

AI

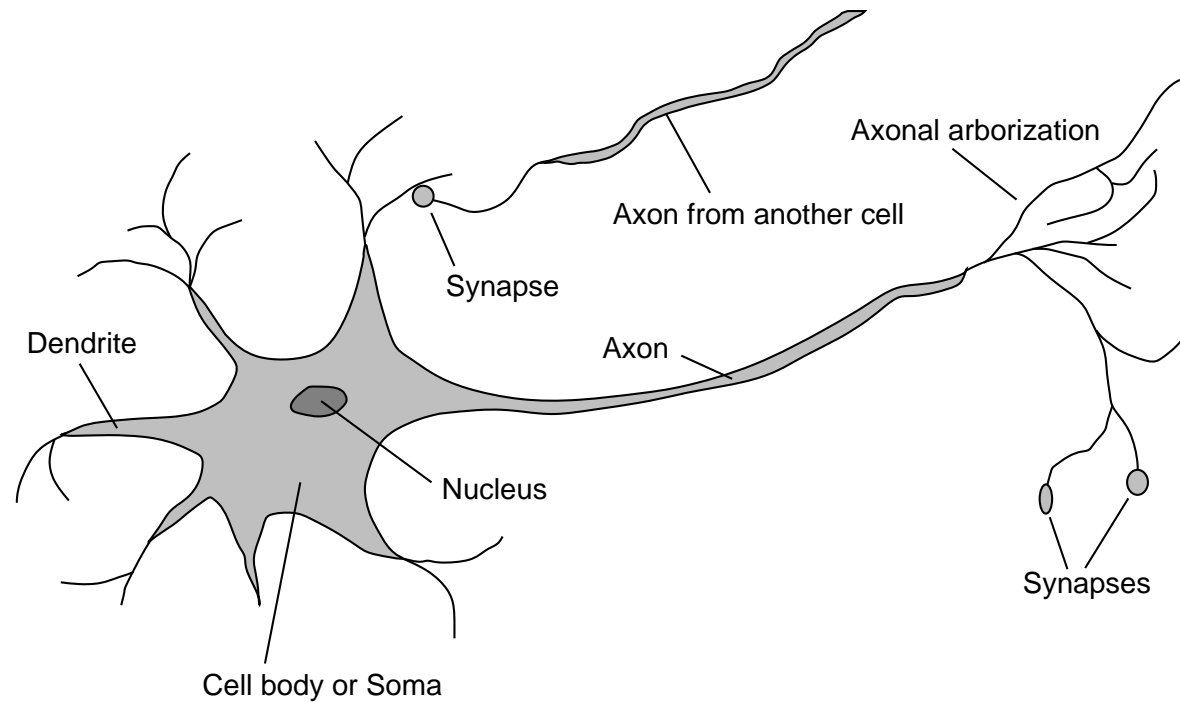
Kunstmatige Intelligentie (AI)

Hoofdstuk 18.7 van Russell/Norvig = [RN]
Neurale Netwerken (NN's)

voorjaar 2011 — College 7, 22 maart 2011

www.liacs.nl/home/kosters/AI/

De menselijke hersenen bestaan uit 10^{11} **neuronen** (grootte ≈ 0.1 mm; meer dan 20 types), die onderling met **axonen** (lengte ≈ 1 cm) en **synapsen** verbonden zijn:

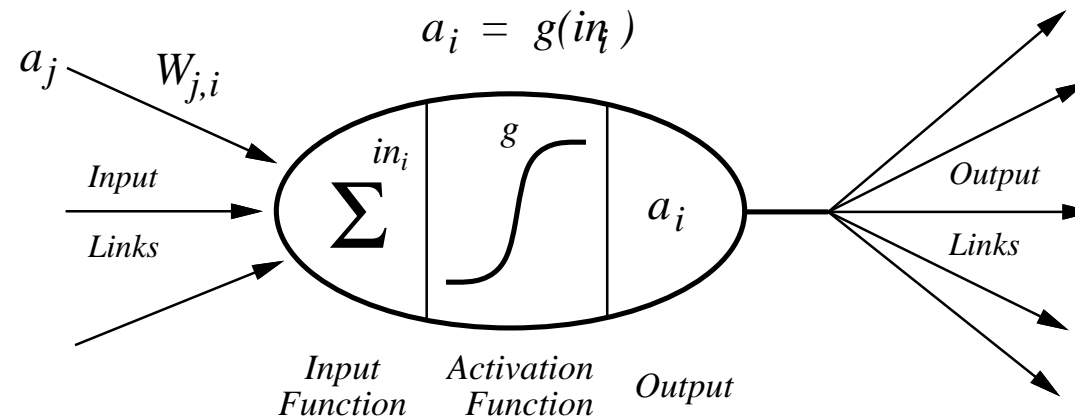


Signalen worden doorgegeven via een nogal gecompliceerde electro-chemische reactie.

