Backpropagation: The Good, the Bad and the Ugly

Keith L. Downing

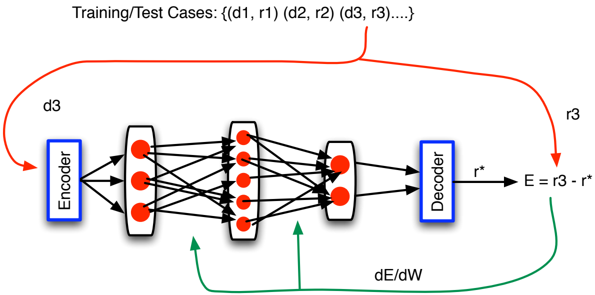
The Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
keithd@idi.ntnu.no

January 28, 2020

Supervised Learning

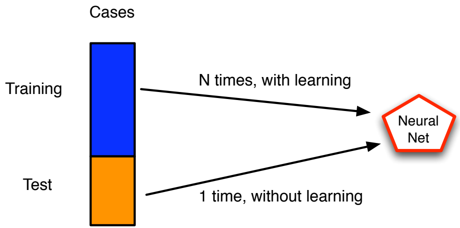
- Constant feedback from an instructor, indicating not only right/wrong, but also the **correct** answer for each training case.
- Many cases (i.e., input-output pairs) to be learned.
- Weights are modified by a complex procedure (back-propagation) based on output error.
- Feed-forward networks with back-propagation learning are the standard implementation.
- 99% of neural network applications use this.
- Typical usage: problems with a) lots of input-output training data, and b) goal of a mapping (function) from inputs to outputs.
- **Not** biologically plausible, although the cerebellum appears to exhibit some aspects.
- But, the **result** of backprop, a trained ANN to perform some function, can be very useful to neuroscientists as a **sufficiency proof**.

Backpropagation Overview



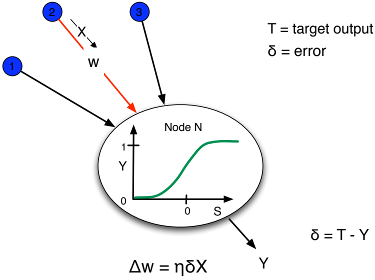
- **Feed-Forward Phase** - Inputs sent through the ANN to compute outputs.
- **Feedback Phase** - Error passed back from output to input layers and used to update weights along the way.

Training -vs- Testing



- **Generalization** - correctly handling test cases (that ANN has not been trained on).
- **Over-Training** - weights become so fine-tuned to the training cases that generalization suffers: failure on many test cases.

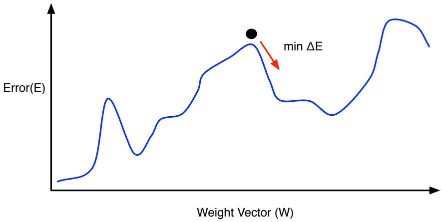
Widrow-Hoff (a.k.a. Delta) Rule



- Delta (δ) = error; Eta (η) = learning rate
- Goal: change w so as to reduce $\mid \delta \mid$.
- Intuitive: If $\delta > 0$, then we want to decrease it, so we must increase Y. Thus, we must increase the sum of weighted inputs to N, and we do that by increasing (decreasing) w if X is positive (negative).
- Similar for $\delta < 0$
- Assumes derivative of N's transfer function is everywhere non-negative.

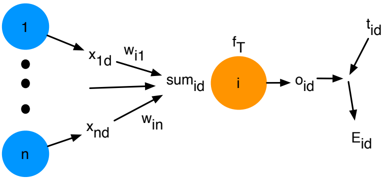
Gradient Descent

- Goal = minimize total error across all output nodes
- Method = modify weights throughout the network (i.e., at all levels) to follow the route of steepest descent in error space.



$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}}$$

Computing $\frac{\partial E_i}{\partial w_{ij}}$

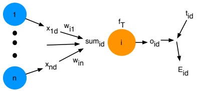


Sum of Squared Errors (SSE)

$$E_i = \frac{1}{2} \sum_{d \in D} (t_{id} - o_{id})^2$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{2} \sum_{d \in D} 2(t_{id} - o_{id}) \frac{\partial (t_{id} - o_{id})}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial (-o_{id})}{\partial w_{ij}}$$

Computing $\frac{\partial(-o_{id})}{\partial w_{ij}}$



Since output = f(sum weighted inputs)

$$\frac{\partial E}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}}$$

where

$$sum_{id} = \sum_{k=1}^{n} w_{ik} x_{kd}$$

Using Chain Rule: $\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g(x)} \times \frac{\partial g(x)}{\partial x}$

$$\frac{\partial(f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times x_{jd}$$

Computing $\frac{\partial sum_{id}}{\partial w_{ij}}$ - Easy!!

$$\frac{\partial sum_{id}}{\partial w_{ij}} = \frac{\partial \left(\sum_{k=1}^{n} w_{ik} x_{kd}\right)}{\partial w_{ij}} = \frac{\partial \left(w_{i1} x_{1d} + w_{i2} x_{2d} + \ldots + w_{ij} x_{jd} + \ldots + w_{in} x_{nd}\right)}{\partial w_{ij}}$$

$$= \frac{\partial (w_{i1} x_{1d})}{\partial w_{ij}} + \frac{\partial (w_{i2} x_{2d})}{\partial w_{ij}} + \ldots + \frac{\partial (w_{ij} x_{jd})}{\partial w_{ij}} + \ldots + \frac{\partial (w_{in} x_{nd})}{\partial w_{ij}}$$

$$= 0 + 0 + \ldots + x_{jd} + \ldots + 0 = x_{jd}$$

Computing $\frac{\partial f_T(sum_{id})}{\partial sum_{id}}$ - Harder for some f_T

f_T = Identity function: $f_T(sum_{id}) = sum_{id}$

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = 1$$

Thus:

$$\frac{\partial(f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} = 1 \times x_{jd} = x_{jd}$$

f_T = Sigmoid: $f_T(sum_{id}) = \frac{1}{1+e^{-sum_{id}}}$

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = o_{id}(1 - o_{id})$$

Thus:

$$\frac{\partial(f_T(sum_{id}))}{\partial w_{ij}} = \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} = o_{id}(1 - o_{id}) \times x_{jd} = o_{id}(1 - o_{id})x_{jd}$$

