50.039 – Theory and Practice of Deep Learning

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.]

1 Neural nets – an intro

Key takeaways:

- definition of a neuron and a neural net
- the capability of neural networks to separate non-linear problems
- the universal approximation theorem
- insights in the approximation capabilities of neural networks by constructive approaches
- neural networks for multiclass classification problems: one hot vectors, softmax for generating probabilities, the cross-entropy loss function

1.1 A single Neuron - loose biological analogy

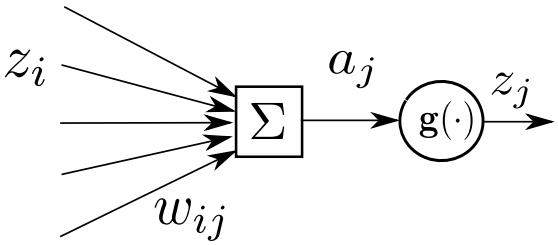
- set of inputs $\{z_i\}_{i=1}^{d}$
- output z_j
- weights on inputs w_{ij}, bias b

  Forward equation:

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

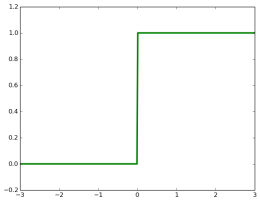
$$z_j = g(a_j) \qquad \text{nonlinear activation}$$

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

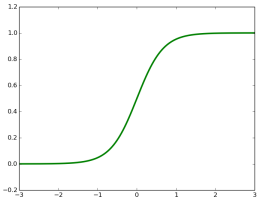


Lets consider examples of **activation functions:**

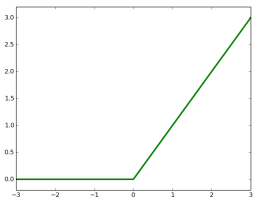
- function $g : \mathbb{R}^1 \longrightarrow \mathbb{R}^1$
- intuition: larger inputs $\rightarrow$ neuron fires more strong



Threshold



Sigmoid

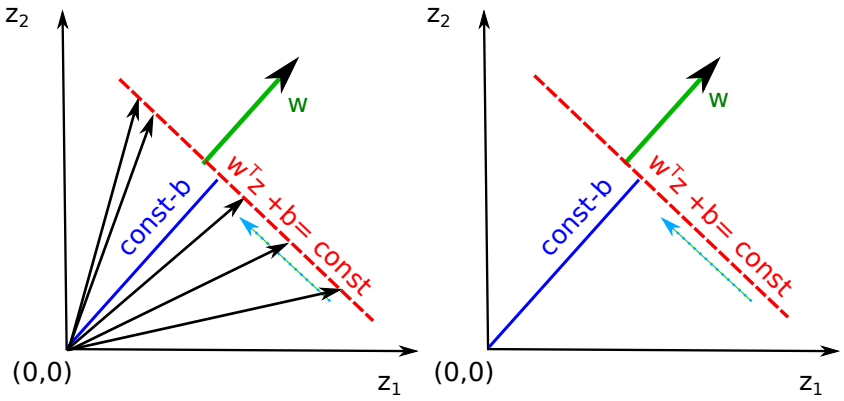


ReLU

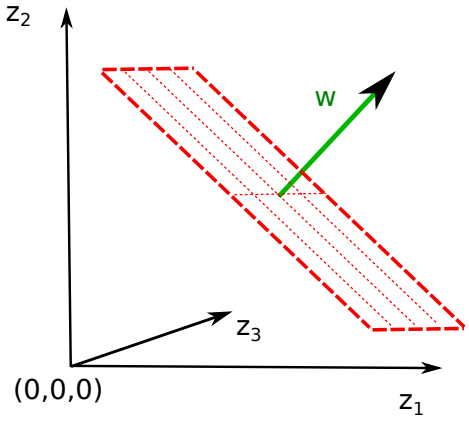
$$g(a) = [a > 0] \qquad g(a) = \frac{1}{1 + exp(-a)} \qquad g(a) = \max(0, a)$$

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

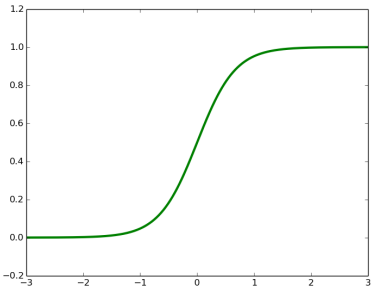
We have seen this in a previous lecture already.

