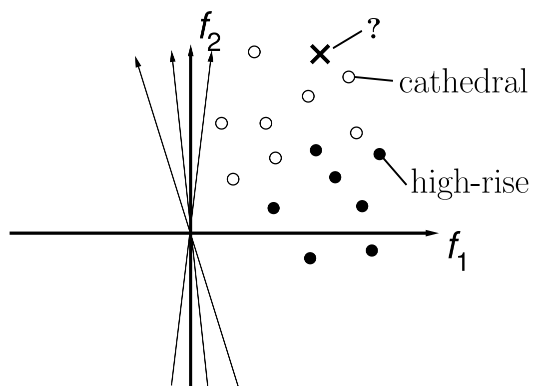
Machine learning for vision

Winter 2016

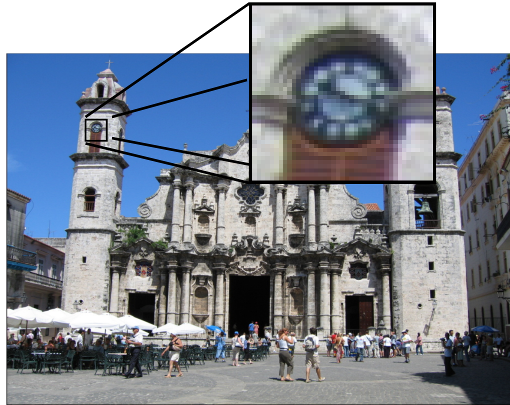
Roland Memisevic

Lecture 2, January 26, 2016



“It’s the features, stupid!”

“It’s the features, stupid!”

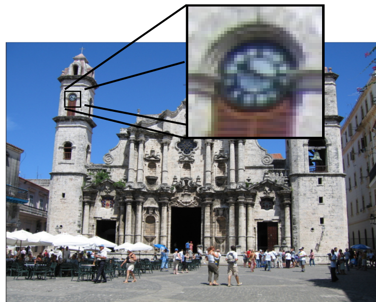


“It’s the features, stupid!”

A common computer vision pipeline before 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 of the image.

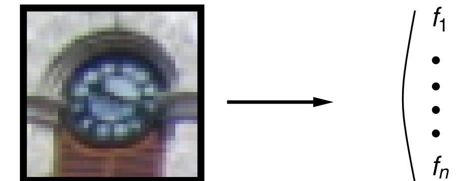
“It’s the features, stupid!”



A common computer vision pipeline before 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 of the image.

“It’s the features, stupid!”



A common computer vision pipeline before 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 of the image.

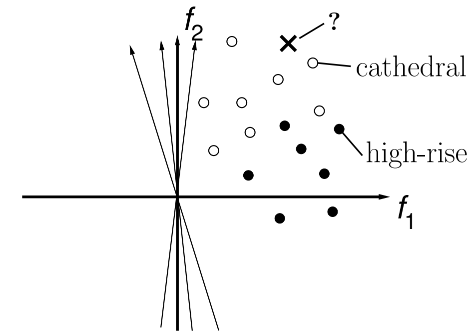
“It’s the features, stupid!”

$$\begin{pmatrix} f_1^1 \\ \vdots \\ f_n^1 \end{pmatrix} + \cdots\cdots + \begin{pmatrix} f_1^M \\ \vdots \\ f_n^M \end{pmatrix}$$

A common computer vision pipeline before 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 of the image.

“It’s the features, stupid!”



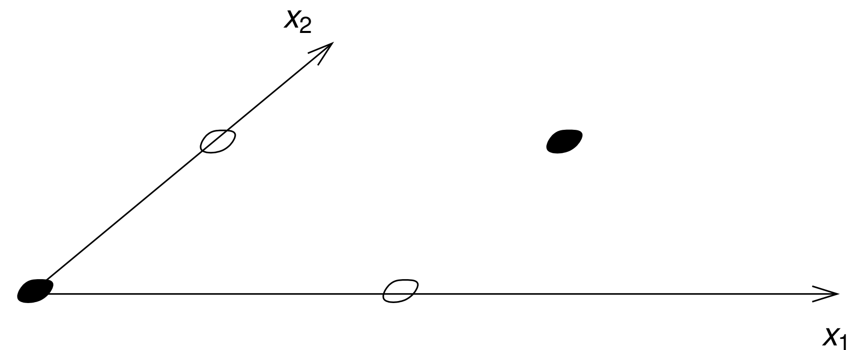
- This creates a representation that even a linear classifier can deal with.