The only non-trivial calculation

$$\frac{\partial f_T(sum_{id})}{\partial sum_{id}} = \frac{\partial\left((1+e^{-sum_{id}})^{-1}\right)}{\partial sum_{id}} = (-1)\frac{\partial(1+e^{-sum_{id}})}{\partial sum_{id}}(1+e^{-sum_{id}})^{-2}$$

$$= (-1)(-1)e^{-sum_{id}}(1+e^{-sum_{id}})^{-2} = \frac{e^{-sum_{id}}}{(1+e^{-sum_{id}})^2}$$

But notice that:

$$\frac{e^{-sum_{id}}}{(1+e^{-sum_{id}})^2} = f_T(sum_{id})(1-f_T(sum_{id})) = o_{id}(1-o_{id})$$

Putting it all together

$$\frac{\partial E_i}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial(-f_T(sum_{id}))}{\partial w_{ij}} = -\sum_{d \in D} \left((t_{id} - o_{id}) \frac{\partial f_T(sum_{id})}{\partial sum_{id}} \times \frac{\partial sum_{id}}{\partial w_{ij}} \right)$$

So for f_T = Identity:

$$\frac{\partial E_i}{\partial w_{ij}} = -\sum_{d \in D} (t_{id} - o_{id}) x_{jd}$$

and for f_T= Sigmoid:

$$\frac{\partial E_i}{\partial w_{ij}} = -\sum_{d \in D} (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd}$$

Weight Updates (f_T = Sigmoid)

Batch: update weights after each training epoch

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} = \eta \sum_{d \in D} (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd}$$

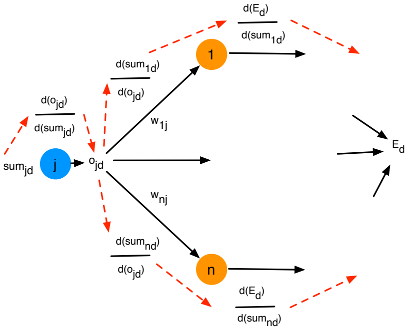
The weight changes are actually computed after each training case, but w_{ij} is not updated until the epoch's end.

Incremental: update weights after each training case

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} = \eta (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd}$$

- A lower learning rate (η) recommended here than for batch method.
- Can be dependent upon case-presentation order. So randomly sort the cases after each epoch.

Backpropagation in Multi-Layered Neural Networks

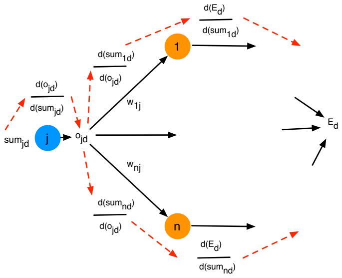


For each node (j) and each training case (d), backpropagation computes an error term:

$$\delta_{jd} = -\frac{\partial E_d}{\partial sum_{jd}}$$

by calculating the influence of sum_{jd} along each connection from node j to the next downstream layer.

Computing δ_{jd}



Along the upper path, the contribution to $\frac{\partial E_d}{\partial sum_{jd}}$ is:

$$\frac{\partial o_{jd}}{\partial sum_{jd}} \times \frac{\partial sum_{1d}}{\partial o_{jd}} \times \frac{\partial E_d}{\partial sum_{1d}}$$

So summing along all paths:

$$\frac{\partial E_d}{\partial sum_{jd}} = \frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} \frac{\partial sum_{kd}}{\partial o_{jd}} \frac{\partial E_d}{\partial sum_{kd}}$$

Computing δ_{jd}

Just as before, most terms are 0 in the derivative of the sum, so:

$$\frac{\partial sum_{kd}}{\partial o_{jd}} = w_{kj}$$

Assuming f_T = a sigmoid:

$$\frac{\partial o_{jd}}{\partial sum_{jd}} = \frac{\partial f_T(sum_{jd})}{\partial sum_{jd}} = o_{jd}(1 - o_{jd})$$

Thus:

$$\delta_{jd} = -\frac{\partial E_d}{\partial sum_{jd}} = -\frac{\partial o_{jd}}{\partial sum_{jd}} \sum_{k=1}^{n} \frac{\partial sum_{kd}}{\partial o_{jd}} \frac{\partial E_d}{\partial sum_{kd}}$$

$$= -o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}(-\delta_{kd}) = o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}\delta_{kd}$$

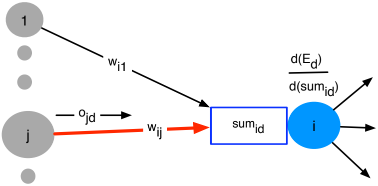
Computing δ_{jd}

Note that δ_{jd} is defined recursively in terms of the δ values in the next downstream layer:

$$\delta_{jd} = o_{jd}(1 - o_{jd}) \sum_{k=1}^{n} w_{kj}\delta_{kd}$$

So all δ values in the network can be computed by moving backwards, one layer at a time.

Computing $\frac{\partial E_d}{\partial w_{ij}}$ from δ_{jd} - Easy!!



The only effect of w_{ij} upon the error is via its effect upon sum_{id}, which is:

$$\frac{\partial sum_{id}}{\partial w_{ij}} = o_{jd}$$

So:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial sum_{id}}{\partial w_{ij}} \times \frac{\partial E_d}{\partial sum_{id}} = \frac{\partial sum_{id}}{\partial w_{ij}} \times (-\delta_{id}) = -o_{jd}\delta_{id}$$

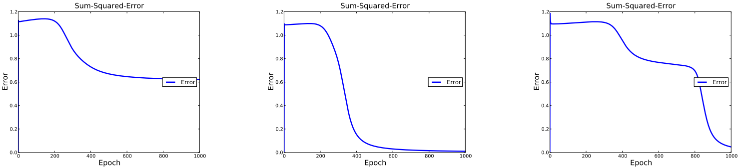
Computing Δw_{ij}

Given an error term, δ_{id} (for node i on training case d), the update of w_{ij} for all nodes j that feed into i is:

$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}} = -\eta(-o_{jd}\delta_{id}) = \eta \delta_{id} o_{jd}$$

So given δ_i, you can easily calculate Δw_{ij} for all incoming arcs.

