A neural network with a single hidden layer

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

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

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

▶ In practice, we can add more hidden layers. → 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 (e.g. 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:
  \[ E(w) = \sum_{n=1}^{N} E_n(w) \]
- Now we can compute derivatives \( \frac{\partial E_n}{\partial w} \) 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 \( z_i = h(a_i) \).
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!

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

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).

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 summary (Bishop, page 244):
1. Given input \( x_n \), propagate forward to compute activations for all hidden \( z_i \) 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
- 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.

```python
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)
```
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

- 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)
- (strangely, that paper introduced another concept now heavily used in vision: conditional random fields (CRF))
Autoencoders are simple neural networks that are trained to reconstruct their input:
\[
\text{cost} = \| r(x) - x \|^2
\]
The hidden layer is a bottleneck that forces the model to compress its input.
Linear autoencoders implement a variation of PCA (Baldi, Hornik; 1989)

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):
\[
\text{cost} = \| r(x + \text{noise}) - x \|^2
\]

Denoising autoencoders
- Minimizing reconstruction error forces to keep variations along manifold.
- Regularizer wants to throw away all variations.
- With both, keep ONLY sensitivity to variations ON the manifold.

(slides by Yoshua Bengio)
Denoising autoencoders

**Contractive Auto-Encoders**

- **Rifai, Vincent, Muller, Glorot ICML 2011; Rifai, Mesnil, Vincent, Bengio, Dauphin, Glorot ECML 2011; Rifai, Dauphin, Vincent, Bengio, Muller NIPS 2011**

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

Training criterion:

\[ J_{\text{CAE}}(\theta) = \sum_{x \in D_{\text{train}}} \left( \lambda \sum_{j} \left( \frac{\partial h_j(x)}{\partial x_i} \right)^2 \right)^2 + L(x, \text{reconstruction}(x)) \]

- wants contraction in all directions
- cannot afford contraction in manifold directions

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

\[ (dh_j(x)/dxi)^2 = h_j(1-h_j)W_{ji}^2 \]

**Contractive AE in theano**

```python
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:

\[
\text{minimize} \sum_{t} \| W W^T z_i - z_i \|^2 + \lambda \sum_{t} \sum_{i} |w_i^T z_i|
\]

- 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" ($W = A^T$)
- 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 in terms of the original, unwhitened data as:
  $$ s(x) = W^T z(x) = W^T L^{-1/2} U^T x =: \tilde{W} x $$
  and the decoder weights in terms of the original, unwhitened data as:
  $$ x(s) = U L^{1/2} W s =: \tilde{A} s $$

Now multiply the encoder weights by the data covariance matrix:

$$ C \tilde{W}^T = C U L^{-1/2} W \\
= U L^T U L^{-1/2} W \\
= U L^{1/2} W \\
= \tilde{A} $$
Autoencoders as dynamical systems

- An autoencoder maps points \( x \in \mathbb{R}^n \) to reconstructions \( r(x) \in \mathbb{R}^n \).
- This defines a dynamical system!
- We can plot \( r(x) - x \) as a vector field.
- (Seung, 1998), (Alain, Bengio; 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”) \( E(x) \).
- 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 \( W = A^T \) (Poincare’s Lemma)

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)

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

\[
E(x) = \int (r(x) - x) \, dx
\]

\[
= \sum_k \int h(s_k) \, ds_k - \frac{1}{2} \| x - b_r \|^2 + \text{const}
\]

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 (e.g., softplus for sigmoid, half-square for rectifier, etc.).
  2. Sum up these new activations.
  3. Subtract $\frac{1}{2}\|x - br\|^2$
- For binary-output models, the last term turns into $b^T r x$
- For sigmoid hidden, 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 $\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$:

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

Linear activation $h(s) = s$:

$$E_{\text{linear}}(x) = \frac{1}{2} (Wx + bh)^T (Wx + bh) - \frac{1}{2} (x - b_r)^2 + \text{const}$$

⇒ the norm of the latent representation, same as PCA generative classifier

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

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

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 $r(x) - x$) to be negative at the data (because they force $\frac{\partial r(x)}{\partial x}$ 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.