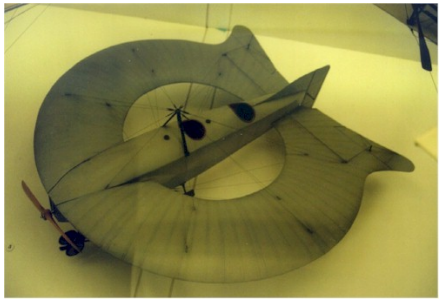
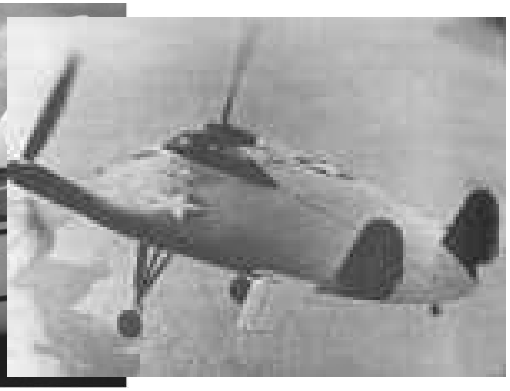
Learning Hierarchies of Invariant Features

Yann LeCun
Courant Institute of Mathematical Sciences
and
Center for Neural Science,
New York University

Challenges for Machine Learning, Vision, Signal Processing, AI, Neuroscience

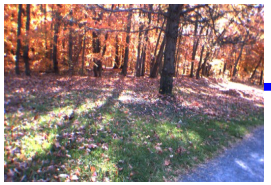
- How can learning build a perceptual system?
- How do we learn representations of the perceptual world?
- In ML/CV/ASR/MIR: How do we learn features (not just classifiers)?
- With good representations, we can learn categories from just a few examples.
- ML has neglected the question of learning representations, relying instead on domain expertise to engineer features and kernels.
- Deep Learning addresses the problem of learning representations
- Goal 1: biologically-plausible methods for deep learning
- Goal 2: representation learning for computer perception





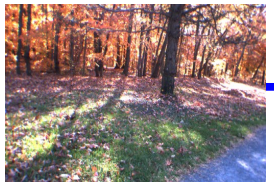
Architecture of "Mainstream" Image and Audio Recognition Systems

- Traditional way: handcrafted features + classifier



(hand-crafted) Feature Extraction → "Simple" Trainable Classifier →

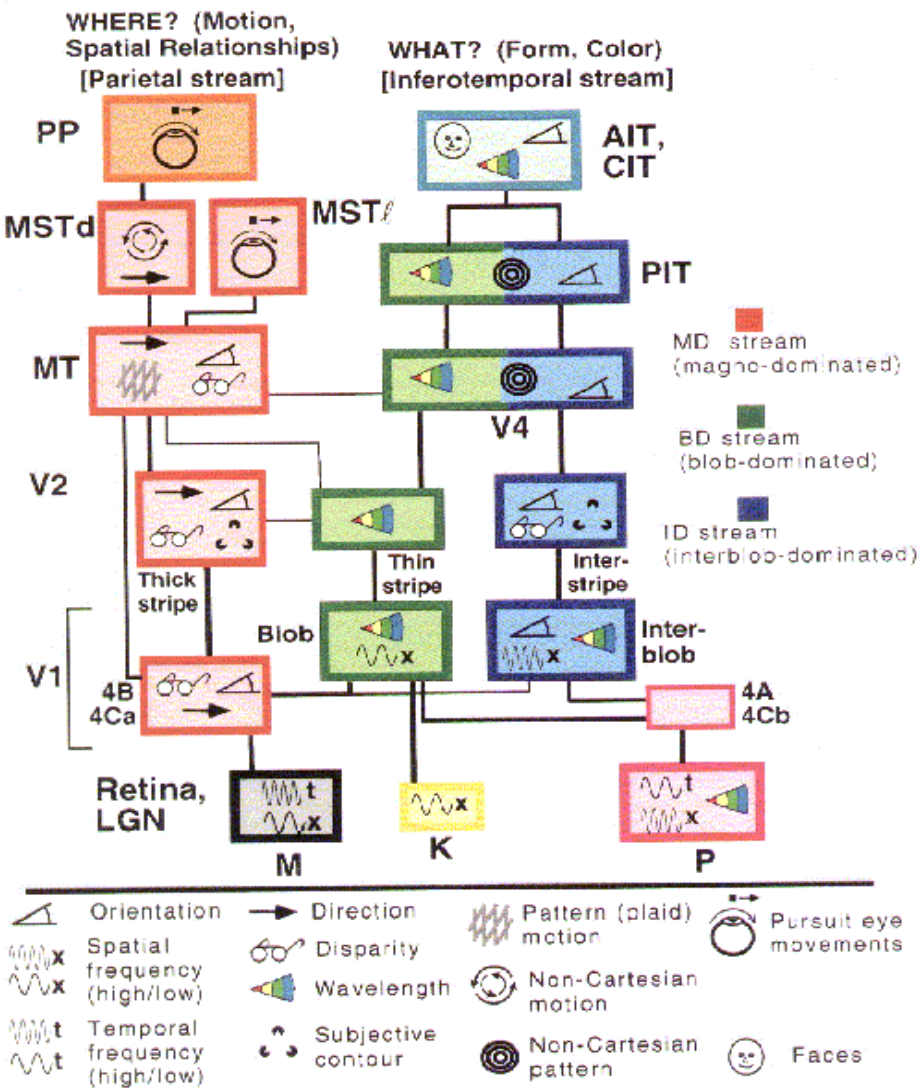
- Modern mainstream Approaches to Image and Speech Recognition
  - Mid-level features, often trained unsupervised

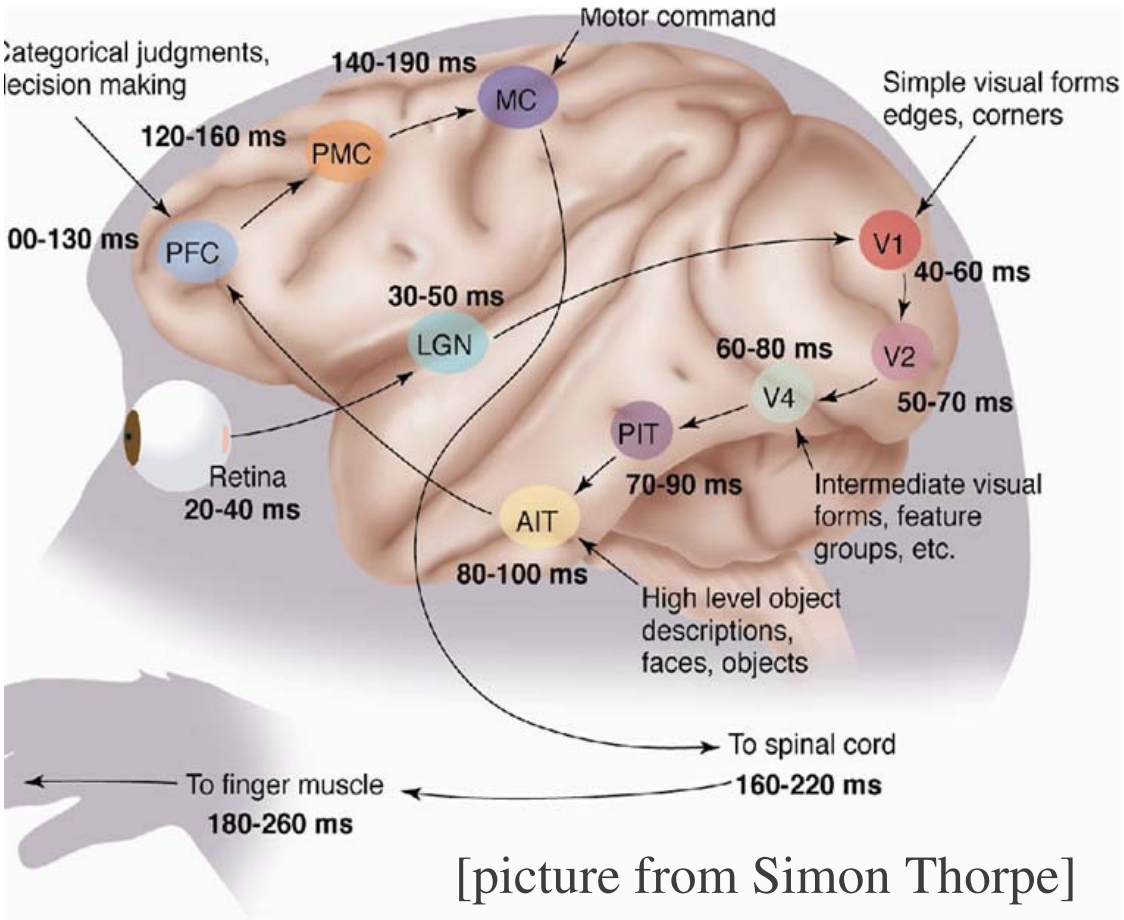


Low-Level Features (fixed) MFCC SIFT HoG → Mid-Level Features (unsupervised) Mix of Gaussians K-means Sparse Coding → Classifier (Supervised) Sometimes structured →

The Mammalian Visual Cortex is Hierarchical

- The ventral (recognition) pathway in the visual cortex has multiple stages
- Retina - LGN - V1 - V2 - V4 - PIT - AIT ....



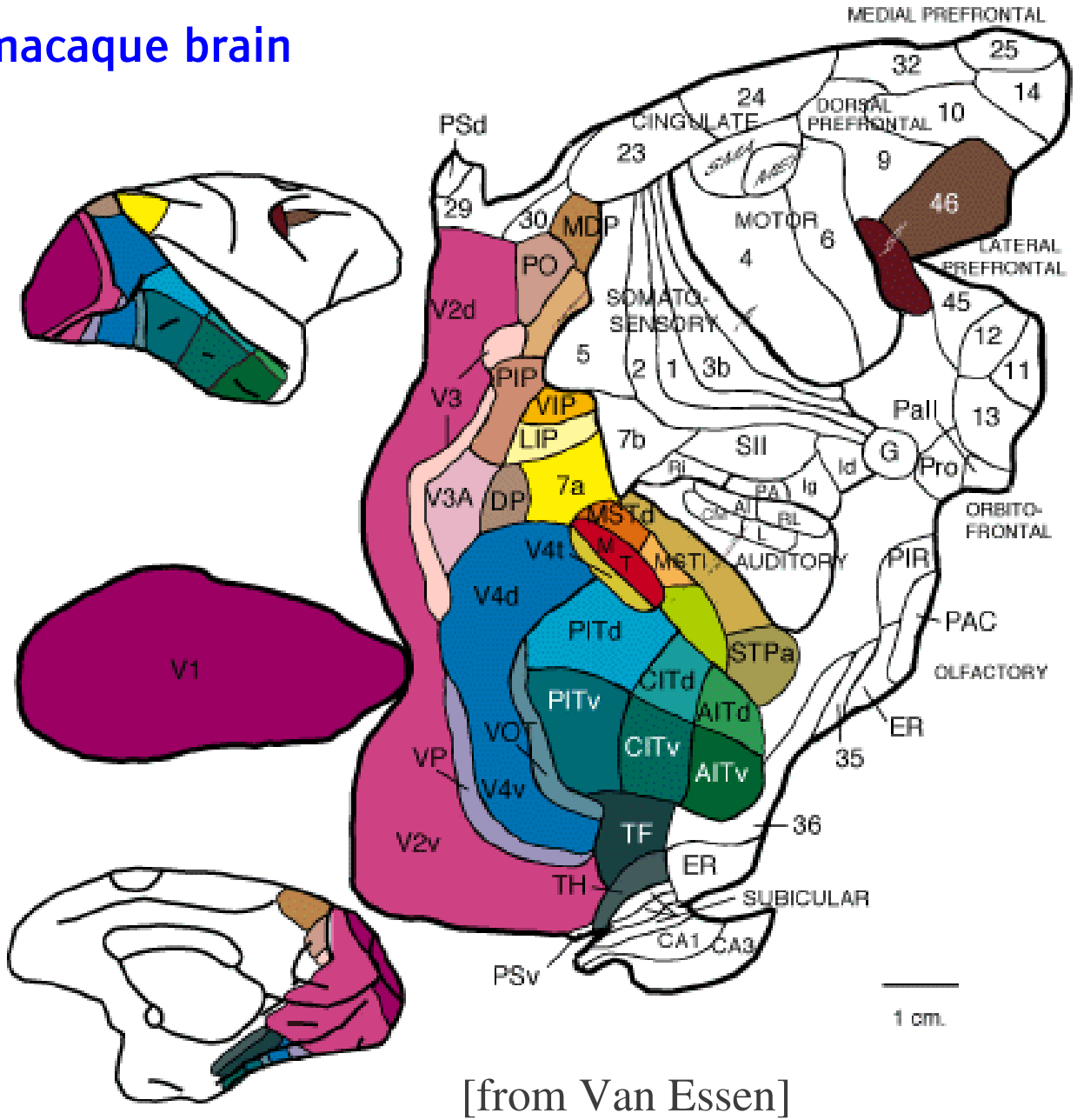


[picture from Simon Thorpe]

[Gallant & Van Essen]

Vision occupies a big chunk of our brains

- 1/3 of the macaque brain



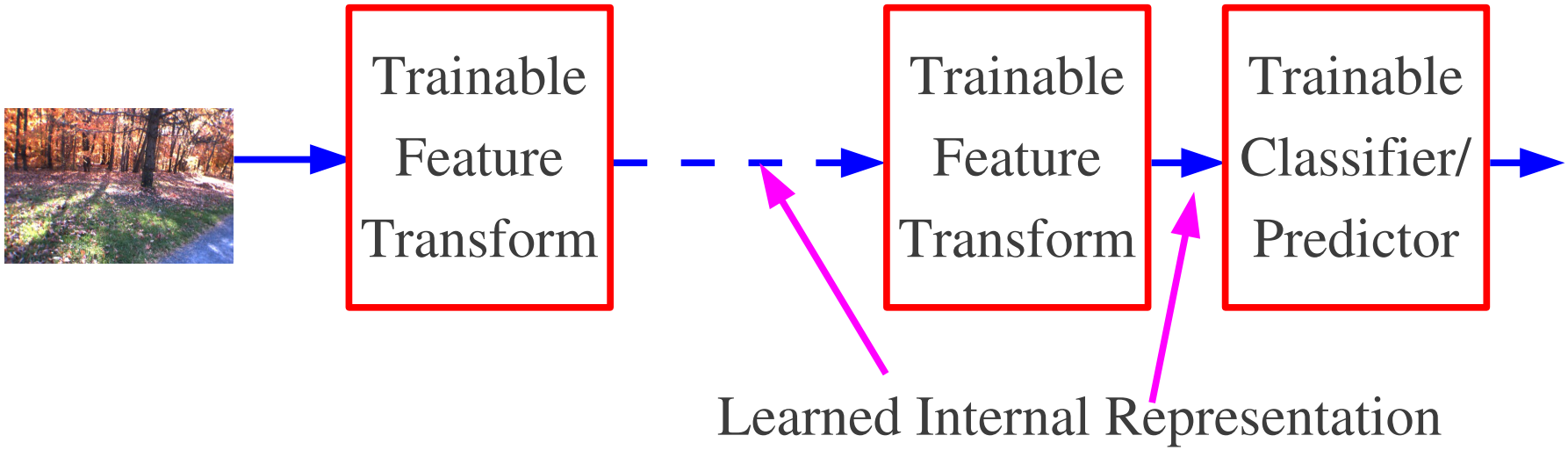
[from Van Essen]

The Primate's Visual System is Deep (LGN->V1->V2->V4->IT)

- The recognition of everyday objects is a very fast process.
  - The recognition of common objects is essentially "feed forward."
  - But not all of vision is feed forward.
- Much of the visual system (all of it?) is the result of learning
  - How much prior structure is there?
- If the visual system is deep (around 10 layers) and learned
- what is the learning algorithm of the visual cortex?
  - What learning algorithm can train neural nets as "deep" as the visual system (10 layers?).
  - Unsupervised vs Supervised learning
  - What is the loss function?
  - What is the organizing principle?
  - Broader question (Hinton): what is the learning algorithm of the neo-cortex?

Trainable Feature Hierarchies

- Why can't we make all the modules trainable?
- Proposed way: hierarchy of trained features



Do we really need deep architectures?

- We can approximate any function as close as we want with shallow architecture. Why would we need deep ones?

$$y = \sum_{i=1}^{P} \alpha_i K(X, X^i) \qquad y = F(W^1.F(W^0.X))$$

  - kernel machines and 2-layer neural net are "universal".
- Deep learning machines

$$y = F(W^K.F(W^{K-1}.F(....F(W^0.X)...)))$$

- Deep machines are more efficient for representing certain classes of functions, particularly those involved in visual recognition
  - they can represent more complex functions with less "hardware"
- We need an efficient parameterization of the class of functions that are useful for "AI" tasks.

Why are Deep Architectures More Efficient?

[Bengio & LeCun 2007 "Scaling Learning Algorithms Towards AI"]

- **A deep architecture trades space for time (or breadth for depth)**
  - more layers (more sequential computation),
  - but less hardware (less parallel computation).
  - Depth-Breadth tradoff
- **Example1: N-bit parity**
  - requires N-1 XOR gates in a tree of depth log(N).
  - requires an exponential number of gates of we restrict ourselves to 2 layers (DNF formula with exponential number of minterms).
- **Example2: circuit for addition of 2 N-bit binary numbers**
  - Requires O(N) gates, and O(N) layers using N one-bit adders with ripple carry propagation.
  - Requires lots of gates (some polynomial in N) if we restrict ourselves to two layers (e.g. Disjunctive Normal Form).
  - Bad news: almost all boolean functions have a DNF formula with an exponential number of minterms O(2^N).....

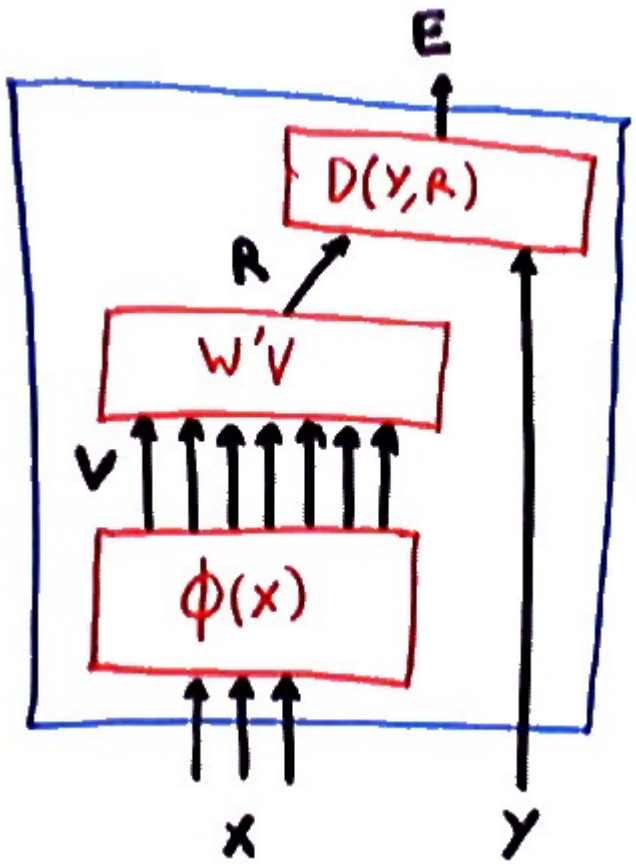
Strategies (a parody of [Hinton 2007])

- **Defeatism:** since no good parameterization of the "AI-set" is available, let's parameterize a much smaller set for each specific task through careful engineering (preprocessing, kernel....).
- **Denial:** kernel machines can approximate anything we want, and the VC-bounds guarantee generalization. Why would we need anything else?
  - unfortunately, kernel machines with common kernels can only represent a tiny subset of functions efficiently
- **Optimism:** Let's look for learning models that can be applied to the largest possible subset of the AI-set, while requiring the smallest amount of task-specific knowledge for each task.
  - There is a parameterization of the AI-set with neurons.
  - Is there an efficient parameterization of the AI-set with computer technology?
- Until very recently, much of the ML community oscillated between defeatism and denial.

Deep Supervised Learning is Hard

- The loss surface is non-convex, ill-conditioned, has saddle points, has flat spots.....
- For large networks, it will be horrible! (not really, actually)
- Back-prop doesn't work well with networks that are tall and skinny.
  - Lots of layers with few hidden units.
- Back-prop works fine with short and fat networks
  - But over-parameterization becomes a problem without regularization
  - Short and fat nets with fixed first layers aren't very different from SVMs.
- For reasons that are not well understood theoretically, back-prop works well when they are highly structured
  - e.g. convolutional networks.

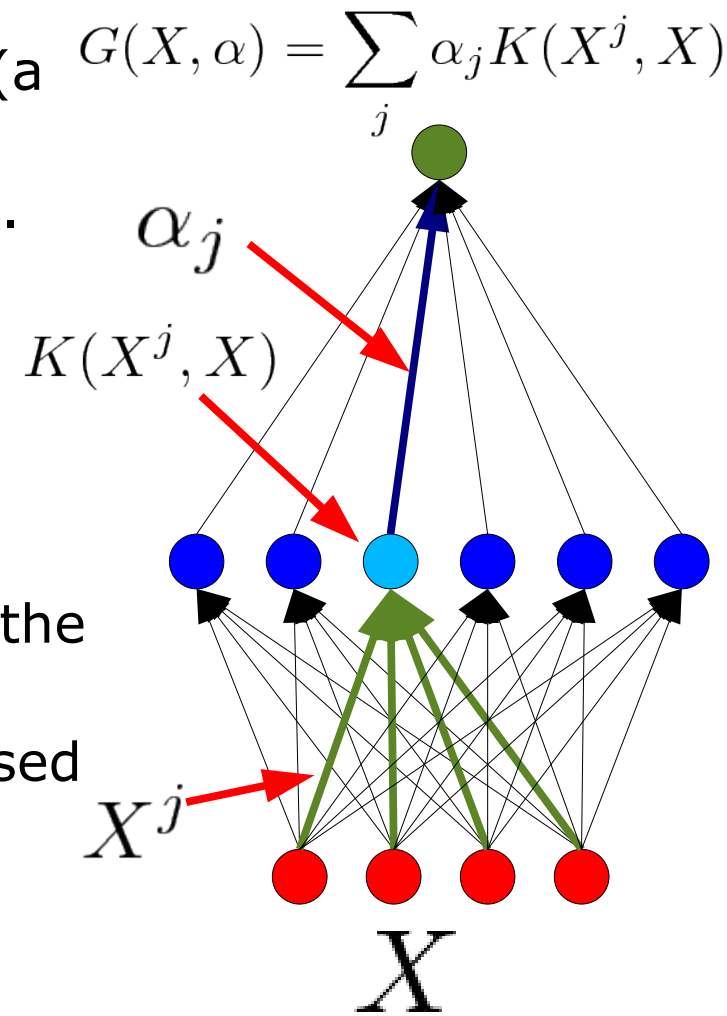
Fixed Preprocessing (features, kernels, basis functions)



- Map the inputs into a (higher dimensional) "feature" space
  - With more dimensions, the task is more likely to be linearly separable.
- Problem: how should we pick the features so that the task becomes linearly separable in the feature space?
  - **Classical approach 1**: we use our prior knowledge about the problem to hand-craft an appropriate feature set.
  - **Classical Approach 2**: we use a "standard" set of basis functions (RBFs....)

The Simplest form of Unsupervised Feature Extraction: Kernels Machines:

- Simplest approach: The Kernel Method (thanks to Grace Wahba's Representer Theorem)
  - Make each basis function a "bump" function (a template matcher).
  - Place one bump around each training sample.
  - Compute a linear combination of the bumps.
  - In the "bump space", we get one separate dimension for each training sample, so if the bumps are narrow enough, we can learn any mapping on the training set.
  - To generalize on unseen samples, we adjust the bump widths and we regularize the weights.
  - Regularize, so only the "useful" bumps are used
  - We get a **Support Vector Machine.**
- Problem: an SVM is a glorified template matcher which is only as good as its kernel.



Trainable Front-End, Structured Architectures

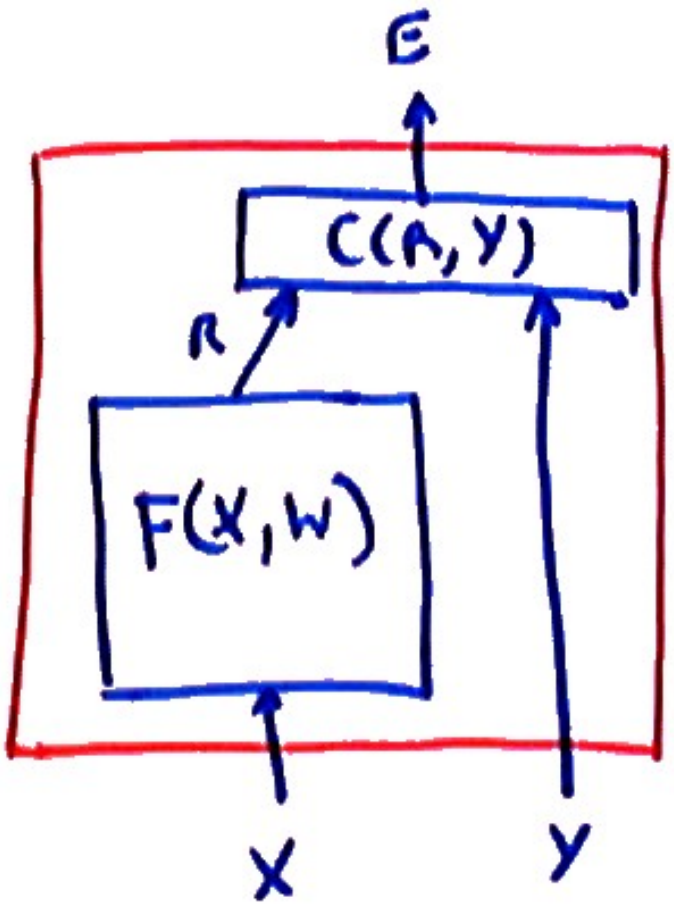
- **The Solutions:**
  - Invariance: Do not use a fixed front-end, **make it trainable**, so it can learn to extract invariant representations
  - Structure: Do not use simple linearly-parameterized classifiers, use architectures whose inference process involves multiple non-linear decisions, as well as search and "reasoning".
- **We need total flexibility in the design of the architecture of the machine:**
  - So that we can tailor the architecture to the task
  - So that we can build our prior knowledge about the task into the architecture
- **Multi-Module Architectures.**

Backprop and Training Multi-Module Architectures

[Bottou & Gallinari NIPS 1991]

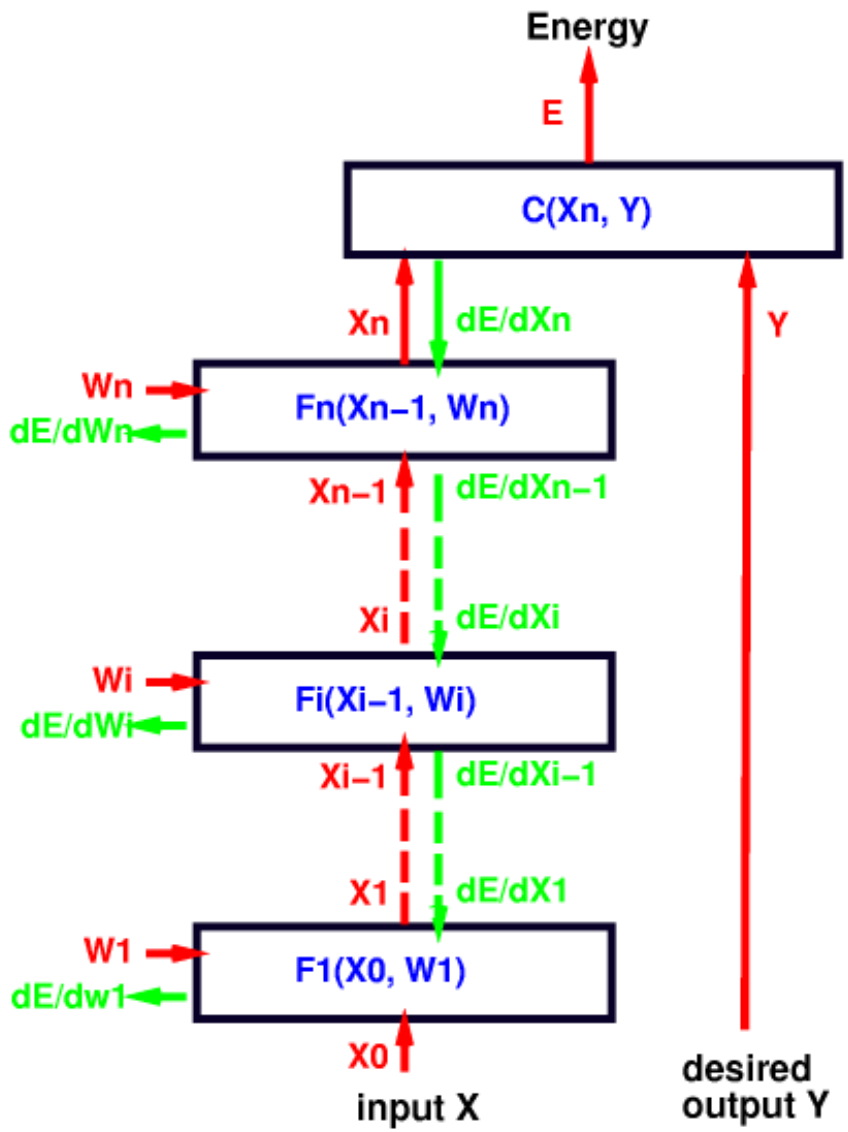
[LeCun, Bottou, Bengio, Haffner, 1998]

Multi-Module Architectures and Backprop



- **For Supervised Learning**
  - We allow the function F(W,X) to be non-linearly parameterized in W.
  - This allows us to play with a large repertoire of functions with rich class boundaries.
  - We assume that F(W,X) is differentiable almost everywhere with respect to W.

Multimodule Systems: Cascade



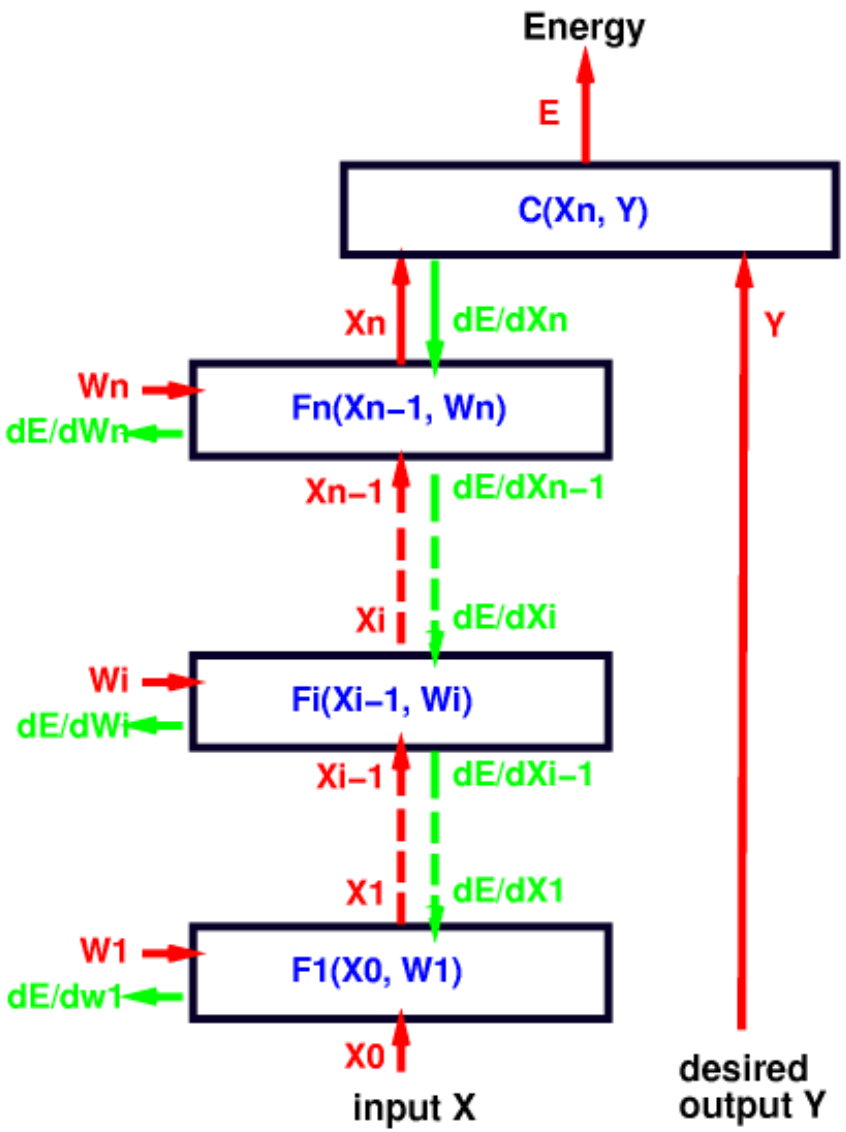
- Complex learning machines can be built by assembling modules into networks
- Simple example: sequential/layered feed-forward architecture (cascade)
- Forward Propagation:

$$\text{let } X = X_0,$$

$$X_i = F_i(X_{i-1}, W_i) \quad \forall i \in [1, n]$$

$$E(Y, X, W) = C(X_n, Y)$$

Multimodule Systems: Implementation



- **Each module is an object**
  - Contains trainable parameters
  - Inputs are arguments
  - Output is returned, but also stored internally
  - Example: 2 modules m1, m2
- **Torch7 (by hand)**
  - `hid = m1:forward(in)`
  - `out = m2:forward(hid)`
- **Torch7 (using the nn.Sequential class)**
  - `model = nn.Sequential()`
  - `model:add(m1)`
  - `model:add(m2)`
  - `out = model:forward(in)`

Gradient of the Loss, gradient of the Energy

- We assumed early on that the loss depends on W only through the terms $E(W, Y, X^i)$:

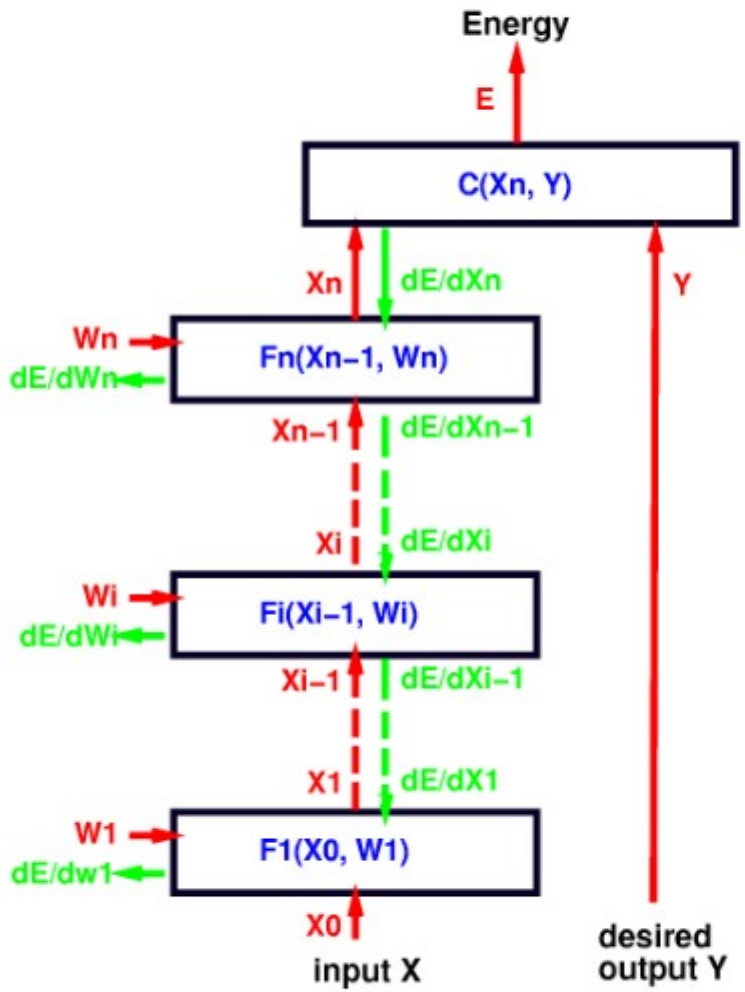
$$L(W, Y^i, X^i) = L(Y^i, E(W, 0, X^i), E(W, 1, X^i), \ldots, E(W, k-1, X^i))$$

- therefore:

$$\frac{\partial L(W, Y^i, X^i)}{\partial W} = \sum_Y \frac{\partial L(W, Y^i, X^i)}{\partial E(W, Y, X^i)} \frac{\partial E(W, Y, X^i)}{\partial W}]$$

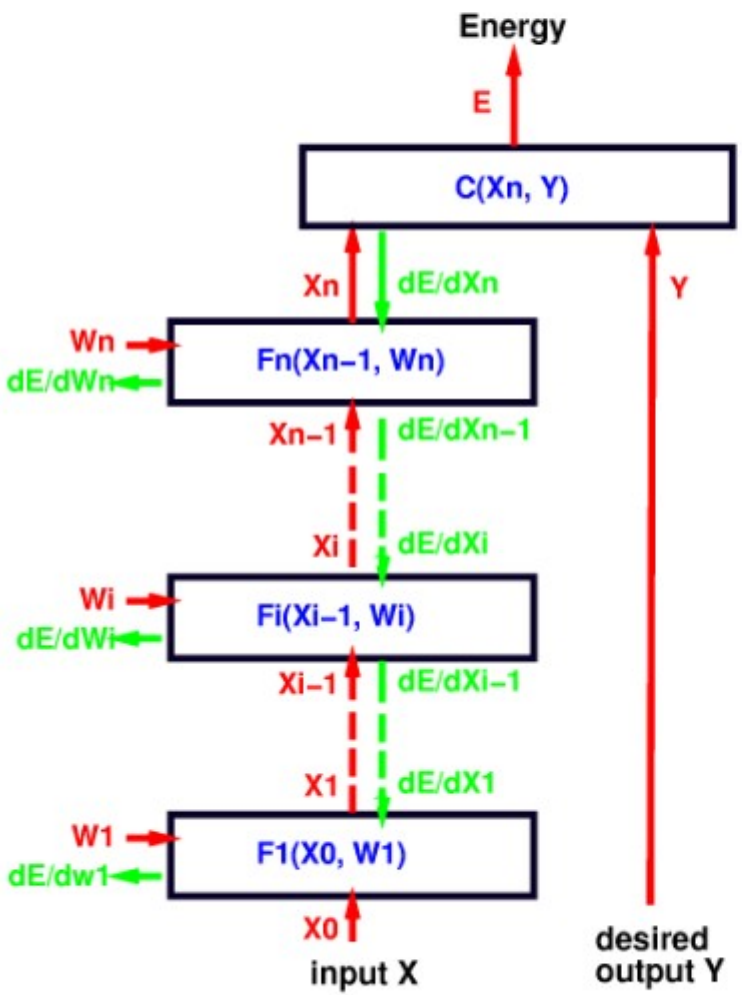
- We only need to compute the terms $\frac{\partial E(W,Y,X^i)}{\partial W}$
- Question: How do we compute those terms efficiently?

Computing the Gradients in Multi-Layer Systems



- To train a multi-module system, we must compute the gradient of E with respect to all the parameters in the system (all the W_i).
- Let's consider module i whose fprop method computes $X_i = F_i(X_{i-1}, W_i)$.
- Let's assume that we already know $\frac{\partial E}{\partial X_i}$, in other words, for each component of vector X_i we know how much E would wiggle if we wiggled that component of X_i.

Computing the Gradients in Multi-Layer Systems



- We can apply chain rule to compute $\frac{\partial E}{\partial W_i}$ (how much E would wiggle if we wiggled each component of W_i):

$$\frac{\partial E}{\partial W_i} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}$$

$$[1 \times N_w] = [1 \times N_x].[N_x \times N_w]$$

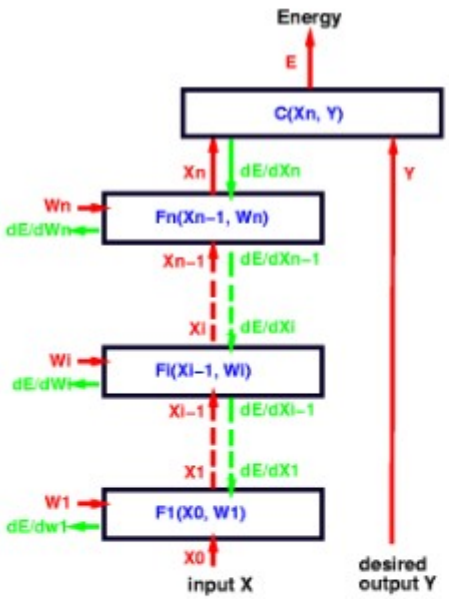
- $\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}$ is the *Jacobian matrix* of F_i with respect to W_i.

$$\left[\frac{\partial F_i(X_{i-1}, W_i)}{\partial W_i}\right]_{kl} = \frac{\partial \left[F_i(X_{i-1}, W_i)\right]_k}{\partial [W_i]_l}$$

- Element (k, l) of the Jacobian indicates how much the k-th output wiggles when we wiggle the l-th weight.

Computing the Gradients in Multi-Layer Systems

Using the same trick, we can compute $\frac{\partial E}{\partial X_{i-1}}$. Let's assume again that we already know $\frac{\partial E}{\partial X_i}$, in other words, for each component of vector X_i we know how much E would wiggle if we wiggled that component of X_i.



- We can apply chain rule to compute $\frac{\partial E}{\partial X_{i-1}}$ (how much E would wiggle if we wiggled each component of X_{i-1}):

$$\frac{\partial E}{\partial X_{i-1}} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}$$

- $\frac{\partial F_i(X_{i-1},W_i)}{\partial X_{i-1}}$ is the *Jacobian matrix* of F_i with respect to X_{i-1}.
- F_i has two Jacobian matrices, because it has to arguments.
- Element (k, l) of this Jacobian indicates how much the k-th output wiggles when we wiggle the l-th input.
- **The equation above is a recurrence equation!**

Jacobians and Dimensions

- derivatives with respect to a column vector are line vectors (dimensions: $[1 \times N_{i-1}] = [1 \times N_i] * [N_i \times N_{i-1}]$)

$$\frac{\partial E}{\partial X_{i-1}} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}$$

- (dimensions: $[1 \times N_{wi}] = [1 \times N_i] * [N_i \times N_{wi}]$):

$$\frac{\partial E}{\partial W_i} = \frac{\partial E}{\partial X_i} \frac{\partial F_i(X_{i-1}, W_i)}{\partial W}$$

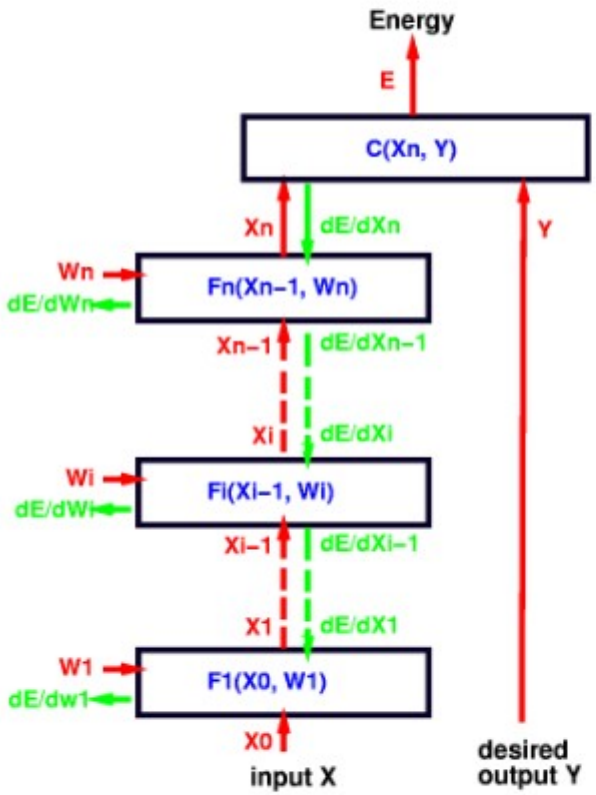
- we may prefer to write those equation with column vectors:

$$\frac{\partial E}{\partial X_{i-1}}' = \frac{\partial F_i(X_{i-1}, W_i)}{\partial X_{i-1}}' \frac{\partial E}{\partial X_i}'$$

$$\frac{\partial E}{\partial W_i}' = \frac{\partial F_i(X_{i-1}, W_i)}{\partial W}' \frac{\partial E}{\partial X_i}'$$

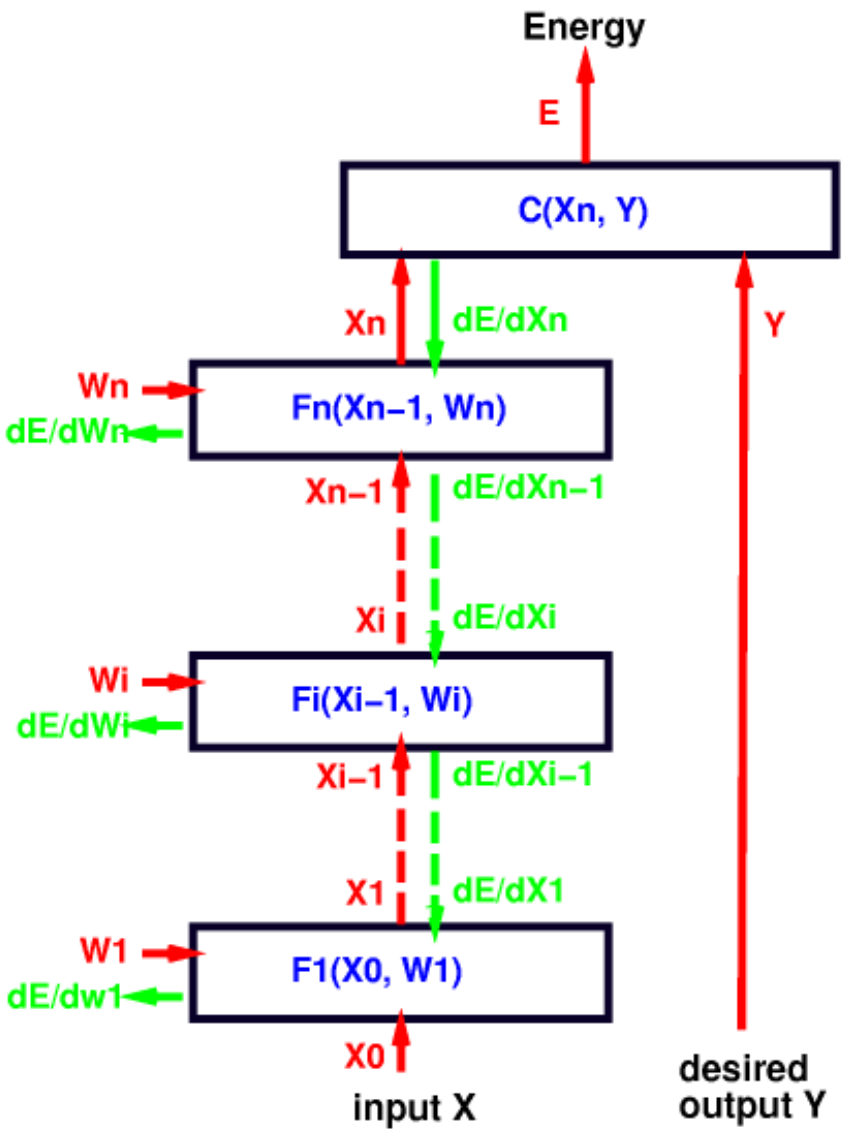
Back-propagation

To compute all the derivatives, we use a backward sweep called the **back-propagation algorithm** that uses the recurrence equation for $\frac{\partial E}{\partial X_i}$



- $\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n, Y)}{\partial X_n}$
- $\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial X_{n-1}}$
- $\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial W_n}$
- $\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial X_{n-2}}$
- $\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial W_{n-1}}$
- ....etc, until we reach the first module.
- we now have all the $\frac{\partial E}{\partial W_i}$ for $i \in [1, n]$.

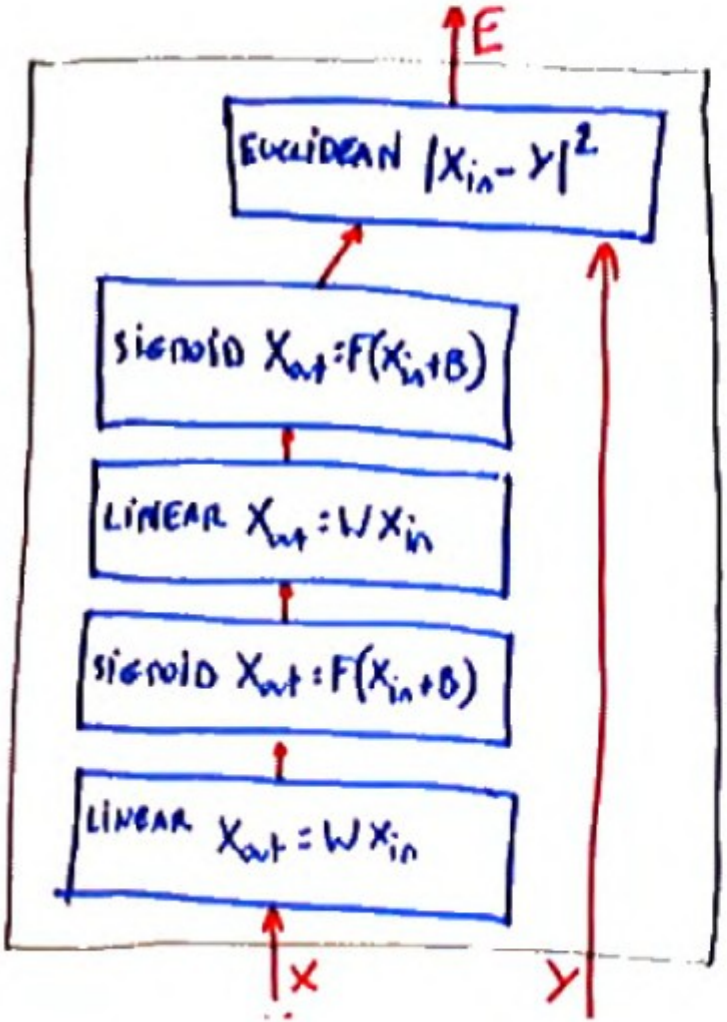
Multimodule Systems: Implementation



- **Backpropagation through a module**
  - Contains trainable parameters
  - Inputs are arguments
  - Gradient with respect to input is returned.
  - Arguments are input and gradient with respect to output
- **Torch7 (by hand)**
  - `hidg = m2:backward(hid,outg)`
  - `ing = m1:backward(in,hidg)`
- **Torch7 (using the nn.Sequential class)**
  - `ing = model:backward(in,outg)`

Modules in a Multi-layer Neural Net

A fully-connected, feed-forward, multi-layer neural nets can be implemented by stacking three types of modules.

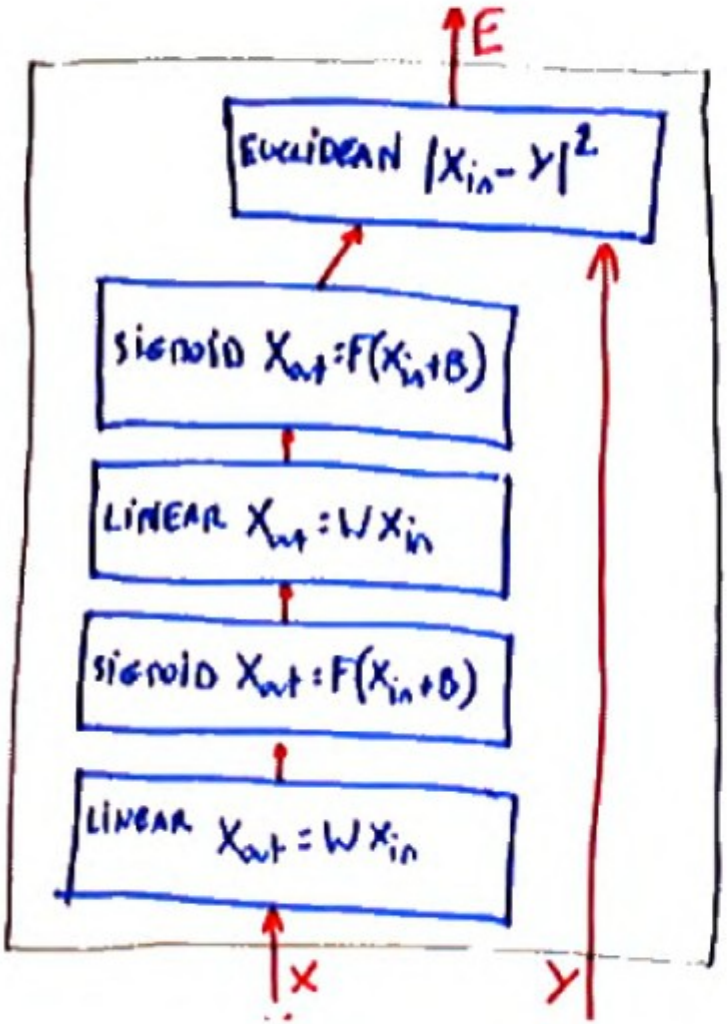


- Linear modules: X_{in} and X_{out} are vectors, and W is a weight matrix.

$$X_{\text{out}} = W X_{\text{in}}$$

- Sigmoid modules: $(X_{\text{out}})_i = \sigma((X_{\text{in}})_i + B_i)$ where B is a vector of trainable "biases", and σ is a sigmoid function such as tanh or the logistic function.
- a Euclidean Distance module $E = \frac{1}{2}||Y - X_{\text{in}}||^2$. With this energy function, we will use the neural network as a regressor rather than a classifier.

Loss Function

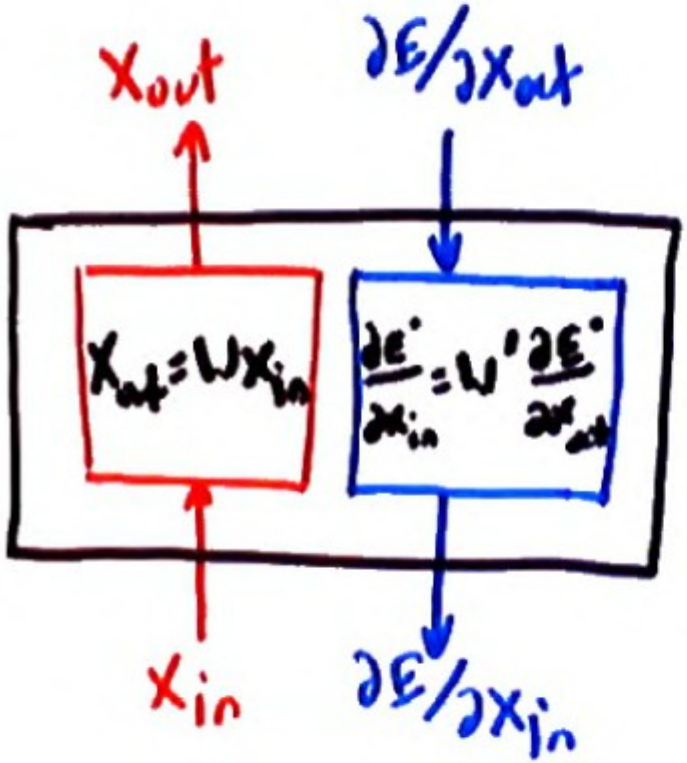


Here, we will us the simple Energy Loss function L_{energy}:

$$L_{\text{energy}}(W, Y^i, X^i) = E(W, Y^i, X^i)$$

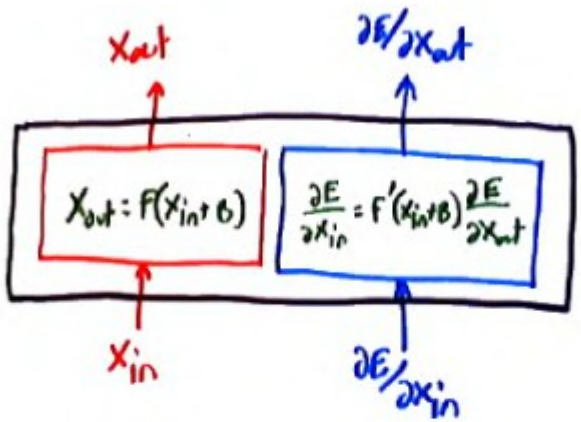
Linear Module

The input vector is multiplied by the weight matrix.



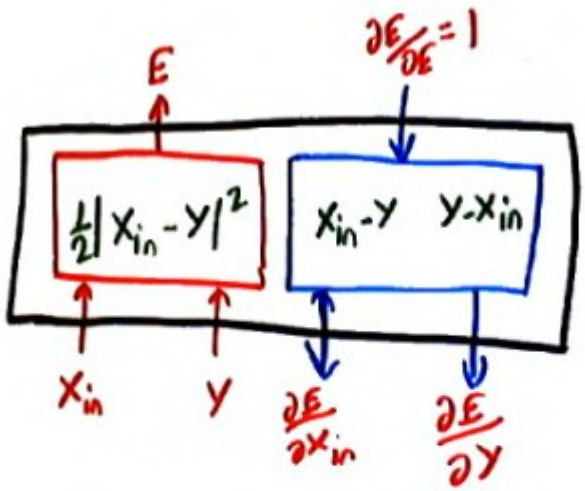
- fprop: $X_{\text{out}} = W X_{\text{in}}$
- bprop to input:
  $$\frac{\partial E}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} \frac{\partial X_{\text{out}}}{\partial X_{\text{in}}} = \frac{\partial E}{\partial X_{\text{out}}} W$$
- by transposing, we get column vectors:
  $$\frac{\partial E}{\partial X_{\text{in}}}' = W' \frac{\partial E}{\partial X_{\text{out}}}'$$
- bprop to weights:
  $$\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial X_{\text{out}i}} \frac{\partial X_{\text{out}i}}{\partial W_{ij}} = X_{\text{in}j} \frac{\partial E}{\partial X_{\text{out}i}}$$
- We can write this as an outer-product:
  $$\frac{\partial E}{\partial W}' = \frac{\partial E}{\partial X_{\text{out}}}' X'_{in}$$

Sigmoid Module (tanh: hyperbolic tangent)



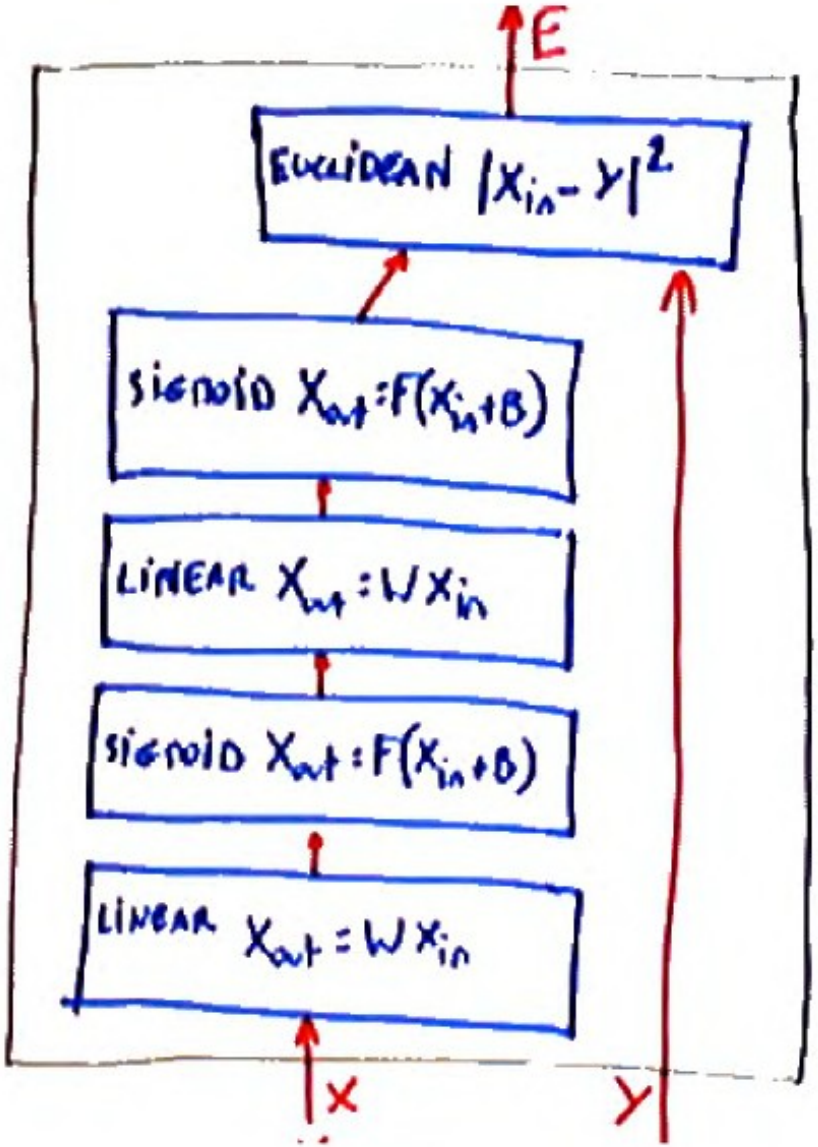
- fprop: $(X_{\text{out}})_i = \tanh((X_{\text{in}})_i + B_i)$
- bprop to input:
  $(\frac{\partial E}{\partial X_{\text{in}}})_i = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'((X_{\text{in}})_i + B_i)$
- bprop to bias:
  $\frac{\partial E}{\partial B_i} = (\frac{\partial E}{\partial X_{\text{out}}})_i \tanh'((X_{\text{in}})_i + B_i)$
- $\tanh(x) = \frac{2}{1+\exp -x} - 1 = \frac{1-\exp(-x)}{1+\exp(-x)}$

Euclidean Module



- fprop: $X_{\text{out}} = \frac{1}{2}||X_{\text{in}} - Y||^2$
- bprop to X input: $\frac{\partial E}{\partial X_{\text{in}}} = X_{\text{in}} - Y$
- bprop to Y input: $\frac{\partial E}{\partial Y} = Y - X_{\text{in}}$

Assembling a simple neural net



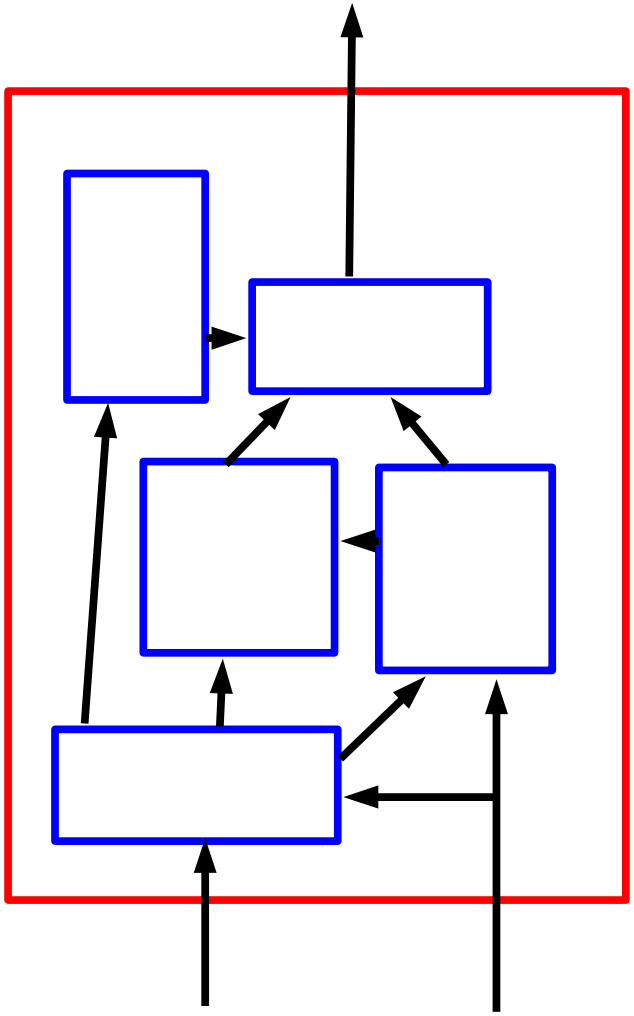

```
model = nn.Sequential()
model:add(nn.Reshape(ninputs))
model:add(nn.Linear(ninputs,nhiddens))
model:add(nn.Tanh())
model:add(nn.Linear(nhiddens,noutputs))
model:add(nn.Tanh())

-- add cost module
criterion = nn.MSECriterion()

-- get the weight and grads in vectors
parameters,gradParameters =
  model:getParameters()
```

Assembling a simple neural net

- We create a function feval(w) that, for the current input, will compute the cost and the gradient of the cost with respect to the parameters.
- This function (actually a closure) can be passed to an optimizer

```
-- create closure to evaluate f(param) and df/dparam
local feval = function(w)
  parameters:copy(w)
  -- reset gradients
  gradParameters:zero()
  local output = model:forward(input)
  local f = criterion:forward(output, target)
  recordstuff(output,target,f)
  -- estimate df/dW
  local df_dout = criterion:backward(output, target)
  model:backward(input, df_dout)
  return f,gradParameters
end
```

Any Architecture works

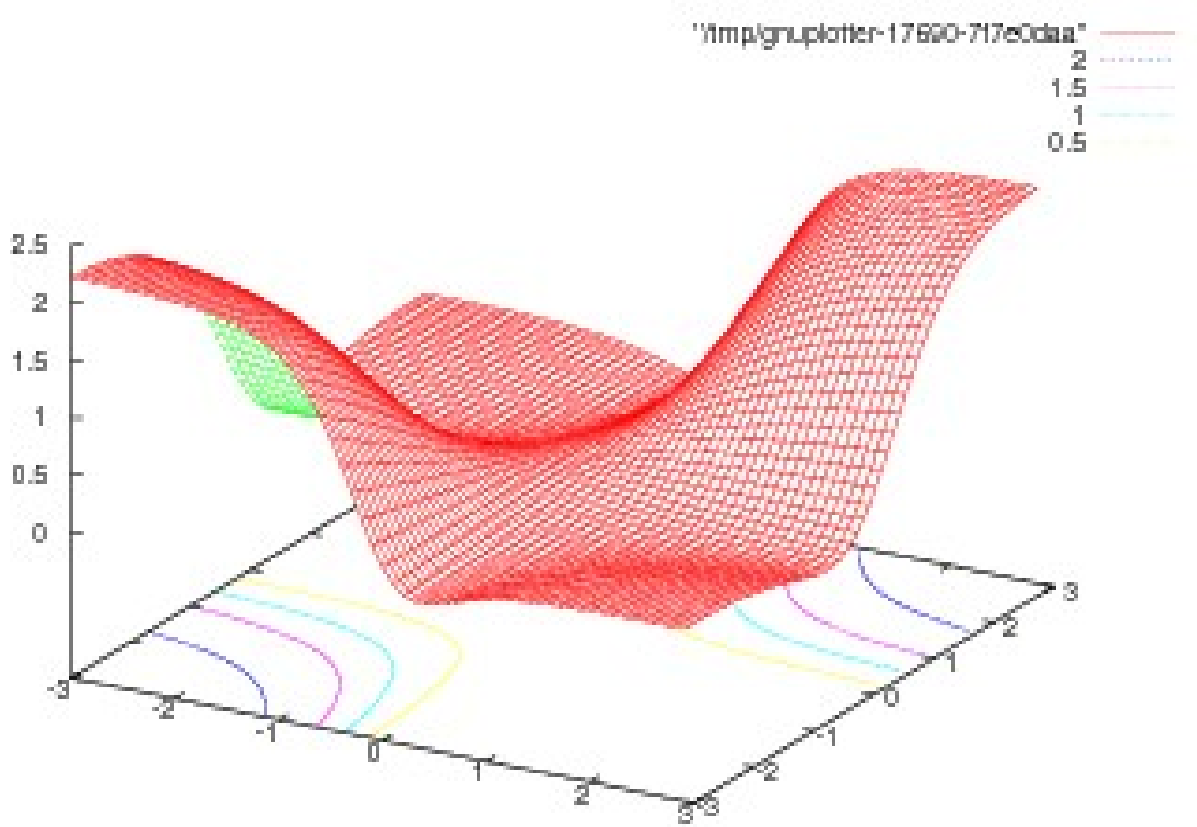


- **Any connection is permissible**
  - Networks with loops must be "unfolded in time".
- **Any module is permissible**
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.

