3. Neurale netwerken

In dit hoofdstuk breiden we het enkele neuron dat we in het vorige hoofdstuk zijn tegengekomen uit tot een multi-layer perceptron model met meerdere lagen van neuronen met meerdere neuronen per laag. Dit model is in staat om in principe elke distributie van klasselabels te modelleren en kan daarmee onder andere het XOR-probleem oplossen. Na een initialisatie met willekeurige gewichten en bias nul kunnen dergelijke neurale netwerken worden geoptimaliseerd met stochastic gradient descent. Hierbij wordt back-propagation toegepast om de gradiënt van de loss naar alle modelparameters te berekenen.

3.1. Het xor-probleem

Het perceptron heeft een aantal tekortkomingen die er in de praktijk voor zorgen dat het maar beperkt toepasbaar is. Een belangrijke hiervan is gelegen in het feit dat het alleen in staat is om lineair separabele data exact te modelleren. Echte data zijn echter slechts zelden lineair separabel. Hieraan kan deels tegemoet worden gekomen door de attributen te transformeren. Zo kan bijvoorbeeld een log-transformatie worden toegepast (bijvoorbeeld $x_1 \leftarrow \ln(x_1)$), of kunnen extra attributen worden toegevoegd die een functie zijn van de bestaande (bijvoorbeeld tweedegraads termen als $x_3 \leftarrow x_1^2$, $x_4 \leftarrow x_2^2$ en $x_5 \leftarrow x_1 \cdot x_2$). In dit hoofdstuk zullen we een andere aanpak gebruiken die leidt tot een type model waarvan kan worden bewezen dat het in theorie alle typen data exact kan beschrijven.

De rekeneenheid van een computer maakt intern gebruik van logische schakelingen. Denk hierbij aan NOT-, OR- en AND-operaties (de *negatie*, de *disjunctie* en de *conjunctie* genaamd), maar zoals we zullen zien bestaan er ook nog wel een aantal andere. Een computer kan hiermee in principe geprogrammeerd worden om élke mogelijke functie te berekenen die maar berekenbaar is; we noemen dit ook wel een *universele Turing-machine* en zeggen dan dat een dergelijk apparaat *Turing-compleet* is. Wat heeft dit nu te maken met machine learning? Welnu, het doen van een voorspelling middels een classificatie- of regressiemodel bestaat eigenlijk ook uit het berekenen van een functie, namelijk een functie die een voorspelde uitkomst $\hat{y}$ berekent uit de waarden van de attributen x_i van een instance. Op grond van het bovenstaande lijkt het daardoor in theorie mogelijk om elke mogelijke dataset optimaal te beschrijven als je model in staat is om operaties als NOT, OR en AND te representeren. Dat zou een *universele approximator* opleveren.

Laten we eens kijken hoe ver we hier nu al mee komen. Omdat we tot dusverre gewerkt hebben met klasselabels ± 1 zullen we hier afspreken dat de logische waarde TRUE overeenkomt met $+1$ en de logische waarde FALSE met -1. Omdat we met dichotome klasselabels werken ligt het perceptron-model $\hat{y} = \text{sgn}(b + \sum_i w_i \cdot x_i)$ voor de hand.

Laten we eerst samenvatten wat de *unaire* NOT-operatie op één attribuut x_1, genoteerd als $\neg x_1$, precies inhoudt. We doen dit met een *waarheidstabel* die voor alle mogelijke invoerwaarden de uitvoer opnoemt.

x_1	$y = \neg x_1$
-1	$+1$
$+1$	-1

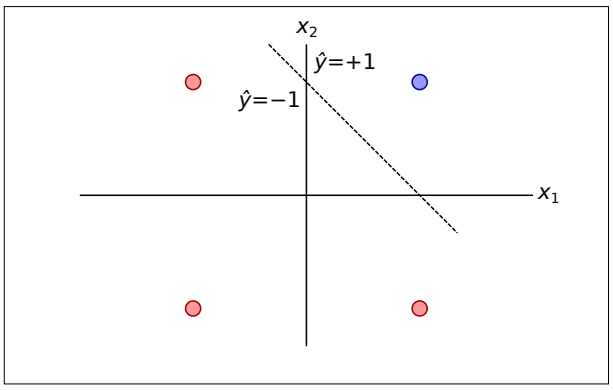
In dit geval is er maar één attribuut, dus vereenvoudigt het perceptron model tot $\hat{y} = \operatorname{sgn}(b + w_1 \cdot x_1)$. Kunnen we een bias b en een gewicht w_1 bedenken waarmee de bovenstaande waarden worden verkregen? Het is niet zo heel moeilijk om in te zien dat dit kan. Door bijvoorbeeld $b = 0$ te kiezen en $w_1 = -1$ krijgen we het model $\hat{y} = \operatorname{sgn}(-x_1)$ dat altijd de juiste voorspelling oplevert. In dit geval is de signum-functie niet eens noodzakelijk; de identiteitsfunctie had reeds volstaan. Er zijn overigens ook diverse andere combinaties van bias en gewicht te vinden die altijd een correcte voorspelling opleveren; de zojuist gegeven oplossing is dus niet uniek, maar dat terzijde.

Het wordt iets lastiger wanneer we naar de *binaire* OR- en AND-operatoren op twéé operanden x_1 en x_2 gaan, die respectievelijk genoteerd worden als $x_1 \vee x_2$ en $x_1 \wedge x_2$. De waarheidstabel moet nu vier mogelijkheden opsommen.

x_1	x_2	$y = x_1 \vee x_2$
-1	-1	-1
-1	$+1$	$+1$
$+1$	-1	$+1$
$+1$	$+1$	$+1$

x_1	x_2	$y = x_1 \wedge x_2$
-1	-1	-1
-1	$+1$	-1
$+1$	-1	-1
$+1$	$+1$	$+1$

Het perceptron model luidt nu $\hat{y} = \operatorname{sgn}(b + w_1 \cdot x_1 + w_2 \cdot x_2)$. Dankzij het feit dat de signum-functie alle uitkomsten naar ± 1 dwingt kunnen hier ook modellen voor gevonden worden. Ga voor jezelf na dat voor de OR-operatie gezegd kan worden dat $\hat{y} = \operatorname{sgn}(x_1 + x_2 + 1)$ en voor de AND-operatie geldt $\hat{y} = \operatorname{sgn}(x_1 + x_2 - 1)$. Dit komt in beide gevallen overeen met gewichten $\boldsymbol{w} = [1, 1]$ en bias $b = +1$ (voor OR) of $b = -1$ (voor AND).