bottom line: **computer vision is all about building non-linear pipelines**
(aka *“the representation matters”*)

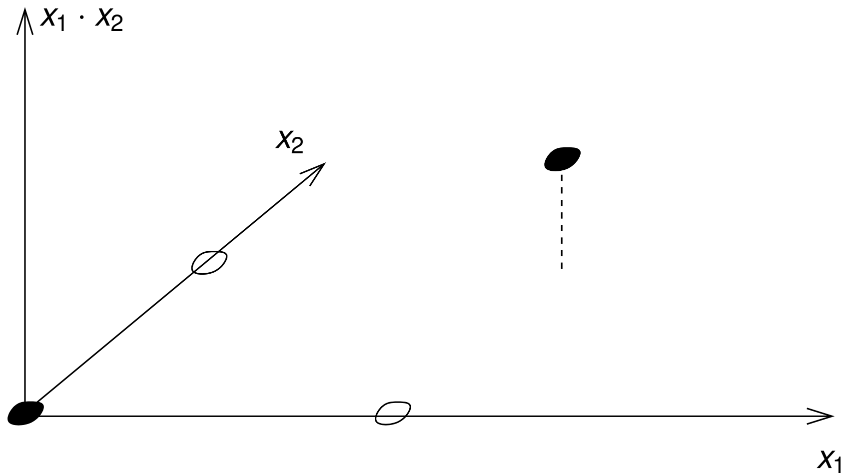
What do good low-level features look like?

- Local features that are often found to work well are based on oriented structure (such as Gabor features)
- These were discovered again and again (also in other areas) and are closely related to the Short Time Fourier Transform.

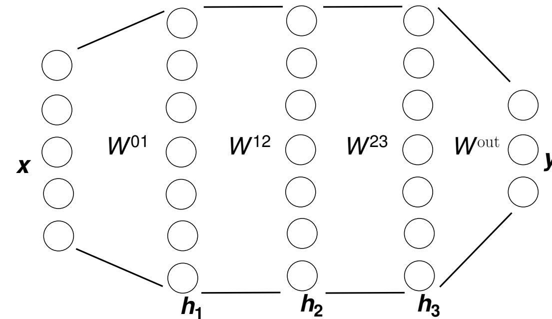
The XOR problem



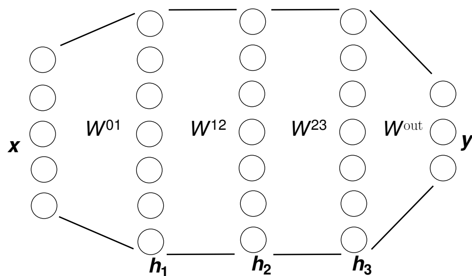
The XOR problem



Neural networks are *trainable* pipelines



Neural networks are *trainable* pipelines

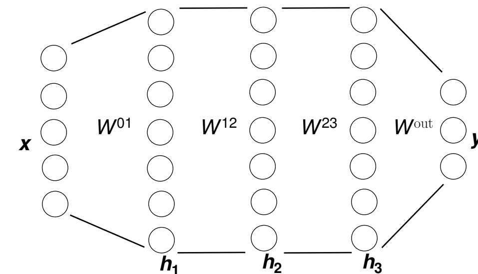


Most common networks interleave **matrix multiplies** with **element-wise non-linearities**:

$$\boldsymbol{y}(\mathrm{x}) = W^{\text{out}} h(W^{23} h(W^{12} h(W^{01} \boldsymbol{x})))$$

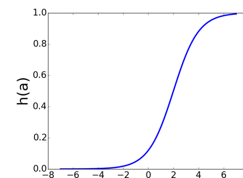
Usually there are constant “bias”-terms as well.