Deep Supervised Learning is Hard

- **Example: what is the loss function for the simplest 2-layer neural net ever**
  - Function: 1-1-1 neural net. Map 0.5 to 0.5 and -0.5 to -0.5 (identity function) with quadratic cost:

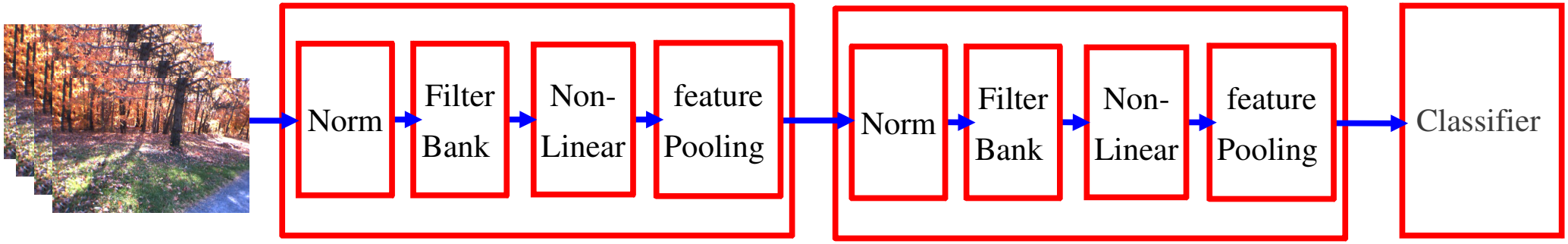
$$y = \tanh(W_1 \tanh(W_0.x)) \quad L = (0.5 - \tanh(W_1 \tanh(W_0 0.5)^2$$



Convolutional Nets

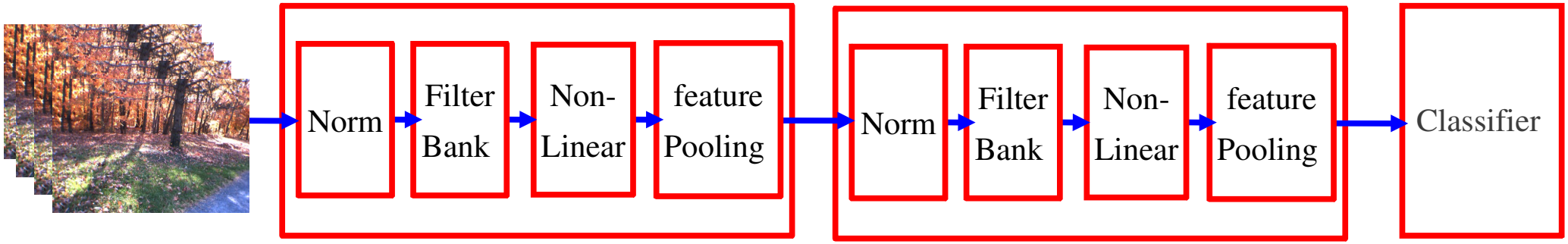
[LeCun, Bottou, Bengio, Haffner 1998]

Feature Transform = Normalization → Filter Bank → Non-Linearity → Pooling



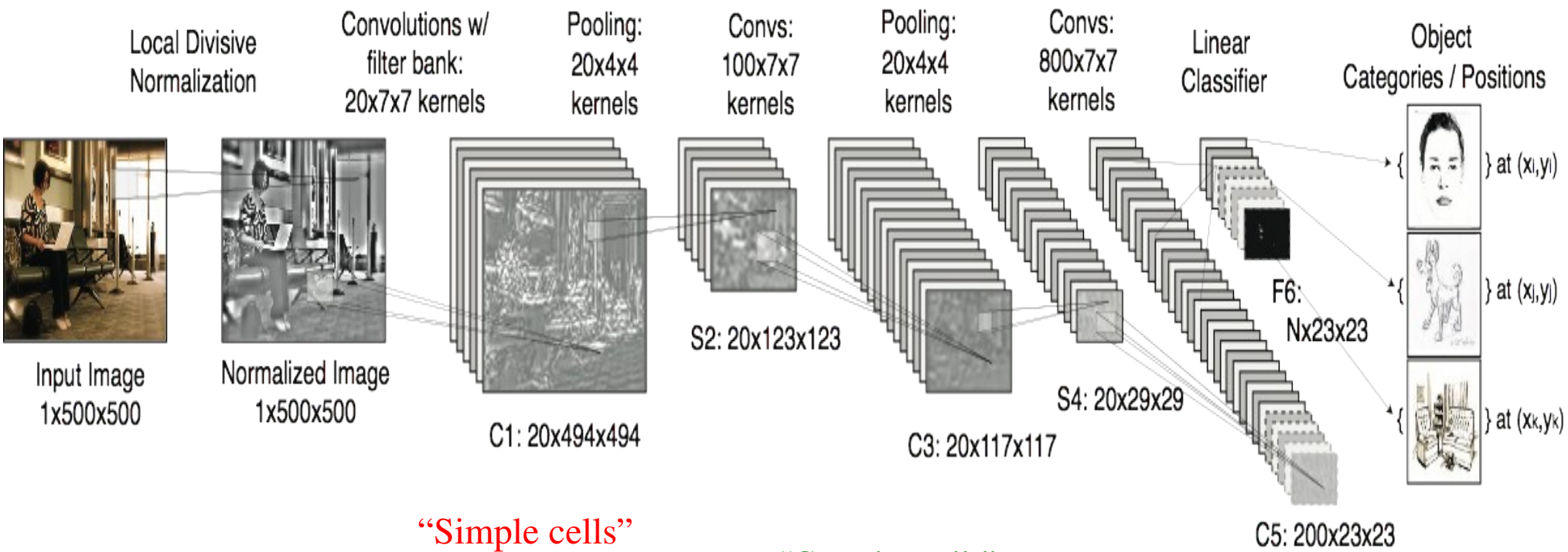
- **Stacking multiple stages of**
  - [Normalization → Filter Bank → Non-Linearity → Pooling].
- **Normalization: variations on whitening**
  - Subtractive: average removal, high pass filtering
  - Divisive: local contrast normalization, variance normalization
- **Filter Bank: dimension expansion, projection on overcomplete basis**
- **Non-Linearity: sparsification, saturation, lateral inhibition....**
  - Component-wise shrinkage or tanh, winner-takes-all
- **Pooling: aggregation over space or feature type, subsampling**
  - $AVERAGE: \frac{1}{K}\sum_i X_i; \quad MAX: \underset{i}{Max}\, X_i; \quad L_p: \sqrt[p]{X_i^p}; \quad PROB: \frac{1}{b}\log\left(\sum_i e^{b X_i}\right)$

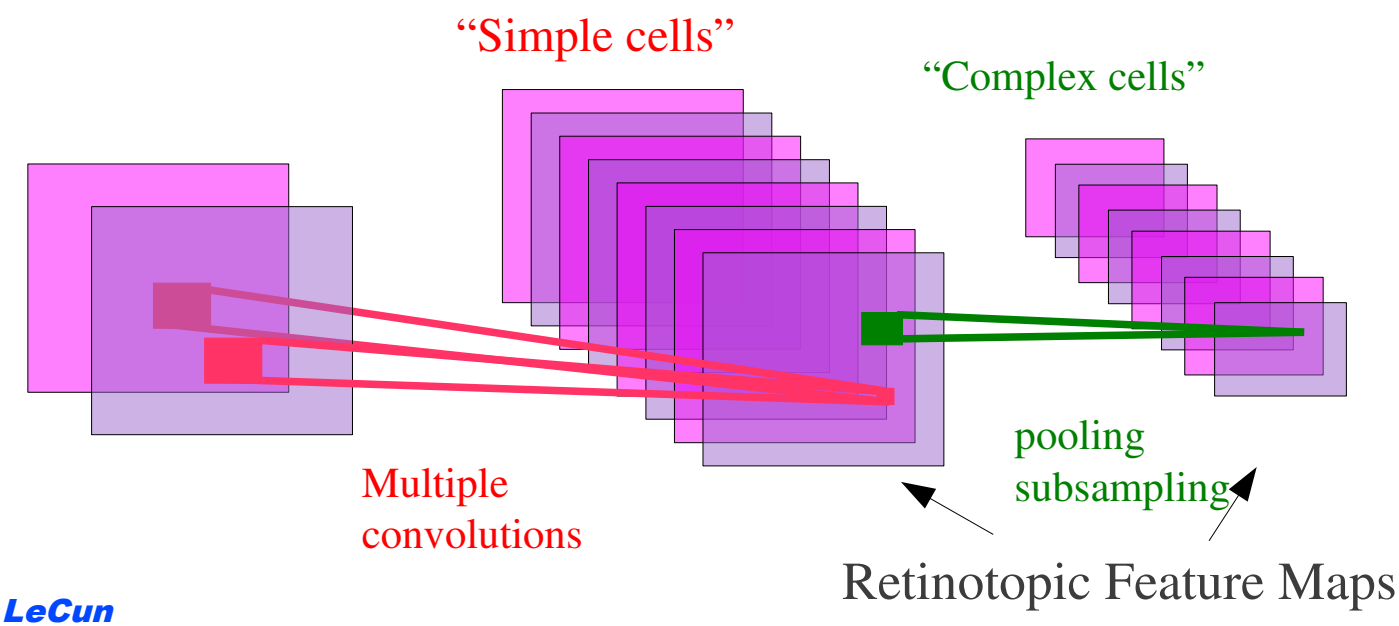
Feature Transform = Normalization → Filter Bank → Non-Linearity → Pooling



- Filter Bank → Non-Linearity = Non-linear embedding in high dimension
- Feature Pooling = contraction, dimensionality reduction, smoothing
- Learning the filter banks at every stage
- Creating a hierarchy of features
- Basic elements are inspired by models of the visual (and auditory) cortex
  - Simple Cell + Complex Cell model of [Hubel and Wiesel 1962]
  - Many "traditional" feature extraction methods are based on this
  - SIFT, GIST, HoG, Convolutional networks.....
- [Fukushima 1974-1982], [LeCun 1988-now], [Poggio 2005-now], [Ng 2006-now], many others....

Basic Convolutional Network Architecture

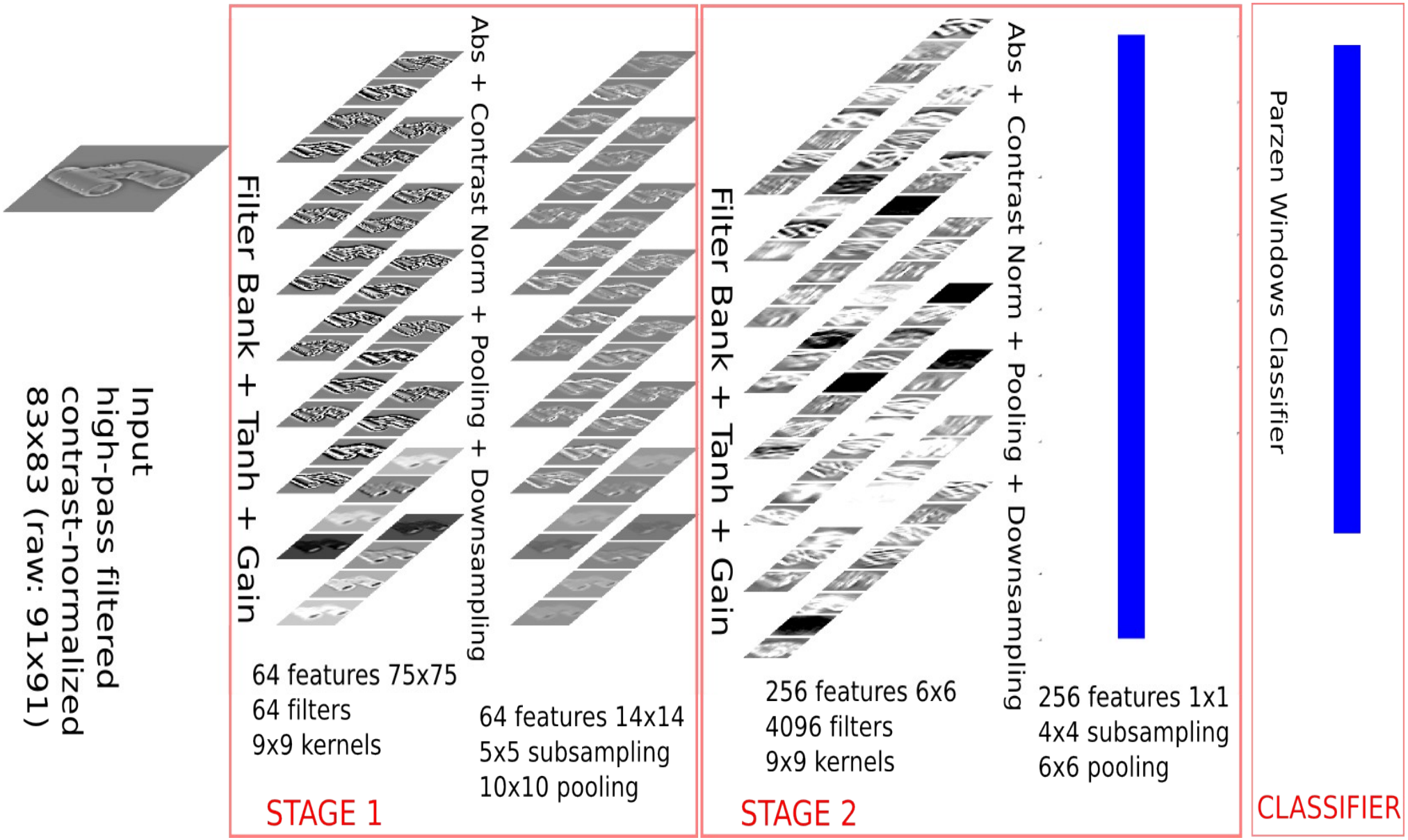




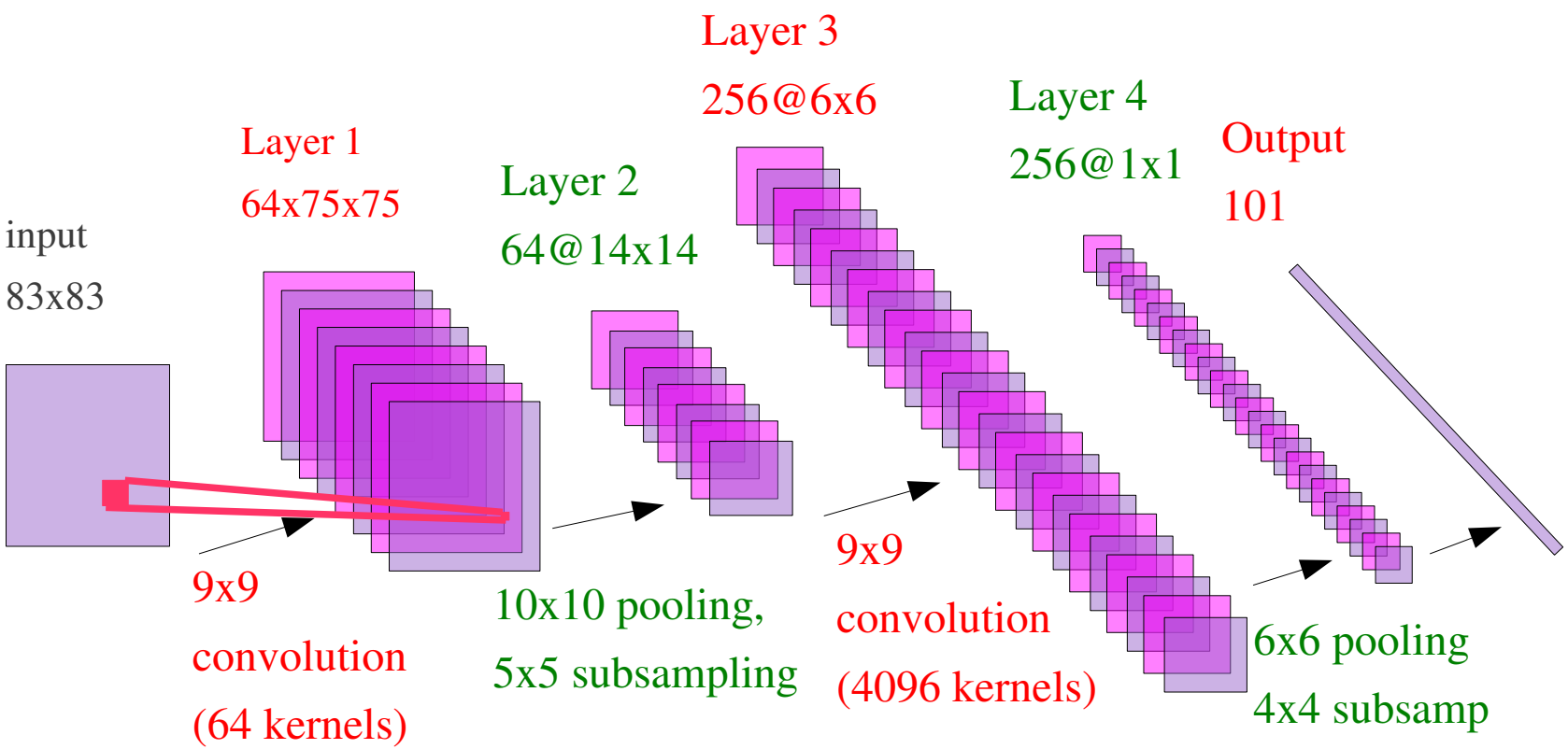
[LeCun et al. 89]

[LeCun et al. 98]

Convolutional Network Architecture



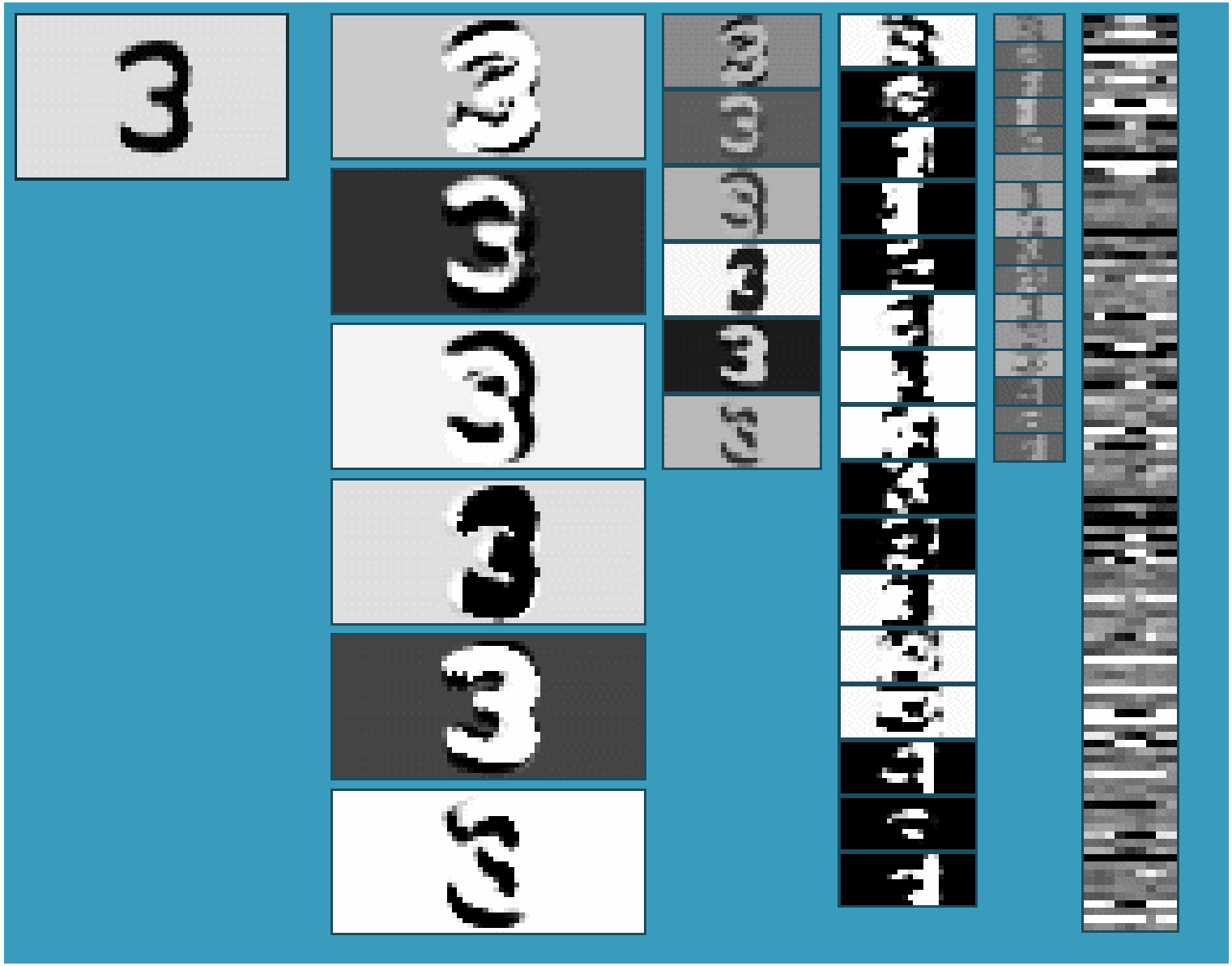
Convolutional Network (ConvNet)



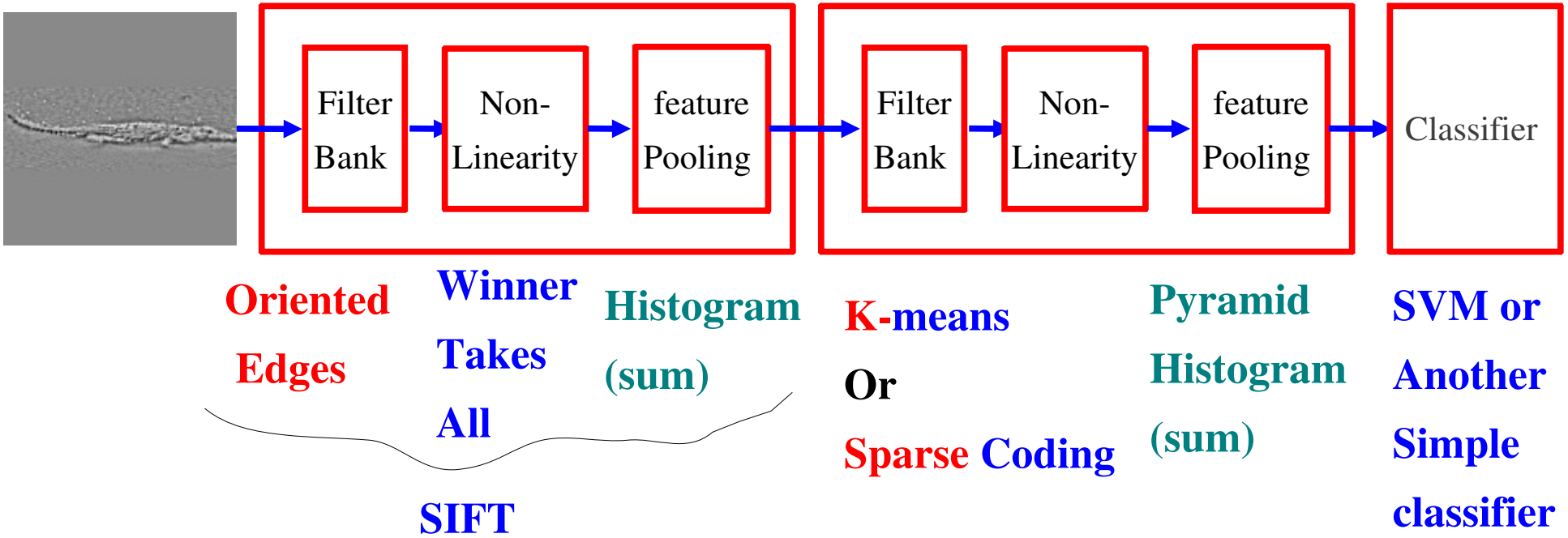
- Non-Linearity: shrinkage function, tanh
- Pooling: L2, average, max, average→tanh
- Training: Supervised (1988-2006), Unsupervised+Supervised (2006-now)

Convolutional Network (vintage 1990)

- filters → tanh → average-tanh → filters → tanh → average-tanh → filters → tanh



"Mainstream" object recognition pipeline 2006-2010: similar to ConvNets



- Fixed low-level Features + unsupervised mid-level features + simple classifier
- Example (on Caltech 101 dataset):
  - SIFT + Vector Quantization + Pyramid pooling + SVM: **>65%**
    - [Lazebnik et al. CVPR 2006]
  - SIFT + Local Sparse Coding Macrofeat. + Pyr/ pooling + SVM: **>77%**
    - [Boureau et al. ICCV 2011]

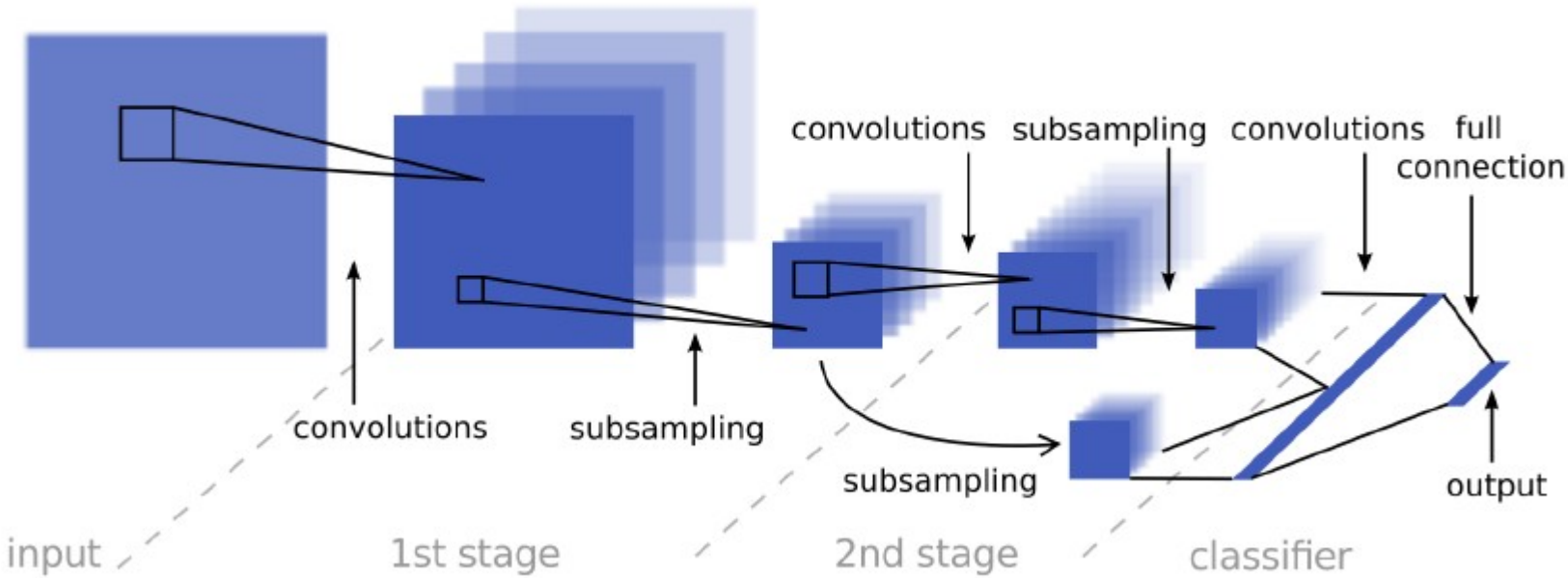
Other Applications with State-of-the-Art Performance

- **Traffic Sign Recognition (GTSRB)**
  - German Traffic Sign Reco Bench
  - 97.2% accuracy

- **House Number Recognition (Google)**
  - Street View House Numbers
  - 94.8% accuracy

ConvNet Architecture with Multi-Stage Features

- Feature maps from all stages are pooled/subsampled and sent to the final classification layers
  - Pooled low-level features: good for textures and local motifs
  - High-level features: good for "gestalt" and global shape

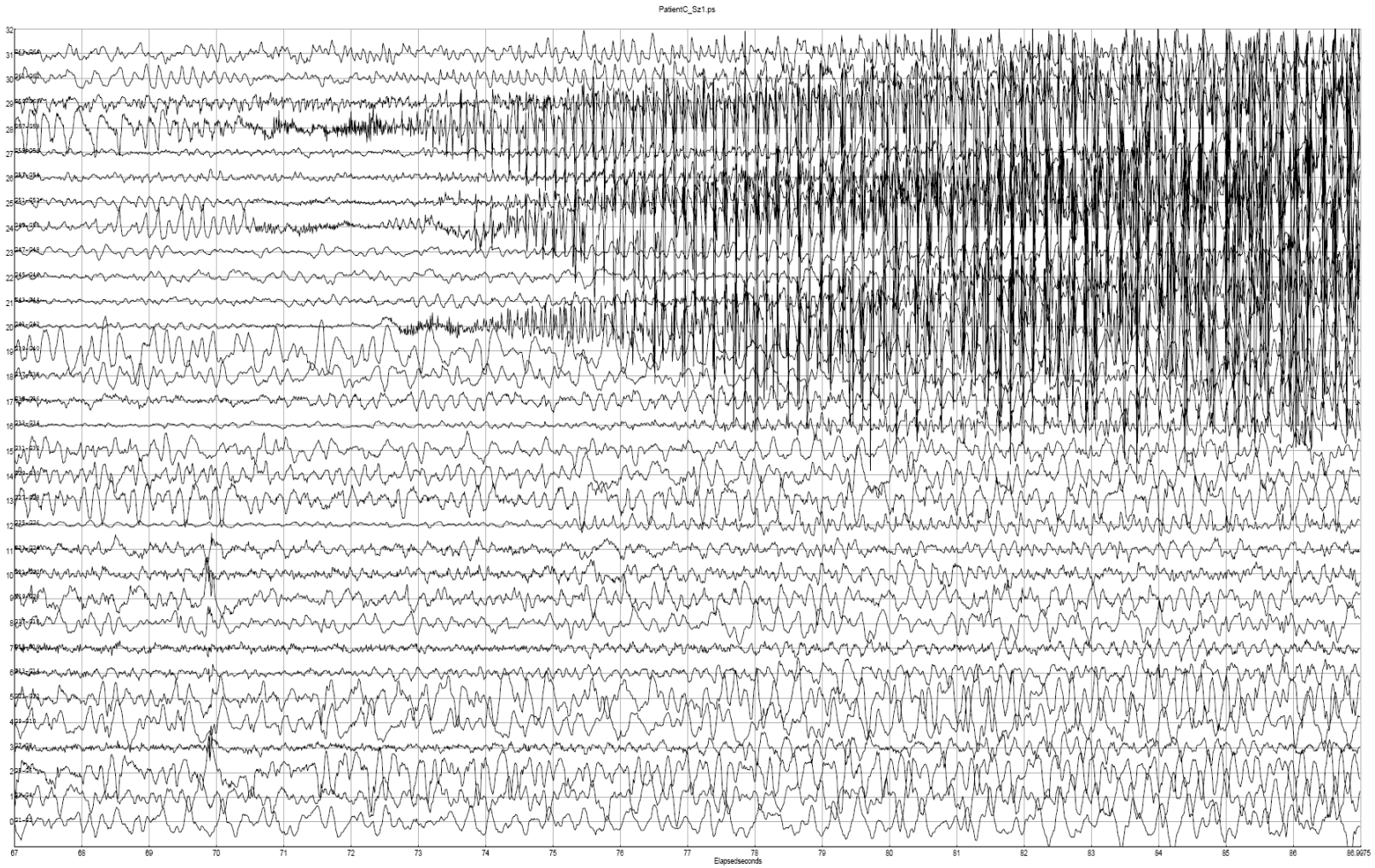


Task	Single-Stage features	Multi-Stage features	Improvement %
Pedestrians detection (INRIA) [9]	14.26%	9.85%	31%
Traffic Signs classification (GTSRB) [11]	1.80%	0.83%	54%
House Numbers classification (SVHN)	5.72%	5.67%	0.9%

[Sermanet, Chintala, LeCun ArXiv:1204.3968, 2012]

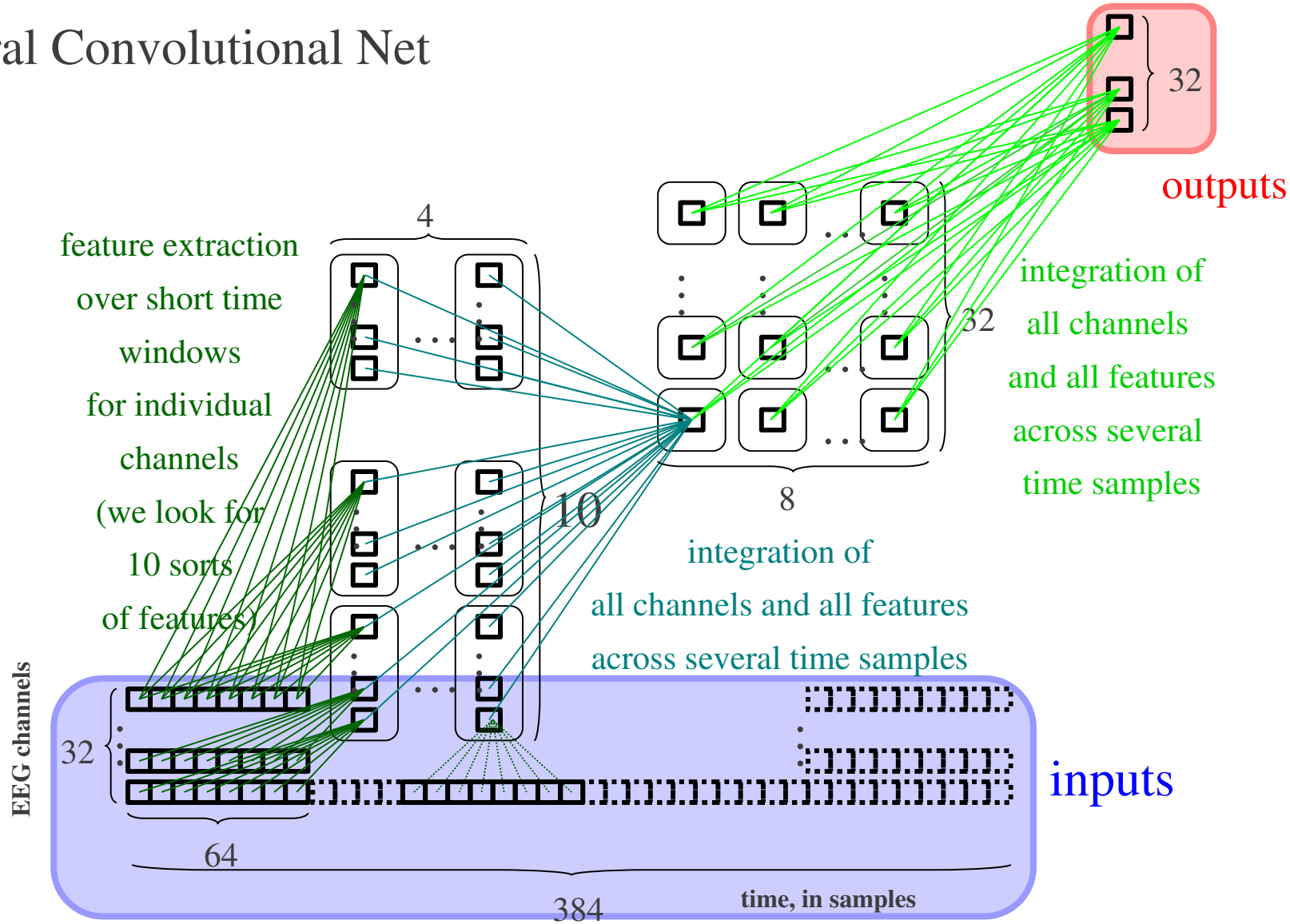
Prediction of Epilepsy Seizures from Intra-Cranial EEG

Piotr Mirowski, Deepak Mahdevan (NYU Neurology), Yann LeCun



Epilepsy Prediction

Temporal Convolutional Net



Industrial Applications of ConvNets

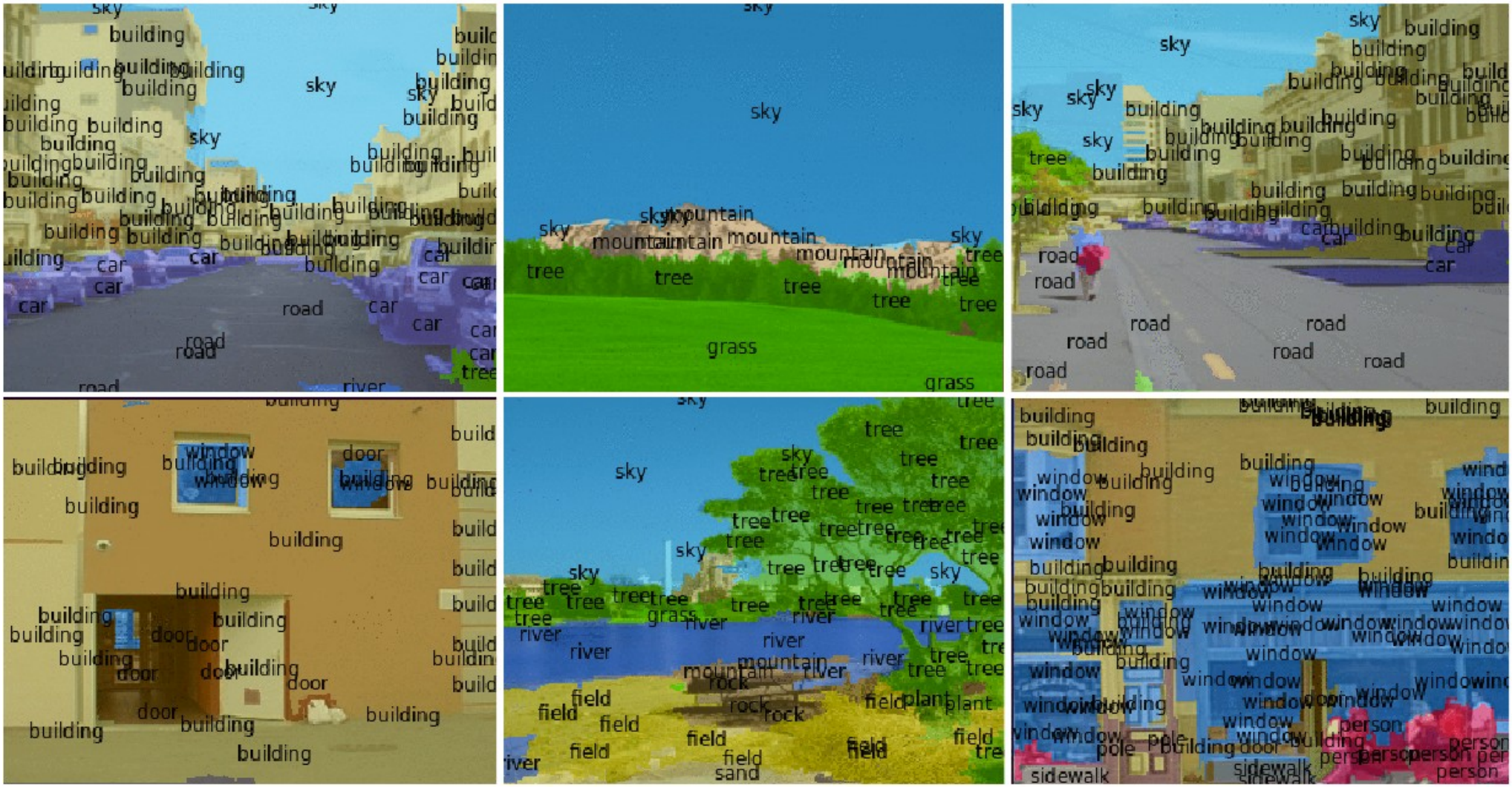
- AT&T/Lucent/NCR
  - Check reading, OCR, handwriting recognition (deployed 1996)
- NEC
  - Intelligent vending machines and advertizing posters, cancer cell detection, automotive applications
- Google
  - Face and license plate removal from StreetView images
- Microsoft
  - Handwriting recognition, speech detection
- Orange
  - Face detection, HCI, cell phone-based applications
- Startups, other companies…

Fast Scene Parsing

[Farabet, Couprie, Najman, LeCun, ICML 2012]

Labeling every pixel with the object it belongs to

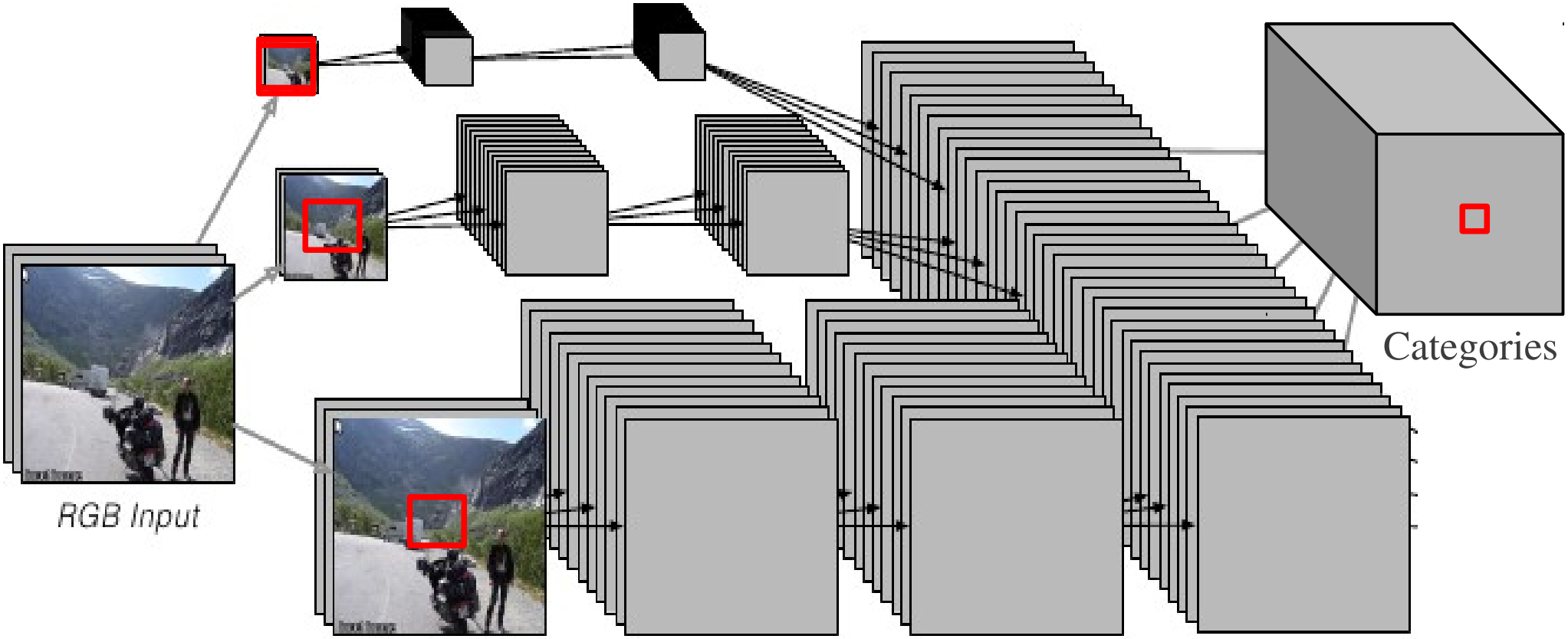
- Would help identify obstacles, targets, landing sites, dangerous areas
- Would help line up depth map with edge maps



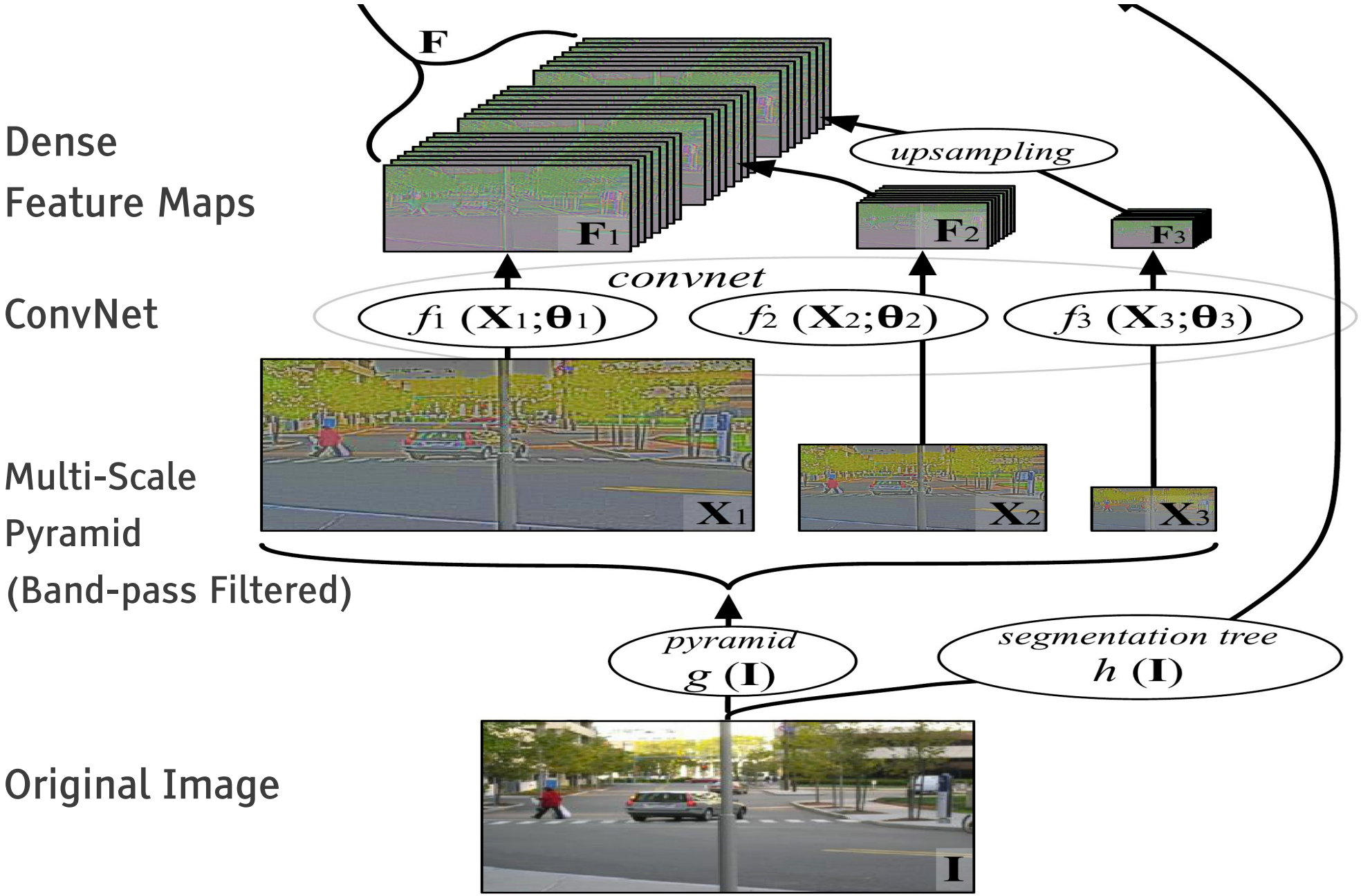
[Farabet et al. ICML 2012]

Scene Parsing/Labeling: ConvNet Architecture

- Each output sees a large input context:
  - **46x46** window at full rez; **92x92** at ½ rez; **184x184** at ¼ rez
  - [7x7conv]->[2x2pool]->[7x7conv]->[2x2pool]->[7x7conv]->
  - Trained supervised on fully-labeled images



Scene Parsing/Labeling: System Architecture



Method 1: majority over super-pixel regions

Input image

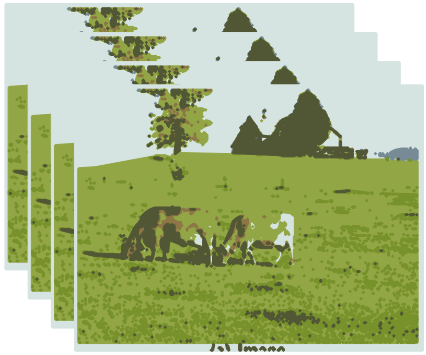
Super-pixel boundary hypotheses

Superpixel boundaries

Majority Vote Over Superpixels

Categories aligned With region boundaries

Multi-scale ConvNet



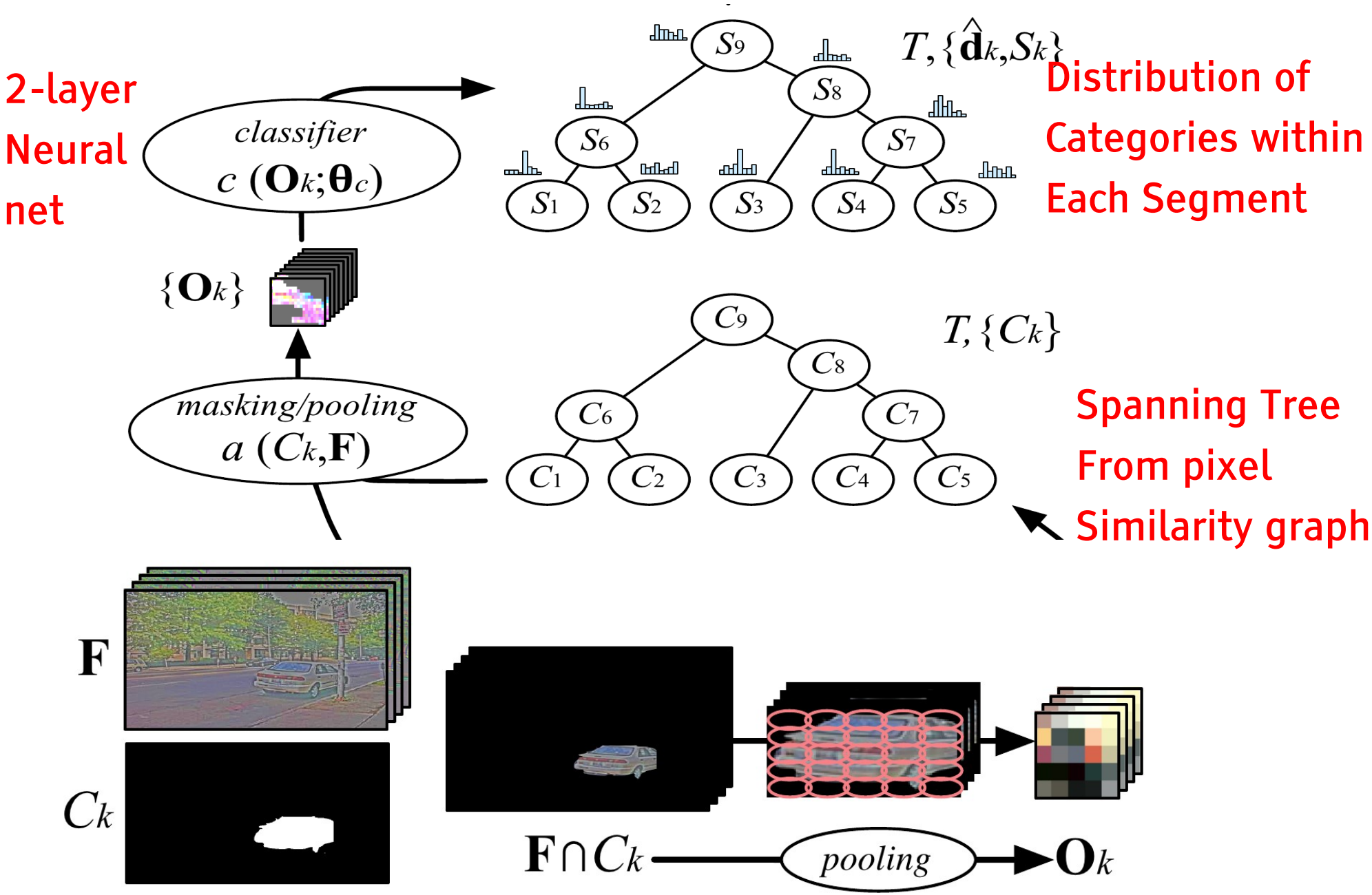
Features from Convolutional net (d=768 per pixel)

Convolutional classifier

"soft" categories scores

[Farabet et al. IEEE T. PAMI 2012]

Method 2: optimal cover of purity tree



[Farabet et al. ICML 2012]

Scene Parsing/Labeling: Performance

- Stanford Background Dataset [Gould 1009]: 8 categories

	Pixel Acc.	Class Acc.	CT (sec.)
Gould *et al.* 2009 [14]	76.4%	-	10 to 600s
Munoz *et al.* 2010 [32]	76.9%	66.2%	12s
Tighe *et al.* 2010 [46]	77.5%	-	10 to 300s
Socher *et al.* 2011 [45]	78.1%	-	?
Kumar *et al.* 2010 [22]	79.4%	-	$< 600s$
Lempitzky *et al.* 2011 [28]	**81.9%**	72.4%	$> 60s$
singlescale convnet	66.0 %	56.5 %	0.35s
multiscale convnet	78.8 %	72.4%	0.6s
multiscale net + superpixels	80.4%	74.56%	**0.7s**
multiscale net + gPb + cover	80.4%	75.24%	61s
multiscale net + CRF on gPb	81.4%	**76.0%**	60.5s

- SIFT Flow dataset [Liu 2009]: 33 categories

	Pixel Acc.	Class Acc.
Liu *et al.* 2009 [29]	74.75%	-
Tighe *et al.* 2010 [46]	76.9%	29.4%
multiscale net + cover[1]	**78.5%**	29.6%
multiscale net + cover[2]	74.2%	**46.0%**

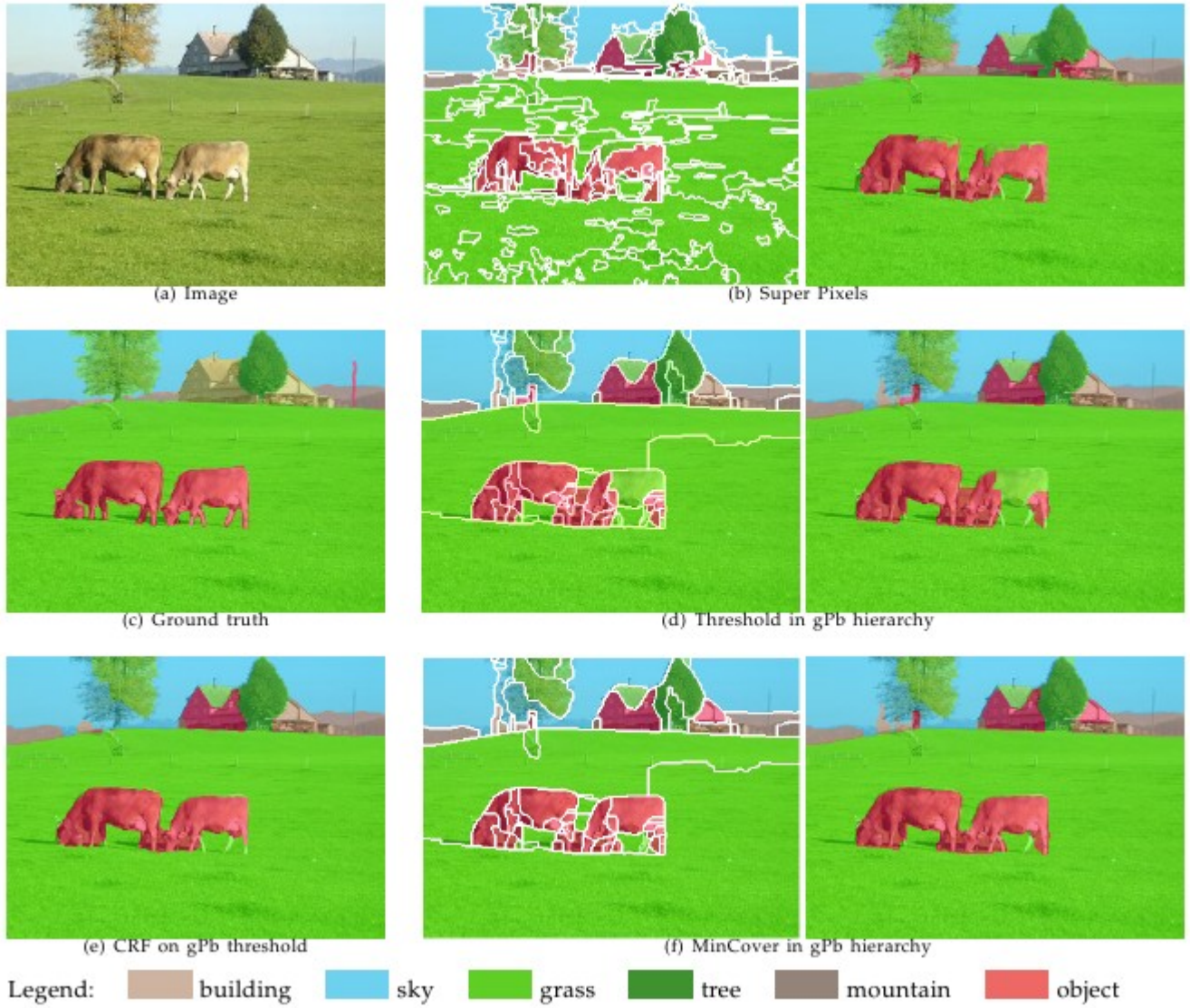
- Barcelona dataset[Tighe 2010]: 170 categories.