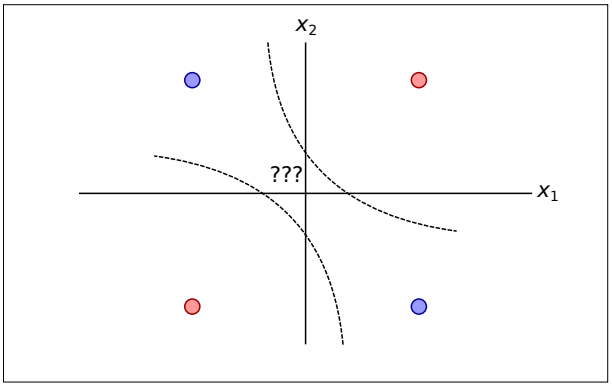
Hierboven wordt het model voor de AND-operatie geïllustreerd. De uitkomst $y = +1$ komt alleen voor bij $x_1 = x_2 = +1$, in blauw getoond in de rechterbovenhoek. De andere drie mogelijkheden hebben $y = -1$ en zijn weergegeven in rood. Een diagonale rechte

lijn scheidt de beide uitkomsten perfect. Het perceptron is in staat een dergelijke rechte lijn te modelleren. Iets soortgelijks geldt voor de OR-operatie.

We lijken er dus goed voor te staan: het eenvoudige perceptron is in staat om zowel NOT als OR als AND te modelleren. Echter, deze vlieger gaat helaas niet op voor sommige andere binaire operatoren, zoals de XOR-operatie $x_1 \oplus x_2$ (de *exclusieve disjunctie*) met de onderstaande waarheidstabel.

x_1	x_2	$y = x_1 \oplus x_2$
-1	-1	-1
-1	$+1$	$+1$
$+1$	-1	$+1$
$+1$	$+1$	-1

De onmogelijkheid van deze operaties in de vorm van een perceptron-model kan wellicht het beste worden afgelezen uit een figuur. Het perceptron is in staat om data van twee klassen perfect te scheiden dan en slechts dan als de data lineair separabel zijn. Overtuig jezelf ervan dat dat hieronder duidelijk niet het geval is.



Het perceptron faalt dus op de XOR-operatie. Omdat dit desalniettemin ogenschijnlijk een erg eenvoudige dataset is wordt dit het XOR-*probleem* genoemd. Hier worden de beperkingen van een simpel model als het perceptron pijnlijk duidelijk.

Overigens lukt het het lineaire of logistische regressie niet om het beter te doen, omdat ook deze beide modellen een rechte contourlijn hebben op $\hat{y} = 0$.

Opgave 38. *

In de tekst werd de NOT-operator gemodelleerd met een perceptron met bias $b = 0$ en gewicht $w_1 = -1$. Geef een voorbeeld van een bias $b \neq 0$ en gewicht $w_1 \neq -1$ die samen eveneens in staat zijn de NOT-operator correct te modelleren volgens het perceptron.

Opgave 39. **

De IMP-operator (de *implicatie*, genoteerd als $x_1 \Rightarrow x_2$) wordt gekarakteriseerd door de onderstaande waarheidstabel.

x_1	x_2	$y = x_1 \Rightarrow x_2$
-1	-1	$+1$
-1	$+1$	$+1$
$+1$	-1	-1
$+1$	$+1$	$+1$

Kan deze operator gemodelleerd worden met een single-layer perceptron? Zo nee, leg uit waarom niet; zo ja, hoe dienen de bias en gewichten dan bijvoorbeeld gekozen te worden?

Opgave 40. **

De EQV-operator (de *equivalentie*, genoteerd als $x_1 \Leftrightarrow x_2$) wordt gekarakteriseerd door de onderstaande waarheidstabel.

x_1	x_2	$y = x_1 \Leftrightarrow x_2$
-1	-1	$+1$
-1	$+1$	-1
$+1$	-1	-1
$+1$	$+1$	$+1$

Kan deze operator gemodelleerd worden met een single-layer perceptron? Zo nee, leg uit waarom niet; zo ja, hoe dienen de bias en gewichten dan bijvoorbeeld gekozen te worden?

Opgave 41. **

In de tekst werden gewichten en biases gegeven die het perceptron in staat stellen om de OR- en AND-operatoren te modelleren. Als je begint met een ongetraind perceptron (met gewichten en bias gelijk aan nul), en je itereert dan door de bijbehorende waarheidstabel om het perceptron te trainen, net zo veel epochs als nodig zijn om te convergeren zodat alle gevallen correct worden beschreven, kom je dan uit op diezelfde waarden voor deze gewichten en biases?

Opgave 42. ***

De *rectified linear unit* (*relu*) activatiefunctie is gelijk aan $\varphi(a) = a$ voor $a \geq 0$, en gelijk aan $\varphi(a) = 0$ voor $a < 0$; met andere woorden, $\varphi(a) = \max(a, 0)$. Maak voor jezelf een schets van deze functie. Stel, we coderen de waarde FALSE als 0 en TRUE als $+1$. Construeer dan single-layer modellen met één neuron met een relu-activatiefunctie die in staat zijn om de NOT-, AND- en OR-operatoren afzonderlijk te beschrijven, voor zover dat mogelijk is. Schrijf eerst de waarheidstabellen weer uit.

