Kunstmatige Intelligentie (AI)

Hoofdstuk 21 van Russell/Norvig = [RN]

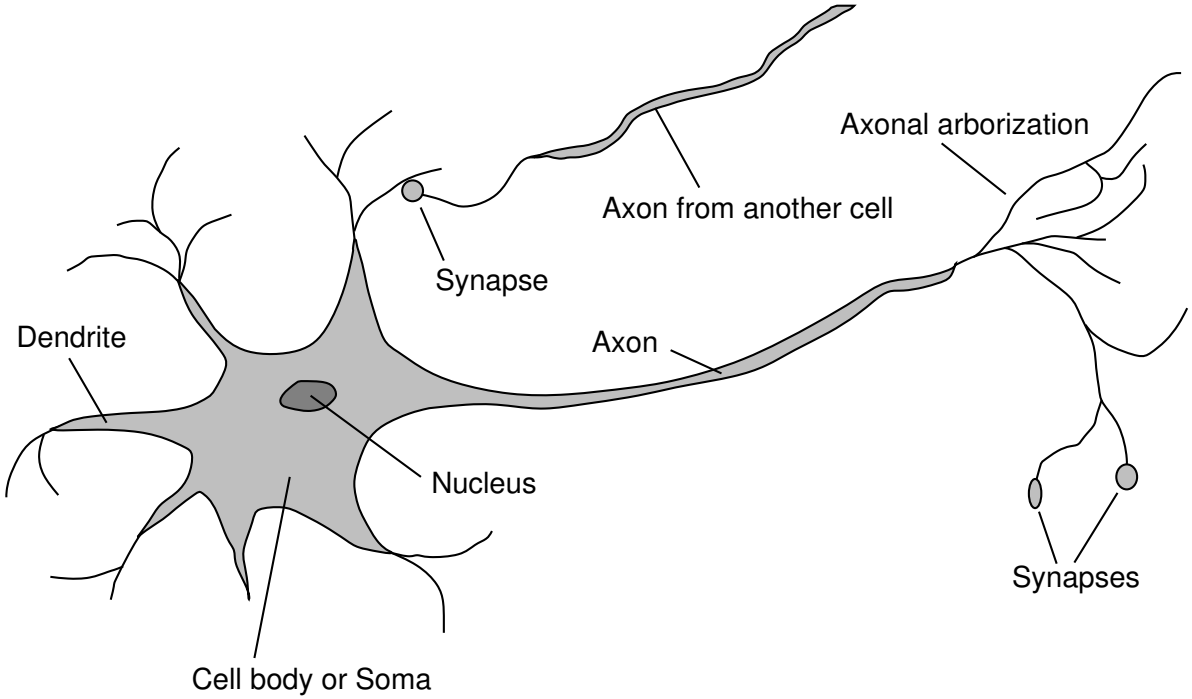
Deep learning, Neurale netwerken

voorjaar 2024

College 10 en 11, 17 en 24 april 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/neuraal.pdf

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



Werking hersenen

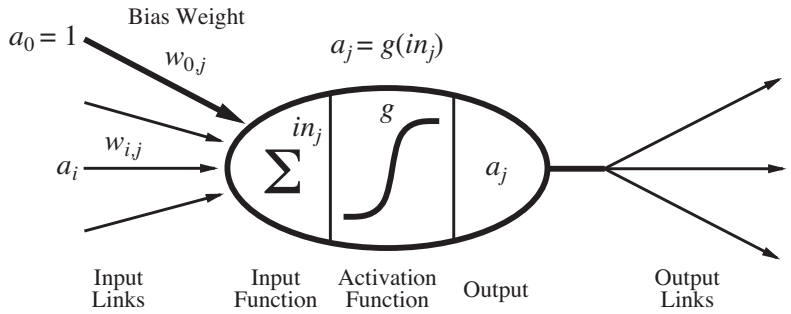
Signalen worden doorgegeven via een nogal gecompliceerde electro-chemische reactie.

Als de electrische 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.