	Pixel Acc.	Class Acc.
Tighe *et al.* 2010 [46]	66.9%	7.6%
multiscale net + cover[1]	**67.8%**	%
multiscale net + cover[2]	39.1%	**10.7%**

[Farabet et al. IEEE T. PAMI 2012]

Scene Parsing/Labeling: Results

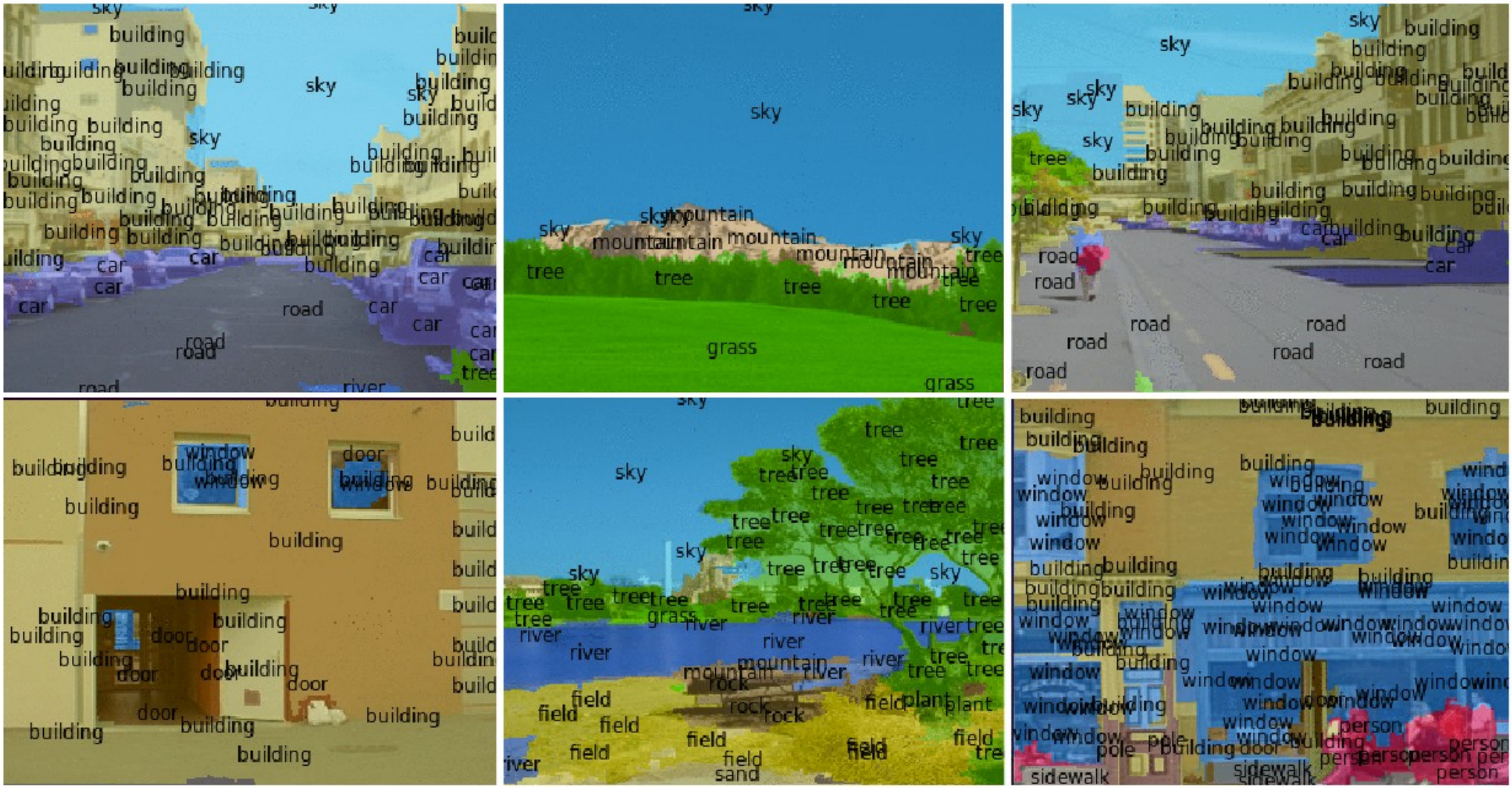
- Samples from the SIFT-Flow dataset (Liu)



[Farabet et al. 2012]

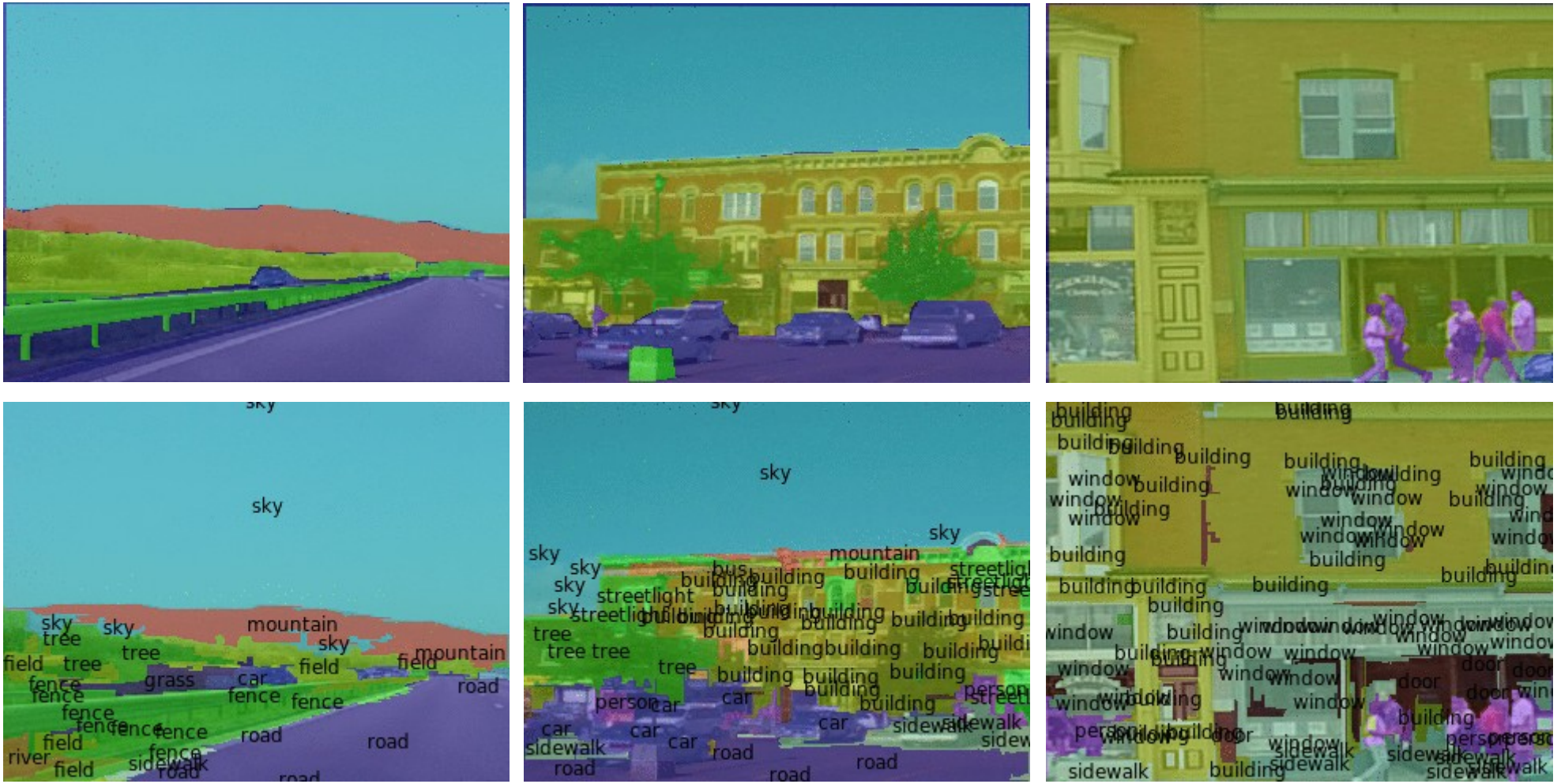
Scene Parsing/Labeling: SIFT Flow dataset (33 categories)

- Samples from the SIFT-Flow dataset (Liu)



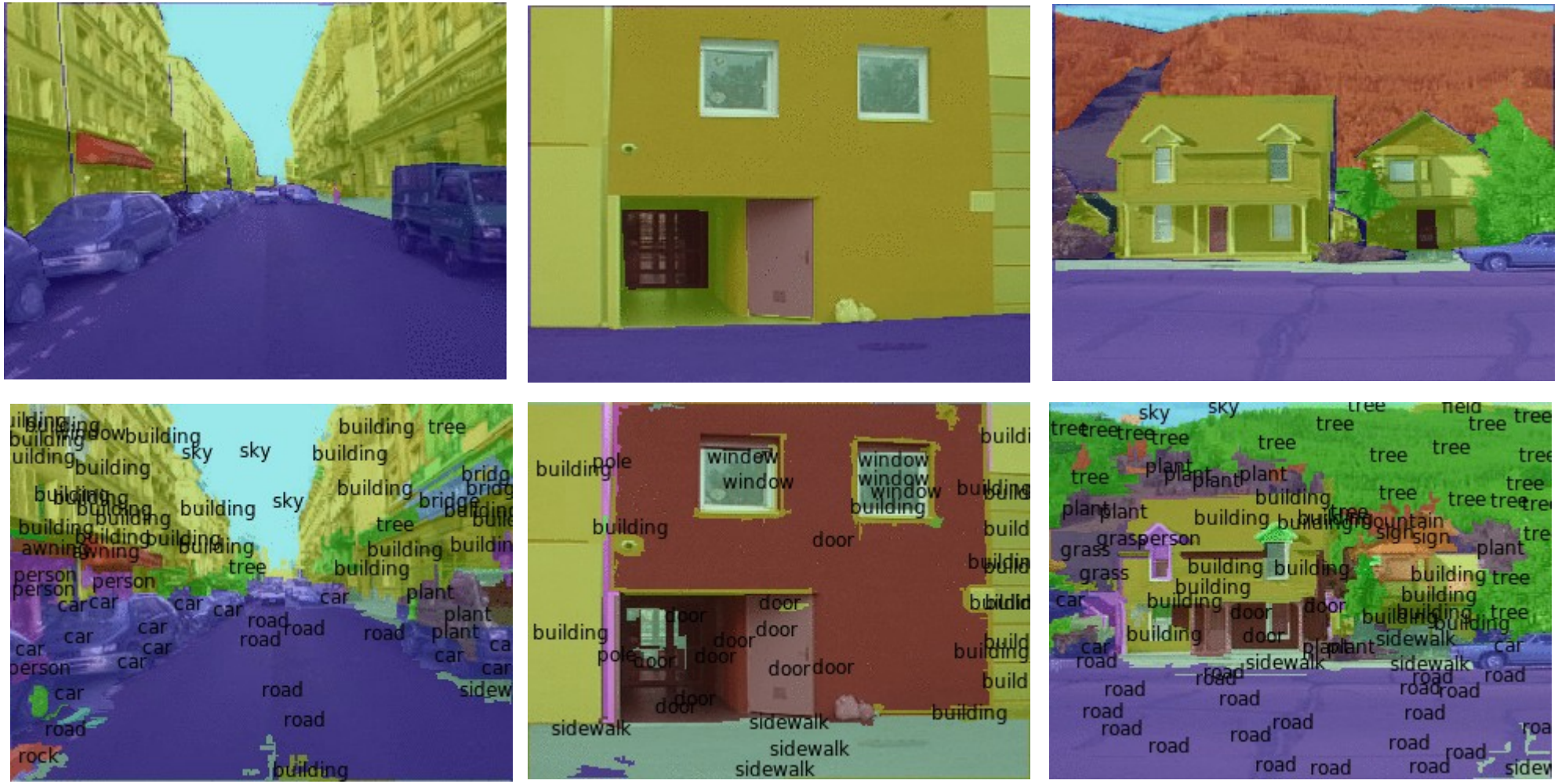
[Farabet et al. ICML 2012]

Scene Parsing/Labeling: SIFT Flow dataset (33 categories)



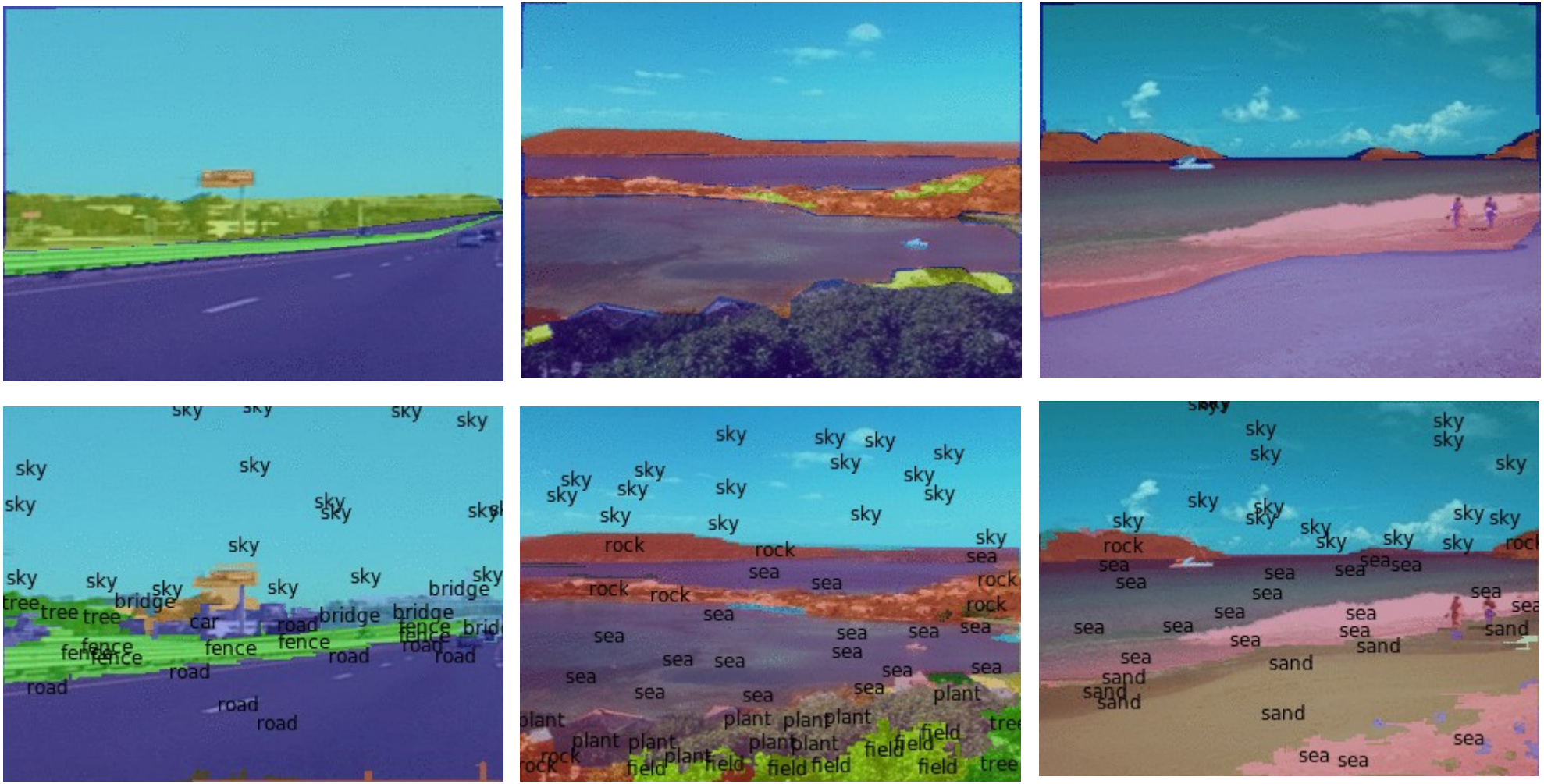
[Farabet et al. ICML 2012]

Scene Parsing/Labeling



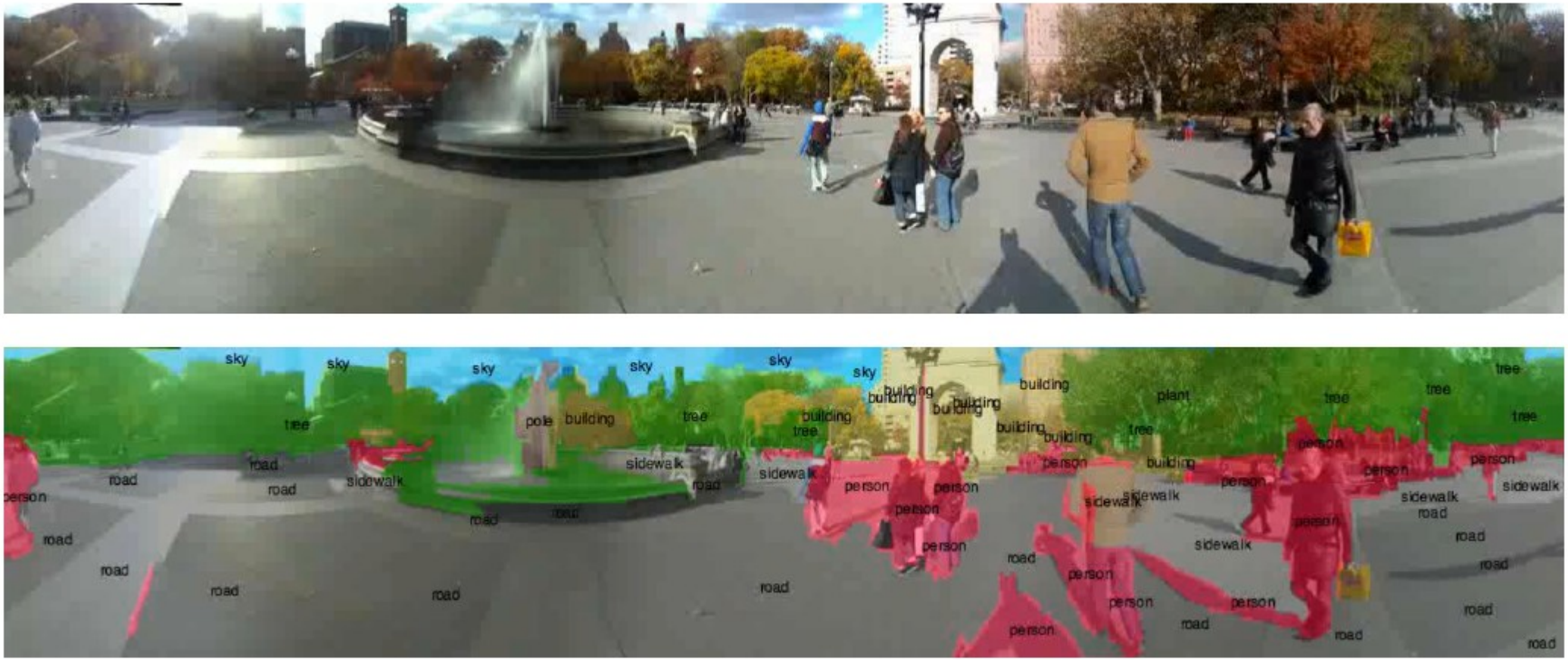
[Farabet et al. ICML 2012]

Scene Parsing/Labeling



[Farabet et al. ICML 2012]

Scene Parsing/Labeling



[Farabet et al. 2012]

Scene Parsing/Labeling

[Farabet et al. 2012]

Scene Parsing/Labeling

- No post-processing
- Frame-by-frame
- ConvNet runs at 50ms/frame on Virtex-6 FPGA hardware
  - But communicating the features over ethernet limits system perf.

Scene Parsing/Labeling: Temporal Consistency

- Majority Vote on Spatio-Temporal Super-Pixels
- Reset every second

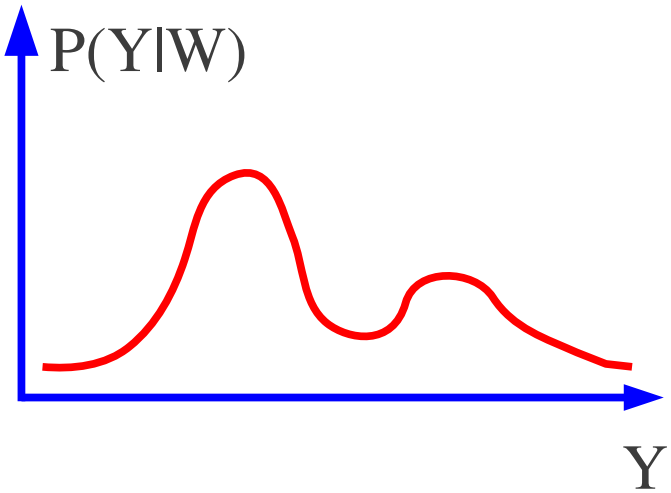
A General View of Unsupervised Learning

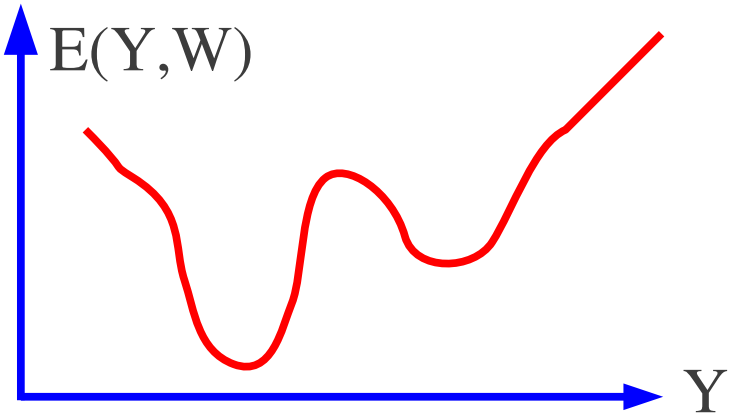
Learning Features with Unsupervised Pre-Training

- Supervised learning requires lots of labeled samples
- Most available data is unlabeled
- Models need to be large to "understand" the task
- But large models have many parameters and require many labeled samples
- Unsupervised learning can be used to pre-train the system before supervised refinement
- Unsupervised pre-training "consumes" degrees of freedom while placing the system in a favorable region of parameter space.
- Supervised refinement merely find the closest local minimum within the attractor found by unsupervised pre-training.
- Unsupervised feature learning through sparse/overcomplete auto-encoders
- With high-dimensional and sparse representations, the data manifold is "flattened" (any collection of points is flatter in higher dimension)

Unsupervised Learning: Capturing Dependencies Between Variables

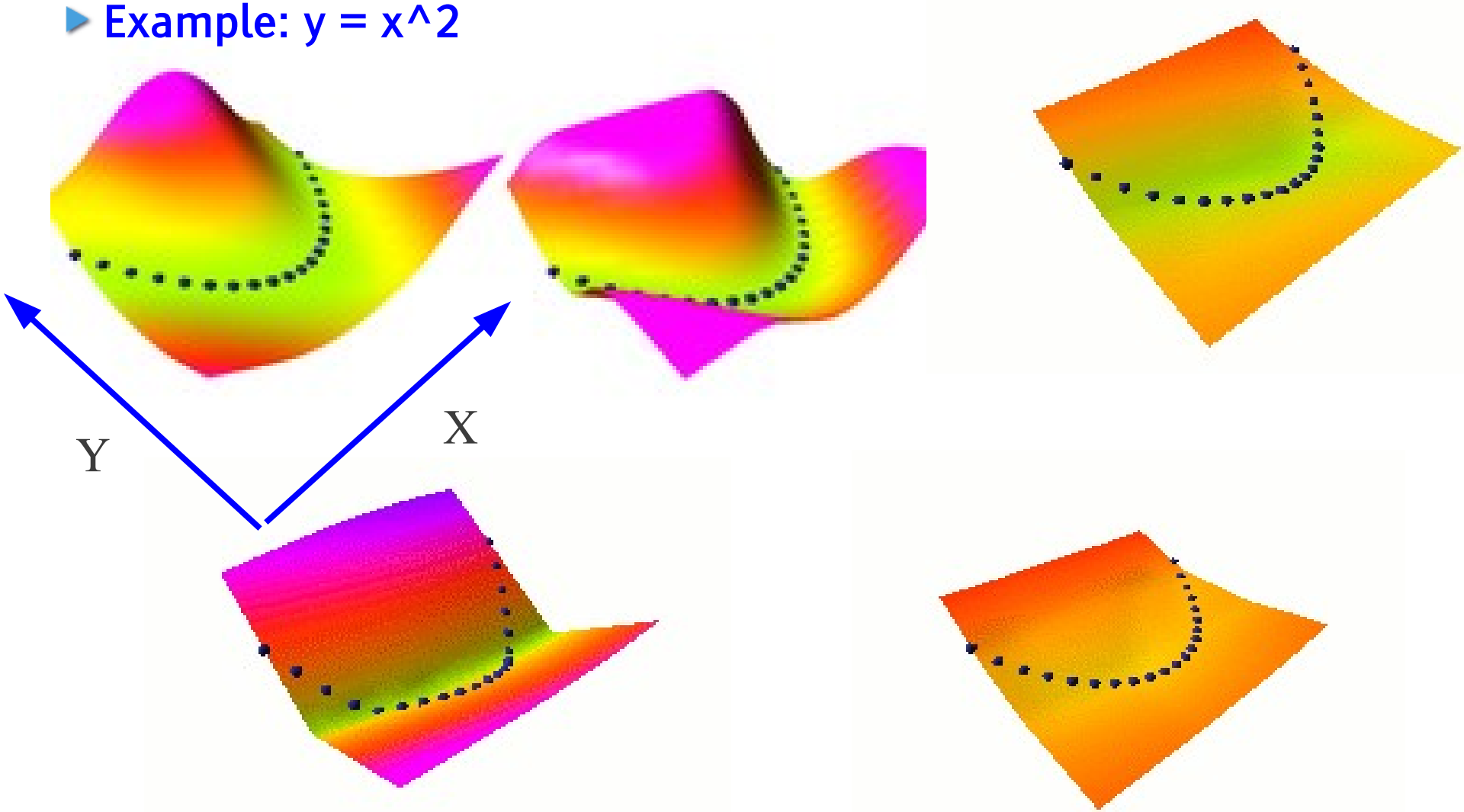
- **Energy function: viewed as a negative log probability density**
- **Probabilistic View:**
  - Produce a probability density function that:
  - has high value in regions of high sample density
  - has low value everywhere else (integral = 1).
- **Energy-Based View:**
  - produce an energy function E(Y,W) that:
  - has low value in regions of high sample density
  - has high(er) value everywhere else





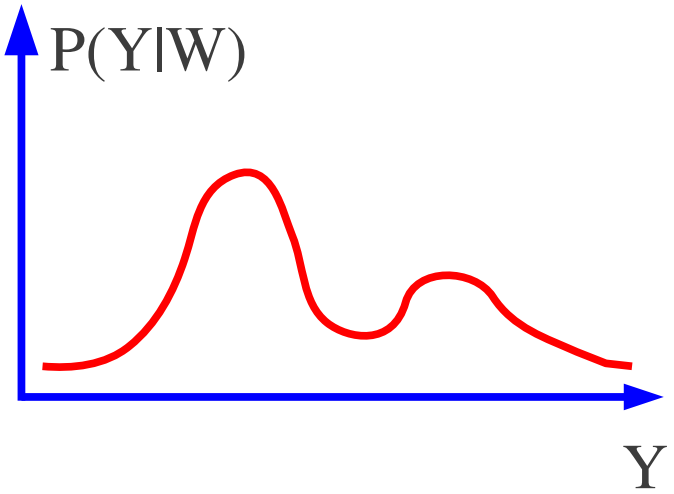
Unsupervised Learning: Capturing Dependencies Between Variables

- Energy function viewed as a negative log density
  - Example: y = x^2

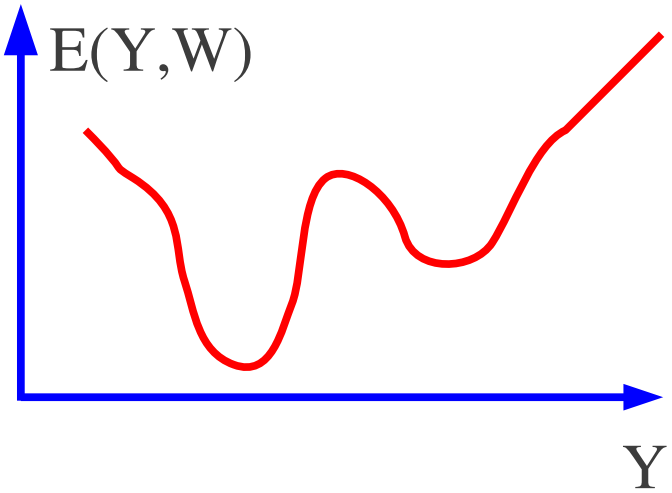


Energy <-> Probability

$$P(Y|W) = \frac{e^{-\beta E(Y,W)}}{\int_y e^{-\beta E(y,W)}}$$

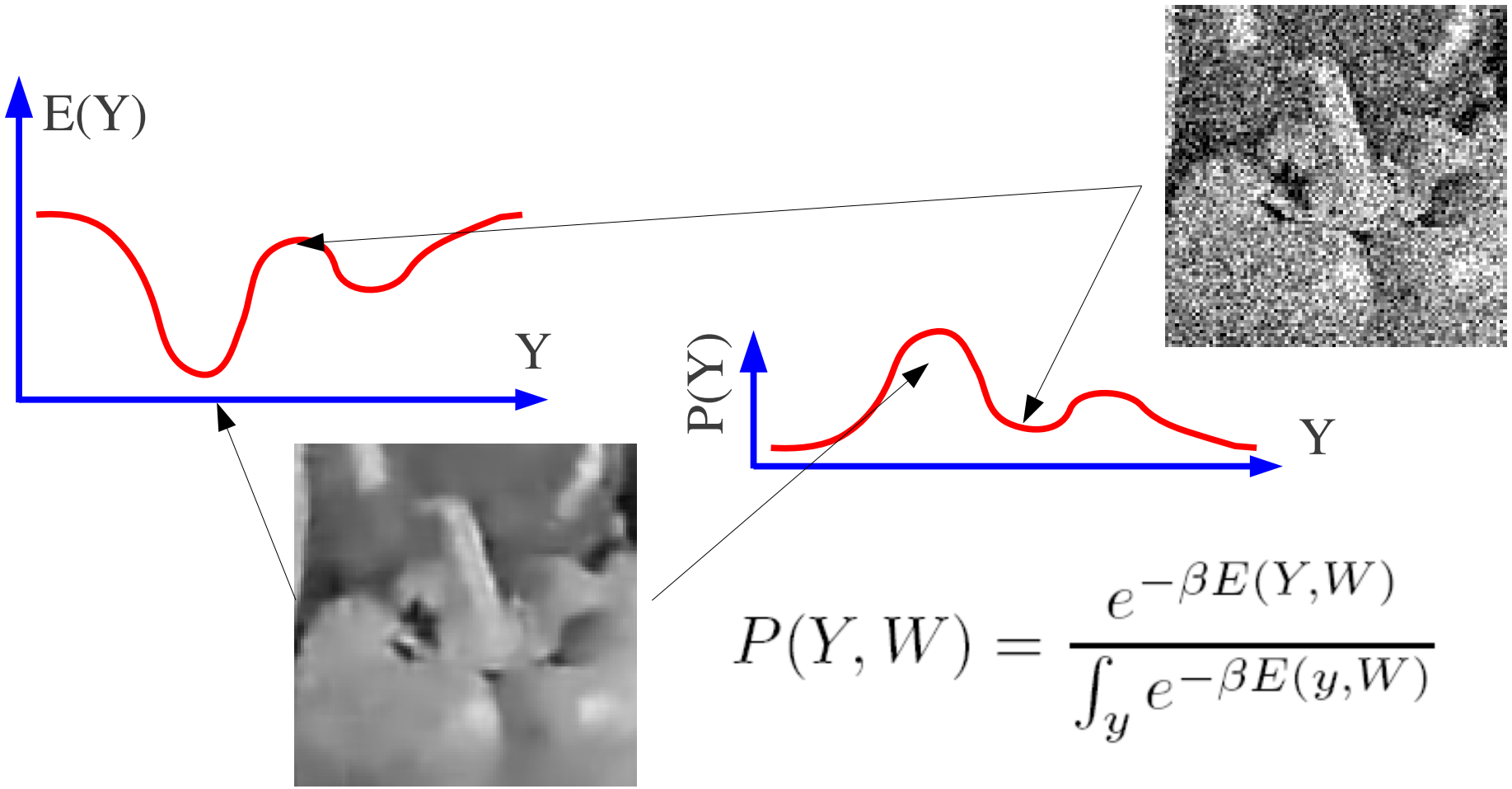


$$E(Y,W) \propto -\log P(Y|W)$$



Training an Energy-Based Model

- Make the energy around training samples low
- Make the energy everywhere else higher



$$P(Y,W) = \frac{e^{-\beta E(Y,W)}}{\int_y e^{-\beta E(y,W)}}$$

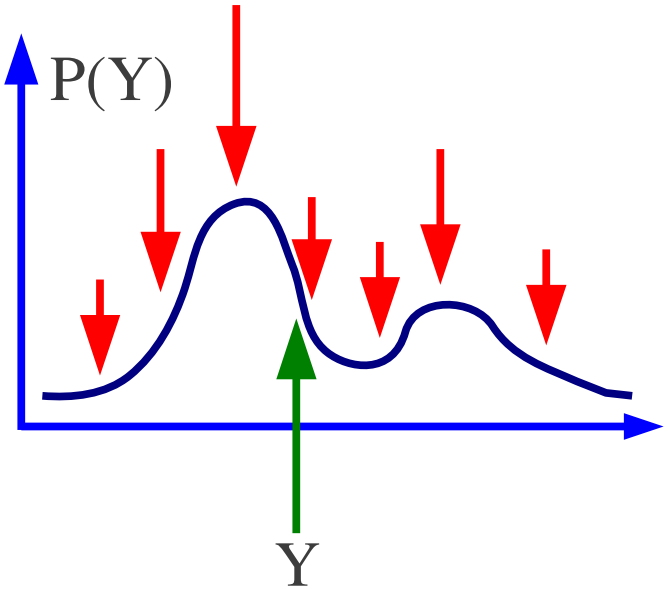
Training an Energy-Based Model to Approximate a Density

Maximizing P(Y|W) on training samples

make this big

$$P(Y|W) = \frac{e^{-\beta E(Y,W)}}{\int_y e^{-\beta E(y,W)}}$$

make this small

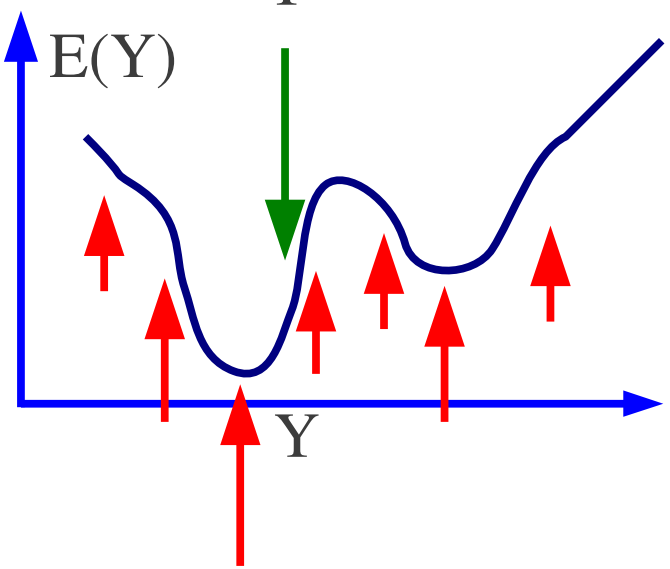


Minimizing -log P(Y,W) on training samples

$$L(Y,W) = E(Y,W) + \frac{1}{\beta} \log \int_y e^{-\beta E(y,W)}$$

make this small

make this big



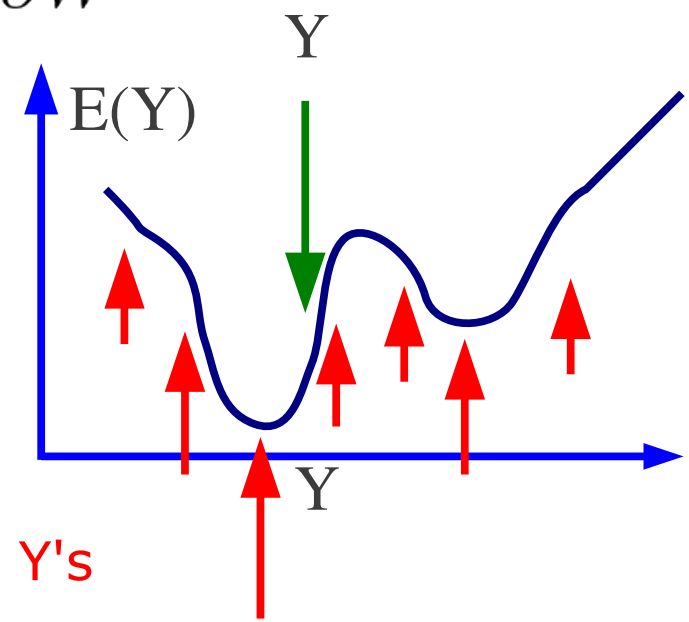
Training an Energy-Based Model with Gradient Descent

- Gradient of the negative log-likelihood loss for one sample Y:

$$\frac{\partial L(Y,W)}{\partial W} = \frac{\partial E(Y,W)}{\partial W} - \int_y P(y|W)\frac{\partial E(y,W)}{\partial W}$$

- Gradient descent:

$$W \leftarrow W - \eta \frac{\partial L(Y,W)}{\partial W}$$



Pushes down on the energy of the samples

Pulls up on the energy of low-energy Y's

$$W \leftarrow W - \eta \frac{\partial E(Y,W)}{\partial W} + \eta \int_y P(y|W)\frac{\partial E(y,W)}{\partial W}$$

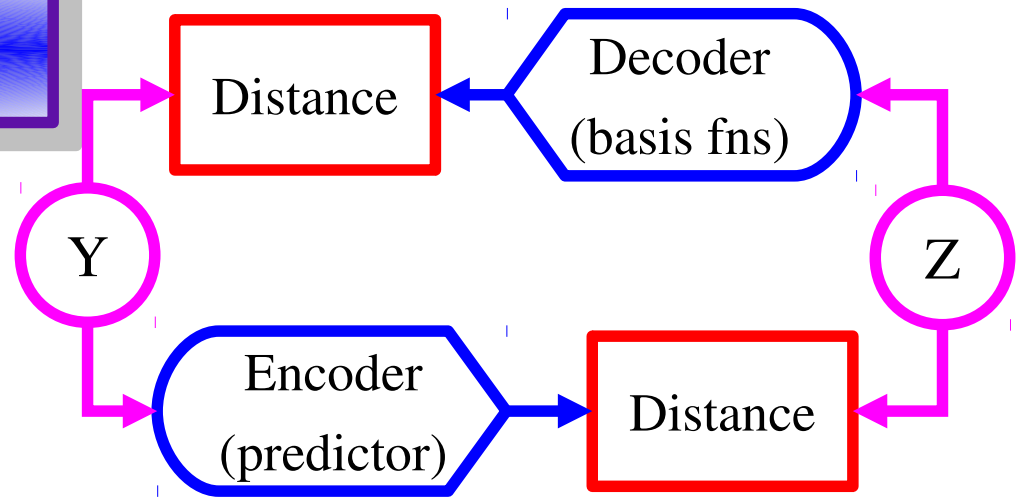
How do we push up on the energy of everything else?

- Solution 1: contrastive divergence [Hinton 2000]
  - Move away from a training sample a bit
  - Push up on that
- Solution 2: score matching
  - On the training samples: minimize the gradient of the energy, and maximize the trace of its Hessian.
- Solution 3: denoising or Contracting auto-encoder (not energy-based)
  - Train the inference dynamics to map noisy samples to clean samples, or regularize the mapping so as not to vary as one moves away from the data manifold.
- Solution 4: MAIN INSIGHT! [Ranzato, ..., LeCun AI-Stat 2007]
  - **Restrict the information content of the code (features) Z**
  - If the code Z can only take a few different configurations, only a correspondingly small number of Ys can be perfectly reconstructed
  - Idea: impose a sparsity prior on Z
  - This is reminiscent of sparse coding [Olshausen & Field 1997]

Restricted Boltzmann Machines

[Hinton & Salakhutdinov 2005]

- Y and Z are binary
- Enc and Dec are linear
- Distance is negative dot product



$$E(Y,Z) = \mathrm{Dist}[Y, \mathrm{Dec}(Z)] + \mathrm{Dist}[Z, \mathrm{Enc}(Y)]$$

$$\mathrm{Enc}(Y) = -W.Y \quad \mathrm{Dist}(Z, W.Y) = -\frac{1}{2} Z^T.W.Y$$

$$\mathrm{Dec}(Y) = -W^T.Z \quad \mathrm{Dist}(Y, E^T.Z) = -\frac{1}{2} Y^T.W^T.Z$$

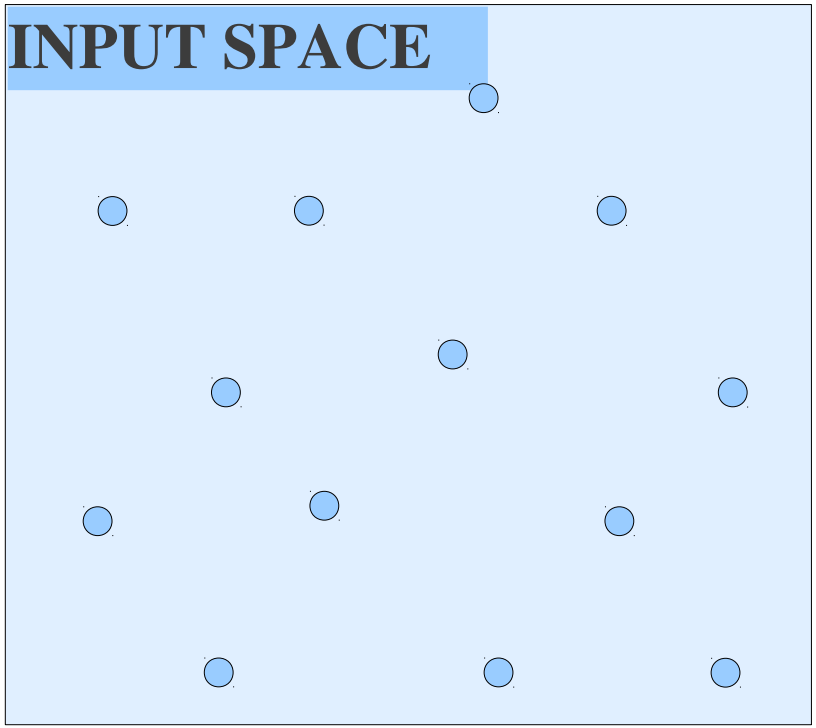
$$E(Y,Z) = -Z^T.W.Y \qquad F(Y) = -\log \sum_z e^{Z^T.W.Y}$$

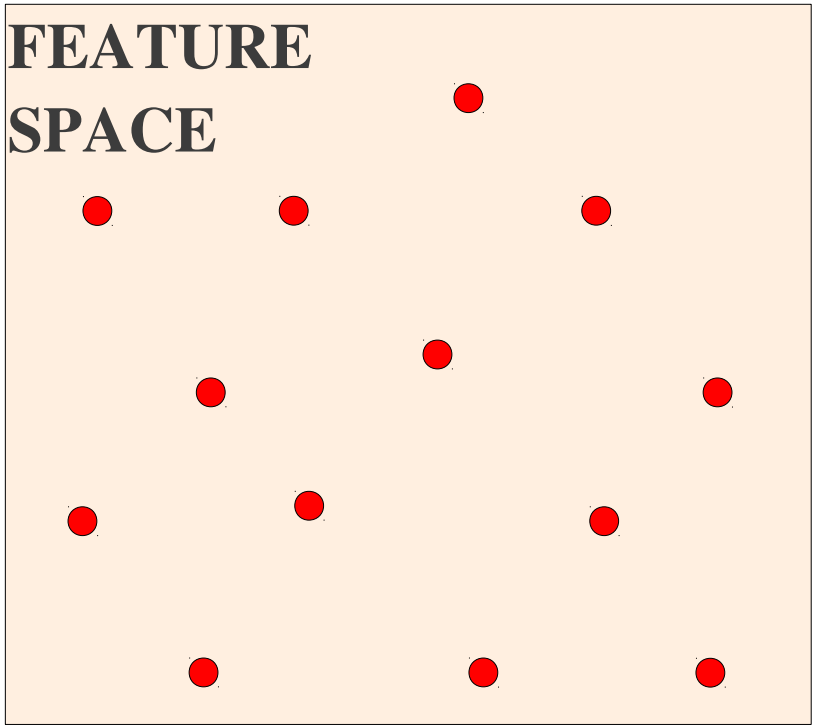
The Main Insight [Ranzato et al. AISTATS 2007]

- **If the information content of the feature vector is limited (e.g. by imposing sparsity constraints), the energy MUST be large in most of the space.**
  - pulling down on the energy of the training samples will necessarily make a groove
- **The volume of the space over which the energy is low is limited by the entropy of the feature vector**
  - Input vectors are reconstructed from feature vectors.
  - If few feature configurations are possible, few input vectors can be reconstructed properly

Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

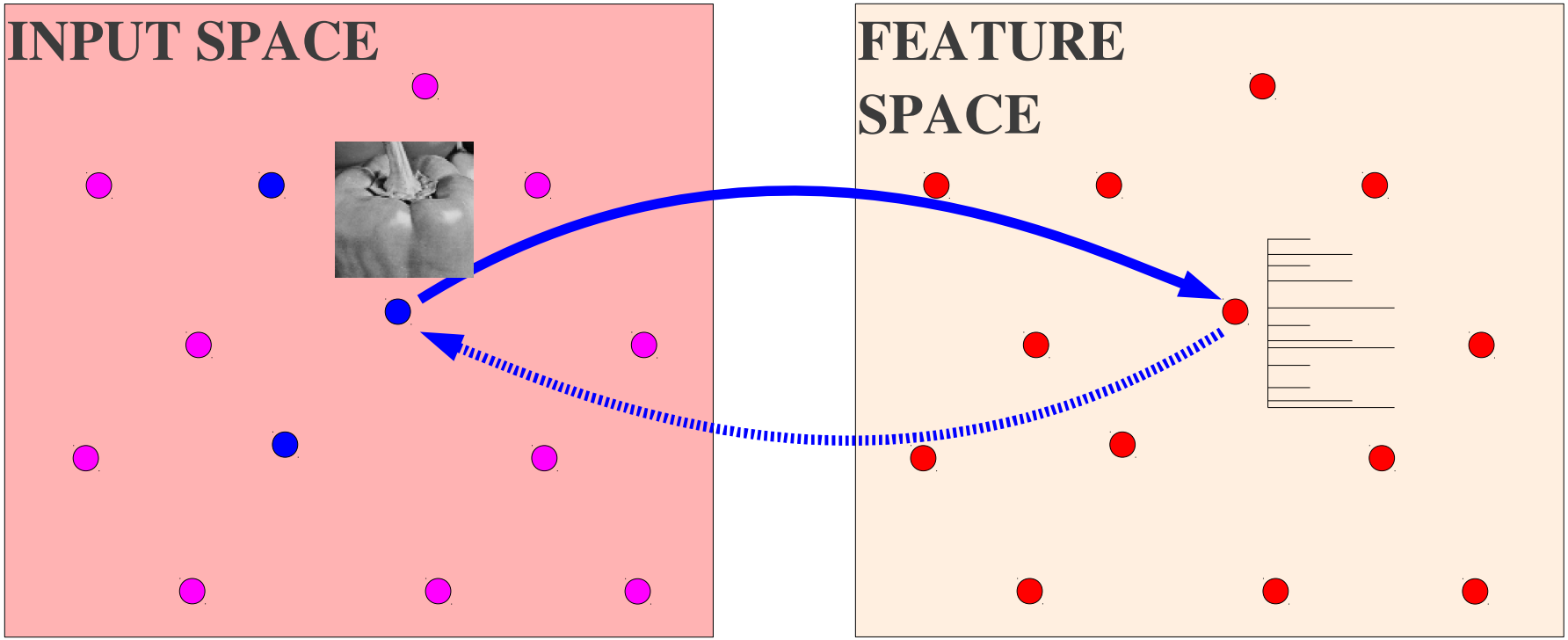




Why Limit the Information Content of the Code?

- **Training sample**
- **Input vector which is NOT a training sample**
- **Feature vector**

Training based on minimizing the reconstruction error over the training set

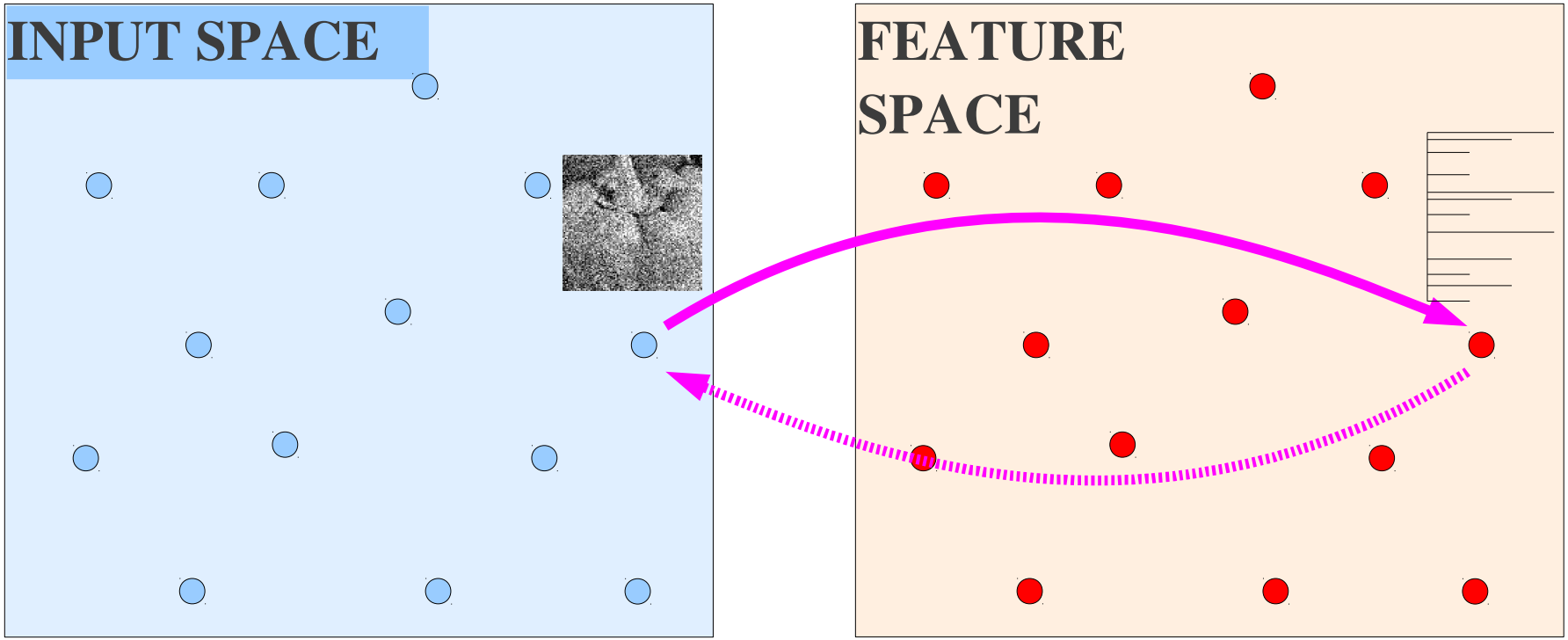


Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

BAD: machine does not learn structure from training data!!

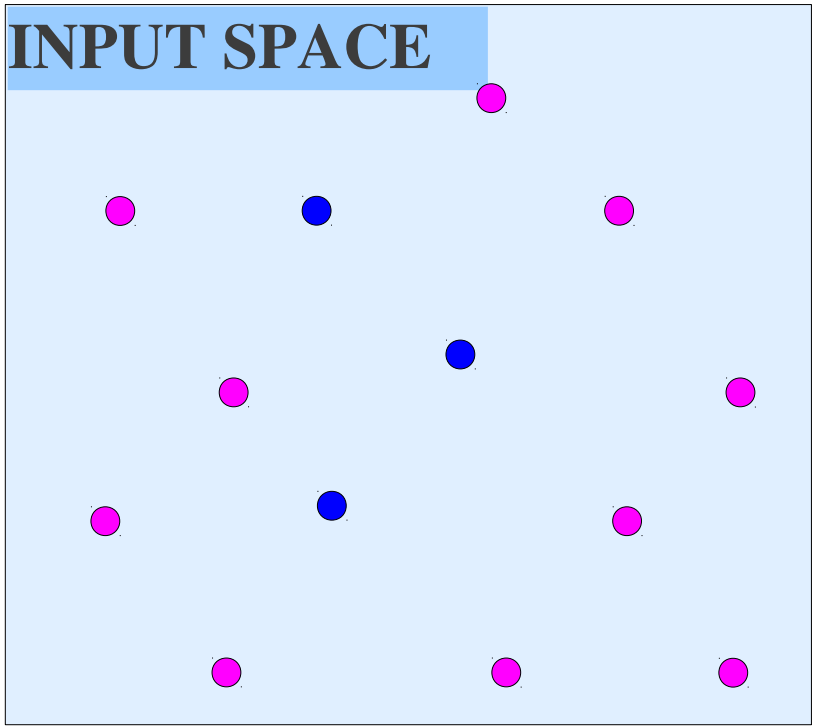
It just copies the data.

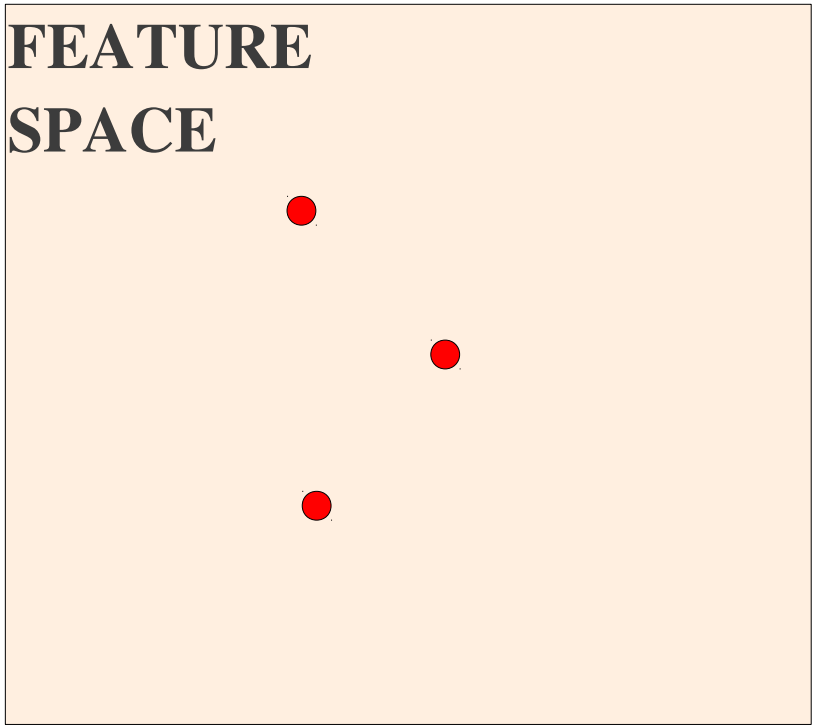


Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

IDEA: reduce number of available codes.

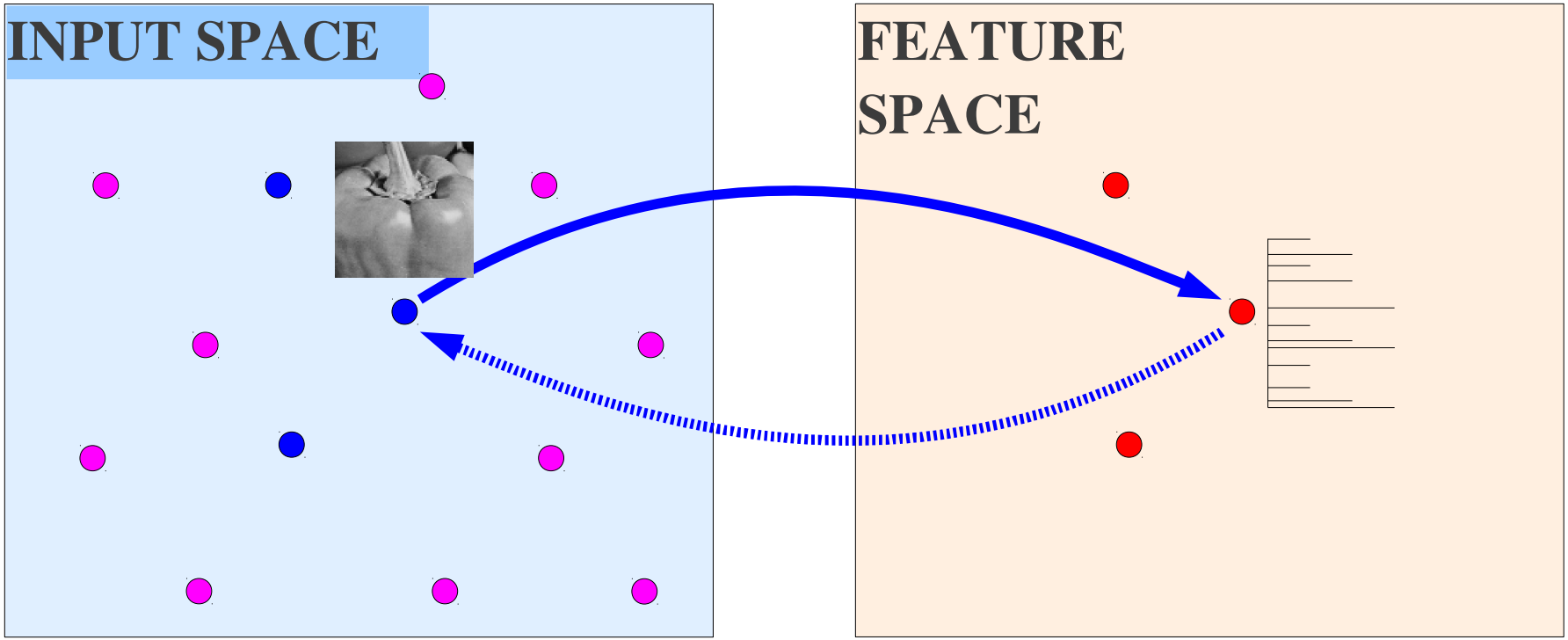




Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

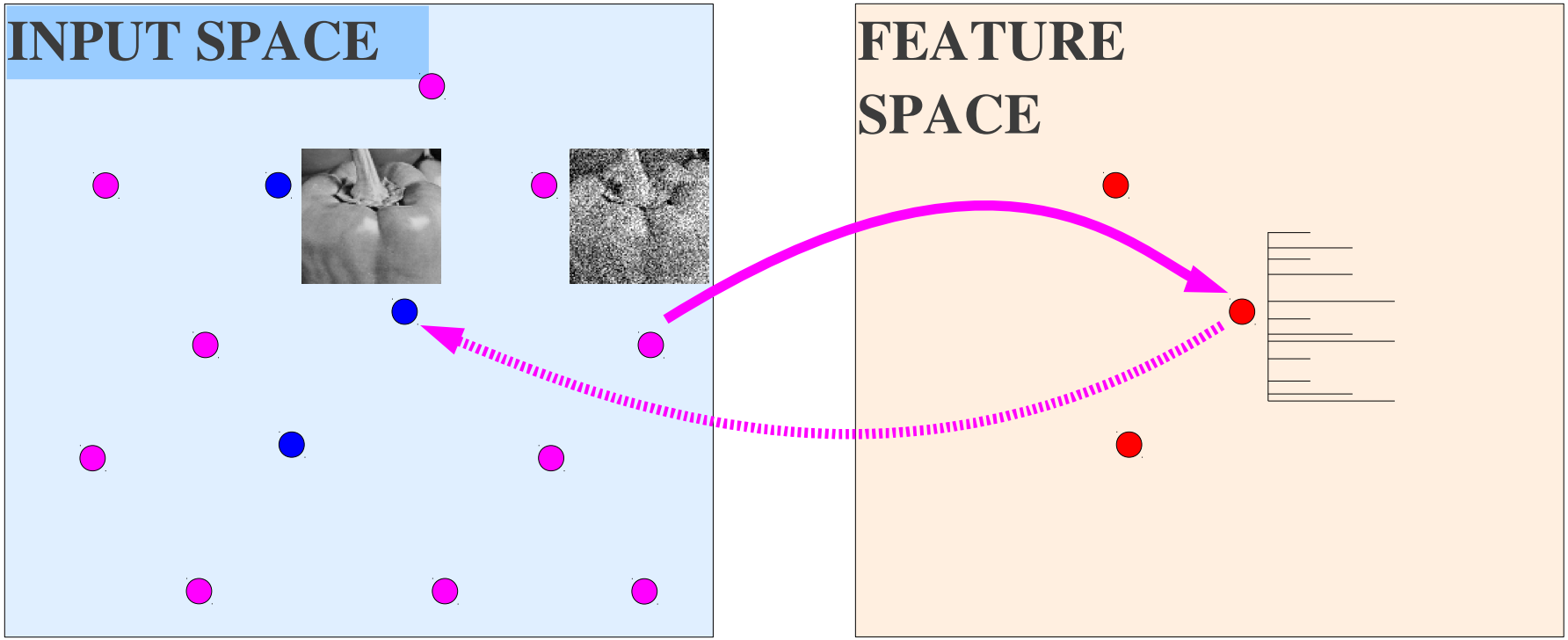
IDEA: reduce number of available codes.



Why Limit the Information Content of the Code?

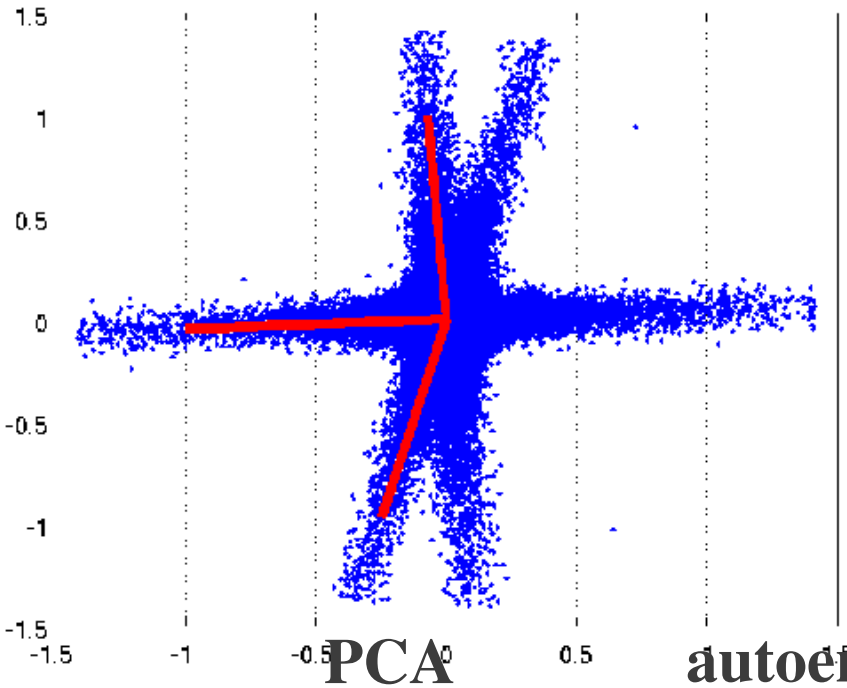
- Training sample
- Input vector which is NOT a training sample
- Feature vector

IDEA: reduce number of available codes.



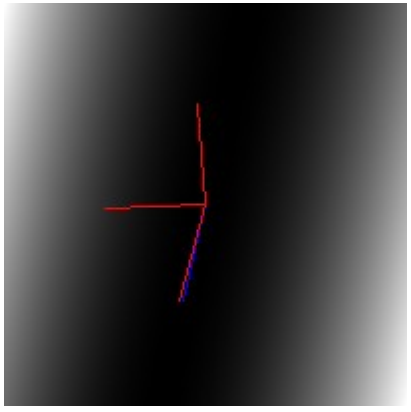
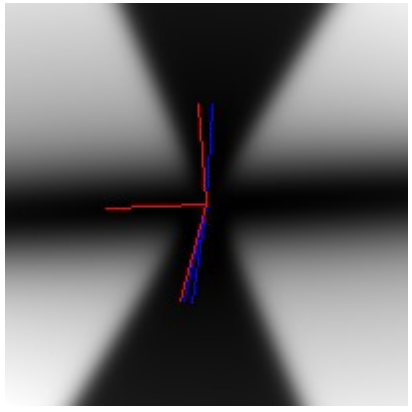
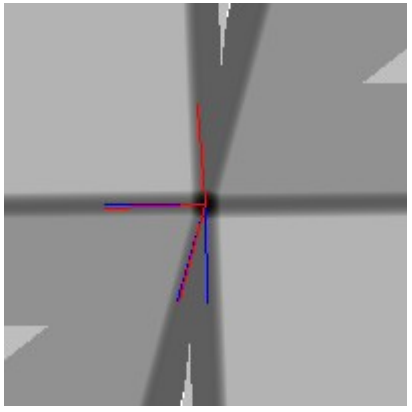
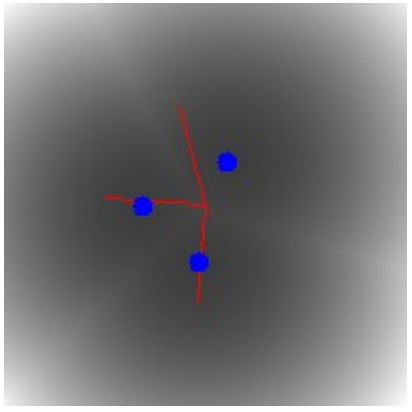
Sparsity Penalty to Restrict the Code

- We are going to impose a sparsity penalty on the code to restrict its information content.
- We will allow the code to have higher dimension than the input
- Categories are more easily separable in high-dim sparse feature spaces
  - This is a trick that SVM use: they have one dimension per sample
- Sparse features are optimal when an active feature costs more than an inactive one (zero).
  - e.g. neurons that spike consume more energy
  - The brain is about 2% active on average.

