\[ \frac{\partial C(w)}{\partial w_*} = ? \]

some weight (or bias), somewhere in the net
It’s time to use the chain rule!

\[
\frac{\partial C(w)}{\partial w^*} = ?
\]

some weight (or bias), somewhere in the net
Small network: Calculate derivative of cost function “by hand”
Small network: Calculate derivative of cost function “by hand”

\[
z = w_1 y_1 + w_2 y_2 + b
\]

\[
C(w) = \frac{1}{2} \left\langle (f(z) - F(y_1, y_2))^2 \right\rangle
\]

- INPUT: \( y_1, y_2 \)
- OUTPUT: \( f(z) \)
Small network: Calculate derivative of cost function “by hand”

\[
\frac{\partial C}{\partial w_1} = \left\langle (f(z) - F)f'(z) \frac{\partial z}{\partial w_1} \right\rangle
\]

\[
\frac{\partial z}{\partial w_1} = y_1
\]

\[
z = w_1 y_1 + w_2 y_2 + b
\]

\[
C(w) = \frac{1}{2} \left\langle (f(z) - F(y_1, y_2))^2 \right\rangle
\]
Now for the full network!

Need to keep track of indices carefully:

- $y_j^{(n)}$: Value of neuron j in layer n
- $z_j^{(n)}$: Input value for “$y=f(z)$”
- $w_{jk}^{n,n-1}$: Weight (neuron k in layer n-1 feeding into neuron j in layer n)
Backpropagation

We have:

\[ C(w) = \langle C(w, y^{in}) \rangle \]

cost value for one particular input

We get:

\[
\frac{\partial C(w, y^{in})}{\partial w_*} = \sum_j (y_j^{(n)} - F_j(y^{in})) \frac{\partial y_j^{(n)}}{\partial w_*}
\]

\[
= \sum_j (y_j^{(n)} - F_j(y^{in})) f'(z_j^{(n)}) \frac{\partial z_j^{(n)}}{\partial w_*}
\]

(we used:)

\[ y_j^{(n)} = f(z_j^{(n)}) \]

some weight (or bias), somewhere in the net
Backpropagation

Apply chain rule repeatedly

We want: Change of neuron \( j \) in layer \( n \) due to change of some arbitrary weight \( \mathbf{w}_* \):

\[
\frac{\partial z_j^{(n)}}{\partial \mathbf{w}_*} = \sum_k \frac{\partial z_j^{(n)}}{\partial y_k^{(n-1)}} \frac{\partial y_k^{(n-1)}}{\partial \mathbf{w}_*} = \sum_k \mathbf{w}_{jk}^{n,n-1} f'(z_k^{(n-1)}) \frac{\partial z_k^{(n-1)}}{\partial \mathbf{w}_*}
\]

And now: the same again (recursion)
Important insight: Each pair of layers \([n,n-1]\) contributes multiplication with the following matrix:

\[
M_{jk}^{(n,n-1)} = w_{jk}^{(n,n-1)} f'(z_k^{(n-1)})
\]
Backpropagation

Repeated matrix multiplication, going down the net:

\[
\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_{k,l,\ldots,u,v} M_{jk}^{n,n-1} M_{kl}^{n-1,n-2} \ldots M_{uv}^{\tilde{n}+1,\tilde{n}} \frac{\partial z_v^{(\tilde{n})}}{\partial w_*}
\]
Backpropagation

What happens when we finally encounter the weight with respect to which we wanted to calculate the derivative of the cost function?

If \( w_* \) was really a weight:

\[
\frac{\partial z^{(\tilde{n})}_j}{\partial w_{\tilde{n},\tilde{n}-1}^{jk}} = y_{k}^{(\tilde{n}-1)}
\]

...if it was a bias:

\[
\frac{\partial z^{(\tilde{n})}_j}{\partial b^{\tilde{n}}_j} = 1
\]
Backpropagation

