Learning Systems 2018:
Lecture 6 – Back Propagation

Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \tag{1}$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \tag{2}$$

NATURE VOL. 323 9 OCTOBER 1986

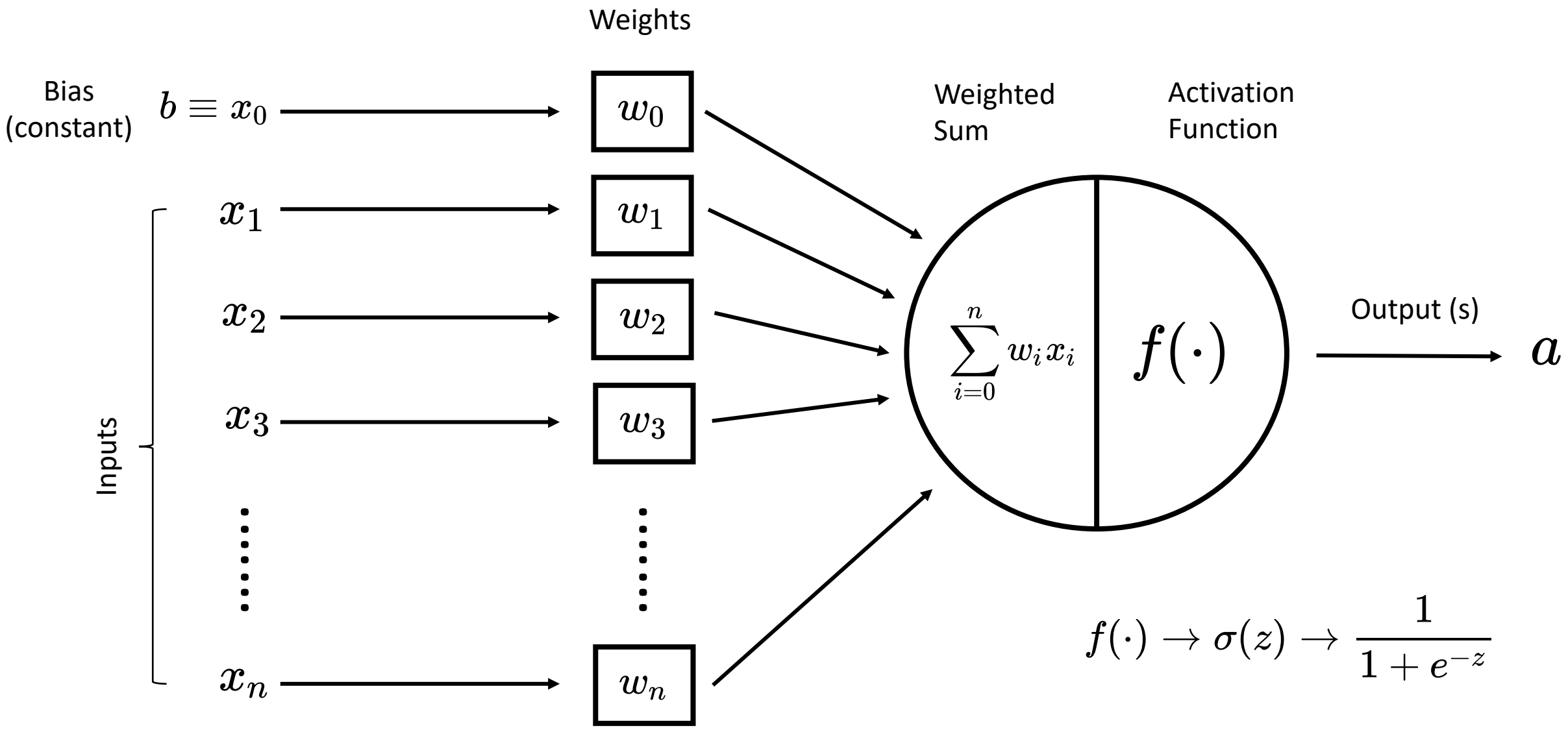
At its essence backpropagation is just a clever application of the chain rule.

