Neurale Netwerken

In dit hoofdstuk bespreken een methode die enigszins verwant is aan regressieanalyse: neurale netwerken. Waar regressieanalyse uitgaat van een door de analist/onderzoeker gespecificeerd model, met verklarende en te verklaren variabelen, gaan neurale netwerken uit van een *black box*. De relaties tussen “te verklaren” en “verklarende” variabelen zijn niet op voorhand duidelijk, en het streven is ook niet om die relaties te begrijpen.

Ook zullen we in deze les dieper ingaan op maatstaven voor het beoordelen van de prestaties van het model. Hoe goed is het model in staat om te voorspellen?

Neurale netwerken: het algoritme

Een veelgebruikte illustratie van neurale netwerken is het herkennen van handschriften. Het is fascinerend te zien op hoeveel verschillende manieren letters en getallen worden geschreven, terwijl toch in vrijwel alle gevallen alle lezers er de juiste interpretatie aan geven.

De MNIST-database (http://yann.lecun.com/exdb/mnist/) van handgeschreven getallen wordt gebruikt als een training- en testset voor het verschillende algoritmes, waaronder neurale netwerken. De onderstaande figuur geeft enkele voorbeelden.

Figuur 1. Illustratie uit de MNIST-database. Bron: http://yann.lecun.com/exdb/mnist/.

Omdat de meesten van ons gemakkelijk alle cijfers in de figuur juist zullen interpreteren, is het ogenschijnlijk gemakkelijk om deze taak te automatiseren. Maar dat is zeker niet het geval!

Om handschriftherkenning te automatiseren, stellen we eerst de vraag: hoe herkent ons brein getallen en cijfers? Zonder al te diep in te gaan op de werking van het brein, geven we een korte inleiding[1].

Het menselijk brein is opgebouwd uit twee hersenhelften. Iedere hersenhelft beschikt over een primaire visuele cortex (gezichtsschors) die is opgebouwd uit miljoenen neuronen, en miljarden verbindingen tussen die neuronen. Naast de primaire visuele cortex (V1) beschikken we ook nog over de visuele cortices V2, V3, V4 en V5, die steeds complexere beelden kunnen verwerken. Ons brein werkt als het ware als een supercomputer die geëvolueerd is over vele eeuwen.

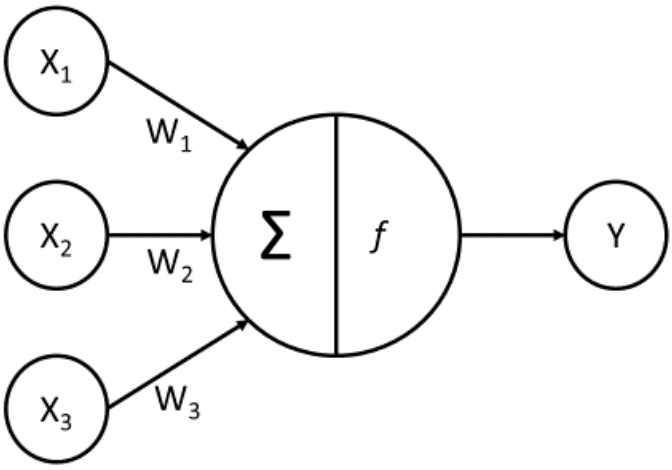
Met een kunstmatig neuraal netwerk (of *Artificial Neural Network*, kortweg ANN) proberen we een deel van die supercomputer na te bootsen. Natuurlijk zijn de resulterende algoritmes slechts een zeer beperkte afspiegeling van de complexe werking van het menselijk brein, maar in veel gevallen zijn ze in staat om goede prestaties te leveren.

> *In een neuraal netwerk zouden we de rechthoeken die de getallen bevatten kunnen opdelen in een groot aantal kleine vierkantjes (zeg, 25*25 pixels) die allemaal de waarde 0 (wit) of 1 (zwart) hebben. Vervolgens kunnen we met een neuraal netwerk de patronen analyseren die de cijfers 0 t/m 9 representeren. Hoe meer divers de patronen zijn, des te complexer het neurale netwerk dat bijna alle cijfers goed "leest".*

De terminologie van ANNs is afgeleid van de werking van het menselijk brein.

Wanneer we als mens iets waarnemen (of zien, of voelen), dan is er sprake van een veelheid van inkomende signalen die door onze zenuwcellen worden opgevangen, via dendrieten. In een chemisch proces worden de diverse signalen "gewogen", op basis van intensiteit en frequentie. Als een zekere drempelwaarde is bereikt dan zal de cel een signaal gaan doorgeven, aan andere zenuwcellen.

In een simpel model, zijn X_1, X_2 en X_3 de inkomende signalen die worden gewogen met de gewichten W_1, W_2 en W_3. De gewogen signalen worden vervolgens vertaald, in een activeringsfunctie, naar een zekere output Y. In dit simpele model is er één neuron, met de drie X-variabelen als de drie dendrieten, en één outputsignaal.



Figuur 2. Input- en outputsignalen. *Bron: OGN.*

[1] Voor wie meer wil weten, zie de YouTube video https://www.youtube.com/watch?v=j-hs0ifskt8

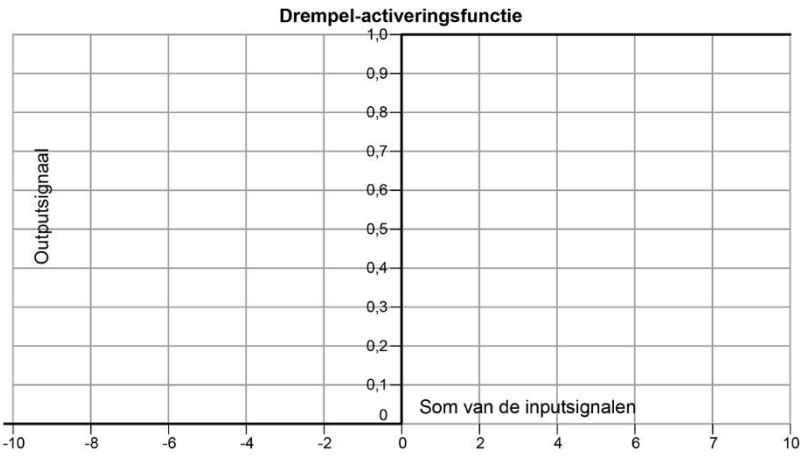
Kunstmatige neurale netwerken zijn te beschrijven aan de hand van drie kenmerken.

1. Het type activeringsfunctie.
2. De netwerkstructuur.
3. Het trainingsalgoritme.

De activeringsfunctie

In de analogie van de menselijke zenuwcel geeft de activeringsfunctie een signaal zodra er een drempelwaarde is bereikt. Is dat niet het geval, dan doet de cel niets. We spreken dan van de "drempelactiveringsfunctie".

Deze activeringsfunctie is als volgt grafisch weer te geven.



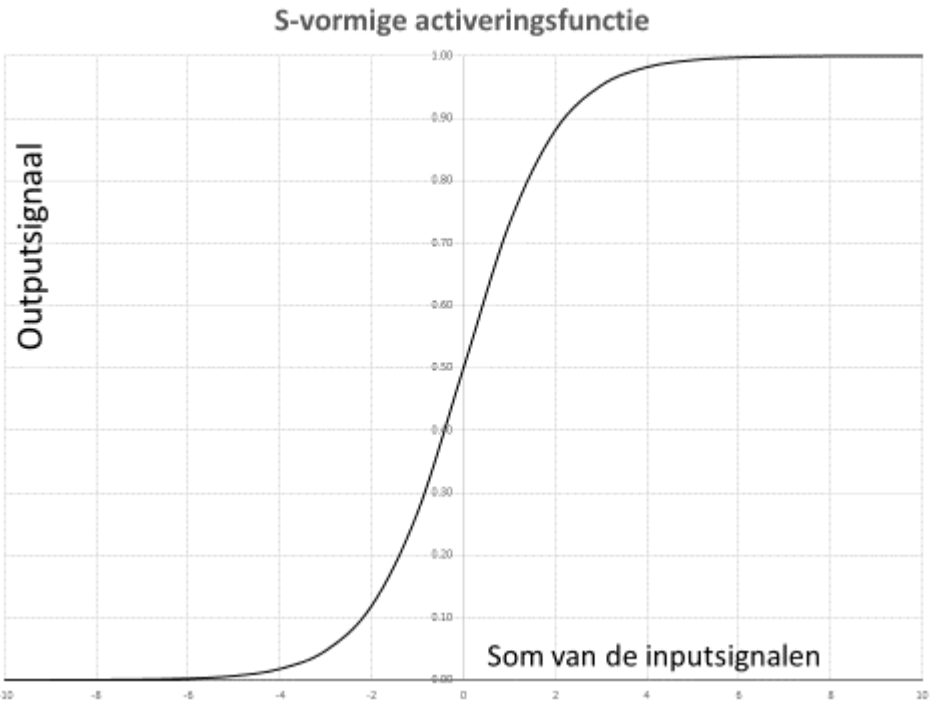
Figuur 3. Drempelactiveringsfunctie. *Bron: OGN.*

De drempelactiveringsfunctie is weliswaar interessant vanwege de parallel met het biochemische proces, maar wordt zelden gebruikt in de kunstmatige neurale netwerken (ANNs) die voor ons van belang zijn. De reden is dat dit type activeringsfunctie te weinig flexibel is om accuraat data te kunnen beschrijven.

Waarom is dat? Wiskundig gezien, proberen we in de ANNs te komen tot die set van gewichten die het beste de data kan beschrijven. In dit optimaliseringsproces is het nodig om de functie te kunnen differentiëren, over het volledige bereik van inputsignalen. Bij differentiëren kijken we wat kleine veranderingen in de inputsignalen betekenen voor het outputsignaal.

En juist in het meest belangrijke deel (rond de drempelwaarde), is de drempelactiveringsfunctie wiskundig niet differentieerbaar! In een differentieerbare functie, hoort bij iedere x-waarde (inputsignaal) één y-waarde. We willen werken met een functie die juist rond de kritische waarden van de inputsignalen een verbetering laat zien bij kleine veranderingen. Maar de drempelactiveringsfunctie verloopt vlak aan de linker- en rechterkant van 0, en is onbepaald voor de waarde 0 zelf: "oneindig" kleine veranderingen hebben geen effect!

Om die praktische reden gebruiken we meestal andere functies. De meestgebruikte activeringsfunctie is de S-vormige curve (ook wel *sigmoïd* genoemd). Het belangrijkste verschil met de drempelactiveringsfunctie is dat nu in principe voor de output alle waarden tussen 0 en 1 kunnen voorkomen, en niet alleen de waarden 0 en 1.



Figuur 4. S-vormige activeringsfunctie. *Bron: OGN.*

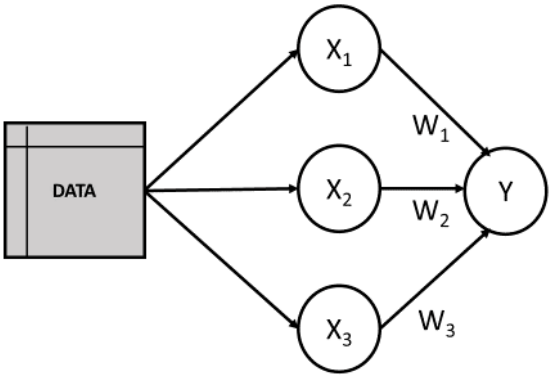
Naast de *sigmoïd* zijn er nog tal van andere specificaties voor de activeringsfunctie, maar die worden in de praktijk zelden gebruikt. Het is van belang te constateren dat het gebruik van de sigmoïd voor inputsignalen kleiner dan -5 en groter dan 5 altijd leidt tot outputsignalen van vrijwel 0 of 1. De mogelijke inputsignalen worden samengedrukt (*squashed*) in een klein bereik van outputsignalen. Om dit *squashing problem* te omzeilen, is het nodig om binnen ANNs de inputsignalen zoveel mogelijk om te zetten in waardes dichtbij 0. Meestal doen we dit door de waarden te normaliseren.