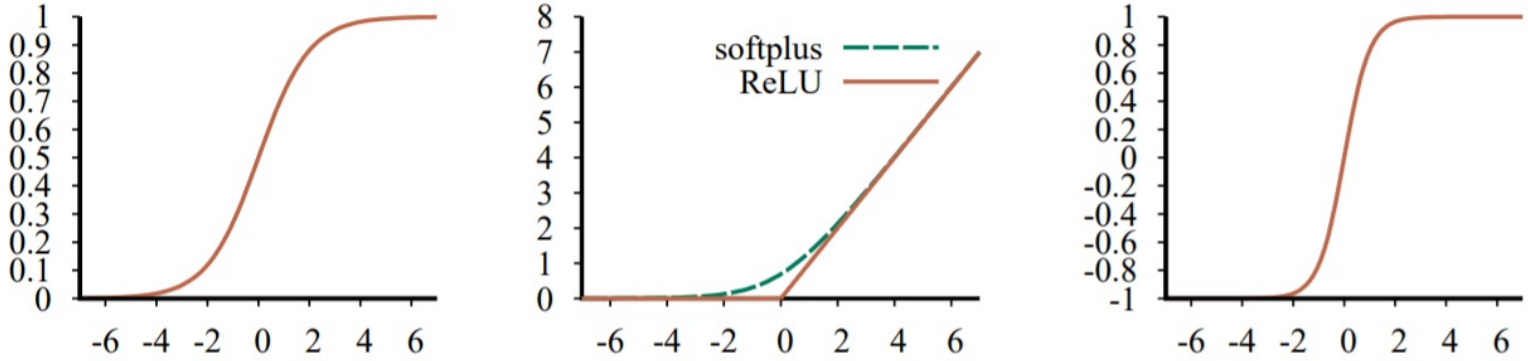
Ooit imiteerden/modelleerden Neurale netwerken menselijke hersenen. Of andersom?

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



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

Veelgebruikte activatie-functies zijn:



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

ReLU$(x) = \max(0, x)$ en softplus$(x) = \log(1 + e^{x})$

$\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1) = 2\sigma(2x) - 1$

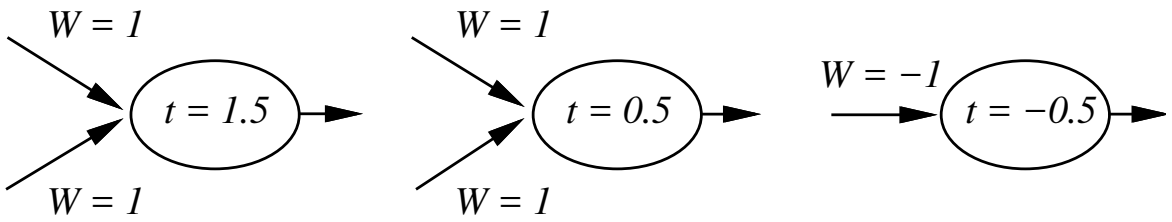
ReLU = rectified linear unit; sigmoid = logistic

De activatie-functies "kantelen" bij 0; De drempelwaarde t binnen de neuronen kan "gesimuleerd" worden door een extra (constante) activatie -1 (of $+1$) in een zogeheten bias-knoop 0 met gewicht $W_{0,j} = t$ op de verbinding van neuron 0 naar neuron j:

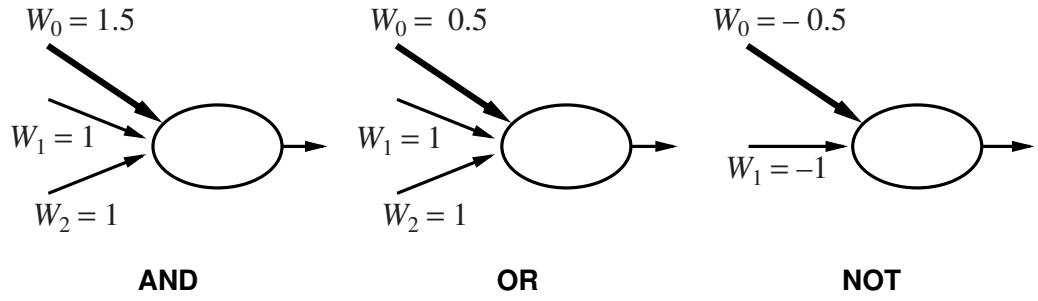
$$\sum_{i=1}^{n} W_{i,j}\ a_i > t \Leftrightarrow \sum_{i=0}^{n} W_{i,j}\ a_i > 0$$

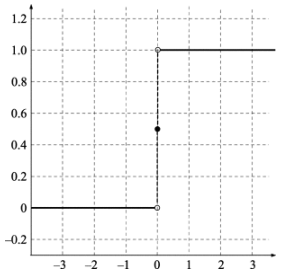
Zo behandelen we drempelwaardes en gewichten uniform.

