Numerical Optimization and Neural Networks

Emo Todorov

Applied Mathematics
Computer Science and Engineering

University of Washington

Error and gradient for a single unit

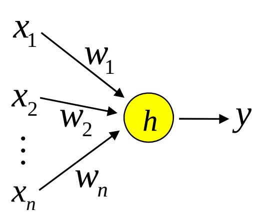
Input: $\mathbf{x}^{(k)}$

Desired output: $y^{(k)}$

Activation: $z^{(k)} = \sum_{i=1}^{n} x_i^{(k)} w_i$

Output: $y = h\left(z^{(k)}\right)$

Architecture:



Error:

$$E(\mathbf{w}) = \frac{1}{2}\sum_k \left(y^{(k)} - h\left(z^{(k)}\right)\right)^2 = \frac{1}{2}\sum_k \left(y^{(k)} - h\left(\sum_i x_i^{(k)} w_i\right)\right)^2$$

The slope of the error function along axis p (partial derivative with respect to w_p):

$$\frac{\partial E(\mathbf{w})}{\partial w_p} = -\sum_k \left(y^{(k)} - h\left(\sum_i x_i^{(k)} w_i\right)\right) h'\left(\sum_i x_i^{(k)} w_i\right) x_p^{(k)} = -\sum_k \underbrace{\left(y^{(k)} - h\left(z^{(k)}\right)\right) h'\left(z^{(k)}\right)}_{\text{does not depend on p}} x_p^{(k)}$$

Gradient of error function:

$$-\nabla E(\mathbf{w}) = -\begin{bmatrix} \partial E(\mathbf{w}) / \partial w_1 \\ \vdots \\ \partial E(\mathbf{w}) / \partial w_n \end{bmatrix} = \sum_k \left(y^{(k)} - h\left(z^{(k)}\right)\right) h'\left(z^{(k)}\right) \mathbf{x}^{(k)}$$

desired – actual output

scalar

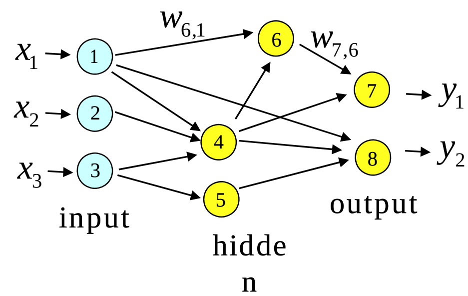
Input

Delta rule!

Arbitrary feed-forward networks

Preliminaries:

- Fix one of the inputs to 1, connect it to each unit that has a bias... no need to worry about biases anymore!
- For each input line, introduce a fake unit that simply copies the corresponding input
- Enumerate all units, starting with the inputs and ending with the outputs, so that all arrows point from the smaller to the larger number (this guarantees that the network has no loops)



$In(6) = \{1,4\}$ $\quad Out(6) = \{7\}$

Notation:

Weight from unit i to unit j: w_{ji}

Set of units that provide input to j: $In(j)$

Set of units that j sends output to: $Out(j)$

Transfer function for non-fake units: $h_j(.)$

Output of unit j (note: o = x for input units):

$$o_j = h_j(z_j)$$

Internal activation of unit j:

$$z_j = \sum_{i \in In(j)} o_i w_{ji}$$

Gradients

Error function and its gradient:

$$\nabla E(\mathbf{w}; D) = \nabla\left(\frac{1}{2}\sum_k \left\|\mathbf{y}^{(k)} - \mathbf{y}\left(\mathbf{x}^{(k)}\right)\right\|^2\right) = \sum_k \nabla\left(\frac{1}{2}\left\|\mathbf{y}^{(k)} - \mathbf{y}\left(\mathbf{x}^{(k)}\right)\right\|^2\right)$$

compute the gradient for each data point, then add up the results

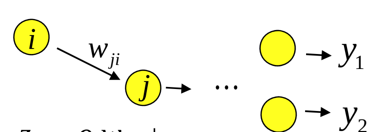
Suppress data index k for clarity:

$$E = \frac{1}{2}\left\|\mathbf{y} - \mathbf{y}(\mathbf{x}; \mathbf{w})\right\|^2 = \frac{1}{2}\sum_i \left(y_i - y_i(\mathbf{x}; \mathbf{w})\right)^2$$

How does changing one weight affect the error?

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} o_i$$

$$\boxed{\delta_j = \frac{\partial E}{\partial z_j}}$$



$$z_j = o_i w_{ji} + \ldots$$

$$\partial z_j / \partial w_{ji} = o_i$$

If we somehow compute all δs, we are done!
The gradient is simply the list of all $\delta_j o_i$

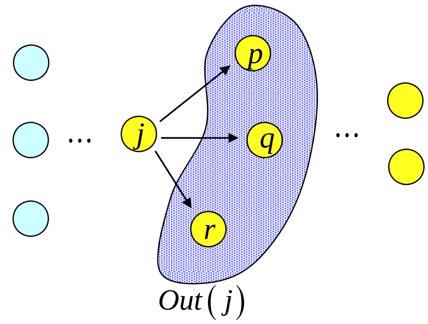
$$\nabla E(\mathbf{w}) = \begin{bmatrix} \vdots \\ \partial E / \partial w_{ji} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \delta_j o_i \\ \vdots \end{bmatrix}$$

Back-propagation

$$\delta_j = \frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial o_j}\frac{do_j}{dz_j} = -\left(y_s - o_j\right) h_j^{'}\left(z_j\right)$$ since $o_j = h_j\left(z_j\right)$ and $E = \frac{1}{2}\left(y_s - o_j\right)^2 + \ldots$

What is δ_j for a non-output unit?

