50.021 -AI

Alex

Week 03: Basic neural networks

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Neural nets – an intro

A single Neuron - loose biological analogy

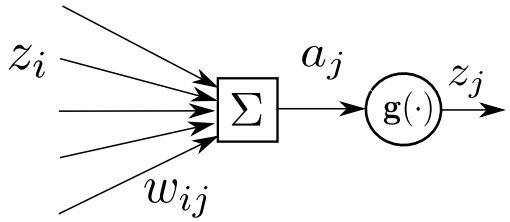
- set of inputs $\{z_i\}_{i=1}^{d}$ vector
- output z_j
- weights on inputs w_{ij}, bias b

Forward equation:

preactivation $a_j = \sum_{i=1}^{d} z_i w_{ij} + b$ linear mapping

$z_j = g(a_j)$ nonlinear activation (could be logistic function)

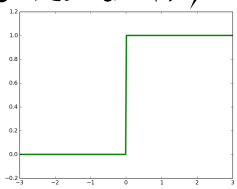
- non-linear activation function $g(\cdot)$



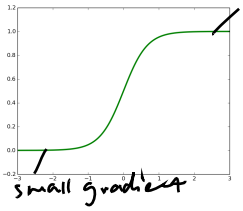
Lets consider examples of **activation functions:**

- function $g : \mathbb{R}^1 \longrightarrow \mathbb{R}^1$
- intuition: larger inputs $\rightarrow$ neuron fires more strong

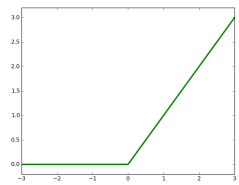
- can't be used for gradient-based optimization
- hard to reduce error



Threshold



Sigmoid

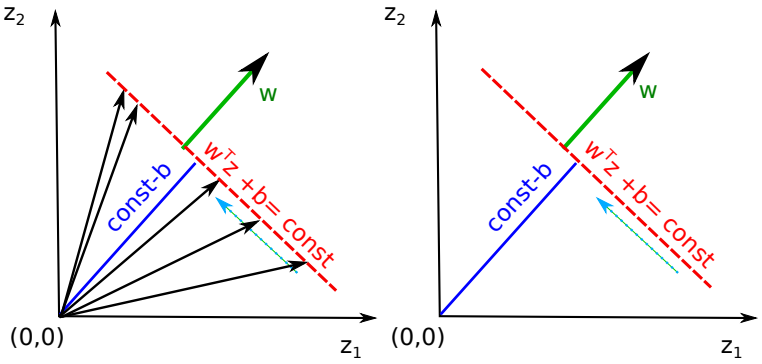


ReLU → better than Sigmoid

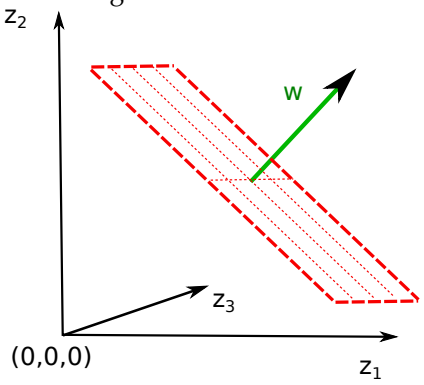
$$g(a) = [a > 0] \qquad g(a) = \frac{1}{1 + exp(-a)} \qquad g(a) = \max(0, a)$$

A single Neuron - What is the meaning of an activation function?

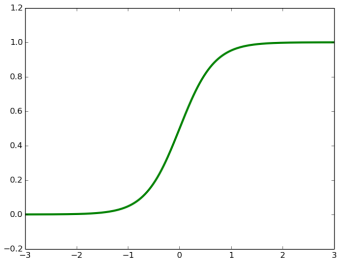
We have seen this in a previous lecture already.

