Outline

1 Introduction
   - Feature Learning
   - Correspondence in Computer Vision
   - Relational feature learning

2 Learning relational features
   - Sparse Coding Review
   - Encoding relations
   - Inference
   - Learning

3 Factorization, eigen-spaces and complex cells
   - Factorization
   - Eigen-spaces, energy models, complex cells

4 Applications
   - Applications
   - Conclusions
Outline

1 Introduction
   - Feature Learning
   - Correspondence in Computer Vision
   - Relational feature learning

2 Learning relational features
   - Sparse Coding Review
   - Encoding relations
   - Inference
   - Learning

3 Factorization, eigen-spaces and complex cells
   - Factorization
   - Eigen-spaces, energy models, complex cells

4 Applications
   - Applications
   - Conclusions
The number of parameters is about $n \times n \times n$ (!)
More, if we want sparse, overcomplete hiddens.
There is a simple, yet far-reaching, way to reduce that number.
Factorization

\[ w_{ijk} = \sum_{ij} \sum_{f} w_{if}^x w_{jf}^y w_{kf}^z \]
Factorization is \textit{filter matching}

\[
\begin{align*}
    z_k &= \sum_{ij} w_{ijk} x_i y_j \\
    &= \sum_{ij} \left( \sum_{f} w^x_{if} w^y_{jf} w^z_{kf} \right) x_i y_j \\
    &= \sum_{f} w^y_{jf} \cdot \left( \sum_{i} w^x_{if} x_i \right) \cdot \left( \sum_{j} w^y_{kf} y_j \right)
\end{align*}
\]
Factorization is *filter matching*

\[
E = \sum_{ijk} \left( \sum_{f} w_{i,f}^x w_{j,f}^y w_{k,f}^z \right) x_i y_j z_k = \sum_{f} \left( \sum_{i} w_{i,f}^x x_i \right) \left( \sum_{j} w_{j,f}^y y_j \right) \left( \sum_{k} w_{k,f}^z z_k \right)
\]
Factorized Gated Boltzmann machines

- Exponentiate and normalize energy (just like RBM).
- Learning and inference exactly like before.
- (Taylor, 2009), (Memisevic, Hinton; 2009)
Factored Relational Autoencoders

- Again, everything like before. Back-propagate through the filters.
- Conditional learning trivial.
- Joint learning by adding two asymmetric objectives.
Square pooling models

Square pooling:

Another way to learn filter matching models are square pooling models, for example:

- ASSOM (Kohonen, 1996)
- ISA (Hyvarinen, 2000)
- Product of T-distributions (Osindero et al., 2006)
- (Karklin, Lewicki; 2008)
- cRBM (Ranzato et al., 2009)

Often, $W^z$ is constrained so each hidden sees only a few squared inputs. That way hiddens can be thought of as encoding subspace norms.
Square pooling models

Square pooling:

- Why is square pooling the same?

The activity that a hidden unit gets is:

$$\sum_f w^z_k f (W^x f^T x + W^y f^T y)^2$$

$$= \sum_f w^z_k f (2 (W^x f^T x) (W^y f^T y) + (W^x f^T x)^2 + (W^y f^T y)^2)$$

Roland Memisevic (Uni Frankfurt)  
Multiview Feature Learning  
Tutorial at CVPR 2012  
79 / 174
Square pooling:

- Why is square pooling the same?

The activity that a hidden unit gets is:

$$\sum_f w^z_{k_f} (W^x_f \mathbf{x} + W^y_f \mathbf{y})^2$$

$$= \sum_f w^z_{k_f} \left(2(W^x_f \mathbf{x})(W^y_f \mathbf{y}) + (W^x_f \mathbf{x})^2 + (W^y_f \mathbf{y})^2\right)$$
Square pooling: Learning is somewhat more difficult than with factored gated feature learning.

