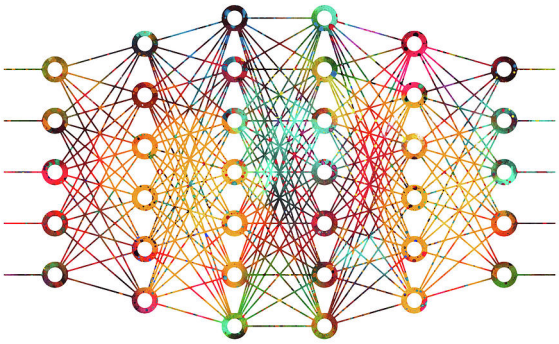
Syllabus BFVH4DMN2
Advanced Data-mining

"Your Hello (to the) World of Neural Networks"

Docent: Dave Langers <d.r.m.langers@pl.hanze.nl>



Inhoudsopgave

Deel I.

Machine learning

1. Het perceptron

Vooruitblik

In dit hoofdstuk maken we kennis met een van de meest eenvoudige machine learning methoden die lineair separabele datasets met nominale uitkomsten perfect kan classificeren: het perceptron. We formuleren het model waarmee instances van een klasselabel worden voorzien, en leiden eveneens een algoritme af dat in staat is de parameters van het model te fitten. Tenslotte generaliseren we deze methode naar instances met numerieke uitkomsten, waarna zal blijken dat we met een soortgelijk model eveneens lineaire regressie kunnen uitvoeren.

1.1. Inleiding

Machine learning algoritmen leren op grond van beschikbare data aangaande een aantal *attributen* om een gewenste uitkomst toe te kennen aan een onbekende *instance*. Hierbij kun je onderscheid maken tussen *nominale* en *numerieke* uitkomsten (d.w.z. categorische klasselabels danwel continue getalwaarden). Algoritmen kunnen daarnaast worden onderverdeeld in twee typen: *supervised* en *unsupervised*.

Supervised algoritmen worden getraind met instances die reeds voorzien zijn van de gewenste uitkomst, het zogenaamde *target*. Het doel van supervised algoritmen is om te leren hoe je deze gewenste uitkomst kan afleiden uit de waarden van de verschillende attributen. *Classificatie* en *regressie* zijn typische voorbeelden van supervised algoritmen met respectievelijk nominale en numerieke uitkomsten. In de *trainingsfase* worden door het algoritme een aantal *parameters* bepaald om tot een optimaal model te komen: het *concept* (bv. een beslisboom of een lineaire vergelijking). Daarna dient het algoritme het gefitte model in de *testfase* toe te passen om een zo goed mogelijke inschatting te geven van de uitkomst van een nieuwe set onbekende instances waarvan de juiste uitkomst niet tevoren gegeven is. Vaak wordt er ook nog een *validatiefase* tussengevoegd waarin de onderzoeker een aantal *hyperparameters* van het algoritme varieert om zo tot een optimaal model te komen (bv. de hoeveelheid pruning van de beslisboom of de graad van het polynoom in de vergelijking).

Unsupervised algoritmen krijgen een hoeveelheid data voorgeschoteld en moeten zelf een relevante structuur in de data zien te ontdekken. *Clustering* en *dimensiereductie* zijn typische categorieën van unsupervised algoritmen met respectievelijk nominale en numerieke uitkomsten. In het geval van clustering wordt aan elke instance een label toegekend behorende bij het cluster waartoe het instance op grond van diens attributen behoort; in het geval van dimensiereductie wordt aan elke instance een beperkt aantal getalm-

atige scores toegekend. Opnieuw leidt dit tot een geoptimaliseerd model, het concept (bijvoorbeeld de cluster centroïden in het geval van k-means clustering, of de loadings bij principale componenten analyse). Kenmerkend is dat unsupervised algoritmen noch tijdens de trainings-, de validatie-, of de testfase ooit te horen hoeven te krijgen wat een ware gewenste uitkomst is. Het nadeel wat hier tegenover staat is dat de uitkomsten van unsupervised algoritmen achteraf vaak lastig te interpreteren zijn.

Trainingsdata bestaan uit een (groot) aantal instances met attributen en een bijbehorende uitkomst: een nominaal klasselabel voor classificatie of een numerieke getalwaarde voor regressie. Gewoonlijk worden de attributen gerepresenteerd door de kolommen van een datatabel, waarbij één speciale kolom (meestal de laatste kolom) de te voorspellen uitkomsten bevat. De waarden van de attributen vormen input voor het algoritme en duiden we aan als $x_1, x_2, \ldots, x_d$; d geeft hierbij het aantal attributen aan, d.w.z. de *dimensionaliteit*. We zullen er in het vervolg van uit gaan dat alle attributen x_i numeriek worden gerepresenteerd als getalwaarden; nominale invoerwaarden zijn op zich wel mogelijk, maar zullen desgewenst in numerieke waarden worden omgezet. De bijbehorende te voorspellen nominale of numerieke uitkomst duiden we aan met y. Deze wordt toegekend door een *expert*, soms ook wel een *orakel* genoemd. Omdat het model in de praktijk niet perfect is, zal de voorspelde uitkomst, aangeduid met $\hat{y}$, kunnen afwijken van de werkelijke waarde, y. We gebruiken hier de gebruikelijke notatie uit de statistiek waarbij een dakje grootheden aanduidt die een schatting zijn van de echte waarde.

Het model kunnen we zien als een abstracte wiskundige functie f die afhankelijk van de attributen een voorspelling genereert: $\hat{y} = f(x_1, x_2, \ldots, x_d)$. De precieze vorm van die functie is bij aanvang onbekend, maar je stopt er de attributen van een instance in en krijgt er de voorspelde uitkomst uit. De *fout* die hierbij gemaakt wordt is dan gelijk aan $e = \hat{y} - y$; deze wordt ook wel de *afwijking*, het *residu* of de *error* genoemd. Het doel van machine learning is natuurlijk om het model zo te optimaliseren dat fouten zo min mogelijk voorkomen (bij classificatie) of zo klein mogelijk zijn (bij regressie). Hiertoe kunnen gewoonlijk een aantal parameters van het model worden gevarieerd; de aard hiervan hangt sterk af van het type model.

Een ideaal model is in principe in staat om elke denkbare vorm voor de functie f te fitten die het beste past bij de data, en vormt daarmee een *universele approximator*. In de praktijk dient een goede balans gevonden te worden tussen *underfitting* omdat het model niet flexibel genoeg is enerzijds, en *overfitting* omdat toevallige details in de trainingsdata geleerd worden die niet generaliseren naar nieuwe testdata anderzijds.

Neurale netwerken zijn voorbeelden van supervised machine learning. Ze worden getraind met voorbeelden die van een uitkomst zijn voorzien. Zoals we later zullen zien zijn neurale netwerken voldoende flexibel om nominale óf numerieke uitkomsten te kunnen produceren. Ze kunnen daarmee gebruikt worden om zowel classificatie- als regressieproblemen op te lossen. Met wat extra kunstgrepen die wij niet zullen verkennen kunnen ze zelfs unsupervised problemen aan. Het zijn dus zeer universeel toepasbare algoritmen.

Er is al werk gedaan aan neurale netwerken sinds halverwege de vorige eeuw. Een van de oorspronkelijke motivaties om onderzoek te doen naar neurale netwerken was om beter inzicht te krijgen in de werking van de hersenen. Een aantal van de vroegste algoritmen waren bedoeld als biologische modellen. Sommige moderne algoritmen zoals *spiking neu-*

ral networks hebben nog steeds een nauw verband met de werking van hersencellen. De meeste kunnen echter beter niet gezien worden als realistische modellen van hersenfunctie en zijn zo ook al lang niet meer bedoeld.

In de jaren negentig werden een aantal serieuze beperkingen van neurale netwerken duidelijk. Hoewel neurale netwerken potentieel zeer krachtig zijn, is het niet altijd eenvoudig om complexe modellen te trainen zodat ze ook goede prestaties behalen. Hiervoor is veel rekenkracht en veel beschikbare trainingsdata nodig. Met andere woorden, neurale netwerken kunnen met de juiste parameters de meest ingewikkelde voorspellingen correct maken, maar het bepalen wát die juiste parameters precies zouden moeten zijn bleek zeer moeilijk en soms onmogelijk. Het feit dat neurale netwerken vaak miljoenen parameters hebben die allemaal tegelijkertijd geöptimaliseerd moeten worden maakt dit probleem niet eenvoudiger. Deze beperkingen zorgden ervoor dat neurale netwerken rond de eeuwwisseling minder populair werden. In de laatste tien tot twintig jaar zijn echter nieuwe technieken ontwikkeld die het mogelijk maken om zogenaamde diepe neurale netwerken te trainen middels *deep learning*, hierbij geholpen door het beschikbaar komen van enorme datasets en krachtige hardware. Dit heeft ertoe geleid dat de huidige neurale netwerken vaak betere prestaties behalen dan vele andere algoritmen, en het regelmatig beter doen dan menselijke deskundigen, wat hun populariteit ten goede is gekomen.

Het doel van dit vak is om inzicht te krijgen in de werking van neurale netwerken, en als het ware een kijkje onder de motorkap te nemen. Daartoe gaan we o.a. van de grond af aan een eigen neuraal netwerk implementeren in Python. Hoewel we natuurlijk niet zullen komen tot een implementatie die zich qua efficiëntie en mogelijkheden kan meten met bestaande bibliotheken met kant-en-klare routines (zoals keras, tensorflow, theano, pytorch, of scikit-learn), zullen we eigen neurale netwerken kunnen opzetten die alle essentiële kenmerken in zich bergen.

Opgave 1. *

Maak een kruistabel met twee rijen voor "supervised" en "unsupervised" algoritmen, en twee kolommen voor "nominale" en "numerieke" uitkomsten. Vul elke cel van deze tabel met een voorbeeld van een bijpassend algoritme, beschrijf in woorden de vorm van hun concepten, en geef een mogelijke toepassing.

Opgave 2. *

Is overfitting het beste te onderscheiden van underfitting aan de hand van de resubstitution error (op de trainingsdata) of aan de hand van de cross-validation error (op de testdata)? Motiveer je antwoord.

Opgave 3. *

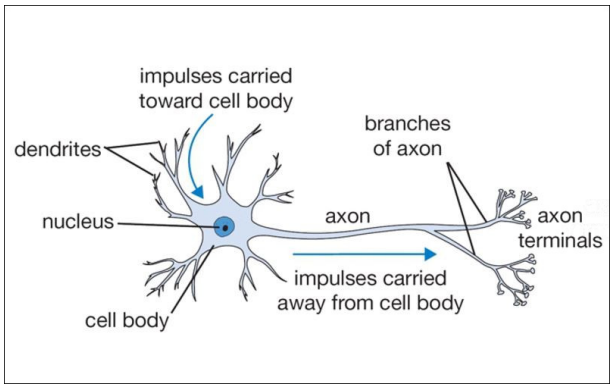
Naast supervised en unsupervised learning wordt vaak ook nog een categorie algoritmen onderscheiden onder de noemer *reinforcement learning*. Nog weer andere algoritmen heten *semi-supervised*. Wat wordt met deze termen bedoeld? Zoek hierover zonodig informatie op.

Opgave 4. **

Stel dat er onbeperkt veel trainingsdata beschikbaar zou zijn, zou het One-R algoritme dan gezien kunnen worden als een universele approximator? En hoe zit dat voor k-nearest neighbor, een (J48-)tree, Naive Bayes, of logistische regressie?

1.2. Het perceptron model

Het *perceptron* is in 1958 door Rosenblatt ontworpen als opvolger van het *McCulloch-Pitts neuron* en is de simpelste vorm van een neuraal netwerk. Dit zal door ons als uitgangspunt genomen worden om uiteindelijk een volledig neuraal netwerk op te bouwen. Om het te onderscheiden van multi-layer perceptrons (die we later zullen bekijken) wordt dit eerste eenvoudige model ook wel het *single-layer* perceptron genoemd. Een dergelijk perceptron bestaat uit één enkele rekeneenheid. Zoals een biologisch neuron typisch duizenden dendrieten heeft die invoer verzamelen vanuit andere neuronen, een cellichaam dat deze inputs combineert, en één axon dat vervolgens de uitkomst doorgeeft aan andere neuronen, zo kent ook het perceptron een meervoudig aantal inputs (de attributen $x_1, x_2, \ldots, x_d$), een bewerking die hierop toe wordt gepast (de functie f), en één uitkomst die de voorspelde klasse representeert (de waarde $\hat{y}$).



Het doel van het perceptron algoritme is om *binaire* (of *dichotome* of *binomiale*) classificatie uit te voeren op twee klassen. Het is gebruikelijk om een getalwaarde $y = -1$ te kiezen om de ene klasse aan te duiden en een waarde $y = +1$ voor de andere klasse. (Soms worden de waarden 0 en 1 gebruikt; de wiskundige formulering verandert dan licht maar de principes blijven hetzelfde.) Het perceptron begint ermee de attributen x_i te wegen met een serie gewichten w_i, en ze vervolgens op te tellen. Dit wordt een *lineaire combinatie* genoemd. Je verkrijgt hiermee een uitkomst

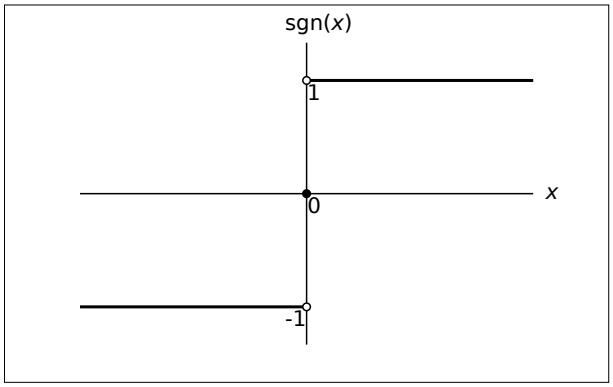
$$a = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_d \cdot x_d$$

Omdat zowel de attributen x_i als de gewichten w_i numerieke getalwaarden bevatten, is ook de uitkomst a numeriek. Aangezien het hier echter niet om regressie maar classificatie gaat, moet deze nog worden omgezet naar een voorspeld klasselabel $\hat{y}$. Het perceptron doet dit door het ene klasselabel $\hat{y} = -1$ toe te kennen als $a < 0$, of het andere klasselabel $\hat{y} = +1$ als $a > 0$. (Voor een uitkomst $a = 0$ die exact op de grens ligt kan een willekeurig

klasselabel worden toegekend, of wordt een uitkomst $\hat{y} = 0$ gekozen om aan te geven dat de voorspelling onzeker is; dit theoretische grensgeval is voor het perceptron echter nooit van praktische betekenis.)

De gewichten w_i geven aan wat de invloed van de waarde van een attribuut is op de uitkomst. Een gewicht nabij nul duidt erop dat het attribuut x_i niet of nauwelijks invloed heeft op de uitkomst en dus van weinig voorspellende waarde is. Een positief gewicht w_i geeft aan dat de waarde van het attribuut x_i typisch hoger is in de klasse $y = +1$ dan in de klasse $y = -1$; een negatief gewicht geeft het omgekeerde aan.

We kunnen het model summier neerschrijven door gebruik te maken van de *signum-functie* $\operatorname{sgn}(x)$ die een waarde ± 1 geeft (of eventueel 0) als het argument negatief of positief is (of eventueel nul).

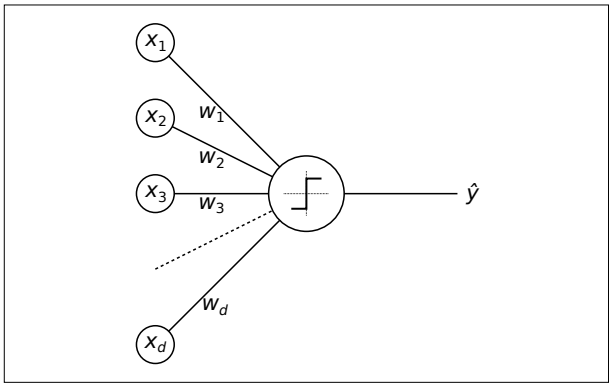


We krijgen dan de volgende formule voor het perceptron:

$$\hat{y} = \operatorname{sgn}\left(\sum_i w_i \cdot x_i\right)$$

We zullen later naast de signum-functie ook nog andere functies tegenkomen. In de context van neurale netwerken worden dit *activatiefuncties* genoemd. Dit grijpt weer terug op de analogie met een biologisch neuron waar het cellichaam bepaalt of het neuron actief wordt door een actiepotentiaal af te voeren langs het axon, of niet. De waarde a noemen we dan ook wel de *pre-activatiewaarde* (de waarde "voordat" de activatiefunctie wordt toegepast), en de waarde $\hat{y}$ is daarmee de *post-activatiewaarde* (de waarde "na-dat" de activatiefunctie wordt toegepast). Als we de activatiefunctie in het algemeen aanduiden met φ zijn de pre- en postactivatiewaarden aan elkaar gerelateerd volgens $\hat{y} = \varphi(a)$.

Het is vrij gebruikelijk om dergelijke formules en modellen te visualiseren in de vorm van een diagram. Hierin wordt een cirkel gebruikt om een neuron te symboliseren, vaak voorzien van een symbool dat de activatiefunctie weergeeft. Aan de linkerkant geeft een aantal inkomende lijnen of pijlen de inputs x_i vanuit alle attributen weer, elk voorzien van een indicatie van het gewicht w_i; aan de rechterkant verlaat één lijn of pijl de cirkel met de voorspelde uitkomst $\hat{y}$.



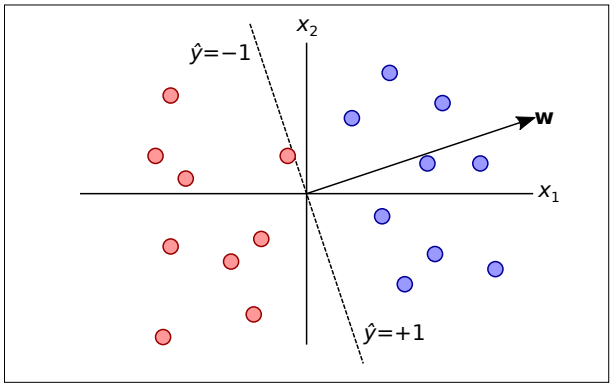
Een andere manier om de notatie te vereenvoudigen is door gebruik te maken van *vector* notatie. Vectoren worden vaak voorgesteld als pijlen met een zekere grootte en richting. Voor de discussie hier is het voldoende om een vector te zien als een reeks getalwaarden, vergelijkbaar met een *list* of *array* met een zekere vaste lengte. Zo kunnen we de attributen kortweg schrijven als een vector $\boldsymbol{x}$, waarbij $\boldsymbol{x} = [x_1, x_2, \ldots, x_d]$, en evenzo de bijbehorende gewichten $\boldsymbol{w}$ als $\boldsymbol{w} = [w_1, w_2, \ldots, w_d]$. De eerdere formule $a = \sum_i w_i \cdot x_i$ om de pre-activatiewaarde te berekenen kan dan worden vereenvoudigd tot

$$a = \boldsymbol{w} \cdot \boldsymbol{x}$$

Hierin staat de dot-notatie voor het *inproduct* van twee vectoren dat berekend wordt door de som te nemen van de producten van alle overeenkomstige elementen. Je kunt in de literatuur ook de matrixnotatie $a = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ voor kolomvectoren tegenkomen die op hetzelfde neerkomt. In het vervolg zullen wij de formules hier echter blijven uitschrijven met een sommatieteken.

Voor het perceptron ligt de grens tussen de twee klassen op die plek waarvoor de pre-activatiewaarde a gelijk is aan nul. Dat wil zeggen dat de grenslijn tussen de twee klassen wordt gedefinieerd door $w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_d \cdot x_d = 0$. Hierdoor is de grens tussen de twee klassen altijd een rechte lijn (of een recht vlak in drie dimensies, of een recht hypervlak in meerdere dimensies). Als de data daadwerkelijk op deze manier exact gescheiden kan worden dan noemen we deze data *lineair separabel*.

Voor het voorgestelde model is het niet moeilijk om in te zien dat deze rechte lijn precies door de oorsprong moet lopen. Immers, in de oorsprong zijn alle x_i aan nul. Als alle attributen gelijk zijn aan nul dan is ook elke gewogen combinatie van deze waarden gelijk aan nul, hetgeen $a = 0$ oplevert. Met andere woorden, de oorsprong van het coördinatenstelsel ligt precies op de scheidingslijn tussen de klassen. De richting van de scheidingslijn is op het eerste gezicht wat lastiger in te zien, maar kan worden begrepen door de gewichten w_i als een vector te bekijken. Het inproduct tussen twee vectoren is gelijk aan nul als de twee vectoren loodrecht op elkaar staan. Dat wil zeggen dat de scheidingslijn bestaat uit alle punten die ten opzichte van de oorsprong loodrecht staan op de vector $\boldsymbol{w}$.



Kortom, teken vanuit de oorsprong de vector $\boldsymbol{w}$, bepaal de hierop loodrechte lijn door de oorsprong, en je hebt de scheidingslijn volgens het model voor een gegeven set gewichten w_i gevonden. In de figuur hierboven is de grenslijn loodrecht op de vector $\boldsymbol{w}$ aangegeven met een stippellijn. Alle (blauwe) punten die aan de kant van de lijn liggen waar de vector $\boldsymbol{w}$ naartoe wijst krijgen het label $\hat{y} = +1$ toegewezen; de (rode) punten aan de andere kant van de lijn krijgen het label $\hat{y} = -1$.

Opgave 5. *
Teken een grafiek met daarin de negen punten $(\pm 2, \pm 1)$, $(\pm 1, \pm 2)$ en $(0, 0)$. Bepaal voor al deze punten tot welke klasse $\hat{y} = \pm 1$ deze punten behoren volgens een perceptron met gewichten $\boldsymbol{w} = [-3, 2]$. Schets in je figuur de gemodelleerde scheidingslijn tussen de twee klassen, en teken de vector $\boldsymbol{w}$.

Opgave 6. *
Stel we beschouwen een dataset met drie attributen x_1, x_2 en x_3. We passen hierop een perceptron toe met gewichten $\boldsymbol{w} = [1, -2, 3]$. Welk label wordt toegekend aan een instance A met $\boldsymbol{x}_A = [0, 1, 2]$? En een instance B met $\boldsymbol{x}_B = \left[\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right]$? Bedenk zelf een voorbeeld van een instance C met attributen die precies op de grenslijn van dit perceptron liggen (anders dan $\boldsymbol{x}_C = [0, 0, 0]$).

Opgave 7. **
Is het waar dat voor alle getallen a en b geldt dat $\operatorname{sgn}(a + b) = \operatorname{sgn}(a) + \operatorname{sgn}(b)$? En is $\operatorname{sgn}(ab) = \operatorname{sgn}(a) \cdot \operatorname{sgn}(b)$? Geef telkens ofwel een voorbeeld waaruit blijkt dat het niet geldt, of leg uit waarom het wel geldt.

Opgave 8. **
Leg uit dat de signum-functie geschreven kan worden als $\operatorname{sgn}(x) = \frac{|x|}{x}$ voor $x \neq 0$, waarbij $|x|$ voor de absolute waarde van x staat.

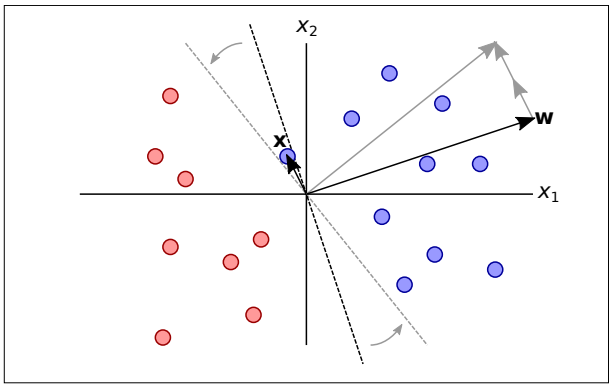
1.3. Optimalisatie

We weten nu dus hoe je de scheidingslijn kan bepalen voor een model met een zekere set gewichten, maar we weten nog niet hoe je geschikte waarden voor die gewichten kan

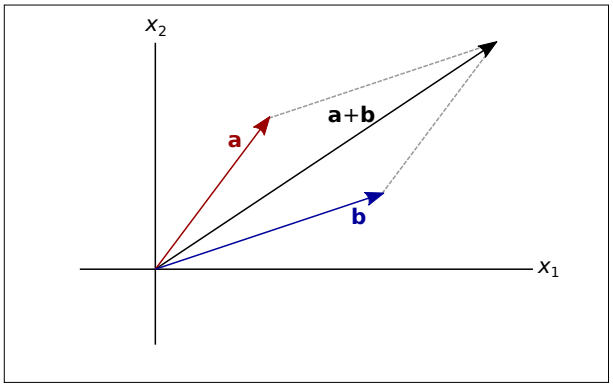
bepalen die de datapunten van twee lineair separabele klassen weten te scheiden. Het perceptron kent hiervoor gelukkig een buitengewoon eenvoudig recept.

Het idee is om voor alle instances één voor één te controleren of ze juist geclassificeerd worden. Als dat voor alle instances het geval is ben je klaar met de optimalisatie. Als er echter een instance gevonden wordt die verkeerd wordt geclassificeerd, dat wil zeggen het datapunt ligt aan de verkeerde kant van de scheidingslijn, dan wordt de scheidingslijn een stapje gedraaid in een zodanige richting dat het foutief geclassificeerde punt beter komt te liggen ten opzichte van de lijn.

Laten we een voorbeeld nemen van een punt $\boldsymbol{x}$ met als gewenst klasselabel $y = +1$, maar met als voorspelde uitkomst $\hat{y} = -1$. De error bedraagt dan $e = \hat{y} - y = (-1) - 1 = -2$. In de onderstaande figuur staat een voorbeeld van een dergelijke situatie geschetst. Er is één blauw punt $\boldsymbol{x}$ zichtbaar dat aan de verkeerde ("rode") kant van de lijn is komen te liggen.



Uit de figuur blijkt dat het wenselijk is om in dit geval de scheidingslijn tegen de klok in te roteren. Hiervoor is het noodzakelijk om ook de vector $\boldsymbol{w}$ tegen de klok in te roteren. Dit kunnen we bereiken door er een kleine vector bij op te tellen met een soortgelijke richting als de vector $\boldsymbol{x}$ zelf. Vectoren kun je optellen door alle overeenkomende elementen van de vector op te tellen (bv. $[1, 2] + [3, 1] = [4, 3]$). Of, in een plaatje kunnen we gebruik maken van de *kop-staart-regel*.



De precieze updateregel van het perceptron is iets algemener en luidt:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - (\hat{y} - y)\,\boldsymbol{x}$$

Door het meenemen van de fout $e = \hat{y} - y$ wordt de scheidingslijn de juiste kant op gedraaid. Voor het geval hierboven is $\hat{y} - y = -2$ negatief en wordt de vector $\boldsymbol{x}$ tweemaal bij de vector $\boldsymbol{w}$ opgeteld. Het resultaat is de in grijs afgebeelde gedraaide vector en scheidingslijn. In dit geval is het oorspronkelijk foutief geclassificeerde punt aan de juiste kant van de lijn komen te liggen. Dit hoeft overigens niet altijd meteen het geval te zijn, maar de lijn verschuift wel altijd in de goede richting. Merk tevens op dat wanneer het ene punt beter komt te liggen, het best zo kan zijn dat een ander punt juist slechter komt te liggen.

Wanneer een omgekeerde classificatiefout wordt gemaakt, dat wil zeggen aan een instance met klasselabel $y = -1$ wordt $\hat{y} = +1$ toegekend, dan moet de scheidingslijn andersom gedraaid en moeten de vectoren van elkaar worden afgetrokken. De factor $\hat{y} - y$ in de formule hierboven zorgt voor het correcte teken. Daarnaast zorgt deze factor er ook voor dat instances die reeds juist worden geclassificeerd geen effect meer hebben op de gewichten w_i. Immers, in dat geval is de fout $\hat{y} - y = 0$ en wordt de vector $\boldsymbol{w}$ bijgewerkt met de *nulvector*, hetgeen niet leidt tot een andere waarde.

Op deze manier kunnen één voor één alle n instances behandeld worden. Alle fout geclassificeerde instances leiden tot het bijwerken van de gewichten w_i. Zodra er geen verkeerd geclassificeerde instances meer zijn wordt het algoritme beëindigd en is een optimale classifier gevonden. Mogelijk zijn hier meerdere gangen door alle instances voor nodig, genaamd *epochs*.

Resteert nog de vraag hoe je de waarde van $\boldsymbol{w}$ het beste kan initialiseren. Aannemende dat van tevoren niet bekend is in welke richting de scheidingslijn ongeveer zou moeten lopen is elke richting voor de vector $\boldsymbol{w}$ even geschikt. Het is daarom gebruikelijk om aanvankelijk eenvoudigweg alle $w_i = 0$ te kiezen.

Opgave 9. *

Maak zelf een soortgelijke schets als de figuur hierboven, maar dan voor een instance met als gewenste klasselabel $y = -1$ en voorspelde uitkomst $\hat{y} = +1$. Ga na dat ook hiervoor de updateregel van het perceptron de vector $\boldsymbol{w}$ aanpast in de juiste richting.

Opgave 10. *

Stel, een perceptron met gewichten $\boldsymbol{w} = [1, -4, 5]$ wordt toegepast op een instance $\boldsymbol{x} = \left[0, \frac{1}{2}, 1\right]$ en met het ware label $y = -1$. Wat worden de nieuwe gewichten nadat de update regel één keer is toegepast? Herhaal de updateregel totdat het perceptron deze instance correct classificeert: wat zijn de uiteindelijke gewichten $\boldsymbol{w}$?

Opgave 11. **

Als in plaats van labels $y = \pm 1$ de waarden $y = 0$ en $y = +1$ gebruikt zouden worden, samen met een model met een activatiefunctie die eveneens leidt tot voorspellingen $\hat{y} = 0$ of $\hat{y} = +1$ (de heaviside-stapfunctie), zou dan de updateregel $\boldsymbol{w} \leftarrow \boldsymbol{w} - (\hat{y} - y)\,\boldsymbol{x}$ nog correct blijven werken, of moet deze veranderd worden?