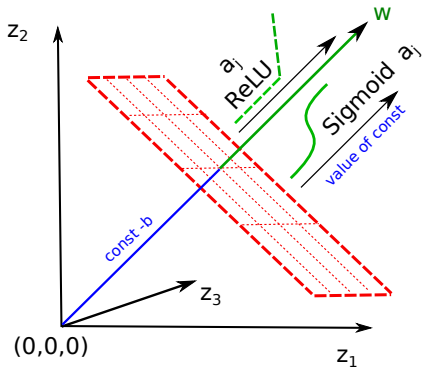


higher dimensions ??



activations do what?



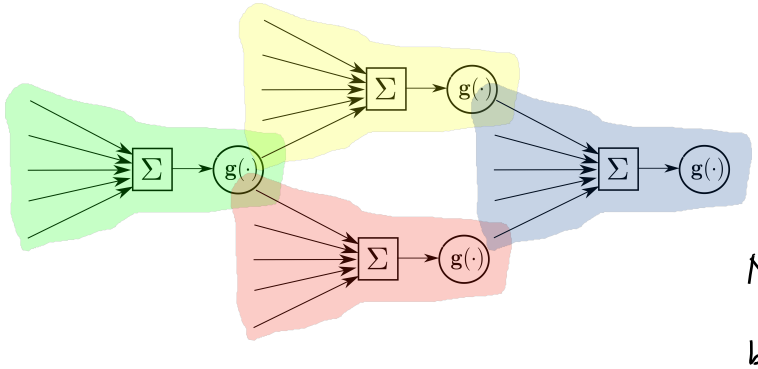


nonlinearly changing w in the direction

- output of activation function a_j is constant in the red plane
- output of activation function a_j varies along the direction of w

A neural network is a graph structure made from connected neurons.

- a neuron can be input to many other neurons
- a neuron can receive input to many other neurons
- each neuron has same structure $(g(\cdot), \sum)$ but different parameters (w_{ij}, b_j)
- can stack neurons in layers



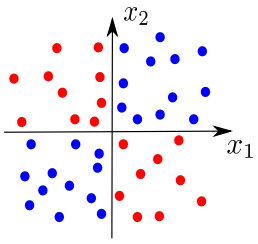
Non-linear activation function is important, because if we stack linear functions still result in linear function

The XOR problem ++ +- -- -+

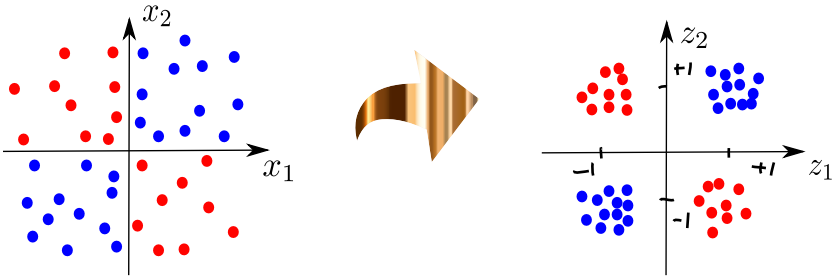
This is an example how a concatenation of neurons allows to learn non-linear mappings!

separate red from blue samples.

Samples lie in quadrants around coordinated $(\pm 1, \pm 1)$

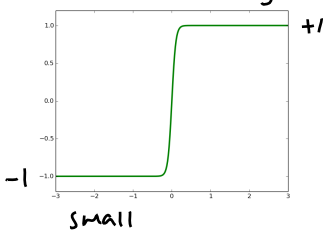


We will show: a neural network with some weights can separate these two classes.



