Universiteit Hasselt

Bachelorproef

Concepten en technieken voor het trainen van deep neural networks

Auteur:
Michiel Vanmunster

Promotor:
Jan Van den Bussche
Begeleider:
Tom Ameloot

Bachelorproef voorgedragen tot het behalen van de graad van Bachelor in de Informatica

Academiejaar 2015–2016

Inhoudsopgave

Voorwoord

Deze bachelorproef was een zeer boeiende ervaring. Voor ik eraan begon, had ik nog nooit gehoord over machine learning, laat staan over neurale netwerken. Opzoekingen rond dit onderwerp wekten een grote interesse in machine learning bij me op. Het fascineert me namelijk dat machines in staat zijn om zelfstandig bepaalde problemen te leren oplossen door ze simpelweg voorbeelddata voor te schotelen. Daardoor maakt machine learning het mogelijk om toepassingen te realiseren waarvoor het schrijven van een conventioneel programma, bestaande uit een precieze opeenvolging van instructies, zeer moeilijk zou zijn. Deze bachelorproef me toegestaan te begrijpen hoe dit mogelijk is, wat voor grote voldoening zorgde. Zelf heb ik ook twee zulke toepassingen gerealiseerd: het herkennen van een handgeschreven cijfer, en het bepalen van het muziekgenre waartoe een muziekbestand behoort. Dankzij deze bachelorproef heb ik uitgebreide en diepgaande kennis verworven over neurale netwerken. De focus lag steeds op 'het begrijpen' in de mate van het mogelijke: de werking van neurale netwerken is vandaag de dag nog niet volledig doorgrond, wat dit onderwerp alleen maar fascinerender maakt. Mijn bachelorproef heeft me dan ook geen moment verveeld.

Graag wil een aantal mensen bedanken voor hun steun bij het tot stand komen van dit werk. In de eerste plaats wil ik Prof. Dr. Jan Van den Bussche, promotor van deze bachelorproef, bedanken. Zijn begeleiding en aanmoediging waren een grote hulp bij het uitwerken van deze bachelorproef. Tevens wil ik Dr. Tom Ameloot en Dr. Geert Jan Bex bedanken om me te helpen bij het beter begrijpen van de werking van neurale netwerken. Ten slotte wil ik ook mijn ouders bedanken om me door het stresserend academisch jaar te loodsen.

Samenvatting

Neurale netwerken situeren zich binnen de supervised machine learning. Deze tak binnen de Artificiële Intelligentie (AI) onderzoekt methoden om machines zelfstandig te laten leren op basis van voorbeelddata. Aangezien voor iedere input de bijhorende gewenste output gekend is, noemt men deze voorbeelddata ook wel gelabelde training data. In die data zit als het ware een functie verborgen die de input-output-mappings realiseert. Tijdens het trainen gaat de machine hiernaar op zoek. Men hoopt dan dat de machine een degelijke benadering van de verborgen functie leert en bijgevolg in staat is om ook voor andere inputs, die ze niet eerder zag, de correcte output te geven. Men hoopt dus dat de machine zal leren veralgemenen. Deze bachelorproef legt de focus op classificatieproblemen, gekenmerkt door hun eindig bereik aan outputs ('klassen').

Een neuraal netwerk is opgebouwd uit verschillende lagen ('layers'), op hun beurt opgebouwd uit neuronen. Ieder netwerk bestaat uit een inputlaag en een outputlaag. Tussenliggende lagen worden hidden layers genoemd. Een 'shallow neural network' gebruikt slechts één hidden layer. De neuronen propageren hun output per laag naar alle neuronen in de volgende laag, tot uiteindelijk de outputlaag bereikt wordt. Een veel gebruikt type neuron is het 'sigmoid neuron', dat zijn output ('activatie') berekent als $a = \sigma(\sum_{i=1}^{N} w_i x_i + b)$. Het aantal inputs van het neuron is N. Aan iedere input x_i associeert het neuron een gewicht w_i, om een gewogen som te berekenen. Daar wordt ook nog de bias b van het neuron bij opgeteld. De activatie wordt vervolgens bepaald door het toepassen van een niet-lineaire functie, de 'activatiefunctie' genaamd, op die gewogen som. Voor het sigmoid neuron is dat de 'logistic sigmoid', σ genoteerd, met $\sigma(z) = 1/(1 + e^{-z})$. De tekst bespreekt ook nog andere activatiefuncties, namelijk de softmax, lineaire, tanh en ReLU.

De gewichten en de bias vormen de parameters van een neuron. Die van alle neuronen samen vormen de parameters van het netwerk, en worden op willekeurige waarden geïnitialiseerd. Het trainen van het netwerk houdt in dat het netwerk die waarden van de parameters bepaalt waarvoor het netwerk optimaal presteert op de voorbeelddata. Een 'kostfunctie' geeft aan hoe goed het netwerk presteert in functie van de huidige waarden van de parameters: hoe beter, hoe lager de kost. Die kost hangt af van de mate waarin de output van het netwerk verschilt van de gewenste output voor de voorbeelddata. Het trainen van het netwerk bestaat er dus in deze kostfunctie te minimaliseren. De tekst bespreekt verschillende kostfuncties, namelijk de mean squared error, cross-entropy en log-likelihood. Het minimaliseren van de kostfunctie gebeurt met het zgn. 'gradient descent' algoritme. Dit algoritme wijzigt iteratief de waarden van de parameters, op basis van de partieel afgeleiden van de kostfunctie tegenover iedere parameter. De tekst geeft ook een inleiding tot geavanceerde leeralgoritmes: de Hessian techniek en het momentum-based gradient descent algoritme worden toegelicht. Om de partieel afgeleiden snel te berekenen, wordt 'backpropagation' gebruikt. Dit algoritme bepaalt eerst de 'error' voor ieder neuron in de outputlaag. Daaruit worden vervolgens laag per laag de errors in voorgaande lagen berekend. Uit de errors worden tot slot de partieel afgeleiden berekend.

Naast de parameters zijn er ook hyper-parameters. Dit zijn parameters die de gebruiker initieel instelt en typisch niet meer wijzigen gedurende het trainen. Voorbeelden hiervan zijn het aantal lagen waaruit het netwerk is opgebouwd en het aantal neuronen per laag. De tekst bespreekt nog talloze andere hyper-

parameters, en bespreekt hoe men er geschikte waarden voor kiest.

Een vaak voorkomend probleem bij het trainen is 'overfitting'. Bij overfitting neemt de benadering eventuele ruis in de voorbeelddata in acht. Zulke ruis kan men zien als data die net langs de verborgen functie, bv. een rechte, vallen. Overfitting houdt in dit geval in dat de benadering een functie is met vele parameters en bijgevolg precies kan samenvallen met al deze data (incl. de ruis). Zulke benaderingen zorgen ervoor dat het netwerk minder goed kan veralgemenen. Andere data zullen immers ook in de buurt van die rechte liggen, waardoor de benadering niet geschikt is. De verzameling technieken waarmee overfitting beperkt kan worden, valt onder de noemer 'regularisatie-technieken', waarvan 'L2 regularisatie' de populairste is. Deze techniek past de kostfunctie aan zodat de kost ook afhangt van de grootte van de gewichten: hoe groter de gewichten, hoe hoger de kost. Op die manier promoot L2 regularisatie kleine gewichten. De intuïtie hierachter is dat kleine gewichten minder gevoelig zijn voor ruis, net doordat ze iedere input weinig laten meetellen en bijgevolg alleen de algemene trend beschouwen. De tekst bespreekt ook nog andere regularisatie-technieken, namelijk L1 regularisatie, dropout en het artificieel uitbreiden van de dataset.

De tekst formuleert een visueel bewijs van het Universal Approximation Theorem (UAT), dat stelt dat een shallow neural network eender welke functie kan benaderen. Toch bestaan er ook 'deep neural networks', neurale netwerken met meerdere hidden layers. Zulke netwerken kunnen complexe verborgen functies makkelijker ontdekken, iets waar het UAT niets over zegt, en presteren bijgevolg beter. De hidden layers kunnen als abstractieniveaus gezien worden, waardoor het netwerk laag voor laag een betere benadering kan maken. Een zeer bekend type deep neural network is het 'convolutional neural network' (CNN). De netwerkarchitectuur van zo'n CNN neemt de spatiale structuur van de input in acht, waardoor het netwerk deze niet meer zelf moet afleiden uit de voorbeelddata. Bijgevolg kan het CNN de verborgen functie makkelijker achterhalen en presteert het beter. De tekst gaat ook kort in op een ander populair type deep neural network, het 'recurrent neural network' genaamd.

Deze bachelorproef omvat bovendien twee toepassingen. Doorheen de tekst gebruikte ik een toepassing als voorbeeld om met verschillende concepten en technieken te experimenteren. Deze voorbeeldtoepassing bestaat erin een afbeelding van een handgeschreven cijfer tussen 0 en 9 te classificeren. Mijn eerste toepassing bouwt hier een GUI rond, die de gebruiker toelaat zelf een cijfer te tekenen in een canvas en dit te laten classificeren. Door deze toepassing te gebruiken, deed ik twee opmerkelijke vaststellingen: de schrijfstijl en de positie van het cijfer binnen het canvas zijn cruciaal voor een correcte classificatie. Mijn tweede toepassing laat de gebruiker toe een zelfgekozen muziekbestand te classificeren op basis van het muziekgenre. De toepassing onderscheidt 10 muziekgenres, en voorziet een lijnplot en een spider diagram om het verloop van de classificatie in de tijd te visualiseren. Ik voerde een kleine literatuurstudie uit om de relevante features van het audiosignaal te weten te komen. Uit een muziekbestand worden 68 features geëxtraheerd ('feature extraction'), die het netwerk gebruikt om de classificatie te realiseren. Uit mijn proof of concept bleek dat de toepassing bruikbaar zou zijn: als het netwerk 10 genres beschouwt, haalt het een gemiddelde classificatie nauwkeurigheid van 68% en hoe minder genres het netwerk beschouwt, hoe hoger de classificatie nauwkeurigheid. Om te weten te komen welke van de 68 features de belangrijkste zijn, deed ik aan 'feature selection'. Hier kwam echter geen eenduidig resultaat uit voort.

1 Inleiding

In deze bachelorproeftekst zullen neurale netwerken worden toegelicht als een vorm van supervised machine learning, een tak binnen de artificiële intelligentie (Artificial Intelligence, kortweg AI). De voornaamste bron voor dit werk is het uitstekende online boek 'Neural Networks and Deep Learning', geschreven door Michael A. Nielsen [1]. Ook vele oefeningen die hierin aan bod komen, werden in de bachelorproeftekst verwerkt om een beter inzicht te verwerven in de materie.

1.1 Situering

Het doel van AI is om intelligente machines te creëren, die de menselijke intelligentie nabootsen [2]. Dit is een heel breed onderzoeksgebied, ontstaan eind jaren 1940.

De tak van supervised machine learning onderzoekt methoden om machines, zoals een computer, zelfstandig te laten leren op basis van een gelabelde training set. Een training set is een verzameling van mogelijke inputs voor de machine. Men spreekt van 'gelabeld' omdat voor iedere input ook de bijhorende output die men van de machine wenst, gekend is. De machine zal deze input-ouput-mappings als voorbeeld gebruiken, om zo zelfstandig te achterhalen (= leren) hoe die mappings gemaakt kunnen worden. De machine gaat als het ware op zoek naar een verborgen functie f in de gelabelde training set, die voor iedere input x de gewenste output $f(x)$ geeft. We hopen dat de functie die de machine zal leren, laat ons zeggen de functie g, een goede benadering is voor de functie f: $g \approx f$. Indien g een goede benadering is, geeft de machine ook voor een input y die niet in de training set zat, een (bij benadering) correcte output: $g(y) \approx f(y)$. Men zegt dan dat de machine in staat is om veralgemeningen te maken, wat steeds het uiteindelijke doel is van het leren.

In 1958 ontwikkelde Frank Rosenblatt het eerste perceptron (zie 2.1) [3]. Hiermee legde hij de basis voor neurale netwerken. Hij baseerde zich hiervoor op het werk van McCulloch en Pitts uit 1943.

1.2 Inspiratie uit de biologie

Neurale netwerken zijn geïnspireerd op de biologie, meer bepaald op zenuwstelsels. Een zenuwstelsel kan als een netwerk van zenuwcellen (ook wel neuronen genaamd) gezien worden. Een neuron ontvangt prikkels van verschillende inputs. Stel dat je een auto recht op jou ziet afstormen. Dit zal ongetwijfeld voor enorme prikkels zorgen in een aantal neuronen. Indien zulke prikkels sterk genoeg zijn om een neuron te activeren, zal het ook andere neuronen gaan prikkelen. Deze hele kettingreactie, begonnen bij het zien van de auto, zal uiteindelijk ons lichaam tot actie aanzetten, door bv. weg te rennen.

De computermodellen voor neurale netwerken gebruiken uiteraard geen echte neuronen, waardoor 'artificieel neuraal netwerk' (ANN) een correctere benaming is.

1.3 Toepassingsgebieden

ANNs zijn hoofdzakelijk geschikt voor het oplossen van classificatieproblemen en voor regressie.

Bij een classificatieprobleem tracht het netwerk te bepalen tot welke categorie (of klasse) de input behoort. Het aantal klassen is eindig. Een voorbeeld van een classificatieprobleem is gezichtsherkenning, over 10 personen bijvoorbeeld. De inputs zouden de afstand tussen de ogen, afstand tussen ogen en mond, en kleur van de ogen kunnen zijn. De verschillende klassen zouden de 10 personen zijn. In de context van de verborgen functie f uit paragraaf 1.1, heeft f dus een eindig bereik.

Bij regressie, daarentegen, tracht het netwerk mappings te maken tussen inputs en een oneindig bereik aan outputs. Een neuraal netwerk kan dus gebruikt worden om een continue functie mee te benaderen. Zo zou het netwerk bijvoorbeeld de functie x^2 kunnen leren op basis van voorbeeld-inputs x met bijhorende functiewaarde x^2. In de context van de verborgen functie f, heeft f dus een oneindig bereik, zoals bv. de verzameling van de reële getallen $\mathbb{R}$.

Doordat classificatieproblemen en regressie vaak voorkomen in de praktijk, kennen ANNs vele toepassingsgebieden. Typische voorbeelden zijn de medische sector (classificeren van kankers bv.) en de financiële sector (voorspellingen voor de beurs bv.).

1.4 Beperkingen en nadelen

Zoals hierboven beschreven, blijken ANNs enorm krachtig en breed toepasbaar. Men kan zich dan ook terecht afvragen waarom men niet overal en altijd neurale netwerken gebruikt. Dat komt omdat neurale netwerken ook een aantal beperkingen en nadelen kennen [4, paragraaf 13.4.2]:

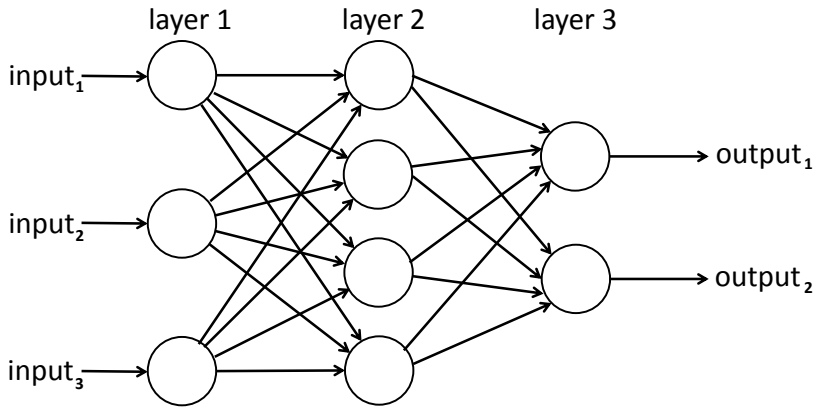
- Om een kwalitatief netwerk te verkrijgen, zijn veel training data nodig. Deze moeten tevens kwalitatief zijn. Zulke data bekomen is tijdrovend.
- Het trainingsproces is in het algemeen computationeel zwaar, waardoor het veel tijd kost.
- De output is niet gegarandeerd correct, wat inherent is aan machine learning.

Dit zorgt ervoor dat men vaak een traditioneel algoritme (niet op machine learning gebaseerd) zal verkiezen boven een ANN gebaseerd, indien beschikbaar uiteraard.

1.5 Globale werking van een ANN

In deze paragraaf wordt kort de globale werking van een artificieel neuraal netwerk (ANN) uitgelegd. Het is belangrijk dat de globale werking van een ANN gekend is om het overzicht te kunnen bewaren. In de volgende hoofdstukken zullen we het ANN in meer detail uitleggen.

Figuur 1 toont een eenvoudig ANN. Een netwerk is opgebouwd uit verschillende lagen ('layers'). Iedere laag bestaat uit een aantal neuronen, voorgesteld in de figuur door cirkels. De in- en output van een neuron zijn een getal, typisch tussen 0 en 1. De laag aan de linkerkant heet de input layer; die aan de rechterkant heet de output layer. Tussenliggende lagen heten hidden layers. De in- en output van het netwerk zijn eigenlijk vectoren. In de figuur bestaat de inputvector uit 3 componenten: $input_1$, $input_2$ en $input_3$. De outputvector



Figuur 1: Eenvoudig ANN.

bestaat uit 2 componenten: $output_1$ en $output_2$. In dit netwerk zijn er dus 3 lagen bestaande uit 3, 4 en 2 neuronen, van links naar rechts.

De input wordt laag per laag verwerkt; de outputvector van een laag vormt de inputvector voor de volgende laag. Vermits de data voorwaarts worden gepropageerd, spreekt men van een 'feedforward neural network'.

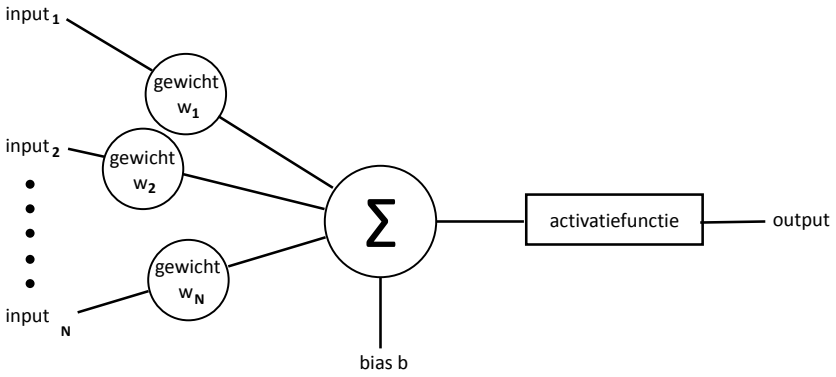
De output van een neuron wordt beïnvloed door een aantal parameters inherent aan dat neuron, namelijk zijn gewichten en bias. De parameters van alle neuronen samen vormen de parameters van het netwerk. Deze worden op willekeurige waarden geïnitialiseerd. Het leerproces zal de training data gebruiken om al die parameters aan te passen zodat de output van het netwerk, voor een bepaalde input, korter bij de gewenste output ligt. Zodra deze trainingsfase voltooid is, kan het netwerk ook gebruikt worden voor andere data; data die het nog niet eerder te zien kreeg.

2 Opbouw van een ANN

Eerst wordt de werking van een eenvoudig artificieel neuron, een 'perceptron' genaamd, uitgelegd en hoe hiermee een netwerk kan worden opgebouwd. Daaruit wordt vervolgens de nood aan het 'sigmoid neuron', een ander artificieel neuron, duidelijk.

2.1 Perceptron

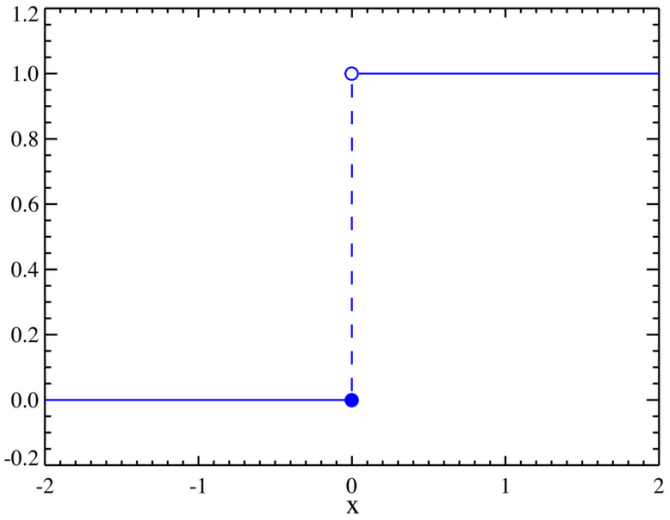
Een perceptron is een eenvoudig artificieel neuron. Ter visualisatie kan figuur 2 gebruikt worden. Zoals in de figuur te zien is, heeft een perceptron N inputs en één output. Dit zijn allemaal binaire waarden. Indien de output '1' is, zegt men dat het perceptron geactiveerd is. Verder is er één gewicht per verbinding en één bias per perceptron.



Figuur 2: Artificieel neuron.

De eerste stap in het berekenen van de output van een neuron, is het bepalen van de gewogen som van de inputs. Bij iedere input hoort een gewicht, dat hier gebruikt wordt. De inputs worden typisch x genoemd, en de gewichten w. In de volgende stap wordt er rekening gehouden met de bias, typisch b genaamd. De bias dient bij de gewogen som opgeteld te worden. Het resultaat wordt typisch z genoemd, dus $z = \sum_{i=1}^{N} w_i x_i + b$ of verkort, $z = wx + b$ (dot-product van w en x). De laatste stap is het toepassen van een zgn. activatiefunctie op z. Voor een perceptron is dat de stapfunctie S, zoals gedefinieerd in figuur 3. S mapt een input z naar 0 indien $z \leq 0$; naar 1 indien $z > 0$. Deze functie bepaalt dus of het perceptron al dan niet geactiveerd wordt. De waarde na toepassing van de activatiefunctie noemt men typisch de activatie a, dus $a = S(z) = S(wx+b)$. De activatiefunctie wordt soms ook de transferfunctie genoemd.

Om bovenstaande uitleg concreter te maken, wordt nu een klein voorbeeldje uitgewerkt. Stel dat $N = 3$ en dat $x = (1, 0, 1)$ de input(vector) is. Stel dat de bijhorende gewichten $w = (0.5, 2.0, 1.2)$ zijn. Inputs, outputs en gewichten worden typisch in zo'n vectornotatie geschreven. De gewogen som zou hier $wx = (1 \cdot 0.5 + 0 \cdot 2.0 + 1 \cdot 1.2) = 1.7$ zijn. Stel dat de bias $b = -1$, dan is $wx + b = 1.7 - 1 = 0.7$. Voor $a = S(0.7) = 1$ zou het perceptron dus wel geactiveerd worden. Stel echter dat $b = -3$, en dus $a = -2.7$, dan zou het perceptron niet geactiveerd worden. Stel dat we het eerste gewicht aanpassen



Figuur 3: Stapfunctie.

zodat $w = (2.5, 2.0, 1.2)$, dan is $wx + b = 0.7$ en zal het perceptron dus wel geactiveerd worden.

De conclusie is dat de gewichten de relatieve belangrijkheid van iedere input bepalen. De bias bepaalt hoe snel het perceptron geactiveerd wordt; hoe gevoelig het is.

2.2 Netwerk

Met behulp van neuronen, zoals perceptrons, kan een ANN worden opgebouwd. Figuur 1 uit hoofdstuk 1.5 illustreert dit.

Een ANN bestaat uit verschillende lagen ('layers'), op hun beurt opgebouwd uit neuronen. Het netwerk in de figuur bestaat uit 3 lagen. Er is steeds een input layer, die aan de linkerkant wordt afgebeeld. In dit geval bestaat deze uit 3 inputneuronen. Ook moet er steeds een output layer aanwezig zijn. In het netwerk in figuur 1 telt de output layer 2 neuronen. Verder zijn er nog een aantal tussenliggende lagen, hidden layers genaamd. Een eenvoudig ANN, zoals hierboven afgebeeld, heeft 1 hidden layer en wordt een ondiep netwerk ('shallow neural network') genoemd. In hoofdstuk 15 hebben we het over 'deep neural networks', die uit meerdere hidden layers bestaan. Het aantal neuronen in opeenvolgende hidden layers kan sterk verschillen.

In de context van neurale netwerken worden ook een aantal hyper-parameters gebruikt. Dit zijn parameters die door de gebruiker worden ingesteld. In tegenstelling tot de gewone parameters (gewichten en biases), wijzigen deze (vaak) niet meer. Het aantal lagen en het aantal neuronen per laag zijn hier voorbeelden van. Andere hyper-parameters zullen geleidelijk geïntroduceerd worden doorheen de tekst. In hoofdstuk 11 worden heuristieken toegelicht voor het bepalen van de waarden van de hyper-parameters.

Nu de terminologie en structuur van een ANN werden toegelicht, kunnen we ingaan op de betekenis van deze componenten. Dit doen we aan de hand van een voorbeeldtoepassing, die doorheen de tekst gebruikt zal worden [1]. In

hoofdstuk 5 bespreken we deze in detail. Die toepassing is het classificeren van een handgeschreven cijfer tussen 0 en 9. De dataset bestaat uit 70 000 afbeeldingen, met op iedere afbeelding één cijfer tussen 0 en 9. De afbeeldingen zijn 28×28 pixels. Een input voor het netwerk is zo'n afbeelding, herleid naar een vector van $28 \times 28 = 784$ grijswaarden (tussen 0 en 1). De output is een vector van lengte 10, waarbij een '1' staat op de index overeenstemmend met het cijfer, en voor de rest allemaal 0'en.