We have:

\[ C(w) = \langle C(w, y^{\text{in}}) \rangle \]

cost value for one particular input

In total, we get:

\[
\frac{\partial C(w, y^{\text{in}})}{\partial w_*} = \sum_j (y_j^{(n)} - F_j(y^{\text{in}})) \frac{\partial y_j^{(n)}}{\partial w_*}
\]

\[
= \sum_j (y_j^{(n)} - F_j(y^{\text{in}})) f'(z_j^{(n)}) \frac{\partial z_j^{(n)}}{\partial w_*}
\]

How to evaluate this: construct vector for output layer n, and then multiply with matrices from the right (as shown above)
Backpropagation

1. Initialize vector from output layer:
\[ \Delta_j = (y^n_j - F_j(y^{in})) f'(z^n_j) \]

2. For each layer: store outcomes (cost derivatives) for all weights and biases \( w_* \) in that layer
\[ \frac{\partial C(w, y^{in})}{\partial w_*} = \Delta_j \frac{\partial z^{(n)}_j}{\partial w_*} \]
(\( j \) is the index where this particular weight appears)

3. Multiply vector by matrix
\[ \Delta_{k}^{\text{new}} = \sum_j \Delta_j M^n_{j,k}^{n,n-1} \]
(see above for \( M \))

Summary

Multiply vector by matrix
\[ \partial C(w, y^{in}) \]
\[ \frac{\partial C(w, y^{in})}{\partial w_*} = \Delta_j \frac{\partial z^{(n)}_j}{\partial w_*} \]
(\( j \) is the index where this particular weight appears)
Backpropagation

Very efficient: One single backpropagation pass through the network yields ALL the derivatives of C with respect to all the weights and biases!

No more effort than forward propagation!

Huge ("million-fold") advantage over naive approach of calculating numerically derivatives for all weights individually!
Backpropagation

Physical intuitive picture:

“force” tries to pull into direction of correct outcome

adjusts all weights (& biases) in layers below
Backpropagation

In each layer:
\[
\frac{\partial C(w, y^{in})}{\partial w_*} = \Delta_j \frac{\partial z_j^{(n)}}{\partial w_*}
\]

<table>
<thead>
<tr>
<th>Weight:</th>
<th>Bias:</th>
</tr>
</thead>
</table>
| \[
\frac{\partial z_j^{(n)}}{\partial w_{n,n-1}^{jk}} = y_k^{(n-1)}
\] | \[
\frac{\partial z_j^{(n)}}{\partial b_j^n} = 1
\] |

--- Averaging over samples: ---

<table>
<thead>
<tr>
<th>Weight:</th>
<th>Bias:</th>
</tr>
</thead>
</table>
| \[
\frac{\partial C(w)}{\partial w_{n,n-1}^{jk}} = \left\langle \Delta_j y_k^{(n-1)} \right\rangle
\] | \[
\frac{\partial C(w)}{\partial b_j^n} = \left\langle \Delta_j \right\rangle
\] |
We are doing batch processing of many samples!

### Implementation

<table>
<thead>
<tr>
<th>Variable</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>y[layer]</td>
<td>batchsize x neurons[layer]</td>
</tr>
<tr>
<td>Delta</td>
<td>batchsize x neurons[layer]</td>
</tr>
<tr>
<td>Weights[layer]</td>
<td>neurons[lower layer] x neurons[layer]</td>
</tr>
<tr>
<td>Biases[layer]</td>
<td>neurons[layer]</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\frac{\partial C(w)}{\partial w_{jk}^{n,n-1}} &= \left\langle \Delta_j y_k^{(n-1)} \right\rangle \\
\frac{\partial C(w)}{\partial b_j^n} &= \left\langle \Delta_j \right\rangle
\end{align*}
\]

*averaging: sum over batch index!*

\[
d\text{Weights[layer]} = \text{dot}(\text{transpose}(y[lower\ layer]),\text{Delta})/\text{batchsize}
\]

\[
d\text{Biases[layer]} = \text{Delta}.\text{sum}(0)/\text{batchsize}
\]

(summation over index 0=batch index)
We are doing batch processing of many samples!