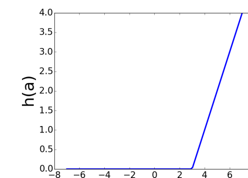
Neural networks are *trainable* pipelines

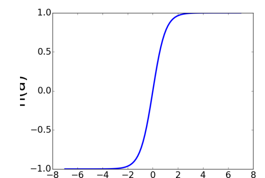


Common non-linearities:

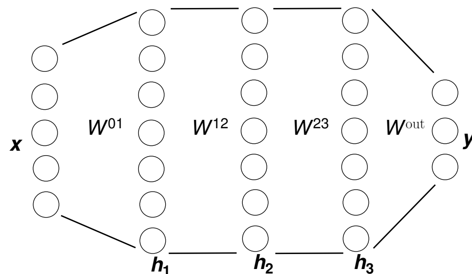
sigmoid: $h(a) = \frac{1}{1+\exp(a)}$ ReLU: $h(a) = a \cdot [a > 0]$ tanh: $h(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$







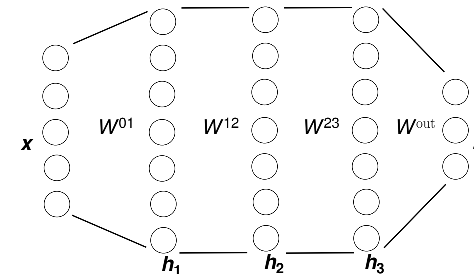
Neural networks are *trainable* pipelines



For classification tasks, turn class outputs into probabilities using the "softargmax" function:

$$p(\mathcal{C}_k|\boldsymbol{x}) = \frac{\exp(y_k(\boldsymbol{x}))}{\sum_j \exp(y_j(\boldsymbol{x}))}$$

Neural networks are *trainable* pipelines



For training, use a (large) training set $(\boldsymbol{x}_n, \boldsymbol{t}_n)_{n=1...N}$ and minimize a suitable *cost*-function, using stochastic gradient descent (SGD).

The most common choices of cost function

- **Regression** (predict real values):

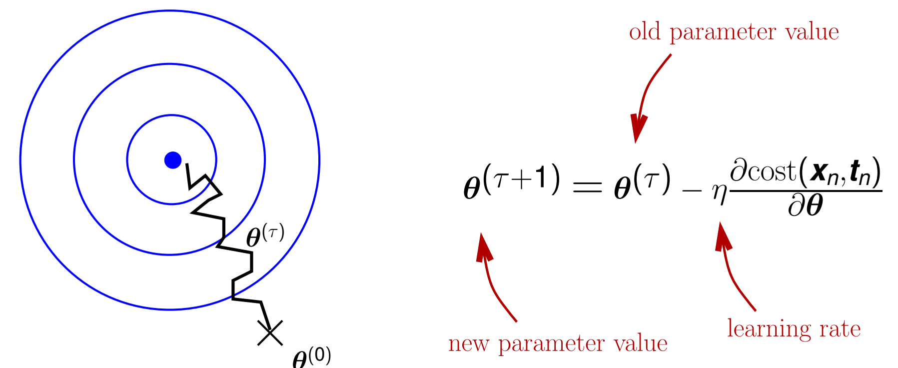
$$\text{cost} = \frac{1}{2}\sum_{n=1}^{N} \|\boldsymbol{y}(\boldsymbol{x}_n) - \boldsymbol{t}_n\|^2$$

- **Classification** (predict discrete labels):

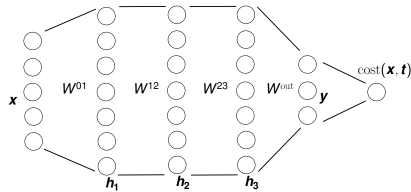
$$\text{cost} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \log p(\mathcal{C}_k|\boldsymbol{x}_n)$$

where $t_{nk} = 1$ iff training case n belongs to class k.

