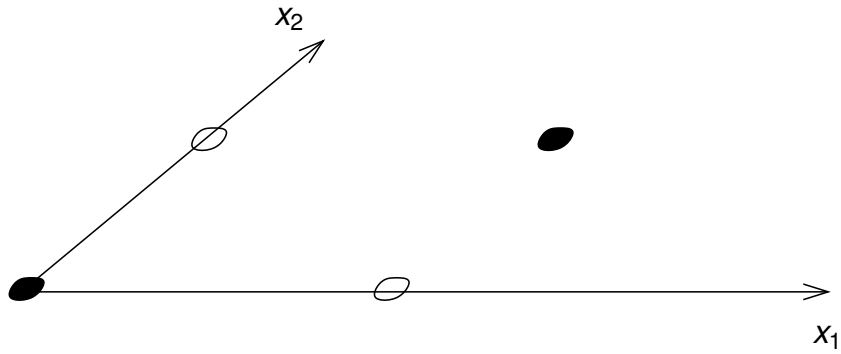
Machine learning for vision

Fall 2013

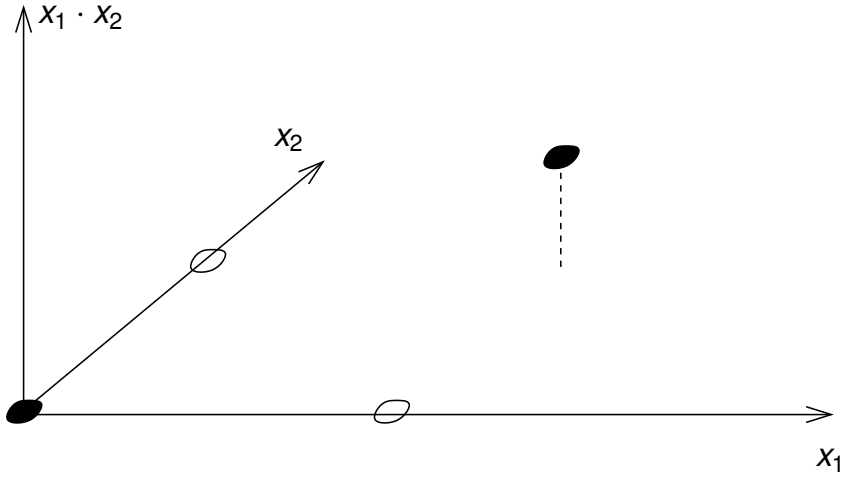
Roland Memisevic

Lecture 9, February 25, 2015

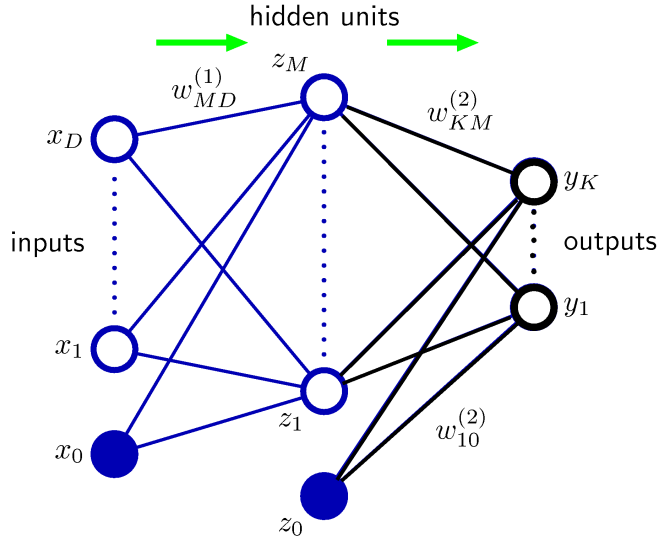
The XOR problem



The XOR problem



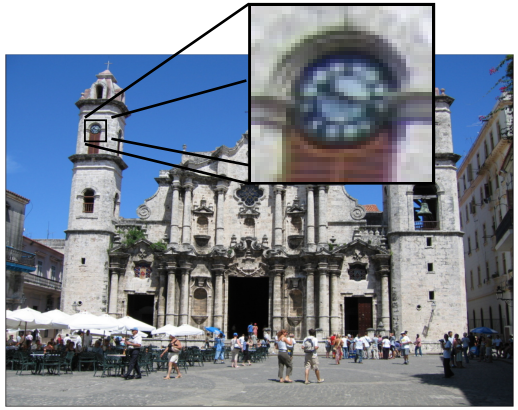
The XOR problem



(picture adapted from Bishop 2006)

