Deep Learning

(Multi-Layer Perceptrons with Many Layers)

Deep Learning: Motivation

- Reminder: **Universal Approximation Theorem**
  Any continous function on an arbitrary compact subspace of $\mathbb{R}^n$
  can be approximated arbitrarily well with a three-layer perceptron.
- This theorem is often cited as (allegedly!) meaning that
  - we can confine ourselves to multi-layer perceptrons with only one hidden layer,
  - there is no real need to look at multi-layer perceptrons with more hidden layers.
- **However:** The theorem says nothing about the number of hidden neurons
  that may be needed to achieve a desired approximation accuracy.
- Depending on the function to approximate,
  a very large number of neurons may be necessary.
- Allowing for more hidden layers may enable us
  to achieve the same approximation quality
  with a significantly lower number of neurons.

Deep Learning: Motivation

- Very simple and commonly used example is the **n-bit parity function**:
  Output is 1 if an even number of inputs is 1; output is 0 otherwise.
- Can easily be represented by a multi-layer perceptron with only one hidden layer.
  (See reminder of TLU algorithm on next slide; adapt to MLPs.)
- However, the solution has $\mathbf{2^{n-1}}$ **hidden neurons**:
  disjunctive normal form is a disjunction of 2^{n-1} conjunctions,
  which represent the 2^{n-1} input combinations with an even number of set bits.
- Number of hidden neurons **grows exponentially** with the number of inputs.
- However, if more hidden layers are admissible, **linear growth** is possible:
  - Start with a *biimplication* of two inputs.
  - Continue with a chain of *exclusive or*s, each of which adds another input.
  - Such a network needs $n + 3(n-1) = 4n - 3$ neurons in total
    (n input neurons, $\mathbf{3(n-1)-1}$ **hidden neurons**, 1 output neuron)

Reminder: Representing Arbitrary Boolean Functions

Algorithm: Let $y = f(x_1, \ldots, x_n)$ be a Boolean function of n variables.

(i) Represent the given function $f(x_1, \ldots, x_n)$ in disjunctive normal form. That is, determine where all K_j are conjunctions of n literals, that is, $K_j = l_{j1} \wedge \ldots \wedge l_{jn}$ with $l_{ji} = x_i$ (positive literal) or $l_{ji} = \neg x_i$ (negative literal).

(ii) Create a neuron for each conjunction K_j of the disjunctive normal form (having n inputs — one input for each variable), where

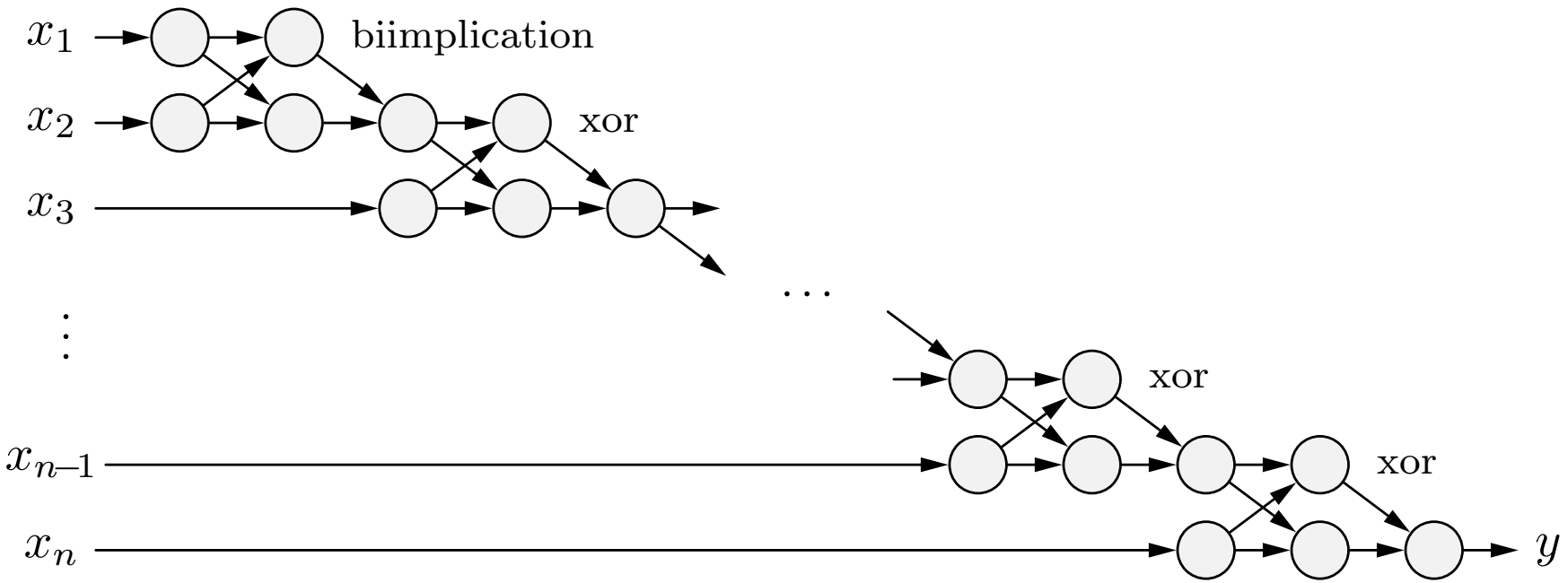
$$w_{ji} = \begin{cases} 2, & \text{if } l_{ji} = x_i, \\ -2, & \text{if } l_{ji} = \neg x_i, \end{cases} \qquad \text{and} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{ji}.$$

(iii) Create an output neuron (having m inputs — one input for each neuron that was created in step (ii)), where

$$w_{(n+1)k} = 2, \quad k = 1, \ldots, m, \qquad \text{and} \qquad \theta_{n+1} = 1.$$

Remark: weights are set to ± 2 instead of ± 1 in order to ensure integer thresholds.

Deep Learning: n-bit Parity Function



- Implementation of the n-bit parity function with a chain of one *biimplication* and $n-2$ *exclusive or* sub-networks.
- Note that the structure is not strictly layered, but could be completed.
  If this is done, the number of neurons increases to $n(n+1)-1$.

