Indledning

Den menneskelige hjerne har inspireret til at lave kunstige systemer, der efterligner dens virkemåde. Det kalder man for *neurale netværk*, og de fungerer som en matematisk funktion ved at give et vist output for et givet input. I modsætning til funktioner skal man *træne* neurale netværk til at give det ønskede svar. Det svarer altså til menneskelig indlæring.

Historisk note

Neurale netværk blev foreslået af Warren McCulloch (1898-1969) og Walter Pitts (1923-1969) i 1943, og i 1958 introducerede den amerikanske psykolog Frank Rosenblatt (1928-1971) den såkaldte *perceptron*, som jeg kommer ind på i næste afsnit. Senere kom *multilags* neurale netværk, som man kan træne ved hjælp af den såkaldte *back-propagation algoritme*. Det var de amerikanske psykologer David Rumelhart (1942-2011) og James McClelland (født 1948), der i 1986 introducerede algoritmen. Rosenblatt implementerede sin perceptron på Cornell universitetets MARK 1 computer i 1960, og med moderne computere kan man relativt nemt lave neurale netværk. Jeg vil vise dig, hvordan du gør i Excel.

Indhold

Kapitlet indeholder følgende tre afsnit:

1. Hjernen
2. Perceptron
3. Multilags neurale netværk

Titelbladet

Titelbladet er inspireret af *neuroner* i hjernen.

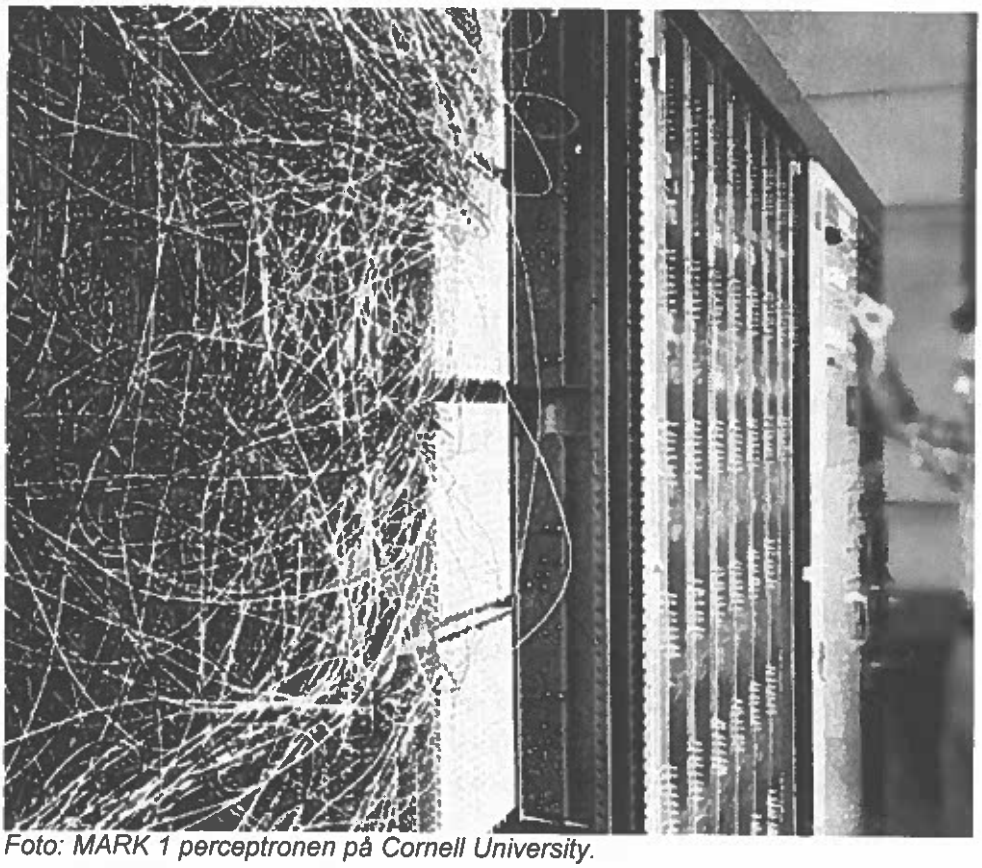
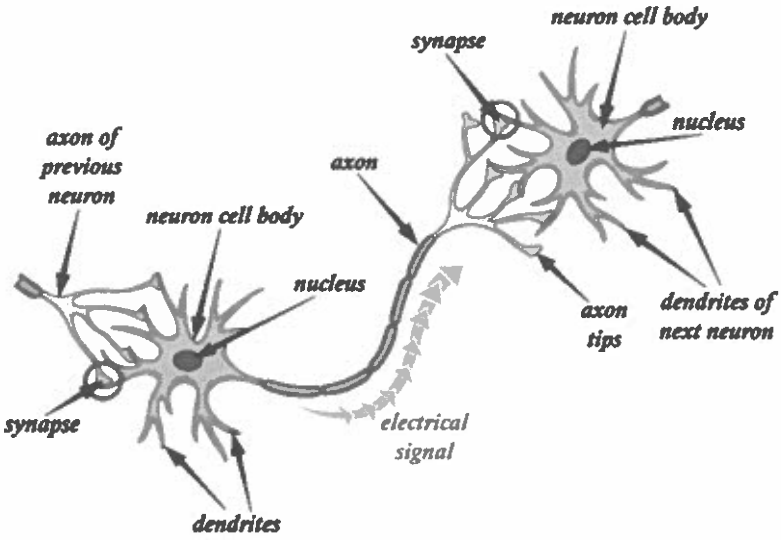


Foto: MARK 1 perceptronen på Cornell University.