De netwerkstructuur

Met de netwerkstructuur doelen we op een aantal kenmerken.

1. Het aantal lagen in het netwerk;
2. De terugkoppeling van “hogere” lagen (dichterbij het outputsignaal) naar “lagere” lagen (dichterbij het inputsignaal);
3. Het aantal knooppunten in elke laag.

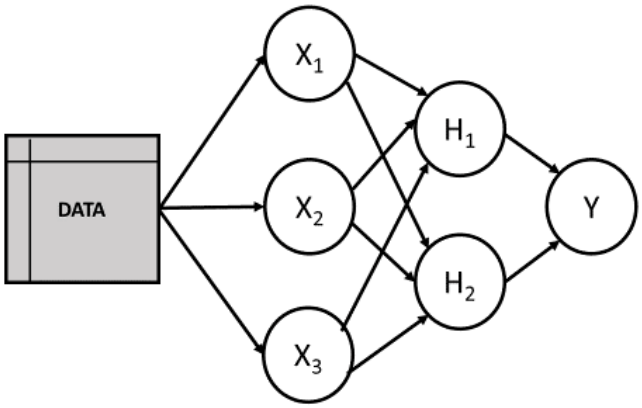
In het simpelste model hebben we alleen een set van inputsignalen. De inputsignalen noemen we de knooppunten (de *nodes*), binnen de laag.



Figuur 5: Single-layer network. Bron: OGN.

Een dergelijk eenvoudig *single-layer network* kan worden gebruikt om eenvoudige patronen te herkennen. In de meeste gevallen hebben we echter een complexer model nodig, met meerdere lagen.

Een model met meerdere lagen noemen we een *multilayer network*. Zo'n network heeft naast de inputknooppunten, ook een verborgen laag (*hidden layer*) met één of meer knooppunten. De knooppunten van de eerste laag (de input) zijn gekoppeld aan alle knooppunten van de volgende laag: ze zijn, zoals we dat noemen, *fully connected*.



Figuur 6: Multilayer network, fully connected. Bron: OGN.

De richting van de informatie

De richting van de informatie is doorgaans van links (input) naar rechts (output). In theorie is het mogelijk ook een *feedback*-stroom in te bouwen die van rechts naar links gaat, maar in de praktijk van ANNs wordt daarvan zelden gebruikgemaakt. Anders gezegd, de modellen die we gebruiken zijn meestal *feed forward* modellen.

Het aantal knooppunten per laag

Het aantal knooppunten in de inputlaag ligt vast met de informatie (de variabelen) die we gebruiken, vanuit de dataset. Natuurlijk kunnen we wel – bijvoorbeeld uit oogpunt van

"spaarzaamheid" en overzichtelijkheid – proberen of een subset van de informatie leidt tot nagenoeg even goede voorspellingen. Of, als het model geen goede resultaten geeft, kunnen we in de toekomst proberen meer relevante variabelen aan het model toe te voegen.

Het aantal knooppunten in de verborgen laag (de *hidden layer*), is aan de analist! In het algemeen leidt een grotere aantal knooppunten tot een betere beschrijving van de training set. Echter, door *overfitting*, is het resultaat voor de test set niet altijd beter!

> Overfitting, in regressiemodellen en artificiële neurale netwerken, is het verschijnsel dat meer "verklarende variabelen", inputsignalen en knooppunten leiden tot een betere beschrijving van de data (in de trainingset). Ter vergelijking, in regressieanalyse kan gemakkelijk worden aangetoond dat een extra verklarende variabele altijd leidt tot een R^2 die groter of gelijk is aan R^2 in een model zonder die variabele! De beschrijving van de data in de trainingset wordt dus alsmaar beter bij het toevoegen van verklarende variabelen. Dat komt doordat elke variabele wel een zekere (kleine) toevallige correlatie heeft met de "ruis" (toevallige afwijkingen) in ons model. Hoe meer van die ruis wordt "verklaard" des te beter zal de beschrijving zijn. Echter, omdat zowel de ruis in ons model als de correlatie tussen ruis en allerlei verklarende variabelen het gevolg zijn van willekeurig processen, zal de voorspelkracht in een test set gering zijn. Het gedachteloos toevoegen van meer variabelen (of inputsignalen, of knooppunten) is daarom een slecht idee!

De resultaten op de testset zijn de beste basis voor een beslissing over het optimale aantal knooppunten. Het is verstandig te beginnen met een klein aantal knooppunten (of zelfs een knooppunt), en vervolgens te experimenteren met het aantal. Een zo klein mogelijk aantal knooppunten met een bevredigend resultaat, is een goede beslisregel.

We hebben ANNs gepresenteerd als een *blackbox* model. De verborgen lagen, en de knooppunten binnen de verborgen lagen, representeren de werking van een complex system. Net zoals bij de visuele cortices in ons brein, is het systeem te complex om in detail te begrijpen. Waarom we zelfs cijfers en letters die "raar" geschreven zijn, juist interpreteren, is in een *blackbox* model van ondergeschikt belang. Wat we willen is een sterk versimpelde weergave van het systeem dat de juiste voorspellingen doet!

Neurale netwerken voor numerieke data

We zullen de werking van ANNs illustreren aan de hand van een voorbeeld.

> Het voorbeeld heeft betrekking op de aantrekkelijkheid van steden, als vestigingsplaats voor investeerders.
> Er is veel informatie bekend over de aantrekkelijkheid van landen, regio's en steden. Helaas is die informatie niet voorhanden voor tal van kleinere landen, regio's en (kleinere) steden. Een belangrijke bron van informatie is de Ease-of-Doing-Business (EODB) studie van de Wereldbank (*http://www.doingbusiness.org/rankings*). In de EODB worden tal van harde en zachte indicatoren verzameld, en gebundeld in een overall indicator. Hoe die bundeling plaatsvindt, is niet helemaal duidelijk. En als dat al duidelijk is, dan biedt dat vaak geen uitkomst voor nieuwe situaties waarin slechts een deel van de benodigde data voorhanden is. In die zin hebben we dus te maken met een daadwerkelijke black box!
> Een internationaal opererend consultancy bedrijf dat investeerders adviseert, probeert nu een EODB indicator te repliceren aan de hand van een relatief geringe set van gemakkelijk te verkrijgen indicatoren voor landen en regio's die nog niet zijn onderzocht. De (harde) gegevens die voorhanden zijn, hebben betrekking op de benodigde tijd om een onderneming te starten, het aantal procedures en het benodigde startkapitaal (alle verbijzonderd naar mannelijke en vrouwelijke ondernemers).

In dit voorbeeld is de variabele die we willen voorspellen een numerieke variabele. De **score**, die is berekend door het bedrijf, is gebaseerd op (of preciezer gezegd, een gewogen gemiddelde van) twee numerieke scores die bekend zijn uit de gegevens van de Wereldbank.

Stap 1: inlezen van de data

De data staan in een csv-bestand, **ML6_OEFEN1.csv.** We lezen het bestand in en geven een overzicht van de structuur van het bestand.

```
> eodb <- read.csv("ML6_OEFEN1.csv", header=TRUE)
> str(eodb)
'data.frame':   182 obs. of  11 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ score        : int  6 8 3 3 5 3 10 10 7 9 ...
 $ eodbrank     : int  183 58 156 182 113 116 38 15 19 65 ...
 $ startbusrank: int  42 46 142 144 124 157 9 7 111 5 ...
 $ procmen      : num  3 5 12 8 9 14 3 3 8 2 ...
 $ timemen      : num  7 5 20 36 22 25 4 2.5 21 3 ...
 $ costmen      : num  19.9 10.1 11.1 27.5 9.4 ...
 $ procwomen    : num  4 5 12 8 9 14 3 3 8 2 ...
 $ timewomen    : num  8 5 20 36 22 25 4 2.5 21 3 ...
 $ costwomen    : num  19.9 10.1 11.1 27.5 9.4 ...
 $ capital      : num  0 0 0 0 0 0 0 0 12.8 0 ...
```

Het bestand telt 11 variabelen, voor in totaal 182 gebieden (landen, en regio's of steden binnen landen).

Omdat de interesse uitgaat naar de aantrekkelijkheid van het gebied, en in het bijzonder naar de aantrekkelijkheid van het gebied als het gaat om het starten van een onderneming door kleine investeerders, heeft het consultancy bedrijf zelf een **score** berekend op basis van de overall EODB ranking door de Wereldbank, en een sub-ranking voor het starten van een eigen onderneming. Het bedrijf wil nu een ANN ontwikkelen om te zien of die **score** kan worden voorspeld met harde gegevens over aantal procedures (**procmen** en **procwomen**), benodigde tijd (**timemen** en **timewomen**), en vereist startkapitaal (**capital**).

We zullen niet alle variabelen gaan gebruiken. Daarom overschrijven we **eodb** met een selectie van die variabelen die we wel nodig hebben in de analyse.

- De variabele **id** is slechts een volgnummer, voor de gebieden waarop de data betrekking hebben.
- De variabelen **eodbrank** en **startbusrank** liggen aan de basis van de **score** maar zullen niet afzonderlijk worden gebruikt. We houden dan de variabelen in de kolommen 2, en 5 t/m 11 over.

  Noot: met grote bestanden is het "tellen" van de kolommen lastig, en foutgevoelig. Veel analisten geven er de voorkeur aan de selectie te doen met de namen van de variabelen in plaats van de nummers van de kolommen! Deze optie is ook in het script opgenomen.

In het eerste commando hieronder staat tussen de vierkante haken niets voor de komma (want we houden alle 182 records), en de combinatie van kolomnummers die we willen behouden, met de `c()` functie na de komma.