“It’s the features, stupid”

“It’s the features, stupid”

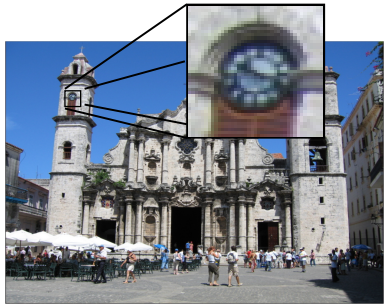


“It’s the features, stupid”

The common vision pipeline (prior 2012)

1. Find **interest points**.
2. Crop patches around them.
3. Represent each patch with a **sparse local descriptor**.
4. **Combine** the descriptors into a representation for the

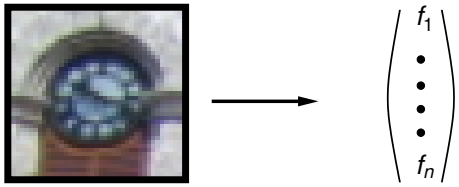
“It’s the features, stupid”



The common vision pipeline (prior 2012)

1. Find **interest points**.
2. Crop patches around them.
3. Represent each patch with a **sparse local descriptor**.
4. **Combine** the descriptors into a representation for the

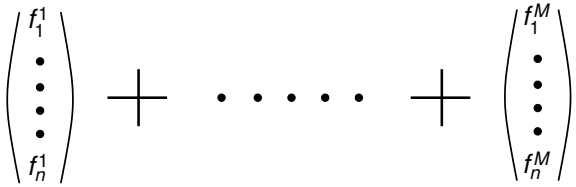
“It’s the features, stupid”



The common vision pipeline (prior 2012)

1. Find **interest points**.
2. Crop patches around them.
3. Represent each patch with a **sparse local descriptor**.
4. **Combine** the descriptors into a representation for the

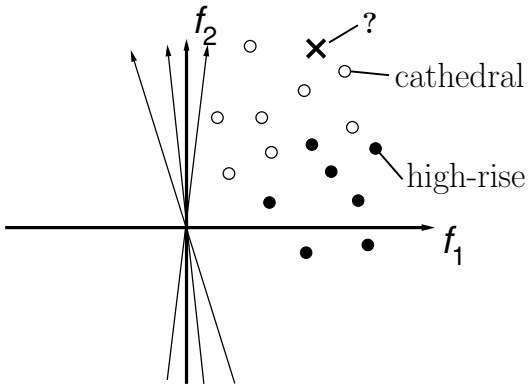
“It’s the features, stupid”



The common vision pipeline (prior 2012)

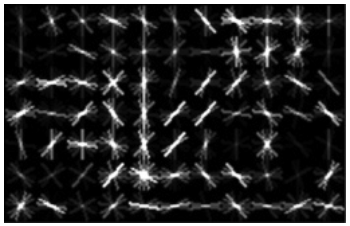
1. Find **interest points**.
2. Crop patches around them.
3. Represent each patch with a **sparse local descriptor**.
4. **Combine** the descriptors into a representation for the

“It’s the features, stupid”



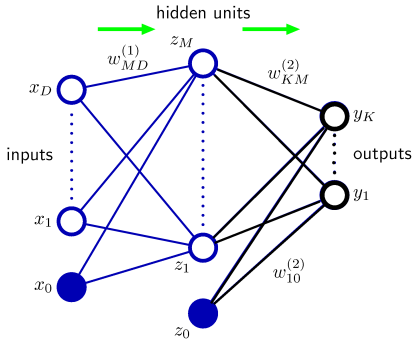
- This creates a representation that even a linear classifier can deal with.

What do good features look like?



- There are many incarnations: SIFT, HOG, GIST, SURF, ...
- Common to all is that they summarize local oriented structure, followed by several stages of non-linear processing.