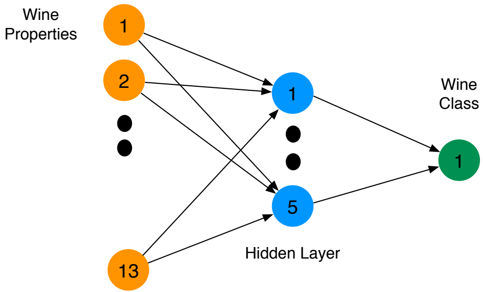
Learning XOR



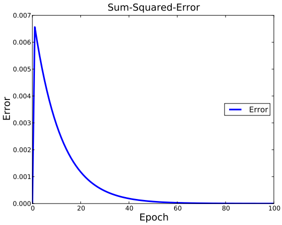
- Epoch = All 4 entries of the XOR truth table.
- 2 (inputs) X 2 (hidden) X 1 (output) network
- Random init of all weights in [-1 1].
- Not linearly separable, so it takes awhile!
- Each run is different due to random weight init.

Learning to Classify Wines

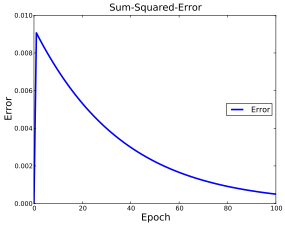
Class	Properties							
1	14.23	1.71	2.43	15.6	127	2.8	$\cdots$	$\cdots$
1	13.2	1.78	2.14	11.2	100	2.65	$\cdots$	$\cdots$
2	13.11	1.01	1.7	15	78	2.98	$\cdots$	$\cdots$
3	13.17	2.59	2.37	20	120	1.65	$\cdots$	$\cdots$
$\vdots$								



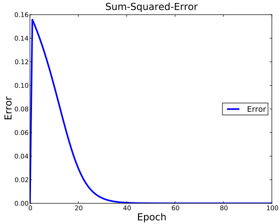
Wine Runs



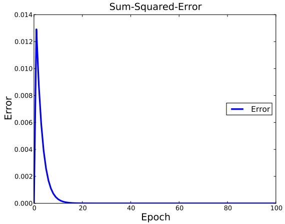
13x5x1 (lrate = 0.3)



13x5x1 (lrate = 0.1)



13 x 10 x 1 (lrate = 0.3)



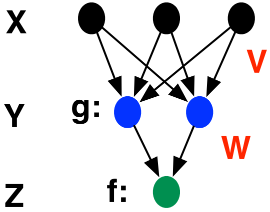
13 x 25 x 1 (lrate = 0.3)

WARNING: Notation Change

- When working with matrices, it is often easier to let w_{ij} denote the weight on the connection FROM node i TO node j.
- Then, the normal syntax for matrix row-column indices implies that:
  - The values in row i are written as $w_{i?}$ and denote the weights on the **outputs** of node i.
  - The values in column j are written as $w_{?j}$ and denote the weights on the **inputs** to node j.

$$\begin{array}{rc} & \\ \text{Weights from node 2} \rightarrow & \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \\ & \begin{array}{ccc} & & \downarrow \\ & & \text{Weights to node 3} \end{array} \end{array}$$

Backpropagation using Tensors



$R = XV$

$Y = g(R)$

$S = YW$

$Z = f(S)$

$X = [x_1, x_2, x_3]$

$Y = [y_1, y_2]$

$Z = [z_1]$

$$V = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \\ v_{31} & v_{32} \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} \\ w_{21} \end{bmatrix}$$

$f = g = \text{sigmoid}$

v_{ij} = weight from i to j

Goal: Compute $\frac{\partial Z}{\partial W}$

- Using the Chain Rule of Tensor Calculus:

$$\frac{\partial Z}{\partial W} = \frac{\partial f(S)}{\partial W} = \frac{\partial f(S)}{\partial S} \times \frac{\partial S}{\partial W} = \frac{\partial f(S)}{\partial S} \times \frac{\partial (Y \bullet W)}{\partial W}$$

- f(S) is a sigmoid, with derivative f(S)(1-f(S)), so simplify to:

$$z_1(1 - z_1)\frac{\partial (Y \bullet W)}{\partial W}$$

- Using the product rule:

$$z_1(1 - z_1) \times \frac{\partial (Y \bullet W)}{\partial W} = z_1(1 - z_1)[\color{red}{\frac{\partial Y}{\partial W} \bullet W} + Y \bullet \frac{\partial W}{\partial W}]$$

- Y is independent of W, so red term = 0. But $\frac{\partial W}{\partial W}$ is not simply 1, but:

$$\begin{pmatrix} \left| \begin{matrix} 1 \\ 0 \end{matrix} \right| \\ \\ \left| \begin{matrix} 0 \\ 1 \end{matrix} \right| \end{pmatrix}$$

Computing $\frac{\partial Z}{\partial W}$

$$Y \bullet \frac{\partial W}{\partial W} = (y_1, y_2) \bullet \begin{pmatrix} \begin{vmatrix} 1 \\ 0 \end{vmatrix} \\ \\ \begin{vmatrix} 0 \\ 1 \end{vmatrix} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

- Putting it all together:

$$\frac{\partial Z}{\partial W} = z_1(1 - z_1) \times Y \bullet \frac{\partial W}{\partial W} = \begin{pmatrix} z_1(1 - z_1)y_1 \\ \\ z_1(1 - z_1)y_2 \end{pmatrix} = \begin{pmatrix} \frac{\partial Z}{\partial w_{1,1}} \\ \\ \frac{\partial Z}{\partial w_{2,1}} \end{pmatrix}$$

- For each case, c_k, in a minibatch, the forward pass will produce values for X, Y, and Z. Those values will be used to compute actual numeric gradients by filling in these symbolic gradients:

$$\frac{\partial Z}{\partial W}\Big|_{c_k} = \begin{pmatrix} z_1(1 - z_1)y_1 \\ \\ z_1(1 - z_1)y_2 \end{pmatrix}\Bigg|_{c_k}$$

Tensor Calculations across Several Layers

Goal: Compute $\frac{\partial Z}{\partial V}$

- Expanding using the Chain Rule:

$$\frac{\partial Z}{\partial V} = \frac{\partial f(S)}{\partial V} = \frac{\partial f(S)}{\partial S} \times \frac{\partial S}{\partial V} = z_1(1 - z_1) \times \frac{\partial (Y \bullet W)}{\partial V}$$

- Using the Product Rule:

$$\frac{\partial (Y \bullet W)}{\partial V} = \frac{\partial Y}{\partial V} \bullet W + {\color{red} Y \bullet \frac{\partial W}{\partial V}} = \frac{\partial (g(R))}{\partial V} \bullet W$$