Als de elektrische potentiaal van het cellichaam een zekere drempelwaarde haalt, wordt een puls/actie-potentiaal/spike-train op het axon gezet.

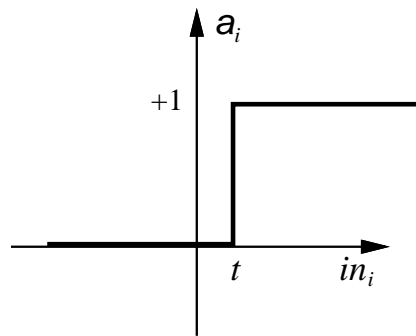
Er zijn excitatory (verhogen potentiaal) en inhibitory (verlagen potentiaal) synapsen.

We modelleren de neuronen door middel van eenheden (units) die we ook weer **neuronen** noemen:

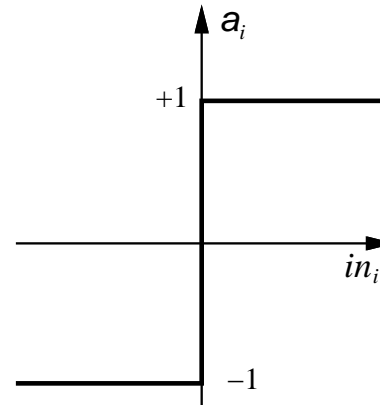


Er geldt: de input van neuron i is $in_i = \sum_j W_{j,i} a_j$ (de gewogen som van de inputs), waarbij de a_j 's de **activaties** van de inkomende verbindingen (inputs) zijn, en $W_{j,i}$ hun gewichten ($W_{j,i}$ weegt de verbinding tussen neuronen j en i). De activatie (output) van neuron i is $a_i = g(in_i)$, met g de **activatie-functie**.

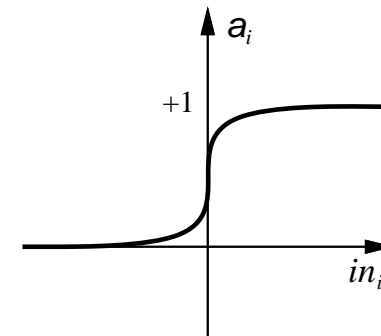
Veelgebruikte activatie-functies zijn:



(a) Step function



(b) Sign function



(c) Sigmoid function

$\text{step}_t(x) = 1$ als $x \geq t$; 0 als $x < t$ (drempelwaarde t)

$\text{sign}(x) = 1$ als $x \geq 0$; -1 als $x < 0$

$\text{sigmoid}(x) = 1/(1 + e^{-\beta x})$ (vaak kiest men $\beta = 1$)

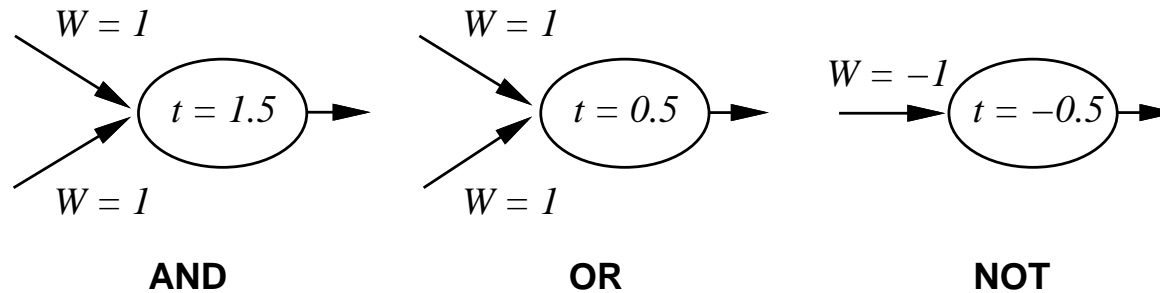
De drempelwaarde t binnen de neuronen kan “gesimuleerd” worden door een extra (constante) activatie -1 in een zogeheten **bias-knoop** 0 met gewicht $W_{0,i} = t$ op de verbinding van 0 naar i :

$$\text{step}_t \left(\sum_{j=1}^n W_{j,i} a_j \right) = \text{step}_0 \left(\sum_{j=0}^n W_{j,i} a_j \right)$$