2.2.1 Input layer

In de context van bovenstaande voorbeeldtoepassing zal de input layer uit 784 neuronen bestaan. De input layer verschilt van andere lagen. Zoals figuur 1 voor een eenvoudig netwerk illustreert, is hier namelijk geen sprake van gewichten. Er zijn geen verbindingen naar de inputneuronen toe, net omdat ze de input representeren. Er wordt dus geen gewogen som gemaakt. Ook is er geen bias of activatiefunctie. Ze geven simpelweg de invoer door richting de volgende laag. Op die verbindingen staan wel gewichten uiteraard. In de voorbeeldtoepassing geeft een inputneuron dus gewoon de intensiteitswaarde naar de volgende laag door, zonder er iets mee te doen.

Doordat de input layer geen échte laag is, wordt deze laag in sommige literatuur niet meegeteld wanneer men over het totale aantal lagen spreekt. Men zou dan zeggen dat het netwerk uit 2 lagen bestaat.

2.2.2 Hidden layer

De inputs worden vervolgens naar ieder neuron in de volgende laag, de hidden layer in dit geval, gepropageerd. Merk op dat alle neuronen in eenzelfde laag verbonden zijn met alle neuronen in de volgende laag, wat kenmerkend is voor neurale netwerken. Ieder neuron in deze laag verwerkt zijn input zoals in 2.1 uitgelegd; niets bijzonders. Herinner dat de output van ieder perceptron een binaire waarde is.

Het aantal neuronen in de hidden layer heeft grote invloed op de kwaliteit van het netwerk. Er zijn echter geen vaste regels over hoe dit aantal bepaald moet worden; men moet hiermee experimenteren. Een veelvuldig gebruikte heuristiek echter, stelt dat het aantal hidden neurons afhangt van de complexiteit van de input-output relatie: hoe complexer de relatie, hoe groter het aantal hidden neurons [5].

Het aantal hidden layers is ook van groot belang voor een netwerk. Een hidden layer zorgt voor een niveau van abstractie. Door meerdere hidden layers te gebruiken, kan het netwerk dus alsmaar abstractere beslissingen nemen. Ook dit is slechts een heuristiek [6]. Een voorbeeld: stel dat we een netwerk zouden ontwikkelen dat bepaalt of er een auto in zijaanzicht te zien is in een afbeelding. De input layer zou de intensiteitswaarde per pixel voorstellen. In de eerste hidden layer zou deze informatie gebruikt worden om laag-niveau beslissingen te maken, zoals: is er links een velg te zien, is er links een band te zien, is er een deurklink te zien, is er een raam te zien? De volgende hidden layer zou deze info gebruiken om te bepalen of er links een wiel (bestaande uit een band en velg) te zien is, en of er een deur (bestaande uit een deurklink en raam) te zien is. De output layer zou die hoog-niveau beslissingen gebruiken om te bepalen of er al dan niet een auto (bestaande uit onder meer een wiel en deur) te zien is.

Voor relatief eenvoudige toepassingen volstaat 1 hidden layer, zoals ook in de voorbeeldtoepassing over handschriftherkenning. Nielsen [1] stelde vast dat 30 neuronen hier optimaal is voor de hidden layer.

2.2.3 Output layer

Nadat de hidden layer zijn inputs verwerkt heeft, propageert deze zijn outputs naar de volgende laag. In het netwerk in figuur 1 is dat de output layer, maar zoals reeds vermeld zouden er nog tussenliggende hidden layers kunnen zijn. De output layer is gewoon opgebouwd uit perceptrons; niets bijzonders dus.

Het aantal outputneuronen hangt af van het aantal mogelijke klassen waarin het netwerk een input kan klasseren. In de voorbeeldtoepassing zijn er 10 mogelijke outputs en dus 10 outputneuronen. De reden dat er per klasse een neuron gebruikt wordt, is, in essentie, omdat een neuraal netwerk patronen herkent. Het zou bv. kunnen dat het netwerk het moeilijk heeft om een 3 en een 5 (die een beetje op elkaar lijken) van elkaar te onderscheiden. Het netwerk zou dan als output (0,0,0,1,0,1,0,0,0,0) kunnen geven, waaruit dit zou blijken. Een andere optie zou een binaire representatie zijn, met 4 outputneuronen dus. Maar met zo'n representatie kan het netwerk geen combinatie van kandidaat-cijfers voorstellen. Dit maakt een binaire representatie ongeschikt.

2.2.4 Notatie

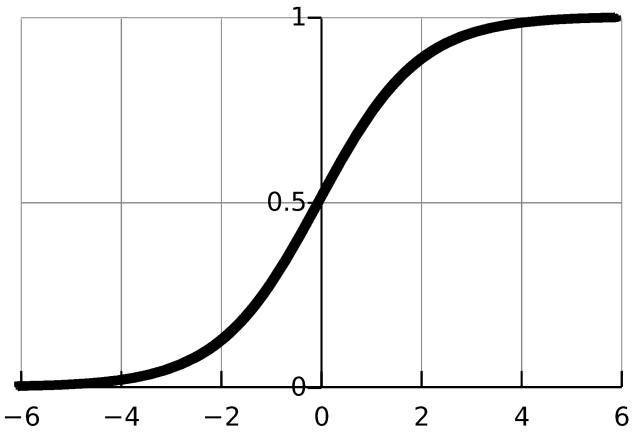
We gebruiken de notatie z om naar de gewogen som van een willekeurig neuron te verwijzen. Analoog verwijzen we met a, b en w_i naar de activatie, bias en het i-de gewicht van een willekeurig neuron. Wanneer het neuron waarnaar we willen verwijzen niet willekeurig, maar specifiek is, gebruiken we volgende notatie.

Met z_j^l bedoelen we de gewogen som van het j-de neuron uit de l-de laag. De analoge notatie a_j^l verwijst naar de activatie van datzelfde neuron. Het superscript l ligt tussen 1 en L, met L het totale aantal lagen in het netwerk. Zo verwijst a_j^L bv. naar de activatie van het j-de outputneuron. Ook voor de bias en een gewicht van het neuron voeren we een analoge notatie in. Zo stelt b_j^l de bias van het j-de neuron uit de l-de laag voor. Voor een gewicht hebben we 2 subscripts nodig: w_{jk}^l. Dit is het gewicht tussen het k-de neuron in laag $l-1$ en het j-de neuron in laag l. Deze vreemde indexatie wordt in paragraaf 4.1 verklaard, maar is voorlopig niet van belang.

2.3 Sigmoid neuron

In de praktijk zijn ANNs opgebouwd uit zgn. sigmoid neurons. Een perceptron netwerk is namelijk niet bruikbaar. Om dit te motiveren moet echter eerst de essentie van het leerproces worden toegelicht.

Herinner dat een neuron een gewicht per input heeft, en één bias. De gewichten en biases van alle neuronen samen vormen de parameters van ANN. Initieel zijn dit willekeurige waarden. Het netwerk wordt getraind m.b.v. een verzameling van voorbeelden. Een voorbeeld bestaat uit een input voor het netwerk, *input*, en de bijhorende gewenste output, *target*. Door kleine aanpassingen aan de gewichten en biases tracht het netwerk zijn parameters aan te passen, zodat zo veel mogelijk inputs de gewenste output geven.



Figuur 4: Logistic sigmoid.

Stel dat een netwerk, bv. om handgeschreven cijfers te classificeren, momenteel 10/15 inputs correct classificeert. Stel dat input x foutief geclassificeerd wordt. Het netwerk wijzigt één van de gewichten een heel klein beetje waardoor x nu correct geclassificeerd wordt. Men zou verwachten dat ondanks deze kleine wijziging de (meeste) andere inputs nog steeds correct geclassificeerd worden, net omdat het om een kleine wijziging gaat. Dit blijkt in de praktijk echter niet het geval.

De vorm van de stapfunctie, zie figuur 3, vormt de bron van het probleem. Stel dat in één van de perceptrons $z = wx + b = -0.05$, waardoor $a = S(z) = 0$. Stel dat een kleine wijziging van één van de gewichten zorgt dat $z = +0.01$. De waarde van z verschilt amper, maar $a = 1$, wat de tegengestelde output is en dus enorm verschilt. Hiermee zou het duidelijk moeten zijn dat een kleine wijziging van een parameter van één perceptron een grote impact kan hebben op het gedrag van het netwerk. Dit zorgt ervoor dat perceptrons niet geschikt zijn voor een netwerk dat moet kunnen leren.

Een sigmoid neuron verschilt alleen in activatiefunctie van een perceptron. De gebruikte activatiefunctie is de 'logistic sigmoid', σ genoteerd. Deze wordt in figuur 4 afgebeeld en is gedefinieerd als $\sigma(z) = 1/(1 + e^{-z})$. In tegenstelling tot de stapfunctie kan de output van de logistic sigmoid reële waarden tussen 0 en 1 aannemen. De logistic sigmoid behoort tot de familie van de sigmoid activatiefuncties. Ze hebben allemaal dezelfde vorm. Een andere sigmoid activatiefunctie is de tangens hyperbolicus, die in paragraaf 9.1 besproken wordt. Door hun geleidelijke vorm zal een kleine wijziging in één gewicht van een neuron, voor een kleine wijziging in de output van het neuron zorgen. Ter illustratie worden dezelfde waarden als in voorgaande paragraaf gebruikt: $\sigma(-0.05) = 0.4875$ en $\sigma(+0.01) = 0.5025$; een klein verschil dus. Hierdoor kan slechts een kleine wijziging in het gedrag van het netwerk te weeg worden gebracht, wat het leren mogelijk maakt.

Voor de logistic sigmoid geldt: $\lim_{z\to-\infty} \sigma(z) = 0$ en $\lim_{z\to\infty} \sigma(z) = 1$. Beschouw één sigmoid neuron, wiens output $z = wx + b \neq 0$. Stel dat we de gewichten en de bias met een constante $c > 0$ vermenigvuldigen. Het zou

duidelijk moeten zijn dat z dan met factor c in absolute waarde toeneemt; het teken blijft ongewijzigd. Bijgevolg:

1. Als $c \to \infty$ en $z > 0$, dan $cz \to \infty$ en $\sigma(cz) = 1$.
2. Als $c \to \infty$ en $z < 0$, dan $cz \to -\infty$ en $\sigma(cz) = 0$.

Dit komt overeen met het gedrag van een perceptron.[1] Uiteraard kan dit idee worden uitgebreid naar een volledig netwerk, wat het verband tussen een sigmoid neural network en een perceptron neural network aangeeft.

Een sigmoid neural network lijkt dus erg op een perceptron netwerk, maar is door de geleidelijke vorm van de sigmoidfunctie wel in staat om te leren. In de literatuur wordt een sigmoid neural netwerk soms een multi-layer perceptron (MLP) netwerk genoemd, wat erg verwarrend kan zijn.

2.4 Softmax en lineaire activatiefuncties

Een activatiefunctie wordt ook wel eens een transferfunctie genoemd. Tot nog toe hebben we 2 mogelijke activatiefuncties beschouwd: de stapfunctie en de logistic sigmoid. De gekozen activatiefunctie is ook een hyper-parameter. Hieronder worden nog twee populaire activatiefuncties uitgewerkt. In hoofdstuk 9 worden er nog twee beschreven.

2.4.1 Softmax activatiefunctie

Indien we met sigmoid neuronen werken, is het interpreteren van de outputactivaties van het netwerk niet eenvoudig. Beschouw de hoogste outputactivatie a_r^L, i.e. die outputactivatie die het kortst bij 1.0 ligt. We zouden de overeenkomstige klasse, de r-de klasse dus, als resultaat kunnen beschouwen. Het gevolg hiervan is dat er geen rekening wordt gehouden met de activaties voor de andere klassen. Indien sommige van die activaties in de buurt van a_r^L liggen, is het netwerk duidelijk niet zo zeker over de bepaalde klasse.

Waar we dus eigenlijk in geïnteresseerd zijn, is het aandeel van a_r^L tegenover alle outputactivaties. Hoe groter dat aandeel, hoe zekerder het netwerk is over de bepaalde klasse. Als we dat aandeel voor iedere outputactivatie bepalen, verkrijgen we dus een kansdistributie. Meer formeel zouden we iedere outputactivatie a_j^L m.b.v. de activatiefunctie P bepalen, met P als volgt gedefinieerd:

$$a_j^L = P(z_j^L) = \frac{\sigma(z_j^L)}{\sum_k \sigma(z_j^L)}$$

met k het aantal neuronen in de outputlaag.

In de praktijk wordt niet bovenstaande activatiefunctie gebruikt, maar wel [5]:

$$a_j^L = P(z_j^L) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

met k het aantal neuronen in de outputlaag. Deze activatiefunctie wordt de 'softmax' genoemd en wordt uiteraard op alle neuronen in de outputlaag toegepast, waardoor men van een 'softmax output layer' spreekt. De reden dat

[1] Alleen voor het geval $z = 0$ geldt dit niet. Dan is $\sigma(z) = 0.5$, ongeacht c.

de exponentiële functie gebruikt wordt, en niet de sigmoid, wordt zo meteen gemotiveerd.

Als we z_j^L verhogen, zal a_j^L verhogen. Vermits de activaties steeds tot 1 sommeren, heeft dit als gevolg dat de overige activaties a_k^L allemaal zullen afnemen. Dit zou zeer logisch moeten lijken. Formeel komt dit neer op: $\partial a_j^L / \partial z_j^L > 0$ en $\partial a_j^L / \partial z_k^L < 0$, wat we als volgt kunnen bewijzen:[2]

$$\frac{\partial a_j^L}{\partial z_j^L} = \frac{e^{z_j^L} \cdot \sum_k e^{z_k^L} - e^{z_j^L} \cdot e^{z_j^L}}{(\sum_k e^{z_k^L})^2} = \frac{e^{z_j^L} \cdot \sum_k e^{z_k^L}}{(\sum_k e^{z_k^L})^2} - \frac{e^{z_j^L} \cdot e^{z_j^L}}{(\sum_k e^{z_k^L})^2}$$
$$= a_j^L - (a_j^L)^2 = a_j^L(1 - a_j^L) \ > 0 \text{ want } 0 \leq a_j^L \leq 1$$

$$\frac{\partial a_j^L}{\partial z_k^L} = \frac{0 \cdot \sum_k e^{z_k^L} - e^{z_j^L} \cdot e^{z_k^L}}{(\sum_k e^{z_k^L})^2} = -a_j^L \cdot a_k^L < 0$$

Men kan zich afvragen waar de naam 'softmax' eigenlijk vandaan komt. Beschouw daartoe de meer algemene vorm:

$$a_j = \frac{e^{cz_j}}{\sum_k e^{cz_k}} \text{ met c een positieve constante}$$

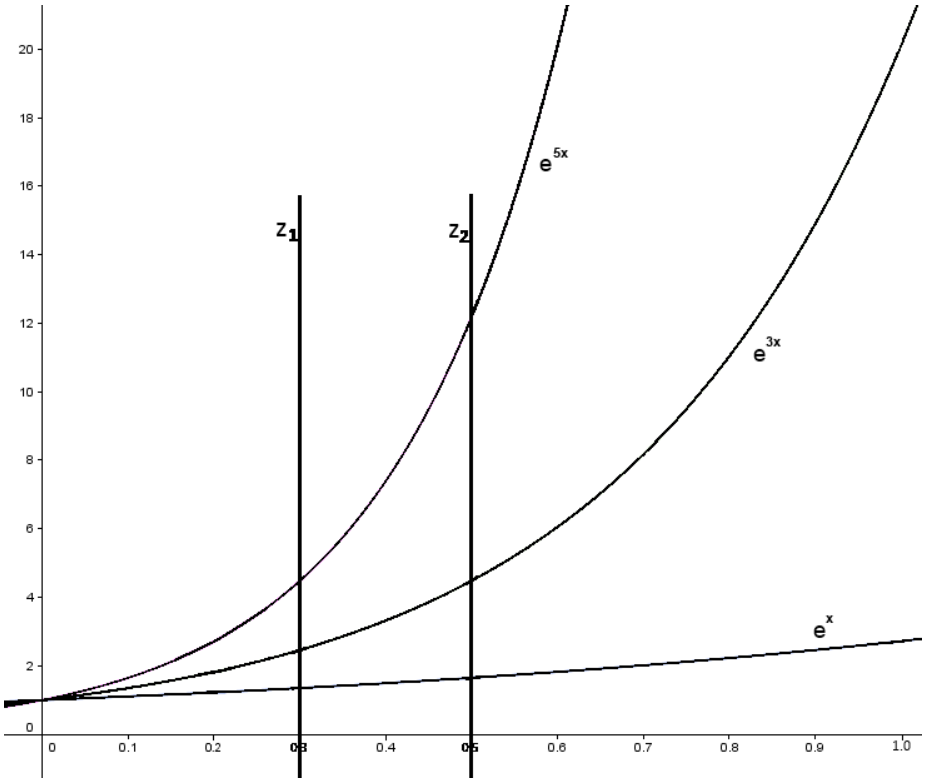
Het geval $c = 1$ komt overeen met bovenstaande definitie van de softmax. De exponentiële functie e^{cz_j} stijgt sneller naarmate c groter wordt. Figuur 5 illustreert dit voor $c = 1, c = 3$ en $c = 5$. Stel dat $z_1 = 0.3$ en $z_2 = 0.5$ (zie figuur). Het is duidelijk dat naarmate c toeneemt, het verschil tussen z_1 en z_2 exponentieel toeneemt. Het gevolg hiervan is dat a_2 een alsmaar grotere fractie van het totaal ($= a_1 + a_2$) inneemt. In limiet, $c \to \infty$, zal de maximale waarde, a_2 in dit voorbeeld, naar 1 gaan en de overige waarden, a_1 in dit voorbeeld, allemaal naar 0. Men zegt daarom dat deze functie het maximum berekent op een "harde" manier. De softmax kan als het 'softened maximum' gezien worden: deze bepaalt ook het maximum, maar de onderlinge verschillen worden minder extreem, "zachter", voorgesteld.

Bovenstaande uitleg geeft al een beetje aan waarom de exponentiële functie wordt gebruikt, en niet de sigmoid bijvoorbeeld. Stel dat er 3 gewogen sommen $z_1 = 4.0$, $z_2 = 5.0$ en $z_3 = 6.0$ zijn. Uit de vorm van de exponentiële functie zou het duidelijk moeten zijn dat $|P(z_1) - P(z_2)|$ veel kleiner is dan $|P(z_2) - P(z_3)|$. Meer algemeen vergroot de exponentiële functie het onderlinge verschil van steeds grotere z's. Bij de sigmoid is dat net omgekeerd. Uit de afgeplatte vorm van de sigmoid blijkt duidelijk dat het onderlinge verschil van steeds grotere z's, steeds kleiner wordt. Door het gebruik van de exponentiële functie worden de delen van de z's t.o.v. het geheel dus wat extremer gemaakt.

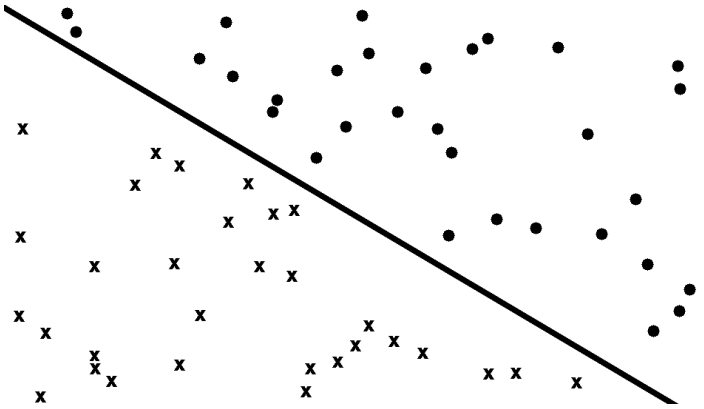
2.4.2 Lineaire activatiefunctie

Nog een vaak gebruikte activatiefunctie is de lineaire activatiefunctie $I(z_j) = z_j$. Deze wordt ook 'de identiteitsfunctie' genoemd. De output van I wordt niet beperkt en ligt dus in $]-\infty; +\infty[$. De identiteitsfunctie wordt in de outputlaag gebruikt wanneer een neuraal netwerk voor regressie opgebouwd wordt [7]. In zo'n netwerk is er slechts 1 outputneuron. Het is zeer belangrijk dat de andere

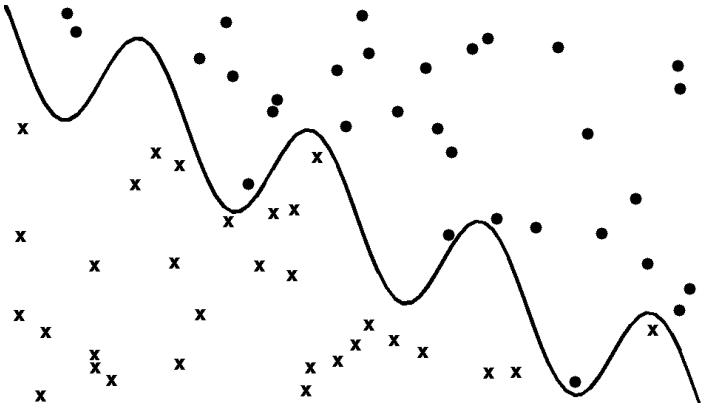
[2] Herinner dat $(e^x)' = e^x$. En dat $[f(x)/g(x)]' = [f'(x)g(x) - f(x)g'(x)]/[g(x)]^2$.



Figuur 5: De exponentiële functies e^x, e^{3x} en e^{5x}. Merk op dat het assenstelsel werd uitgerekt langs de horizontale as.



Figuur 6: Lineair probleem.



Figuur 7: Niet-lineair probleem.

lagen een sigmoid activatiefunctie gebruiken. Immers, indien alle neuronen de identiteitsfunctie gebruiken, kan het netwerk gereduceerd worden tot een netwerk bestaande uit 2 lagen: de inputlaag en een lineaire outputlaag. Iedere output is dan namelijk een lineaire combinatie van de inputs.

Een netwerk waarin alleen lineaire activatiefuncties gebruikt worden, is alleen geschikt voor lineaire problemen. Vermits de meeste problemen niet-lineair zijn, worden meestal niet-lineaire activatiefuncties gebruikt, waarvan de sigmoid het bekendste voorbeeld is. Daarom wordt de sigmoid ook wel eens 'de niet-lineariteit van het netwerk' genoemd. Het verschil tussen een lineair en niet-lineair probleem kan als volgt worden geïnterpreteerd. Beschouw de datasets in figuren 6 en 7, die beiden uit 2 klassen bestaan. Indien het een lineair probleem betreft, kan men een rechte bepalen die beide datasets van elkaar scheidt, zoals bij figuur 6. Voor een niet-lineair probleem voldoet een rechte niet, en is een niet-lineaire curve nodig, zoals bij figuur 7.

3 Het leerproces

Zodra het 'sigmoid feedforward neural network' opgebouwd is (zie vorig hoofdstuk), kan het netwerk beginnen leren.

Als een mens iets bij wilt leren, zal hij zich hierin trainen. Hetzelfde geldt voor neurale netwerken. Het neuraal netwerk leert uit voorbeeldata, zgn. gelabelde training data. Deze verzameling wordt de training set genoemd, als T genoteerd. Voor iedere input $x \in T$ is de bijhorende gewenste output y gekend. De gewichten en biases van alle neuronen in het netwerk vormen de parameters van het netwerk. Deze worden op willekeurige waarden geïnitialiseerd, meer bepaald volgens de standaard normale distributie (gemiddelde is 0, standaarddeviatie is 1). Het trainen houdt in dat deze waarden zullen worden aangepast opdat het netwerk voor zo veel mogelijk training data de gewenste output geeft. Het uiteindelijke doel van het trainen, is dat het netwerk veralgemeningen leert. Daardoor geeft het netwerk ook de gewenste output voor inputs die het niet eerder zag.

3.1 De kostfunctie

Om voor zo veel mogelijk training data de gewenste output te geven, moet het netwerk op zoek gaan naar zijn best mogelijke configuratie. Dit zijn de waarden van gewichten en biases waarvoor de activaties van het netwerk zo kort mogelijk bij de gewenste outputs liggen. De 'kost' geeft aan hoe goed het netwerk dit momenteel doet: hoe lager de kost, hoe beter; hoe hoger de kost, hoe slechter. Het doel van het trainen zal dus zijn om de kost te minimaliseren, door de bijhorende configuratie te vinden. Deze kost wordt bepaald door de kostfunctie C:

$$C(w,b) = \frac{1}{N} \sum_{x \in T} \frac{\|y(x) - a(x)\|^2}{2}$$

Hierbij zijn w en b de gewichten en biases van het volledige netwerk. N is het aantal training samples in de training set T. De som gaat over ieder training sample x, wat we in de toekomst simpelweg als $\sum_x$ zullen noteren. Verder stelt $a(x)$ de outputvector voor input x voor, en $y(x)$ de gewenste outputvector (targetvector).

Met $\|v\|$ wordt de norm van de vector $v = (v_1, v_2, \ldots, v_n)$ bedoeld. Dit wordt ook de lengte of magnitude van de vector genoemd. De norm is gedefinieerd als: $\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$. Indien v het verschil is van 2 vectoren, zoals het geval is voor de kostfunctie, komt de norm neer op de Euclidische afstand tussen beide vectoren. De kost kan bijgevolg ook als volgt geschreven worden:

$$C(w,b) = \frac{1}{N} \sum_x \frac{1}{2} \sum_i (y_i(x) - a_i(x))^2$$

Hierbij gaat i over alle neuronen in de outputlaag.

Zoals later zal blijken, hebben we de afgeleide van C nodig. Door het kwadraat wordt een factor 2 geïntroduceerd in die afgeleide. De factor $1/2$ in C zorgt dat deze de factor 2 opheft in de afgeleide. Hierdoor krijgt de afgeleide een eenvoudigere vorm. In paragraaf 6.4 zal dit duidelijker worden, maar ter motivatie van de factor $1/2$ werd het alvast vermeld.