- W is indep. of V, so red term = 0; and Y = g(R).
- Using the Chain Rule:

$$\frac{\partial (g(R))}{\partial V} \bullet W = [\frac{\partial g(R)}{\partial R} \times \frac{\partial R}{\partial V}] \bullet W$$

- g(R) is also a sigmoid, so:

$$\frac{\partial g(R)}{\partial R} = g(R)(1 - g(R)) = Y(1 - Y) = [y_1(1 - y_1), y_2(1 - y_2)]$$

Computing $\frac{\partial Z}{\partial V}$

- Since $R = X \bullet V$, and using the Product Rule:

$$\frac{\partial R}{\partial V} = \frac{\partial (X \bullet V)}{\partial V} = X \bullet \frac{\partial V}{\partial V} + {\color{red}\frac{\partial X}{\partial V} \bullet V}$$

- X is indep. of V, so red term = 0; $\frac{\partial V}{\partial V}$ is 4-dimensional:

$$\frac{\partial V}{\partial V} = \begin{pmatrix} \begin{vmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{vmatrix} & \begin{vmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{vmatrix} \\ \begin{vmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{vmatrix} & \begin{vmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{vmatrix} \\ \begin{vmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{vmatrix} & \begin{vmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{vmatrix} \end{pmatrix}$$

Computing $\frac{\partial Z}{\partial V}$

- Take dot product of X = $[x_1, x_2, x_3]$ with each internal maxtrix of $\frac{\partial V}{\partial V}$:

$$X \bullet \frac{\partial V}{\partial V} = [x_1, x_2, x_3] \bullet \begin{pmatrix} \begin{vmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{vmatrix} & \begin{vmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{vmatrix} \\ \\ \begin{vmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{vmatrix} & \begin{vmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{vmatrix} \\ \\ \begin{vmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{vmatrix} & \begin{vmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{vmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} \begin{vmatrix} x_1 & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_1 \end{vmatrix} \\ \\ \begin{vmatrix} x_2 & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_2 \end{vmatrix} \\ \\ \begin{vmatrix} x_3 & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_3 \end{vmatrix} \end{pmatrix}$$

Computing $\frac{\partial Z}{\partial V}$

- Putting the pieces back together:

$$\frac{\partial g(R)}{\partial R} \times \frac{\partial R}{\partial V} = [y_1(1-y_1), y_2(1-y_2)] \times \begin{pmatrix} \begin{vmatrix} x_1 & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_1 \end{vmatrix} \\ \begin{vmatrix} x_2 & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_2 \end{vmatrix} \\ \begin{vmatrix} x_3 & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_3 \end{vmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} \begin{vmatrix} x_1 y_1(1-y_1) & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_1 y_2(1-y_2) \end{vmatrix} \\ \begin{vmatrix} x_2 y_1(1-y_1) & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_2 y_2(1-y_2) \end{vmatrix} \\ \begin{vmatrix} x_3 y_1(1-y_1) & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_3 y_2(1-y_2) \end{vmatrix} \end{pmatrix}$$

Computing $\frac{\partial Z}{\partial V}$

$$\frac{\partial(g(R))}{\partial V} \bullet W = [\frac{\partial g(R)}{\partial R} \times \frac{\partial R}{\partial V}] \bullet W$$

$$= \begin{pmatrix} \begin{vmatrix} x_1 y_1(1-y_1) & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_1 y_2(1-y_2) \end{vmatrix} \\ \begin{vmatrix} x_2 y_1(1-y_1) & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_2 y_2(1-y_2) \end{vmatrix} \\ \begin{vmatrix} x_3 y_1(1-y_1) & 0 \end{vmatrix} & \begin{vmatrix} 0 & x_3 y_2(1-y_2) \end{vmatrix} \end{pmatrix} \bullet \begin{pmatrix} w_{11} \\ w_{21} \end{pmatrix}$$

$$= \begin{pmatrix} x_1 y_1(1-y_1)w_{11} & x_1 y_2(1-y_2)w_{21} \\ x_2 y_1(1-y_1)w_{11} & x_2 y_2(1-y_2)w_{21} \\ x_3 y_1(1-y_1)w_{11} & x_3 y_2(1-y_2)w_{21} \end{pmatrix} = \frac{\partial(Y \bullet W)}{\partial V}$$

Computing $\frac{\partial Z}{\partial V}$

...and finally...

$$\frac{\partial Z}{\partial V} = \frac{\partial f(S)}{\partial V} = z_1(1-z_1) \times \frac{\partial (Y \bullet W)}{\partial V}$$

$$= z_1(1-z_1) \times \begin{pmatrix} x_1 y_1(1-y_1)w_{11} & x_1 y_2(1-y_2)w_{21} \\ x_2 y_1(1-y_1)w_{11} & x_2 y_2(1-y_2)w_{21} \\ x_3 y_1(1-y_1)w_{11} & x_3 y_2(1-y_2)w_{21} \end{pmatrix}$$

$$= \begin{pmatrix} z_1(1-z_1)x_1 y_1(1-y_1)w_{11} & z_1(1-z_1)x_1 y_2(1-y_2)w_{21} \\ z_1(1-z_1)x_2 y_1(1-y_1)w_{11} & z_1(1-z_1)x_2 y_2(1-y_2)w_{21} \\ z_1(1-z_1)x_3 y_1(1-y_1)w_{11} & z_1(1-z_1)x_3 y_2(1-y_2)w_{21} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\partial Z}{\partial v_{11}} & \frac{\partial Z}{\partial v_{12}} \\ \frac{\partial Z}{\partial v_{21}} & \frac{\partial Z}{\partial v_{22}} \\ \frac{\partial Z}{\partial v_{31}} & \frac{\partial Z}{\partial v_{32}} \end{pmatrix}$$

Gradients and the Minibatch

- For every case c_k in a minibatch, the values of X, Y and Z are computed during the forward pass. The symbolic gradients in $\frac{\partial Z}{\partial V}$ are filled in using these values in X,Y and Z (along with W), producing one numeric, 3 x 2 matrix **per case**:

$$\left(\frac{\partial Z}{\partial V}\right)|_{c_k} = \begin{pmatrix} \frac{\partial Z}{\partial v_{11}} & \frac{\partial Z}{\partial v_{12}} \\ \frac{\partial Z}{\partial v_{21}} & \frac{\partial Z}{\partial v_{22}} \\ \frac{\partial Z}{\partial v_{31}} & \frac{\partial Z}{\partial v_{32}} \end{pmatrix}|_{c_k}$$

$$= \begin{pmatrix} z_1(1-z_1)x_1y_1(1-y_1)w_{11} & z_1(1-z_1)x_1y_2(1-y_2)w_{21} \\ z_1(1-z_1)x_2y_1(1-y_1)w_{11} & z_1(1-z_1)x_2y_2(1-y_2)w_{21} \\ z_1(1-z_1)x_3y_1(1-y_1)w_{11} & z_1(1-z_1)x_3y_2(1-y_2)w_{21} \end{pmatrix}|_{c_k}$$