Opgave 43. ***

Diverse programmeertalen ondersteunen een *ternaire* logische operator met drie invoerwaarden: de *conditionele* operator, genoteerd als $x_1 \,?\, x_2 : x_3$. Deze retourneert de waarde van x_2 als x_1 TRUE is en de waarde van x_3 als x_1 FALSE is. Het werkt als een soort verkorte IF-THEN-ELSE constructie. Ga voor jezelf na dat deze wordt gekarakteriseerd door de onderstaande waarheidstabel. Schets voor jezelf deze data in een "drie-dimensionale grafiek" net zoals de OR- en AND-operatoren hierboven in twee-dimensionale grafieken werden geïllustreerd. Probeer in te schatten of de uitkomsten y lineair separabel zijn. Is het perceptron met geschikt gekozen gewichten w_1, w_2, w_3, en bias b in staat om de conditionele operator te beschrijven?

x_1	x_2	x_3	$y = x_1 \,?\, x_2 : x_3$
-1	-1	-1	-1
-1	-1	$+1$	$+1$
-1	$+1$	-1	-1
-1	$+1$	$+1$	$+1$
$+1$	-1	-1	-1
$+1$	-1	$+1$	-1
$+1$	$+1$	-1	$+1$
$+1$	$+1$	$+1$	$+1$

3.2. Het multi-layer perceptron

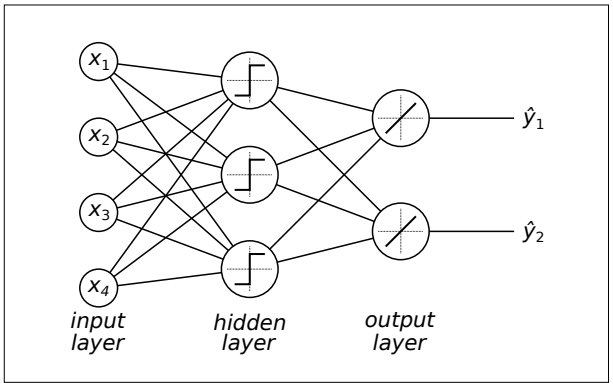
In de vorige paragraaf kwamen we een belangrijke beperking tegen van het perceptron, in de vorm van het XOR-probleem. We halen nu echter weer inspiratie uit de rekeneenheden van computers. Die bestaan immers ook niet simpelweg uit één enkele logische schakeling, maar zijn in staat om krachtige berekeningen uit te voeren door vele van dergelijke schakelingen aan elkaar te verbinden. Dit gebeurt zowel parallel, door meerdere operaties naast elkaar uit te voeren op dezelfde invoergegevens, als serieel, door meerdere operaties achter elkaar uit te voeren waarbij de uitvoer van de ene operatie dient als invoer voor de volgende.

Ditzelfde idee passen we toe op het perceptron.

- We plaatsen een aantal neuronen naast elkaar, die daarmee een *neurale laag* vormen van een bepaalde *breedte*. Dit is enigszins te vergelijken met *ensemble learning*, waarbij meerdere eenvoudige classificatiemodellen worden samengevoegd om te komen tot een complexer classificatiemodel met betere prestaties. Het gebruik van een brede laag met meerdere parallelle neuronen lijkt op classificatie door middel van *voting*, *bagging*, of *randomisatie*, waarbij je ook meerdere *base-learners* naast elkaar gebruikt.

- Vervolgens hangen we een aantal van dergelijke lagen achter elkaar om een neuraal netwerk te vormen met een bepaalde *diepte*. Dit is enigszins te vergelijken met ensemble-learning door middel van *stacking*, waarbij je een *meta-learner* op de uitvoer van de base-learners toepast.

We krijgen dan een *multi-layer perceptron* zoals hieronder geïllustreerd.



In dit geval zijn er vier attributen x_1 tot en met x_4 die als invoer dienen van de eerste laag neuronen. De attributen zou je zelf ook als een soort van neurale laag kunnen beschouwen die zelf geen invoer heeft, maar wel een vectoriële uitvoer $\boldsymbol{x}$ geeft met vier elementen x_i. Deze laag wordt ook wel de *input layer* genoemd omdat de gebruiker hierin de attributen van een instance invoert. In deze laag vindt echter geen berekening plaats.

De waarden in $\boldsymbol{x}$ worden doorgegeven naar de volgende laag, die in bovenstaand voorbeeld uit drie neuronen bestaat. Het is gebruikelijk dat alle neuronen in een laag dezelfde activatiefunctie gebruiken, zoals hier de signum-functie. Wel heeft elk neuron in de laag zijn eigen bias en gewichten. Omdat de gebruiker geen directe interactie heeft met deze laag wordt deze laag de *hidden layer* genoemd. Deze laag produceert op zijn beurt een vector met drie uitvoerwaarden die we zullen aanduiden als h_1 tot en met h_3, oftewel tezamen $\boldsymbol{h}$; dit zijn als het ware tussenresultaten van de berekening. Afhankelijk van de complexiteit van het model kunnen er meerdere van dergelijke hidden layers achter elkaar worden gedefinieerd die hun tussenuitkomsten $\boldsymbol{h}, \boldsymbol{h}', \boldsymbol{h}'', \ldots$ telkens aan elkaar doorgeven. Deze lagen kunnen uiteenlopende breedtes hebben. In de figuur hierboven is voor het gemak maar één hidden layer gebruikt, maar voor zeer complexe problemen zijn tientallen tot honderden (of tegenwoordig soms zelfs duizenden) lagen met vele duizenden tot miljoenen (of tegenwoordig soms zelfs miljarden) neuronen geen uitzondering.

Tenslotte wordt de uitvoer doorgegeven naar een laatste laag. In dit voorbeeld bestaat die uit twee neuronen met lineaire modellen. Deze produceren een vector $\hat{\boldsymbol{y}}$ met twee uitvoerwaarden die als voorspelling fungeren. Omdat dit hetgeen is waar de gebruiker naar op zoek was noemen we deze laatste laag de *output layer*. Voor regressie is het vrij gebruikelijk om de identiteitsfunctie als activatiefunctie in de output layer te kiezen; voor classificatie wordt vaak gekozen voor een functie met een bereik van 0 tot 1, zoals de logistische sigmoïde functie, maar hier komen we in het volgende hoofdstuk nog uitgebreider op terug. De diepte van het model is hier gelijk aan twee, omdat er in twee lagen een lineaire combinatie en activatiefunctie wordt toegepast; de input layer wordt gewoonlijk niet meegeteld bij het bepalen van de diepte. Zodra de diepte van een model groter is dan één, oftewel indien het model naast een input- en output-layer ook één of meer hidden layers bevat, dan kan worden gesproken van *deep learning*.