$$z_1 = tanh(10x_1)$$
$$z_2 = tanh(10x_2)$$

large ↳ steep step

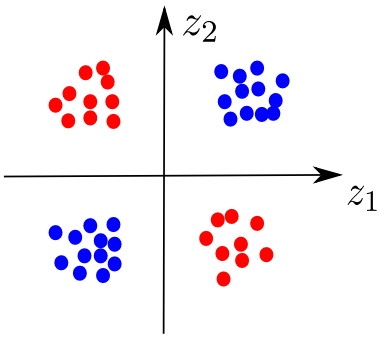


- *tanh* pushes points towards $(\pm 1, \pm 1)$

all points close to $(\pm 1, \pm 1)$

$$z_3 = tanh(z_1 + z_2 - 1)$$
$$z_4 = tanh(-(z_1 + z_2) - 1)$$

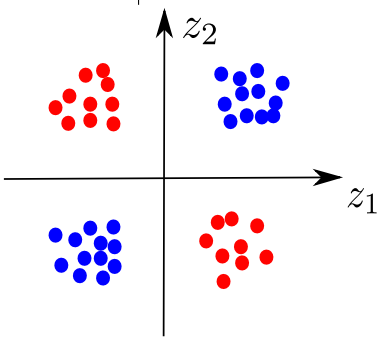
	(z_1, z_2)	$z_1 + z_2 - 1$	z_3 $tanh(\cdot)$	$-(z_1 + z_2) - 1$	z_4 $tanh(\cdot)$	
Blue	$(-1, -1)$	-3	-1	$+1$	$+1$	either z_3 or z_4 is $+1$
	$(+1, +1)$	$+1$	$+1$	-3	-1	
Red	$(-1, +1)$	-1	-1	-1	-1	both z_3 & z_4 are -1
	$(+1, -1)$	-1	-1	-1	-1	



as a consequence:
$z_3 + z_4 \approx 0$ for $(z_1, z_2) = (-1,-1),(+1,+1)$
$z_3 + z_4 \approx -2$ for $(z_1, z_2) = (+1,-1),(-1,+1)$
$z_3 + z_4$ Separates samples!

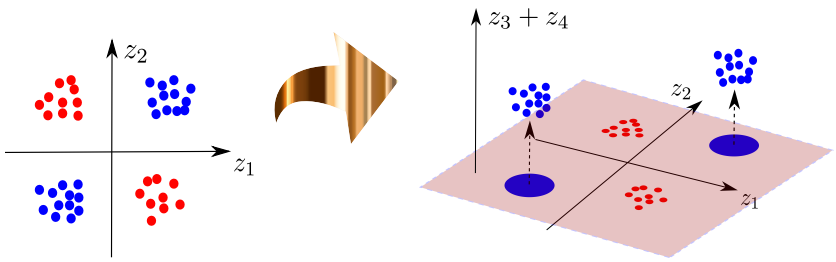
$$z_3 = tanh(1 + (z_1 - z_2))$$
$$z_4 = tanh(1 - (z_1 - z_2))$$

z_1	z_2	$1 + (z_1 - z_2)$	z_3	$1 - (z_1 - z_2)$	z_4
-1	-1	$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	$+1$	$+1$	$+1$	$+1$
-1	$+1$	-1	-1	$+3$	$+1$
$+1$	-1	$+3$	$+1$	-1	-1

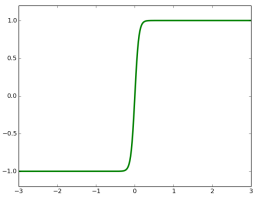


as a consequence:
$z_3 + z_4 \approx 2$ for $(z_1, z_2) = (-1,-1),(+1,+1)$
$z_3 + z_4 \approx 0$ for $(z_1, z_2) = (+1,-1),(-1,+1)$
$z_3 + z_4$ Separates samples! ~ $z_3 + z_4$ has learned to classify

OOPS ... multiple solutions! (just like multiple local maxima)



$$z_3 = tanh(z_1 + z_2 - 1)$$
$$z_4 = tanh(-(z_1 + z_2) - 1)$$



- output $z_3 + z_4$ separates!

