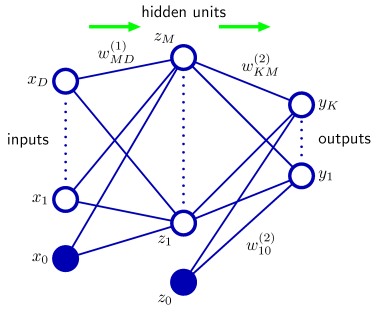
Machine learning for vision

Fall 2013

Roland Memisevic

Lecture 10, November 5, 2013

A neural network with a single hidden layer

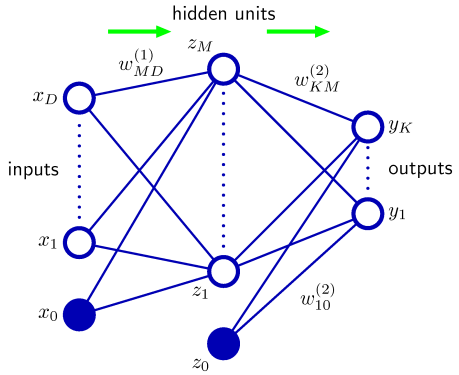


- A feed-forward neural net (AKA backprop net) computes its output layer-by-layer:

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

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

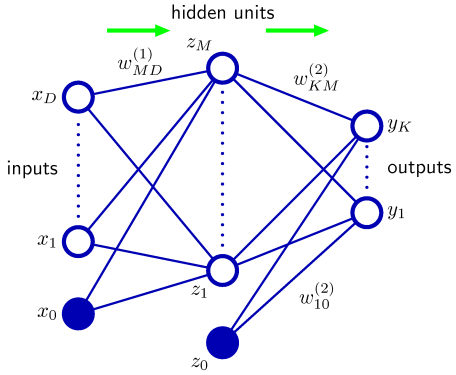
A neural network with a single hidden layer



- With explicit bias terms:

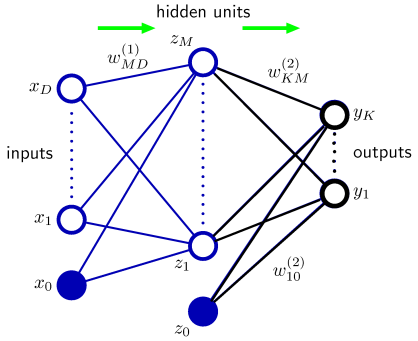
$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w^{(2)}_{kj} h\Big(\sum_{i=1}^{D} w^{(1)}_{ji} x_i + w^{(1)}_{j0} \Big) + w^{(2)}_{k0}$$

A neural network with a single hidden layer



- In practice, we can add more hidden layers. $\rightarrow$ compute derivatives using back-prop.

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.

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.

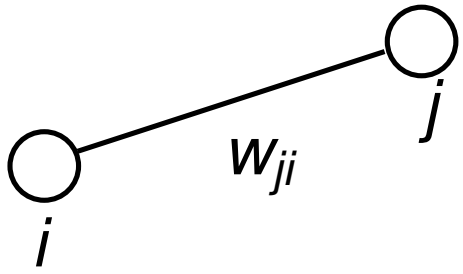
Backpropagation

- 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



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

Backpropagation

- 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

- 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

- 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

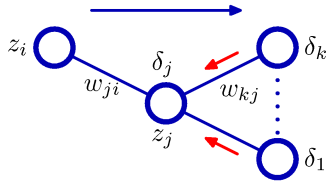
- If h is the logistic sigmoid, we have:

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

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



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.

Implementing backprop

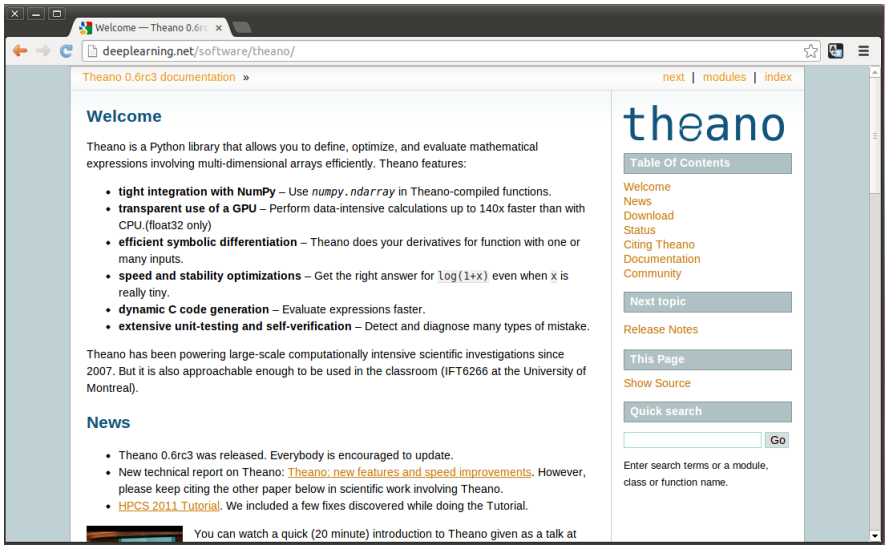
- More generally, backprop tells us that it is easy to compose systems consisting of modules, as long as the modules provide the following three functions:
  - A function **fprop()** to compute outputs, given inputs and parameters.
  - A function **bprop()** to compute derivatives of some function wrt. 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.
- Building large complicated systems is then just a matter of sticking together these modules, and gradients can be computed fully automatically.