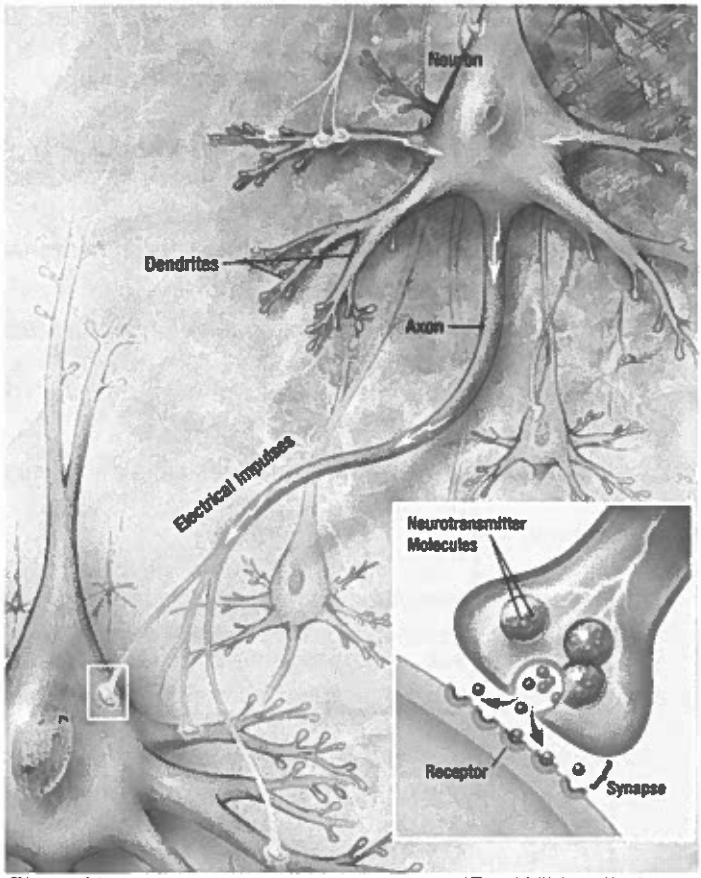
1: Hjernen

Neuroner

Menneskets fantastiske organ hjernen består af række såkaldte *neuroner*, der er celler, som er forbundet med hinanden via såkaldte *axoner*.



I axonerne løber elektriske signaler med en fart på op til 100 m/s. Signalerne ankommer til neuronerne via såkaldte *synapser*, som menneskehjernen har ca. 10^{14} af. Her udløser de kemiske stoffer kaldet *neurotransmittere*.



Figur: Neuroner, axoner og synapser. (Fra Wikipedia.)

Fra neuronerne bliver signalerne enten hæmmet eller forstærket og derpå sendt videre. Input-signaler fra en del af kroppen giver derved anledning til output-signaler i en anden. Det er dette princip, som har inspireret til neurale netværk.

Opgaver

1	Studér hjernen og find ud af, hvilke funktioner de enkelte dele har.

2: Perceptron

Perceptron

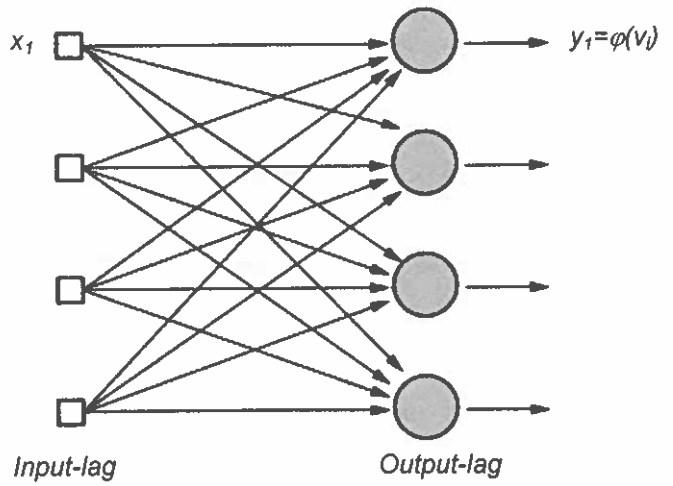
Perceptronen er et simpelt neuralt netværk, som virker på følgende måde:

Input-lag

Input-laget består af en række tal

$x_1, x_2, x_3, ..., x_n$

som grafisk er placeret i firkanterne til venstre på figuren nedenunder.



Figur: Input- og output-laget i en perceptron.

Output-lag

Hver firkant er forbundet til en række cirkler, som udgør *output-laget*. Disse forbindelser svarer til linjerne ovenover, og til hver er der knyttet en såkaldt *vægt* w_{ij}, som er et tal, der indgår i senere beregninger.

Neuroner

Cirklerne kalder man for *neuroner*, og de producerer output-værdier på samme måde som en matematisk funktion:

$y_1, y_2, y_3, ..., y_m$

Det foregår sådan: For hver output-neuron beregner man en vægtet sum

$v_i = x_i w_{i1} + x_2 w_{i2} + ...x_n w_{in} + b$

hvor b er et tal, som man på engelsk kalder for *bias*.

Derefter anvender man en såkaldt *overførselsfunktion* φ:

$y_i = \varphi(x_1 w_{i1} + x_2 w_{i2} + ...x_n w_{in} + b)$

Denne funktion kan f.eks. være den såkaldte *signum-funktion*:

$$sign(v) = \begin{cases} 1 & for \quad v > 0 \\ -1 & for \quad v \le 0 \end{cases}$$

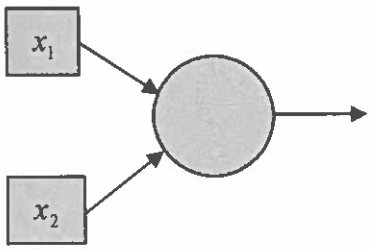
Her ændrer man tit funktionsværdierne til 1 og 0 i stedet for 1 og -1.