Omdat de attributen aan de linkerkant worden ingevoerd, dan stap voor stap wordt doorgerekend en doorgegeven van laag tot laag, om uiteindelijk uiterst rechts een voorspelling op te leveren, noemen we dit ook wel een *feed-forward* netwerk. Als elk neuron

toegang heeft tot alle uitvoerwaarden uit de vorige laag dan noemen we dit een *fully-connected layer*. Soms worden netwerken gedefinieerd met speciale *topologieën* die ook bijvoorbeeld verbindingen bevatten die lagen overslaan (bijvoorbeeld in *residuele* neurale netwerken), of die van latere lagen terugvoeren naar eerdere lagen (bijvoorbeeld in *recurrente* neurale netwerken), of waar de verbindingen slechts tussen bepaalde neuronen plaatsvinden in plaats van tussen alle (bijvoorbeeld in *convolutionele* neurale netwerken). Dergelijke netwerken hebben specialistische toepassingen in de analyse van onder andere tijdsignalen en beelden; die zullen we hier niet nader bespreken.

Merk overigens op dat we nu op een vrij vanzelfsprekende wijze een vector van voorspellingen $\hat{\boldsymbol{y}}$ als uitvoer hebben gekregen, waar we eerder slechts één *scalar* $\hat{y}$ hadden. Natuurlijk kunnen we eenvoudig toch een enkele voorspelling verkrijgen door simpelweg slechts één neuron in de output layer te plaatsen. Deze produceert dan een vector $\hat{\boldsymbol{y}}$ met lengte 1. Echter, het hebben van meerdere uitkomsten is vaak nuttig! Wanneer we bijvoorbeeld *multinomiale* classificatie uitvoeren waarbij er meer dan twee klassen zijn kunnen we elke uitvoer beschouwen als een indicator van hoe waarschijnlijk een bepaald klasselabel is. De gewenste uitvoer $\boldsymbol{y}$ zou er dan uit kunnen zien als een vector van nullen, met op één positie het cijfer één. Deze positie codeert dan het klasselabel. Dit wordt een *one-hot encoding* genoemd. Zo zou je de klasse $y = -1$ kunnen omcoderen naar $y = [1, 0]$ en de klasse $y = +1$ naar $y = [0, 1]$. Als er drie klassen zijn kun je die labelen met de one-hot vectoren $[1, 0, 0]$, $[0, 1, 0]$, en $[0, 0, 1]$, enzovoorts.

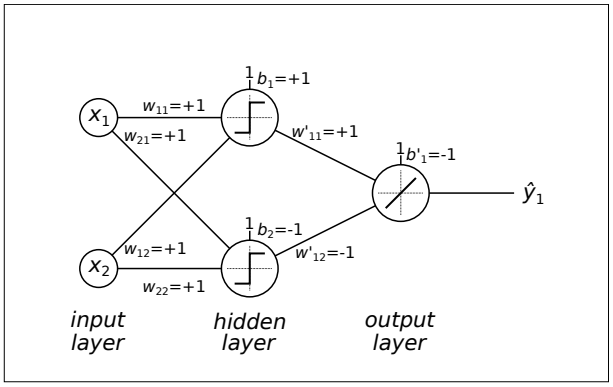
Een dergelijke one-hot codering kan ook gebruikt worden wanneer je een nominale variabele met een beperkt aantal mogelijkheden als attribuut wil gebruiken in de input layer. Je zet deze dan om in een verzameling *indicator-* of *dummy-variabelen* waarbij je alleen de ene waarde die overeenkomt met de waarde van het attribuut gelijk aan één stelt, en alle andere op nul. Zo zou je bijvoorbeeld iemands oogkleur kunnen coderen als $x = [1, 0, 0]$ voor bruin, $x = [0, 1, 0]$ voor blauw, en $x = [0, 0, 1]$ voor grijs of anders. Je gebruikt dan drie attributen x_i voor het coderen van wat eigenlijk één meetuitkomst is.

Opmerkelijk genoeg is bewezen dat een netwerk met één voldoende brede hidden layer met een niet-lineaire activatiefunctie in staat is om in principe elk mogelijk classificatie- of regressieprobleem willekeurig nauwkeurig te modelleren. Dit wordt het *universele approximatie theorema* genoemd. Voor realistische problemen met veel attributen en ingewikkelde klassegrenzen kan het dan echter nodig zijn om gigantisch veel neuronen in die ene layer te stoppen. Desalniettemin is het interessant te weten dat het multi-layer perceptron op zijn minst in theorie altijd in staat is om een goede oplossing te beschrijven. In de praktijk blijken modellen met meerdere smallere lagen het echter vaak beter te doen, hoewel dergelijke diepere modellen evenzeer problemen kunnen geven bij het trainen. Het feit dat een goede oplossing in theorie mogelijk is wil in dergelijke gevallen namelijk nog niet zeggen dat je zo'n goede oplossing in de praktijk ook daadwerkelijk kan vinden!

Met het multi-layer perceptron is het XOR-probleem netjes op te lossen. Begin bijvoorbeeld eens met een hidden layer die twee perceptron neuronen bevat, elk met de signum-functie als activatiefunctie. Kies de gewichten van de ene gelijk aan die van de OR-operator en die van de andere gelijk aan die van de AND-operator, zoals we die eerder bepaalden. Voeg nu een output layer toe met slechts één lineair neuron, met de identiteitsfunctie als activatiefunctie. Hoe kan deze de uitvoer van de OR- en AND-operatoren

combineren om een XOR-operator te verkrijgen? De OR-operatie lijkt best veel op de XOR-operatie, dus dat lijkt een goed uitgangspunt. Alleen als zowel x_1 als x_2 gelijk zijn aan $+1$ gaat het mis: $x_1 \vee x_2$ levert dan $+1$, terwijl $x_1 \oplus x_2$ daarentegen -1 zou moeten opleveren. Dit is precies de ene combinatie van x_1 en x_2 waar de AND-operatie een afwijkende uitkomst voor geeft. Deze eigenschap kunnen we gebruiken om in dat geval de uitkomst te verlagen. Als je de uitkomst van de AND-operator nu eens aftrekt van die van de OR-operator zit je al bijna goed. Je vind dan alleen uitkomsten gelijk aan 0 en 2 in plaats van -1 en $+1$. Maar dat is op te lossen door er in de output layer een negatieve bias bij te stoppen. Dit leidt tot exact de gewenste uitkomsten.

In grafische vorm krijgen we dan het volgende model.



