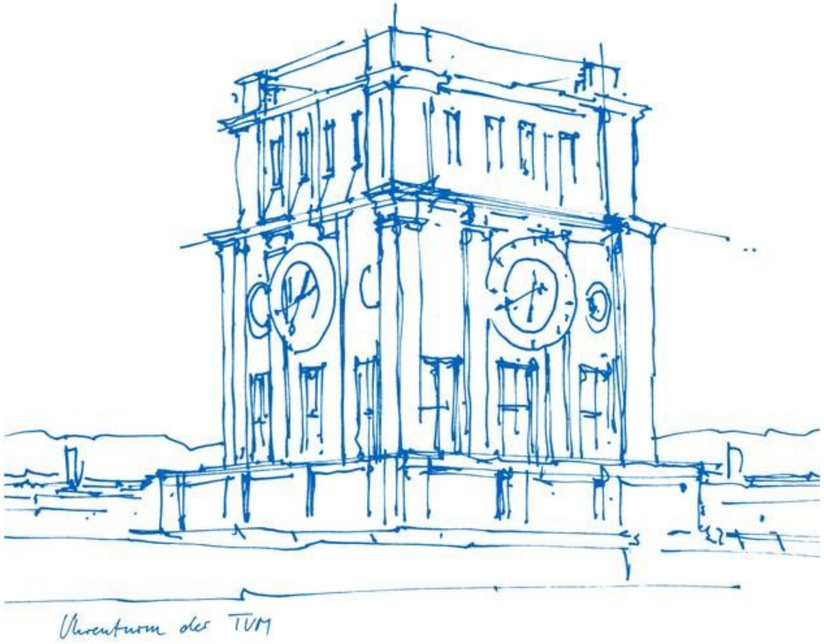
# Deep Learning

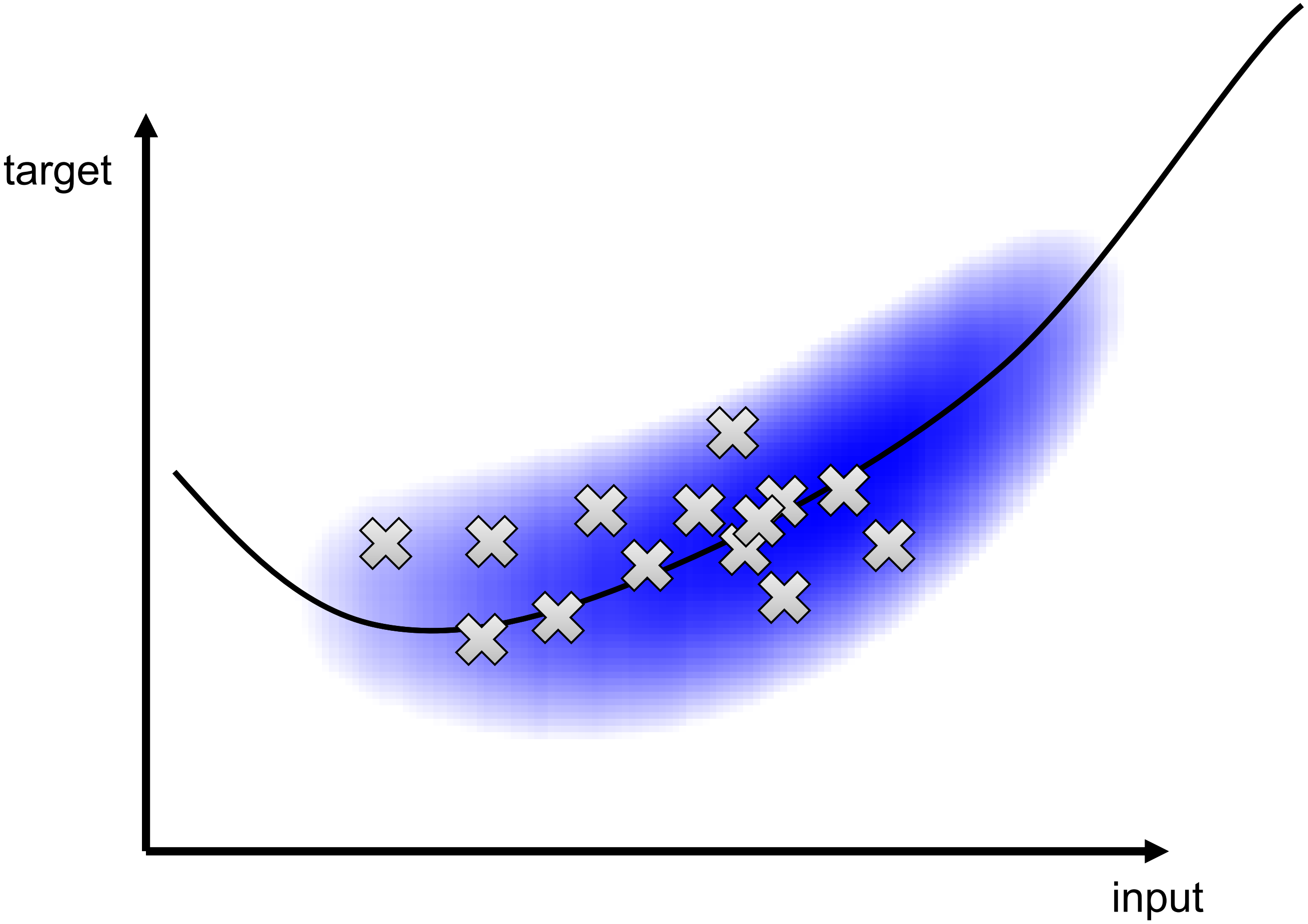
Vladimir Golkov

Technical University of Munich

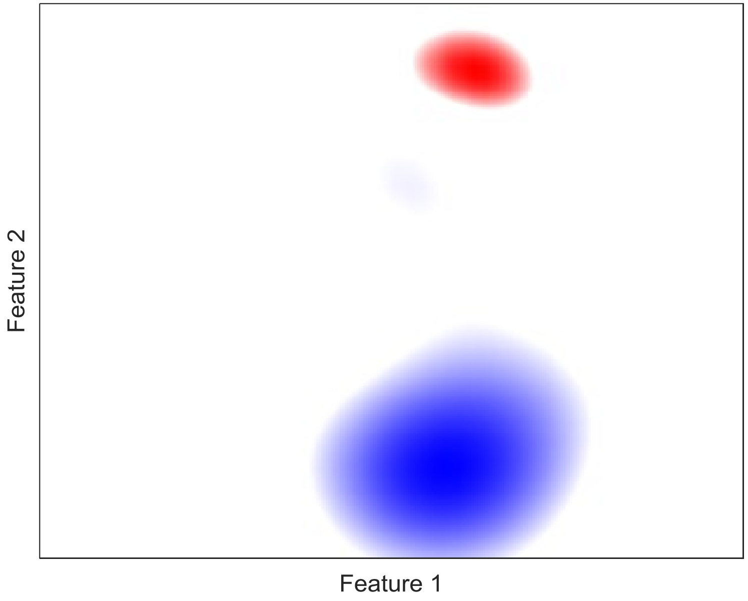
Computer Vision Group

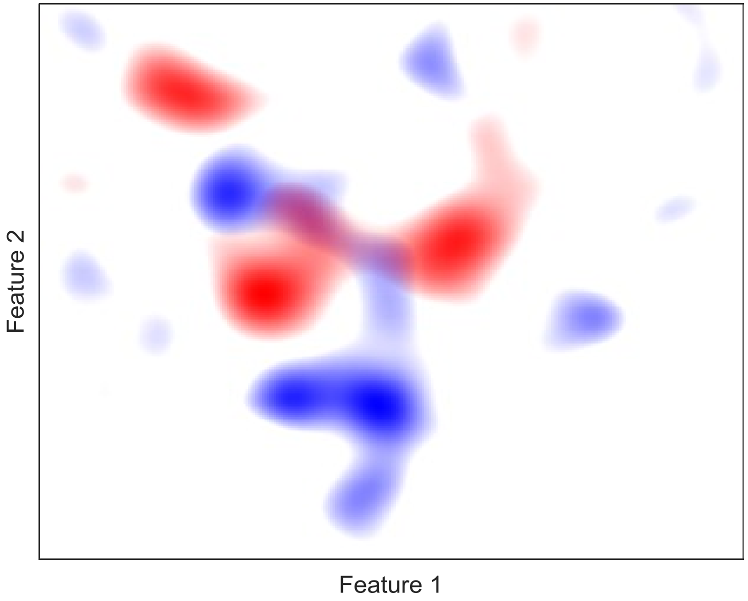


# 1D Input, 1D Output



# 2D Input, 1D Output: Data Distribution Complexity

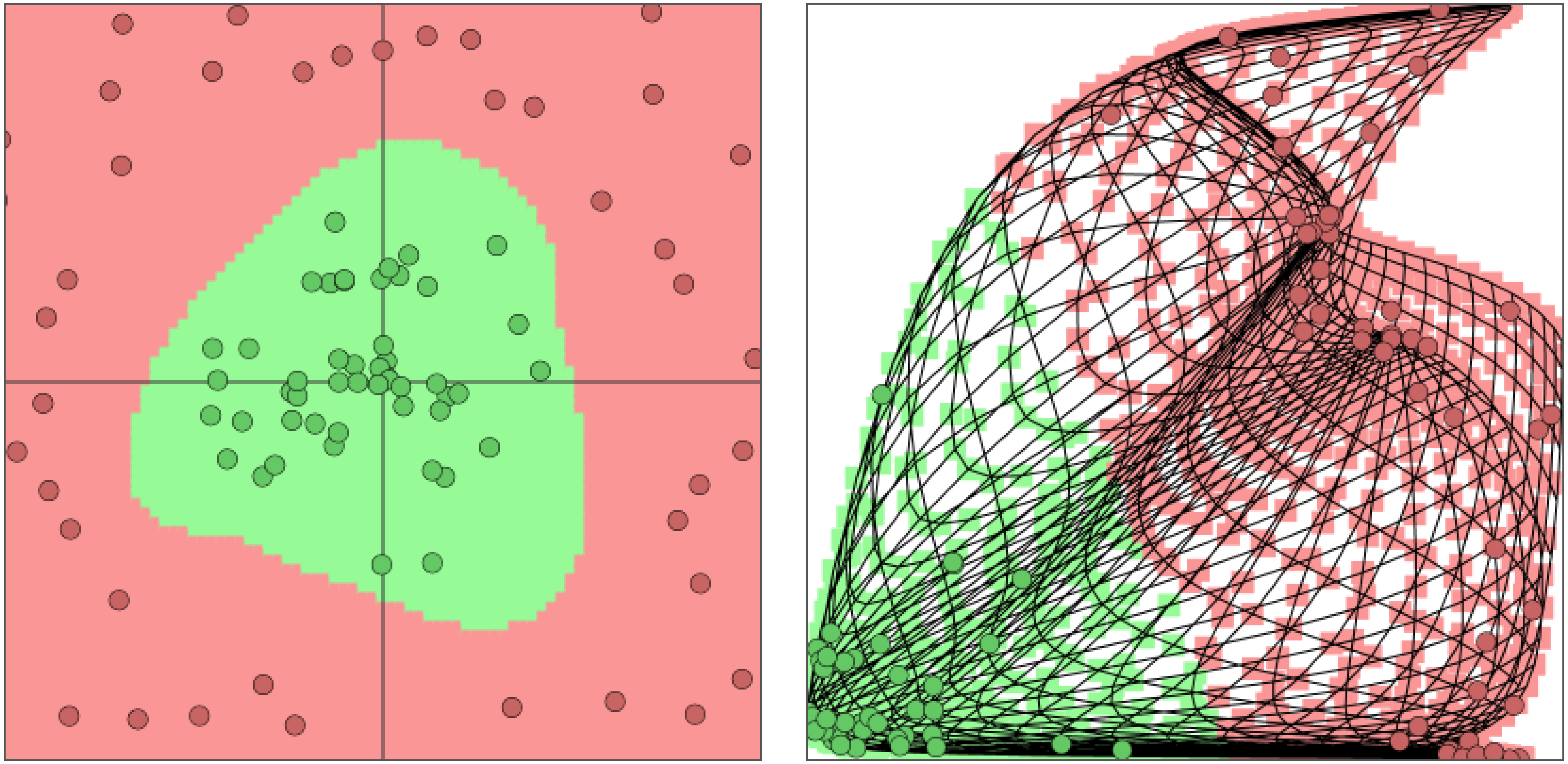




Imagine many dimensions
(data occupies sparse entangled regions)

Deep network: sequence of (simple) nonlinear disentangling transformations
(Transformation parameters are optimization variables)

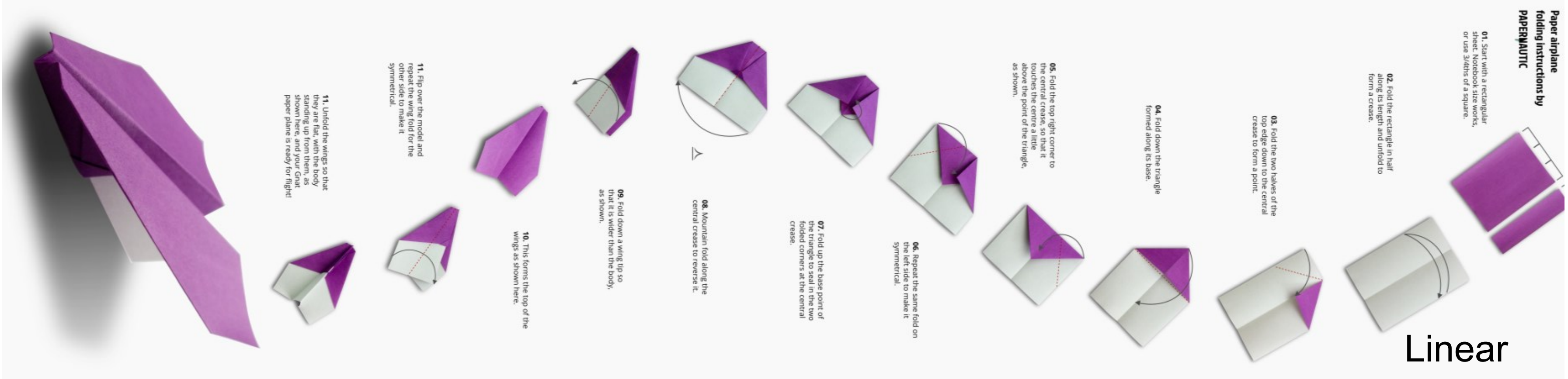
# Nonlinear Coordinate Transformation



http://cs.stanford.edu/people/karpathy/convnetjs/

Dimensionality may change!

# Sequence of Simple Nonlinear Coordinate Transformations



**Paper airplane folding instructions by PAPERNAUTIC**

**01.** Start with a rectangular sheet. Notebook size works, or use 3/4ths of a square.

**02.** Fold the rectangle in half along its length and unfold to form a crease.

**03.** Fold the two halves of the top edge down to the central crease to form a point.

**04.** Fold down the triangle formed along its base.

**05.** Fold the top right corner to the central crease, so that it touches the centre a little above the point of the triangle, as shown.

**06.** Repeat the same fold on the left side to make it symmetrical.

**07.** Fold up the base point of the triangle to seal in the two folded corners at the central crease.

**08.** Mountain fold along the central crease to reverse it.

**09.** Fold down a wing tip so that it is wider than the body, as shown.

**10.** This forms the top of the wings as shown here.

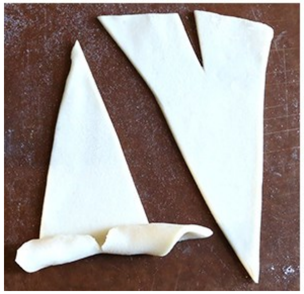
**11.** Flip over the model and repeat the wing fold for the other side to make it symmetrical.