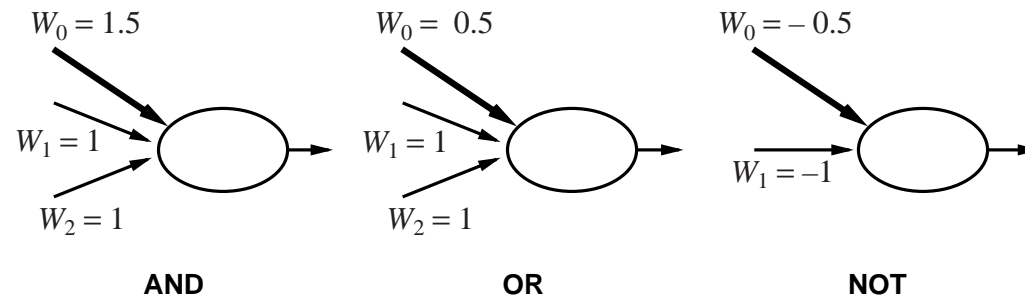
Zo worden drempelwaardes en gewichten uniform behandeld.

De functie step_0 heet ook wel de Heaviside-functie, en lijkt op de functie sign .

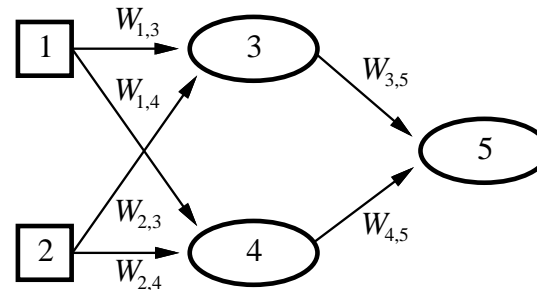
Alle Boolese functies kunnen gerepresenteerd worden door netwerken met geschikte neuronen:



En mét bias-knoppen:



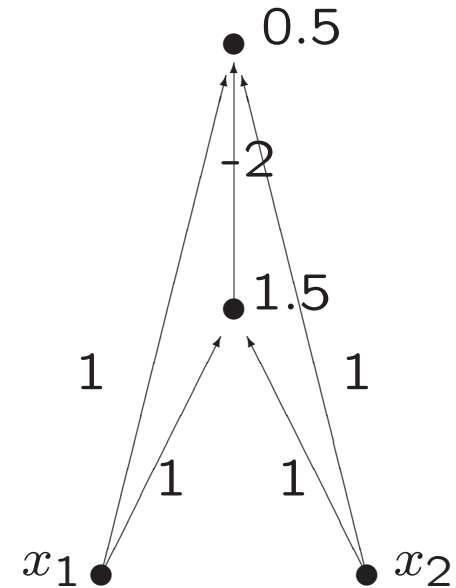
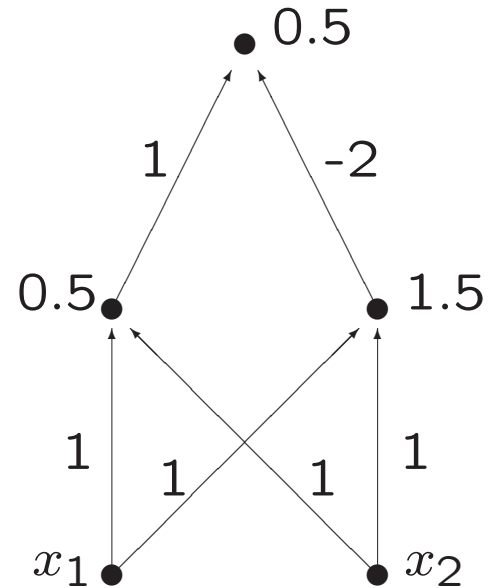
Een **feed-forward netwerk** heeft gerichte takken en geen cykels (in tegenstelling tot een **recurrent netwerk**). Vaak is zo'n netwerk in **lagen** georganiseerd:



Dit netwerk heeft twee input-knopen 1 en 2, twee **verborgen** (= hidden) knopen 3 en 4, en één uitvoer-knoop 5. Het representeert de volgende functie:

$$\begin{aligned} a_5 &= g(W_{3,5} a_3 + W_{4,5} a_4) \\ &= g(W_{3,5} g(W_{1,3} a_1 + W_{2,3} a_2) \\ &\quad + W_{4,5} g(W_{1,4} a_1 + W_{2,4} a_2)) \end{aligned}$$

Voorbeelden:



De drempels staan naast de knopen.

Het linker feed-forward netwerk representeert de XOR-functie, de verborgen units zijn in feite een OR en een AND.

Het rechter netwerk representeert dezelfde functie, maar is niet “conventioneel” (geen lagen).

Een feed-forward netwerk zonder verborgen neuronen heet een **perceptron**. Een **meerlaags** (= multi-layer) netwerk heeft één of meer verborgen lagen, en alle pijlen van laag ℓ gaan naar laag $(\ell + 1)$.

