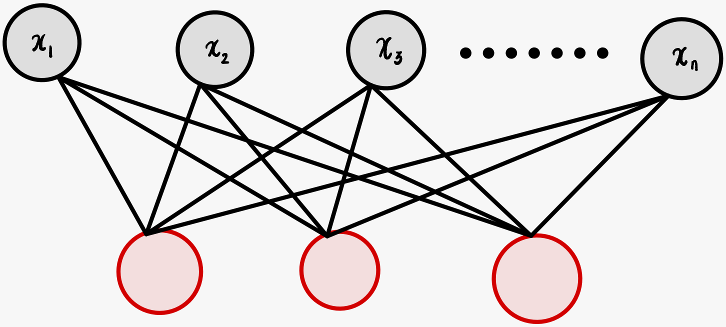
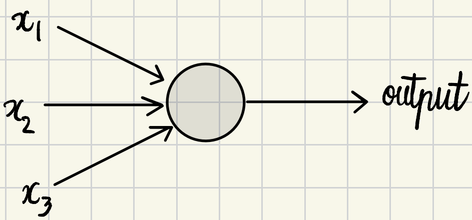


Neural networks

" its just matrix multiplication "

Perceptrons (a type of artificial neuron)

– it takes several binary inputs, $x_1, x_2, \dots, x_n$ & produces a single binary output



- **Weights**, $w_1, w_2, \dots w_n$ are real numbers that express the importance of the respective inputs to the outputs.

- neuron's output (0 or 1) = if the weighted sum $\left(\sum_j w_j x_j\right)$ is less than or greater than some threshold value.

  a real number that is a parameter of the neuron

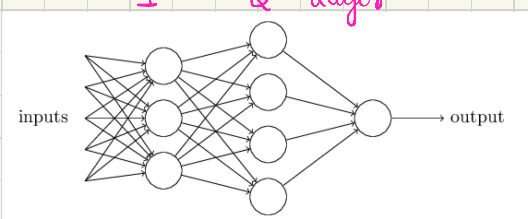
$$\text{Output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \le \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j \ge \text{threshold} \end{cases}$$

$$\boxed{w \cdot x \equiv \sum_j w_j x_j}$$

– in a multi-layer network of perceptrons, the output from the first layer is used as input to several other perceptrons in the second layer.

"collection of neurons"

1st 2nd layer



- the perceptrons in the second layer make more complex & abstract level decisions than perceptrons in the first layer.

- perceptron's bias, $b = -$ threshold

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

- Bias can be thought of as how easy it is to get a perceptron to output 1. Really big bias means its extremely easy for the perceptron to output 1.

Sigmoid neurons

- small change in their weights & biases causes only a small change in their output.

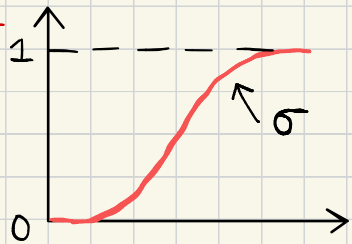
- inputs can take any values b/w 0 & 1.

- output is $\sigma(w \cdot x + b)$ where σ is called the sigmoid / logistic func.

$$\boxed{\sigma(z) = \frac{1}{1 + e^{-z}}}$$

- The output of a sigmoid neuron w/ inputs $x_1, x_2, \ldots$ weights $w_1, w_2, \ldots$ and bias b is $\boxed{1 / \left(1 + \exp\left(-\sum_j w_j x_j - b\right)\right)}$

- if z is a large positive no. $\Rightarrow e^{-z} \approx 0 \Rightarrow \sigma(z) \approx 1$
- if z is very negative $\Rightarrow e^{-z} \to \infty \Rightarrow \sigma(z) \to 0$

- 

The smoothness of σ means that small changes Δw_j in the weights & Δb in bias will produce a small change Δ output in the output from neuron.

- $$\boxed{\Delta \text{output} \approx \sum \frac{\partial \text{ output}}{\partial w_j} \Delta w_j + \frac{\partial \text{ output}}{\partial b} \Delta b}$$

it means that Δ output is a linear func. of the changes Δw_j & Δb in the weights & biases

The architecture of neural networks

- "hidden layer" – not an input or an output"
- feedforward – output from one layer is used as input to the next layer. There are no loops in the network, information is always fed forward, never fed back.