Neural networks



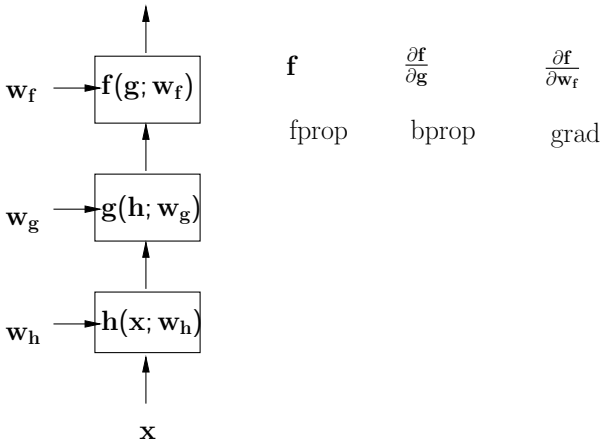
- Neural networks can in principle perform the same computation – if the weights are set appropriately.
- Finding the right weigts is slow in principle, but trivially parallelizable.

(picture adapted from Bishop 2006)

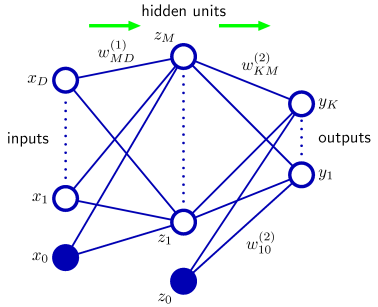
Back-prop

- It is easy to compute derivatives if the network is composed of modules that provide the following three functions:
  - A function **fprop()** to compute outputs, given inputs and parameters.
  - A function **bprop()** to compute derivatives of some function wrt. its inputs, given the derivatives of that function wrt. its outputs.
  - A function **grad()** to compute derivatives of some function wrt. its parameters, given inputs and the derivatives of that function wrt. its outputs.
- This is based of the chain-rule of differentiation, but it is better than using the chain rule “on paper”, because the computation of derivatives can now be automated.

Back-prop



A neural network with a single hidden layer

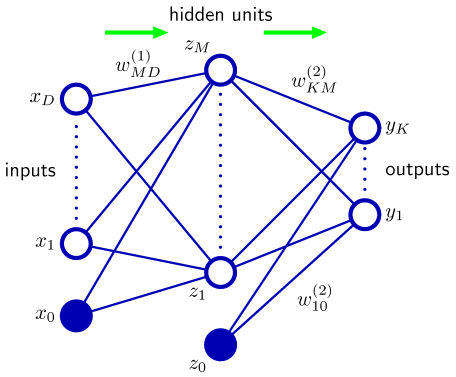


- A feed-forward neural net (AKA backprop net) computes its output layer-by-layer:

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}^{(2)} h\Big(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \Big)$$

this and most of the following images from: (Bishop, 2006)

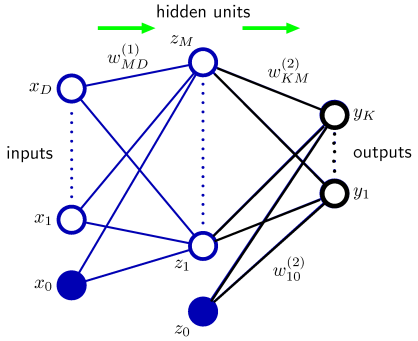
A neural network with a single hidden layer



- With explicit bias terms:

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}^{(2)} h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \Big) + w_{k0}^{(2)}$$

Output activation functions



- The last layer determines the functionality of the network. For example:
- linear outputs + squared error loss = non-linear regression
- softmax outputs + log-loss = non-linear logistic regression

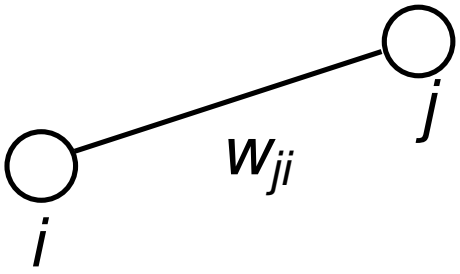
Backpropagation in detail

- Define the training cost as the sum over per-example costs:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- Now we can compute derivatives $\frac{\partial E_n}{\partial \mathbf{w}}$ individually for each training case and add them up afterwards.
- Definitions:
  - Let $a_j = \sum_i w_{ji} z_i$ be the *net input* of unit j.
  - Let z_i be the *output* of unit i, in other words $z_i = h(a_i)$.