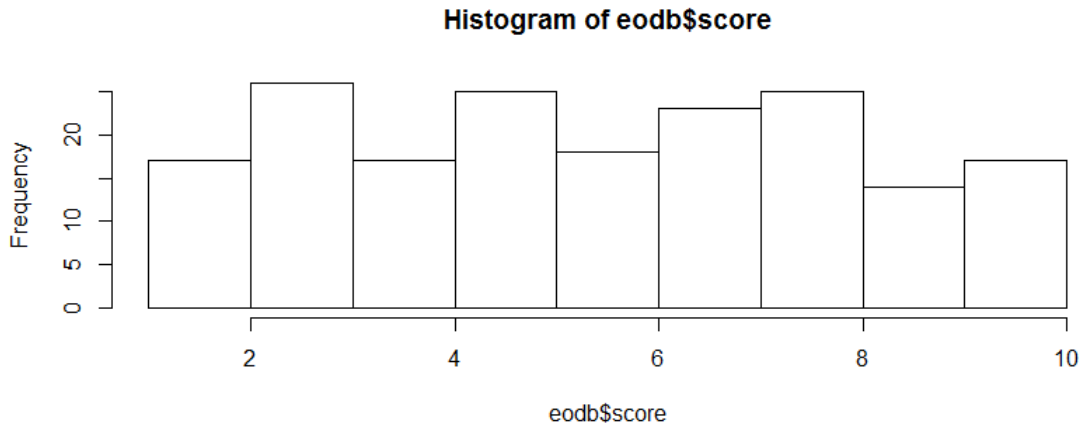
```
> eodb <- eodb[,c(2,5:11)]
> str(eodb)
'data.frame':   182 obs. of  8 variables:
 $ score    : int  6 8 3 3 5 3 10 10 7 9 ...
 $ procmen  : num  3 5 12 8 9 14 3 3 8 2 ...
 $ timemen  : num  7 5 20 36 22 25 4 2.5 21 3 ...
 $ costmen  : num  19.9 10.1 11.1 27.5 9.4 ...
 $ procwomen: num  4 5 12 8 9 14 3 3 8 2 ...
 $ timewomen: num  8 5 20 36 22 25 4 2.5 21 3 ...
 $ costwomen: num  19.9 10.1 11.1 27.5 9.4 ...
 $ capital  : num  0 0 0 0 0 0 0 0 12.8 0 ...
```

Stap 2: verkennen en prepareren van de data

Het is belangrijk te onderkennen dat de variabelen een groot bereik kunnen hebben, en wellicht niet normaal verdeeld zijn.

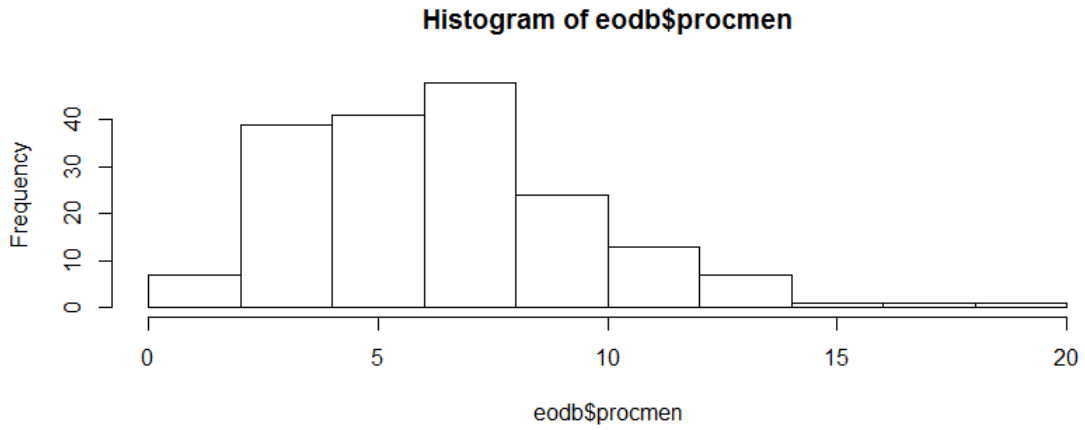
Om inzicht te krijgen in het bereik van de data en in de verdeling, maken we een histogram, met het **`hist()`** commando.

De scores voor aantrekkelijkheid, zijn vrij gelijkmatig verdeeld over de range van 1 tot 10.



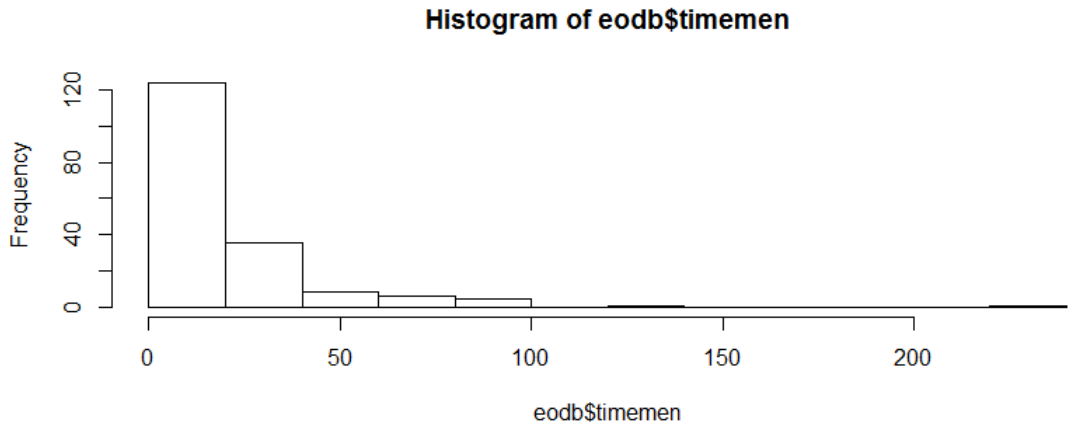
Figuur 7.a. Histogram van de variabele score. *Bron: OGN.*

Het aantal procedures heeft een piek rond de 6 of 7 procedures, maar heeft een scheve verdeling. In sommige gebieden loopt het aantal procedures op tot 20.



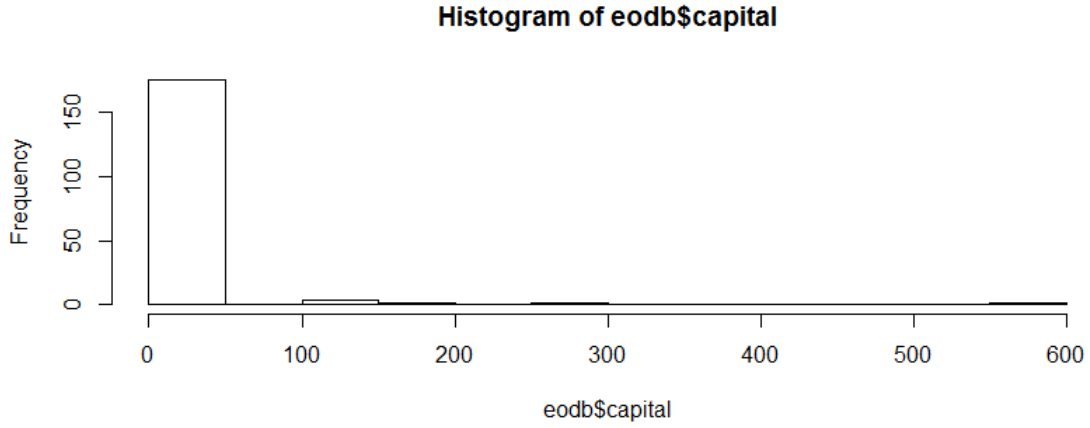
Figuur 7.b. Histogram van de variabele procmen. *Bron: OGN.*

De scheefheid van de verdeling is een groter probleem bij de tijd die een en ander in beslag neemt. Hoewel het opstarten van een onderneming in de meerderheid (ongeveer 120 van 182 gebieden) van de gevallen niet langer duurt dan ongeveer 20 dagen, zijn er uitschieters tot boven de 200 dagen. De verdeling is dus verre van normaal!



Figuur 7.c. Histogram van de variabele timemen. *Bron: OGN.*

Nog schever verdeeld is het benodigde startkapitaal. In het overgrote deel van de gevallen ligt dit tussen de 0 en 50 (het gemiddelde is ongeveer 12; verifieer dit!). Echter er zijn uitschieters tot boven de 600.



Figuur 7.d. Histogram van de variabele capital. *Bron: OGN.*

Controleer voor uzelf of de verdelingen voor mannen en vrouwen (bijvoorbeeld **procmen** versus **procwomen**) vrijwel hetzelfde zijn.

Omdat het bereik van de diverse (verklarende) variabelen zover uiteenloopt, en om aan het *squashing problem* tegemoet te komen is het absoluut noodzakelijk dat we de data normaliseren.

We doen dat door, voor iedere variabele, de werkelijke waarde ten opzichte van het minimum te berekenen, en dat te delen door het bereik.

Om veel typewerk te vermijden, gebruiken we weer een zelfgeschreven functie, **`normalize()`**.

```
normalize<-function(x){
  return((x-min(x))/(max(x)-min(x)))
}
```

Als voorbeeld van normaliseren: stel dat de examencijfers van alle leerlingen in een klas liggen tussen 4 (minimum) en 9 (maximum). Een leerling met een 6, krijgt dan de genormaliseerde score (6-4)/(9-4)=0.40. Een leerling met een 4 krijgt een genormaliseerde score van 0, en een leerling met een 9, een genormaliseerde score van 1. Genormaliseerde scores lager dan 0 of hoger dan 1 kunnen per definitie niet voorkomen.

Maar let op: het is van groot belang dat de normalisering wordt toegepast op de complete (training- plus test-) dataset! Het minimum en het maximum (en dus het bereik) in de twee sets kunnen immers verschillen! Dus de regel is: eerst normaliseren, en dan pas splitsen in training- en testset!

We passen die functie toe op **eodb**. We gebruiken weer de **`lapply()`** functie, om de functie in een keer toe te passen op alle variabelen in het dataframe. Om van de uitkomst weer een dataframe (in plaats van een matrix) te maken, gebruiken we de **`as.data.frame()`** functie.

Uit de **`summary()`** blijkt dat, inderdaad, alle genormaliseerde variabelen een waarde hebben tussen 0 en 1.

```
> eodb.n <- as.data.frame(lapply(eodb, normalize))
> summary(eodb.n)
     score            procmen           timemen           costmen       
Min.   :0.0000   Min.   :0.0000   Min.   :0.00000   Min.   :0.000000  
1st Qu.:0.3333   1st Qu.:0.1711   1st Qu.:0.02669   1st Qu.:0.009234  
Median :0.5556   Median :0.3158   Median :0.05447   Median :0.044004  
Mean   :0.5379   Mean   :0.3092   Mean   :0.08972   Mean   :0.106513  
3rd Qu.:0.7778   3rd Qu.:0.4171   3rd Qu.:0.10741   3rd Qu.:0.115025  
Max.   :1.0000   Max.   :1.0000   Max.   :1.00000   Max.   :1.000000  
   procwomen         timewomen         costwomen           capital       
Min.   :0.0000   Min.   :0.00000   Min.   :0.000000   Min.   :0.00000  
1st Qu.:0.2105   1st Qu.:0.02832   1st Qu.:0.009234   1st Qu.:0.00000  
Median :0.3158   Median :0.05447   Median :0.044004   Median :0.00000  
Mean   :0.3159   Mean   :0.09027   Mean   :0.106525   Mean   :0.02122  
3rd Qu.:0.4211   3rd Qu.:0.10741   3rd Qu.:0.115025   3rd Qu.:0.00786  
Max.   :1.0000   Max.   :1.00000   Max.   :1.000000   Max.   :1.00000  
```

Vervolgens splitsen we de dataset in een trainingset en een testset. We gebruiken 75% van alle records voor training, en de overige 25% voor de test. Om de stabiliteit van de uitkomsten te meten kun je experimenteren met hogere of lagere percentages.

Omdat we niet zeker weten of, en hoe de data gesorteerd zijn, trekken we een willekeurige steekproef van 75% van het aantal records in de dataset. Bij het selecteren van de eerste 75% van alle records lopen we het gevaar dat de trainingset niet representatief is voor het geheel.