1.4. Bias

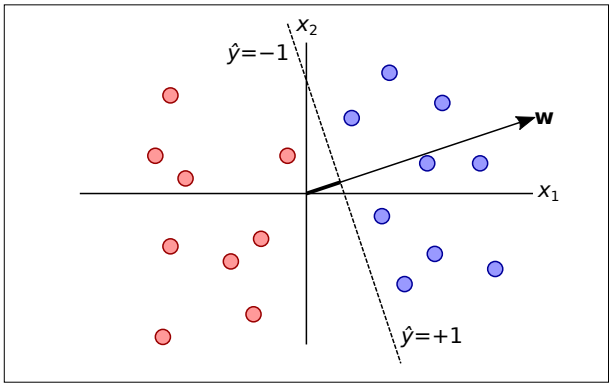
Tot dusverre hebben we als eis gesteld dat de data lineair separabel moeten zijn middels een rechte lijn door de oorsprong. In de praktijk is dat een onrealistisch zware eis. Het is echter vrij eenvoudig om dit te veralgemeniseren tot een rechte lijn die niet per se door de oorsprong hoeft te gaan. We kunnen dit bereiken door aan de pre-activatiewaarde een extra *bias* b toe te voegen, zodat we krijgen

$$a = b + w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_d \cdot x_d$$

De classificatieregel wordt dan:

$$\hat{y} = \operatorname{sgn}\left(b + \sum_i w_i \cdot x_i\right)$$

Het is nu niet moeilijk in te zien dat de oorsprong waar alle attributen de waarde nul hebben niet langer op de scheidingslijn hoeft te liggen. In dat geval is immers alle $x_i = 0$ en wordt $\hat{y} = \operatorname{sgn}(b)$, oftewel de oorsprong wordt hetzelfde label toegekend als het teken van de bias b. De bias geeft aan hoe sterk het model een voorkeur heeft om een bepaald klasselabel toe te kennen als er verder geen inputs x_i zouden hoeven te worden meegewogen. Als de bias $b > 0$, dan heeft het model een ingebouwde voorkeur voor het label $\hat{y} = +1$; voor $b < 0$ neigt het eerder naar $\hat{y} = -1$. Hoe groter de absolute waarde van b, hoe verder de scheidingslijn van de oorsprong komt te liggen.

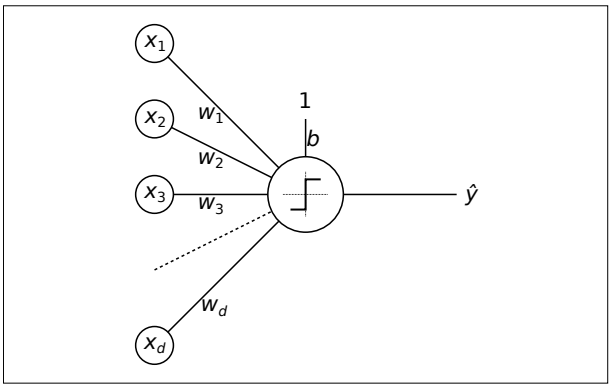


Hiermee is nog niet gezegd hoe de gewichten w_i en de bias b geoptimaliseerd kunnen worden aan de hand van een concrete lineair separabele dataset. Echter, we kunnen een slimme truc toepassen om de bovenstaande formule te reduceren tot het simpelere geval zonder bias. Beschouw hiertoe de bias b als onderdeel van de te optimaliseren gewichten w_i: laten we voor het gemak een tijdelijk extra gewicht w_0 toevoegen dat we identificeren met b.

$$a = \underbrace{w_0}_{=b} \cdot \underbrace{x_0}_{=1} + w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_d \cdot x_d$$

In de formule wordt w_0 vermenigvuldigd met x_0. Als we elke instance een denkbeeldig attribuut $x_0 = 1$ meegeven, en de sommatie laten lopen vanaf $i = 0$, dan is het effect

dat er een extra term $w_0 \cdot x_0$ wordt opgeteld bestaande uit een gewicht dat altijd wordt vermenigvuldigd met 1. Dit heeft natuurlijk exact dezelfde uitwerking als het optellen van een constante bias, als we w_0 terug hernoemen naar b. In het diagram wordt dit soms expliciet aangegeven met een extra input met waarde 1 en gewicht b.

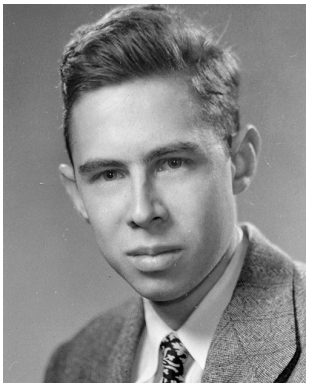


We kunnen precies dezelfde updateregel toepassen op het gewicht w_0 als hierboven op alle andere gewichten werd toegepast. Wanneer we ons realiseren dat x_0 altijd gelijk is aan 1, dan levert dit op $b \leftarrow b - (\hat{y} - y) \cdot 1$, oftewel samengevat:

$$\begin{cases} b \leftarrow b - (\hat{y} - y) \\ w_i \leftarrow w_i - (\hat{y} - y)\, x_i \end{cases}$$

Het algoritme is voltooid zodra alle instances juist worden geclassificeerd. Er worden daarna immers geen updates meer uitgevoerd.

Rosenblatt bewees rigoreus dat wanneer de data twee lineair separabele klassen vormen, het perceptron algoritme altijd convergeert en een scheidingslijn bepaalt die de klassen perfect van elkaar onderscheidt!



Opgave 12. *

Teken een grafiek met daarin de negen punten $(\pm 2, \pm 1)$, $(\pm 1, \pm 2)$ en $(0, 0)$, en bepaal tot welke klasse deze punten behoren volgens een perceptron met gewichten $\boldsymbol{w} = [-3, 2]$ en bias $b = -4$. Schets in je figuur de gemodelleerde scheidingslijn tussen de twee klassen, samen met de vector $\boldsymbol{w}$.

Opgave 13. *
Is het teken van de bias b in de eerste figuur van deze paragraaf hierboven positief of negatief? Leg je antwoord uit.

Opgave 14. **
Gegeven is het onderstaande datasetje met twee numerieke attributen x_1 en x_2 en één klasselabel $y = \pm 1$. Maak een schets van de ligging van deze datapunten in een coördinatenstelsel met twee assen. Is deze dataset lineair separabel? Voer handmatig het perceptron algoritme uit, geïnitialiseerd met $b = w_1 = w_2 = 0$, beginnend met het eerste datapunt, net zo lang door de opeenvolgende datapunten heen roulerend tot alle instances juist geclassificeerd worden. Hoe ligt de uiteindelijke scheidingslijn?

x_1	x_2	y
1	0	+1
0	1	+1
−1	0	−1
0	−1	−1

Opgave 15. **
Gegeven is het onderstaande datasetje met twee numerieke attributen x_1 en x_2 en één klasselabel $y = \pm 1$. Maak een schets van de ligging van deze datapunten in een coördinatenstelsel met twee assen. Is deze dataset lineair separabel? Voeg nu een extra attribuut toe dat je berekent uit de bestaande attributen volgens $x_3 = x_1 \cdot x_2$. Zal het perceptron algoritme convergeren als je het toepast op deze uitgebreide dataset?

x_1	x_2	y
1	1	+1
1	−1	−1
−1	1	−1
−1	−1	+1

1.5. Lineaire regressie

Met een paar aanpassingen kan het perceptron ook worden ingezet om lineaire regressie uit te voeren. Om te beginnen zal de uitkomst niet moeten worden omgezet naar waarden $\hat{y} = \pm 1$, maar dienen numerieke getalwaarden te worden geproduceerd. Dit kan eenvoudig worden bereikt door de signum-functie te verwijderen. We krijgen dan:

$$\hat{y} = b + \sum_i w_i \cdot x_i$$

Dit is eigenlijk equivalent aan het toepassen van een activatiefunctie $\varphi(a) = a$, dat wil zeggen de *identiteitsfunctie*, aangezien deze de post-activatiewaarde gewoon gelijk houdt aan de pre-activatiewaarde. Het resultaat is het hopelijk bekende (multi-)lineaire regressie model.

De updateregel blijkt ook vrijwel dezelfde vorm te krijgen. We moeten echter opletten dat voor het perceptron eigenlijk alleen de richting van de vector $\boldsymbol{w}$ relevant is. De

signum-functie kijkt namelijk niet naar de grootte van de pre-activatiewaarde a, alleen naar het teken. Nu de signum-functie is verwijderd gaat de grootte van $\boldsymbol{w}$ wel een belangrijke rol spelen. Immers, hoe groter de gewichten, hoe groter ook de uitkomsten. Het gevolg hiervan is dat we ook de grootte van de updates zorgvuldiger moeten kiezen om tot een goede oplossing te komen. We doen dit door een extra coëfficient α te introduceren, de *learning rate*. Hoe groter deze factor, hoe groter de updates van de waarden w_i en b. De updateregel voor lineaire regressie luidt:

$$\begin{cases} b \leftarrow b - \alpha\,(\hat{y} - y) \\ w_i \leftarrow w_i - \alpha\,(\hat{y} - y)\,x_i \end{cases}$$

Deze is nagenoeg identiek aan die van het perceptron. Merk wel op dat in dit geval zowel $\hat{y}$ als y numerieke getalwaarden zijn en de fout $e = \hat{y} - y$ daardoor allerlei mogelijke waarden kan aannemen. Omgekeerd kun je bovenstaande updateregel prima op het perceptron toepassen; het ligt voor de hand om dan $\alpha = 1$ te kiezen, hoewel je kan aantonen dat de waarde van α er voor het perceptron eigenlijk niet toe doet.

Als je de waarde van α bij lineaire regressie te klein neemt, dan zijn er een heleboel iteraties nodig om tot een optimale oplossing te komen omdat het model telkens maar een klein beetje wordt aangepast. Kies je α echter te groot, dan is het mogelijk dat je de vergelijking telkens te sterk aanpast, over de juiste oplossing heenschiet, en daardoor nooit convergeert. In complexere neurale netwerken wordt α vaak rond de 0.001 of 0.01 gekozen. Dit kan afhangen van de dataset en van het model en is veelal een kwestie van experimenteren: als het algoritme langzamer convergeert dan je zou verwachten heeft het soms zin om α te vergroten; als de parameters wild heen en weer springen en helemaal niet convergeren is dat een teken dat α wel eens te groot gekozen zou kunnen zijn. We zullen in latere hoofdstukken *adaptive learning* heuristieken bekijken die in staat zijn om de waarde van α automatisch te kiezen en gaandeweg aan te passen. Voor nu laten we het probleem van het kiezen van een geschikte learning rate aan de gebruiker van het perceptron om op te lossen.

Een belangrijk verschil tussen het perceptron en lineaire regressie is dat het perceptron voor lineair separabele data gegarandeerd in een eindig aantal stappen convergeert naar een optimaal model dat alle trainingsinstances correct classificeert. Voor lineaire regressie is dat meestal niet het geval: hierbij zie je dat de oplossing geleidelijk steeds beter wordt, maar de fout typisch nooit helemaal gelijk aan nul wordt.

Overigens bestaan er voor lineaire regressie exacte (matrix-)formules die in één keer de juiste oplossing geven. De hier behandelde numerieke oplossingsmethode is dus niet per se het meest praktische model, maar maakt wel de elegante parallel met het perceptron duidelijk en toont aan dat dit soort modellen zowel voor classificatie als voor regressie kunnen worden ingezet.

Opgave 16. *

Teken een grafiek met daarin de negen punten $(\pm 2, \pm 1)$, $(\pm 1, \pm 2)$ en $(0, 0)$, en bepaal de voorspellingen die aan deze punten worden toegekend door een lineair regressiemodel met gewichten $\boldsymbol{w} = [-3, 2]$ en bias $b = -4$.

Opgave 17. **

Gegeven is het onderstaande datasetje met slechts één numeriek attribuut x_1 en één numerieke uitkomst y. Maak een schets van de ligging van deze datapunten in een grafiek. Voer handmatig het lineaire regressie algoritme uit met learning rate $\alpha = \frac{1}{2}$, geïnitialiseerd met $b = w_1 = 0$, beginnend met het eerste datapunt, en stop na maximaal drie epochs. Hoe ligt de uiteindelijk gefitte lijn?

x_1	y
-1	-1
1	3

Opgave 18. **

Herhaal de bovenstaande opgave met learning rate $\alpha = 1$; wat valt je op?

Opgave 19. ***

Leg uit waarom het voor het perceptron eigenlijk niet uitmaakt hoe groot je de learning rate α kiest als je de updateregel van lineaire regressie zou toepassen.

2. Generieke neuronen

Vooruitblik

In dit hoofdstuk zullen we het perceptron en lineaire regressie algoritme veralgemeniseren tot het model van een neuron dat een activatiefunctie en een lossfunctie kent. Door de loss te minimaliseren middels stochastic gradient descent blijken we de modellen uit het vorige hoofdstuk te kunnen reproduceren. Met andere keuzes van deze functies kunnen we bijvoorbeeld ook logistische regressie uitvoeren. Dit leidt uiteindelijk tot een zeer algemeen model dat de bouwsteen van neurale netwerken zal worden.

2.1. Lossfunctie

In het voorgaande hoofdstuk hebben we gezien dat het perceptron algoritme in staat is om lineair separabele trainingsdata foutloos te classificeren. Het deed dat door, beginnend met een bias $b = 0$ en gewichten $w_i = 0$, voor de instances een klasselabel $\hat{y} = \operatorname{sgn}(b + \sum_i w_i \cdot x_i)$ te voorspellen, en aan de hand daarvan de parameters bij te werken volgens de regels $b \leftarrow b - (\hat{y} - y)$ en $w_i \leftarrow w_i - (\hat{y} - y)\, x_i$, net zolang totdat alle instances juist worden voorspeld. Een soortgelijk model kon worden geformuleerd voor lineaire regressie, waarbij dan de signum-functie werd vervangen door de identiteitsfunctie en een learning rate α als extra coëfficiënt werd toegevoegd aan de updateregel. In dit hoofdstuk gaan we dezelfde modellen wat rigoreuzer afleiden, met als bonus dat we ook een aantal andere modellen (zoals logistische regressie) kunnen begrijpen en optimaliseren, waardoor we niet langer beperkt blijven tot lineair separabele datasets.

We beginnen met terug te kijken naar hoe regressiemodellen traditioneel worden geoptimaliseerd. Hierbij wordt gebruik gemaakt van de *kleinste-kwadraten-methode*, oftewel *least squares*. Een regressiemodel dat aan een instance met een uitkomst y een voorspelling toekent gelijk aan $\hat{y}$ maakt hierbij een fout $e = \hat{y} - y$. Omdat fouten naar boven en naar beneden even ernstig zijn worden deze fouten gewoonlijk gekwadrateerd om het teken kwijt te raken. We definiëren hiertoe een kwadratische *lossfunctie* $\mathcal{L}(\hat{y}; y) = (\hat{y} - y)^2$ die aangeeft hoe ernstig een gemaakte fout is.

De lossfunctie levert een loss $l = \mathcal{L}(\hat{y}; y) = 0$ als de voorspelde uitkomst exact overeenkomt met de gewenste uitkomst, en heeft altijd een strict positieve waarde $l > 0$ als hierin een afwijking optreedt. Kortom, we kunnen nu zeggen dat een oplossing beter is naarmate l lager is. Hiermee is het probleem omgezet in een optimalisatievraagstuk waarin we op zoek moeten naar het *minimum* van een functie. De parameters die we kunnen variëren zijn de bias b en de gewichten w_i.

We zullen voor het gemak de bias weer even tijdelijk beschouwen als een van de ge-

wichten (w_0) die altijd gecombineerd wordt met een constant attribuut ($x_0 = 1$), zodat we niet speciaal rekening hoeven te houden met de bias als we de gewichten w_i tezamen optimaliseren. Alles wat hieronder over de gewichten w_i wordt geschreven geldt dus soortgelijk ook voor de bias b.

De vraag is nu: hoe kunnen we $\boldsymbol{w}$ zodanig kiezen dat de uitkomst l zo klein mogelijk is?

De berekening van de loss l geschiedt eigenlijk in drie stappen:

1. Eerst worden de gewichten gecombineerd met de waarden van de attributen om de pre-activatiewaarde te berekenen:

$$a = \sum_i w_i \cdot x_i$$

   Kortom, a hangt rechtstreeks af van w_i, en als je wil weten hoe a varieert als w_i verandert dan hoef je alleen de waarden van de attributen x_i te kennen.

2. Daarna wordt de pre-activatiewaarde in de activatiefunctie gestopt om de post-activatiewaarde te berekenen:

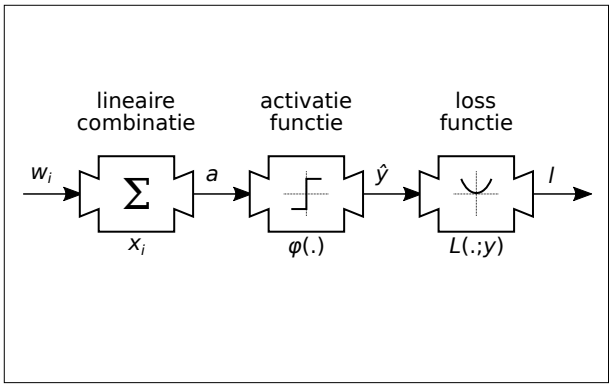
$$\hat{y} = \varphi(a)$$

   Kortom, $\hat{y}$ hangt rechtstreeks af van a, en als je wil weten hoe $\hat{y}$ varieert als je a verandert dan hoef je alleen de vorm van de activatiefunctie φ te kennen.

3. Tenslotte wordt de post-activatiewaarde in de lossfunctie gestopt om de loss te berekenen:

$$l = \mathcal{L}(\hat{y}; y)$$

   Kortom, l hangt rechtstreeks af van $\hat{y}$, en als je wil weten hoe l varieert als je $\hat{y}$ verandert dan hoef je alleen de vorm van de lossfunctie $\mathcal{L}$ te kennen (voor een gegeven gewenste uitkomst y voor het betreffende instance).



We kunnen dit proces visualiseren als een reeks van drie transformaties, zoals in de figuur hierboven. Links gaan de gewichten erin, die worden omgezet in de pre-activatiewaarde middels de attributen x_i, die worden omgezet in de post-activatiewaarde

middels de activatiefunctie φ, die worden omgezet in de loss middels de lossfunctie $\mathcal{L}$. Dit machientje wordt ook wel een *neuron* genoemd, naar analogie met hersencellen.

Om de loss te minimaliseren dienen we te weten hoe l reageert als we de waarden van alle w_i één voor één veranderen. Alle gewichten zijn als het ware instelknoppen van het neuron waaraan we kunnen draaien. Hoe moeten we nu aan die knoppen draaien om de uitvoer (dat wil zeggen: de loss) omlaag te krijgen? Om dat te bepalen is het voldoende om te weten hoe de uitvoer van elk van de drie genoemde stappen afhangt van hun invoer. Preciezer geformuleerd, als we één van de gewichten verhogen van de huidige waarde w_i naar een nieuwe waarde $w_i + \Delta w_i$, dan zal de pre-activatiewaarde meeveranderen van a naar een zekere $a + \Delta a$, wat op zijn beurt leidt tot een verandering van de post-activatiewaarde van $\hat{y}$ naar $\hat{y} + \Delta\hat{y}$, hetgeen tenslotte leidt tot een verandering van de loss van l naar $l + \Delta l$. We gebruiken de notatie Δ om veranderingen aan te geven; dit kan gaan om een toename als $\Delta w_i > 0$ of ook om een afname als $\Delta w_i < 0$. We willen nu weten, als we de gewichten veranderen met een hoeveelheid Δw_i, hoe groot zijn dan achtereenvolgens de bijbehorende Δa, $\Delta\hat{y}$, en Δl? Het doel is daarbij om de loss l te verlagen, dus te bereiken dat $\Delta l < 0$.

Opgave 20. *

Leg uit waarom zelfs de meest optimale oplossing die bij lineaire regressie gevonden kan worden slechts zelden een loss $l = 0$ oplevert.

Opgave 21. ***

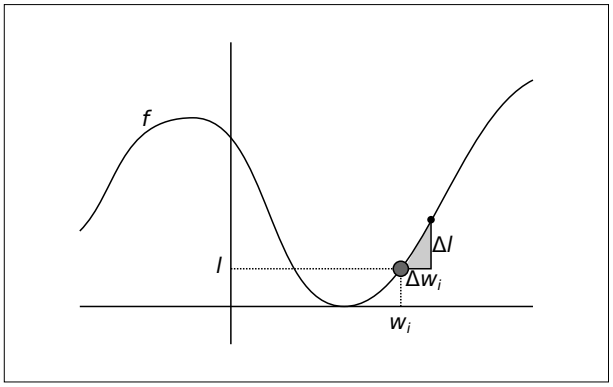
Welke van de onderstaande functies zou je in principe ook als lossfunctie kunnen gebruiken? Bedenk wat vereiste eigenschappen zijn voor een lossfunctie en leg uit wat er mis is met sommige van deze functies.

- $\mathcal{L}(\hat{y}; y) = \hat{y} - y$
- $\mathcal{L}(\hat{y}; y) = (\hat{y} - y)^2$
- $\mathcal{L}(\hat{y}; y) = |\hat{y} - y|$
- $\mathcal{L}(\hat{y}; y) = \hat{y}^2 - y^2$
- $\mathcal{L}(\hat{y}; y) = (\hat{y} + y)^2$

2.2. Gradient descent

Er zijn diverse methoden om minima van een functie te bepalen, maar een algemeen bruikbare methode is *gradient descent*. Deze methode kan ook toegepast worden op de afhankelijkheid in de loss van de gewichten. Gradient descent werkt door te beginnen met een willekeurige startwaarde voor w_i, en dan te kijken naar de helling van de functie die beschrijft hoe de loss afhangt van de gewichten. Als de helling ongelijk aan nul is dan kan de loss worden verkleind door een stap te zetten in de neergaande richting van de helling.

In het voorbeeld hieronder is een willekeurig verloop van de loss l geschetst met slechts één argument w_i. Noemen we deze functie even f, dan is dus $l = f(w_i)$; in de vorige paragraaf hebben we gezien dat deze functie f eigenlijk uit drie opeenvolgende transformaties bestaat die afhangen van x_i, φ en $\mathcal{L}$.



De helling van de functie f ter plekke van het aangegeven grijze punt kun je in de praktijk redelijk schatten door een klein stapje opzij te zetten, bijvoorbeeld naar de zwarte stip. Laat je w_i toenemen tot $w_i + \Delta w_i$ dan correspondeert dat met een toename van l tot een zekere nieuwe waarde $l + \Delta l$. De helling van de functie ter plekke van dit punt komt overeen met de steilheid van het getekende grijze driehoekje als je zorgt dat de grootte van het stapje voldoende klein is. Deze steilheid kun je karakteriseren met de verhouding $\frac{\Delta l}{\Delta w_i}$, ook wel de *afgeleide* genoemd. Als de functie f *lineair* is en een rechte lijn beschrijft ken je deze helling vermoedelijk als de *richtingscoëfficient*.

Omdat enerzijds voor het grijze punt geldt dat $l = f(w_i)$ en anderzijds voor de zwarte stip geldt dat $l + \Delta l = f(w_i + \Delta w_i)$, kun je schrijven $\Delta l = f(w_i + \Delta w_i) - f(w_i)$. Hieruit volgt voor de afgeleide:

$$\frac{\Delta l}{\Delta w_i} = \frac{f\left(w_i + \Delta w_i\right) - f\left(w_i\right)}{\Delta w_i}$$

Hetzelfde kan bereikt worden door een stapje in de ene richting én een stapje in de andere richting te maken en die twee punten met elkaar te vergelijken. Dat leidt tot dit resultaat:

$$\frac{\Delta l}{\Delta w_i} = \frac{f\left(w_i + \Delta w_i\right) - f\left(w_i - \Delta w_i\right)}{2\Delta w_i}$$

Als de stapjes maar voldoende klein zijn en f een gladde functie is dan is de uitkomst van beide uitdrukkingen gelijk, maar de tweede uitdrukking is in de praktijk iets nauwkeuriger.

Als de stapjes *infinitesimaal* klein gemaakt worden, wat wil zeggen willekeurig klein maar nog net niet nul, dan spreken we ook wel van een *differentiaal* $\mathrm{d}l$ of $\mathrm{d}w_i$; de notatie met de griekse letter Δ wordt dan vervangen door een d, en we vinden een afgeleide $\frac{\mathrm{d}l}{\mathrm{d}w_i}$. Deze notatie ken je misschien al wel. Als daarenboven de functie f afhangt van meerdere argumenten $w_0, w_1, w_2, \ldots, w_d$ en we alleen maar kijken naar de invloed van een verandering in één van de argumenten tegelijkertijd, dan spreken we van een *partiële*

differentiaal ∂l of ∂w_i. We zullen vanaf nu deze laatste notatie gebruiken omdat die het meest algemeen is en ook in boeken en artikelen op het gebied van machine learning veel voorkomt. De helling van de lossfunctie wordt daarmee genoteerd als $\frac{\partial l}{\partial w_i}$. In gedachten mag je houden dat het bij al deze verschillende notaties steeds over de verhouding tussen kleine verschillen gaat, zoals afgebeeld in de figuur hierboven.

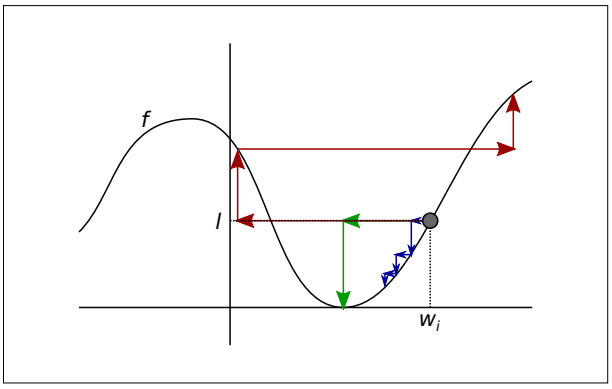
Keren we terug naar gradient descent, dan is het idee dat als de functie toeneemt, en dus de helling van de functie positief is, dat we dan de waarde van l kunnen laten afnemen door de waarde van w_i te verkleinen. Omgekeerd, als de functie afneemt, en de helling negatief is, dan moet w_i worden vergroot om in de richting van een minimum in l te bewegen. Hieruit kunnen we de updateregel $w_i \leftarrow w_i - \frac{\partial l}{\partial w_i}$ afleiden. Ga voor jezelf na dat de gewichten dan inderdaad worden verkleind als de helling $\frac{\partial l}{\partial w_i}$ positief is en worden vergroot als de helling negatief is.

Hoe groot de stap in het ideale geval moet zijn hangt af van de sterkte van de kromming van de functie. Deze kennen we niet. Daarom gebruiken we wederom de learning rate α als coëfficiënt om de stapgrootte mee in te kunnen stellen. We krijgen uiteindelijk:

$$w_i \leftarrow w_i - \alpha \cdot \frac{\partial l}{\partial w_i}$$

Soms wordt hiervoor ook wel de vectornotatie $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \cdot \boldsymbol{\nabla_w} l$ gebruikt, waarbij dan de *gradiënt* een vector is gelijk aan $\boldsymbol{\nabla_w} l = \left[\frac{\partial l}{\partial w_0}, \frac{\partial l}{\partial w_1}, \frac{\partial l}{\partial w_2}, \ldots, \frac{\partial l}{\partial w_d}\right]$. Hier komt de naam van de methode vandaan.

In de figuur hieronder is te zien hoe het bijwerken van de gewichten kan verlopen. Beginnen we weer met een willekeurige waarde van w_i, gemarkeerd met het grijze punt. De functie loopt hier naar boven, oftewel de loss neemt toe met het gewicht. Om een minimum op te sporen in het geval van een dergelijke positieve helling moeten we de waarde van het gewicht verkleinen. Met de gekleurde pijlen is aangegeven hoe dit in zijn werk kan gaan.



- Voor de groene pijlen is de learning rate α toevallig optimaal gekozen. De waarde van w_i wordt in één keer verlaagd tot in het "dal" van de functie. Hier loopt de functie horizontaal en is de helling gelijk aan nul, dus hierna hoeft het gewicht niet meer bijgewerkt te worden. Dit blijkt ook uit de gradient descent regel: als de

afgeleide $\frac{\partial l}{\partial w_i}$ gelijk is aan nul worden de gewichten niet meer veranderd. Overigens is het niet gegarandeerd dat je nu ook in het *globale* minimum terecht bent gekomen; het zou ook een sub-optimaal *lokaal* minimum kunnen zijn. Maar in de praktijk van machine learning blijkt dit meestal niet tot al te grote problemen te leiden.

- Als je de waarde van de learning rate α te klein neemt, dan neem je te kleine stapjes. Dit is aangegeven in blauw. Weliswaar nader je in de richting van het minimum, maar de stap is niet groot genoeg om dit helemaal te bereiken. Pas je daarna nogmaals gradient descent toe, dan zul je dus nog een verder stapje zetten in de goede richting, enzovoorts. Op deze manier zul je uiteindelijk wel uiterst nabij het minimum uitkomen, maar je bereikt het nooit helemaal exact, en je hebt er ook veel meer stappen voor nodig dan strict noodzakelijk. Vroeg of laat wordt de oplossing echter willekeurig goed, dus dit gedrag is acceptabel.

- In rood is geïllustreerd wat er kan gebeuren als je de learning rate α te groot neemt, met een gigantische stapgrootte tot gevolg. Dit kan ertoe leiden dat je over het minimum heen schiet en op een ander deel van de functie terechtkomt. In de figuur beland je op een deel met een steile negatieve helling, waardoor de volgende stap de andere kant opgaat en nog groter is. Je merkt dat je in dit geval het risico loopt alleen maar verder van het minimum vandaan te belanden. De methode convergeert dan niet. Dit is natuurlijk onwenselijk.

Helaas is in de praktijk de ideale waarde van de learning rate α niet goed te voorspellen, aangezien dat van de vorm van de te optimaliseren functie afhangt. Er bestaan heuristieken om deze geschikt te kiezen of gaandeweg aan te passen, maar in afwachting daarvan is een veilige waarde zoals $\alpha = 0.01$ of kleiner aan te raden.

Bij het minimaliseren van de loss wordt door alle trainingsinstances heen gelopen. Elke volgende stap die je zet wordt berekend op grond van telkens een andere instance. Eigenlijk optimaliseren we met elke instance een net iets andere functie: de ene iteratiestap trainen we het model richting het beter herkennen van een eerste instance, en in een volgende stap trainen we het model richting het beter herkennen van een tweede instance, enzovoorts. Door de trainingsinstances telkens vrij willekeurig af te wisselen hopen we terecht te komen in een punt dat voor alle instances tezamen een optimaal compromis is. We spreken vanwege deze willekeurige afwisseling ook wel van *stochastic* gradient descent.

Opgave 22. *

Beschrijf in je eigen woorden het verschil tussen de notaties $\frac{\Delta y}{\Delta x}$, $\frac{\mathrm{d}y}{\mathrm{d}x}$, $\frac{\partial y}{\partial x}$, en $\boldsymbol{\nabla}_{\boldsymbol{x}} y$.

Opgave 23. *

Als je k-means clustering uitvoert is het verstandig om het algoritme vele malen te herhalen waarbij je elke keer de cluster centroïden willekeurig initialiseert. Leg uit waarom dit nodig is, en gebruik in je antwoord de termen "lokaal minimum" en "globaal minimum". Hint: zoek de betekenis van de parameter NSTART op in de implementatie van k-means in R, of die van N_INIT in die van python's module *scikit-learn*.

Opgave 24. *
Geef een synoniem voor het woord "stochastisch"; zoek deze term indien nodig op in een woordenboek.

Opgave 25. **
Laat zien dat de helling $\frac{\partial l}{\partial w_i}$ benaderd kan worden door $\frac{\Delta l}{\Delta w_i} = \frac{f(w_i+\Delta w_i)-f(w_i-\Delta w_i)}{2\Delta w_i}$ als je deze baseert op twee punten, eentje een stap Δw_i naar links en eentje een stap Δw_i naar rechts.