Stochastic gradient descent (SGD)



Error back-propagation (backprop)



- **Use the chainrule:** For regression and classification we get:

$$\frac{\partial \text{cost}}{\partial \boldsymbol{y}(\boldsymbol{x}_n)} = \boldsymbol{y}(\boldsymbol{x}_n) - \boldsymbol{t}_n$$

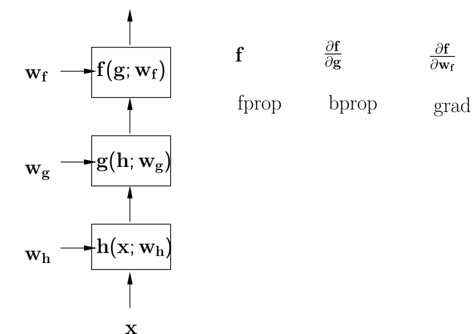
- Next: If $\boldsymbol{y}$ has any parameters, W^{out}, collect them using:

$$\frac{\partial \text{cost}}{\partial W^{\text{out}}} = (\boldsymbol{y}(\boldsymbol{x}_n) - \boldsymbol{t}_n) \cdot \frac{\partial \boldsymbol{y}(\boldsymbol{x}_n)}{\partial W^{\text{out}}}$$

- Next: Descend to the next layer by computing

$$\frac{\partial \text{cost}}{\partial \boldsymbol{h}_3} = \frac{\partial \text{cost}}{\partial \boldsymbol{y}(\boldsymbol{x}_n)} \cdot \frac{\partial \boldsymbol{y}(\boldsymbol{x}_n)}{\partial \boldsymbol{h}_3(\boldsymbol{x}_n)} \qquad \text{...and so on...}$$

Backprop general form



$\mathbf{f}$ $\frac{\partial \mathbf{f}}{\partial \mathbf{g}}$ $\frac{\partial \mathbf{f}}{\partial \mathbf{w_f}}$

fprop   bprop   grad

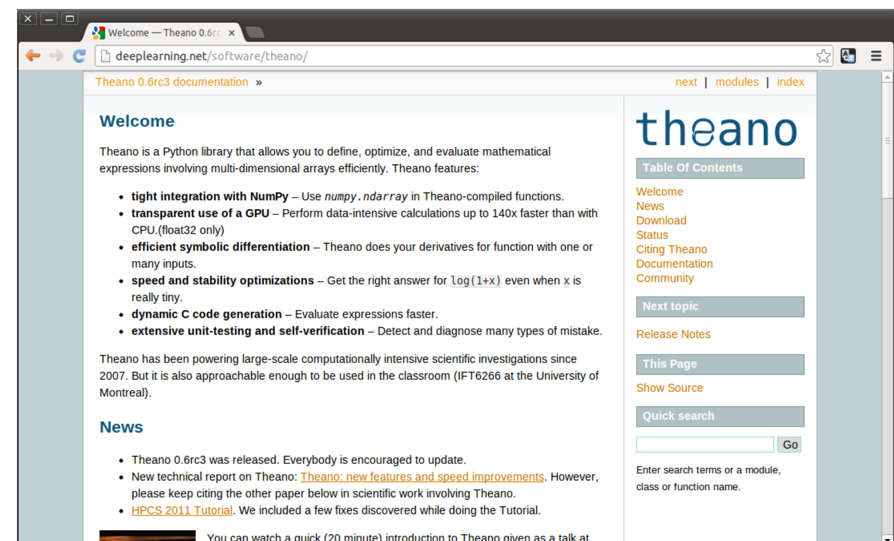
- Backprop can be thought of as an engineering principle, that prescribes how to design an end-to-end train-able system from differentiable components:
- Use components which provide the methods **fprop**, **bprop** and **grad**. Then backprop can be automated.
- Well-suited for support by software frameworks