Gradients and the Minibatch

- In most Deep Learning situations, the gradients will be based on a loss function, L, not simply the output of the final layer. But that's just one more level of derivative calculations (see below).
- At the completion of a minibatch, the numeric gradient matrices are added together to yield the complete gradient, which is then used to update the weights.
- For any weight matrix U in the network, and minibatch M, update weight $u_{i,j}$ as follows:

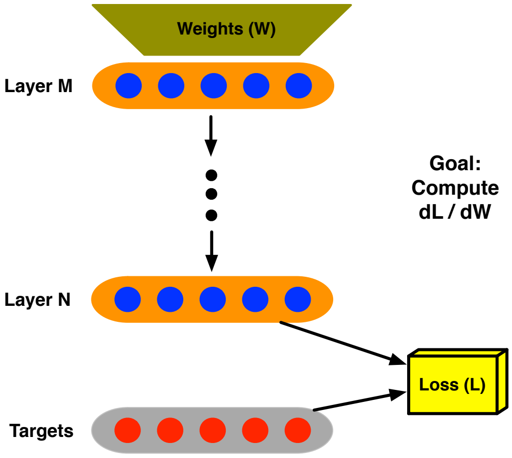
$$\left(\frac{\partial L}{\partial u_{i,j}}\right)\Big|_{M} = \sum_{c_k \in M} \left(\frac{\partial L}{\partial u_{i,j}}\right)\Big|_{c_k}$$

η = learning rate

$$\triangle u_{i,j} = -\eta \left(\frac{\partial L}{\partial u_{i,j}}\right)\Big|_{M}$$

- Tensorflow and PyTorch do all of this for you.
- Everytime you write Deep Learning code, be grateful!!

The Jacobian Product Chain



$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial N} \bullet \frac{\partial N}{\partial N-1} \bullet \cdots \bullet \frac{\partial M+1}{\partial M} \bullet \frac{\partial M}{\partial W}$$

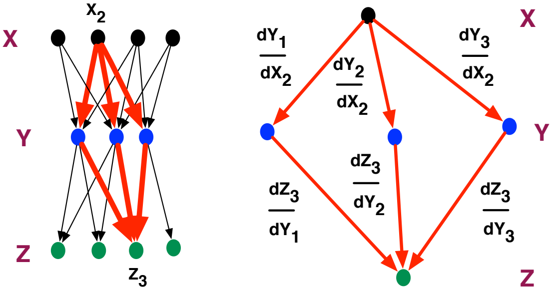
Jacobian Matrix Linking Neighboring Layers: $Y \rightarrow Z$

$$\mathbf{J}_Y^Z = \begin{pmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_1}{\partial y_2} & \cdots & \frac{\partial z_1}{\partial y_n} \\ \frac{\partial z_2}{\partial y_1} & \frac{\partial z_2}{\partial y_2} & \cdots & \frac{\partial z_2}{\partial y_n} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial z_m}{\partial y_1} & \frac{\partial z_m}{\partial y_2} & \cdots & \frac{\partial z_m}{\partial y_n} \end{pmatrix}$$

Jacobian Linking a Layer's Outputs(Z) to its Incoming Weights (W)

$$\mathbf{J}^Z_W = \begin{pmatrix} \frac{\partial Z}{\partial w_{11}} & \frac{\partial Z}{\partial w_{12}} & \cdots & \frac{\partial Z}{\partial w_{1m}} \\ \frac{\partial Z}{\partial w_{21}} & \frac{\partial Z}{\partial w_{22}} & \cdots & \frac{\partial Z}{\partial w_{2m}} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial Z}{\partial w_{n1}} & \frac{\partial Z}{\partial w_{n2}} & \cdots & \frac{\partial Z}{\partial w_{nm}} \end{pmatrix}$$

Distant Gradients = Sums of Path Products



$$\frac{\partial z_3}{\partial x_2} = \frac{\partial z_3}{\partial y_1}\frac{\partial y_1}{\partial x_2} + \frac{\partial z_3}{\partial y_2}\frac{\partial y_2}{\partial x_2} + \frac{\partial z_3}{\partial y_3}\frac{\partial y_3}{\partial x_2}$$

More generally:

$$\frac{\partial z_i}{\partial x_j} = \sum_{k \in Y} \frac{\partial z_i}{\partial y_k}\frac{\partial y_k}{\partial x_j}$$

This results from the multiplication of two Jacobian matrices.

Multiplying Jacobian Matrices

$$J_y^z \bullet J_x^y = \begin{vmatrix} \vdots & \vdots & \vdots \\ \frac{\partial z_i}{\partial y_1} & \frac{\partial z_i}{\partial y_2} & \frac{\partial z_i}{\partial y_3} \\ \vdots & \vdots & \vdots \end{vmatrix} \bullet \begin{vmatrix} \cdots & \frac{\partial y_1}{\partial x_j} & \cdots \\ \cdots & \frac{\partial y_2}{\partial x_j} & \cdots \\ \cdots & \frac{\partial y_3}{\partial x_j} & \cdots \end{vmatrix}$$

$$= \begin{vmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & \frac{\partial z_i}{\partial y_1}\frac{\partial y_1}{\partial x_j} + \frac{\partial z_i}{\partial y_2}\frac{\partial y_2}{\partial x_j} + \frac{\partial z_i}{\partial y_3}\frac{\partial y_3}{\partial x_j} \cdots & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{vmatrix} = J_x^z$$

We can do this repeatedly to form J_m^n, the Jacobian relating the activation levels of upstream layer m with those of downstream layer n:

- R = Identity Matrix
- For q = n down to m+1 do:
  - $R \leftarrow R \bullet J_{q-1}^q$
- $J_m^n \leftarrow R$

Last of the Jacobians

Once we've computed J_m^n, we need one final Jacobian: J_w^m, where w are the weights feeding into layer m. Then we have the standard Jacobian, J_w^n needed for updating all weights in matrix w.