Eksempel

XOR

Nedenunder har jeg implementeret en simpel perceptron i Excel:

	A	B	C	D
1				
2	Neurale netværk			
3	Perceptron			
4				
5	OR			
6	Nr	Input x	Vægt w	Output
7	1	1	1	1
8	2	1	1	1
9	Bias			0
10				1



Formlerne er:
D7 = B7*C7
D8 = B8*C8
D10 = HVIS(SUM(D7:D9)<=0;0;1) *(Jeg har ændret lidt i signum-funktionen)*

Denne perceptron giver den logiske funktion ELLER (på engelsk OR), som er defineret sådan her:

x_1	x_2	*OR*
0	0	0
0	1	1
1	0	1
1	1	1

Man forstår bedre perceptronen, hvis man opfatter input-laget som en vektor:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}$$

På samme måde kan man opfatte vægtene for forbindelser til en given output-neuron "i" som en vektor:

$$\vec{w}_i = \begin{pmatrix} w_{i1} \\ w_{i2} \\ \dots \\ w_{in} \end{pmatrix}$$

Den vægtede sum bliver nemlig så *prikproduktet* af de to vektorer (plus bias):

$$v_i = x_i w_{i1} + x_2 w_{i2} + \dots x_n w_{in} + b = \vec{x} \cdot \vec{w}_i + b$$

Tænk nu på vektorer i planen. Ligningen for en linje med normalvektor $\vec{n}$ gennem punktet P_0 får man sådan:

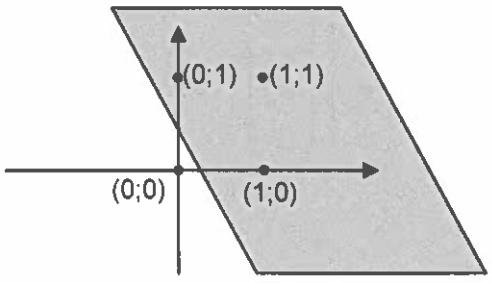
$\vec{n} \cdot \overrightarrow{P_0 P} = 0$ for et givet P på linjen

Med andre ord:

$$\begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix} = a(x - x_0) + b(y - y_0) = ax + by + c = 0$$

Punkter, som ligger på linjen, har ax+by+c lig med 0. Punkter, der ligger henholdsvis på den ene eller anden side af linjen, har ax+by+c større eller mindre end 0. Linjen skiller dermed to delplaner.

Sammenlign summen ax+by+c med den vægtede sum for perceptronen. Det er samme type. Signum-funktionen ser på, om summen er større eller mindre end nul. Den skiller dermed to delplaner ad. Perceptronen afgør med andre ord, om punkter ligger i det ene eller det andet plan.



Figur: Delplaner for OR-funktionen.

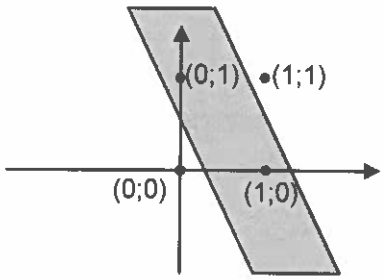
På figuren ovenover ser du OR-funktion, hvor tre input skal give output 1. Disse input svarer til punkter i en bestemt plan.

Eksempel

XOR-funktionen

Den logiske XOR-funktion er defineret sådan her;

x_1	x_2	*OR*
0	0	0
0	1	1
1	0	1
1	1	0



Figur: Problemet med XOR-funktionen.

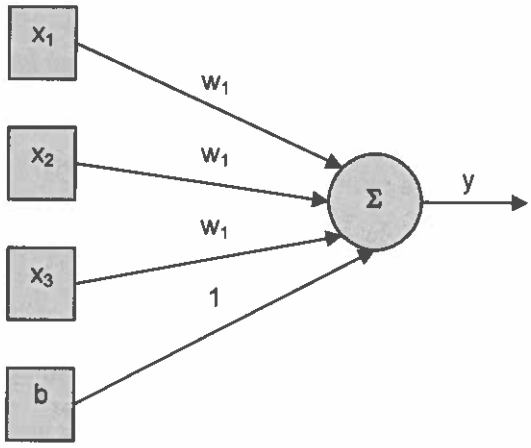
De input, som skal give 1, ligger i et bælte, som man ikke kan skille ad med en enkelt linje som før. Det er derfor *umuligt* for en perceptron at levere XOR-funktionen.

Træning af perceptronen

Det er vægtene, som styrer, hvordan en perceptron reagerer. Hvordan finder man ud af, hvad disse vægte skal være for at få det rigtige output?

Det findes der en *algoritme* for, dvs. en metode bestående af en række trin, som man kører igennem evt. i flere omgange.

Jeg nøjes med at kigge på en perceptron med blot en enkelt output-neuron:



Bemærk, at jeg har placeret biasen i input-laget og lavet en forbindelse med vægt 1. Det betyder, at jeg kan skrive den vægtede sum som prikproduktet mellem to vektorer:

$$v = \vec{x} \cdot \vec{w} = \begin{pmatrix} x_1 \\ \cdots \\ x_n \\ b \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ \cdots \\ w_n \\ 1 \end{pmatrix} = x_i w_1 + x_2 w_2 + \ldots x_n w_n + b$$

Algoritme

Her er algoritmen, som *træner* perceptronen til den ønskede funktion:

Trin 0

Sæt alle vægte til nul:

$$\vec{w} = \vec{0}$$

Trin 1

Lav en række træningsopgaver, dvs. input-vektorer $\vec{x}$ og ønskede output-værdi d.

Trin 2

Vælg en bestemt træningsopgave og udregn den faktiske output-værdi:

$$y = sign(\vec{x} \cdot \vec{w})$$

Trin 3

Justér vægtene således:

$$w_i(ny) = w_i + \eta \cdot (d - y) \cdot x_i$$

Her er η (det græske bogstav *eta*) et lille tal, som man kalder *indlæringsraten*.

Trin 4

Gentag trin 2-3.

Iteration

Bemærk, at man bruger den justerede vægt fra trin 3 som input til trin 2, hvor algoritmen starter forfra. Man kalder dette for *iteration* og siger, at man *itererer* for at justere vægtene.