Deze kostfunctie wordt in de praktijk vaak gebruikt en heet de 'mean squared error' (MSE) [1]. Het verschil tussen de activatie- en de targetvector voor sample x, $y(x) - a(x)$, noemt men de error voor dit sample. De kost voor één sample is $(\|y(x) - a(x)\|^2)/2$ of $0.5 \sum_i (y_i(x) - a_i(x))^2$ met i de output neuronen; de 'squared error' C_x. Deze wordt voor iedere sample berekend, en het gemiddelde stelt de kost C voor. Een andere notatie zou dus zijn:

$$C(w, b) = \frac{1}{N} \sum_x C_x$$

Merk op dat de kost eigenlijk afhangt van de activaties. Maar vermits de activaties uiteraard afhangen van de parameters van het netwerk (gewichten en biases), schrijft men de kostfunctie in functie van deze parameters.

Ter verduidelijking zal nu een klein voorbeeldje gegeven worden. Stel dat er slechts 2 testsamples zijn: x_1 en x_2. Stel dat $y(x_1) = (0, 1, 0)$ en $a(x_1) = (0.1, 0.8, 0.1)$. De kost van sample 1:

$$C_1 = \frac{\|(-0.1, 0.2, -0.1)\|^2}{2} = \frac{\sqrt{0.01 + 0.04 + 0.01}^2}{2} = \frac{0.06}{2} = 0.03$$

Stel dat $y(x_2) = (0, 0, 1)$ en $a(x_2) = (0.8, 0.1, 0.1)$. De kost van sample 2:

$$C_2 = \frac{\|(-0.8, -0.1, 0.9)\|^2}{2} = \frac{\sqrt{0.64 + 0.01 + 0.81}^2}{2} = \frac{1.46}{2} = 0.73$$

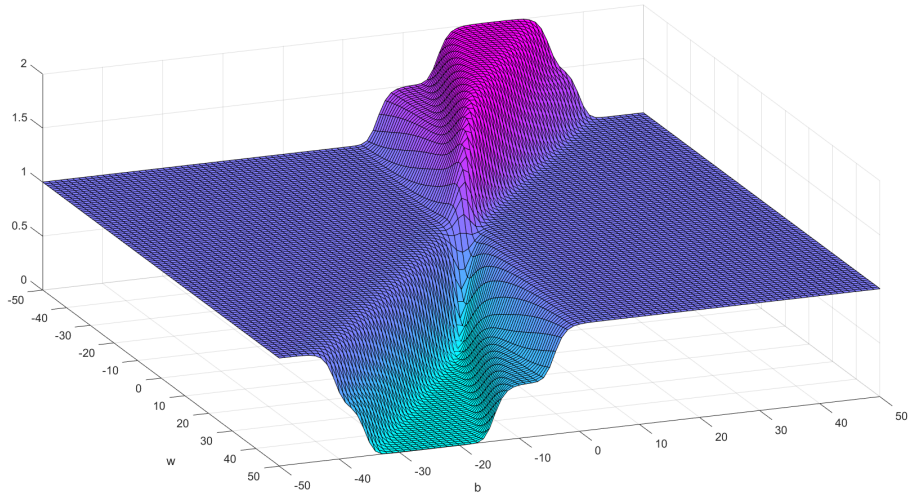
De uiteindelijke kost is dus:

$$C(w, b) = \frac{1}{2}(C_1 + C_2) = \frac{0.03 + 0.73}{2} = 0.38$$

Over de vorm van de kostfunctie weten we op voorhand niets. We weten alleen dat als alle gewichten en biases hun ideale waardes hebben, de kost minimaal is. Naarmate ze verder van hun ideale waardes afwijken, stijgt de kost. Laat ons ter illustratie een netwerk beschouwen, bestaande uit 1 input neuron, 1 output sigmoid neuron en geen hidden layer. Er zijn dus twee variabelen: het gewicht w en de bias b. Stel dat een input x in $[0.0; 1.0]$ ligt, en de gewenste output y is 0.0 als $x < 0.5$ en anders 1.0. Het spreekt voor zich dat zo'n verborgen functie enorm eenvoudig is. Als training set T genereren we een aantal gepaste tuples. De plot van de MSE kostfunctie wordt in figuur 8 getoond. We zien duidelijk dat de kost minimaal is als we w groot genoeg kiezen, en $b = -0.5 * w$ kiezen. Merk op dat de kostfunctie in haar minimum gelijk is aan 0.0, wat niet altijd zo is; de minimale kost kan ook veel groter zijn dan 0.0.

Stel dat het netwerk geleerd zou hebben dat $w = 50$ en $b = -25$. De output van het getrainde netwerk wordt dan gegeven door $a = \sigma(50 * x - 25)$. Voor input $x = 0.6$, die niet in T zit, is de outputactivatie $a = 0.99$, wat overeenkomt met de gewenste output van 1.0. Voor een netwerk met meer dan twee parameters kunnen we de kostfunctie niet visualiseren, aangezien dat in een dimensie hoger dan 3D is. Daar moeten we vertrouwen op het leeralgoritme om de kostfunctie te minimaliseren.

Er werd nog geen duidelijke motivatie gegeven voor de keuze van deze specifieke functie als kostfunctie. Voorlopig volstaat het te weten dat deze functie



Figuur 8: Plot van kostfunctie voor een neuraal netwerk bestaande uit twee parameters. De verborgen functie is 'is de input groter dan 0.5?'.

altijd positief is, en de kost toeneemt naarmate de activaties verder afwijken van de gewenste output. In paragraaf 6.1 worden nog een aantal nuttige eigenschappen van deze functie toegelicht.

3.2 Gradient descent

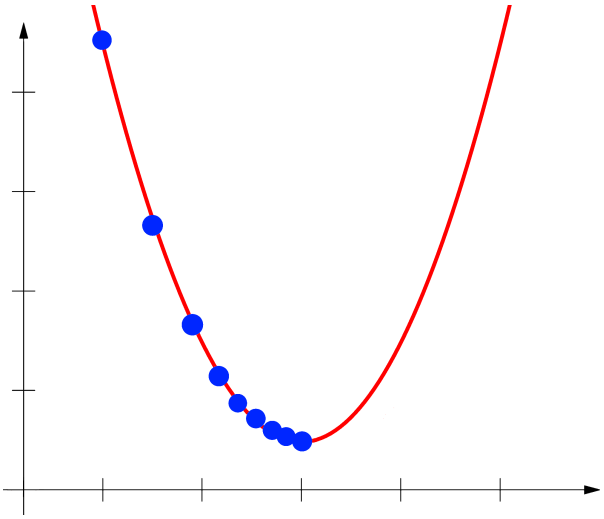
Door het gebruik van een kostfunctie C reduceert het trainen zich tot het bepalen van de waarden van de gewichten en biases waarvoor C haar minimum bereikt. Men zou dit analytisch kunnen doen: het minimum van een functie berekenen, gebruik makend van de afgeleide. Doordat de kostfunctie echter uit enorm veel variabelen (de gewichten en biases) bestaat, is deze aanpak niet haalbaar [1].

Het 'gradient descent' algoritme bestaat erin de parameters iteratief aan te passen, waarbij de kost in iedere iteratie verminderd wordt, zoals in figuur 9 geïllustreerd. Laat ons ervan uit gaan dat de kostfunctie een kwadratische vorm heeft. Dit doen we puur ter illustratie; zoals in vorige paragraaf uitgelegd, weten we niets over de vorm van de kostfunctie. De uitleg over gradient descent zal vanuit de calculus gebeuren, pas daarna wordt deze op neurale netwerken toegepast.

3.2.1 Eén variabele

We beginnen de uitleg vanuit de (eerste) afgeleide van een functie in één variabele, waar iedereen vertrouwd mee is. Ter illustratie beschouwen we de kwadratische functie $f(x) = x^2$, maar onderstaande redenering geldt voor eender welke functie. Figuur 10 toont de afgeleide van deze voorbeeldfunctie, $f'(x) = 2x$. We kunnen volgende observaties maken op basis van de afgeleide van een functie:

1. Het teken bepaalt of x zich links (negatief) of rechts (positief) van het minimum bevindt. Bijgevolg weten we of x moet toenemen (links) of



Figuur 9: Iteratieve berekening van het minimum (te beginnen vanuit het punt linksboven).

afnemen (rechts) om dichter bij het minimum te komen.

2. De absolute waarde bepaalt in welke mate x van het minimum afwijkt. De absolute waarde wordt immers groter naarmate x zich verder van het minimum bevindt.

Hierdoor zou het duidelijk moeten zijn dat we x als volgt kunnen aanpassen, opdat x naar het minimum toe gaat:

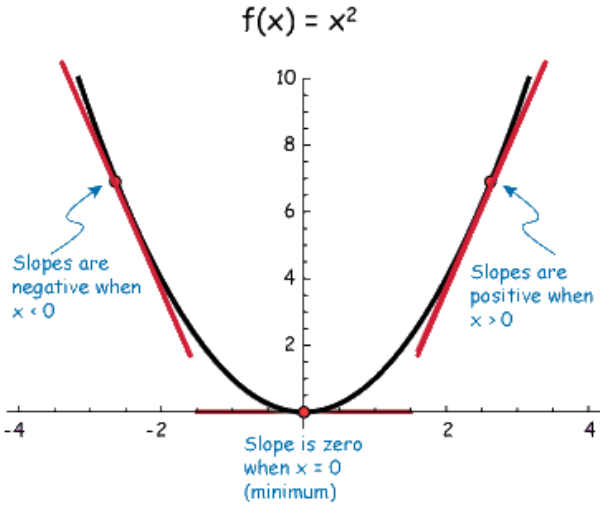
$$x \to x - \eta \cdot f'(x)$$

met η een positieve constante.

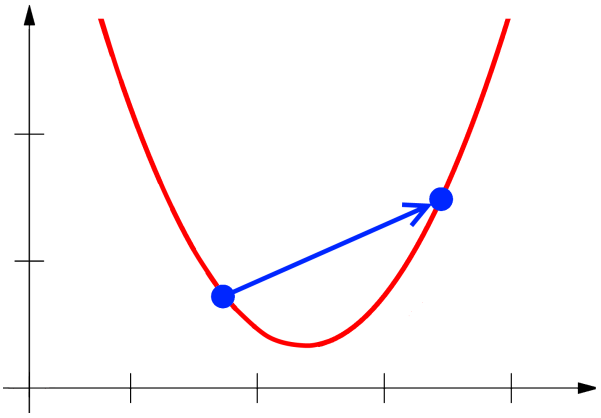
Met η kan de grootte van de aanpassing van x beïnvloed worden. De rol van η mag niet onderschat worden. Kiest men η te groot, dan kan het zijn dat de gemaakte stap zodanig groot is dat de kost zelfs gaat toenemen. Men springt als het ware voorbij het minimum, waardoor men van 'overshooting' spreekt. Dit wordt geïllustreerd door figuur 11. Kiest men η te klein, dan zullen zeer veel iteraties nodig zijn vooraleer de functie haar minimum bereikt.

3.2.2 Meerdere variabelen

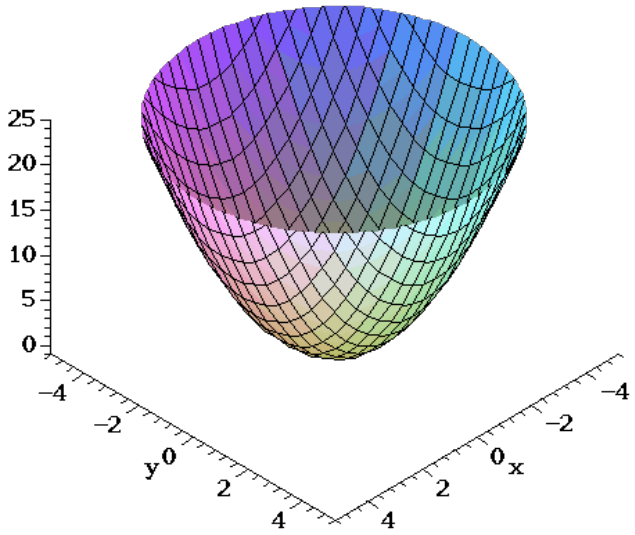
Het bovenstaande geldt ook voor hogere dimensies. Figuur 12 toont de kwadratische functie $f(x, y) = x^2 + y^2$ als voorbeeld van een functie in twee variabelen. Weet dat onderstaande redenering (opnieuw) geldt voor eender welke functie. Vermits er hier twee variabelen zijn, moeten ze allebei worden aangepast. Aangezien we nu de invloed van iedere variabele apart op de kost willen kennen, hebben we de partieel afgeleiden nodig. De partieel afgeleide van f naar x, $\partial f/\partial x$, geeft de informatie over de helling in de x-richting; de partieel afgeleide van f naar y, $\partial f/\partial y$, die in de y-richting. De update-regels worden dus als



Figuur 10: Afgeleide van kwadratische functie in één variabele.



Figuur 11: Stijgende kost bij te grote η.



Figuur 12: Kwadratische 3D-functie.

volgt:

$$x \to x - \eta \cdot \frac{\partial f}{\partial x} \text{ en } y \to y - \eta \cdot \frac{\partial f}{\partial y}$$

Calculus intermezzo — Partiële differentiatie: Bij partiële differentiatie dient men alleen de variabele waarnaar men afleidt als variabele te beschouwen; de andere variabelen moeten als constanten beschouwd worden. Voorbeeld: $\partial f/\partial x = 2x$ en $\partial f/\partial y = 2y$.

We hebben bovenstaande aanpak nu voor één en twee variabelen uitgewerkt. Het zou logisch moeten lijken dat die aanpak veralgemeend mag worden naar meerdere variabelen. Laat ons de gradiënt van $f(x, y)$ beschouwen: $\nabla f = (\partial f/\partial x, \partial f/\partial y)$. De gradiënt is dus de vector bestaande uit de partieel afgeleide van iedere variabele (van de functie). Beschouw $v = (v_1, v_2, \ldots, v_n)$, een vector van n variabelen, en $f(v)$. De meer algemene update-regel wordt dan:

$$v \to v - \eta \cdot \nabla f$$

3.2.3 Neurale netwerken

Bovenstaande 'gradient descent' aanpak uit de multivariabele calculus kan ook worden toegepast op neurale netwerken. Herinner dat de gewichten en biases van alle neuronen samen, de variabelen voorstellen. De functie die bij neurale netwerken beschouwd wordt, is natuurlijk de kostfunctie $C(w, b)$ (uit paragraaf 3.1), die gemakkelijkheidshalve herhaald wordt:

$$C(w, b) = \frac{1}{N} \sum_x C_x.$$

De update-regels voor een willekeurig gewicht w_k en bias b_l zijn dus:

$$w_k \to w_k - \eta \frac{\partial C}{\partial w_k} \text{ en } b_l \to b_l - \eta \frac{\partial C}{\partial b_l}$$

We hebben de partieel afgeleide ten opzichte van ieder gewicht en ieder bias nodig, wat verkort als ∇C (de gradiënt) genoteerd kan worden. Calculus zegt 'de afgeleide van een som = de som van de afgeleiden', waardoor:

$$\nabla C = \frac{1}{N} \sum_x \nabla C_x$$

De update-regels kunnen dus als volgt geschreven worden:

$$w_k \to w_k - \frac{\eta}{N} \sum_x \frac{\partial C_x}{\partial w_k} \text{ en } b_l \to b_l - \frac{\eta}{N} \sum_x \frac{\partial C_x}{\partial b_l}$$

Herinner het belang van de keuze van η, zoals in paragraaf 3.2.1 uitgelegd. In de context van neurale netwerken noemt men deze parameter de 'learning rate', vermits men hiermee beïnvloedt hoe snel het netwerk leert (i.e. de kost geminimaliseerd wordt). De learning rate is een hyper-parameter. Een heuristieke aanpak voor het bepalen van de learning rate is te beginnen met een kleine waarde, zoals 3.0. Indien blijkt dat het netwerk bijleert, kan de learning rate geleidelijk verhoogd worden. Indien het netwerk niet lijkt te leren, kan men de learning rate verlagen [1].

3.3 Mini-batch stochastic gradient descent

In principe weten we nu hoe een neuraal netwerk kan leren. Om de gewichten en biases te updaten (a.d.h.v. bovenstaande updateregels), moeten de partieel afgeleiden t.o.v. ieder gewicht en bias voor álle training samples berekend worden. Dit noemt men 'batch learning'.

In de praktijk is deze aanpak veel te langzaam, net omdat álle training samples nodig zijn voor de berekening van ∇C. Het is echter mogelijk om snel een goede benadering van de gradiënt te berekenen, door een beperkt aantal, $m < N$, willekeurige samples te gebruiken. Deze optimalisatie heet 'mini-batch learning', of 'stochastic gradient descent', aangezien de benadering als een stochastische variabele beschouwd kan worden. De grootte van de mini-batch, m, is een hyper-parameter. Ook met deze hyper-parameter moet men een beetje experimenteren om tot een geschikte waarde te komen. Meer formeel geldt:

$$\nabla C = \frac{1}{N} \sum_x \nabla C_x \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{Xj}$$

waardoor de update-regels als volgt worden:

$$w_k \to w_k - \frac{\eta}{m} \sum_{j=1}^{m} \frac{\partial C_{Xj}}{\partial w_k} \text{ en } b_l \to b_l - \frac{\eta}{m} \sum_{j=1}^{m} \frac{\partial C_{Xj}}{\partial b_l}$$

Indien een mini-batchgrootte van 1 gebruikt wordt, spreekt men van 'online learning'. Het zou duidelijk moeten zijn dat deze aanpak sneller is, maar niet

erg bruikbaar is. De partieel afgeleiden van individuele samples verschillen hier immers onderling te sterk voor. Het ene sample zorgt bv. voor een toename, terwijl het andere voor een afname zorgt. Door meerdere samples te combineren, zijn de globale richting en grootte van de aanpassing gekend.

3.4 Iteraties en epochs

We weten ondertussen dat een mini-batch gebruikt moet worden binnen het leerproces, en kennen de bijhorende update-regels. De uitleg over het leerproces is echter nog niet volledig.

In het leerproces moet de training data willekeurig worden ingedeeld in een aantal mini-batches (van dezelfde grootte). Stel dat de training data uit 1000 samples bestaan, dan zou men bv. 50 mini-batches kunnen maken met ieder 20 samples in. Het updaten van de gewichten en biases a.d.h.v. één zo'n mini-batch, zoals in paragraaf 3.3 beschreven, noemt men een 'iteratie'. In voorgaand voorbeeld zouden alle training data éénmaal gebruikt zijn na 50 iteraties. Dit noemt men één 'epoch'. Het leerproces bestaat uit meerdere epochs. Net zoals de mini-batch grootte, is ook het aantal epochs een hyper-parameter die experimenteel bepaald moet worden.

Het leerproces bestaat dus uit een aantal epochs, en iedere epoch bestaat uit een aantal iteraties (afhankelijk van de mini-batch grootte). We weten nu dus hoe een neuraal netwerk kan leren, tenminste indien we in staat zijn om de partieel afgeleiden te berekenen. Dit is echter niet voor de hand liggend. In hoofdstuk 6 wordt uitgelegd hoe dit m.b.v. backpropagation mogelijk is.

4 ANNs in de praktijk

In dit hoofdstuk worden een aantal onderwerpen toegelicht die hun praktisch nut hebben.

4.1 Notatie en implementatie

Herinner de notaties z en z_j^l uit paragraaf 2.2.4, waarmee we naar de gewogen som van één (al dan niet willekeurig) neuron verwijzen. Hetzelfde geldt voor de activaties, biases en gewichten. Zoals in paragraaf 2.2 werd uitgelegd, worden de outputs van de neuronen laag per laag verder gepropageerd naar de outputlaag.

Een handigere notatie is daarom $a^l = \sigma(z^l)$, met $z^l = w^l a^{l-1} + b^l$. Deze notatie is laag-gebaseerd. Zo stelt a^l een kolomvector voor, bestaande uit de activatie van ieder neuron in laag l. Analoog voor z^l en b^l. De activatiefunctie, σ hier als voorbeeld, wordt elementsgewijs toegepast.

De meest bijzondere component is w^l, die de gewichtenvector van ieder neuron in laag l bevat en dus een matrix is. Een component w_{jk}^l van de gewichtenmatrix heeft als betekenis: het gewicht om van het k-de neuron in laag $l-1$ naar het j-de neuron in laag l te gaan. Stel bijvoorbeeld dat laag $l-1$ uit 3 neuronen bestaat en laag l uit 4, dan heeft de gewichtenmatrix als dimensies 4×3. Het ordenen van de gewichten in die vorm is niet intuïtief, maar heeft als grote voordeel dat $w^l a^{l-1}$ d.m.v. een matrixvermenigvuldiging kan worden berekend i.p.v. over ieder van de neuronen te itereren en telkens het dotproduct tussen de gewichten- en activatievector te berekenen. In de praktijk wordt steeds zo'n matrix-gebaseerde implementatie gebruikt omdat vele libraries enorm snelle matrixvermenigvuldigingen voorzien.

4.2 Data verwerving

Zoals reeds uitgelegd, worden gelabelde training data, <*input, target*>, gebruikt om het netwerk te trainen. In de praktijk gaat men echter typisch als volgt te werk. Eerst bekomt men een gelabelde dataset. Vervolgens deelt men deze dataset op in 3: training data (zo'n 70%), test data (zo'n 15%) en validatie data (zo'n 15%). Op het gebruik van validatie data komen we in paragraaf 10.3 terug. Met de training data wordt het netwerk getraind, zoals reeds uitgelegd. Typisch laat men het netwerk de test data na ieder epoch evalueren. Dit houdt in dat men bepaalt hoeveel procent van die test data correct geclassificeerd worden door het netwerk. Indien het netwerk het na een aantal epochs ook voor deze data goed doet, kan men aannemen dat het netwerk in staat is om veralgemeningen te maken. Die test data werden immers niet gebruikt voor de training zelf.

Een grote moeilijkheid in de praktijk is het bekomen van die gelabelde dataset. De dataset moet kwantitatief en kwalitatief zijn. Men kan gebruik maken van reeds bestaande datasets. Een uitstekende bron hiervoor is de 'UCI Machine Learning Repository' [8]. Soms is er echter geen gelabelde dataset voor handen, en moet men deze zelf opstellen. Dit is een tijdrovende bezigheid.

4.3 Data preprocessing

Sommige gelabelde datasets zijn niet rechtstreeks geschikt als input voor neurale netwerken. Beschouw opnieuw de voorbeeldtoepassing voor het herkennen van handgeschreven cijfers. Het zou kunnen dat de dataset uit een folder met 10 subfolders bestaat. Iedere subfolder zou afbeeldingen van eenzelfde cijfer kunnen bevatten, door de naam van de subfolder aangegeven. Een resterende preprocessingstap is het inlezen van de kleurwaarden van iedere afbeelding en deze telkens omvormen naar een vector. Dit proces noemt men 'feature extraction': men extraheert kenmerken uit de data, zoals kleurwaarden. Nu hebben we de inputs ter beschikking. Ook moet het getal in de foldernaam nog in vectornotatie worden omgezet, bv. $3 \rightarrow (0,0,0,1,0,0,0,0,0,0)$. Zo hebben we ook de gewenste outputs ter beschikking.

Een preprocessing stap die in de praktijk veel gebruikt wordt, is het normaliseren van de input data. Er wordt per feature over alle inputs heen genormaliseerd, niet per inputvector zelf. Hierdoor wordt voor alle features als het ware eenzelfde schaal/eenheid gebruikt. Zonder normalisatie van de input data zou het netwerk appels met peren vergelijken, zoals men in de volksmond wel eens zegt. De 2 meest gebruikte technieken hiervoor zijn 'Z-score normalization' en 'Min-max normalization'.

Bij 'Z-score normalization' worden het gemiddelde μ en de standaarddeviatie σ gebruikt om input x te normaliseren:

$$x' = \frac{x - \mu}{\sigma}$$

De resulterende genormaliseerde dataset heeft als gemiddelde 0 en als standaarddeviatie 1. Bij 'Min-max normalization' worden de minimale en maximale waarde die de feature kan aannemen, x_{min} en x_{max}, gebruikt:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Alle waarden in de genormaliseerde dataset liggen dan in het interval [0;1].

We zullen beide technieken nu kort vergelijken met elkaar [9]. Min-max normalisatie heeft als voordeel, t.o.v. Z-score normalisatie, dat alle data begrensd is. Deze techniek is echter gevoelig voor uitschieters: indien er één opmerkelijk lage en één opmerkelijk hoge waarde is, zullen alle andere waarden zeer kort bij elkaar liggen. Daar heeft Z-score normalisatie dan weer geen last van. In de praktijk gebruikt men meestal min-max normalisatie. Merk ten slotte op dat op de data van de voorbeeldtoepassing reeds een min-max normalisatie werd toegepast; alle waarden liggen in [0;1].

5 Herkennen van handgeschreven cijfers

In dit hoofdstuk wordt met de voorbeeldtoepassing rond het herkennen van handgeschreven cijfers geëxperimenteerd. Vervolgens wordt een GUI rond deze toepassing geïmplementeerd, waarmee de gebruiker een zelf geschreven cijfer kan laten classificeren.

5.1 De toepassing

Het herkennen van handgeschreven cijfers werd in paragraaf 2.2 als voorbeeldtoepassing geïntroduceerd. Hier volgt wat meer uitleg.

De bedoeling is een neuraal netwerk te creëren dat in staat is om een cijfer tussen 0 en 9 te herkennen. Er wordt een bestaande gelabelde dataset gebruikt, 'The MNIST database of handwritten digits' [10] genaamd. Deze dataset bestaat uit een training dataset van 60 000 afbeeldingen en een test dataset van 10 000 afbeeldingen. Het aantal afbeeldingen per cijfer, van 0 naar 9, is respectievelijk: 6934, 7932, 6950, 7165, 6819, 6232, 6979, 7315, 6730 en 6944. Op ieder van die afbeeldingen staat één cijfer tussen 0 en 9. De afbeeldingen zijn 28×28 pixels. Een input voor het netwerk is zo'n afbeelding, herleid naar een vector van $28 \times 28 = 784$ grijswaarden (tussen 0 en 1). De afbeeldingen zijn ingescande handgeschreven cijfers van 500 Amerikanen [1]. In figuur 13 worden een aantal voorbeelden getoond.