Universal approximation theorem

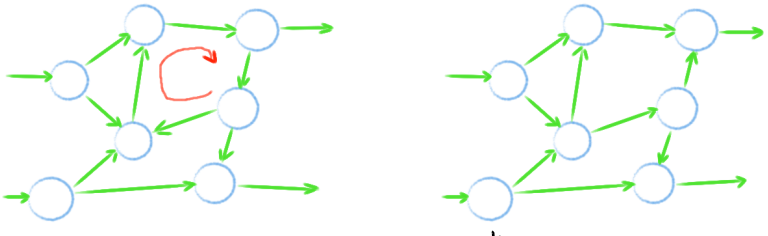
It has many versions of it.

- Kolmogorov (1957), Hornik (1989), Cybenko (1989) and others: Neural network can approximate *any continuous function on a compact region*
- can approximate any function $\neq$ able to learn well from training data (!!)
- training of neural networks is often difficult with many "surprises"
- can represent but maybe cannot learn the necessary representation

Definition: Neural Network

Any directed graph built from neurons is a neural network! –The Definition ends here.

- two types: recurrent and feedforward neural networks



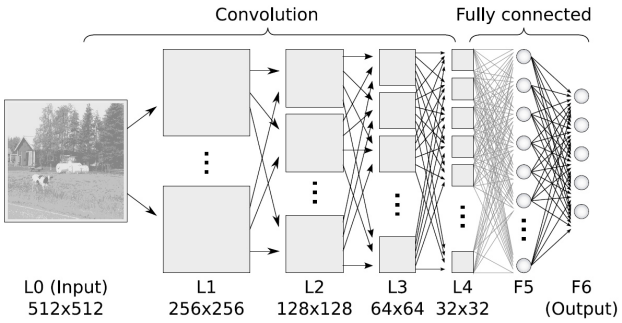
recurrent (not covered in this lecture) e.g. Speech processing

feedforward e.g. Image classification

Quick over view over Neural Networks so far:

- neuron is a simple unit
- complexity by combining many simple units
- classification: can learn non-linear boundaries
- example: XOR-problem

How to learn the parameters in the neurons? Consider a multi-layer neural network for image classification



taken from:`http://www.ais.uni-bonn.de/deep_learning/`

- use gradient! — for all weights of all neurons
- how to compute gradient for one weight w_{ij} deep in this nested structure ?

how to learn neural network parameters?

- define a loss function E on outputs $f(x)$ of the NN. NN has parameters w
- compute gradient of the loss function w.r.t. NN parameters w
- optimize parameters by gradient descent using $\frac{\partial E(f(x),y)}{\partial w}$

The loss function

- need to define a loss E
- example loss E for: *Multiclass Classification*
- assume: have 3 output neurons for three classes z_1,z_2,z_3 – ice cream, oranges, everything else.
- a sample x_i is fed into a neural network, produces 3 outputs $z_1(x_i), z_2(x_i), z_3(x_i)$
- map outputs $z_c(x_i)$ to "probabilities" p_c, then apply loss to p_c

$$p_c = \frac{e^{z_c}}{\sum_{c'=1}^{3} e^{z_{c'}}}$$
$$\sum_{c=1}^{3} p_c = \frac{\sum_{c=1}^{3} e^{z_c}}{\sum_{c'=1}^{3} e^{z_{c'}}} = 1$$
$$p_c = \frac{e^{z_c}}{\sum_{c'=1}^{3} e^{z_{c'}}}$$

Note: this is a generalization of the logistic function to more than two classes. See it for two classes:

$$p_1(z_1, z_2) = \frac{e^{z_1}}{\sum_{c'=1}^{2} e^{z_{c'}}} = \frac{e^{z_1}}{e^{z_1} + e^{z_2}} = \frac{e^{z_1}}{e^{z_1} + C}$$
$$p_1(z_1, z_2 = 0) = \frac{e^{z_1}}{e^{z_1} + 1} = \frac{1}{e^{-z_1} + 1}$$