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



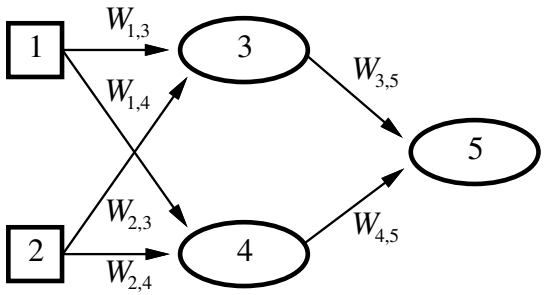
En mét bias-knopen:





Met activatie-functie $\mathtt{Y}(x) = 1$ als $x \geq 0$; 0 als $x < 0$.

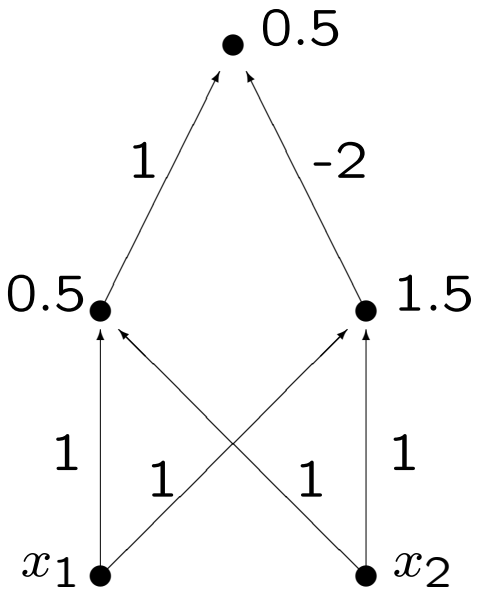
Een feed-forward netwerk heeft gerichte takken en geen cykels (in tegenstelling tot een recurrent netwerk). Vaak is zo'n netwerk in lagen georganiseerd:

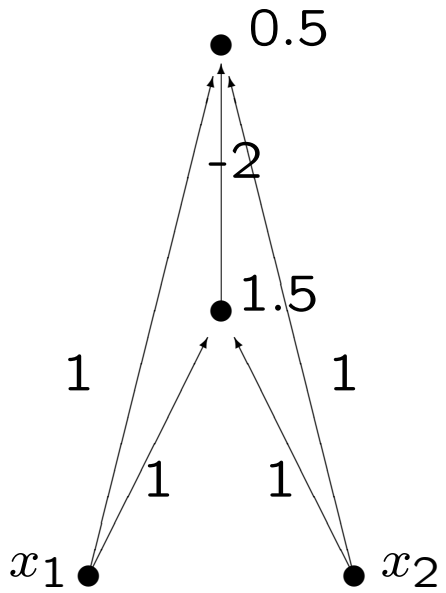


Dit netwerk heeft twee input-knopen 1 en 2, twee verborgen (= hidden) knopen 3 en 4, en één uitvoer-knoop 5. Het representeert de volgende functie:

$$\begin{aligned} a_5 &= g_5\left(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4\right) \\ &= g_5\left(W_{3,5} \cdot g_3\left(W_{1,3} \cdot x_1 + W_{2,3} \cdot x_2\right)\right. \\ &\quad \left. + W_{4,5} \cdot g_4\left(W_{1,4} \cdot x_1 + W_{2,4} \cdot x_2\right)\right) \end{aligned}$$

Voorbeelden:





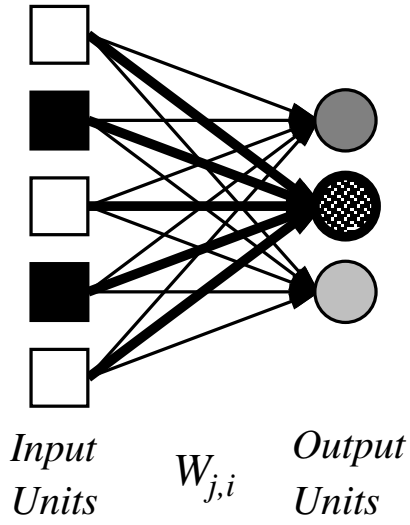
De drempels staan naast de knopen.
Het linker feed-forward netwerk representeert de `XOR`-functie, de verborgen units zijn in feite een `OR` en een `AND`.
Het rechter netwerk representeert dezelfde functie, maar is niet "conventioneel" (geen lagen).

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.

In een perceptron (geen verborgen units!) zijn de uitvoerknopen onafhankelijk:



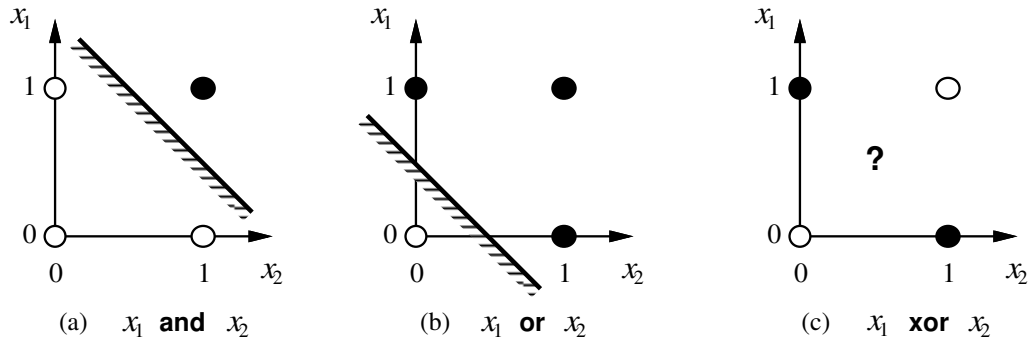
Voor een enkele output-unit geldt dat de uitvoer 1 is indien $\sum_j W_j x_j = W_0 x_0 + W_1 x_1 + \ldots + W_n x_n \geq 0$ en anders 0. Hierbij is $(x_1, \ldots, x_n)$ de invoer en $x_0 = -1$ de bias-knoop. We gebruiken dus weer even de Heaviside/stap-functie

$$\mathtt{Y}(x) = 1 \text{ als } x \geq 0;\ 0 \text{ als } x < 0$$

Wat kan een perceptron?

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 $j = 1, 2, \ldots, n$.

De vergelijking $-W_0 + W_1 x_1 + \ldots + W_n x_n \geq 0$ laat zien dat je precies Booleaanse functies kunt maken die lineair te scheiden zijn, de `XOR`-functie dus niet:



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Hoe leert een perceptron?

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

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

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

met Error = correcte uitvoer − net-uitvoer en $\alpha > 0$ de leersnelheid (= learning rate). De correcte uitvoer heet wel de target, het doel.
Als Error = 1 en $x_j = 1$, wordt W_j ietsje opgehoogd, in de hoop dat de net-uitvoer hoger wordt.

Perceptron — voorbeeld

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?

Perceptron — anders leren

We kunnen in plaats van de discontinue functie $\mathtt{Y}$ ook een differentieerbare activatie-functie g gebruiken in de knopen. Een vergelijkbaar leeralgoritme gaat dan als volgt.