Deep Learning: Motivation

- Similar to the situation w.r.t. linearly separable functions:
  - Only few n-ary Boolean functions are linearly separable (see next slide).
  - Only few n-ary Boolean functions need few hidden neurons in a single layer.
- An n-ary Boolean function has k_0 input combinations with an output of 0 and k_1 input combinations with an output of 1 (clearly $k_0 + k_1 = 2^n$).
- We need at most $2^{\min\{k_0,k_1\}}$ hidden neurons if we choose disjunctive normal form for $k_1 \leq k_0$ and conjunctive normal form for $k_1 > k_0$.
- As there are $\binom{2^n}{k_0}$ possible functions with k_0 input combinations mapped to 0 and k_1 mapped to 1, many functions require a substantial number of hidden neurons.
- Although the number of neurons may be reduced with minimization methods like the Quine–McCluskey algorithm [Quine 1952, 1955; McCluskey 1956], the fundamental problem remains.

Reminder: Limitations of Threshold Logic Units

Total number and number of linearly separable Boolean functions
(On-Line Encyclopedia of Integer Sequences, `oeis.org`, A001146 and A000609):

inputs	Boolean functions	linearly separable functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	4,294,967,296	94,572
6	18,446,744,073,709,551,616	15,028,134
n	$2^{(2^n)}$	no general formula known

- For many inputs a threshold logic unit can compute almost no functions.
- Networks of threshold logic units are needed to overcome the limitations.

Deep Learning: Motivation

- In practice the problem is mitigated considerably by the simple fact that **training data sets are necessarily limited in size**.
- Complete training data for an n-ary Boolean function has 2^n training examples. Data sets for practical problems usually contain much fewer sample cases.

  This leads to many input configurations for which no desired output is given; freedom to assign outputs to them allows for a simpler representation.
- Nevertheless, using more than one hidden layer promises in many cases to reduce the number of needed neurons.
- This is the focus of the area of **deep learning**.
  (**Depth** means the length of the longest path in the network graph.)
- For multilayer perceptrons (longest path: number of hidden layers plus one), deep learning starts with more than one hidden layer.
- For 10 or more hidden layers, one sometimes speaks of **very deep learning**.

Deep Learning: Main Problems

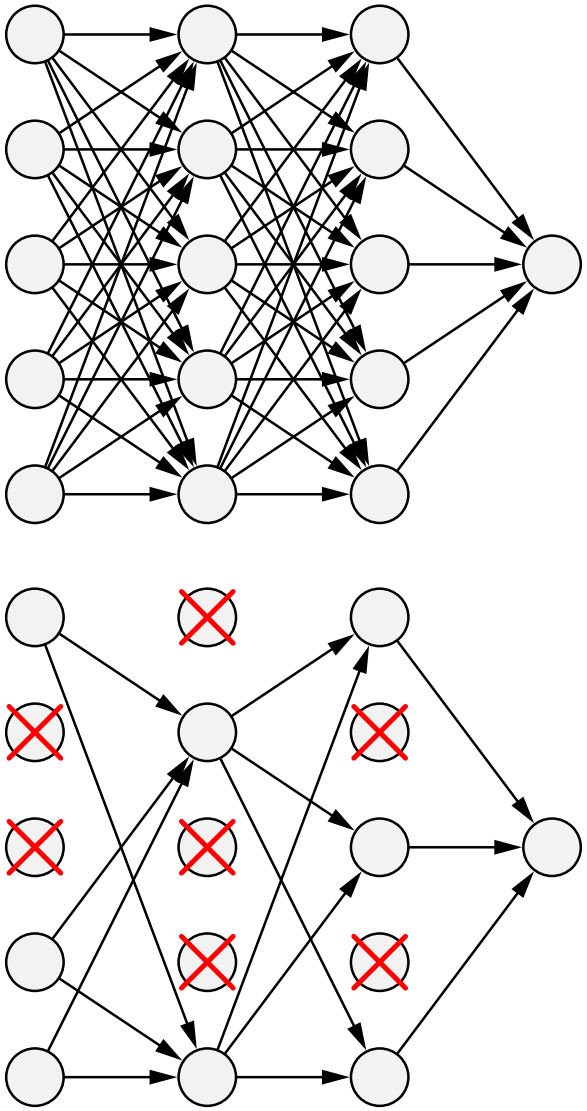
- Deep learning multilayer perceptrons suffer from two main problems:

  overfitting and **vanishing gradient**.

- Overfitting results mainly from the increased number of adaptable parameters.
- **Weight decay** prevents large weights and thus an overly precise adaptation.
- **Sparsity constraints** help to avoid overfitting:
  - there is only a restricted number of neurons in the hidden layers or
  - only few of the neurons in the hidden layers should be active (on average).

    May be achieved by adding a regularization term to the error function (compares the observed number of active neurons with desired number and pushes adaptations into a direction that tries to match these numbers).
- Furthermore, a training method called **dropout training** may be applied: some units are randomly omitted from the input/hidden layers during training.