- maximizing the likelihood of p_c is equivalent to minimizing the neg log likelihood
- minimize loss as: neg log likelihood
- likelihood for one datapoint x_i:

$$\prod_{c=1}^{3} p_c(x_i)^{(y_{c,i}+1)/2}$$
$$y_{c,i} = \begin{cases} 1 & \text{if } c \text{ is the class label for } x_i \\ -1 & \text{else} \end{cases}$$

similar to $h(x)^y (1 - h(x))^{1-y}$

- neg log likelihood for one sample x_i

$$-1 \sum_{c=1}^{K} \log(p_c(x_i)) \frac{y_{ci} + 1}{2}$$

- minimize loss as: minimize neg log likelihood
- neg log likelihood for one sample x_i

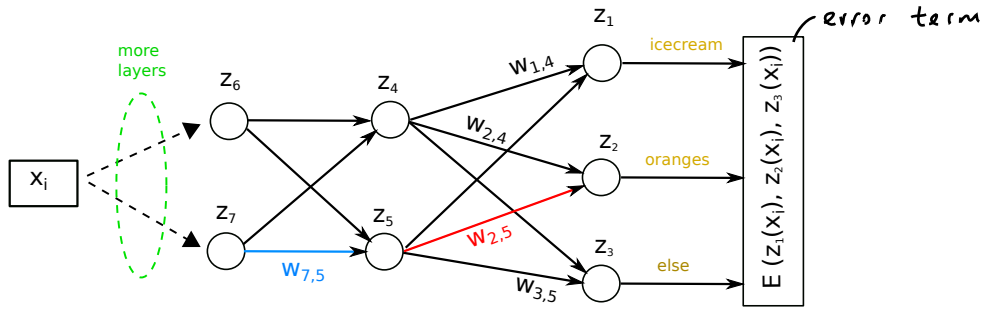
$$-1 \sum_{c=1}^{K} \log(p_c(x_i)) \frac{y_{ci} + 1}{2}$$

- loss E for a set of x_i is sum over all x_i:

$$E = -1 \sum_{i \in data} \sum_{c=1}^{K} \log(p_c(x_i)) \frac{y_{ci} + 1}{2}$$
$$\Rightarrow E = -1 \sum_{i \in data} \sum_{c=1}^{K} t(z_c(x_i)) \frac{y_{ci} + 1}{2}$$

(with $t(z_c) := log(\frac{e^{z_c}}{\sum_{c'=1}^{3} e^{z_{c'}}})$)

- E as a function of the neuron outputs z_1, z_2, z_3



$\frac{\partial E}{\partial w_{2,5}} = ???$ back propagation

have now a loss function for the three neuron outputs z_1, z_2, z_3

- example: compute gradient with respect to parameters w_{25} of neuron z_2

$$\frac{\partial E}{\partial w_{2,5}} = \frac{\partial E}{\partial z_2} \frac{z_2}{\partial w_{2,5}}$$

out put

output

chain rule

- important: both terms $\frac{\partial E}{\partial z_2}$ and $\frac{z_2}{\partial w_{2,5}}$ easily computable in terms of NN structure

$$z_2(x_i) = g(w_{2,4} z_4(x_i) + w_{2,5} z_5(x_i))$$

look at the graph

$$\frac{\partial z_2}{\partial w_{2,5}} = g'(w_{2,4} z_4(x_i) + w_{2,5} z_5(x_i)) \cdot z_5(x_i)$$

derivative of the inner term

- E is defined directly as a function of z_2 and z_1, z_3

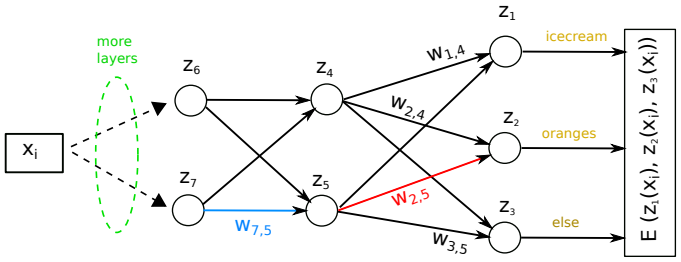
$$\frac{\partial E}{\partial z_2} = -1 \sum_{i \in data} \left(\frac{y_{2i} + 1}{2} - \frac{e^{z_2(x_i)}}{\sum_{c'=1}^{3} e^{z_{c'}(x_i)}} \right)$$