The chain rule is a fundamental property of derivatives taught in any undergraduate curriculum. It states that if you have 3 functions *f*, *g* and *h* with *f* being a function of *g* and *g* being a function of *h* then the derivative of *f* with respect to *h* is equal to the product of the derivative of *f* with respect to *g* and the derivative of *g* with respect to *h.* This can be neatly written as:

$$f(g(h(x)))$$

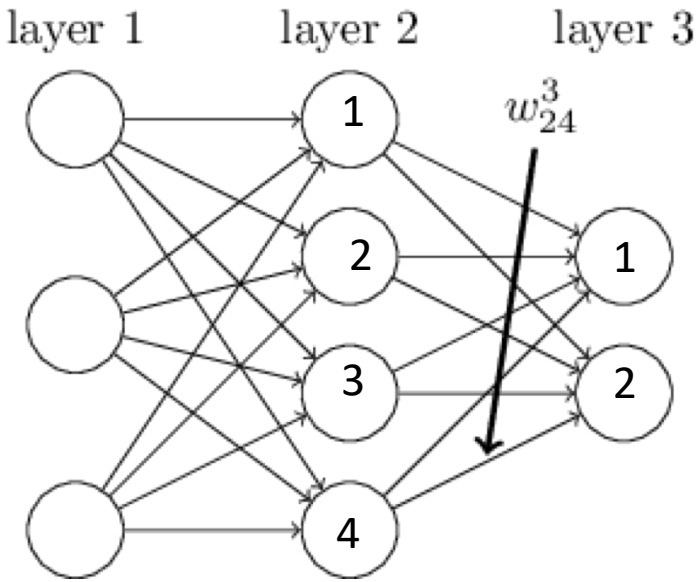
$$\frac{df}{dh} = \frac{df}{dg}\frac{dg}{dh}$$

Quick Review from Last Time


Weights
Bias
(constant)
$b \equiv x_0$
w_0
Weighted
Sum
Activation
Function
x_1
w_1
x_2
w_2
x_3
w_3
Inputs
$\sum_{i=0}^{n} w_i x_i$
$f(\cdot)$
Output (s)
a
x_n
w_n
$f(\cdot) \rightarrow \sigma(z) \rightarrow \frac{1}{1+e^{-z}}$

Convention for naming things

$(l-2)^{th}$ $(l-1)^{th}$ $(l)^{th}$



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

a^l_j

Activation of neuron j in layer l

w^l_{jk}

Weight

l == to (⟶) layer

j == to (⟶) neuron

k == from (⟵) neuron

With these notations, the activation a^l_j of the j^{th} neuron in the l^{th} layer is related to the activations in the $(l-1)^{th}$ layer by the equation

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{(l-1)}_k + b^l_j\right)$$

To rewrite this expression in a matrix form we define a *weight matrix* w^l for each layer, l. The entries of the weight matrix w^l are just the weights connecting to the l^{th} layer of neurons, that is, the entry in the j^{th} row and k^{th} column is w^l_{jk} . We can simplify our vector notation a bit by dropping the explicit bias term.

$$a^l = \sigma\left(w^l a^{(l-1)} + b^l\right) \rightarrow \boxed{a^l = \sigma\left(w^l a^{(l-1)}\right)}$$

Definition of Loss (e.g. for MSE)

$$L(w,b) \equiv \frac{1}{2n} \sum_x \| \, y(x) - a \|^2$$

$y(x)$	The ground truth label for each sample x
a	The activation of the last layer for sample x

Loss Definition (simpler)

$$L(w) \equiv \frac{1}{2n} \sum_{x} \| y - \hat{y} \|^2$$

y is the label for sample *x* and *y* (hat) is the prediction for sample *x*

we also collapse the bias into the weights for simplicity of presentation

Overall Loss is Mean Loss over Samples

$$L = \frac{1}{n} \sum_{x} L_x$$

Computing the loss over all the samples would be one “batch”. Since the number of samples can be very large, in practice We estimate the loss by selected a random subset of samples and computing the mean loss for that set and then adjust weights and then iterate. This approach is called the “mini-batch”. Mini-batches are usually from 1-100 or so in size.