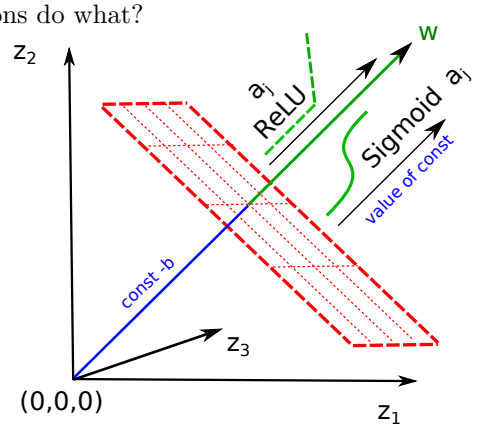


higher dimensions ??



activations do what?

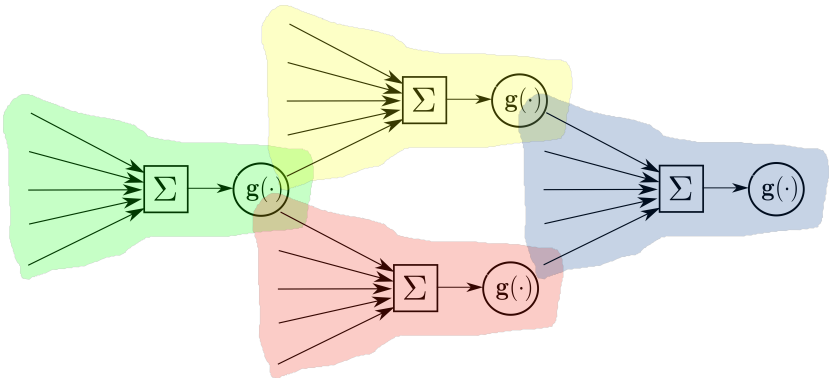




- output of activation function a_j is constant in the red plane
- output of activation function a_j varies along the direction of w

A neural network is a directed 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

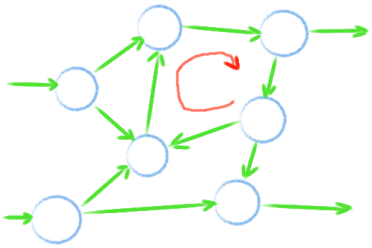


Definition: Neural Network

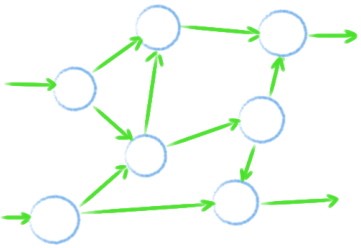
Any directed graph built from neurons is a neural network! –The Definition ends here.

The art lies in: loss design, algorithms to update weights, data representation as inputs, data augmentation, model design (not the most important)

- two types: recurrent and feedforward neural networks



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



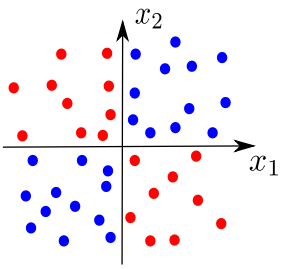
feedforward
e.g. Image classification

1.3 The XOR problem

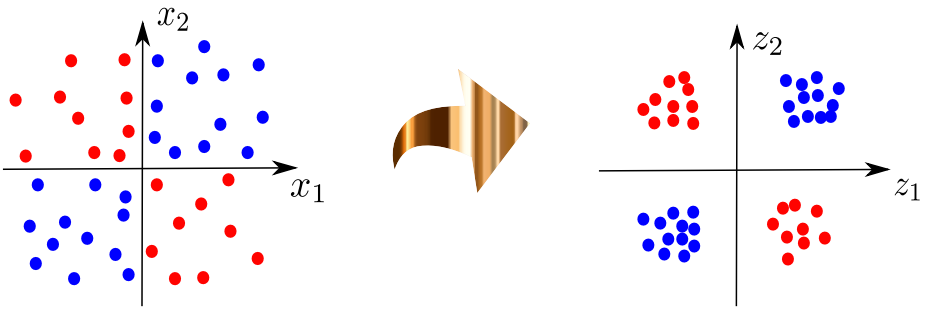
This is an example how a concatenation of neurons allows to learn non-linear mappings!

separate red from blue samples.