**11.** Unfold the wings so that they are flat, with the body standing up from them, as shown here, and your Gnat paper plane is ready for flight!

Linear separation of purple and white sheet

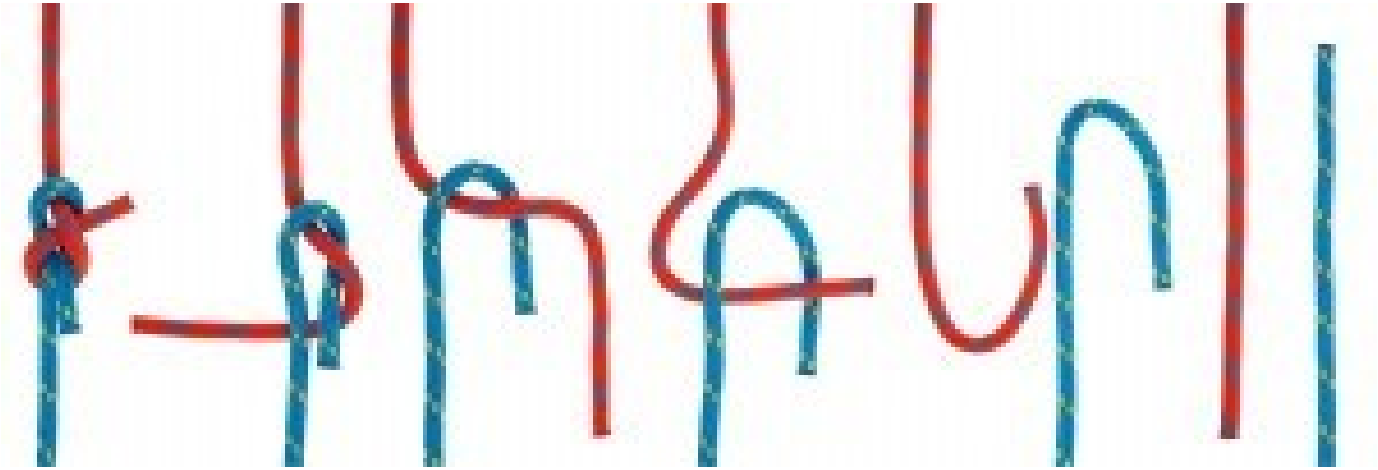
Rectified linear units do exactly such kinks

# Sequence of Simple Nonlinear Coordinate Transformations



Linear separation of dough and butter

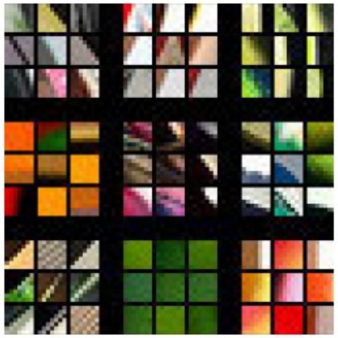
# Sequence of Simple Nonlinear Coordinate Transformations



Data is sparse (almost lower-dimensional)

# The Increasing Complexity of Features

Input features: RGB



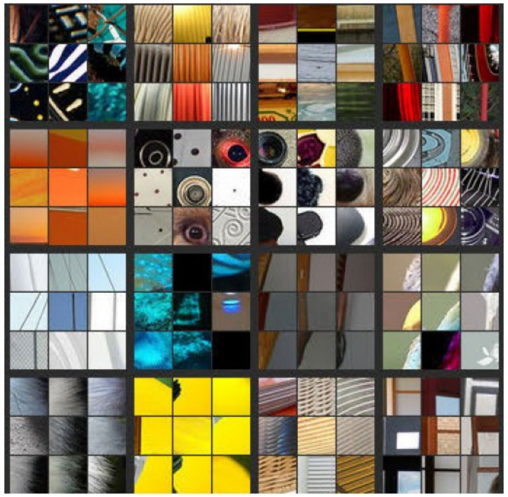
Layer 1

Feature space coordinates:

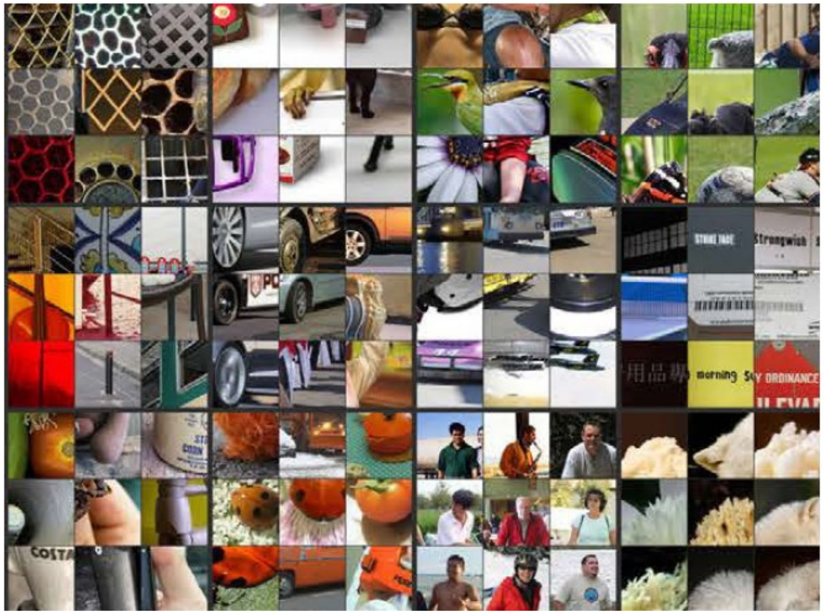
45° edge yes/no

Green patch yes/no

…



Layer 2



Layer 3

Feature space coordinates:

Person yes/no

Car wheel yes/no

…

[Zeiler & Fergus, ECCV 2014]

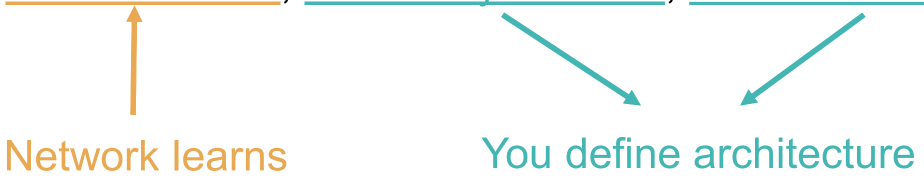
Increasing dimensionality, receptive field, invariance, complexity

"by design" "by convergence"

# Network Architecture: Your Decision!

Informed approach:

💡 If I were to choose layer-wise features *by hand* in a **smart, optimal** way, which features, how many features, with which receptive fields would I choose?



Network learns

You define architecture

Bottom-up approach: make dataset simple (e.g. simple samples and/or few samples), get a simple network (few layers, few neurons/filters) to work at least on the training set, then re-train increasingly complex networks on increasingly complex data

Top-down approach: use a network architecture that is known to work well on similar data, get it to work on your data, then tune the architecture if necessary

# FORWARD PASS

# (DATA TRANSFORMATION)

# Fully-Connected Layer

$x^{(0)}$ is input feature vector for neural network (one sample).

$x^{(L)}$ is output vector of neural network with $L$ layers.