Gradient of the Loss

The gradient of the loss is equal to the mean of the gradient over each sample loss

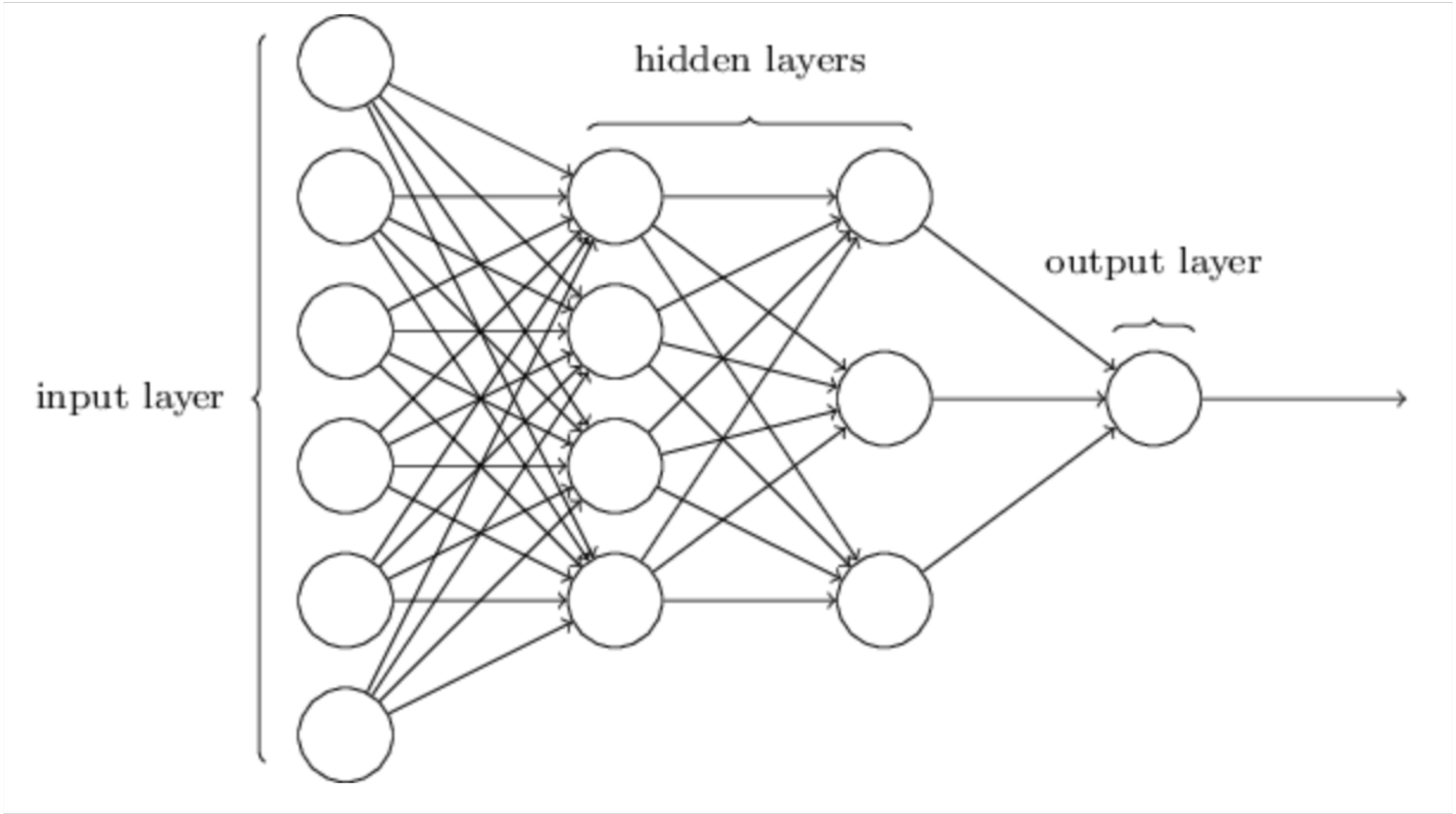
$$\nabla L = \frac{1}{n} \sum_{x} \nabla L_x$$