Opgave 26. **
Bereken met je rekenmachine (of computer) de afgeleide van de functie $y = \ln(x)$ ter plekke van het punt $x = 2$. Gebruik een numerieke benadering waarbij je gebruik maakt van een stapgrootte $\Delta x = 0.01$ om Δy te bepalen. Maakt het uit of je $\Delta y = f(x + \Delta x) - f(x)$ gebruikt of $\Delta y = \frac{f(x+\Delta x)-f(x-\Delta x)}{2}$? Het exacte antwoord dat je kan vinden door analytisch te differentiëren luidt $\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{2}$; komen je numerieke resultaten hiermee overeen?

Opgave 27. **
Gebruik de formule voor $\frac{\Delta y}{\Delta x}$ om de numerieke afgeleide te bepalen van de functie $y = f(x) = x^2$. Komt het resultaat overeen met wat je zou krijgen met analytische wiskundige differentiatie?

2.3. Optimalisatie

Laten we de voorgaande resultaten combineren om ons in staat te stellen de gewichten w_i te variëren op een dusdanige manier dat l geminimaliseerd wordt. We gaan uit van de gradient descent updateregel $w_i \leftarrow w_i - \alpha \cdot \frac{\partial l}{\partial w_i}$. Het probleem is dat we geen eenvoudige manier hebben om de helling van de lossfunctie in termen van de gewichten w_i te bepalen. We moeten immers drie verschillende transformaties achter elkaar toepassen om van de gewichten w_i uiteindelijk tot de loss l te komen. Echter, met de *kettingregel* kunnen we de ene afgeleide in deze formule omzetten in drie afzonderlijke afgeleiden die overeenkomen met elk van de drie transformaties:

$$w_i \leftarrow w_i - \alpha \cdot \frac{\partial a}{\partial w_i} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial l}{\partial \hat{y}}$$

Ga voor jezelf na dat dit precies hetzelfde is als de eerdere updateregel omdat de partiële differentialen $\partial\hat{y}$ en ∂a boven en onder de deelstreep telkens tegen elkaar wegvallen.

We bekijken de drie partiële afgeleiden nu elk afzonderlijk.

1. De uitdrukking $\frac{\partial a}{\partial w_i}$ geeft aan hoe groot de verandering in de pre-activatiewaarde a is als je het gewicht w_i een klein beetje verandert. Dit zegt iets over de eerste van de drie transformaties $a = \sum_i w_i \cdot x_i$. Het is duidelijk dat als we w_i bijvoorbeeld met 1 laten toenemen, dat dan de uitkomst a met een hoeveelheid x_i toeneemt. Immers,

w_i wordt vermenigvuldigd met x_i. Kortom, voor $\Delta w_i = 1$ is de bijbehorende $\Delta a = x_i$, zodat $\frac{\Delta a}{\Delta w_i} = \frac{x_i}{1} = x_i$. Hieruit kunnen we concluderen:

$$\frac{\partial a}{\partial w_i} = x_i$$

2. De uitdrukking $\frac{\partial \hat{y}}{\partial a}$ geeft aan hoe groot de verandering in de post-activatiewaarde $\hat{y}$ is als je de pre-activatiewaarde a een klein beetje verandert. Dit zegt iets over de tweede van de drie transformaties $\hat{y} = \varphi(a)$. Deze afgeleide zegt iets over de helling van de activatiefunctie. Voor veel standaardfuncties kun je de helling bepalen door analytisch te *differentiëren*; deze uitkomsten zijn vaak ook op te zoeken in tabellenboeken. In het algemeen kun je deze afgeleide ook numeriek bepalen, bijvoorbeeld middels de eerder genoemde benadering:

$$\frac{\partial \hat{y}}{\partial a} \approx \frac{\varphi(a + \Delta a) - \varphi(a - \Delta a)}{2\Delta a}$$

voor een voldoende kleine waarde van Δa.

3. De uitdrukking $\frac{\partial l}{\partial \hat{y}}$ tenslotte geeft aan hoe groot de verandering in de loss l is als je de post-activatiewaarde $\hat{y}$ een klein beetje verandert. Dit zegt iets over de derde van de drie transformaties $l = \mathcal{L}(\hat{y}; y)$. Deze afgeleide zegt iets over de helling van de lossfunctie. Ook deze is vaak op te zoeken, of je kunt weer een numerieke benadering gebruiken:

$$\frac{\partial l}{\partial \hat{y}} \approx \frac{\mathcal{L}(\hat{y} + \Delta\hat{y}; y) - \mathcal{L}(\hat{y} - \Delta\hat{y}; y)}{2\Delta\hat{y}}$$

voor een voldoende kleine waarde van $\Delta\hat{y}$.

Samengevat, voor elk model dat kan worden geschreven in termen van een lineaire combinatie, een activatiefunctie, en een lossfunctie hebben we hierboven de updateregel $w_i \leftarrow w_i - \alpha \cdot \frac{\partial a}{\partial w_i} \frac{\partial \hat{y}}{\partial a} \frac{\partial l}{\partial \hat{y}}$ afgeleid. Weliswaar komen hier een drietal afgeleiden in voor, maar die kunnen allemaal uitgerekend worden met de formules hierboven (hetzij analytisch, hetzij numeriek).

De modellen die we in het vorige hoofdstuk bekeken hebben, te weten het perceptron en het lineaire regressiemodel, zijn beide voorbeelden van modellen die zo uitgewerkt kunnen worden. Daarnaast zijn er echter nog vele andere modellen mogelijk, waaronder logistische regressie.

Opgave 28. *

Bepaal analytisch of numeriek de afgeleide $\frac{\partial \hat{y}}{\partial a}$ voor de identiteitsfunctie $\varphi(a) = a$.

Opgave 29. *

Omschrijf in woorden de vorm van de afgeleide $\frac{\partial \hat{y}}{\partial a}$ wanneer je de signum-functie als activatiefunctie neemt, dat wil zeggen $\varphi(a) = \operatorname{sgn}(a)$. Deze afgeleide is nauw gerelateerd aan de zogenaamde *Dirac-deltafunctie*.

Opgave 30. *

Leg uit dat de afgeleide van de pre-activatiewaarde a naar de bias b gelijk is aan $\frac{\partial a}{\partial b} = 1$. Je kunt dit bijvoorbeeld doen aan de hand van de gegeven formule $\frac{\partial a}{\partial w_i} = x_i$.

Opgave 31. **

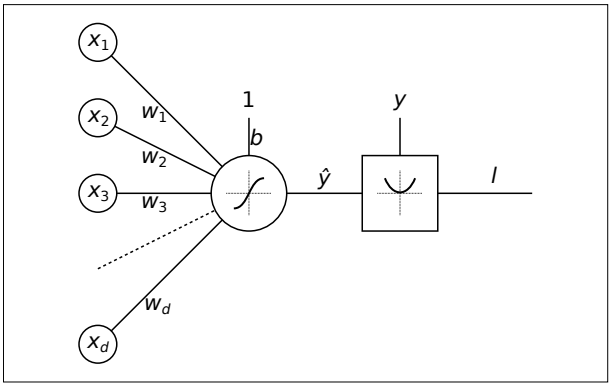
Laat zien dat voor de kwadratische lossfunctie $\mathcal{L}(\hat{y}; y) = (\hat{y} - y)^2$ de partiële afgeleide gelijk is aan $\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y)$. Gebruik hiervoor de numerieke formule $\frac{\partial \mathcal{L}}{\partial \hat{y}} \approx \frac{\mathcal{L}(\hat{y}+\Delta\hat{y};y) - \mathcal{L}(\hat{y}-\Delta\hat{y};y)}{2\Delta\hat{y}}$ en vereenvoudig het resultaat.

Opgave 32. **

Motiveer dat voor de absolute lossfunctie $\mathcal{L}(\hat{y}; y) = |\hat{y} - y|$ de partiële afgeleide geschreven kan worden als $\frac{\partial \mathcal{L}}{\partial \hat{y}} = \operatorname{sgn}(\hat{y} - y)$.

2.4. Het generieke model

Het totale model van een neuron, inclusief lossfunctie, is hieronder schematisch getoond in een diagram. Er is nu een extra component bijgekomen, gesymboliseerd met een vierkant dat een kwadratische functie bevat, die de voorspelde waarde $\hat{y}$ binnenkrijgt en in combinatie met het ware klasselabel y omzet in de loss l. Stochastic gradient descent wordt nu toegepast op dit diagram in zijn geheel: alle beschikbare vrije parameters (dat wil zeggen de bias b en gewichten $w_1, w_2, \ldots, w_d$) worden in kleine stapjes bijgesteld om het optimalisatiecriterium l aan de rechterkant zo laag mogelijk te krijgen.



Zoals we eerder al zagen wordt lineaire regressie gekarakteriseerd door een activatiefunctie die gelijk is aan de identiteitsfunctie $\varphi(a) = a$. Deze functie doet eigenlijk niets: de post-activatiewaarde is simpelweg gelijk aan de pre-activatiewaarde. Verder wordt bij regressie gebruik gemaakt van de eerder genoemde kwadratische lossfunctie $\mathcal{L}(\hat{y}; y) = (\hat{y} - y)^2$.

De afgeleide van de kwadratische lossfunctie is $\frac{\partial l}{\partial \hat{y}} = 2(\hat{y} - y)$; de afgeleide van de identiteitsfunctie is $\frac{\partial \hat{y}}{\partial a} = 1$; en de afgeleide van de pre-activatiewaarde naar de gewichten is immer $\frac{\partial a}{\partial w_i} = x_i$. Combineren we dit met de stochastic gradient descent updateregel

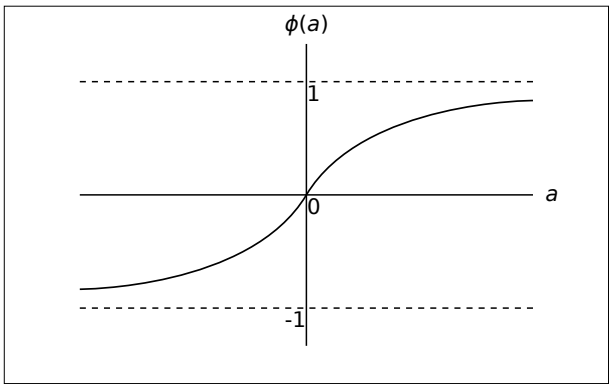
$w_i \leftarrow w_i - \alpha \cdot \frac{\partial a}{\partial w_i} \frac{\partial \hat{y}}{\partial a} \frac{\partial l}{\partial \hat{y}}$ zoals hierboven uiteengezet, dan vinden we:

$$\begin{cases} b \leftarrow b - 2\alpha\,(\hat{y} - y) \\ w_i \leftarrow w_i - 2\alpha\,(\hat{y} - y)\,x_i \end{cases}$$

De formule voor b is verkregen door naar de uitdrukking voor w_0 te kijken en $x_0 = 1$ te stellen. Dit is bijna exact de updateregel voor lineaire regressie die we in het vorige hoofdstuk al hadden gevonden! Het enige verschil is de factor 2; soms wordt daarom ook wel een loss-functie $\mathcal{L}\,(\hat{y}; y) = \frac{1}{2}\,(\hat{y} - y)^2$ genomen met een extra factor $\frac{1}{2}$, maar door een twee keer zo kleine learning rate α te kiezen kun je het effect van de factor 2 ook volkomen opheffen. Deze updateregel komt dus op hetzelfde neer. Het nieuwe inzicht is dat onze updateregel uit het vorige hoofdstuk de lossfunctie minimaliseert over alle modelparameters middels stochastic gradient descent.

Voorheen zagen we dat het perceptron gebruikt kan worden voor classificatieproblemen. Het perceptron had als kenmerk dat de signum-functie als activatiefunctie wordt gebruikt. Helaas is deze functie niet zo geschikt voor onze huidige aanpak omdat diens helling zich niet netjes gedraagt. Rond $a = 0$ maakt $\operatorname{sgn}(a)$ een sprong. De functie is hier *discontinu* en daardoor niet differentieerbaar: de helling is eventjes oneindig sterk. Ook voor positieve en negatieve a gaat het echter mis. Elders heeft de signum-functie immers overal een helling gelijk aan nul, zodat $\frac{\partial \hat{y}}{\partial a} = 0$. Dit zou betekenen dat updateregel zegt dat je de gewichten moet bijwerken met een waarde nul. Hierdoor verandert er niets aan de parameters, en wordt het model dus helemaal niet geoptimaliseerd.

Met de resultaten hierboven is het niet zo heel moeilijk om deze problemen te omzeilen. Door de signum-functie af te vlakken tot een gladde S-vormige *sigmoïde* functie kan zowel het probleem van de oneindige helling rondom nul als dat van de helling nul elders worden opgelost. We verkrijgen dan een vorm van *logistische* regressie. De activatiefunctie ziet er dan ongeveer uit zoals hieronder.



Er zijn diverse activatiefuncties $\varphi\,(a)$ te verzinnen die deze vorm hebben. Een aantal bekende voorbeelden zijn:

- de *tangens-hyperbolicus-functie* $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$;
- de *softsign-functie* $\operatorname{softsign}(a) = \frac{a}{1 + |a|}$;

- de *inverse square root unit* functie $\text{isru}(a) = \frac{a}{\sqrt{1+a^2}}$.

Voor uitkomsten met een bereik tussen 0 en 1 kan de *logistische functie* $\sigma(a) = \frac{1}{1+e^{-a}}$ worden gekozen. Hier is de naam logistische regressie aan gerelateerd.

De tangens-hyperbolicus heeft de prettige eigenschap dat diens afgeleide eenvoudig kan worden geformuleerd in termen van zichzelf. Er geldt namelijk dat $\frac{\mathrm{d}\tanh(a)}{\mathrm{d}a} = 1 - \tanh^2(a)$. Voor een model dat een activatiefunctie heeft gelijk aan $\hat{y} = \tanh(a)$ geldt dan dat $\frac{\partial \hat{y}}{\partial a} = 1 - \hat{y}^2$. De updateregel $w_i \leftarrow w_i - \alpha \cdot \frac{\partial a}{\partial w_i} \frac{\partial \hat{y}}{\partial a} \frac{\partial l}{\partial \hat{y}}$ voor stochastic gradient descent leidt dan uiteindelijk tot:

$$\begin{cases} b \leftarrow b - \alpha(\hat{y} - y)\left(1 - \hat{y}^2\right) \\ w_i \leftarrow w_i - \alpha(\hat{y} - y)\left(1 - \hat{y}^2\right) x_i \end{cases}$$

Alle factoren in deze formules zijn bekend of worden reeds berekend tijdens het doen van een voorspelling.

Vergeleken met lineaire regressie is er een extra factor bijgekomen gelijk aan $\left(1 - \hat{y}^2\right)$. Deze gedraagt zich als een wegingsfactor. Voor instances waarvoor $\hat{y} = 0$, dat wil zeggen instances die op de geschatte scheidingslijn tussen de klassen liggen, is deze wegingsfactor gelijk aan 1. Deze instances worden dus als het ware voor 100% meegewogen bij het bepalen of de scheidingslijn moet worden verschoven. Voor instances waarvoor $\hat{y} = \pm 1$, dat wil zeggen instances die ver van scheidingslijn vandaan liggen en daarom reeds met grote zekerheid als de ene of de andere klasse worden gelabeld, nadert de wegingsfactor naar 0. Deze instances worden dus niet of nauwelijks meegewogen bij het aanpassen van de modelparameters. Het idee hierachter is dat instances die ver van de scheidingslijn af liggen na een kleine aanpassing toch nog immer ver aan dezelfde kant van de lijn blijven liggen, dus voor hen heeft het niet zoveel zin om de lijn te verschuiven. Voor instances die vlak bij de scheidingslijn liggen is het echter wel degelijk zeer relevant of die lijn enigszins verschoven wordt, dus deze worden het zwaarst meegeteld bij het aanpassen van de modelparameters. Dit gedrag is anders dan bij lineaire regressie, waar alle instances altijd even zwaar meewegen.

Opgave 33. *

Het perceptron en logistische regressie zijn beide modellen om classificatie uit te voeren. Beschrijf een aantal verschillen tussen deze twee methoden.

Opgave 34. *

Leg uit dat een helling exact gelijk aan nul voor de activatiefunctie, dat wil zeggen $\frac{\partial \hat{y}}{\partial a} = 0$, ertoe leidt dat de updateregel de gewichten niet langer bijwerkt. Is dit een probleem?

Opgave 35. **

Voor een *even* functie geldt dat $f(-x) = f(x)$, terwijl voor een *oneven* functie $f(-x) = -f(x)$. Toon aan dat de activatiefuncties $\tanh(a)$, $\text{softsign}(a)$, en $\text{isru}(a)$ alledrie oneven functies zijn.

Opgave 36. **

Laat zien dat de logistische functie $\sigma(a) = \frac{1}{1+e^{-a}}$ gerelateerd is aan de tanh-functie volgens $\tanh(a) = 2 \cdot \sigma(2a) - 1$, of omgekeerd $\sigma(a) = \frac{1+\tanh(a/2)}{2}$. Schets de beide functies samen in één grafiek.

Opgave 37. ***

De afgeleiden van de softsign- en isru-functies kunnen eveneens geschreven worden in termen van zichzelf, namelijk $\frac{\mathrm{dsoftsign}(a)}{\mathrm{d}a} = (1 - |\mathrm{softsign}(a)|)^2$ en $\frac{\mathrm{disru}(a)}{\mathrm{d}a} = \left(1 - \mathrm{isru}^2(a)\right)^{\frac{3}{2}}$. Leid hiermee de updateregels af voor logistische regressiemodellen waarin deze functies worden gebruikt als activatiefunctie, samen met een kwadratische lossfunctie. Leg uit dat deze modellen verschillen van lineaire regressie door een hoge weging van instances die dichtbij de geschatte scheidingslijn tussen de twee klassen liggen en een lage weging van instances die hier ver vandaan liggen.

Opgave 38. ***

Zoek zelf informatie op over de sigmoïde Gudermannfunctie $\mathrm{gd}(a)$ en de formule voor diens afgeleide. Kun je de afgeleide van de Gudermannfunctie schrijven in termen van de Gudermannfunctie zelf (vergelijkbaar met hoe dat in de vorige opgave voor de softsign en isru functies werd gedaan)?

Deel II.

Deep learning

3. Neurale netwerken

Vooruitblik

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

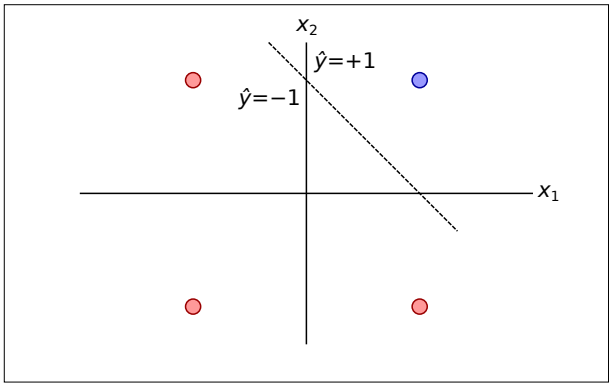
In dit geval is er maar één attribuut, dus vereenvoudigt het perceptron model tot $\hat{y} = \text{sgn}\,(b + w_1 \cdot x_1)$. Kunnen we een bias b en een gewicht w_1 bedenken waarmee de bovenstaande waarden worden verkregen? Het is niet zo heel moeilijk om in te zien dat dit kan. Door bijvoorbeeld $b = 0$ te kiezen en $w_1 = -1$ krijgen we het model $\hat{y} = \text{sgn}\,(-x_1)$ 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 $\hat{y} = \text{sgn}\,(b + w_1 \cdot x_1 + w_2 \cdot x_2)$. 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 $\hat{y} = \text{sgn}\,(x_1 + x_2 + 1)$ en voor de AND-operatie geldt $\hat{y} = \text{sgn}\,(x_1 + x_2 - 1)$. Dit komt in beide gevallen overeen met gewichten $\boldsymbol{w} = [1, 1]$ en bias $b = +1$ (voor OR) of $b = -1$ (voor AND).



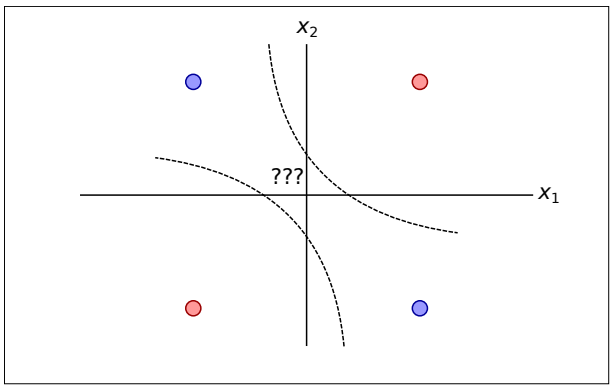
Hierboven wordt het model voor de AND-operatie geïllustreerd. De uitkomst $y = +1$ komt alleen voor bij $x_1 = x_2 = +1$, in blauw getoond in de rechterbovenhoek. De andere

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 39. *

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 40. **

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

x_1	x_2	$y = x_1 \to x_2$
-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. **

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 42. **

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 43. ***

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 44. ***

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

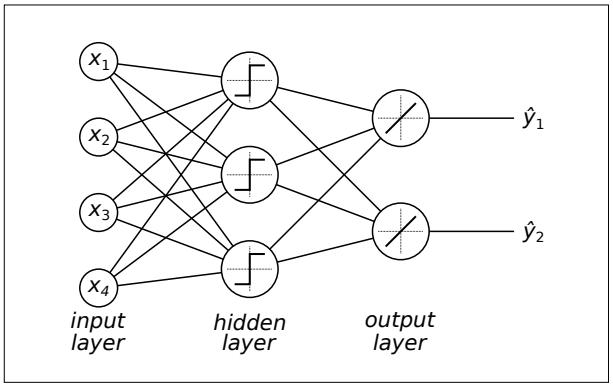
In de vorige paragraaf kwamen we een belangrijke beperking tegen van het perceptron, in de vorm van het XOR-probleem. We halen nu echter weer inspiratie uit de rekeneenheden van computers. Die bestaan immers ook niet simpelweg uit één enkele logische schakeling, maar zijn in staat om krachtige berekeningen 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.

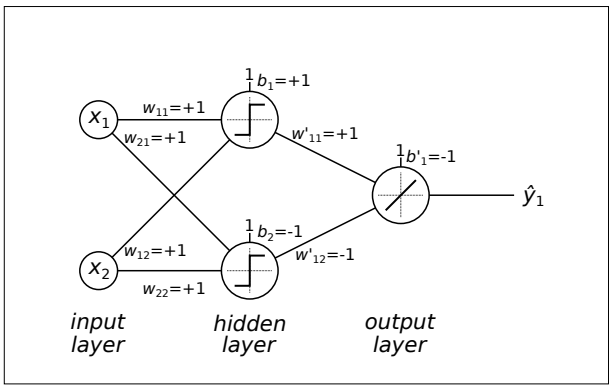
Een dergelijke one-hot codering kan ook gebruikt worden wanneer je een nominale variabele met een beperkt aantal mogelijkheden als attribuut wil gebruiken in de input layer. Je zet deze dan om in een verzameling *indicator-* of *dummy-variabelen* waarbij je alleen de ene waarde die overeenkomt met de 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 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 identi-

teitsfunctie 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 45. *

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 46. **

In de opgaven bij de vorige paragraaf werd gevraagd om de IMP-operator $x_1 \to x_2$ 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 47. **

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 48. **
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 49. ***
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 50. ***
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 51. ***
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.

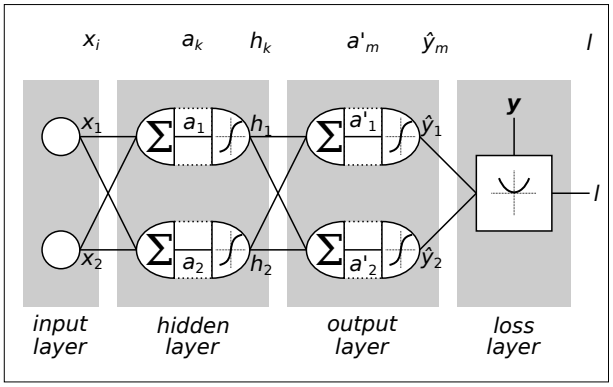
3.3. Back-propagation

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 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 52. *

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 53. **

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 54. **

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 theoretisch precies de beste aanpak is in bepaalde gevallen is eigenlijk niet goed bekend.

Opgave 55. *

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 56. *

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 57. **

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 58. ***

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 59. ***

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?

4. Multinomiale classificatie

Vooruitblik

In dit hoofdstuk zetten we de uitvoer van het multi-layer perceptron ten behoeve van classificatie om in probabilistische waarden die de kans op elk klasselabel aangeven. We leiden de feed-forward en back-propagation formules af voor een softmax-layer en combineren deze met de cross-entropy lossfunctie om de uitkomst robuuster te maken tegen saturatie. Het resultaat blijkt een multinomiale generalisatie van logistische regressie op te leveren, waarmee we eveneens een formulering van het oorspronkelijke perceptron in termen van activatie- en lossfuncties kunnen afleiden.

4.1. Softmax

In het vorige hoofdstuk zijn we gekomen tot een multi-layer perceptron dat in staat is om classificatie danwel regressie uit te voeren door middel van meerdere neurale lagen waarin neuronen parallel aan elkaar functioneren om uiteindelijk te komen tot een voorspelling $\hat{\boldsymbol{y}}$. In het geval van classificatie kan de uitvoer gezien worden als een vector van waarschijnlijkheden voor de verschillende mogelijke klasselabels. In dat geval wordt de gewenste uitkomst $\boldsymbol{y}$ gegeven door een one-hot encoding. Immers, de kans op het juiste klasselabel zou idealiter gelijk moeten zijn aan 100%, oftewel 1, en de kans op alle andere labels gelijk aan 0%, oftewel 0.

Tot dusverre hebben we geen bijzondere eisen gesteld aan de uitkomst van het model, behalve dan dat deze zo goed mogelijk met de gewenste uitkomsten dient overeen te komen. Wanneer de voorspellingen geïnterpreteerd dienen te worden als kansen kunnen echter een aantal randvoorwaarden worden opgelegd waaraan de voorspelling van de output layer sowieso móet voldoen. Deze omvat de volgende kenmerken:

- Omdat kansen nooit een negatieve waarde kunnen aannemen dienen de voorspellingen $\hat{y}_n$ non-negatief te zijn; dat wil zeggen, positief of eventueel gelijk aan nul.

$$\hat{y}_n \geq 0$$

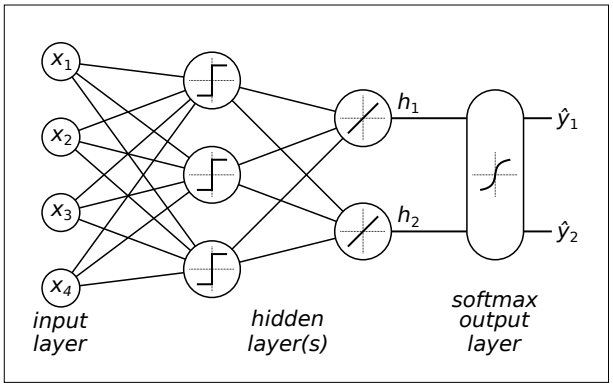
- Omdat de mogelijke klasselabels elkaar uitsluiten maar er wel altijd één label juist is, dienen de voorspellingen voor alle mogelijke klasselabels $\hat{y}_n$ tezamen op te tellen tot 100%, oftewel 1.

$$\sum_n \hat{y}_n = 1$$

- Omdat de uitkomst van de output neuronen rechtstreeks verband dient te houden met de voorspelde kans, eisen we dat de kans *monotoon* toeneemt met de invoerwaarden h_n van de output neuronen; dat wil zeggen, hoe hoger de invoer van een neuron in de output layer, hoe hoger de corresponderende kans op het bijbehorende klasselabel.

$$\frac{\partial \hat{y}_m}{\partial h_m} > 0$$

Om te garanderen dat aan deze eisen wordt voldaan kan de output layer aan het einde van het neurale netwerk worden aangepast. Om redenen die nog nader zullen worden toegelicht noemen we deze een *softmax* layer. In de figuur hieronder wordt deze getoond.



Duiden we de invoer naar deze laatste laag aan met de vector $\boldsymbol{h}$, dan transformeert de softmax layer deze naar voorspellingen $\hat{\boldsymbol{y}}$ die voldoen aan de eerder genoemde drie eigenschappen. Merk op dat de softmax layer niet bestaat uit meerdere functies die parallel en onafhankelijk op één invoerwaarde elk worden toegepast. Dit komt omdat elke uitvoer $\hat{y}_n$ af zal hangen van alle invoerwaarden h_m tezamen. Om dit te benadrukken beschouwen we de softmax layer als een *vectorfunctie* die wordt toegepast op een complete vector tegelijkertijd ($\boldsymbol{h}$) en ook weer een nieuwe vector ($\hat{\boldsymbol{y}}$) oplevert.

Kansen moeten zoals gezegd non-negatief zijn. Positieve uitkomsten kunnen worden verkregen door diverse soorten transformaties toe te passen. We zijn dat eerder tegengekomen bij de lossfunctie, waarbij bijvoorbeeld gekozen werd voor het kwadraat. In dit geval zouden deze mogelijkheden echter niet tot een monotone functie leiden omdat een parabool ook een neergaande flank heeft, dus dit is in strijd met het derde genoemde kenmerk hierboven. Een functie die wel monotoon toeneemt en altijd positieve uitkomsten oplevert is bijvoorbeeld de exponentiële functie $f(x) = e^x$. Als we voor elke invoer h_n een uitvoer berekenen volgens $\hat{y}_n = e^{h_n}$ is echter nog niet automatisch voldaan aan de eis dat de som van alle kansen opgeteld 1 moet opleveren. De exponentiële functie kan immers willekeurig grote uitkomsten opleveren die het gewenste totaal van 1 ver overtreffen. Dit kan worden gecorrigeerd door de uitkomsten te *normaliseren*. Je deelt in dit geval elke individuele uitkomst door het totaal, om te komen tot onderstaande formule voor de *softmax-functie*.

$$\hat{y}_n = \frac{e^{h_n}}{\sum_j e^{h_j}}$$

De naam van deze functie is eigenlijk ietwat ongelukkig gekozen. De welbekende *max*-functie retourneert de hoogste waarde die voorkomt of mogelijk is; de *argmax*-functie geeft de index of het argument waarvoor die maximale waarde wordt bereikt. Voor de vector $\boldsymbol{v} = [-1, 4, 2, 0]$ is bijvoorbeeld $\max(\boldsymbol{v}) = 4$, maar $\operatorname{argmax}(\boldsymbol{v}) = 2$ omdat het tweede element van de vector de hoogste waarde heeft. Wanneer indices vanaf nul geteld worden, zoals gebruikelijk in de informatica, zou gesteld kunnen worden dat $\operatorname{argmax}(\boldsymbol{v}) = 1$, of wanneer een one-hot encoding wordt gebruikt is $\operatorname{argmax}(\boldsymbol{v}) = [0, 1, 0, 0]$, waarbij de positie van het maximum als het ware met booleans wordt aangegeven. Deze laatste definitie van $\operatorname{argmax}(\boldsymbol{v})$ is geschikt om als laatste laag van het netwerk te fungeren, want deze produceert precies het type uitvoer dat we wensen. Maar deze heeft als nadeel dat deze functie discontinu is. Dat wil zeggen, een willekeurig kleine verandering in de invoer kan tot een grote verandering in de uitvoer leiden. Dergelijke discontinue functies zijn niet gewenst omdat deze een helling opleveren die enerzijds oneindig groot kan zijn, en anderzijds vaak gelijk is aan nul. Vergelijk dit met het gedrag van de signum-functie. Om dit probleem op te lossen kan de functie worden afgevlakt. De signum-functie gaf daarbij aanleiding tot bijvoorbeeld de softsign-functie; de argmax-functie leidt soortgelijk tot de softmax-functie. Deze laatste had dus eigenlijk beter de "softargmax-functie" kunnen heten, maar helaas is de naam anders ingeburgerd geraakt.