Zij $E = \frac{1}{2}\text{Error}^2 = \frac{1}{2}\Big(y - g(\sum_{j=0}^{n} W_j x_j)\Big)^2$ 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 $-$, we willen de fout laten *dalen*; leersnelheid $\alpha > 0$):

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

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

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

We rekenen uit

$$\left.\frac{\partial E}{\partial W_1}\right|_{(W_1,W_2)=(1,1)} = (5 - (-W_1 + 4W_2))\Big|_{(W_1,W_2)=(1,1)} = 2$$

$$\left.\frac{\partial E}{\partial W_2}\right|_{(W_1,W_2)=(1,1)} = -4(5 - (-W_1 + 4W_2))\Big|_{(W_1,W_2)=(1,1)} = -8$$

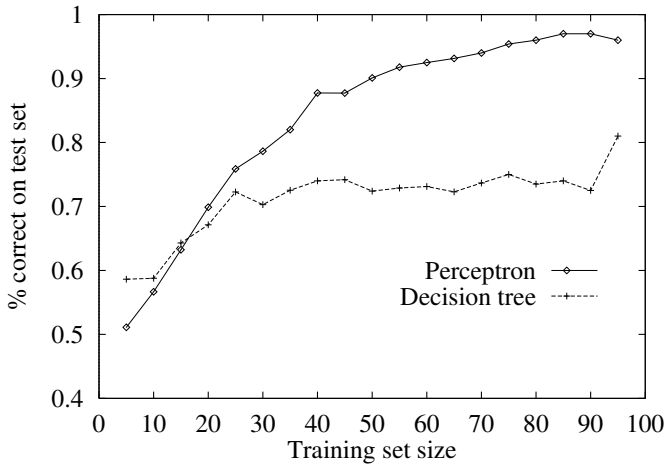
en passen aan (met kleine $\alpha > 0$; let op de $-$):

$$W_1 \longleftarrow 1 - 2\alpha, \quad W_2 \longleftarrow 1 + 8\alpha$$

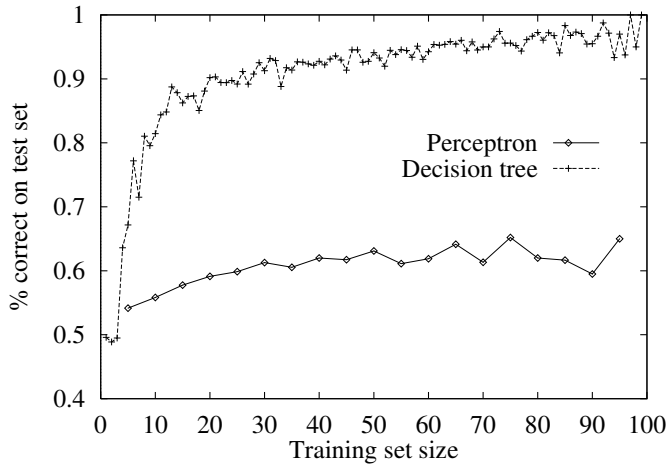
Dus W_2 wordt meer (omhoog) aangepast!

Perceptron — vergelijking

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 het vorige college "Leren"):



Maar voor een meer complex probleem als het "restaurant" 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 waarschijnlijk beter met drie aparte knopen, en dan respectievelijk als 1–0–0, 0–1–0 en 0–0–1: gedistribueerd coderen = one-hot encoding.

Je kunt zelfs bij de foutmaat ervoor zorgen dat je waarden als 1–1–0.9 niet fijn vindt, en daar van weg trainen met een extra laag.
Oftewel: we willen kansen ≥ 0 die samen 1 zijn:

$$\text{softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}} \right)$$

Leerproces: training

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).

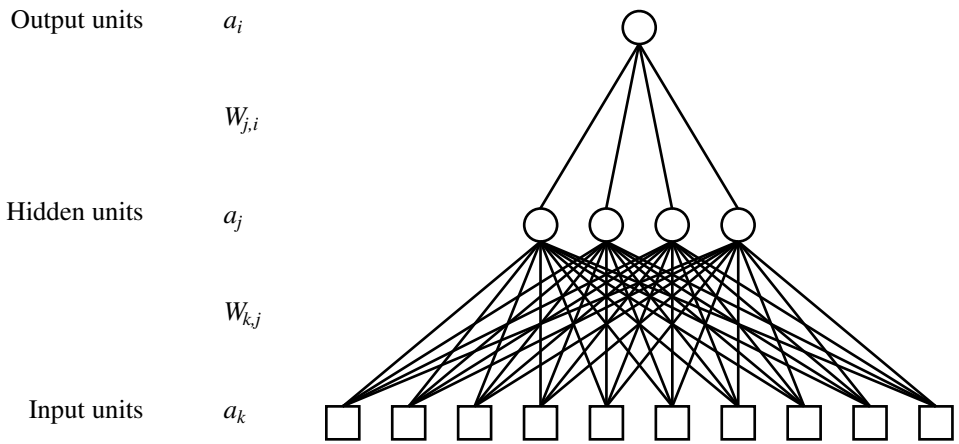
Je rapporteert tot slot over het gedrag op een (weer apart gehouden) testset.