Example ISA: Gradient-based, while enforcing $W_{xy}^T W_{xy} = I$ after every gradient step (eigen-decomposition).
Toy examples:
There is no structure in these images.
Only in *how they change.*
Learned filters $w^x_{if}$
Learned filters $w_{ij}^y$
Frequency/orientation histograms

combined (freq, orient) usage of all filters by channel (left/right)
Frequency/orientation histograms
Velocity tuning of mapping units

Roland Memisevic (Uni Frankfurt)

Multiview Feature Learning

Tutorial at CVPR 2012

Filters learned from split-screen shifts
Affine filters
“Filtering” - filters
Rotation filters
Rotation filters
Rotation filters
Filters learned by watching TV
Filters learned by watching TV
“Bag-Of-Warps”
Outline

1 Introduction
   - Feature Learning
   - Correspondence in Computer Vision
   - Relational feature learning

2 Learning relational features
   - Sparse Coding Review
   - Encoding relations
   - Inference
   - Learning

3 Factorization, eigen-spaces and complex cells
   - Factorization
   - Eigen-spaces, energy models, complex cells

4 Applications
   - Applications
   - Conclusions

Roland Memisevic (Uni Frankfurt)
Linear image warps

Consider a linear transformation in pixel space ("warp"):

\[ y = Lx \]

Now consider the following task:

Given two images \( x, y \), what is the warp that relates them?

This is exactly the problem that mapping units should be able to solve.
Orthogonal image warps

$$y = Lx$$

- We restrict our attention to **orthogonal warps** in the following, that is:
  $$L^TL = I$$

- These include all permutations (“shuffling pixels”).
- Orthogonal warps are the *only* transformations we can see anyway, if all our images are *white*:
  $$I = Cy = LC_xL^T = LL^T$$

  (Bethge, 2007)

- To get a better understanding of what mapping units really do, we make use of two properties of orthogonal image warps:
Properties of orthogonal image warps

(I) Orthogonal transformations decompose into 2-D rotations

- An orthogonal matrix is similar to a matrix that performs axis-aligned two-dimensional rotations:

\[
V^T L V = \begin{bmatrix}
R_1 \\
\vdots \\
R_k
\end{bmatrix}
\]

\[
R_i = \begin{bmatrix}
\cos(\theta_i) & -\sin(\theta_i) \\
\sin(\theta_i) & \cos(\theta_i)
\end{bmatrix}
\]

- This follows, for example, from the fact that the eigen-decomposition

\[
L = V D V^T
\]

has complex eigenvalues of length 1.

- The eigenspaces are also known as **invariant subspaces**.
Example: Translation and the Fourier spectrum

- **Translation** is an example of an orthogonal warp.
- 1-D translation matrices are *circulants*, which have ones along an off-diagonal, like so:

\[
L = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

- The two-dimensional eigen-features of this matrix turn out to be sine-/cosine-pairs (Fourier features).
Properties of orthogonal image warps

Quadrature pairs

- Since the invariant subspaces of orthogonal warps are two-dimensional, eigenvectors come in pairs:

\((v_R, v_I)\)

They form an orthogonal basis for the invariant subspace.

- In the case of translation, \(v_I\) is a sine and \(v_R\) is a cosine feature.

- Waves with 90 degrees phase difference are known as “quadrature pair”.

- But the concept is more general and applies to all orthogonal matrices.

- The eigenvector pairs of orthogonal transformations have been referred to as “generalized quadrature pairs” (Bethge et al., 2007).
(II) Commuting transformations share an eigen-basis

- Any two transformations that commute share a single eigen-basis.
- They differ only in their eigenvalues.

“Proof”: Consider $A$ and $B$ with $AB = BA$ and the eigenvector $v$ of $B$ with $\lambda$ an eigenvalue with multiplicity one. We have

$$BAv = ABv = \lambda Av.$$ 

So $Av$ is also an eigenvector of $B$ with the same eigenvalue.
Properties of commuting image warps

Translation Example continued

- All circulants have the Fourier basis as eigen-basis.