Opgave 60. *

Gegeven een vector $\boldsymbol{x} = [-1, 0, +1]$. Bereken $\operatorname{softmax}(\boldsymbol{x})$. Controleer dat de uitkomst aan de drie hierboven genoemde vereisten voldoet.

Opgave 61. *

Gegeven een vector $\boldsymbol{x} = \left[-10^6, \frac{1+\sqrt{5}}{2}, \ln(2), \pi, e\right]$. Bepaal $\max(\boldsymbol{x})$, $\operatorname{argmax}(\boldsymbol{x})$, en $\operatorname{softmax}(\boldsymbol{x})$.

Opgave 62. **

In het lijstje met vereisten voor kanswaarden wordt wel genoemd dat kansen niet negatief mogen zijn, te weten $\hat{y}_n \geq 0$, maar er wordt niet genoemd dat kansen niet groter dan 100% mogen zijn, te weten $\hat{y}_n \leq 1$. Leg uit dat dit laatste kenmerk toch reeds vanzelf uit de drie genoemde vereisten voortvloeit.

Opgave 63. **

Laat zien dat de uitkomst van de softmax-functie niet verandert als je alle invoerwaarden h_m met dezelfde waarde verhoogt of verlaagt. Dat wil zeggen, gegeven een vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_m]$ en een aangepaste vector $\boldsymbol{x'} = [x_1 + c, x_2 + c, \ldots, x_m + c]$ met een willekeurige offset c, dan is $\operatorname{softmax}(\boldsymbol{x}) = \operatorname{softmax}(\boldsymbol{x'})$.

Opgave 64. ***

Laat zien dat voor alle vectoren $\boldsymbol{v}$ geldt dat $\max(\boldsymbol{v}) = \boldsymbol{v} \cdot \operatorname{argmax}(\boldsymbol{v})$, waarbij de argmax-functie een vectoriële uitvoer geeft volgens een one-hot encoding en de punt staat voor het inproduct van twee vectoren.

Opgave 65. ***

Leg uit hoezo de argmax-functie met een one-hot encoding als resultaat noch continu noch differentieerbaar is.

4.2. Optimalisatie

Zojuist is de formule afgeleid waarmee de uitvoer van de softmax layer kan worden afgeleid uit diens invoer. Dit is nodig tijdens de feed-forward fase tijdens de evaluatie van een neuraal netwerk. Tijdens de optimalisatie van een dergelijk netwerk met behulp van back-propagation dient echter ook de gradiënt van de loss naar de invoer $\boldsymbol{\nabla_h} l$ te kunnen worden bepaald als de gradiënt naar de uitvoer $\boldsymbol{\nabla_{\hat{y}}} l$ bekend is. Met andere woorden, als je weet hoe de loss verandert als de uitvoer $\hat{\boldsymbol{y}}$ van de softmax layer varieert, willen we nu nagaan hoe de loss afhangt van de invoer $\boldsymbol{h}$ van de softmax layer. Dit kan in principe numeriek gedaan worden door de waarde van een invoer h_m een klein beetje te veranderen en te bezien wat de invloed op alle uitvoeren $\hat{y}_n$ is. Er kan dan gesteld worden dat

$$\frac{\partial l}{\partial h_m} = \sum_n \frac{\partial l}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial h_m}$$

Hierbij mag $\frac{\partial l}{\partial \hat{y}_n}$ recursief bekend worden verondersteld uit eerdere stappen van de back-propagation fase. De afgeleide $\frac{\partial \hat{y}_n}{\partial h_m}$ die aangeeft hoe sterk uitvoer nummer n van de softmax layer verandert als invoer nummer m een beetje verandert zou nu numeriek kunnen worden bepaald. Dit is vergelijkbaar met hoe de gradiënt van de loss door een neurale laag wordt teruggepropageerd. Echter, in dit geval is de functie altijd gelijk aan de softmax-functie, terwijl we bij neurale lagen met diverse soorten activatiefuncties rekening moesten houden. Daarnaast blijkt de softmax-functie een elegante afgeleide te hebben. Daarom kiezen we er hier voor om deze afgeleide eenmalig analytisch af te leiden door middel van differentiëren, in plaats van dit telkens numeriek door de computer te moeten laten doen.

We maken gebruik van de regel omtrent het differentiëren van quotiënten die zegt dat

$$\frac{\partial}{\partial x} \frac{f}{g} = \frac{g \cdot \frac{\partial f}{\partial x} - f \cdot \frac{\partial g}{\partial x}}{g^2}$$

Passen we dit toe op de softmax-functie $\hat{y}_n = \frac{e^{h_n}}{\sum_j e^{h_j}}$ dan dienen we eerst de afgeleiden van de teller en de noemer afzonderlijk te kennen.

- Voor de exponent in de teller geldt dat $\frac{\partial}{\partial h_m} e^{h_n}$ gelijk is aan nul indien $n \neq m$ omdat in dat geval de exponent helemaal niet afhangt van h_m, en gelijk is aan e^{h_n} als $m = n$ omdat de afgeleide van de exponentiële functie gelijk is aan zichzelf. Dit kan korter worden genoteerd door gebruik te maken van de *Kronecker-delta* notatie

  $$\delta_{mn} = \begin{cases} 0 \text{ voor } m \neq n \\ 1 \text{ voor } m = n \end{cases}$$

  wat dan oplevert $\frac{\partial}{\partial h_m} e^{h_n} = \delta_{mn} \cdot e^{h_n}$.

- Voor de afgeleide van de som in de noemer geldt dat $\frac{\partial}{\partial h_m} \sum_j e^{h_j} = e^{h_m}$ omdat er precies één term e^{h_m} in de som zit die afhangt van h_m: diens afgeleide is e^{h_m}, en de afgeleide van elke andere term is weer nul.

Voegen we dit allemaal samen dan vinden we

$$\begin{aligned}
\frac{\partial \hat{y}_n}{\partial h_m} &= \frac{\left(\sum_j e^{h_j}\right) \cdot \left(\frac{\partial}{\partial h_m} e^{h_n}\right) - \left(e^{h_n}\right) \cdot \left(\frac{\partial}{\partial h_m} \sum_j e^{h_j}\right)}{\left(\sum_j e^{h_j}\right)^2} \\
&= \frac{\left(\sum_j e^{h_j}\right) \cdot \left(\delta_{mn} \cdot e^{h_n}\right) - \left(e^{h_n}\right) \cdot \left(e^{h_m}\right)}{\left(\sum_j e^{h_j}\right)^2} \\
&= \frac{\left(\sum_j e^{h_j}\right) \cdot \left(\delta_{mn} \cdot e^{h_n}\right)}{\left(\sum_j e^{h_j}\right) \cdot \left(\sum_j e^{h_j}\right)} - \frac{\left(e^{h_n}\right) \cdot \left(e^{h_m}\right)}{\left(\sum_j e^{h_j}\right) \cdot \left(\sum_j e^{h_j}\right)} \\
&= \delta_{mn} \cdot \frac{e^{h_n}}{\sum_j e^{h_j}} - \frac{e^{h_n}}{\sum_j e^{h_j}} \cdot \frac{e^{h_m}}{\sum_j e^{h_j}} \\
&= \frac{e^{h_n}}{\sum_j e^{h_j}} \left(\delta_{mn} - \frac{e^{h_m}}{\sum_j e^{h_j}}\right)
\end{aligned}$$

Dit kan worden vereenvoudigd door te herkennen dat hierin twee maal de softmax-formule $\hat{y}_n = \frac{e^{h_n}}{\sum_j e^{h_j}}$ voorkomt, voor twee verschillende neuronen n en m. We krijgen dan

$$\frac{\partial \hat{y}_n}{\partial h_m} = \hat{y}_n \cdot (\delta_{mn} - \hat{y}_m)$$

De resulterende formule beschrijft de verandering in elke voorspelling $\hat{y}_n$ als één van de invoeren h_m verandert. Hieruit blijkt dat elke uitvoer afhangt van elke invoer. Er kan worden aangetoond dat de voorspelde kans $\hat{y}_n$ op een zeker klasselabel n geleidelijk toeneemt als de daarmee overeenkomende invoer h_n toeneemt, maar dat deze kans afneemt als de andere invoeren h_m (met $m \neq n$) toenemen.

Opgave 66. *
Laat zien dat de afgeleide van de softmax-functie geschreven kan worden als

$$\frac{\partial \hat{y}_n}{\partial h_m} = \begin{cases} \hat{y}_n \cdot (1 - \hat{y}_m) & \text{voor } m = n \\ -\hat{y}_n \cdot \hat{y}_m & \text{voor } m \neq n \end{cases}$$

Opgave 67. **
Leg uit dat afgeleide van de softmax-functie ook geschreven kan worden als

$$\frac{\partial \hat{y}_n}{\partial h_m} = \hat{y}_m \cdot (\delta_{mn} - \hat{y}_n)$$

Opgave 68. **
Laat zien dat voor de softmax-functie geldt dat $\frac{\partial \hat{y}_n}{\partial h_m} > 0$ voor $m = n$ en $\frac{\partial \hat{y}_n}{\partial h_m} < 0$ voor $m \neq n$. Leg uit dat dit aantoont dat aan de derde voorwaarde omtrent de monotoniciteit van de kansverdelingsfunctie wordt voldaan.

Opgave 69. **

Gegeven een neuraal netwerk dat gebruikt wordt om data te classificeren met N verschillende klasselabels. Stel dat alle invoerwaarden naar de softmax laag identiek zijn, dat wil zeggen alle h_m zijn aan elkaar gelijk. Toon aan dat dan $\hat{y}_n = \frac{1}{N}$ voor alle klassen. Hoe groot is in dit geval $\frac{\partial \hat{y}_n}{\partial h_m}$ voor alle combinaties van n en m?

Opgave 70. ***

Toon aan dat voor de softmax-functie $\sum_n \frac{\partial \hat{y}_n}{\partial h_m} = 0$. Wat betekent dit in woorden? Hint: je kan schrijven $\sum_n \frac{\partial \hat{y}_n}{\partial h_m} = \frac{\partial}{\partial h_m}\left(\sum_n \hat{y}_n\right)$; waarvoor staat de grootheid $\sum_n \hat{y}_n$ en wat moet diens waarde dus wel zijn en blijven, ongeacht de waarde van h_m?

4.3. Dichotome classificatie

Als er maar twee mogelijke klassen zijn dan vereenvoudigt de softmax-functie tot

$$\hat{y}_1 = \frac{e^{h_1}}{e^{h_1} + e^{h_2}} = \frac{1}{1 + e^{-(h_1 - h_2)}}$$

$$\hat{y}_2 = \frac{e^{h_2}}{e^{h_1} + e^{h_2}} = \frac{1}{1 + e^{-(h_2 - h_1)}}$$

Hierin herken je misschien de logistische functie $\sigma(a) = \frac{1}{1+e^{-a}}$. We vinden dan $\hat{y}_1 = \sigma(\Delta h)$ en $\hat{y}_2 = \sigma(-\Delta h)$, met $\Delta h = h_1 - h_2$. De kans op het eerste klasselabel neemt dus geleidelijk toe volgens een sigmoïde curve naarmate de pre-activatiewaarde h_1 van de output layer verder uitstijgt boven de waarde h_2. Dit is ook precies wat je zou verwachten, want h_1 codeert in zekere zin de kans op het eerste klasselabel. Tegelijkertijd is wel gegarandeerd dat $\hat{y}_1 + \hat{y}_2 = 1$, wat ook vereist is om de voorspellingen als kansen te kunnen interpreteren.

Voor de afgeleiden geldt op basis van het voorgaande dat

$$\frac{\partial \hat{y}_1}{\partial h_1} = \hat{y}_1 \cdot (1 - \hat{y}_1) = +\hat{y}_1 \hat{y}_2$$

$$\frac{\partial \hat{y}_1}{\partial h_2} = -\hat{y}_1 \cdot \hat{y}_2 = -\hat{y}_1 \hat{y}_2$$

$$\frac{\partial \hat{y}_2}{\partial h_1} = -\hat{y}_2 \cdot \hat{y}_1 = -\hat{y}_1 \hat{y}_2$$

$$\frac{\partial \hat{y}_2}{\partial h_2} = \hat{y}_2 \cdot (1 - \hat{y}_2) = +\hat{y}_1 \hat{y}_2$$

Deze hebben dus allemaal dezelfde grootte, alleen niet hetzelfde teken. Omdat $\hat{y}_1$ en $\hat{y}_2$ allebei tussen de 0 en 1 liggen volgt dat $\frac{\partial \hat{y}_1}{\partial h_1}, \frac{\partial \hat{y}_2}{\partial h_2} > 0$ en $\frac{\partial \hat{y}_1}{\partial h_2}, \frac{\partial \hat{y}_2}{\partial h_1} < 0$. Oftewel, als h_1 toeneemt dan neemt ook $\hat{y}_1$ toe maar daalt $\hat{y}_2$, terwijl als h_2 toeneemt dan stijgt $\hat{y}_2$ maar daalt $\hat{y}_1$.

De mate waarin de voorspelde kansen veranderen is klein als hetzij $\hat{y}_1$ of $\hat{y}_2$ in de buurt ligt van 0, want dan is immers ook het product $\hat{y}_1 \cdot \hat{y}_2$ klein. In dat geval is het model

overtuigd van de juistheid van de classificatie en is de kans op het ene klasselabel ongeveer gelijk aan 100% en die op het andere ongeveer gelijk aan 0%. Dan is de verandering die in het model zal optreden klein. Als het model het inderdaad bij het juiste eind heeft dan is dit wenselijk, want dan wil je niet dat het model zou veranderen. Echter, als het model ernaast zit (maar het wel zeker "denkt" te weten) dan leidt dit ook nauwelijks tot veranderingen, waardoor het model moeilijk leert van zijn fouten. Dit is een stuk minder wenselijk.

Het model leert in dit geval het snelst als het getraind wordt met instances die twijfelgevallen vormen. Dat wil zeggen, als zowel $\hat{y}_1$ en $\hat{y}_2$ ergens in de buurt liggen van $\frac{1}{2}$. In hoofdstuk 2 zagen we ook al dat uit de updateregel van het logistische regressiemodel bleek dat instances die nabij de scheidingslijn tussen de twee klassen liggen het zwaarst worden meegewogen, en instances die hier ver vandaan liggen een lagere weging krijgen. Dit is exact hetzelfde gedrag.

Kortom, een enkel neuron met een logistische functie als activatiefunctie komt in essentie op precies hetzelfde neer als een neuraal netwerk met een softmax layer als output layer. Een softmax layer heeft echter als voordeel dat deze ook toepasbaar is op multinomiale classificatie met meer dan twee klassen.

Opgave 71. *

Laat zien dat voor de logistische functie geldt dat $\sigma(a) + \sigma(-a) = 1$.

Opgave 72. ***

Toon aan dat voor de afgeleide van de logistische functie geldt dat $\frac{\partial}{\partial a}\sigma(a) = \sigma(a) \cdot (1 - \sigma(a))$.

4.4. Cross-entropy

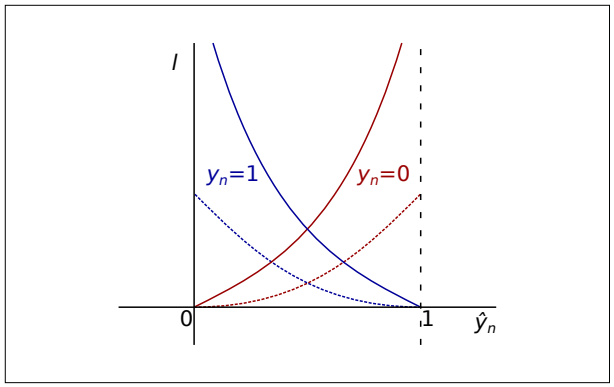
We hebben net een belangrijk nadeel ontdekt van de softmax functie zoals we die tot nu toe gebruiken. Als het model overtuigd is van de juistheid van de klasselabels, dan zullen de modelparameters maar weinig worden bijgewerkt. Dat is hiervoor aangetoond voor gevallen met twee klasselabels, maar dit geldt ook in het algemeen. Voor een deel is dit gewenst. Immers, als het model terecht een hoge kans toekent aan de juiste klasse, dan duidt dit op een goede classificatie, en dan hoef je het model helemaal niet aan te passen. Echter, als het model het weliswaar zeker meent te weten maar het eigenlijk mis heeft, dan is het ongewenst dat de gewichten niet of nauwelijks worden aangepast. Dit probleem heeft te maken met de *verzadiging* of *saturatie* die optreedt in de softmax-functie (en evenzo de logistische functie). Hiermee wordt bedoeld dat de helling uitermate klein wordt in de "staarten" van de functie. Dit betekent dat je maar weinig kan veranderen aan de uitkomsten van de functie door de parameters enigszins aan te passen, en het principe van gradient descent dicteert dan dat de modelparameters ook maar weinig aangepast zullen worden. Het model blijft dan steken in een slechte oplossing. Dit is in elk geval onwenselijk als de voorspelling onjuist is.

We zijn dit probleem eerder tegengekomen bij het perceptron, waar de signum-functie evenzo een kleine helling had. In dat geval trad het probleem in zijn meest extreme

vorm op: de helling van de signum-functie is vrijwel overal exact gelijk aan nul. Voor de softmax- en logistische functies geldt dit in afgezwakte vorm. Weliswaar wordt de helling nooit helemaal nul, en is deze ook niet letterlijk overal klein, maar als een instance per ongeluk in de staart van de functie terecht komt en verkeerd geclassificeerd wordt, dan is het heel moeilijk voor het model om hiervan te herstellen.

Het probleem zit eigenlijk niet alleen in de softmax- en logistische functie. Het heeft ook te maken met de lossfunctie die we gebruiken. Stel bijvoorbeeld dat een instance eigenlijk tot klasse nummer één behoort. Dus $y_1 = 1$, en alle andere $y_n = 0$. Dan bestraffen we het model als het komt tot een voorspelling met $\hat{y}_1 \neq 1$. Als het model bijvoorbeeld een kans van slechts 10% toekent aan klasse één, in plaats van 100%, dan is $\hat{y}_1 = 0.1$ en vinden we een kwadratisch loss $l = \mathcal{L}(0.1; 1) = 0.9^2 = 0.81$. Als het model echter een kans van 0% toekent aan die instance geeft dat een loss $l = 1.00$. In het eerste geval is het model weliswaar niet nauwkeurig, maar het geeft wel nog de mogelijkheid aan dat de instance tot klasse één behoort. In het tweede geval beweert het model absoluut zeker te zijn dat het onmogelijk klasse één kan zijn. Hoewel beide modellen ernaast zitten is dat tweede model qua interpretatie veel slechter dan het eerste. Dat komt niet goed tot uiting in de loss, die nauwelijks 25% gestegen is. We zouden voor classificatie-problemen eigenlijk beter een lossfunctie hanteren die een steeds sterker toenemende loss toekent naarmate de toegekende kans op een klasselabel dat eigenlijk juist is nadert naar nul.

In de onderstaande grafiek is een dergelijk gedrag geïllustreerd. In blauw is de loss geplot van een instance die eigenlijk een hoge gewenste kans dient te worden toegekend, dat wil zeggen $y_n = 1$. Gestippeld is de waarde van de kwadratische lossfunctie. Als de voorspelling exact juist is zodat $\hat{y}_n = 1$, dan wordt een loss $l = 0$ toegekend, zoals het hoort. Als echter $\hat{y}_n$ daalt naar nul, dan stijgt de loss geleidelijk naar 1. Belangrijk is dat voor alle $\hat{y}_n$ in de buurt van nul de loss in de buurt ligt van 1.0; er wordt weinig onderscheid gemaakt tussen een ingeschatte kleine kans $\hat{y}_n > 0$ en een nulkans $\hat{y}_n = 0$. We zouden mogelijk liever een andere lossfunctie kiezen die steeds steiler loopt naarmate $\hat{y}_n = 0$ wordt genaderd, en dus *asymptotisch* tot de x-as nadert. Dit betere gedrag is geschetst met een doorlopende blauwe lijn.



Voor de rode curven geldt iets soortgelijks. Als een instance eigenlijk niet tot een zekere klasse behoort zodat $y_n = 0$, dan is het model dat $\hat{y}_n = 0$ voorspelt natuurlijk het beste. Maar een model dat absoluut zeker denkt te zijn dat het wél tot die klasse behoort

met $\hat{y}_n = 1$ zou eigenlijk veel sterker moeten worden bestraft dan een ander model dat het alleen waarschijnlijk acht met bijvoorbeeld $\hat{y}_n = 0.9$.

Opnieuw zijn er tal van functies te bedenken met deze vorm. Een functie die we al eerder zijn tegengekomen (bij het vak *Introduction Datamining*) is de hoeveelheid *informatie* die een gebeurtenis met kans p met zich meebrengt. Deze is gelijk aan $H(p) = -\lg(p)$. In dit geval werd de binaire logaritme met grondtal 2 gebruikt, waarbij de uitkomst kan worden uitgedrukt in *bits*. Het is ook mogelijk om de natuurlijke logaritme $-\ln(p)$ met grondtal e te kiezen of de decimale logaritme $-\log(p)$ met grondtal 10; die schelen slechts een constante factor die niet tot een andere optimale oplossing zal leiden. We zullen vanaf hier werken met de natuurlijke logaritme.

De logaritme heeft precies een staart nabij de x-as die gebruikt kan worden om het gedrag in de bovenstaande figuur te modelleren. We vinden dan:

$$\mathcal{L}(\hat{y}_n; y_n) = \begin{cases} -\ln(1-\hat{y}_n) & \text{voor } y_n = 0 \\ -\ln(\hat{y}_n) & \text{voor } y_n = 1 \end{cases}$$

Ga voor jezelf na dat dit kan worden samengevat in één uitdrukking van de volgende vorm die klopt voor zowel $y_n = 0$ als $y_n = 1$:

$$\mathcal{L}(\hat{y}_n; y_n) = -y_n \cdot \ln(\hat{y}_n) - (1-y_n) \cdot \ln(1-\hat{y}_n)$$

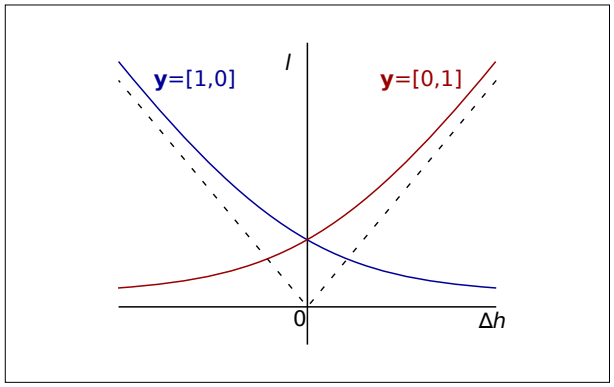
Deze formule heeft enige gelijkenis met de *entropie* $H = \sum -p \cdot \lg(p)$, zij het dat hier twee kansgrootheden $\hat{y}_n$ en y_n door elkaar worden gebruikt. De bovenstaande lossfunctie staat bekend als de *binary cross-entropy*. De cross-entropy geeft een maat voor hoe sterk de werkelijke en de geschatte kansverdelingen y_n en $\hat{y}_n$ van elkaar afwijken.

Als we meerdere uitvoeren $\hat{y}_n$ hebben wordt de loss reeds gesommeerd over al die uitvoeren: $l = \sum_n \mathcal{L}(\hat{y}_n; y_n)$. Het volstaat dan eigenlijk om de loss-functie te definiëren als

$$\mathcal{L}(\hat{y}_n; y_n) = -y_n \cdot \ln(\hat{y}_n)$$

Immers, de andere uitvoeren hebben samen een kans $1 - y_n$, dus de tweede term in de binary cross-entropy wordt al verdisconteert doordat de loss ook op die andere uitvoeren van de softmax-functie wordt toegepast. Deze vereenvoudigde versie van de cross-entropy heet de *categorical cross-entropy*. Als je de softmax-functie gebruikt en elk mogelijk klasselabel een eigen voorspelde kans heeft, dan is de categorische variant geschikt; als je maar twee klassen hebt en alleen de kans op de ene klasse voorspelt met een logistische activatiefunctie (en de andere klasse impliciet laat), dan dien je de binaire variant te kiezen.

Passen we deze lossfunctie toe op de uitkomst van de softmax layer dan krijgen we voor het eerdere vereenvoudigde dichotome geval met twee klassen $l = \sum_n \mathcal{L}(\hat{y}_n; y_n) = \mathcal{L}\left(\frac{1}{1+e^{-(h_1-h_2)}}; y_1\right) + \mathcal{L}\left(\frac{1}{1+e^{-(h_2-h_1)}}; y_2\right)$. Noemen we het verschil tussen de twee klassen $\Delta h = h_1 - h_2$, en passen we dit toe op een instance met het gewenste klasselabel één, oftewel de one-hot encoding $\boldsymbol{y} = [1, 0]$, dan leidt dit tot $l = \mathcal{L}\left(\frac{1}{1+e^{-\Delta h}}; 1\right) + \mathcal{L}\left(\frac{1}{1+e^{+\Delta h}}; 0\right)$ hetgeen in een aantal stappen vereenvoudigd kan worden tot $l = \ln\left(1 + e^{-\Delta h}\right)$. Het verloop van deze functie is hieronder geschetst in blauw.



Het gedrag van deze functie kan uitgelegd worden. Als h_1 groter is dan h_2 leidt dit tot een voorspelling waarin de hoogste kans aan het eerste klasselabel wordt gegeven. Dit is correct. In dit geval is $h_1 - h_2$ positief, en voor positieve Δh daalt de loss volgens de blauwe curve naar nul. Dat is juist, want een correcte classificatie dient een lage loss toegekend te worden. Omgekeerd wordt voor een model dat een incorrecte classificatie toekent een negatieve Δh gevonden. Hierbij hoort een toenemende loss, dus dit model wordt bestraft. Belangrijk is dat de helling van de blauwe curve aan de linkerkant niet satureert. Kortom, naarmate het model het slechter doet blijft de loss steeds verder toenemen en gaat gradient descent ervoor zorgen dat de modelparameters worden aangepast. Dit is precies het gewenste gedrag waar we naar op zoek waren. Wat er in feite gebeurt is dat de steeds vlakker lopende softmax-functie wordt gecompenseerd met een steeds steiler lopende cross-entropy. Deze twee werken elkaar precies zodanig tegen dat voor foutief geclassificeerde instances de gradiënt naar een constante nadert, maar niet langer naar nul gaat zoals bij de kwadratische lossfunctie. Voor correct geclassificeerde instances vlakt de curve wel af, maar dat is niet erg want die worden reeds juist geclassificeerd dus het model hoeft die niet nog beter te leren.

Voor een instance met het andere klasselabel $\boldsymbol{y} = [0, 1]$ geldt min of meer hetzelfde verhaal, maar dan gespiegeld. Je verkrijgt daarvoor de rode curve. De vorm van deze curve volgt nauw de *softplus-functie* $S^+(x) = \ln(1 + e^x)$. Deze heeft twee asymptoten, $S^+(x) = 0$ voor $x \downarrow -\infty$ en $S^+(x) = x$ voor $x \uparrow +\infty$. Deze curve satureert voor negatieve Δh, waarvoor het model een juiste voorspelling doet, maar satureert nooit voor positieve Δh waar het model het mis heeft. Hierdoor kan het model van zijn fouten blijven leren en is het probleem van de verzadiging van de softmax-functie verholpen.

Opgave 73. *

Stel we hebben een dichotoom classificatieprobleem met twee klassen: A en B. Schets in de vorm van een diagram twee verschillende neurale netwerken die in staat zijn om een dergelijk probleem te modelleren: eentje met één output die alleen de kans op klasse A voorspelt; en een ander met twee outputs die zowel de kans op A als de kans op B voorspelt. Hoe dien je je output layer in te richten en hoe dien je de loss-functie te kiezen in die beide gevallen?

Opgave 74. **

Gegeven is een classificatiemodel dat een voorspelling $\hat{\boldsymbol{y}} = [0.1, 0.4, 0.2, 0.3]$ toekent aan een instance die, van vier mogelijk labels A tot en met D, eigenlijk het ware label B heeft. (Merk op dat het model inderdaad de hoogste kans van 40% toekent aan dit tweede label B.) Bereken de loss als je gebruik zou maken van de kwadratische loss-functie en doe hetzelfde voor de cross-entropy loss-functie. Stel vervolgens dat het model deze uitkomst zou voorspellen voor een instance met waar label A. (Merk op dat het model dan de laagste kans van 10% toekent aan label A.) Bepaal ook dan weer de loss voor beide genoemde loss-functies.

Opgave 75. ***

Stel, we hebben een classificatieprobleem met honderd identieke instances, waarvan er 60 tot klasse A behoren en 40 tot klasse B. Laat zien dat de behaalde cross-entropy loss daadwerkelijk minimaal is als het model voor deze instances een waarschijnlijkheid $\hat{y}_A = 0.6$ toekent aan klasse A en $\hat{y}_B = 0.4$ aan klasse B.

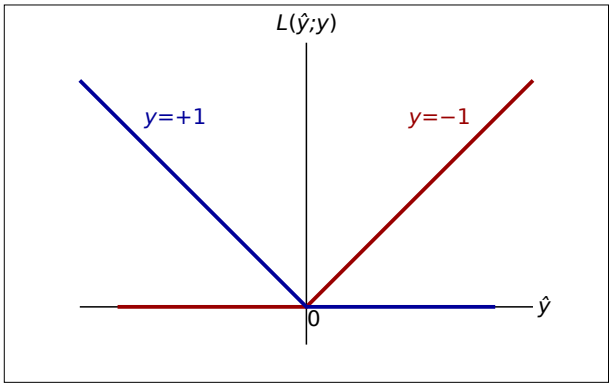
Opgave 76. ***

Leid af dat voor een instance met de gewenste one-hot classificatie $\boldsymbol{y} = [0, 1]$, gebruikmakend van een model met een hidden layer met twee uitvoerwaarden h_1 en h_2, een softmax layer, en een categorical cross-entropy lossfunctie, de totale loss gelijk is aan $l = S^+(h_1 - h_2)$.

4.5. Tot slot

Hoewel we oorspronkelijk zijn begonnen met het perceptron model en dat al doende hebben gegeneraliseerd tot algemene neuronen die veel meer kunnen dan wat het perceptron kan, zijn we er tot dusverre niet in geslaagd om het perceptron zelf te formuleren in termen van lossfuncties. Dit had ermee te maken dat de signum-functie die het perceptron gebruikt niet differentieerbaar is rond nul en helling nul heeft daarbuiten. Met het resultaat van de vorige paragraaf kunnen we echter toch komen tot een model van het perceptron op basis van lossfuncties.