Natuurlijk is het behoorlijk een kwestie van ervaring, inzicht, uitproberen, en soms geluk om op de zojuist beschreven manier tot een goede oplossing te komen. Als we voor complexere problemen het model willen trainen zijn we op geautomatiseerde optimalisatie-algoritmen aangewezen. De werking hiervan vormt de kern van machine learning; enig begrip hiervan is essentieel om een neuraal netwerk zinvol te kunnen gebruiken, dus hier gaan we in de rest van dit hoofdstuk nader op in.

Opgave 44. *

Naast voting, bagging, randomisatie, en stacking is ook *boosting* een vorm van ensemble learning. Is dit meer een vorm van parallelle of seriële gegevensverwerking, vind je? Motiveer je antwoord.

Opgave 45. **

In de opgaven bij de vorige paragraaf werd gevraagd om de IMP-operator $x_1 \Rightarrow x_2$ en de EQV-operator $x_1 \Leftrightarrow x_2$ zo mogelijk te beschrijven met een single-layer perceptron. Voor zover dat onmogelijk gedaan kon worden, modelleer deze operatoren dan nu met behulp van een multi-layer perceptron.

Opgave 46. **

Gegeven de modellen voor de OR-, AND-, en XOR-operatoren, teken het diagram van een enkel neuraal netwerk met diepte 2, met een input layer met breedte 2, een hidden layer met breedte 2, en een output layer met breedte 3, dat al deze drie operaties als

parallelle uitvoeren heeft. Met andere woorden, voor een input $[x_1, x_2]$ dient de output gelijk te zijn aan $[x_1 \vee x_2, x_1 \wedge x_2, x_1 \oplus x_2]$. Vermeld in je diagram de waarden van alle modelparameters.

Opgave 47. **
Breid het model uit de vorige opgave uit zodat het ook de waarden $\neg x_1$ en $\neg x_2$ uitvoert. Je krijgt dan een model met een output layer met breedte vijf. Bedenk zelf of de breedte van de input en/of hidden layer ook uitgebreid moeten worden.

Opgave 48. ***
Leg uit waarom het meestal geen zin heeft om de identiteitsfunctie als activatiefunctie te kiezen in hidden layers (hoewel het wel zinvol kan zijn in de output layer).

Opgave 49. ***
Beschouw de onderstaande dataset van vijf punten.

x_1	x_2	y
-1	0	$+1$
0	-1	$+1$
0	0	-1
0	$+1$	$+1$
$+1$	0	$+1$

Schets deze punten in een grafiek. Kan deze dataset met een single-layer perceptron gemodelleerd worden? Ontwerp een multi-layer perceptron met één hidden layer met de relu-activatiefunctie $\text{relu}(a) = \max(a, 0)$ (zie de opgaven uit de vorige sectie), en één lineaire output layer met de identiteitsfunctie als activatiefunctie. Bepaal zelf de breedte van je netwerk. Hoe zou je de bias en gewichten van alle neuronen kunnen kiezen om alle uitkomsten correct te voorspellen? Hint: het kan met twee neuronen in de hidden layer, maar met vier is waarschijnlijk makkelijker.

Opgave 50. ***
Construeer een multi-layer perceptron met diepte, breedten, en activatiefuncties naar keuze waarmee de ternaire conditionele operator $x_1 \,?\, x_2 : x_3$ (zie de opgaven uit de vorige sectie) precies gemodelleerd kan worden.

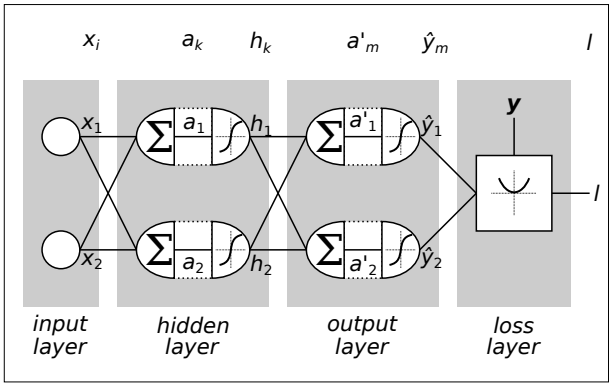
3.3. Back-propagation

Om te komen tot geschikte biases en gewichten voor alle neuronen zullen we voortbouwen op het algemene raamwerk uit het vorige hoofdstuk dat gebruik maakte van het minimaliseren van de lossfunctie middels stochastic gradient descent. Tot dusverre hebben we de lossfunctie van het multi-layer perceptron nog niet bekeken. Je kunt deze beschouwen als een soort extra laag op het eind die de voorspellingen $\hat{\boldsymbol{y}}$ (een vector) in combinatie met de gewenste uitkomsten $\boldsymbol{y}$ (ook een vector) omzet in één einduitkomst, de loss l (een scalar). Een verschil met het vorige hoofdstuk is dat we nu één loss willen berekenen uit een hele vector van voorspelde en gewenste uitkomsten.

Het uitgangspunt blijft hierbij dat een goede oplossing de loss dient te minimaliseren. We willen dat geldt dat $l = 0$ als álle elementen van de uitkomstenvector juist worden voorspeld; als dat niet het geval is willen we dat strict $l > 0$. En liefst willen we dat de loss geleidelijk toeneemt naarmate de oplossing slechter wordt. Hoe valt dit te bereiken? Gewoonlijk worden simpelweg de losses van elk van de elementen $\hat{y}_m$ in de uitvoer opgeteld, dat wil zeggen $l = \sum_m \mathcal{L}(\hat{y}_m; y_m)$. Ga voor jezelf na dat deze uitkomst alleen nul is als alle y_m juist voorspeld worden, en strict groter dan nul is als één of meer van de y_m verkeerd worden voorspeld.

Om stochastic gradient descent toe te passen moeten we de afgeleide van de loss naar alle gewichten kennen. Elke bias b beschouwen we weer als een gewicht w_0 voor een denkbeeldig extra attribuut x_0 dat altijd gelijk is aan 1. De partiële afgeleide $\frac{\partial l}{\partial w_i}$ is echter niet eenvoudig te berekenen. Je zou natuurlijk het gewicht een kleine hoeveelheid Δw_i kunnen veranderen en dan het complete netwerk opnieuw doorrekenen om te bezien hoeveel de verandering in de loss Δl daardoor zou bedragen. Maar die aanpak is onbeheersbaar: een netwerk kan duizenden gewichten bevatten, en ook duizenden optellingen en vermenigvuldigingen vereisen om door te rekenen. De hoeveelheid rekentijd zou hierdoor onaanvaardbaar groot worden. Het blijkt echter mogelijk om de afgeleiden van de loss naar alle gewichten in één keer te bepalen met een enkele gang door het netwerk. Waar de evaluatie van een feed-forward network van de invoer- naar de uitvoerzijde geschiedt, gaat de berekening van de afgeleiden in omgekeerde richting, van de uitvoer- naar de invoerzijde. Dit process staat bekend onder de naam *back-propagation*.