Deep Learning: Dropout



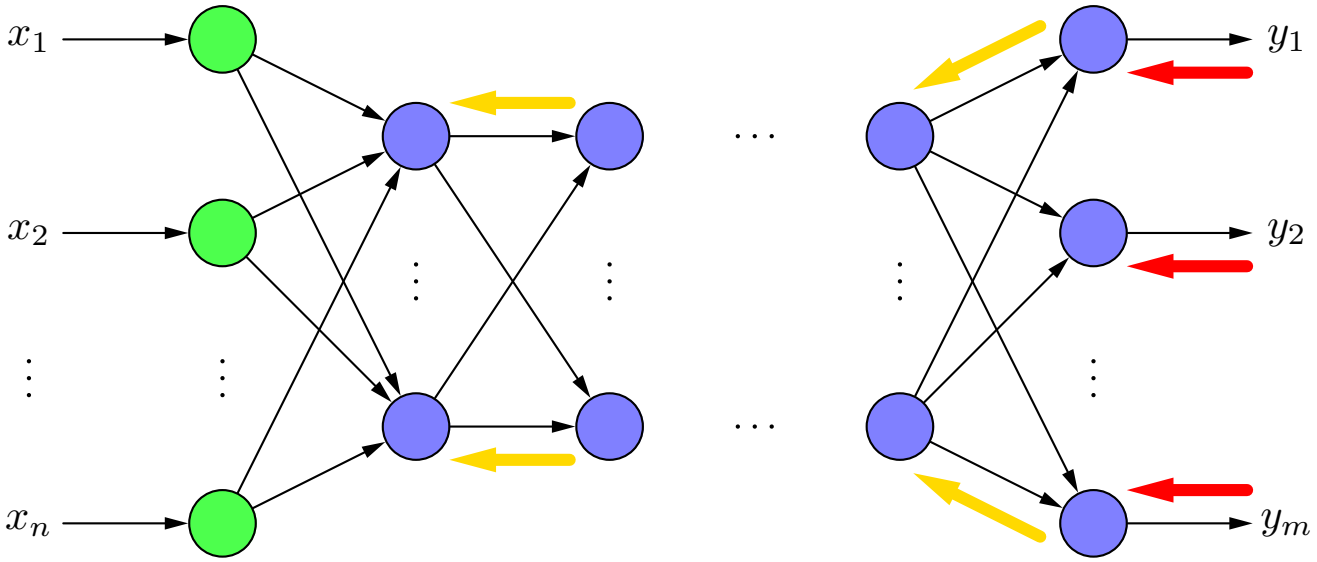
- Desired characteristic:
  Robustness against neuron failures.
- Approach during training:
  - Use only $p\%$ of the neurons
    (e.g. $p = 50$: half of the neurons).
  - Choose dropout neurons randomly.
- Approach during execution:
  - Use all of the neurons.
  - Multiply all weights by $p\%$.
- Result of this approach:
  - More robust representation.
  - Better generalization.

Reminder: Cookbook Recipe for Error Backpropagation

$\forall u \in U_{\text{in}}$:
$\text{out}_u^{(l)} = \text{ext}_u^{(l)}$

forward propagation:

$\forall u \in U_{\text{hidden}} \cup U_{\text{out}}$:
$\text{out}_u^{(l)} = \left(1 + \exp\left(-\sum_{p \in \text{pred}(u)} w_{up}\, \text{out}_p^{(l)}\right)\right)^{-1}$



- logistic activation function
- implicit bias value

backward propagation:

$\forall u \in U_{\text{hidden}}$:
$\delta_u^{(l)} = \left(\sum_{s \in \text{succ}(u)} \delta_s^{(l)} w_{su}\right) \lambda_u^{(l)}$

error factor:

$\forall u \in U_{\text{out}}$:
$\delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \lambda_u^{(l)}$

activation derivative:

$\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$

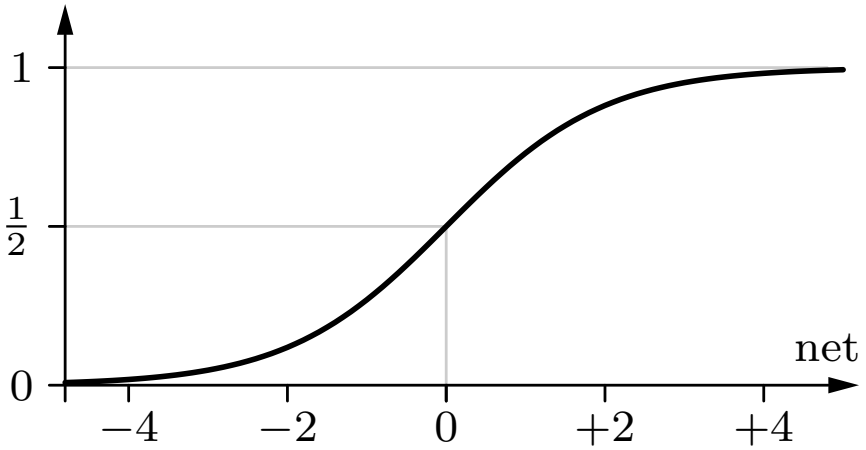
weight change:

$\Delta w_{up}^{(l)} = \eta\, \delta_u^{(l)}\, \text{out}_p^{(l)}$

Deep Learning: Vanishing Gradient

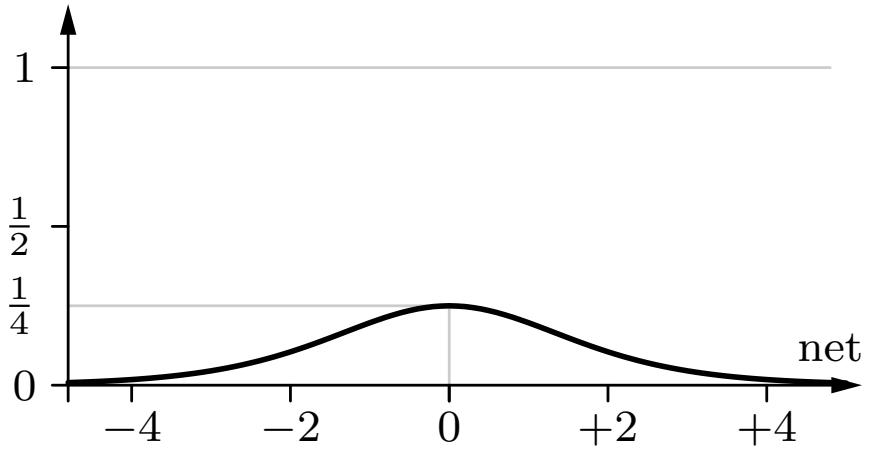
logistic activation function:

$$f_{\text{act}}(\text{net}_u^{(l)}) = \frac{1}{1 + e^{-\text{net}_u^{(l)}}}$$



derivative of logistic function:

$$f'_{\text{act}}(\text{net}_u^{(l)}) = f_{\text{act}}(\text{net}_u^{(l)}) \cdot (1 - f_{\text{act}}(\text{net}_u^{(l)}))$$



- If a logistic activation function is used (shown on left), the weight changes are proportional to $\lambda_u^{(l)} = \text{out}_u^{(l)} \left(1 - \text{out}_u^{(l)}\right)$ (shown on right; see recipe).
- This factor is also propagated back, but cannot be larger than $\frac{1}{4}$ (see right).

$\Rightarrow$ The gradient tends to vanish if many layers are backpropagated through. Learning in the early hidden layers can become very slow [Hochreiter 1991].