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 bachelorcollege Data Mining. En Statistiek.

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 sheet 8), 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 bekende leerschema voor Neurale Netwerken is back-propagation (1969, 198?), waarbij we — nadat een invoer een uitvoer heeft opgeleverd — 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, dus $y_i - a_i$). De leerregel is dan net als eerder:

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

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

Voor gewicht $W_{k,j}$ van de k-de invoerknoop naar knoop j in de verborgen laag is de leerregel:

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

Hierbij geldt:

α is de leersnelheid

$a_k = x_k$ 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

We leiden de leerregel met gradient descent af. De gradient wijst in de richting van de sterkste toename.

De fout per voorbeeld (x, y) is $E = \frac{1}{2}\sum_i (y_i - a_i)^2$, waarbij de y_i's samen de target y vormen. Definieer $\text{Error}_i = y_i - a_i$.

Dan geldt voor het gewicht $W_{j,i}$ op de verbinding van de j-de verborgen knoop naar de i-de uitvoerknoop:

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

We hebben namelijk $a_i = g(\text{in}_i) = g(\sum_\ell W_{\ell,i} a_\ell)$.

Dus wordt de leerregel:

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

We hebben $E = \frac{1}{2}\sum_i (y_i - a_i)^2$, $a_i = g(\sum_\ell W_{\ell,i} a_\ell)$ en $a_j = g(\text{in}_j) = g(\sum_q W_{q,j} a_q)$, en dus:

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

Hier hebben we gedefinieerd $\tilde{\Delta}_j = g'(\text{in}_j)\sum_i W_{j,i}\Delta_i$ en $\Delta_i = (y_i - a_i) g'(\text{in}_i)$, met leerregel $W_{k,j} \longleftarrow W_{k,j} + \alpha \cdot a_k \cdot \tilde{\Delta}_j$.

In diepe netwerken gebruikt men automatisch differentiëren om dit soort regels af te leiden.

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

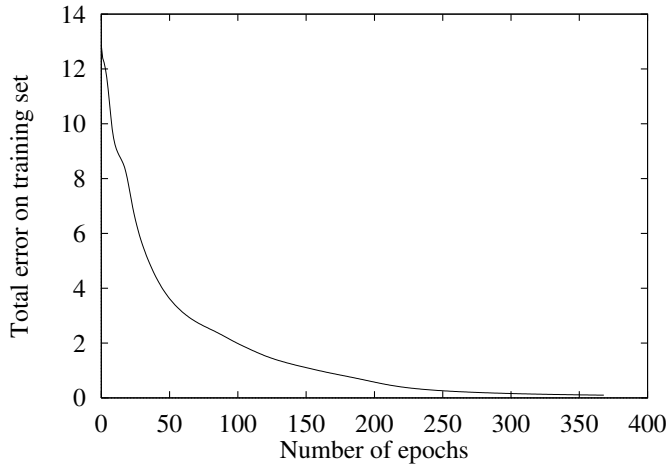
Het back-propagation-algoritme voor een netwerk met één verborgen laag gaat nu als volgt:

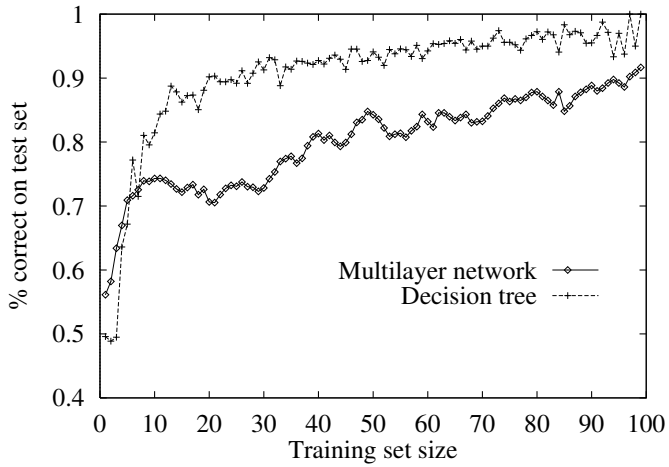
repeat
  for each e **in** trainingsset **do**
    maak de a_k's gelijk aan de x_k's
    bereken de a_j's en de a_i's (outputs)
    bereken de Δ_i's en de $\tilde{\Delta}_j$'s
    update de $W_{j,i}$'s en de $W_{k,j}$'s
until netwerk "geconvergeerd"

Let erop dat de gewichten goed random (?) geinitialiseerd worden. En bied de voorbeelden in random volgorde aan.

hint-sheet

Respectievelijk een trainings-curve en een leercurve:

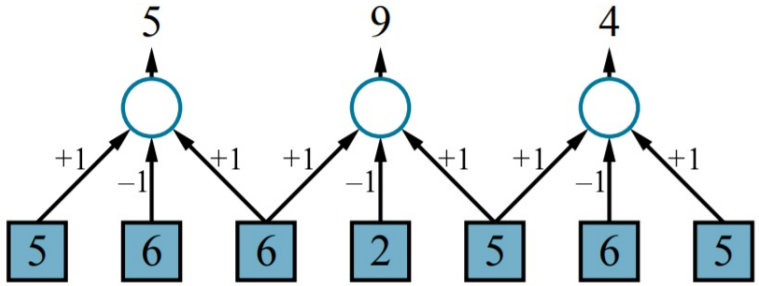




Een epoch is een ronde waarin alle (of één) voorbeelden uit de trainingsset één keer één voor één in random volgorde door het netwerk gegaan zijn. Soms heb je ∞ veel voorbeelden.

Er bestaat naast deze incrementele benadering ook een batch-benadering.

Deep learning staat volop in de belangstelling: netwerken met zeer veel lagen (50–100) van verschillende types. De architectuur is heel complex. Het werkt erg goed bij beeldherkenning (met pixels).
Men gebruikt Convolutional Neural Networks (CNN's): groepjes neuronen in een laag hebben dezelfde gewichten (dat is goed voor ruimtelijke invariantie, een oog in een plaatje ziet er overal hetzelfde uit): een "kernel".



We berekenen output z_i ($i = 1, 3, 5$) via $\sum_{j=1}^{\ell} k_j x_{j+i-(\ell+1)/2}$ uit invoer $(x_0, \ldots, x_6)$. Hier is $\ell = 3$ de grootte van de kernel $k = [+1, -1, +1]$, en stride (= stap) $s = 2$.

Vaak worden pooling layers (met vaste kernels) gebruikt. Bijvoorbeeld max pooling bepaalt het maximum van ℓ pixels.

Het werkt met speciale hardware (GPU's, TPU's) die efficiënt CNN's kan doorrekenen (de T is van "Tensor" = meer-dimensionaal array).

Zie verder Residual networks, . . .