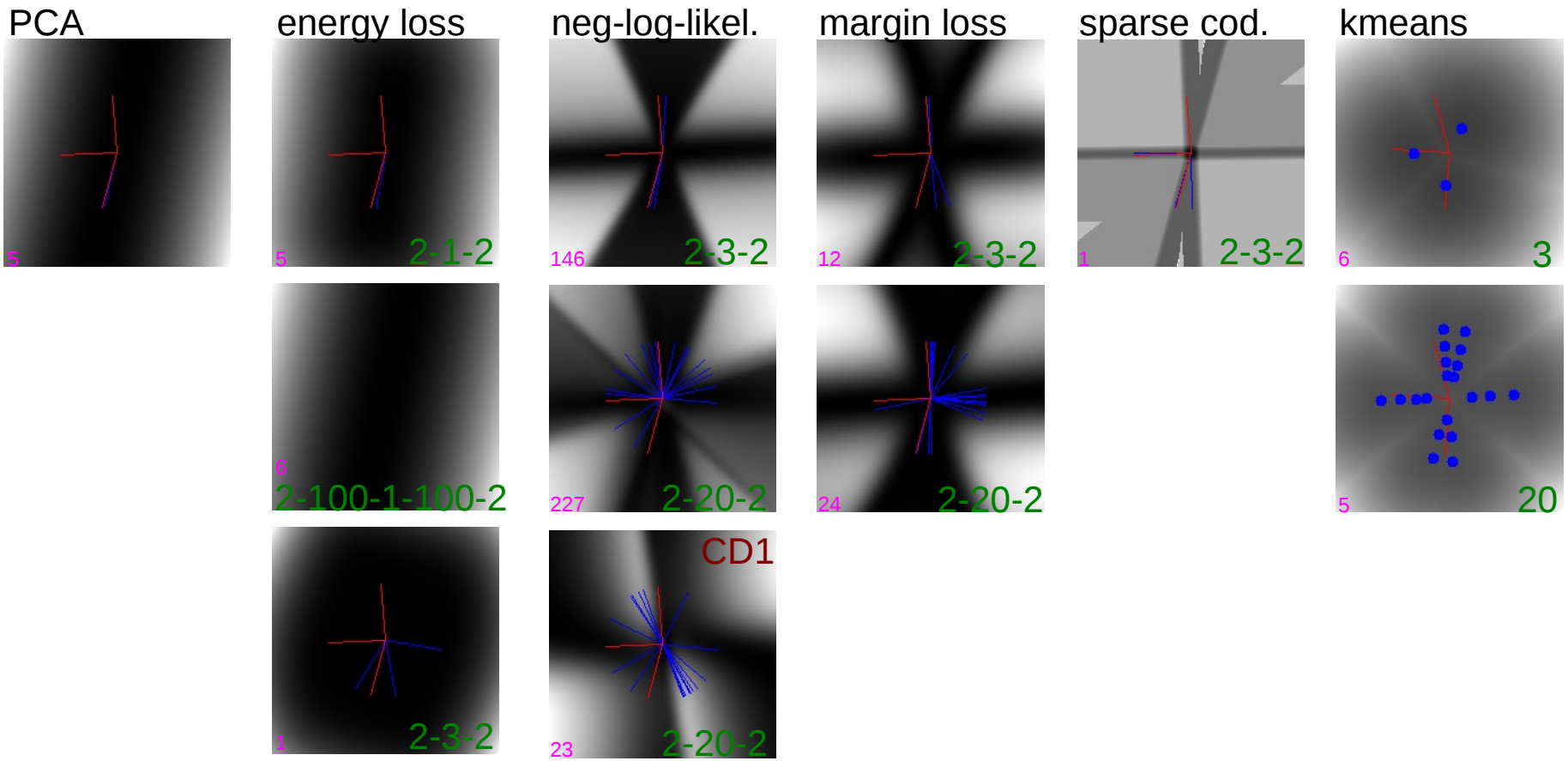


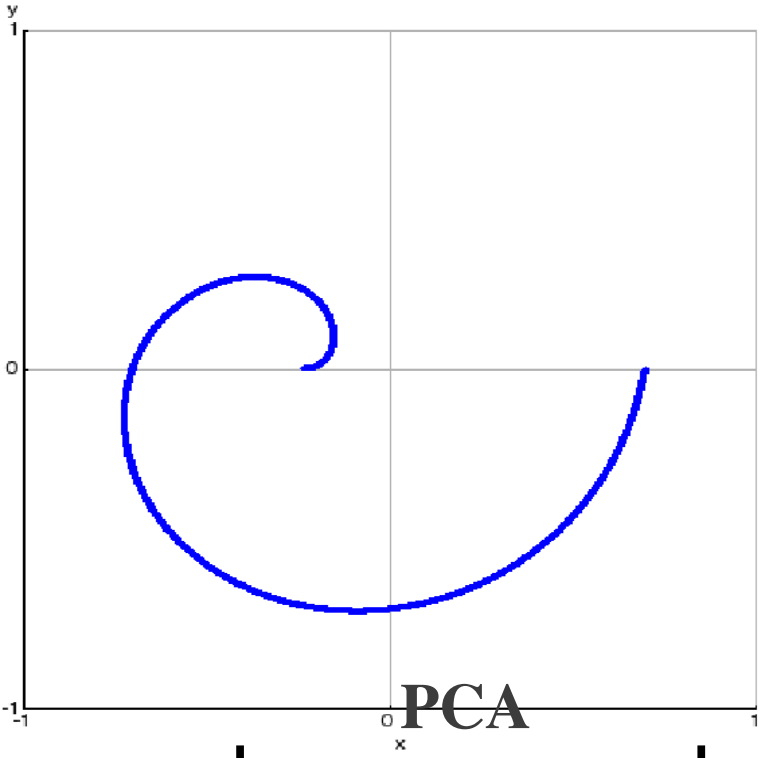
- 2 dimensional toy dataset
  - Mixture of 3 Cauchy distrib.
- Visualizing energy surface (black = low, white = high)

[Ranzato 's PhD thesis 2009]

	PCA (1 code unit)	autoencoder (3 code units)	sparse coding (3 code units)	K-Means (3 code units)
encoder	$W'Y$	$\sigma(W_e Y)$	–	–
decoder	WZ	$W_d Z$	WZ	WZ
energy	$\|\|Y-WZ\|\|^2$	$\|\|Y-WZ\|\|^2$	$\|\|Y-WZ\|\|^2+\lambda\|Z\|$	$\|\|Y-WZ\|\|^2$
loss	$F(Y)$	$F(Y)+\log\Gamma$	$F(Y)$	$F(Y)$
pull-up	dimens.	part. func.	sparsity	1-of-N code
				

Energies Surfaces for Various Loss Functions

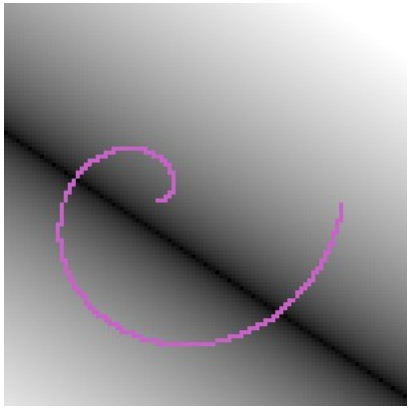


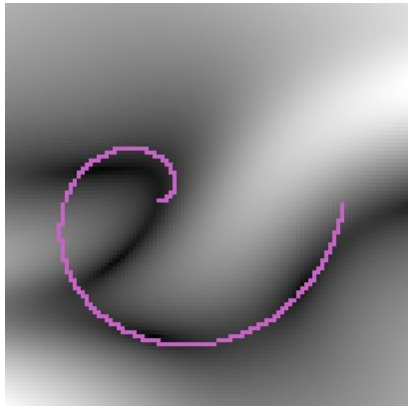


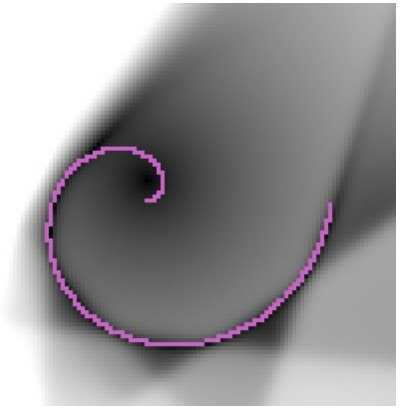
- 2 dimensional toy dataset
  - spiral
- Visualizing energy surface

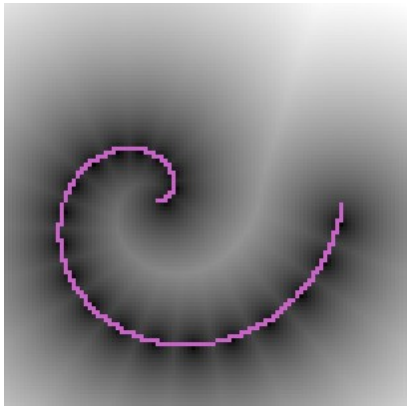
(black = low, white = high)

	PCA (1 code unit)	autoencoder (1 code unit)	sparse coding (20 code units)	K-Means (20 code units)
encoder	$W'Y$	$\sigma(W_e Y)$	$\sigma(W_e Z)$	–
decoder	WZ	$W_d Z$	$W_d Z$	WZ
energy	$\|\|Y-WZ\|\|^2$	$\|\|Y-WZ\|\|^2$	$\|\|Y-WZ\|\|^2$	$\|\|Y-WZ\|\|^2$
loss	$F(Y)$	$F(Y)$	$F(Y)$	$F(Y)$
pull-up	dimens.	dimens.	sparsity	1-of-N code



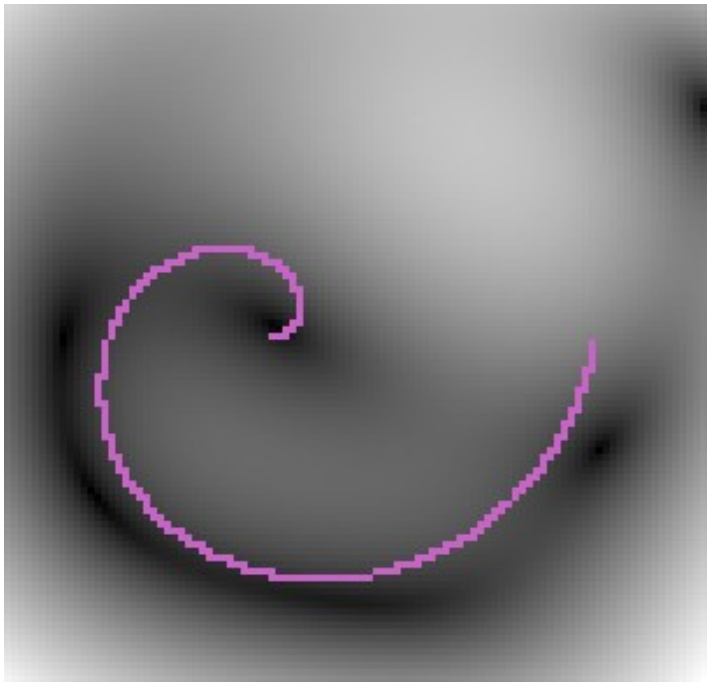




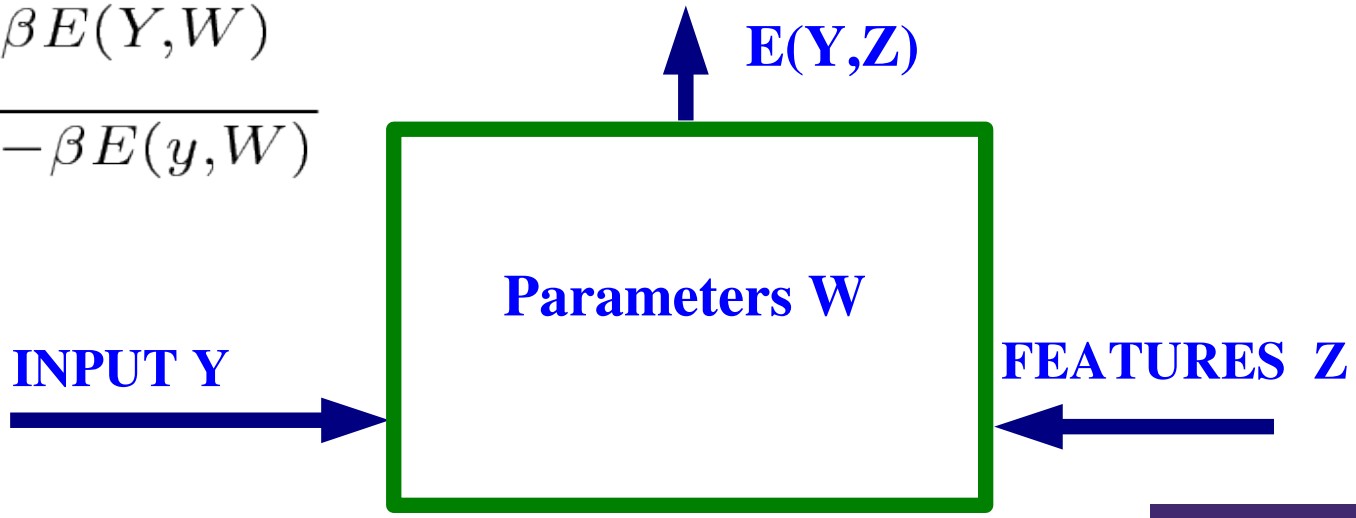


Unsupervised Feature Learning as Density Estimation

- **Contrast Function or Energy function:**
  - $E(Y,W) = \mathrm{MIN}_Z\ E(Y,Z,W)$
  - Y: input
  - Z: "features", representation, latent variables
  - W: parameters of the model (to be learned)
  - Maximum A Posteriori approximation for Z
- **Density function P(Y|W)**
  - Learn W so as to maximize the likelihood of the training data under the model

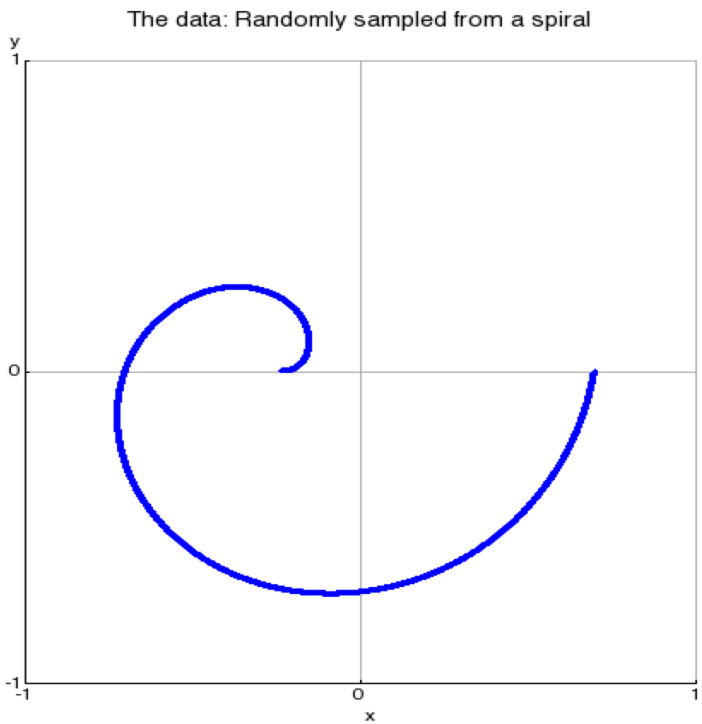


$$P(Y|W) = \frac{e^{-\beta E(Y,W)}}{\int_y e^{-\beta E(y,W)}}$$



Example: A Toy Problem. The Spiral

- Dataset
  - 10,000 random points along a spiral in a 2D plane
  - The spiral fits in a square with opposite corners (-1,1), (1,-1)
  - The spiral is designed so that no function can predict a single value of from
- Goal
  - Learn an energy surface with low energies along the spiral and high energy everywhere else



PCA (Principal Component Analysis)

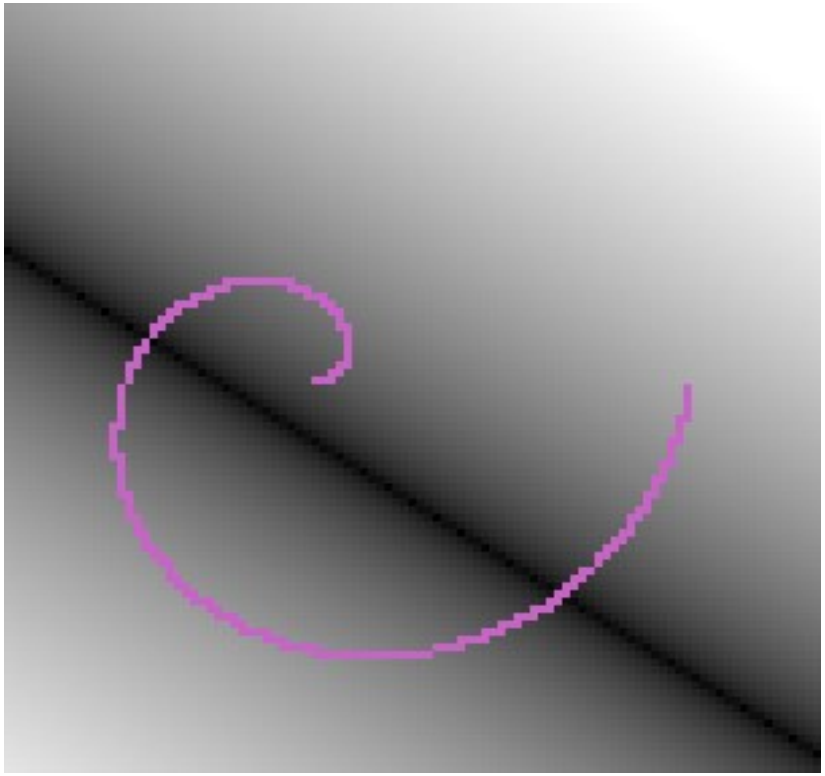
- Can be seen as encoder-decoder architecture that minimizes mean square reconstruction error (energy loss)
- Optimal code is constrained to be equal to the value predicted by encoder
- Flat surfaces are avoided by using a code of low dimension

$$Enc(Y) = WY$$

$$Dec(Z) = W^T Y, \text{ where } W \in \Re^{NxM}$$

$$C_e(Y, Z) = \|WY - Z\|^2$$

$$C_d(Y, Z) = \|W^T Z - Y\|^2$$



- For large value of γ energy reduces to

$$E(Y, W) = \|W^T WY - Y\|^2$$

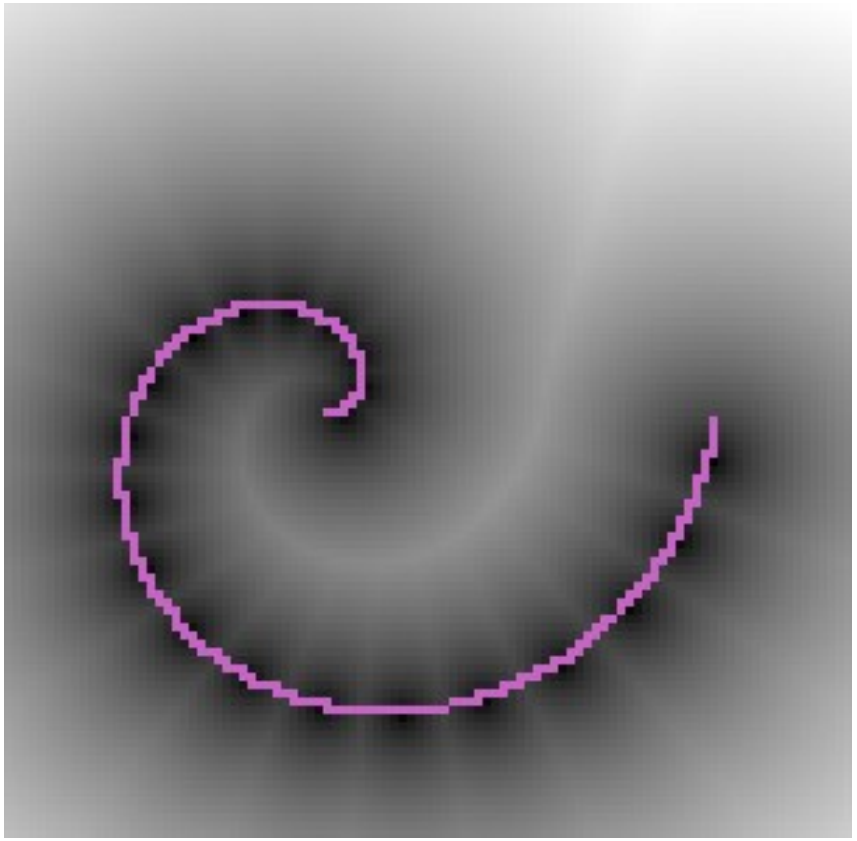
K-Means

- In this architecture the code Z is a binary vector of size N (N being the number of prototypes)
- For any sample Y, only one component is active (equal to 1) and all the others are inactive (equal to 0).
- This is a one-of-N sparse binary code
- The energy is:

$$E(Y,Z)=\sum_i Z_i \|Y-W_i\|^2$$

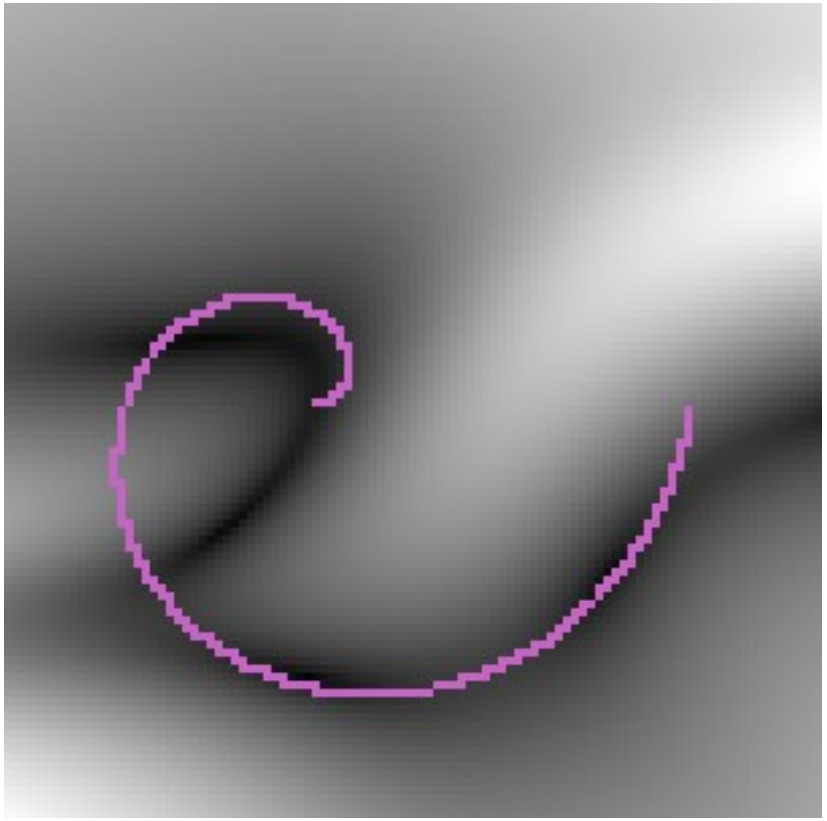
$$W_i \text{ is the } i^{th} \text{ prototype}$$

- Inference involves finding Z that minimizes the energy



Auto-encoder neural net (2-100-1-100-2 architecture)

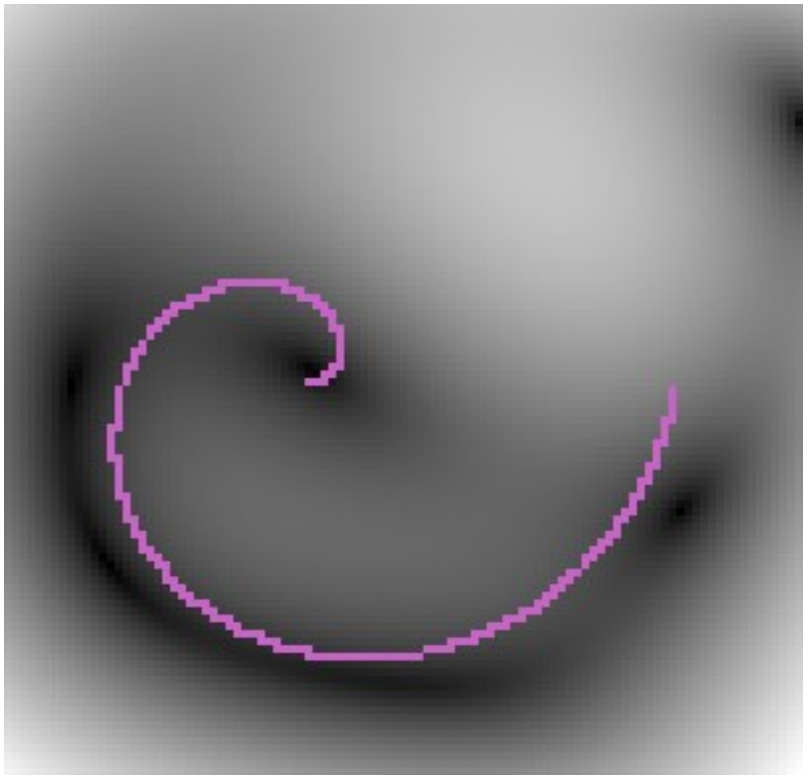
- A neural network autoencoder that learns a low dimensional representation.
- Architecture used
  - Input layer: 2 units
  - First hidden layer: 100 units
  - Second hidden layer (the code): 1 unit
  - Third hidden layer: 100 units
  - Output layer: 2 units
- Similar to PCA but non-linear
- Energy is



$$E(Y) = |Dec(Enc(Y)) - Y|^2$$

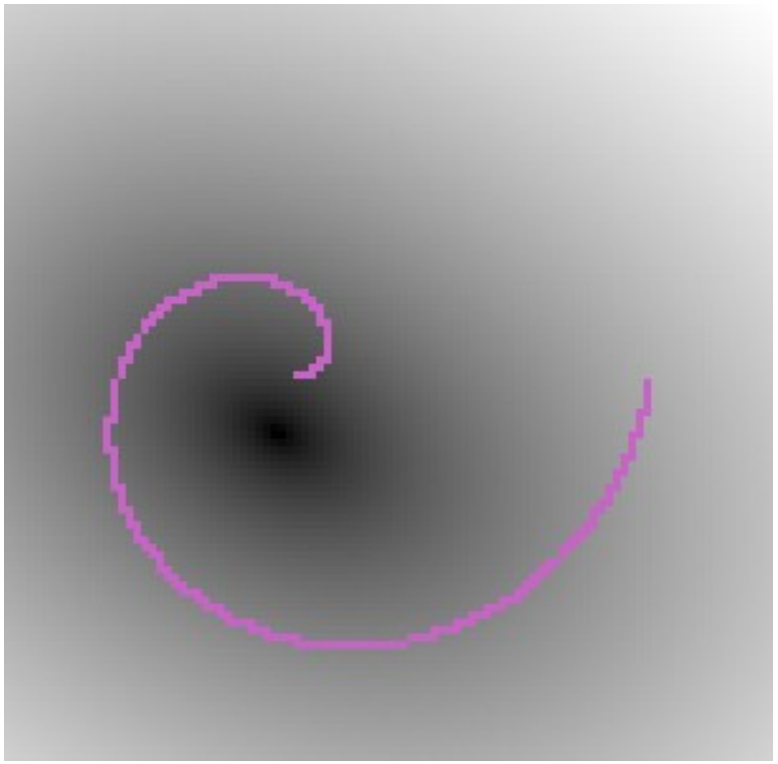
Wide auto-encoder neural net with energy loss

- The energy loss simply pulls down on the energy of the training samples (no contrastive term).
- Because the dimension of the code is larger than the input, nothing prevents the architecture from learning the identity function, which gives a very flat energy surface (a collapse): everything is perfectly reconstructed.
- Simplest example: a multi layer neural network with identical input and output layers and a large hidden layer.
- Architecture used
  - Input layer: 2 units
  - Hidden layer (the code): 20 units
  - Output layer: 2 units
- Energy loss leads to a collapse
- Tried a number of loss functions



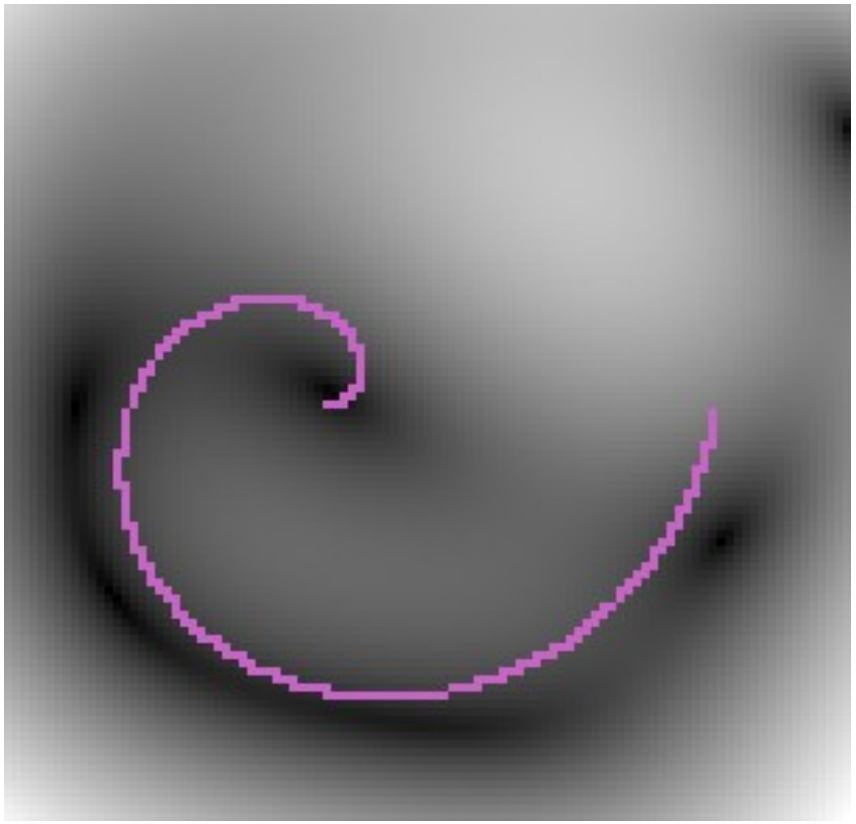
Wide auto-encoder with negative-log-likelihood loss

- Negative Log Likelihood Loss
  - Pull down on the energy of training (observed) samples
  - Pull up on the energies of all the other (unobserved) samples
  - Approximate the log of partition function through dense sampling.
  - Energy surface is very "stiff" because of small number of parameters.
  - Hence the energy surface is not perfect.



Wide auto-encoder with energy-based contrastive loss

- Linear-Linear Contrastive Loss
  - Avoid the cost associated with minimizing negative log likelihood
  - Idea is to pull up on unobserved points in the vicinity of training samples
  - We use Langevin dynamics to generate such points



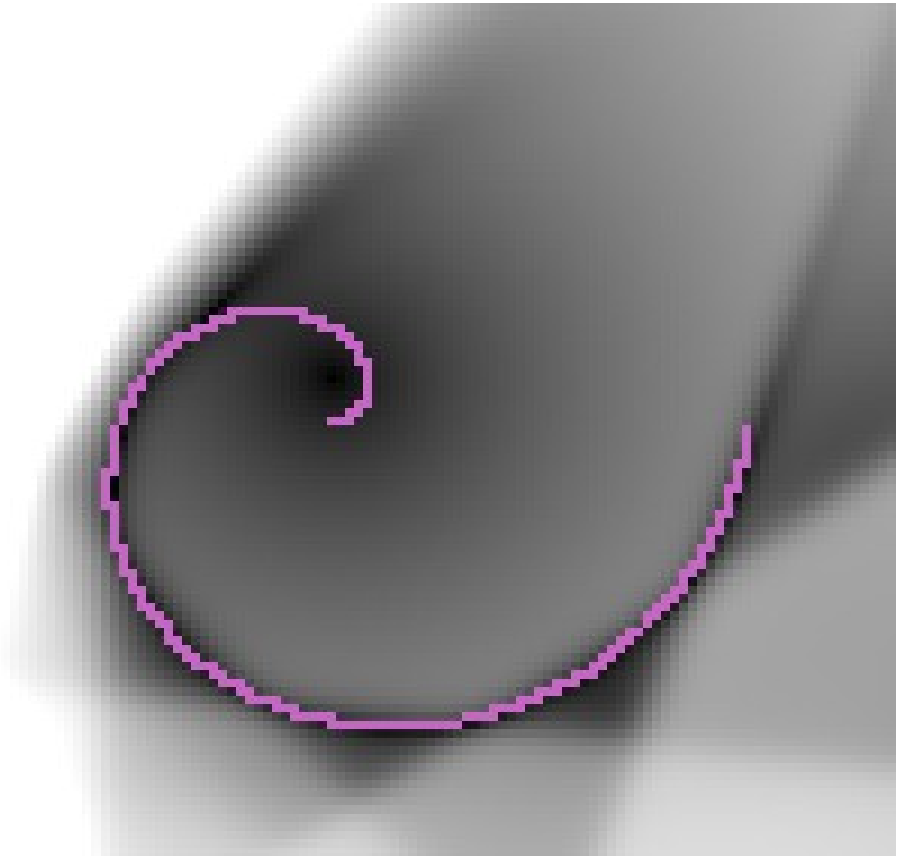
$$\bar{Y} \leftarrow \bar{Y} - \eta \frac{\delta E(\bar{Y})}{\delta Y} + \epsilon$$

  - The loss function is

$$L(Y, W) = \alpha E(Y, W) + max(0, m - E(\bar{Y}, W))$$

Wide auto-encoder with sparse code

- Sparse Codes
  - Limiting the information content of the code prevents flat energy surfaces, without the need to explicitly push up the bad points
  - Idea is to make the high dimensional code sparse by forcing each variable to be zero most of the time



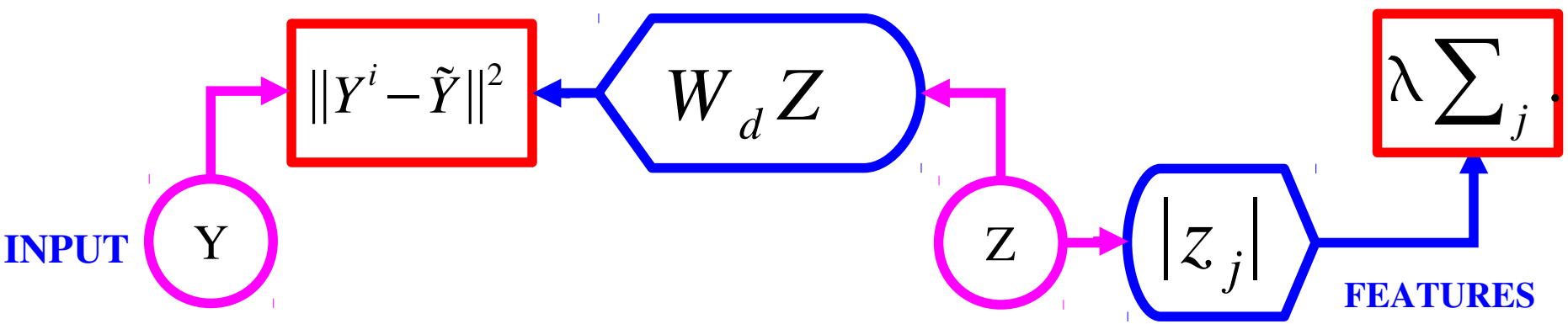
Unsupervised Feature Learning: variations on the sparse auto-encoder theme

Sparse Coding & Sparse Modeling

[Olshausen & Field 1997]

- Sparse linear reconstruction
- Energy = reconstruction_error + code_prediction_error + code_sparsity

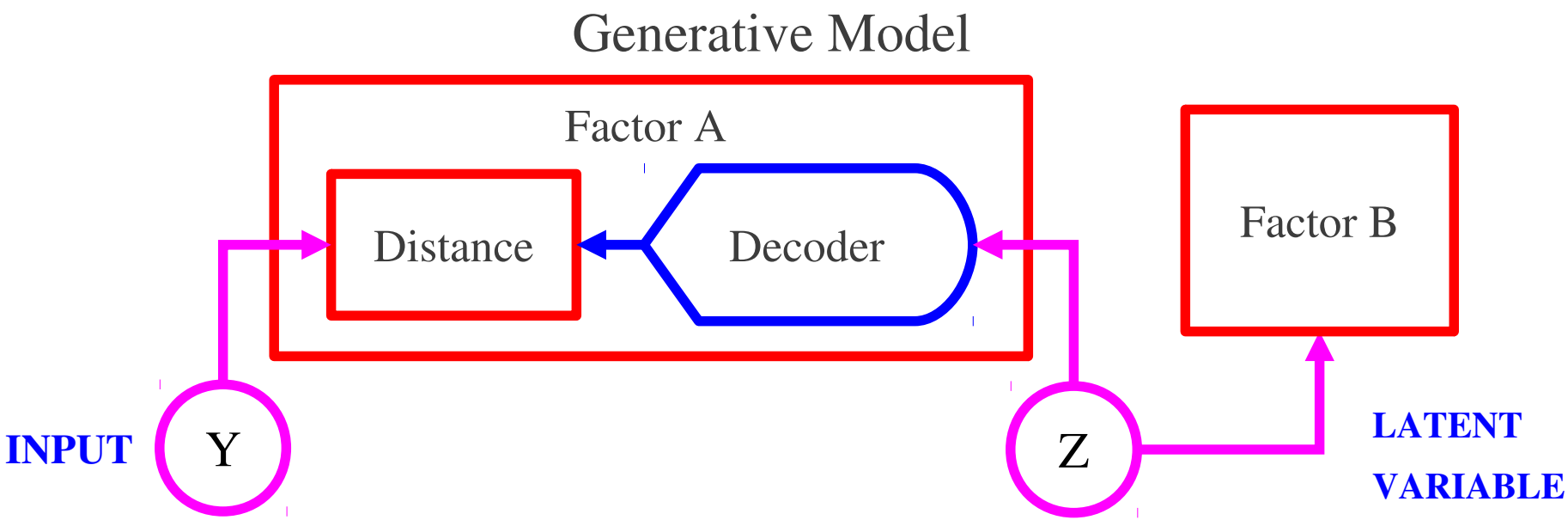
$$E(Y^i, Z) = \|Y^i - W_d Z\|^2 + \lambda \sum_j |z_j|$$



- Inference is slow $Y \rightarrow \hat{Z} = argmin_Z E(Y, Z)$

How to Speed Up Inference in a Generative Model?

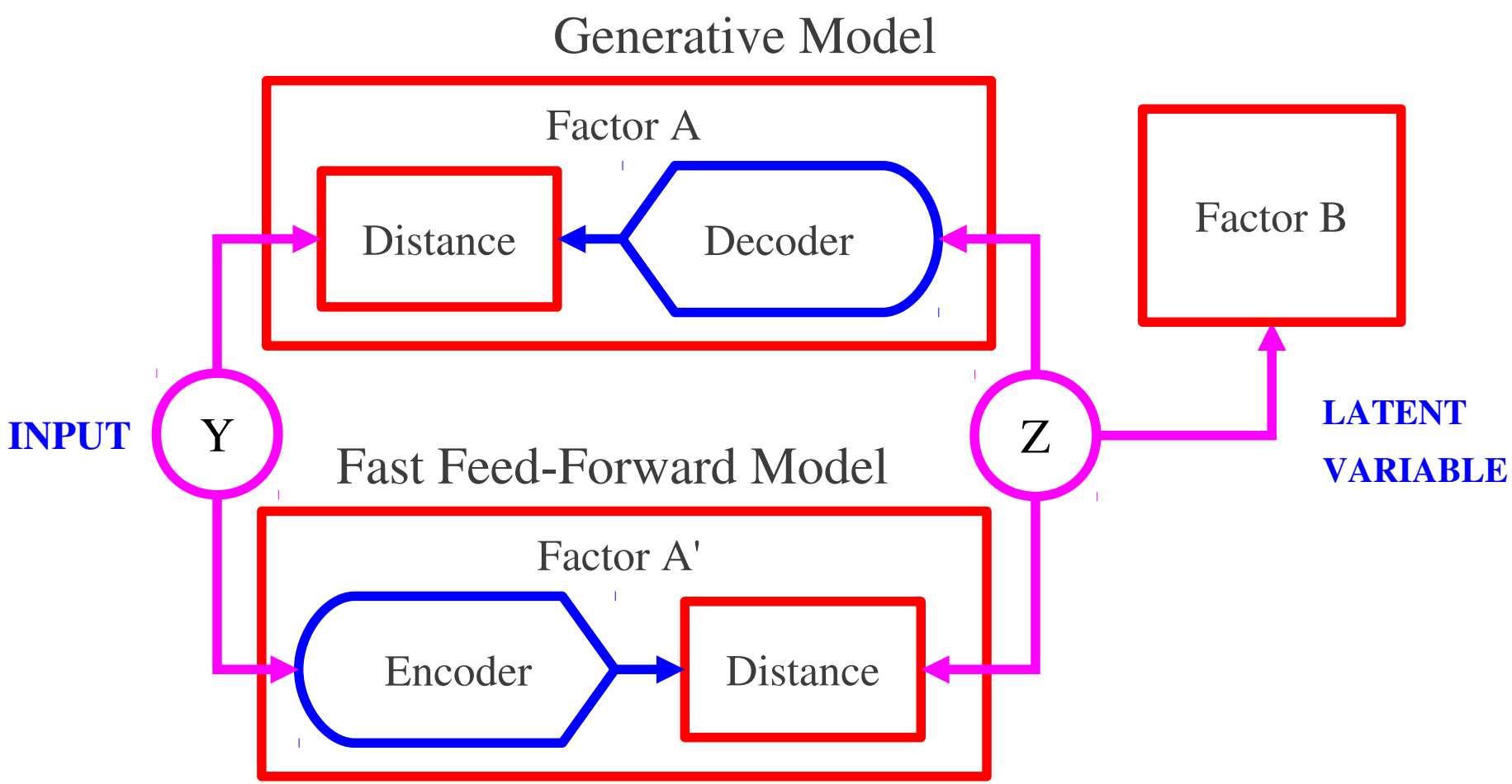
- Factor Graph with an asymmetric factor
- Inference Z → Y is easy
  - Run Z through deterministic decoder, and sample Y
- Inference Y → Z is hard, particularly if Decoder function is many-to-one
  - MAP: minimize sum of two factors with respect to Z
  - Z* = argmin_z Distance[Decoder(Z), Y] + FactorB(Z)



Idea: Train a "simple" function to approximate the solution

[Kavukcuoglu, Ranzato, LeCun, rejected by every conference, 2008-2009]

- Train a "simple" feed-forward function to predict the result of a complex optimization on the data points of interest



- 1. Find optimal Zi for all Yi; 2. Train Encoder to predict Zi from Yi

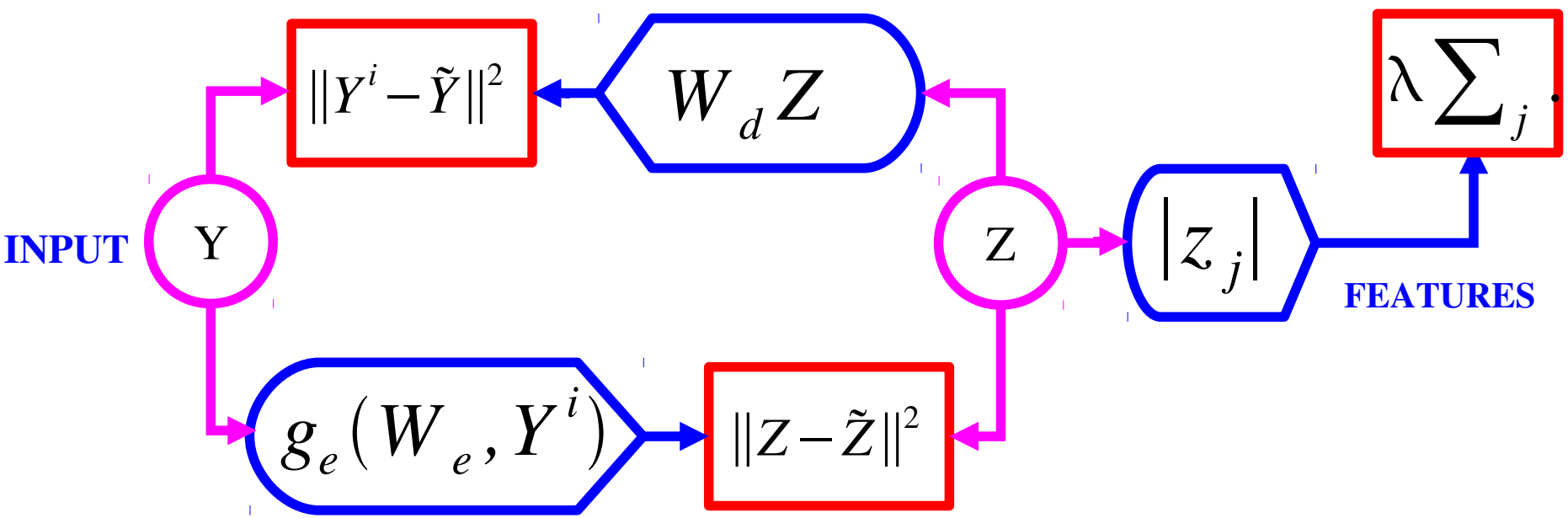
Predictive Sparse Decomposition (PSD): sparse auto-encoder

[Kavukcuoglu, Ranzato, LeCun, 2008 → arXiv:1010.3467],

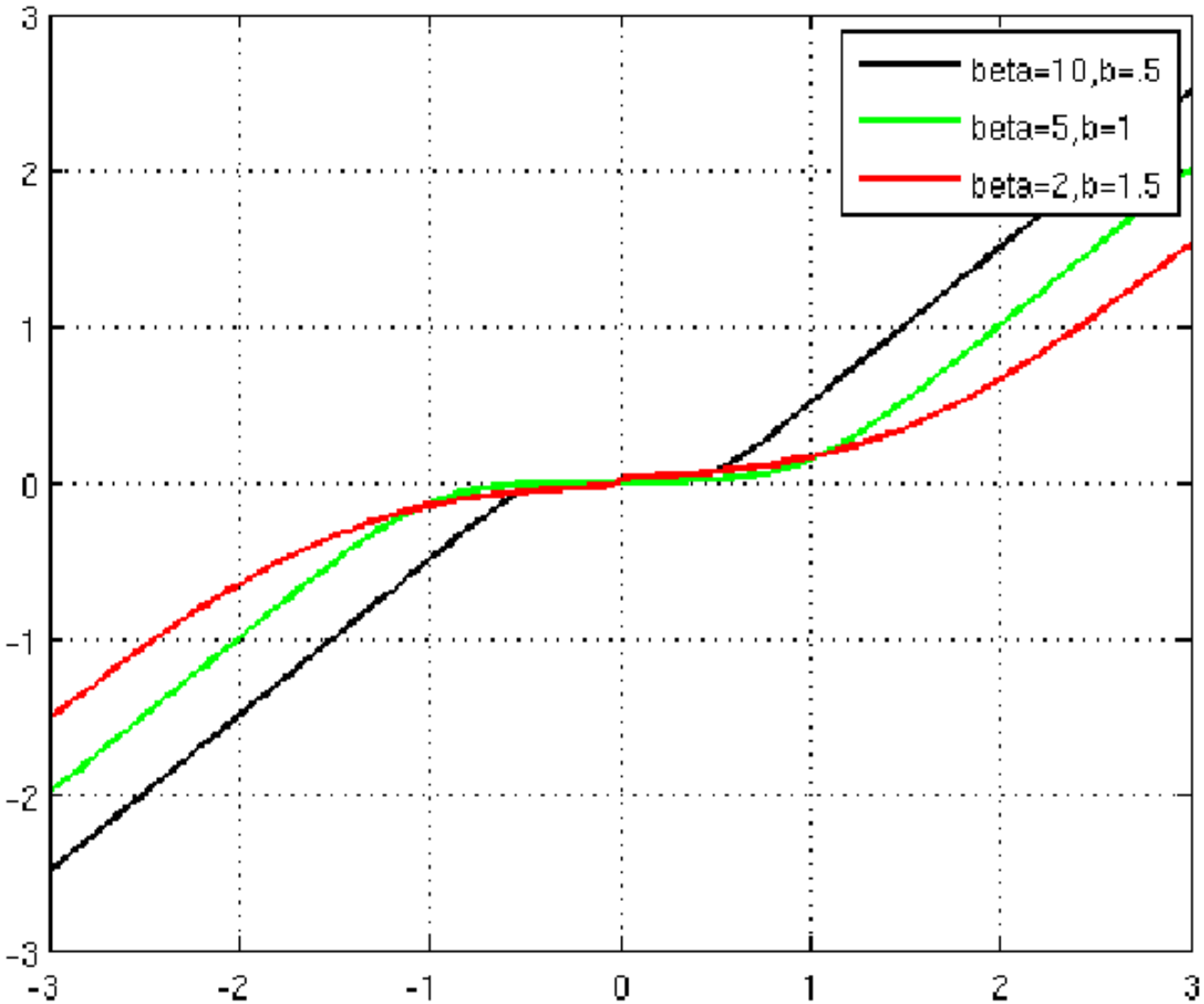
- Prediction the optimal code with a trained encoder
- Energy = reconstruction_error + code_prediction_error + code_sparsity

$$E(Y^i, Z) = \|Y^i - W_d Z\|^2 + \|Z - g_e(W_e, Y^i)\|^2 + \lambda \sum_j |z_j|$$

$$g_e(W_e, Y^i) = shrinkage(W_e Y^i)$$

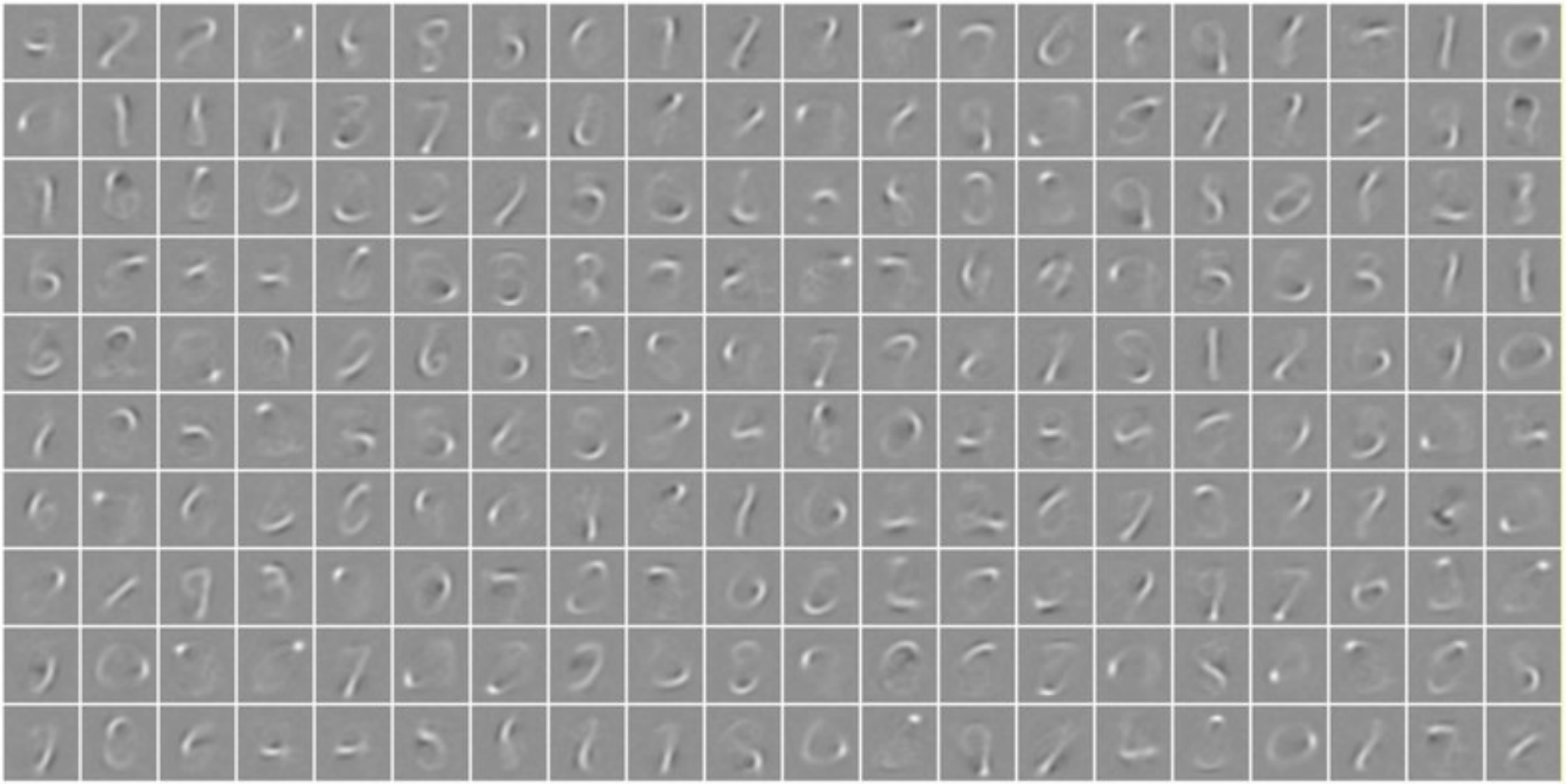


Soft Shrinkage Non-Linearity



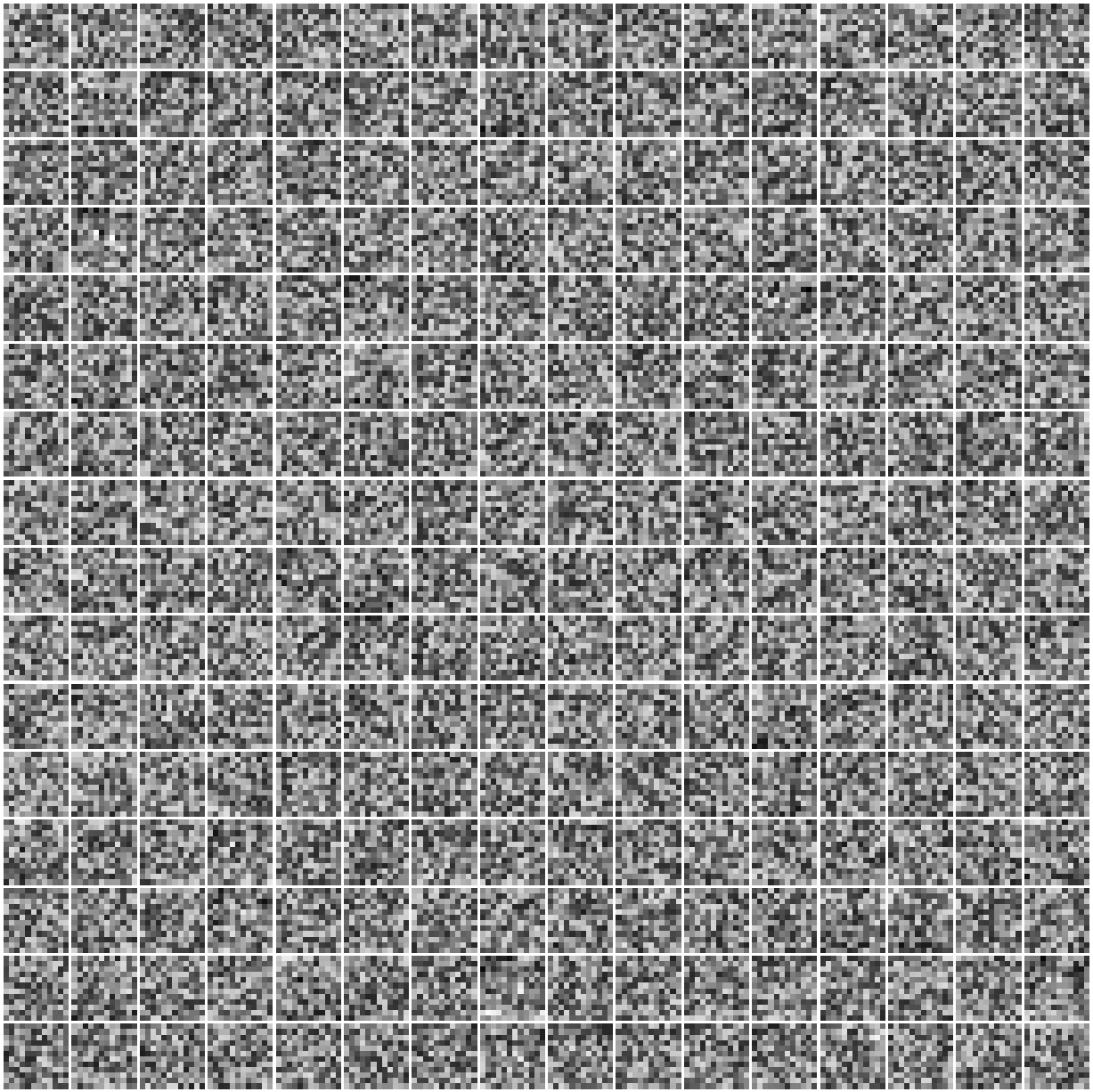
PSD: Basis Functions on MNIST

- Basis functions (and encoder matrix) are digit parts



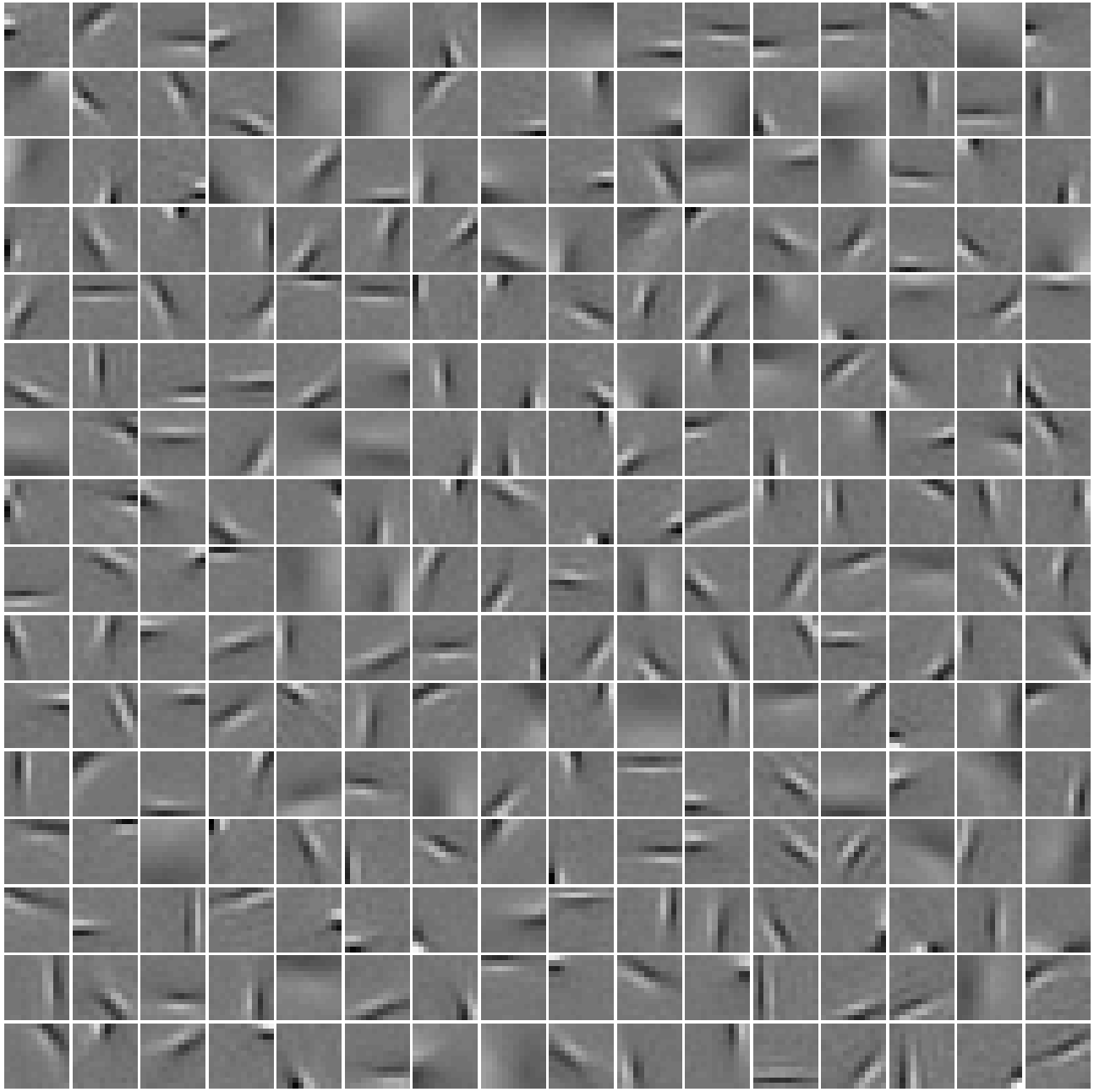
Predictive Sparse Decomposition (PSD): Training

- Training on natural images patches.
  - 12X12
  - 256 basis functions



iteration no 0

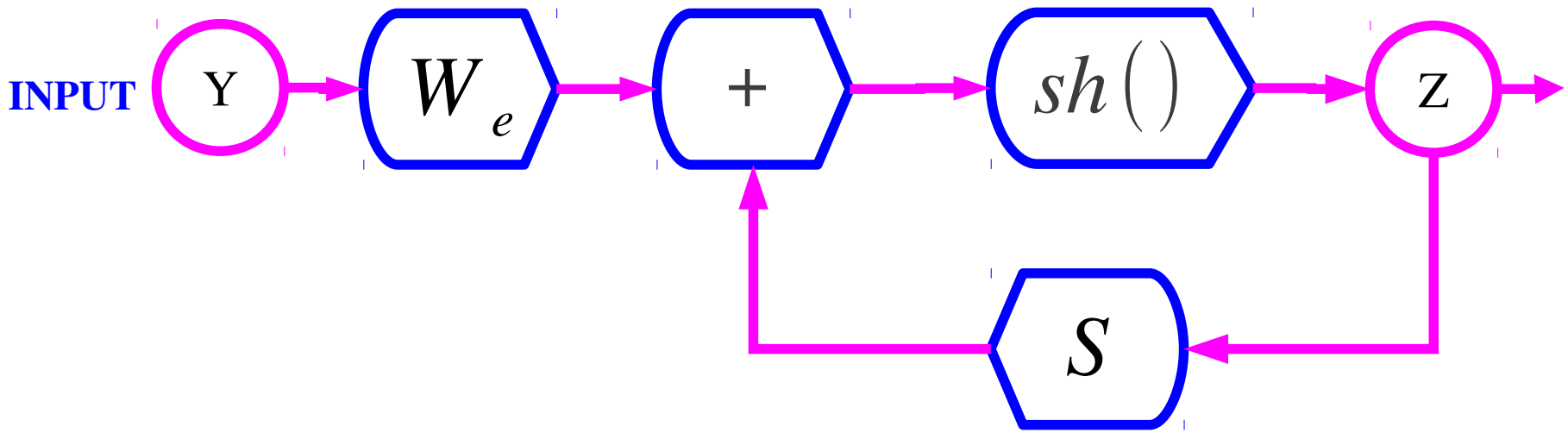
Learned Features on natural patches: V1-like receptive fields



Better Idea: Give the "right" structure to the encoder

[Gregor & LeCun, ICML 2010], [Bronstein et al. ICML 2012]

- ISTA/FISTA: iterative algorithm that converges to optimal sparse code

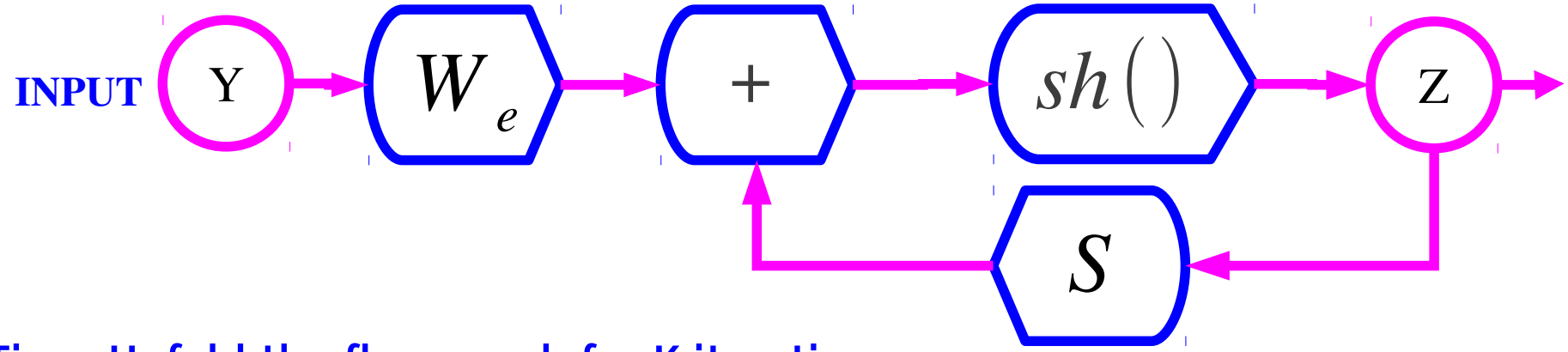


$$Z(t+1) = \text{Shrinkage}_{\lambda/L}\left[Z(t) - \frac{1}{L}W_d^T(W_d Z(t) - Y)\right]$$

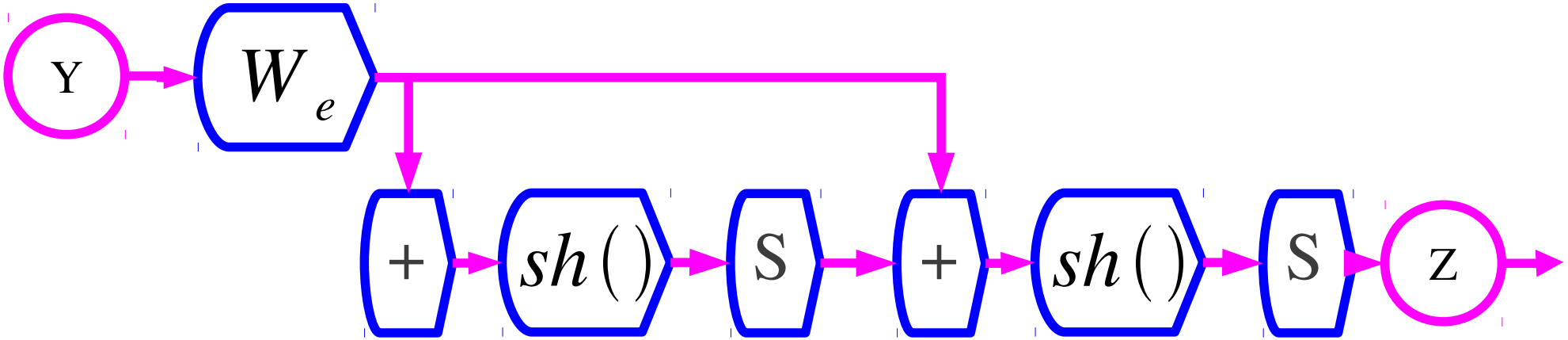
$$Z(t+1) = \text{Shrinkage}_{\lambda/L}\left[W_e^T Y + SZ(t)\right]; \quad W_e = \frac{1}{L}W_d; \quad S = I - \frac{1}{L}W_d^T W_d$$