Backpropagation in detail



- We need derivatives of the cost, E_n, with respect to each weight w_{ji} connecting nodes i and j in the network.

Backpropagation in detail

- By the chain-rule of differentiation, we have

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$$

- The second factor is easy:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

And to compute it for all i: Just run the network!

Backpropagation in detail

- Apply the chain-rule once more to get

$$\frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}$$

where the sum is over those units k connected to j.

- Intuitively, this reflects the fact that if we wiggle a_j this will affect the cost function through all the a_k.
- With

$$\frac{\partial a_k}{\partial a_j} = w_{kj} h'(a_j)$$

this simplifies to:

$$\frac{\partial E_n}{\partial a_j} = h'(a_j)\sum_k w_{kj}\frac{\partial E_n}{\partial a_k}$$

Backpropagation in detail

- Thus, we can use a recursion to compute all $\frac{\partial E_n}{\partial a_k}$ starting at the outputs.
- For squared error (regression), we have

$$\frac{\partial}{\partial a_k} E_n = \frac{\partial}{\partial a_k}\frac{1}{2}\|\mathbf{y}^{(n)}(\mathbf{x},\mathbf{w}) - \mathbf{t}^{(n)}\|^2 = y_k^{(n)} - t_k^{(n)}$$

since $y_k^{(n)} = a_k^{(n)}$ in the case of regression.

- Same for classification if we define the log-probability as softmax and use negative log-probability as the loss (exercise).

Backpropagation in detail

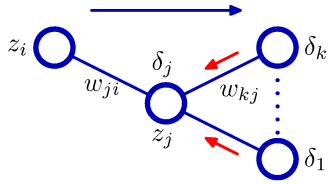
- If h is the logistic sigmoid, we have:

$$h'(a_j) = h(a_j)(1 - h(a_j))$$

So in this case we may use the activations themselves to compute the derivatives.

- But any activation function that is differentiable almost everywhere will work.

Backpropagation in detail



Backpropagation summary (Bishop, page 244):

1. Given input $\mathbf{x}_n$, propagate forward to compute activations for all hiddens z_i and outputs.
2. Evaluate E_n and $\frac{\partial E_n}{\partial a_k}$ for all output units.
3. Compute $\frac{\partial E_n}{\partial a_k}$ recursively for each hidden unit.
4. Compute the derivatives for each w_{ji} by multiplying the appropriate $\frac{\partial E_n}{\partial a_j}$ and z_i terms.

Learning arbitrary non-linear functions

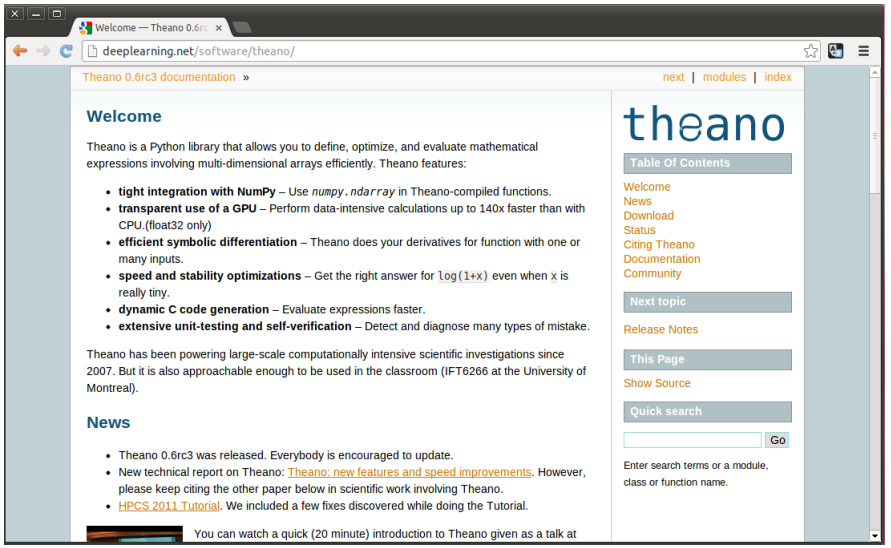
- A network with a single hidden layer can model any non-linear function under fairly mild conditions to arbitrary accuracy (eg. Funahashi, 1989).
- Unfortunately, the proof relies on using an exponentially large number of hidden units.
- So the practical relevance of this result is very limited.
- In practice, networks with many layers have proven to be much more useful.