How about $\frac{\partial E}{\partial w_{7,5}} = ???$

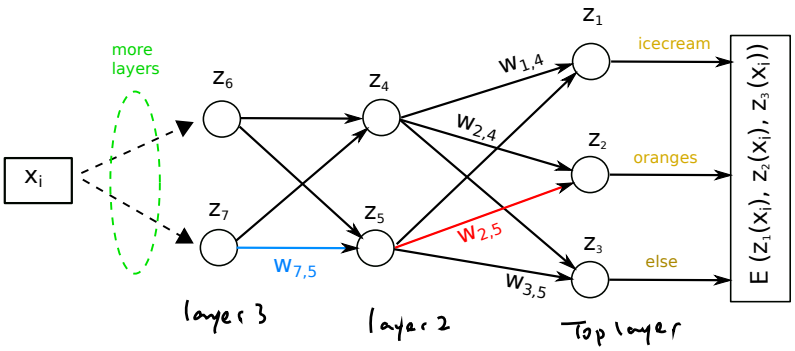
- same approach - using chain rule

$$\frac{\partial E}{\partial w_{7,5}} = \frac{\partial E}{\partial z_5} \frac{\partial z_5}{\partial w_{7,5}}$$

- $\frac{\partial E}{\partial z_5}$ not directly computable (z_5 is no input to E)



- idea: apply top-down chainrule to compute $\frac{\partial E}{\partial z_5}$ – we know all the functions that depend on z_5 as input: z_1, z_2, z_3. Use this:



$$\frac{\partial E}{\partial z_5} = \frac{\partial E}{\partial z_1}\frac{\partial z_1}{\partial z_5} + \frac{\partial E}{\partial z_2}\frac{\partial z_2}{\partial z_5} + \frac{\partial E}{\partial z_3}\frac{\partial z_3}{\partial z_5}$$

- know already how to compute $\frac{\partial E}{\partial z_1}, \frac{\partial E}{\partial z_2}, \frac{\partial E}{\partial z_3}$ – can be done directly by differentiating E.
- open is only $\frac{\partial z_1}{\partial z_5}, \frac{\partial z_2}{\partial z_5}, \frac{\partial z_3}{\partial z_5}$

$$z_2(x_i) = g(w_{2,4}z_4(x_i) + w_{2,5}z_5(x_i))$$
$$\frac{\partial z_2}{\partial z_5} = g'(w_{2,4}z_4(x_i) + w_{2,5}z_5(x_i)) \cdot w_{2,5}$$

very similar to

$$\frac{\partial z_2}{\partial w_{2,5}} = g'(w_{2,4}z_4(x_i) + w_{2,5}z_5(x_i)) \cdot z_5$$

Observation

- $\frac{\partial E}{\partial w_{7,5}} = \frac{\partial E}{\partial z_5}\frac{\partial z_5}{\partial w_{7,5}}$
- $\frac{\partial E}{\partial z_5}$ can be computed by top-down chain rule

$$\frac{\partial E}{\partial z_5} = \frac{\partial E}{\partial z_1}\frac{\partial z_1}{\partial z_5} + \frac{\partial E}{\partial z_2}\frac{\partial z_2}{\partial z_5} + \frac{\partial E}{\partial z_3}\frac{\partial z_3}{\partial z_5}$$

- $\frac{\partial z_2}{\partial z_5}$ is easy to compute by NN structure, similar to $\frac{\partial z_2}{\partial w_{25}}$
- (final result:)

$$\frac{\partial E}{\partial w_{5,7}} = \frac{\partial E}{\partial z_1}\frac{\partial z_1}{\partial z_5}\frac{\partial z_5}{\partial w_{5,7}} + \frac{\partial E}{\partial z_2}\frac{\partial z_2}{\partial z_5}\frac{\partial z_5}{\partial w_{5,7}} + \frac{\partial E}{\partial z_3}\frac{\partial z_3}{\partial z_5}\frac{\partial z_5}{\partial w_{5,7}}$$

