LING3804: Lecture Notes 6
Backpropagation in Multi-layer Neural Networks

We know single neurons only learn linear separators; they can't pick out arbitrary regions of input.

Multi-neuron states also can't – they just learn 'planar' separators in multiple dimensions.

We also know multi-***layer*** neural networks have ***no*** limitations – they can estimate any function.

But these have to be trained a bit differently...

Contents

6.1 Backpropagation brings error to interior layers of multi-layer networks

For single-neuron learning, we repeatedly adjust synaptic weights in a direction to minimize error.

- if the error is negative (the prediction is too low): increase the used weights
- if the error is positive (the prediction is too high): decrease the used weights

Then repeat until convergence.

But if we have multiple layers of neurons, how do we know which direction to move their weights?

- we can minimize the error of a single-layer network based on the correct outputs,
- but what should we use for the interior layer's correct output and its error?

Solution: propagate error back to interior layers by exploiting causal relationships between layers.

Think of these relationships as the effect on error of adjusting each weight an infinitesimal amount (like wiggling each gear of a mechanical clock to see what effect it has on the clock hands).

This is called **backpropagation**.

6.2 Jacobian matrices for error propagation

We already know these effects for synaptic weights, since the weights directly encode the effects!

But the threshold functions are a bit trickier...

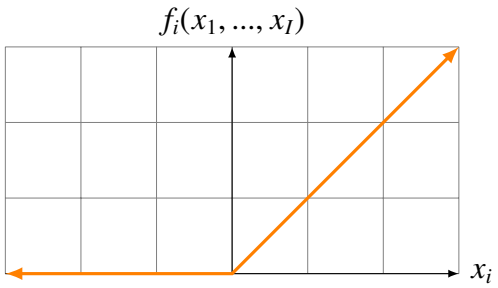
Threshold functions aren't 'linear' so we can't define a linear effect of each input *in general*
(a linear effect is one you can graph as a straight line, or a flat plane in multiple dimensions)
(so non-linear effects vary depending on the value of each input – i.e. the function has curves).
If a function is not linear, it can't be represented as a matrix, which just calculates sums of products.

But we're adjusting weights for *particular states*, so we don't have to calculate general functions.
We can calculate the effect on error of infinitesimal change in each input *from its current value*!
That actually *is* a linear function, so we can use that to define our Jacobian matrices.

Effects of infinitesimal input changes are **derivatives**. Effects for vectors of input are **gradients**.
A matrix of effects (weights) of each input on each output of a threshold function is a **Jacobian**.

For example, the rectified linear threshold function:

$$f_i(x_1, ..., x_I) = \begin{cases} x_i & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$



has slope 0 for negative input and 1 for positive input, so the Jacobian of f at the vector below is:

0.3
-1.8
-0.2
0.4
0.9

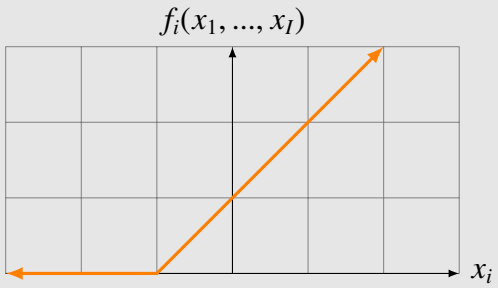
1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	1	0
0	0	0	0	1

It's zero off the diagonal because the neurons don't interact in the threshold function.

Practice 6.1:

For the following threshold function:

$$f_i(x_1, ..., x_I) = \begin{cases} x_i & \text{if } x_i > -1 \\ 0 & \text{otherwise} \end{cases}$$

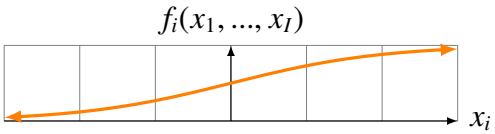


what would the Jacobian be at the below vector?

0.3
-1.8
-0.2
0.4
0.9

Functions like 'softmax' are normalized to be probability distributions, so they show interaction:

$$f_i(x_1, ..., x_I) = \frac{e^{x_i}}{\sum_k e^{x_k}}$$



The Jacobian of the softmax function at the below vector is the below matrix:

0.3
-1.8
-0.2
0.4
0.9

0.2	-0	-0	-0.1	-0.1
-0	0	-0	-0	-0
-0	-0	0.1	-0	-0.1
-0.1	-0	-0	0.2	-0.1
-0.1	-0	-0.1	-0.1	0.2

Note that increasing the value of any output neuron decreases the values of its competitors.