This fact will be useful when we are trying to estimate the overall loss gradient from subset of samples

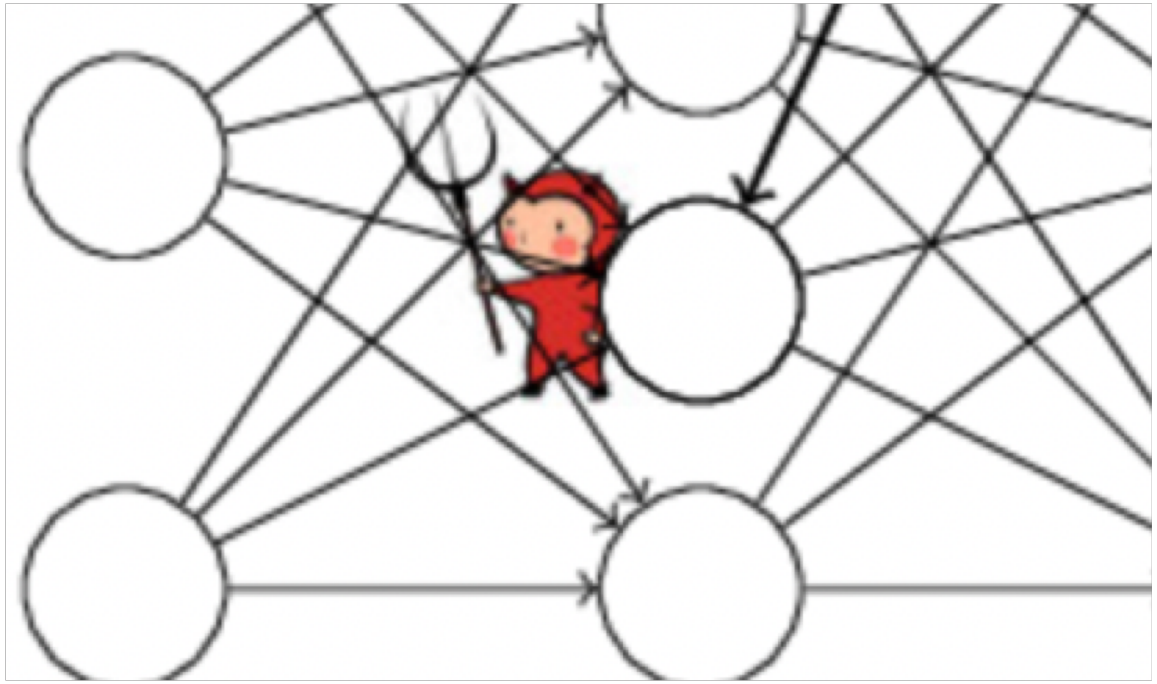
$$\nabla L \equiv \left(\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_n} \right)^{\mathsf{T}}$$

$$\sigma(z_j^l) \rightarrow \sigma(z_j^l + \Delta z_j^l)$$

Our goal is to work out how to nudge weights to improve our error



Gradient Descent



$$\sigma(z_j^l) \rightarrow \sigma(z_j^l + \Delta z_j^l)$$

$$\Delta w = -\eta \nabla L$$

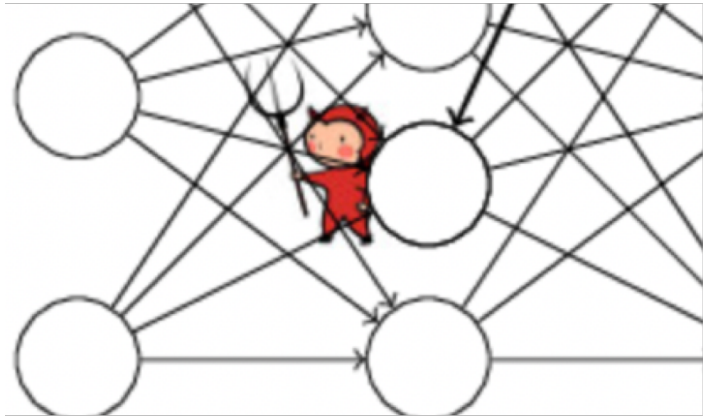
Small change to the weight is
learning rate times the gradient of Loss