- have shown: $\frac{\partial z_i}{\partial w_{i,j}}, \frac{\partial z_i}{\partial z_j}$ are simple terms
- can use chainrule to get every $\frac{\partial E}{\partial z_i}$

- there is a pattern in these computations – formalize it

Backprop Takeaway 1

- goal: compute $\frac{\partial E}{\partial w_v^{(l)}}$, need to get: $\frac{\partial E}{\partial z^{(l)}}$ so that we can compute

$$\frac{\partial E}{\partial w_v^{(l)}} = \frac{\partial E}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w_v^{(l)}}$$

- assume each neuron $z^{(l)}$ has an index (l) denoting its layer.
- assume there are no shortcuts between layers, can be treated similarly, but messy to write down
- **backpropagation is essentialy only a top-down application of chainrule**

$$\begin{aligned}
\text{init: } & \frac{\partial E}{\partial z^{(TOP)}} && = \text{direct compute} \\
\text{layer } (TOP-1) & \frac{\partial E}{\partial z^{(TOP-1)}} && = \sum_{z^{(TOP)}} && \frac{\partial E}{\partial z^{(TOP)}} \frac{\partial z^{(TOP)}}{\partial z^{(TOP-1)}} \\
& \cdots\cdots && \cdots && \cdots\cdots \\
\text{layer } (k) & \frac{\partial E}{\partial z^{(k)}} && = \sum_{z^{(k+1)}} && \frac{\partial E}{\partial z^{(k+1)}} \frac{\partial z^{(k+1)}}{\partial z^{(k)}} \\
& \cdots\cdots && \cdots && \cdots\cdots \\
\text{layer } (l) & \frac{\partial E}{\partial z^{(l)}} && = \sum_{z^{(l+1)}} && \frac{\partial E}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial z^{(l)}} \\
\text{layer } (l) \text{ of weight } & \frac{\partial E}{\partial w_v^{(l)}} && = \sum_{z^{(l)}} && \frac{\partial E}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w_v^{(l)}}
\end{aligned}$$

layer with weights

↳ this step can be repeated for all layers, to calculate all weights

Backprop Takeaway 2

- above approach can compute also derivatives $\frac{\partial E}{\partial w_v^{(k)}}$ at all layers (k), not only (l) when going down the layers
- general top-down algorithm – see below

Back propagation:

TOP DOWN CHAIN RULE

Backpropagation as algorithm for layered NNs

for *layerindex* $l = N \rightarrow 1$

- if $l == N$: compute $\frac{\partial E}{\partial z_i^{(l)}}$ directly
  (error as a function of NN outputs)

- else: use precomputed $\frac{\partial E}{\partial z_j^{(l+1)}}$,

  compute and store $\frac{\partial E}{\partial z_i^{(l)}} = \sum_{z_j^{(l+1)}} \frac{\partial E}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}}$

- compute partial derivative for variable $w_{hi}^{(l)}$ as $\frac{\partial E}{\partial w_{hi}^{(l)}} = \frac{\partial E}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{hi}^{(l)}}$

  end for

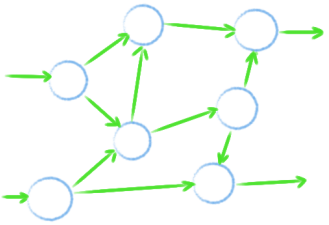
Now: assemble partial derivatives for variables w_{ij} into one huuuge gradient vector ... and descend in w-space!