That's because a softmax is normalized (divided) to be a probability distribution so it sums to one.

This is like a 'zero-sum' game: if you increase your score, other players' scores decrease.

Now we can use these Jacobian matrices to propagate error to interior layers of a network.

6.3 Matrix chains

Ok, Jacobians let us propagate error one layer back, but what if we have more than two layers?

Recall from the lecture on associative memory how to cue a weight matrix with a state vector:

$$\begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} \overset{\text{def}}{=} \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Well, we can similarly 'cue' an error vector with the columns of a weight matrix:

$$\begin{bmatrix} e'_1 & e'_2 & e'_3 \end{bmatrix} \overset{\text{def}}{=} \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix}$$

This is again a matrix product, multiplying rows of the left operand by columns of the right.

For example:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 20 \end{bmatrix}$$

$$(1 \times 3) + (2 \times 5) = 3 + 10 = 13$$
$$(1 \times 4) + (2 \times 6) = 4 + 12 = 16$$

Practice 6.2:

Calculate the error through the below Jacobian:

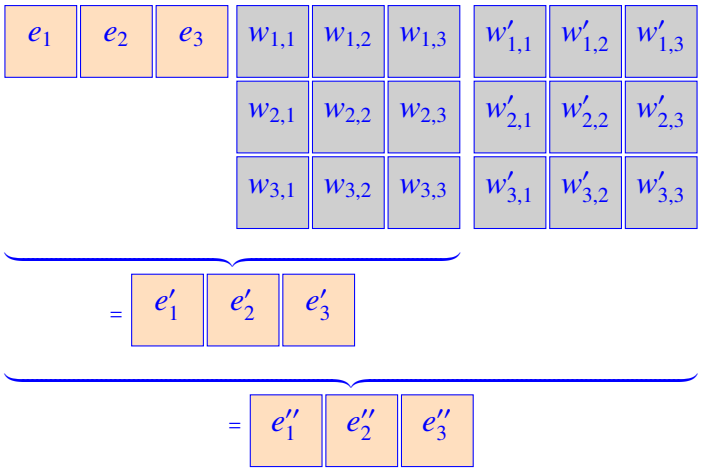
$$\begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 3 & -3 \\ 4 & -4 \end{bmatrix} = \begin{bmatrix} & \end{bmatrix}$$

Note that both of the above pairs of inputs and outputs have the same form.

This means outputs of matrix products can be used as inputs to other matrices.

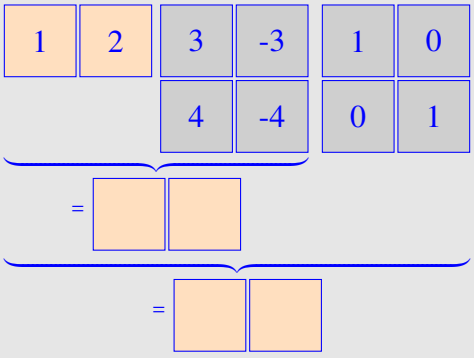
These are called **matrix chains**.

We use these chains to propagate activation forward or error backward through several Jacobians:



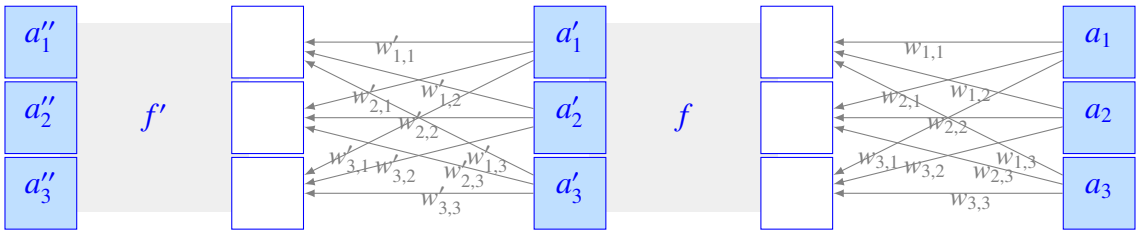
Practice 6.3:

Calculate the error through the below Jacobians:



6.4 Backpropagation in multi-layer neural networks

Suppose we are training a network with several layers of weights w and threshold functions f:



We want to update the weights to better predict each observed output and thus minimize the error.