Implementing backprop

- There are several software packages that implement backprop.
- theano (http://deeplearning.net/software/theano) takes the idea to the extreme, by using *symbolic differentiation*, so you don't even need to implement bprop and grad yourself.

```
import theano
import theano.tensor as T
x = T.dmatrix("x")
w = T.dmatrix("w")
somefunction = T.dot(w,x).sum()
python_function = theano.function([x,w], somefunction)
python_function(randn(100, 10), randn(10, 100))
derivative = T.grad(somefunction, w)
```


theano



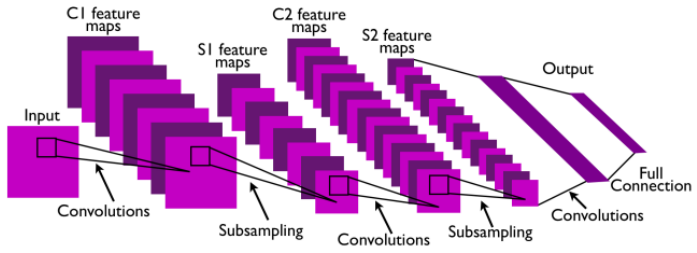
Weight sharing

- A common approach to reducing the number of model parameters is weight sharing:
- Force different parts of the network to use *the same* parameters.
- It is trivial to implement weight sharing using backprop:
- Just let your modules make use of the same parameter array.
- Derivatives for these parameters will simply accumulate.

Convolutional networks

- Convolutional networks ("conv nets") are probably the most common application of weight sharing.
- They are neural networks designed specifically for visual tasks (though there are examples for conv nets used in other domains).
- Since structure in images is local and invariant, they use local receptive fields with weight-sharing.
- This defines a convolution with (flipped) filters which are learned *discriminatively* using back-prop.

Convolutional networks

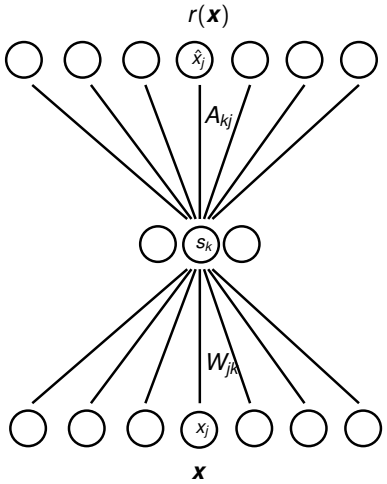


- Alternating sub-sampling layers are commonly used to get invariance to small shifts and to reduce the spatial extent of the representation towards the higher layers.
- (LeCun et al., 1989)

Convolutional networks

- Convolutional networks were inspired by Hubel & Wiesel's complex/simple cells results.
- There are various related models (but without back-prop learning): Neocognitron (Fukushima, 1980), HMAX (Riesenhuber & Poggio, 1999)
- A standard reference for conv nets is: "Gradient-based learning applied to document recognition." Y. LeCun, et al. 1998.
- (interestingly, that paper introduced another concept now heavily used in vision: conditional random fields)

Autoencoders

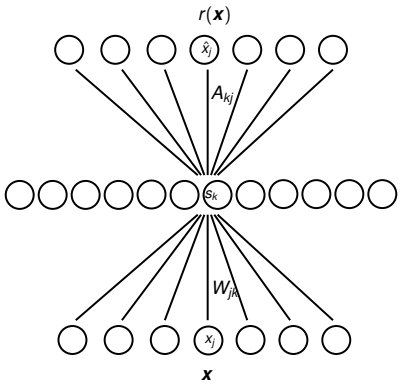


- Autoencoders are simple neural networks that are trained to reconstruct their input:

$$\text{cost} = \|r(\boldsymbol{x}) - \boldsymbol{x}\|^2$$

- The hidden layer is a bottleneck that forces the model to compress its input.
- Linear autoencoders implement a variation of PCA (Baldi, Hornik; 1989)