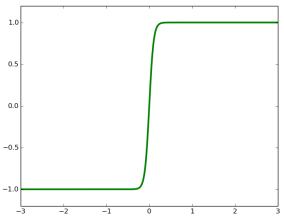
Samples lie in quadrants around coordinated $(\pm 1, \pm 1)$



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



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

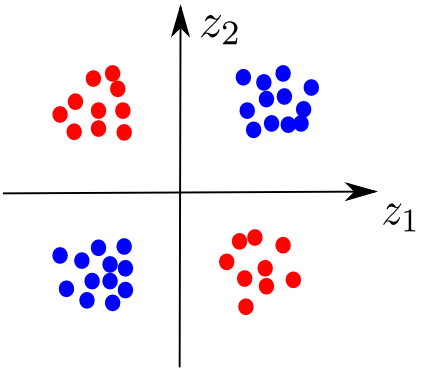


- *tanh* pushes points towards $(\pm 1, \pm 1)$

Next idea: Have a detector z_3 which fires only for $(-1, -1)$, and a detector z_4 which fires only for $(+1, +1)$. Then their sum would fire if any of these cases is present.

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)	$z_1 + z_2 - 1$	$tanh(\cdot)$	$-(z_1 + z_2) - 1$	$tanh(\cdot)$
$(-1, -1)$	-3	-1	$+1$	$+1$
$(+1, +1)$	$+1$	$+1$	-3	-1
$(-1, +1)$	-1	-1	-1	-1
$(+1, -1)$	-1	-1	-1	-1

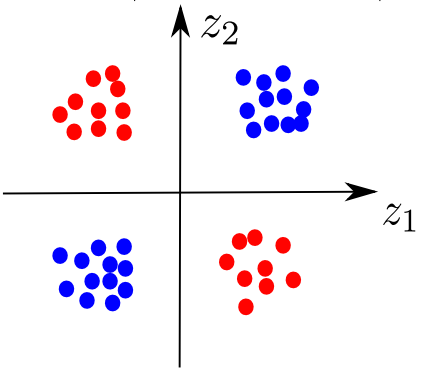


as a consequence:

$z_3 + z_4 \approx 0$ 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!

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

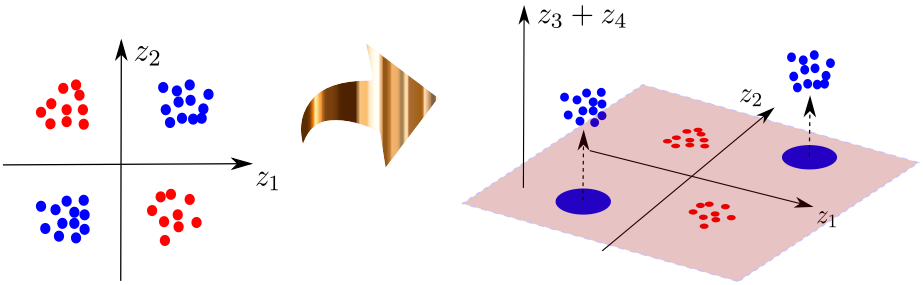
z_1	z_2	$1 + (z_1 - z_2)$	z_3	$1 - (z_1 - z_2)$	z_4
-1	-1	$+1$	$+1$	$+1$	$+1$
$+1$	$+1$	$+1$	$+1$	$+1$	$+1$
-1	$+1$	-1	-1	$+3$	$+1$
$+1$	-1	$+3$	$+1$	-1	-1



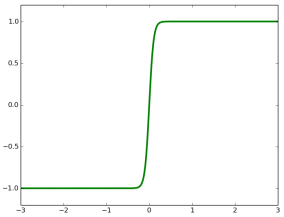
as a consequence:

$z_3 + z_4 \approx 2$ 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!

OOPS ... multiple solutions!



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



- output $z_3 + z_4$ separates!

1.4 Universal approximation theorem

It has many versions of it. One:

Universal approximation theorem:

Let $g(\cdot)$ be a continuous function on a m-dimensional hypercube $[0,1]^m$. Let $a(\cdot)$ be a non-constant, bounded, continuous (activation) function. Then $f(\cdot)$ can be approximated arbitrarily well, that is for every maximal deviation $\epsilon > 0$, there exists a set of weights u_i, w_i and biases b_i such that

$$\forall x \in [0,1]^m : \ |g(x) - (\sum_{i=1} u_i a(w_i x + b_i) + b)| < \epsilon$$

Neural networks with one hidden layer, and two layers of weights can approximate any smooth function on a compact hypercube. If there is a good algorithm for learning the parameters from data, we are done!

Universal approximation theorem is the arch-evil of neural network research. Why?