Layer number $l$ has:

- Inputs (usually $x^{(l-1)}$, i.e. outputs of layer number $l-1$)
- Weight matrix $W^{(l)}$, bias vector $b^{(l)}$ - both trained (e.g. with stochastic gradient descent) such that network output $x^{(L)}$ for the training samples minimizes some objective (loss)
- Nonlinearity $s_l$ (fixed in advance, for example $\mathrm{ReLU}(z) \coloneqq \max\{0, z\}$)
  - Quiz: why is nonlinearity useful?
- Output $x^{(l)}$ of layer $l$

Transformation from $x^{(l-1)}$ to $x^{(l)}$ performed by layer $l$:

$$x^{(l)} = s_l\left(W^{(l)}x^{(l-1)} + b^{(l)}\right)$$

# Example

$$W^{(l)} = \begin{pmatrix} 0 & 0.1 & -1 \\ -0.2 & 0 & 1 \end{pmatrix}$$

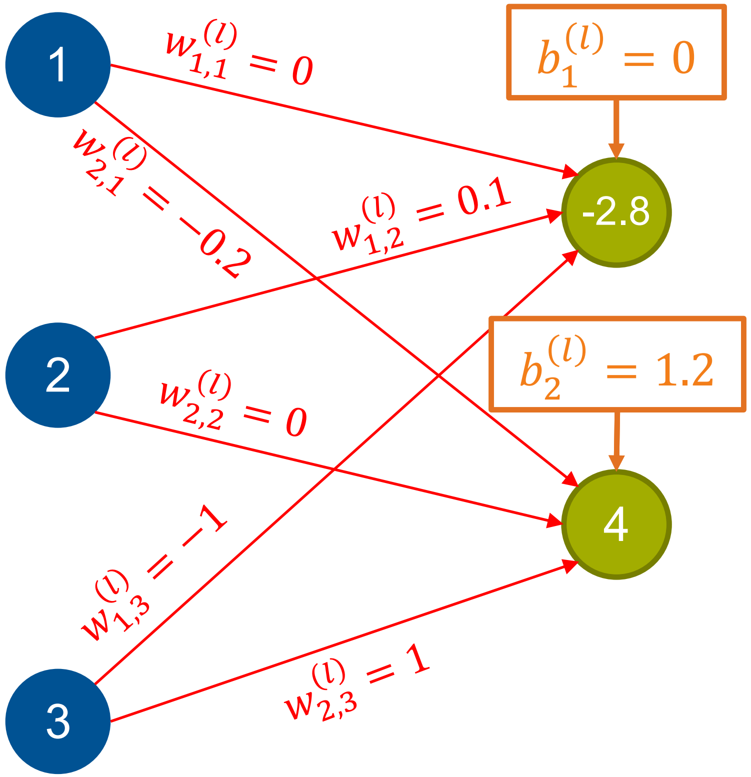
$$x^{(l-1)} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

$$b^{(l)} = \begin{pmatrix} 0 \\ 1.2 \end{pmatrix}$$

$$W^{(l)} x^{(l-1)} + b^{(l)} =$$

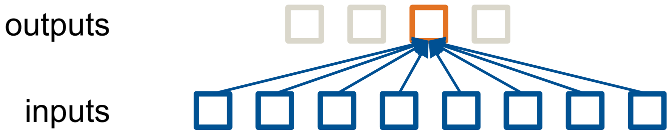
$$= \begin{pmatrix} 0 \cdot 1 + 0.1 \cdot 2 - 1 \cdot 3 + 0 \\ -0.2 \cdot 1 + 0 \cdot 2 + 1 \cdot 3 + 1.2 \end{pmatrix}$$

$$= \begin{pmatrix} -2.8 \\ 4 \end{pmatrix}$$

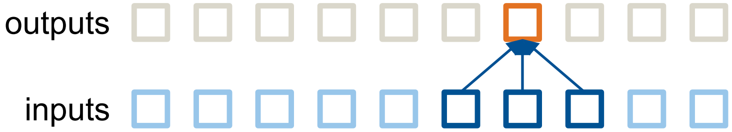


# Structured Data

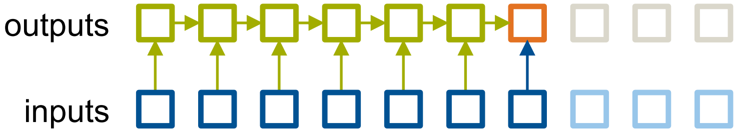
- “Zero-dimensional” data: multilayer perceptron



- Structured data: translation-covariant operations
  - Neighborhood structure: convolutional networks (2D/3D images, 1D bio. sequences, …)



  - Sequential structure (memory): recurrent networks (1D text, 1D audio, …)



# 2D Convolutional Layer

Appropriate for 2D structured data (e.g. images) where we want:

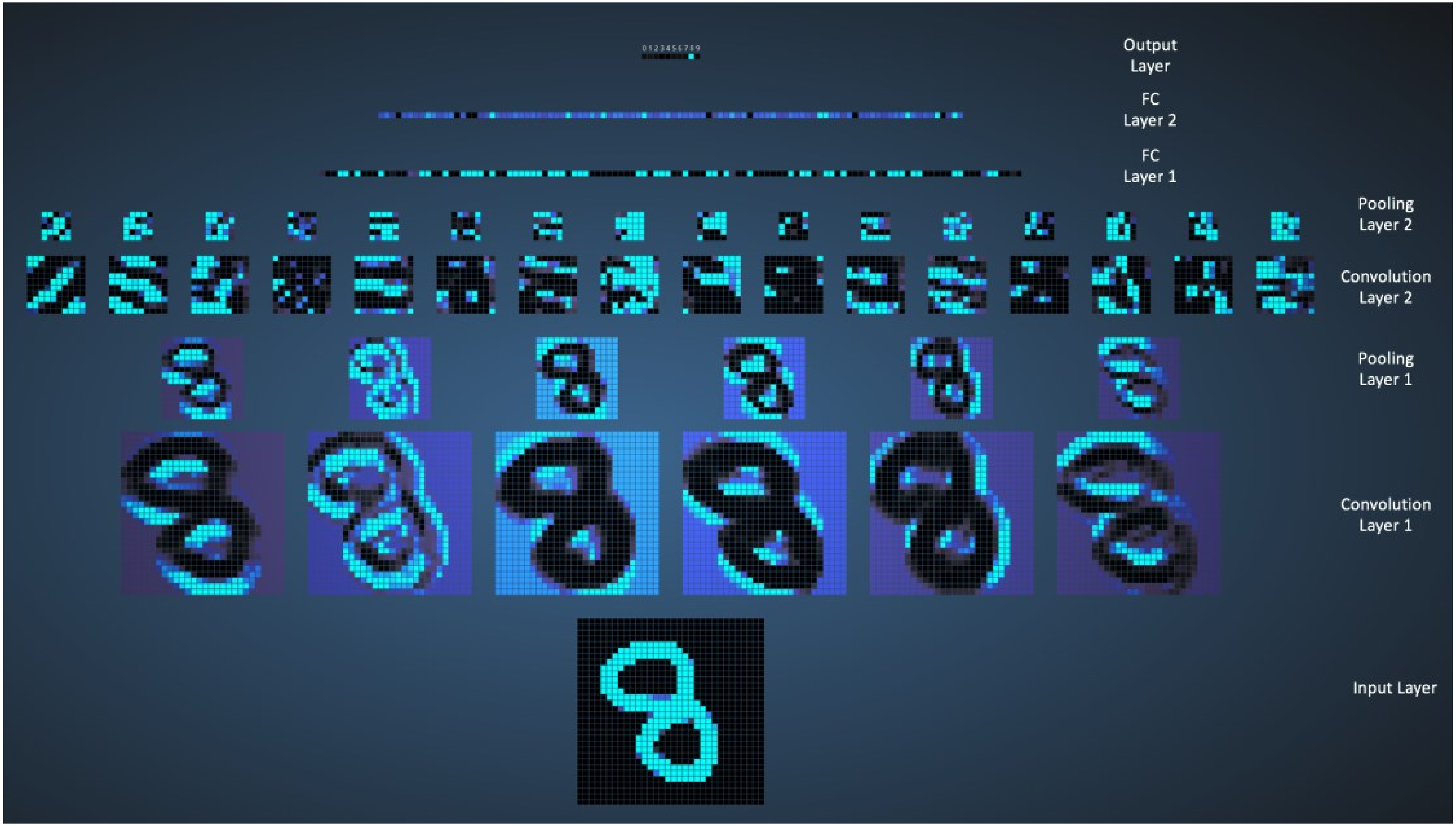
- Locality of feature extraction (far-away pixels do not influence local output)
- Translation-equivariance (shifting input in space ($i, j$ dimensions) yields same output shifted in the same way)

$$x_{i,j,k}^{(l)} = s_l\left(b_k^{(l)} + \sum_{\hat{\imath},\hat{\jmath},\hat{k}} w_{i-\hat{\imath},j-\hat{\jmath},\hat{k},k}^{(l)} x_{i,j,\hat{k}}^{(l-1)}\right)$$