De **`sample()`** functie werkt met aantallen (en niet met percentages). We bepalen daarom eerst het aantal records waaruit we eens steekproef trekken (**x**), en de omvang van de steekproef (**xs**). Om een afgerond geheel getal te krijgen, gebruiken we de **`ceiling()`** functie. Door de commando's tussen haakjes te plaatsen, worden de resultaten meteen afgedrukt in de console.

```
(x  <- nrow(eodb.n))
(xs <- ceiling(.75*x))
```

Om ervoor te zorgen dat je dezelfde steekproef als en dezelfde resultaten krijgt, gebruiken we het **`set.seed()`** commando. Een ander getal dan het hier gebruikte getal (765) levert een andere steekproef op.

Vervolgens genereren we, met **`sample()`** een steekproef **train**. De steekproef is een vector van **xs** trekkingen uit de populatie van **x** records. We kunnen de structuur van de vector (de lengte, en de eerste records) weergeven met **`str()`**.

```
set.seed(765)
train <- sample(x, xs)
```

De resultaten zijn:

```
> (x  <- nrow(eodb.n))
[1] 182
> (xs <- ceiling(.75*x))
[1] 137
> set.seed(765)
> train <- sample(x, xs)
```

```
> str(train)
 int [1:137] 84 112 138 134 124 72 139 73 82 7 ...
```

Nu we de steekproef (in de vector **train**) hebben, kunnen we onze training- en testset definiëren.

We indiceren het genormaliseerde bestand **eodn.n**, met de vector **train**, voor de training set. En voor de test set indiceren we **eodn.n** met alles wat niet in de steekproef valt (simpelweg, "–**train**").

Het is altijd goed om, met **`summary()`** en/of **`str()`**, te kijken of alles werkt zoals we verwachten!

```
> eodb_train <- eodb.n[train,]
> eodb_test  <- eodb.n[-train,]
> str(eodb_train); str(eodb_test)
'data.frame':  137 obs. of  8 variables:
 $ score    : num  0.222 0.444 0.778 0.778 0.333 ...
 $ procmen  : num  0.579 0.316 0.211 0.158 0.316 ...
 $ timemen  : num  0.2636 0.0719 0.0283 0.037 0.1503 ...
 $ costmen  : num  0.0128 0.119 0.0296 0.0351 0.1815 ...
 $ procwomen: num  0.632 0.316 0.211 0.158 0.316 ...
 $ timewomen: num  0.268 0.0719 0.0283 0.037 0.1503 ...
 $ costwomen: num  0.0128 0.119 0.0296 0.0351 0.1815 ...
 $ capital  : num  0.0183 0 0 0 0 ...
'data.frame':  45 obs. of  8 variables:
 $ score    : num  0.222 0.444 0.556 0.222 0.222 ...
 $ procmen  : num  0.579 0.368 0.368 0.526 0.211 ...
 $ timemen  : num  0.085 0.0915 0.0632 0.3442 0.0632 ...
 $ costmen  : num  0.0506 0.0629 0.0173 0.0237 0.1459 ...
 $ procwomen: num  0.579 0.368 0.368 0.526 0.263 ...
 $ timewomen: num  0.085 0.0915 0.0632 0.3442 0.0675 ...
 $ costwomen: num  0.0506 0.0629 0.0173 0.0237 0.1477 ...
 $ capital  : num  0 0 0 0 0.247 ...
```

Stap 3: Het trainen van het model

Voor het trainen van het model zijn er diverse *packages* beschikbaar. Ze verschillen in de algoritmes die worden gebruikt om de gewichten te schatten.

Het waarschijnlijk vaakst gebruikte *package* is **nnet**, dat onderdeel is van standaard **R**. Dat wil zeggen, het bevindt zich in de *system library* van **R**. Het hoeft dus niet afzonderlijk geïnstalleerd te worden met **`install.packages()`**, maar de functies van **nnet** moeten wel worden opgeroepen met **`library()`**.

Alternatieve *packages* voor artificiële neurale netwerken zijn **neuralnet**, en **RSNSS**.

Het eerstgenoemde *package*, **neuralnet**, werkt in veel gevallen erg traag, en biedt – afgezien van grafische weergave van het model – geen bijzondere voordelen ten opzichte van **nnet**.

Het **RSNSS** *package* biedt wel uitgebreidere mogelijkheden maar is moeilijk te leren. Binnen **nnet** is er geen optie om meer dan één *hidden layer* toe te voegen. Volgens de vele discussies over ANNs (zie bijvoorbeeld http://www.faqs.org/faqs/ai-faq/neural-nets/, of http://stats.stackexchange.com/), is de conclusie dat in vrijwel alle gevallen een *multilayer* netwerk met één *hidden layer* afdoende is. Meerdere *hidden layers* dragen zelden bij aan een verbetering van de resultaten.

We zullen om al deze redenen gebruikmaken van **nnet**.

We zeiden al dat ANNs enigszins verwant zijn aan regressieanalyse. We zien dat terug in de structuur van de **`nnet()`** functie. We maken een object aan (hier **eodb_model**), met de functie **`nnet()`**. In die functie beginnen we, tussen de haakjes, met een formule die lijkt op de formule in de **`lm()`** functie voor regressieanalyse.

- De tilde (“~”) betekent “wordt verklaard door”;
- We zetten links van de tilde, de te verklaren (output) variabele (**score**);
- Rechts van de tilde komen de verklarende (input) variabelen, gescheiden door “+”;
- Met **`data`** geven we de naam van de set met data die we voor de training van het model gaan gebruiken;
- Vervolgens kunnen we een aantal opties gebruiken waarvan de belangrijkste **`size`** is. In deze opties geven we het aantal knooppunten (*nodes*) in de *hidden layer* aan;
- Met **`linout=TRUE`** geven we aan dat de output is gemeten op een numerieke (lineaire) schaal. De *default* waarde is een “logistieke” schaal met de outputwaarden 0 of 1.

In de output die verschijnt in de console, is de belangrijkste informatie het feit dat de schattingen convergeren (*converged*). De zoektocht naar de beste gewichten begint met willekeurige startwaardes waarna het algoritme de waarden slim gaat bijstellen net zolang totdat er geen betere gewichten meer te vinden zijn.

Soms convergeert het model niet, of althans niet binnen het maximale aantal iteraties. Een hoger aantal iteraties (met **`maxiter`**), kan dan een oplossing zijn.

Het vinden van goede gewichten lukt vaak niet als de data niet genormaliseerd zijn. Ook wanneer gebruik wordt gemaakt van standaardisering in plaats van normalisering, kunnen er problemen ontstaan als de data zeer scheef (asymmetrisch; niet-normaal) zijn verdeeld. In onze ervaring verdient normaliseren de voorkeur!

Soms is er ogenschijnlijk geen probleem: zonder enige output of foutmelding (en zonder iteraties) is het neurale netwerk geschat. Dit wijst er echter op dat de startwaarden voor de gewichten zijn geaccepteerd als de beste waarden. Het is “theoretisch” niet uitgesloten dat deze willekeurige waardes meteen de beste zijn, maar in bijna alle gevallen wijst dit op een probleem met de data (en meestal op data die niet zijn genormaliseerd).

```
> library(nnet)
> eodb_model <- nnet(score ~ procmen + timemen + costmen +
+                    procwomen + timewomen + costwomen + capital,
+                    data = eodb_train, maxit=10000, size=1, linout=TRUE
)
# weights:  10
initial  value 94.541416
iter  10 value 10.210761
iter  20 value 2.588115
```

[output weggelaten]

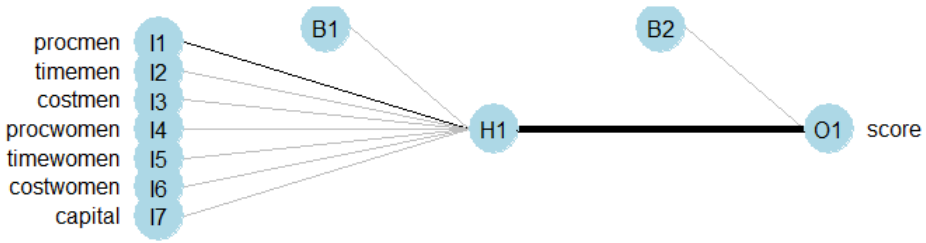
```
iter 150 value 0.701431
final  value 0.701430
converged

> summary(eodb_model)
a 7-1-1 network with 10 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1 i7->h1
  0.59   2.32  -2.78  -2.05  -4.66  -2.84  -2.25  -3.57
 b->o h1->o
 0.05  1.63
```

De output van **`summary()`** van het model is op het eerste gezicht niet erg behulpzaam. Allereerst wordt gerapporteerd hoeveel gewichten (*weights*) zijn geschat. Vervolgens zien we een overzicht van de gewichten die horen bij de “paden” in ons neurale netwerk. Aangezien we 7 input knooppunten hebben, zullen **i1** t/m **i7** wel betrekking hebben op de gewichten van deze

knooppunten op het enkele knooppunt (**size=1**) in het neurale netwerk, maar een grafische presentatie zou een welkome aanvulling zijn.

Zo'n grafische voorstelling van het model kan worden verkregen met het *package* **devtools**. Met **`plot()`** kunnen we dan het model afdrukken (maar zonder de gewichten). Omdat neurale netwerken vaak een zeer groot aantal knooppunten bevat, is het gebruikelijk de gewichten niet weer te geven. In een *blackbox* model is uiteindelijk ook de werking van het model belangrijker dan de interpretatie van de gewichten! Let op: de **`source-url()`** regel moet ook worden gerund!

```
#import the function from Github
# install.packages("devtools")
library(devtools)
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c
1474d0a505ff044412703516c34f1a4684a5/nnet_plot_update.r')
plot(eodb_model)
```



Figuur 8: de structuur van het ANN. Bron: OGN.

Met de plot is het gemakkelijker om de output te interpreteren. De **i1** t/m **i7** hebben inderdaad betrekking op de input knooppunten, en **h1** is het knooppunt in de *hidden layer*. De gewichten **b** in de output, komen overeen met **B1** en **B2** in de figuur. Deze gewichten hebben dezelfde rol als de constante in een regressiemodel, en zijn minder interessant.

Stap 4: het evalueren van het model

Aangezien het gaat om een *blackbox model* zijn we vooral geïnteresseerd in het vermogen van het model om de goede voorspellingen te doen in een testset met die data die niet zijn gebruikt in het ontwikkelen van het model.

Voor het testen van het model zijn slechts twee eenvoudige stappen nodig.

In de eerste stap passen we het model toe op de test set (in **eodb_test**). Dat doen we met de **`predict()`** functie, waarin we tussen haakjes de naam van het model en van de test set aangeven. We schrijven het resultaat weg naar een vector (**x**). Het resultaat is een vector met dezelfde lengte (aantal records) als die van de test set (**eodb_test**), in dit geval dus 45. We kunnen deze 45 voorspellingen vergelijken met de werkelijke scores in de test set, die zijn opgeslagen in de variabele **score**. Omdat het gaat om numerieke waarden, ligt het voor de hand om de correlatie tussen deze waarden te berekenen, met de **`cor()`** functie.

```
> cor(x,eodb_test$score)
          [,1]
[1,] 0.9708145
```

De correlatie tussen de werkelijke en voorspelde waarden voor de score, in de testset, is bijzonder goed, met 0.97. Dat betekent dat we met een beperkte set van gegevens tot een bijna perfecte voorspelling kunnen komen!

Stap 5: verbeteren van het model

Gegeven het goede resultaat, met een correlatie tussen werkelijke en voorspelde waarden van 0.97, is het nauwelijks mogelijk om nog te verbeteren. Maar zeker als het resultaat minder goed is, kunnen we experimenteren met andere waarden voor het aantal knooppunten in de *hidden layer*. Er is geen duidelijke regel – of zelfs maar een vuistregel – voor het beste aantal knooppunten in de *hidden layer*, anders dan dat het uiteenloopt van het aantal *output nodes* (hier 1) en het aantal *input nodes* (hier 7).