Eksempel

Her er en perception, som jeg vil træne til at reagere som OR-funktionen:

	A	B	C	D	E	F	G
1							
2	Neurale netværk						
3	Perceptron. Iterer med Ctrl p					Eta	0,1
4							
5	OR						
6	Nr	Input x	Vægt w	Output			Nye vægte
7	1	1	-1,1	-1,1			-1
8	2	0	0,1	0			0,1
9	Bias			0			
10				0		Ønsket	1

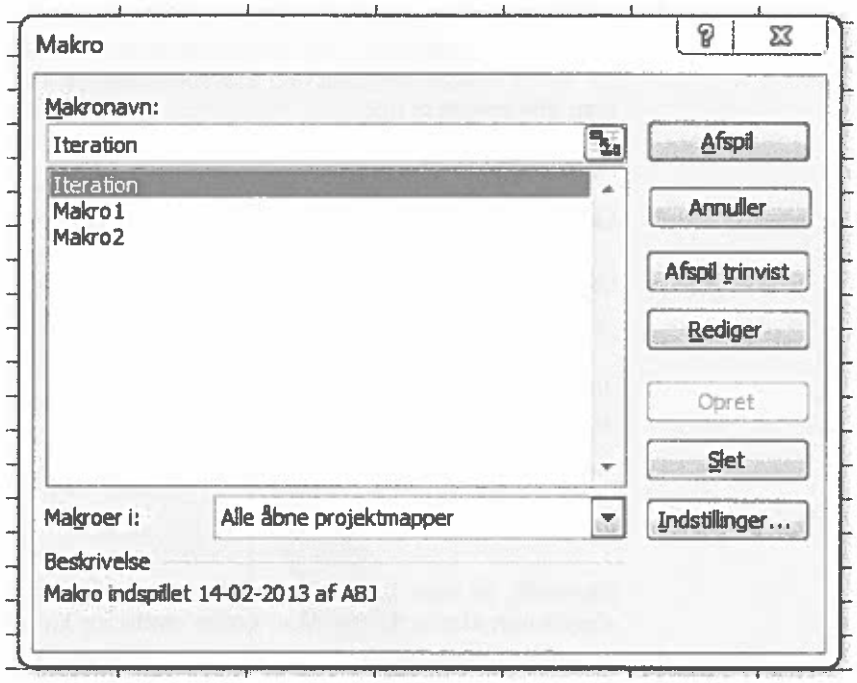
De nye formler er:
G7 = C7+$G$3*($G$10-$D$10)*B7
G8 = C8+$G$3*($G$10-$D$10)*B8

Makro

Når formlerne er på plads, kan jeg iterere én gang, idet Excel regner de nye vægte ud. Men jeg vil gerne iterere igen og igen, og her er det smart at indspille en såkaldt *makro*.

I Excel under *Funktioner* vælger jeg *Makro* og *Indspil ny makro*. Jeg kalder makroen *Iteration* og vælger genvejstasten Ctrl i.

Så kopierer jeg de nye vægte og laver en *indsæt specielt* med værdier på et frit område af celler. Derpå kopierer jeg dette område og indsætter specielt med værdier i cellerne E10 til E17. Endelig afslutter jeg indspilningen af makroen.



Konvergens

Som nævnt er det ikke sikkert, at perceptronen kommer til at virke. Det så du med XOR-problemet. Men hvis input-vektorerne for de to ønskede output-værdier -1 og 1 ligger i to lineært adskilte planer, vil algoritmen ovenover *konvergere*, dvs. give et sæt af vægte, som ikke ændrer sig ret meget ved yderligere iteration.

Det er indholdet af perception-konvergens sætningen:

Sætning

Perception-konvergens sætningen

Træningsalgoritmen for perceptronen konvergerer, dvs.

$w_i(ny) = w_i$ fra et vist trin

hvis input-vektorerne for de to ønskede output-værdier -1 og 1 ligger i to lineært adskilte planer i det n-dimensionelle rum.

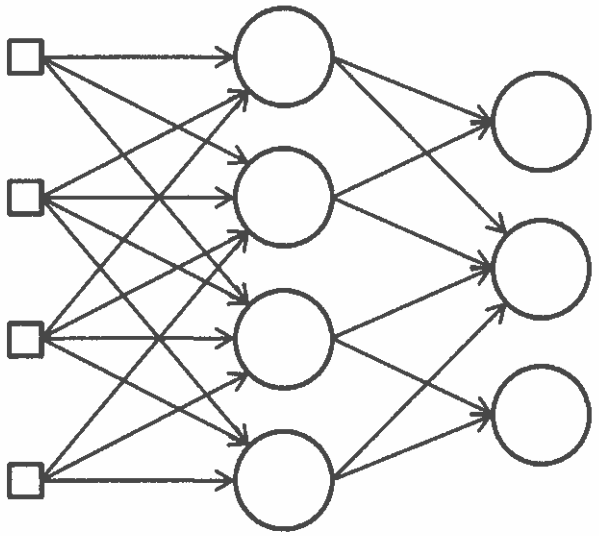
Opgaver	
1	Lav en perceptron, som giver den logiske funktion AND.
2	Bevis Perception-konvergens sætningen!

Foto: Frank Rosenblatt - opfinderen af perceptronen.

3: Multilags neurale netværk

Skjulte lag

Perceptronen har et input-lag af neuroner, som er forbundet til et output-lag, hvor der foregår en vægtning og evaluering. I *multilags neurale netværk* er der ét eller flere *skjulte lag* mellem input- og output-laget. Disse skjulte lag af neuroner er forbundet med hinanden, og i hver neuron foregår der en vægtning og evaluering af input, der producerer et output, som derpå bliver sendt videre til neuronerne i næste lag.



Figur: En multilags neuralt netværk med ét skjult lag.

Formålet med netværket er at få det til at virke som en *funktion*, der til et givet sæt af input-tal x producerer et bestemt sæt af output-tal y.

Notation

For at forklare detaljerne er det nødvendig med noget notation. Her er der flere ting at tænke på.

For det første er tal, som er output fra det ene lag, input til det næste. Man kan derfor ikke sige, at x står for input og y for output.

For det andet er der brug for indicering, f.eks. x_i, hvor "i" står for lagets nummer og "j" for neuronens nummer i det givne lag.

For det tredje er det en fordel at bruge en notation, som stemmer nogenlunde med den, som findes i litteraturen om emnet.