$$J_w^n \leftarrow J_m^n \bullet J_w^m$$

Weights V from layer X to Y (from the earlier example)

$$J_V^Y = \frac{\partial Y}{\partial V} = \begin{vmatrix} \begin{pmatrix} \frac{\partial y_1}{\partial v_{11}} & \frac{\partial y_2}{\partial v_{11}} \end{pmatrix}^T & \begin{pmatrix} \frac{\partial y_1}{\partial v_{12}} & \frac{\partial y_2}{\partial v_{12}} \end{pmatrix}^T \\ \begin{pmatrix} \frac{\partial y_1}{\partial v_{21}} & \frac{\partial y_2}{\partial v_{21}} \end{pmatrix}^T & \cdots \\ \begin{pmatrix} \frac{\partial y_1}{\partial v_{31}} & \frac{\partial y_2}{\partial v_{31}} \end{pmatrix}^T & \cdots \end{vmatrix}$$

- Since a) the act func for Y is sigmoid, and b) $\frac{\partial y_k}{\partial v_{ij}} = 0$ when $j \neq k$:

$$J_V^Y = \frac{\partial Y}{\partial V} = \begin{vmatrix} \begin{pmatrix} y_1(1-y_1)x_1 & 0 \end{pmatrix}^T & \begin{pmatrix} 0 & y_2(1-y_2)x_1 \end{pmatrix}^T \\ \begin{pmatrix} y_1(1-y_1)x_2 & 0 \end{pmatrix}^T & \begin{pmatrix} 0 & y_2(1-y_2)x_2 \end{pmatrix}^T \\ \begin{pmatrix} y_1(1-y_1)x_3 & 0 \end{pmatrix}^T & \begin{pmatrix} 0 & y_2(1-y_2)x_3 \end{pmatrix}^T \end{vmatrix}$$

- Inner vectors are transposed to fit them on the page.

Last of the Jacobian Multiplications

From the previous example (the network with layer sizes 3-2-1), once we have J_Y^Z and J_V^Y, we can multiply (taking dot products of J_Y^Z with the inner vectors of J_V^Y) to produce J_V^Z: the matrix of gradients that backprop uses to modify the weights in V.

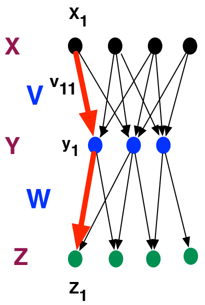
$$J_V^Z = J_Y^Z \bullet J_V^Y = \left| \begin{array}{cc} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_1}{\partial y_2} \end{array} \right| \bullet \frac{\partial Y}{\partial V}$$

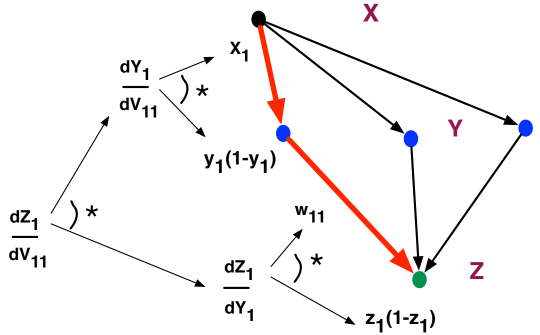
- Since a) the act func for Z is sigmoid, and b) $\frac{\partial sum(z_k)}{\partial y_i} = w_{ik}$ (where $sum(z_k)$ = sum of weighted inputs to node z_k):

$$= \left| \begin{array}{cc} z_1(1-z_1)w_{11} & z_1(1-z_1)w_{21} \end{array} \right| \bullet \frac{\partial Y}{\partial V}$$

$$J_V^Z = \begin{pmatrix} z_1(1-z_1)x_1y_1(1-y_1)w_{11} & z_1(1-z_1)x_1y_2(1-y_2)w_{21} \\ z_1(1-z_1)x_2y_1(1-y_1)w_{11} & z_1(1-z_1)x_2y_2(1-y_2)w_{21} \\ z_1(1-z_1)x_3y_1(1-y_1)w_{11} & z_1(1-z_1)x_3y_2(1-y_2)w_{21} \end{pmatrix}$$

One entry of J_V^Z





$$J_V^Z(1,1) = z_1(1-z_1)x_1 y_1(1-y_1)w_{11}$$

First of the Jacobians

- This iterative process is very general, and permits the calculation of J_M^N ($M < N$) for any layers M and N, or J_W^N for an weights (W) upstream of N.
- However, the standard situation in backpropagation is to compute $J_{W_i}^L \forall i$ where L is the objective (loss) function and W_i are the weight matrices.
- This follows the same procedure as sketched above, but the series of dot products **begins** with J_N^L: the Jacobian of derivatives of the loss function w.r.t. the activations of the output layer.
- For example, assume L = Mean Squared Error (MSE), Z is an output layer of 3 sigmoid nodes, and t_i are the target values for those 3 nodes for a particular case (c):

$$L(c) = \frac{1}{3}\sum_{i=1}^{3}(z_i - t_i)^2$$

Taking partial derivatives of L(c) w.r.t. the z_i yields:

$$J_Z^L = \frac{\partial L}{\partial Z} = \left| \begin{array}{ccc} \frac{2}{3}(z_1 - t_1) & \frac{2}{3}(z_2 - t_2) & \frac{2}{3}(z_3 - t_3) \end{array} \right|$$

First of the Jacobian Multiplications

- Continuing the example: Assume that layer Y (of size 2) feeds into layer Z, using weights W, then:

$$J_Y^Z = \begin{vmatrix} z_1(1-z_1)w_{11} & z_1(1-z_1)w_{21} \\ z_2(1-z_2)w_{12} & z_2(1-z_2)w_{22} \\ z_3(1-z_3)w_{13} & z_3(1-z_3)w_{23} \end{vmatrix}$$

- Our first Jacobian multiplication yields J_Y^L, a 1 x 2 row vector (shown transposed to fit on the page):

$$J_Z^L \bullet J_Y^Z = J_Y^L = \begin{vmatrix} \frac{\partial L}{\partial y_1} & \frac{\partial L}{\partial y_2} \end{vmatrix} =$$

$$\begin{vmatrix} \frac{2}{3}(z_1-t_1)z_1(1-z_1)w_{11} + \frac{2}{3}(z_2-t_2)z_2(1-z_2)w_{12} + \frac{2}{3}(z_3-t_3)z_3(1-z_3)w_{13} \\ \frac{2}{3}(z_1-t_1)z_1(1-z_1)w_{21} + \frac{2}{3}(z_2-t_2)z_2(1-z_2)w_{22} + \frac{2}{3}(z_3-t_3)z_3(1-z_3)w_{23} \end{vmatrix}^T$$

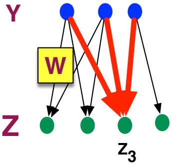
Backpropagation with Tensors: The Big Picture

General Algorithm

- Assume Layer M is upstream of Layer N (the output layer). So $M < N$.
- Assume V is the tensor of weights feeding into Layer M.
- Assume L is the loss function.
- Goal: Compute $J_V^L = \frac{\partial L}{\partial V}$
- R = J_N^L (the partial derivatives of the loss function w.r.t. the output layer)
- If output layer is Softmax: R $\leftarrow R \bullet J^{soft}$
- For q = N down to M + 1 do:
  - $R \leftarrow R \bullet J_{q-1}^{q}$
- $J_V^L \leftarrow R \bullet J_V^M$
- Use J_V^L to update the weights in V.