Deep Learning: Vanishing Gradient

- In principle, a small gradient may be counteracted by a large weight.

$$\delta_u^{(l)} = \Big(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \Big) \lambda_u^{(l)}.$$

- However, usually a large weight, since it also enters the activation function, drives the activation function to its **saturation regions**.

- Thus, (the absolute value of) the derivative factor is usually the smaller, the larger (the absolute value of) the weights.

- Furthermore, the connection weights are commonly initialized to a random value in the range from -1 to $+1$.

  $\Rightarrow$ **Initial training steps are particularly affected:**
  both the gradient as well as the weights provide a factor less than 1.

- Theoretically, there can be exceptions to this description.
  However, they are rare in practice and one usually observes a vanishing gradient.

Deep Learning: Vanishing Gradient

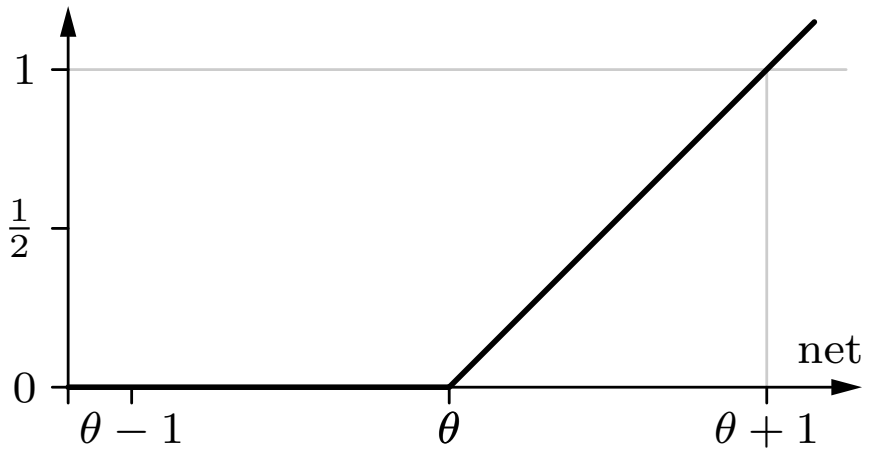
Alternative way to understand the vanishing gradient effect:

- The logistic activation function is a **contracting function**:
  for any two arguments x and y, $x \neq y$, we have $|f_{\text{act}}(x) - f_{\text{act}}(y)| < |x - y|$.
  (Obvious from the fact that its derivative is always < 1; actually $\leq \frac{1}{4}$.)
- If several logistic functions are chained, these contractions combine
  and yield an even stronger contraction of the input range.
- As a consequence, a rather large change of the input values
  will produce only a rather small change in the output values,
  and the more so, the more logistic functions are chained together.
- Therefore the function that maps the inputs of a multilayer perceptron to its outputs usually becomes the flatter the more layers the multilayer perceptron has.
- Consequently the gradient in the first hidden layer
  (were the inputs are processed) becomes the smaller.

Deep Learning: Different Activation Functions

rectified maximum/ramp function:

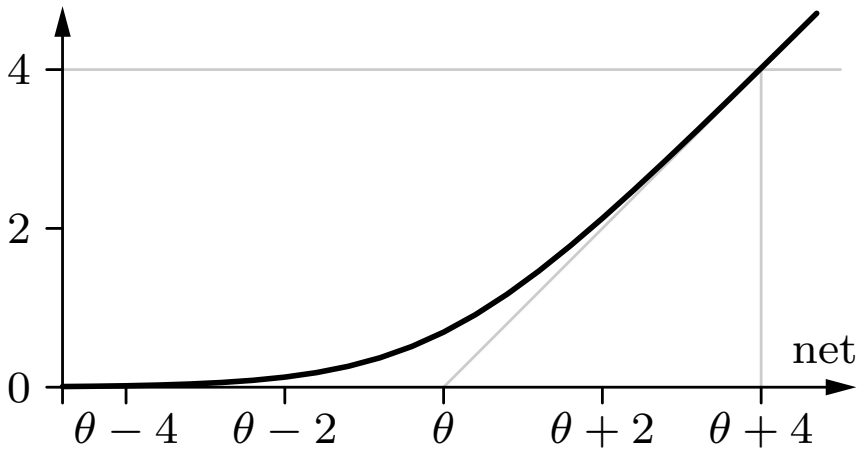
$f_{\text{act}}(\text{net}, \theta) = \max\{0, \text{net} - \theta\}$



softplus function:

Note the scale!

$f_{\text{act}}(\text{net}, \theta) = \ln(1 + e^{\text{net} - \theta})$

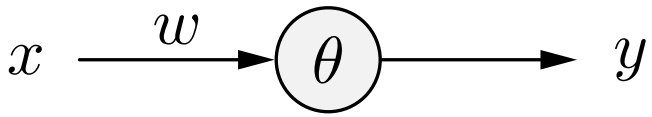


- The vanishing gradient problem may be battled with other activation functions.
- These activation functions yield so-called **rectified linear units (ReLUs)**.
- **Rectified maximum/ramp function**

  Advantages: simple computation, simple derivative, zeros simplify learning
  Disadvantages: no learning $\leq \theta$, rather inelegant, not continously differentiable

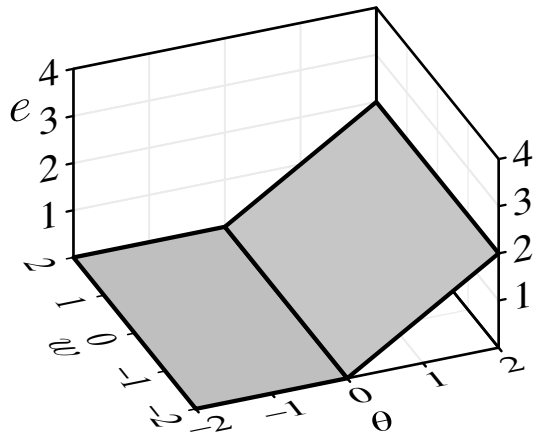
Reminder: Training a TLU for the Negation

Single input
threshold logic unit
for the negation $\neg x$.