- activation func. – its just a func. used to calculate the output of neuron (node)

Gradient Descent

- for MNIST, each training input, x is a 784 dimensional vector (28 x 28 pixels) & the "desired" output y is a 10 dimensional vector.

cost func / objective func / loss :

output of the neural network

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

number of training inputs

mean - squared error (MSE)

- $\| v \|$ means length of vector $\vec{v}$
- $C(w,b)$ is always non-negative & $C(w,b) \approx 0$ when $y(x)$ is approximately equal to the output, a for all inputs x.
- The aim of the training algorithm is to minimize the $C(w,b)$ as a func of weights & biases, i.e. find a set of weights & biases that makes cost as small as possible which is done using Gradient Descent

- Consider C as a func. of two variables, v_1 & v_2 as a kind of a valley & imagine a ball rolling down the slope of the valley. Suppose we move the ball a small amount Δv_1 & Δv_2 in the v_1 & v_2 dirⁿ

$$\boxed{\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2}$$

we know, $\frac{dy}{dx} = \frac{\Delta y}{\Delta x} \Rightarrow$

$\Delta y = \Delta x \frac{dy}{dx}$

- we have to find a way of choosing Δv_1 & Δv_2 so as ΔC is negative, i.e. choose them so the ball is rolling down into the valley.

- $\Delta v = \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \end{bmatrix}$ gradient of C = $\nabla C = \begin{bmatrix} \partial C / \partial v_1 \\ \partial C / \partial v_2 \end{bmatrix}$

- "Gradient" of a function is a vector that points in the dir^n of greatest rate of increase of the func. $\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$

- $\Delta C \approx \nabla C \cdot \Delta v$

- $\Delta v = -\eta \nabla C$ where η is a small, positive parameter known as 'learning rate'

- $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \overbrace{\|\nabla C\|^2}^{\geq 0} \Rightarrow \Delta C \leq 0 \Rightarrow C$ will always decrease if $\Delta v = -\eta \nabla C$

- update rule $\Rightarrow v \longrightarrow v' = v - \eta C \Rightarrow$ if we keep doing this, we'll keep decreasing C until we reach a global minimum.

current position

for neural networks, the idea is to use gradient descent to find the weights, w_k & biases, b_l which minimizes the $C(w, b)$.

- update rule $\Rightarrow w_k \xrightarrow{\text{current} \to \text{updated}} w_k' = w_k - \eta \nabla C = w_k - \eta \frac{\partial C}{\partial w_k}$

$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$

- for individual training examples, $C_x = \frac{\| y(x) - a \|^2}{2} \Rightarrow C = \frac{1}{n} \sum_x C_x$ (for n training examples). To compute ∇C we need to compute gradients ∇C_x separately for each training input x & then average them which is computationally long for large no. of training inputs.

- Stochastic Gradient Descent (SGD) can be used to speed up learning. The idea is to estimate gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs.

- algorithm :- randomly pick a small no. m of randomly chosen training inputs, $x_1, x_2, \ldots, x_m$ & is referred as mini-batch.

- for large enough sample size m, average value of $\nabla C_{x_j} \approx$ average over all ∇C_x

- $\frac{\sum_{j=1}^{m} \nabla C_{x_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \Rightarrow \boxed{\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{x_j}}$

- SGD works by picking a randomly chosen mini-batch of training inputs & training w/ those,

$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k}$ $\qquad$ $b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_k}$

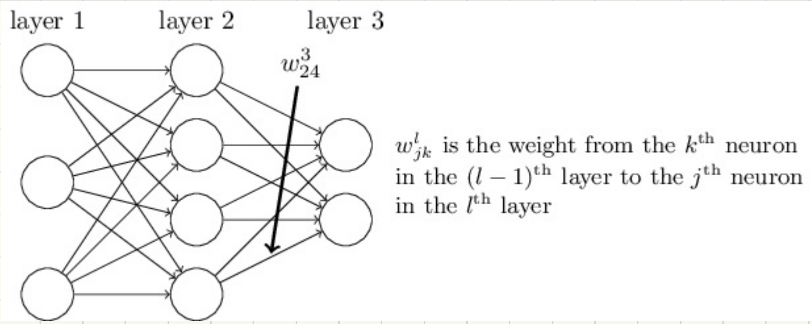
- then we pick out another randomly chosen mini-batch & train w/ those until we've exhausted all training inputs which is called an "epoch" of training.