- the size of $W$ along the $i, j$ dimensions is called “filter size”
- the size of $W$ along the $\hat{k}$ dimension is the number of input channels (e.g. three (red, green, blue) in first layer)
- the size of $W$ along the $k$ dimension is the number of filters (number of output channels)

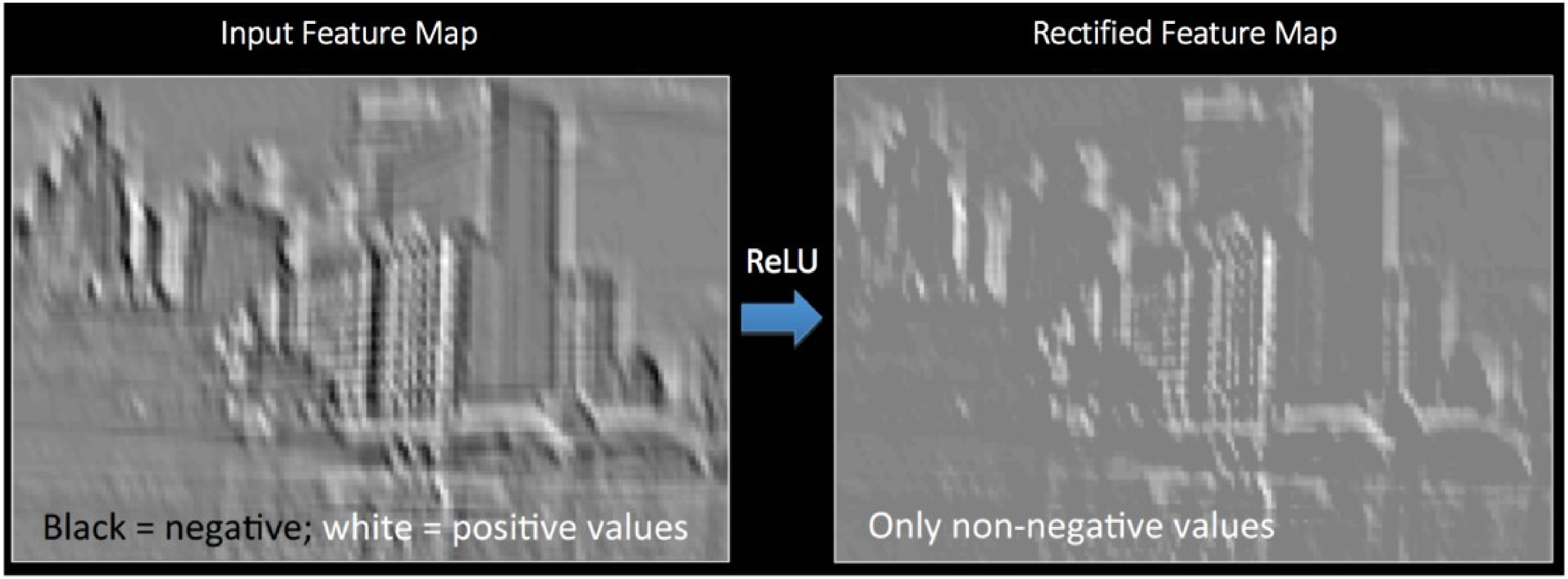
- Equivalent for 1D, 3D, ...
- http://cs231n.github.io/assets/conv-demo/

# Convolutional Network



Interactive: http://scs.ryerson.ca/~aharley/vis/conv/flat.html

# Nonlinearity



# Loss Functions

N-class classification:

- N outputs
- nonlinearity in last layer: softmax
- loss: categorical cross-entropy between outputs $x^{(L)}$ and targets $t$ (sum over all training samples)

2-class classification:

- 1 output
- nonlinearity in last layer: sigmoid
- loss: binary cross-entropy between outputs $x^{(L)}$ and targets $t$ (sum over all training samples)

2-class classification (alternative formulation)

- 2 outputs
- nonlinearity in last layer: softmax
- loss: categorical cross-entropy between outputs $x^{(L)}$ and targets $t$ (sum over all training samples)

Many regression tasks:

- linear output in last layer
- loss: mean squared error between outputs $x^{(L)}$ and targets $t$ (sum over all training samples)

# Neural Network Training Procedure

- Fix number $L$ of layers
- Fix sizes of weight arrays and bias vectors
  - For a fully-connected layer, this corresponds to the “number of neurons”
- Fix nonlinearities
- Initialize weights and biases with random numbers
- Repeat:
  - Select mini-batch (i.e. small subset) of training samples
  - Compute the gradient of the loss with respect to all trainable parameters (all weights and biases)
    - Use chain rule (“error backpropagation”) to compute gradient for hidden layers
  - Perform a gradient-descent step (or similar) towards minimizing the error
    - (Called “stochastic” gradient descent because every mini-batch is a random subset of the entire training set)
    - Important hyperparameter: learning rate (i.e. step length factor)
  - “Early stopping”: Stop when loss on validation set starts increasing (to avoid overfitting)

# Data Representation: Your Decision!

- The data representation should be natural
  (do not “outsource” known data transformations to the learning of the mapping)

- Make it easy for the network
  - For angles, we use sine and cosine to avoid the jump from 360° to 0°
    - Redundancy is okay!
  - Fair scale of features (and initial weights and learning rate) to facilitate optimization

- Data augmentation using natural assumptions

- Features from different distributions or missing: use several disentangled inputs to tell the network!
  - Trade-off: The more a network should be able to do, the *much* more data and/or better techniques are required

- https://en.wikipedia.org/wiki/Statistical_data_type
  - Categorical variable: one-hot encoding
  - Ordinal variable: cumulative sum of one-hot encoding [cf. Jianlin Cheng, arXiv:0704.1028]
  - etc