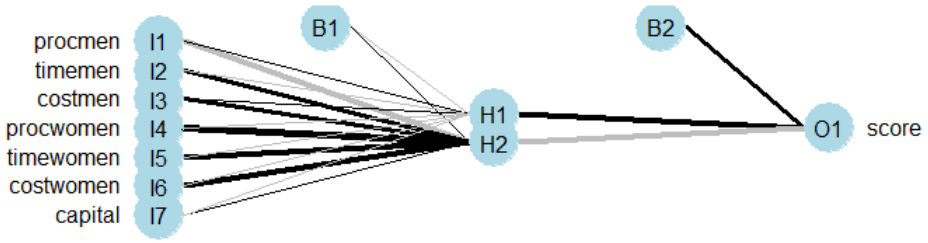
Wat we kunnen doen is de correlaties berekenen, als we de **size** laten variëren tussen 1 en 7.

Natuurlijk kunnen we, zoals hieronder, het script voor **size=1** kopiëren, een aanpassen met andere waarden voor **size**.

```
> # size = 2
> eodb_model <- nnet(score ~ procmen + timemen + costmen +
+                      procwomen + timewomen + costwomen + capital,
+                    data = eodb_train, maxit=10000, size=2)
# weights:  19
initial  value 14.681216
iter  10 value 1.367160
iter  20 value 0.781868
```

[output weggelaten]

```
iter 610 value 0.593788
iter 620 value 0.593768
final  value 0.593763
converged
> summary(eodb_model)
a 7-2-1 network with 19 weights
options were -
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1 i7->h1
 -0.36   0.03  -0.69   1.68  -1.09  -0.91  -2.92  -1.72
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2 i7->h2
  1.50 -17.04  11.51  13.38  17.95  13.99  14.69   2.83
  b->o  h1->o  h2->o
 11.27  14.90 -15.60
> plot(eodb_model)
```



Figuur 9. de structuur van het ANN (met size=2). Bron: OGN.

```
> x<-predict(eodb_model, eodb_test)
> cor(x,eodb_test$score)
          [,1]
[1,] 0.9715117
```

De correlatie is iets hoger dan voor **size=1**.

Handiger is het om gebruik te maken van een *for loop*. Omdat we niet geïnteresseerd zijn in het verloop van het iteratieproces, geven we als optie aan om de iteraties te onderdrukken, met de **`trace=FALSE`** optie.

```
> for (x in c(1:7)){
+   m <- nnet(score ~ procmen + timemen + costmen +
+                     procwomen + timewomen + costwomen + capital,
+                     data = eodb_train, maxit=10000,
+                     size=x, trace=FALSE)
+   p <- predict(m, eodb_test)
+   c <- cor(p,eodb_test$score)
+   cat("Corr(size=",x,")",round(c,3),"\n")
+ }
Corr(size= 1 ) 0.971
Corr(size= 2 ) 0.972
Corr(size= 3 ) 0.969
Corr(size= 4 ) 0.615
Corr(size= 5 ) 0.854
Corr(size= 6 ) 0.955
Corr(size= 7 ) 0.68
```

De modellen met **size=1, 2** of **3**, geven nagenoeg gelijke prestaties. Omdat de prestaties ook afhankelijk kunnen zijn van de steekproef, is het aan te bevelen om de exercitie te herhalen met een andere steekproef (door een andere waarde aan te geven in **`set.seed()`**. Probeer dit zelf uit!

Als **size=2** steevast tot betere resultaten leidt dan is het te overwegen om dat model als uitgangspunt te nemen. Als de resultaten wisselen per steekproef, dan gebruiken we meestal het principe van spaarzaamheid: simpele modellen – hier, met minder *nodes* in de *hidden layer* – verdienen de voorkeur.

Overigens is het interessant om voor dezelfde dataset ook eens regressieanalyse te proberen. In de door ons gebruikte steekproeven werken ANNs systematisch beter dan regressiemodellen, wat waarschijnlijk komt door de niet-normaal verdeelde (verklarende) variabelen. Het is wel mogelijk de regressiemodellen te verbeteren door het toepassen van allerlei transformaties op de verklarende variabelen, maar dat maakt de analyse een stuk complexer. Artificiële neurale netwerken kunnen beter overweg met allerlei soorten die niet normaal verdeeld zijn.

Neurale netwerken: voor groepen

Net zoals regressieanalyse met de nodige aanpassingen kan worden toegepast in situaties waarin de verklarende variabelen en/of de te verklaren variabele niet-numeriek zijn maar betrekking hebben op groepen, is het ook voor neurale netwerken mogelijk te werken met categoriale variabelen.

Een voorbeeld van zo'n toepassing hebben we eigenlijk al geïntroduceerd aan het begin van deze les: handschriftherkenning. De geschreven letters en cijfers (a t/m z, en 0 t/m 9) zijn de bouwstenen die we kunnen onderscheiden.

Als voorbeeld voor het gebruik van ANNs en andere technieken, hebben Frey & Slate in 1991 een data set ontwikkeld met 20,000 letters, die vervolgens beschreven zijn aan de hand van zestien kenmerken (voor details, zie https://archive.ics.uci.edu/ml/datasets/Letter+Recognition).

Stap 1: inlezen van de data

We gaan deze dataset inlezen, en dan weer een ANN bouwen met dezelfde **`nnet()`** functie in het gelijknamige *package*) die we hiervoor hebben gebruikt. Wel moeten we enkele opties aanpassen aan de nieuwe situatie met een categoriale te verklaren variabele(de 26 letters van het alfabet).

En even belangrijk, we zullen ingaan op maatstaven die we kunnen gebruiken om het model te evalueren. Dat is nodig omdat de eenvoudige maatstaf (correlatie) die we hiervoor hebben gebruikt voor numerieke variabelen, niet kan worden gebruikt voor categorale variabelen.

```
> letter <- read.csv("letterdata.csv", header=TRUE)
> str(letter)
'data.frame':  20000 obs. of  17 variables:
 $ letter: Factor w/ 26 levels "A","B","C","D",..: 20 9 4 14 7 19 2 1 10
13 ...
 $ xbox  : int  2 5 4 7 2 4 4 1 2 11 ...
 $ ybox  : int  8 12 11 11 1 11 2 1 2 15 ...
 $ width : int  3 3 6 6 3 5 5 3 4 13 ...
 $ height: int  5 7 8 6 1 8 4 2 4 9 ...
 $ onpix : int  1 2 6 3 1 3 4 1 2 7 ...
 $ xbar  : int  8 10 10 5 8 8 8 8 10 13 ...
 $ ybar  : int  13 5 6 9 6 8 7 2 6 2 ...
 $ x2bar : int  0 5 2 4 6 6 6 2 2 6 ...
 $ y2bar : int  6 4 6 6 6 9 6 2 6 2 ...
 $ xybar : int  6 13 10 4 6 5 7 8 12 12 ...
 $ x2ybar: int  10 3 3 4 5 6 6 2 4 1 ...
 $ xy2bar: int  8 9 7 10 9 6 6 8 8 9 ...
 $ xedge : int  0 2 3 6 1 0 2 1 1 8 ...
 $ xedgey: int  8 8 7 10 7 8 8 6 6 1 ...
 $ yedge : int  0 4 3 2 5 9 7 2 1 1 ...
 $ yedgex: int  8 10 9 8 10 7 10 7 7 8 ...
> summary(letter)
     letter           xbox             ybox            width
 U      :  813   Min.   : 0.000   Min.   : 0.000   Min.   : 0.000
 D      :  805   1st Qu.: 3.000   1st Qu.: 5.000   1st Qu.: 4.000
 P      :  803   Median : 4.000   Median : 7.000   Median : 5.000
 T      :  796   Mean   : 4.024   Mean   : 7.035   Mean   : 5.122
 M      :  792   3rd Qu.: 5.000   3rd Qu.: 9.000   3rd Qu.: 6.000
 A      :  789   Max.   :15.000   Max.   :15.000   Max.   :15.000
 (Other):15202
```

[output weggelaten]

Merk op dat de eerste variabele in het bestand een factor variabele is (**letter$letter**), die 26 niveaus heeft (de 26 letters van het alfabet).

De 16 verklarende variabelen zijn gehele getallen, allemaal gemeten op een schaal van 1 tot 15. De verklarende variabelen hebben betrekking op de hoogte, de breedte, de spreiding, en dergelijke van de "gemeten" letters. De vraag is nu in hoeverre we met deze "statistische" metingen de 26 letters kunnen duiden.

De werkwijze is hetzelfde als in het eerste voorbeeld:

1. We gaan de data (voor zover relevant) normaliseren;
2. We maken een training- en een testset;
3. We trainen het model op een deel van de data (de training set);

4. We testen het model op de overige data (de training set);
5. We evalueren het model, en zoeken naar mogelijke verbeteringen.

Stap 2: normaliseren van de data

Het normaliseren van de data hoeft alleen maar voor de 16 verklarende variabelen. De te verklaren variabele is een factor variabele (26 letters), en die kunnen niet worden genormaliseerd.

Het is overzichtelijk om dit te doen in stapjes. Eerst maken we een data frame met de genormaliseerde 16 verklarende variabelen (in de kolommen 2 t/m 17). En vervolgens plakken we daaraan vast de te verklaren factor variabele.

Misschien hebt u de **`normalize()`** functie nog niet verwijderd, en kunt u hem direct gebruiken. Zo niet, dan moet u hem opnieuw definiëren.