Herinner je je dat we het logistische model bedacht hebben als een afgevlakte versie van het perceptron dat een te "harde" stapvormige activatiefunctie had? Het ligt voor de hand dat je omgekeerd dus het perceptron model kan verkrijgen door een afgevlakt logistisch regressie model weer "hard" te maken. Hierboven hebben we gezien dat als we de softmax-functie en de cross-entropy combineren, dat we dan een logistisch regressiemodel verkrijgen dat niet satureert. De combinatie van softmax en cross-entropy kwam neer op een softplus-functie die aan één zijde naar nul gaat en aan de andere zijde blijft hellen. Voor de softplus-functie worden deze twee regimes verbonden door een gladde overgang. Het lijkt aannemelijk dat voor het perceptron dan hetzelfde soort gedrag gekozen moet worden, maar met een harde overgang.



Hierboven staat een dergelijke "harde" functie geschetst. Hierbij zijn tijdelijk weer klasselabels van de vorm $y = \pm 1$ gebruikt omdat het perceptron hiermee werkte. Het functievoorschrift bij deze grafiek luidt $\mathcal{L}(\hat{y}; y) = \max(-\hat{y} \cdot y, 0)$. Het perceptron kan dus worden verkregen door deze lossfunctie toe te passen op een enkel neuron met de identiteitsfunctie als activatiefunctie. Merk op dat nu niet langer de signum-functie als activatiefunctie optreedt, vanwege diens genoemde problemen. De "harde" overgang die eerst in de activatiefunctie zat is nu ingebakken in de lossfunctie. In tegenstelling tot de signum-functie is echter de bovenstaande perceptron-lossfunctie wél numeriek en satureert deze niet. Wel moeten we nu de voorspellingen $\hat{y}$ iets ruimer interpreteren. Dit model produceert namelijk niet strict voorspellingen met waarden $\hat{y} = \pm 1$. Als de voorspelling $\hat{y} > 0$ dan dient dit geïnterpreteerd te worden als klasselabel $\hat{y} = +1$ en als $\hat{y} < 0$ dan duidt dit op $\hat{y} = -1$.

Voor instances met het klasselabel $y = -1$ is de lossfunctie in rood afgebeeld. Voor $\hat{y} < 0$ (links van de y-as) is de loss gelijk aan nul. Dit is wel te begrijpen, omdat een negatieve waarde van $\hat{y}$ zoals zojuist gezegd als een voorspelling van het klasselabel -1 moet worden gezien. Deze voorspelling is correct voor deze instances, dus deze mogen loss nul toegewezen krijgen. Als de voorspelling het verkeerde teken heeft (rechts van de y-as) loopt de lossfunctie echter steeds verder op. De helling van de lossfunctie is dan constant, wat ertoe leidt dat de updateregel de parameters altijd op dezelfde wijze bijwerkt, ongeacht hoe ver het punt aan de verkeerde kant van de scheidingslijn ligt. Dit is precies hoe het perceptron werkt: punten aan de verkeerde kant van de lijn leiden altijd tot eenzelfde update; punten aan de goede kant van de lijn leiden niet tot een update. Voor instances met het klasselabel $y = +1$ komt de lossfunctie neer op de blauwe lijn. Dan worden voorspellingen met $\hat{y} > 0$ niet bestraft, en is de helling constant voor $\hat{y} < 0$.

Hierboven hebben we laten zien dat het probleem van saturatie van de softmax-functie kan worden opgelost door een andere lossfunctie te kiezen. Zo leidde het gebruik van de cross-entropy ertoe dat het model samen met de softmax laag effectief een softplus-functie optimaliseert. Deze heeft als prettig kenmerk dat deze niet satureert; althans, niet voor instances die verkeerd worden geclassificeerd. Tijdens de optimalisatie van het model leidt dit ertoe dat foutief geclassificeerde instances blijven zorgen voor aanpassingen in de modelparameters.

Het genoemde probleem betreffende saturatie treedt echter niet alleen op voor de softmax-functie. Ook activatiefuncties in eerdere lagen van het neurale netwerk zijn hier

gevoelig voor. Zo heeft de tanh-functie net als de logistische functie twee staarten die geleidelijk vlak gaan lopen. Als het model per toeval in die staarten terecht komt zorgt dit ervoor dat de gewichten en biases van de neurale lagen nauwelijks meer aangepast zullen worden. Als dit voor vrijwel alle instances geldt wordt ook wel gezegd dat het neuron is *overleden*, aangezien het niet meer bijdraagt aan het leerproces.

Saturatie kan voor hidden layers niet tegengegaan worden door een andere lossfunctie te kiezen. Immers, de lossfunctie wordt pas achteraan het model toegepast en kan in de praktijk daardoor hoogstens de saturatie van de output layer beïnvloeden. Wel is het mogelijk om simpelweg een andere activatiefunctie te nemen. Jaren geleden was de tanh-functie de meest gebruikte activatiefunctie omdat deze wel enige gelijkenis toont met de manier waarop neuronen in de hersenen actief worden als ze voldoende geëxciteerd worden. Recentelijk wordt echter steeds vaker gekozen voor activatie-functies die minder gevoelig blijken te zijn voor saturatie, zoals de hierboven reeds genoemde softplus-functie.

Nog gebruikelijker is een vereenvoudigde vorm van de softplus-functie die geen gladde overgang heeft rondom nul, maar bestaat uit twee lineaire functies die met een knik aan elkaar vast zitten. Deze zou je de "hardplus-functie" kunnen noemen, maar staat beter bekend als de *rectified linear unit*, oftewel de *relu-functie*. Deze wordt gedefinieerd door $f(x) = x$ voor $x \geq 0$ en $f(x) = 0$ voor $x < 0$. Een korte manier om dit te schrijven is $f(x) = \max(x, 0)$ voor alle x. Hij werkt vergelijkbaar met een gelijkrichter diode in de electronica of een terugslagklep in een waterleiding aangezien deze signalen met een positief teken ongewijzigd doorlaat maar negatieve invoerwaarden totaal blokkeert op nul. De relu is daarmee de meest typische vertegenwoordiger van een familie *rectifier* activatiefuncties. De relu heeft als sterke punt dat zowel de functie zelf als diens afgeleide zeer eenvoudig numeriek is te bepalen, wat het een efficiënte keuze maakt.

Samengevat zijn tegenwoordig relu-functies populair als activatiefuncties in de hidden layer(s), hoewel je ook nog wel tanh-functie tegenkomt of andere varianten die hierboven genoemd zijn. Het is soms een kwestie van experimenteren, maar het kan meestal geen kwaad om te beginnen met relu-functies. Als je classificatie uitvoert worden die standaard gevolgd door een softmax layer en een loss layer gebaseerd op cross-entropy; als je regressie uitvoert zijn de laatste lagen gewoonlijk een lineaire output layer met de identiteitsfunctie als activatiefunctie en een kwadratische loss layer. Van deze keuze voor de output en loss layers wordt maar zelden afgeweken.

Hoeveel hidden layers er nodig zijn en hoe breed deze lagen dienen te zien hangt af van de complexiteit van het op te lossen probleem. Dit vereist enige ervaring om goed in te kunnen schatten, en blijft ook dan altijd een kwestie van uitproberen.

Opgave 77. *

Leg uit dat de afgeleide van de relu-functie geschreven kan worden in termen van de signum functie als $\frac{\mathrm{d}}{\mathrm{d}a}\mathrm{relu}(a) = \frac{1+\mathrm{sgn}(a)}{2}$.

Opgave 78. *

Het perceptron kan enerzijds geformuleerd worden door als activatie-functie de signum-functie $\hat{y} = \mathrm{sgn}(a)$ te nemen, of anderzijds door als loss-functie de functie $\mathcal{L}(\hat{y}; y) = \max(-\hat{y} \cdot y, 0)$ te kiezen. Bespreek van beide keuzes een voor- en nadeel.

Opgave 79. **

Zoek zelf informatie over de ELU-functie (exponential linear unit) en de SiLU-functie (Sigmoid Linear Unit). Plot hun vorm in een grafiek samen met de relu- en softplus-functies en ga na dat het allemaal varianten van rectifier-functies zijn.

Opgave 80. **

Druk de lossfunctie van het perceptron $\mathcal{L}(\hat{y}; y) = \max(-\hat{y} \cdot y, 0)$ uit in termen van de relu-functie in plaats van de max-functie.

Deel III.

Uitbreiding

5. Regularisatie

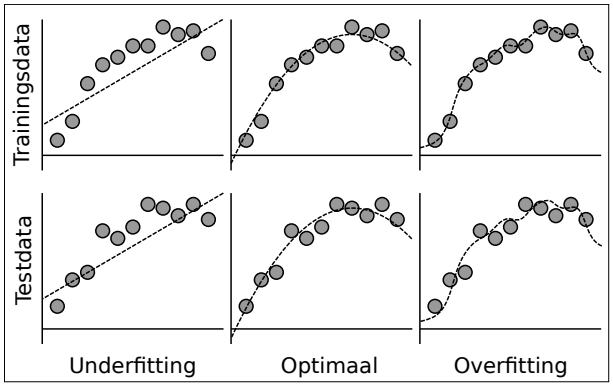
Vooruitblik

Tijdens het trainen van neurale netwerken ligt het risico van overfitting op de loer. In dit hoofdstuk bekijken we een aantal methoden die erop gericht zijn om het aantal (of de grootte) van de fouten tijdens kruis-validatie te beperken, zelfs als dit ten koste gaat van meer (of grotere) fouten op de trainingsdata. Hieronder vallen in het bijzonder early stopping, data augmentatie, L_1- & L_2-regularisatie, en dropout.

5.1. Het bias-variance probleem

Afhankelijk van de complexiteit van het probleem in verhouding tot het model bestaat bij machine learning in het algemeen het risico op underfitting danwel overfitting.

We spreken van *underfitting* (of *undertraining*) wanneer het model niet voldoende flexibel is om de relaties die in de data aanwezig zijn te modelleren. Bijvoorbeeld, wanneer data niet lineair afhankelijk zijn (in het geval van een regressieprobleem) of niet lineair separabel (in het geval van een classificatieprobleem), en je zou deze data toch fitten met een lineair model, dan kan het model misschien wel een lineaire benadering vinden, maar het is niet in staat om de niet-lineariteiten goed te beschrijven. Daarvoor is het model in zekere zin te eenvoudig. Het bevat dan gewoonlijk te weinig parameters oftewel vrijheidsgraden om de ingewikkelde data voldoende nauwkeurig te benaderen. Er wordt dan ook wel gezegd dat het model een hoge *bias* heeft: het model maakt systematisch altijd hetzelfde soort fouten, ongeacht de steekproef aan data. (Let op: dit is een ander gebruik van het woord bias als de modelparameter b.) Dit komt overeen met de linker situatie in de figuur hieronder.



Omgekeerd spreken we van *overfitting* (of *overtraining*) wanneer het model juist té flexibel is en ook niet-bestaande toevallige effecten in de data gaat beschrijven. Wanneer bijvoorbeeld een ingewikkeld model met tal van parameters wordt toegepast op een relatief simpele dataset met weinig datapunten, dan bestaat het risico dat het model niet alleen de systematische relaties gaat beschrijven die in de data aanwezig zijn, maar ook nog vrijheidsgraden "over heeft" om allerlei niet-reproduceerbare effecten die willekeurig in de data aanwezig zijn te fitten. In dit geval spreken we niet van bias maar van *variantie*: de kwaliteit van de fit varieert willekeurig met de toevallige effecten in de dataset. Dit wordt geïllustreerd in de rechter panelen van de vorige figuur.

In de praktijk wil je natuurlijk het liefst een model hebben dat net voldoende vrijheden heeft om alle systematische effecten in de data te beschrijven, maar geen extra vrijheden over heeft om ook de ruis te modelleren. Dit wordt de *bias-variance trade-off* genoemd. Als dat lukt dan noem je het model *parcimonieus*. In de figuur is op het oog redelijk te zien wat er aan de hand is, maar helaas is het in de praktijk meestal niet goed bekend wat de precieze vorm van het model zou moeten zijn of hoeveel parameters daarin zouden moeten worden opgenomen.

Om inzicht te krijgen in parcimonie, of dat je model lijdt aan under- of overfitting, wordt meestal gebruik gemaakt van meerdere afzonderlijke datasets.

1. De *trainingsdata* dienen om de modelparameters van de neuronen in het netwerk te optimaliseren. Dit betreft voor een neuraal netwerk de willekeurig geïnitialiseerde biases b_i en gewichten w_{ij} die automatisch door stochastic gradient descent met back-propagation worden bepaald.

2. De *validatiedata* dienen om de hyperparameters van het model geschikt te kiezen. Dit betreft instellingen die door de gebruiker handmatig worden gekozen, zoals de diepte van het netwerk, de breedte van de lagen, de gebruikte activatiefuncties en learning rate, of het aantal epochs waarover getraind wordt.

3. De *testdata* tenslotte dient om de kwaliteit van het uiteindelijke model betrouwbaar te evalueren. Deze data mag nooit worden gebruikt om model- of hyperparameters mee te optimaliseren; anders gezegd, zodra je de testdata gebruikt mag het model niet meer gewijzigd worden.

Overfitting is herkenbaar doordat het model op de trainingsdata significant beter presteert dan op de validatiedata. Dit duidt erop dat de modelparameters te specifiek zijn afgestemd op de trainingsdata, maar niet generaliseren naar andere steekproeven van soortgelijke data. Je kan overigens ook overfitting krijgen als je als onderzoeker te lang blijft tweaken aan je model en je hyperparameters te ver door optimaliseert. Dan presteert het model op de testdata slechter dan op de validatiedata. Het principe dat je een model dient te evalueren op een andere dataset dan waarmee je het optimaliseert wordt *kruis-validatie* genoemd.

In het geval van neurale netwerken speelt het bias-variance probleem een extra belangrijke rol omdat je als ontwerper van een neuraal netwerk de vrijheid hebt om zelf te kiezen hoeveel lagen je in een model opneemt, en hoeveel neuronen je in elk van die lagen

stopt. Elk neuron heeft naast een bias-parameter b_i ook nog eens evenveel gewichten w_{ij} als het aantal inputs van de desbetreffende laag, dus daarmee loopt het totale aantal te fitten parameters al gauw enorm op. Het aantal parameters bepaalt onder andere hoe goed het model in staat is om diverse soorten datasets te fitten. Dit wordt de *capaciteit* van het model genoemd. Een model met een te lage capaciteit zal underfitten; een model met een te hoge capaciteit neigt naar overfitten.

De capaciteit is niet in zijn eentje bepalend of er under- of overfitting plaatsvindt. Dit hangt ook samen met de hoeveelheid data en de complexiteit van het probleem. Een kleine hoeveelheid data met een eenvoudige structuur zal zelfs een eenvoudig model al gauw "van buiten" kunnen leren, terwijl grote hoeveelheden data met daarin ingewikkelde verbanden een complex model zullen vereisen. In de praktijk is het dan ook de verhouding tussen de capaciteit van het model enerzijds en de complexiteit van de data en het probleem anderzijds die bepaalt of er under- of overfitting optreedt.

		Model-capaciteit:	
		laag	*hoog*
Data-complexiteit:	*laag*	parcimonieus	overfitting
	hoog	underfitting	parcimonieus

In de praktijk ben je er niet zozeer in geïnteresseerd of je model het goed doet op de trainingsdata. Het gaat erom je model uiteindelijk toe te passen op nieuwe, onbekende data. Het eigenlijke doel is om de nauwkeurigheid op de testdata zo hoog mogelijk te maken, of de gemiddelde loss op de testdata zo laag mogelijk krijgen. Tot dusverre deden we dat door te proberen de loss op de trainingsdata te minimaliseren middels stochastic gradient descent en back-propagation. We gebruikten de loss op de trainingsdata als een meetbare surrogaatuitkomst of *proxy* voor de dan nog onbekende loss op de testdata.

Er zijn methoden die weliswaar de prestaties op de trainingsdata verslechteren, maar toch de prestaties op de testdata typisch verbeteren. Deze heten *regularisatiemethoden* en zijn geschikt om overfitting tegen te gaan. In dit hoofdstuk zullen we er hiervan een aantal bekijken.

Opgave 81. *

Zoek de letterlijke betekenis van het woord parcimonie/parcimonieus (Engels: parsimony/parsimonious) op in een woordenboek. Kun je dit relateren aan de betekenis in de context van een Machine Learning model?

Opgave 82. *

Deel de volgende paren begrippen op in twee aparte groepjes van termen die allemaal bij elkaar passen: overfitting / underfitting; te simpel model / te complex model; hoge capaciteit / lage capaciteit; bias / variance; te veel vrijheidsgraden / te weinig vrijheidsgraden; eenvoudige datastructuur / ingewikkelde datastructuur; undertraining / overtraining; systematische fouten / willekeurige fouten; te veel modelparameters / te weinig modelparameters.

Opgave 83. *

Hoeveel modelparameters in totaal heeft een neuraal netwerk met een input layer met tien inputs, drie hidden layers met elk een breedte van honderd neuronen en een tanh-activatiefunctie, en een output layer met wederom tien neuronen en een softmax-activatiefunctie?

Opgave 84. *
Leg in je eigen woorden uit waarom je een model alleen maar in zijn definitieve uiteindelijke vorm mag toepassen op testdata, maar niet gaandeweg tijdens de ontwikkeling en optimalisatie.

Opgave 85. **
Een manier om overfitting tegen te gaan is door een grotere hoeveelheid trainingsdata te verzamelen. Denk na wat een dergelijke vergroting betekent voor de nauwkeurigheid op de trainingsdata en testdata in het geval van een model dat neigt naar overfitting. Is het vergroten van de trainingsdataset te beschouwen als een regularisatiemethode?

Opgave 86. **
Een andere manier om overfitting tegen te gaan is door de capaciteit van het gebruikte model te verlagen, bijvoorbeeld door minder neuronen in je model te verwerken. Denk na wat een dergelijke capaciteitsverlaging betekent voor de nauwkeurigheid op de trainingsdata en testdata in het geval van een model dat neigt naar overfitting. Is het verlagen van de capaciteit van het model te beschouwen als een regularisatiemethode?

Opgave 87. **
Anita beweert: "een model met een te *hoge* capaciteit vertoont *bias* als diens parameters onjuist geoptimaliseerd zijn", waarop Bob tegenwerpt: "een model met een te *lage* capaciteit vertoont *variantie* als diens parameters onjuist geoptimaliseerd zijn." Met wie ben je het eens?

Opgave 88. **
Regularisatiemethoden verbeteren de loss op testdata ten koste van een verslechtering in de loss op trainingsdata. Leg uit dat dit overfitting kan tegengaan.

Opgave 89. ***
Bij classificatie kan de nauwkeurigheid gemeten worden aan de hand van het percentage instances dat juist wordt geclassificeerd, dat wil zeggen de accuracy. Een hoge nauwkeurigheid en een lage loss zijn in de praktijk weliswaar gerelateerd, maar het verband is niet exact. Beschrijf een denkbeeldig voorbeeld van een classificatieprobleem waarin een (behoorlijk) hoge nauwkeurigheid toch kan samengaan met een (behoorlijk) hoge gemiddelde loss.

Opgave 90. ***
Motiveer welk van de volgende klassieke machine learning algoritmen in jouw ogen de allerlaagste capaciteit heeft, en welk de allerhoogste: J48-Tree, Naive Bayes, ZeroR, OneR, k-Nearest Neighbor, Logistische Regressie, Random Forest, Support Vector Machine.

5.2. Early stopping

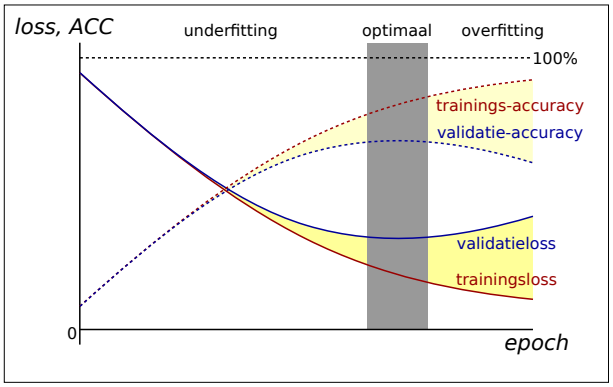
Overfitting treedt op wanneer een model te zeer is getraind op de fijne details van een trainingsdataset die niet generaliseren naar nieuwe data. Als je mag aannemen dat het model aanvankelijk een grove benadering vormt van de data en gaandeweg steeds fijner afgestemd raakt, dan is het aannemelijk dat het model tijdens de eerste epochs van de training vooral de algemene structuur van alle trainingsdata leert die de trainingsdata wél gemeenschappelijk heeft met de testdata, en pas in latere epochs de kleine toevalligheden in sommige trainingsdata leert die niet reproduceerbaar hoeven te zijn in onafhankelijke data.

De training kan dan worden verdeeld in verscheidene fasen. Aanvankelijk is het model willekeurig geïnitialiseerd en ongetraind. Het benadert de data dan sowieso slecht, dus het vertoont bias, indicatief voor underfitting. Na verloop van een aantal epochs raakt het model echter getraind op de relevante structuur van de data en gaat het beter presteren op zowel de trainings- als testdata. Als het model een voldoende grote capaciteit heeft gaat het model uiteindelijk echter ook toevallige kenmerken van de trainingsdata leren. Het gaat dan geleidelijk lijden aan variance, indicatief voor overfitting. Het presteert dan nog wel steeds beter op de trainingsdata, maar verslechtert op de testdata.

Ergens daar tussenin ligt vermoedelijk een optimum waar het model een balans heeft tussen underfitting en overfitting. Dit zouden we graag bereiken, maar we weten natuurlijk niet van tevoren hoeveel epochs hiervoor nodig zijn. We kunnen dit echter doen door tijdens de training de loss op zowel de trainingsdata (waarmee we optimaliseren) als de validatiedata (waarmee we niet optimaliseren) bij te houden. De loss op de validatiedata gebruiken we als proxy voor de loss op de testdata, waarin we eigenlijk geïnteresseerd zijn.

Aanvankelijk zal de loss op zowel de trainings- als validatiedata gezamenlijk afnemen naarmate het model verbetert. Op een gegeven moment vlakt de validatieloss echter af, en gaat deze mogelijk zelfs toenemen, terwijl de loss op de trainingsdata langzaam verder afneemt. Het moment waar deze twee losses uit elkaar gaan lopen is het moment waarop overfitting een rol begint te spelen. Het moment daarna waarop de validatieloss net vlak begint te lopen en tenslotte mogelijk zelfs toeneemt is het moment waarop een optimale balans tussen underfitting en overfitting bestaat.

Een zinvol hulpmiddel om dit gedrag te visualiseren is de *validatiecurve*. Hierin zet je als functie van het volgnummer van de epoch (op de horizontale as) de trainings- en validatielosses (op de vertikale as) uit. Als alternatief kan ook de behaalde nauwkeurigheid worden uitgezet, of een andere relevante maat die iets zegt over de kwaliteit van het model (zoals de false positive rate of positive predictive value, de area under the ROC-curve, de F_1-score, enzovoorts). Hieronder is een dergelijke validatiecurve geschetst met de gemiddelde loss in doorgetrokken lijnen, en in dezelfde figuur de nauwkeurigheid (accuracy, ACC) in stippellijnen. In de praktijk zullen deze grafieken overigens veel ruiziger en minder glad verlopen, deels omdat het model tijdelijk nabij lokale minima van de loss-functie kan blijven steken, en deels omdat de verschillen tussen de trainings- en testdata toevallig zijn.



Je kunt de informatie in deze validatiecurve op verschillende manieren gebruiken. Je zou tijdens het trainen de modelparameters kunnen onthouden van het model met de laagste gemiddelde loss (of hoogste nauwkeurigheid) op basis van de validatiedata. Of je zou het model een aantal keren kunnen trainen met willekeurige beginwaarden en telkens kijken na hoeveel epochs ongeveer het optimum in de validatieloss wordt bereikt; je weet dit pas zodra het model reeds overfit is geraakt, maar daarna kun je een nieuw model precies zoveel epochs trainen als optimaal nodig is.

Het zal je niet verbazen dat deze methode *early stopping* wordt genoemd. Het is een regularisatiemethode. Immers, door vroegtijdig te stoppen wordt de trainingsloss niet zo klein als deze had kunnen zijn geweest door langer door te trainen; de validatieloss echter is wel beter dan wat je zou bereiken als je langer door was gegaan.

Opgave 91. *

Verwacht je dat de trainings-loss willekeurig dicht naar nul zou dalen, en de trainings-accuracy helemaal naar 100% zou naderen, als je "oneindig lang" zou kunnen door blijven trainen?

Opgave 92. **

Hoe hangt het antwoord op de vorige vraag af van de capaciteit van je model?

Opgave 93. **

Normaal neemt de loss op de trainingsdata geleidelijk aan af, in het begin snel en geleidelijk aan langzamer, soms met haperingen. De loss op de trainingsdata zou echter nooit systematisch mogen toenemen: immers, gradient descent probeert precies de loss op de trainingsdata stapsgewijs te minimaliseren. Toch komt het wel eens voor dat de loss op de trainingsdata zichtbaar wild heen en weer springt en gedurende sommige epochs flink toeneemt. Wat zou er dan aan de hand kunnen zijn?

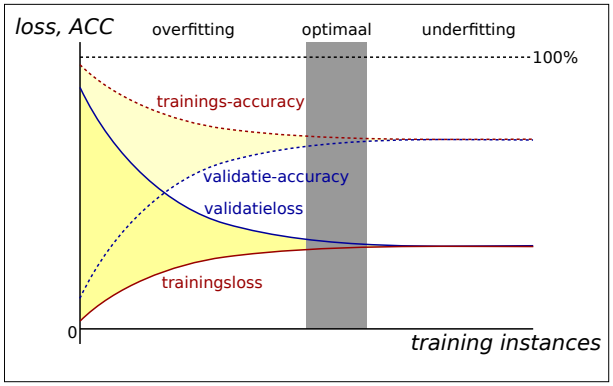
Opgave 94. **

Stel dat de validatiecurves nogal ruizig zijn, en je kiest als uiteindelijke modelparameters van je optimaal getrainde model die waarden die actueel waren toen de validatie-loss minimaal was. Is deze behaalde minimale validatie-loss dan een betrouwbare maat voor de kwaliteit van het model die generaliseert naar de testdataset, denk je? Motiveer je antwoord.

5.3. Data augmentatie

Het gedrag van een model omtrent over- en underfitting hangt af van de trainingsdata. Immers, overfitting treedt op wanneer een model in staat is om toevallige details in de trainingsdata te beschrijven. Wanneer de trainingsdataset echter groter en groter wordt, wordt het steeds onwaarschijnlijker dat die toevallige details voor álle trainingsdata blijven gelden. Met andere woorden, toevalligheden middelen steeds meer uit in de trainingsdata naarmate er hier meer van beschikbaar zijn. Hierdoor gaat de trainingsloss omhoog, in de richting van de validatieloss. Tegelijkertijd zal het model representatiever worden naarmate er meer trainingsdata is om van te leren. Dit zal ook tot uiting komen in de validatiedata: de prestaties hierop worden geleidelijk iets beter.

Ook dit gedrag kunnen we samenvatten in een grafiek. Dit keer zetten we op de horizontale as niet een hyperparameter zoals het aantal epochs uit, maar het aantal trainingsinstances in de dataset waarmee het model getraind wordt. Langs de vertikale as zetten we wederom de loss of de nauwkeurigheid (of een soortgelijke kwaliteitsmaat) uit; in het voorbeeld hieronder zijn die weer respectievelijk met doorgetrokken en gestippelde lijnen geschetst. De resulterende grafiek wordt de *leercurve* of *trainingscurve* genoemd.



Wanneer er maar weinig trainingsdata beschikbaar is, is het risico op overfitting het grootst. Naarmate er meer data voorhanden is reduceert dat risico en lopen de curven geleidelijk naar elkaar toe. Er is dan geen sprake meer van overfitting. Wel zou er underfitting kunnen optreden. Dit hangt af van de capaciteit van het model in relatie tot de complexiteit van de data: als de data nog structuur bevat die het model niet kan beschrijven, dan is er sprake van underfitting; echter, als het model in essentie alle structuur te pakken heeft die er in de data aanwezig is, dan blijft het model optimaal.

Vanaf het moment dat de curven niet meer significant van elkaar verschillen heeft het niet veel zin om nog meer trainingsdata toe te voegen. Omdat het in de praktijk vaak duur is om betrouwbare data te verzamelen kan de leercurve een goede indicatie vormen of je je energie het beste kan steken in het vergaren van meer trainingsdata of in het pogen te verbeteren van de opzet van het model.

Er is een goedkoper alternatief voor het verzamelen van meer trainingsdata. Vaak is het mogelijk om de attributen van trainingsinstances enigszins te wijzigen, waarna nieuwe instances ontstaan die eveneens mogelijk hadden kunnen zijn. Het hangt van

de aard van het probleem af welk soort transformaties mogelijk zijn. Soms kan volstaan worden met het toevoegen van een hoeveelheid ruis, bijvoorbeeld door normaal verdeelde random waarden op te tellen bij numerieke attributen. Natuurlijk dient de hoeveelheid ruis niet dusdanig groot te zijn dat het karakter van de instance zodanig verandert dat deze bijvoorbeeld niet meer als een realistische instance kan worden herkend of zelfs tot een andere klasse zou kunnen gaan behoren. Soms is het mogelijk om attributen van op elkaar lijkende instances van eenzelfde klasse te mengen of te middelen.

Als instances bestaan uit afbeeldingen zijn transformaties met name nuttig. Zo kan het bijvoorbeeld toelaatbaar zijn om de afbeelding te spiegelen, roteren, transleren, schalen, of croppen, of om de kleur, contrast, helderheid, saturatie, of scherpte te veranderen. Een microscopiefoto van cellen in een weefsel zou bijvoorbeeld prima op allerhande manieren getransformeerd kunnen worden, zolang hoogstens niet dusdanig ver in-/uitgezoomd wordt dat de celgrootte onrealistisch wordt; afbeeldingen van een handschrift kunnen daarentegen niet zomaar gespiegeld of gedraaid worden omdat dan p's, q's, b's, en d's in elkaar zouden worden omgezet, of 6en dan 9s zouden worden. Hoe dan ook, de te leren uitkomsten dienen *invariant* te zijn onder de transformatie.

Het proces van het modificeren van data waarbij varianten van instances geproduceerd worden heet *data augmentatie*. Data augmentatie is niet zo effectief als het verzamelen van meer trainingsdata, omdat er niet werkelijk meer diversiteit in de trainingsdata ontstaat, maar het kan desalniettemin bijdragen aan het verminderen van overfitting. Het kan vooraf worden toegepast om de trainingsdataset kunstmatig te vergroten, maar het is gebruikelijker om tijdens het leren elke instance opnieuw willekeurig te transformeren. In dit laatste geval hoeft tijdens de training nooit twee keer exact dezelfde instance te worden aangeboden aan het model, maar dat is alleen haalbaar als het uitvoeren van de transformatie zelf niet een tijdrovende operatie is.

Data augmentatie is een vorm van regularisatie omdat het de prestaties op trainingsdata verslechtert, ten gunste van de prestaties op de test- en validatiedata. Dit komt duidelijk tot uitdrukking in de leercurve hierboven.