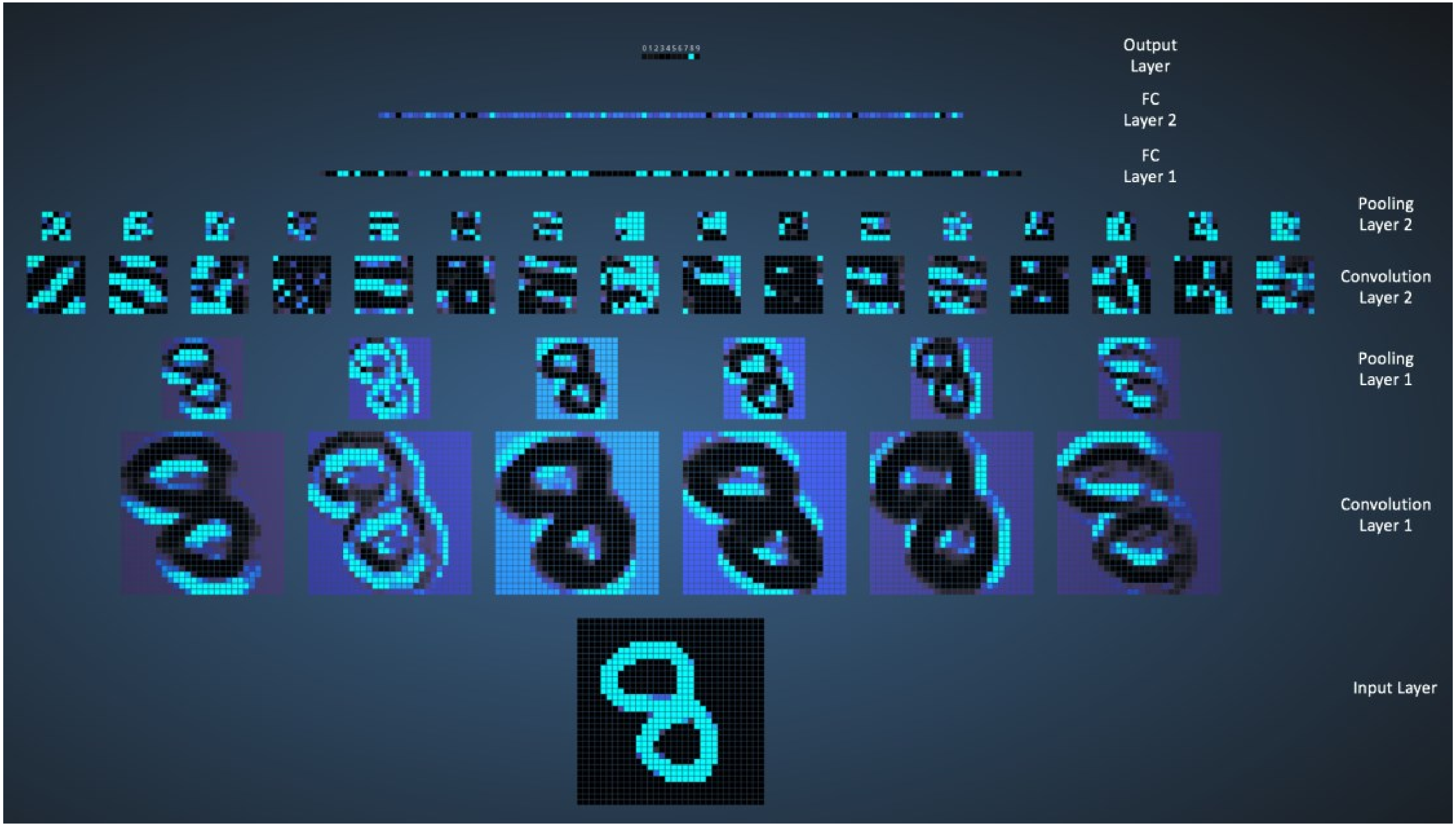
# Regularization to Avoid Overfitting

Impose meaningful invariances/assumptions about mapping in a hard or soft manner:

- Limited complexity of model: few layers, few neurons, weight sharing
- Locality and shift-equivariance of feature extraction: ConvNets
- Exact spatial locations of features don’t matter: pooling; strided convolutions
    - http://cs231n.github.io/assets/conv-demo/
- Deep features shouldn’t strongly rely on each other: dropout (randomly setting some deep features to zero during training)
- Data augmentation:
    - Known meaningful transformations of training samples
    - Random noise (e.g. dropout) in first or hidden layers
- Optimization algorithm tricks, e.g. early stopping

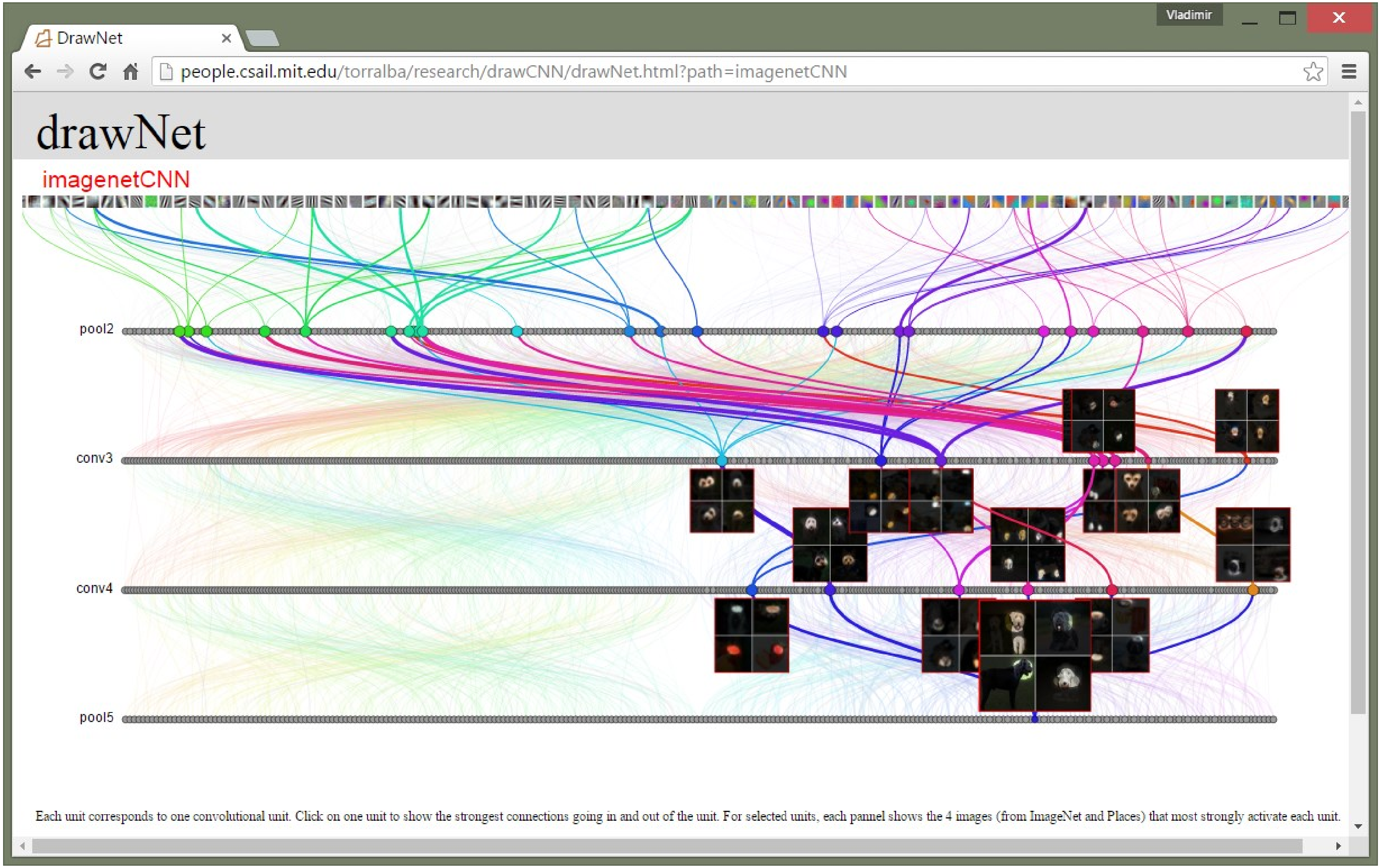
# WHAT DID THE NETWORK LEARN ?

# Convolutional Network



Interactive: http://scs.ryerson.ca/~aharley/vis/conv/flat.html

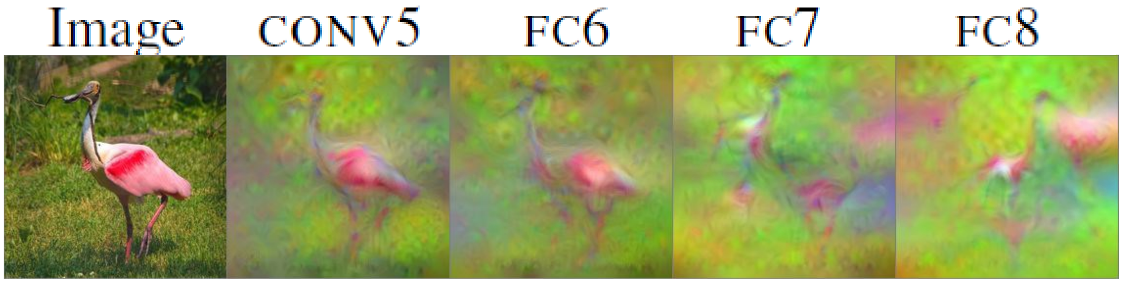
- Visualization: creative, no canonical way
- Look at standard networks to gain intuition