Implementing backprop

- There are several software packages that implement backprop.
- Software packages like *theano* and *tensorflow*, take 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)
```

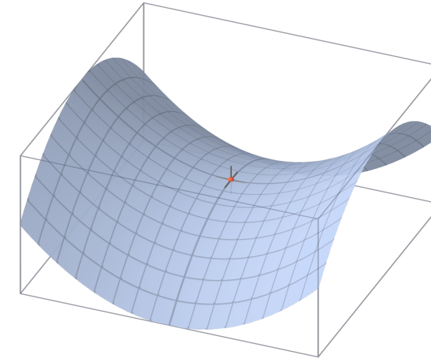

theano



Potential Issues

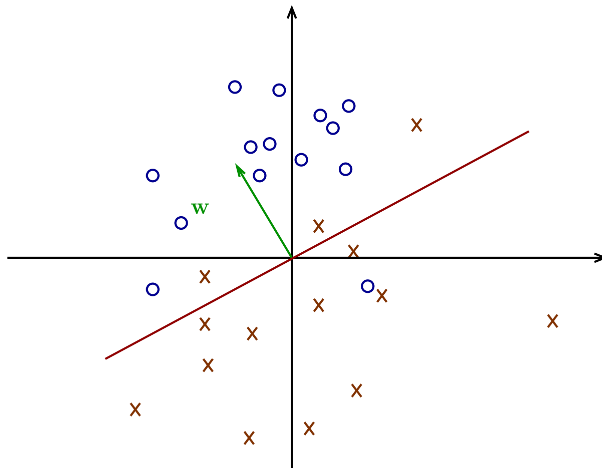
- "But what about local minima?"
- "But what about overfitting?"
- Vanishing gradients

The cost surface/local optima

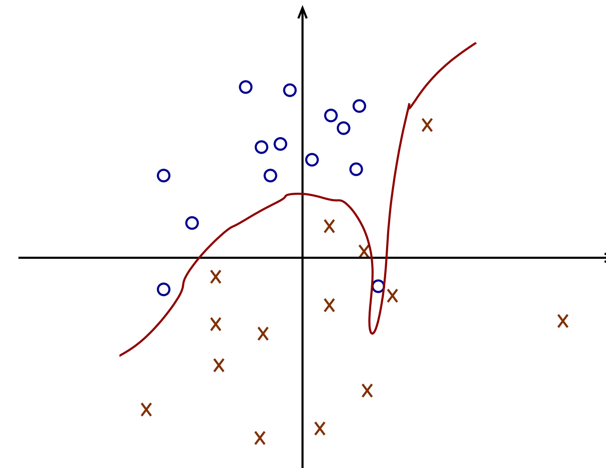


- Local minima not an issue in practice
- This is probably due to high dimensional parameter space, which causes most critical points to be **saddle points** not local optima.
- Some recent theoretical work supports this view (Choromanska et al. 2014); (Dauphin, et al. 2014)

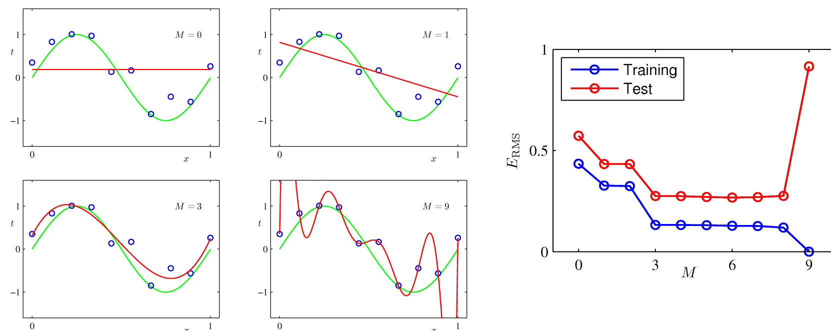
Overfitting



Overfitting



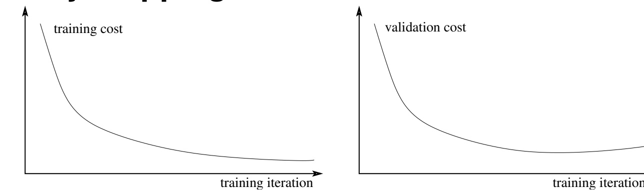
Overfitting in regression



(Bishop 2006: Pattern recognition and machine learning)

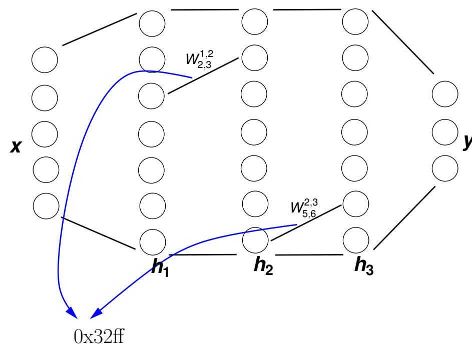
Preventing overfitting in neural networks

- **Early stopping**:



- **Weight decay** (somewhat outdated): add a weight penalty to the training objective (weight constraints now more common)
- **Dropout (Hinton et al., 2012)**: Corrupt hidden unit activations (multiplicatively) during training
- **More data**
- **Weight sharing (reduce the number of parameters):**

Weight sharing

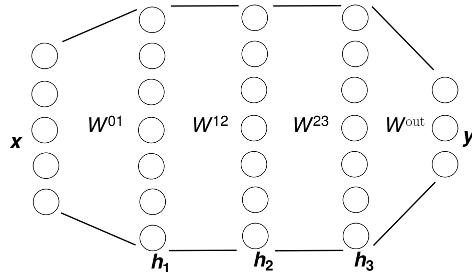


- Parameters can be shared by having them point to the same memory location.
- Very common way to reduce parameters and encode prior knowledge.
- Central ingredient in conv-nets (CNNs) and recurrent nets (RNNs). *But: It requires long-range communication.*

Batch normalization (Ioffe, Szegedy 2015)

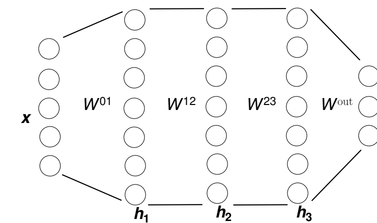
- Normalize the “pre-activation” (activation before non-linearity) of each neuron *averaged over the current mini-batch* to have mean 0 and standard-deviation 1.
- To allow the network to learn the original, unnormalized function, two parameters are added that allow it to undo the normalization.