LISTA: Train We and S matrices to give a good approximation quickly

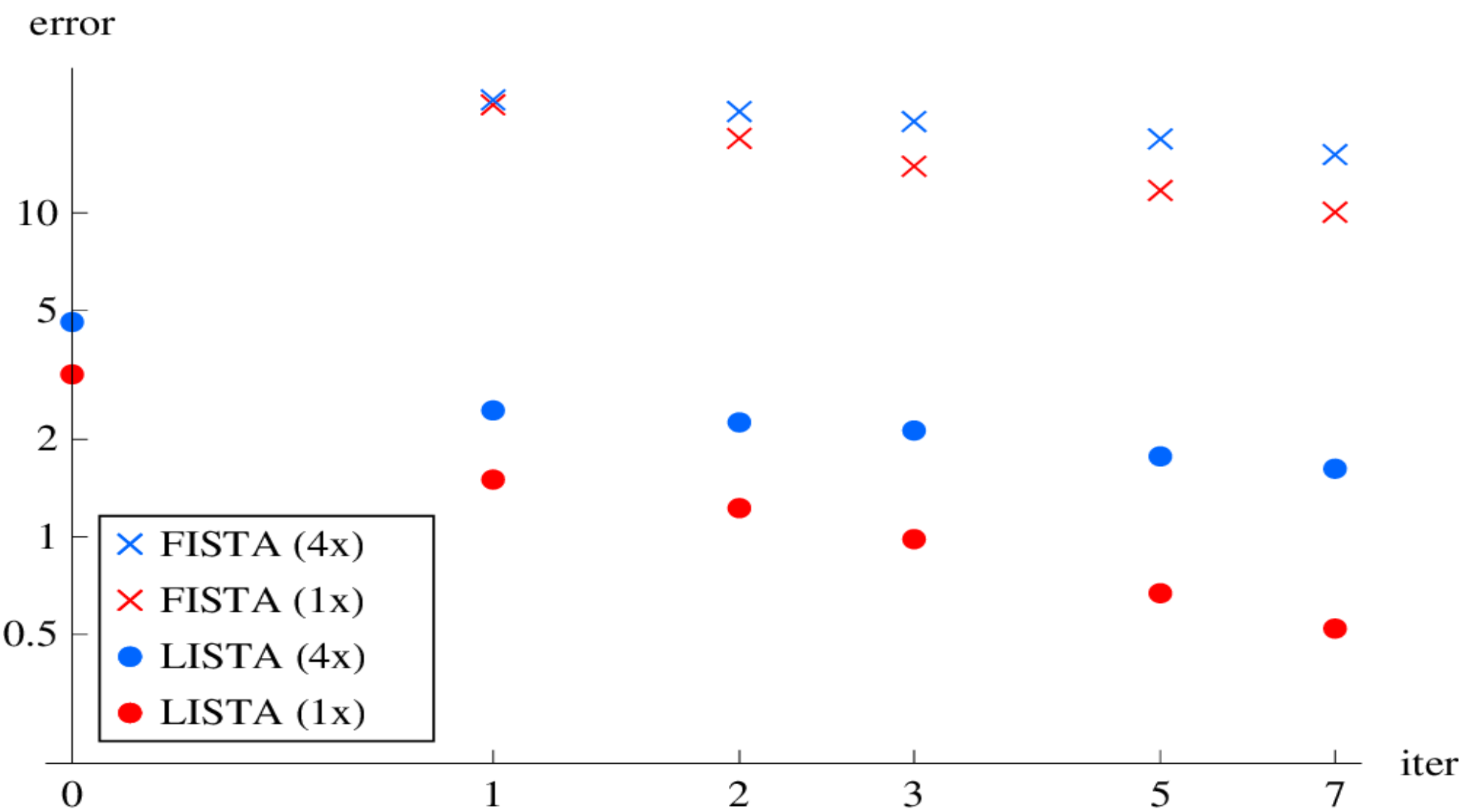
- Think of the FISTA flow graph as a recurrent neural net where We and S are trainable parameters



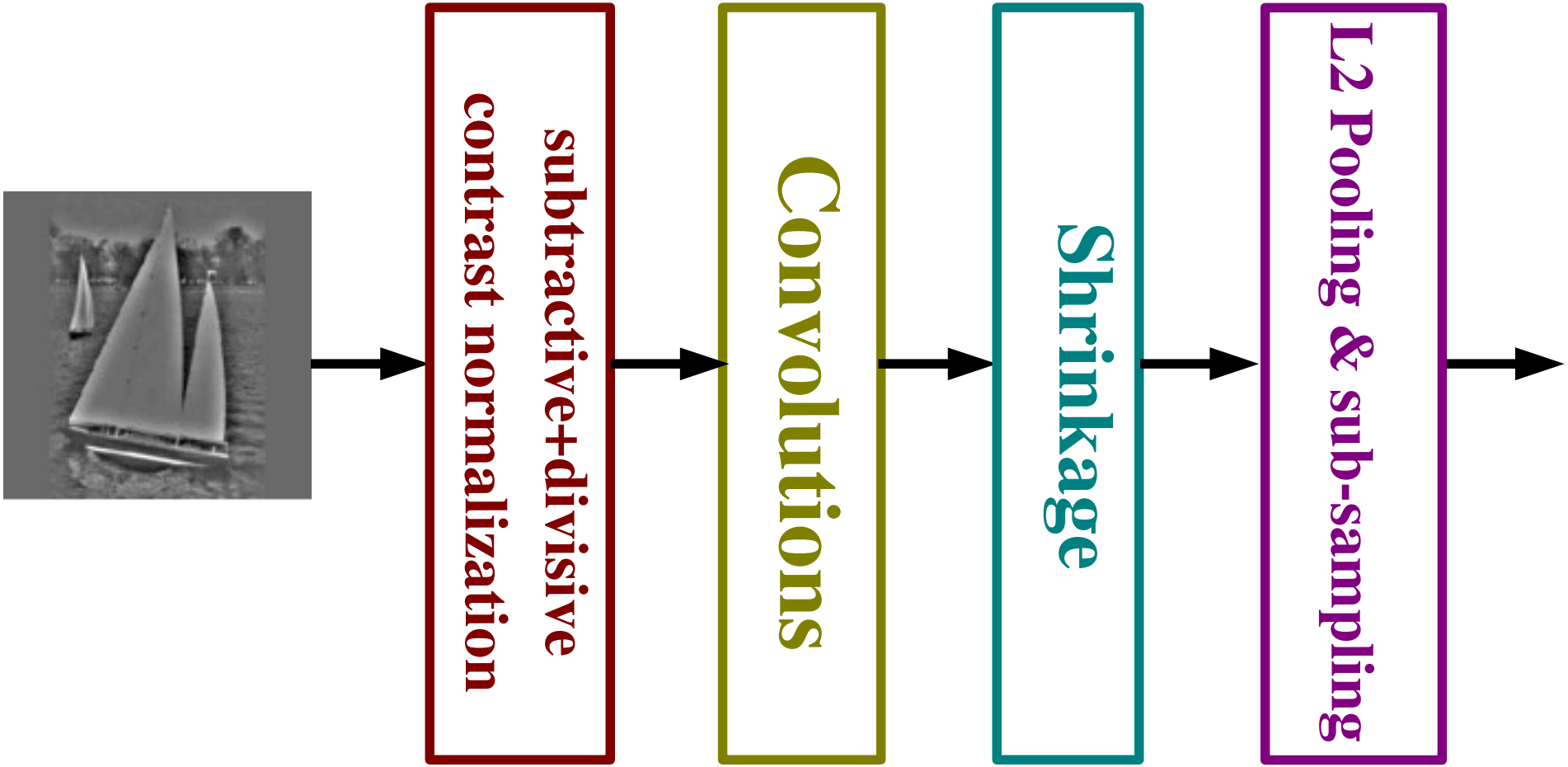
- Time-Unfold the flow graph for K iterations
- Learn the We and S matrices with "backprop-through-time"
- Get the best approximate solution within K iterations



Learning ISTA (LISTA) vs ISTA/FISTA



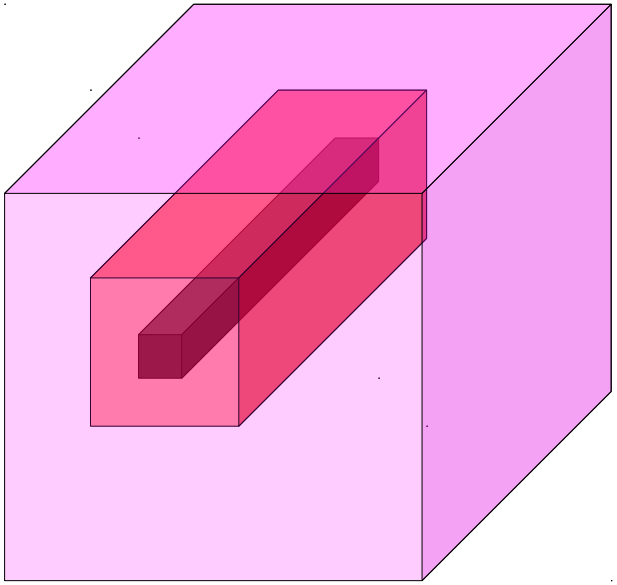
One Stage: filter → Shrinkage → L2 Pooling → Contrast Norm



THIS IS ONE STAGE OF THE CONVNET

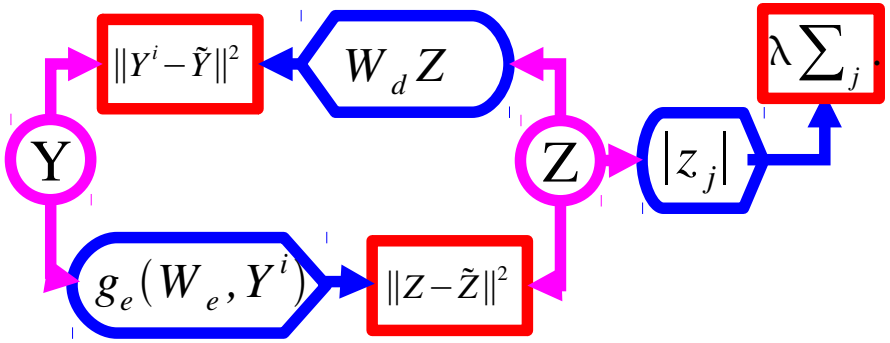
Local Contrast Normalization

- Performed on the state of every layer, including the input
- Subtractive Local Contrast Normalization
  - Subtracts from every value in a feature a Gaussian-weighted average of its neighbors (high-pass filter)
- Divisive Local Contrast Normalization
  - Divides every value in a layer by the standard deviation of its neighbors over space and over all feature maps
- Subtractive + Divisive LCN performs a kind of approximate whitening.



Using PSD to Train a Hierarchy of Features

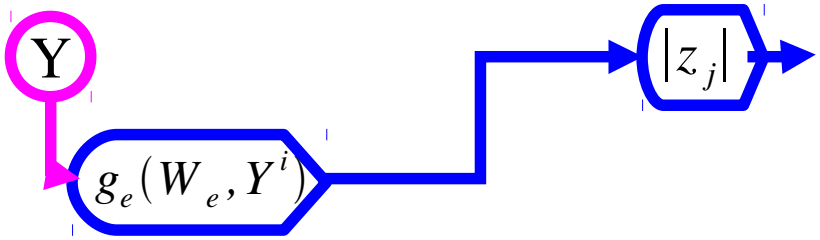
- Phase 1: train first layer using PSD



FEATURES

Using PSD to Train a Hierarchy of Features

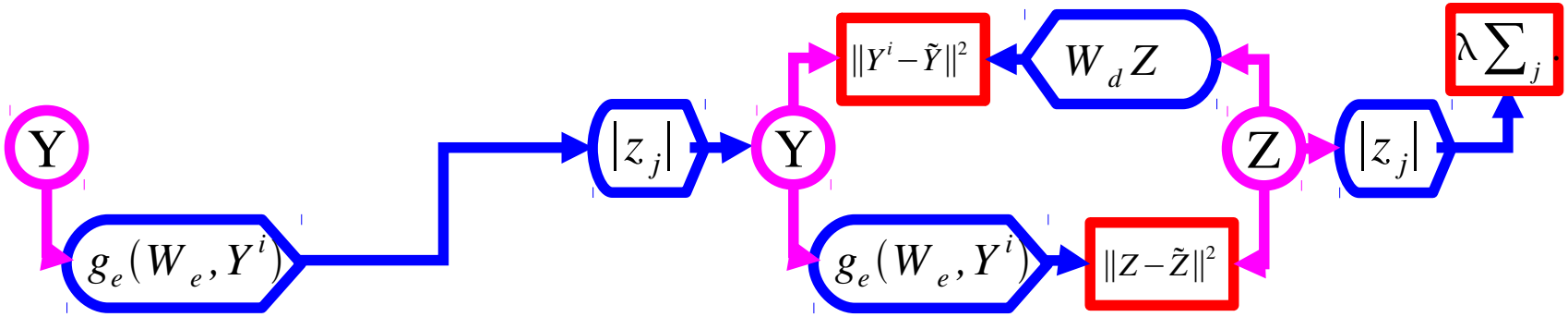
- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor



FEATURES

Using PSD to Train a Hierarchy of Features

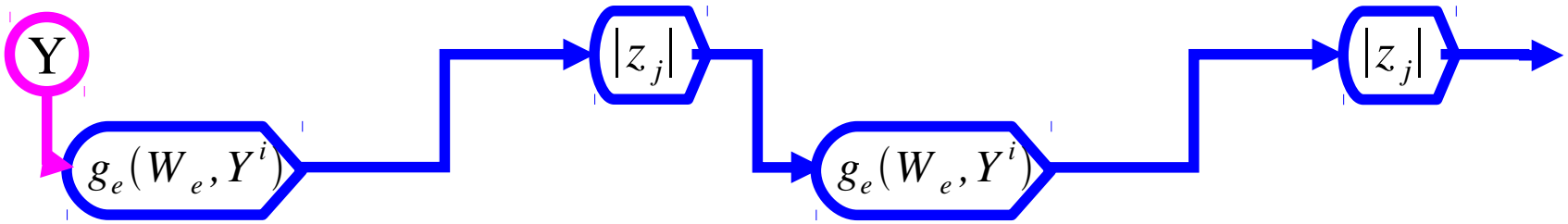
- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor
- Phase 3: train the second layer using PSD



FEATURES

Using PSD to Train a Hierarchy of Features

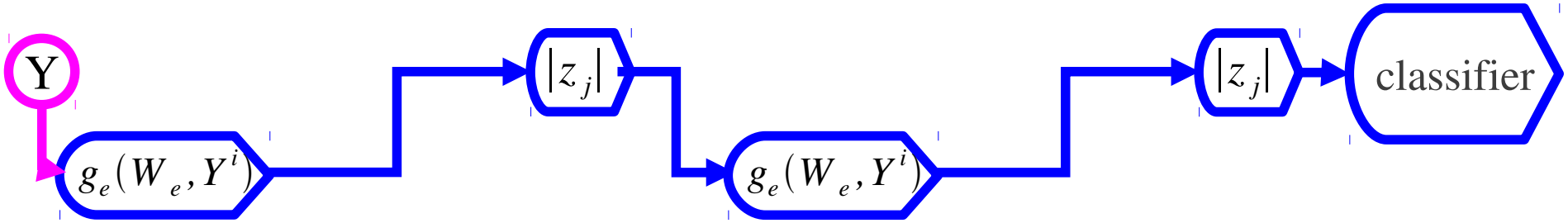
- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor
- Phase 3: train the second layer using PSD
- Phase 4: use encoder + absolute value as 2nd feature extractor



FEATURES

Using PSD to Train a Hierarchy of Features

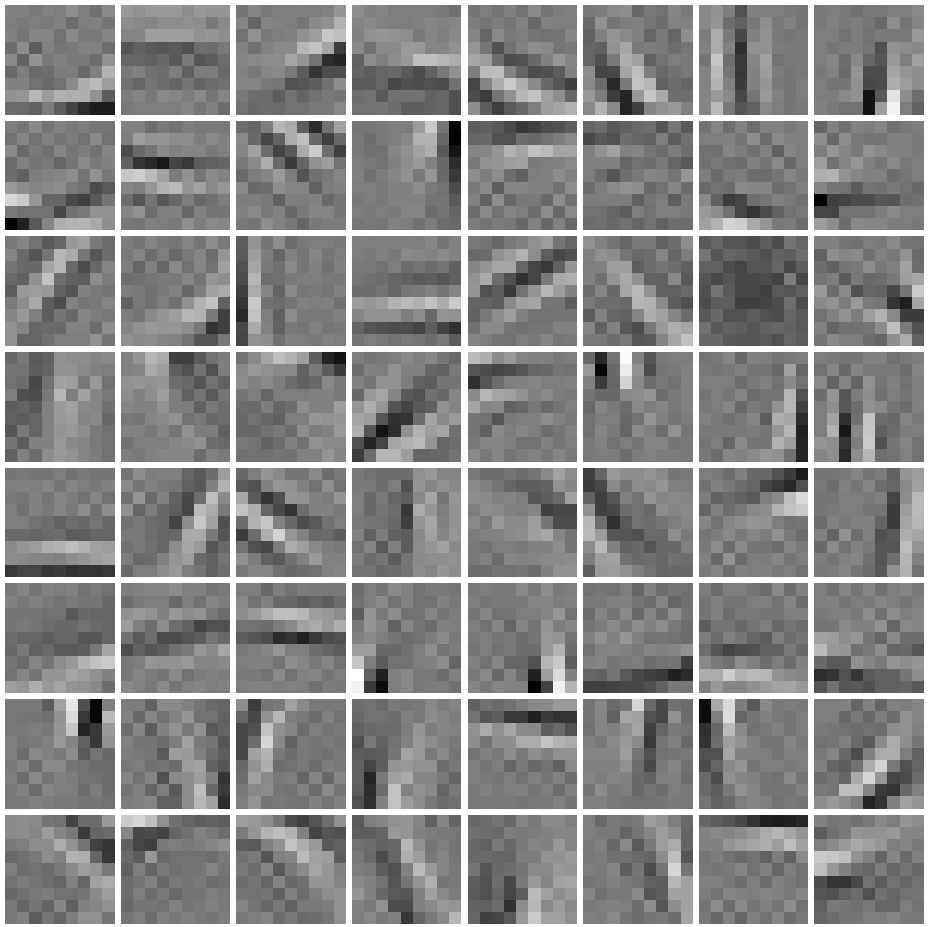
- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor
- Phase 3: train the second layer using PSD
- Phase 4: use encoder + absolute value as 2nd feature extractor
- Phase 5: train a supervised classifier on top
- Phase 6 (optional): train the entire system with supervised back-propagation



FEATURES

Using PSD Features for Object Recognition

- 64 filters on 9x9 patches trained with PSD
  - with Linear-Sigmoid-Diagonal Encoder

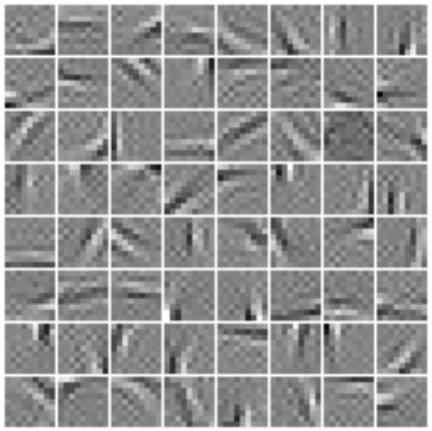


weights :-0.2828 - 0.3043

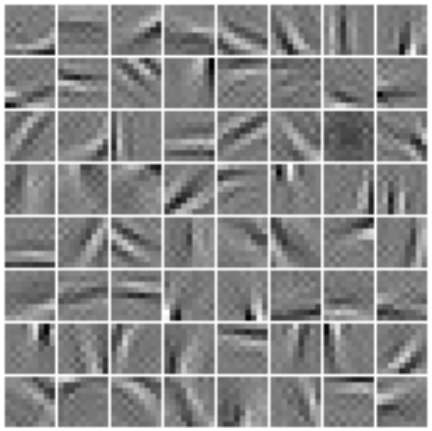
Multistage Hubel-Wiesel Architecture: Filters

- After PSD
- After supervised refinement

- Stage 1

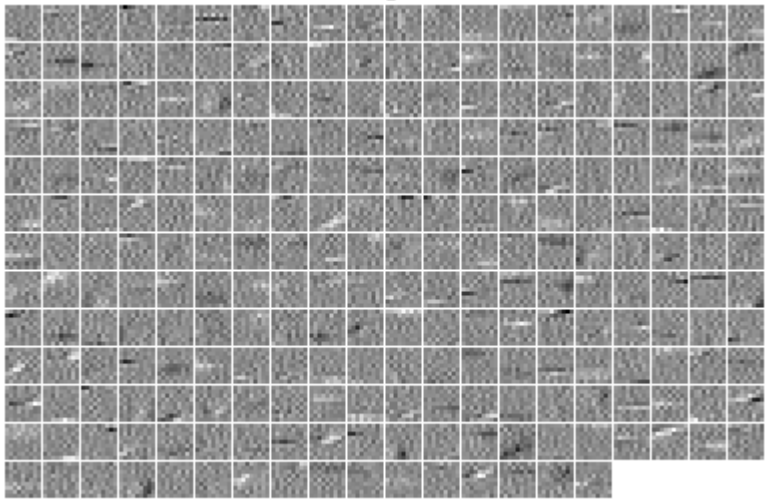


weights :-0.2232 - 0.2075

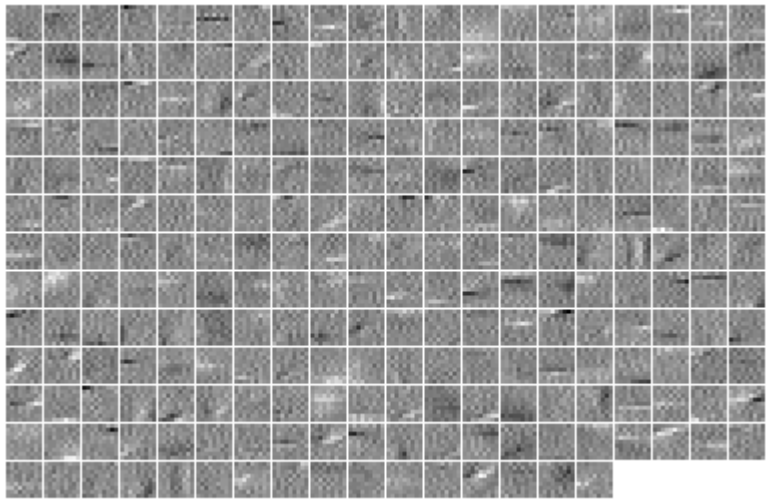


weights :-0.2828 - 0.3043

- Stage2



weights :-0.0778 - 0.064



weights :-0.0929 - 0.0784

Results on Caltech101 with sigmoid non-linearity

Single Stage System: $[64.F^{9\times9}_{CSG} - R/N/P^{5\times5}]$ - log_reg					
R/N/P	$R_{abs} - N - P_A$	$R_{abs} - P_A$	$N - P_M$	$N - P_A$	P_A
U^+	54.2%	50.0%	44.3%	18.5%	14.5%
R^+	54.8%	47.0%	38.0%	16.3%	14.3%
U	52.2%	43.3%(±1.6)	44.0%	17.2%	13.4%
R	53.3%	31.7%	32.1%	15.3%	12.1%(±2.2)
G	52.3%				

Two Stage System: $[64.F^{9\times9}_{CSG} - R/N/P^{5\times5}] - [256.F^{9\times9}_{CSG} - R/N/P^{4\times4}]$ - log_reg					
R/N/P	$R_{abs} - N - P_A$	$R_{abs} - P_A$	$N - P_M$	$N - P_A$	P_A
U^+U^+	65.5%	60.5%	61.0%	34.0%	32.0%
R^+R^+	64.7%	59.5%	60.0%	31.0%	29.7%
UU	63.7%	46.7%	56.0%	23.1%	9.1%
RR	62.9%	33.7%(±1.5)	37.6%(±1.9)	19.6%	8.8%
GT	55.8%	← like HMAX model			

Single Stage: $[64.F^{9\times9}_{CSG} - R/N/P^{5\times5}]$ - PMK-SVM		
U	64.0%	

Two Stages: $[64.F^{9\times9}_{CSG} - R/N/P^{5\times5}] - [256.F^{9\times9}_{CSG} - R/N]$ - PMK-SVM		
UU	52.8%	

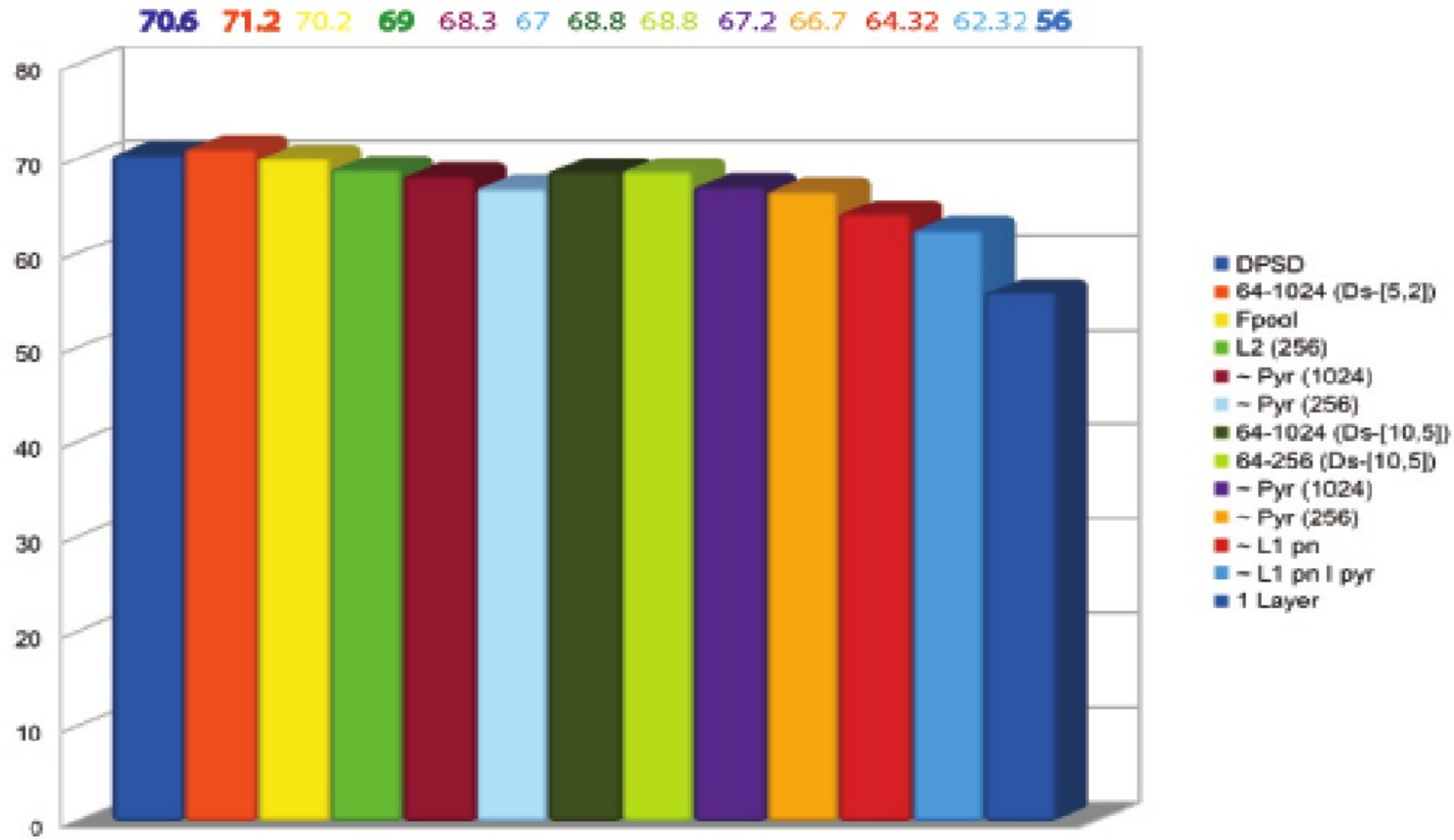
Using a few more tricks...

- Pyramid pooling on last layer: 1% improvement over regular pooling
- Shrinkage non-linearity + lateral inhibition: 1.6% improvement over tanh
- Discriminative term in sparse coding: 2.8% improvement

Architecture	Protocol	%
(1) $F_{\tanh} - R_{abs} - N - P_A^{pyr}$	$\mathbf{R^+R^+}$	65.4 ± 1.0
(2) $F_{\tanh} - R_{abs} - N - P_A^{pyr}$	$\mathbf{U^+U^+}$	66.2 ± 1.0
(3) $F_{si} - R_{abs} - N - P_A$	$\mathbf{R^+R^+}$	63.3 ± 1.0
(4) $F_{si} - R_{abs} - N - P_A$	$\mathbf{UU}$	60.4 ± 0.6
(5) $F_{si} - R_{abs} - N - P_A$	$\mathbf{U^+U^+}$	66.4 ± 0.5
(6) $F_{si} - R_{abs} - N - P_A^{pyr}$	$\mathbf{U^+U^+}$	67.8 ± 0.4
(7) $F_{si} - R_{abs} - N - P_A$	$\mathbf{DD}$	66.0 ± 0.3
(8) $F_{si} - R_{abs} - N - P_A$	$\mathbf{D^+D^+}$	68.7 ± 0.2
(9) $F_{si} - R_{abs} - N - P_A^{pyr}$	$\mathbf{D^+D^+}$	$\mathbf{70.6 \pm 0.3}$

Results on Caltech101: purely supervised with soft-shrink, L2 pooling, contrast normalization

- Supervised learning with soft-shrinkage non-linearity, L2 complex cells, and sparsity penalty on the complex cell outputs: 71%



What does Local Contrast Normalization Do?

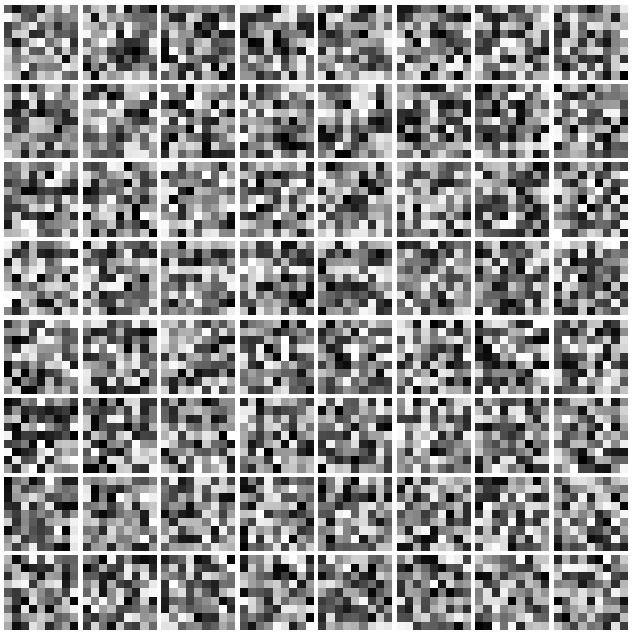
Original

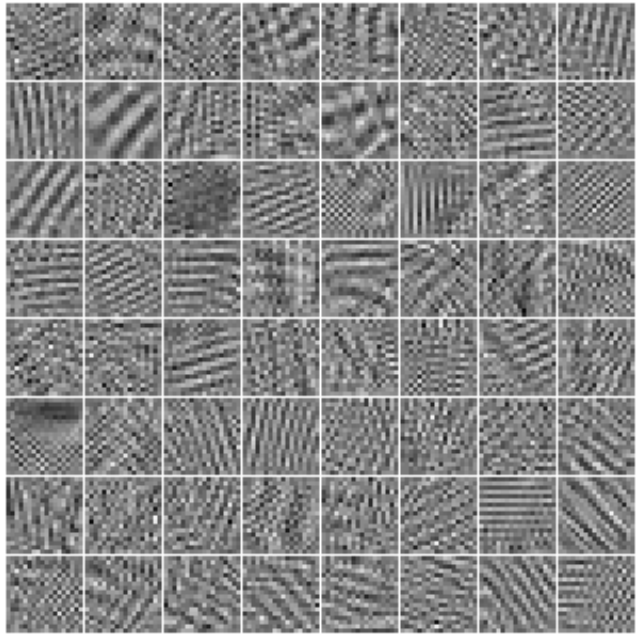
Reconstuction
With LCN

Reconstruction
Without LCN

Why Do Random Filters Work?

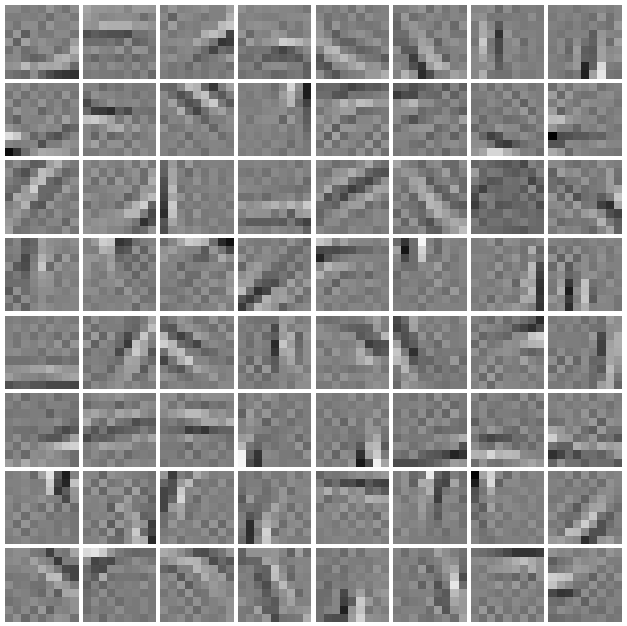
Random Filters For Simple Cells

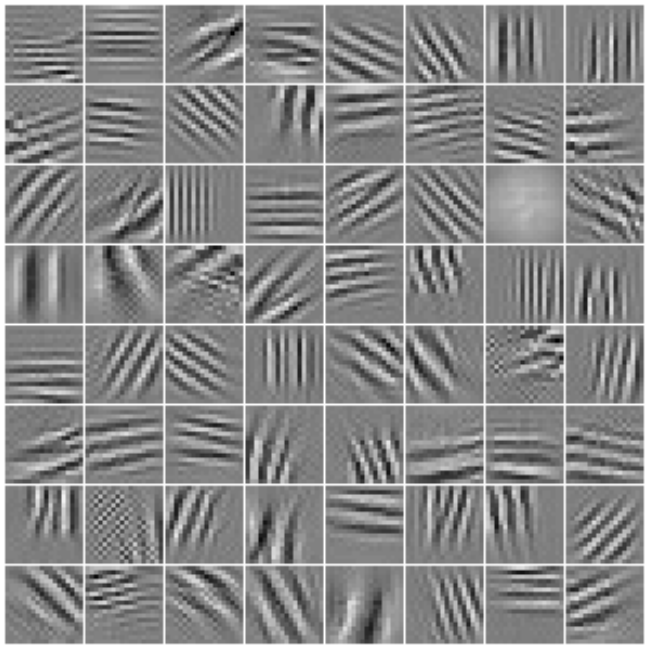




Optimal Stimuli for each Complex Cell

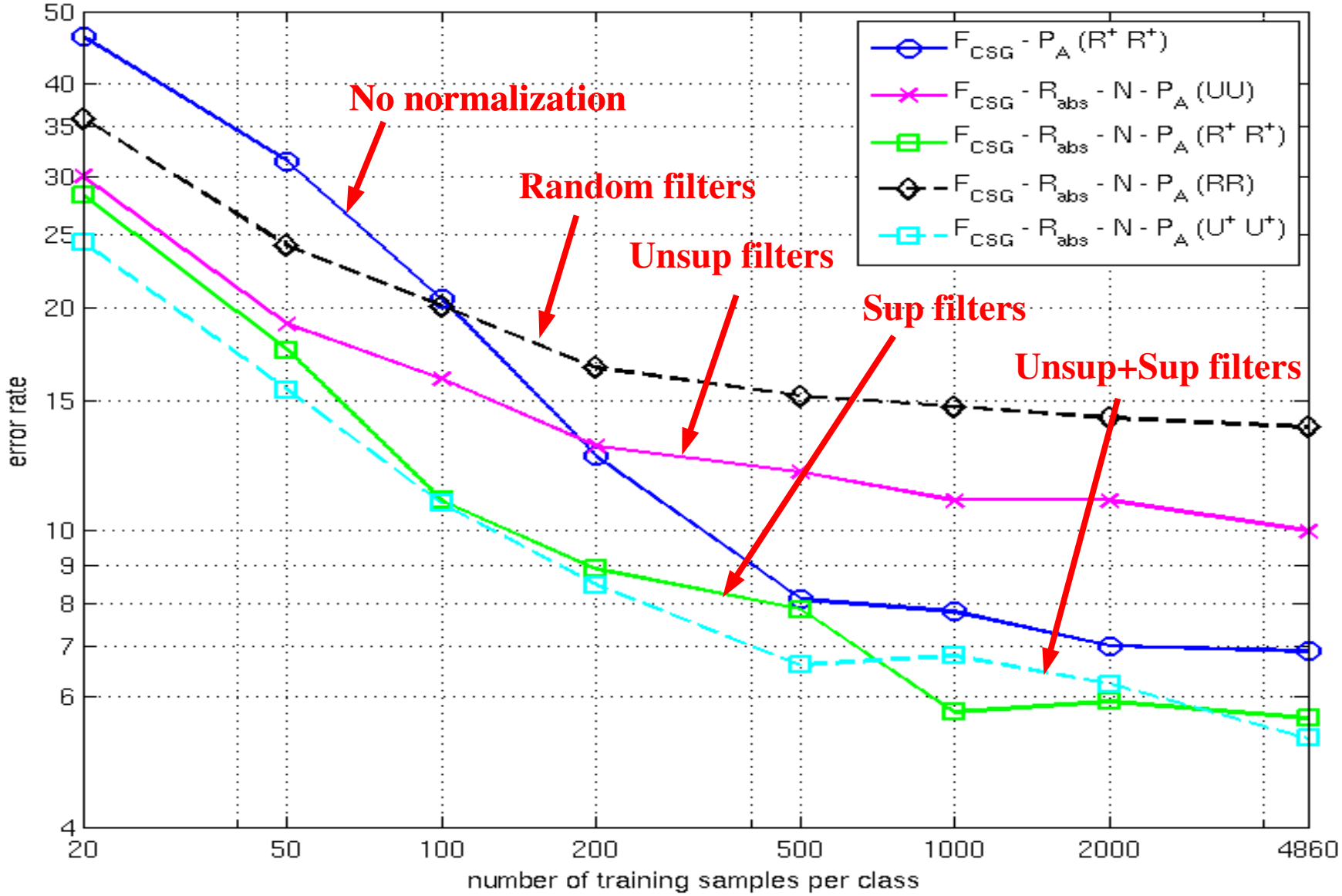
Trained Filters For Simple Cells





Small NORB dataset

- Two-stage system: error rate versus number of labeled training samples



Convolutional Sparse Coding
Convolutional PSD

[Kavukcuoglu, Sermanet, Boureau, Mathieu, LeCun. NIPS 2010]: convolutional PSD

[Zeiler, Krishnan, Taylor, Fergus, CVPR 2010]: Deconvolutional Network

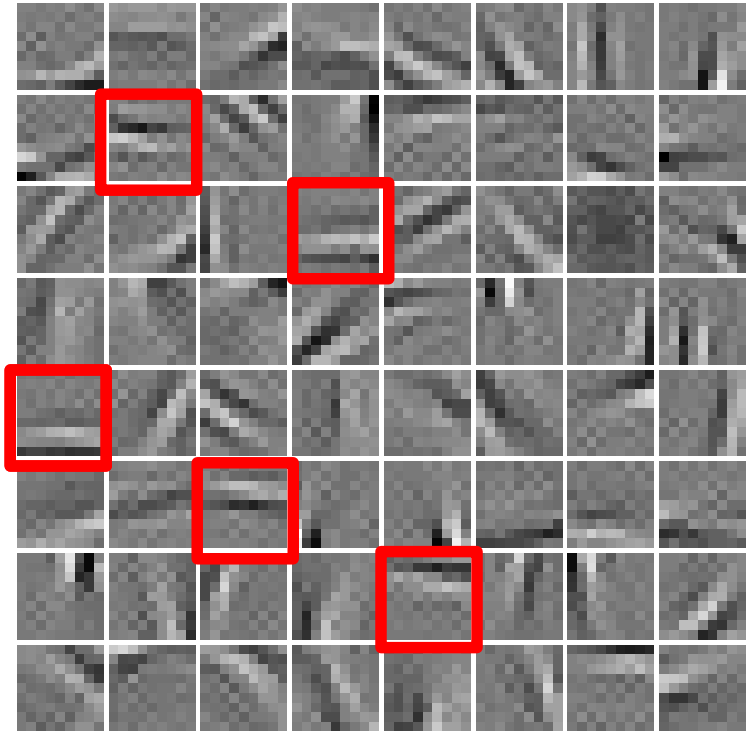
[Lee, Gross, Ranganath, Ng, ICML 2009]: Convolutional Boltzmann Machine

[Norouzi, Ranjbar, Mori, CVPR 2009]: Convolutional Boltzmann Machine

[Chen, Sapiro, Dunson, Carin, Preprint 2010]: Deconvolutional Network with automatic adjustment of code dimension.

Convolutional Training

- **Problem:**
  - With patch-level training, the learning algorithm must reconstruct the entire patch with a single feature vector
  - But when the filters are used convolutionally, neighboring feature vectors will be highly redundant

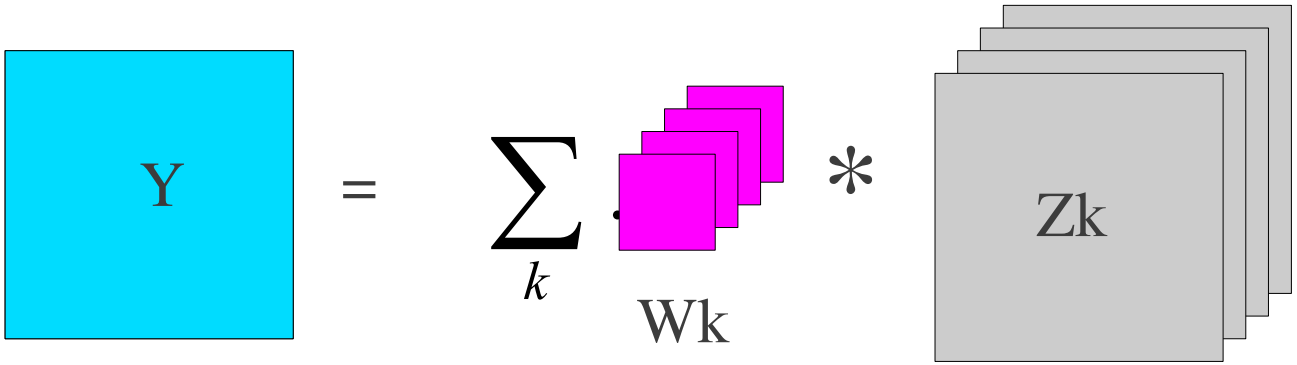


weights :-0.2828 - 0.3043

Patch-level training produces lots of filters that are shifted versions of each other.

Convolutional Sparse Coding

- **Replace the dot products with dictionary element by convolutions.**
  - Input Y is a full image
  - Each code component Zk is a feature map (an image)
  - Each dictionary element is a convolution kernel
- **Regular sparse coding** $$E(Y,Z) = ||Y - \sum_k W_k Z_k||^2 + \alpha \sum_k |Z_k|$$
- **Convolutional S.C.** $$E(Y,Z) = ||Y - \sum_k W_k * Z_k||^2 + \alpha \sum_k |Z_k|$$

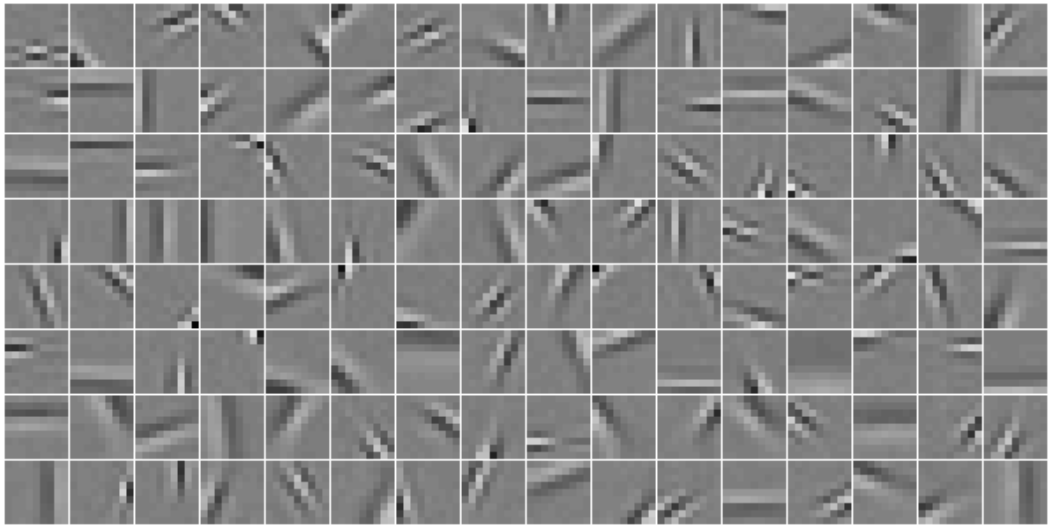


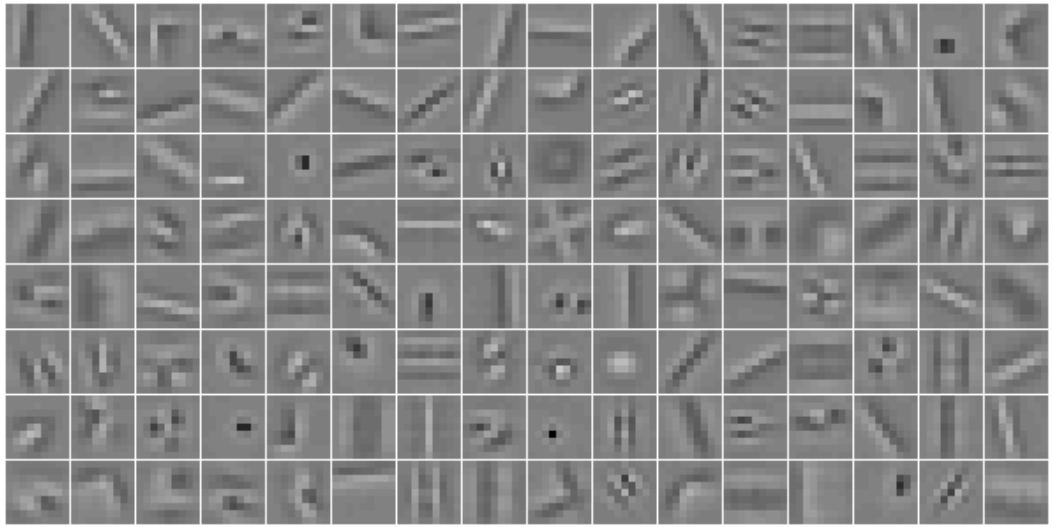
"deconvolutional networks" [Zeiler, Taylor, Fergus CVPR 2010]

Convolutional PSD: Encoder with a soft sh() Function

- Convolutional Formulation
  - Extend sparse coding from ***PATCH*** to ***IMAGE***

$$\mathcal{L}(x, z, \mathcal{D}) = \frac{1}{2}||x - \sum_{k=1}^{K} \mathcal{D}_k * z_k||_2^2 + \sum_{k=1}^{K} ||z_k - f(W^k * x)||_2^2 + |z|_1$$





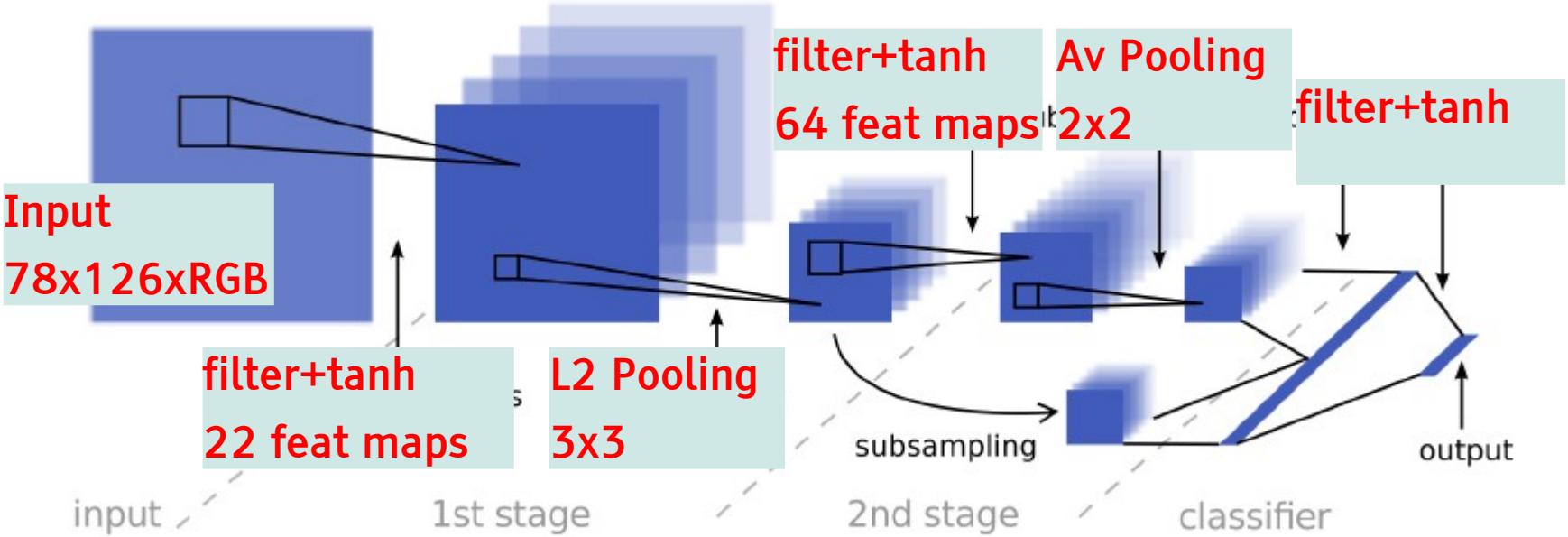
- ***PATCH*** based learning
- ***CONVOLUTIONAL*** learning

Pedestrian Detection, Face Detection

[Osadchy,Miller LeCun JMLR 2007],[Kavukcuoglu et al. NIPS 2010]

ConvNet Architecture with Multi-Stage Features

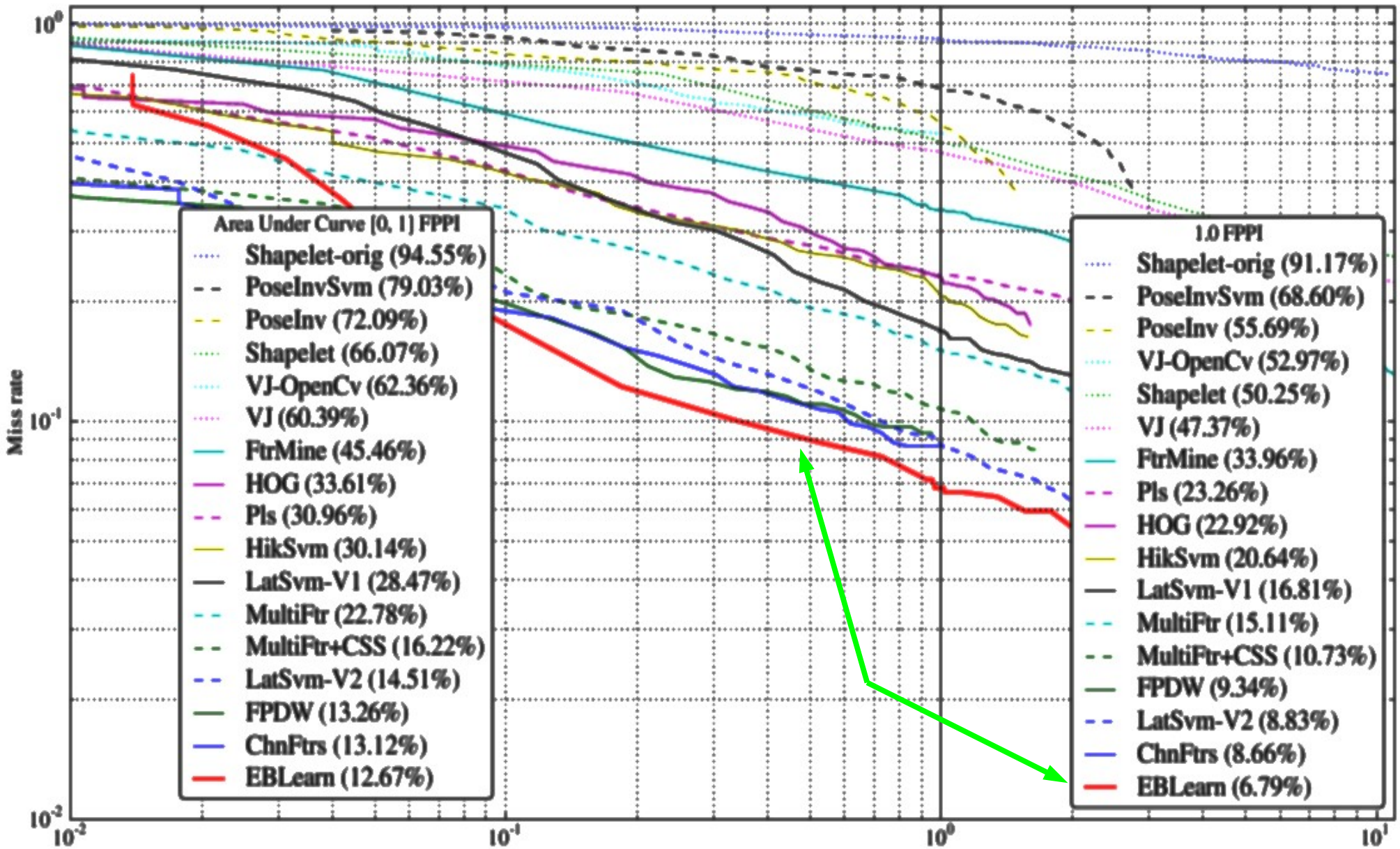
- Feature maps from all stages are pooled/subsampled and sent to the final classification layers
  - Pooled low-level features: good for textures and local motifs
  - High-level features: good for "gestalt" and global shape



Task	Single-Stage features	Multi-Stage features	Improvement %
Pedestrians detection (INRIA) [9]	14.26%	9.85%	31%
Traffic Signs classification (GTSRB) [11]	1.80%	0.83%	54%
House Numbers classification (SVHN)	5.72%	5.67%	0.9%

[Sermanet, Chintala, LeCun ArXiv:1204.3968, 2012]

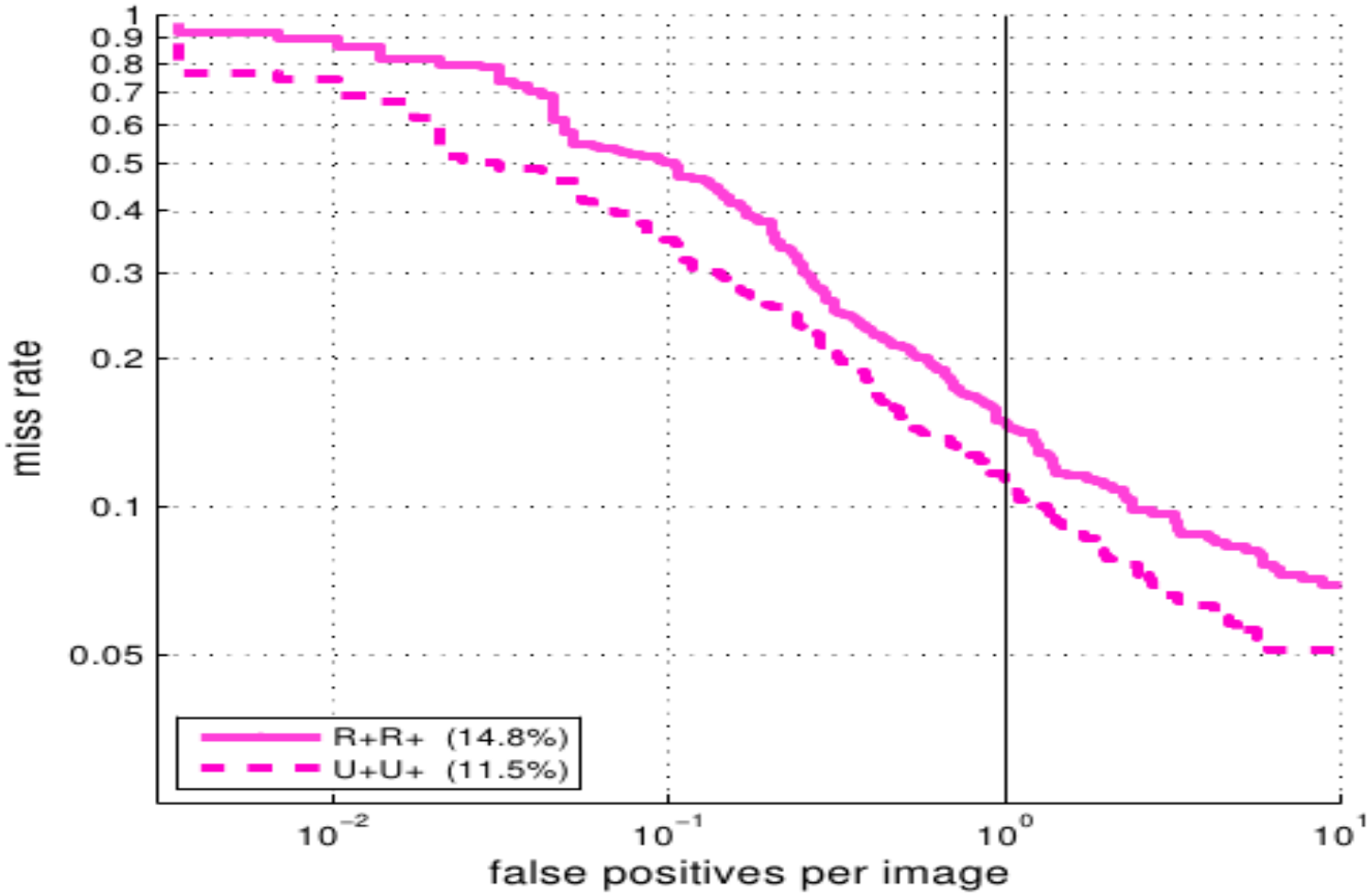
Pedestrian Detection (INRIA Dataset)



[Kavukcuoglu et al. NIPS 2010]

Convolutional PSD pre-training for pedestrian detection

- ConvPSD pre-training improves the accuracy of pedestrian detection over purely supervised training from random initial conditions.