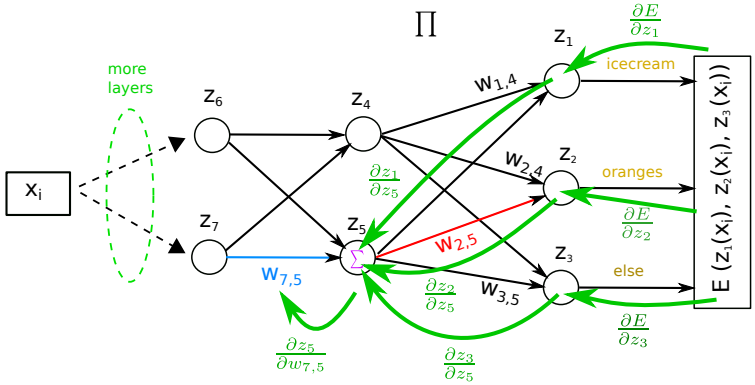
another Backprop Takeaway

- backprop: chainrule PLUS we can reuse terms:
- reuse $\frac{\partial z^{(k)}}{\partial z^{(k-1)}}$ for all weights in levels below $k-1$
- recycling of $\frac{\partial z^{(k)}}{\partial z^{(k-1)}}$ is the key to fast computation of gradient

Backpropagation in general

- does work without layer assumptions.
- replace *neurons of next higher layer* $(l+1)$ by
  neurons such that the current neuron is input to them





- "flow" pattern

Neural network Implementation: Backpropagation with batch sizes

One important trick: when running toolbox code, the toolboxes create of every layer as many copies as the batchsize (or they create a layer with dimensionality being multiplied by the batch size). The forward computation runs in parallel for all samples in a batch.

Care needs to be taken when implementing backpropagation in such setup. For neuron activations implementation is as usual

$$Y = \frac{1}{1 + e^{-x}}$$

Then what one needs to return to the layer below is:

$$\frac{dE}{dX} = \frac{dE}{dY} * \frac{dY}{dX}$$

however there are layers with shared parameters. For example a fully connected layer

$$Y = W \cdot X + b$$

There are multiple copies of X – one for every sample in the batch, but the parameters (W, b) are shared across all samples in the batchsize. Suppose that X would have dimension K if one would use a batchsize of 1, and Y would have dimension M if one would use a batchsize of 1. Then W would have dimension $K \times M$, no matter what batchsize.

matrix multiplication

Now $\frac{dE}{dX}$ and $\frac{dE}{dW}$ must be treated differently. $\frac{dE}{dX}$ is same as above for the activation layer. Note: if it has a batch size of 5, then $\frac{dE}{dX}$ has dimensionality $5 \cdot K$.

Following the formula:

$$\nabla \sum_{(x_i, y_i) \in Batch} E(f(x_i), y_i)$$

The

however there are layers with shared parameters. For example a fully connected layer

$$Y = W \cdot X + b$$

There are multiple copies of X – one for every sample in the batch, but the parameters (W, b) are shared across all samples in the batchsize. Suppose that X would have dimension K if one would use a batchsize of 1, and Y would have dimension M if one would use a batchsize of 1. Then W would have dimension $M \times N$, no matter what batchsize.

Now $\frac{dE}{dX}$ and $\frac{dE}{dW}$ must be treated differently. $\frac{dE}{dX}$ is same as above for the activation layer. Note: if it has a batch size of 5, then $\frac{dE}{dX}$ has dimensionality $5 \cdot K$.

For $\frac{dE}{dW}$, using the formula:

$$\nabla \left(\sum_{(x_i, y_i) \in Batch} E(f(x_i), y_i) \right) : \text{ The } \cdot \text{ in the term } \frac{dE}{dW} = \frac{dE}{dY} \cdot \frac{dY}{dW}$$

says that it needs to be summed over all elements of the batch, when computing $\frac{dE}{dW}$ for updating W.

If we have a batch size of 5, then (up to transposition and reshaping)

$X.shape = (5, K)$ $W.shape = (K, M)$

$Y.shape = (5, M)$

$\frac{dE}{dY}.shape = Y.shape = (5, M)$

$\frac{dY}{dW}.shape = (5K, M)$ or $(K, 5M)$ or $(5, K, M)$ – be aware of that additional 5

(5k, 1) (k, 5) (5, k)