Vervolgens definiëren we **l1** als de eerste kolom uit het data frame **letter**. Daarna voegen we **l1** en de genormaliseerde variabelen in **l.n** samen tot **l2**. Het is verstandig te checken of inderdaad alle (17) variabelen in het nieuwe bestand staan, en of ze de juiste waarden hebben (tussen 0 en 1, voor de genormaliseerde variabelen).

```
> l.n <- as.data.frame(lapply(letter[,2:17], normalize))
> str(l.n)
'data.frame':  20000 obs. of  16 variables:
 $ xbox  : num  0.133 0.333 0.267 0.467 0.133 ...
 $ ybox  : num  0.5333 0.8 0.7333 0.7333 0.0667 ...
 $ width : num  0.2 0.2 0.4 0.4 0.2 ...
 $ height: num  0.3333 0.4667 0.5333 0.4 0.0667 ...
 $ onpix : num  0.0667 0.1333 0.4 0.2 0.0667 ...
 $ xbar  : num  0.533 0.667 0.667 0.333 0.533 ...
 $ ybar  : num  0.867 0.333 0.4 0.6 0.4 ...
 $ x2bar : num  0 0.333 0.133 0.267 0.4 ...
 $ y2bar : num  0.4 0.267 0.4 0.4 0.4 ...
 $ xybar : num  0.4 0.867 0.667 0.267 0.4 ...
 $ x2ybar: num  0.667 0.2 0.2 0.267 0.333 ...
 $ xy2bar: num  0.533 0.6 0.467 0.667 0.6 ...
 $ xedge : num  0 0.1333 0.2 0.4 0.0667 ...
 $ xedgey: num  0.533 0.533 0.467 0.667 0.467 ...
 $ yedge : num  0 0.267 0.2 0.133 0.333 ...
 $ yedgex: num  0.533 0.667 0.6 0.533 0.667 ...
> l1 <- letter[,1]; head(l1); str(l1); length(l1)
[1] T I D N G S
Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 Factor w/ 26 levels "A","B","C","D",..: 20 9 4 14 7 19 2 1 10 13 ...
[1] 20000
> l2 <- cbind.data.frame(l1,l.n)
> str(l2)
'data.frame':  20000 obs. of  17 variables:
 $ l1    : Factor w/ 26 levels "A","B","C","D",..: 20 9 4 14 7 19 2 1 10
13 ...
 $ xbox  : num  0.133 0.333 0.267 0.467 0.133 ...
 $ ybox  : num  0.5333 0.8 0.7333 0.7333 0.0667 ...
 $ width : num  0.2 0.2 0.4 0.4 0.2 ...
 $ height: num  0.3333 0.4667 0.5333 0.4 0.0667 ...
 $ onpix : num  0.0667 0.1333 0.4 0.2 0.0667 ...
 $ xbar  : num  0.533 0.667 0.667 0.333 0.533 ...
 $ ybar  : num  0.867 0.333 0.4 0.6 0.4 ...
 $ x2bar : num  0 0.333 0.133 0.267 0.4 ...
 $ y2bar : num  0.4 0.267 0.4 0.4 0.4 ...
 $ xybar : num  0.4 0.867 0.667 0.267 0.4 ...
 $ x2ybar: num  0.667 0.2 0.2 0.267 0.333 ...
```

```
 $ xy2bar: num  0.533 0.6 0.467 0.667 0.6 ...
 $ xedge : num  0 0.1333 0.2 0.4 0.0667 ...
 $ xedgey: num  0.533 0.533 0.467 0.667 0.467 ...
 $ yedge : num  0 0.267 0.2 0.133 0.333 ...
 $ yedgex: num  0.533 0.667 0.6 0.533 0.667 ...
```

Stap 3: training- en testset

De bedenkers van de dataset (Frey en Slate) splitsen de data in een trainingset van 16,000 letters (80%), en een testset van 4,000 letters (20%). Bij zoveel data zijn dat inderdaad de gebruikelijke getallen. Hier hebben we gebruikgemaakt van 75% en 25%. Het is altijd goed om zelf de procedure te herhalen met verschillende steekproeven van dezelfde omvang, en met steekproeven van een andere omvang!

De procedure is identiek aan die in het eerdere voorbeeld.

- We bepalen het aantal records, **y**, met, bijvoorbeeld, **`nrow()`**.
- Aan de hand daarvan bepalen we de steekproef **ys**, een afgerond geheel getal gelijk aan 75% van het totale aantal records **y**. Voor de reproduceerbaarheid gebruiken we **`set.seed()`**.
- Dan trekken we een steekproef **ltrain** (met 75%, of 15,000 records).

  De steekproef **ltrain** is een vector die we kunnen gebruiken als index, voor het bepalen van de training- en de testset.

Natuurlijk controleren we steeds of het resultaat is zoals verwacht! Een vergissing is snel gemaakt.

```
> (y <- nrow(l2))
[1] 20000
> (ys <- ceiling(.75*y))
[1] 15000
> set.seed(2345432)
> ltrain <- sample(y, ys)
> str(ltrain)
 int [1:15000] 11829 10132 12039 1968 9699 9485 17442 2205 14788 15930 .
..
> l2_train <- l2[ltrain,]
> l2_test  <- l2[-ltrain,]
> str(l2_train); str(l2_test)
'data.frame':  15000 obs. of  17 variables:
 $ ll    : Factor w/ 26 levels "A","B","C","D",..: 23 22 15 21 25 6 26 1
22 13 ...
 $ xbox  : num  0.533 0.333 0.2 0.133 0.467 ...
 $ ybox  : num  0.8 0.6667 0.2667 0.0667 0.6667 ...
 $ width : num  0.6 0.333 0.267 0.133 0.467 ...

… [output weggelaten]

 $ yedgex: num  0.4 0.533 0.533 0.533 0.4 ...

'data.frame':  5000 obs. of  17 variables:
 $ ll    : Factor w/ 26 levels "A","B","C","D",..: 9 7 15 3 10 12 16 5 7
25 ...
 $ xbox  : num  0.333 0.267 0.2 0.467 0.133 ...
 $ ybox  : num  0.8 0.6 0.267 0.667 0.133 ...
 $ width : num  0.2 0.4 0.267 0.333 0.2 ...

… [output weggelaten]

$ yedgex: num  0.667 0.533 0.533 0.6 0.467 ...
```

Stap 4: Het trainen van het model

Voor het trainen van het ANN maken we weer gebruik van de **`nnet()`** functie. In de formule gebruiken we links van de tilde ("~") de factor variabele **ll** (de letters van het alfabet) als de variabele die we willen voorspellen. Achter de tilde kunnen we volstaan met een punt (".") die staat voor alle andere variabelen in de dataset.

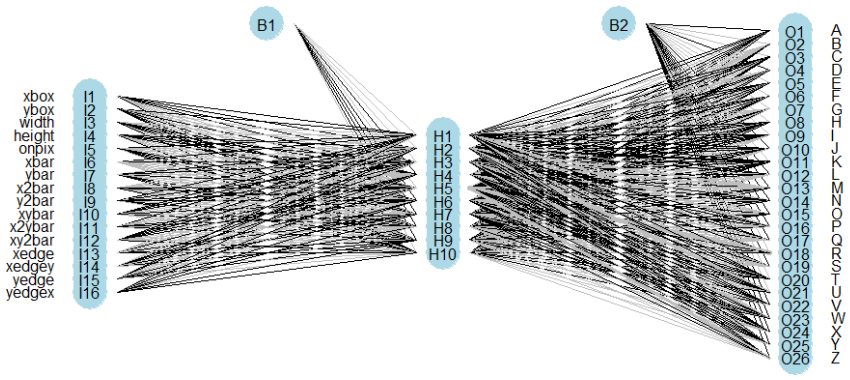
De **`nnet()`** functie ziet dat de te verklaren variabele een factor variabele is, en interpreteert de 26 *levels* als even zoveel outputsignalen. We hebben dus een model met 16 inputsignalen en 26 outputsignalen. In een situatie met zoveel input- en outputsignalen is het uitzonderlijk dat een ANN met slechts één knooppunt in de *hidden layer* een goed resultaat geeft. Enigszins arbitrair zouden we kunnen beginnen met de 10 knooppunten (ongeveer de helft van het minimum aantal input- of outputsignalen), en daarna kijken of een kleiner of groter aantal knooppunten betere resultaten geeft.

Het model wordt nu complexer, en u zult merken dat het model niet convergeert na het *default* maximale aantal iteraties. We kunnen het maximale aantal iteraties aanpassen met **`maxiter`**. Het model convergeert na honderden iteraties (afhankelijk van de startwaarden). In totaal moeten er 456 gewichten worden geschat:

- 16 verklarende variabelen (inputsignalen) met elk een gewicht op 10 knooppunten in de *hidden layer*, geeft **160** gewichten;
- 10 knooppunten met elk een gewicht op 26 outputsignalen, is **260** gewichten;
- 10 plus 26, dus **36 bias terms** (de B's; vergelijkbaar met de constanten in regressievergelijkingen).

Dit vertaalt zich in een enorme output, die – mede gegeven het *blackbox* karakter – moeilijk te interpreteren is. Het is ongebruikelijk om al die gewichten in het schema op te nemen. Maar het weergeven van de structuur van het schema, zonder de gewichten, is altijd nuttig.

We kunnen die weer oproepen met de **`plot()`** functie vanuit het *package* **devtools** (zie de commando's in het script).



Figuur 10. de structuur van het ANN (met 26 levels). Bron: OGN.

```
> library(nnet)
> l_model <- nnet(l1 ~ ., data = l2_train, maxit=10000,
+                 size=10, linout=FALSE)
# weights:  456
initial  value 54930.153079
iter  10 value 39780.036633
iter  20 value 31600.233467
```

[output weggelaten]

```
iter1540 value 9881.538941
iter1550 value 9881.396404
converged
```

```
> summary(l_model)
a 16-10-26 network with 456 weights
options were - softmax modelling
   b->h1   i1->h1   i2->h1   i3->h1   i4->h1   i5->h1   i6->h1   i7->h1
   -3.78    -0.95     0.73    -0.54    -0.75    -0.27    -0.05     0.00
  i8->h1   i9->h1  i10->h1  i11->h1  i12->h1  i13->h1  i14->h1  i15->h1
   -3.50     4.12    -0.96     0.44     0.46    -1.33    -0.90     1.55
 i16->h1
    0.65
   b->h2   i1->h2   i2->h2   i3->h2   i4->h2   i5->h2   i6->h2   i7->h2
    0.94    -1.50    -0.03     3.99     2.01    -5.93    -1.49     1.86
  i8->h2   i9->h2  i10->h2  i11->h2  i12->h2  i13->h2  i14->h2  i15->h2
   -1.25    -0.45    -2.72     0.14     0.43    -2.33    -1.05    -4.84
```

[output weggelaten]

```
h8->o25  h9->o25 h10->o25
  -57.18    -7.58   127.30
  b->o26  h1->o26  h2->o26  h3->o26  h4->o26  h5->o26  h6->o26  h7->o26
   10.87   329.06   -30.11   -81.24   131.01    -7.43    50.92   124.14
 h8->o26  h9->o26 h10->o26
  228.47   -19.95   265.81
```

Stap 5: Evalueren van het model

Voor het evalueren van het model maken we weer gebruik van de **`predict()`** functie. Hierin geven we aan dat we te maken met groepen (of categorieën of klassen), met de optie **`type="class"`**.

Omdat de te verklaren variabele een (categoriale) factor is, maken we een tabel waarin de voorspelde waarde en de werkelijke waarde tegen elkaar worden afgezet. Die tabel heeft dus 26 rijen en 26 kolommen, voor alle letters van het alfabet. De hoop is dat alle (5,000) records op de "diagonaal" van de tabel staan: die geeft immers aan dat de voorspelde letter gelijk is aan de werkelijke letter.

In het kleine stukje van de tabel dat we hieronder hebben weergegeven zien wij bijvoorbeeld dat in **161** een "A" juist voorspeld is. Maar ook zien we (de **rood** afgedrukte getallen) dat het model in een aantal gevallen een "A" voorspelt terwijl de werkelijke letter een andere is. Bijvoorbeeld, in 8 gevallen wordt een "Q" voorspeld als een "A". Het kan natuurlijk ook voorkomen dat een "A" wordt voorspeld als een andere letter.

Check dat voor uzelf aan de hand van de volledige output in de console! Uit de tweede rij, voor de letter "B", lijkt het aantal misclassificaties groter.

> *Noot: het aantal iteraties en de uiteindelijke oplossing is deels afhankelijk van de startwaarden. Door willekeurige keuzes in het algoritme, kunnen de oplossingen steeds iets anders zijn!*

```
> lp<-predict(l_model, l2_test, type="class")
> ltable <- table(lp,l2_test$ll); ltable

lp    A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R
  A 161   0   0   1   0   0   0   1   0   3   0   0   0   0   0   0   8   0
  B   0 165   0  14   1   5   3   4   3   0   7   0   3   3   0   2   4   6
  C   1   0 160   1   0   1   8   1   0   1   3   0   0   0   1   0   2   0
```

[output weggelaten]

In het geval van numerieke variabelen hadden we met één simpele maatstaf (de correlatie) voldoende om de voorspelkracht van het model compact samen te vatten. Maar in dit geval is dat veel lastiger.