The basic update operation

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial L_{x_j}}{\partial w_k}$$

The basic update operation

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial L}{\partial w_k}$$



Review for back propagation

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{(l-1)} + b_j^l\right)$$ Activation of neuron in *j* in layer *l*

$$a^l = \sigma\left(w^l a^{(l-1)} + b^l\right) \rightarrow a^l = \sigma\left(w^l a^{(l-1)}\right)$$ Vector form

$$z^l \equiv w^l a^{(l-1)}$$ Intermediate product before “squash” with activation (vector form)

$$a^l = \sigma(z^l)$$ z “squashed” with activation (vector form)

Assumptions about the loss function

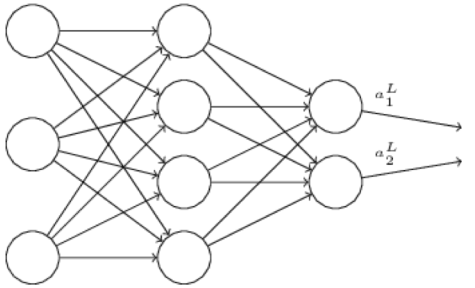
The goal of backpropagation is to compute the partial derivates

$$\partial L/\partial w \qquad \partial L_x/\partial w_x$$

Of the loss function L with respect to any weight w or bias b in the network

1. Loss function can be written as an average [batch $\Longrightarrow$ mini-batch]
2. It can be written only in terms of a function of the outputs (last layers) from the neural network [ground case for induction]

$$L = \frac{1}{2}\|y - a^{last}\|^2 = \frac{1}{2}\sum_j (y_j - a_j^{last})^2$$



$$L = L(a^l)$$

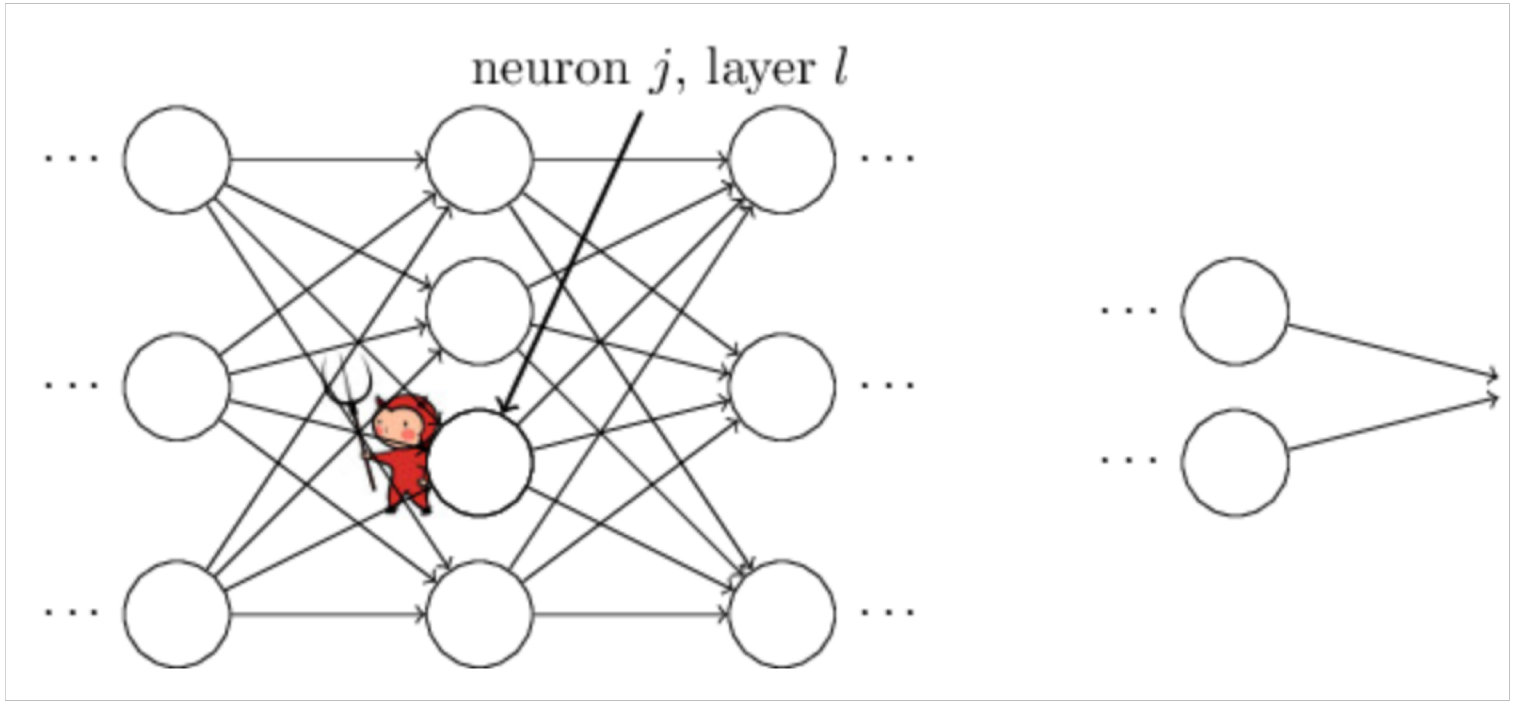
The Hadamard Product, s ⊙ t (aka element wise product)