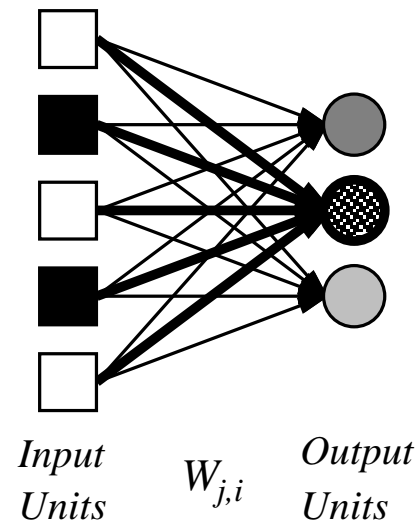
Met één (voldoende grote) laag met verborgen units kunt je elke continue functie benaderen, en met twee lagen zelfs elke discontinue functie (Cybenko).

De output van een netwerk hangt af van de parameters, de gewichten. Bij *te veel* parameters is er gevaar voor **overfitting**: het netwerk generaliseert dan niet goed.

Er zijn nog vele andere soorten netwerken, zoals

- Hopfield netwerken (associatief geheugen), met symmetrische gewichten $w_{i,j} = w_{j,i}$
- Boltzmann machines
- Kohonen's Self Organizing Maps (SOM's)
- Support Vector Machines (SVM's), kernel machines
- ...

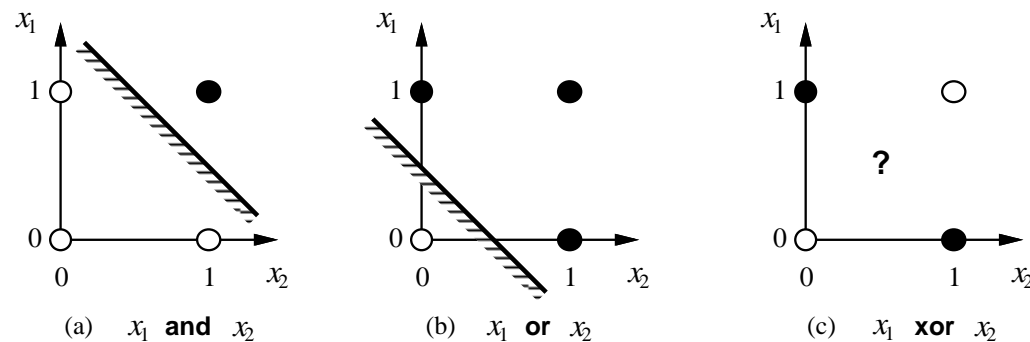
In een perceptron (geen verborgen units!) zijn de uitvoer-knoppen onafhankelijk:



Voor een enkele output-unit geldt dat de uitvoer 1 is indien $\sum_j W_j x_j = W_0 x_0 + W_1 x_1 + \dots + W_n x_n \geq 0$ en anders 0. Hierbij is (x_1, \dots, x_n) de invoer en $x_0 = -1$ de bias-knoop.

De **majority functie** (uitvoer 1 \Leftrightarrow meer dan de helft van de n inputs is 1) kan eenvoudig gemaakt worden: kies $W_0 = n/2$ en $W_j = 1$ voor $j = 1, 2, \dots, n$.

De vergelijking $-W_0 + W_1x_1 + \dots + W_nx_n \geq 0$ laat zien dat je precies Boolese functies kunt maken die **lineair te scheiden** zijn, de XOR-functie dus niet:



Gegeven genoeg trainings-voorbeelden, kan een perceptron elke Boolese lineair te scheiden functie leren. Een (hyper)vlak scheidt positieve en negatieve voorbeelden.

Rosenblatt's algoritme uit 1957/60 werkt als volgt:

$$W_j \leftarrow W_j + \alpha \cdot x_j \cdot \text{Error}$$

met $\text{Error} = \text{correcte uitvoer} - \text{net uitvoer}$ en $\alpha > 0$ de **leersnelheid** (= learning rate). De correcte uitvoer heet wel de **target**, het doel.

Als $\text{Error} = 1$ en $x_j = 1$, wordt W_j ietsje opgehoogd, in de hoop dat de net uitvoer hoger wordt.

We willen een perceptron x_1 AND x_2 leren, met $\alpha = 0.1$:

	x_0	x_1	x_2	W_0	W_1	W_2	uitvoer	target	Error
1	-1	1	1	-0.160	-0.606	-0.217	0.000	1.000	1.000
2	-1	1	0	-0.260	-0.506	-0.117	0.000	0.000	0.000
3	-1	0	1	-0.260	-0.506	-0.117	1.000	0.000	-1.000
4	-1	1	0	-0.160	-0.506	-0.217	0.000	0.000	0.000
5	-1	1	0	-0.160	-0.506	-0.217	0.000	0.000	0.000
6	-1	1	1	-0.160	-0.506	-0.217	0.000	1.000	1.000
7	-1	0	0	-0.260	-0.406	-0.117	1.000	0.000	-1.000
8	-1	0	0	-0.160	-0.406	-0.117	1.000	0.000	-1.000
...									
70	-1	1	0	0.140	0.194	0.183	1.000	0.000	-1.000
71	-1	1	1	0.240	0.094	0.183	1.000	1.000	0.000
...									