- Batch normalization is a somewhat unusual operation, because it couples (independently for each neuron) all examples within the minibatch (and requires to back-prop through them).
- Shown to stabilize training and prevent overfitting.
- Explanation attempt by (Ioffe, Szegedy 2015): it prevents “covariate-shift”.

The vanishing gradients problem



- The backward-pass is a sequence of matrix multiplies.
- Depending on the magnitude of the eigenvalues, initial values can blow up or decay to zero.
- This can may learning difficult or slow.
- Potential solutions: architectural tricks (for example, the “LSTM” unit)

Neural nets learn distributed representations



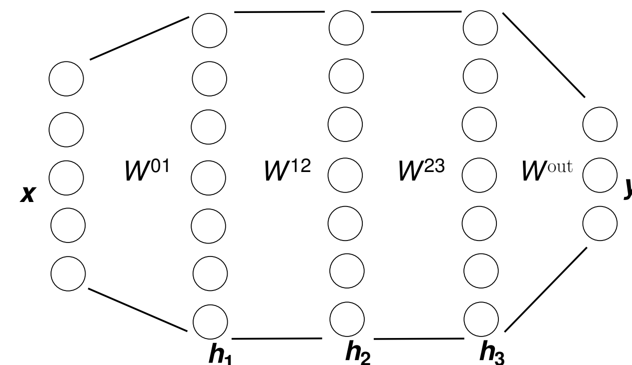
- Neural networks encode information as vectors of real values.
- This makes it easy to encode conceptual similarities. In a text processing task, for example:
  - If user searches for **Dell notebook battery size**, we would like to match documents with “Dell laptop battery capacity”
  - If user searches for **Seattle motel**, we would like to match documents containing “Seattle hotel”

(Example from Chris Manning)

Universal approximation

- 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.

Back-prop using asynchronous, local computations?



In the brain, where is the backward channel?

Towards back-prop using local computations

- Hinton 2007: Use the **temporal derivative** to encode the error derivative!
- (see also: Bengio et al. 2015)
- Recall that the derivative of most common cost functions is, conveniently, given by

$$\frac{\partial \text{cost}}{\partial \boldsymbol{y}(\boldsymbol{x}_n)} = \boldsymbol{y}(\boldsymbol{x}_n) - \boldsymbol{t}_n$$

How local back-prop may work

- Let top-layer drive the activations towards the correct value.
- Let feedback weights transport that change downward.
- Make weight changes proportional to the *rate of change* of a postsynaptic neuron and the *value* of the pre-synaptic neuron.

Is the brain doing local back-prop?

- (Hinton 2007): “What would neuro-scientists see if this is what’s happening in the brain?”
- They should see this (and they do!):