Figuur 13: Een aantal afbeeldingen uit 'The MNIST database of handwritten digits'.

Nielsen [1], wiens boek de voornaamste bron van dit werk is, voorziet een eenvoudige implementatie van deze toepassing in Python 2.7 [11]. Deze maakt tevens gebruik van de third-party library Numpy [12]. We zullen dan ook zijn code gebruiken om een aantal experimenten rond deze toepassing uit te voeren. De code kan gevonden worden op zijn GitHub account [13]. Met de kennis die we uit voorgaande hoofdstukken hebben opgedaan, komen we tot Nielsens implementatie in 'network.py'. Backpropagation, een algoritme om de partieel afgeleiden te berekenen, werd daarin uiteraard ook geïmplementeerd. Dat wordt pas in hoofdstuk 6 toegelicht, maar zal niet van belang zijn bij het experimenteren.

De inputlaag van het neuraal netwerk bestaat uit $28 \times 28 = 784$ neuronen. De outputlaag bestaat uit 10 neuronen, wat overeenkomt met de 10 mogelijke cijfers. We gebruiken 1 hidden layer, kenmerkend voor een shallow network. Nielsen stelde vast dat 30 hidden neurons, een learning rate van 3.0, een mini-batch grootte van 10 en 30 epochs goede keuzes voor de hyper-parameters zijn. Heuristieken voor het bepalen van die waarden komen in hoofdstuk 11 aan bod.

Om het netwerk te trainen dient volgende code uitgevoerd te worden:

```
import mnist_loader
import network
```

```
tr_data, val_data, test_data = mnist_loader.load_data_wrapper()

net = network.Network([784, 30, 10])
net.SGD(tr_data, 30, 10, 3.0, test_data)
```

Met de functie *load_data_wrapper* wordt de MNIST database ingeladen en in 3 datasets opgedeeld. De 60 000 samples uit de MNIST training dataset worden door deze functie opgedeeld in 50 000 test samples en 10 000 validatie samples. De validatie dataset wordt voorlopig niet gebruikt, daar komen later op terug. De test dataset blijft ongewijzigd t.o.v. die van de MNIST database. Bij het aanmaken van het netwerk wordt het aantal neuronen per laag meegegeven, in een lijst: $[784, 30, 10]$. Om het gecreëerde netwerk te trainen wordt de functie *SGD* opgeroepen, wat voor 'Stochastic Gradient Descent' staat. De parameters zijn respectievelijk de training dataset, het aantal epochs, de mini-batch grootte, de learning rate en de test dataset.

Tijdens het trainen zal na iedere epoch worden uitgeschreven hoeveel cijfers het netwerk correct kan classificeren. Het idee hierachter werd in paragraaf 4.2 toegelicht. Het netwerk zou zo'n 95.50% correct moeten classificeren. Dit verschilt bij iedere uitvoering een klein beetje, aangezien de gewichten en biases op willekeurige waarden geïnitialiseerd worden.

5.2 Experimenten

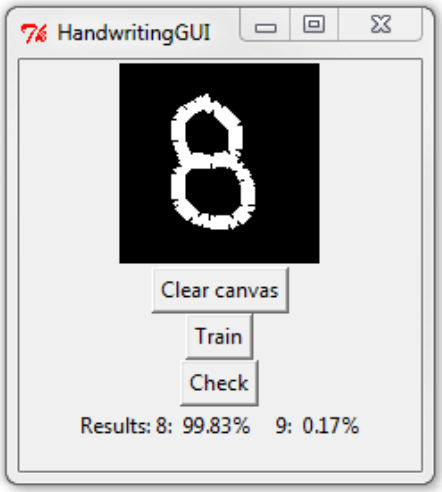
In deze paragraaf beschrijven we enkele experimenten die we hebben uitgevoerd omtrent de keuze van de hyper-parameters. We kijken steeds naar de impact van deze keuze op de classificatie-nauwkeurigheid.

Laat ons ten eerste met het aantal hidden neurons experimenteren. Stel dat we dit aantal van 30 naar 10 verlagen. Intuïtief verwachten we dat classificatie-nauwkeurigheid zou dalen. De resulterende nauwkeurigheid is zo'n 91%.[3] Desondanks significant minder hidden neuronen gebruikt werden, bleef de daling van de classificatie-nauwkeurigheid beperkt. Laat ons dit aantal daarom nog verder verlagen, naar 5 hidden neuronen. Nu haalt het netwerk nog slechts zo'n 83% nauwkeurigheid. Deze significante daling toont aan dat het netwerk nu onvoldoende parameters heeft om een accuraat model te kunnen vormen.

Laat ons het aantal hidden neuronen nu eens verhogen, naar 100. De overige parameters blijven onveranderd. Intuïtief verwachten we een stijging van de classificatie-nauwkeurigheid, aangezien het netwerk nu meer parameters ter beschikking heeft. Het netwerk haalt echter slechts zo'n 87% nauwkeurigheid. De meest waarschijnlijke verklaring voor deze observatie is dat 'overfitting' zich nu erg duidelijk manifesteert. Dit betekent dat het netwerk zo veel parameters ter beschikking heeft dat het alle input-ouput-mappings van de trainig data ongeveer kan memoriseren. Het netwerk leert nu echter geen veralgemeningen te maken, waardoor het de test data niet goed kan classificeren. Meer uitleg over overfitting volgt in paragraaf 10.1.

Het laatste experiment bestaat er in de learning rate te verhogen. Het oorspronkelijke aantal hidden neurons, 30, wordt gebruikt. Laten we de learning rate verhogen naar 50. De resulterende classificatie-nauwkeurigheid is zo'n 15%.

[3] Iedere beschreven classificatie-nauwkeurigheid in dit hoofdstuk werd bepaald door de mediaan van 5 uitvoeringen van het experiment.



Figuur 14: HandwritingGUI: voorbeeld van een classificatie.

Dit illustreert duidelijk dat het netwerk nu te grote aanpassingen/sprongen maakt, waardoor de minimale kost nooit bereikt kan worden. Figuur 11 illustreerde dit fenomeen reeds.

5.3 Toepassing: HandwritingGUI

In deze paragraaf wordt een kleine toepassing gerealiseerd die een GUI voorziet rond het eenvoudige netwerk in 'network.py'. De toepassing zal de gebruiker toelaten om zelf een cijfer te schrijven in een canvas, en dit cijfer door het netwerk te laten classificeren. Op die manier kan men zich een beter idee vormen over wat die 95.50% correctheid betekent. Men zal kunnen aanvoelen hoe duidelijk men moet schrijven. Voor deze toepassing zal ook de Python Imaging Library (PIL) [14] gebruikt worden. Als GUI framework wordt Tkinter [15] gebruikt, wat standaard bij Python zit.

Nielsens implementatie voorziet de functie *feedforward* in 'network.py' die de outputvector teruggeeft voor een gegeven inputvector. Deze gebruikt steeds de logistic sigmoid activatiefunctie, ook in de outputlaag dus. Om de output als kansdistributie te kunnen interpreteren, zullen we in de outputlaag de softmax activatiefunctie gebruiken. Hiervoor wordt naar paragraaf 2.4 verwezen. Op de precieze code gaan we niet in.

Figuur 14 illustreert de werking van de toepassing. Na het starten van de toepassing dient het netwerk getraind te worden, door op de 'Train'-knop te klikken. Voor het trainen worden de hyper-parameters uit paragraaf 5.1 gebruikt. Zodra het trainen voltooid is, kan men zelf tekenen in het canvas. Wanneer op de 'Check'-knop geklikt wordt, wordt een afbeelding gegenereerd op basis van het canvas. Deze wordt vervolgens door het netwerk geclassificeerd, door de aangepaste *feedforward* aan te roepen. De resultaten worden beneden in de GUI getoond.

Wat de implementatie betreft, beperken we ons tot het kort vermelden van de voornaamste elementen:

1. Er worden een Canvas (Tkinter), Image (PIL) en ImageDraw (PIL) gebruikt. Het 'canvas' en de 'image' zijn 112×112, wat 4 keer de grootte is van de afbeelding die het netwerk als invoer krijgt. Een canvas van 28×28 is gewoon te onhandig voor de gebruiker.

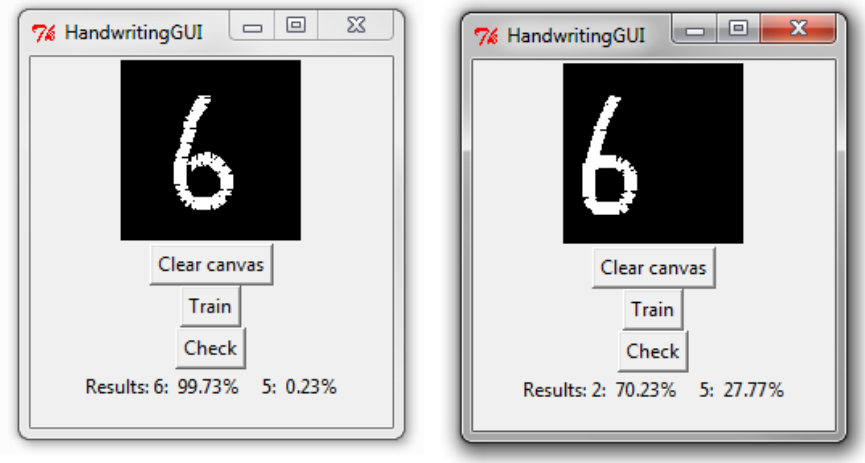
2. De inhoud van het canvas kan niet rechtstreeks als 2D-array van intensiteiten worden opgevraagd. Daarom wordt een 'imagedraw' gebruikt, waarmee op een 'image' getekend kan worden. Wanneer de gebruiker op het 'canvas' tekent, wordt hetzelfde op de 'image' getekend.

3. Wanneer op de 'Check'-knop geklikt wordt, wordt de inhoud van de 'image' opgevraagd. Deze 2D-array van intensiteiten wordt naar een 1D-array omgezet. Vervolgens wordt de functie *feedforward* gebruikt om de outputvector te genereren. De 2 hoogste waarden, met hun overeenkomstig cijfer, worden als resultaat getoond.

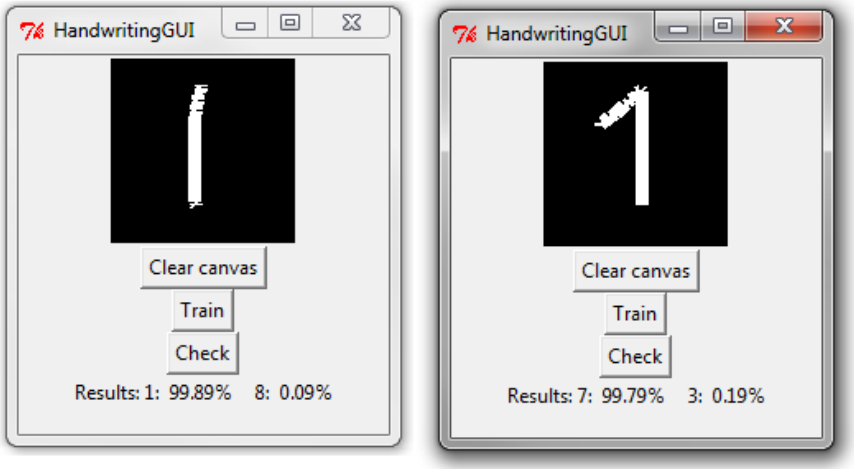
Door wat te experimenteren met deze toepassing kan men volgende twee opmerkelijke vaststellingen doen, waaruit het belang van geschikte voorbeelddata blijkt.

Ten eerste is de positie van het cijfer binnen het canvas cruciaal. Beschouw daartoe figuur 15. In beide gevallen werd de '6' praktisch exact hetzelfde geschreven, alleen de positie binnen het canvas verschilt. De classificatie is in het eerste geval perfect, maar in het tweede geval waardeloos. De verklaring hiervoor vinden we in de documentatie over de MNIST database [10]. Om tot de training data te komen, ging men als volgt te werk. Voor een ingescand cijfer werd een vierkante bounding box berekend, en naar 20×20 pixels geschaald. Vervolgens werd het zwaartepunt van die afbeelding berekend. De 20×20 afbeelding werd ten slotte in een 28×28 canvas geplaatst, zo, dat het zwaartepunt in het centrum van het canvas zou staan. Bij de '6' aan de linkerkant is dit (bij benadering) het geval, maar bij die aan de rechterkant ligt het zwaartepunt een behoorlijke afstand links van het centrum. Het netwerk kreeg dus nooit een '6' te zien waarvan het zwaartepunt ver van het centrum verwijderd was, waardoor het dit ook niet kon leren.

De tweede vaststelling is dat ook de schrijfstijl cruciaal is. Figuur 16 illustreert dit. De personen waarvan de handgeschreven cijfers gebruikt werden, waren allemaal Amerikanen. Zij schrijven hun '1' vaak zoals aan de linkerzijde van de figuur getoond wordt, terwijl Europeanen ze meestal schrijven zoals aan de rechterzijde. Vermits het netwerk de Europese schrijfstijl nooit te zien kreeg, kon het de '1' niet correct classificeren.



Figuur 15: HandwritingGUI: gevoeligheid voor de positie van het cijfer binnen het canvas. Links een 6 waarvan het zwaartepunt op het centrum ligt; rechts een 6 waarvan het zwaartepunt links van het centrum ligt.



Figuur 16: HandwritingGUI: gevoeligheid voor de schrijfstijl. Links een Amerikaanse 1, rechts een Europese.

6 Backpropagation

In hoofdstuk 3 werd uitgelegd hoe een ANN kan leren. We zagen dat het netwerk op zoek moest gaan naar die waarden van gewichten en biases waarvoor de activaties van het netwerk zo goed kort mogelijk bij de gewenste output liggen. Daarbij introduceerden we de kostfunctie, en gaven we aan dat we voor alle training samples de partieel afgeleiden moeten berekenen t.o.v. ieder gewicht en ieder bias. In dit hoofdstuk wordt uitgelegd hoe die partieel afgeleiden berekend kunnen worden.

6.1 Vereisten van de kostfunctie

In paragraaf 3.1 gaven we een beperkte motivatie voor de keuze van de MSE (mean squared error) als kostfunctie. De MSE voldoet nog aan een aantal andere eigenschappen, die bijdragen tot deze motivatie. Die eigenschappen, met hun bijhorend belang, zijn:

1. De kost is steeds positief, en naarmate de activaties en gewenste output verder uit elkaar liggen, neemt de kost toe. Dit is de definitie van een kostfunctie.
2. De kostfunctie is afleidbaar. Het gradient descent algoritme steunt hier op.
3. Met backpropagation kan alleen de partieel afgeleide van de kost voor één sample berekend worden. De afgeleide van de kostfunctie moet dus opgebouwd zijn uit de afgeleiden per sample. De afgeleide van de MSE is het gemiddelde van de afgeleiden van de kost per sample, en voldoet hier dus aan.
4. De kostfunctie moet geschreven kunnen worden als functie van de outputactiviaties.[4] Er mogen dus geen andere activaties in voorkomen bijvoorbeeld. Dit is een logische eigenschap van een kostfunctie, maar is wel van groot belang. Zoals immers zal blijken, steunt het backpropagation algoritme hier op. De MSE voldoet hier duidelijk aan.

Het backpropagation algoritme kan dus gebruikt worden met iedere kostfunctie die aan deze eigenschappen voldoet.

6.2 Introductie tot backpropagation

De term 'backpropagation' werd in 1962 door Frank Rosenblatt uitgevonden. Hij probeerde het backpropagation algoritme te ontwikkelen voor neurale netwerken die uit meerdere lagen bestaan, maar faalde. In 1982 slaagde onder meer David Rumelhart er alsnog in. Het duurde echter tot 1986 vooraleer dit de standaard werd binnen neurale netwerken [16]. David Rumelhart, Geoffrey Hinton en Ronald Williams brachten toen immers een paper [17] uit waarin de snelheidswinst duidelijk werd.

De eigenlijke definitie van de afgeleide van een functie f in x is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

[4] Natuurlijk staat de kostfunctie in functie van de gewichten en biases in het netwerk, maar de outputactivaties zijn op hun beurt een functie daarvan.

Stel dat we de partieel afgeleide van de kost van één specifiek sample tegenover een willekeurig gewicht w_j, dus $\partial C_x/\partial w_j$, willen berekenen. Een voor de hand liggende mogelijkheid, met h bijna 0, is dan:

$$\frac{\partial C_x}{\partial w_j} = C'_x(w_j) \approx \frac{C_x(w_j + h) - C_x(w_j)}{h}$$

Dit noemt men numerieke differentiatie. Om de partieel afgeleide te kennen, zou dus de kost voor het beschouwde sample moeten berekend worden voor (1) de huidige gewichten en biases ($C_x(w_j)$), en voor (2) de situatie waarbij w_j een minimale wijziging onderging ($C_x(w_j + h)$).

Laat ons zeggen dat we de partieel afgeleiden van C_x tegenover alle gewichten en biases willen berekenen, ∇C_x. Stel dat het aantal gewichten en biases tesamen n is. Ter berekening van ∇C_x, zou de output van het netwerk dus $n+1$ keer berekend moeten worden. Bovenstaande aanpak zou in de praktijk zo veel tijd kosten dat neurale netwerken onbruikbaar zouden zijn [1].

Aan het einde van volgende paragraaf zal het duidelijk zijn dat het backpropagation algoritme toelaat om ∇C_x veel sneller te berekenen. Het algoritme werkt, zoals de naam zegt, achterwaarts. Vertrekkend vanuit de output wordt een zgn. 'error' vanuit de output layer achterwaarts doorheen het netwerk gepropageerd, tot deze uiteindelijk bij de input layer terecht komt. Op basis van die error kunnen de partieel afgeleiden tegenover de gewichten en biases berekend worden. Het backpropagation algoritme steunt op 4 vergelijkingen, die hieronder uitgelegd worden.

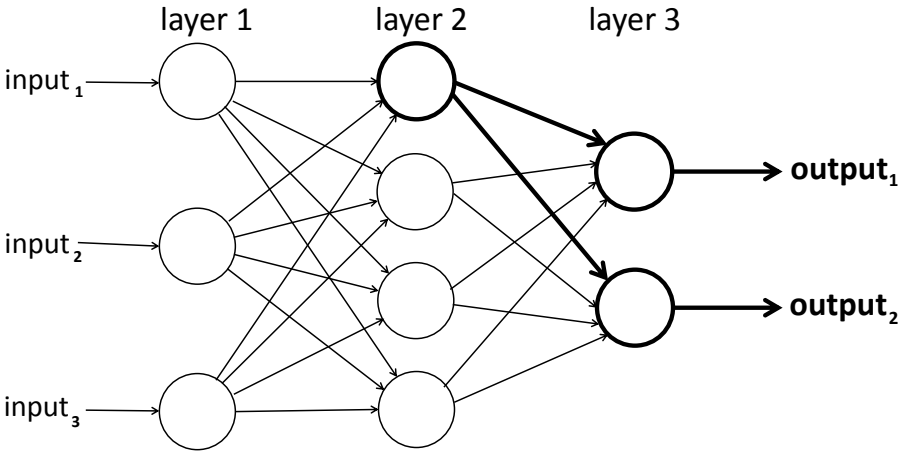
6.3 De vier vergelijkingen

We zijn geïnteresseerd in de partieel afgeleide van de kost van één sample tegenover een willekeurig gewicht of bias. Laat ons dit even schrijven als $\partial C_x/\partial w$ en $\partial C_x/\partial b$ respectievelijk. Laat ons echter vertrekken vanuit de meer algemene $\partial C_x/\partial z$, en later terug komen op $\partial C_x/\partial w$ en $\partial C_x/\partial b$. Zoals in paragraaf 3.2 werd toegelicht, zou $\partial C_x/\partial z$ intuïtief als 'de invloed/impact van z op C_x' gelezen kunnen worden.

Beschouw het ANN in figuur 17. Stel dat we z_1^2 een beetje wijzigen, namelijk $z_1^2 \to z_1^2 + \Delta z$. We zijn geïnteresseerd in de impact van deze wijziging op C_x. Als z_1^2 wijzigt, zal ook de bijhorende activatie, a_1^2, wijzigen. Dat zorgt voor een wijziging van de outputvector, a^3. Dat resulteert in een wijziging van de kost, die we ΔC_x noemen. Al deze wijzigingen werden in het vet aangeduid in de figuur. Het zou duidelijk moeten zijn dat ΔC_x afhangt van (1) de grootte van Δz en (2) de invloed/impact van z_1^2 op de kost. Stel dat we voor Δz een eenheidsgrootte gebruiken, zoals '1'. Dan kunnen we stellen dat ΔC_x volledig afhangt van $\partial C_x/\partial z_1^2$. Laat ons δ_j^l, 'de error', voor een willekeurige z_j^l definiëren als:

$$\delta_j^l = \frac{\partial C_x}{\partial z_j^l}$$

Calculus intermezzo — Kettingregel: Beschouw de functies $f(x)$ en $g(x)$. We definiëren de samengestelde functie $F(x) = (f \circ g)(x) = f(g(x))$. Wat is de afgeleide van $F(x)$? De kettingregel stelt dat $F'(x) = f'(g(x)) \cdot g'(x)$. Laat ons



Figuur 17: Wijziging van z_1^2 zorgt voor wijziging van de kost.

het bovenstaande nu schrijven a.d.h.v. partiële afgeleiden. Dan krijgen we:

$$\frac{\partial F(x)}{\partial x} = \frac{\partial F(x)}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

6.3.1 Eerste vergelijking

We beschouwen de error in neuron j van de laatste laag L, δ_j^L dus. Wegens bovenstaande definitie van 'de error' geldt: $\delta_j^L = \partial C_x / \partial z_j^L$. We weten: z_j^L beïnvloedt a_j^L en a_j^L beïnvloedt C_x. Dat is exact de intuïtie die we kunnen afleiden uit het toepassen van de kettingregel:

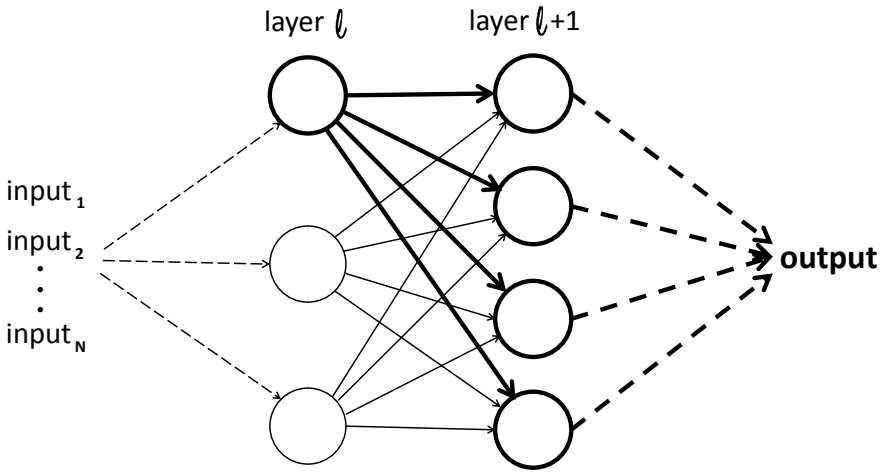
$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial C_x}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}$$

Herinner uit paragraaf 3.1 dat $C_x = \frac{1}{2} \sum_i (y_i(x) - a_i(x))^2$ met de som over de i output neuronen. Indien we dit afleiden naar de activatie van het j-de output neuron, a_j^L, krijgen we:

$$\frac{\partial C_x}{\partial a_j^L} = \frac{2 \cdot (y_j - a_j^L) \cdot (y_j - a_j^L)'}{2} = (y_j - a_j^L) \cdot (0 - 1) = a_j^L - y_j$$

Vermits a_j^L in functie van z_j^L staat (herinner $a = \sigma(z)$), geldt:

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$$



Figuur 18: Invloed van z_1^l op de kost.

Herinner uit paragraaf 2.3 dat $\sigma(z) = \frac{1}{1+e^{-z}}$. We bepalen diens afgeleide:

$$\sigma'(z) = -1 \cdot (1+e^{-z})^{-2} \cdot (1+e^{-z})' = \frac{1}{(1+e^{-z})^2} \cdot e^{-z} = \frac{(1+e^{-z})-1}{(1+e^{-z})^2}$$
$$= \frac{1}{(1+e^{-z})} - \frac{1}{(1+e^{-z})^2} = \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) = \sigma(z) \cdot (1-\sigma(z))$$

Deze berekening ligt niet voor de hand, maar heeft als voordeel dat de implementatie van $\sigma(z)$ herbruikt kan worden. Omdat $\sigma'(z)$ korter is qua notatie, zal in het verloop van deze tekst vaak die notatie gebruikt worden.

Laat ons δ^L definiëren als de vector van de error voor ieder neuron in laag L. Dan geldt:

$$\delta^L = (a^L - y) \odot (\sigma'(z^L)) \tag{1}$$

met $\odot$ de elementsgewijze vermenigvuldiging.