We bekijken een schematische weergave van een multi-layer perceptron. Dit model heeft een diepte van twee en alle neurale lagen hebben hier een breedte van twee omwille van de eenvoud, maar dit kan in werkelijkheid natuurlijk variëren.



Hierboven wordt links een input layer getoond met uitvoer x_i. Daarna volgen middenin twee fully-connected layers met neuronen. Hun werking is opgesplitst in het maken van een lineaire combinatie enerzijds en het toepassen van de activatiefunctie anderzijds. Deze ontvangen als invoer x_i (danwel h_k). Vervolgens maken ze een gewogen lineaire combinatie om tot de pre-activatiewaarden a_k (en a'_m) te komen, die worden omgezet in post-activatiewaarden h_k (en $\hat{y}_m$) door toepassing van een activatiefunctie. Tenslotte hebben we de loss layer die een aantal voorspellingen $\hat{y}_m$ ontvangt en die tezamen met de gewenste uitkomsten y_m omzet in de loss l.

Tijdens de feed-forward fase wordt het model doorgerekend met dezelfde transformaties als in het vorige hoofdstuk, alleen worden de fully-connected layers vaker herhaald. In formulevorm komt dit neer op de volgende stappen:

1. $a_k = b_k + \sum_i w_{ki} \cdot x_i$;
2. $h_k = \varphi(a_k)$;
3. $a'_m = b'_m + \sum_k w'_{mk} \cdot h_k$;
4. $\hat{y}_m = \varphi'(a'_m)$;
5. $l = \sum_m \mathcal{L}(\hat{y}_m; y_m)$.

Hierin heeft de index i betrekking op de input layer, de index k op de hidden layer, en de index m op de output layer. De overeenkomstige variabelen zonder en met accenten hebben respectievelijk betrekking op de hidden layer en output layer. De gewichten hebben dit keer twee indices omdat elk neuron van elke invoer afhangt: een gewicht w_{ki} beschrijft de sterkte van de verbinding van invoer i naar neuron k.

Als er nog meer hidden layers zijn worden er nog meer vergelijkingen aan het model toegevoegd; stappen 2 en 3 worden dan min of meer herhaald om extra tussenuitkomsten $\boldsymbol{h'}, \ldots$ en $\boldsymbol{a''}, \ldots$ te berekenen. Voor het gemak houden we het nu bij één hidden layer, maar het principe blijft hetzelfde als er meerdere hidden layers zijn.

Nu gaan we van achteren naar voren op zoek naar de gradiënten van de loss.

1. Eerst bepalen we de gradiënt van de loss naar de voorspellingen $\hat{y}_m$. Dat wil zeggen, als een voorspelling $\hat{y}_m$ een beetje zou veranderen, wat gebeurt er dan met de loss? De loss kan geschreven worden als $l = \mathcal{L}(\hat{y}_1; y_1) + \mathcal{L}(\hat{y}_2; y_2) + \ldots$. Als we de afgeleide naar de voorspelling $\hat{y}_1$ willen weten hoeven we alleen naar de eerste term te kijken. Immers, de termen $\mathcal{L}(\hat{y}_2; y_2)$ enzovoorts hangen helemaal niet af van $\hat{y}_1$ en hebben daardoor geen invloed op de helling. De afgeleide van de lossfunctie $\mathcal{L}$ bepalen we analytisch of numeriek, net als in het vorige hoofdstuk voor het single-layer perceptron. Voor de kwadratische lossfunctie is deze bijvoorbeeld gelijk aan $2(\hat{y}_m - y_m)$. Een verschil is dat we nu niet slechts één afgeleide $\frac{\partial l}{\partial \hat{y}_m}$ moeten bepalen, maar we dit moeten herhalen voor elke voorspelling $\hat{y}_m$ en alle resultaten moeten verzamelen in een gradiëntvector die we noteren als $\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}} l$.

2. Vervolgens kijken we naar de gradiënt van de loss naar de pre-activatiewaarden a'_m in de output layer. We maken gebruik van $\frac{\partial l}{\partial a'_m} = \frac{\partial l}{\partial \hat{y}_m} \cdot \frac{\partial \hat{y}_m}{\partial a'_m}$. De eerste factor $\frac{\partial l}{\partial \hat{y}_m}$ halen we uit de voorgaande stap. De tweede factor $\frac{\partial \hat{y}_m}{\partial a'_m}$ beschrijft hoe de post-activatiewaarde verandert als de pre-activatiewaarde wijzigt. Dit is gelijk aan de helling van de activatiefunctie φ'. Wij zullen ook deze analytisch of numeriek bepalen, opnieuw niet slechts éénmaal maar herhaaldelijk voor elk neuron in de laag. Voor de tangens-hyperbolicus-functie is die helling bijvoorbeeld gelijk aan $1 - \hat{y}_m^2$, zoals in het vorige hoofdstuk beschreven. Hiermee kan de gradiënt $\boldsymbol{\nabla}_{\boldsymbol{a'}} l$ worden berekend.

3. Dan gaan we naar de gradiënt van de loss naar de invoerwaarden h_k van de output layer. Weer splitsen we de afgeleide op, alleen moeten we er nu rekening mee houden dat in een fully-connected layer elke invoer invloed heeft op alle neuronen. Die invloeden moeten allemaal bij elkaar worden geteld, waardoor we verkrijgen $\frac{\partial l}{\partial h_k} = \sum_m \frac{\partial l}{\partial a'_m} \cdot \frac{\partial a'_m}{\partial h_k}$. De waarden van $\frac{\partial l}{\partial a'_m}$ voor alle neuronen in de laag bepaalden we in de vorige stap. De afgeleide $\frac{\partial a'_m}{\partial h_k}$ moeten we afleiden uit hoe de lineaire combinatie werkt. Als h_k met 1 eenheid toeneemt, dan neemt a'_m met w'_{mk} toe, aangezien elke invoer met een gewicht wordt vermenigvuldigd. De conclusie luidt dat $\frac{\partial a'_m}{\partial h_k} = w'_{mk}$. Hiermee kan de gradiënt $\boldsymbol{\nabla_h} l$ worden berekend.