Med disse forhold i mente er jeg nu klar til at definere netværkets variable.

x og y

Tallene i input-laget, dvs. i firkanterne ovenover, kalder jeg x_i. Input-laget gør intet andet end at sende tallene videre til det skjulte lag. Output fra input-laget kalder jeg

$y_i^{(1)}$

Jeg har dermed to betegnelser for det samme nemlig tallene i input-laget:

$x_i = y_i^{(1)}$

Vægt w

I det første skjulte lag (lag 2) bliver tallene fra input-laget vægtet og evalueret.

Det foregår sådan i den i'te neuron: Til hver forbindelse (dvs. pil ovenover) til neuronen knytter jeg et tal, som hedder en vægt, og som jeg betegner

$w_{ij}^{(2)}$

Den i'te neuron i lag 2 producerer nu et tal, som bygger på de forskellige input:

$$v_i^{(2)} = \sum_j w_{ij}^{(2)} y_j^{(1)}$$ (sum over alle output j fra input-laget)

Overførsel

Dette tal bliver *evalueret* af en såkaldt *overførselsfunktion* φ (det græske bogstav phi), og resultatet definerer man som output for det i'te neuron:

$y_i^{(2)} = \varphi(v_i^{(2)})$

Jeg vender straks tilbage til, hvad overførselsfunktionen konkret kan være.

Tallet bliver nu sendt videre til alle neuroner i næste lag i en proces, som man kalder *feed forward.*

For den i'te neuron i lag l+1 har jeg:

$$v_i^{(l+1)} = \sum_j w_{ij}^{(l+1)} y_j^{(l)}$$ (sum over alle output fra lag l)

Her er:

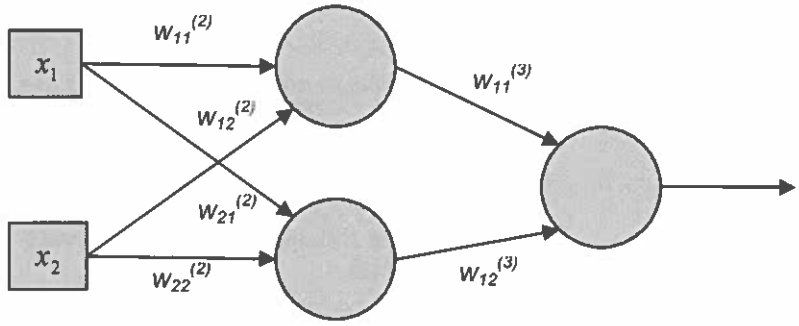
$y_j^{(l)}$ værdi af input fra den j'te neuron i lag l.

$w_{ij}^{(l+1)}$ vægt af input fra den j'te neuron i lag l til den i'te neuron i lag l+1.

Eksempel

XOR

Her er et simpelt eksempel, der giver XOR-funktionen.



Figur: Et simpelt neuralt netværk til XOR-funktionen.

Dette netværk skal til to givne input-tal x_1 og x_2 producere et output-tal $y_1^{(3)}$.

Jeg skal definere netværkets vægte - én for hver pil:

$w_{11}^{(2)} = 1$, $w_{12}^{(2)} = -1$, $w_{21}^{(2)} = -1$, $w_{22}^{(2)} = 1$ samt

$w_{11}^{(3)} = 1$, $w_{12}^{(3)} = 1$

Jeg bruger følgende overførselsfunktion:

$$\varphi(x) = \begin{cases} 0 & for \quad x \le 0 \\ 1 & for \quad x > 0 \end{cases}$$

Netværket implementerer jeg i Excel:

	A	B	C	D	E	F	G
1							
2	**Neurale netværk**						
3	XOR med hård overførselsfunktion						
4							
5	Lag					Vægt	
6	1	Input		Output		w1	w2
7		x1	1	y1	1	1	-1
8		x2	1	y2	1	-1	1
9					Nyt input	x1	x2
10						0	0
11							
12	2	Input		Output		w1	
13		x1	0	y1	0	1	
14		x2	0	y2	0	1	
15					Nyt input	x1	
16						0	
17	3	Input		Output			
18	3	x1	0	y1	0		

Formlerne er:
E7 = C7
E8 = C8

F10 = E7*F7+E8*F8
G10 = E7*G7+E8*G8

C13 = F10
C14 = G10

E13 = HVIS(C13>0;1;0)
E14 = HVIS(C14>0;1;0)

F16 = E13*F13+E14*F14

C18 = F16
E18 = HVIS(C18>0;1;0)

Dette netværk giver som nævnt XOR-funktionen, som perceptronen i det foregående afsnit ikke kunne levere. Prøv selv!

Træning ved back-propagation

Hvordan vælger man vægtene w i et neuralt netværk, så det kan levere den ønskede funktion? Det findes der en algoritme til, som hedder *back-propagation*.

Jeg beskriver først algoritmen generelt, derpå konkret via et eksempel, og til sidst viser jeg, hvorfor den virker.

Algoritme

Jeg ønsker, at et neuralt netværk for input-sættet
$(x_1, \ x_2, \ \ldots \ x_n)$
skal give output-sættet
$(d_1, \ d_2, \ \ldots \ d_m)$

Trin 1

Jeg sætter vægtene w tilfældigt.

Trin 2

Jeg kører input-sættet igennem netværket og beregner fejlene, dvs. afvigelserne mellem de ønskede output-værdier og det faktuelle:
$e_i = d_i - y_i^{(L)}$

Trin 3

Jeg udregner følgende hjælpestørrelser:

$\delta_i^{(L)} = e_i \cdot \varphi'(v_i^{(L)})$ for hver neuron i output-laget L

$\delta_i^{(l)} = \varphi'(v_i^{(l)}) \cdot \sum_k \delta_k^{(l+1)} w_{ki}^{(l+1)}$ for hver neuron i det l'te lag

Her summerer jeg over alle neuroner k i det næste lag l+1.

Trin 4

Med en selvvalgt konstant η kaldet *indlæringsraten* (græske bogstav eta) justerer jeg vægtene således:

$w_{ij}^{(l)}(ny) = w_{ij}^{(l)} + \eta \cdot \delta_i^{(l)} \cdot y_j^{(l-1)}$

Trin 5

Gentag trin 2-4 i et antal iterationer, indtil netværket fungerer så godt som muligt.