6.3.2 Tweede vergelijking

Het doel van de tweede vergelijking is de error achterwaarts te propageren. We moeten dus de error bepalen op basis van die uit de volgende laag; δ_j^l uitdrukken in functie van δ^{l+1}. Beschouw daartoe figuur 18 met 2 willekeurige opeenvolgende lagen l en $l+1$. De invloed van z_1^l op de kost hangt af van de invloed van z_1^l op alle z in laag $l+1$, en hun invloed op de kost (δ^{l+1}). Het toepassen van de kettingregel voor een willekeurige δ_j^l bevestigt dit:

$$\delta_j^l = \frac{\partial C_x}{\partial z_j^l} = \sum_k \frac{\partial C_x}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

Beschouw $\partial z_k^{l+1} / \partial z_j^l$. Intuïtief vragen we ons dus af hoe groot de invloed van z_j^l is op een z uit de volgende laag. We weten: z_j^l beïnvloedt a_j^l en a_j^l beïnvloedt

z_k^{l+1}. Voor dat eerste vonden we reeds de uitdrukking $\sigma'(z_j^l)$. Voor dat tweede zou het logisch moeten lijken dat het gewicht tussen beide neuronen bepalend is. Herinner dat $z_k^{l+1} = \sum_n w_{kn}^{l+1} a_n^l + b_k^{l+1} = \sum_n w_{kn}^{l+1} \sigma(z_n^l) + b_k^{l+1}$. De afgeleide hiervan tegenover één specifieke z_j^l geeft dan: $w_{kj}^{l+1} \sigma'(z_j^l)$; zoals verwacht. Als we dit nu samenvoegen met het bovenstaande, krijgen we:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \sigma'(z_j^l) \delta_k^{l+1} = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l)$$

Herinner de laag-gebaseerde forward propagation notatie (waarbij de implementatie de matrixvermenigvuldiging gebruikt) uit paragraaf 4.1. Overeenkomstig met die notatie hebben we nu de gewichten uit één kolom nodig, voor het berekenen van δ_j^l. Indien we echter de getransponeerde van de gewichtenmatrix gebruiken, moeten de gewichten uit één rij gebruikt worden. De vector bestaande uit de error in ieder neuron van laag l wordt dan:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2}$$

6.3.3 Derde vergelijking

Met 'vergelijking 1' kunnen de errors in de laatste laag berekend worden. Door 'vergelijking 2' herhaaldelijk toe te passen, kunnen ook de errors in de voorgaande lagen berekend worden. Zoals in het begin van deze paragraaf aangegeven, zullen we hieruit $\partial C_x / \partial w$ en $\partial C_x / \partial b$ berekenen. In dit deel bekijken we $\partial C_x / \partial b$.

Beschouw een willekeurig bias b_j^l. We weten: b_j^l beïnvloedt z_j^l en z_j^l beïnvloedt C_x. Om $\partial z_j^l / \partial b_j^l$ te bepalen, schrijven we $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$. Dit afleiden naar b_j^l geeft '1'. Dus:

$$\frac{\partial C_x}{\partial b_j^l} = \frac{\partial C_x}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \cdot 1 = \delta_j^l$$

of in vectornotatie:

$$\frac{\partial C_x}{\partial b^l} = \delta^l \tag{3}$$

6.3.4 Vierde vergelijking

Beschouw een willekeurig gewicht w_{jk}^l. We zullen nu $\partial C_x / \partial w_{jk}^l$ berekenen op basis van de error. Opnieuw weten we: w_{jk}^l beïnvloedt z_j^l en z_j^l beïnvloedt C_x. Om $\partial z_j^l / \partial w_{jk}^l$ te bepalen, schrijven we $z_j^l = \sum_n w_{jn}^l a_n^{l-1} + b_j^l$. Herinner 'de afgeleide van een som is de som van de afgeleiden'. Afleiden naar w_{jk}^l geeft a_k^{l-1}. Dus:

$$\frac{\partial C_x}{\partial w_{jk}^l} = \frac{\partial C_x}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \cdot a_k^{l-1}$$

of in matrixnotatie:

$$\frac{\partial C_x}{\partial w^l} = \delta^l (a^{l-1})^T \tag{4}$$

De resulterende matrix heeft dezelfde dimensies als de bijhorende gewichtenmatrix. Het element op index $[j][k]$ is dus de partieel afgeleide tegenover gewicht w_{jk}.

6.4 Het algoritme

Het backpropagation algortime zelf bestaat uit volgende stappen:

1. Bereken z^l en a^l voor iedere laag l.
2. Bereken de errors in de laatste laag, m.b.v. vergelijking 1.
3. Propageer de errors laag-gebaseerd achterwaarts door vergelijking 2 herhaaldelijk toe te passen.
4. Voor iedere laag: pas vergelijking 3 en 4 toe om de partieel afgeleiden tegenover de biases en gewichten in die laag te berekenen, uit de errors.

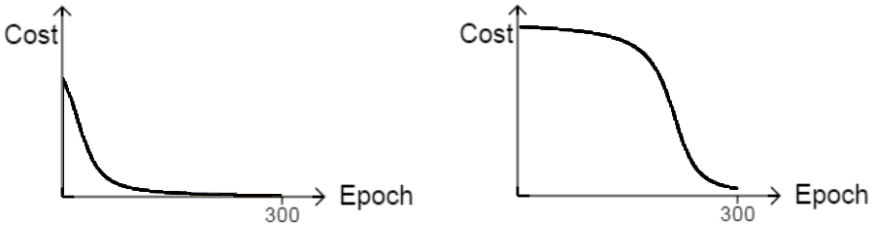
Het backpropagation algoritme berekent de gradiënt voor één sample; ∇C_x. Indien we het backpropagation algoritme dus toepassen op ieder van de samples in een mini-batch, kunnen de update-regels uit paragraaf 3.3 gebruikt worden. Hiermee is de cirkel rond; het volledige leerproces is gekend!

6.5 Een laatste woord

In de introductie tot backpropagation (paragraaf 6.2) werd uitgelegd dat numerieke differentiatie te traag zou zijn. Het vereist immers dat de output van het netwerk $n+1$ keer berekend moet worden, met n het aantal gewichten en biases samen.

De kost van het backpropagation algoritme daarentegen komt neer op 2 keer de output berekenen. Ten eerste moet de output voor de beschouwde input x berekend worden, zie stap 1 in het algoritme (paragraaf 6.4). Hierbij worden de inputs laag per laag richting de output gepropageerd. Herinner dat hiervoor matrixvermenigvuldigingen gebruikt worden. Dit is computationeel gezien de zwaarste component in de outputberekening. Herinner dat de errors laag per laag berekend worden, door de error uit de ene laag naar de voorgaande laag te propageren. Zoals vergelijking 2 aangeeft, gebeurt ook deze achterwaartse propagatie met een matrixvermenigvuldiging. Het zou dan ook logisch moeten lijken dat deze achterwaartse propagatie computationeel ongeveer even zwaar is als de voorwaartse. Hierdoor komen, computationeel gezien, de achterwaartse propagaties samen neer op één outputberekening, en dus op de tweede outputberekening in totaal [1].

Er worden in de bewijzen van de 4 vergelijkingen geen specifieke eigenschappen van de logistic sigmoid gebruikt. Het zou daarom duidelijk moeten zijn dat het backpropagation algoritme met eender welke activatiefunctie f gebruikt kan worden, zolang deze afleidbaar is. Waar nu σ' gebruikt werd, zal f' gebruikt moeten worden. Voor de rest wijzigt er niets.



Figuur 19: Snel leren (links); traag leren (rechts).

7 Cross-entropy en log-likelihood kostfuncties

In dit hoofdstuk worden twee kostfuncties voorgesteld die een mogelijke verbetering zijn ten opzichte van de mean squared error (MSE) uit paragraaf 3.1.

7.1 Cross-entropy kostfunctie

7.1.1 Traag leren en saturatie

We zouden intuïtief verwachten dat een neuraal netwerk sneller leert als het grotere fouten maakt, en minder sneller leert als het kleinere fouten maakt. Bij de mens is dat immers het geval, maar dat is niet altijd zo voor een neuraal netwerk. Voor een netwerk dat traag leert, duurt het lang vooraleer de kost daalt, ondanks het feit dat de activaties en gewenste output ver uit elkaar liggen.

Stel dat we één neuron hebben, waarvoor we willen dat het voor input 1.0 als output 0.0 geeft. We gebruiken de MSE als kostfunctie. Vergelijk het kostverloop links en rechts in figuur 19. Voor de situatie aan de linkerkant was het gewicht en bias zo gekozen dat de initiële activatie 0.82 was. Voor de situatie aan de rechterkant was de initiële activatie 0.98. Ondanks de grotere fout in de tweede situatie, leert het netwerk veel trager dan in de eerste situatie!

Op zich is het 'traag leren' een zeer groot probleem, dat overal in het netwerk kan optreden. Laat ons voorlopig op de outputlaag focussen en het probleem alleen daar proberen te vermijden. We komen hier in hoofdstuk 15 op terug.

Beschouw w^L_{jk} en b^L_j; een gewicht en de bias van het j-de outputneuron. Om de bron van het probleem te begrijpen, moeten we naar hun partieel afgeleiden kijken. Door enerzijds vergelijkingen 1 en 4 van backpropagation (zie paragraaf 6.3) te combineren, en anderzijds vergelijkingen 1 en 3, krijgen we:

$$\frac{\partial C_x}{\partial w^L_{jk}} = \frac{\partial C_x}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j}\frac{\partial z^L_j}{\partial w^L_{jk}} = (a^L_j - y_j)\cdot\sigma'(z^L_j)\cdot a^{L-1}_k \tag{5}$$

$$\frac{\partial C_x}{\partial b^L_j} = \frac{\partial C_x}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j}\frac{\partial z^L_j}{\partial b^L_j} = (a^L_j - y_j)\cdot\sigma'(z^L_j)\cdot 1 \tag{6}$$

Herinner de vorm van de (logistic) sigmoid uit paragraaf 2.3. Voor een activatie a^L_j die bijna 0 of 1 is, is de sigmoid erg vlak en is $\sigma'(z^L_j)$ dus bijna 0. Bijgevolg is de partiële afgeleide, zowel die tegenover een gewicht als die tegenover een bias, in dit geval bijna 0. Hierdoor heeft het toepassen van de update-regels (zie paragraaf 3.3) amper invloed; het gewicht of bias wijzigt amper. Bijgevolg

kan de activatie amper wijzigen. De activatie zit dus min of meer vast op zijn huidige waarde. Dit noemt men 'saturatie' of 'een gesatureerd neuron'.

Beschouw een neuron dat in de buurt van 1.0 gesatureerd is, maar eigenlijk 0.0 als output zou moeten geven, zoals in het voorbeeld aan het begin van deze paragraaf. We weten nu dat het vele epochs zal duren vooraleer de activatie uit de buurt van 1.0 raakt. Het zal bijgevolg lang duren vooraleer de kost daalt, waardoor het netwerk traag leert.

7.1.2 Nieuwe kostfunctie

We weten nu dat de $\sigma'(z_j^L)$ de bron van het probleem is. Laat ons dit verhelpen in de outputlaag. We zouden de $\sigma'(z_j^L)$ weg willen uit bovenstaande vergelijkingen 5 en 6, zodat we overhouden:

$$\frac{\partial C_x}{\partial w_{jk}^L} = (a_j^L - y_j) \cdot a_k^{L-1} \tag{7}$$

$$\frac{\partial C_x}{\partial b_j^L} = a_j^L - y_j \tag{8}$$

Op die manier is niet alleen het traag leren verholpen, maar hangt een update van een gewicht of bias alleen af van de afstand tussen de activatie en de gewenste output, zoals we intuïtief zouden verwachten.

Maar hoe kunnen we dat realiseren? Een voor de hand liggende oplossing zou zijn om de identiteitsfunctie als activatiefunctie te gebruiken, aangezien dan $\partial a_j^L / \partial z_j^L = 1$. Maar dat is alleen geschikt voor regressie, zoals in paragraaf 2.4.2 werd uitgelegd. Verder spreekt het voor zich dat we niets kunnen wijzigen aan $\partial z_j^L / \partial w_{jk}^L$ of $\partial z_j^L / \partial b_j^L$. De enige optie is dus een andere kostfunctie te gebruiken, waarvoor geldt:

$$\frac{\partial C_x}{\partial a_j^L} = \frac{a_j^L - y_j}{\sigma'(z_j^L)}$$

Zo kan deze $\sigma'(z_j^L)$ factor worden weggedeeld met die in vergelijkingen 5 en 6, waardoor we, zoals gewenst, vergelijkingen 7 en 8 bekomen.

Om de kostfunctie C_x te bepalen met bovenstaande afgeleide, moeten we $\partial C_x / \partial a_j^L$ integreren. Dat geeft ons:

$$C_x = -\sum_j [y_j \ln(a_j^L) + (1 - y_j) \ln(1 - a_j^L)] + \textit{constante}$$

Om het bovenstaande te bewijzen, zullen we C_x afleiden naar a_j^L:

$$C_x' = -[y_j \ln(a_j^L) + (1 - y_j) \ln(1 - a_j^L)]' = -\left(\frac{y_j}{a_j^L} - \frac{1 - y_j}{1 - a_j^L}\right)$$
$$= -\frac{y_j - y_j a_j^L - a_j^L + y_j a_j^L}{a_j^L (1 - a_j^L)} = \frac{a_j^L - y_j}{a_j^L (1 - a_j^L)} \overset{5}{=} \frac{a_j^L - y_j}{\sigma'(z_j^L)}$$

[5] De gelijkheid $\sigma'(z_j^L) = a_j^L (1 - a_j^L)$ werd in paragraaf 6.3.1 berekend.

Laat ons de kostfunctie veralgemenen naar meerdere samples:

$$C = \frac{1}{N}\sum_x C_x = -\frac{1}{N}\sum_x \sum_j [y_j \ln(a_j^L) + (1-y_j)\ln(1-a_j^L)] \tag{9}$$

Dit noemt men de cross-entropy kostfunctie.

Bijgevolg geldt:

$$\delta_j^L = \frac{\partial C_x}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{a_j^L - y_j}{\sigma'(z_j^L)} \cdot \sigma'(z_j^L) = a_j^L - y_j$$

waardoor we, zoals gewenst, bekomen:

$$\frac{\partial C_x}{\partial w_{jk}^L} = a_k^{L-1} \cdot \delta_j^L = (a_j^L - y_j) \cdot a_k^{L-1}$$

$$\frac{\partial C_x}{\partial b_j^L} = \delta_j^L = a_j^L - y_j$$

We hebben nu de $\sigma'(z_j^L)$ factor weggewerkt in de partieel afgeleiden. Maar in principe kan de factor a_k^{L-1} in de partieel afgeleide tegenover een gewicht alsnog die partieel afgeleide naar bijna 0 brengen. Een slimme keuze voor de kostfunctie kan dit probleem echter niet verhelpen. We zouden dan de factor a_k^{L-1} in de kostfunctie introduceren, zodat deze uiteindelijk zou kunnen weggedeeld worden. Hierdoor zou de gecreëerde kostfunctie niet meer voldoen aan de vierde vereiste van de kostfunctie (zie paragraaf 6.1), die stelt dat de kostfunctie alleen in functie van de outputactivaties mag staan.

Het interpreteren van de betekenis van de cross-entropy kostfunctie is veel moeilijker dan bij de MSE. Laat ons C_x beschouwen in het geval er slechts 1 outputneuron is: $C_x = -[y\ln(a) + (1-y)\ln(1-a)]$. Figuur 20 toont aan de linkerkant de kost in functie van de activatie voor het geval $y = 0.8$. Het minimum van de kost ligt duidelijk in $a = 0.8 = y$. Aan de rechterkant wordt een contour plot getoond (in functie van a en y). We zien dat, voor iedere waarde van y de kost toeneemt naarmate de activatie a verder van y ligt; voor $a = y$ bereikt de kost haar minimum. We kunnen formeel aantonen dat $C(a) = -[y\ln(a) + (1-y)\ln(1-a)]$ een minimum bereikt in y, met $y \in [0;1]$, als volgt. Ten eerste geldt dit triviaal voor de gevallen $y = 0$ en $y = 1$. Ten tweede kunnen we vanuit de calculus het geval $y \in]0;1[$ aantonen:

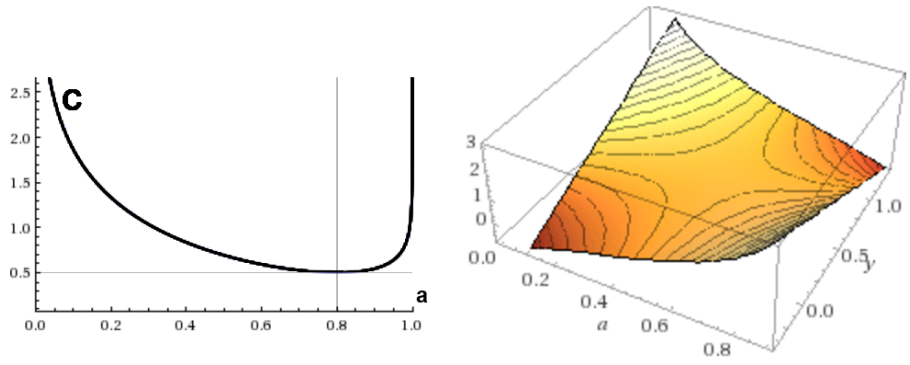
$$f'(a) = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) = -\frac{y - ya - a + ya}{a(1-a)} = \frac{a-y}{a(1-a)} = 0 \iff a = y$$

$$f''(a) = -\left(\frac{-y}{a^2} - \frac{1-y}{(1-a)^2}\right) = \frac{y}{a^2} + \frac{1-y}{(1-a)^2}$$

$$f''(y) = y + \frac{1}{1-y} \text{ met y in }]0;1[\ > 0 \ \Rightarrow \ \text{minimum in y}$$

Nu is het duidelijk dat deze kostfunctie aan de 4 vereisten (zie paragraaf 6.1) voldoet.

De cross-entropy kostfunctie in vergelijking 9 lost het probleem van traag



Figuur 20: Cross-entropy kost voor $y = 0.8$ (links); Cross-entropy contour plot (rechts).

leren op in de laatste laag. We weten ondertussen dat de $\sigma'(z)$ de bron van het probleem is. Voor de berekening van de error in voorgaande lagen gebruiken we vergelijking 2 van backpropagation (zie paragraaf 6.3.2), die alsnog $\sigma'(z)$ bevat. Hierdoor lost de slimme keuze van kostfunctie het probleem niet op in voorgaande lagen. Op dit probleem komen we terug in hoofdstuk 15.

7.1.3 Experiment en conclusie

Herinner het verloop van de kost bij een initiële activatie van 0.98 uit paragraaf 7.1.1, gebruik makend van de MSE kostfunctie. Dit wordt aan de linkerkant in figuur 21 opnieuw getoond. Als we echter de cross-entropy kostfunctie gebruiken, verloopt de kost zoals aan de rechterkant in die figuur getoond wordt.

Op Nielsens GitHub account [13] vinden we 'network2.py' terug. Dit is het vervolg op 'network.py', in paragraaf 5.1 geïntroduceerd. Vele optimalisaties, waaronder de cross-entropy kostfunctie, werden hierin door hem geïmplementeerd. Nielsen bracht ook een aantal wijzigingen aan om de code flexibeler in gebruik te maken. Voor het experimenteren met verschillende optimalisatietechnieken zullen we zijn code gebruiken.

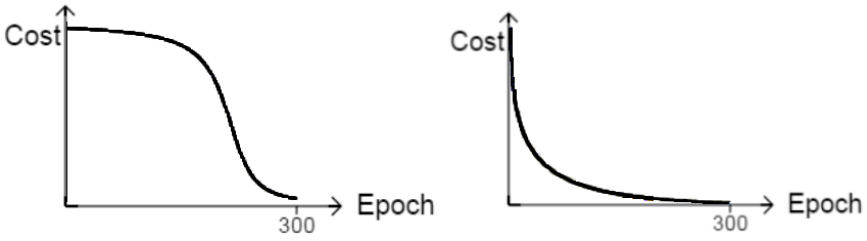
Laat ons volgende code uitvoeren:

```
import mnist_loader
import network2

tr_data, val_data, test_data = mnist_loader.load_data_wrapper()

net = network.Network([784, 30, 10], network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(tr_data, 30, 10, 0.5, test_data)
```

Vooraleer we de resultaten van de aanroep bespreken, zullen we even naar de code zelf kijken. Een eerste wijziging t.o.v. 'network.py' is dat bij het aanmaken van het netwerk een extra parameter beschikbaar is om de te gebruiken kostfunctie in te stellen. De opties zijn de nieuwe *CrossEntropyCost* en de oude *QuadraticCost* (MSE). Een tweede wijziging is de oproep van *large_weight_initializer()*.



Figuur 21: Gebruik van MSE (links); gebruik van cross-entropy (rechts).

In hoofdstuk 8 zullen we een beter manier zien om de gewichten te initialiseren, die default gebruikt wordt. Door *large_weight_initializer()* op te roepen, forceren we de tot nog toe gebruikte manier alsnog te gebruiken.

Voor het experiment in paragraaf 5.1 gebruikten we 3.0 als learning rate. We haalden toen zo'n 95.50% classificatie nauwkeurigheid. Hier gebruiken we een learning rate van 0.5. De geschikte learning rate is afhankelijk van de gebruikte kostfunctie. Nielsen stelde vast dat deze waarden goede keuzes waren. Voor de rest blijven de hyper-parameters ongewijzigd. Door in essentie alleen de kostfunctie te veranderen, halen we nu opnieuw zo'n 95.50% classificatie nauwkeurigheid. Voor onze voorbeeldtoepassing zien we dus amper impact van de keuze van de kostfunctie. Het netwerk doet het niet slechter, maar ook niet beter. Maar aangezien de cross-entropy kostfunctie zeer vaak gebruikt wordt en toelaat om 'traag leren' uit te leggen, was het zeker nuttig hier op in te gaan.

7.2 Log-likelihood kostfunctie

7.2.1 Definitie

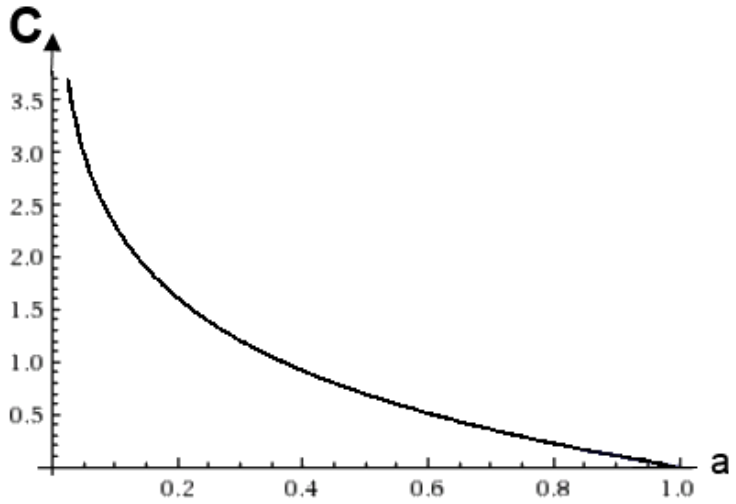
De log-likelihood kostfunctie is als volgt gedefinieerd:

$$C_x = -\ln(a_y^L) \quad \text{en} \quad C = \frac{1}{N}\sum_x C_x$$

Index y hierboven is die index in de targetvector waar een '1' staat. We geven een klein voorbeeld in de context van de voorbeeldtoepassing over het classificeren van handgeschreven cijfers. Stel dat een beschouwde input als een '7' zou geclassificeerd moeten worden, dan staat op index 7 van de targetvector een '1' en op de overige 9 indices een '0'. De kost wordt dan bepaald op basis van de activatie van het 7e outputneuron. Hoe korter a_7^L bij 1 ligt, hoe lager de kost moet zijn; hoe korter a_7^L bij 0 ligt, hoe hoger de kost moet zijn.

Beschouw het functieverloop van deze kostfunctie, zoals in figuur 22 getoond wordt. Dat bevestigt de hierboven beschreven eis over de kost. Het zou nu ook duidelijk moeten zijn dat aan alle 4 vereisten van de kostfunctie (zie paragraaf 6.1) voldaan is.

Het lijkt misschien vreemd dat deze kostfunctie slechts één specifieke outputactivatie gebruikt om de kost te definiëren. Herinner namelijk dat de MSE en cross-entropy kostfuncties over álle j outputactivaties sommeren. We kunnen



Figuur 22: Plot van de log-likelihood kostfunctie.

voor de log-likelihood echter ook zo'n vorm opstellen:

$$C_x = -\sum_j y_j \cdot \ln(a_j^L)$$

Hierbij is y_j de j-de waarde uit de targetvector y. Vermits voor alle y_j, op één na, geldt $y_j = 0$, reduceert deze uitdrukking tot bovenstaande $C_x = -\ln(a_y^L)$.

7.2.2 Gebruik

De log-likelihood kostfunctie wordt soms gebruikt in combinatie met een softmax outputlaag. Zoals we zo dadelijk zullen aantonen, voorkomt men zo het 'traag leren' in de outputlaag. De log-likelihood kostfunctie in combinatie met een softmax outputlaag is dus een beetje het equivalent van de cross-entropy kostfunctie in combinatie met een sigmoid outputlaag. In de praktijk is de log-likelihood kostfunctie niet erg populair. Men gebruikt meestal gewoon de cross-entropy kostfunctie, ook in combinatie met een softmax outputlaag.

We zullen nu aantonen dat de combinatie van deze kostfunctie met de softmax outputlaag geen last heeft van traag leren. Herinner $C_x = -\ln(a_y^L)$ en $\delta_j^L = \partial C_x / \partial z_j^L$. We moeten rekening houden met volgende twee mogelijkheden:

1. $j = y$:
   De kost staat in functie van de activatie van het beschouwde neuron, a_j^L. Voor het j-de outputneuron is de gewenste output dus '1'. Als we y als vector (targetvector) beschouwen, geldt dus $y_j = 1$.
2. $j \neq y$:
   De kost staat in functie van de activatie van een ander neuron, a_k^L. Voor het j-de outputneuron is de gewenste output dus '0'. Als we y als vector (targetvector) beschouwen, geldt dus $y_j = 0$.

In het eerste geval:

$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial C_x}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} = \frac{-1}{a_j^L} \cdot a_j^L(1 - a_j^L) = a_j^L - 1 = a_j^L - y_j$$

In het tweede geval:

$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial C_x}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L} = \frac{+1}{a_k^L} \cdot a_k^L \cdot a_j^L = a_j^L = a_j^L - 0 = a_j^L - y_j$$

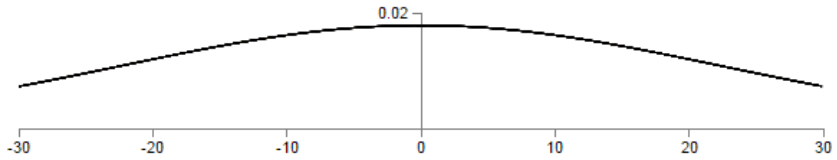
De partieel afgeleiden $\partial a_j^L / \partial z_j^L$ en $\partial a_k^L / \partial z_j^L$ werden in paragraaf 2.4.1 berekend.