4. De bovenstaande stappen 2 en 3 herhalen we nu van achter naar voren voor alle neurale lagen totdat we beland zijn bij de laag die rechtstreeks de invoer $\boldsymbol{x}$ ontvangt. In dit geval leidt dat tot $\frac{\partial l}{\partial a_k} = \frac{\partial l}{\partial h_k} \cdot \frac{\partial h_k}{\partial a_k}$ wat neerkomt op vermenigvuldigen met de helling van activatiefunctie φ, en $\frac{\partial l}{\partial x_i} = \sum_k \frac{\partial l}{\partial a_k} \cdot \frac{\partial a_k}{\partial x_i}$ wat neerkomt op vermenigvuldigen met w_{ki}. Het resultaat is dat we ook de gradiënten $\boldsymbol{\nabla_a} l$ en $\boldsymbol{\nabla_x} l$ kennen.

Voor elke vector $\boldsymbol{x}$, $\boldsymbol{a}$, $\boldsymbol{h}$, $\boldsymbol{a'}$ of $\hat{\boldsymbol{y}}$ weten we nu precies hoe veranderingen daarin de loss l beïnvloeden!

Tenslotte rest ons alleen nog te bepalen wat het effect is van veranderingen in de gewichten op de loss, want daar was het ons eigenlijk om te doen. Als we gewichten aanpassen hebben die rechtstreeks effect op de pre-activatiewaarde $\boldsymbol{a}$ of $\boldsymbol{a'}$ die daaruit berekend wordt. Verhogen we een gewicht w_{ki} met één eenheid, dan wordt de pre-activatiewaarde a_k met een hoeveelheid x_i opgehoogd, omdat de gewichten met de invoerwaarden worden vermenigvuldigd (vergelijkbaar met stap 3 hierboven). We weten dus dat $\frac{\partial a_k}{\partial w_{ki}} = x_i$. Maar als we nu de afgeleide van de loss naar de gewichten schrijven als $\frac{\partial l}{\partial w_{ki}} = \frac{\partial l}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{ki}}$, dan kennen we de beide factoren aan de rechterzijde: alle $\frac{\partial l}{\partial a_k}$ hebben we bepaald in stap 4 hierboven, en $\frac{\partial a_k}{\partial w_{ki}}$ stelden we zojuist gelijk aan de input van de betreffende neurale laag x_i. Iets soortgelijks geldt voor de gewichten van de output layer.

Hiermee zijn alle afgeleiden van de loss naar de gewichten $\boldsymbol{\nabla_w} l$ (en soortgelijk de biases) eindelijk berekend. Hiermee kunnen de gewichten worden bijgewerkt volgens gradient descent.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \cdot \boldsymbol{\nabla_w} l$$

Samengevat hebben we dus teruggewerkt om afgeleiden te bepalen, waarbij elke keer de gradiënt van de loss kon worden bijgewerkt door te vermenigvuldigen met een extra factor, te weten de helling van de lossfunctie, de helling van de activatiefunctie, of de gewichten. De afgeleide van de loss naar de modelparameters kon tenslotte worden bepaald door een vermenigvuldiging met de invoerwaarden van een laag. Deze kunnen we gebruiken om gradient descent mee uit te voeren om de loss mee te minimaliseren.

Al met al is het hier beschreven back-propagation proces zonder meer het meest ingewikkelde wat er over een neuraal netwerk te weten valt. Tegelijkertijd is het zo mogelijk ook het meest belangrijke proces, juist omdat dit gebruikt wordt om het netwerk te trainen en zonder goede training is zelfs het beste soort model volkomen nutteloos.

Conceptueel komt het er op neer dat gekeken wordt hoe een kleine verandering in de modelparameters, dat wil zeggen de gewichten en biases, de verschillende tussenuitkomsten in het netwerk beïnvloeden: denk hierbij aan de pre- en post-activatiewaarden van alle layers. Uiteindelijk verandert hierdoor helemaal aan het einde de post-activatiewaarde van de output layer, die gelijk is aan de voorspelling, en hiermee beïnvloed je dus ook de loss. Deze informatie gebruikt de stochastic gradient descent procedure om geleidelijk een steeds beter model te verkrijgen.

Opgave 51. *

De loss wordt in het multi-layer perceptron meestal bepaald als de *som* van de losses van elk van de voorspelde uitkomsten $\hat{y}_m$, dat wil zeggen $l = \sum_m \mathcal{L}(\hat{y}_m; y_m)$. Waarom is het een minder goed idee om het *product* van de individuele losses te nemen, dat wil zeggen $l = \prod_m \mathcal{L}(\hat{y}_m; y_m)$; deze zou toch immers ook $l = 0$ opleveren als alle uitkomsten juist voorspeld worden?

Opgave 52. **

Zou het een goed idee zijn om de loss te definiëren als het maximum van alle losses van de individuele uitkomsten, dat wil zeggen $l = \max_m \mathcal{L}(\hat{y}_m; y_m)$? Bespreek hiervan enkele voor- en nadelen.

Opgave 53. **

Hierboven werd uitgeschreven hoe je de afgeleide van de loss naar de gewichten w_{ki} van de hidden layer kan bepalen, te weten $\frac{\partial l}{\partial w_{ki}} = \frac{\partial l}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{ki}}$. Werk zelf uit hoe je de afgeleide van de loss naar de gewichten w'_{mk} van de output layer kan bepalen, te weten $\frac{\partial l}{\partial w'_{mk}}$.

3.4. Initialisatie

We sluiten af met wat opmerkingen over de initialisatie van de bias en gewichten van neuronen. Tot dusverre hebben we die meestal op nul gesteld, maar voor de meerdere neuronen in een neurale laag is dit niet verstandig. Immers, als alle neuronen beginnen met dezelfde parameters en ze krijgen ook elke keer allemaal dezelfde input, dan zullen ze ook allemaal op dezelfde manier bijgewerkt worden. Alle neuronen blijven dan identieke kopieën van elkaar. Ze voegen dan geen extra informatie toe. Daarom worden neuronen geïnitialiseerd met willekeurige waarden.