Backpropagation

- it is used to compute the gradient of the cost function.

• matrix based algorithm to compute output from a neural net

- let w_{jk}^{l} denote the weight for the connection from the k^{th} neuron in $(l-1)^{th}$ layer to the j^{th} neuron in l^{th} layer.



- b_j^l is the bias of the j^{th} neuron in the l^{th} layer & a_j^l is the activation of the j^{th} neuron in the l^{th} layer.

- the activation a_j^l of the j^{th} neuron in l^{th} layer is related to the activations in the $(l-1)^{th}$ layer as :-

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

where the sum is over all neurons k in the $(l-1)^{th}$ layer.

- in matrix form we define a weight matrix w^l for each layer l & the entries of w^l are just the weights connecting to the l^{th} layer of neurons & the entry in j^{th} row & k^{th} column is w_{jk}^l. for each layer l we define a bias vector, b^l & the components of bias vector are just b_j^l, one component for each neuron in the l^{th} layer. And finally, an activation vector a^l whose components are a_j^l.

- applying a func. such as σ to every element in a vector v is called vectorization denoted as $\sigma(v)$ & $\sigma(v)_j = \sigma(v_j)$

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

- $z^l = w^l a^{l-1} + b^l$ & z^l is called the weighted input to the neurons in layer l.

- $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$ where z_j^l is the weighted input to the activation func. for neuron j in layer l.

- The goal of backpropagation is to compute the partial derivatives $\partial C / \partial w$ & $\partial C / \partial b$ of the cost function C w/ respect to any weight w or bias b in the network.
- quadratic cost func. $\Rightarrow$ $\boxed{C = \frac{1}{2n} \sum_x \| y(x) - a^L(x) \|^2}$

where $y(x)$ is desired output & $a^L = a^L(x)$ is the vector of activations output from the network when x is input.

- cost func^n C is a function of the outputs from neural network $\Rightarrow$ $C = C(a^L) = \frac{1}{2} \| y(x) - a^L \|^2$. C is not a function of $y(x)$ as for a fixed training input, x, the output y is also fixed.

The Hadamard Product (Schur)

- suppose s & t are two vectors of same dimension than $s \odot t$ denotes elementwise product of two vectors & $(s \odot t)_j = s_j t_j$

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

- let δ_j^l be the error in the j^{th} neuron in the l^{th} layer.
- consider a little change Δz_j^l to the neuron's weighted input

so that instead of outputting $\sigma(z_j)$, the neuron outputs $\sigma(z_j^l + \Delta z_j^l)$

This change propagates through later layers in the network, causing the overall cost to change by an amount $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$.

- if $\partial C / \partial z_j^l$ has a large value then Δz_j^l can lower the cost by having opposite sign to $\partial C / \partial z_j^l$. But if $\partial C / \partial z_j^l$ is already close to zero then Δz_j^l can't improve the cost much. Hence, $\partial C / \partial z_j^l$ is a measure of error in the neuron.

$$\boxed{\delta_j^l = \frac{\partial C}{\partial z_j^l}}$$

<u>eq^n for error in the output layer</u> (δ^L)

The components δ^L are given by,

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

- $\partial C / \partial a_j^L$ measures how fast the cost is changing as a func of the j^{th} output activation. If C doesn't depend on a particular output neuron j then δ_j^L will be small.
- $\sigma'(z_j^L)$ measures how fast the activation func σ is changing at z_j^L

- $\dfrac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$

- in matrix form, $\boxed{\delta^L = \nabla_a C \odot \sigma'(z^L)}$ ——— (i)

$$\Rightarrow \delta^L = (a^L - y) \odot \sigma'(z^L)$$

<u>error δ^l in terms of δ^{l+1}</u>

$$\boxed{\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)} \quad \text{——— (ii)}$$

where $(w^{l+1})^T$ is the transpose of weight matrix w^{l+1} for the $(l+1)^{th}$ layer.