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



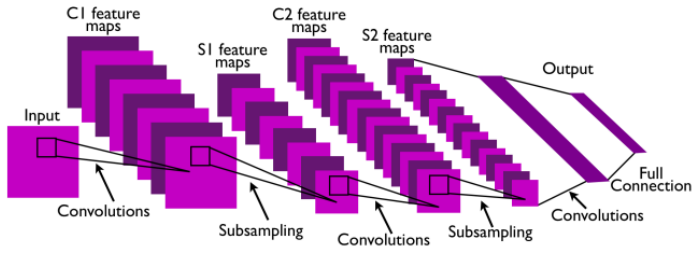
Weight sharing

- 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

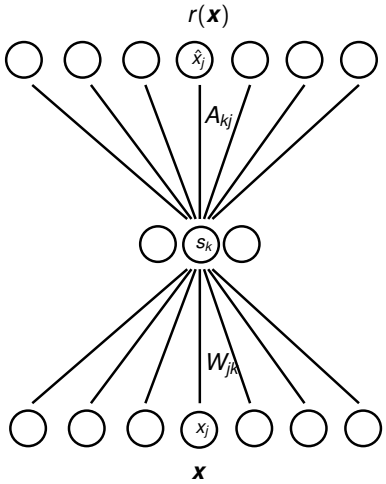


- Alternating sub-sampling layers are commonly used to get invariance to small shifts and to reduce the spatial extent of the representation towards the higher layers.
- (LeCun et al., 1989)

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.
- (strangely, that paper introduced another concept now heavily used in vision: conditional random fields (CRF))

Autoencoders

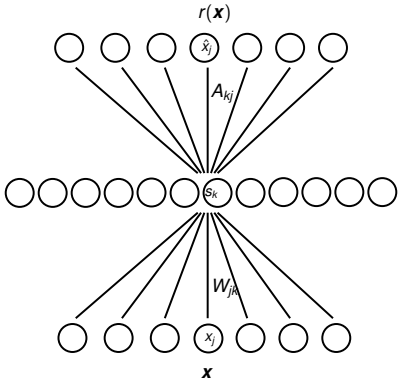


- Autoencoders are simple neural networks that are trained to reconstruct their input:

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

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

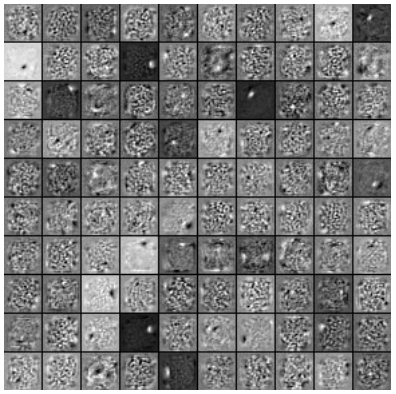
Overcomplete autoencoders



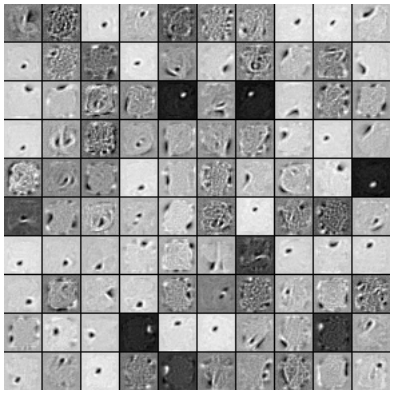
- With overcomplete hiddens, the model can cheat and learn the identity.
- One solution: Corrupt the inputs during training, but train the model to reconstruct the original, uncorrupted inputs (Vincent et al. 2008):

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

Denoising autoencoders

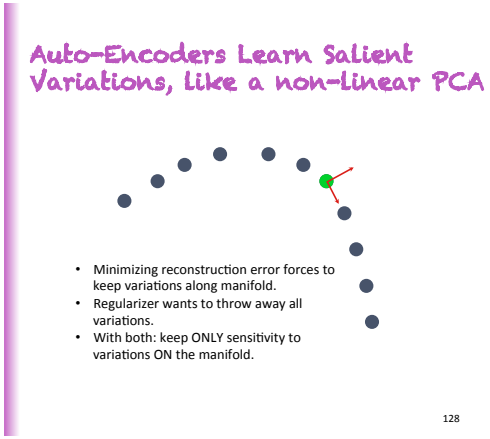


no denoising



denoising

Denoising autoencoders



(slides by Yoshua Bengio)