- Properties (I) and (II) taken together now allow us to state the following:
Properties of commuting image warps

Any two orthogonal, commuting transformations differ only with respect to the rotation angles in the eigenpaces.

So to *apply* a transformation you can equivalently perform a set of independent two-D rotations.

\[ y = Lx \]
Properties of commuting image warps

Any two orthogonal, commuting transformations differ only with respect to the rotation angles in the eigenpaces.

- So to apply a transformation you can equivalently perform a set of independent two-D rotations.

\[ x \Rightarrow y = Lx \]

- To infer the transformation, given two images \( x \) and \( y \): Project \( x \) and \( y \) onto the eigenvectors, then compute the rotation angles!
Properties of commuting image warps

Any two orthogonal, commuting transformations differ only with respect to the rotation angles in the eigenpaces.

- So to apply a transformation you can equivalently perform a set of independent two-D rotations.

\[ y = Lx \]

- To infer the transformation, given two images \( x \) and \( y \): Project \( x \) and \( y \) onto the eigenvectors, then compute the rotation angles!
In each subspace:
- Normalize the 2-D projections to unit norm, then read off the angle between them.
In each subspace:
- Normalize the 2-D projections to unit norm, then read off the angle between them.
In each subspace:

- Normalize the 2-D projections to unit norm, then read off the angle between them.
In each subspace:

- Normalize the 2-D projections to unit norm, then read off the angle between them.
Extracting sub-space rotations, naive approach

To read off the angle, compute the inner product (after normalizing projections to unit-norm).

Formally,

$$\cos(\phi_y - \phi_x) = \cos \phi_y \cos \phi_x + \sin \phi_y \sin \phi_x$$

$$= (v_R^T y)(v_R^T x) + (v_I^T y)(v_I^T x)$$

Compute the sum over products of filter responses.
Extracting sub-space rotations, naive approach

To read off the angle, compute the **inner product** (after normalizing projections to unit-norm).

Formally,

$$\cos(\phi_y - \phi_x) = \cos \phi_y \cos \phi_x + \sin \phi_y \sin \phi_x$$

$$= (v_R^T y)(v_R^T x) + (v_I^T y)(v_I^T x)$$

Compute the sum over products of filter responses.
Sub-space rotation detectors

- Normalizing to unit norm can be a bad idea, if projections are small:

### The aperture problem

- Consider the left shift of a horizontal bar.
- It is impossible to see the transformation in this case.
- This is known as the **aperture problem**.
- Normalizing subspace projections would amount to pretending we could see the transformation!

- A second way to get the rotations:
  - Absorb the rotation into one of the eigenvectors, then try to **detect** rotation angles.
Sub-space rotation detectors

Extracting rotations by detecting angles

Formally, let the output filter pair $v_\theta R, v_\theta I$ be the input filter rotated by $\theta$ degrees (in complex notation: $v_\theta = \exp(i\theta)v$).

Measure how well the image pair $x, y$ conforms with this rotation:

$$r_\theta := \cos(\phi_y - \phi_x - \theta) = \cos(\phi_y) \cos(\phi_x + \theta) + \sin(\phi_y) \sin(\phi_x + \theta) = (v_\theta R^Ty)(v_R^Tx) + (v_\theta I^Ty)(v_I^Tx)$$

Again we have to sum over products of filter responses.
Sub-space rotation detectors

Formally, let the output filter pair $v_{θ_R}, v_{θ_I}$ be the input filter rotated by $θ$ degrees (in complex notation: $v_{θ} = \exp(iθ)v$).

Measure how well the image pair $x, y$ conforms with this rotation:

$$r_θ := \cos(φ_y - φ_x - θ) = \cos(φ_y) \cos(φ_x + θ) + \sin(φ_y) \sin(φ_x + θ) = (v_{θ_R}^T y)(v_{R}^T x) + (v_{θ_I}^T y)(v_{I}^T x)$$

Again we have to sum over products of filter responses.
Sub-space rotation detectors

Extracting rotations by detecting angles

Formally, let the output filter pair $v_{\theta}^R, v_{\theta}^I$ be the input filter rotated by $\theta$ degrees (in complex notation: $v_{\theta} = \exp(i\theta)v$).