Her kommer et eksempel på denne algoritme.

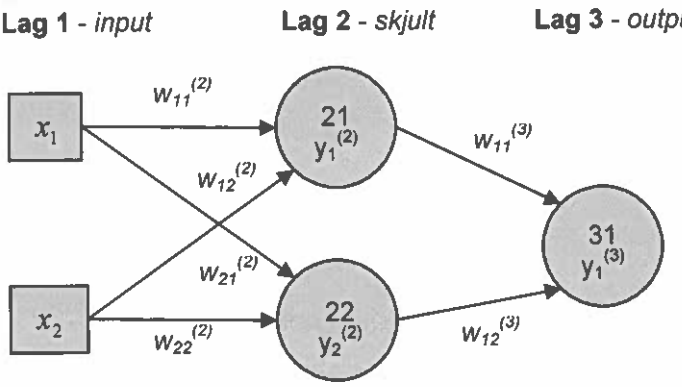
Eksempel

XOR

Jeg prøver igen at lave et neuralt netværk for XOR-funktionen, men denne gang vil jeg selv træne det. Som overførselsfunktion bruger jeg følgende såkaldte *logistiske* funktion:

$$\varphi(x) = \frac{1}{1+e^{-x}}$$

I modsætning overførselsfunktionen i det foregående eksempel er funktionen her differentiabel, hvilket er nødvendigt for back-propagation algoritmen, hvor der jo indgår et φ'.



Figur: Et simpelt neuralt netværk til XOR-funktionen.

Når jeg skal bruge formlerne, får jeg brug for at differentiere overførselsfunktionen. Det giver i dette tilfælde et pænt resultat. Se her:

$$\varphi'(x) = \frac{0-1\cdot(-e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}-1}{(1+e^{-x})^2}$$

$$= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) = \varphi(x)\cdot(1-\varphi(x))$$

De centrale formler i algoritmen er nu:

$\delta_1^{(3)} = (d_1 - y_1^{(3)}) y_1^{(3)} (1 - y_1^{(3)})$

$\delta_1^{(2)} = y_1^{(2)}(1 - y_1^{(2)}) \cdot \delta_1^{(3)} w_{11}^{(3)}$

$\delta_2^{(2)} = y_2^{(2)}(1 - y_2^{(2)}) \cdot \delta_1^{(3)} w_{12}^{(3)}$

$w_{11}^{(3)}(ny) = w_{11}^{(3)} + \eta\delta_1^{(3)} y_1^{(2)}$ og $w_{12}^{(3)}(ny) = w_{12}^{(3)} + \eta\delta_1^{(3)} y_2^{(2)}$

$w_{11}^{(2)}(ny) = w_{11}^{(2)} + \eta\delta_1^{(2)} y_1^{(1)}$ og $w_{12}^{(2)}(ny) = w_{12}^{(2)} + \eta\delta_1^{(2)} y_2^{(1)}$ samt

$w_{21}^{(2)}(ny) = w_{21}^{(2)} + \eta\delta_2^{(2)} y_1^{(1)}$ og $w_{22}^{(2)}(ny) = w_{22}^{(2)} + \eta\delta_2^{(2)} y_2^{(1)}$

Jeg implementerer igen netværket i Excel:

	A	B	C	D	E	F	G	H	I	J
1										
2	**Neurale netværk**									
3	XOR med logistisk overførselsfunktion, iterer med makro Ctrl i									
4										
5	Lag 1	*Input*		*Output*				Eta:	0,01	
6		x1	1	y1(1)	1					
7		x2	1	y2(1)	1					
8										
9	Lag 2	*Input*		*Vægt*		*Sum*	*Output*	*Gradient*	*Nye vægte*	
10	Neuron1	x11=y1(1)	1	*w11(2)*	0,9	*v1(2)*	*y1(2)*	*delta1(2)*	*w11(2)*	0,90082815
11		x12=y2(1)	1	*w12(2)*	0,9	1,8	0,8581489	0,08281536	*w12(2)*	0,90082815
12	Neuron2	x21=y1(1)	1	*w21(2)*	6,6	*v2(2)*	*y2(2)*	*delta2(2)*	*w21(2)*	6,59999999
13		x22=y2(1)	1	*w22(2)*	6,6	13,2	0,9999981	-9,94E-07	*w22(2)*	6,59999999
14										
15	Lag 3	*Input*		*Vægt*		*Sum*	*Output*	*Gradient*	*Nye vægte*	
16	Neuron1	*x11=y1(2)*	0,858149	*w11(3)*	-19	*v1(3)*	y1(3)	*delta1(3)*	*w11(3)*	-19,000307
17		*x12=y2(2)*	0,999998	*w12(3)*	15	-1,304858	0,2133486	-0,0358065	*w12(3)*	14,9996419
18										
19						Ønsket	0			

Formlerne er:

E6 = C6 $= y_1^{(1)}$

E7 = C7 $= y_2^{(1)}$

C10 = E6
C11 = E7
C12 = E6
C13 = E7

F11 = C10*E10+C11*E11 $= v_1^{(2)}$

F13 = C12*E12+C13*E13 $= v_2^{(2)}$

G11 = 1/(1+EKSP(-F11)) $= y_1^{(2)}$

G13 = 1/(1+EKSP(-F13)) $= y_2^{(2)}$

C16 = G11
C17 = G13

Formlerne for back-propagation er:

H17 = (G19-G17)*G17*(1-G17) $= \delta_1^{(3)}$

J16 = E16+$I$5*H17*C16 $= w_{11}^{(3)}$

J17 = E17+$I$5*H17*C17 $= w_{12}^{(3)}$

H11 = G11*(1-G11)*H17*E16 $= \delta_1^{(2)}$

H13 = G13*(1-G13)*H17*E17 $= \delta_2^{(2)}$

J10 = E10+$I$5*H11*C10 $= w_{11}^{(2)}$

J11 = E11+$I$5*H11*C11 $= w_{12}^{(2)}$

J12 = E12+$I$5*H13*C12 $= w_{21}^{(2)}$

J13 = E13+$I$5*H13*C13 $= w_{21}^{(2)}$

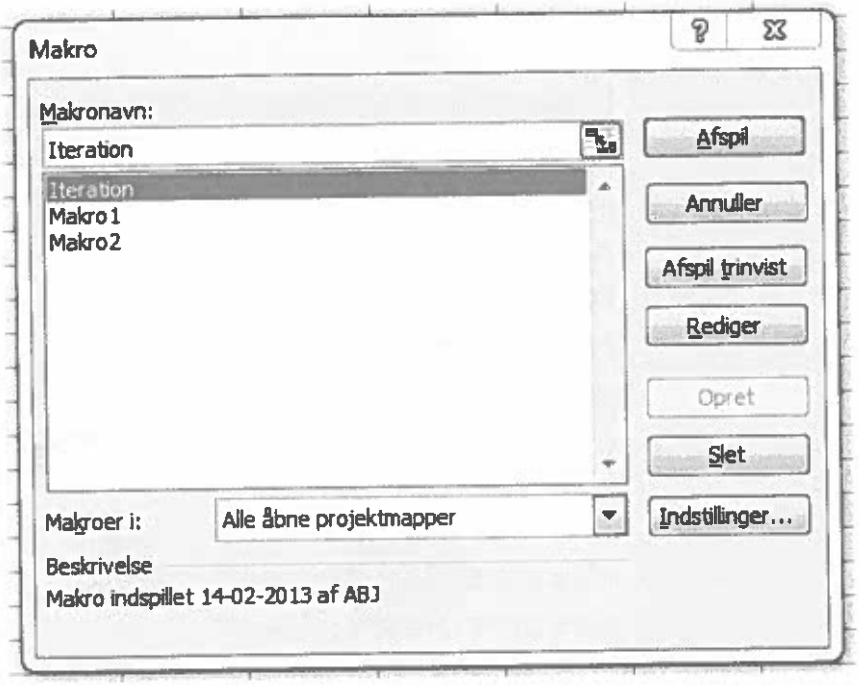
Bemærk, hvordan justeringerne af vægtene begynder i lag 3 og fortsætter i lag 2, idet ændringerne i et lag afhænger af det følgende. Det er derfor, man kalder algoritmen for back-propagation.

Makro

Når formlerne er på plads, kan man iterere én gang, idet Excel regner de nye vægte ud. Men man vil gerne iterere igen og igen, og her er det smart at indspille en makro.

I Excel under *Funktioner* vælger jeg *Makro* og *Indspil ny makro*. Jeg kalder makroen *Iteration* og vælger genvejstasten Ctrl i.

Så kopierer jeg de nye vægte og laver en *indsæt specielt* med værdier på et frit område af celler. Derpå kopierer jeg dette område og indsætter specielt med værdier i cellerne E10 til E17. Endelig afslutter jeg indspilningen af makroen



Man kan også skrive koden for makroen direkte:

```
Sub Iteration()
'
' Iteration Makro
' Makro indspillet 14-02-2013 af ABJ
'
' Genvejstast:Ctrl+i
'
    Range("J10:J17").Select
    Selection.Copy
    Range("J22:J29").Select
    Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone, SkipBlanks
_
        :=False, Transpose:=False
    Range("J22:J29").Select
    Application.CutCopyMode = False
    Selection.Copy
    Range("E10:E17").Select
    Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone, SkipBlanks
_
        :=False, Transpose:=False
End Sub
```

Det er nu let at lave iterationer med netværket, da jeg bare skal trykke Ctrl i. Så er jeg klar til at lære netværket XOR-funktionen.

Det gør jeg ved at opstille de forskellige input-værdier med deres ønskede output og så iterere.

Udledning af formlerne

Til sidst vil jeg udlede de formler, som back-propagation algoritmen bygger på.

Jeg deler op i to tilfælde:

- Neuroner i output-laget
- Neuroner i skjulte lag

Neuroner i output-laget

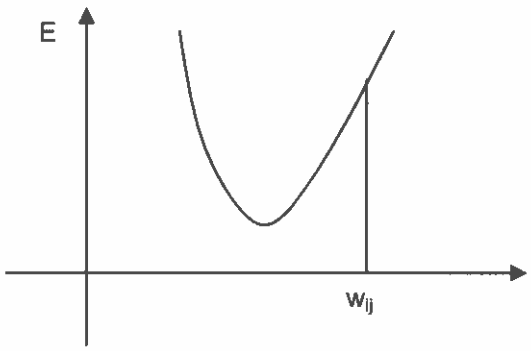
Neuronerne i output-laget leverer hver især et output y_i, som man direkte kan sammenligne med den ønskede værdi d_i. Fejlen er forskellen

$$e_i = d_i - y_i$$

Den samlede fejl måler jeg med størrelsen

$$E = \frac{1}{2}\sum_k e_k^2 \qquad \text{(sum over alle neuroner i output-laget)}$$

Kig på den i'te neuron i output-laget. Den j'te forbindelse til denne neutron har en vægt w_{ij}, som jeg gerne vil justere, så den totale fejl bliver minimeret. Lad os sige, at den totale fejl E afhænger således af w_{ij}:



For en given værdi w_{kj} vil jeg undersøge, om jeg skal lægge noget til eller trække noget fra for at få en mindre fejl, dvs. om Δw_{ij} i formlen

$$w_{ij}(ny) = w_{ij} + \Delta w_{ij}$$

skal være positiv eller negativ.

Det finder jeg ud af ved at kigge på hældningskoefficienten til grafen. Hvis den som vist er positiv, skal jeg trække noget fra. Hvis den omvendt var negativ, skulle jeg lægge noget til.

E afhænger af en masse vægte, men nu fokuserer jeg altså på w_{ij}. Hældningskoefficienten er derfor givet med den partielt afledede, som man betegner

$$\frac{\partial E}{\partial w_{ij}}$$

Bemærk, at jeg bruger såkaldte *bløde* d-er, fordi E er en funktion af flere variable.

Kender jeg den, kan jeg forsøge mig med følgende justering:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}}$$

hvor η er et lille tal (kaldet *indlæringsraten*), som jeg gange på "retningen" af justeringen.