Probleem: wanneer stop je?

We kunnen in plaats van de discontinue stapfunctie ook een gladde sigmoïde g gebruiken in de knopen. Een vergelijkbaar leeralgoritme gaat dan als volgt.

Zij $E = \frac{1}{2}\text{Error}^2 = \frac{1}{2}\left(y - g\left(\sum_{j=0}^n W_j x_j\right)\right)^2$ met y de target. Met behulp van **gradient descent** bepalen we in welke richting de fout het snelst stijgt:

$$\frac{\partial E}{\partial W_j} = \text{Error} \cdot \frac{\partial \text{Error}}{\partial W_j} = -\text{Error} \cdot g'(\text{in}) \cdot x_j$$

met $\text{in} = \sum_{j=0}^n W_j x_j$. Dus leerregel (let op de extra $-$):

$$W_j \longleftarrow W_j + \alpha \cdot \text{Error} \cdot g'(\text{in}) \cdot x_j \quad .$$

Een kunstmatig voorbeeld: stel we proberen de uitvoer $y = 5$ te bereiken met de functie $W_1 + 3W_2$, en we zitten in $W_1 = W_2 = 1$; de gok is dus 4. De E is dan gelijk aan:

$$E = \frac{1}{2}(5 - (W_1 + 3W_2))^2 = \frac{1}{2}(5 - 4)^2 = 1/2$$

We rekenen uit

$$\frac{\partial E}{\partial W_1} = -(5 - (W_1 + 3W_2)) = -1$$

$$\frac{\partial E}{\partial W_2} = -3(5 - (W_1 + 3W_2)) = -3$$

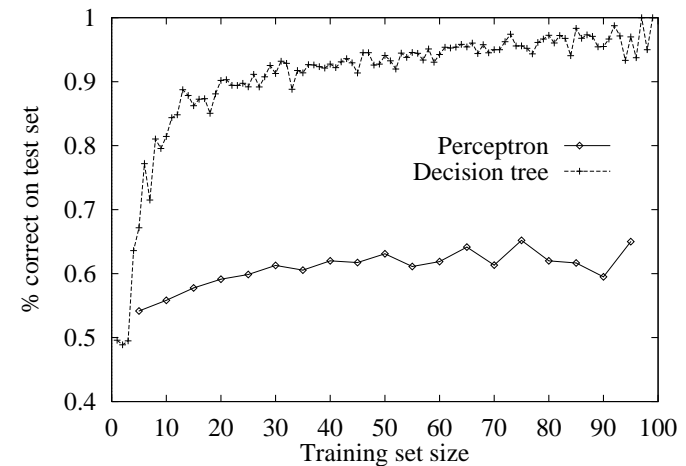
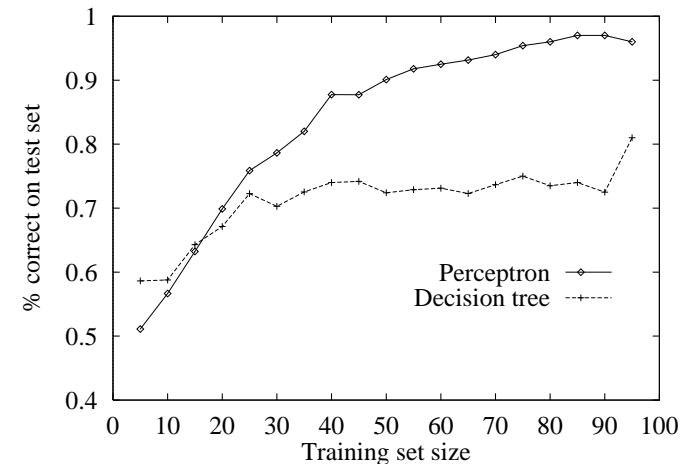
en passen aan (met kleine $\alpha > 0$):

$$W_1 \longleftarrow 1 + \alpha, \quad W_2 \longleftarrow 1 + 3\alpha$$

Dus W_2 wordt meer (omhoog) aangepast!