- Suppose we know the error δ^{l+1} at the $(l+1)^{th}$ layer. Applying the transpose weight matrix, $(w^{l+1})^T$ intuitively means moving the error "backward" through the network, giving us measure of the error at the output of the l^{th} layer. Then $\odot \sigma'(z^l)$ moves the error backward through the activation func in layer l, giving δ^l in the weighted input to layer l.

- by using eqⁿ (i) we compute δ^L then applying (ii) to compute δ^{L-1} and so on all the way back through the network.

eqⁿ for rate of change of cost w.r.t. any bias

$$\boxed{\frac{\partial C}{\partial b_j^l} = \delta_j^l} \quad \text{———— (iii)}$$

- error δ_j^l is exactly equal to the rate of change $\partial C / \partial b_j^l$

$$\boxed{\frac{\partial C}{\partial b} = \delta}$$

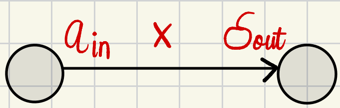
eqⁿ for rate of change of cost w.r.t. any weight

$$\boxed{\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l} \quad \text{———— (iv)}$$

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

where a_{in} is the activation of the neuron input to weight w & δ_{out} is the error of the neuron output from the weight w.

$\partial C / \partial w =$



- when $a_{in} \approx 0$, the gradient $\partial C/\partial w$ will also be small meaning the weights will learn slowly. The weights output from low-activations neuron learn slowly.

- The σ func becomes very flat when $\sigma(z_j^L)$ is approx 0 or 1 & then $\sigma'(z_j^L) \approx 0$. So the weights in the final layer will learn slowly if the output neuron has either low or high activations & is said that output neuron has saturated since the weight has stopped learning.

Proof of the four eq^n

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

applying chain rule, $\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}$ where sum is over all neurons k in the output layer.

- the output activation a_k^L of the k^{th} neuron depends only on the weighted input z_j^L for the j^{th} neuron when $k = j$ & so $\partial a_k^L / \partial z_j^L$ vanishes when $k \neq j$.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}$$

$$a_j^L = \sigma(z_j^L)$$

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

$$\Rightarrow \quad \delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

$$\Rightarrow \quad z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

differentiating, $\dfrac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

$\Rightarrow$ now, $\dfrac{\partial C}{\partial b_j^l} = \dfrac{\partial C}{\partial z_j^l} \times \dfrac{\partial z_j^l}{\partial b_j^l} = \dfrac{\partial C}{\partial z_j^l} \cdot \dfrac{\partial \sum_j w_{kj}^l a_j^{l-1} + b_k^l}{\partial b_j^l}$

$$= \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial b_k^l}{\partial b_j^l}$$

$$= \frac{\partial C}{\partial z_j^l} = \delta_j^l$$

$\Rightarrow \dfrac{\partial C}{\partial w_{jk}^l} = \dfrac{\partial C}{\partial z_j^l} \cdot \dfrac{\partial z_j^l}{\partial w_{jk}^l} = \dfrac{\partial C}{\partial z_j^l} \times \dfrac{\partial \sum_j w_{kj}^l a_j^{l-1} + b_k^l}{\partial w_{jk}^l}$

$$= \frac{\partial C}{\partial z_j^l} a_j^{l-1}$$

$$= a_j^{l-1} \delta_j^l$$

The backpropagation algorithm

1. **input x** – set the corresponding activation a^l for the input layer.