Det hele handler nu om at finde ovenstående partielt afledede.

Ved hjælp af *kædereglen* får jeg:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e_i}\frac{\partial e_i}{\partial y_i}\frac{\partial y_i}{\partial v_i}\frac{\partial v_i}{\partial w_{ij}}$$

De fire differentialkvotienter finder jeg én ad gangen:

$\frac{\partial E}{\partial e_i}$

Da

$$E = \frac{1}{2}\sum_k e_k^2$$

er

$$\frac{\partial E}{\partial e_i} = e_i$$

(Her ser du, hvorfor jeg satte en ½ foran - det bliver lidt pænere.)

$\frac{\partial e_i}{\partial y_i}$

Da

$$e_i = d_i - y_i$$

er

$$\frac{\partial e_i}{\partial y_i} = -1$$

$\frac{\partial y_i}{\partial v_i}$

Da

$$y_i = \varphi(v_i)$$

er

$$\frac{\partial y_i}{\partial v_i} = \varphi'(v_i)$$

$\frac{\partial v_i}{\partial w_{ij}}$

Endelig da

$$v_i = \sum_j y_j w_{ij}$$

er

$$\frac{\partial v_i}{\partial w_{ij}} = y_j$$

Sammenlagt får jeg derfor:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e_i}\frac{\partial e_i}{\partial y_i}\frac{\partial y_i}{\partial v_i}\frac{\partial v_i}{\partial w_{ij}}$$

$$= -e_i \cdot \varphi'(v_i) \cdot y_j$$

Justeringen bliver nu:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} = \eta \cdot e_i \cdot \varphi'(v_i) \cdot y_j$$

$$= \eta \cdot \delta_i \cdot y_j$$

hvor

$$\delta_i = e_i \cdot \varphi'(v_i) = (d_i - y_i) \cdot \varphi'(v_i)$$

hedder den *lokale gradient* for den i'te neuron i output-laget.

Neuroner i skjulte lag

Kig videre på den i'te neuron i det skjulte lag lige før output-laget. Den leverer output til neuronerne i output-laget, som videre forårsager en vis samlet fejl:

$$E = \frac{1}{2}\sum_k e_k^2$$ (sum over alle neuroner i output-laget)

Kig igen på den i'te neuron i det skjulte lag. Den j'te forbindelse til denne neutron har en vægt w_{ij}, som jeg gerne vil justere, så den totale fejl bliver minimeret.

Jeg ser igen på hældningskoefficienten og bruger kædereglen:

(*) $$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial v_i}\frac{\partial v_i}{\partial w_{ij}}$$

Jeg udregner de tre differentialkvotienter i (*) én ad gangen:

$\frac{\partial E}{\partial y_i}$

Da

$$E = \frac{1}{2}\sum_k e_k^2$$

får jeg i første omgang

$$\frac{\partial E}{\partial y_i} = \sum_k e_k \frac{\partial e_k}{\partial y_i} = \sum_k e_k \frac{\partial e_k}{\partial v_k}\frac{\partial v_k}{\partial y_i}$$

Men

$$e_k = d_k - y_k = d_k - \varphi(v_k)$$

så

$$\frac{\partial e_k}{\partial v_k} = -\varphi'(v_k)$$

Videre er

$$v_k = \sum_j y_j w_{kj}$$

så

$$\frac{\partial v_k}{\partial y_i} = w_{ki}$$

Alt i alt:

$$\frac{\partial E}{\partial y_i} = \sum_k e_k \frac{\partial e_k}{\partial v_k}\frac{\partial v_k}{\partial y_i} = -\sum_k e_k \varphi'(v_k) w_{ki}$$

I denne formel genkender jeg noget, nemlig de lokale gradienter defineret tidligere for output-laget:

$$\delta_k = e_k \cdot \varphi'(v_k)$$

Jeg får derfor:

$$\frac{\partial E}{\partial y_i} = -\sum_k \delta_k w_{ki}$$ (sum over neuronerne i output-laget)

$\frac{\partial y_i}{\partial v_i}$

Så til den næste differentialkvotient i (*):

$$\frac{\partial y_i}{\partial v_i} = \varphi'(v_i) \quad \text{idet} \quad y_i = \varphi(v_i)$$

$\frac{\partial v_i}{\partial w_{ij}}$

Og den sidste differentialkvotient i (*):

$$\frac{\partial v_i}{\partial w_{ij}} = y_j \quad \text{da} \quad v_i = \sum_j y_j w_{ij}$$

Justeringen bliver derfor:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} = \eta \cdot \left(\sum_k \delta_k w_{ki}\right) \cdot \varphi'(v_i) \cdot y_j$$

$$= \eta \cdot \delta_i \cdot y_j$$

hvor

$$\delta_i = \left(\sum_k \delta_k w_{ki} \right) \cdot \varphi'(v_i)$$ (sum over neuronerne i output-laget)

igen hedder den lokale gradient.

Bemærk, at jeg nu har samme type formel til justering af vægte. Både for output-laget L og for det skjulte lag lige før L-1 afhænger justeringen af de lokale gradienter.

Det kan jeg bruge, hvis jeg går endnu et skjult lag L-2 tilbage. Her kan jeg nemlig definere nye lokale gradienter ved hjælp af gradienterne for lag L-1.

Jeg kan derfor generalisere gradientformlen rekursivt bagud:

$$\delta_i^{(l)} = \varphi'(v_i^{(l)}) \cdot \sum_k \delta_k^{(l+1)} w_{ki}^{(l+1)}$$ for hver neuron i det l'te lag

Det er netop på denne måde, at back-propagation fungerer.

Opgaver	
1	Lav regnearket for XOR. Kan du forbedre netværkets vægte?
2	Lav et neuralt netværk, som kan spille kryds og bolle!

Foto: David Rumelhart og James McClelland.

Referencer

- Søren Brunak: Neurale netværk. Computere med intuition. (ISBN: 87-1610-003-4)
- Simon Haykin: Neural Networks and Learning Machines (ISBN: 978-81-203-4000-8)