Voor een lineair te scheiden majority probleem (11 inputs) is het perceptron veel beter dan een techniek als beslissingsbomen (decision trees, zie Data Mining en ook College 11 over Leren):

Maar voor een meer complex probleem is het omgekeerd:



Er zijn allerlei keuzes om invoer en uitvoer te coderen. Je kunt bijvoorbeeld **locaal coderen**: stel dat een variabele 3 waarden kan hebben: Geen, Gemiddeld en Veel; dit kun je dan in één knoop coderen als 0.0, 0.5 en 1.0 respectievelijk. Maar ook met drie aparte knopen, en dan als 1–0–0, 0–1–0 en 0–0–1 respectievelijk: **gedistribueerd coderen**.

Je kunt zelfs bij de foutmaat ervoor zorgen dat je waarden als 1–1–0.9 niet fijn vindt, en daar van weg trainen (**“softmax”**).

Hoe verloopt nu het leren bij Neurale Netwerken, het aanpassen van de gewichten, kortom: de **training**?

Je begint met een **random initialisatie** van de gewichten. Vervolgens biedt je één voor één voorbeelden aan uit een zogeheten **trainingsset**. Deze voorbeelden moeten in een willekeurige volgorde staan. Steeds geef je een invoer, en met behulp van de juiste uitvoer pas je volgens je trainings-algoritme de gewichten aan.

Je gaat net zolang door totdat (bijvoorbeeld) de fout op een apart gehouden **validatieset** niet meer daalt — of zelfs gaat stijgen (overtraining).

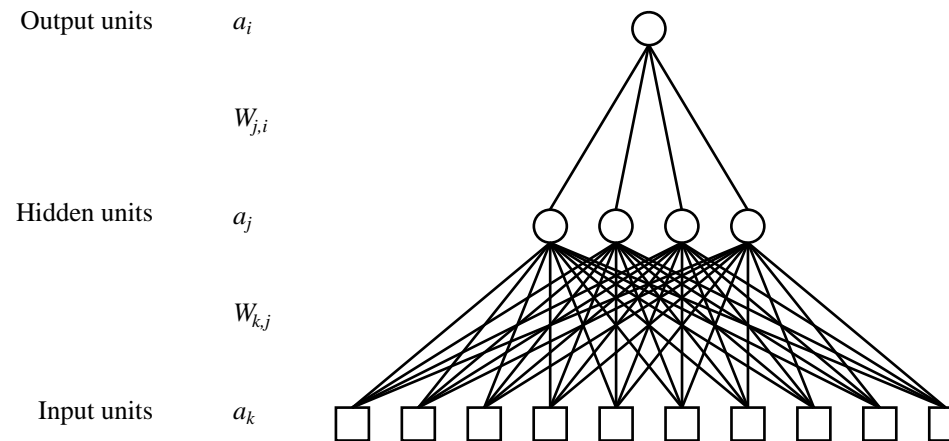
Je rapporteert tot slot over het gedrag op een (weer apart gehouden) **testset**.

Bij elk leer-algoritme kun je **cross-validation** gebruiken om overfitting tegen te gaan.

Bij “ k -fold cross-validation” (vaak $k = 5$ of $k = 10$) draai je k experimenten, waarbij je steeds een $1/k$ -de deel van de data apart zet om als testset te gebruiken — en de rest als trainingsset. De testset is dus steeds een ander random gekozen deel!

Als $k = n$ met n de grootte van de dataset, heet deze techniek wel “leave-one-out”. En dan heb je ook nog “ensemble leren”, AdaBoost, enzovoorts. Zie verder het college Data Mining.

We kijken nu naar een **meerlaags neuraal netwerk**. Meestal zijn alle lagen onderling *volledig* verbonden.



De notatie is ietwat dubbelzinnig: zit $W_{1,2}$ op twee plaatsen? Je kunt of knopen doornummeren (zie elders), of meerdere W 's hanteren, of hopen dat er geen misverstand ontstaat. We gebruiken index i voor de uitvoerlaag, j voor de verborgen laag en k voor de invoerlaag. De a_k 's zijn de input(s) — wat we eerder x_k noemden.

Met

$$\begin{aligned} a_5 &= g(in_5) = g(W_{3,5} a_3 + W_{4,5} a_4) \\ &= g(W_{3,5} g(W_{1,3} a_1 + W_{2,3} a_2) \\ &\quad + W_{4,5} g(W_{1,4} a_1 + W_{2,4} a_2)) \end{aligned}$$

geldt

$$\frac{\partial a_5}{\partial W_{3,5}} = g'(in_5) \cdot a_3$$

en

$$\frac{\partial a_5}{\partial W_{1,3}} = g'(in_5) \cdot W_{3,5} \cdot g'(in_3) \cdot a_1 \quad .$$

Hierbij: $in_5 = W_{3,5} a_3 + W_{4,5} a_4$ en $in_3 = W_{1,3} a_1 + W_{2,3} a_2$.
In dit geval hebben we dus doorgenummerd.