We zullen het *package* **caret** introduceren dat de belangrijkste maatstaven voor het evalueren van het model, aan de hand van de tabel, bevat. Deze maatstaven worden berekend met een zogenoemde *confusion matrix*. Het principe van zo'n matrix is eigenlijk heel eenvoudig.

In de matrix maken we zowel voor de voorspelling als voor de werkelijke uitkomst, een tweedeling in "positief" en "negatief". Natuurlijk hebben we in ons voorbeeld niet twee maar liefst 26 categorieën, maar we kunnen de exercitie categorie voor categorie (hier dus, letter voor letter) doen, te beginnen met de letter "A".

We vatten de 26*26 tabel eerst samen in een 2*2 tabel, met in de rijen en in de kolommen "A" en "niet-A" (dat is, B t/m Z).

		Werkelijk	
		A	**Niet-A**
Voorspeld	**A**	**TP**	**FP**
	Niet-A	**FN**	**TN**

De volgende gevallen kunnen zich voordoen:

- De werkelijke waarde "A" is juist voorspeld. We noemen dat "**True Positive**", of afgekort TP.
- De werkelijke waarde "niet-A" is juist voorspeld, als "niet-A". Dat heet "**True Negative**", of TN.
- Een werkelijke "niet-A" is incorrect voorspeld als "A": "**False Positive (FP)**".
- Een werkelijke "A" is incorrect voorspeld als "niet-A": "**False Negative (FN)**".

Dit zijn verwarrende begrippen. Ze zijn het gemakkelijkste te onthouden door de termen in tweeën te splitsen. De "**true**" en "**false**" delen hebben betrekking op de diagonalen in de tabel, terwijl "**positive**" en "**negative**" betrekking hebben op de rijen.

Ieder van de cellen is op zich interessant. Bovenal willen we weten in hoeveel gevallen een "A" juist wordt voorspeld (**TP**). Maar meestal maken we gebruik van maatstaven die de informatie in alle vier cellen combineert.

We zullen deze maatstaven bespreken, aan de hand van (een deel van) de output van de functie **`confusionMatrix()`** in het **caret** *package*.

Uit de 26*26 tabel (met alle letters in de rijen en kolommen) kunnen we zelf, handmatig of automatisch, 2*2 tabellen afleiden, zoals "A" versus alle andere letters. Voor de berekeningen van de maatstaven doet **`confusionMatrix()`** dat voor ons, maar zonder die tabellen te laten zien. Ter illustratie maken we de tabel voor "A" en "niet-A" handmatig. De 163 juiste **TP** classificaties zien we op de diagonaal voor rij en kolom "A". De **TF** zouden we kunnen berekenen door de overige getallen op de diagonaal op te tellen. Voor **FP** tellen we de getallen in de eerste rij, in de kolommen "B" t/m "Z" bij elkaar op. En voor **FN**, de getallen in de kolom "A", en rijen "B" t/m

“Z”. Omdat we het totale aantal records weten is het voldoende om drie van de vier cellen te kennen; de vierde ligt dan vast.

```
> lcm <- confusionMatrix(ltable); lcm
Confusion Matrix and Statistics

lp    A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R
  A 161   0   0   1   0   0   0   1   0   3   0   0   0   0   0   0   8   0
  B   0 165   0  14   1   5   3   4   3   0   7   0   3   3   0   2   4   6
  C   1   0 160   1   0   1   8   1   0   1   3   0   0   0   1   0   2   0
  D   0  10   0 187   1   4   1  10   5   2   0   0   0   3  10   1   1   7
  E   0   0   7   0 130   2   0   0   1   1   2   6   0   0   0   0   7   0
  F   0   0   0   0   0 153   0   1   4   0   0   0   0   0   1   9   0   0
  G   1   5   8   0   8   2 153   3   1   1   2   6   0   2   8   7   7   5
  H   2   1   2   4   1   4   2  98   0   7   8   0   0   6   5   3   2  19
  I   0   0   0   0   0   0   0   0 140   0   0   0   0   0   0   0   0   0
  J   3   0   0   1   0   0   0   3   4 157   0   0   0   0   1   1   0   0
  K   2   3   3   0   5   0   0  10   0   0 129   0   0   1   0   0   1   4
  L   2   0   0   0   2   0   0   0   1   0   2 145   1   0   0   0   2   1
  M   1   0   0   2   0   0   0   4   0   0   0   3 189   5   0   0   0   0
  N   0   0   0   2   0   3   0   3   0   0   1   0   6 175   0   0   0   4
  O   0   0   1   7   0   0   3   1   1   3   0   0   4   5 156   1  15   4
  P   0   0   0   0   0  12   7   1   1   2   0   0   0   0   1 179   1   3
  Q   5   0   1   0   4   0   6   6   3   6   1   5   0   0   5   0 107   1
  R   0   4   0   6   0   0   4  14   0   0   8   4   0   1   1   1   0 149
  S   0   8   0   0   8   4   5   0  16   3   0   2   0   0   0   0   7   0
  T   0   0   1   4   0   4   0   0   0   0   2   1   0   0   0   0   0   1
  U   1   0   1   2   0   0   0   3   0   0   7   0   3   1   3   0   0   0
  V   0   2   0   0   0   1   2   3   0   0   1   0   0   1   0   1   0   0
  W   0   0   0   0   0   1   0   0   0   0   0   0   4   3   4   0   0   5
  X   1   1   0   2   4   1   0   2   4   2   6   0   0   0   0   0   4   0
  Y   3   0   0   0   3   0   2   0   0   0   0   2   0   0   0   5   0   0
  Z   0   0   0   0   3   0   0   0   5   2   0   0   0   0   0   0   3   0

lp    S   T   U   V   W   X   Y   Z
  A   0   0   1   1   0   0   2   1
  B   7   4   0   3   2   1   0   0
  C   0   0   2   0   0   0   0   0
  D   3   2   1   0   0   1   1   0
  E  12   8   0   0   0   9   0   4
  F   5   4   0   0   0   1   1   1
  G   2   5   1   2   0   0   2   0
  H   0   1   0   0   0   1   0   1
  I   6   0   0   0   0   0   0   1
  J   2   0   0   0   0   2   4  13
  K   0   5   2   0   0   3   0   0
  L   3   3   0   0   0   4   1   0
  M   0   0   7   1  14   0   0   0
  N   0   0   4   0   5   0   0   0
  O   2   0   2   0   0   0   0   0
  P   8   0   0   2   0   0   1   0
  Q  16   0   0   0   0   4  10   0
  R   2   0   0   2   0   0   0   0
  S 103   1   0   0   0   2   0  10
  T   0 145   1   0   0   5   7   0
  U   0   1 183   0   0   0   1   0
  V   0   2   1 168   2   0   5   0
  W   0   0   0   3 155   0   2   0
  X   0   5   1   0   0 144   2   3
  Y  10   7   0  11   0   6 153   4
  Z   9   4   0   0   0   2   2 155

Overall Statistics

               Accuracy : 0.7878
                 95% CI : (0.7762, 0.7991)
    No Information Rate : 0.0466
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7793
 Mcnemar's Test P-Value : NA
```

```
Statistics by Class:

                     Class: A Class: B Class: C Class: D Class: E Class: F
Sensitivity            0.8798   0.8291   0.8696   0.8026   0.7647   0.7766
Specificity            0.9963   0.9850   0.9956   0.9868   0.9878   0.9944
Pos Pred Value         0.8994   0.6962   0.8840   0.7480   0.6878   0.8500
Neg Pred Value         0.9954   0.9929   0.9950   0.9903   0.9917   0.9909
Prevalence             0.0366   0.0398   0.0368   0.0466   0.0340   0.0394
Detection Rate         0.0322   0.0330   0.0320   0.0374   0.0260   0.0306
Detection Prevalence   0.0358   0.0474   0.0362   0.0500   0.0378   0.0360
Balanced Accuracy      0.9380   0.9071   0.9326   0.8947   0.8762   0.8855
```

[output weggelaten, voor letters G t/m Z]

De samengevatte 2*2 tabel, berekend met de geel gemarkeerde rij en kolom, voor "A", luidt:

		Werkelijk		
		A	**Niet-A**	**Totaal**
Voorspeld	**A**	**TP=161**	**FP=18**	**179**
	Niet-A	**FN=22**	**TN=4,799**	**4,821**
	Totaal	**183**	**4,817**	**5,000**

Onderaan in de output van `confusionMatrix()`, staat een groot aantal maatstaven.

Verreweg de belangrijkste, en meest gebruikte zijn, **sensitivity** en **specificity**.

- De **sensitivity** is gedefinieerd als het aantal werkelijke voorkomens van "A" dat door het model correct is voorspeld:

$$Sensitivity = \frac{TP}{TP + FN} = \frac{161}{161 + 22} = 0.8798$$

  Soms wordt ook wel de maatstaf **recall** gehanteerd, gedefinieerd als de compleetheid van de resultaten. De formule voor **recall** is gelijk aan die voor **sensitivity**.

- De **specificity** is de tegenhanger, voor "niet-A": het aantal voorkomens van "niet-A" dat juist wordt voorspeld:

$$Specificity = \frac{TN}{TN + FP} = \frac{4{,}799}{4{,}799 + 18} = 0.9963$$

- Omdat beide een maatstaf zijn voor de accuraatheid waarmee voorspeld wordt (de ene voor "A", en de andere voor "niet-A"), wordt ook weleens het gemiddelde van de twee genomen, als **balanced accuracy**: (0.8798+0.9963)/2 = 0.9380

Minder vaak gebruikte maatstaven zijn:

- De **precision** (of *positive predicted value*) geeft weer hoeveel van de "A"-voorspellingen correct zijn.

  Omdat **FP** in dit voorbeeld gelijk is aan **FN**, is de uitkomst gelijk aan die voor **sensitivity**, maar dat is louter toevallig.

$$Precision = \frac{TP}{TP + FP} = \frac{161}{161 + 18} = 0.8994$$

- De **F-score** is een "harmonisch" gemiddelde van **precision** en **recall** (gelijk aan **sensitivity**). De F-score is geen deel van de output van `confusionMatrix()`.

$$F\ Score = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * 0.8994 * 0.8798}{0.8994 + 0.8978} = 0.8806$$

- De **negative predicted value**, is de tegenhanger van **precision**, voor "niet-A".

$$Negative\ Predicted\ Value = \frac{TN}{TN + FN} = \frac{4{,}799}{4{,}799 + 22} = 0.9954$$

- De **prevalence** is het aantal werkelijke voorkomens van "A".

$$Prevalence = \frac{TP + FN}{TP + FN + FP + FN} = \frac{161 + 22}{5{,}000} = 0.0366$$

- De **detection prevalence** is het aantal keer dat "A" wordt voorspeld:

$$Detection\ Prevalence = \frac{TP + FP}{TP + FP + FN + TN} = \frac{161 + 18}{5{,}000} = 0.0358$$

- De **detection rate** is het aantal keer dat "A" correct wordt voorspeld:

$$Detection\ Rate = \frac{TP}{TP + FP + FN + TN} = \frac{161}{5{,}000} = 0.0322$$

De maatstaven hierboven hebben alle betrekking op een van de afzonderlijke categorieën (in dit geval "A").

Een vaak gebruikte overall maatstaf voor het model, is **accuracy**. De **accuracy** kijkt naar de diagonaal in de 26*26 tabel, voor alle groepen. Van de 5,000 gevallen wordt 78.78% juist geclassificeerd.