- 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 (!!)
- can represent the necessary representation but maybe almost impossible to learn from data

1.5 More insights into approximation capabilities

Michael Nielsen gives an insight into approximation capabilites by showing that two layers of a neural network can be used to approximate step functions. Then any 1-dimensional function can be approximated by a step function.

One can show:

- one neuron with sufficiently large weight separates a single hyperplane.
- a set of neurons represents a set of hyperplanes
- adding a layer on top can learn to detect an intersection of hyperplanes in an AND-like fashion. (assume the activations have $g(0) = 0$, then for N inputs, if the sum is above $N - 1$, all N neurons must have fired, or if the minimum is above 0)
- result is an approximate intersection of hyperplanes.
- for a larger set of neurons: approximate generic convex shapes in the input space (what is a convex set of images?)
- two disjoint shapes or one non-convex? add a layer for an OR-combination of convex sets. (assume the activations have $g(0) = 0$, then for N inputs, if the maximum is above 0, one neuron must have fired)
- Important: I do NOT CLAIM that neural networks learn in this way. They do not (in particular when weights are regularized). Though, it can be used that they have the capability to represent very general shapes as decision boundaries for classification.

Quick overview over Neural Networks so far

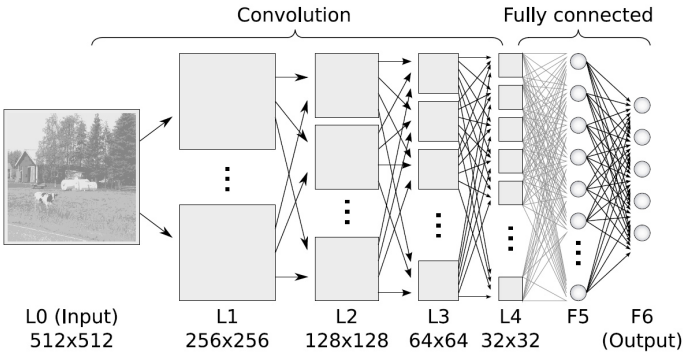
- neuron is a simple unit
- complexity by combining many simple units
- classification: can learn non-linear boundaries
- example: XOR-problem

1.6 How to learn neural network parameters?

1. define a loss function L on outputs $f(x)$ of the NN.
   - for the supervised case we have pairs (x, y) of samples x and ground truth labels y (see second lecture)
   - loss function $L(f(x), y)$ measures difference between the prediction $f(x)$ and what should be the correct outcome – given by the label y.

2. The NN (which models $f(x)$) has parameters w, so actually $f(x) = f_w(x)$. Consider the loss over a minibatch

$$L(\text{Minibatch}, f) = \sum_{i \in \text{minibatch}} L(f(x_i), y_i)$$



   Compute gradient $\frac{\partial L(\text{Minibatch}, f)}{\partial w}$ of the loss function w.r.t. NN parameters w. $\longrightarrow$ See later lecture on backpropagation.

3. optimize parameters by gradient descent using $\frac{\partial L(\text{Minibatch}, f)}{\partial w}$

$$w_{t+1} = w_t - \epsilon_t \frac{\partial L(\text{Minibatch}, f)}{\partial w}$$

   - ϵ_t is a step size parameter of the gradient. In toolboxes: the learning rate schedulers define how ϵ_t change over time.
   - More information later: 1. lecture on backpropagation, 2. lecture on better methods to apply gradients

1.7 The loss function for Multiclass Classification

- Input space $\mathcal{X} = \mathbb{R}^d$ - some vector space
- Output space $\mathcal{Y} = \{0, \ldots, C-1\}$ - indices for C classes

Multiclass Classification / Multilabel Classification:

Multiclass Classification is a setting where we want to predict whether one input sample x_i falls into one of **mutually exclusive** classes $\{0, \ldots, C-1\}$.

Multilabel Classification is a setting where we want to predict whether an input sample x_i belongs to classes $\{0, \ldots, C-1\}$, which are not mutually exclusive. That is a sample can have multiple ground truth labels. In that case one has C binary classification problems.

Before defining the loss function, lets introduce a more convenient output space, and find a way to map neural network outputs into a vector of probabilities for C classes.

output space representation: one-hot vectors

We need to model the ground truth label y_i for sample x_i. A common way is a **one-hot-vector**, that is $y_i \in \mathbb{R}^C$ a vector of all zeros, except the one class, which is present.

$$y_i = (0, \ldots, 0, \underbrace{1}_{\text{index of true class}}, 0, \ldots)$$

This gives rise to a different output space

$$\mathcal{Y} = \{y \in \mathbb{R}^C : y_c = 0 \text{ or } y_c = 1, \sum_c y_c = 1\}$$

In the following we will denote

$$y_{i,k}$$

as the k-th component of the vector y_i in this representation.

creating probabilistic outputs for C classes

In last lecture we have done this for two-classes with the logistic sigmoid.