We first calculate all the neuron activations as usual, based on their antecedent neurons.

Then we substitute Jacobians and propagate error (the initial error includes the top-layer Jacobian).

We do this for each weight matrix – for example, here we update the lowest layer weight matrix:

$$\underbrace{\begin{bmatrix} e''_1 & e''_2 & e''_3 \end{bmatrix} \begin{bmatrix} w'_{1,1} & w'_{1,2} & w'_{1,3} \\ w'_{2,1} & w'_{2,2} & w'_{2,3} \\ w'_{3,1} & w'_{3,2} & w'_{3,3} \end{bmatrix} \begin{bmatrix} j_{1,1} & j_{1,2} & j_{1,3} \\ j_{2,1} & j_{2,2} & j_{2,3} \\ j_{3,1} & j_{3,2} & j_{3,3} \end{bmatrix}}_{= \begin{bmatrix} e'_1 & e'_2 & e'_3 \end{bmatrix}} \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$= \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix}$$

Once we propagate the error to the desired weight matrix, we update the weights as before

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \overset{\text{def}}{=} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$$

(this is an inner product of the error and the antecedent layer's activation vector).

6.5 Problems with backpropagation

There are a couple of problems with backpropagation:

First, cognitive scientists and neuroscientists don't find backpropagating models very human-like.

That's because there is no analogue in biology for 'cueing' synaptic weights backward with errors.

But also, the error (or 'gradient') attenuates with each backpropagation.

Consider again the Jacobian of the softmax of this vector:

0.3
-1.8
-0.2
0.4
0.9

0.2	-0	-0	-0.1	-0.1
-0	0	-0	-0	-0
-0	-0	0.1	-0	-0.1
-0.1	-0	-0	0.2	-0.1
-0.1	-0	-0.1	-0.1	0.2

Note that the weights are all very small.

This attenuation makes the error signal tend to fade away over long chains of Jacobians.

Rectified linear threshold functions don't attenuate, but they do become 'sparse': mostly zero.

An opposite problem happens with large initial weights, which amplifies the error at each layer.

This is called a problem of **vanishing and exploding gradients** – one motivation for transformers.

6.6 Extra: derivation of Jacobian for softmax

The derivative (slope) for each neuron i given each input j to a softmax at state $x_1, ..., x_J$ is:

$$\begin{aligned}
\frac{\partial}{\partial x_j}\frac{e^{x_i}}{\sum_k e^{x_k}} &= e^{x_i}\frac{\partial}{\partial x_j}\frac{1}{\sum_k e^{x_k}} + \frac{1}{\sum_k e^{x_k}}\frac{\partial}{\partial x_j}e^{x_i} && \text{derivative of product} \\
&= e^{x_i}(-1)\frac{1}{(\sum_k e^{x_k})^2}\frac{\partial}{\partial x_j}\sum_k e^{x_k} + \frac{1}{\sum_k e^{x_k}}\frac{\partial}{\partial x_j}e^{x_i} && \text{derivative of power} \\
&= e^{x_i}(-1)\frac{1}{(\sum_k e^{x_k})^2}\sum_k \frac{\partial}{\partial x_j}e^{x_k} + \frac{1}{\sum_k e^{x_k}}\frac{\partial}{\partial x_j}e^{x_i} && \text{derivative of sum} \\
&= e^{x_i}(-1)\frac{1}{(\sum_k e^{x_k})^2}e^{x_j} + \frac{1}{\sum_k e^{x_k}}e^{x_j}\begin{cases}1 & \text{if } i = j \\ 0 & \text{if } i \neq j\end{cases} && \text{derivative of exponential} \\
&= \begin{cases}\left(1 - \frac{e^{x_i}}{\sum_k e^{x_k}}\right)\frac{e^{x_i}}{\sum_k e^{x_k}} & \text{if } i = j \\ \left(0 - \frac{e^{x_i}}{\sum_k e^{x_k}}\right)\frac{e^{x_j}}{\sum_k e^{x_k}} & \text{if } i \neq j\end{cases} && \text{distributive axiom}
\end{aligned}$$