Measure how well the image pair $x, y$ conforms with this rotation:

$$r_{\theta} := \cos(\phi_y - \phi_x - \theta) = \cos(\phi_y) \cos(\phi_x + \theta) + \sin(\phi_y) \sin(\phi_x + \theta)$$

$$= (v_{\theta}^R)^T y (v_{\theta}^I)^T x + (v_{\theta}^I)^T y (v_{\theta}^R)^T x$$

Again we have to sum over products of filter responses.
Sub-space rotation detectors

Extracting rotations by detecting angles

Formally, let the output filter pair $v_{θ R}, v_{θ I}$ be the input filter rotated by $θ$ degrees (in complex notation: $v_{θ} = \exp(iθ)v$).

Measure how well the image pair $x, y$ conforms with this rotation:

$$r_θ := \cos(φ_y - φ_x - θ) = \cos(φ_y) \cos(φ_x + θ) + \sin(φ_y) \sin(φ_x + θ) = (v_{θ R}^T y)(v_{R}^T x) + (v_{θ I}^T y)(v_{I}^T x)$$

Again we have to sum over products of filter responses.
Extracting rotations by detecting angles

- Formally, let the output filter pair $v^\theta_R, v^\theta_I$ be the input filter rotated by $\theta$ degrees (in complex notation: $v^\theta = \exp(i\theta)v$).
- Measure how well the image pair $x, y$ conforms with this rotation:

\[
\begin{align*}
    r^\theta & := \cos(\phi_y - \phi_x - \theta) \\
    &= \cos(\phi_y) \cos(\phi_x + \theta) + \sin(\phi_y) \sin(\phi_x + \theta) \\
    &= (v^\theta_R^T y)(v^T_R x) + (v^\theta_I^T y)(v^T_I x)
\end{align*}
\]

- Again we have to sum over products of filter responses.
For each subspace, we will need several mapping units, each tuned to a different angle, $\theta_i$.

The set of mapping unit responses will now constitute a **population code** that represents the observed transformation.

A mapping unit is *conservative*: It fires only if a transform is present *and* if it is visible in the image pair.
But the aperture problem causes another problem:
Take a video showing translations and generate two copies:
Low-pass filter each frame in the first; High-pass filter each frame in the second.
Now the transformation will be visible only in some components in the first and in other components in the second video.
These subspace features are content-dependent!
The solution:

Let hiddens pool within \textit{and} pool across subspaces.

This is exactly the factored bilinear model.
The cross-correlation model

- A hidden variable that computes the sum over products of filter responses can detect rotations, $\theta$, in an invariant subspace.
- To reconstruct the transformed output from the input image, it has to pool over multiple 2-dimensional subspaces.
- The population code of such hidden units is a good code for image transformations.
- Learning requires contrast normalization + keeping the scales of filters roughly the same!
Learning quadrature features

We can see the quadrature features, if we outsource the across-subspace pooling into a separate layer.
Learning quadrature Features
Learning quadrature Features
Rotation “quadrature” filters
Rotation “quadrature” filters
Mixed transformations
Mixed transformations
Quadrature features from natural video
Quadrature features from natural video
Energy models

\[ z_k = \sum_f w_{fk} (u_f^T x + v_f^T y)^2 \]

\[ = 2 \sum_f w_{fk} (u_f^T x)(v_f^T y) \]

\[ + \sum_f w_{fk} (u_f^T x)^2 + \sum_f w_{fk} (v_f^T y)^2 \]

- When we apply energy models to the concatenation of two images, we add square terms in inference.
- This may make the rotation detectors more conservative. Otherwise inference is the same!
Energy models

When we apply energy models to the concatenation of two images, we add square terms in inference.