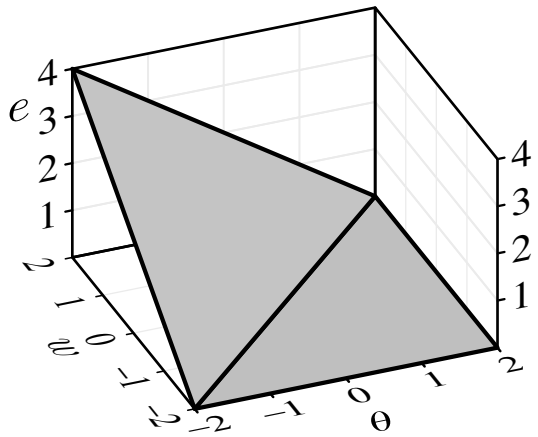


x	y
0	1
1	0

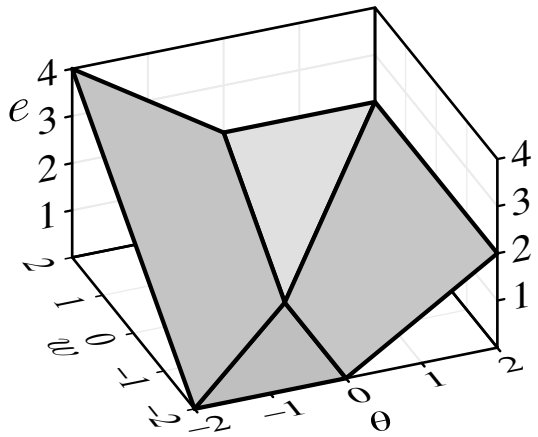
Modified output error as a function of weight and threshold:



error for $x = 0$



error for $x = 1$



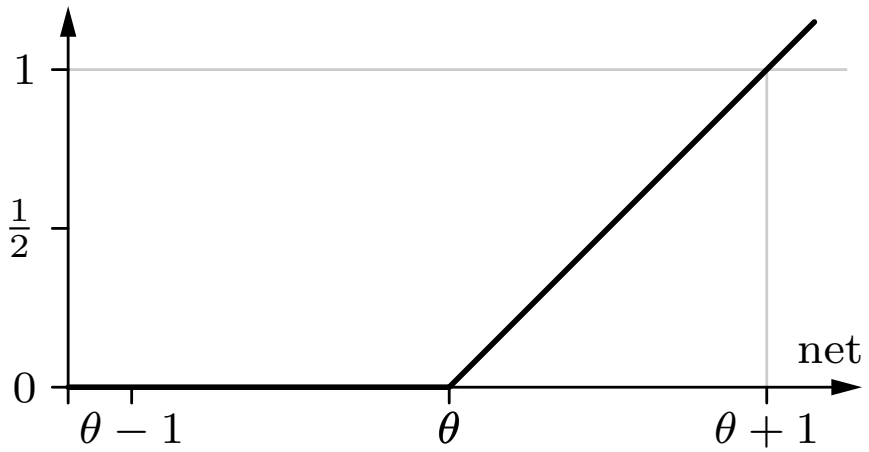
sum of errors

A rectified maximum/ramp function yields these errors directly.

Deep Learning: Different Activation Functions

rectified maximum/ramp function:

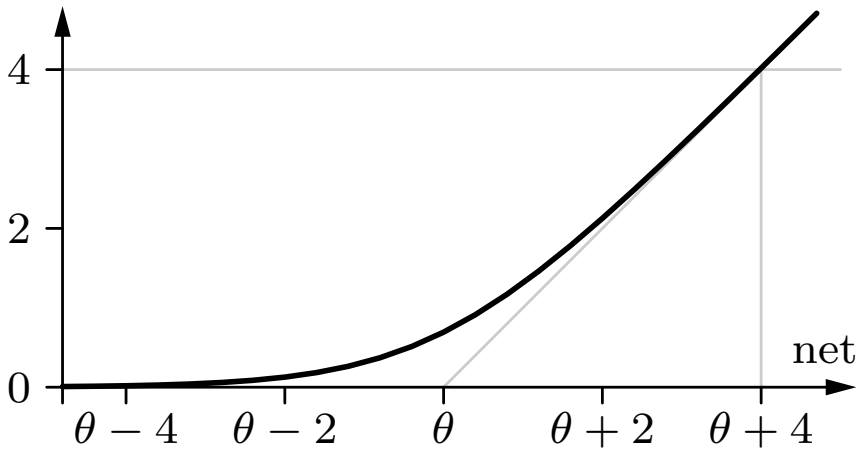
$f_{\text{act}}(\text{net}, \theta) = \max\{0, \text{net} - \theta\}$



softplus function:

Note the scale!

$f_{\text{act}}(\text{net}, \theta) = \ln(1 + e^{\text{net} - \theta})$

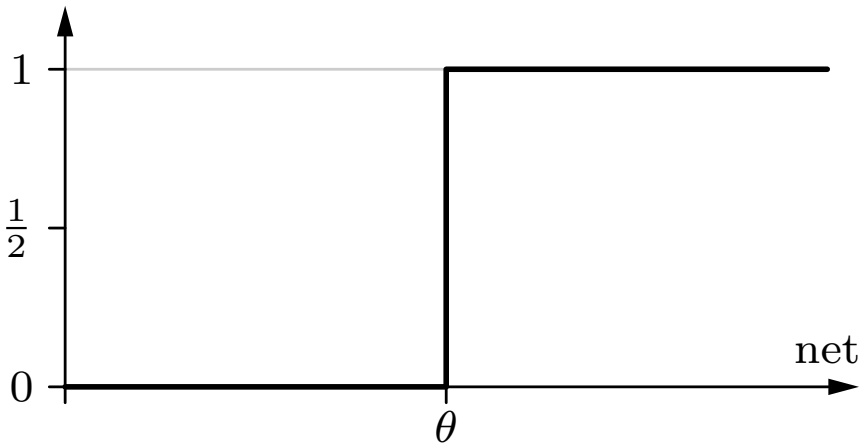


- **Softplus function**: continuously differentiable, but much more complex
- Alternatives:
  - **Leaky ReLU:** $f(x) = \begin{cases} x & \text{if } x > 0, \\ \nu x & \text{otherwise.} \end{cases}$ (adds learning $\leq \theta$; $\nu \approx 0.01$)
  - **Noisy ReLU:** $f(x) = \max\{0, x + \mathcal{N}(0, \sigma(x))\}$ (adds Gaussian noise)