Vladimir
DrawNet
people.csail.mit.edu/torralba/research/drawCNN/drawNet.html?path=imagenetCNN
drawNet
imagenetCNN
pool2
conv3
conv4
pool5

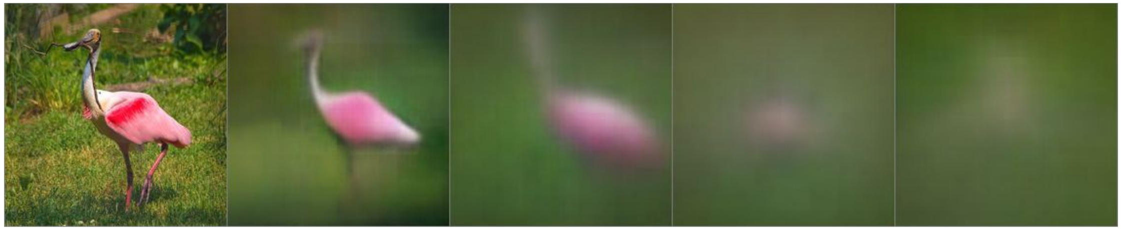
Each unit corresponds to one convolutional unit. Click on one unit to show the strongest connections going in and out of the unit. For selected units, each pannel shows the 4 images (from ImageNet and Places) that most strongly activate each unit.

# Image Reconstruction from Deep-Layer Features

Inverse problem:
Loss in feature space
[Mahendran & Vedaldi, CVPR 2015]

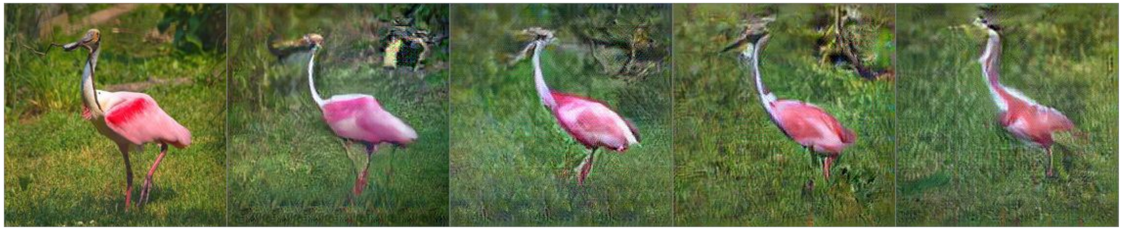


Another network:
Loss in image space
[Dosovitskiy & Brox, CVPR 2016]



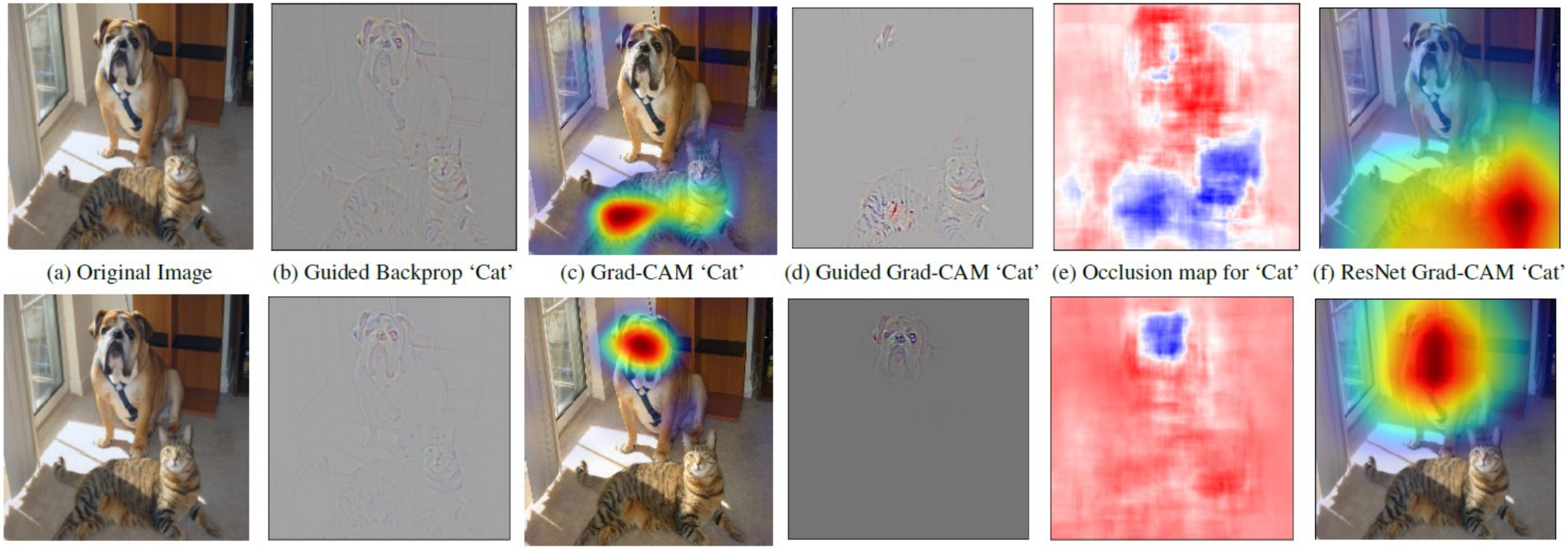
not entire information content can be retrieved, e.g. many solutions may be “known”, but only their “least-squares compromise” can be shown

Another network:
Loss in feature space + adversarial loss
+ loss in image space
[Dosovitskiy & Brox, NIPS 2016]



generative model “improvises” realistic details

# Reasons for Activations of Each Neuron

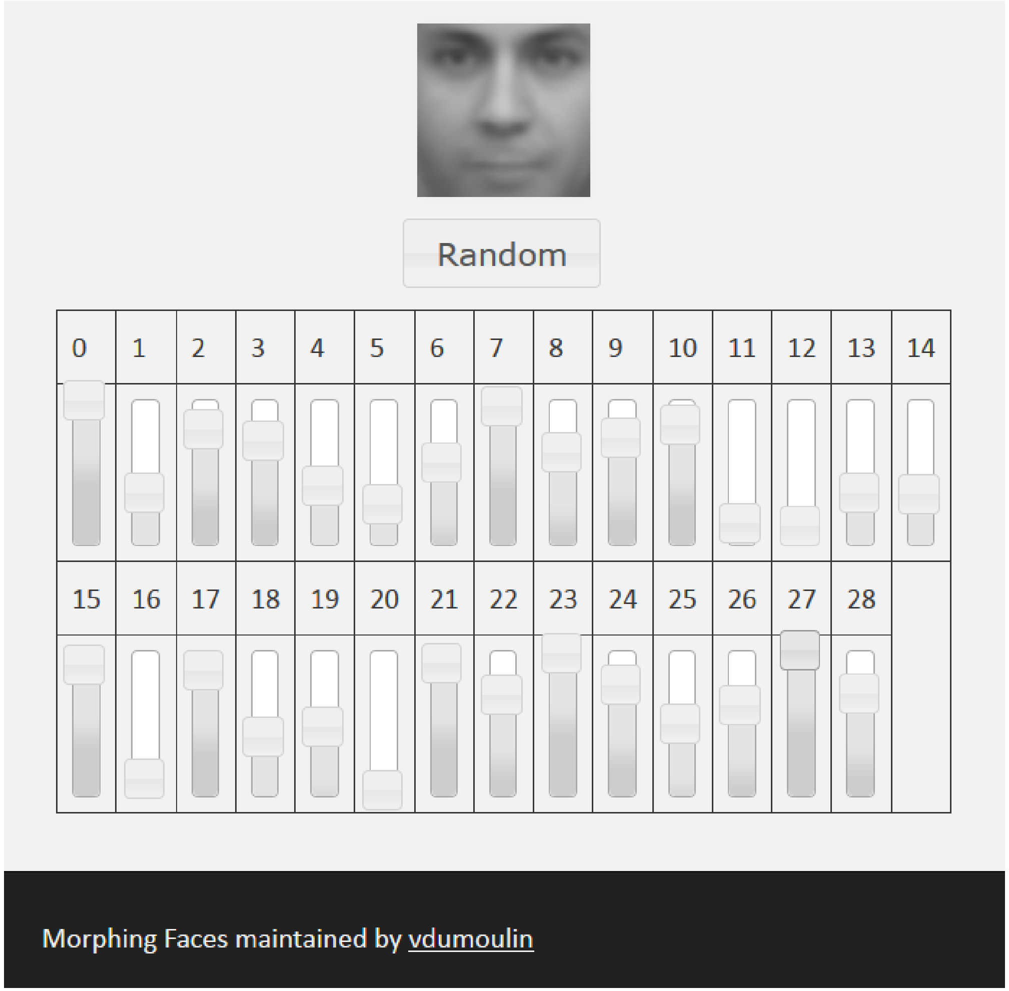


(a) Original Image (b) Guided Backprop 'Cat' (c) Grad-CAM 'Cat' (d) Guided Grad-CAM 'Cat' (e) Occlusion map for 'Cat' (f) ResNet Grad-CAM 'Cat'