Suppose we have a neural network which produces for each input sample x_i a vector $z(x_i)$ of C outputs. Lets denote these outputs as a vector-valued function $z(x_i) = (z_1(x_i), \ldots, z_C(x_i))$. These outputs are real numbers, possibly negative.

for a simple example consider as a neural network:

$$z_c(x_i) = w_c \cdot x_i + b_c, \;\; c = 0, \ldots, C-1$$

That is, one has for every class c a linear model with parameters w_c, b_c. Now lets introduce the softmax mapping.

Softmax function:

$$p_c(x_i) = \frac{e^{z_c(x_i)}}{\sum_{c'} e^{z_{c'}(x_i)}}$$

is a way to map any real-valued vector $z(x_i)$ onto a set of positive numbers that sum up to one. It has an interpretation as $p_c(x_i) = P(Y = c|X = x_i)$

How can we remember this ?

We need to map the vector $z(x_i)$, which can have negative real numbers, onto a vector of non-negative probabilities. It should have the property, that larger outputs $z_c(x_i)$ result in larger probabilities. Firstly, the mapping

$$z_c(x_i) \mapsto e^{z_c(x_i)} > 0$$

creates a strictly positive number, and it is strictly monotonously increasing. Finally, these numbers need to sum up to one, so we can simply normalize them by their sum

$$z_c(x_i) \mapsto e^{z_c(x_i)} \mapsto \frac{e^{z_c(x_i)}}{\sum_{c'} e^{z_{c'}(x_i)}}$$

This is still strictly positive, and sums up to one, and normalization does not break the requirements of larger outputs to larger probabilities

$$\sum_c \frac{e^{z_c(x_i)}}{\sum_{c'} e^{z_{c'}(x_i)}} = \frac{\sum_c e^{z_c(x_i)}}{\sum_{c'} e^{z_{c'}(x_i)}} = 1$$

So we have something that can be used as a probability.
For two outputs $C = 2$ compare this to the logistic sigmoid:

$$\frac{e^{z_1}}{e^{z_1} + e^{z_2}} = \frac{1}{1 + e^{z_2 - z_1}} = \frac{1}{1 + e^{-(z_1 - z_2)}}$$

$$\underline{z} := z_1 - z_2$$

$$\Rightarrow \frac{1}{1 + e^{-\underline{z}}} = \frac{e^{z_1}}{e^{z_1} + e^{z_2}}$$

What does this mean ?

- For 2 classes, taking the difference of both outputs $z_1 - z_2$ and plugging that differential output into the logistic sigmoid gives the softmax for a neural network with two outputs.

- if we use a logistic sigmoid on a neural network which has one output $\underline{z}$ (as we did with logistic regression!) – to compute the probability of $P(Y = +1|X = x)$ in a two class problem, then this one output can be understood as the difference of two outputs of a neural network with two outputs for two different classes

Since $p_c > 0$ and $\sum_c p_c = 1$, we can interpret p_c as conditional probability to predict class c: $p_c(x_i) = P(Y = c|X = x_i)$.
Compare this to $h(x_i) = P(Y = +1|X = x_i)$ in the case of the logistic sigmoid.

loss function for multi-class classification with C classes
We go here in an analogy from the two class case from last lecture:

Definition: cross-entropy loss for 2-class classification

Suppose we have for every sample x_i a function $p(x_i) = (p_1(x_i), p_2(x_i))$ which provides a probability $p_1(x_i) = P(Y = +1|X = x_i)$ for the positive class and $p_2(x_i) = P(Y = -1|X = x_i)$ for the negative class.
Furthermore let $\bar{y}_i = (y_i + 1)/2$ be the mapping of the labels $y_i \in \{-1, +1\}$ onto $\{0, 1\}$.
Then the crossentropy loss for a sample (x_i, y_i) is given as

$$\begin{aligned} L(p(x_i), y_i) &= \begin{cases} -\log(p_1(x_i)) & \text{if } y_i = +1 \\ -\log(p_2(x_i)) & \text{if } y_i = -1 \end{cases} \\ &= -\bar{y}_i \log(p_1(x_i)) - (1 - \bar{y}_i) \log(1 - p_1(x_i)) \end{aligned}$$