Deep Learning: Different Activation Functions

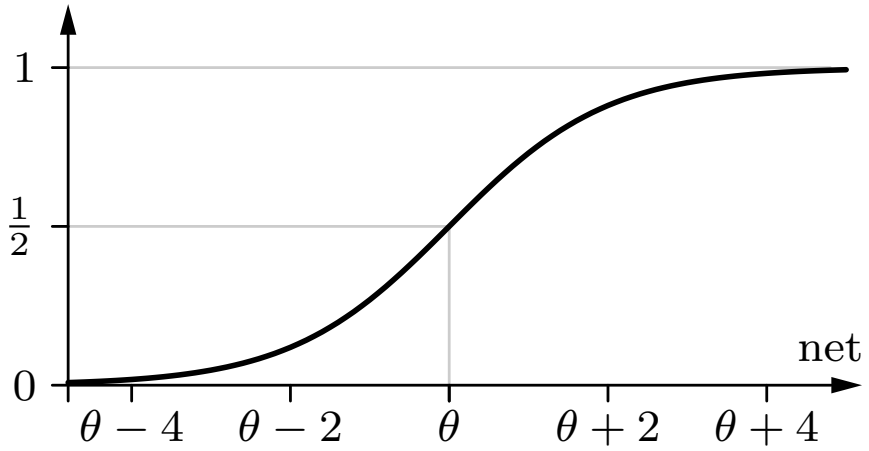
derivative of ramp function:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{if net} \geq \theta, \\ 0, & \text{otherwise.} \end{cases}$$



derivative of softplus function:

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$



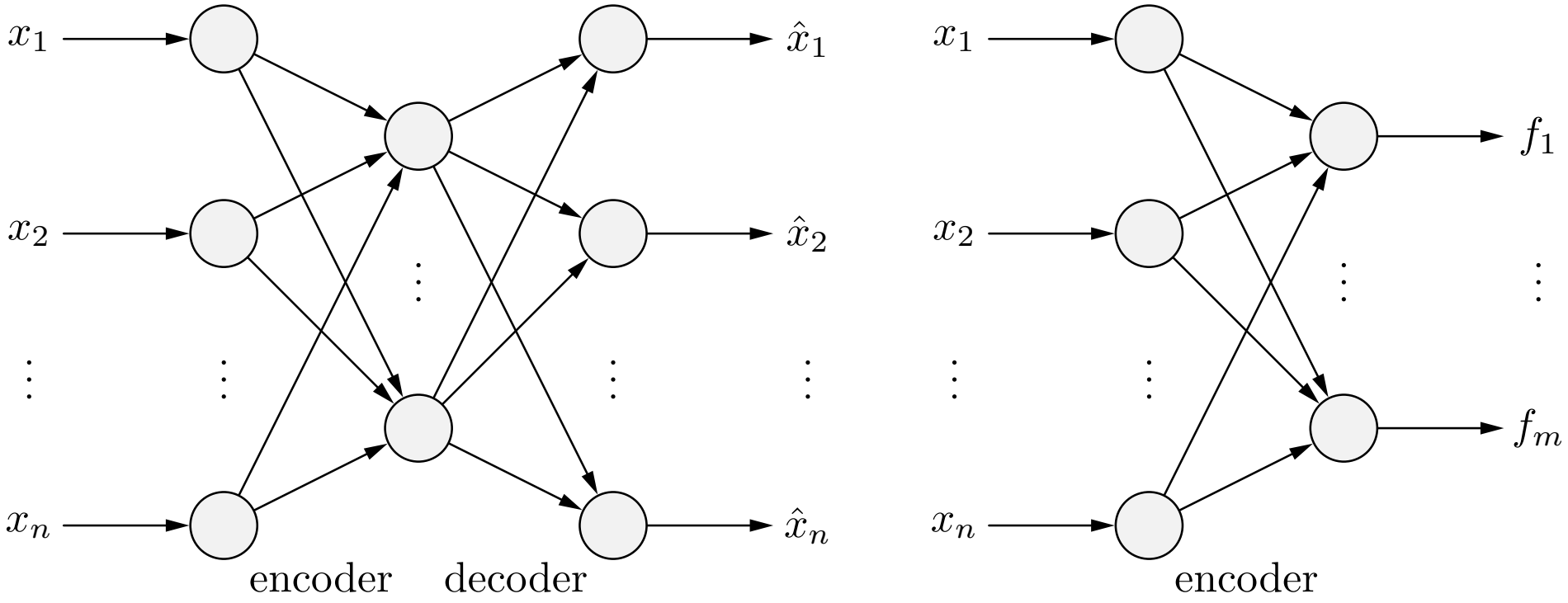
- The derivative of the rectified maximum/ramp function is the **step function**.
- The derivative of the softplus function is the **logistic function**.
- Factor resulting from derivative of activation function can be much larger.
  $\Rightarrow$ Larger gradient in early layers of the network; training is (much) faster.
- What also helps: **faster hardware**, **implementations on GPUs** etc.

Deep Learning: Auto-Encoders

- Reminder: Networks of threshold logic units cannot be trained, because
  - there are no desired values for the neurons of the first layer(s),
  - the problem can usually be solved with several different functions computed by the neurons of the first layer(s) (non-unique solution).
- Although multi-layer percentrons, even with many hidden layers, can be trained, the same problems still affect the effectiveness and efficiency of training.
- Alternative: **Build network layer by layer, train only newly added layer in each step.**

  Popular: Build the network as **stacked auto-encoders**.
- An **auto-encoder** is a 3-layer perceptron that maps its inputs to approximations of these inputs.
  - The hidden layer forms an encoder into some form of internal representation.
  - The output layer forms a decoder that (approximately) reconstructs the inputs.

Deep Learning: Auto-Encoders



An auto-encoder/decoder (left), of which only the encoder part (right) is later used.

The x_i are the given inputs, the $\hat{x}_i$ are the reconstructed inputs, and the f_i are the constructed features; the error is $e = \sum_{i=1}^{n} (\hat{x}_i - x_i)^2$.

Training is conducted with error backpropagation.