If we have two vectors (or matrices) of the same size then we just multiple the corresponding elements

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

This will appear in our final equations

Equations Behind Backpropagation



$$\partial L / \partial w^l_{jk}$$

Partial per weight

$$L = L(a^l)$$

Only need Last layer Activation to Compute L

$$\delta^l_j \equiv error$$ Layer l and neuron j

$$\delta^l_j \equiv \frac{\partial L}{\partial z^l_j}$$

Error per Neuron per Layer on Z

Nudging W $\Longrightarrow$ Nudges Z $\Longrightarrow$ Nudges L

$$z^l \equiv w^l a^{(l-1)}$$

Nudging W will nudge Z

$$\sigma(z_j^l) \rightarrow \sigma(z_j^l + \Delta z_j^l)$$

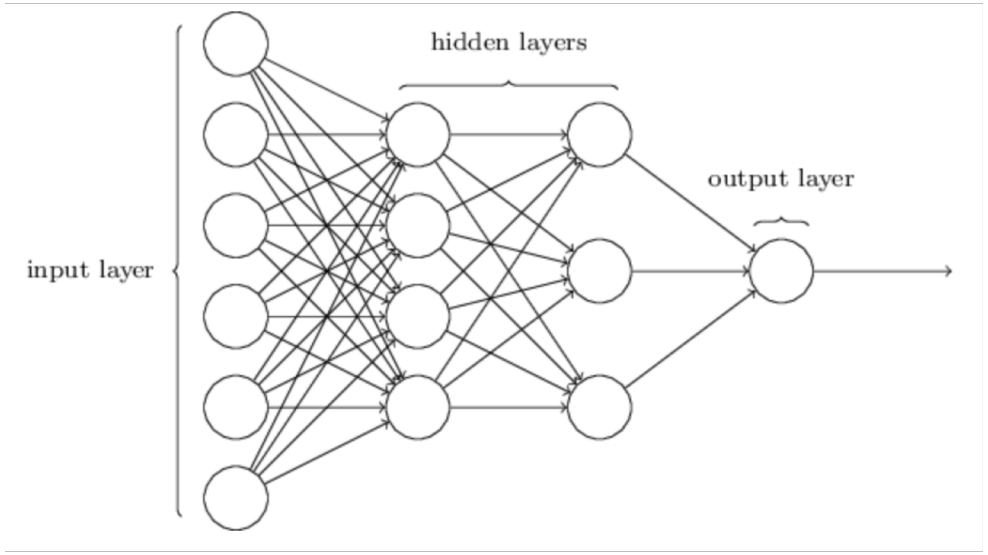
Nudging the Z by nudging the weights that sum into Z