The Error depends on z_j only through the activations of the units in the set $Out(j)$



Using the multivariate chain rule, we get:

$$\delta_j = \frac{\partial E\left(z_p, z_q, z_r, \cdots\right)}{\partial z_j} = \frac{\partial E}{\partial z_p}\frac{\partial z_p}{\partial z_j} + \frac{\partial E}{\partial z_q}\frac{\partial z_q}{\partial z_j} + \frac{\partial E}{\partial z_r}\frac{\partial z_r}{\partial z_j}$$

In general, we have to sum over all units $i \in Out(j)$

$$\delta_j = \sum_{i \in Out(j)} \frac{\partial E}{\partial z_i}\frac{\partial z_i}{\partial z_j} = \sum \delta_i \frac{\partial z_i}{\partial o_j}\frac{do_j}{dz_j}$$

Recall that: $z_i = w_{ij} o_j + \cdots \quad \Rightarrow \quad \partial z_i / \partial o_j = w_{ij}$

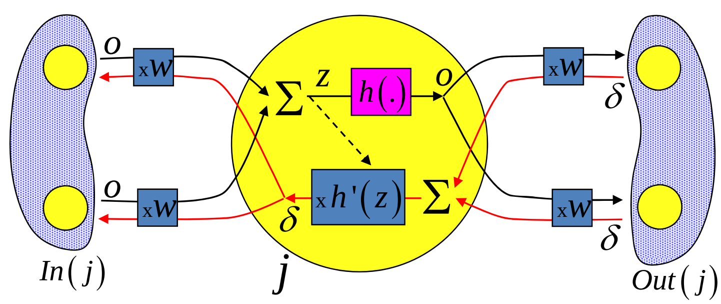
$o_j = h_j\left(z_j\right) \quad \Rightarrow \quad \partial o_j / \partial z_j = h_j^{'}\left(z_j\right)$

Error Back-propagation

$$\boxed{\delta_j = h_j^{'}\left(z_j\right) \sum_{i \in Out(j)} \delta_i w_{ij}}$$

Properties of back-propagation

Forward pass: compute z, o, h'



Backward pass: compute δ

Both the forward and backward pass use the same connections

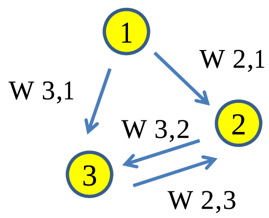
The algorithm is computationally efficient, i.e. it avoids re-computing quantities that are already computed (a bit like dynamic programming)

Each weight is adapted on the basis of local information only: $\frac{\partial E}{\partial w_{ji}} = \delta_j o_i$

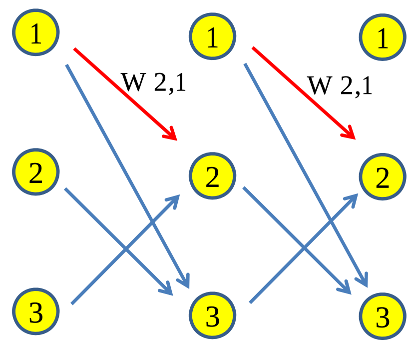
Biologically realistic: synapses in the brain adapt as a function of pre- and post-synaptic activity

Biologically unrealistic: neurons in the brain have no mechanism for propagating δ-like quantities backwards

Unfolding recurrent networks



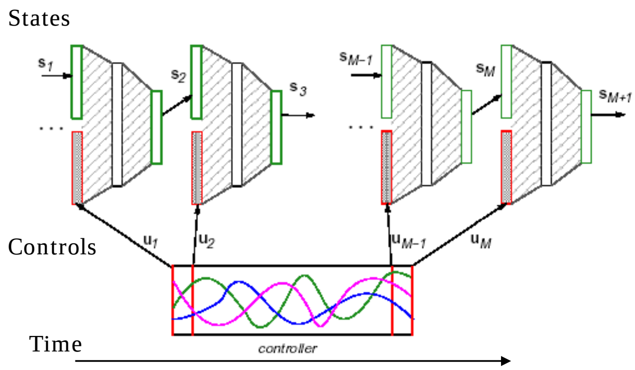
shared weights



time (iterations)

Neuro-animator

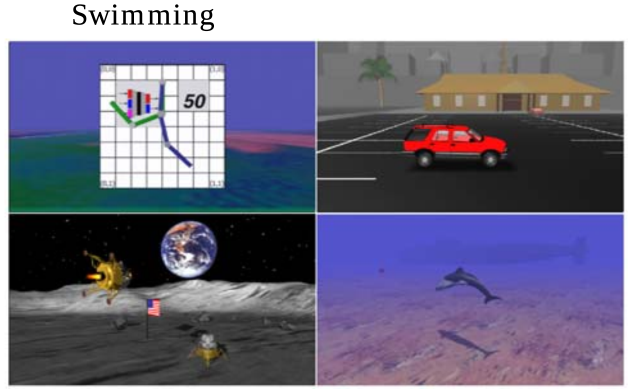
send controls u(t) to the dynamical system, observe resulting states x(t)
train a recurrent neural network that predicts the system dynamics...



The network computes the next state given the current state and control.

The same network is then replicated over time, using weight sharing.

Successful applications:
Reaching, Parking, Landing, Swimming



Given a desired final state, “invert” the network to get the appropriate control signals.

This is done by gradient descent, where the weight are fixed and the unknown controls u are treated as the parameters to be optimized.

It is straightforward to modify back-propagation to do this.

Control of a 2-link 6-muscle arm

(Huh and Todorov)