Deep Learning: Auto-Encoders

- Rationale of training an auto-encoder:
  hidden layer is expected to construct features.
- Hope: Features capture the information contained in the input in a compressed form (encoder), so that the input can be well reconstructed from it (decoder).
- Note: this implicitly assumes that features that are well suited to represent the inputs in a compressed way are also useful to predict some desired output.
  Experience shows that this assumption is often justified.
- Main problems:
  - how many units should be chosen for the hidden layer?
  - how should this layer be treated during training?
- If there are as many (or even more) hidden units as there are inputs, it is likely that it will merely pass through its inputs to the output layer.
- The most straightforward solution are **sparse auto-encoders**:
  There should be (considerably) fewer hidden neurons than there are inputs.

Deep Learning: Sparse and Denoising Auto-Encoders

- Few hidden neurons force the auto-encoder to learn relevant features (since it is not possible to simply pass through the inputs).
- How many hidden neurons are a good choice?
  Note that **cross-validation** not necessarily a good approach!
- Alternative to few hidden neurons: **sparse activation scheme**
  The number of active neurons in the hidden layer is restricted to a small number.

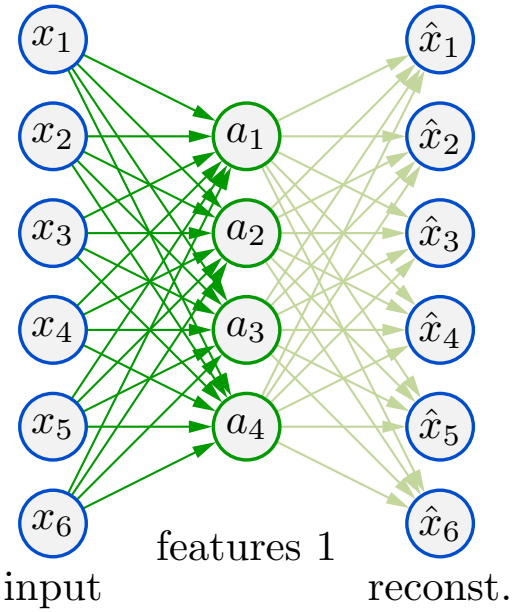
  May be enforced by either adding a regularization term to the error function that punishes a larger number of active hidden neurons or by explicitly deactivating all but a few neurons with the highest activations.
- A third approach is to add noise (that is, random variations) to the input (but not to the copy of the input used to evaluate the reconstruction error): **denoising auto-encoders**.

  (The auto-encoder to be trained is expected to map the input with noise to (a copy of) the input without noise, thus preventing simple passing through.)

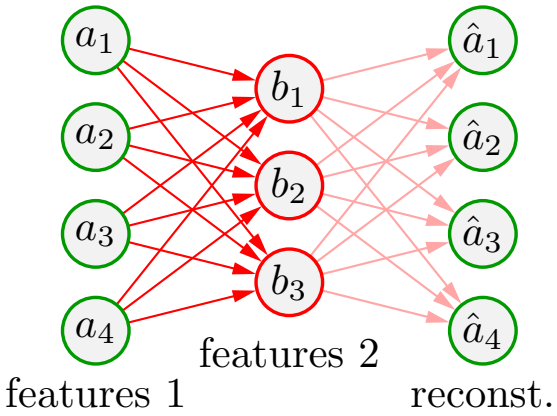
Deep Learning: Stacked Auto-Encoders

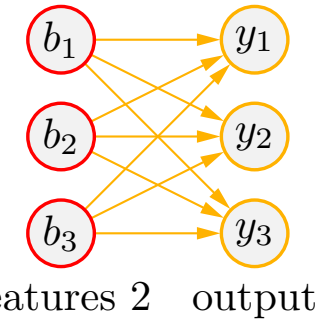
- A popular way to initialize/pre-train a stacked auto-encoder is a **greedy layer-wise training approach**.
- In the first step, a (sparse) auto-encoder is trained for the raw input features. The hidden layer constructs **primary features** useful for reconstruction.
- A primary feature data set is obtained by propagating the raw input features up to the hidden layer and recording the hidden neuron activations.
- In the seocnd step, a (sparse) auto-encoder is trained for the obtained feature data set. The hidden layer constructs **secondary features** useful for reconstruction.
- A secondary feature data set is obtained by propagating the primary features up to the hidden layer and recording the hidden neuron activations.
- This **process is repeated** as many times as hidden layers are desired.
- Finally the encoder parts of the trained **auto-encoders are stacked** and the resulting network is **fine-tuned with error backpropagation**.

Deep Learning: Stacked Auto-Encoders

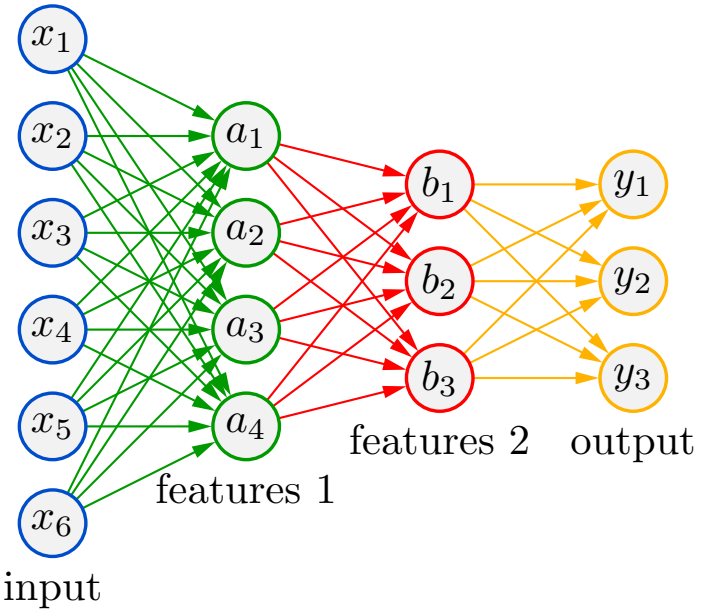


Layer-wise training of auto-encoders:





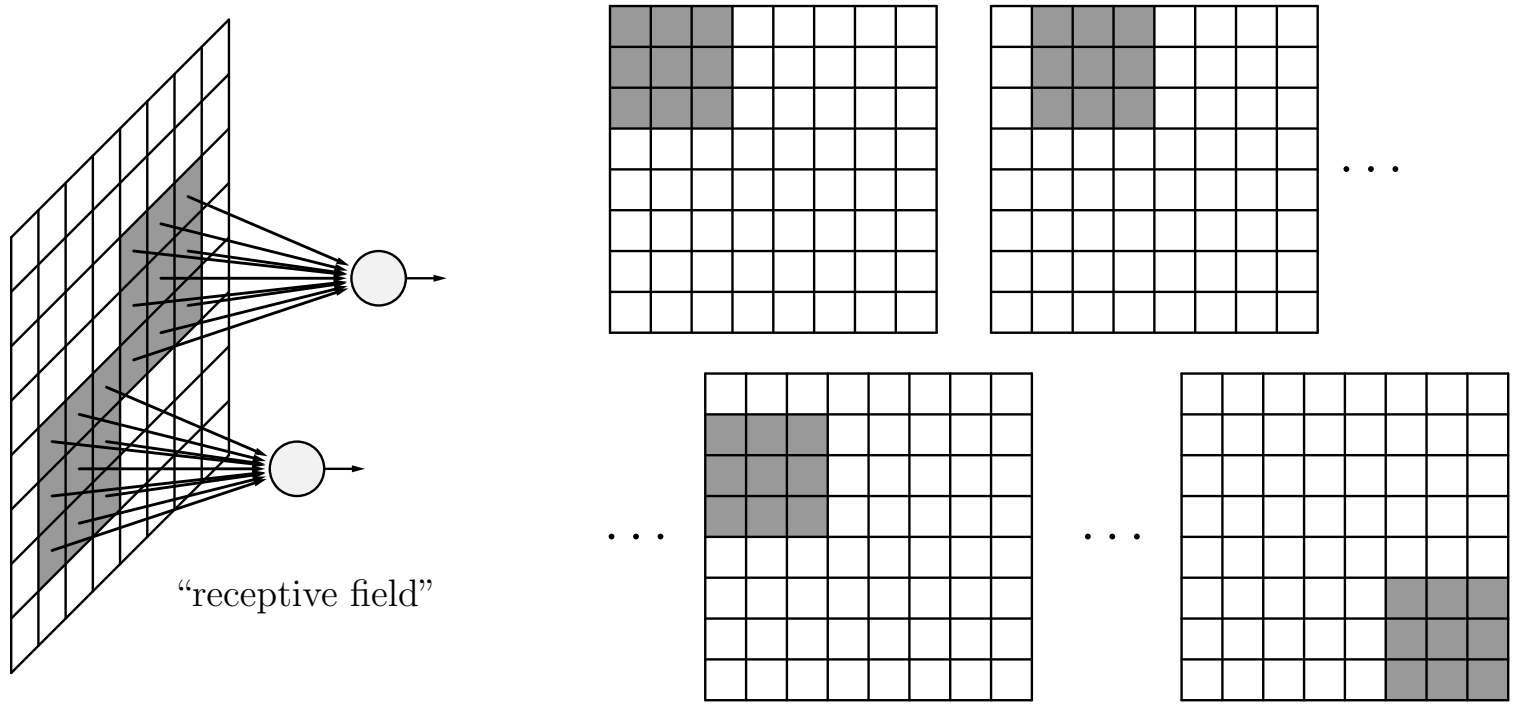
1. Train auto-encoder for the raw input;
   this yields a primary feature set.
2. Train auto-encoder for the primary features;
   this yields a secondary feature set.
3. A classifier/predictor for the output
   is trained from the secondary feature set.
4. The resulting networks are stacked.



Deep Learning: Convolutional Neural Networks (CNNs)

- Multilayer perceptrons with several hidden layers built in the way described have been applied very successfully for handwritten digit recognition.
- In such an application it is assumed, though, that the handwriting has already been preprocessed in order to separate the digits (or, in other cases, the letters) from each other.
- However, one would like to use similar networks also for more general applications, for example, recognizing whole lines of handwriting or analyzing photos to identify their parts as sky, landscape, house, pavement, tree, human being etc.
- For such applications it is advantageous that the features constructed in hidden layers are not localized to a specific part of the image.
- Special form of deep learning multi-layer perceptron: **convolutional neural network**.
- Inspired by the human retina, where sensory neurons have a **receptive field**, that is, a limited region in which they respond to a (visual) stimulus.

Reminder: Receptive Field, Convolution



- Each neuron of the (first) hidden layer is connected to a small number of input neurons that refer to a contiguous region of the input image (left).
- Connection weights are shared, same network is evaluated at different locations. The input field is "moved" step by step over the whole image (right).
- Equivalent to a **convolution** with a small size kernel.

Deep Learning: AlphaGo

- **AlphaGo** is a computer program developed by Alphabet Inc.'s Google DeepMind to play the board game Go.
- It uses a combination of machine learning and tree search techniques, combined with extensive training, both from human and computer play.
- AlphaGo uses Monte Carlo tree search, guided by a "value network" and a "policy network", both of which are implemented using **deep neural networks**.
- A limited amount of game-specific feature detection is applied to the input before it is sent to the neural networks.
- The neural networks were bootstrapped from human gameplay experience. Later AlphaGo was set up to play against other instances of itself, using reinforcement learning to improve its play.

source: Wikipedia

Deep Learning: AlphaGo

Match against **Fan Hui**
(Elo 3016 on 01-01-2016,
#512 world ranking list),
best European player
at the time of the match

AlphaGo wins 5 : 0

Match against **Lee Sedol**
(Elo 3546 on 01-01-2016,
#1 world ranking list 2007–2010,
#3 at the time of the match)

AlphaGo wins 4 : 1

`www.goratings.org`

AlphaGo	threads	CPUs	GPUs	Elo
Async.	1	48	8	2203
Async.	2	48	8	2393
Async.	4	48	8	2564
Async.	8	48	8	2665
Async.	16	48	8	2778
Async.	32	48	8	2867
Async.	40	48	1	2181
Async.	40	48	2	2738
Async.	40	48	4	2850
Async.	40	48	8	2890
Distrib.	12	428	64	2937
Distrib.	24	764	112	3079
Distrib.	40	1202	176	3140
Distrib.	64	1920	280	3168