$$L \rightarrow L + \frac{\partial L}{\partial z_j^l} \Delta z_j^l$$

Move the loss value by nudging weights that nudge z

We need to work backwards from the error estimated from L

- Basic idea
- Figure out how to compute the “nudges” in the last layer from L
- Then recursively apply the nudges backwards through the network
- Stop when at the input layer



An equation for the error in the output layer (bp1)

$$\delta_j^{last} = \frac{\partial L}{\partial a_j^{last}} \sigma'(z_j^{last})$$

Error in the last layer
For neuron j

$$\delta^{last} = \nabla_a L \odot \sigma'(z^{last})$$

$$\delta^{last} = (a^{last} - y) \odot \sigma'(z^{last})$$

Vector form
in terms of
previously
computed values

An equation for the error δ^l in terms of the error in the next layer $(l+1)$ (bp2)

$$\delta^l = ((w^{l+1})^\intercal \delta^{l+1}) \odot \sigma'(z^l)$$

Working backwards from layer $l+1$ to l

Note which quantities have already been computed on the forward pass

By combining (bp2) with (bp1) we can compute the error δ^l for any layer in the network

We start by using (bp1) to compute δ^{last}, then apply Equation (bp2) to compute $\delta^{(l-1)}$, then Equation (bp2) again to compute $\delta^{(l-2)}$, and so on, all the way back through the network

An equation for the rate of change of the loss with respect to any bias in the network (bp3)

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l \quad \longrightarrow \quad \frac{\partial L}{\partial b} = \delta$$

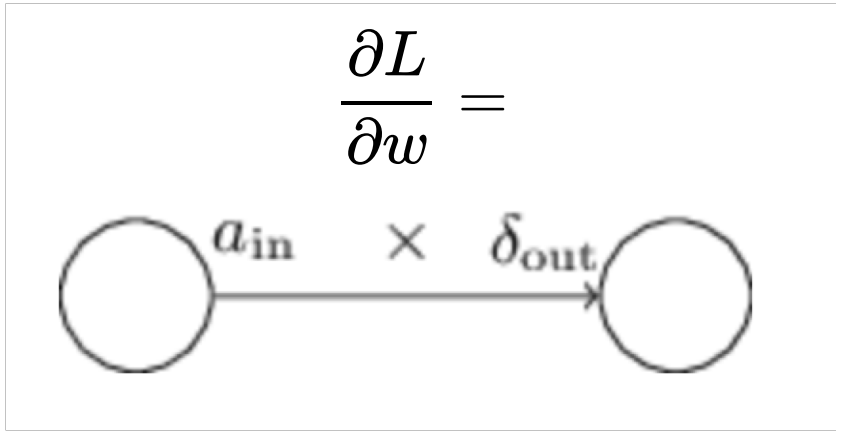
I've included this to be consistent with

http://neuralnetworksanddeeplearning.com/chap2.html

An equation for the rate of change of the loss with respect to any weight in the network (bp4)

$$\frac{\partial L}{\partial w^l_{jk}} = a^{(l-1)}_k \delta^l_j$$

one consequence of (bp4) is that weights output from low-activation neurons learn slowly.



$$\frac{\partial L}{\partial w} = a_{in} \delta_{out}$$

Summary: the equations of backpropagation

$$\delta^{last} = (a^{last} - y) \odot \sigma'(z^{last})$$

$$\delta^{l} = ((w^{l+1})^{\intercal}\delta^{l+1}) \odot \sigma'(z^{l})$$

$$\frac{\partial L}{\partial b} = \delta$$

$$\frac{\partial L}{\partial w^{l}_{jk}} = a^{(l-1)}_{k}\delta^{l}_{j}$$

For a good explaination of all of this read
http://neuralnetworksanddeeplearning.com/chap2.html