Een geschikte keuze van de gewichten is nog best een lastige zaak. Als de gewichten veel groter dan 1 zijn worden de invoerwaarden vermenigvuldigd met grote getallen. Dat wil zeggen dat de uitkomsten in latere lagen steeds grotere getallen gaan bevatten wat tot slechte predicties kan leiden, zeker als je veel lagen hebt. In omgekeerde richting worden de gradiënten van de eerdere lagen dan ook te groot als je back-propagation toepast. Maak je de gewichten veel kleiner dan 1 dan geldt het omgekeerde: de voorspelde uitkomsten van de latere lagen en de gradiënten in de eerdere lagen worden dan te klein. Kortom, zijn de aanvankelijke gewichten te groot, dan "exploderen" de getallen die je

berekent tijdens de feed-forward en back-propagation fases; zijn ze te klein, dan "doven" ze al gauw uit tot iets wat niet meer meetbaar is.

Dit maakt het moeilijk voor het netwerk om te leren: als de eerste lagen snel leren, leren de latere lagen te langzaam, of omgekeerd. Om zowel de lagen vooraan in het netwerk als de lagen achteraan in het netwerk optimaal te laten leren moeten de gewichten op delicate wijze zo gekozen worden dat de waarden $\boldsymbol{h}, \boldsymbol{h}', \ldots$ die van laag naar laag worden doorgegeven een "redelijke" grootte blijven houden.

Wat je eigenlijk wil met je beginwaarden is dat een invoer van getallen met een typische grootte van ± 1 (zoals bijvoorbeeld standaard-normaalverdeelde waarden) ook weer leidt tot een uitvoer van getallen met een typische grootte van ± 1. In dat geval worden alle lagen aanvankelijk voorzien van redelijke getalwaarden. Je moet hierbij ook rekening houden met het feit dat neuronen meerdere invoeren krijgen die ze gewogen optellen. Voor normaal verdeelde getallen geldt dat de som van N willekeurige waarden in grootte groeit evenredig met $\sqrt{N}$. Om de grootte van de combinaties niet te laten toenemen moet je de gewichten hiervoor laten compenseren, dus die dien je evenredig te nemen aan $w \sim \frac{1}{\sqrt{N}}$. Een mogelijkheid is derhalve om de gewichten te kiezen volgens een *normale* verdeling met een gemiddelde $\mu = 0$ en een standaardafwijking $\sigma = \frac{1}{\sqrt{N}}$. Hierbij kies je N meestal gelijk aan het gemiddelde van het aantal invoeren N_{in} en het aantal uitvoeren N_{uit} van de betreffende neurale laag, dat wil zeggen $N = \frac{N_{\text{in}}+N_{\text{uit}}}{2}$, omdat je daarmee zowel de feed-forward als back-propagation fase meeweegt.

Een alternatief is om gewichten te kiezen volgens een *uniforme* verdeling. Hierbij zijn alle waarden tussen een zekere boven- en ondergrens even waarschijnlijk. Deze wordt gewoonlijk symmetrisch rondom nul gekozen. Een uniforme verdeling van waarden tussen -1 en $+1$ heeft van zichzelf een standaardafwijking $\sigma = \frac{1}{\sqrt{3}}$. Om te komen tot een verdeling met standaarddeviatie $\sigma = \frac{1}{\sqrt{N}}$ dien je dan de gewichten uniform te kiezen tussen uitersten van $\pm\sqrt{\frac{6}{N_{\text{in}}+N_{\text{uit}}}}$. Dit wordt *(genormaliseerde) Xavier initialisatie* genoemd

De bias kiest men gewoonlijk wel altijd gelijk aan nul. Dit doet men om ervoor te zorgen dat neuronen "ergens in het midden" van de activatiefunctie beginnen. Sigmoïde functies lopen bijvoorbeeld nogal vlak ver van nul vandaan, en als een neuron daar begint leert het slechts heel langzaam. Aangezien de gewichten van de neuronen toch al verschillend zijn kunnen de biases best allemaal hetzelfde gekozen worden.

Er bestaan diverse ingewikkeldere heuristieken om beginwaarden te genereren. Wat precies de beste aanpak is in bepaalde gevallen is eigenlijk niet goed bekend.

Opgave 54. *

Leg iets beter uit wat er bedoeld wordt met "sigmoïde functies lopen bijvoorbeeld nogal vlak ver van nul vandaan, en als een neuron daar begint leert het slechts heel langzaam."

Opgave 55. *

Wanneer een relu-functie $\varphi(a) = \max(a, 0)$ als activatiefunctie wordt gebruikt, wordt er soms voor gekozen om de bias met een lichte positieve waarde te initialiseren. Wat zou hier de reden van zijn, denk je?

Opgave 56. **

Simuleer in een omgeving naar keuze (bijvoorbeeld MS-Excel, R, Python, Java, enzovoorts) duizend maal de worp van een enkele dobbelsteen. Wat is het gemiddelde aantal ogen dat je gooit en wat is hiervan de standaardafwijking? Simuleer vervolgens duizend maal de som van ogen van honderd dobbelstenen. Wat is nu het gemiddelde en de standaardafwijking? Komen je bevindingen overeen met de regel dat de som van willekeurige waarden groeit volgens $\sqrt{N}$?

Opgave 57. ***

Laat op grond van de gegeven informatie zien dat voor gewichten met een uniforme verdeling tussen $\pm\sqrt{\frac{6}{N_{\text{in}}+N_{\text{uit}}}}$ geldt dat de standaardafwijking gelijk is aan $\sigma = \frac{1}{\sqrt{N}}$.

Opgave 58. ***

De gegeven redenering houdt wel rekening met hoe de gewichten doorwerken op de grootte van de uitkomsten en gradiënten, maar negeert de activatiefunctie. Beredeneer of een activatiefunctie zoals de tangens-hyperbolicus de typische grootte van uitkomsten van laag tot laag zal laten toe- of afnemen als je de gewichten kiest met een standaardafwijking $\sigma = \frac{1}{\sqrt{N}}$, zoals hierboven geadviseerd. Evenzo de gradiënten in omgekeerde richting. Hint: heeft de typische uitkomst van de activatiefunctie een grootte van 1, en heeft de typische helling van de activatiefunctie een grootte van 1?