Opgave 95. *

De bekende iris-dataset bevat voor drie verschillende soorten irissen (*Iris setosa*, *Iris virginica* en *Iris versicolor*) de lengte- en breedte-afmetingen van kelk- en kroonbladeren. Het totale aantal instances (50 per klasse) is echter zeer beperkt. Hoe zou je deze dataset kunnen augmenteren?

Opgave 96. *

Welke vormen van data augmentatie acht je zoal geschikt wanneer je een model traint om aan de hand van kleurenpasfoto's gezichten te herkennen?

Opgave 97. *

Iemand wil een model ontwikkelen dat zeedieren van landdieren kan onderscheiden op basis van de lichaamsafmetingen zoals lengte en lichaamsgewicht. Enerzijds zitten er gegevens over zeedieren zoals met name zalmen en walvissen in de data; anderzijds zitten er gegevens over landdieren zoals met name honden en olifanten in de dataset. Om de data te augmenteren wil deze persoon twee instances van dezelfde klasse nemen en hun

gegevens middelen om een nieuwe virtuele instance van die klasse te creëren. Vind je dit een verstandig idee?

Opgave 98. *

Stel je wil de verschillende soorten terreinen in satellietfoto's classificeren; denk aan water, bos, heide, stedelijk gebied, weilanden, zandvlaktes, enzovoorts. Je hebt een beperkt aantal voorbeelden met de hand gelabeld, maar omdat dit intensief werk is overweeg je augmentatie toe te passen. Noem een aantal soorten transformaties die wel geschikt zijn, en een aantal die niet geschikt zijn.

Opgave 99. **

In plaats van ruis toe te voegen aan instances tijdens data augmentatie is het ook mogelijk om tijdens het trainen een klein beetje ruis toe te voegen aan de uitvoer van elk neuron in een netwerk zelf. De post-activatiewaarde wordt dan berekend als $h_j = \varphi\left(\sum_i w_{ij} \cdot x_i\right) + \varepsilon_j$, waarin ε_j voor elke instance opnieuw getrokken wordt uit een random verdeling. Wanneer je niet traint, tijdens het doen van voorspellingen op nieuwe data, voeg je echter geen ruis toe. Motiveer of je dit zou kunnen beschouwen als een regularisatiemethode.

5.4. Weight regularisation

Een eenvoudig idee om overfitting tegen te gaan is door het model simpeler te kiezen. Als we het aantal modelparameters verlagen reduceren we echter ook de capaciteit van het model om diverse verdelingen van data te fitten. Als we tevoren niet goed weten hoe de data eruitziet, of niet goed weten hoe dat te vertalen in een simpel model, dan lopen we het risico op underfitting.

Een benadering die hieraan tegemoet komt bestaat eruit weliswaar een model op te zetten dat veel modelparameters bevat, maar tijdens het trainen het model te bestraffen als het gebruik maakt van al deze parameters. Of, omgekeerd, we stimuleren dat het model de gewichten gelijk stelt aan nul zodat die effectief "niet gebruikt" worden. We doen dit door de bestaande berekening van de loss volgens $l = \sum_m \mathcal{L}\left(\hat{y}_m; y_m\right)$ uit te breiden met extra termen die een positieve loss toekennen als de gewichten w_{ij} ongelijk aan nul worden gekozen. We verkrijgen dan een uitdrukking van de vorm

$$l = \lambda_1 \sum_{i,j} |w_{ij}| + \sum_m \mathcal{L}\left(\hat{y}_m; y_m\right)$$

of

$$l = \lambda_2 \sum_{i,j} w_{ij}^2 + \sum_m \mathcal{L}\left(\hat{y}_m; y_m\right)$$

Hierin dienen we te sommeren over alle gewichten i van alle neuronen j. De coëfficiënten λ_1 en λ_2 bepalen hoe zwaar gewichten ongelijk aan nul bestraft worden. Biases worden meestal niet bestraft. De reden is dat gewichten overeenkomen met verbindingen tussen

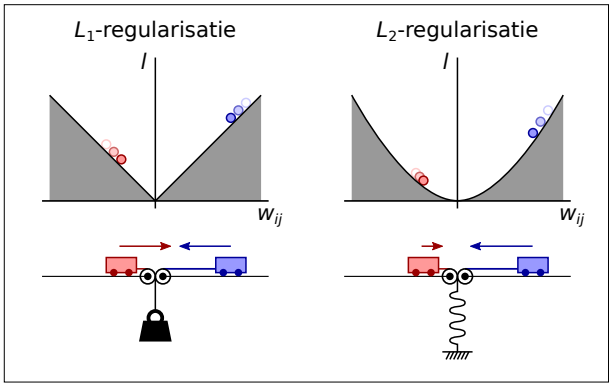
neuronen in het model, en als een verbinding een sterkte nul heeft kan deze effectief worden weggesnoeid waardoor het model eenvoudiger wordt; biases zijn meer een eigenschap van een neuron zelf, en deze snoei je niet weg.

Net als de loss-functie $\mathcal{L}$ dient de extra loss-term voor de gewichten strict positief te zijn, of eventueel nul als het gewicht nul is. In de eerste uitdrukking hierboven wordt dit bereikt door de absolute waarde te nemen van de gewichten. Deze aanpak wordt L_1-*regularisatie* genoemd, of *LASSO-regressie* als het een regressieprobleem betreft. In de tweede uitdrukking daaronder wordt voor kwadraten gekozen, hetgeen bekend staat als L_2-*regularisatie*, maar je kan ook de term *Tikhonov-regularisatie* tegenkomen of *ridge-regressie* in het geval van een regressieprobleem. De naamgeving komt van de L_1- en L_2-norm van een vector, waar we hier niet verder op ingaan, maar het cijfer 1 of 2 slaat op de macht waartoe een gewicht verheven wordt. Soms worden zowel de absolute waarden en de kwadraten tegelijkertijd gebruikt, elk met hun eigen λ-coëfficiënt, zodat

$$l = \lambda_1 \sum_{i,j} |w_{ij}| + \lambda_2 \sum_{i,j} w_{ij}^2 + \sum_m \mathcal{L}\left(\hat{y}_m; y_m\right)$$

Het resultaat wordt *elastic net regularisatie* genoemd. L_2-regularisatie is iets gangbaarder dan L_1-regularisatie, mede omdat het zich rekenkundig iets netter gedraagt doordat de kwadratische functie eenvoudig differentieerbaar is, maar de combinatie van beide is tegenwoordig steeds populairder.

De regularisatietermen kunnen denkbeeldig worden gezien als krachten die de gewichten w_{ij} naar nul trekken. L_1- en L_2-regularisatie gedragen zich in dit opzicht een beetje anders.



De absolute L_1-norm heeft altijd een even grote helling, hoe dicht je ook bij het minimum in de buurt komt. Hierdoor is ook de kracht waarmee deze de gewichten naar nul probeert te dwingen altijd even groot. Het is als het ware alsof een touwtje met een massa het gewicht naar beneden sleurt. De kwadratische L_2-norm zorgt echter voor een helling die geleidelijk afneemt naarmate je dichter bij het minimum in de buurt komt. Hierdoor is ook de kracht waarmee deze de gewichten naar nul trekt steeds kleiner naarmate de waarde dichter bij nul komt. Dit lijkt meer op een veer die steeds minder kracht uitoefent naarmate hij dichter bij de evenwichtsstand komt.

Hoe dan ook, het netto effect hiervan is dat L_2-regularisatie de gewichten wel in de buurt van nul probeert te trekken, maar deze nooit exact tot nul dwingt. Immers, hoe dichter een gewicht naar nul nadert, hoe zwakker de werking van de L_2-regularisatie wordt. L_1-regularisatie daarentegen probeert de gewichten exact op nul te krijgen. Beide methoden laten de gewichten in grootte krimpen, en worden dan ook wel *shrinkage methoden* genoemd; de term *weight decay* is ook gangbaar, met dezelfde betekenis. Gewichten gelijk aan nul leiden zoals gezegd tot een eenvoudiger interpreteerbaar model omdat er feitelijk minder werkzame verbindingen tussen neuronen aanwezig zijn. Zelfs al schakelt L_2-regularisatie verbindigen nooit helemaal uit, beide vormen van regularisatie blijken overfitting tegen te gaan.

Het verkleinen van de waarde van de gewichten w_{ij} gebeurt overigens niet koste wat het kost. Immers, ook de fouten in de uiteindelijke voorspellingen leveren een loss op zoals voorgeschreven door de loss-functie $\mathcal{L}$, en als de sterkte van neurale verbindingen wordt verkleind vanwege L_1- of L_2-regularisatie gaat dit meestal ten koste van de juistheid van de voorspellingen. De gewichten kunnen dus alleen in de buurt komen van nul als de winst die geboekt kan worden vanwege het minimaliseren van de L_1- of L_2-norm opweegt tegen een slechts geringe verslechtering in de voorspellingen van het model voor de trainingsdata. Of, omgekeerd, de gewichten worden niet sterk naar nul gedwongen als dit in een te grote afname van de kwaliteit van de voorspellingen zou resulteren. Met andere woorden, verbindingen tussen neuronen worden alleen verzwakt of verwijderd als het model die specifieke verbindingen blijkbaar toch al niet zo hard nodig had.

Desalniettemin zal het introduceren van L_1- en/of L_2-regularisatie leiden tot slechtere prestaties op de trainingsdata, die echter vanwege de vereenvoudiging van het model minder snel tot overfitting zal leiden. Hiermee gedragen L_1- en L_2-regularisatie zich allebei als regularisatiemethoden, zoals de naam al aangeeft.

Opgave 100. *

Wat gebeurt er als je weight-regularisation uitvoert met coëfficiënten λ gelijk aan nul?

Opgave 101. *

Leg uit dat de loss van gewichten volgens L_2-regularisatie wel differentieerbaar is, maar die volgens L_1-regularisatie niet.

Opgave 102. **

Een onderzoeker stel L_3-regularisatie voor volgens de formule $l = \lambda_3 \sum_{i,j} w_{ij}^3 + \sum_m \mathcal{L}(\hat{y}_m; y_m)$. Werkt dit? Zo ja, leg uit hoe; zo nee, pas de formule aan zodat het idee gehandhaafd blijft maar dit wel werkt.

Opgave 103. **

Stel, we voeren lineaire regressie uit met een kwadratisch model $y = w_2x^2 + w_1x + w_0$, maar we hebben slechts twee datapunten: $(x, y) = (-1, +1)$ en $(x, y) = (2, 4)$. Er zijn verschillende geschikte kandidaat-modellen, waaronder [i] $y = x + 2$, [ii] $y = x^2$, en [iii] $y = \frac{3}{4}x^2 + \frac{1}{4}x + \frac{1}{2}$. Ga na dat die alledrie exact door de twee punten heen gaan. Welk van deze drie modellen zou de beste oplossing zijn (dat wil zeggen, de laagste loss opleveren) als je L_1-regularisatie zou toepassen, welk als je L_2-regularisatie zou toepassen. En welk wint als je géén regularisatie zou toepassen?

Opgave 104. ***

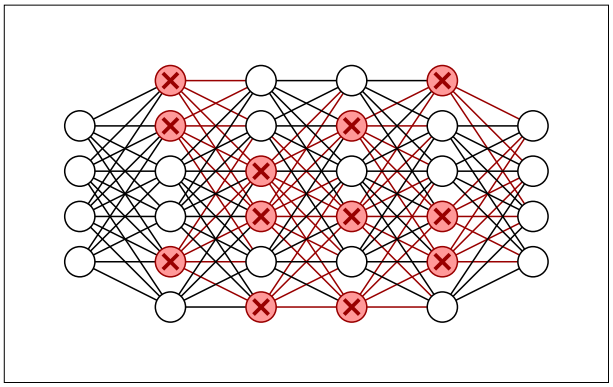
Stel we kijken alleen naar de loss-term van de L_2-regularisatie volgens $l = \lambda_2 \sum_{i,j} w_{ij}^2$. Laat zien dat dan de algemene updateregel van stochastic gradient descent $w_{ij} \leftarrow w_{ij} - \alpha \cdot \frac{\partial l}{\partial w_{ij}}$ leidt tot $w_{ij} \leftarrow \kappa w_{ij}$, waarbij $\kappa = 1 - 2\alpha\lambda_2$ typisch een getal is tussen nul en één. Oftewel, het effect van L_2-regularisatie in z'n eentje is dat de waarden van de gewichten tijdens elke trainingsstap worden gereduceerd met een constante vermenigvuldigingsfactor κ.

Opgave 105. ***

Stel we kijken alleen naar de loss-term van de L_1-regularisatie volgens $l = \lambda_1 \sum_{i,j} |w_{ij}|$. Laat zien dat dan de algemene updateregel van stochastic gradient descent $w_{ij} \leftarrow w_{ij} - \alpha \cdot \frac{\partial l}{\partial w_{ij}}$ leidt tot $|w_{ij}| \leftarrow |w_{ij}| - \kappa$, waarbij $\kappa = \alpha\lambda_1$ typisch een getal is tussen nul en één. Oftewel, het effect van L_1-regularisatie in z'n eentje is dat de grootten van de gewichten tijdens elke trainingsstap worden gereduceerd met een constante afname κ.

5.5. Dropout

De laatste regularisatiemethode die we bekijken in dit hoofdstuk is de minst intuïtieve. Deze is gebaseerd op het idee van ensemble learning. Dat wil zeggen, meerdere simpele machine learning modellen presteren tezamen vaak beter dan één ingewikkeld model. In het geval van neurale netwerken kunnen we opmerkelijk genoeg een simpeler model verkrijgen door willekeurig een aantal neuronen uit te schakelen. De output van deze neuronen wordt dan op nul gesteld: ze dragen niet bij tot de activatie van de neuronen in de volgende laag tijdens forward-propagation, en omdat ze geen invloed hebben op de uitkomst van de voorspelling worden ze ook niet bijgewerkt tijdens back-propagation. Echter, afhankelijk van welke neuronen we precies uitschakelen en welke we intact laten kunnen we met dit recept een heleboel verschillende simpelere modellen genereren. Sterker nog, het gehele neurale netwerk waarin al deze neuronen wél intact zijn kun je als het ware zien als een ensemble van al deze simpelere neurale netwerkjes die elk voor zich bijdragen tot de voorspelling.



Met dit in gedachten is *dropout* een methode waarbij tijdens het trainen van een model willekeurig een zekere fractie p van de neuronen in een laag wordt behouden en de overige

fractie $1-p$ wordt uitgeschakeld. Typisch wordt p gekozen tussen 0.5 en 0.8, waarbij dus een willekeurig gekozen 20 à 50% van de neuronen wordt uitgeschakeld. Andere waarden zijn ook mogelijk; er is vrij weinig bekend over wat in theorie het beste is en waar dit van afhangt.

Om te compenseren voor het wegvallen van een deel van de neuronen in de laag, wordt de uitvoerwaarde van alle andere neuronen vermenigvuldigd met een waarde $\frac{1}{p}$. In het geval $p = \frac{1}{2}$ bijvoorbeeld wordt de helft van de neuronen uitgeschakeld, maar wordt de uitvoer van de overlevende neuronen verdubbeld. Het idee achter deze *weight scaling rule* is dat de volgende laag in totaal evengoed "evenveel" invoer te verwerken krijgt. Hoewel er geen theoretisch bewijs is voor deze vuistregel blijkt dit in de praktijk goed te functioneren.

In elke stap van de training wordt een andere subset aan neuronen uitgeschakeld. De neuronen in de output layer worden vanzelfsprekend onaangeroerd gelaten omdat anders niet alle voorspellingen gedaan worden. De neuronen in de input layer kunnen eventueel deels worden uitgeschakeld, maar noodzakelijk is dit niet; $p = 0.8$ is hier redelijk gebruikelijk. Verschillende hidden layers kunnen zonder bezwaar een uiteenlopende dropout rate hebben.

Wanneer tenslotte echter het model toegepast wordt op nieuwe data, dat wil zeggen de test- of validatiedata, dan worden alle neuronen wél ingeschakeld en worden hun uitvoerwaarden niet vermenigvuldigd. Oftewel, tijdens het trainen train je telkens afzonderlijke eenvoudigere subnetwerken met minder actieve neuronen erin, maar bij het toepassen van het model gebruik je wel het hele ensemble van al die subnetwerken tezamen. Het resultaat blijkt robuustere voorspellingen te geven.

Deze procedure lijkt uitermate onintuïtief. Immers, waarom zou een netwerk dat je tijdens het trainen moedwillig toetakelt beter presteren dan een intact netwerk? Dat zou bijna betekenen dat iemand met een herseninfarct beter functioneert dan iemand met een gezond brein? Het is echter belangrijk om te realiseren dat je het model bij het toepassen op nieuwe data níet deels uitschakelt. Het is als het ware alsof je het neurale netwerk onder moeilijke omstandigheden traint om een taak uit te voeren. Dan is het wellicht niet zo vreemd dat het model het opeens verbazingwekkend goed doet als de omstandigheden niet meer moeilijk zijn zodra het ertoe doet. Vergelijk het met hoogtetraining van een atleet: de training vind plaats onder extra zware omstandigheden, maar daardoor presteer je beter als de omstandigheden normaal zijn, is het idee.

Dropout werkt omdat in een volledig neuraal netwerk neuronen soms heel delicaat op elkaar ingespeeld raken om precies de trainingsdata goed te verwerken. Dit samenspel genaamd *feature co-adaptation* blijkt niet altijd goed vertaalbaar naar nieuwe testdata. Dropout zorgt ervoor dat neuronen er niet altijd op kunnen vertrouwen dat andere neuronen in de laag ook aanwezig zijn. Daardoor worden neuronen gedwongen elk voor zich een uitvoer te produceren die erg robuust is. In zekere zin moeten de neuronen *redundante* informatie produceren, dat wil zeggen dat ze gedeeltelijk overlappende informatie doorgeven naar volgende lagen. En een dergelijke robuustere uitvoer blijkt een grotere kans te hebben om ook op nieuwe data goed te werken.

Een andere manier om tegen dropout aan te kijken is als het toevoegen van (multipli-

catieve) maskeerruis, aangezien de uitvoerwaarden van sommige neuronen willekeurig op nul gesteld worden. We hebben al eerder gezien bij data augmentatie dat het toevoegen van ruis een gunstig effect kan hebben.

Een variatie van dropout is *dropconnect*; daarbij worden niet de neuronen maar de verbindingen ertussen willekeurig uitgeschakeld. Oftewel, tijdens het trainen (maar niet tijdens het toepassen) van het model wordt een willekeurige fractie aan gewichten w_{ij} gelijk aan nul gesteld. Dit lijkt een beetje op het idee achter L_1- of L_2-regularisatie.

Opnieuw zorgt het uitschakelen van neuronen door middel van dropout ervoor dat de prestaties op de trainingsdata verminderen, maar het robuustere netwerk presteert dan vervolgens beter op de testdata. Daarmee kwalificeert het als een regularisatiemethode. Dropout heeft als grote voordelen dat het erg effectief kan zijn, eenvoudig te implementeren is, en zeer flexibel op allerlei soorten neurale netwerken kan worden toegepast.

Opgave 106. *

Waarom is het toevoegen van dropout niet zinvol als $p = 0$ of $p = 1$ gekozen wordt?

Opgave 107. **

In de figuur hierboven wordt een neuraal netwerk getoond met vier hidden layers met breedte zes waarbinnen per laag precies 50% van de neuronen wordt uitgeschakeld. Hoeveel verschillende "subnetwerken" van overlevende neuronen kunnen op die manier verkregen worden?

Opgave 108. **

Stel je hebt een model dat getraind wordt op tien attributen, en je past dropout toe op de input layer met een dropout rate van 90% (wat bizar hoog is, maar toch). Een collega beweert dat je daarmee eigenlijk het One-R algoritme krijgt. Immers, er is dan maar één input over die gebruikt kan worden. Ben je het hiermee eens?

Opgave 109. **

Je zou dropout kunnen toepassen door de pre-activatiewaarden van een willekeurige fractie neuronen gelijk aan nul te stellen, of door dit te doen met de post-activatiewaarden. Wat zou in jouw ogen beter werken, of maakt het vermoedelijk niet veel uit? Hangt je antwoord af van de gebruikte activatiefunctie?

6. Adaptive learning

Vooruitblik

In dit hoofdstuk bekijken we een aantal varianten op stochastic gradient descent die erop gericht zijn om de convergentie naar optimale modelparameters te bevorderen. We combineren hiertoe ideeën gerelateerd aan (mini)batch learning, learning rate decay, momentum, en schaling van gradiënten om uiteindelijk te komen tot de methode van adaptive moments estimation.

6.1. Minibatch learning

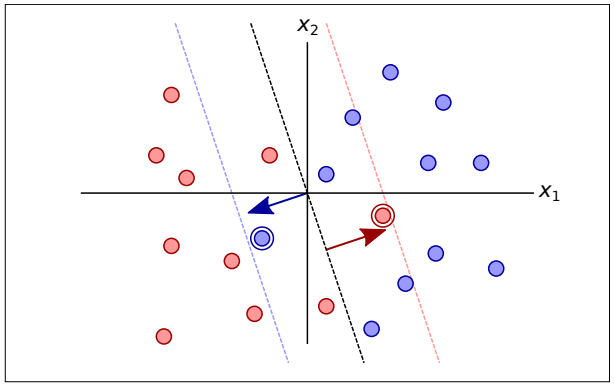
Tot dusverre trainen we het neurale netwerk elke keer met precies één instance tegelijkertijd. Het effect hiervan is dat je het model telkens aanleert om die ene instance goed te kunnen voorspellen. Door de instances snel en willekeurig af te wisselen hoop je dat het model bij een oplossing terecht komt die het voor alle instances redelijk goed doet.

Deze benadering werkt in de praktijk behoorlijk goed. Ook kan worden aangetoond dat voor een voldoende kleine learning rate α dit gegarandeerd leidt tot een optimum. Als je α heel klein kiest werk je de modelparameters immers maar heel weinig bij. Het effect daarvan is dat het model goed in staat is om de geleerde gewichten op grond van de instances die het eerder gezien heeft te "onthouden". Op die manier onthoudt het model effectief alle instances, en optimaliseert het dus ook de prestaties voor alle instances.

Het nadeel van een lage α is natuurlijk echter dat het model slechts heel langzaam leert. Als de learning rate groter wordt gekozen kan het model weliswaar sneller convergeren naar een lage waarde voor de loss l, maar deze is dan voornamelijk gebaseerd op de laatste instances waarmee het model getraind is. Eerdere instances worden dan snel weer "vergeten" omdat de gewichten al heel gauw flink veranderen. Een goede prestatie voor de ene instance hoeft niet ook een goede prestatie voor een andere instance te betekenen. Omdat bij hoge α vooral de meest recente trainingsinstances de uitkomst bepalen en het model dus vooral die instances correct leert voorspellen, maakt dit het soms moeilijk om een model te trainen dat het goed doet voor álle data.

Bekijk bijvoorbeeld eens de figuur hieronder. Hier wordt een situatie getoond waarin op enig moment een model zoals logistische regressie is getraind om de zwart gestippelde scheidingslijn middenin te trekken. Er worden dan twee instances verkeerd geclassificeerd, een blauwe en een rode, beide omcirkeld. Zodra het model op de blauwe misgeclassificeerde instance wordt getraind zal de scheidingslijn naar links worden getrokken. Het hangt ervan af hoe groot de learning rate is hoe ver de lijn verschuift, maar een model dat deze ene instance goed classificeert zal het op de dataset als geheel waarschijnlijk slecht

doen omdat een boel andere instances dan aan de verkeerde kant van de lijn komen te liggen.



Zodra even later het model op de rode omcirkelde instance wordt getraind zal de scheidingslijn terug naar rechts worden getrokken. Opnieuw leidt dat tot een verbetering van het model voor die ene instance, maar kunnen andere instances daardoor verslechteren. Als je telkens maar op één instance tegelijk traint wordt de scheidingslijn numeriek heen en weer bewogen. De oplossing heeft dan sterk de neiging om heen en weer te springen: op zeker moment presteert je model misschien goed op de ene instance maar niet de andere, en even later presteert het dan weer goed op die andere instance maar niet meer op die ene. Je komt dan weliswaar ergens in de buurt van een redelijk model, maar op welk moment je ook stopt zul je niet noodzakelijkerwijs een optimaal model vinden voor alle instances samen.

Dit kan worden opgelost door niet om en om te trainen op één individuele instance, maar op de hele dataset tegelijkertijd. Dit wordt ook wel *batch learning* genoemd omdat je je baseert op een hele trainingsbatch met instances. Elke iteratie van het trainingsproces komt dan in één keer overeen met een epoch. Wanneer je traint op één instance per keer noem je dat *online* learning; dit is soms noodzakelijk als de trainingsinstances één voor één gegenereerd worden en je elke keer onmiddellijk je model bij wil werken.

Tot nu toe probeerden we de loss $l = \sum_m \mathcal{L}(\hat{y}_m; y_m)$ van een enkele instance te minimaliseren. De sommatie m liep hier over de output neuronen van het model. Beter lijkt het nu om te kijken naar de gemiddelde loss van alle instances gezamenlijk. Deze wordt ook wel de *cost* genoemd. Overigens worden de termen loss en cost vaak door elkaar gebruikt. Wij hanteren hier de definitie dat de loss beschrijft hoe uit $\hat{y}_m$ en y_m kan worden berekend hoe slecht de voorspelling van één instance is; de cost beschrijft wat het criterium is dat wordt geminimaliseerd, meestal de gemiddelde of totale loss over vele instances in de trainingsdata (soms met inbegrip van extra termen, zoals voor L_1- of L_2-regularisatie uit het vorige hoofdstuk). Tot dusverre waren die twee begrippen identiek, maar vanaf nu lopen ze uiteen.

We kunnen hier stellen dat de cost berekend wordt als

$$J = \frac{1}{N} \sum_{n=1}^{N} l_n$$

waarbij we middelen over de trainingsinstances genummerd $n = 1$ tot en met N. De stochastic gradient descent regel voor een gewicht w_{ij} wordt dan lichtjes aangepast tot

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial J}{\partial w_{ij}}$$

of een vergelijkbare uitdrukking voor de bias b_j. De gradiënt van de cost is gelijk aan de gradiënten van de losses, gemiddeld over alle instances.

Overigens wordt de cost ook vaak gelijk gesteld aan de totale loss over alle instances, dat wil zeggen de som in plaats van het gemiddelde. Dit scheelt slechts de vermenigvuldigingsfactor $\frac{1}{N}$. Dit leidt niet tot een andere oplossing: het minimum voor de een is ook een minimum voor de ander, hoewel de minima niet even "diep" zijn. Wel heeft het effect op de optimale grootte van de learning rate die gekozen dient te worden.

Wanneer je bij elke update van je model alle instances moet doorrekenen leidt dit tot erg veel rekenwerk. Voor grote datasets kan dit nogal uit de klauwen lopen. Vandaar dat in de praktijk een tussenoplossing wordt gekozen die alle voordelen combineert. De dataset wordt dan opgedeeld in kleinere *minibatches* die wel zoveel instances bevatten dat ze voldoende representatief zijn voor de hele dataset, maar toch ook weer niet zoveel dat er vrijwel identieke instances als het ware "dubbelop" in voorkomen. Hoe groot deze minibatches moeten zijn hangt ervan af hoeveel trainingsdata er beschikbaar zijn en hoe ingewikkeld het model en de verdeling van de data zijn, maar als vuistregel worden vaak enkele tientallen tot honderden instances genomen.

Wanneer kortweg gesproken wordt over batch learning wordt soms ook wel eens minibatch learning bedoeld. Het woordgebruik hieromtrent is niet altijd consequent. Om dit onderscheid expliciet te maken kun je spreken van *full batch learning* als je alle instances tezamen gebruikt. Het woord minibatch slaat wel altijd op slechts een deel van de instances. Kies je de grootte van de minibatch gelijk aan 1 dan verkrijg je online learning.

Bij minibatch learning of online learning maak je gebruik van stochastic gradient descent omdat er een toevalsfactor bij komt kijken die bepaalt op welke deel van de instances per keer getraind wordt: de instances in de minibatch worden willekeurig getrokken uit de trainingsdata, dus de cost die wordt geoptimaliseerd is ook in elke minibatch een klein beetje anders. Bij full batch learning kun je spreken van traditionele gradient descent omdat dan de te optimaliseren cost in alle iteraties identiek blijft.

Minibatch learning heeft ook een paar andere praktische voordelen. Enerzijds wordt het model telkens getraind met net een iets andere verzameling van instances. Vergeleken met (full) batch learning heeft dit als voordeel dat de methode wat ongevoeliger wordt voor overfitting. Bij overfitting wordt het model te sterk aangepast aan de toevallige waarden van de gebruikte trainingsdataset; door de trainingsdata te verdelen in steeds weer andere deelverzamelingen wordt overfitting verminderd. Dit lijkt een beetje op het toevoegen van ruis aan de trainingsdata bij data augmentatie. Anderzijds geeft minibatch learning de mogelijkheid om gebruik te maken van parallellisatie van de berekening. Vergeleken met online learning wordt hetzelfde model immers parallel en identiek doorgerekend voor diverse instances, en dit leent zich goed voor speciale hardware, zoals

een GPU. Dit kan het doorrekenen en optimaliseren van neurale netwerken behoorlijk versnellen.

Opgave 110. *
Leg uit dat je online en batch learning allebei kunt zien als speciale gevallen van minibatch learning.

Opgave 111. *
Waarom kunnen opeenvolgende instances bij online learning niet parallel worden doorgerekend?

Opgave 112. **
Stel dat een optimale learning rate $\alpha = 0.1$ zou gelden als je de cost definieert als de *gemiddelde* loss over 100 instances in een minibatch. Hoe zou je in dezelfde situatie de learning rate moeten kiezen als je de cost had gedefinieerd als de *totale* gesommeerde loss over een minibatch van 100 instances?

Opgave 113. ***
Schets een voorbeeld van een dataset met twee numerieke attributen en één nominale uitkomst die prima met kleine minibatches getraind kan worden, en een andere die grote minibatches zal vereisen. Hint: de distributie van data in een minibatch dient representatief te zijn voor de distributie van data in de hele dataset; hoe hangt de haalbaarheid hiervan af van de complexiteit van de vorm van de ware scheidingsgrens tussen de twee klassen?

6.2. Learning rate decay

We hebben het zojuist al even gehad over de learning rate α. Deze belangrijke parameter bepaalt de convergentie naar een optimale oplossing. Wordt deze te groot genomen dan kan het zijn dat het model heen en weer blijft springen en nooit convergeert naar een redelijke oplossing; wordt deze te klein gekozen dan kan het heel lang duren voordat een goede oplossing gevonden wordt. Je kan dit diagnostiseren aan de hand van de validatiecurve: in het eerste geval zal de validatiecurve grillig op en neer springen; in het laatste geval zal deze slechts heel langzaam verbetering vertonen. In de praktijk kunnen beide situaties ertoe leiden dat je niet tot een geschikt model komt.