Overcomplete autoencoders



- With overcomplete hiddens, the model can "cheat" and learn the identity.
- One solution: Corrupt the inputs during training, but train the model to reconstruct the original, uncorrupted inputs (Vincent et al. 2008):

$$\text{cost} = \|r(\boldsymbol{x} + \text{noise}) - \boldsymbol{x}\|^2$$

Sparse autoencoders

- As discussed in the context of ICA, another way to define an overcomplete autoencoder, is by forcing hiddens to be sparse.
- This will also let the hidden layer act like a bottleneck.
- For example, to train a linear autoencoder with L_1 sparsity term:

$$\text{minimize} \quad \sum_t \|\boldsymbol{W}\boldsymbol{W}^{\mathrm{T}}\boldsymbol{z}_t - \boldsymbol{z}_t\|^2 + \lambda \sum_t \sum_i |\boldsymbol{w}_i^{\mathrm{T}}\boldsymbol{z}_t|$$

- K-mean clustering can be viewed as a (very) sparse autoencoder, too.
- Other sparse autoencoders include: contractive autoencoders, "shrinking" autoencoders, and others

Factorial representations

- In K-means clustering, each hidden unit represents a convex blob in the data-space.
- Thus, hidden units cannot collaborate to define regions in input space.
- A sparse autoencoder may be viewed as a way to allow for *some* collaboration between hiddens.
- The number of "blobs" that a set of hiddens can represent thus becomes, in principle, exponential in the number of hiddens involved.
- Codes that can collaborate are commonly called "factorial".

Relationship between encoder and decoder weights

- Take an autoencoder with "tied weights" ($\mathbf{W} = \mathbf{A}^{\mathrm{T}}$)
- If the model is defined on whitened data, the decoder weights (in terms of the original data) will be smoothed encoder weights (also in terms of the original data).
- To see this, write the encoder weights in terms of the original, unwhitened data as:

$$\boldsymbol{s}(\boldsymbol{x}) = \mathbf{W}\boldsymbol{z}(\boldsymbol{x}) = \mathbf{W}\mathbf{L}^{-\frac{1}{2}}\mathbf{U}^{\mathrm{T}}\boldsymbol{x} =: \bar{\mathbf{W}}\boldsymbol{x}$$

and the decoder weights in terms of the original, unwhitened data as:

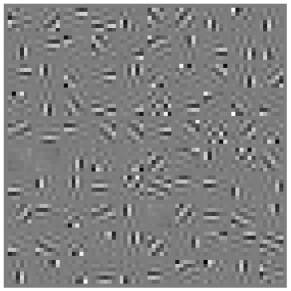
$$\boldsymbol{x}(\boldsymbol{s}) = \mathbf{U}\mathbf{L}^{\frac{1}{2}}\mathbf{W}^{\mathrm{T}}\boldsymbol{s} =: \bar{\mathbf{A}}\boldsymbol{s}$$

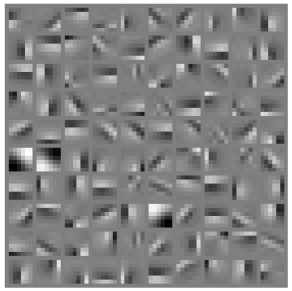
Relationship between encoder and decoder weights

- Now multiply the encoder weights by the data covariance matrix:

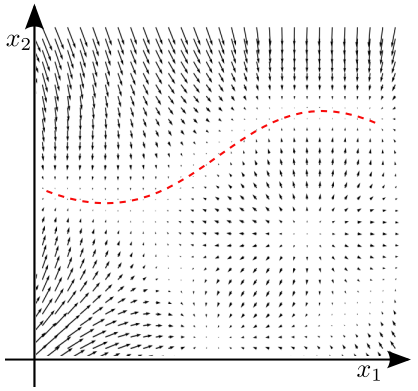
$$\begin{aligned}
\mathbf{C}\bar{\mathbf{W}}^{\mathrm{T}} &= \mathbf{C}\left(\mathbf{W}\mathbf{L}^{-\frac{1}{2}}\mathbf{U}^{\mathrm{T}}\right)^{\mathrm{T}} \\
&= \mathbf{C}\mathbf{U}\mathbf{L}^{-\frac{1}{2}}\mathbf{W}^{\mathrm{T}} \\
&= \mathbf{U}\mathbf{L}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{L}^{-\frac{1}{2}}\mathbf{W}^{\mathrm{T}} \\
&= \mathbf{U}\mathbf{L}^{\frac{1}{2}}\mathbf{W}^{\mathrm{T}} \\
&= \bar{\mathbf{A}}
\end{aligned}$$