We weten nu dus dat voor de error in de outputlaag geldt: $\delta_j^L = a_j^L - y_j$. Hieruit kunnen we de partieel afgeleide tegenover een bias en tegenover een gewicht bepalen. Door respectievelijk vergelijkingen 3 en 4 van backpropagation (zie paragraaf 6.3) te gebruiken, bekomen we:

$$\frac{\partial C_x}{\partial b_j^L} = \delta_j^L = a_j^L - y_j$$

$$\frac{\partial C_x}{\partial w_{jk}^L} = a_k^{L-1} \cdot \delta_j^L = a_k^{L-1}(a_j^L - y_j)$$

Nu is het duidelijk dat de log-likelihood in combinatie met de softmax outputlaag geen last heeft van 'traag leren'; alleen het verschil tussen de activatie en de gewenste output is van belang. Aangezien deze kostfunctie niet erg populair is, voeren we er geen experiment mee uit.



Figuur 23: Brede kansdistributie van z indien de gewichten volgens de standaard normale distributie geïnitialiseerd worden.

8 Initialisatie van de gewichten

Herinner uit hoofdstuk 3 dat de parameters van het neuraal netwerk op willekeurige waarden worden geïnitialiseerd. Voor ieder gewicht en ieder bias werd een willekeurige waarde gekozen volgens de standaard normale distributie (gemiddelde is 0, standaarddeviatie is 1). In dit hoofdstuk wordt een betere keuze toegelicht.

8.1 Probleem met oude initialisatie

We zullen eerst uitleggen wat het probleem is met de oude initialisatie; waarom die standaard normale distributie geen goede keuze is. Beschouw even een willekeurig neuron uit de eerste hidden layer van een willekeurig netwerk. Stel dat 500 van de 1000 inputs '1' zijn, en de rest '0'. In de gewogen som, $z = wx+b$, worden dan 501 (500 gewichten en 1 bias) toevalsvariabelen gesommeerd die ieder een gemiddelde van 0 en standaarddeviatie van 1 hebben. Hierdoor wordt z een toevalsvariabele met een gemiddelde van 0 en standaarddeviatie van $\sqrt{501} \approx 22.4$. Die normale kansdistributie wordt in figuur 23 afgebeeld.

Let vooral op de breedte van deze distributie. Dit betekent dat er veel kans is dat z een grote absolute waarde zal hebben, veel groter dan 1. Hierdoor is $a = \sigma(z) \approx 0$ of $a = \sigma(z) \approx 1$, en $\sigma'(z) \approx 0$. Herinner uit paragraaf 7.1 dat het neuron dus gesatureerd is en dit tot 'traag leren' kan leiden. Uiteraard geldt dit voor alle neuronen in de eerste hidden layer. Vermits alle activaties uit de eerste hidden layer dus ongeveer 0 of 1 zijn, verkrijgen we een gelijkaardige situatie in de volgende laag. Dit probleem zet zich dus verder in iedere volgende laag.

Hoewel we een specifiek voorbeeld hebben gebruikt in deze uitleg, zal de kansdistributie van z altijd deze brede vorm hebben wanneer de parameters volgens de standaard normale distributie geïnitialiseerd werden [1].

8.2 Nieuwe initialisatie

Om dit probleem op te lossen, moeten we zorgen dat de kansdistributie van z smaller wordt. We moeten dus zorgen dat de standaarddeviatie kleiner wordt. Daartoe moet de standaarddeviatie van de individuele parameters kleiner worden. Een geschikte keuze is de gewichten te initialiseren volgens de normale distributie met gemiddelde 0 en standaarddeviatie $1/\sqrt{n}$, met n het aantal neuronen uit de vorige laag. Vermits er slechts één bias is tegenover de vele gewichten, heeft de bias amper impact op de uiteindelijke kansdistributie van z. Daarom kunnen we voor de bias de oude initialisatie blijven gebruiken; vol-

gens de standaard normale distributie initialiseren. De hierboven beschreven verbetering van de initialisatie is slechts één van de vele mogelijkheden in de literatuur beschreven.

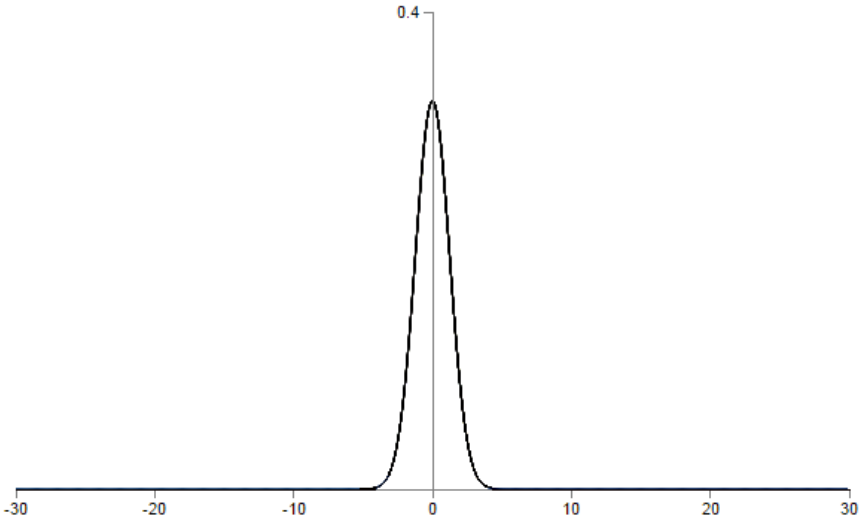
Laat ons de nieuwe standaarddeviatie van z berekenen. Voor één gewicht wordt de variantie[6] $(1/\sqrt{1000})^2 = 1/1000$. Vermits we de initialisatie van een bias niet gewijzigd hebben, blijft de variantie hiervoor 1. Er waren 500 inputs gelijk aan 1, waardoor de variantie voor z als volgt is:[7] $500/1000 + 1 = 3/2$. Bijgevolg is de standaarddeviatie voor z: $\sqrt{3/2} = 1.22$. De kansdistributie van z is nu dus veel smaller, zoals in figuur 24 getoond wordt. Het gevolg hiervan is dat er veel minder kans is dat $\sigma'(z) \approx 0$ en dus is er veel minder kans op 'traag leren'.

8.3 Experiment

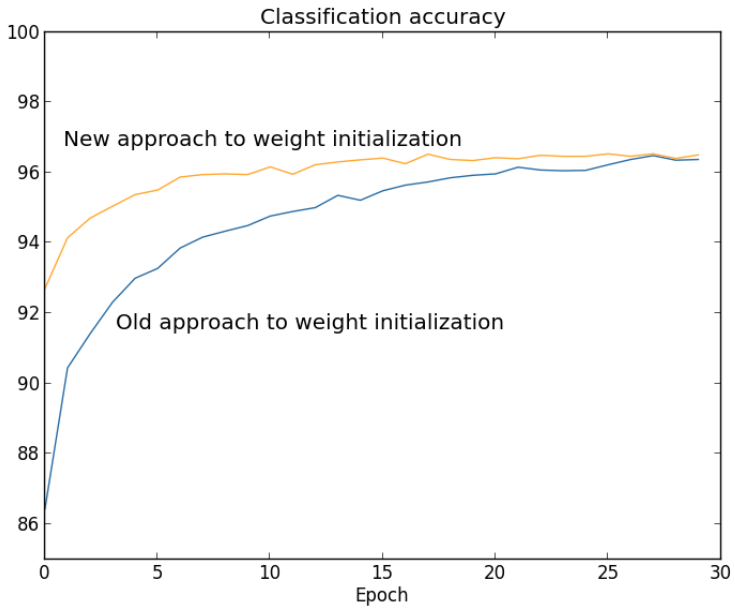
Nielsen implementeerde zowel de oude als de nieuwe techniek in zijn 'network2.py'. De default initialisatie gebeurt volgens de nieuwe techniek. Met `net.large_weight_initializer()` kunnen we expliciet aangeven om alsnog de oude initialisatie te gebruiken. Op die manier kunnen we de nauwkeurigheid van classificatie voor beide technieken onderling vergelijken. Figuur 25 toont het resultaat.

De gebruikte waarden van de hyper-parameters, alsook de hoge classificatie nauwkeurigheden, zijn momenteel niet van belang. Nielsen implementeerde namelijk nog andere optimalisaties, die pas in volgende hoofdstukken besproken worden. Wat wel van belang is, is dat de nieuwe techniek er duidelijk voor zorgt dat het netwerk sneller convergeert naar de uiteindelijke classificatie nauwkeurigheid (van zo'n 96%). Het netwerk bereikt deze nauwkeurigheid nu reeds rond epoch 21, in plaats van pas rond epoch 27. Dat komt omdat er nu initieel veel minder kans is op gesatureerde neuronen, en dus op traag leren. Zoals verwacht is er geen verschil in de uiteindelijke nauwkeurigheid van classificatie.

[6] De variantie is het kwadraat van de standaarddeviatie.
[7] De variantie van de som van toevalsvariabelen is de som van hun varianties.



Figuur 24: Smalle kansdistributie van z indien de gewichten volgens de normale distributie met gemiddelde 0 en standaarddeviatie $1/\sqrt{n}$ geïnitialiseerd worden.



Figuur 25: Vergelijking van de 2 mogelijke technieken om de parameters te initialiseren.

9 Tanh en ReLU activatiefuncties

In paragraaf 2.3 werd de logistic sigmoid activatiefunctie besproken. Nog twee populaire activatiefuncties, de softmax en identeitsfunctie, werden in paragraaf 2.4 besproken. In dit hoofdstuk worden nog twee activatiefuncties kort besproken. Van deze activatiefuncties konden we het nut voorheen nog niet beargumenteren.

9.1 Tangens hyperbolicus

Een activatiefunctie die vaak in de literatuur voorkomt, is de tangens hyperbolicus (tanh). Deze is gedefinieerd als volgt:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

De tanh behoort tot de sigmoid activatiefuncties, en heeft dus dezelfde vorm als de logistic sigmoid uit paragraaf 2.3. Figuur 26 beeldt deze functie af. Het verband tussen de logistic sigmoid en de tanh is $\sigma(z) = (1 + \tanh(z/2))/2$:

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

De logistic sigmoid mapt zijn inputs op [0;1], terwijl de tanh ze op [-1;1] mapt. Indien de tanh gebruikt wordt als activatiefunctie, moeten de gewenste outputs in de training data herschaald worden naar dit domein.

Het theoretische voordeel van het gebruik van de tanh is het volgende. Beschouw alle k gewichten w_{jk}^{l+1} naar het j-de neuron uit laag $l+1$. Herinner dat de bijhorende partieel afgeleiden $a_k^l \delta_j^{l+1}$ zijn. Vermits de activaties a_k^l van een sigmoid neuron steeds positief zijn, hangt het teken van de partieel afgeleiden af van dat van de error δ_j^{l+1}. Het gevolg hiervan is dat alle w_{jk}^{l+1} zullen toenemen óf afnemen, wat niet altijd gewenst is. Het kan namelijk zijn dat het ene gewicht moet toenemen, terwijl een ander moet dalen om de kost te verlagen. Vermits de activatie bij gebruik van de tanh ook negatief kan zijn, wordt dit probleem verholpen.

Empirisch is er echter weinig tot geen verbetering te zien. Men begrijpt (nog) niet waarom. Waarschijnlijk heeft de logistic sigmoid nog ongekende voordelen ten opzichte van de tanh.

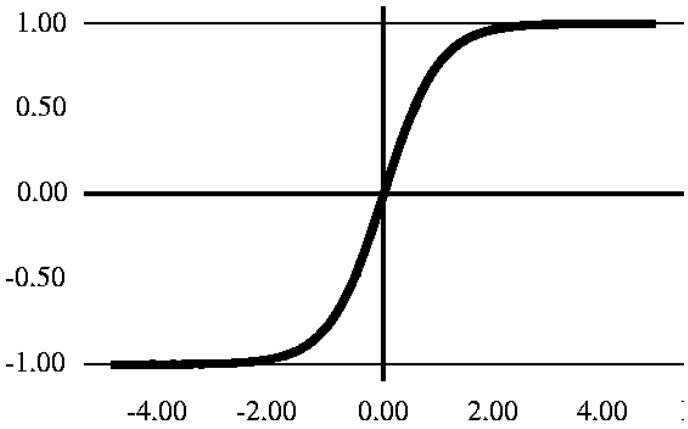
9.2 Rectified Linear Unit

De rectified linear unit (ReLU) is gedefinieerd als volgt:

$$ReLU(z) = \max(0, z)$$

Het functieverloop wordt in figuur 27 getoond. Voor $z \geq 0$ geldt $ReLU'(z) = 1$; voor $z < 0$ geldt $ReLU'(z) = 0$.

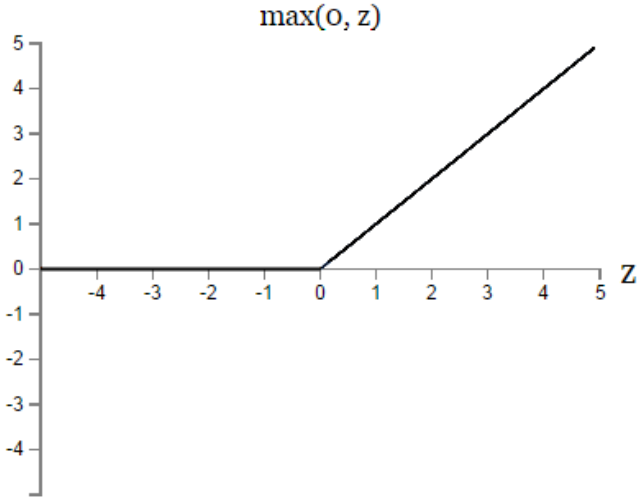
De theoretische motivatie achter deze activatiefunctie is de volgende. Herinner het probleem 'traag leren' uit paragraaf 7.1.1 en de cross-entropy kostfunctie



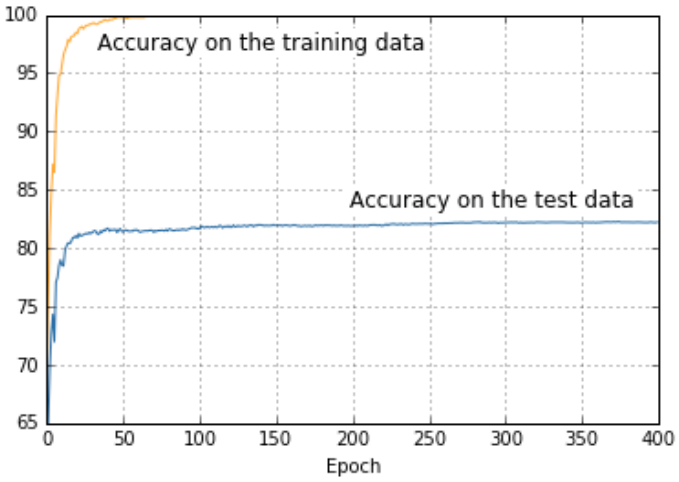
Figuur 26: Tanh activatiefunctie.

die daartoe geïntroduceerd werd. Zoals aan het einde van paragraaf 7.1.2 vermeld, kan het traag leren nog steeds optreden in voorgaande lagen door de $\sigma'(z)$ in vergelijking 2 van backpropagation. Laat ons nu de ReLU gebruiken als activatie in combinatie met de mean squared error als kostfunctie. Beschouw het geval $z \geq 0$. Vermits de afgeleide van de activatiefunctie steeds 1 is, kan in geen enkele laag saturatie voorkomen. Dit is een groot voordeel van de ReLU t.o.v. de logistic sigmoid. Beschouw nu het geval $z < 0$. Vermits de afgeleide van de activatiefunctie 0 is, worden de partieel afgeleiden van de gewichten en biases precies 0. De gradiënt is als het ware verdwenen. Eens $z < 0$ stopt het neuron dus met leren, wat zeker niet altijd gewenst is.

Vreemd genoeg heeft de ReLU zijn nut toch bewezen in de praktijk [1]. Opnieuw begrijpt men niet goed waarom deze activatiefunctie soms nuttig blijkt.



Figuur 27: ReLU activatiefunctie.



Figuur 28: Verloop van de classificatie nauwkeurigheid voor de training data en test data in een niet-geregulariseerd netwerk.

10 Overfitting en regularisatie

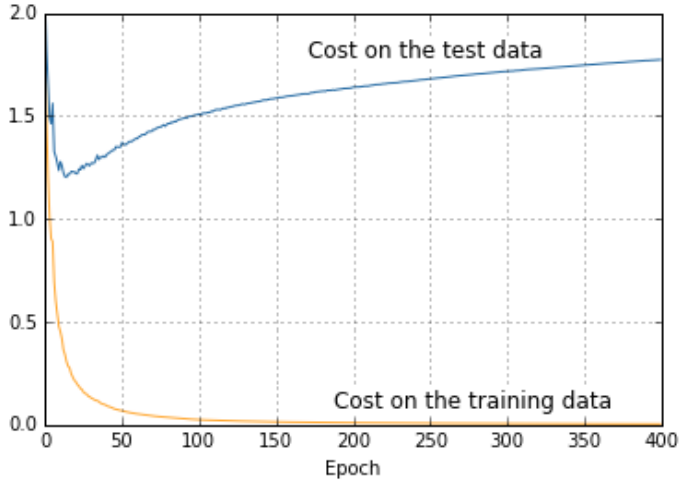
Naast 'traag leren' is er nog een groot probleem binnen neurale netwerken, namelijk 'overfitting'. In dit hoofdstuk worden verschillende technieken besproken om overfitting te beperken, zgn. regularisatie-technieken.

10.1 Overfitting

Om overfitting uit te leggen, zullen we ten eerste het gevolg ervan observeren in een experiment. We gebruiken hiervoor de voorbeeldtoepassing rond het classificeren van handgeschreven cijfers. Dit maal trainen we 400 epochs en gebruiken we slechts 1000 training samples. Het verloop van de classificatie nauwkeurigheid voor de training data en test data wordt in figuur 28 getoond. Het verloop van de kost voor de training data en test data wordt in figuur 29 getoond.

Voor de classificatie nauwkeurigheden valt meteen op dat die voor de training data blijft stijgen, tot 100%. Die voor de test data echter niet, die stopt op ongeveer 83%. Vanaf ongeveer epoch 280 blijkt het trainen geen nut meer te hebben, aangezien het netwerk niet meer bijleert. Voor de kost kunnen we een analoge redenering maken. De kost voor de training data blijft namelijk dalen, tot 0.0, maar niet voor de test data. Vanaf het 15e epoch neemt deze kost zelfs toe. We zien dus dat vanaf een bepaald aantal epochs, het netwerk niet langer beter wordt in het classificeren van de test data, terwijl het wel beter blijft worden in het classificeren van de training data. Zodra dit fenomeen zich voordoet, spreekt men van 'overfitting' of 'overtraining'. Overfitting wordt ook gekenmerkt doordat de kost voor de training data ver van de kost voor de test data afwijkt. Hetzelfde geldt uiteraard voor de classificatie nauwkeurigheden.

De bron van het probleem is de grote hoeveelheid parameters in het netwerk. Ons 3-lagig neuraal netwerk (784, 30 en 10 neuronen) bestaat uit maar liefst $30 \cdot (784 + 1) + 10 \cdot (30 + 1) = 23860$ parameters. Doordat er zo veel parameters



Figuur 29: Verloop van de kost voor de training data en test data in een niet-geregulariseerd netwerk.

zijn, is het netwerk in staat alle input-ouput-mappings ongeveer te memoriseren. Het netwerk kan een te goede benadering vormen voor de training data. Dit zorgt ervoor dat het netwerk de patronen niet leert herkennen, en bijgevolg niet in staat is om goede veralgemeningen te maken.

Om het bovenstaande te illustreren, kunnen we best even los van neurale netwerken denken. Stel dat we een zo goed mogelijk model, onder de vorm van een functievoorschrift, willen bepalen voor de data in figuur 30. In figuur 31 worden 2 mogelijke modellen getoond. Het model in het blauw gebruikt veel parameters, 10 om precies te zijn (een 9e orde polynoom); het model in het oranje gebuikt weinig parameters, 2 om precies te zijn (een rechte). De rechte is waarschijnlijk het betere model van de 2. Het is namelijk waarschijnlijker dat andere punten in de buurt van deze rechte zullen liggen, en dat de kleine afwijkingen gewoon ruis voorstellen. Dit is echter geen garantie; soms is deze "keep it simple" aanpak de foute keuze.

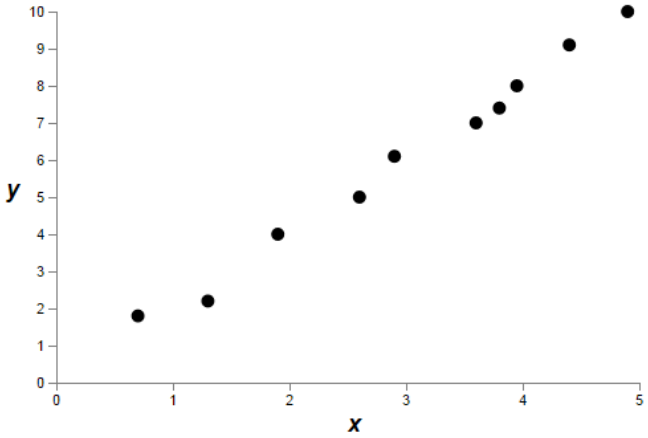
Computers worden alsmaar krachtiger, waardoor ook steeds grotere neurale netwerken gebruikt kunnen worden, met miljoenen parameters. Deze trend maakt het zeer relevant om te proberen om overfitting te voorkomen.

10.2 Intro tot regularisatie-technieken

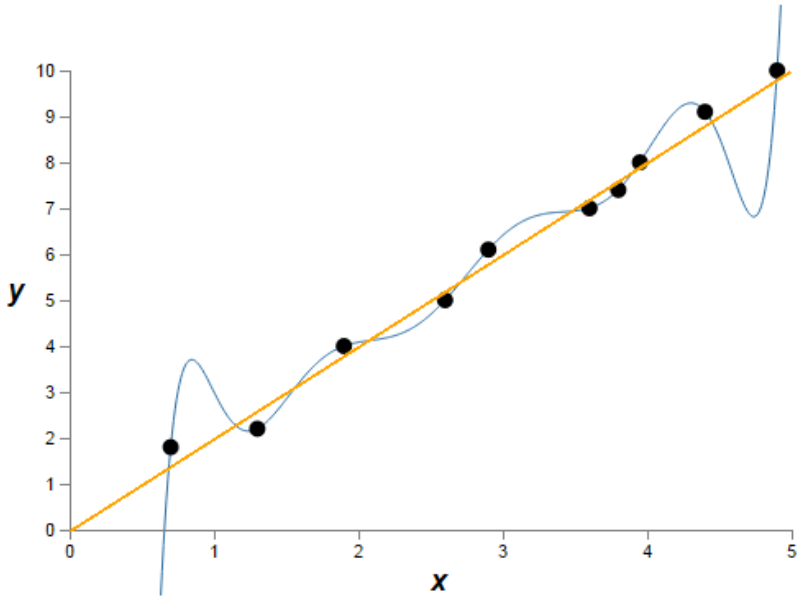
We weten nu dat de grote hoeveelheid parameters in het netwerk verantwoordelijk is voor overfitting. In de praktijk is een eenvoudig model (met weinig parameters) daarom vaak goed in het maken van veralgemeningen.

Men zou dan ook volgende 3 voor de hand liggende technieken kunnen suggereren om overfitting te beperken in de context van neurale netwerken:

1. Minder parameters gebruiken:
   De impact van deze aanpak hebben we eigenlijk al gezien in paragraaf 5.2. Voor onze voorbeeldtoepassing haalde een netwerk met 100 hidden neuronen zo'n 87% classificatie nauwkeurigheid. Indien we 30 hidden neu-



Figuur 30: Data die met een functievoorschrift gemodelleerd moet worden.



Figuur 31: Model met veel parameters (blauwe curve); model met weinig parameters (oranje rechte).

ronen gebruikten, een netwerk met minder parameters dus, haalden we zo'n 95.50%. Het netwerk werd duidelijk beter in het maken van veralgemeningen. Toch is deze aanpak geen goed idee. Grotere netwerken, met veel parameters, zijn namelijk in staat om veel complexere mappings te leren.

2. Early stopping:
   We zouden het trainen gewoon kunnen beëindigen zodra de classificatie-nauwkeurigheid niet meer stijgt (of de kost niet meer daalt). Deze techniek heet 'early stopping', maar maakt het netwerk niet beter in het maken van veralgemeningen. Het voorkomt alleen dat het netwerk onnodig lang traint, maar de uiteindelijke classificatie nauwkeurigheid neemt niet toe. Op deze techniek komen we in paragraaf 11.2.1 terug, in de context van het bepalen van de hyper-parameters.

3. Meer training data:
   Door meer training data te gebruiken, kan het netwerk de specifieke input-output-mappings onmogelijk nauwkeurig memoriseren. Hierdoor leert het netwerk alleen de patronen in de data. In de context van onze voorbeeldtoepassing kennen we de impact van meer training data te gebruiken al: met 1000 training samples bereikten we een nauwkeurigheid van zo'n 83%, terwijl we met de volledige training dataset van 50 000 samples zo'n 95.50% haalden. Hoewel het uitbreiden van de dataset zeker nuttig is, is het vaak niet haalbaar. Zoals we in paragraaf 4.2 hebben gezien, is het verwerven van data namelijk een moeilijke klus. Anderzijds bestaat een eenvoudige regularisatie-techniek erin de dataset op een artificiële manier uit te breiden (zie paragraaf 10.4.4), wat zeker in de buurt komt.