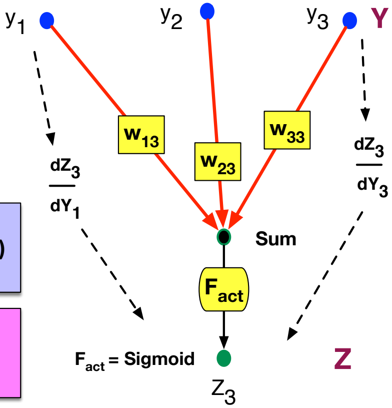
... And now some practical details on implementing this...

Building the J_Y^Z and J_W^Z Jacobians



$$\frac{dZ_3}{dy_1} = w_{13}z_3(1-z_3)$$

$$\frac{dZ_3}{dw_{13}} = y_1z_3(1-z_3)$$



Detailed Entries of J_Y^Z

$$J_Y^Z =$$

$$\begin{pmatrix} w_{11}z_1(1-z_1) & w_{21}z_1(1-z_1) & \cdots & w_{n1}z_1(1-z_1) \\ w_{12}z_2(1-z_2) & w_{22}z_2(1-z_2) & \cdots & w_{n2}z_2(1-z_2) \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ w_{1m}z_m(1-z_m) & w_{2m}z_m(1-z_m) & \cdots & w_{nm}z_m(1-z_m) \end{pmatrix}$$

More succinctly:

$$J_Y^Z = (W \bullet J_{Sum}^Z)^T = J_{Sum}^Z \bullet W^T$$

Jacobian J^{Z}_{Sum} Linking Z to Summed Inputs

$$J^{Z}_{Sum} =$$

$$\begin{pmatrix} z_1(1-z_1) & 0 & \cdots & 0 \\ 0 & z_2(1-z_2) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & z_m(1-z_m) \end{pmatrix}$$

Simplifying J_W^Z

In an earlier slide, J_W^Z was presented. Note that it has the same shape as the weight matrix, W:

$$\mathbf{J}_W^Z = \begin{pmatrix} \frac{\partial Z}{\partial w_{11}} & \frac{\partial Z}{\partial w_{12}} & \cdots & \frac{\partial Z}{\partial w_{1m}} \\ \frac{\partial Z}{\partial w_{21}} & \frac{\partial Z}{\partial w_{22}} & \cdots & \frac{\partial Z}{\partial w_{2m}} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial Z}{\partial w_{n1}} & \frac{\partial Z}{\partial w_{n2}} & \cdots & \frac{\partial Z}{\partial w_{nm}} \end{pmatrix}$$

- Weight w_{ik} connects y_i to z_k.
- So z_k is the only element of Z that w_{ik} affects.
- Thus, $\frac{\partial Z}{\partial w_{ik}} = [0, \cdots, y_i z_k(1 - z_k), \cdots, 0]^T$ (only kth entry is non-zero).
- Make a simpler, more practical, matrix of only these positive values: $\hat{J}_W^Z$.

The Simplified Jacobian: $\hat{J}_W^Z$

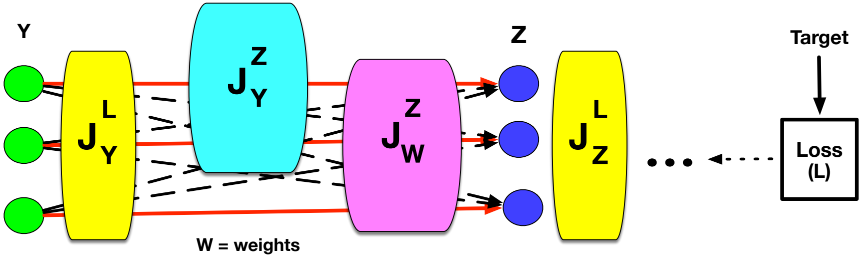
$$\begin{pmatrix} \frac{\partial z_1}{\partial w_{11}} & \frac{\partial z_2}{\partial w_{12}} & \cdots & \frac{\partial z_m}{\partial w_{1m}} \\ \frac{\partial z_1}{\partial w_{21}} & \frac{\partial z_2}{\partial w_{22}} & \cdots & \frac{\partial z_m}{\partial w_{2m}} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial z_1}{\partial w_{n1}} & \frac{\partial z_2}{\partial w_{n2}} & \cdots & \frac{\partial z_m}{\partial w_{nm}} \end{pmatrix} = \begin{pmatrix} y_1 z_1(1-z_1) & y_1 z_2(1-z_2) & \cdots & y_1 z_m(1-z_m) \\ y_2 z_1(1-z_1) & y_2 z_2(1-z_2) & \cdots & y_2 z_m(1-z_m) \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ y_n z_1(1-z_1) & y_n z_2(1-z_2) & \cdots & y_n z_m(1-z_m) \end{pmatrix}$$

More succinctly:

$$\hat{J}_W^Z = Y \otimes Diag(J_{Sum}^Z)$$

where $\otimes$ = Outer Product and Diag(J_{Sum}^Z) = diagonal of J_{Sum}^Z.

The Backward Pass across a Layer



1. Receive J^L_Z from downstream.
2. Compute $J^L_W \leftarrow J^L_Z \bullet J^Z_W$ and use it to update W.
3. Compute $J^L_Y \leftarrow J^L_Z \bullet J^Z_Y$ and pass it back (upstream).