(g) Original Image (h) Guided Backprop 'Dog' (i) Grad-CAM 'Dog' (j) Guided Grad-CAM 'Dog' (k) Occlusion map for 'Dog' (l) ResNet Grad-CAM 'Dog'

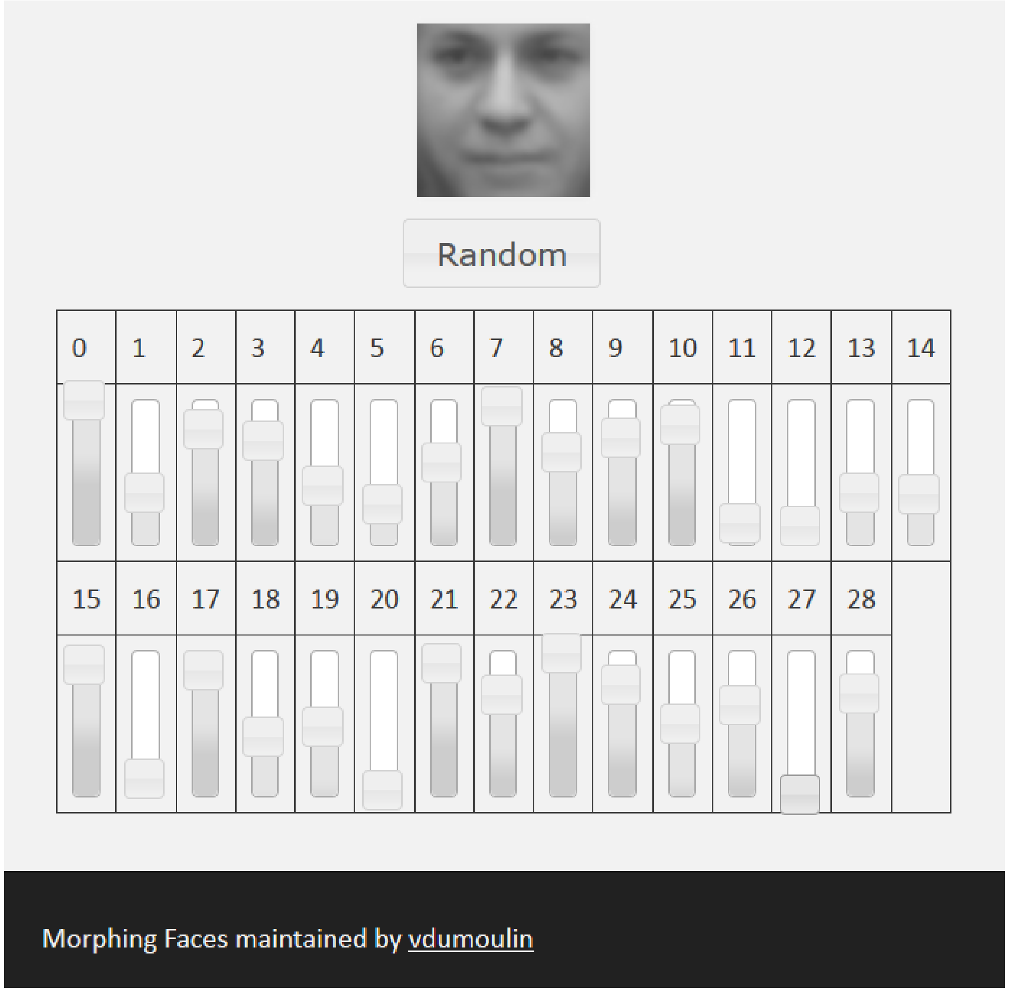
[Selvaraju et al., arXiv 2016]

# Network Bottleneck



Morphing Faces maintained by vdumoulin

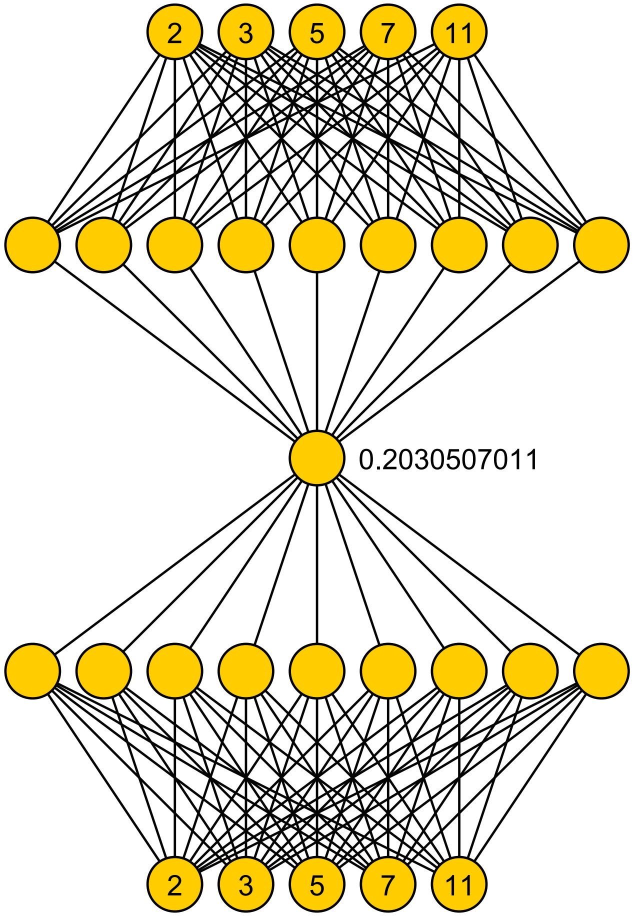
# Network Bottleneck



Morphing Faces maintained by vdumoulin

# An Unpopular and Counterintuitive Fact

- Any amount of information can pass through a layer with just one neuron! ("smuggled" as fractional digits)
- Quiz time: So why don't we have one-neuron layers?
- Learning to entangle (before) and disentangle (after) is difficult for the usual reasons:
  - Solution is not global optimum and not perfect
  - Optimum on training set ≠ optimum on test set
- Difficulty to entangle/disentangle used to our advantage: hard bottleneck (few neurons), soft bottleneck (additional loss terms):
  - dimensionality reduction
  - regularizing effect
  - understanding of intrinsic factors of variation of data



# Cost Functions: Your Decision!

- Appropriate measure of quality
- Desired properties of output (e.g. image smoothness, anatomic plausibility, ...)
- Desired properties of mapping (e.g. $f(x_1) \approx f(x_2)$ for $x_1 \approx x_2$)
- Weird data distributions: Class imbalance, domain adaptation, ...
- If derivative is zero on more than a null set, better use a “smoothed” version
- Apart from that, all you need is subdifferentiability almost everywhere (not even everywhere)
  - i.e. kinks and jumps are OK
  - e.g. ReLU

THANK YOU!