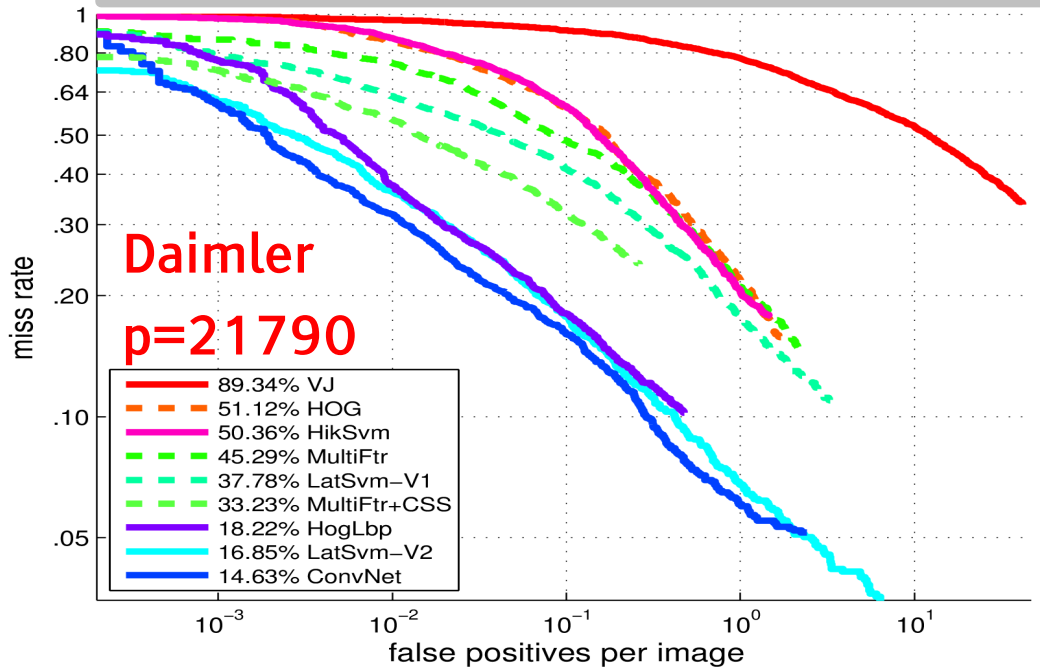


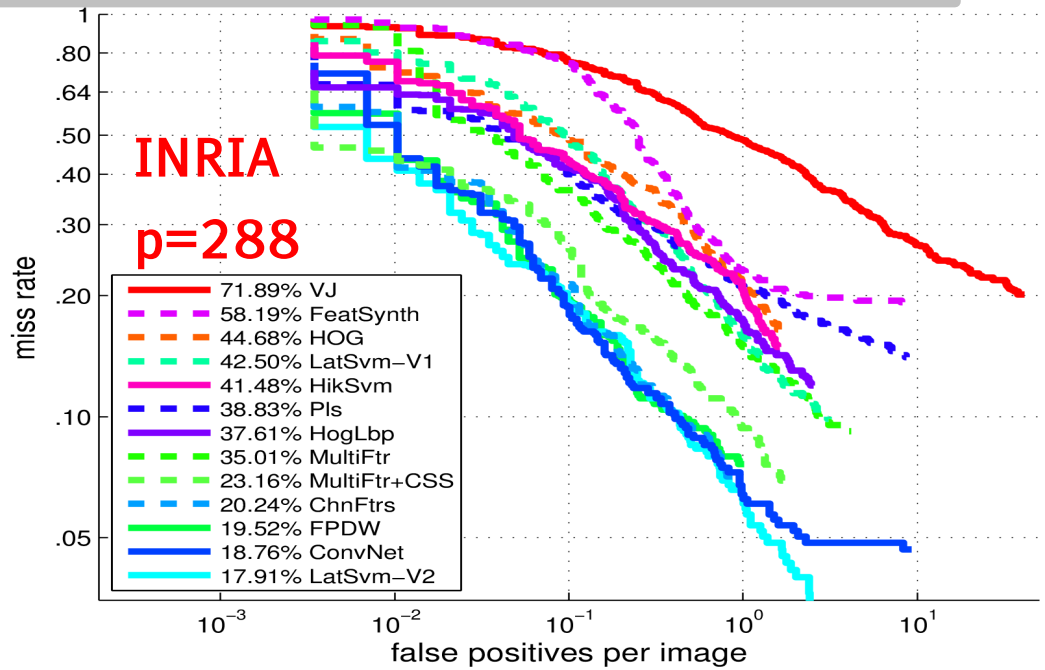
Convolutional PSD pre-training for pedestrian detection

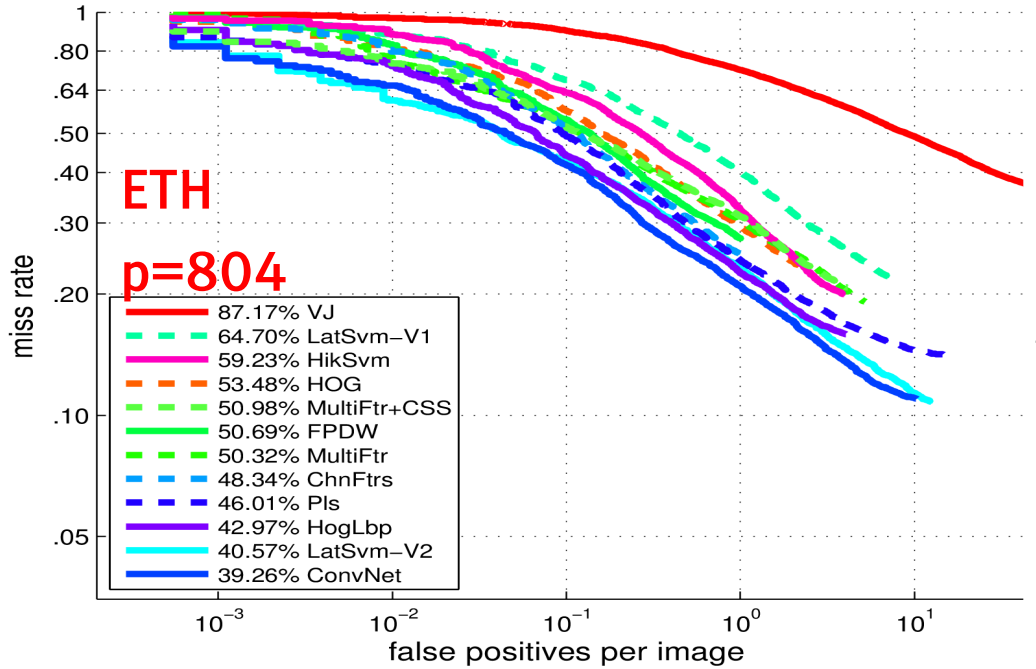
- Trained on INRIA. Tested on INRIA, Daimler, TUD-Brussles, ETH
- Same testing protocol as in [Dollar et al. T.PAMI 2011]

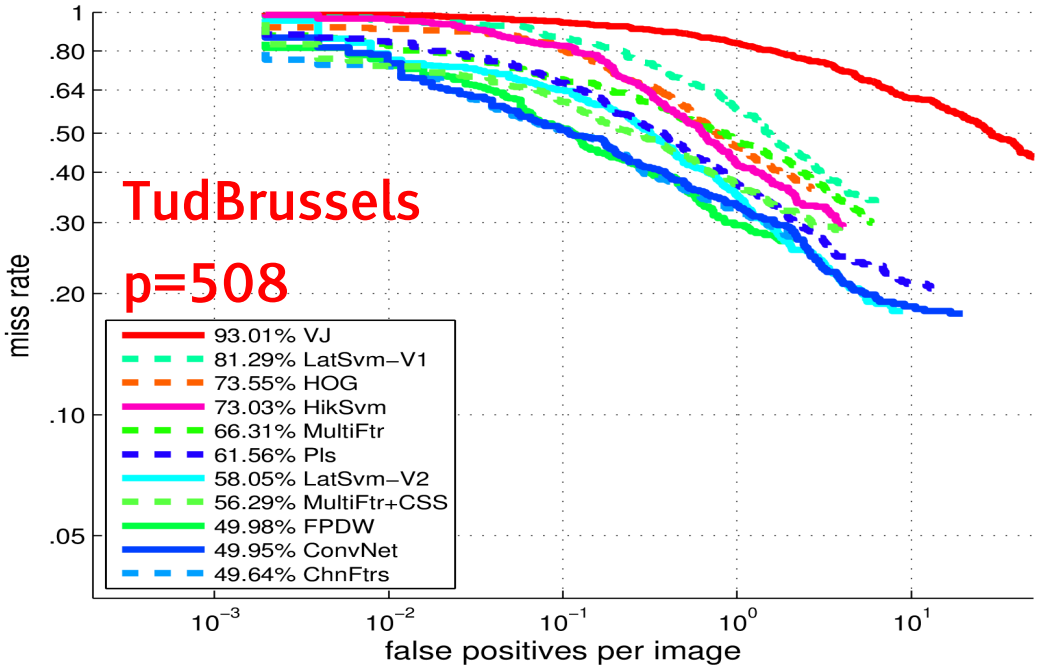
	INRIA (all positives labeled)		INRIA (missing positive labels)	Caltech	Daimler	TUD	ETH
# test images	288		288	4024	21790	508	1804
Error Percentage type	1FPPI	AUC	AUC	AUC	AUC	AUC	AUC
Reasonable: at least 50 pixels tall + no or partial occlusion							
ConvNet's rank			**1st**	12th	**1st**	4th	**1st**
ConvNet			**19.78 %**	77.2 %	**32.66 %**	68.81 %	**50.31 %**
LatSvm-V2			19.96 %	63.26 %	37.9 %	69.59 %	50.89 %
ChnFtrs			22.18 %	**56.34 %**	-	60.33 %	57.47 %
MultiFtr+CSS			24.74 %	60.89 %	42.25 %	**59.49 %**	60.74 %
FPDW			21.47 %	57.40 %	-	63.03 %	60.10 %
Near: at least 80 pixels tall							
ConvNet's rank	**1st**	**1st**	2nd	4th	**1st**	2nd	**1st**
ConvNet	**5.77 %**	**11.26 %**	18.76 %	36.68 %	**14.63 %**	49.95 %	**39.26 %**
LatSvm-V2	8.83 %	14.51 %	**17.91 %**	34.27 %	16.85 %	58.05 %	40.57 %
ChnFtrs	8.66 %	13.12 %	20.24 %	35.12 %	-	**49.64 %**	48.34 %
HogLpb	-	-	37.61 %	**30.79 %**	18.22 %	-	42.97 %
Large: at least 100 pixels tall							
ConvNet's rank			2nd	**1st**	2nd	2nd	**1st**
ConvNet			17.27 %	**21.37 %**	11.81 %	**41.15 %**	**34.56 %**
LatSvm-V2			**16.31 %**	28.23 %	**11.52 %**	54.16 %	35.27 %
HogLpb			37.71 %	22.67 %	20.2 %	-	40.84 %

Results on "Near Scale" Images (>80 pixels tall, no occlusions)

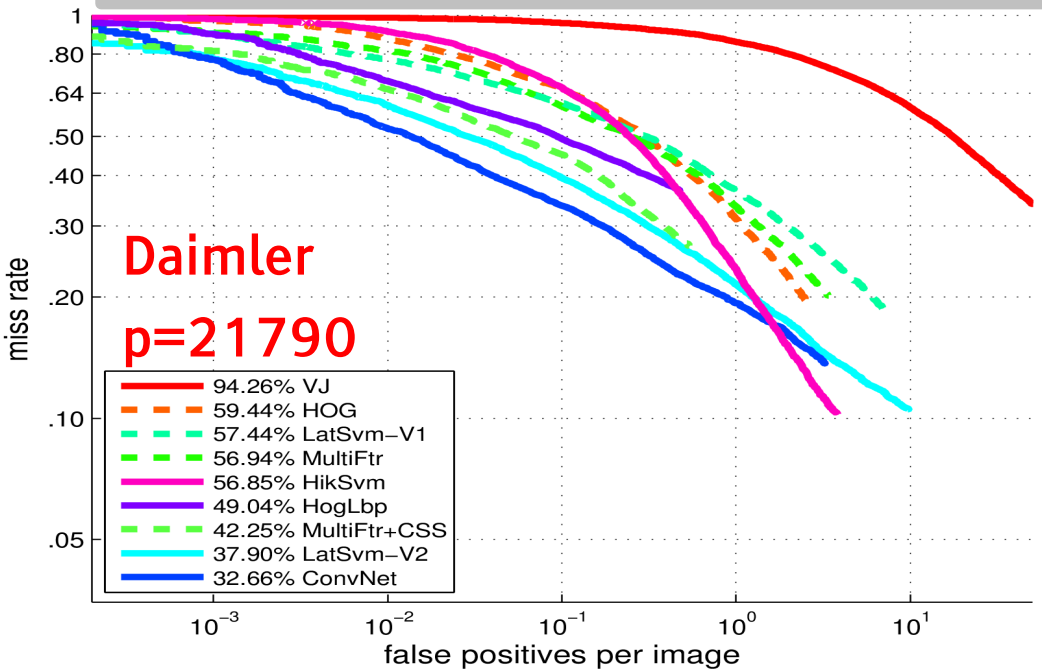


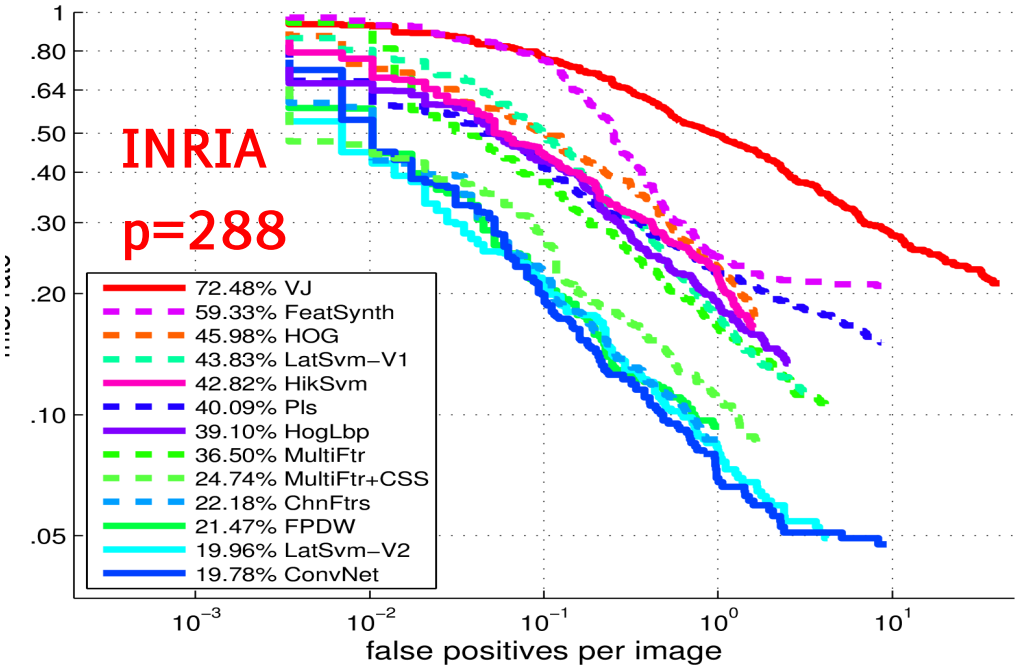


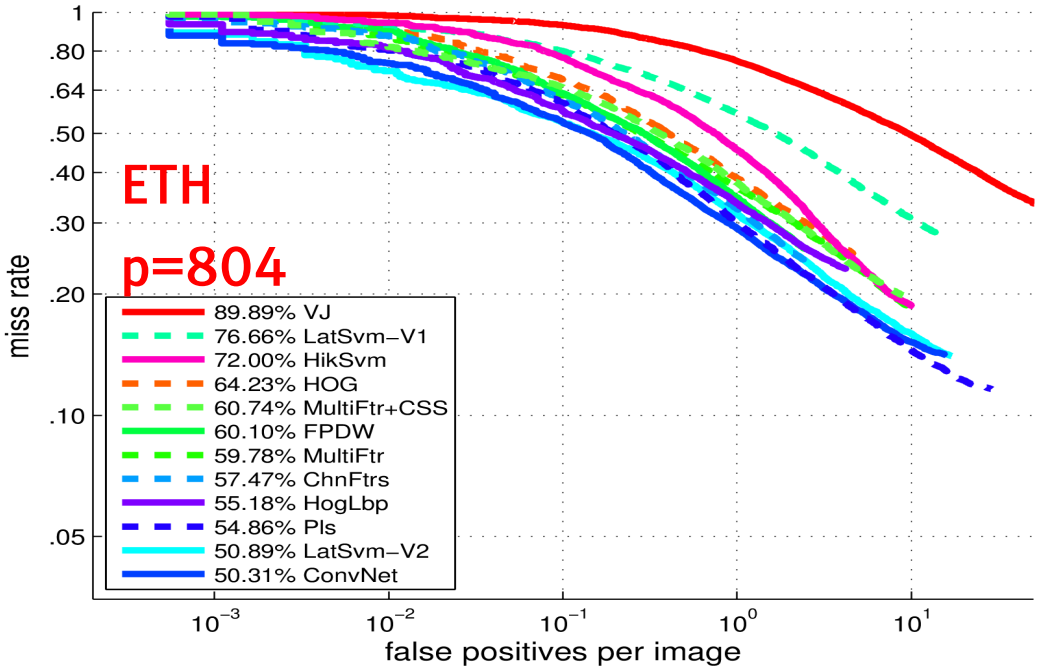


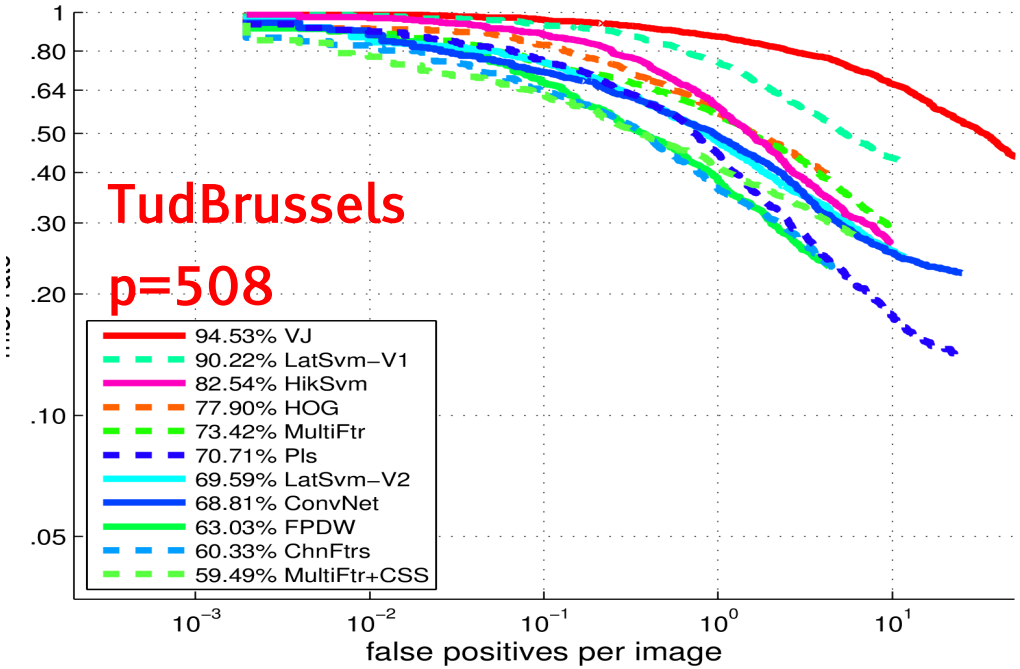


Results on "Reasonable" Images (>50 pixels tall, few occlusions)









Musical Genre Recognition

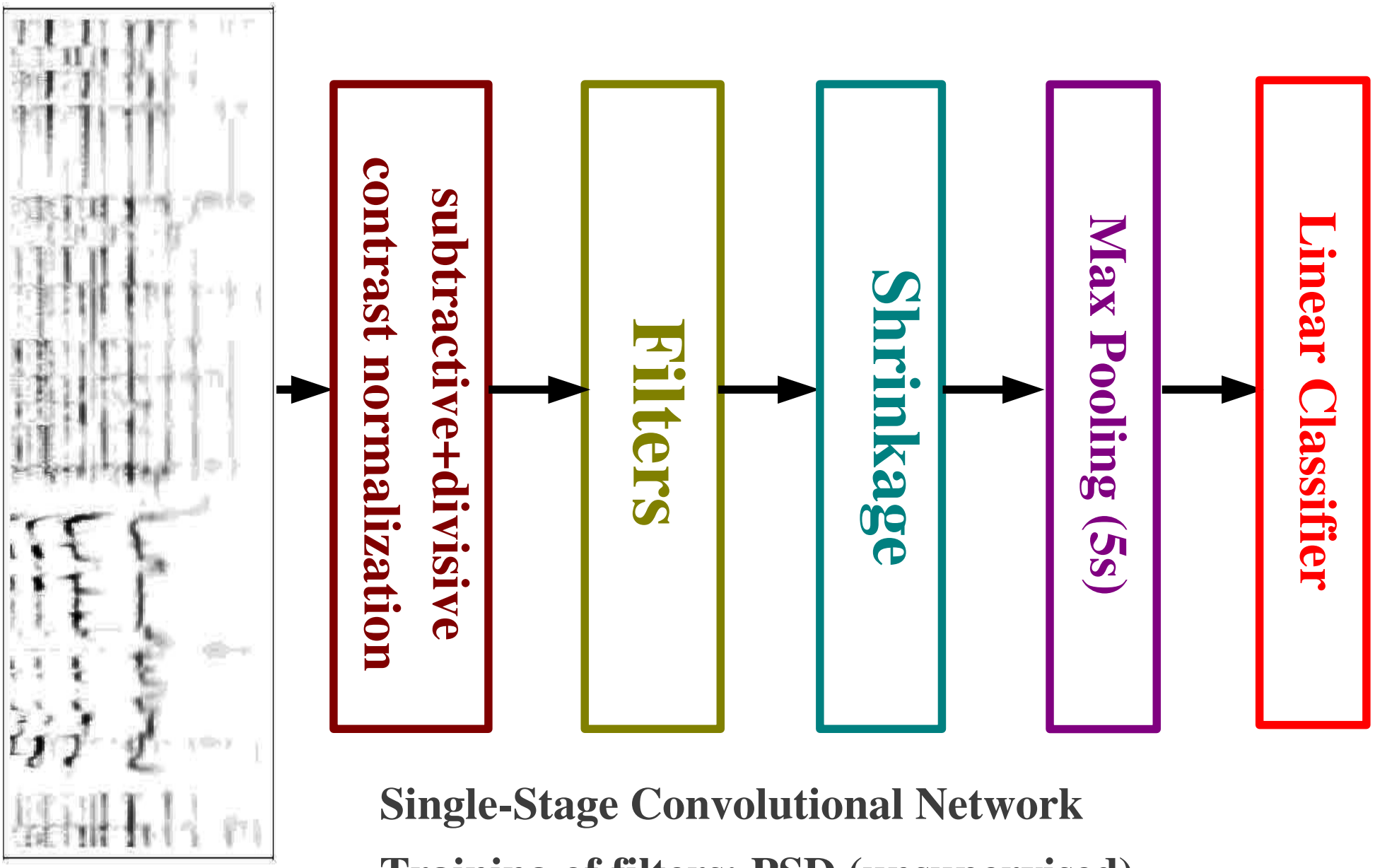
Same Architecture, Different Data

[Henaff et al. ISMIR 2011]

Convolutional PSD Features on Time-Frequency Signals

- **Input: "Constant Q Transform" over 46.4ms windows (1024 samples)**
  - 96 filters, with frequencies spaced every quarter tone (4 octaves)
- **Architecture:**
  - Input: sequence of contrast-normalized CQT vectors
  - 1: PSD features, 512 trained filters
  - 2: shrinkage function → rectification
  - 3: pooling over 5 seconds
  - 4: linear SVM classifier
  - 5: pooling of SVM categories over 30 seconds
- **GTZAN Dataset**
  - 1000 clips, 30 second each
  - 10 genres: blues, classical, country, disco, hiphop, jazz, metal, pop, reggae and rock.
- **Results**
  - 84% correct classification
  - (state of the art is at 92% with many features)

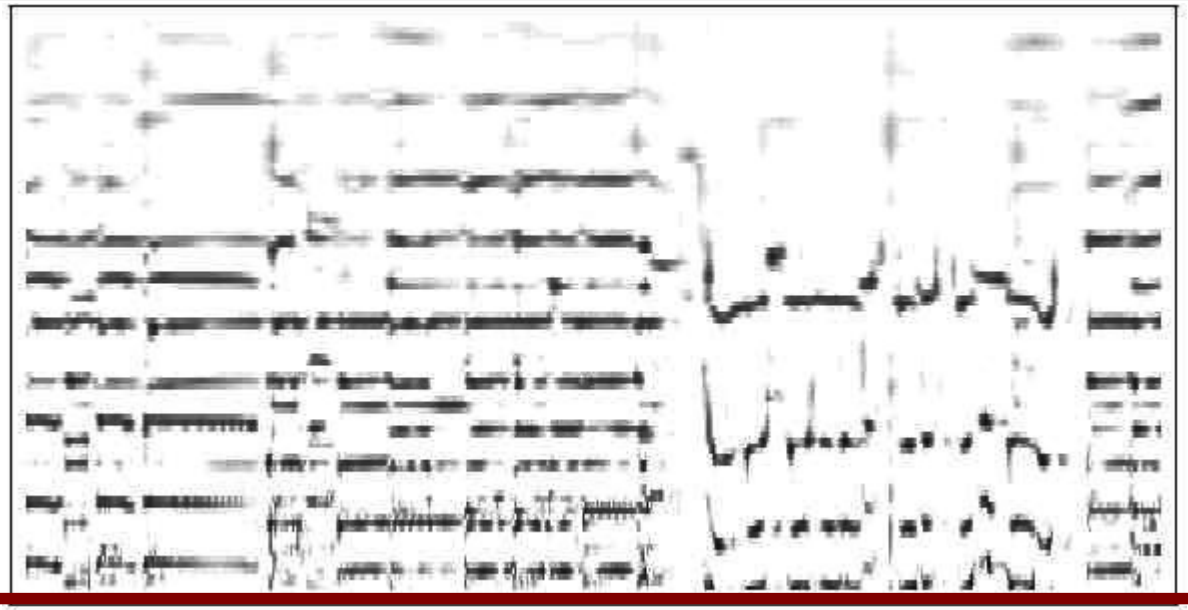
Architecture: contrast norm → filters → shrink → max pooling



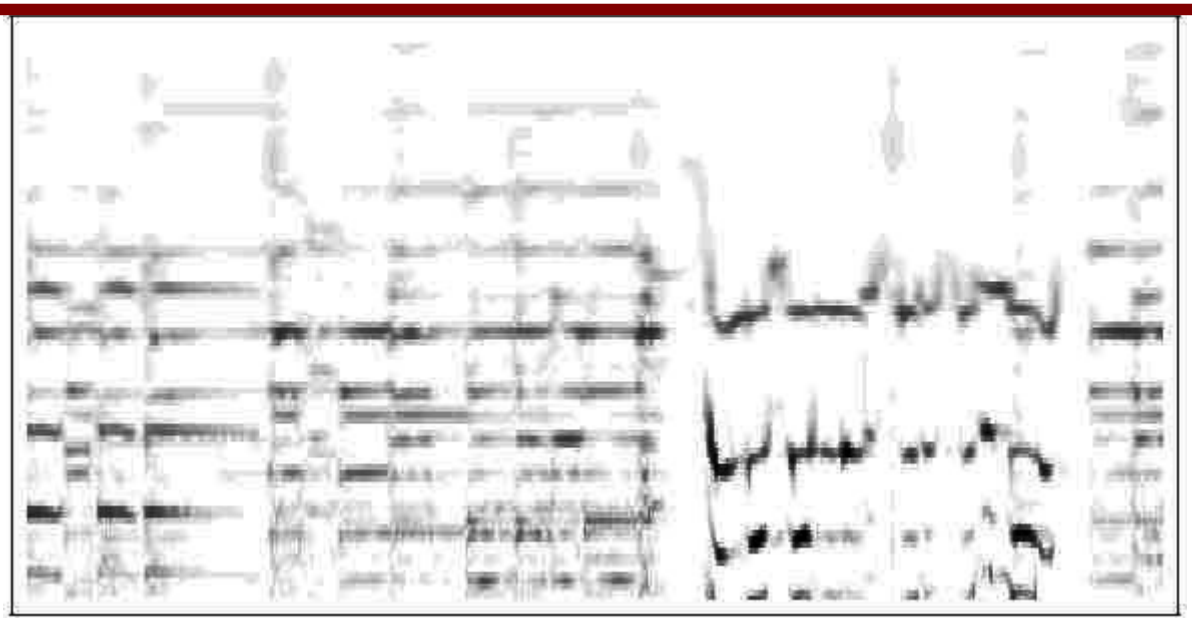
Single-Stage Convolutional Network

Training of filters: PSD (unsupervised)

Constant Q Transform over 46.4 ms → Contrast Normalization

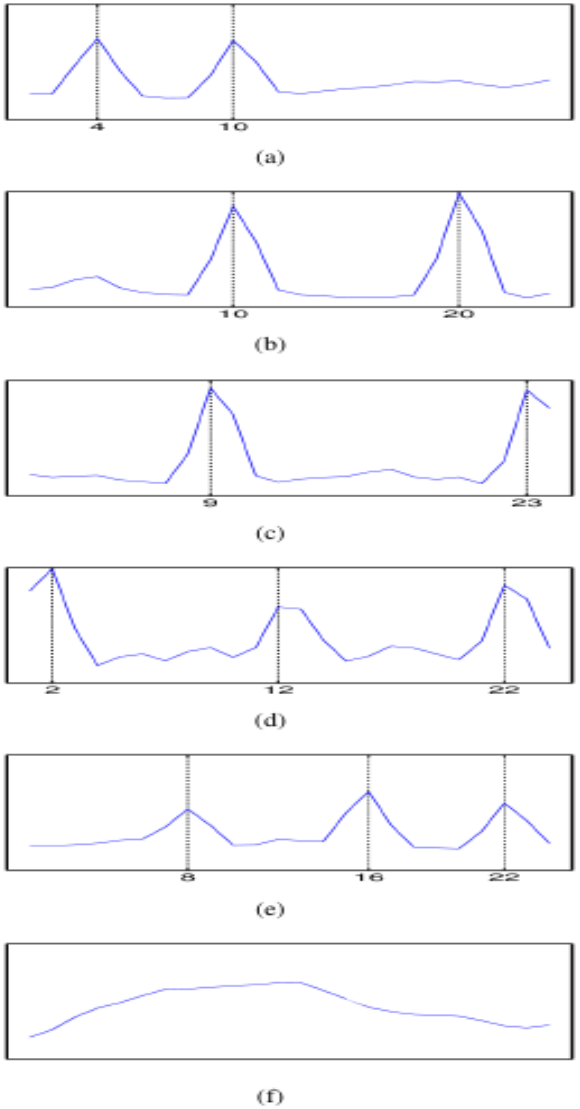


subtractive+divisive contrast normalization



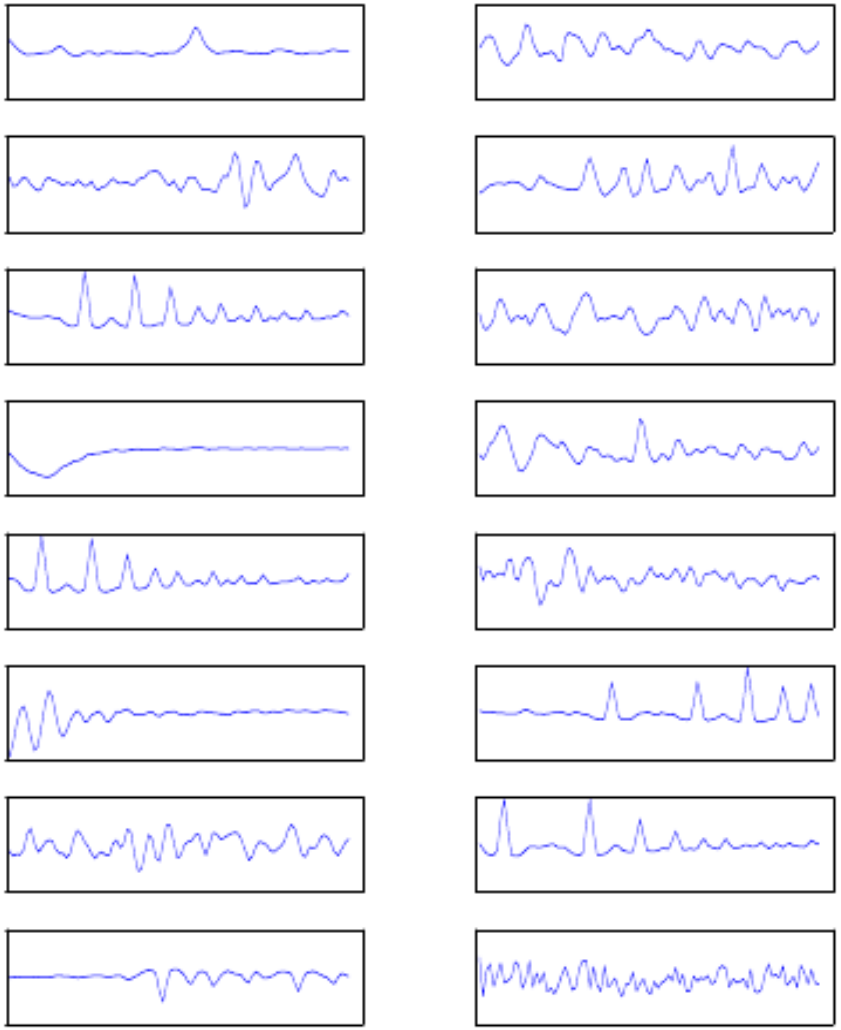
Convolutional PSD Features on Time-Frequency Signals

- Octave-wide features



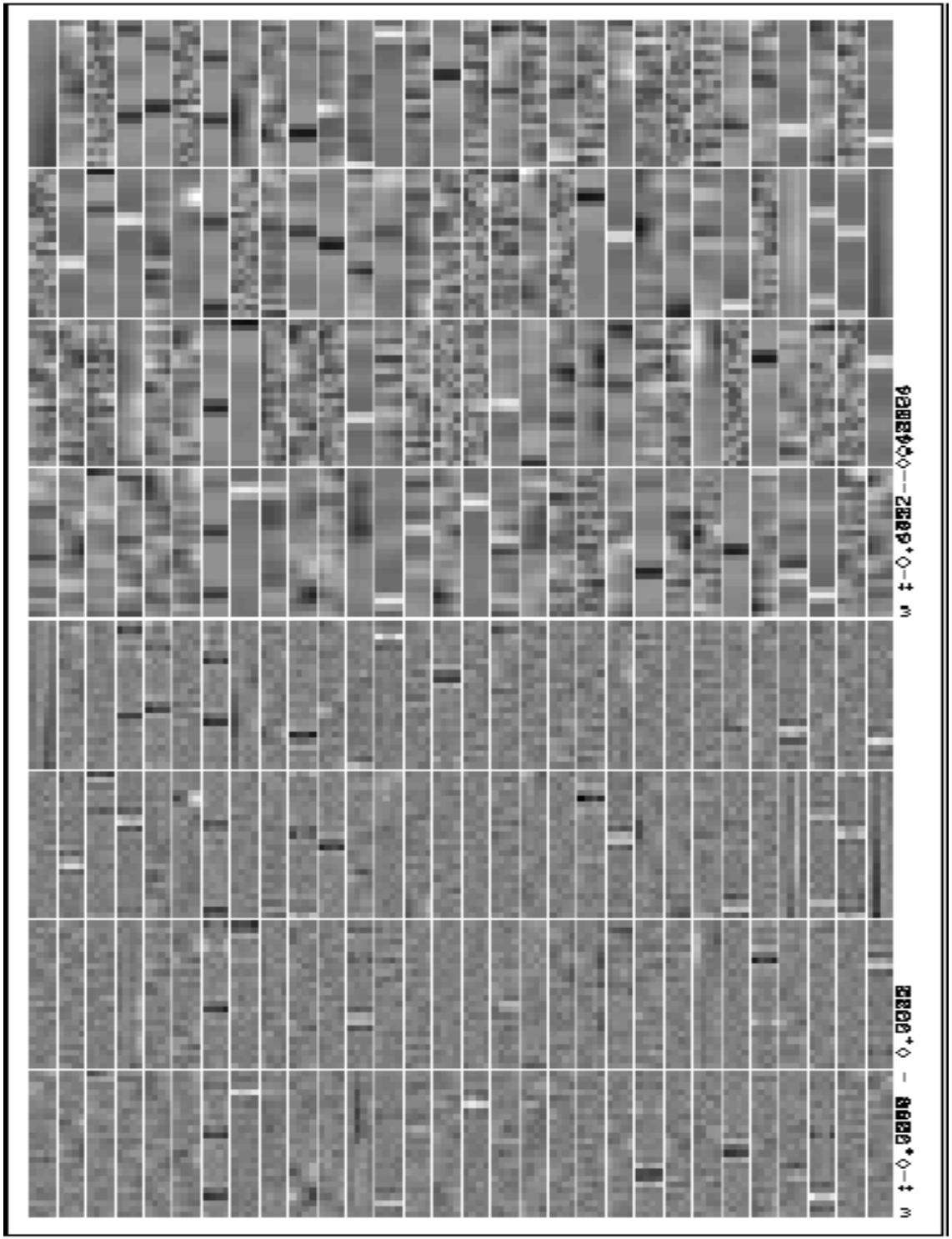
- Minor 3rd
- Perfect 4th
- Perfect 5th
- Quartal chord
- Major triad
- transient

full 4-octave features



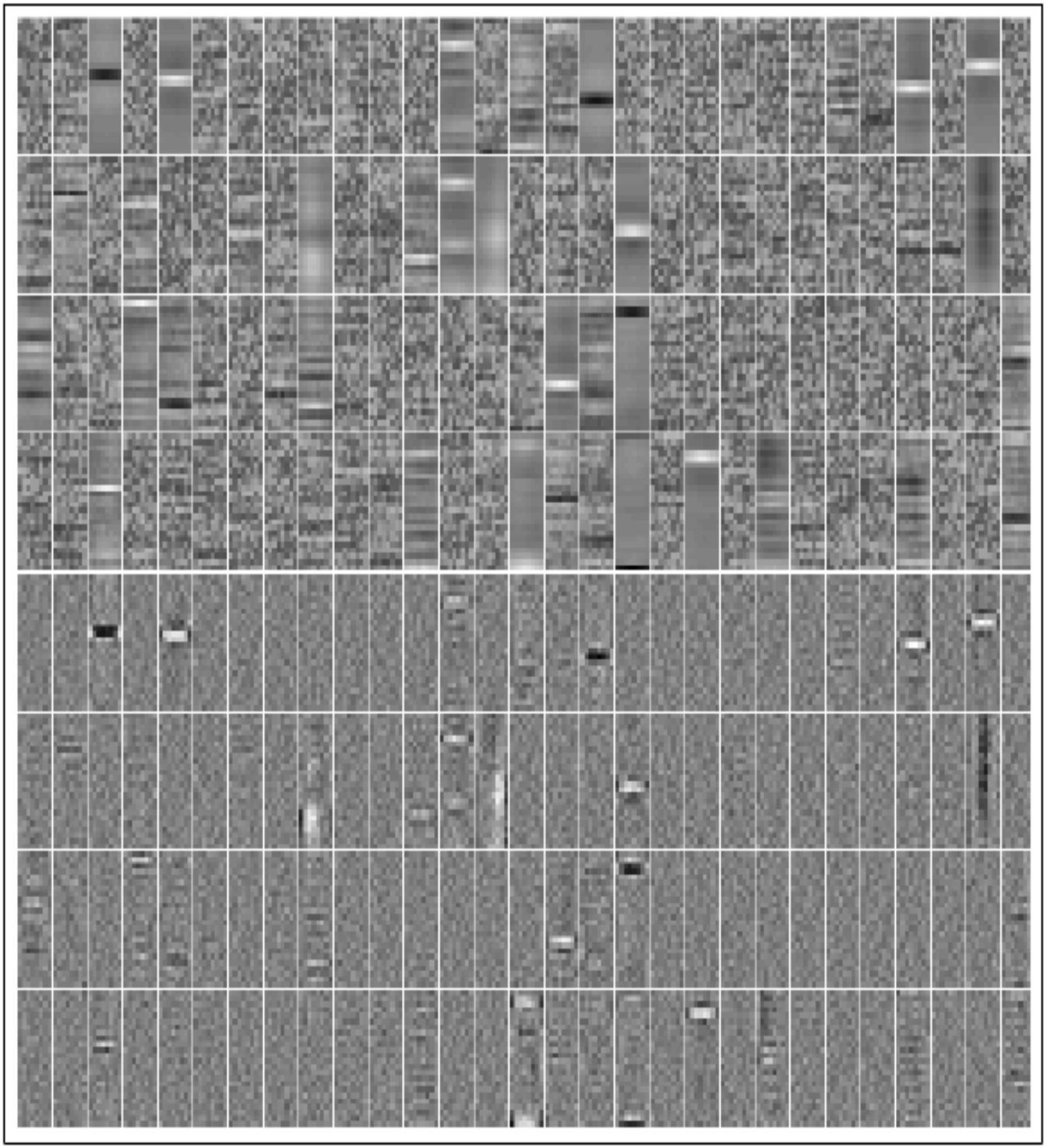
PSD Features on Constant-Q Transform

- Octave-wide features
  - Encoder basis functions
  - Decoder basis functions



Time-Frequency Features

- Octave-wide features on 8 successive acoustic vectors
  - Almost no temporal structure in the filters!



Accuracy on GTZAN dataset (small, old, etc...)

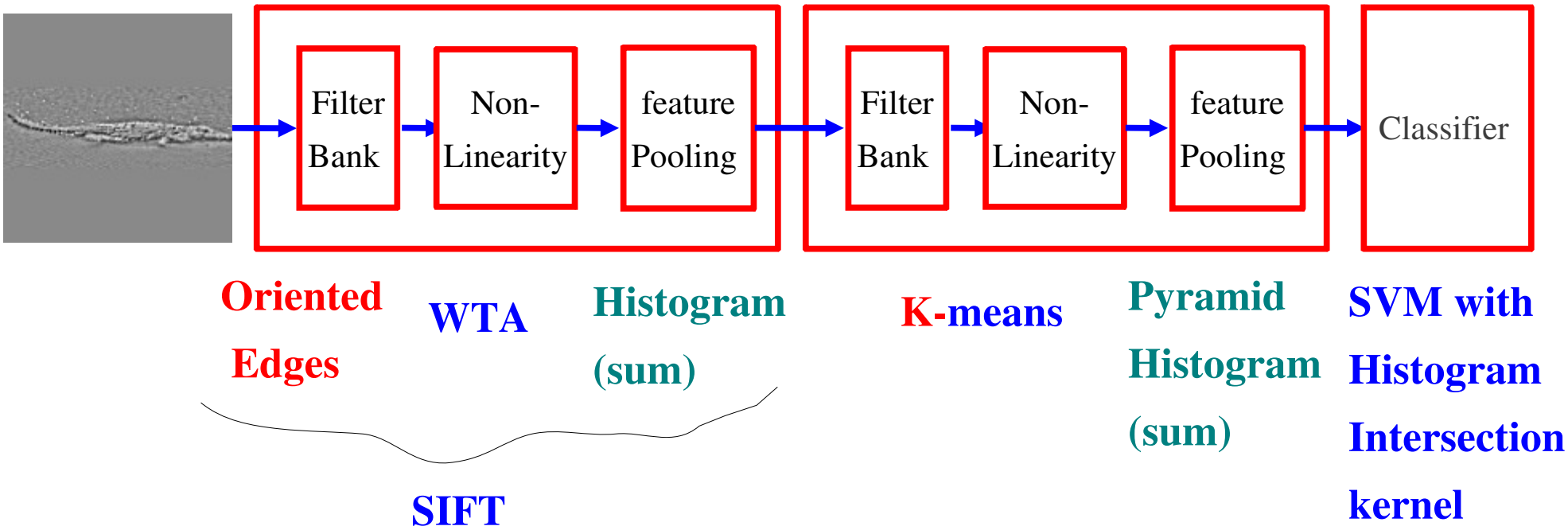
- Accuracy: 83.4%. State of the Art: 84.3%
- Very fast

Classifier	Features	Acc. (%)
CSC	Many features [6]	92.7
SRC	Auditory cortical feat. [25]	92
RBF-SVM	Learned using DBN [12]	84.3
Linear SVM	Learned using PSD on octaves	$\mathbf{83.4 \pm 3.1}$
AdaBoost	Many features [2]	83
Linear SVM	Learned using PSD on frames	$\mathbf{79.4 \pm 2.8}$
SVM	Daubechies Wavelets [19]	78.5
Log. Reg.	Spectral Covariance [3]	77
LDA	MFCC + other [18]	71
Linear SVM	Auditory cortical feat. [25]	70
GMM	MFCC + other [29]	61

Learning Mid-Level Features

[Boureau et al., CVPR 2010, ICML 2010, CVPR 2011]

ConvNets and "Conventional" Vision Architectures are Similar

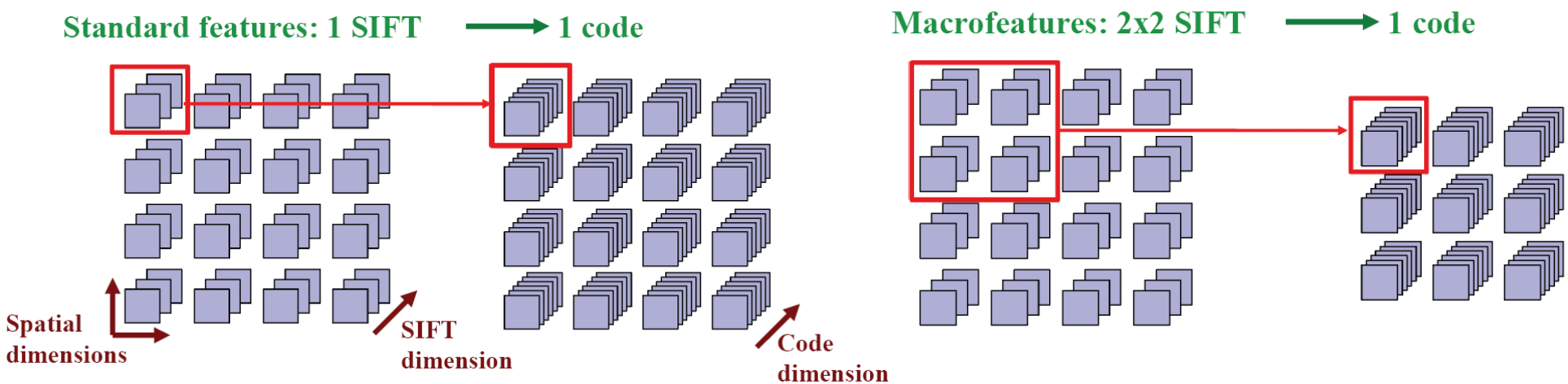


- Can't we use the same tricks as ConvNets to train the second stage of a "conventional vision architecture?
- Stage 1: SIFT
- Stage 2: sparse coding over neighborhoods + pooling

Using DL/ConvNet ideas in "conventional" recognition systems

[Boureau et al. CVPR 2010]

- **Adapting insights from ConvNets:**
  - Jointly encoding spatial neighborhoods instead of single points: increase spatial receptive fields for higher-level features



  - Use max pooling instead of average pooling
  - Train supervised dictionary for sparse coding

- **This yields state-of-the-art results:**
  - **75.7%** on Caltech-101 (+/-1.1%): record for single system
  - **85.6%** on 15-Scenes (+/- 0.2): record!

The Competition: SIFT + Sparse-Coding + PMK-SVM

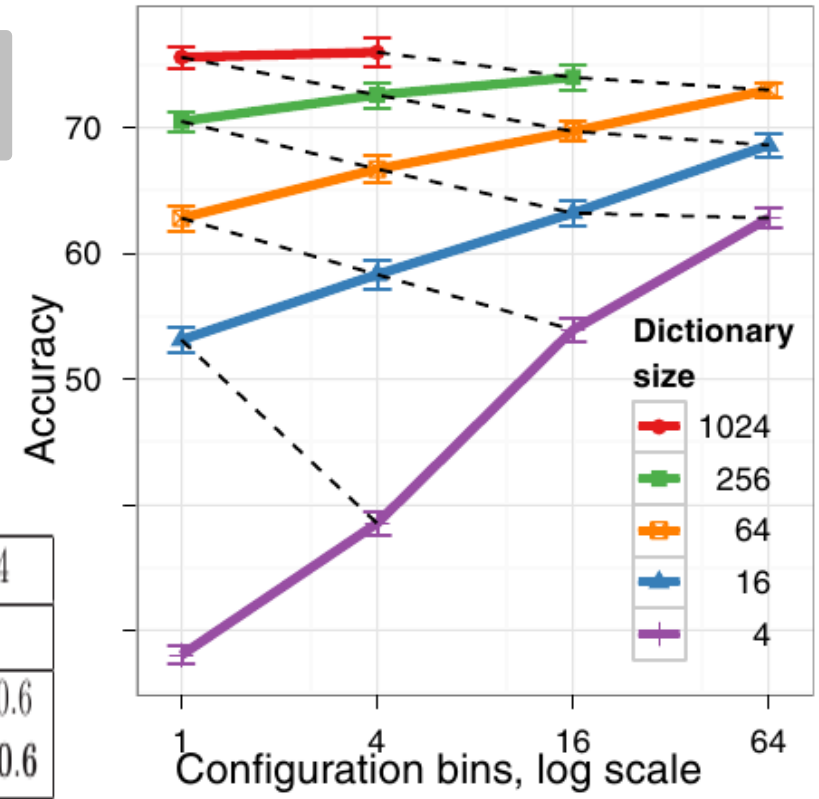
- **Replacing K-means with Sparse Coding**
  - [Yang 2008] [Boureau, Bach, Ponce, LeCun 2010]

	Method	Caltech 15	Caltech 30	Scenes
Boiman et al. [1]	Nearest neighbor + spatial correspondence	65.00 ± 1.14	70.40	-
Jain et al. [8]	Fast image search for learned metrics	61.00	69.60	-
Lazebnik et al. [12]	Spatial Pyramid + hard quantization + kernel SVM	56.40	64.40 ± 0.80	81.40 ± 0.50
van Gemert et al. [24]	Spatial Pyramid + soft quantization + kernel SVM	–	64.14 ± 1.18	76.67 ± 0.39
Yang et al. [26]	SP + sparse codes + max pooling + linear	**67.00±0.45**	**73.2±0.54**	80.28 ± 0.93
Zhang et al. [27]	*k*NN-SVM	59.10 ± 0.60	66.20 ± 0.50	-
Zhou et al. [29]	SP + Gaussian mixture	–	–	**84.1 ± 0.5**
Baseline:	SP + hard quantization + avg pool + kernel SVM	56.74 ± 1.31	64.19 ± 0.94	80.89 ± 0.21
Unsupervised coding	SP + soft quantization + avg pool + kernel SVM	59.12 ± 1.51	66.42 ± 1.26	81.52 ± 0.54
1×1 features	SP + soft quantization + max pool + kernel SVM	63.61 ± 0.88	–	83.41 ± 0.57
8 pixel grid resolution	SP + sparse codes + avg pool + kernel SVM	62.85 ± 1.22	70.27 ± 1.29	83.15 ± 0.35
	SP + sparse codes + max pool + kernel SVM	64.62 ± 0.94	**71.81±0.96**	**84.25 ± 0.35**
	SP + sparse codes + max pool + linear	**64.71 ± 1.05**	71.52 ± 1.13	83.78 ± 0.53
Macrofeatures +	SP + sparse codes + max pool + kernel SVM	**69.03±1.17**	**75.72±1.06**	**84.60 ± 0.38**
Finer grid resolution	SP + sparse codes + max pool + linear	68.78 ± 1.09	75.14 ± 0.86	84.41 ± 0.26

Sparse Coding within Clusters

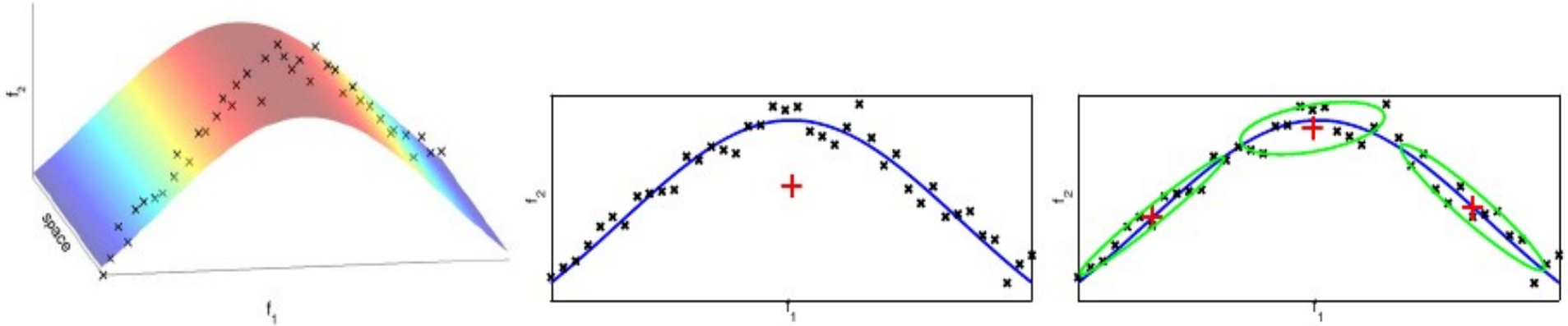
- **Splitting the Sparse Coding into Clusters**
  - only similar things get pooled together

[Boureau, et al. CVPR 2011]



p	1	4	16	64	1+4	1+16	1+64
Caltech-101							
$k = 256$	70.5 ± 0.8	72.6 ± 1.0	74.0 ± 1.0	75.0 ± 0.8	72.5 ± 1.0	74.2 ± 1.1	75.6 ± 0.6
$k = 1024$	75.6 ± 0.9	76.0 ± 1.2	76.3 ± 1.1	76.2 ± 0.8	76.3 ± 1.2	$\mathbf{76.9 \pm 1.0}$	$\mathbf{77.3 \pm 0.6}$
Scenes							
$k = 256$	78.8 ± 0.6	80.9 ± 0.7	81.5 ± 0.8	81.1 ± 0.5	80.8 ± 0.8	81.5 ± 0.8	81.9 ± 0.7
$k = 1024$	82.7 ± 0.7	83.0 ± 0.7	82.7 ± 0.9	81.4 ± 0.7	83.3 ± 0.8	83.3 ± 1.0	83.1 ± 0.7

Results on Caltech 101 (30 training examples per class), and 15-Scenes (100 training examples per class) as a func



Learning Invariant Features (learning complex cells)

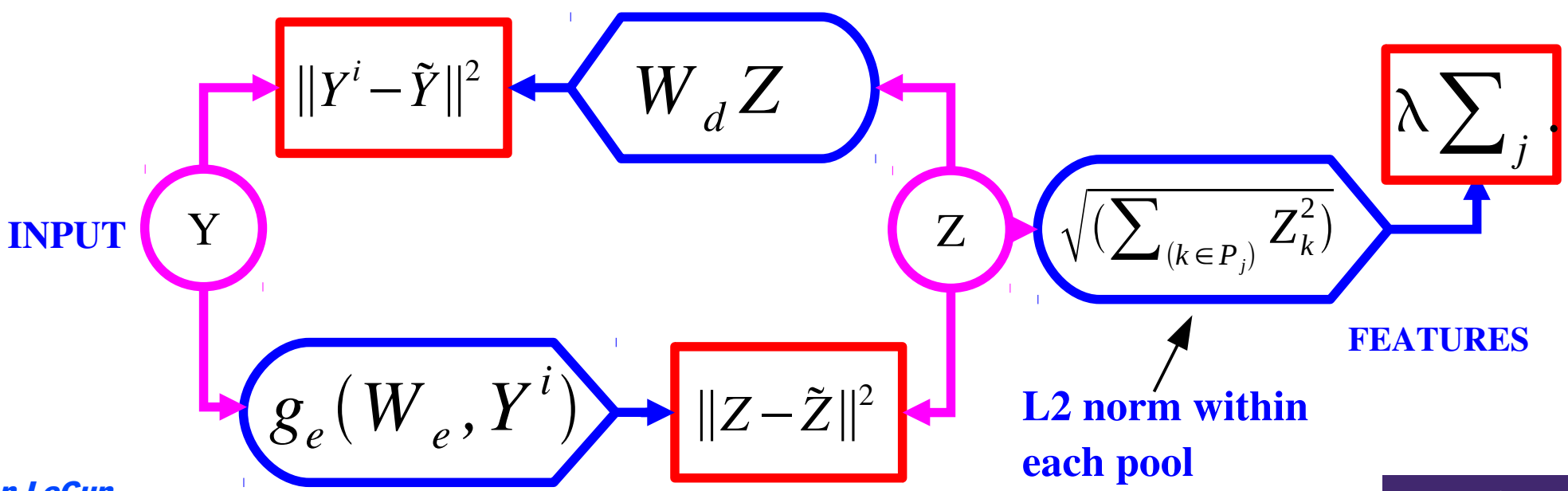
[Kavukcuoglu, Ranzato, Fergus, LeCun, CVPR 2009]

[Gregor & LeCun 2010]

Learning Invariant Features with L2 Group Sparsity

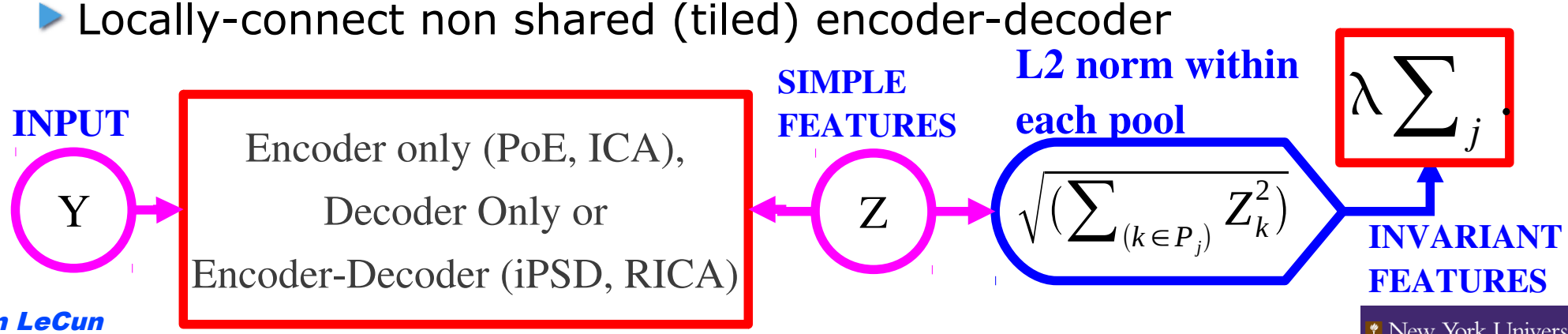
- Unsupervised PSD ignores the spatial pooling step.
- Could we devise a similar method that learns the pooling layer as well?
- Idea [Hyvarinen & Hoyer 2001]: group sparsity on pools of features
  - Minimum number of pools must be non-zero
  - Number of features that are on within a pool doesn't matter
  - Pools tend to regroup similar features

$$E(Y,Z)=\|Y-W_d Z\|^2+\|Z-g_e(W_e,Y)\|^2+\sum_j \sqrt{\sum_{k\in P_j} Z_k^2}$$



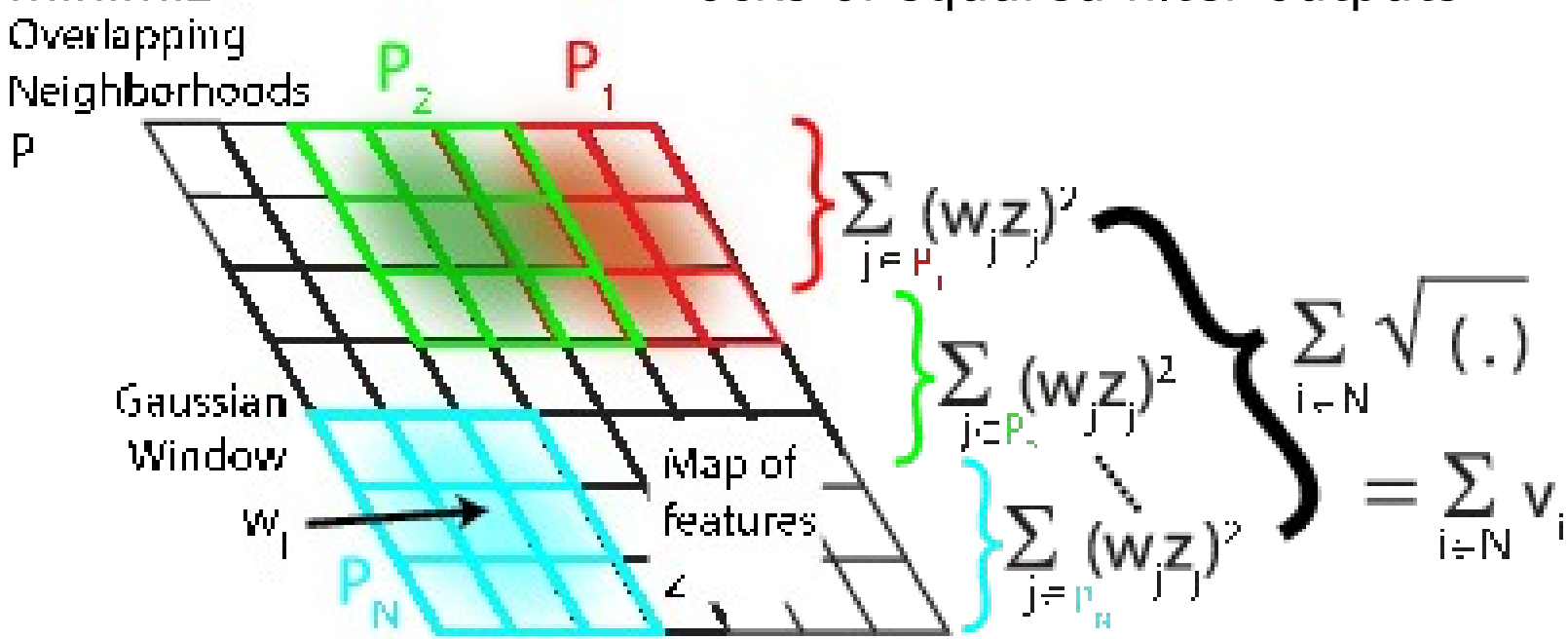
Learning Invariant Features with L2 Group Sparsity

- **Idea: features are pooled in group.**
  - sparsity: sum over groups of L2 norm of activity in group.
- **[Cardoso 1998, Hyvärinen Hoyer 2001]: multidim ICA, subspace ICA**
  - square, decoder only (Cardoso), encoder only (Hyvärinen),
- **[Welling, Hinton, Osindero NIPS 2002]: pooled product of experts (PoE)**
  - overcomplete, encoder only, log student-T penalty on L2 pooling
- **[Kavukcuoglu, Ranzato, Fergus LeCun, CVPR 2009]: Invariant PSD (iPSD)**
  - overcomplete, encoder-decoder (like PSD), L2 pooling
- **[Le et al. NIPS 2011]: Reconstruction ICA (RICA)**
  - Same as [Kavukcuoglu 2010] with linear encoder and tied decoder
- **[Gregor & LeCun arXiv:1006:0448, 2010] [Le et al. ICML 2012]**
  - Locally-connect non shared (tiled) encoder-decoder



Pooling Similar Features using Group Sparsity

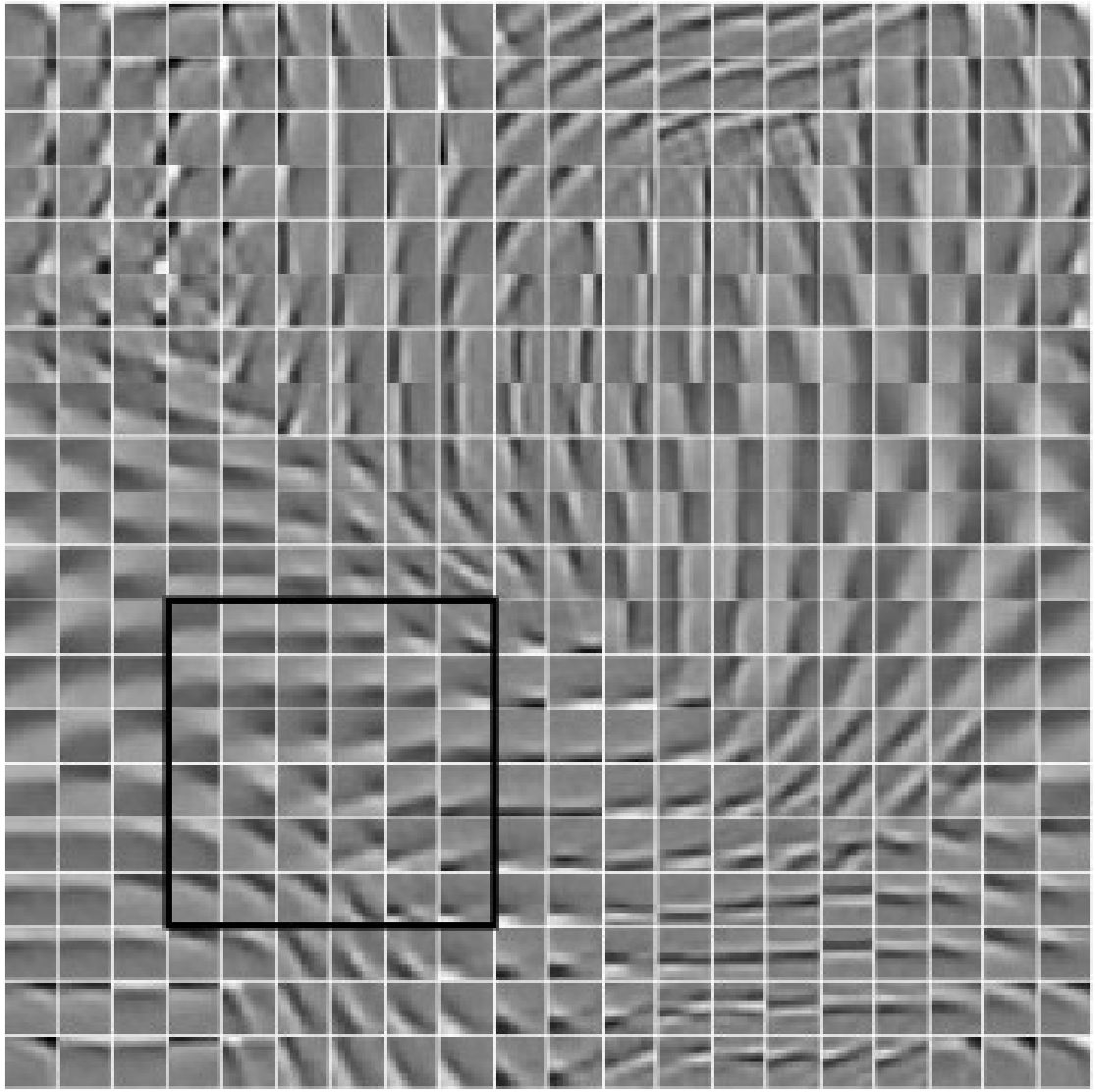
- A sparse-overcomplete version of Hyvarinen's subspace ICA
- Decoder ensures reconstruction (unlike ICA which requires orthonogonal matrix)
  - 1. Apply filters on a patch (with suitable non-linearity)
  - 2. Arrange filter outputs on a 2D plane
  - 3. square filter outputs
  - 4. minimize sqrt of sum of blocks of squared filter outputs



[Kavukcuoglu, Ranzato, Fergu, LeCun, CVPR 2009]
[Jenatton, Obozinski, Bach AISTATS 2010] [Le et al. NIPS2011]

Groups are local in a 2D Topographic Map

- The filters arrange themselves spontaneously so that similar filters enter the same pool.
- The pooling units can be seen as complex cells
- Outputs of pooling units are invariant to local transformations of the input
  - For some it's translations, for others rotations, or other transformations.



Pinwheels?

- Does that look pinwheely to you?