Vaak is het zinvol om aanvankelijk de learning rate behoorlijk groot te kiezen. Dit zorgt ervoor dat je vrij vlot in elk geval tot een redelijke oplossing kan komen. Daarna kan de learning rate dan worden verlaagd om te zorgen dat je niet om een oplossing heen blijft springen maar er geleidelijk naartoe convergeert. Een eenvoudige manier om hiermee om te gaan is door de learning rate handmatig bij te stellen. Je zou kunnen instellen dat je eerst een aantal epochs met hoge α traint en daarna epochs met telkens lagere waarden kiest. Je zou dit ook interactief kunnen doen door bijvoorbeeld "live" de cost van het actuele model te plotten en dan de learning rate met de hand aanpasbaar te maken. Je kan dan direct zien of het model verder verbetert door de learning rate te verkleinen.

De learning rate kan ook volgens een vast functievoorschrift geleidelijk worden verkleind. Als het aantal gedraaide epochs wordt aangeduid met t dan zijn de meest gebruikte mogelijkheden de volgende.

- Lineaire afname: $\alpha = \max(\alpha_0 - \beta \cdot t, \alpha_1)$. De waarde van α begint dan bij α_0, neemt elke epoch met een constante hoeveelheid β af, totdat een ondergrens α_1 is bereikt die dan wordt gehandhaafd.
- Exponentiële afname: $\alpha = \alpha_0 \cdot e^{-\beta \cdot t}$. De waarde begint wederom bij α_0, maar neemt elke epoch met een constante factor gerelateerd aan β af.
- Inverse afname: $\alpha = \frac{\alpha_0}{1+\beta \cdot t}$. De waarde begint bij α_0, en elke epoch wordt deze door een met β toenemend groter getal gedeeld.

Voor al deze varianten geldt dat hoe hoger β, hoe sneller de learning rate afneemt.

Tenslotte is het ook mogelijk te kiezen voor een getrapte afname. Bijvoorbeeld door elke tien epochs de learning rate te halveren, of iets dergelijks. Iets geavanceerdere methoden laten de learning rate afnemen wanneer bijvoorbeeld een minimum in de validatiecurve wordt bereikt en de cost op de validatiedata niet meer afneemt. Dat is immers het moment dat het model niet meer verder generaliseert maar dreigt te gaan overfitten.

De genoemde methoden hebben het voordeel dat ze relatief eenvoudig te implementeren zijn. Hierna zullen we echter een aantal *adaptieve* methoden bekijken die minder gevoelig zijn voor de precieze waarde van α, of die de grootte van de aanpassingen aan de modelparameters gaandeweg bepalen.

Opgave 114. *

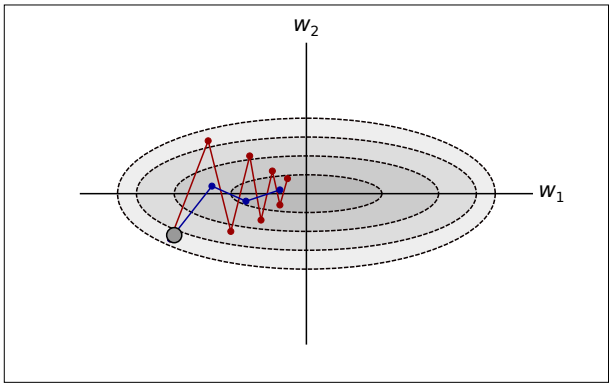
Orden de drie strategieën van afname (lineair, exponentieel, invers) van de meest langzame afname tot de meest snelle afname. Kijk hierbij naar het gedrag van de afname op de lange termijn, ongeacht de precieze waarde van de parameter β.

Opgave 115. **

Iemand suggereert de volgende strategie om de learning rate aan te passen: begonnen wordt met een zekere waarde α_0, en elke keer wanneer het teken van de verandering in een te optimaliseren modelparameter verandert (dus een positieve update in de vorige iteratie wordt gevolgd door een negatieve update in de volgende iteratie, of omgekeerd) wordt de learning rate verkleind door deze te delen door een zekere factor $\beta > 1$. Wat vind je van deze suggestie?

6.3. (Nesterov) momentum

In de onderstaande figuur staat een fenomeen geïllustreerd wat relatief vaak voorkomt. De cost J hangt hier af van twee gewichten w_1 en w_2. Dit is hier getekend in de vorm van gestippelde hoogtelijnen, zoals je die met een *contour plot* maakt. In de oorsprong van het assenstelsel behaalt de cost een minimum, aangeduid door grijstinten. Dit is in dit voorbeeld dus de optimale oplossing. Daaromheen neemt de cost geleidelijk toe. In dit geval neemt de cost sterker toe als functie van w_2 langs de y-as dan als functie van w_1 langs de x-as. De cost vormt als het ware een langgerekt dal in horizontale richting.



Gradient descent dicteert dat de stapgrootte evenredig is met de helling van de cost. In dit geval is de helling langs de horizontale richting kleiner dan die langs de vertikale richting. De stappen waarmee de gewichten worden bijgewerkt zullen dus ook groter zijn voor w_2 dan voor w_1. De gradiënt staat altijd exact loodrecht op de hoogtelijnen en is groter naarmate de hoogtelijnen dichter op elkaar lopen. Hetzelfde geldt voor de stappen.

Stel, de gewichten hebben op een zeker moment waarden die overeenkomen met de grijze stip. Dan zal, om de helling af te gaan, dus een stap gezet worden die groot is naar boven en klein is naar rechts. Voor de volgende stappen geldt elke keer iets vergelijkbaars. Deze volgen dan ongeveer de rode zigzaglijn. In deze illustratie kom je telkens helemaal aan de andere kant van het dal terecht en ben je tegelijkertijd een beetje opgeschoven naar rechts. In dit geval nadert de serie rode stappen volgens gradient descent weliswaar het optimum, maar de zwalkende manier waarop is verre van ideaal.

Afhankelijk van de grootte van de learning rate α kan de stapgrootte iets te groot zijn waardoor je over de ideale waarde heenschiet of te klein waardoor je niet ver genoeg komt. Wat het geval is kan echter per modelparameter verschillen, zien we nu. Zo is hierboven voor w_1 de learning rate eigenlijk te klein, want je gaat niet zover naar rechts als je wel zou willen om in het optimum te belanden. Maar tegelijkertijd is voor w_2 de learning rate te groot, want je schiet in vertikale richting helemaal door het dal heen om aan de andere kant van het laagste punt te belanden. Het is dus onmogelijk een learning rate te kiezen die alle parameters even goed bijwerkt.

Een manier om dit gedrag te verbeteren is door niet alleen te kijken naar wat de gradiënt op een zeker moment is, maar ook naar wat de gradiënt de afgelopen stappen was. Zou je bijvoorbeeld naar het gemiddelde kijken over meerdere stappen, dan krijg je meteen al een veel beter gedrag. Immers, de gradiënten in de vertikale richting wisselen steeds af van teken. Als je die middelt dan werken ze tegen elkaar in en blijft er een kleine stapgrootte over. De gradiënten in de horizontale richting daarentegen wijzen allemaal dezelfde kant op. Als je deze middelt dan blijven ze dus aanzienlijk. Als je stappen zou zetten die gelijk zijn aan het gemiddelde van de afgelopen gradiënten vermindert het zig-zag patroon sterk. Je kan dan veel sneller convergeren naar het optimum, zoals getoond met de blauwe lijn.

Je kan dit voorstellen als een zekere massa die aan de stip wordt gehangen. Als die eenmaal in beweging is verandert die niet zomaar weer. In vertikale richting werkt

de gradiëntkracht op het punt steeds andersom, waardoor de beweging uitmiddelt. In horizontale richting werkt de gradiëntkracht telkens dezelfde kant op, waardoor deze beweging relatief wordt versterkt. De uitmiddelende werking van een dergelijke massa wordt ook wel *momentum* genoemd.

De traditionele manier waarop gradient descent werkt is dat er voor elke parameter w een gradiënt $G = \frac{\partial J}{\partial w}$ wordt bepaald en dat vervolgens de parameter wordt bijgewerkt volgens de regel

$$w \leftarrow w - \alpha G$$

Dit is nog steeds gewoon de updateregel volgens gradient descent, alleen slaan we de gradiënt nu op in een speciale variabele G. Wanneer we gebruik maken van momentum moeten we niet bijwerken met de gradiënt, maar met een gewogen voortschrijdend gemiddelde of *moving average* daarvan. We bereiken dit door G zelf geleidelijk bij te werken volgens

$$G \leftarrow \beta G + \frac{\partial J}{\partial w}$$

De coëfficiënt $0 < \beta < 1$ bepaalt in dit geval hoe lang de eerdere gradiënten blijven meewegen. Dit komt als het ware overeen met de massa van de zwarte stip. Kiezen we β nabij 0, dan krijgen we precies onze "oude" gradient descent updateregel terug. De massa is dan nul. Nadert β naar 1 dan wordt G steeds sterker bepaald door de voorafgaande gradiënten in plaats van door de huidige gradiënt $\frac{\partial J}{\partial w}$. Dit helpt om een steeds grotere snelheid in de juiste richting op te bouwen. De massa is dan hoog. De coëfficiënt β wordt ook wel de *momentum parameter* of *frictie parameter* genoemd. De term frictie duidt erop dat kleine waarden van β een afremmende werking hebben en lijken op wrijving. Een typische waarde voor β is 0.9. Dit betekent losjes gesproken dat je de stap voor 90% laat afhangen van de voorafgaande gradiënten en voor de resterende 10% van de huidige nieuwe gradiënt.

Met behulp van momentum kan het leerproces versnellen omdat de voorkeur wordt gegeven aan richtingen die consistent zijn over meerdere iteraties heen, terwijl nutteloze zijwaartse oscillaties worden gedempt. Het leidt er echter ook toe dat je over de optimale oplossing heen kan schieten zodra je die bereikt hebt, omdat de opgebouwde snelheid blijft doorwerken. Vergelijk dit met een knikker in een ronde kom die naar beneden rolt en dan doorschiet. Echter, de winst die je kan bereiken door sneller naar het optimum te convergeren compenseert gewoonlijk ruimschoots voor de extra tijd die je nodig hebt om rond het optimum tot rust te komen. Het hebben van een zekere *overshoot* kan zelfs gunstige gevolgen hebben als het bereikte minimum slechts een lokaal minimum blijkt te zijn: de opgebouwde impuls kan dan voldoende zijn om door het lokale minimum heen te schieten om daarna verderop wellicht in een beter minimum te belanden.

Het gebruik van momentum heeft nog een extra voordeel. Omdat je continu een schatting G bijhoudt van de gradiënt en die geleidelijk bijwerkt, weet je tijdens de forward-propagation fase al redelijk hoe de nieuwe gewichten er uit gaan zien, zelfs als die nog niet zijn bijgewerkt voor de lopende minibatch. Immers, je weet dat je de updateregel $w \leftarrow w - \alpha G$ zal gaan toepassen, en hoewel je nog niet exact weet wat de waarde van de nieuwe gradiënt G gaat zijn, is de huidige waarde daarvan al best een aardige

schatting. Je kunt dus tijdens de forward-propagation alvast een voorschot nemen op de update, en werken met de gewichten zoals je die zou gaan krijgen met de huidige waarde van G. Vervolgens bereken je de gradiënt $\frac{\partial J}{\partial w}$ voor de huidige minibatch, uitgaande van die schatting van de nieuwe gewichten. Daarmee werk je dan de gradiënt bij volgens $G \leftarrow \beta G + \frac{\partial J}{\partial w}$, en pas dan update je de gewichten definitief met behulp van die nieuwe waarde van G. Het gevolg is dat je de gewichten kan bijwerken met een nauwkeurigere nieuwe schatting van de gradiënt. Je hebt als het ware al vooruit kunnen kijken naar waar je terecht gaat komen. Hierdoor kun je bijvoorbeeld al beginnen "af te remmen" nog vóórdat je in een minimum bent beland, waardoor je er dus minder sterk doorheen zal schieten. Het idee om tijdens de forward propagation alvast gebruik te maken van de bestaande gemiddelde gradiënt om een betere schatting te krijgen van de nieuwe gradiënt wordt *Nesterov momentum* of *Nesterov accelerated gradient* genoemd.

Opgave 116. *

Er werd gezegd dat een voordeel van momentum is dat je door een lokaal minimum heen kan rollen om elders in een dieper minimum te belanden. Echter, omgekeerd zou je natuurlijk ook door een globaal minimum heen kunnen rollen en in een ondieper minimum kunnen blijven steken. Bedenk waarom het netto meestal toch voordelig uitwerkt.

Opgave 117. **

Leg uit waarom het idee van Nesterov om vooruit te kijken alleen gebruikt kan worden als je gebruik maakt van momentum, maar niet als je gebruik maakt van "gewone" (stochastic) gradient descent.

Opgave 118. ***

Stel, de update zou worden uitgevoerd volgens $G \leftarrow \beta G + (1 - \beta) \frac{\partial J}{\partial w}$? Wat is hiervan de invloed op de werking van het algoritme? Hoe zou je de waarde van α kunnen aanpassen zodat de uitkomsten identiek zouden worden als bij de update regels eerder hierboven?

6.4. Gradiënten schalen

In de vorige paragraaf werd beschreven hoe door de gradiënten te middelen over meerdere iteraties de convergentie kon worden verbeterd. Dit doet niet af aan de mogelijk langzame convergentie die kan optreden wanneer de activatie- of lossfuncties dreigen te satureren. In hoofdstuk 4 hebben we immers gezien dat weliswaar de saturatie van de softmax-functie kan worden tegengegaan door een cross-entropy lossfunctie te gebruiken, maar saturerende activatiefuncties in eerdere hidden layers kunnen nog steeds optreden en leiden tot stagnerende leerprestaties van het neurale netwerk. Het kan in zo'n geval gebeuren dat een parameter weliswaar consistente stappen maakt in de juiste richting, maar dat die stappen slechts heel klein zijn. Het middelen van gradiënten met momentum werkt dan niet: het gemiddelde van zwakke gradiënten blijft een zwakke gradiënt. Een ander idee is daarom om de gradiënt te schalen. Als de grootte van de gradiënt systematisch klein dreigt te worden dienen we de stapgrootte op te schalen.

Een methode die dit idee handen en voeten geeft heet *RMSProp*, wat staat voor *root-mean-square propagation*. Hierbij wordt een variabele bijgehouden die een idee geeft van de grootte van de gradiënt en die wij hier Q zullen noemen. Dit wordt bereikt door een lopend gemiddelde te nemen van de kwadraten van de gradiënt, volgens

$$Q \leftarrow \beta Q + (1-\beta)\left(\frac{\partial J}{\partial w}\right)^2$$

Hier komen waarden β en $(1-\beta)$ tussen 0 en 1 in voor die zorgen dat voorafgaande iteraties weliswaar worden meegewogen maar ook weer geleidelijk worden vergeten. Opnieuw geeft de *decay parameter* β aan hoe sterk voorafgaande iteraties meetellen. Dit is vergelijkbaar met momentum, zij het dat niet de gradiënt wordt uitgemiddeld maar diens kwadraat.

Hoe groter de gradiënten zijn, ongeacht hun teken, hoe sterker de waarde van Q zal oplopen. Q geeft dus een idee van de sterkte van alle gradiënten tot nu toe, zonder naar hun richting te kijken. De formule die wordt gehanteerd om de modelparameters bij te werken luidt vervolgens

$$w \leftarrow w - \frac{\alpha}{\sqrt{Q+\epsilon}}\frac{\partial J}{\partial w}$$

Hierbij is in de noemer de schaalfactor ongeveer gelijk gekozen aan $\sqrt{Q}$. Immers, Q houdt de kwadraten van de gradiënt bij, dus om iets over de grootte van de gradiënt zelf te kunnen zeggen dien je hiervan de wortel te nemen. Verder is er een constante ϵ toegevoegd om problemen met delen door nul te vermijden. Deze wordt uiterst klein gekozen, bijvoorbeeld 10^{-6} of 10^{-9}, zodat deze meestal geen merkbare invloed heeft.

Als er nu bijvoorbeeld al vrij snel saturatie optreedt en de gradiënt blijft maar klein voor een bepaalde parameter, dan zal ook $\sqrt{Q}$ een betrekkelijk kleine waarde houden. In de updateregel wordt gedeeld door deze kleine waarde, dus dat betekent dat de stapgrootte in dat geval naar verhouding groot kan blijven. Het effect van de bovenstaande formule is daardoor dat het er voor de stapgroottes eigenlijk niet meer zoveel toe doet hoe groot de gradiënten precies zijn: gekeken wordt vooral naar de grootte van de gradiënt in verhouding tot de gradiënten zoals die tot dan toe zijn opgetreden. Hoe groter die verhouding, hoe groter de stappen. Dat betekent dat een modelparameter die satureert en daardoor nog slechts hele kleine stapjes dreigt te maken daardoor minder wordt beïnvloed. Immers, weliswaar wordt de gradiënt dan klein, maar omdat de gradiënt dan ook in het verleden klein was kan in verhouding toch een grote stap worden gezet.

Opgave 119. **

Is het ook redelijk om de update uit te voeren volgens $w \leftarrow w - \frac{\alpha}{\epsilon+\sqrt{Q}}\frac{\partial J}{\partial w}$, denk je? (Let op de verschillende positie van de constante ϵ.) Waarom werkt dit wel/niet?

Opgave 120. **

Stel dat voor een bepaalde modelparameter de grootte en richting van de gradiënt constant zijn. Met andere woorden, in alle iteraties blijkt de gradiënt $\frac{\partial J}{\partial w}$ exact dezelfde

waarde te hebben. Hoe gedraagt zich dan op de lange termijn de stapgrootte volgens de updateregel van RMSProp? Neigt die naar een constante, en zo ja hoe groot is dan die constante, of wordt die steeds groter of kleiner? Wat vertelt je dit over de gevoeligheid voor saturatie?

6.5. Adaptive moments

Tenslotte kunnen de ideeën van momentum en RMSProp worden gecombineerd door zowel een lopend gemiddelde voor de gradiënt als voor diens kwadraat bij te houden. We krijgen dan een lopend gemiddelde voor de gradiënt volgens

$$G \leftarrow \beta_1 G + (1 - \beta_1) \frac{\partial J}{\partial w}$$

en een lopend gemiddelde voor het kwadraat van de gradiënt volgens

$$Q \leftarrow \beta_2 Q + (1 - \beta_2) \left(\frac{\partial J}{\partial w} \right)^2$$

De updateregel wordt daarmee

$$w \leftarrow w - \frac{\alpha G}{\sqrt{Q + \epsilon}}$$

De twee decay parameters β_1 en β_2 geven aan hoe zwaar voorafgaande iteraties worden meegewogen voor respectievelijk de gradiënt zelf en diens kwadraat. Deze hoeven niet noodzakelijk gelijk aan elkaar gekozen te worden. In de praktijk wordt bijvoorbeeld vaak $\beta_1 = 0.9$ en $\beta_2 = 0.999$ genomen. Dit betekent dat de richting van de gradiënt in staat is om zich sneller aan te passen dan de schaling met de diens grootte.

Een probleem dat bij deze methode kan optreden is gerelateerd aan de initialisatie van de parameters G en Q. Die worden normaal aanvankelijk gelijk aan nul gesteld. Maar dat betekent dat in de eerste paar updates de gradiënt te klein of te groot genomen wordt omdat de nulwaarde nog even door blijft werken. Er wordt dan ook wel gesproken van een *bias*: de gemiddelde gradiënt en diens grootte worden aanvankelijk beide systematisch onderschat. Dit kan met name vervelende gevolgen hebben voor de bovenstaande methode omdat hierin G en Q deze nulwaarde met een ander tempo laten uitdoven. Om dit te voorkomen kun je *unbiased* waarden voor G en Q berekenen volgens

$$G' = \frac{G}{1 - \beta_1^t}$$

en

$$Q' = \frac{Q}{1 - \beta_2^t}$$

In de noemer worden de wegingsfactoren tot de t-de macht verheven, waarbij t het aantal uitgevoerde iteraties is, dat wil zeggen het aantal minibatches. Bij de eerste iteratie is $t = 1$. Deze correctie heft precies het effect van de aanvankelijke nulwaarden op.

In de updateregel moeten dan natuurlijk deze unbiased waarden worden verwerkt

$$w \leftarrow w - \frac{\alpha G'}{\sqrt{Q' + \epsilon}}$$

De methode die nu is verkregen staat bekend als *Adam*, wat staat voor *adaptive moments estimation*. Deze methode is zeer populair omdat ie eigenlijk alle ingrediënten van van de eerdere methoden in zich bergt, en tegelijkertijd toch nog (betrekkelijk) eenvoudig te implementeren is.

Het is tenslotte mogelijk om hier ook nog het idee van Nesterov momentum aan toe te voegen, waarbij je tijdens de forward-propagation alvast een voorschot neemt op de update die je zal gaan maken in de gewichten. Immers, ook bij Adam houd je waarden voor G en Q bij die slechts langzaam zullen veranderen en dus redelijk voorspelbaar zijn. Het resultaat wordt *Nadam* genoemd, maar deze methode lijkt niet populairder te zijn dan Adam zelf.

Overigens bestaan er nog diverse andere varianten. Het voert te ver om die hier allemaal uiteen te zetten.

Opgave 121. **

Als je de beide frictieparameters gelijk aan nul stelt, zodat $\beta_1 = 0$ en $\beta_2 = 0$, hoe groot is dan in elke iteratie de stapgrootte volgens Adam? Je mag de constante ϵ verwaarlozen.

Opgave 122. **

Stel, je kiest $\beta_1 = \beta_2$ bij Adam. Zal dan de waarde van Q gelijk zijn en blijven aan het kwadraat van G? Oftewel, is een (gewogen) gemiddelde van kwadraten hetzelfde als het kwadraat van een (gewogen) gemiddelde?

Opgave 123. **

Toon aan dat ongeacht de precieze vorm van de cost na de eerste iteratie $G' = \frac{\partial J}{\partial w}$ en $Q' = \left(\frac{\partial J}{\partial w}\right)^2$. Aannemende dat de constante ϵ verwaarloosbaar klein is, hoe groot is dan de verandering in de modelparameter w?

Opgave 124. **

Hoe moeten de parameters β_1 en β_2 gekozen worden om Adam te reduceren tot RMSProp?

Opgave 125. ***

Leg uit dat je zowel een bias kunt hebben richting te grote updates als richting te kleine updates. Hoe hangt dit af van de parameters β_1 en β_2?

Opgave 126. ***

Is het mogelijk om met een geschikte keuze van de parameters β_1 en β_2 hetzelfde gedrag te krijgen als met momentum alleen? Leg je antwoord uit.

Appendix A

Oplossingen

Antwoorden op een selectie van opgaven worden hieronder gegeven.

Oplossing 1. -

Oplossing 2. Overfitting is te onderscheiden van underfitting aan de hand van een disproportioneel lage resubstitution error; over- en underfitting hebben beide een relatief grote cross-validation error.

Oplossing 3. -

Oplossing 4. One-R, Naive Bayes, en logistische regressie vormen geen universele approximator omdat het concept beperkt is tot classificatiegrenzen die recht zijn (One-R en logistische regressie) of rond (Naive Bayes), maar bijvoorbeeld geen willekeurige kronkelige vormen kunnen aannemen. Een (J48-)tree en k-nearest neigbor classificatie kan classificatiegrenzen produceren die willekeurig fijn gedetailleerd zijn, mits er voldoende data is; deze kunnen daardoor in theorie daadwerkelijk elke verdeling fitten en vormen een universele approximator.

Oplossing 5. -

Oplossing 6. $\hat{y}_A = \text{sgn}\,(1 \cdot 0 + -2 \cdot 1 + 3 \cdot 2) = \text{sgn}\,(4) = +1$; $\hat{y}_B = \text{sgn}\left(1 \cdot \frac{1}{2} + -2 \cdot \frac{1}{3} + 3 \cdot \frac{1}{4}\right) = \text{sgn}\left(\frac{7}{12}\right) = +1$; een voorbeeld met $\hat{y}_C = 0$ is $\boldsymbol{x}_C = [1, 2, 1]$, maar er zijn vele andere voorbeelden.

Oplossing 7. Het is onjuist dat $\text{sgn}\,(a + b) = \text{sgn}\,(a) + \text{sgn}\,(b)$, zoals bv. blijkt door te kiezen $a = 1$ en $b = -2$; het is wel juist dat $\text{sgn}\,(ab) = \text{sgn}\,(a) \cdot \text{sgn}\,(b)$, hetgeen kan worden gecontroleerd door te zien wat er gebeurt voor elk van de negen combinaties waarbij a en b negatief, nul, of positief zijn.

Oplossing 8. Als $x > 0$ is $|x| = x$ zodat $\frac{|x|}{x} = \frac{x}{x} = +1$; als $x < 0$ is $|x| = -x$ zodat $\frac{|x|}{x} = \frac{-x}{x} = -1$. Dit komt overeen met de definitie van $\text{sgn}\,(x)$.

Oplossing 9. -

Oplossing 10. De voorspelling op basis van $\boldsymbol{w} = [1, -4, 5]$ luidt $\hat{y} = \text{sgn}\left(1 \cdot 0 + -4 \cdot \frac{1}{2} + 5 \cdot 1\right) = \text{sgn}\,(3) = +1$, zodat $\hat{y} - y = +2$. De updateregel levert dan $\boldsymbol{w} \leftarrow [1, -4, 5] - 2 \cdot \left[0, \frac{1}{2}, 1\right] = [1, -5, 3]$. Vervolgens is $\hat{y} = \text{sgn}\left(1 \cdot 0 + -5 \cdot \frac{1}{2} + 3 \cdot 1\right) = \text{sgn}\left(\frac{1}{2}\right) = +1$, zodat nogmaals $\hat{y} - y = +2$ en $\boldsymbol{w} \leftarrow [1, -5, 3] - 2 \cdot \left[0, \frac{1}{2}, 1\right] = [1, -6, 1]$. Nu is met $\hat{y} = \text{sgn}\left(1 \cdot 0 + -6 \cdot \frac{1}{2} + 1 \cdot 1\right) = \text{sgn}\,(-2) = -1$ de voorspelling juist. De uiteindelijke gewichten bedragen dus $\boldsymbol{w} = [1, -6, 1]$.

Oplossing 11. De updateregel zou dan wel correct blijven werken; wel worden de gewichten maar half zo sterk bijgewerkt per stap. Als begonnen wordt met alle modelparameters gelijk aan nul maakt dit niet uit.

Oplossing 12. -

Oplossing 13. Het teken van de bias is negatief. In de oorsprong is de waarde van het argument van de signum-functie gelijk aan de bias, en daar levert de signum-functie blijkbaar het klasselabel -1 op.

Oplossing 14. Deze data is wel lineair separabel. Het perceptron algoritme met beginwaarden $b = 0$, $w_1 = 0$, $w_2 = 0$ convergeert als volgt:

1. Voorspelling op instance $x_1 = 1$, $x_2 = 0$ levert $\hat{y} = 0$ zodat $\hat{y} - y = -1$; updateregel levert $b = +1$, $w_1 = +1$, $w_2 = 0$.
2. Voorspelling op instance $x_1 = 0$, $x_2 = 1$ levert $\hat{y} = +1$ zodat $\hat{y} - y = 0$; geen update nodig.
3. Voorspelling op instance $x_1 = -1$, $x_2 = 0$ levert $\hat{y} = 0$ zodat $\hat{y} - y = +1$; updateregel levert $b = 0$, $w_1 = +2$, $w_2 = 0$.
4. Voorspelling op instance $x_1 = 0$, $x_2 = -1$ levert $\hat{y} = 0$ zodat $\hat{y} - y = +1$; updateregel levert $b = -1$, $w_1 = +2$, $w_2 = +1$.
5. Voorspelling op instance $x_1 = 1$, $x_2 = 0$ levert $\hat{y} = +1$ zodat $\hat{y} - y = 0$; geen update nodig.
6. Voorspelling op instance $x_1 = 0$, $x_2 = 1$ levert $\hat{y} = 0$ zodat $\hat{y} - y = -1$; updateregel levert $b = 0$, $w_1 = +2$, $w_2 = +2$.
7. Hierna blijken alle instances juist te worden geclassificeerd met een scheidingslijn die diagonaal naar beneden door de oorsprong loopt; er zijn geen verdere updates meer nodig.

Oplossing 15. -

Oplossing 16. -

Oplossing 17. Het lineaire regressie algoritme met beginwaarden $b = 0$, $w_1 = 0$ convergeert als volgt:

1. Voorspelling op instance $x_1 = -1$ levert $\hat{y} = 0$ zodat $\hat{y} - y = +1$; updateregel levert $b = -\frac{1}{2}$, $w_1 = +\frac{1}{2}$.
2. Voorspelling op instance $x_1 = +1$ levert $\hat{y} = 0$ zodat $\hat{y} - y = -3$; updateregel levert $b = +1$, $w_1 = +2$.
3. Hierna blijken alle instances juist te worden voorspeld; er zijn geen verdere updates meer nodig.

Oplossing 18. -

Oplossing 19. De signum-functie kijkt niet naar de grootte van het argument, alleen naar het teken. De learning rate beïnvloedt de grootte van de updates in de modelparameters, en daardoor de schaling van de modelparameters, maar de signum-functie is hier niet gevoelig voor.

Oplossing 20. Een loss $l = 0$ treedt alleen op als een instance juist wordt voorspeld. In het algemeen zal een model niet alle instances juist voorspellen.

Oplossing 21. Een loss-functie dient minimaal (d.w.z. nul) te zijn voor een optimaal model en groter (d.w.z. positiever) naarmate een model slechter is.

- $\mathcal{L}(\hat{y}; y) = \hat{y} - y$ geeft een negatieve loss voor sommige verkeerd geclassificeerde instances.
- $\mathcal{L}(\hat{y}; y) = (\hat{y} - y)^2$ is geschikt.
- $\mathcal{L}(\hat{y}; y) = |\hat{y} - y|$ is geschikt.
- $\mathcal{L}(\hat{y}; y) = \hat{y}^2 - y^2$ geeft een negatieve loss voor sommige verkeerd geclassificeerde instances.
- $\mathcal{L}(\hat{y}; y) = (\hat{y} + y)^2$ geeft loss nul voor sommige verkeerd geclassificeerde instances.

Oplossing 22. -

Oplossing 23. Het k-means algoritme kan tot verschillende oplossingen komen, afhankelijk van hoe de clusters geïnitialiseerd worden. Hierdoor bestaat het risico in een lokaal minimum te belanden, wat tot een suboptimale uitkomst leidt. Door het algoritme vele malen te herhalen met telkens andere willekeurige startwaarden wordt de kans verhoogd dat de beste gevonden oplossing ook daadwerkelijk de globale optimale uitkomst is.

Oplossing 24. -

Oplossing 25. -

Oplossing 26. In het ene geval is $\frac{\Delta y}{\Delta x} = \frac{\ln(2.01) - \ln(2.00)}{0.01} = \frac{0.698134.. - 0.693147..}{0.01} = 0.498754..$; in het andere geval is $\frac{\Delta y}{\Delta x} = \frac{\ln(2.01) - \ln(1.99)}{0.02} = \frac{0.698134.. - 0.688134..}{0.02} = 0.500004...$ De tweede, symmetrische formule is nauwkeuriger.