Het meest gebruikte leerschema voor Neurale Netwerken is **backpropagation = backprop** (1969, 198?), waarbij we — nadat de invoer een uitvoer heeft gegeven — de fout teruggeven (propageren) van uitvoerlaag richting invoerlaag.

Definieer Error_i als de fout in de i -de uitvoer (dat wil zeggen: i -de target minus i -de uitvoer van het netwerk). De leerregel is dan net als eerder:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \cdot a_j \cdot \Delta_i \quad \text{met} \quad \Delta_i = \text{Error}_i \cdot g'(in_i)$$

voor gewichten $W_{j,i}$ van verborgen laag naar uitvoerlaag.

Voor de gewichten $W_{k,j}$ van invoerlaag naar verborgen laag is de leerregel:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \cdot a_k \cdot \Delta_j \quad \text{met} \quad \Delta_j = g'(\text{in}_j) \sum_i W_{j,i} \Delta_i$$

Hierbij geldt:

α is de leersnelheid

a_k is de activatie van de k -de invoerknoop

in_j is de gewogen invoer voor de j -de verborgen knoop

$W_{j,i}$ is het gewicht op de verbinding tussen

de j -de verborgen knoop en de i -de uitvoerknoop

Δ_i is de i -de “uitvoer-delta”, zie de vorige sheet

AI—Neurale netwerken **Backpropagation — afleiding**

We leiden de leerregel met gradient descent af. De fout per voorbeeld is $E = \frac{1}{2} \sum_i (y_i - a_i)^2$, waarbij de y_i 's de target zijn. Dan geldt:

$$\frac{\partial E}{\partial W_{j,i}} = -(y_i - a_i) \cdot \frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i) \cdot g'(\text{in}_i) \cdot a_j = -a_j \cdot \Delta_i$$

En zo verder:

$$\begin{aligned} \frac{\partial E}{\partial W_{k,j}} &= - \sum_i (y_i - a_i) \cdot \frac{\partial a_i}{\partial W_{k,j}} = - \sum_i \Delta_i \cdot W_{j,i} \cdot \frac{\partial a_j}{\partial W_{k,j}} \\ &= - \sum_i \Delta_i \cdot W_{j,i} \cdot g'(\text{in}_j) \cdot a_k = -a_k \cdot \Delta_j \end{aligned}$$

Voor de sigmoïde $g(x) = 1/(1 + e^{-\beta x})$ geldt overigens dat $g'(x) = \beta g(x)(1 - g(x))$.

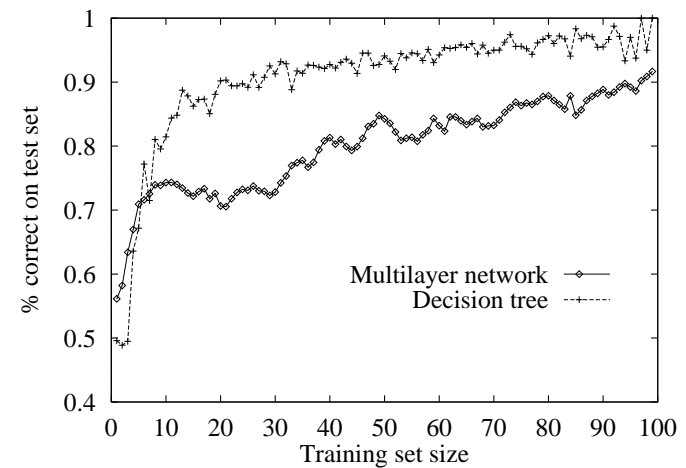
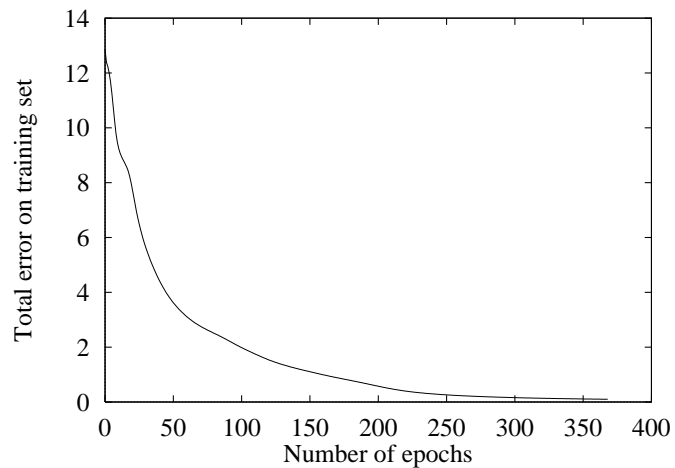
AI—Neurale netwerken **Backpropagation — algoritme**