2. **feedforward** – for each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$

3. **output error δ^L** – compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$

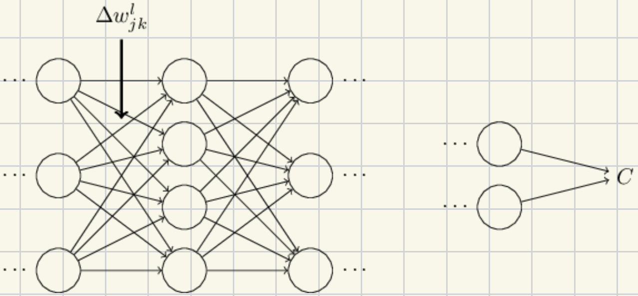
4. **Backpropagate the error** – for each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

5. **Output** – the gradient of the cost func is given by

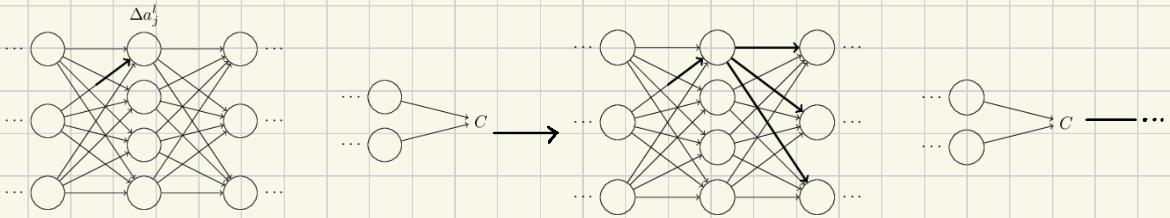
$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{and} \quad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

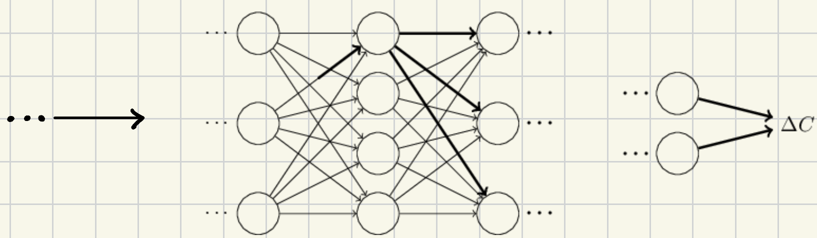
The big picture

– imagine a small change Δw^l_{jk} to some weight in the net, w^l_{jk},



- that change in weight will cause a change in the output activation from the corresponding neuron which in turn will cause a change in all the activations in the next layer. Those changes will cause changes all the way through to final layer & then in cost function.





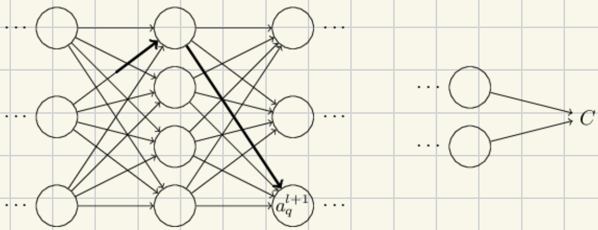
- The change ΔC in the cost is related to the change Δw_{jk}^l in the weight by,

$$\Delta C = \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l$$

- The change Δw_{jk}^l causes a small change Δa_j^l in the activation of the j^{th} neuron in the l^{th} layer which is given by,

$$\Delta a_j^l = \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

- The change in Δa_j^l will cause changes in all the activations in the $(l+1)^{th}$ layer. a single one of these activations say, a_q^{l+1} will cause the change,



$$\Delta a_q^{l+1} = \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l$$

$$\Delta a_q^{l+1} = \frac{\partial a_q^{l+1}}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

- a path all the way through the network from w_{jk}^l to C, with each change in activation causing a change in the next activation & finally a change in C at the output. If path goes through activations $a_j^l, a_q^{l+1}, \ldots, a_n^{L-1}, a_m^L$ then,

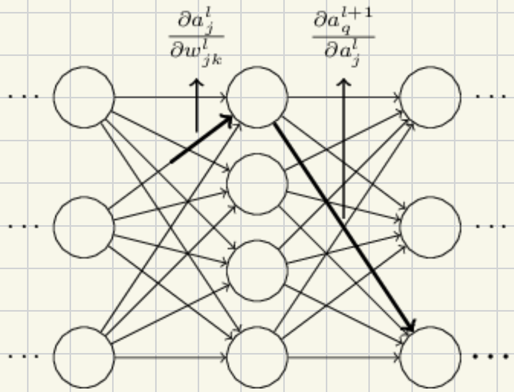
$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

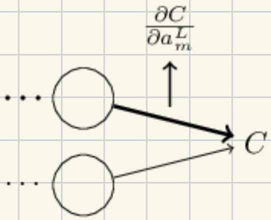
- This represents the change in C due to changes in activations along a particular path through network.
- Every edge between two neurons in the network is associated w/ a rate factor which is just the partial derivative of

one neuron's activation w.r.t the other neuron's activation.