<table>
<thead>
<tr>
<th>Variable</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y[\text{layer}] )</td>
<td>( \text{batchsize} \times \text{neurons}[\text{layer}] )</td>
</tr>
<tr>
<td>( \text{Delta} )</td>
<td>( \text{batchsize} \times \text{neurons}[\text{layer}] )</td>
</tr>
<tr>
<td>( \text{Weights}[\text{layer}] )</td>
<td>( \text{neurons}[\text{lower layer}] \times \text{neurons}[\text{layer}] )</td>
</tr>
<tr>
<td>( \text{Biases}[\text{layer}] )</td>
<td>( \text{neurons}[\text{layer}] )</td>
</tr>
</tbody>
</table>

\[
\Delta_k^{\text{new}} = \sum_j \Delta_j M_{jk}^{n,n-1}
\]

with:
\[
M_{jk}^{(n,n-1)} = w_{jk}^{(n,n-1)} f'(z_k^{(n-1)})
\]

Take step from ‘\text{layer}’ down to ‘\text{lower layer}’:

\[
\text{Delta} = \text{dot}(\text{Delta}, \text{transpose}(\text{Weights})) * \text{df}_\text{layer}[\text{lower layer}]
\]

\( \text{batchsize} \times \text{neurons}[\text{lower layer}] \)

\( f'(z) \) in lower layer

(first dimension will be expanded)
here: NumLayers=3 (count all, except input)

Implementation

_weights[0]
_weights[1]
_weights[2]

Biases[0]
Biases[1]
Biases[2]

_y_layer[0]
_y_layer[1]
_y_layer[2]
_y_layer[3]

df_layer[0]
df_layer[1]
df_layer[2]

(should be a 2x3 matrix)
Now: The full algorithm, with forward propagation and backpropagation!

(will store neuron values and f’(z) values during forward propagation, to be used later during backpropagation)
def net_f_df(z):  # calculate f(z) and f'(z)
    val = 1 / (1 + exp(-z))
    return (val, exp(-z) * (val ** 2))  # return both f and f'

def forward_step(y, w, b):  # calculate values in next layer
    z = dot(y, w) + b  # w=weights, b=bias vector for next layer
    return (net_f_df(z))  # apply nonlinearity

def apply_net(y_in):  # one forward pass through the network
    global Weights, Biases, NumLayers
    global y_layer, df_layer  # store y-values and df/dz
    y = y_in  # start with input values
    y_layer[0] = y
    for j in range(NumLayers):  # loop through all layers
        # j=0 corresponds to the first layer above input
        y, df = forward_step(y, Weights[j], Biases[j])
        df_layer[j] = df  # store f'(z)
        y_layer[j + 1] = y  # store f(z)
    return (y)
def backward_step(delta, w, df):
    # delta at layer N, of batchsize x layersize(N))
    # w [layersize(N-1) x layersize(N) matrix]
    # df = df/dz at layer N-1, of batchsize x layersize(N-1)
    return (dot(delta, transpose(w)) * df)

def backprop(y_target):
    # one backward pass
    # the result will be the 'dw_layer' matrices with
    # the derivatives of the cost function with respect to
    # the corresponding weight (similar for biases)
    global y_layer, df_layer, Weights, Biases, NumLayers
    global dw_layer, db_layer
    # dCost/dw and dCost/db
    #(w,b=weights,biases)
    global batchsize

delta = (y_layer[-1] - y_target) * df_layer[-1]
dw_layer[-1] = dot(transpose(y_layer[-2]), delta) / batchsize
db_layer[-1] = delta.sum(0) / batchsize
for j in range(NumLayers - 1):
    delta = backward_step(delta, Weights[-1-j], ...
    ...df_layer[-2-j])
dw_layer[-2-j] = dot(transpose(y_layer[-3-j]), delta)...
    .../batchsize
db_layer[-2-j] = delta.sum(0) / batchsize
Homework

Try out the effects of:
- Value of the stepsize $\eta$
- Layout of the network (number of neurons and number of layers)
- Initialization of the weights

How do these things affect the speed of learning and the final quality (final value of the cost function)?

Try them out also for other test functions (other than in the example)
Change the output layer $f(z)$ to a LINEAR function, i.e. $f(z)=z$! Implement the required changes to the backpropagation code.

Apply this to the example case (learning a 2D function; see code on the website).