Het doel van regularisatie is het netwerk beter te maken in veralgemenen, en bijgevolg overfitting te beperken. Hiervoor worden regularisatie-technieken gebruikt, die tijdens het trainen gebruikt worden om ervoor te zorgen dat het neuraal netwerk een eenvoudig model voorstelt. De 3 voorgaande technieken kunnen we hier niet toe rekenen omdat ze, zoals gemotiveerd, in het algemeen niet bruikbaar zijn. In paragraaf 10.4 worden 4 bekende regularisatie-technieken besproken.

10.3 Gebruik van validatie data

Herinner dat een neuraal netwerk vele hyper-parameters gebruikt, zoals de learning rate en de mini-batch grootte. Vele regularisatie-technieken introduceren nog extra hyper-parameters. Voor het bepalen van alle hyper-parameters worden de validatie data gebruikt; niet de training of test data.

De reden hiervoor is dezelfde als die voor de splitsing tussen de training en test data: we willen een onafhankelijke evaluatie van het netwerk kunnen maken. Indien we niet alleen de parameters van het netwerk, maar ook de hyper-parameters volgens de training set zouden bepalen, zouden beide verzamelingen van parameters te sterk van elkaar afhankelijk worden; risico op overfitting. We moeten hiervoor dus een andere dataset gebruiken. Indien we echter de test data zouden gebruiken, is het mogelijk dat we de hyper-parameters specifiek voor die test data bepalen; opnieuw risico op overfitting. Daarom moet een derde dataset gebruikt worden, die men de validatie data noemt.

De test data kunnen we nu blijven gebruiken als een onafhankelijke evaluatie. Zo hebben we nog steeds een goede indicatie voor de mate waarin het netwerk veralgemeningen kan maken.

10.4 Regularisatie-technieken

Door het grote belang van regularisatie doet men veel onderzoek naar dit onderwerp. Er zijn dan ook vele regularisatie-technieken terug te vinden in de literatuur. Vele papers concluderen echter in de aard van "Our whiz-bang technique gave us an improvement of X percent on standard benchmark Y" [1]. Met zulke conclusies moet men opletten. Het zou namelijk kunnen dat de techniek afgesteld werd op die specifieke dataset, waardoor de techniek op andere datasets mogelijks waardeloos zou blijken.

In deze paragraaf worden 4 populaire regularisatie-technieken uitgelegd. Ze zijn populair omdat men empirisch heeft kunnen vaststellen dat ze (bijna) altijd nuttig blijken; ze zijn niet alleen bruikbaar voor één specifieke dataset.

10.4.1 L2 regularisatie

Definitie:
De bekendste regularisatie-techniek heet 'L2 regularisatie', ook wel 'weight decay' genoemd. Deze techniek bestaat erin de oorspronkelijke kostfunctie C_0 als volgt aan te passen:

$$C = C_0 + \frac{\lambda}{2N} \sum_w w^2$$

Hierbij heet λ de 'regularisatie parameter', waarvoor geldt dat $\lambda > 0$. N is het totale aantal training samples, en de som gaat over alle gewichten w. Indien C_0 bijvoorbeeld de cross-entropy kostfunctie is, geldt:

$$C = -\frac{1}{N} \sum_x \sum_j [y_j \ln(a_j^L) + (1 - y_j) \ln(1 - a_j^L)] + \frac{\lambda}{2N} \sum_w w^2$$

L2 regularisatie "bestraft" grote gewichten: het is duidelijk dat de kost toeneemt naarmate de gewichten in het netwerk groter zijn. Deze regularisatietechniek zorgt er bijgevolg voor dat, in het algemeen, de gewichten klein zullen zijn (in absolute waarde). Anderzijds blijven grote gewichten wel mogelijk, op voorwaarde dat ze de oorspronkelijke kost, C_0, sterk laten dalen. Met de regularisatie parameter λ kan bepaald worden hoeveel belang men hecht aan de groottes van de gewichten. Hoe kleiner λ is, hoe minder belang gehecht wordt aan kleine gewichten; in het extreme geval $\lambda = 0$ zou C tot C_0 reduceren en wordt er dus geen rekening gehouden met de groottes van de gewichten.

Laat ons nu de nieuwe update-regels bepalen voor een willekeurig gewicht w en een willekeurig bias b. Voor de partieel afgeleiden geldt:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{N} w \quad \text{en} \quad \frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$$

Hierdoor wordt de update-regel voor een gewicht w:

$$w \to w - \eta \frac{\partial C}{\partial w} = w - \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{N} w = \left(1 - \frac{\eta\lambda}{N}\right) w - \eta \frac{\partial C_0}{\partial w}$$

Voor een bias blijft de update-regel ongewijzigd:

$$b \to b - \eta \frac{\partial C_0}{\partial b}$$

In het geval van stochastic gradient descent worden de update regels:

$$w \to \left(1 - \frac{\eta\lambda}{N}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w} \quad \text{en} \quad b \to b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$

Uit deze update-regels blijkt duidelijk waar de benaming 'weight decay' vandaan komt. De eerste stap in het updaten van het gewicht is namelijk een afname ("decay") van het gewicht. De factor waarmee het gewicht afneemt, $(1-\eta\lambda/N)$, noemt men de 'weight decay factor'. Merk op dat deze afname lineair is t.o.v. w.

Intuïtie:
We weten dat L2 regularisatie er, in het algemeen, voor zorgt dat de gewichten klein zullen zijn. De vraag is dus waarom het netwerk daardoor beter wordt in het maken van veralgemeningen. De intuïtie hierachter is dat kleine gewichten minder gevoelig zijn voor ruis, in tegenstelling tot grote gewichten. Kleine gewichten laten iedere input in de dataset een beetje meetellen, waardoor de algemene trend in de inputs beschouwd wordt. Een groot gewicht, daarentegen, laat iedere input sterk meetellen, waardoor rekening gehouden wordt met kleine verschillen tussen de inputs (ruis).

De gewichten bepalen in welke mate de inputs meetellen in de gewogen som. Voor de bias geldt dit niet; deze bepaalt in welke mate de output getriggerd wordt. De groottes van de biases hebben dus niets te maken met hoe goed het netwerk zal kunnen veralgemenen. Daarom houdt L2 regularisatie geen rekening met de groottes van de biases. Men heeft ook empirisch kunnen vaststellen dat de rol van de biases hier onbelangrijk is [1].

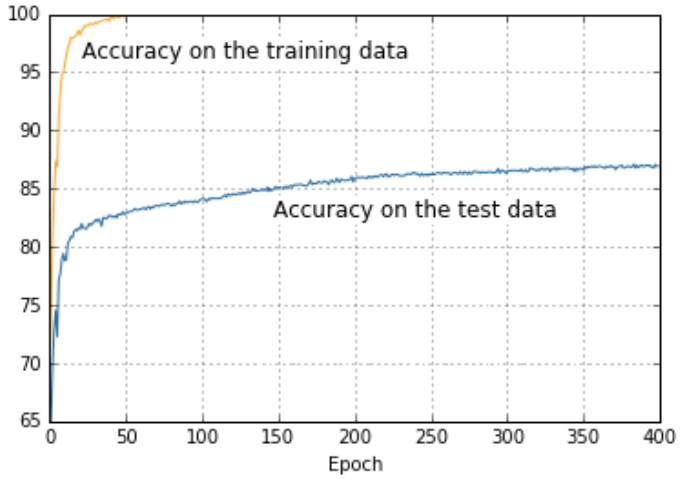
Experimenten:
In 'network2.py' implementeerde Nielsen deze techniek. In de volgende experimenten gebruiken we dus zijn code.

Herinner het verloop van de classificatie nauwkeurigheid (figuur 28) en van de kost (figuur 29) indien we slechts 1000 training samples gebruikten. Door het gebruik van L2 regularisatie met $\lambda = 0.1$ krijgen we nu de resultaten in figuren 32 en 33 getoond. De nauwkeurigheid voor de test data blijft stijgen en is een stuk hoger dan voorheen. De kost voor de test data blijft dalen en is duidelijk lager dan voorheen. Ook liggen de nauwkeurigheden dichter bij elkaar; hetzelfde geldt voor de kosten.

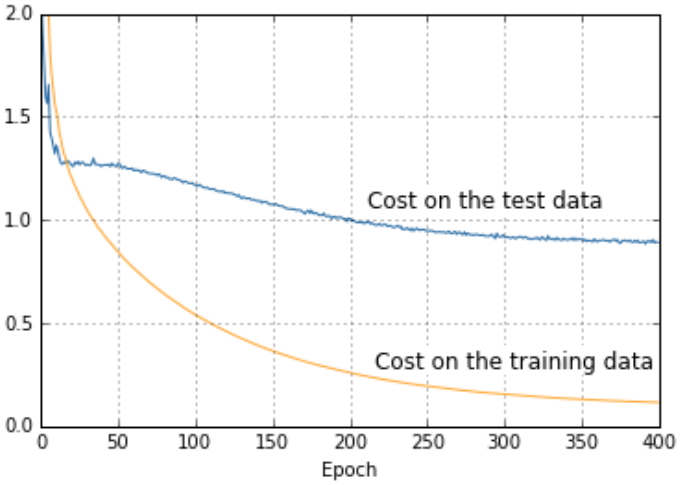
Herinner dat we zo'n 95.50% classificatie nauwkeurigheid haalden indien we de volledige training set van 50000 samples gebruikten. We moeten nu $\lambda = 5.0$ kiezen om de weight decay factor, $(1 - \eta\lambda/N)$, gelijk te houden.[8] Het geregulariseerde netwerk haalt nu zo'n 96.50% classificatie nauwkeurigheid.

Link met de initialisatie van de gewichten:
Herinner dat de nieuwe techniek om gewichten te initialiseren (zie hoofdstuk 8) ervoor zorgt dat het trainen begint met kleine gewichten. Het gebruik van L2

[8] Eerst was $N = 1000$, maar nu is $N = 50000$; een toename met factor 50.



Figuur 32: Verloop van de classificatie nauwkeurigheid voor de training data en test data in een L2-geregulariseerd netwerk.



Figuur 33: Verloop van de kost voor de training data en test data in een L2-geregulariseerd netwerk.

regularisatie in combinatie met de oude techniek voor het initialiseren van de gewichten, heeft hetzelfde effect. Om dit in te zien, dient men de update-regel die bij L2 regularisatie gebruikt wordt in het achterhoofd houden.

Indien de oude gewicht-initialisatie-techniek gebruikt wordt, zijn de gewichten initieel relatief groot. Uit hoofdstuk 8 weten we dat de neuronen hierdoor grote kans hebben om gesatureerd te raken. In de update-regel zijn de partieel afgeleiden bijgevolg zeer klein. Verder zijn de gewichten groot, waardoor de weight decay een grote invloed heeft (lineair verband). De eerste epochs worden dus gedomineerd door weight decay, waardoor de gewichten alsmaar kleiner worden. Uiteindelijk worden de gewichten zo klein dat de neuronen niet langer gesatureerd zijn, en de partieel afgeleiden bijgevolg niet langer klein zijn. Verder wordt de invloed van de weight decay alsmaar kleiner, net doordat de gewichten kleiner worden. Bijgevolg wordt de update-regel vanaf dan gedomineerd door de partieel afgeleiden, wat op de oude update-regel neerkomt.

L2 regularisatie zorgt dus, net als de nieuwe gewicht-initialisatie-techniek, dat de gewichten klein zijn vooraleer de echte training van start gaat.

10.4.2 L1 regularisatie

Definitie:
Deze techniek lijkt erg op L2 regularisatie. Bij L1 regularisatie wordt de kostfunctie als volgt aangepast:

$$C = C_0 + \frac{\lambda}{N} \sum_w |w|$$

waar $|w|$ staat voor de absolute waarde van gewicht w. Doordat de groottes van de gewichten in de kost worden opgenomen, zorgt ook L1 regularisatie ervoor dat de gewichten eerder klein zullen zijn.

De partiële afgeleide tegenover een gewicht w wordt:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{N} sgn(w)^9$$

Die tegenover een bias blijft ongewijzigd. De update-regel voor stochastic gradient descent voor een gewicht w wordt:

$$w \to w - \frac{\eta\lambda}{N} sgn(w) - \eta \frac{\partial C_0}{\partial w} = w - \frac{\eta\lambda}{N} sgn(w) - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}$$

Die voor een bias blijft uiteraard, opnieuw, ongewijzigd.

Intuïtie:
Ook de intuïtie achter deze regularisatie-techniek is gelijkaardig aan die achter L2 regularisatie. Bij L2 regularisatie is de weight decay decay factor lineair t.o.v. w, terwijl dat bij L1 regularisatie constant is. L1 regularisatie vermindert een groot gewicht bijgevolg minder dan L2 regularisatie, en een klein gewicht meer dan L2 regularisatie dat doet. Het resultaat van L1 regularisatie is dat vele gewichten in het netwerk zeer klein zijn, en enkele belangrijke gewichten groot

[9] De 'sign' functie is gedefinieerd als volgt. Als $x < 0$, geldt $sgn(x) = -1$. Als $x > 0$, geldt $sgn(x) = 1$. En $sgn(0) = 0$.

zijn. Het netwerk kijkt bijgevolg alleen naar de belangrijkste inputs, waardoor het een eenvoudig model voorstelt en beter is in het maken van veralgemeningen.

Experimenten:
Nielsens 'network2.py' implementeert deze techniek niet. Met een minimale aanpassing van de broncode kan de L2 regularisatie vervangen worden door L1 regularisatie.

De resultaten zijn volledig analoog aan die van L2 regularisatie, zowel voor de kleine training set als voor de volledige. Voor onze experimenten blijken L1 en L2 regularisatie even geschikt. In de praktijk zijn beide technieken erg populair.

10.4.3 Dropout

Definitie:
Deze techniek bestaat erin de netwerkarchitectuur aan te passen; niet de kostfunctie. Ten eerste wordt de helft van de hidden neuronen uit het netwerk verwijderd. Deze neuronen worden willekeurig gekozen. Ten tweede worden de gewichten en biases aangepast volgens de gekende update-regels van het stochastic gradient descent algoritme. Ten derde worden de verwijderde hidden neuronen weer toegevoegd. Dit proces wordt herhaald voor iedere mini-batch, voor het gekozen aantal epochs.

We benadrukken dat na het trainen alle hidden neuronen in het netwerk aanwezig zijn. De gewichten en biases in de outputlaag werden echter geleerd in een netwerkarchitectuur die slechts uit de helft van het totale aantal hidden neuronen bestaat. Na het trainen krijgt de outputlaag dus dubbel zo veel inputs als tijdens het trainen. Ter compensatie worden de gewichten in de outputlaag gehalveerd. De gewogen sommen in de outputlaag zijn nu equivalent met de gewogen sommen die berekend werden indien slechts de helft van de hidden neuronen gebruikt werden. Daarom mogen de biases ongewijzigd blijven.

Het verwijderen van een hidden neuron moet men niet letterlijk nemen. Men zou namelijk gewoon kunnen instellen dat de activatie van dat neuron '0' is. Ook beschrijft de uitleg hierboven eigenlijk een specifiek geval van dropout. Meer algemeen wordt ieder hidden neuron met een bepaalde kans verwijderd [18]. Bijgevolg wordt niet noodzakelijk de helft van de hidden neuronen verwijderd. Op die algemenere vorm gaan we niet verder in.

Intuïtie:
Voor deze regularisatie-techniek zijn volgende twee intuïtieve verklaringen mogelijk.

Ten eerste zorgt dropout voor een soort uitmiddeling van verschillende netwerken die gelijktijdig getraind worden. Het intuïtieve voordeel hiervan is dat ieder individueel netwerk waarschijnlijk op een andere manier last zal hebben van overfitting. Het uitgemiddelde netwerk beperkt dit effect waardoor het beter kan veralgemenen.

Ten tweede gebruikt de outputlaag tijdens het trainen steeds andere hidden neuronen. Hierdoor vermijdt dropout dat de outputs afhankelijk zijn van een beperkt aantal neuronen in de hidden layer. Vermits de neuronen niet afhankelijk kunnen zijn van elkaar, worden ze geforceerd zich op patronen in de data te

baseren. Hierdoor wordt het netwerk beter in het maken van veralgemeningen.

Experiment:
Deze techniek werd niet geïmplementeerd in 'network2.py' door Nielsen en zullen we ook niet zelf implementeren.

In het paper van Geoffrey Hinton et al. [19] beschrijft men de significante verbeteringen die men haalde door het gebruik van dropout. Zij gebruikten nog andere optimalisaties, maar door het gebruik van dropout steeg hun classificatie nauwkeurigheid van 98.4% naar 98.7% voor de MNIST database.

10.4.4 Dataset artificieel uitbreiden

Definitie:
Deze techniek bestaat er simpelweg in de dataset zelf uit te breiden met realistische variaties op de bestaande data. Wat die variaties precies zijn, hangt af van het domein waarin de dataset past. Voor de MNIST database zouden we kleine rotaties, translaties en schaleringen kunnen maken van de bestaande afbeeldingen. Zulke variaties kunnen namelijk optreden voor handgeschreven cijfers.

Intuïtie:
De intuïtie achter deze regularisatie-techniek is erg eenvoudig. Door het netwerk te trainen op meer data, heeft het meer variaties op de data gezien, waardoor het netwerk vanzelfsprekend beter zal kunnen veralgemenen. Het is, algemeen gesproken, erg kostelijk om meer data te verzamelen, maar de dataset artificieel uitbreiden is dat zeker niet.

Een bijkomstig voordeel van deze techniek is dat er algoritmisch niets verandert. Dit maakt deze regularisatie-techniek enorm eenvoudig te begrijpen. Ook moet men niet op zoek gaan naar geschikte waarden voor mogelijke extra hyper-parameters.

Experimenten:
Op Nielsens GitHub account [13] vinden we 'expand_mnist.py' terug. Hiermee kan de MNIST database (artificieel) worden uitgebreid met translaties over 1 pixel naar boven, rechts, onder en links. De training set bestaat bijgevolg uit 250 000 afbeeldingen en daarom kiezen we $\lambda = 25.0$ als regularisatie parameter. Door het gebruik van deze regularisatie-techniek stijgt de classificatie nauwkeurigheid van zo'n 96.50% naar zo'n 97.50%.

Ook in het paper van P. Simard, D. Steinkraus en J. Platt [20] wordt deze techniek toegepast op de MNIST database. Ze gebruikten translaties, rotaties, schaleringen en nog 2 meer technische variaties (waar we niet op ingaan) in combinatie met een uitgebreidere netwerkarchitectuur. Het gebruik van de uitgebreide dataset zorgde ervoor dat hun classificatie nauwkeurigheid van 98.40% naar 99.30% steeg.

Het artificieel uitbreiden van de dataset is duidelijk een eenvoudige maar doeltreffende techniek.

11 Hyper-parameters kiezen

Met het stochastic gradient descent algoritme kan het neuraal netwerk zijn eigen parameters (gewichten en biases) bepalen. Ondertussen werd ook een groot aantal hyper-parameters geïntroduceerd, namelijk: het aantal hidden layers, het aantal neuronen per hidden layer, de learning rate, de mini-batchgrootte, het aantal te trainen epochs, de regularisatie parameter, de gebruikte activatiefunctie en de gebruikte kostfunctie. De hyper-parameters hebben ook een grote invloed op de kwaliteit van het neuraal netwerk. In dit hoofdstuk worden een algemene strategie en een aantal heuristieken toegelicht om gemakkelijker geschikte waarden voor de hyper-parameters te kunnen bepalen.

11.1 Algemene strategie

De eerste stap in de algemene strategie voor de keuze van de hyper-parameters bestaat erin zo snel mogelijk een beter-dan-gokken-nauwkeurigheid te behalen. De nauwkeurigheid de men haalt door te gokken, hangt af van het aantal klassen waarin geclassificeerd wordt. Voor de voorbeeldtoepassing over handgeschreven cijfers heeft men een kans van 1/10 om juist te gokken, waardoor de nauwkeurigheid door te gokken 10% bedraagt.

Het behalen van een beter-dan-gokken-nauwkeurigheid is niet altijd eenvoudig. Men kan initieel best een aantal vereenvoudigingen doen. Zo kan men minder klassen, een eenvoudigere netwerkarchitectuur en een kleinere test dataset gebruiken. Deze vereenvoudigingen zorgen er tevens voor dat het netwerk sneller feedback geeft tijdens het trainen, waardoor de gebruiker de hyper-parameters sneller kan wijzigen. Verder stelt men best de regularisatie parameter $\lambda = 0.0$ en focust men op de learning rate η. De keuze van de overige hyper-parameters is initieel een kwestie van experimenteren. De heuristieken in de volgende paragraaf beschreven kunnen hiertoe bijdragen.

Zodra met een configuratie van de hyper-parameters gevonden heeft die een beter-dan-gokken-nauwkeurigheid geeft, staat men al ver. De volgende stap in de algemene strategie bestaat erin de hyper-parameters geleidelijk te verfijnen en de gemaakte vereenvoudigingen geleidelijk ongedaan te maken. Zodra men tevreden is over de classificatie nauwkeurigheid die het netwerk haalt, stopt men gewoon met het aanpassen van de hyper-parameters.

11.2 Heuristieken

In paragraaf 2.2.2 gaven we reeds een heuristiek voor het bepalen van het aantal hidden layers en het aantal neuronen per hidden layer. De motivatie achter de cross-entropy kostfunctie (zie paragraaf 7.1) kan als heuristiek dienen om deze kostfunctie te verkiezen boven de mean squared error. De keuze tussen de logistic sigmoid en tanh als activatiefunctie is typisch niet van groot belang. Voor de overige hyper-parameters kan men onderstaande heuristieken raadplegen.

11.2.1 Aantal te trainen epochs

Het aantal te trainen epochs staat eerder los van de overige hyper-parameters. Men kan met een grote waarde beginnen om zo de overige hyper-parameters te bepalen. Zodra deze min of meer vastliggen, kan 'early stopping' gebruikt

worden om het aantal te trainen epochs te verfijnen. Early stopping begint met het maximale aantal te trainen epochs in te stellen op een grote waarde. Na ieder epoch evalueert het netwerk de validatie data, om de validatie nauwkeurigheid te bepalen. Indien deze niet is toegenomen t.o.v. het vorige epoch, wordt het trainen meteen, en dus vroegtijdig, beëindigd. Op die manier krijgt men een indicatie van gedurende hoeveel epochs het trainen nuttig is.

Zoals hierboven beschreven, is de techniek te agressief. Uit verschillende grafieken in deze tekst kunnen we afleiden dat de classificatie nauwkeurigheid soms daalt, maar daarna weer stijgt. Een beter idee is daarom om bovenstaande techniek te veralgemenen naar een 'no-improvement-in-n' regel. Hierbij wordt het trainen vroegtijdig beëindigd indien de validatie nauwkeurigheid niet is toegenomen sinds de laatste n epochs. De parameter n is ook een hyper-parameter, maar deze is eenvoudig te bepalen. Men kan met een grote waarde beginnen en deze geleidelijk blijven verlagen zo lang de uiteindelijke classificatie nauwkeurigheid niet significant daalt.

Nielsen implementeerde deze techniek niet in zijn 'network2.py'. Om met deze techniek te experimenteren, implementeren we deze zelf, wat slechts een kleine uitbreiding vereist. Herinner dat we zo'n 96.50% classificatie nauwkeurigheid haalden indien gedurende 30 epochs getraind werd. In het experiment stellen we $n = 10$ en het maximale aantal te trainen epochs gelijk aan 400. Het trainen gaat gemiddeld 50 epochs door, waarbij een nauwkeurigheid van zo'n 96.70% gehaald wordt. Ondanks het grote verschil in aantal getrainde epochs, is er slechts een kleine verbetering in nauwkeurigheid. Daarom is gedurende 30 epochs trainen een zeer goede trade-off.

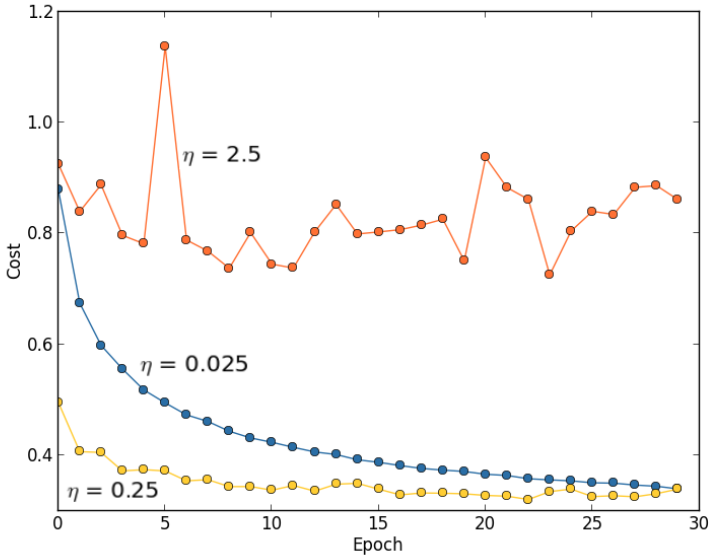
We kunnen nog een betere trade-off bekomen tussen het aantal getrainde epochs en de uiteindelijke classificatie nauwkeurigheid. We zouden namelijk kunnen instellen hoe groot de verbetering in nauwkeurigheid moet zijn. Indien er de laatste n epochs geen significante verbetering was, kunnen we het trainen beter al beëindigen. Laat ons instellen dat de nauwkeurigheid met meer dan 0.05% moet verbeteren t.o.v. de laatste 10 epochs. Het netwerk wordt dan gedurende zo'n 40 epochs getraind en haalt zo'n 96.58% classificatie nauwkeurigheid.

11.2.2 Learning rate

Zoals we in paragraaf 3.2.1 zagen, is de keuze van de learning rate η van groot belang.