- The edge from the first weight to the first neuron has a rate factor $\partial a_j^l / \partial w_{jk}^l$.

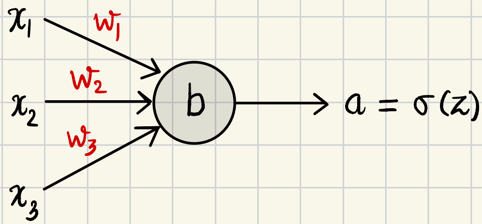
- The rate factor of a path is just the product of rate factors along the path. And total rate of change $\partial C / \partial w_{jk}^l$ is just the sum of rate factors of all paths from the initial weight to the final cost.





Cross - Entropy

- suppose we've a neuron w/ several input variables, $x_1, x_2, \ldots$ corresponding weights, $w_1, w_2, \ldots$ and a bias, b



- The output from the neuron is $a = \sigma(z)$ where $z = \sum_j w_j x_j + b$ is the weighted sum of the inputs.
- The "cross - entropy" cost function is defined as,

$$\boxed{C = -\frac{1}{n} \sum_x \left[y \ln a + (1-y) \ln (1-a) \right]}$$

- Cross - entropy cost function is non - negative, $C > 0$ as all the individual terms in the sum are negative since log of the numbers will range from 0 to 1.
- if the neuron's actual output is close to the desired output for all training inputs, x, then the cross - entropy ≈ 0. for example, when $y = 0$ & $a \approx 0$ then $y \ln a \approx 0$ & $\ln(1-a) \approx 0$.

- now, $\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)\,x_j$$

$$= \frac{1}{n}\sum_x \frac{\sigma'(z)\,x_j}{\sigma(z)(1-\sigma(z))}\,(\sigma(z)-y)$$

- $\sigma'(z) = \sigma(z)(1-\sigma(z))$

$$\boxed{\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y)} \qquad \boxed{\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x(\sigma(z)-y)}$$

- This tells us that the rate at which the weight learns is controlled by $(\sigma(z)-y)$, i.e. by the error in the output. The larger the error, the faster the neuron will learn.
- it also avoids learning slowdown caused by $\sigma'(z)$ in the eq^n $\partial C/\partial w = (a-y)\,\sigma'(z)\,x$ for quadratic cost.

- $\sigma'(z) = \frac{d}{dx}\left(\frac{1}{1+e^{-z}}\right) = \frac{d}{dx}(1+e^{-z})^{-1} = -(1+e^{-z})^{-2}(-e^{-z})$

$$= \frac{1}{(1+e^{-z})}\cdot\frac{e^{-z}}{(1+e^{-z})} = \frac{1}{(1+e^{-z})}\,\frac{(1+e^{-z})-1}{(1+e^{-z})}$$

$$= \sigma(z)\,(1-\sigma(z))$$

- cross-entropy of a multi-layer network with $y = y_1, y_2, \ldots$ as the desired values at the output neurons & $a_1^L, a_2^L, \ldots$ are the actual output values,

$$\boxed{C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1-y_j) \ln (1-a_j^L) \right]}$$

- cross-entropy of two probability distributions, p_j & q_j is

$$\sum_j p_j \ln q_j$$

- for "regression", y can take values intermediate b/w 0 & 1. The cross-entropy is then defined,

$$\boxed{C = -\frac{1}{n} \sum_x \left[y \ln y + (1-y) \ln (1-y) \right]}$$

and is known as binary entropy.

- from information theory, cross-entropy is a measure of surprise when we learn the true value for y.

Softmax

- The idea of softmax is to define a new type of output layer for neural networks.
- instead of applying a sigmoid func to the weighted inputs, z_j^L, we apply the softmax function. The activation of the j^{th} output neuron is,

$$\boxed{a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}}$$

where the sum is over all output neurons.