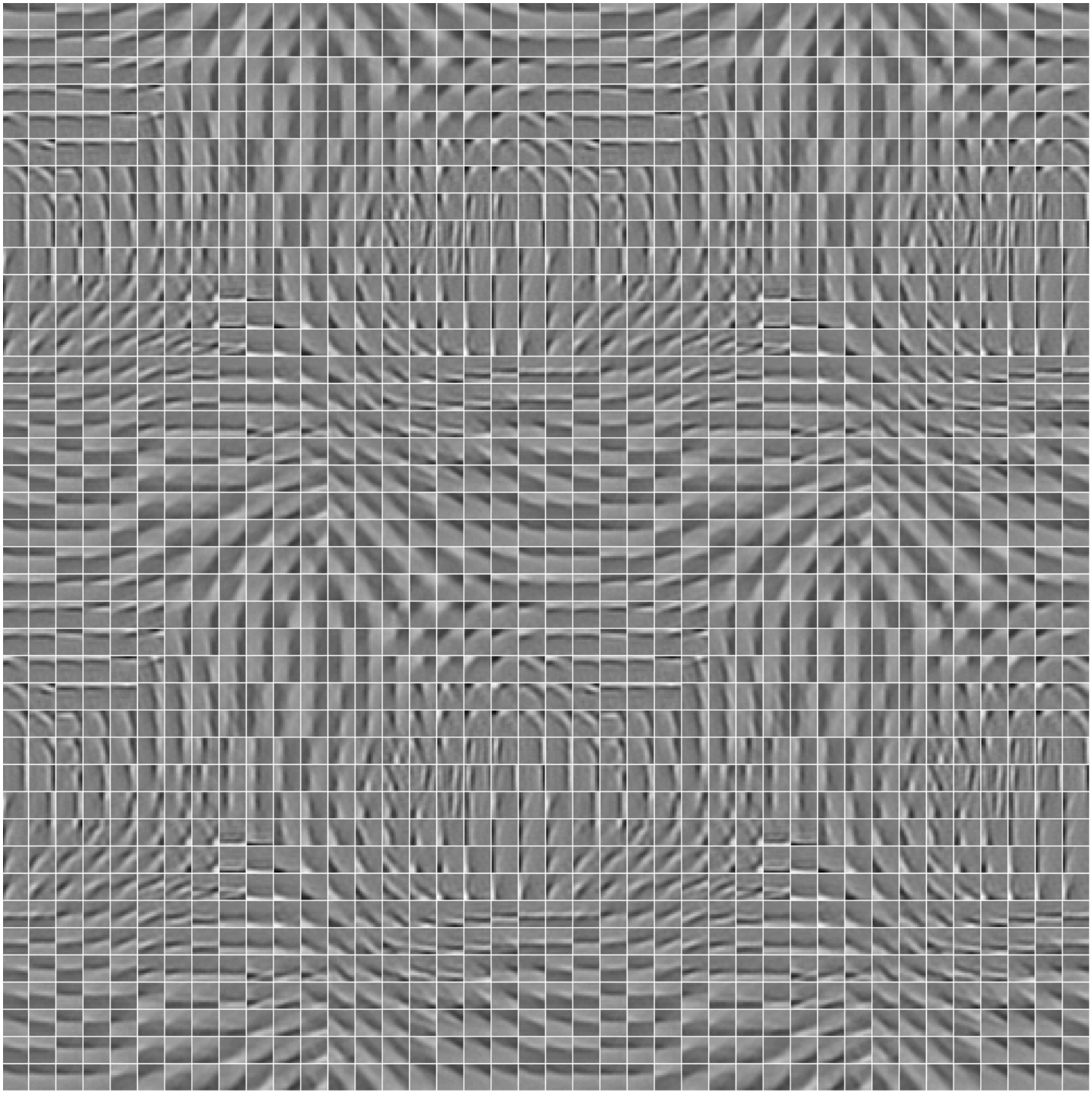


Image-level training, local filters but no weight sharing

- Training on 115x115 images. Kernels are 15x15 (not shared across space!)
  - [Gregor & LeCun 2010]
  - Local receptive fields
  - No shared weights
  - 4x overcomplete
  - L2 pooling
  - Group sparsity over pools

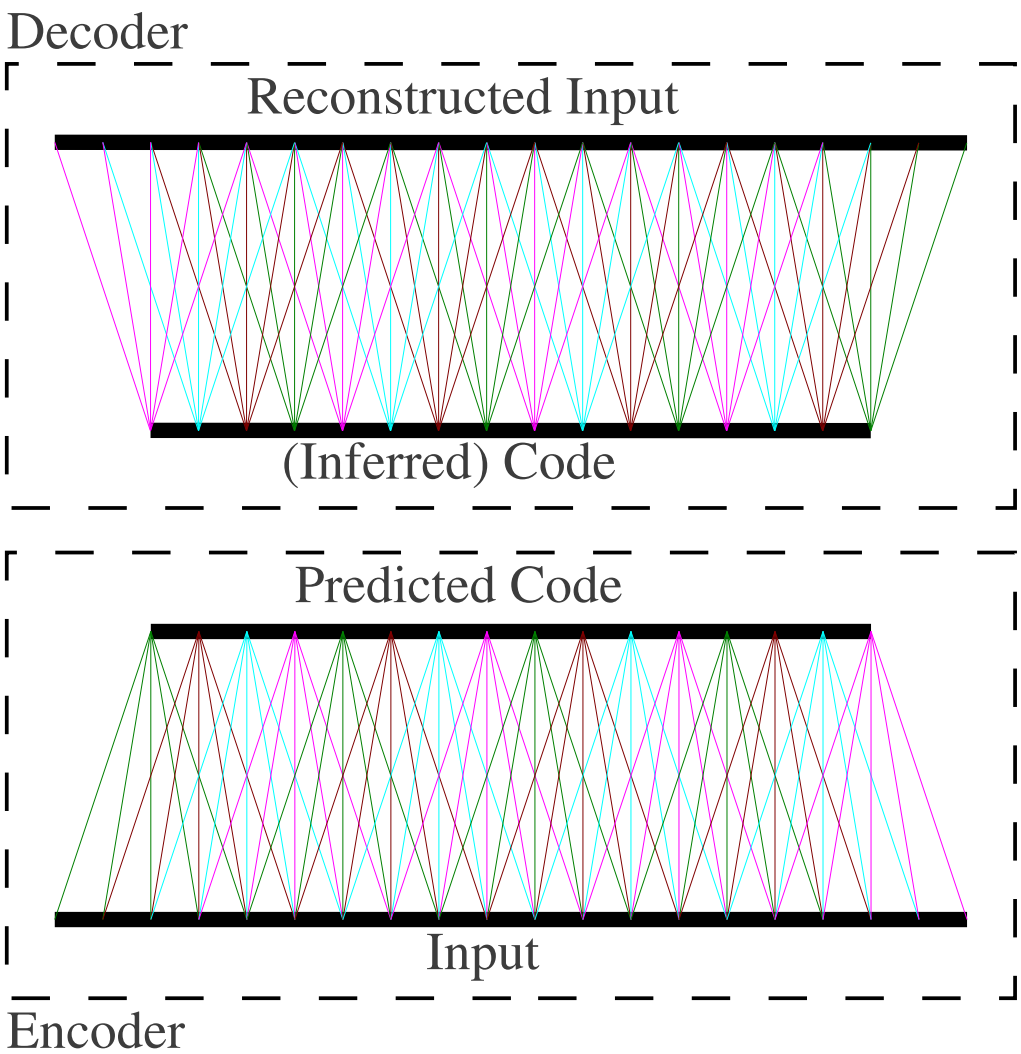


Image-level training, local filters but no weight sharing

- Topographic maps of continuously-varying features
- Local overlapping pools are invariant complex cells
  - [Gregor & LeCun arXiv:1006.0448] (double tanh encoder)
  - [Le et al. ICML'12] (linear encoder)

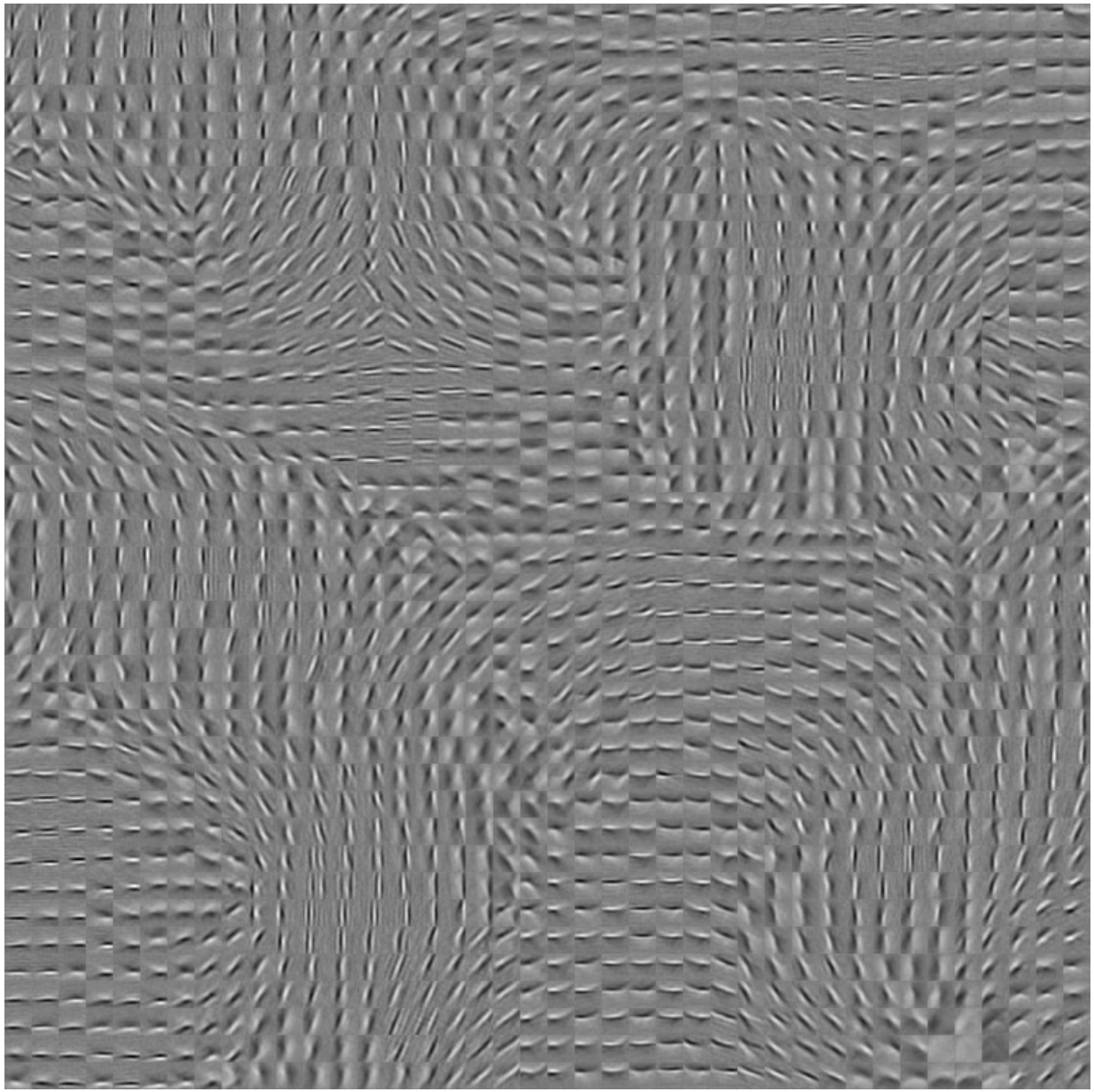
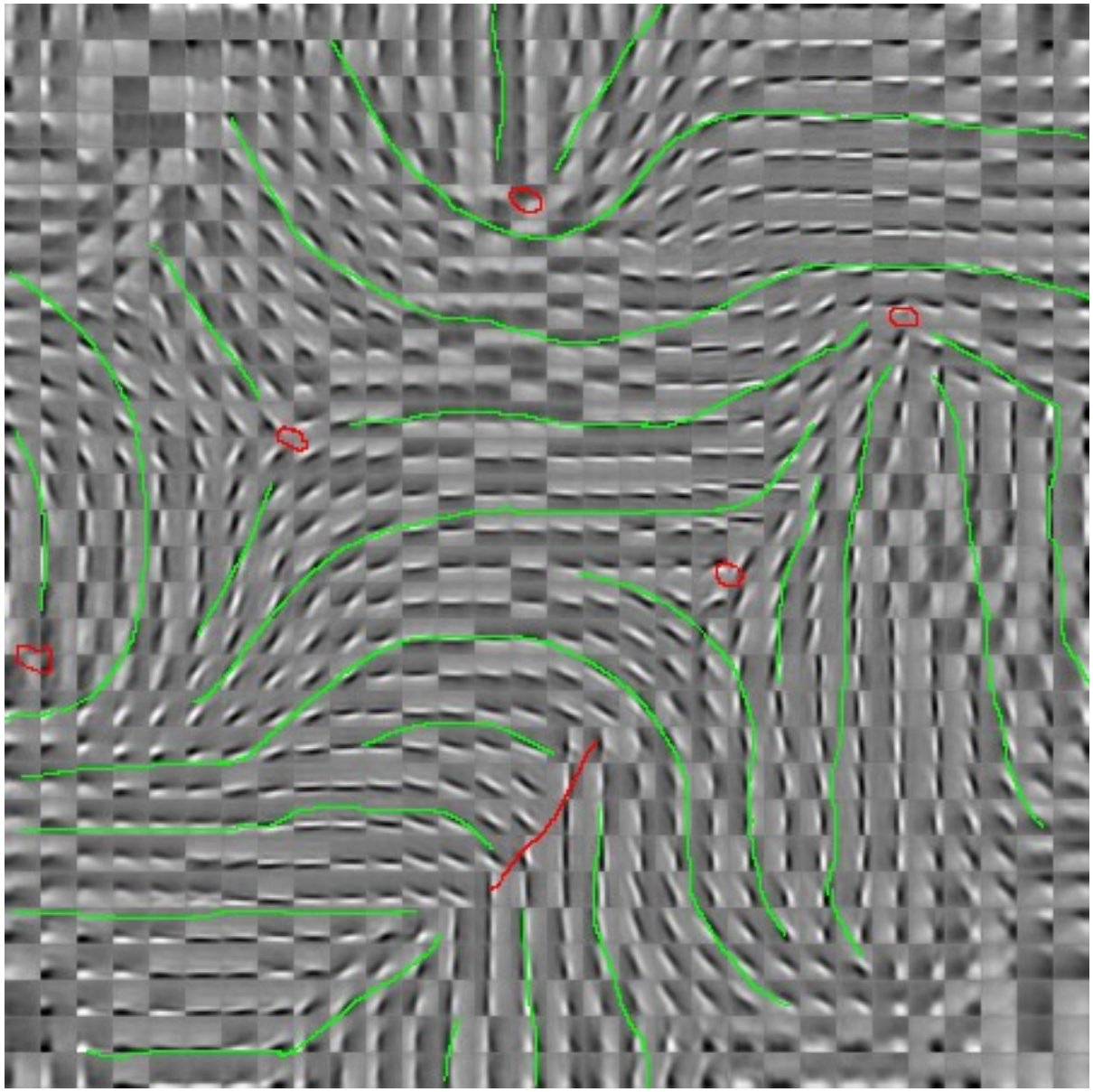
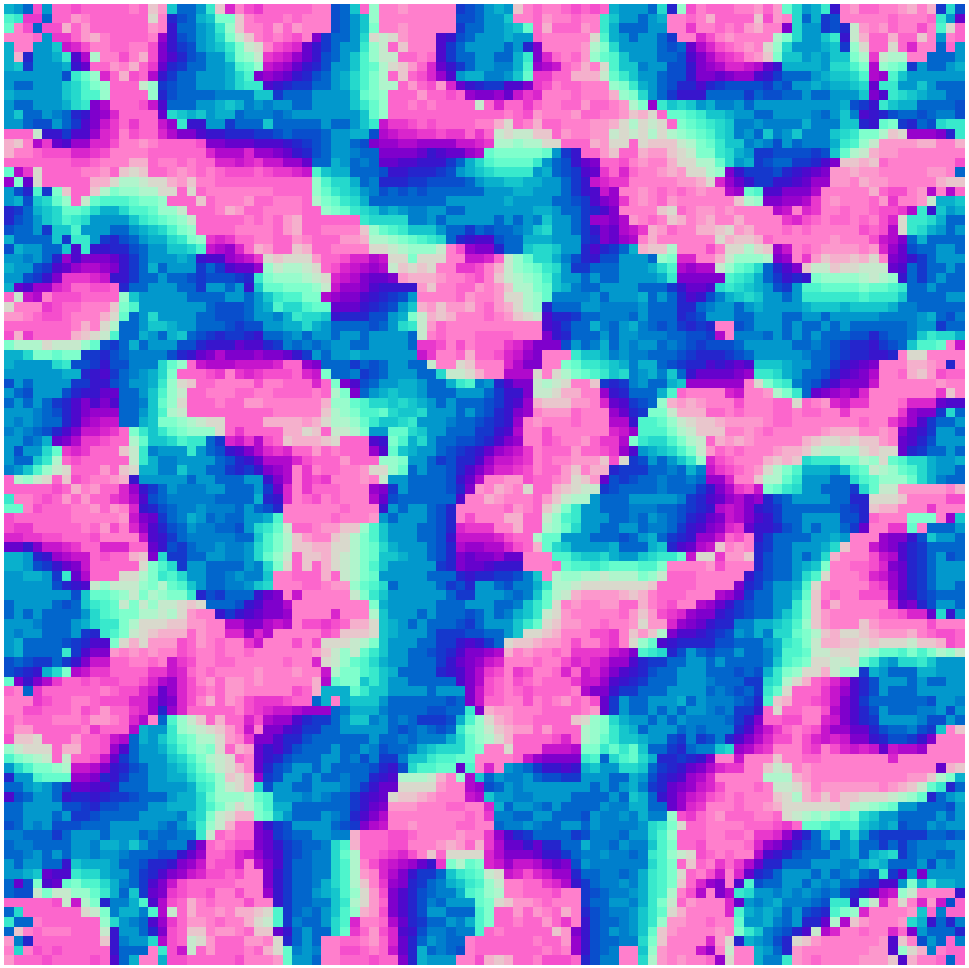


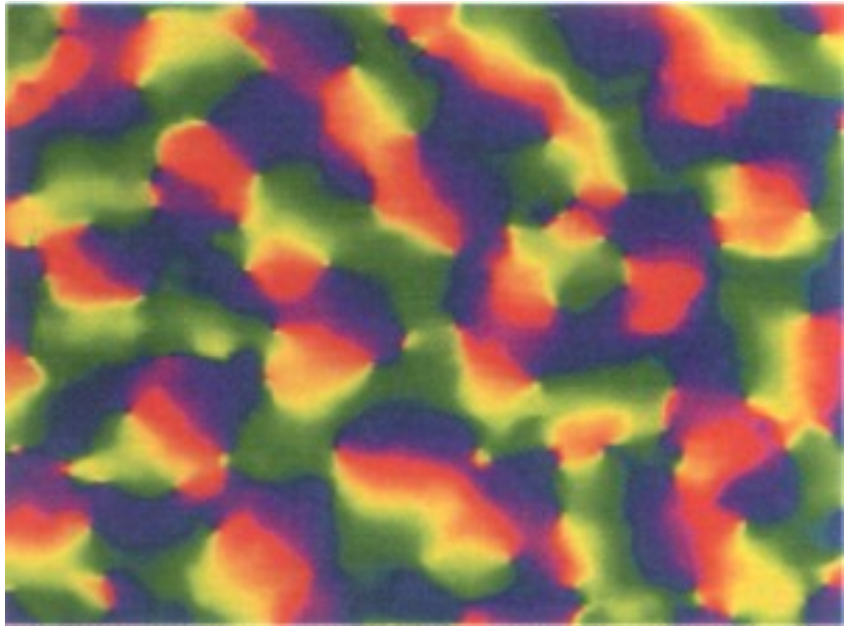
Image-level training, local filters but no weight sharing

- Training on 115x115 images. Kernels are 15x15 (not shared across space!)



K Obermayer and GG Blasdel, Journal of Neuroscience, Vol 13, 4114-4129 (**Monkey**)



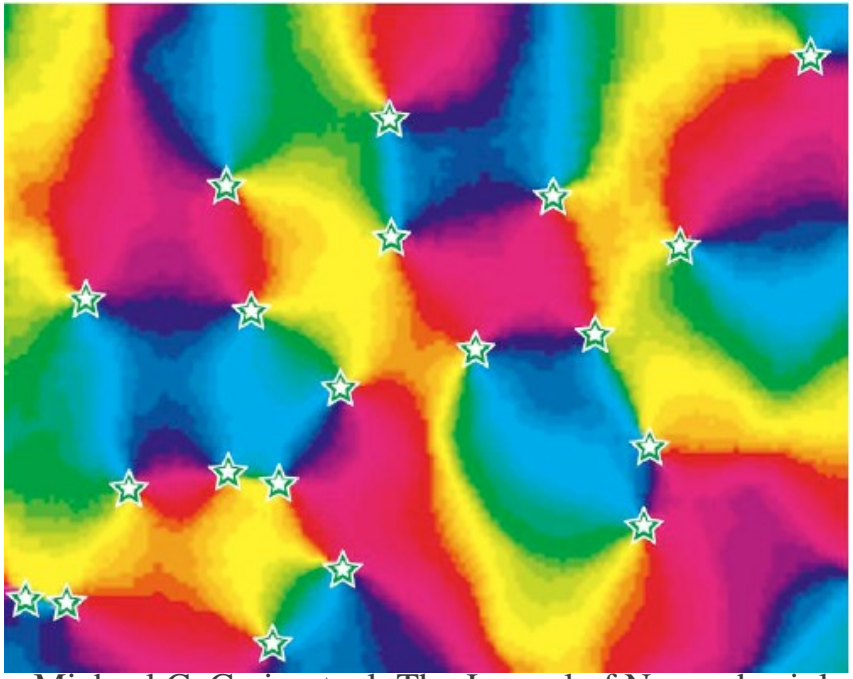


119x119 Image Input

100x100 Code

20x20 Receptive field size

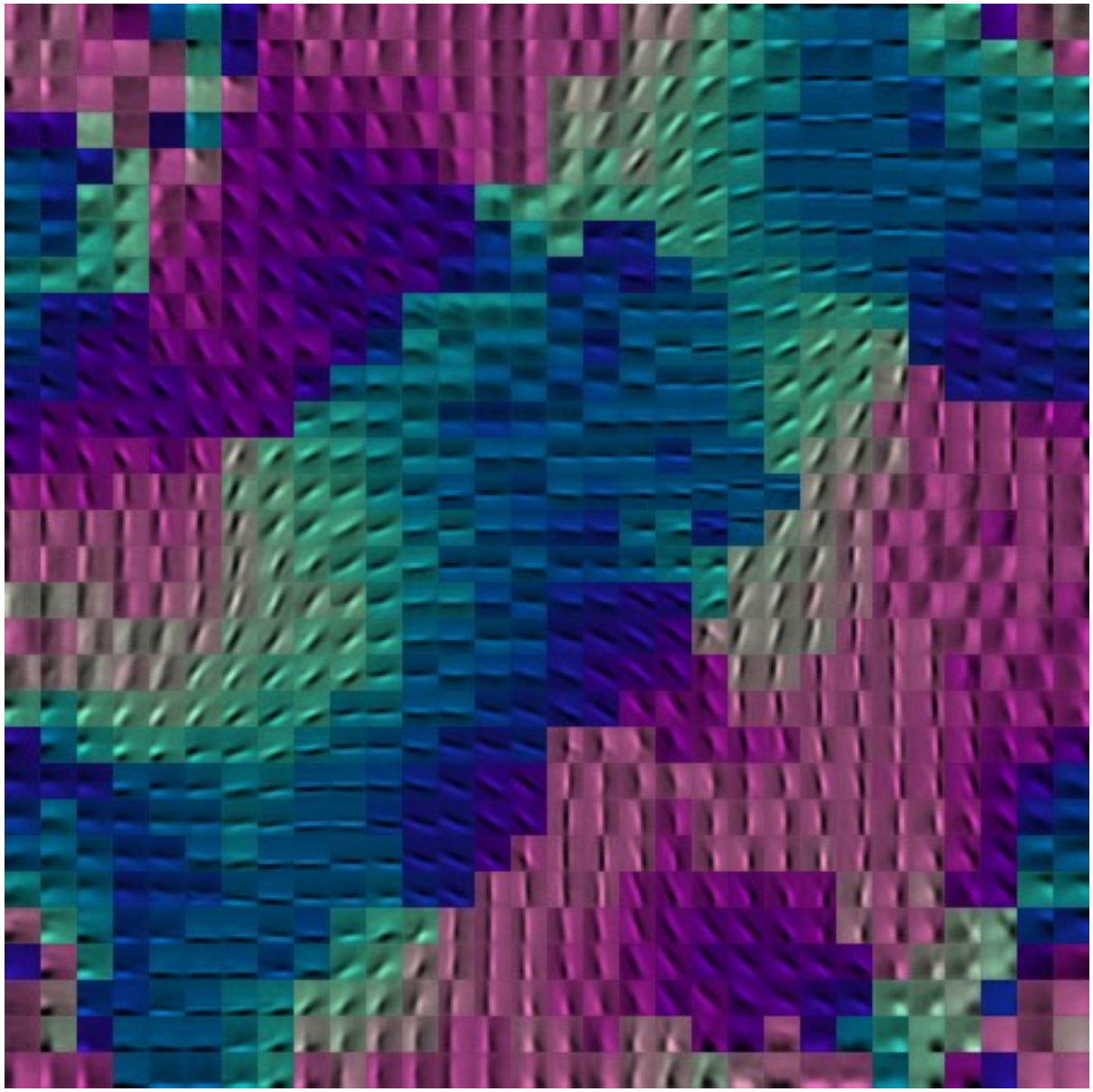
sigma=5



Michael C. Crair, et. al. The Journal of Neurophysiology Vol. 77 No. 6 June 1997, pp. 3381-3385 (**Cat**)

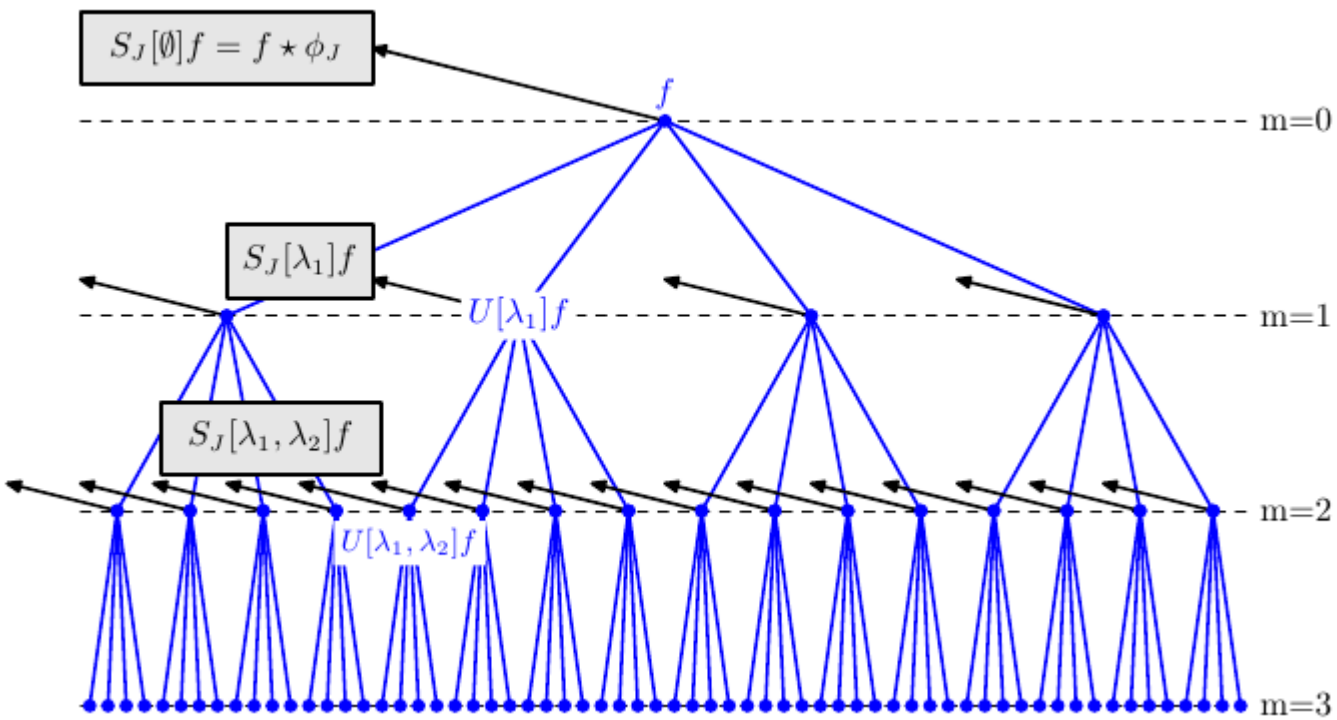
Image-level training, local filters but no weight sharing

- Color indicates orientation (by fitting Gabors)



Theory of Repeated [Filter Bank → L2 Pooling → Average Pooling]

- Stéphane Mallat's "Scattering Transform": Theory of ConvNet-like architectures
- [Mallat & Bruna CVPR 2011] Classification with Scattering Operators
- [Mallat & Bruna, arXiv:1203.1513 2012] Invariant Scattering Convolution Networks
- [Mallat CPAM 2012] Group Invariant Scattering



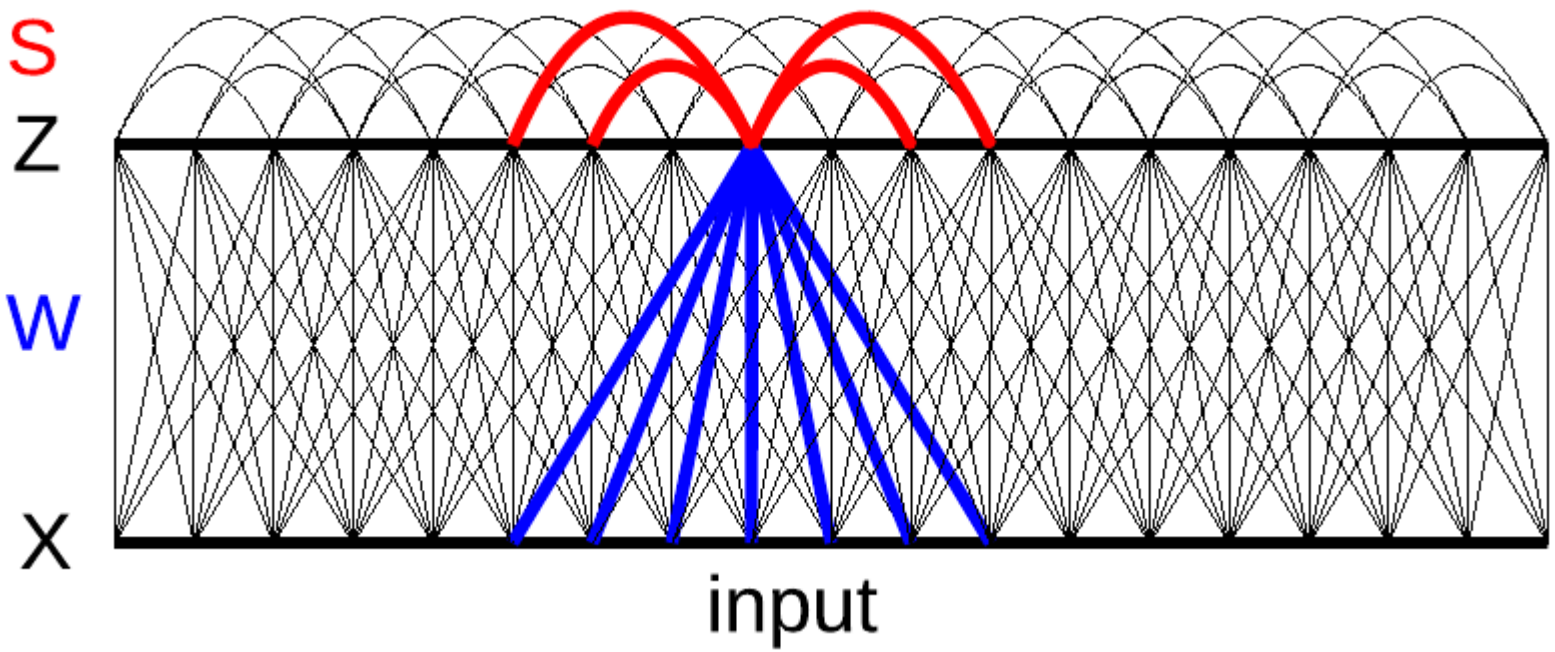
Sparse Coding Using Lateral Inhibition

[Gregor, Szlam, LeCun, NIPS 2011]

Invariant Features Lateral Inhibition

- Replace the L1 sparsity term by a lateral inhibition matrix
- Easy way to impose some structure on the sparsity

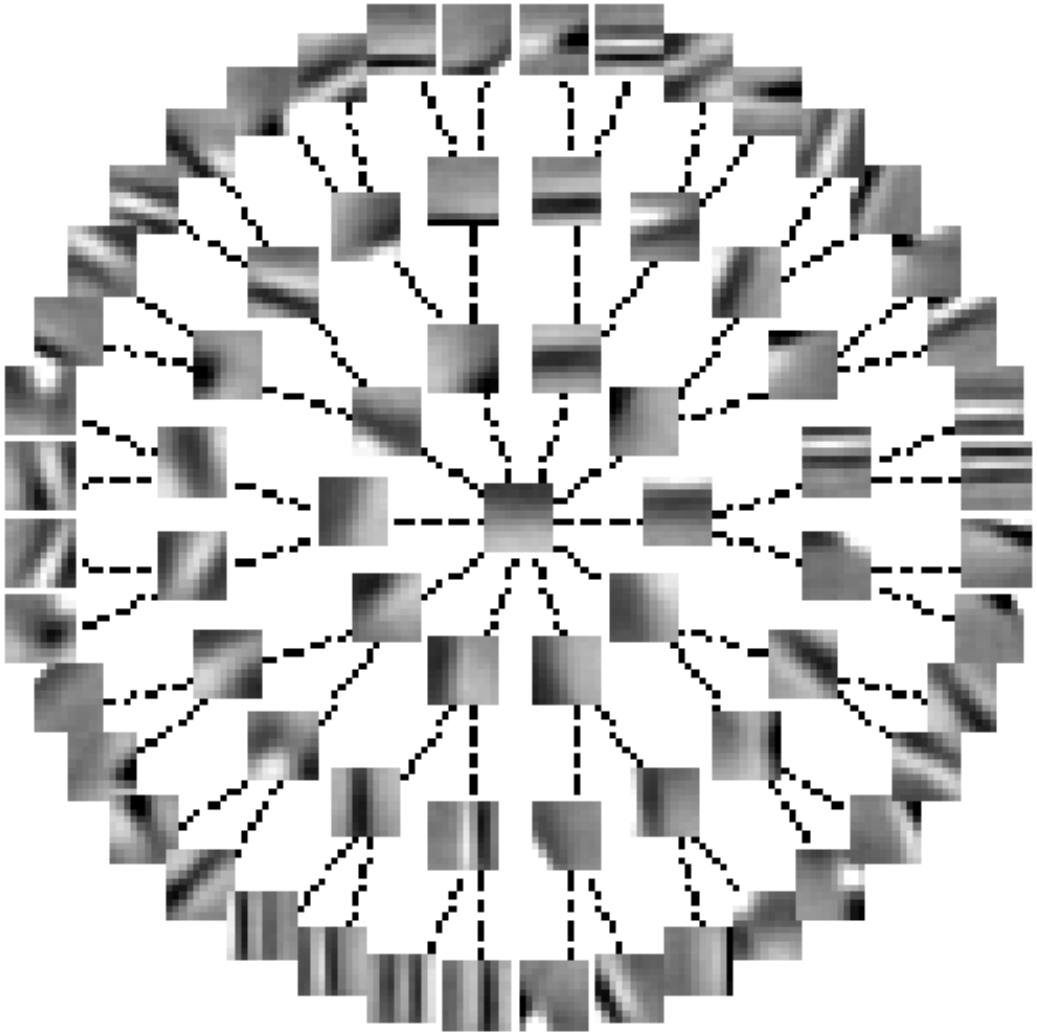
$$\min_{W,Z} \sum_{x \in X} ||Wz - x||^2 + |z|^T S |z|$$

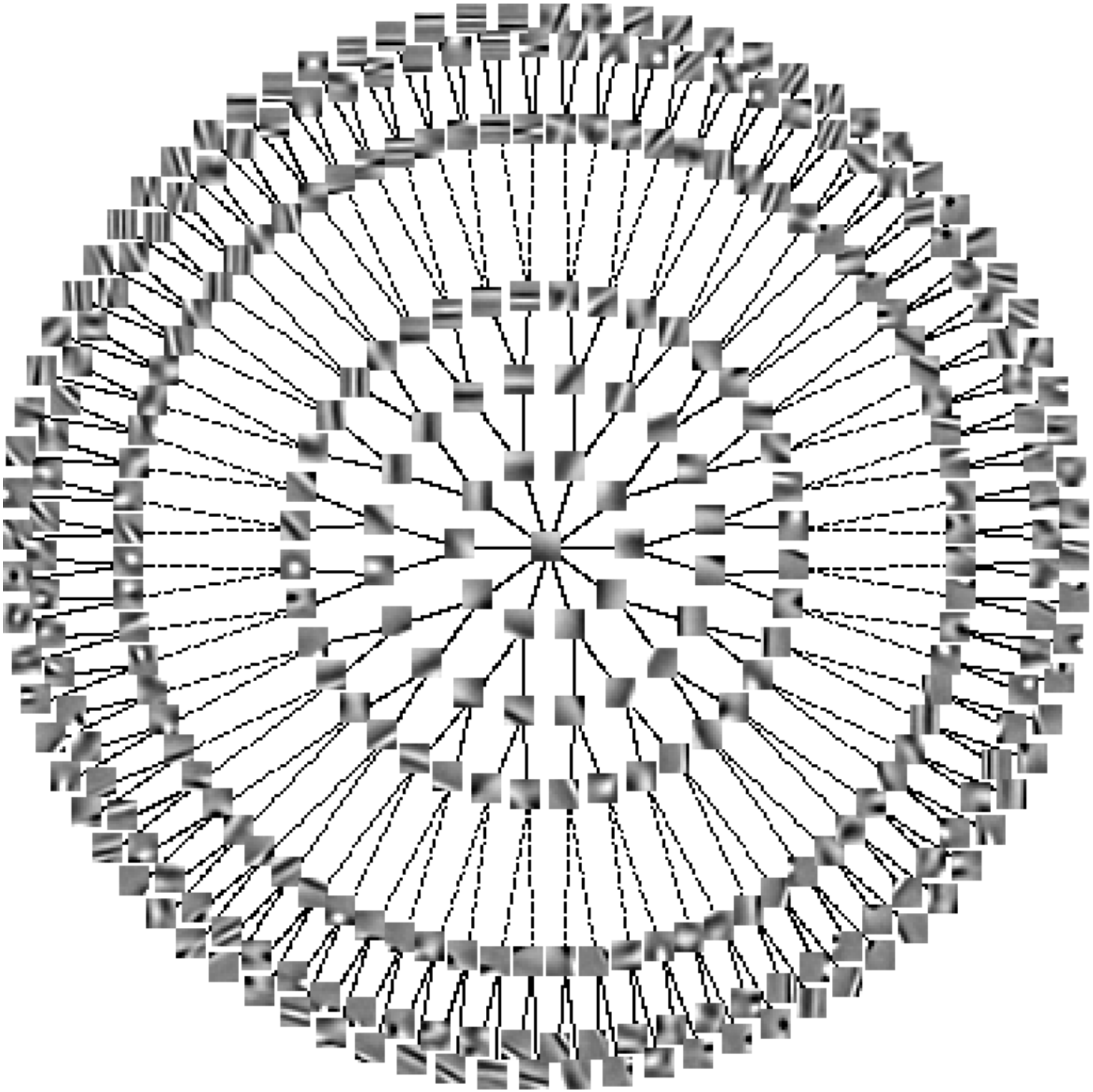


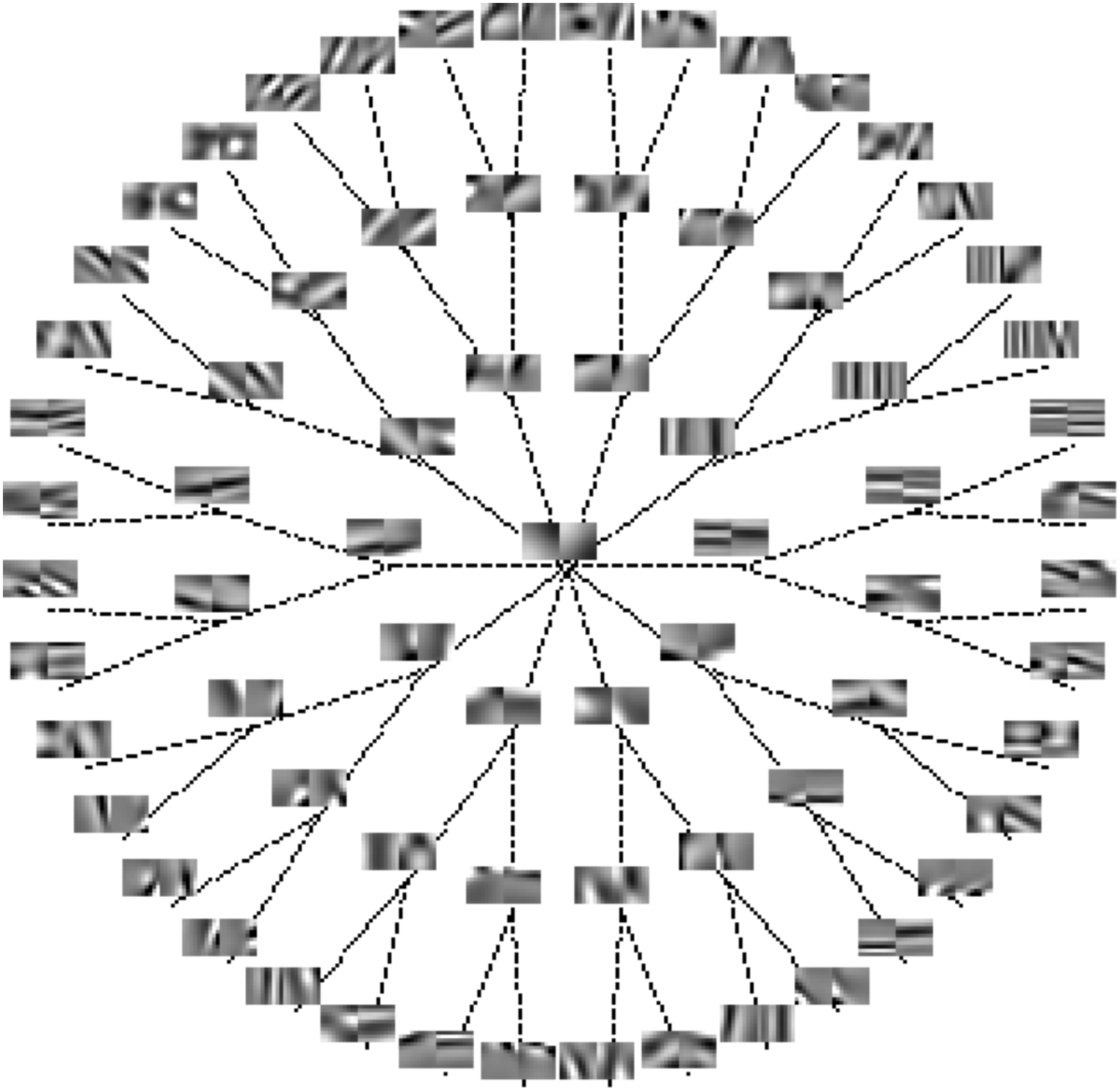
[Gregor, Szlam, LeCun NIPS 2011]

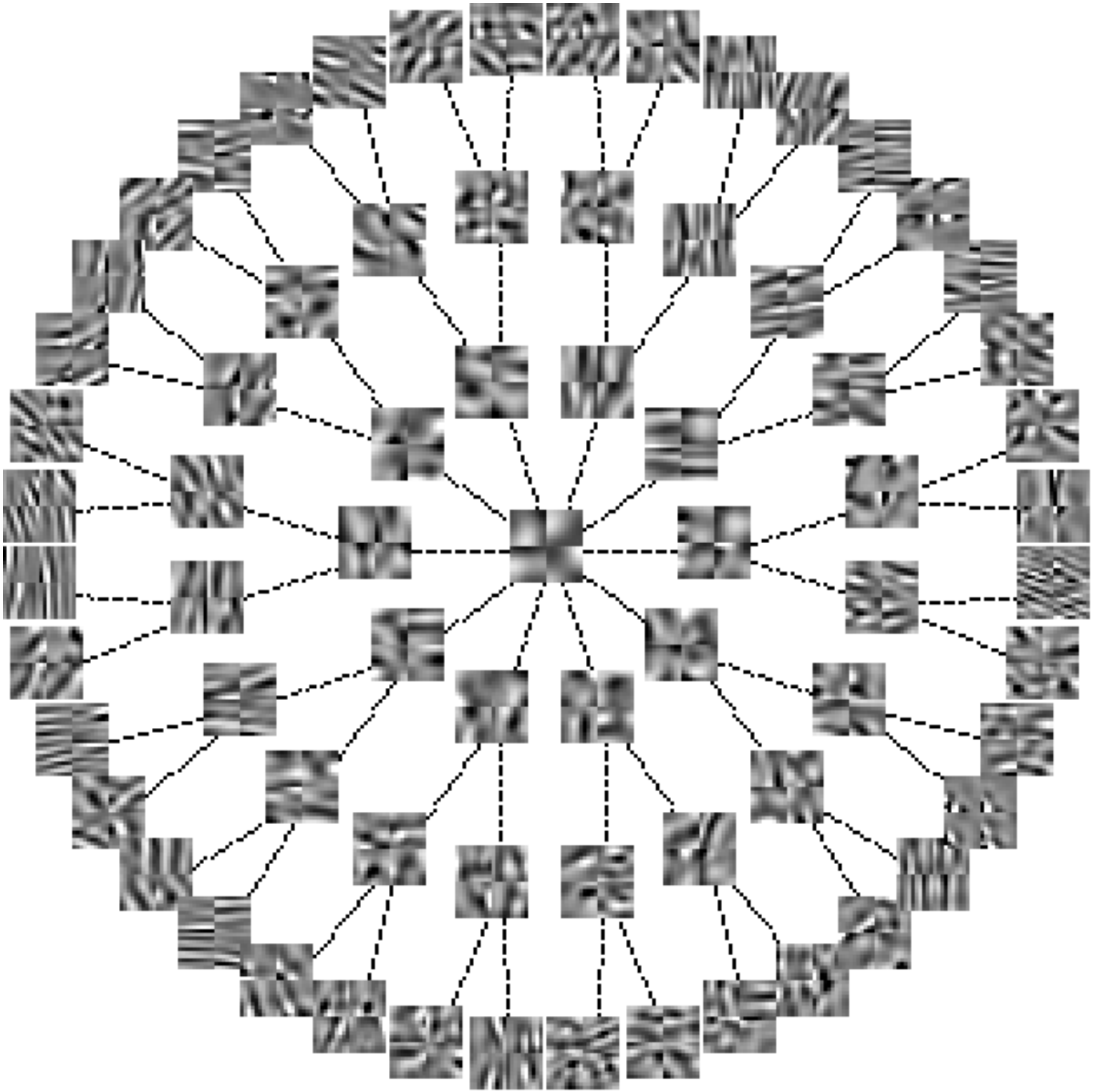
Invariant Features via Lateral Inhibition: Structured Sparsity

- Each edge in the tree indicates a zero in the S matrix (no mutual inhibition)
- Sij is larger if two neurons are far away in the tree



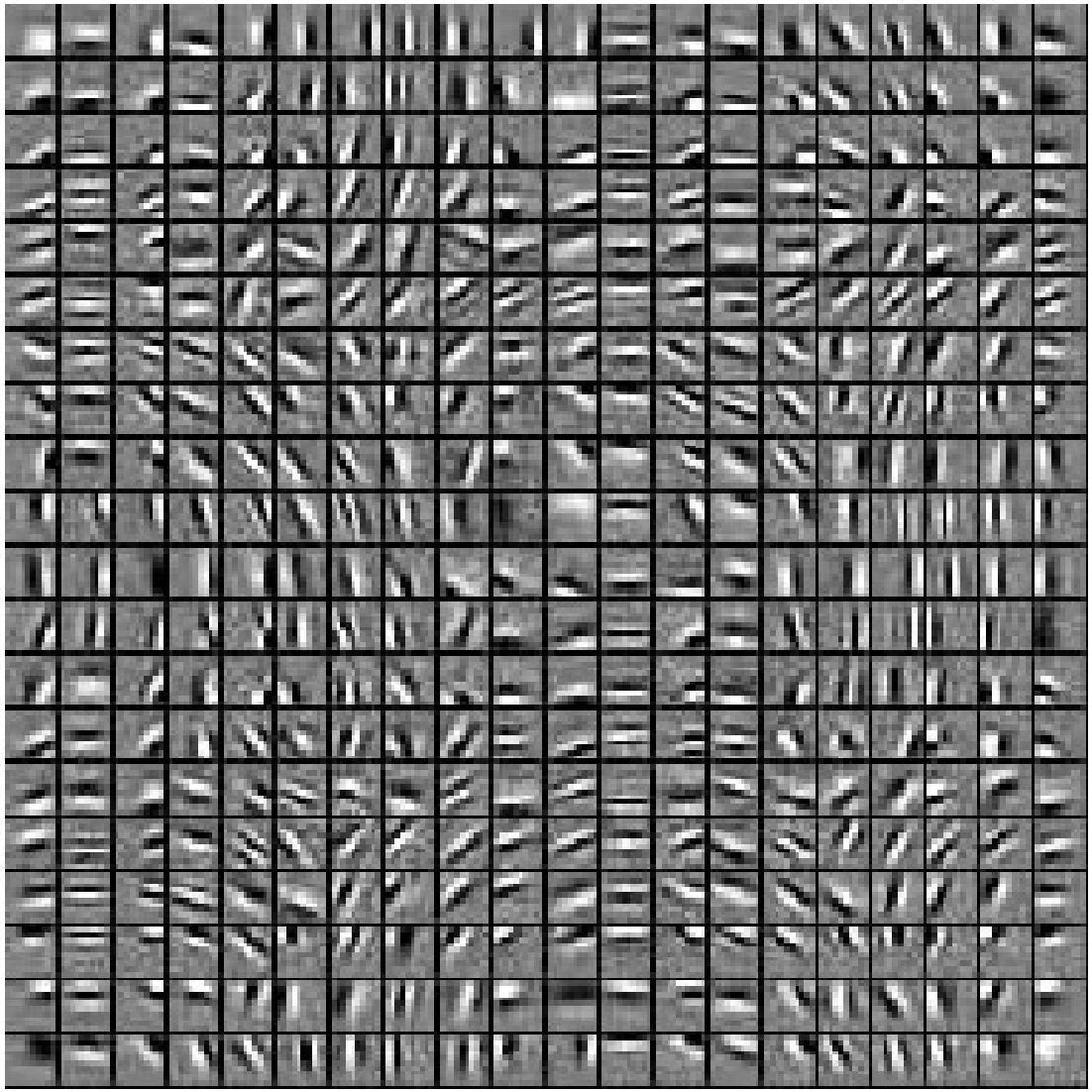






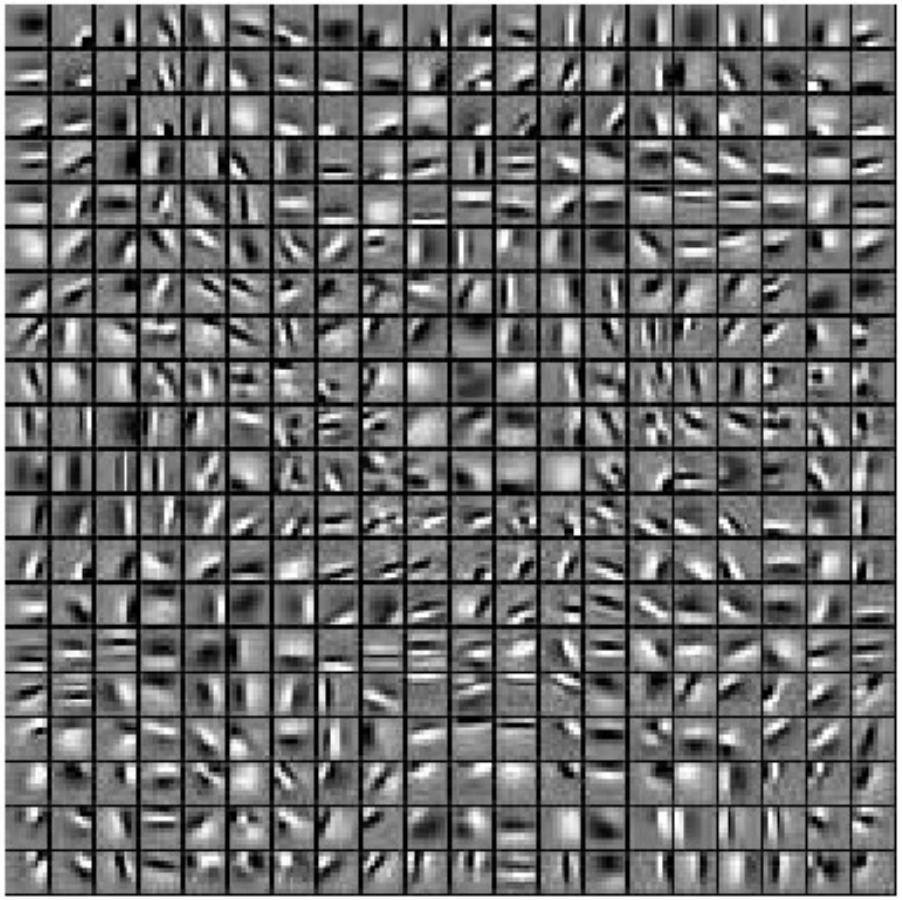
Invariant Features via Lateral Inhibition: Topographic Maps

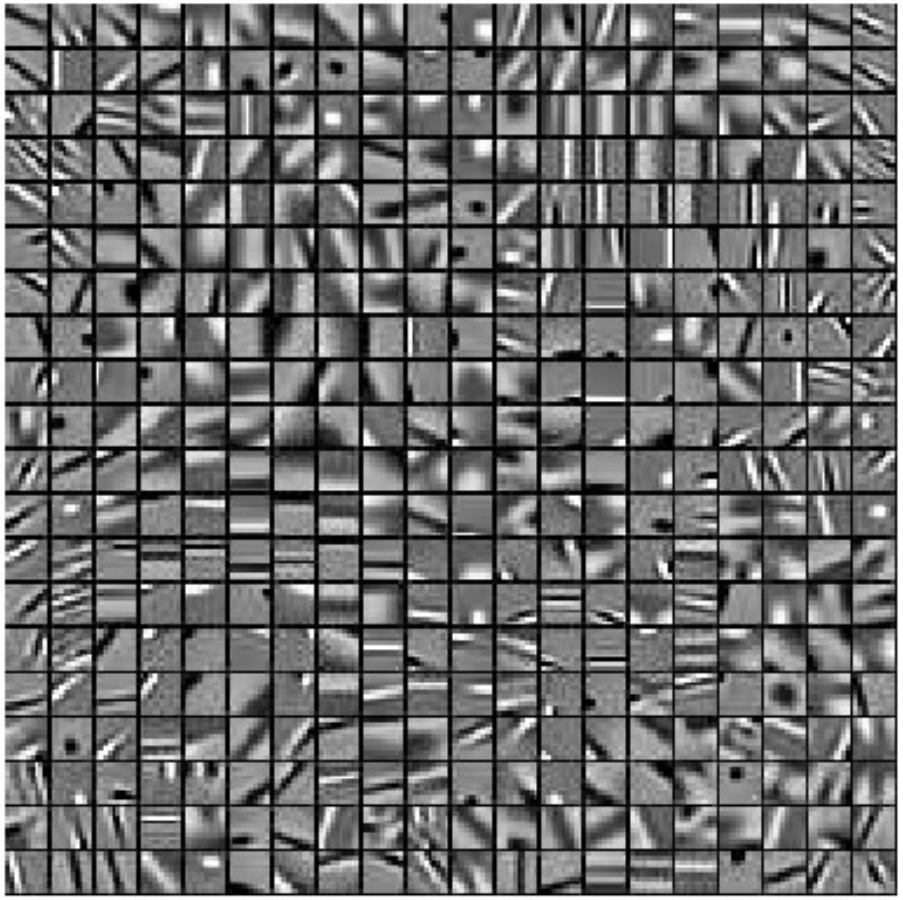
- Non-zero values in S form a ring in a 2D topology
  - Input patches are high-pass filtered



Invariant Features via Lateral Inhibition: Topographic Maps

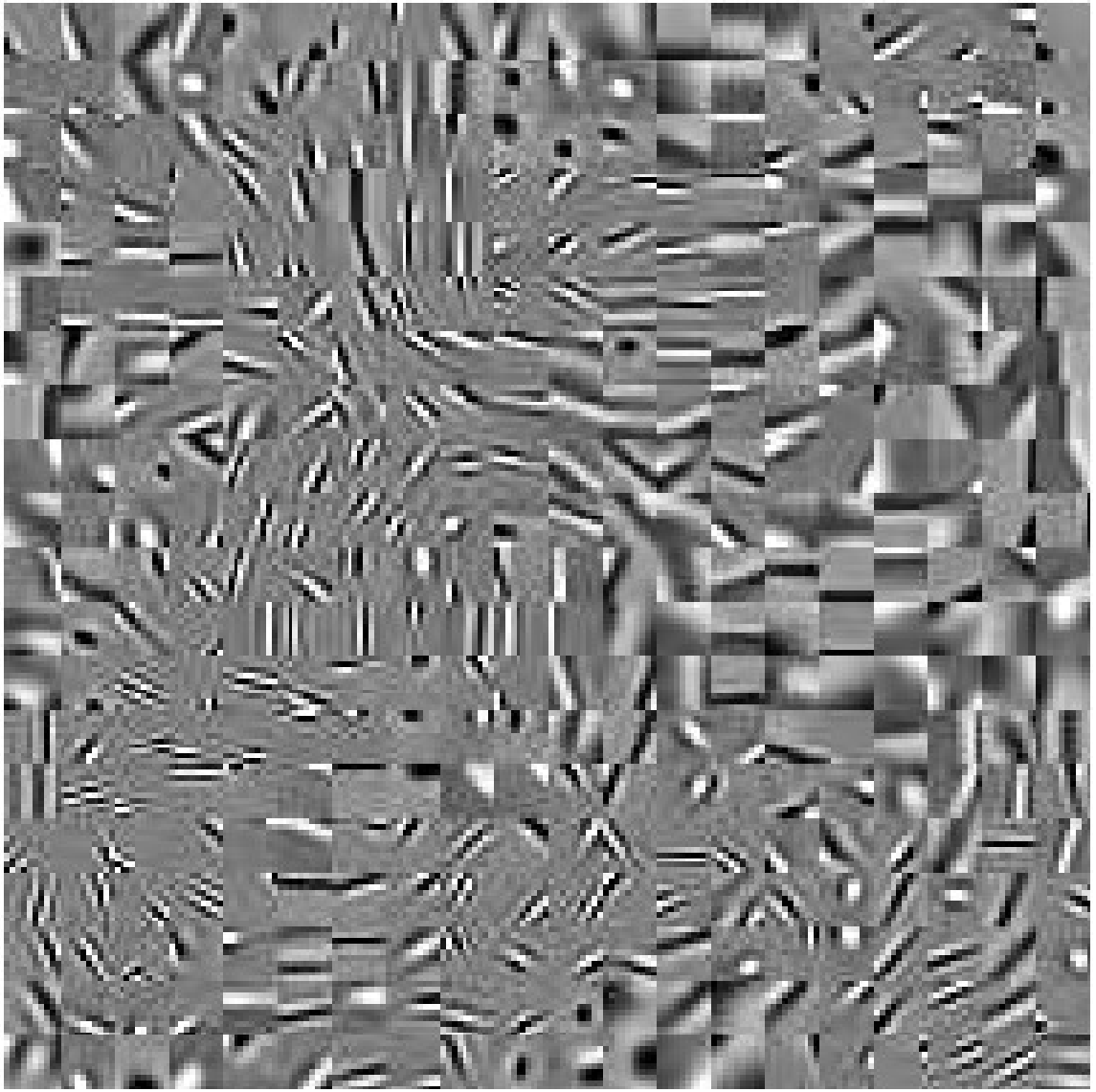
- Non-zero values in S form a ring in a 2D topology
  - Left: no high-pass filtering of input
  - Right: patch-level mean removal





Invariant Features via Lateral Excitation: Topographic Maps

- Short-range lateral excitation + L1 sparsity

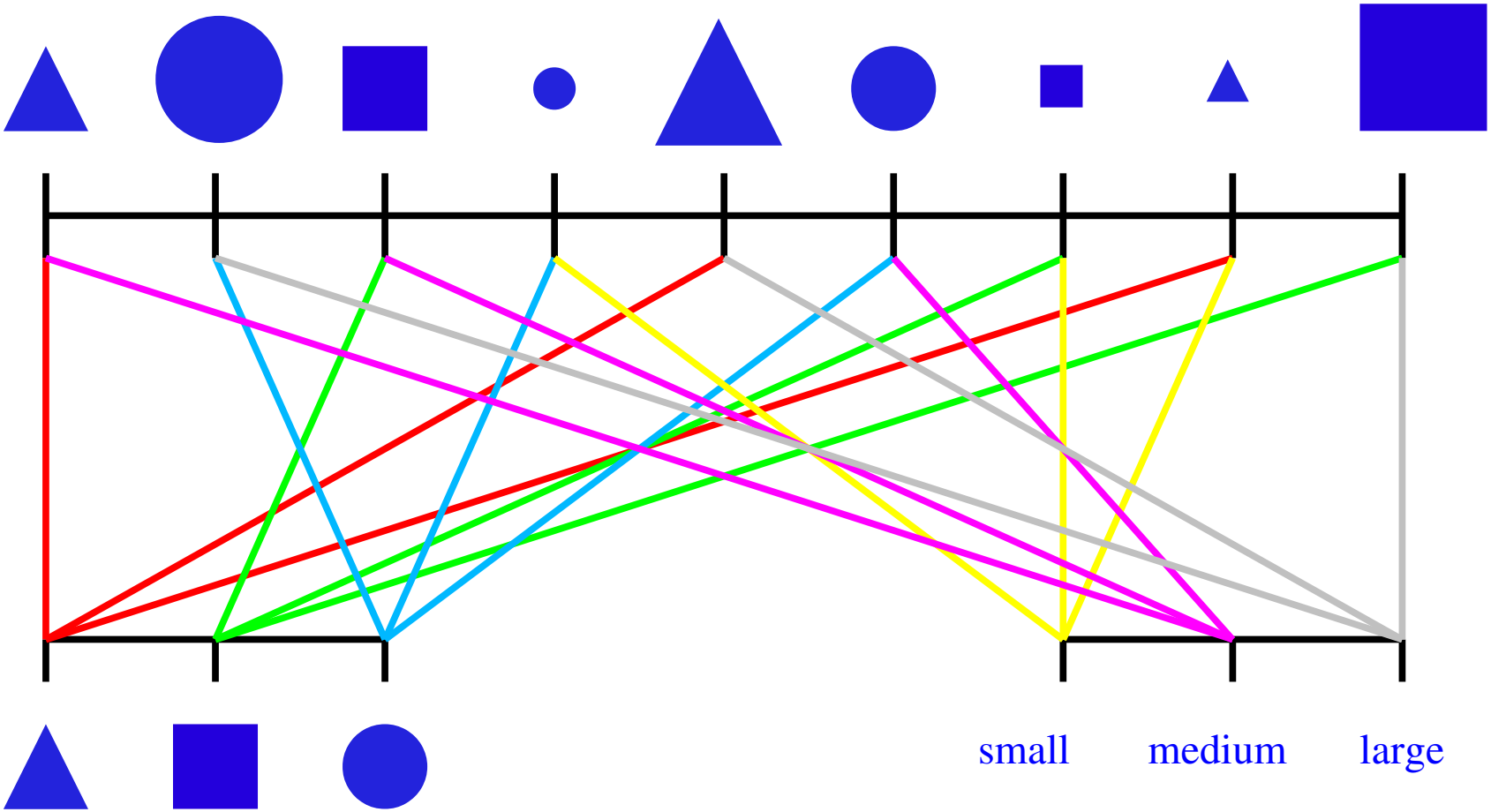


Learning What/Where Features with Temporal Constancy

[Gregor & LeCun arXiv:1006.0448, 2010]

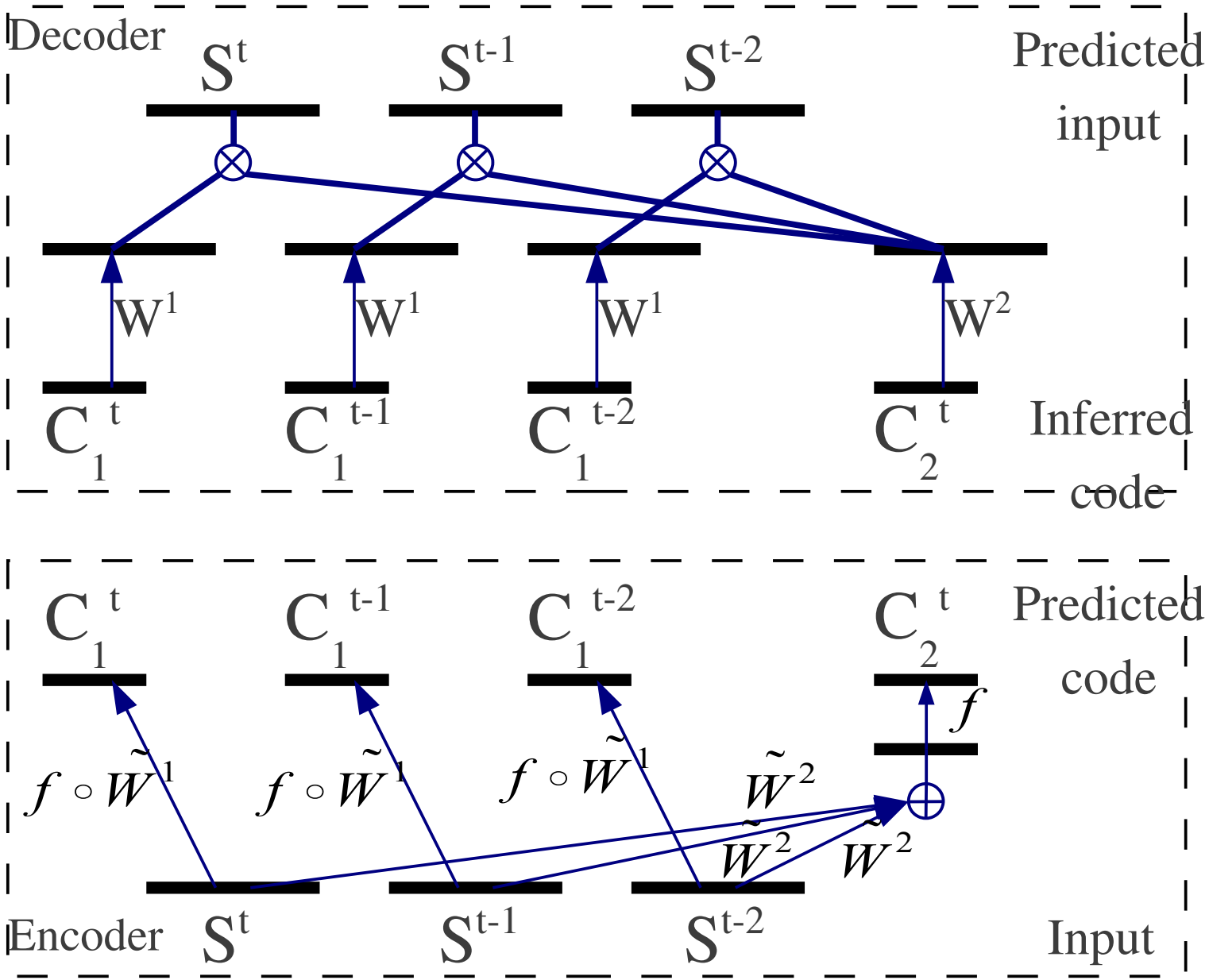
Invariant Features through Temporal Constancy

- Object is cross-product of object type and instantiation parameters
  - Mapping units [Hinton 1981], capsules [Hinton 2011]



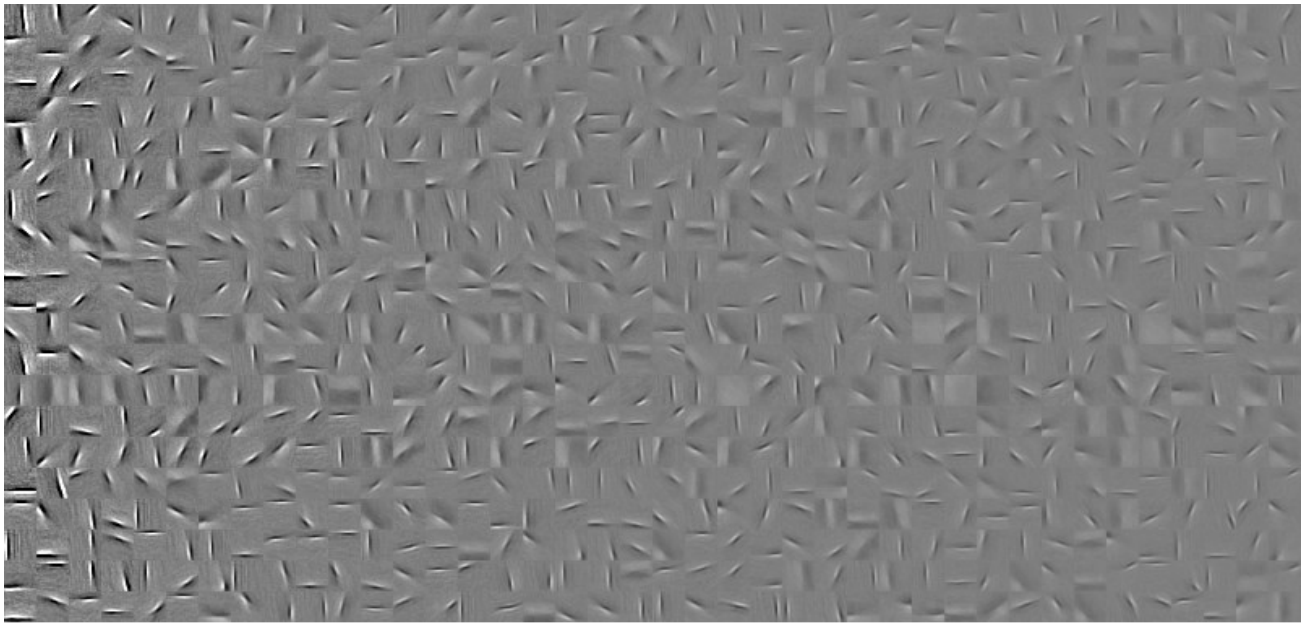
[Karol Gregor et al.]