Now we have $y_{i,c} \in \{0, 1\}$ instead of $y_i \in \{-1, +1\}$. We can generalize accordingly:

Definition: cross-entropy loss for C-class classification

Suppose we have for every sample x_i a function $p(x_i) = (p_1(x_i), p_2(x_i), \ldots p_C(x_i))$ which provides a probability $p_c(x_i) = P(Y = c|X = x_i)$.
Then the crossentropy loss for a sample (x_i, y_i) is given as

$$\begin{aligned} L(p(x_i), y_i) &= \begin{cases} -\log(p_c(x_i)) & \text{if } y_{i,c} = +1 \end{cases} \\ &= \sum_{c=1}^{C} -y_{i,c} \log(p_c(x_i)) \end{aligned}$$

Thus, in a multiclass setting, for a sample $(x_i, y_i) = (x_i, (y_{i,1}, \ldots, y_{i,C}))$, the cross-entropy loss is just the neg log of the predicted probability for the ground truth class: $-\log(p_c(x_i))$ if $y_{i,c} = 1$. This appears now as a straightforward generalization of the two-class case.

Its derivation from the idea of maximum likelihood is similar to the two class case. One starts again from the principle of max likelihood, applies conditional independence to obtain a product of probabilities (last lecture).

We had for the two class case

$$P(Y = y_i|X = x_i) = h(x_i)^{y_i}(1 - h(x_i))^{1-y_i}, y_i \in \{0, 1\}$$

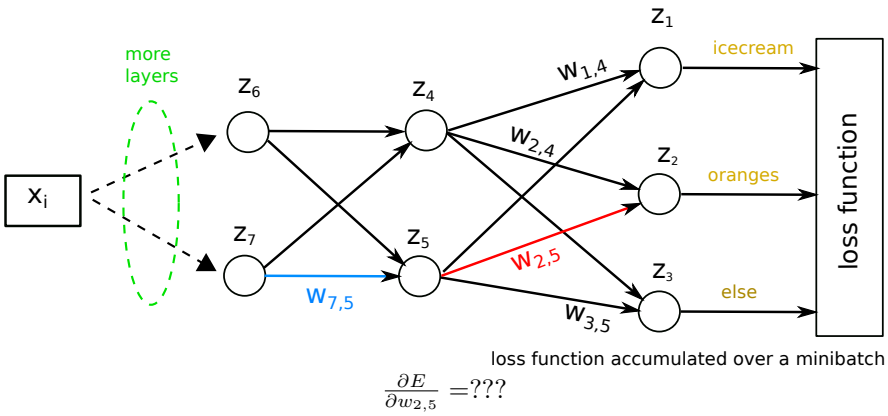
Now we have for C classes

$$\begin{aligned} P(Y = y_i|X = x_i) &= \Big\{ p_c(x_i) \quad \text{if } y_{i,c} = +1 \\ &= p_1(x_i)^{y_{i,1}} p_2(x_i)^{y_{i,2}} p_3(x_i)^{y_{i,3}} \cdot \ldots \cdot p_C(x_i)^{y_{i,C}} \\ &= \prod_{c=1}^{C} p_c(x_i)^{y_{i,c}} \end{aligned}$$

Taking $-\log(\cdot)$ of this yields:

$$-\log(\prod_{c=1}^{C} p_c(x_i)^{y_{i,c}}) = \sum_{c=1}^{C} -y_{i,c} \log(p_c(x_i))$$

which is shown above.



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

Once we compute the gradient with respect to parameters, we can use the gradient descent steps to find weight parameters of neural networks. This applies also to neural networks with many layers, not only for logistic regression.

Remarks:

- the softmax is usually not computed directly as $\frac{e^{z_c}}{\sum_{c'} e^{z_{c'}}}$ - the reason is that e to the power a large number can easily become numerically unstable. Instead one uses

$$\begin{aligned} m &= \max_c z_c \\ \text{softmax} &= \frac{e^{z_c}}{\sum_{c'} e^{z_{c'}}} = \frac{e^{z_c - m}}{\sum_{c'} e^{z_{c'} - m}} \end{aligned}$$

  where all terms are now smaller than 1. Trick: here one multiplied simply by e^{-m}

- for outputs coming directly from fully connected layers, many toolboxes have functions to compute the loss directly over the real-valued outputs in $(-\infty, \infty)$ without the need to explicitly compute softmax probabilities. That uses more tricks to compute the log softmax nicely.