- $\sum_j a_j^L = 1$ for softmax activations. The output from the softmax layer is b/w 0 & 1 & can be thought of as a probability distribution.
- for softmax layer, any particular output activation a_j^L depends on all the weighted inputs.
- Suppose we've a neural net with a softmax output layer & the activations a_j^L are known then,

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \Rightarrow \ln(a_j^L) = z_j^L + \ln\left(\sum_k e^{z_k^L}\right)$$

$$\boxed{z_j^L = \ln(a_j^L) + C}$$

log-likelihood cost func.

$$\boxed{C = -\ln a_y^L}$$

$$\boxed{\frac{\partial C}{\partial b_j^L} = a_j^L - y_j} \qquad \boxed{\frac{\partial C}{\partial w_j} = a_k^{L-1}(a_j^L - y_j)}$$

weight decay / L2 regularization

- the idea is to add an extra term to the cost function called the "regularization term".
- regularized cross-entropy,

$$\boxed{C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2}$$

- $\sum w^2$ is the sum of the squares of all the weights in the network & λ is regularization parameter, where $\lambda > 0$
- the effect is so that the network prefers to learn small weights & large weights are allowed only if they considerably improve the first part of cost func.

- small λ ⇒ minimize the original cost func.
- large λ ⇒ small weights

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b} \Rightarrow b \rightarrow b - \eta \frac{\partial C_0}{\partial b}$$

$$\boxed{\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \eta/n \, \frac{\lambda}{w} \\ &= \left(1 - \eta\lambda/n\right) w - \eta \frac{\partial C_0}{\partial w} \end{aligned}}$$

- the rescaling factor $(1 - \eta\lambda/n)$ is called weight decay since it makes the weights smaller.
- small weights ⇒ lower complexity
- regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data & thus, generalize better from what they learn.

L1 regularization

$$\boxed{C = C_0 + \frac{\lambda}{n} \sum_w |w|}$$

where $\sum_i |w|$ is the sum of absolute values of weights.

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \, \text{sgn}(w)$$

where $\text{sgn}(w)$ is the sign of w, that is, $+1$ if w is positive & -1 if negative.

$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \, \text{sgn}(w) - \eta \frac{\partial C_0}{\partial w}$$

$$= w\left(1 - \frac{\eta\lambda}{n}\right) - \eta \frac{\partial C_0}{\partial w}$$

- in both L1 & L2 the intuition is to penalize larger weights
- in L1, the weights shrink by a constant amount toward 0 while in L2 the weights shrink by an amount proportional to w
- when a particular weight has large magnitude $|w|$, L1 shrinks the weight much less than L2 & vice-versa for when $|w|$ is small.

- L1 tends to concentrate the weight of the network in a relatively small no. of high-importance connections, while the other weights are driven toward zero.

Hessian technique

- consider a cost func C which is a func of many variables, $w = w_1, w_2, \dots$, so $C = C(w)$

by Taylor's theorem,

$$C(w + \Delta w) = C(w) + \sum_j \frac{\partial C}{\partial w_j} \Delta w_j + \frac{1}{2} \sum_{jk} \Delta w_j \frac{\partial^2 C}{\partial w_j \partial w_k} \Delta w_k + \dots$$

$$C(w + \Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w + \dots$$

- H is a "Hessian" matrix, whose jk^{th} term is $\partial^2 C / \partial w_{jk}$

$$\boxed{\Delta w = H^{-1} \nabla C}$$

algorithm:

- choose a starting point, w.

- update w to $w' = w - H^{-1}\nabla C$, where H & ∇C are computed at w.
- update w' to $w'' = w' - H'^{-1}\nabla' C$ where the H' & $\nabla' C$ are computed at w'....
- This approach to minimizing a cost func. is called "Hessian optimization".

tanh func

$$\boxed{\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}}$$

- $\sigma(z) = \frac{1 + \tanh(z/2)}{2}$ ($\tanh(z)$ is just a rescaled version of $\sigma(z)$)

ReLU (rectified linear unit)

- $\boxed{\text{output} = \max(0, w \cdot x + b)}$