Oplossing 27. De eerste formule $\frac{\Delta y}{\Delta x} = \frac{f(x+\Delta x) - f(x)}{\Delta x}$ geeft $\frac{\Delta y}{\Delta x} = \frac{(x+\Delta x)^2 - x^2}{\Delta x} = \frac{x^2 + 2x \cdot \Delta x + \Delta x^2 - x^2}{\Delta x} = 2x + \Delta x$; de tweede formule $\frac{\Delta y}{\Delta x} = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x}$ geeft $\frac{\Delta y}{\Delta x} = \frac{(x+\Delta x)^2 - (x-\Delta x)^2}{2\Delta x} = \frac{x^2 + 2x \cdot \Delta x + \Delta x^2 - x^2 - 2x \cdot \Delta x - \Delta x^2}{2\Delta x} = 2x$. De wiskundige afgeleide van x^2 is $2x$. Dit komt exact overeenkomt met de tweede methode, terwijl de eerste methode alleen redelijk zal kloppen als Δx heel klein is.

Oplossing 28. Als $\hat{y} = \varphi(a) = a$, dan is $\frac{\partial \hat{y}}{\partial a} = \frac{\partial a}{\partial a} = 1$.

Oplossing 29. -

Oplossing 30. -

Oplossing 31. $\frac{\partial \mathcal{L}}{\partial \hat{y}} \approx \frac{\mathcal{L}(\hat{y}+\Delta\hat{y};y)-\mathcal{L}(\hat{y}-\Delta\hat{y};y)}{2\Delta\hat{y}} = \frac{(\hat{y}+\Delta\hat{y}-y)^2-(\hat{y}-\Delta\hat{y}-y)^2}{2\Delta\hat{y}}$ hetgeen vereenvoudigt tot $\frac{\left(\hat{y}^2+\Delta\hat{y}^2+y^2+2\hat{y}\Delta\hat{y}-2\hat{y}y-2\Delta\hat{y}y\right)-\left(\hat{y}^2+\Delta\hat{y}^2+y^2-2\hat{y}\Delta\hat{y}-2\hat{y}y+2\Delta\hat{y}y\right)}{2\Delta\hat{y}} = \frac{4\hat{y}\Delta\hat{y}-4\Delta\hat{y}y}{2\Delta\hat{y}} = 2\left(\hat{y}-y\right)$.

Oplossing 32. -

Oplossing 33. -

Oplossing 34. De algemene updateregel $w_i \leftarrow w_i - \alpha \cdot \frac{\partial a}{\partial w_i}\frac{\partial \hat{y}}{\partial a}\frac{\partial l}{\partial \hat{y}}$ werkt de gewichten (inclusief de bias) bij met nul als $\frac{\partial \hat{y}}{\partial a} = 0$, ongeacht de waarden van de andere factoren. De modelparameters worden dan effectief helemaal niet bijgewerkt, wat ervoor zorgt dat het model niet leert.

Oplossing 35. Het bewijs verloopt als volgt:

- $\tanh(-a) = \frac{e^{(-a)}-e^{-(-a)}}{e^{(-a)}+e^{-(-a)}} = \frac{e^{-a}-e^{a}}{e^{-a}+e^{a}} = \frac{-\left(e^{a}-e^{-a}\right)}{e^{a}+e^{-a}} = -\tanh(a)$;
- $\text{softsign}(-a) = \frac{-a}{1+|-a|} = -\frac{a}{1+|a|} = -\text{softsign}(a)$;
- $\text{isru}(-a) = \frac{-a}{\sqrt{1+(-a)^2}} = -\frac{a}{\sqrt{1+a^2}} = -\text{isru}(a)$.

Oplossing 36. -

Oplossing 37. Voor de softsign-activatiefunctie wordt de updateregel

$$\begin{cases} b \leftarrow b - \alpha\left(\hat{y}-y\right)\left(1-|\hat{y}|\right)^2 \\ w_i \leftarrow w_i - \alpha\left(\hat{y}-y\right)\left(1-|\hat{y}|\right)^2 x_i \end{cases}$$

en voor de isru-activatiefunctie

$$\begin{cases} b \leftarrow b - \alpha\left(\hat{y}-y\right)\left(1-\hat{y}^2\right)^{\frac{3}{2}} \\ w_i \leftarrow w_i - \alpha\left(\hat{y}-y\right)\left(1-\hat{y}^2\right)^{\frac{3}{2}} x_i \end{cases}$$

Deze regels hebben respectievelijk extra weegfactoren $\left(1-|\hat{y}|\right)^2$ en $\left(1-\hat{y}^2\right)^{\frac{3}{2}}$ die in beide gevallen gelijk zijn aan 1 (d.w.z. zwaarste weging) als $\hat{y} = 0$ (d.w.z. op de scheidingslijn tussen klassen, met onzekere classificatie) en gelijk zijn aan 0 (d.w.z. géén weging) als $\hat{y} = \pm 1$ (d.w.z. ver van de scheidingslijn, met zekere classificatie).

Oplossing 38. De Gudermannfunctie kan worden geschreven in termen van hyperbolische en inverse goniometrische functies, zoals $\text{gd}(a) = \arcsin(\tanh(a))$ of $\text{gd}(a) = \arccos\left(\frac{1}{\cosh(a)}\right)$, en diens afgeleide is gelijk aan $\frac{\text{d}}{\text{d}a}\text{gd}(a) = \frac{1}{\cosh(a)}$ (zie bv. de wikipedia pagina over deze functie). De laatste twee formules kunnen worden gecombineerd tot $\frac{\text{d}}{\text{d}a}\text{gd}(a) = \cos(\text{gd}(a))$.

Oplossing 39. -

Oplossing 40. De IMP-operator kan gemodelleerd worden met een perceptron zoals $\hat{y} = \text{sgn}\,(1 - x_1 + x_2)$ oftewel $b = +1$, $w_1 = -1$, $w_2 = +1$.

Oplossing 41. -

Oplossing 42. -

Oplossing 43. De relu-activatiefunctie kan als volgt gebruikt worden:

- $\text{NOT}\,(x_1) = \text{relu}\,(1 - x_1)$, oftewel $b = +1$, $w_1 = -1$;
- $\text{AND}\,(x_1, x_2) = \text{relu}\,(-1 + x_1 + x_2)$, oftewel $b = -1$, $w_1 = +1$, $w_2 = +1$;
- $\text{OR}\,(x_1, x_2)$ is onmogelijk.

Oplossing 44. De ternaire operator kan niet door het perceptron worden beschreven omdat de data niet lineair separabel is.

Oplossing 45. Beide antwoorden zijn verdedigbaar: boosting levert een serie van modellen die *opeenvolgend* de gemaakte fouten van vorige modellen corrigeren, dus de *constructie* van de modellen geschiedt *serieel*; zodra de serie modellen echter is bepaald en wordt toegepast worden de *onafhankelijke* uitkomsten van meerdere modellen gemiddeld/opgeteld, dus de *toepassing* van de modellen geschiedt *parallel*.

Oplossing 46. De EQV-operator kan worden gemodelleerd met een two-layer perceptron. Het kan worden verkregen uit het model voor de XOR-operator door in de eerste laag de gewichten w_1 van beide neuronen te vermenigvuldigen met -1 (of, door het teken van de beide gewichten w_2 te flippen).

Oplossing 47. -

Oplossing 48. -

Oplossing 49. De identiteitsfunctie doet eigenlijk niets. Er wordt dan een lineaire combinatie van een laag effectief gevolgd door een lineaire combinatie van een twee laag. Een lineaire combinatie van een lineaire combinatie is zelf ook weer een linaire combinatie, dus dan voegt de laag met de lineaire activatiefunctie niets toe aan het model.

Oplossing 50. -

Oplossing 51. Je kan de ternaire operator ontbinden in simpelere operaties die af te lezen zijn uit de tabel: de ternaire operator is alleen dan waar als (NOT x_1 AND x_3) OR (x_1 AND x_2). De twee AND-operaties kunnen elk berekend worden door een perceptron in de eerste laag in de vorm $h_1 = \text{sgn}\,(-1 - x_1 + x_3)$ en $h_2 = \text{sgn}\,(-1 + x_1 + x_2)$, en de OR-operatie kan worden uitgevoerd door een perceptron in de tweede laag in de vorm $\hat{y} = \text{sgn}\,(1 + h_1 + h_2)$. Hier is h_i de uitvoer van de eerste laag die de invoer vormt van de tweede laag, en worden logische waarden gecodeerd als ± 1. Vele andere oplossingen zijn mogelijk.

Oplossing 52. Het product zou al nul zijn als slechts één oplossing juist voorspeld wordt. Oftewel, als één van de outputs juist voorspeld wordt, worden fouten in de andere outputs niet meer bestraft. Dit zou onwenselijk zijn.

Oplossing 53. Het maximum van alle losses is weliswaar alleen dan nul als alle losses nul zijn, en positief naarmate de fouten groter worden. Maar deze uitkomst hangt alleen af van de slechtst voorspelde output. Oftewel, het model probeert dan alleen de slechtste voorspelling te verbeteren, maar heeft geen reden om minder slechte outputs nog wat verder te verbeteren.

Oplossing 54. -

Oplossing 55. -

Oplossing 56. De relu-activatiefunctie loopt vlak voor negatieve argumenten. Door een positieve bias te kiezen zullen instances vaker op de positieve helft van de functie terecht komen die een helling heeft dan op de negatieve helft met helling nul. Dit is prettig omdat helling nul betekent dat dat neuron voor die instance op dat moment niet zal leren. Het voorkomt daarmee overmatige saturatie.

Oplossing 57. -

Oplossing 58. Gegeven werd dat een uniforme verdeling tussen -1 en +1 een standaarddeviatie heeft van $\frac{1}{\sqrt{3}}$. Voor een uniforme verdeling tussen grenzen $-g$ en $+g$ schaalt de standaarddeviatie dus mee naar $\frac{g}{\sqrt{3}}$. Als $g = \sqrt{\frac{6}{N_{\text{in}}+N_{\text{uit}}}}$, dan is dus $\sigma = \frac{\sqrt{\frac{6}{N_{\text{in}}+N_{\text{uit}}}}}{\sqrt{3}} = \sqrt{\frac{\frac{6}{N_{\text{in}}+N_{\text{uit}}}}{3}} = \sqrt{\frac{2}{N_{\text{in}}+N_{\text{uit}}}} = \sqrt{\frac{1}{N}} = \frac{1}{\sqrt{N}}$ waarbij $N = \frac{N_{\text{in}}+N_{\text{uit}}}{2}$.

Oplossing 59. Voor de meeste activatiefuncties geldt dat $|\varphi(a)| \leq |a|$, oftewel de grootte van getallen wordt erdoor verkleind. Het gebruik van activatiefuncties bij gewichten met een typische grootte van $\sigma = \frac{1}{\sqrt{N}}$ zorgt er tijdens forward-propagation typisch voor dat de output van een model met veel lagen na initialisatie (iets te) klein is. Omgekeerd geldt voor de meeste activatiefuncties dat $|\varphi'(a)| \leq 1$, oftewel de grootte van gradiënten wordt erdoor verkleind. Dit zorgt er bij back-propagation typisch voor dat de gradiënten waarmee het model leert (iets te) klein zijn. Voor modellen met erg veel lagen kan dit het voorspellen en leren zodanig verslechteren dat er in de praktijk geen goede oplossing gevonden wordt.

Oplossing 60. -

Oplossing 61. $\max(\boldsymbol{x}) = \pi$, $\operatorname{argmax}(\boldsymbol{x}) = 4$ of 3 of $[0,0,0,1,0]$ (afhankelijk van de definitie), $\operatorname{softmax}(\boldsymbol{x}) = [0.000000..,0.111234..,0.044112..,0.510402..,0.334249..]$.

Oplossing 62. Als de som van alle kansen exact 100% moet bedragen, en geen van de kansen mag negatief zijn, dan moet elk element zelf ook al niet hoger zijn dan 100%. Anders gezegd, als een kans hoger dan 100% zou zijn, en alle andere kansen zijn niet negatief, dan zou ook de som hoger dan 100% zijn, en dit voldoet niet aan de gegeven voorwaarden.

Oplossing 63. $\frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i} \cdot e^c}{\sum_j e^{x_j} \cdot e^c} = \frac{e^c \cdot e^{x_i}}{e^c \cdot \sum_j e^{x_j}} = \frac{e^{x_i}}{\sum_j e^{x_j}}$.

Oplossing 64. argmax $(\boldsymbol{v})$ bevat een 1 op de positie van het grootste element van $\boldsymbol{v}$, en elders 0. Het inproduct is gelijk aan de som van de producten van overeenkomstige elementen. De nullen dragen niet bij aan de som, dus het enige element dat overblijft is $v_i \cdot 1$, met index i die verwijst naar het maximale element. Dit is hetzelfde als max $(\boldsymbol{v})$.

Oplossing 65. -

Oplossing 66. -

Oplossing 67. Uit de gegeven formules volgt:

- indien $m = n$, dan $\frac{\partial \hat{y}_n}{\partial h_m} = \hat{y}_n \cdot (1 - \hat{y}_m)$, maar omdat m en n gelijk zijn kunnen de indices worden uitgewisseld, zodat dit overeenkomt met $\frac{\partial \hat{y}_n}{\partial h_m} = \hat{y}_m \cdot (1 - \hat{y}_n)$, en omdat $\delta_{mn} = 1$ kan dit worden geschreven als $\frac{\partial \hat{y}_n}{\partial h_m} = \hat{y}_m \cdot (\delta_{mn} - \hat{y}_n)$;
- indien $m \neq n$, dan $\frac{\partial \hat{y}_n}{\partial h_m} = -\hat{y}_n \cdot \hat{y}_m$, maar omdat het product commutatief is kunnen de factoren worden uitgewisseld, zodat dit overeenkomt met $\frac{\partial \hat{y}_n}{\partial h_m} = -\hat{y}_m \cdot \hat{y}_n$, en omdat $\delta_{mn} = 0$ kan dit worden geschreven als $\frac{\partial \hat{y}_n}{\partial h_m} = \hat{y}_m \cdot (\delta_{mn} - \hat{y}_n)$.

Hiermee zijn beide gevallen bekeken.

Oplossing 68. Dit volgt uit het feit dat de outputs $\hat{y}_n$ van een softmay layer tussen nul en één liggen, waardoor zowel $\hat{y}_n \geq 0$ als $1 - \hat{y}_n \geq 0$ voor alle n. Hierdoor is het product $\hat{y}_n \cdot (1 - \hat{y}_m) \geq 0$ en het product $-\hat{y}_n \cdot \hat{y}_m \leq 0$.

Oplossing 69. -

Oplossing 70. $\sum_n \frac{\partial \hat{y}_n}{\partial h_m} = \frac{\partial}{\partial h_m} \left(\sum_n \hat{y}_n\right) = \frac{\partial}{\partial h_m} 1 = 0$ omdat de som van alle outputs van de softmax-functie gelijk is aan 1 en de afgeleide van een constante functie gelijk is aan 0. Het kan ook omslachtiger worden afgeleid: $\sum_n \frac{\partial \hat{y}_n}{\partial h_m} = \sum_n \hat{y}_m \cdot (\delta_{mn} - \hat{y}_n) = \hat{y}_m \cdot \sum_n (\delta_{mn} - \hat{y}_n) = \hat{y}_m \cdot (\sum_n \delta_{mn} - \sum_n \hat{y}_n) = \hat{y}_m \cdot (1 - 1) = 0$.

Oplossing 71. $\sigma(a) + \sigma(-a) = \frac{1}{1+e^{-a}} + \frac{1}{1+e^{-(-a)}} = \frac{e^a}{e^a(1+e^{-a})} + \frac{1}{1+e^a} = \frac{e^a}{e^a \cdot 1 + e^a \cdot e^{-a}} + \frac{1}{e^a+1} = \frac{e^a}{e^a+1} + \frac{1}{e^a+1} = \frac{e^a+1}{e^a+1} = 1$.

Oplossing 72. -

Oplossing 73. -

Oplossing 74. Voor een instance met label B is de kwadratische loss $l = (0.1-0)^2 + (0.4-1)^2 + (0.2-0)^2 + (0.3-0)^2 = 0.500$ en de cross-entropy $l = -0 \cdot \log(0.1) - 1 \cdot \log(0.4) - 0 \cdot \log(0.2) - 0 \cdot \log(0.3) = 0.916$. Voor een instance met label A is de kwadratische loss $l = (0.1-1)^2 + (0.4-0)^2 + (0.2-0)^2 + (0.3-0)^2 = 1.100$ en de cross-entropy $l = -1 \cdot \log(0.1) - 0 \cdot \log(0.4) - 0 \cdot \log(0.2) - 0 \cdot \log(0.3) = 2.303$.

Oplossing 75. De totale loss is gelijk aan $\sum_i -y \ln(\hat{y}) - (1-y)\ln(1-\hat{y}) = -60 \cdot \ln(\hat{y}_A) - 40 \cdot \ln(\hat{y}_B) = -60 \cdot \ln(\hat{y}_A) - 40 \cdot \ln(1-\hat{y}_A)$. Om het minimum te bepalen bepalen we waar de helling gelijk is aan nul: $\frac{\partial}{\partial \hat{y}_A}[-60 \cdot \ln(\hat{y}_A) - 40 \cdot \ln(1-\hat{y}_A)] = 0 \iff \frac{-60}{\hat{y}_A} + \frac{40}{1-\hat{y}_A} = 0 \iff \frac{-60(1-\hat{y}_A)}{\hat{y}_A(1-\hat{y}_A)} + \frac{40(\hat{y}_A)}{\hat{y}_A(1-\hat{y}_A)} = 0 \iff -60(1-\hat{y}_A) + 40(\hat{y}_A) = 0 \iff -60 + 100\hat{y}_A = 0 \iff \hat{y}_A = 0.60$ en $\hat{y}_B = 1 - \hat{y}_A = 0.40$.

Oplossing 76. -

Oplossing 77. De afgeleide $\frac{\mathrm{d}}{\mathrm{d}a}\mathrm{relu}(a) = 0$ als $a < 0$, terwijl $\frac{\mathrm{d}}{\mathrm{d}a}\mathrm{relu}(a) = 1$ als $a > 0$. Het is eenvoudig na te gaan dat dit ook volgt uit de uitdrukking met de signum-functie. (Voor $a = 0$ is de afgeleide niet gedefinieerd omdat de relu een scherpe "knik" heeft.)

Oplossing 78. De signum-activatiefunctie heeft als voordeel dat de voorspellingen $\hat{y}$ direct geinterpreteerd kunnen worden als labels; dat kan bij de hardmax-lossfunctie niet. De hardmax-lossfunctie heeft als voordeel dat deze differentieerbaar is en met back-propagation kan worden geoptimaliseerd; dat kan bij de signum-activatiefunctie niet.

Oplossing 79. -

Oplossing 80. $\mathcal{L}(\hat{y}; y) = \max(-\hat{y} \cdot y, 0) = \mathrm{relu}(-\hat{y} \cdot y)$.

Oplossing 81. -

Oplossing 82. -

Oplossing 83. $100 \times 11 + 100 \times 101 + 100 \times 101 + 10 \times 101 = 22310$ modelparameters.

Oplossing 84. Als het model (deels) wordt geoptimaliseerd op de testdata gaat het op deze data gemiddeld beter presteren dan op andere willekeurige data. Hiermee is het geen betrouwbare maat meer voor de prestaties van het model op onbekende data.

Oplossing 85. -

Oplossing 86. Een verlaging van de capaciteit vermindert overfitting, laat de fouten op de trainingsdata toenemen en de fouten op de validatiedata afnemen. Dit kan inderdaad worden beschouwd als een regularisatiemethode.

Oplossing 87. Een model met onjuiste parameters maakt systematische fouten, dus vertoont bias. Een model met weinig parameters kan niet de ruis in de trainingsdata fitten, ongeacht of die parameters juist zijn, dus vertoont geen variantie. Anita heeft gelijk, Bob niet.

Oplossing 88. -

Oplossing 89. Stel bijvoorbeeld er zijn honderd instances in een dichotoom classificatie-probleem, en hiervan worden er negentig met een voorspelling van 51% zekerheid op het nippertje juist geclassificeerd, en tien met een voorspelling van bijvoorbeeld 0.00001% volkomen onjuist geclassificeerd. De nauwkeurigheid is dan behoorlijk goed, maar de gemiddelde loss behoorlijk slecht.

Oplossing 90. ZeroR heeft de laagste capaciteit, want dit model heeft helemaal géén modelparameters. k-Nearest Neighbours daarentegen heeft de hoogste capaciteit omdat deze alle trainingsdata onthoudt en in het meest extreme geval ook alle trainingsdata juist classificeert zonder dat dit goed hoeft te generaliseren naar validatiedata (iets soortgelijks kan worden gezegd van een J48-Tree of Random Forest, mits deze niet worden gepruned!).

Oplossing 91. -

Oplossing 92. Een model met voldoende grote capaciteit kan alle trainingsdata goed leren voorspellen (zolang er geen instances voorkomen met identieke attributen maar verschillende uitkomsten). In dat geval kan de trainingsloss dus wel naar nul dalen en de trainingsaccuracy naar 100%. Dit is de meest extreme vorm van overfitting en dus niet wenselijk.

Oplossing 93. Dit duidt meestal op een te grote learning rate α, waardoor stochastic gradient descent niet naar het optimum convergeert maar er overheen schiet en in een slechtere oplossing terecht kan komen.

Oplossing 94. Nee, dit model is dan gebiasd om het goed te doen op de validatiedata maar zal niet vergelijkbaar presteren op testdata.

Oplossing 95. Je zou een kleine hoeveelheid willekeurige ruis aan de lengte- en breedtematen kunnen toevoegen, of je zou in dit geval bijvoorbeeld alle overeenkomstige waarden van twee bloemen van dezelfde klasse kunnen middelen om een nieuwe "virtuele" bloem te genereren.

Oplossing 96. -

Oplossing 97. Nee; door de afmetingen van een zalm en een walvis te middelen kun je wellicht gegevens krijgen die lijken op die van honden of olifanten, en die behoren precies tot de andere klasse.

Oplossing 98. Operaties zoals spiegelen, roteren, of transleren veranderen de aard van het terrein niet en kunnen worden toegepast; operaties die de schaal, kleur, of scherpte beïnvloeden kunnen beter worden vermeden.

Oplossing 99. Deze methode zal de prestaties op de trainingsdata verslechteren, omdat de ruis niet informatief is. Tegelijkertijd kunnen de prestaties op de validatie- en testdata verbeteren als hiermee overfitting wordt tegengegaan. We verbeter hiermee de prestaties op validatiedata ten koste van die op trainingsdata, en dit is de kenmerkende karakteristiek van regularisatiemethoden.

Oplossing 100. Dan krijg je een ongeregulariseerd model, dat wil zeggen zonder L_1- of L_2-regularisatie.

Oplossing 101. -

Oplossing 102. Nee, dit werkt niet omdat w_{ij}^3 negatief kan zijn (als w_{ij} negatief is). Je zou dit op kunnen lossen door de absolute waarde van de derde macht te nemen, bv. $l = \lambda_3 \sum_{i,j} \left| w_{ij}^3 \right| + \sum_m \mathcal{L}\left(\hat{y}_m; y_m\right)$.

Oplossing 103. -

Oplossing 104. $w_{ij} \leftarrow w_{ij} - \alpha \cdot \frac{\partial l}{\partial w_{ij}}$ met $l = \lambda_2 \sum_{i,j} w_{ij}^2$ geeft $w_{ij} \leftarrow w_{ij} - \alpha\lambda_2 \cdot \frac{\partial}{\partial w_{ij}} w_{ij}^2$ omdat alle andere gewichten niet meeveranderen als een gewicht w_{ij} wordt gewijzigd. Dit leidt tot $w_{ij} \leftarrow w_{ij} - \alpha\lambda_2 \cdot 2w_{ij} = w_{ij}\,(1 - 2\alpha\lambda_2)$.

Oplossing 105. $w_{ij} \leftarrow w_{ij} - \alpha \cdot \frac{\partial l}{\partial w_{ij}}$ met $l = \lambda_1 \sum_{i,j} |w_{ij}|$ geeft $w_{ij} \leftarrow w_{ij} - \alpha\lambda_1 \cdot \frac{\partial}{\partial w_{ij}} |w_{ij}|$ omdat alle andere gewichten niet meeveranderen als een gewicht w_{ij} wordt gewijzigd. Omdat de afgeleide van de absolute waarde gelijk is aan de signum functie leidt dit tot $w_{ij} \leftarrow w_{ij} - \alpha\lambda_1 \cdot \operatorname{sgn}(w_{ij}) = \operatorname{sgn}(w_{ij}) \cdot |w_{ij}| - \alpha\lambda_1 \cdot \operatorname{sgn}(w_{ij}) = \operatorname{sgn}(w_{ij})\,(|w_{ij}| - \alpha\lambda_1)$. Hieruit volgt $|w_{ij}| = w_{ij} \cdot \operatorname{sgn}(w_{ij}) \leftarrow \operatorname{sgn}^2(w_{ij})\,(|w_{ij}| - \alpha\lambda_1) = 1 \cdot (|w_{ij}| - \alpha\lambda_1)$ waardoor dus $|w_{ij}| \leftarrow |w_{ij}| - \alpha\lambda_1$. (Overigens geldt dit formeel alleen als het teken van w_{ij} tijdens de update niet verandert, maar dit is voor kleine updates nagenoeg altijd het geval.)

Oplossing 106. -

Oplossing 107. Je kan op twintig manieren drie neuronen uitschakelen van de zes (kwestie van tellen, of berekenen $\begin{pmatrix} 6 \\ 3 \end{pmatrix} = \frac{6!}{3! \cdot 3!} = 20$). Voor vier onafhankelijke hidden layers geeft dit $20^4 = 160000$ verschillende sub-netwerken.

Oplossing 108. Nee; One-R gebruikt inderdaad maar één attribuut, maar dropout gebruikt kan in elke trainingsstap een ander attribuut selecteren en hierdoor toch geleidelijk op alle attributen trainen. Daarnaast wordt tijdens het toepassen van het model wél gebruik gemaakt van alle attributen. Tenslotte is natuurlijk ook de functie van alle hidden layers anders dan in het One-R algoritme.

Oplossing 109. Als voor de activatiefunctie geldt dat $\varphi(0) = 0$, dan maakt het niet uit of je dropout toepast voor of nadat je de activatiefunctie toepast. Voor de meeste activatiefuncties, maar niet alle, is dit het geval.

Oplossing 110. Online learning is minibatch learning met een batchgrootte 1; full batch learning is minibatch learning met een batchgrootte N_{training}.

Oplossing 111. Bij online learning wordt een volgende instance doorgerekend met het model waarin de parameters zijn geüpdate middels een vorige instance. Omdat alle nieuwe modelparameters pas bekend zijn nadat de training met de vorige instances klaar is, kun je een volgende instance niet reeds parallel aan een vorige beginnen door te rekenen.

Oplossing 112. Als je de cost definieert als de som in plaats van als het gemiddelde, wordt de waarde daarvan 100 maal zo hoog. Om dezelfde updates te krijgen moet je de learning rate dan 100 maal kleiner kiezen om daarvoor te compenseren, dat wil zeggen $\alpha = 0.001$.

Oplossing 113. -

Oplossing 114. Exponentiële afname is het snelst. Lineaire afname is op de lange termijn het traagst omdat deze uiteindelijk helemaal niet meer afneemt (hoewel op de middelgrote termijn inverse afname het traagst is).

Oplossing 115. -

Oplossing 116. Als je van een ondiep minimum doorrolt in een diep minimum is de kans groot dat je niet meer uit dat diepere minimum komt en daarin blijft steken, maar als je van een diep minimum doorrolt naar een ondiep minimum is de kans nog behoorlijk dat je ook weer aan dat ondiepere minimum ontsnapt en terugrolt het diepe minimum in.

Oplossing 117. Bij Nesterov momentum wil je tijdens de forward propagation alvast een update maken volgens de gradiënt zoals je verwacht dat die gaat zijn. Als je momentum gebruikt houd je die gradiënt (G) bij en verandert die slechts geleidelijk vanwege het gebruik van de frictie parameter (β). Als je (stochastic) gradient descent gebruikt heb je geen goed idee wat de nieuwe gradient gaat zijn; als je bijvoorbeeld over het minimum heen stapt keert de gradiënt zelfs plotsklaps om. Je kunt hier tijdens forward propagation dus niet gebruik van maken.

Oplossing 118. Deze aanpassing zorgt er eigenlijk voor dat niet de waarde van de gradiënt $\frac{\partial J}{\partial w}$ wordt bijgehouden volgens de formule uit de hoofdtekst, maar de omlaag geschaalde gradiënt $(1-\beta)\frac{\partial J}{\partial w}$. Je zou hiervoor kunnen corrigeren door de learning rate α omhoog te schalen met een factor $\frac{1}{1-\beta}$, zodat het product van learning rate en gradiënt dat in de updateregel voorkomt gelijk blijft.

Oplossing 119. Ja, dit werkt ook, want het vermijdt delen door nul net zozeer en dat is de enige functie van ϵ.

Oplossing 120. Als de gradiënt altijd dezelfde waarde $G = \frac{\partial J}{\partial w}$ blijkt te hebben, convergeert Q_n op den duur naar een constante $Q_{n\to\infty} = G^2$ omdat hier een gewogen gemiddelde van de gradiënten wordt genomen, en als alle gradiënten identiek zijn is ook dat gemiddelde hieraan gelijk. De updateregel wordt dan $w \leftarrow w - \frac{\alpha}{\sqrt{G^2+\epsilon}}G \approx w - \frac{\alpha}{\sqrt{G^2}}G = w - \alpha\frac{G}{|G|} = w - \alpha\text{sgn}\,(G)$. Oftewel, de *grootte* van de stappen is constant en gelijk aan α; de *richting* van de stappen wordt bepaald door het teken van de gradiënt G. Deze methoden is weinig gevoelig voor saturatie omdat de update niet zozeer afhangt van de grootte van de gradiënt, maar voornamelijk bepaalt wordt door de vrij te kiezen learning rate.

Oplossing 121. -

Oplossing 122. Nee; bijvoorbeeld, als de gradiënt telkens wisselt van teken maar gelijk blijft van grootte, dan middelt G uit richting nul maar Q doet dit niet.

Oplossing 123. -

Oplossing 124. RMSProp houdt wel een lopende gemiddelde bij van het kwadraat van de gradiënt, dus β_2 van Adam moet overeenkomen met β van RMSProp. Maar RMSProp houdt géén gemiddelde bij van de gradiënt zelf (zoals momentum dat doet), dus β_1 van Adam moet daartoe gelijk gesteld worden aan nul. (Overigens verschilt Adam dan nog steeds enigszins van RMSProp door het gebruik van unbiased waarden voor G en Q, maar na een aantal stappen dooft dit verschil vanzelf uit.)

Oplossing 125. Als G aanvankelijk gelijk gesteld wordt aan nul, blijft deze doorwerken door het "geheugeneffect" van Adam, en dit neigt ernaar de updates te klein te maken; als Q aanvankelijk gelijk gesteld wordt aan nul, neigt dit er soortgelijk naar de updates te groot te maken (omdat je in de updateregel deelt door Q). Het gevolg is dat als $\beta_1 \gg \beta_2$, dan "wint" het geheugeneffect van G en zijn de biased updates te klein, terwijl als $\beta_1 \ll \beta_2$, dan "wint" het geheugeneffect van Q en zijn de biased updates te groot. De unbiased updates corrigeren hiervoor.

Oplossing 126. Nee; momentum maakt geen gebruik van het schalen van gradiënten middels Q, en hoe je β_2 ook kiest bij Adam, de schaling blijft altijd doorwerken.