Denoising autoencoders

Contractive Auto-Encoders

(Rifai, Vincent, Muller, Glorot, Bengio ICML 2011; Rifai, Mesnil, Vincent, Bengio, Dauphin, Glorot ECML 2011; Rifai, Dauphin, Vincent, Bengio, Muller NIPS 2011)

$$\text{reconstruction}(x) = g(h(x)) = \text{decoder}(\text{encoder}(x))$$

Training criterion:

$$\mathcal{J}_{CAE}(\theta) = \sum_{x \in D_n} \lambda \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2 + L(x, \text{reconstruction}(x))$$

wants contraction in all directions

cannot afford contraction in manifold directions

If $h_j = \text{sigmoid}(b_j + W_j x)$

$(dh_j(x)/dx_i)^2 = h_j^2(1-h_j)^2 W_{ji}^2$

Denoising autoencoders

Denoising auto-encoders are also contractive!

- Taylor-expand Gaussian corruption noise in reconstruction error:

$$\begin{aligned} E\left[\ell(x, r(x+\epsilon))\right] &\approx E\left[\left(x - \left(r(x) + \frac{\partial r(x)}{\partial x}\epsilon\right)\right)^T \left(x - \left(r(x) + \frac{\partial r(x)}{\partial x}\epsilon\right)\right)\right] \\ &= E\left[\|x - r(x)\|^2\right] + \sigma^2 E\left[\left\|\frac{\partial r(x)}{\partial x}\right\|_F^2\right] \end{aligned}$$

- Yields a contractive penalty in the reconstruction function (instead of encoder) proportional to amount of corruption noise

Contractive AE in theano

```
.
.
.
hiddens = T.nnet.sigmoid(T.dot(inputs, W) + bhid)
outputs = T.dot(hiddens, W.T) + bvis
cost = T.mean(T.sum(0.5 * ((inputs - outputs)**2), axis=1))
cost += contraction *
        T.sum( ((hiddens * (1 - hiddens))**2) * T.sum(W**2,
grads = T.grad(cost, params)
.
.
.
```


Sparse autoencoders

- 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:

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

- This can also be viewed as a way to implement ICA (see lecture 6).
- K-mean clustering can be viewed as a (very) sparse autoencoder, too.

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 weight sin terms of the original, unwhitened data as:

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

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

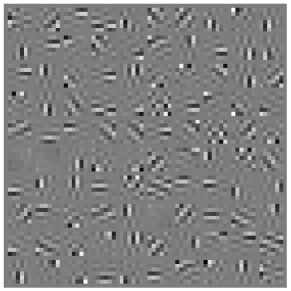
$$\boldsymbol{x}(\boldsymbol{s}) = \mathbf{U}\mathbf{L}^{\frac{1}{2}}\mathbf{W}\boldsymbol{s} =: \bar{\mathbf{A}}\boldsymbol{s}$$

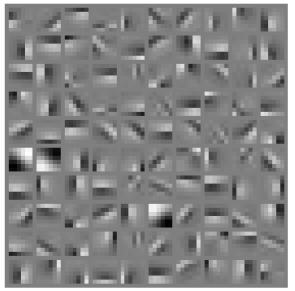
Relationship between encoder and decoder weights

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

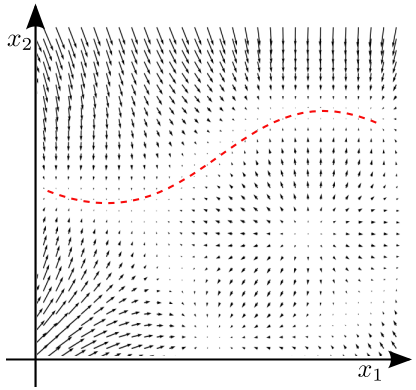
$$\begin{aligned}\mathbf{C}\bar{\mathbf{W}}^{\mathrm{T}} &= \mathbf{C}\mathbf{U}\mathbf{L}^{-\frac{1}{2}}\mathbf{W} \\ &= \mathbf{U}\mathbf{L}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{L}^{-\frac{1}{2}}\mathbf{W} \\ &= \mathbf{U}\mathbf{L}^{\frac{1}{2}}\mathbf{W} \\ &= \bar{\mathbf{A}}\end{aligned}$$

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.
- Which autoencoders have energy functions?
- Answer: Those with tied weights ($\mathbf{W} = \mathbf{A}^{\mathrm{T}}$) (Poincare's Lemma)

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

- 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 it is in general ICA-like general.

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

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

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**.

RBMs are autoencoders

- Minimizing reconstruction error will minimize the magnitude of the *first derivative* of the energy at the data.
- Denoising/contraction penalties will force *second derivatives* of the energy (= first derivatives of $r(\boldsymbol{x}) - \boldsymbol{x}$) to be negative at the data (because they force $\frac{\partial r(\boldsymbol{x})}{\partial \boldsymbol{x}}$ to be small).
- This will encourage datapoints to be local minima of the energy.
- This is *exactly* what RBM training tries to accomplish. The only technical difference is that the RBM optimizes the energy directly (in its positive phase) and its derivative through sampling, whereas the AE optimizes only the derivatives, and it does so analytically.