This may make the rotation detectors more conservative. Otherwise inference is the same!
The energy model

- (Adelson and Bergen, 1985): Motion
- (Ozhawa, DeAngelis, Freeman; 1990): Disparity
- Equivalence to cross-correlation: See, for example, (Fleet et al.; 1994).
What happens when we train energy models on movies?

Hiddens receive all pairs of products between filters applied to frames.

So they detect the **repeated application of the same eigenvalue**:

\[
\left( \sum_s v^s x_s \right)^2 = \sum_s \left( v^s x_s \right)^2 + \sum_{st} \left( v^s x_s \right) \cdot \left( v^t x_t \right)
\]
Learning energy models on movies

What happens when we train energy models on movies?
Hiddens receive all pairs of products between filters applied to frames.
So they detect the repeated application of the same eigenvalue:

\[
\sum_s (v^s x_s)^2 = \sum_s (v^s x_s)^2 + \sum_{st} (v^s x_s) \cdot (v^t x_t)
\]
We can train a cross-correlation model via the energy mechanism.

But we can do the opposite, too:

Plug in *the same* data left and right and tie left and right filters.

So we don’t have to use ISA or PoT to train energy models.
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
A covariance encoder trained on movies
Learning cross-correlation and energy models

Take-home message, factored model

To learn about transformation, let hidden units pool over products of filter responses (gated feature learning) or pool over squares of sums of filter responses (energy model).
A bag of tricks

Tricks for learning:

- Normalize filters during learning, so they grow slowly, and they grow together: Normalize with a running average of the average filter norms.
- Connect top-level hiddens locally to the factors.
- Probably even better: make them locally overlapping ("Topographic ICA").
- DC-centering and contrast-normalization for each patch.
- Plus: Whiten the data before learning, using PCA or ZCA.
- Fast learning: large data-sets essential (use GPU’s...).
Tricks for learning:

- Normalize filters during learning, so they *grow slowly*, and they *grow together*: Normalize with a running average of the average filter norms.

- Connect top-level hiddens *locally* to the factors.

- Probably even better: make them *locally overlapping* ("Topographic ICA").

- *DC-centering and contrast-normalization* for each patch.

- Plus: *Whiten the data* before learning, using PCA or ZCA.

- Fast learning: large data-sets essential (use GPU’s...).
A bag of tricks

Tricks for learning:

- Normalize filters during learning, so they grow slowly, and they grow together: Normalize with a running average of the average filter norms.
- Connect top-level hiddens locally to the factors.
- Probably even better: make them locally overlapping ("Topographic ICA").
- DC-centering and contrast-normalization for each patch.
- Plus: Whiten the data before learning, using PCA or ZCA.
- Fast learning: large data-sets essential (use GPU’s...).
A bag of tricks

**Tricks for learning:**

- Normalize filters during learning, so they *grow slowly*, and they *grow together*: Normalize with a running average of the average filter norms.

- Connect top-level hiddens *locally* to the factors.

- Probably even better: make them *locally overlapping* (“Topographic ICA”).

- *DC-centering and contrast-normalization* for each patch.

- Plus: *Whiten the data* before learning, using PCA or ZCA.

- Fast learning: large data-sets essential (use GPU’s...).
Tricks for learning:

- Normalize filters during learning, so they *grow slowly*, and they *grow together*: Normalize with a running average of the average filter norms.
- Connect top-level hiddens *locally* to the factors.
- Probably even better: make them *locally overlapping* ("Topographic ICA").
- *DC-centering and contrast-normalization* for each patch.
- **Plus:** *Whiten the data* before learning, using PCA or ZCA.
- Fast learning: large data-sets essential (use GPU’s...).
Tricks for learning:

- Normalize filters during learning, so they grow slowly, and they grow together: Normalize with a running average of the average filter norms.
- Connect top-level hiddens locally to the factors.
- Probably even better: make them locally overlapping (“Topographic ICA”).
- DC-centering and contrast-normalization for each patch.
- Plus: Whiten the data before learning, using PCA or ZCA.
- Fast learning: large data-sets essential (use GPU’s...).