Het backpropagation-algoritme voor een netwerk met één verborgen laag gaat nu als volgt:

```
repeat
  for each  $e$  in trainingsset do
    maak de  $a_k$ 's gelijk aan de  $x_k$ 's
    bereken de  $a_j$ 's en de  $a_i$ 's (outputs)
    bereken de  $\Delta_i$ 's en de  $\Delta_j$ 's
    update de  $W_{j,i}$ 's en de  $W_{k,j}$ 's
until netwerk "geconvergeerd"
```

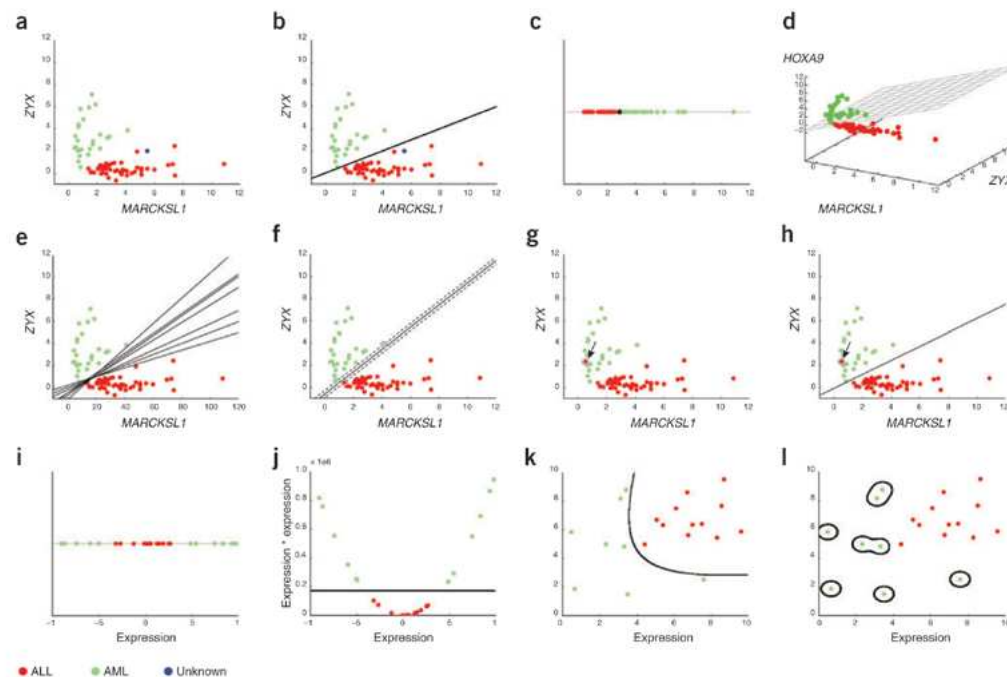
Let erop dat de gewichten goed random geïntialiseerd worden.

Respectievelijk een trainings-curve en een leercurve:



Een **epoch** is een ronde waarin alle voorbeelden uit de trainingsset één keer één voor één in random volgorde door het netwerk gegaan zijn. Soms heb je ∞ veel voorbeelden. Er bestaat naast deze **incrementele** benadering ook een **batch**-benadering.

Een **Support Vector Machine (SVM)** (zie [RN], Hoofdstuk 18.9) probeert de invoerdata zo in een hoger dimensionale ruimte te leggen, dat klassen lineair te scheiden worden.



Uit: W.S. Noble, What is a Support Vector Machine? Nature Biotechnology 24, 1565–1567 (2006).

Er zijn allerlei uitbreidingen. Zo kun je **weight decay** toepassen (laat gewichten in de buurt van 0 verdwijnen), **momentum** (= impuls) toevoegen (onthoud vorige wijziging), met **softmax** verschillende uitvoerunits van elkaar af trainen, . . .

Neurale netwerken worden overal ingezet: voor waterhoogtes, beurskoersen, Backgammon, sonarbeelden, herkenning van nummerborden (classificatie), datacompressie, . . .

Je blijft last houden van locale extremen.

Zie verder het speciale college Neurale Netwerken.

Voor de derde programmeeropgave moet een Neuraal Netwerk (met backpropagation) worden gemaakt om **letters** te herkennen.

Test het netwerk eerst op een paar eenvoudige problemen.

Gebruik zonodig **Perl** om de invoerfiles voor te bewerken.
Denk aan trainingsset en testset.
Gebruik `gnuplot` voor plots/grafieken.



Het huiswerk voor de volgende keer (dinsdag 29 maart 2011): lees **Hoofdstuk 6**, p. 202–233 van [RN] over het onderwerp Constrained Satisfaction Problemen eens door. Kijk ook eens naar de vragen bij dit hoofdstuk.

Denk na over de derde opgave: **Neurale Netwerken**.