Details of $J^L_W \leftarrow J^L_Z \bullet J^Z_W$

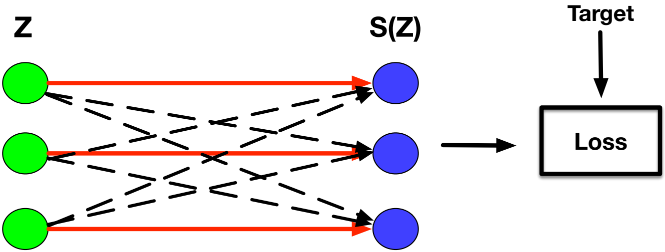
- Remember that every element of J^Z_W is a vector, so $J^L_Z \bullet J^Z_W$ involves many vector dot products: $J^L_Z \bullet Q$ for each vector (Q) in J^Z_W. This produces a final matrix with the same dimensions as W.
- In numpy, the "dot" function produces the desired result, but it is sensitive to argument; the proper call is: numpy.dot(J^Z_W, J^L_Z)
- If we use $\hat{J}^Z_W$ instead of J^Z_W, then the dot product is replaced by this:

$$J^L_W \leftarrow J^L_Z \times \hat{J}^Z_W$$

- The use of "$\times$" means that the ith element of J^L_Z is multiplied by every item in the ith column of $\hat{J}^Z_W$
- In numpy, the standard multiplication operator, "*" between the vector J^L_Z and the matrix $\hat{J}^Z_W$ will perform the desired operation, as long as $\|J^L_Z\|$ = the number of columns in $\hat{J}^Z_W$.

SoftMax: One More Layer (but without weights)

$$Softmax(Z) = S(Z)$$



$$s_i = S(z_i) = \frac{e^{z_i}}{\sum_{z_k \in Z} e^{z_k}} = \frac{e^{z_i}}{\Sigma}$$

Derivatives of Softmax

Effect of z_i on $S(z_i)$

$$\frac{\partial S(z_i)}{\partial z_i} = \frac{\frac{\partial e^{z_i}}{\partial z_i}\Sigma - \frac{\partial \Sigma}{\partial z_i} e^{z_i}}{\Sigma^2} = \frac{e^{z_i}\Sigma - e^{z_i}e^{z_i}}{\Sigma^2} =$$

$$\frac{e^{z_i}}{\Sigma} - \left(\frac{e^{z_i}}{\Sigma}\right)^2 = S(z_i) - S(z_i)^2 = s_i - s_i^2$$

Effect of z_i on $S(z_k)$ where $i \neq k$

$$\frac{\partial S(z_k)}{\partial z_i} = \frac{\frac{\partial e^{z_k}}{\partial z_i}\Sigma - \frac{\partial \Sigma}{\partial z_i} e^{z_k}}{\Sigma^2} = \frac{0 - e^{z_i}e^{z_k}}{\Sigma^2} =$$

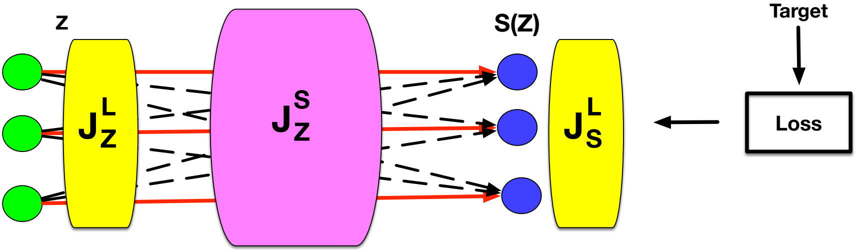
$$-\frac{e^{z_i}}{\Sigma}\frac{e^{z_k}}{\Sigma} = -S(z_i)S(z_k) = -s_i s_k$$

The Softmax (m x m) Jacobian Matrix

$$J^{soft}$$

$$\mathbf{J}_Z^S = \begin{pmatrix} \frac{\partial s_1}{\partial z_1} & \frac{\partial s_1}{\partial z_2} & \cdots & \frac{\partial s_1}{\partial z_m} \\ \frac{\partial s_2}{\partial z_1} & \frac{\partial s_2}{\partial z_2} & \cdots & \frac{\partial s_2}{\partial z_m} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial s_m}{\partial z_1} & \frac{\partial s_m}{\partial z_2} & \cdots & \frac{\partial z_m}{\partial z_m} \end{pmatrix} = \begin{pmatrix} s_1 - s_1^2 & -s_1 s_2 & \cdots & -s_1 s_m \\ -s_2 s_1 & s_2 - s_2^2 & \cdots & -s_2 s_m \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ -s_m s_1 & -s_m s_2 & \cdots & s_m - s_m^2 \end{pmatrix}$$

Backward Pass Across Softmax



$$J^L_Z = J^L_S \bullet J^S_Z$$

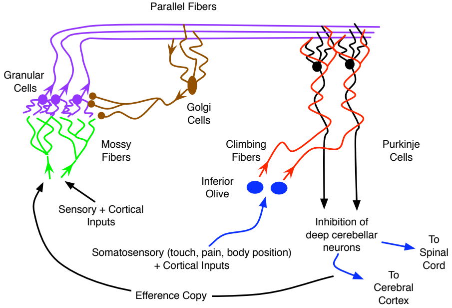
This assumes J^L_S is a row vector. If it's a column vector, then:

$$J^L_Z = \left(J^L_S\right)^T \bullet J^S_Z$$

Practical Tips

1. Only add as many hidden layers and hidden nodes as necessary. Too many $\rightarrow$ more weights to learn + increased chance of over-specialization.
2. Scale all input values to the same range, typically [0 1] or [-1 1].
3. Use target values of 0.1 (for zero) and 0.9 (for 1) to avoid saturation effects of sigmoids.
4. Beware of tricky encodings of input (and decodings of output) values. Don't combine too much info into a single node's activation value (even though it's fun to try), since this can make proper weights difficult (or impossible) to learn.
5. For discrete (e.g. nominal) values, one (input or output) node per value is often most effective. E.g. car model and city of residence -vs- income and education for assessing car-insurance risk.
6. All initial weights should be relatively small: [-0.1 0.1]
7. Bias nodes can be helpful for complicated data sets.
8. Check that all your layer sizes, activation functions, activation ranges, weight ranges, learning rates, etc. *make sense* in terms of each other and your goals for the ANN. One improper choice can ruin the results.

Supervised Learning in the Cerebellum



- Granular cells detect contexts.
- Parallel fibers and Purkinje cells map contexts to actions
- Climbing fibers from Inferior Olive provide (supervisory) feedback signals for LTD on Parallel-Purkinje synapses