Relationship between encoder and decoder weights





Autoencoders as dynamical systems



- An autoencoder maps points $\boldsymbol{x} \in \mathcal{R}^n$ to reconstructions $r(\boldsymbol{x}) \in \mathcal{R}^n$.
- This defines a *dynamical system*.
- We can plot $r(\boldsymbol{x}) - \boldsymbol{x}$ as a vector field.
- (Seung, 1998), (Alain, Bengio; 2013)

Autoencoders as dynamical systems

- The dynamical systems perspective provides another explanation for why denoising and contractive training works:
- Training can be viewed as making training data points *attractive fixed points* of the network dynamics.
- (Seung, 1998), (Swersky et al. 2011), (Vincent 2011), (Alain, Bengio; 2013), (Kamyshanska, Memisevic, 2013)

Computing the energy of an autoencoder

- Some dynamical systems can be defined as the derivative of a scalar function (AKA "scalar field" or "potential energy") $E(\boldsymbol{x})$.
- If such an energy exists, extrema of the scalar field will be fixed points of the dynamical system, and running the dynamical system will amount to performing gradient descent in the energy.
- A sufficient condition for the energy function to exist is that the weights are tied (Clairaut's theorem), in other words $\mathbf{W} = \mathbf{A}^{\mathrm{T}}$

Computing the energy of an autoencoder

- We can compute the energy of these autoencoders by integration (Kamyshanska, Memisevic; 2013):

$$\begin{aligned} E(\boldsymbol{x}) &= \int (r(\boldsymbol{x}) - \boldsymbol{x})\, d\boldsymbol{x} \\ &\vdots \\ &= \sum_k \int h(s_k)\, ds_k - \frac{1}{2}\|\boldsymbol{x} - b_r\|^2 + \text{const} \end{aligned}$$

where b_r is the vector of (visible) bias terms.

Computing the energy of an autoencoder

- Thus, computing the energy boils down to the following recipe:
  1. Replace hidden activation function by its anti-derivative (eg., softplus for sigmoid, half-square for rectifier, etc.).
  2. Sum up these new activations.
  3. subtract $\frac{1}{2}\|\boldsymbol{x} - b_r\|^2$
- For binary-output models, the last term turns into $b_r^{\mathrm{T}}\boldsymbol{x}$
- For sigmoid hiddens, this recipe comes down to computing *exactly* the RBM free energy!
- Potential energies are *additive* in the hiddens, so autoencoders are ICA-like.

Examples

Sigmoid activation $\sigma(x) = (1 + \exp(-x))^{-1}$**:**

$$E_{\text{sigmoid}}(\boldsymbol{x}) = \sum_k \text{softplus}(s_k) - \frac{1}{2}\|\boldsymbol{x} - b_r\|^2 + \text{const}$$

$\Rightarrow$ same as the free energy of a **binary-Gaussian RBM**.

Sigmoid activation (binary inputs):

$$E_{\text{sigmoid}}(\boldsymbol{x}) = \sum_k \text{softplus}(s_k) + b_r^T \boldsymbol{x} + \text{const}$$

$\Rightarrow$ same as the free energy of a **binary-binary RBM**.

Examples

Hyperbolic tangent activation $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$**:**

$$E_{\tanh}(\boldsymbol{x}) = \sum_k \log\left(\cosh(s_k)\right) - \frac{1}{2}\left(\boldsymbol{x} - b_r\right)^2 + \text{const}$$

Linear activation $h(s) = s$**:**

$$E_{\text{linear}}(\boldsymbol{x}) = \frac{1}{2}(W\boldsymbol{x} + b_h)^T(W\boldsymbol{x} + b_h) - \frac{1}{2}\left(\boldsymbol{x} - b_r\right)^2 + \text{const}$$

$\Rightarrow$ the norm of the latent representation, same as PCA generative classifier

Rectifier hiddens $h(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{else} \end{cases}$ **:**

$E_{\text{relu}}(\boldsymbol{x}) = \sum_k \left(\text{sign}(s_k) + 1\right)\frac{s_k^2}{2} - \frac{1}{2}\left(\boldsymbol{x} - b_r\right)^2 + \text{const}$