De **kappa** maatstaf is een variant die de **accuracy** maatstaf corrigeert voor de geluksfactor. Ook al werkt het model helemaal niet, dan zal immers toch in ongeveer 1 op de 26 gevallen de voorspelling juist zijn! Bij een grote test set en veel groepen, zoals hier, zal de **kappa** niet veel lager zijn dan **accuracy**. De waarde van **kappa** is 0.7793. Een prettige eigenschap is dat de **kappa** een algemeen geaccepteerde interpretatie heeft:

kappa < 0.20	Slecht (*poor*)
0.20 ≤ kappa < 0.40	Redelijk (*fair*)
0.40 ≤ kappa < 0.60	Matig (*moderate*)
0.60 ≤ kappa < 0.80	Goed (*good*)
kappa > 0.80	Zeer goed (*very good*)

Dat gezegd hebbende, is het altijd verstandig om de **accuracy** en **kappa** te interpreteren aan de accuraatheid die vereist is in de specifieke situatie. Als alle letters en cijfers juist worden herkend met - zoals hier - een kans van ongeveer 80%, dan is de kans dat een Nederlandse postcode met 4 cijfers en 2 letters juist wordt herkend $0{,}80^6$ (0,80 tot de macht 6, oftewel 0,80*0,80*0,80*0,80*0,80*0,80), is 26%! De automatische postsortering zal leiden tot chaos!

Stap 6: verbeteren van het model

Ten slotte willen we proberen tot een beter model te komen. We kunnen diverse dingen aanpassen, waarvan in een ANN het aantal knooppunten in de *hidden layer* het meest voor de hand ligt. De uitdaging is dat elke *run* veel tijd in beslag neemt, en een veelheid van output.

Voor verschillende aantallen knooppunten, kunnen we ons beperken tot het criterium **accuracy** (of **kappa**). Deze maatstaf is opgeslagen in de output van `confusionMatrix()`, maar waar? Om daarachter te komen, kijken we naar de structuur van het object **lcm**, met de `str()` functie.

We zien dat het object **lcm** een *list* is, met verschillende onderdelen die we met een "$" teken kunnen aanroepen. De voor ons interessante informatie blijkt te vinden te zijn in **lcm$overall**.

```
> str(lcm)
List of 6
 $ positive: NULL
 $ table   : 'table' int [1:26, 1:26] 163 1 0 0 0 0 0 1 0 4 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ lp: chr [1:26] "A" "B" "C" "D" ...
  .. ..$   : chr [1:26] "A" "B" "C" "D" ...
 $ overall : Named num [1:7] 0.8108 0.8032 0.7997 0.8216 0.0466 ...
  ..- attr(*, "names")= chr [1:7] "Accuracy" "Kappa" "AccuracyLower" "Ac
curacyUpper" ...
 $ byClass : num [1:26, 1:11] 0.891 0.809 0.902 0.79 0.735 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:26] "Class: A" "Class: B" "Class: C" "Class: D" ...
  .. ..$ : chr [1:11] "Sensitivity" "Specificity" "Pos Pred Value" "Neg
Pred Value" ...
 $ mode    : chr "sens_spec"
 $ dots    : list()
 - attr(*, "class")= chr "confusionMatrix"
```

We bekijken **lcm$overall**, door het in te typen – in de console of in het script. Zoals we hierboven ook al zagen, heeft **lcm$overall** zeven elementen, waarvan **accuracy** de eerste is, en **kappa** de tweede. De **accuracy** is te benaderen als **lcm$overall[1]**. We kunnen daarvan gebruik maken in een *for loop*.

```
> lcm$overall
      Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNu
ll
     0.7878000      0.7792514      0.7762006      0.7990624      0.04660
00
AccuracyPValue  McnemarPValue
     0.0000000            NaN
> lcm$overall[1]
Accuracy
  0.7878
> lcm$overall[2]
    Kappa
0.7792514
> lcm$overall[1:2]
 Accuracy     Kappa
0.7878000 0.7792514
```

We maken weer een *for loop*, met verschillende waarden voor het aantal knooppunten (**`size`**). We onderdrukken de iteraties in de console, en drukken alleen de **accuracy** af. We beginnen met **`size=5`** en laten die oplopen tot 15. We zetten – om rekentijd te besparen – het maximale iteraties op 100, in de verwachting dat na dat aantal iteraties de oplossing redelijk in de buurt komt van de oplossing na convergeren. Ook als dat niet zo is, dan zal het verloop van **accuracy** inzicht geven in een goed aantal knooppunten!

U ziet dat we het **`set.seed()`** commando opnemen. We doen dat om twee redenen. De eerste reden is om er voor te zorgen dat uw resultaten dezelfde zijn als die van ons. De tweede reden is dat we willen voorkomen dat verschillen in **accuracy** voor de aantallen *layers* wordt veroorzaakt door toevallige verschillen in startwaarden.

Het is aan te bevelen om de stabiliteit van de uitkomsten te testen door, zeg, drie verschillende startwaardes (getallen tussen haakjes, in het **`set.seed()`** commando) te proberen. Hier hebben we ons beperkt tot een startwaarde, aannemend dat de oplossing stabiel is!

```
> for (x in c(5:15))
+   {
+   set.seed(78987)
+   l_model <- nnet(ll ~ ., data = l2_train, maxit=100,
+                   size=x, linout=FALSE, trace=FALSE)
+   lp<-predict(l_model, l2_test, type="class")
+   xx <- table(lp,l2_test$ll)
+   cm <- confusionMatrix(xx)
+   cat("Accuracy (size=",x,"):",cm$overall[1],"\n")
+ }
Accuracy (size= 5 ): 0.5214
Accuracy (size= 6 ): 0.5806
Accuracy (size= 7 ): 0.6706
Accuracy (size= 8 ): 0.6586
Accuracy (size= 9 ): 0.6948
Accuracy (size= 10 ): 0.7332
Accuracy (size= 11 ): 0.7092
Accuracy (size= 12 ): 0.7052
Accuracy (size= 13 ): 0.756
Accuracy (size= 14 ): 0.7492
Accuracy (size= 15 ): 0.757
```

Het lijkt erop dat de prestaties van het model beter worden met een zo groot mogelijk aantal knooppunten. Als vuistregel nemen we niet meer knooppunten dan we inputsignalen (16) hebben. We besluiten het model met 15 knooppunten te nemen, en trainen en testen het model nog eens, maar nu met een groot maximum aantal iteraties.

```
> mf.cm    <- confusionMatrix(mf.table); mf.cm
Confusion Matrix and Statistics

model.final.predict   A   B   C   D   E   F   G   H   I   J   K   L   M   N
                  A 172   0   0   0   0   0   0   0   0   1   0   0   0   1
                  B   0 171   0  10   3   4   2   1   2   1   0   0   0   4
                  C   1   0 168   0   2   0   2   1   0   0   2   1   0   0
                  D   0   2   0 199   1   0   1   8   5   2   1   0   0   2
                  E   0   0   3   0 142   5   0   0   2   0   0   5   0   0
                  F   0   0   0   0   0 156   0   3   2   1   0   0   0   0
                  G   0   0   4   0   8   0 166   0   0   0   2   6   1   0
                  H   0   4   1   3   0   5   2 120   0   2   4   0   0   4
                  I   2   0   1   1   1   2   0   0 153   4   0   0   0   0
                  J   1   0   0   3   0   3   0   0  10 170   0   0   0   0
                  K   2   2   1   0   2   0   0   5   0   0 150   3   0   0
                  L   0   0   0   0   0   0   1   2   2   1   0 145   0   0
                  M   0   0   0   2   0   0   0   4   0   0   0   0 194   3
                  N   0   0   0   2   0   2   0   2   0   2   0   0   4 178
                  O   0   0   3   4   0   0   1   1   0   0   0   0   8   8
                  P   0   0   0   0   0  12   0   1   0   0   0   0   0   0
                  Q   1   0   0   0   5   0   7   1   0   0   0   3   0   0
                  R   1   9   1   2   0   0   2   9   0   0  12   3   0   0
                  S   2   4   0   0   1   2   6   0   5   4   0   0   0   0
                  T   0   2   1   4   0   5   0   0   5   0   1   3   0   0
                  U   0   0   0   1   0   0   1   1   0   0   2   0   0   0
                  V   1   2   1   0   0   1   5   5   0   0   0   0   0   3
                  W   0   0   0   0   0   0   0   0   0   0   0   0   3   3
                  X   0   2   0   1   1   0   0   4   1   2   5   5   0   0
                  Y   0   1   0   0   0   0   0   0   1   0   0   0   0   0
                  Z   0   0   0   1   4   0   0   0   1   0   0   0   0   0

model.final.predict   O   P   Q   R   S   T   U   V   W   X   Y   Z
                  A   0   0   1   0   1   0   1   0   0   0   0   0
```

[output weggelaten]

```
Overall Statistics
```

```
               Accuracy : 0.8518
                 95% CI : (0.8416, 0.8615)
    No Information Rate : 0.0466
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8458
 Mcnemar's Test P-Value : NA

Statistics by Class:
```

	Class: A	Class: B	Class: C	Class: D	Class: E	Class: F
Sensitivity	0.9399	0.8593	0.9130	0.8541	0.8353	0.7919
Specificity	0.9990	0.9869	0.9975	0.9922	0.9921	0.9956
Pos Pred Value	0.9718	0.7308	0.9333	0.8432	0.7889	0.8814
Neg Pred Value	0.9977	0.9941	0.9967	0.9929	0.9942	0.9915
Prevalence	0.0366	0.0398	0.0368	0.0466	0.0340	0.0394
Detection Rate	0.0344	0.0342	0.0336	0.0398	0.0284	0.0312
Detection Prevalence	0.0354	0.0468	0.0360	0.0472	0.0360	0.0354
Balanced Accuracy	0.9694	0.9231	0.9553	0.9232	0.9137	0.8938
	Class: G	Class: H	Class: I	Class: J	Class: K	Class: L
Sensitivity	0.8469	0.7143	0.8095	0.8947	0.8380	0.8333
Specificity	0.9898	0.9897	0.9967	0.9942	0.9942	0.9959
Pos Pred Value	0.7721	0.7059	0.9053	0.8586	0.8427	0.8788
Neg Pred Value	0.9937	0.9901	0.9925	0.9958	0.9940	0.9940
Prevalence	0.0392	0.0336	0.0378	0.0380	0.0358	0.0348
Detection Rate	0.0332	0.0240	0.0306	0.0340	0.0300	0.0290
Detection Prevalence	0.0430	0.0340	0.0338	0.0396	0.0356	0.0330
Balanced Accuracy	0.9184	0.8520	0.9031	0.9445	0.9161	0.9146

[output weggelaten]

De **accuracy** is behoorlijk omhoog gegaan, naar ruim 85%. Zeker onze “A” is goed te voorspellen, met een **sensitivity** van bijna 94%. Het loont dus zeker de moeite om te zoeken naar mogelijke verbeteringen!

Sommige letters (zoals de letter “E”,”F” en “H”) doen het een stuk slechter. Natuurlijk zijn we bij het trainen van het model beperkt door de data (inputsignalen) die we tot onze beschikking hebben. Het analyseren van de letters die moeilijk te voorspellen zijn (en vaak worden verward met andere letters) kan ons op het spoor zetten van extra gegevens die we nodig hebben om de letters beter te kunnen herkennen.