What-Where Auto-Encoder Architecture

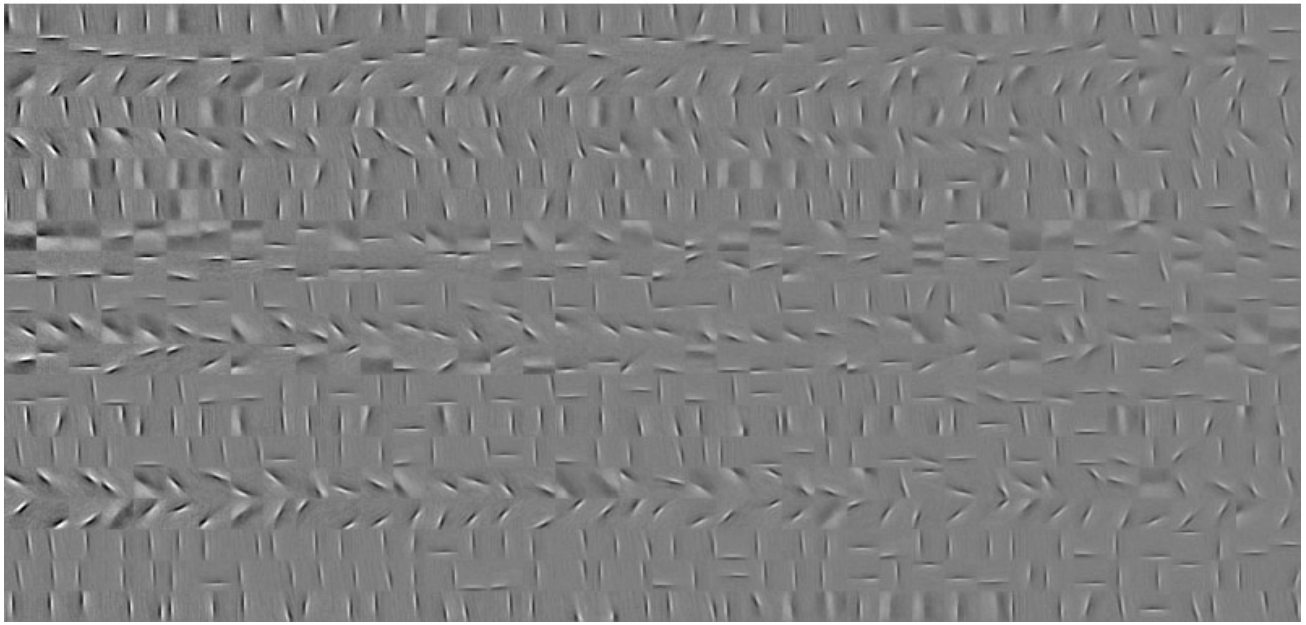


Low-Level Filters Connected to Each Complex Cell

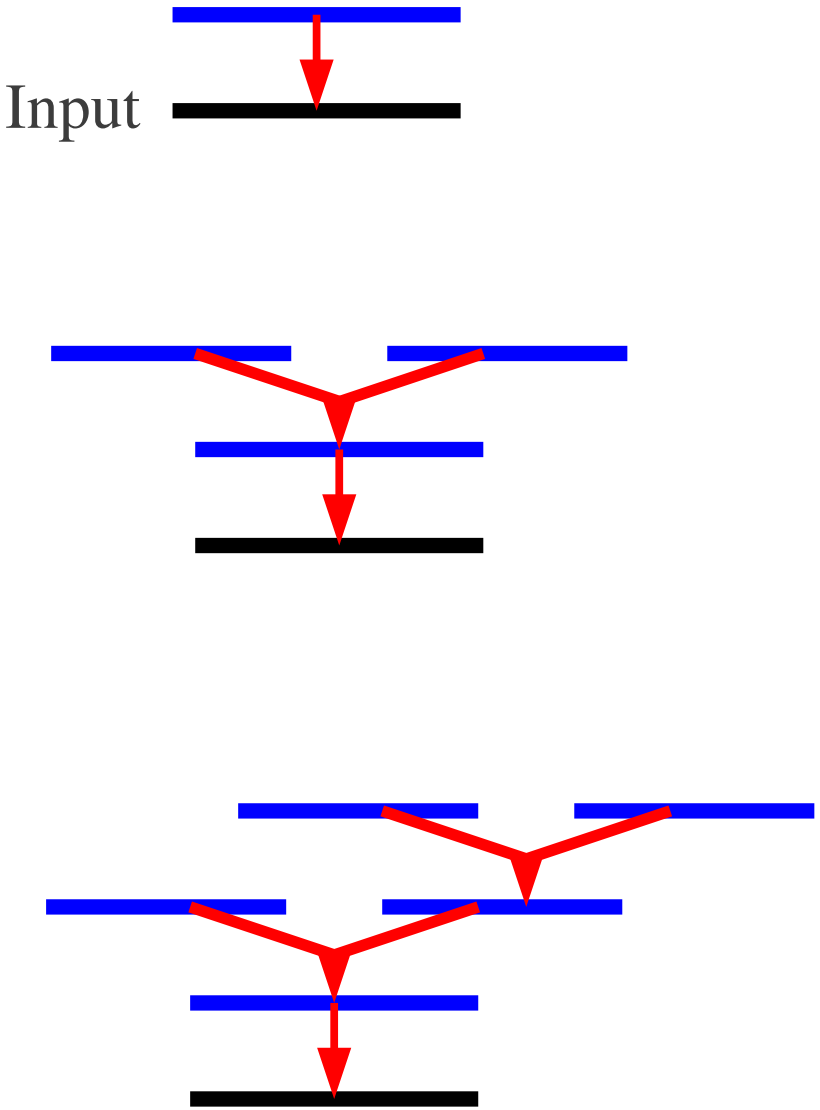
C1
(where)

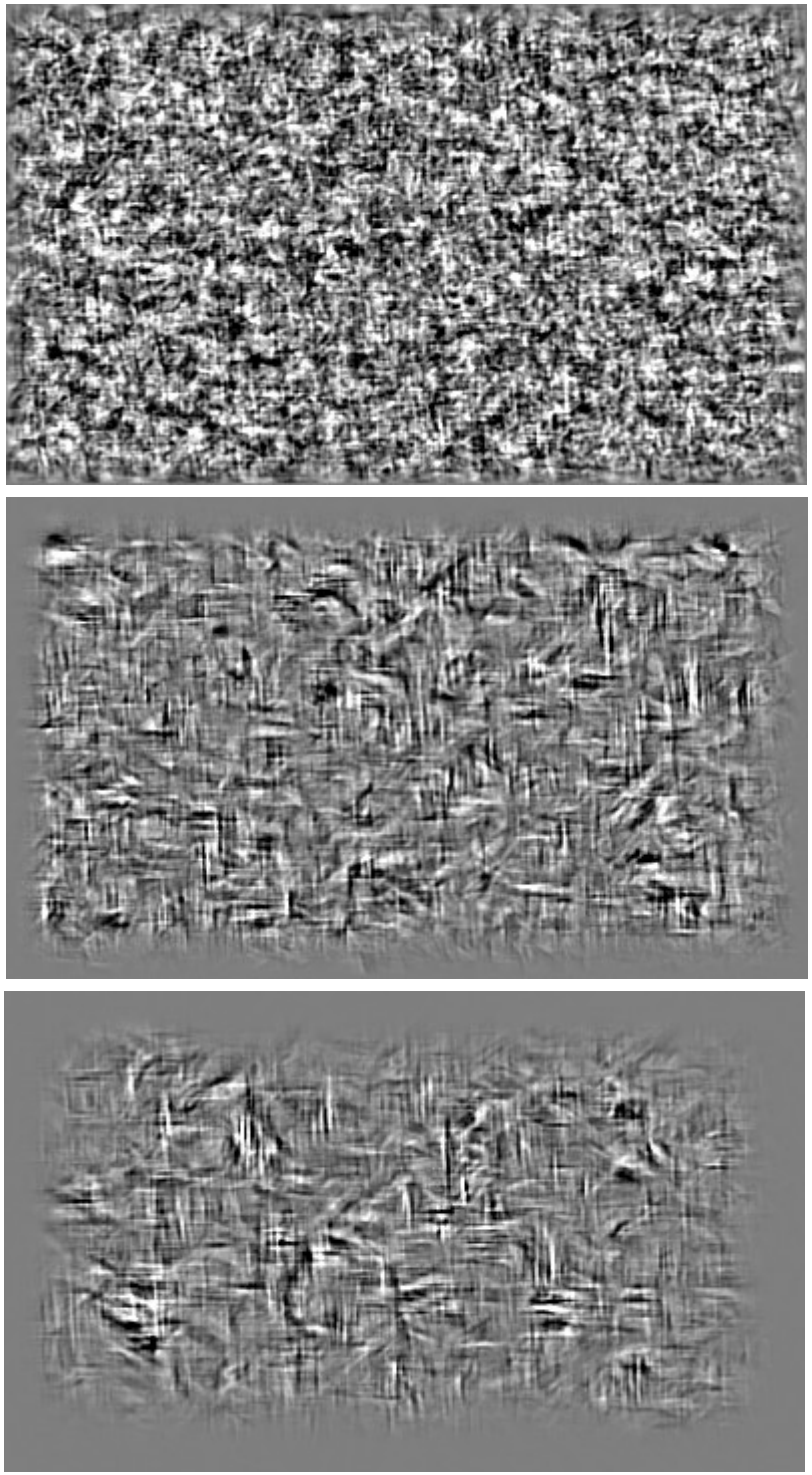


C2
(what)



Generating from the Network





Learning Features and Pose Embedding with DrLIM

Dimensionality Reduction by Learning and Invariant Mapping.

Contrastive Loss Function for Metric Learning

[Goldberger, Roweis, Hinton, Salakhutdinov, NIPS 2004]: NCA

[Chopra, Hadsell, LeCun CVPR 2005]

[Hadsell, Chopra, LeCun, CVPR 2006]: DrLIM

[Taylor, Fergus, LeCun, Bregler, ECCV 2010]

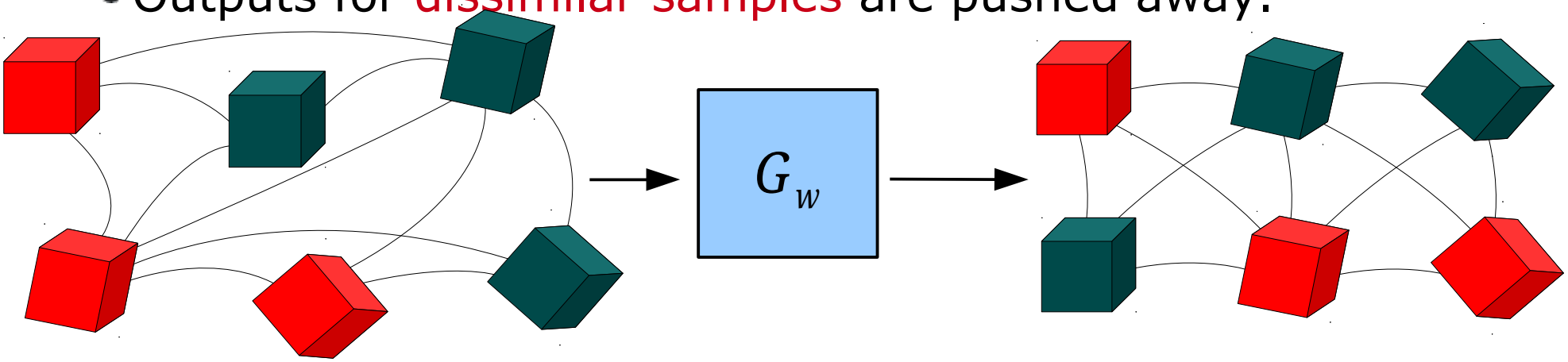
[Taylor, Fergus, Williams, Spiro, Bregler NIPS 2010]

[Taylor, Spiro, Bregler, Fergus, CVPR 2011]

DrLIM: Metric Learning

Dimensionality Reduction by Learning an Invariant Mapping

- **Step 1**: Construct neighborhood graph.
- **Step 2:** Choose a parameterized family of functions.
- **Step 3:** Optimize the parameters such that:
  - Outputs for similar samples are pulled closer.
  - Outputs for dissimilar samples are pushed away.



joint work with Sumit Chopra: Hadsell et al. CVPR 06; Chopra et al., CVPR 05

DrLIM: Contrative Loss function

Dimensionality Reduction by Learning an Invariant Mapping

- **Step 1**: Construct neighborhood graph.
- **Step 2:** Choose a parameterized family of functions.
- **Step 3:** Optimize the parameters such that:
  - Outputs for similar samples are pulled closer.
  - Outputs for dissimilar samples are pushed away.

- Loss function for inputs X_1 and X_2 with binary label Y and $D_W = ||G_W(X_1) - G_W(X_2)||_2$:

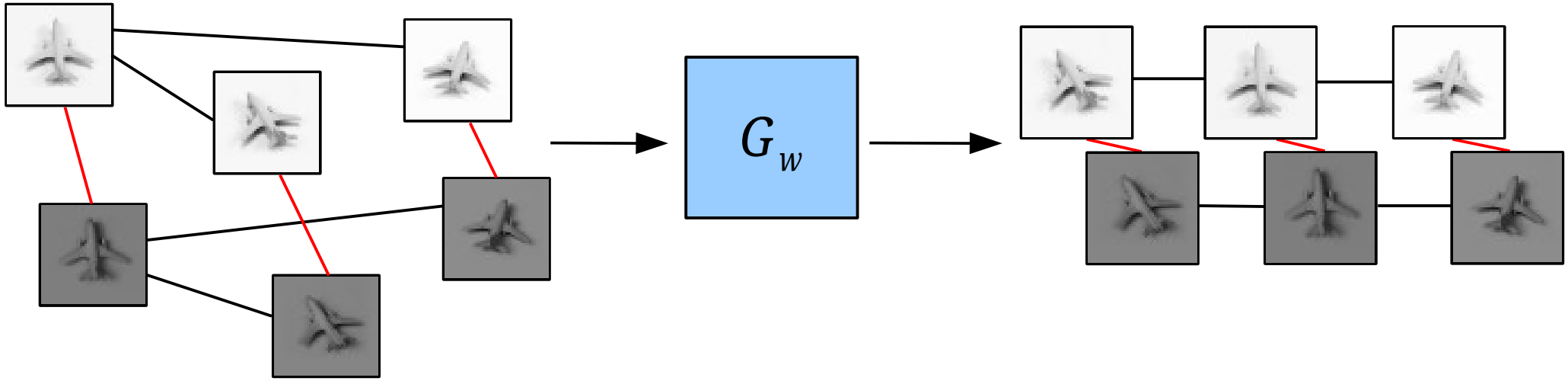
$$L(W, Y, \vec{X}_1, \vec{X}_2) = (1 - Y)\frac{1}{2}(D_W)^2 + (Y)\frac{1}{2}\{max(0, m - D_W)\}^2$$

joint work with Sumit Chopra: Hadsell et al. CVPR 06; Chopra et al., CVPR 05

DrLIM: Contrative Loss function

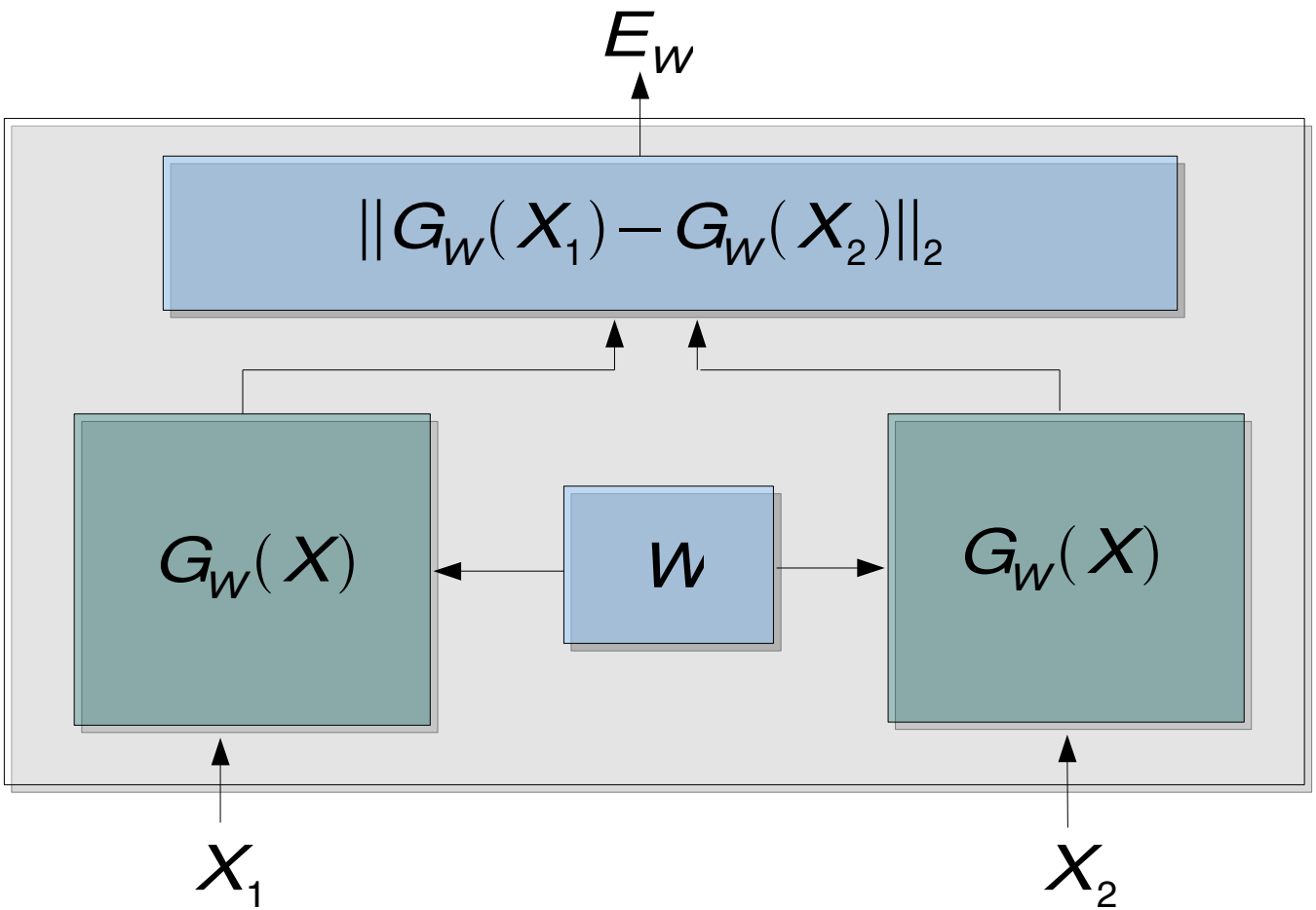
Dimensionality Reduction by Learning an Invariant Mapping

- **Step 1**: Construct neighborhood graph.
- **Step 2:** Choose a parameterized family of functions.
- **Step 3:** Optimize the parameters such that:
  - Outputs for similar samples are pulled closer.
  - Outputs for dissimilar samples are pushed away.



joint work with Sumit Chopra: Hadsell et al. CVPR 06; Chopra et al., CVPR 05

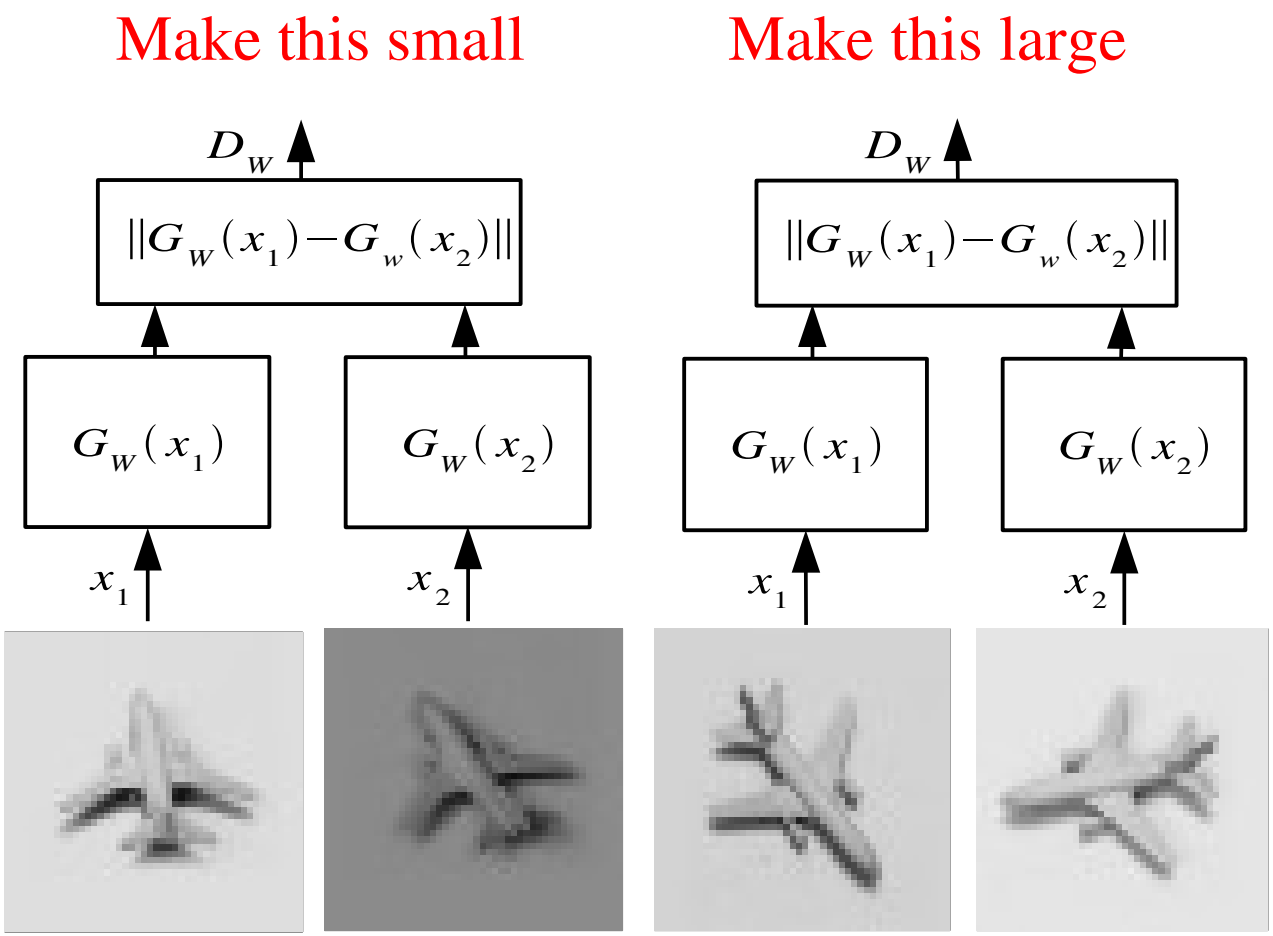
Siamese Architecture



Siamese Architecture [Bromley, Sackinger, Shah, LeCun 1994]

Siamese Architecture and loss function

- **Loss function:**
  - Outputs corresponding to input samples that are neighbors in the neigborhood graph should be nearby
  - Outputs for input samples that are not neighbors should be far away from each other



Similar images (neighbors in the neighborhood graph)

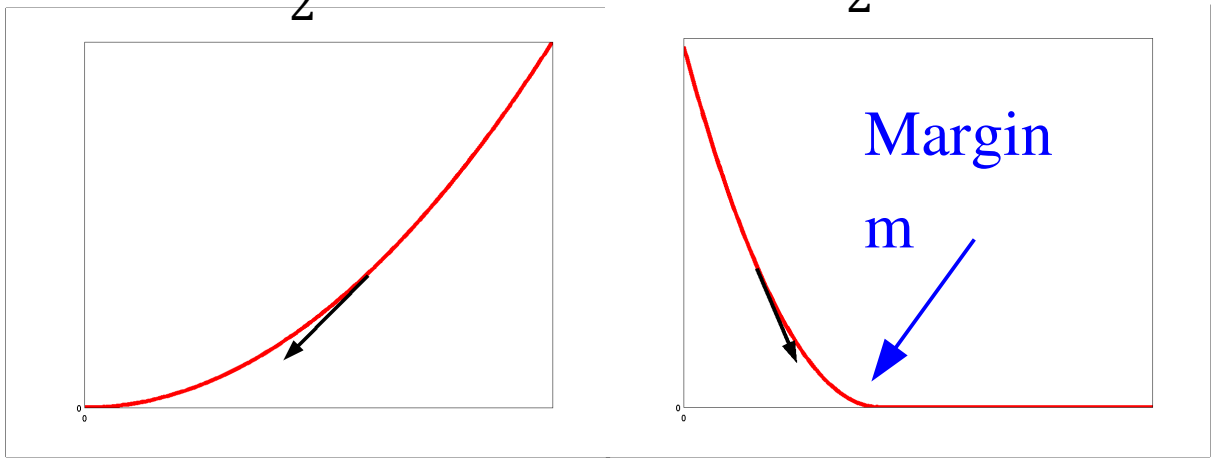
Dissimilar images (non-neighbors in the neighborhood graph)

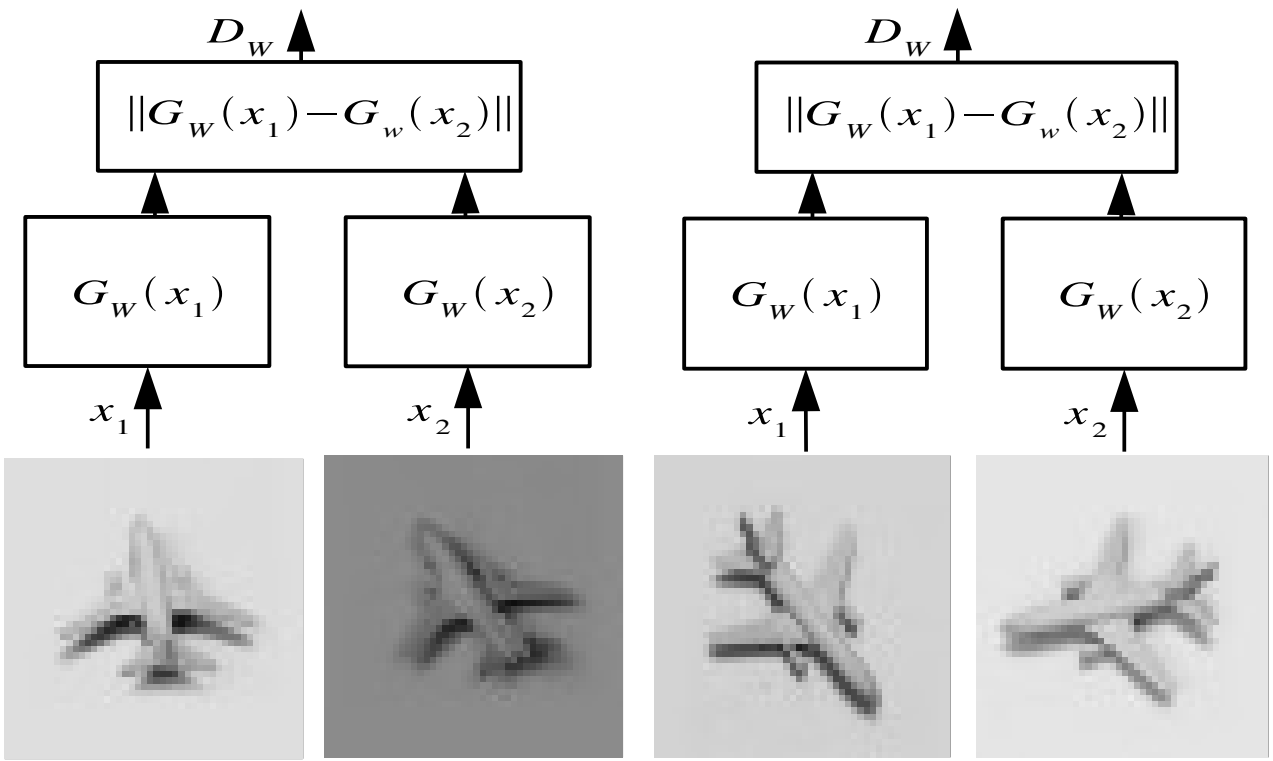
Loss function

- **Loss function:**
  - Pay quadratically for making outputs of neighbors far apart
  - Pay quadratically for making outputs of non-neigbors smaller than a **margin** m

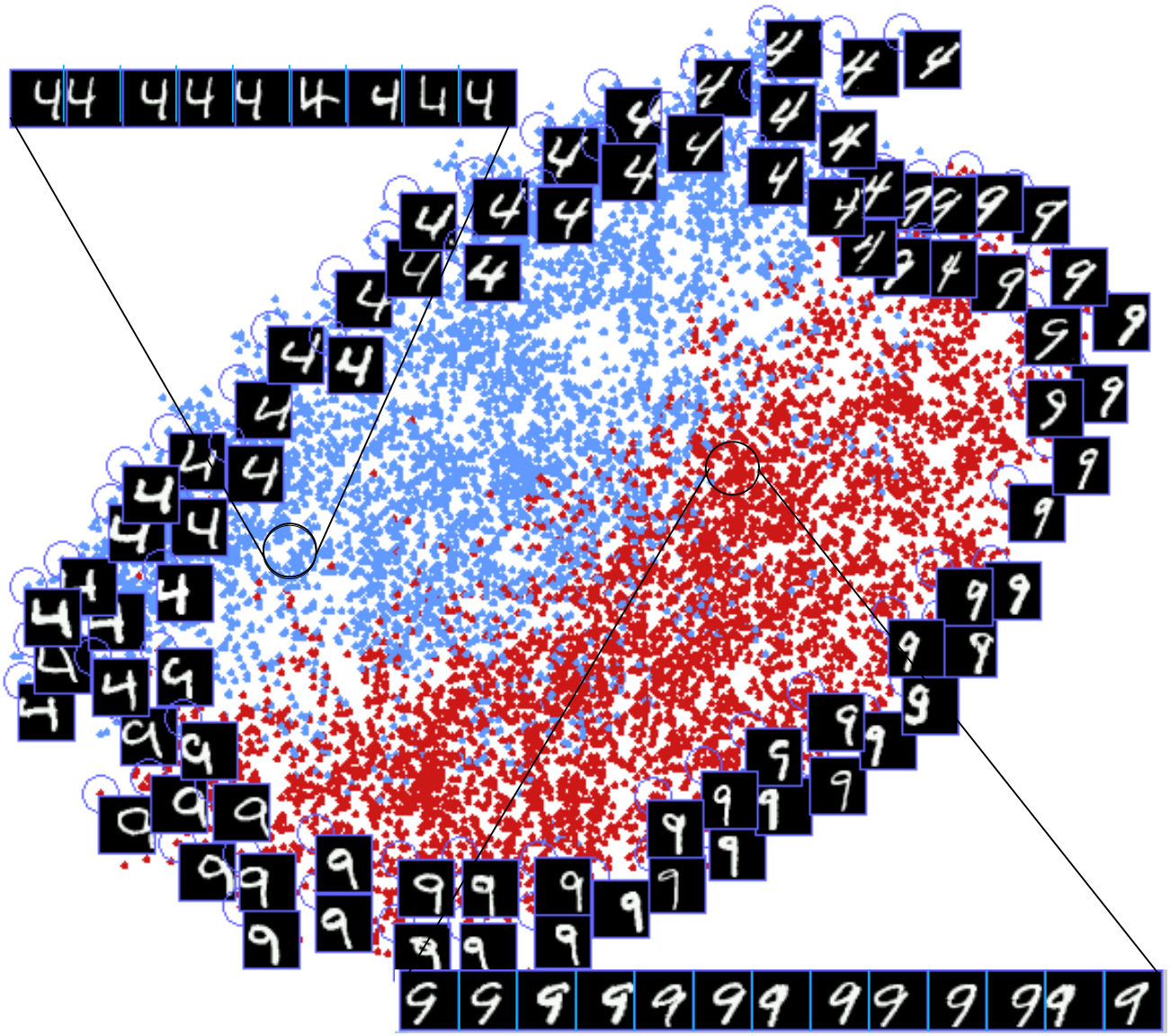
$$L_{similar} = \frac{1}{2} D_w^2$$

$$L_{dissimilar} = \frac{1}{2} \{ max(0, m - D_W) \}^2$$





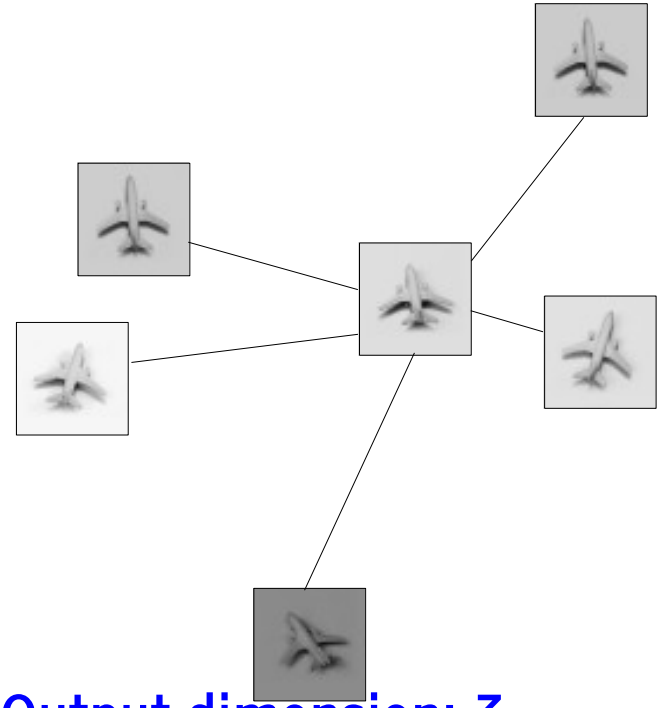
A Digit Manifold with Invariance to Shifts



- Training set: 3000 “4” and 3000 “9” from MNIST. **Each digit is shifted horizontally by -6, -3, 3, and 6 pixels**
- Neighborhood graph: 5 nearest neighbors in Euclidean distance, **and shifted versions of self and nearest neighbors**
- Output Dimension: 2
- Test set (shown) 1000 “4” and 1000 “9”

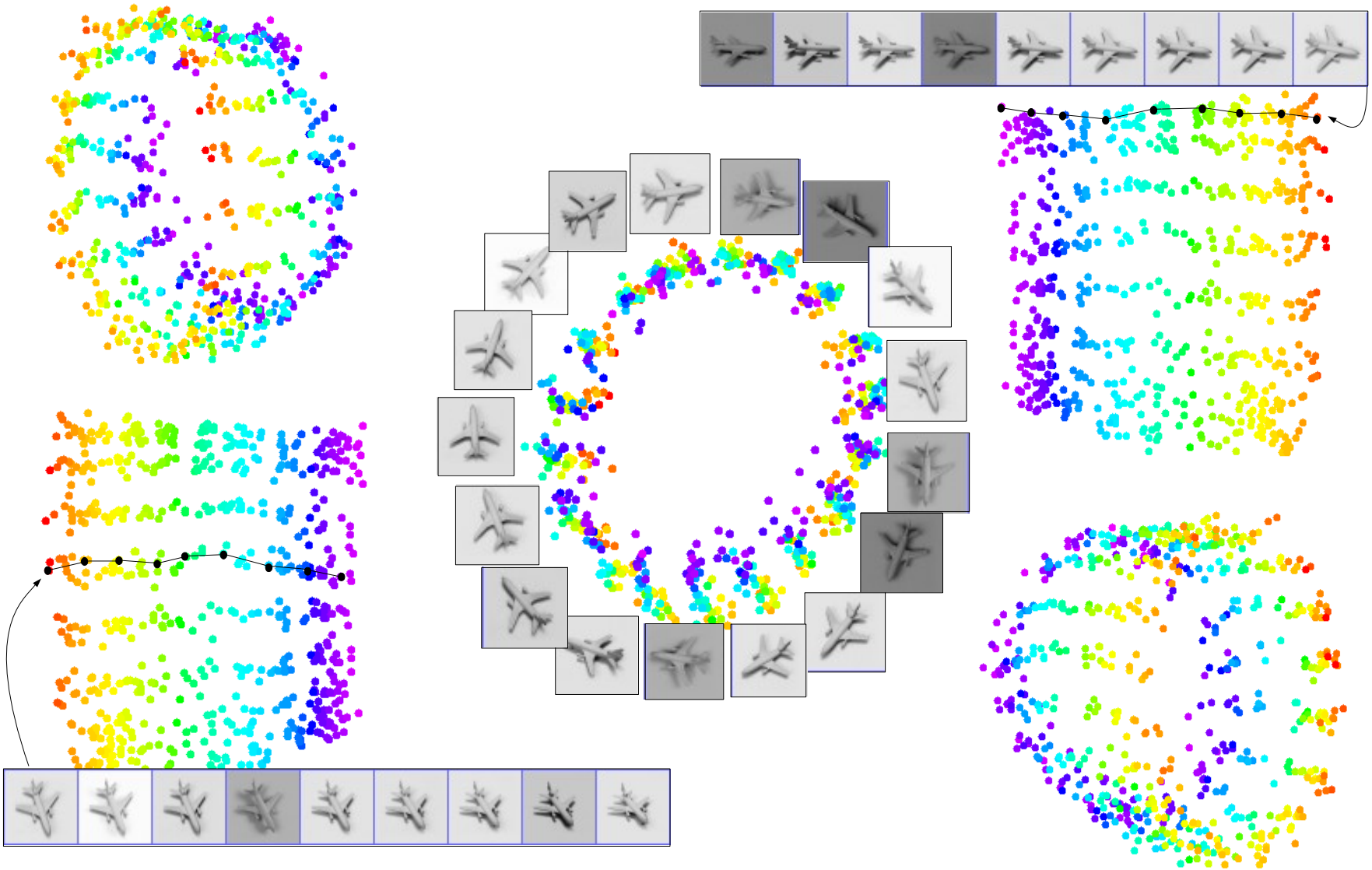
NORB dataset: Pose Manifold with lighting invariance

- **Objective:** Learn a mapping into 3-space that is invariant to lighting.
- **Dataset:** 972 images of a single object.
  18 azimuths,
  9 elevations,
  6 lighting conditions
- **Neighborhood graph:**
  - nearest neighbors in azimuth.
  - nearest neighbors in elevation.
  - Independent of lighting condition.



- **Architecture:** Input dimension: 48x48. Output dimension: 3.
  A 2-layer fully connected neural network.

NORB dataset – Automatic Discovery of the Viewpoint Manifold with Invariance to Lighting



Different views of a single manifold are shown

DrLIM: Learning a Face Manifold for Face Verification

dissimilar similar

dissimilar similar

Face Verification datasets: AT&T/ORL

- The AT&T/ORL dataset
- Total subjects: **40**. Images per subject: **10**. Total images: **400**.
- Images had a moderate degree of variation in pose, lighting, expression and head position.
- Images from **35** subjects were used for training. Images from **5** remaining subjects for testing.
- Training set was taken from: **3500** genuine and **119000** impostor pairs.
- Test set was taken from: **500** genuine and **2000** impostor pairs.
- http://www.uk.research.att.com/facedatabase.html

AT&T/ORL Dataset

Internal state for genuine and impostor pairs

Classification Examples

- **Example: Correctly classified genuine pairs**

energy: 0.3159 energy: 0.0043 energy: 0.0046

- **Example: Correctly classified impostor pairs**

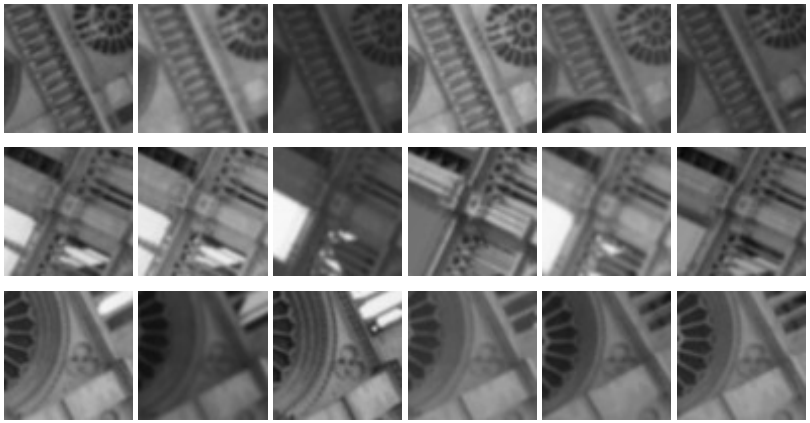
energy: 20.1259 energy: 32.7897 energy: 5.7186

- **Example: Mis-classified pairs**

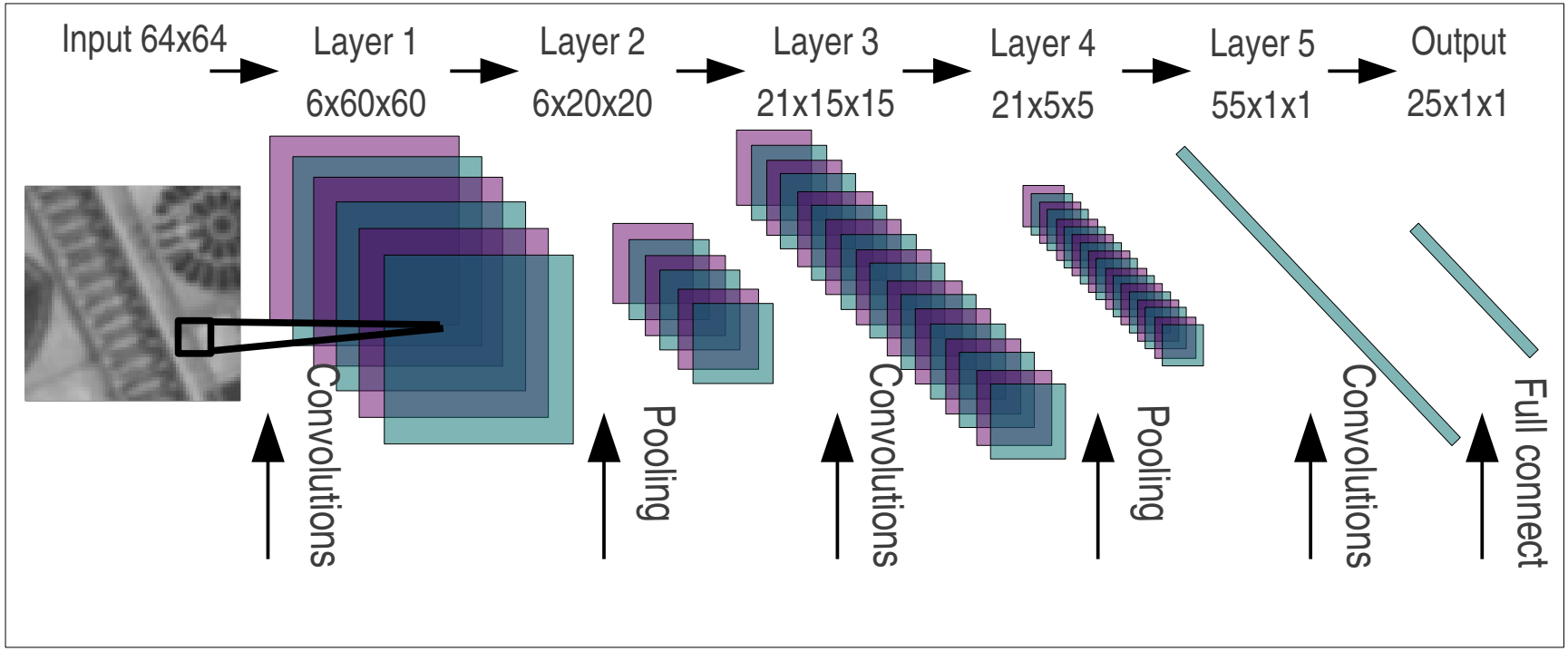
energy: 10.3209 energy: 2.8243

DrLIM for Invariant Feature Learning

[Raia Hadsell's PhD thesis, 2008]



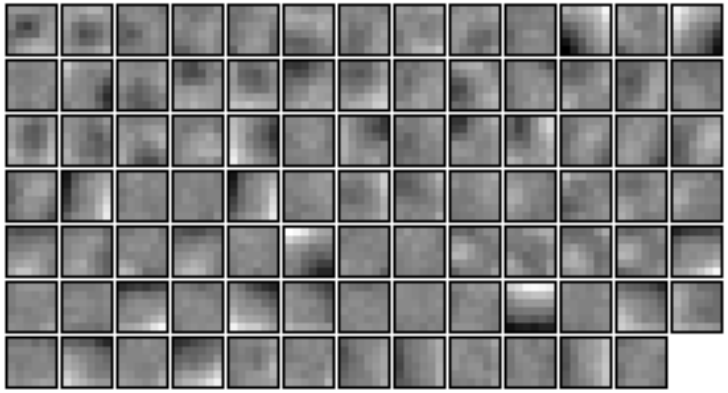
- **Co-location patch data**
  - multiple tourist photos
  - 3d reconstruction
  - groundtruth matches

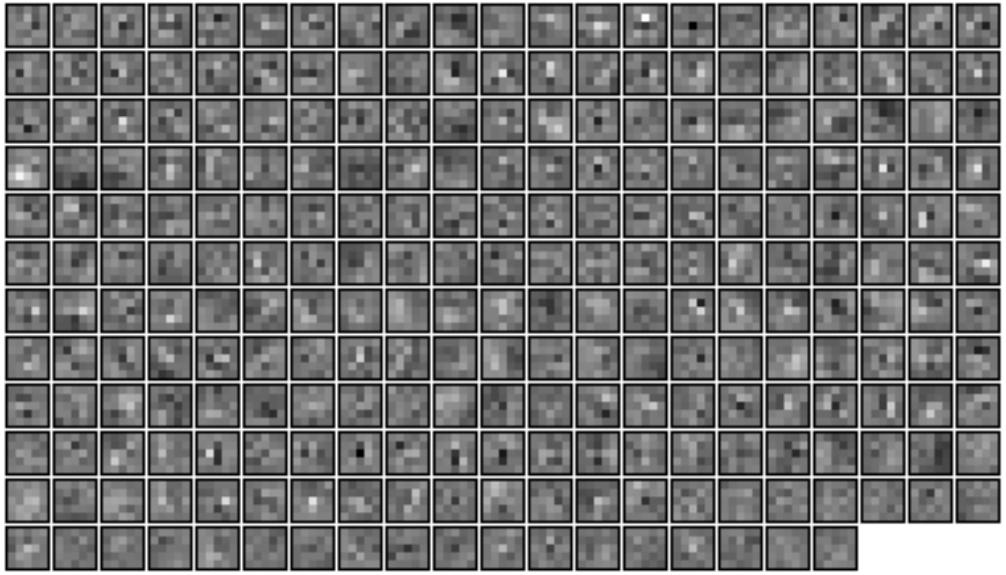


data from: Winder and Brown, CVPR 07

DrLIM for Invariant Feature Learning

- Error on test set at 95% detection rate:
  0.93%

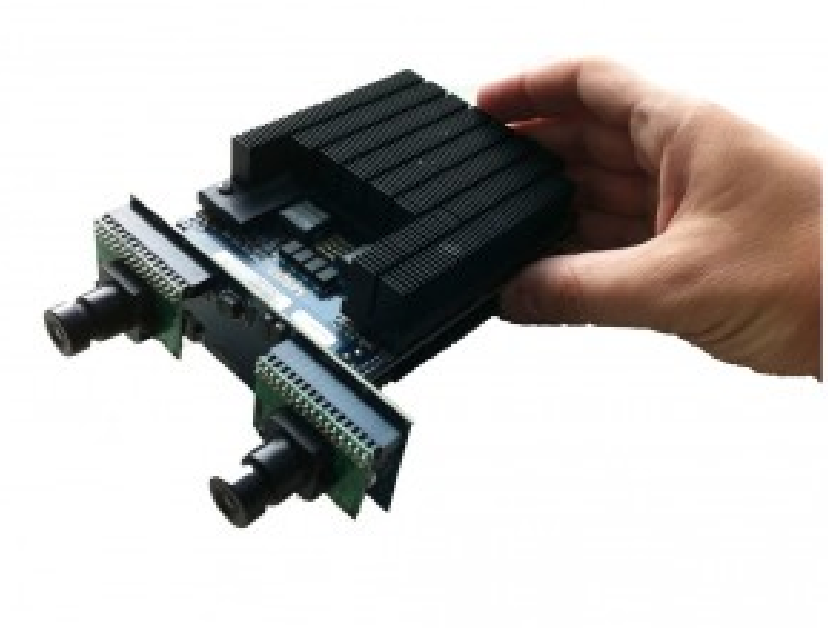




Hardware Implementations

Higher End: FPGA with NeuFlow architecture

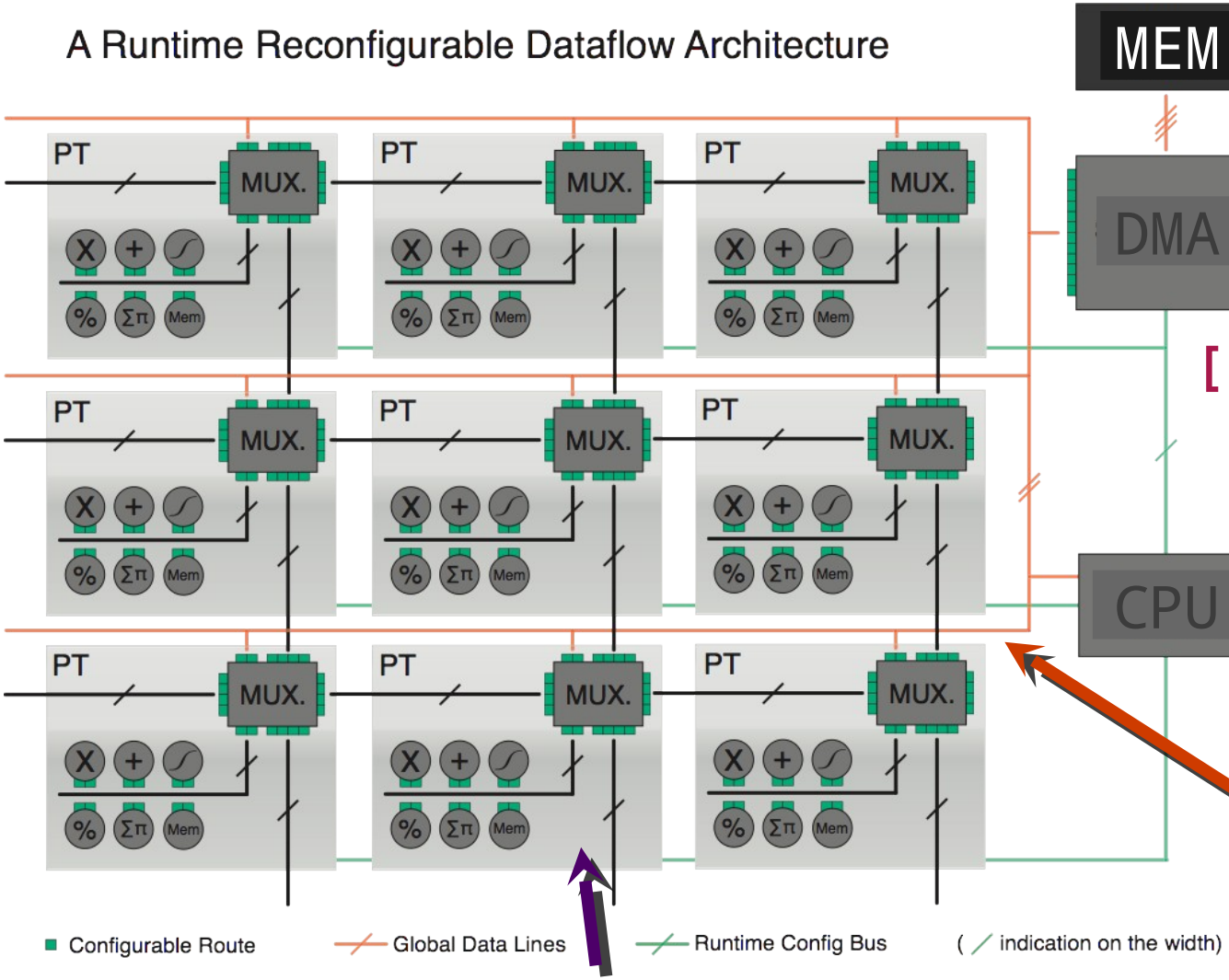
- Now Running on Picocomputing 8x10cm high-performance FPGA board
  - Virtex 6 LX240T: 680 MAC units, 20 neuflow tiles
- Full scene labeling at 20 frames/sec (50ms/frame) at 320x240



New board with Virtex-6

NewFlow: Architecture

A Runtime Reconfigurable Dataflow Architecture



Multi-port memory controller (DMA)

[x12 on a V6 LX240T]

RISC CPU, to reconfigure tiles and data paths, at runtime

global network-on-chip to allow fast reconfiguration

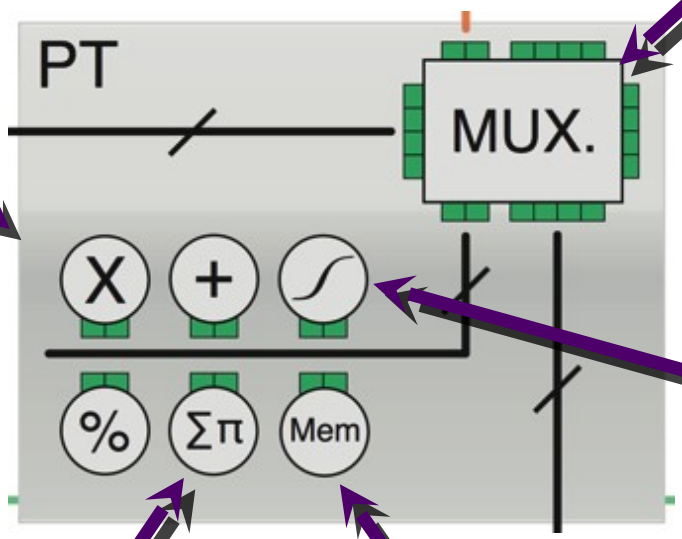
grid of passive processing tiles (PTs)

[x20 on a Virtex6 LX240T]

NewFlow: Processing Tile Architecture

Term-by:-term streaming operators (MUL,DIV,ADD,SUB,MAX)

[x8,2 per tile]



configurable router, to stream data in and out of the tile, to neighbors or DMA ports

[x20]

configurable piece-wise linear or quadratic mapper

[x4]

full 1/2D parallel convolver with 100 MAC units

[x4]

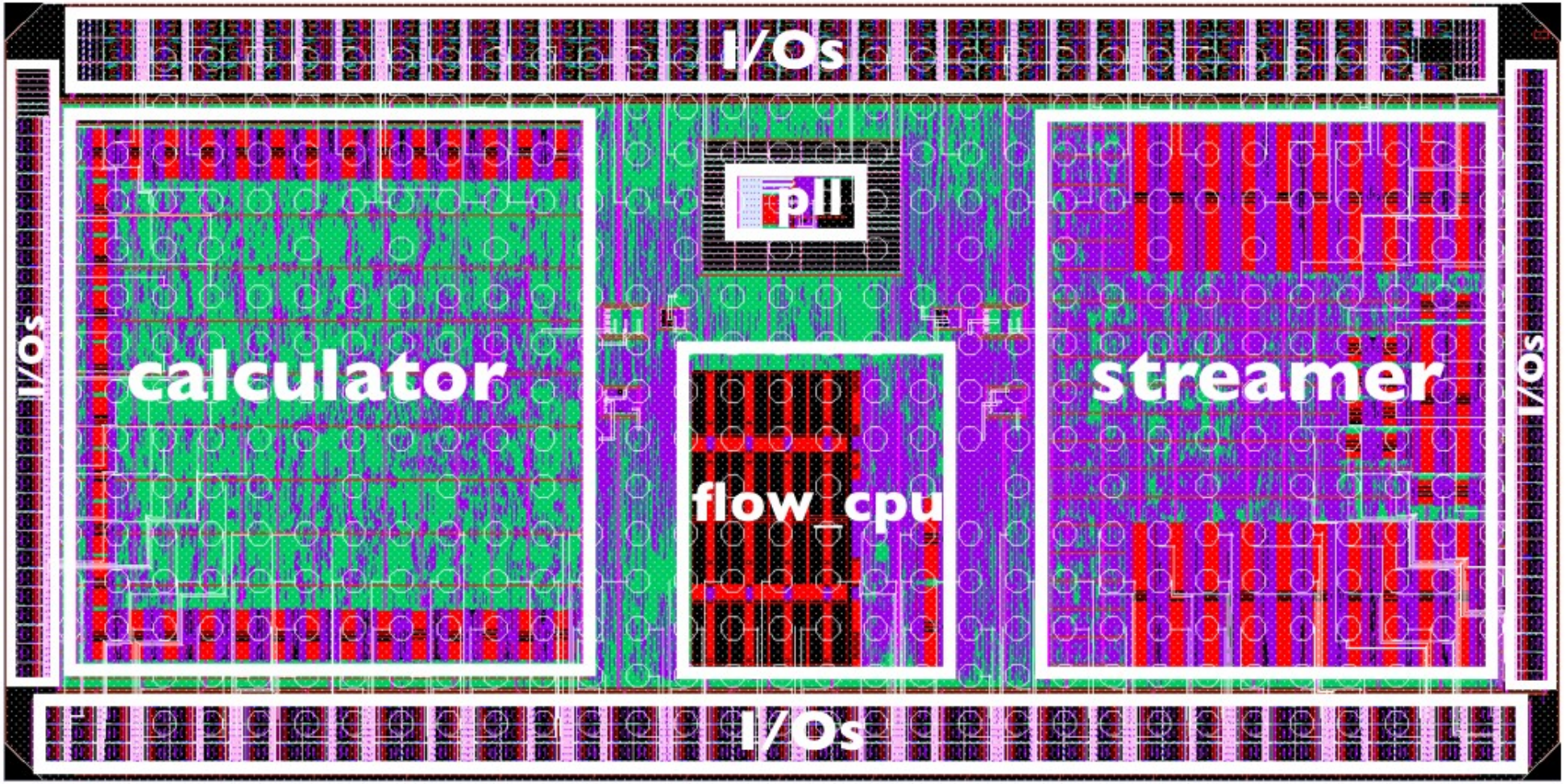
configurable bank of FIFOs , for stream buffering, up to 10kB per PT

[x8]

[Virtex6 LX240T]

NewFlow ASIC: 2.5x5 mm, 45nm, 0.6Watts, 160GOPS

- Collaboration NYU-Purdue (Eugenio Culurciello's group)
- Suitable for vision-enabled embedded and mobile devices
- Status: waiting for first samples end of 2012



Pham, Jelaca, Farabet, Martini, LeCun, Culurciello 2012]

NewFlow: Performance

	Intel I7 4 cores	neuFlow Virtex4	neuFlow Virtex 6	nVidia GT335m	neuFlow IBM 45nm	nVidia GTX480
Peak GOP/sec	40	40	160	182	160	1350
Actual GOP/sec	12	37	147	54	147	294
FPS	14	46	182	67	182	374
Power (W)	50	10	10	30	0.6	220
Embed? (GOP/s/W)	0.24	3.7	14.7	1.8	245	1.34

Software Platforms: Torch7 http://www.torch.ch

[Collobert, Kavukcuoglu, Farabet 2011]

Torch7

- **ML/Vision development environment**
  - Collaboration between IDIAP, NYU, and NEC Labs
  - Uses the Lua interpreter/compiler as a front-end
  - Very simple and efficient scripting language
  - Ultra simple and natural syntax
  - It's like Scheme underneath, but with a “normal” syntax.
  - Lua is widely used in the video game and web industries
- **Torch7 extend Lua with Numerical, ML, Vision libraries**
  - Powerful Vector/Matrix/Tensor Engine (developed by us)
  - Interfaces to numerical libraries, OpenCV, etc
  - Back-ends for SSE, OpenMP, CUDA, ARM-Neon,...
  - Qt-based GUI and graphics
  - Open source: http://www.torch.ch/
  - First released in early 2012

Torch7

- ML/Vision development environment
  - As simple as Matlab, but faster and better designed as a language.

```
2D CONVOLUTION
x = torch.rand(100,100)
k = torch.rand(10,10)
res1 = torch.conv2(x,k)
```

```
MATRIX and VECTOR Operations
> M = torch.Tensor(2,2):fill(2)
> N = torch.Tensor(2,4):fill(3)
> x = torch.Tensor(2):fill(4)
> y = torch.Tensor(2):fill(5)
> = x*y -- dot product
40
> = M*x --- matrix-vector

 16
 16
[torch.Tensor of dimension 2]
```

```
TRAINING A NEURAL NET
mlp = nn.Sequential()
mlp:add( nn.Linear(10, 25) ) -- 10 input, 25 hidden units
mlp:add( nn.Tanh() ) -- some hyperbolic tangent transfer function
mlp:add( nn.Linear(25, 1) ) -- 1 output
criterion = nn.MSECriterion() -- Mean Squared Error criterion
trainer = nn.StochasticGradient(mlp, criterion)
trainer:train(dataset) -- train using some examples
```

Acknowledgements

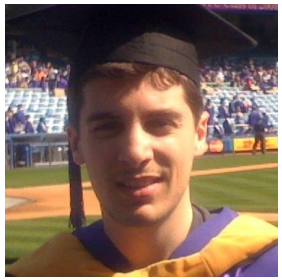
- Y-Lan Boureau

- Kevin Jarrett

- Koray Kavukcuoglu

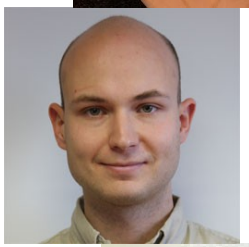
- Marc'Aurelio Ranzato

- Pierre Sermanet

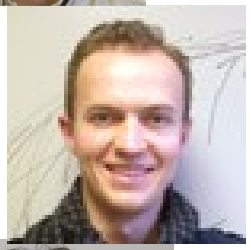


- Camille Couprie

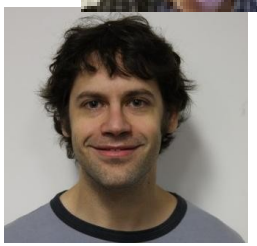
- Karol Gregor



- Clément Farabet



- Arthur Szlam



- Rob Fergus

- Laurent Najman (ESIEE)

- Eugenio Culurciello (Purdue)

The End