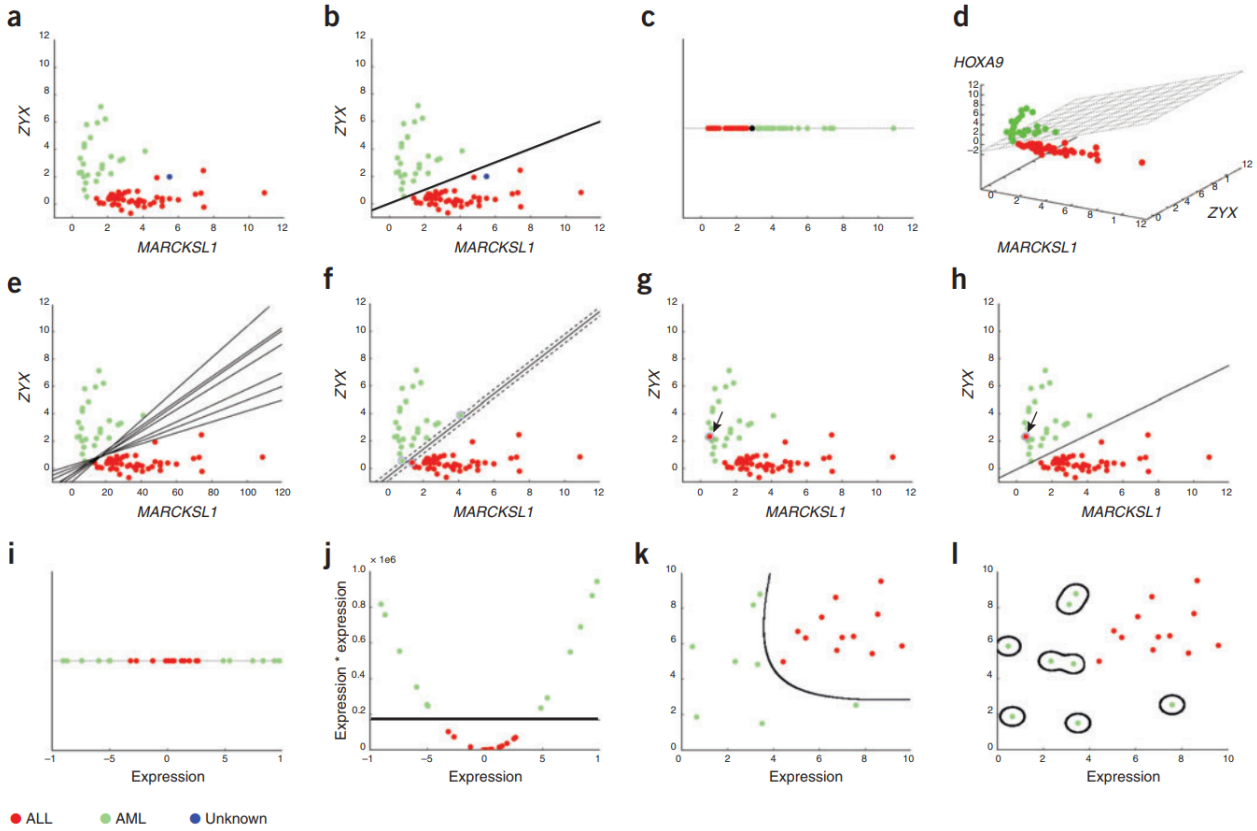
Zie ook `Keras`, TensorFlow, Theano, . . .

Nog meer soorten

Er zijn nog vele andere soorten netwerken, zoals

- Hopfield netwerken (associatief geheugen)
- Kohonen's Self Organizing Maps (SOM's)
- Support Vector Machines (SVM's), kernel machines
- Long short-term memory (LSTM) netwerken: spraakherkenning, tijdreeksen, . . .
- . . .

Een Support Vector Machine (SVM) probeert de invoer-data 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 regularisatie bijvoorbeeld in de vorm van weight decay toepassen (laat gewichten in de buurt van 0 verdwijnen), momentum (= impuls) toevoegen (onthoud vorige wijziging), . . . , en nu dus Deep learning.

Neurale netwerken worden overal ingezet: voor waterhoogtes, beurskoersen, Backgammon, sonarbeelden, beeldherkenning, datacompressie, . . .

Je blijft last houden van locale extremen. En de vanishing gradient?

Zie verder het mastercollege Neurale Netwerken.

Maak voor de vierde programmeeropgave een Neuraal Netwerk (met back-propagation) om eenvoudige functies te voorspellen: (X)OR en AND en . . . pokerhanden.

www.liacs.leidenuniv.nl/~kosterswa/AI/nn24.html

De eerstvolgende keer zijn we nog met Deep learning bezig. Het huiswerk voor de daaropvolgende keer (woensdag 1 mei 2024): lees Hoofdstuk 4.1, p.110–119 van [RN] door (in de derde druk p. 126–129) over het onderwerp Locaal zoeken en Optimalisatie.

Denk aan de "sommen":

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf
www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven2.pdf

Zie ook de video's met uitwerkingen op

www.liacs.leidenuniv.nl/~kosterswa/AI/

Werk aan de vierde opgave: Neurale netwerken; deadline: woensdag 15 mei 2024.