Om de learning rate te bepalen, kan men best eerst de drempelwaarde ('treshold') bepalen. Dit is de waarde voor η waarvoor de kost gedurende de eerste epochs niet geleidelijk daalt, maar wel oscilleert of stijgt. Figuur 34 illustreert dit. Hier zou alleen $\eta = 0.025$ een goede keuze zijn. Men mag de learning rate nooit boven deze drempelwaarde kiezen. Om een eerste ruwe schatting van de drempelwaarde te bepalen, kan men met $\eta = 0.01$ beginnen en deze vervolgens met een factor 10 verhogen of verlagen naargelang het verloop van de kost. Zodra de ruwe schatting gekend is, kan deze verder verfijnd worden. Een goede keuze voor de uiteindelijke learning rate is de helft van de drempelwaarde.

Idealiter is de learning rate groot gedurende de eerste epochs, aangezien de classficatie nauwkeurigheid dan nog sterk toeneemt. Geleidelijk aan worden de verbeteringen in nauwkeurigheid kleiner. Nu zou de learning rate best afnemen zodat er kleinere stappen genomen worden en de nauwkeurigheid verder kan toenemen. Een techniek die het bovenstaande idee realiseert, heet 'learning



Figuur 34: Verschillende mogelijke keuzes voor de learning rate en hun impact op de kost.

rate schedule'. Initieel wordt η ingesteld zoals in de vorige alinea werd uitgelegd. Indien er geen verbetering is van de classificatie nauwkeurigheid gedurende n epochs, wordt de learning rate met een factor f verminderd. Het trainen stopt zodra de learning rate s keer verminderd werd. Voor deze 3 hyper-parameters kan men bv. $n = 10$, $f = 2$ en $s = 7$ kiezen.

Merk op dat deze techniek erg lijkt op early stopping. Nu wordt de learning rate verminderd met factor f, i.p.v. het trainen te stoppen. Het implementeren van deze techniek is bijgevolg eenvoudig. Laat ons een experiment uitvoeren met de hierboven beschreven n, f en s en met initieel $\eta = 0.5$. Na zo'n 145 epochs stopt het trainen, waarbij een classificatie nauwkeurigheid van 96.91% gehaald wordt. Het grote aantal epochs en de minimale verbetering in nauwkeurigheid tonen aan dat learning rate schedule met de gekozen n, f en s niet nuttig is in dit geval.

11.2.3 Regularisatie parameter

Zoals bij de algemene strategie werd toegelicht, stelt men de regularisatie parameter λ initieel best in op 0. Zodra men een beter-dan-gokken-nauwkeurigheid haalt, stelt men $\lambda = 1.0$. Vervolgens kan men λ herhaaldelijk verhogen of verlagen met factor 10 om een ruwe schatting voor λ te vinden. Vertrekkende van deze ruwe schatting kan men λ verder verfijnen om een betere classificatie nauwkeurigheid te bekomen.

11.2.4 Mini-batchgrootte

Herinner het gebruik van een mini-batch binnen stochastic gradient descent, zoals in paragraaf 3.3 werd toegelicht. In de praktijk zorgt een relatief kleine mini-batchgrootte m reeds voor een goede benadering van de gradiënt. De verschillende experimenten die we uitvoerden, getuigen hiervan. We kozen immers steeds $m = 10$ tegenover een training set van grootte $N = 50000$, maar haalden duidelijk goede resultaten. In dit opzicht zouden we steeds een kleine mini-batch gebruiken. Zo worden de parameters ieder epoch vaak geüpdatet, waardoor het trainen minder epochs vereist.

Anderzijds worden in implementaties van performante libraries voor neurale netwerken vaak zeer snelle matrixvermenigvuldigingen gebruikt. Zo worden de gradiënten voor alle samples in de mini-batch typisch met één matrixvermenigvuldiging berekend [1]. In dit opzicht is het beter om grote mini-batches te gebruiken. Hoe groter de mini-batch, hoe relatief sneller de berekening van de gradiënten immers is. Met zo'n grotere mini-batch kan de gradiënt vervolgens preciezer benaderd worden en de parameter (gewicht of bias) dus beter worden geüpdatet.

Bijgevolg dient voor de grootte van de mini-batch een compromis gemaakt te worden tussen vele updates doen enerzijds, en het uitbuiten van de snelheid van matrixvermenigvuldigingen anderzijds. Deze compromis moet de tijd (echte tijd; niet aantal epochs) die het netwerk nodig heeft om te trainen, minimaliseren. Daartoe kan men de validatie nauwkeurigheid plotten tegenover de tijd, voor een aantal mini-batchgroottes. Voor de voorbeeldtoepassing over handgeschreven cijfers stelde Nielsen vast dat $m = 10$ een goede keuze is: hier van afwijken maakt weinig verschil voor de tijd die nodig is voor het trainen.

Zoals men uit bovenstaande uitleg kan afleiden, staat de mini-batch grootte, net als het aantal te trainen epochs, eerder los van de andere hyper-parameters. Men kan dus met een kleine waarde beginnen, zoals $m = 10$. Zodra de overige hyper-parameters min of meer vastliggen, kan de keuze van m verfijnd worden.

11.3 Een laatste woord

De keuze van de hyper-parameters is enorm moeilijk door de vele mogelijke waarden van iedere hyper-parameter en hun invloed op elkaar. Desalniettemin kunnen bovenstaande heuristieken als vuistregels dienen. Het grote belang van hyper-parameters maakt de keuze van de hyper-parameters een enorm populair onderwerp voor onderzoekers. Weet dat de heuristieken uit het ene onderzoek soms tegenstrijdig zijn met die uit een ander onderzoek. Volledigheidshalve vermelden we nog dat er geautomatiseerde technieken bestaan voor het bepalen van de hyper-parameters. Hiervoor kan geen neuraal netwerk gebruikt worden, om de eenvoudige reden dat men het probleem dan alleen verschuift: voor het netwerk dat de hyper-parameters bepaalt, moeten ook nog hyper-parameters bepaald worden.

12 Inleiding tot geavanceerde leeralgoritmes

We zijn ondertussen helemaal vertrouwd met het 'stochastic gradient descent' leeralgoritme. We hebben ook kunnen vaststellen dat het zeer goed werkt. In de praktijk worden vaak geavanceerdere leeralgoritmes gebruikt, die op stochastic gradient descent voortbouwen. Ten eerste wordt de 'Hessian techniek' toegelicht, die aan de basis ligt van de meeste geavanceerde leeralgoritmes. Ten tweede wordt 'Momentum-based gradient descent' uitgelegd als (eenvoudig) voorbeeld van zo'n geavanceerd leeralgoritme.

12.1 Hessian techniek

Het gradient descent algoritme (paragraaf 3.2) laat toe een kostfunctie C op een iteratieve manier te minimaliseren. Laat ons, net als in paragraaf 3.2, v definiëren als de vector bestaande uit alle N parameters van C. Het updaten van v gebeurt volgens de update-regel $v \to v - \eta \cdot \nabla C$. Herinner dat ∇C een vector is, bestaande uit de partieel afgeleide tegenover iedere variabele in v. Voor het updaten gebruiken we dus de eerste partieel afgeleiden, die info geven over de helling van de functie.

Om een nauwkeurigere update te doen, kunnen we ook de tweede partieel afgeleiden in acht nemen. Deze geven info over de wijziging van de helling; of de helling aan het toe- of afnemen is en in welke mate. Om de tweede partieel afgeleiden te berekenen, moet ieder van de N eerste partieel afgeleiden worden afgeleid tegenover alle N variabelen. In totaal krijgen we daardoor $N \cdot N$ tweede partieel afgeleiden. De Hessian matrix, H genoteerd, is een matrix bestaande uit al die tweede partieel afgeleiden, waarbij op index $[j][k]$ de waarde $\partial^2 C / \partial v_j \partial v_k$ staat. De update-regel wordt $v \to v - \eta \cdot H^{-1} \cdot \nabla C$. Merk op dat tussen H^{-1} en ∇C een matrixvermenigvuldiging plaatsvindt, en de vector ∇C dus beschouwd wordt als een matrix bestaande uit 1 kolom.

Door de tweede partieel afgeleiden te gebruiken, worden de parameters beter geüpdatet. Het gevolg hiervan, voor een neuraal netwerk, is dat het trainen sneller gaat; het netwerk convergeert in minder epochs naar de uiteindelijke classificatie nauwkeurigheid. Hoewel dit allemaal zeer mooi klinkt, is er één groot probleem: de Hessian matrix is enorm groot en bijgevolg kostelijk om te berekenen, té kostelijk in de praktijk.

De meeste geavanceerde leeralgoritmes baseren zich daarom wel op de Hessian techniek, maar berekenen de Hessian matrix niet. Zo gebruiken sommige algoritmes een benadering van de Hessian matrix. Andere algoritmes baseren zich alleen maar op het idee achter deze techniek (zijnde het gebruik van info over de wijziging van de helling). In het paper 'Efficient BackProp' van Yann LeCun et al. [21] worden enkele zeer bekende geavanceerde leeralgoritmes uitgelegd, waaronder Levenberg Marquardt (LM), Broyden-Fletcher-Goldfarb-Shanno (BFGS) en Conjugate Gradient Descent.

12.2 Momentum-based gradient descent

Een eenvoudig geavanceerd leeralgoritme is 'momentum-based gradient descent'. Dit populaire leeralgoritme baseert zich alleen op het idee achter de Hessian techniek. De Hessian matrix wordt dus niet berekend, ook geen benadering ervan.

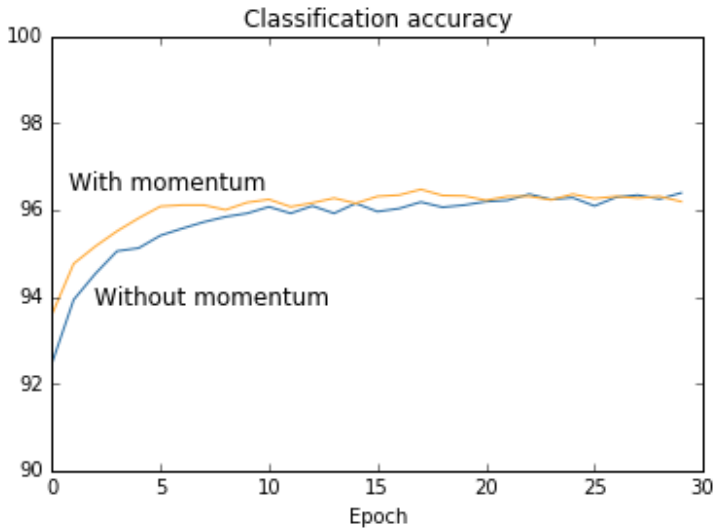
Voor iedere parameter (bias of gewicht) p van het neuraal netwerk wordt ook een snelheid s bijgehouden. De update-regels worden de volgende:

$$s \to \mu s - \eta \nabla C$$
$$p \to p + s$$

De rol van μ negeren we voorlopig, door $\mu = 1$ te beschouwen. Ten eerste wordt de snelheid s geüpdatet. Hiervoor wordt info over de helling (eerste afgeleide) gebruikt. Als het bergaf is, neemt de snelheid toe en hoe steiler de helling, hoe sterker de toename; analoog voor bergop. Meerdere opeenvolgende iteraties van bergaf, bijvoorbeeld, zorgen ervoor dat de snelheid alsmaar toeneemt. De snelheid geeft dus meer info over de helling, wat gelijkaardig is met het idee achter de Hessian techniek (die de wijziging van de helling in acht neemt). Ten tweede wordt p geüpdatet door er de snelheid bij op te tellen. Dit komt intuïtief neer op: de nieuwe positie van de parameter hangt af van zijn huidige snelheid.

We zullen nu de rol van μ bespreken. Als we $\mu = 1$ gebruiken, lopen we in de praktijk het risico dat de snelheid zodanig hoog wordt, dat de waarde van p opeens ver voorbij het minimum terecht komt; 'overshooting'. We moeten de snelheid daarom inperken, door, intuïtief gesproken, frictie te introduceren. De hyper-parameter μ heet de 'momentum co-efficient', en ligt tussen 0 en 1. Deze bepaalt hoeveel procent we van de oorspronkelijke snelheid overhouden. Daarom kunnen we $1-\mu$ als de 'frictie-coëfficiënt' beschouwen. Er is geen frictie als $\mu = 0$; er is enorme frictie als $\mu = 1$, wat neerkomt op de oude update-regel (stochastic gradient descent). Men mag μ nooit buiten het interval $[0; 1]$ kiezen. Indien $\mu > 1$, zal de snelheid na enkele iteraties alleen maar verhogen (in absolute waarde). Eens een beetje snelheid is opgebouwd, geldt dat $\mu s > \eta \nabla C$. Dit betekent dat de snelheid zelfs zou toenemen op een bergop. Indien $\mu < 0$ geldt een analoge redenering. De snelheid neemt dan iedere iteratie toe in absolute waarde, maar wisselt nu ook van teken.

Het implementeren van momentum-based gradient descent is zeer eenvoudig. We moeten gewoon de update regels aanpassen naar de bovenstaande. Deze minimale aanpassing geeft wel een degelijke verbetering. Figuur 35 illusteert het effect van het gebruik van snelheid. Door momentum-based gradient descent te gebruiken, convergeert het netwerk duidelijk sneller (in minder epochs) naar zijn uiteindelijke classificatie nauwkeurigheid; ongeveer 5 epochs tegenover 12 epochs. Aan het einde van de eerste epoch zijn al een aantal iteraties doorlopen, waardoor de parameters al wat snelheid hebben opgebouwd. Hierdoor staan de parameters na de eerste epoch al korter bij hun uiteindelijke waarden, waardoor de classificatie nauwkeurigheid al een stuk hoger is.



Figuur 35: Het gebruik van ‘momentum’ bij het trainen.

13 Toepassing: muziekgenre classificatie

In dit hoofdstuk wordt een toepassing geïmplementeerd die een muziekbestand classificeert op basis van het muziekgenre. Het ligt niet voor de hand hoe zo'n toepassing gerealiseerd zou kunnen worden met een conventioneel computerprogramma. Het spreekt voor zich dat onze implementatie van een neuraal netwerk gebruik zal maken.

13.1 Inleiding en gebruik

Om het precieze doel van deze toepassing duidelijk te maken, kan men best figuur 36 bekijken, die de afgewerkte toepassing toont.

Ten eerste dient de gebruiker linksboven de te ondersteunen muziekgenres aan te vinken. Ieder muziekbestand zal als één van deze genres geclassificeerd worden.

Ten tweede moet het netwerk getraind worden voor te ondersteunen genres. Hiervoor gebruiken we een bestaande gelabelde dataset, waar we later dieper op in gaan. Het neuraal netwerk zal het audiosignaal niet rechtstreeks gebruiken. Er worden namelijk verschillende eigenschappen ('features') van het audiosignaal bepaald, waar de gemiddelde BPM (Beats Per Minute) een eenvoudig voorbeeld van zou zijn. Het netwerk wordt getraind m.b.v. de features van ieder van de liedjes van de geselecteerde genres.

Zodra de training voltooid is, kan de gebruiker een lokaal opgeslagen muziekbestand selecteren en door het netwerk laten classificeren. Rechtsboven worden de resultaten van die classificatie getoond.

Verder voorziet de toepassing ook twee visualisaties om het verloop van de classificatie in de tijd duidelijk te maken. Links beneden staat een gewone plot van de nauwkeurigheid in functie van de tijd, en rechts beneden staat een spider diagram. Door op de 'Play'-knop te klikken, wordt het geselecteerde liedje afgespeeld en worden beide visualisaties iedere halve seconde geüpdatet. De classificatie op het huidige tijdstip gebeurt op basis van de eigenschappen van het liedje, vanaf het begin van het liedje tot het huidige tijdstip. Op die manier lijkt het dus alsof het neuraal netwerk het liedje beluistert en iedere halve seconde zijn classificatie updatet.

13.2 Literatuurstudie

We zullen een aantal bestaande werken rond het classificeren van muziek in genres beschouwen. Ten eerste zijn we geïnteresseerd in welke features van het audiosignaal beschouwd worden, om te weten dewelke we moeten extraheren. Ten tweede kijken we naar de behaalde resultaten binnen hun context (gebruikte dataset, genres en machine learning algoritme), om een ruwe indicatie te krijgen voor de nauwkeurigheid die ons neuraal netwerk zou moeten halen. Zo weten we of de beschreven toepassing überhaupt haalbaar is.

13.2.1 'Automatic Musical Genre Classification Of Audio Signals'

We beschouwen het paper getiteld 'Automatic Musical Genre Classification Of Audio Signals', van G. Tzanetakis, G. Essl en P. Cook [22].

Figuur 36: Screenshot van afgewerkte toepassing.

De dataset werd door de auteurs zelf opgesteld. Men verzamelde 50 liedjes van 6 verschillende genres, nl. klassiek, country, disco, hiphop, jazz en rock. Ieder liedje duurt 30 seconden. Hun dataset bevat ook muziekbestanden met spraak, waarrond ook experimenten werden uitgevoerd, maar daar gaan we niet op in. Voor de classificatie werd geen neuraal netwerk gebruikt, wel een 'Gaussian Bayes Classifier'. Dit is een ander, meer statistisch, machine learning algoritme.

Men deelde de beschouwde features op in twee groepen. Ten eerste waren er de 'Musical Surface Features', bestaande uit de centroid, rollof, flux en het aantal zerocrossings. Zowel hun gemiddelde waarden als hun standaarddeviaties werden beschouwd. De tweede groep waren 'Rhythm Features'. Dit zijn featurs afgeleid uit de BPM (Beats Per Minute). Op de precieze betekenis van de features gaan we niet in; dat heeft op zich niets met neurale netwerken te maken. Het merendeel van deze features komt wel in tabel 37 aan bod, waarin de features beschreven worden die we uiteindelijk zullen gebruiken in onze toepassing.

Het algoritme kon gemiddeld zo'n 62% van de liedjes correct classificeren. Opmerkelijk was dat het algoritme 90% van de liedjes die tot 'klassiek' of 'hip-hop' behoren correct classificeerde, terwijl dat voor de overige genres slechts zo'n 50% was. Men concludeerde ook dat de 'Musical Surface Features' belangrijker waren dan de 'Rhythm features'.

13.2.2 'Audio Feature Engineering for Automatic Music Genre Classification'

We beschouwen het paper getiteld 'Audio Feature Engineering for Automatic Music Genre Classification', van P. Annesi et al. [23].

Men experimenteerde met twee datasets. Ten eerste gebruikte men de 'Magnatune 2004 Corpus', die de genres rock, klassiek, jazz, elektronisch, metal en world onderscheidt; 6 genres. Deze dataset bestaat uit 729 liedjes. Ten tweede werd een zelf samengestelde dataset gebruikt, opgebouwd uit 500 liedjes. Deze noemde men 'RTV corpus' en onderscheidt de genres rock, klassiek, jazz, elektronisch en pop; 5 genres. Als machine learning algoritme werd een 'State Vector Machine' (SVM) gebruikt. Dit is een zeer populaire en, algemeen gesproken, krachtige vorm van machine learning.

De beschouwde features werden opgedeeld in verschillende groepen. De 'Basic Features' bestaan uit het volume, de spectral energy, centroid, pitch en MFCCs. Veel van deze features komen ook in tabel 37 voor. Merk op dat de 'pitch' overeenkomt met de 'Chroma Vector' uit de tabel. Verder is er de 'Beats' groep, die 5 features uit de BPM haalt. Op de overige twee groepen features gaan we niet in. Het zijn de gemiddeldes en standaarddeviaties van deze features die gebruikt worden.

Wanneer alleen de 'Basic Features' gebruikt werden, behaalde de SVM een gemiddelde nauwkeurigheid van 80.5% voor de 'Magnatune 2004 Corpus' en 89.4% voor de 'RTV Corpus'. Het gebruik van de 5 features in de 'Beats' groep had amper invloed op de nauwkeurigheid. Bij de 'Magnatune 2004 Corpus' viel het op dat de SVM alleen voor klassieke muziek een nauwkeurigheid hoger dan het gemiddelde haalde; voor de andere genres lag de nauwkeurigheid eronder.

13.2.3 'Music Genre Classification'

We beschouwen het paper getiteld 'Music Genre Classification', van M. Haggblade, Y. Hong en K. Kao [24].

Men gebruikte de dataset 'GTZAN Genre Collection'. Deze onderscheidt in principe 10 genres, maar men gebruikte er slechts 4 van: klassiek, jazz, metal en pop. Per genre zijn 100 liedjes van 30 seconden beschikbaar. Wat deze paper bijzonder interessant maakt, is dat met verschillende machine learning algoritmes geëxperimenteerd werd. Meer bepaald met een SVM, neuraal netwerk, k-nearest neighbor (k-NN) en k-means.

Deze paper onderscheidt zich ook van andere papers door alleen de eerste 15 Mel Frequency Cepstral Coefficients (MFCCs) te beschouwen. Er worden dus geen andere features gebruikt.

Ondanks het beperkte aantal gebruikte features halen ieder van de machine learning algoritmes hoge nauwkeurigheden. Zo behaalden k-means en k-NN gemiddeld zo'n 80% classificatie nauwkeurigheid. SVM haalde zo'n 87% en het neuraal netwerk zelfs 96%. Het neuraal netwerk haalde voor het genre jazz de hoogste nauwkeurigheid, terwijl de overige machine learning algoritmes daar net de laagste nauwkeurigheid voor behaalden. Voor klassieke muziek deden alle 4 de machine learning algoritmes het zeer goed.

13.2.4 'Automatic Music Genre Classification of Audio Signals with Machine Learning Approaches'

Ten slotte beschouwen we nog een paper van D. Chathuranga en L. Jayaratne, getiteld 'Automatic Music Genre Classification of Audio Signals with Machine Learning Approaches' [25].

Er werd met twee datasets geëxperimenteerd: de 'GTZAN Genre Collection' (nogmaals) en de 'ISMIR2004'. Men gebruikte alle 10 genres van de GTZAN dataset, zijnde blues, klassiek, country, disco, hiphop, jazz, metal, pop, reggae en rock. We herhalen dat deze database 100 liedjes bevat per genre. De ISMIR2004 dataset bevat 1458 liedjes, ongelijkmatig verdeeld over 6 genres: klassiek, elektronisch, jazz/blues, metal/punk, rock/pop en world. Als machine learning algoritme werd een SVM gebruikt.

Op het vlak van gebruikte features onderscheidde men twee groepen: 'Short Term Features' (frequentie, temporeel en cepstraal domein) en 'Long Term Features' (modulatie frequentie domein). In het temporeel domein beschouwde men de Zero Crossing Rate, amplitude gerelateerde features en Energy. In het frequentie domein beschouwde men de Spectral Flux en Spectral Centroid, Spectral Rolloff en Chroma. Ook werden de eerste 13 MFCCs gebruikt (in het cepstraal domein). Het modulatie frequentie domein omvat 3 ritmische features.

Wanneer alleen de 'Short Term Features' features gebruikt werden, haalde het SVM een gemiddelde classificatie nauwkeurigheid van 76.5% voor de GTZAN database. Wanneer alleen de 'Long Term Features' gebruikt werden, haalde men 74.1%. Door beide groepen tezamen te gebruiken, steeg de nauwkeurigheid naar 78%. Voor de ISMIR2004 dataset haalde de SVM dan een classificatie nauwkeurigheid van 80.66%.

13.2.5 Conclusies

Op het vlak van gebruikte features concluderen we dat er vaak een onderscheid wordt gemaakt tussen short- en long-term features.[10] Tabel 37 bevat praktisch alle short-term features die over de verschillende papers heen aan bod kwamen. We concluderen dat al deze features relevant zouden kunnen zijn; in paragraaf 13.5 gaan we hier dieper op in. Verder blijken de long-term features, die met de BPM te maken hebben, niet zo belangrijk zijn; de rol van de short-term features blijkt belangrijker.

Op basis van de gerapporteerde resultaten concluderen we dat het classificeren van muziek in genres zeker haalbaar is m.b.v. machine learning. In de literatuurstudie in paragraaf 13.2.3 stelde men zelfs vast dat het neuraal netwerk de beste resultaten behaalde van de 4 beschouwde machine learning algoritmes. We concluderen ook dat zowel het aantal genres als de grootte van de dataset een grote invloed hebben op de behaalde classificatie nauwkeurigheid. Uit ieder van de beschouwde papers blijkt dat de nauwkeurigheid voor het genre 'klassiek' het hoogst is (van alle beschouwde genres).

13.3 Dataset en feature extraction

Nu we weten dat het classificeren van muziek in genres zeer waarschijnlijk haalbaar is m.b.v. neurale netwerken, gaan we op zoek naar geschikte training data en een manier om aan feature extraction te doen.

13.3.1 Dataset

We gebruiken de dataset 'GTZAN Genre Collection' [26] van G. Tzanetakis binnen onze toepassing. Deze dataset is vrij te verkrijgen en enorm populair, zoals reeds uit de literatuurstudie bleek. De GTZAN dataset onderscheidt volgende 10 muziekgenres: blues, classical, country, disco, hiphop, jazz, metal, pop, reggae en rock. Per genre wordt een aparte folder voorzien met daarin 100 muziekbestanden (van dat genre). Ieder liedje duurt 30 seconden en het bestandsformaat is 'AU'.

Zelf delen we deze dataset verder op in de folders 'Train songs' en 'Test songs'. Per genre plaatsen we 5 willekeurige muziekbestanden in de folder 'Test songs', en de overige 95 in de folder 'Train songs'. Het spreekt voor zich dat we het neuraal netwerk zullen trainen m.b.v. de muziekbestanden in 'Train songs' en onze toepassing zullen uittesten met de muziekbestanden in 'Test songs'.

13.3.2 Feature extraction

Het neuraal netwerk moet allerlei eigenschappen beschouwen van het te classificeren audiosignaal om het bijhorende muziekgenre te bepalen. Het extraheren van die eigenschappen noemt men 'feature extraction'. Uit de literatuurstudie weten we dat de short-term features in tabel 37 relevant zijn. De Python-library *pyAudioAnalysis* [27], van T. Giannakopoulos, kan deze features extraheren uit een WAV-bestand. Daarom zal onze toepassing deze library gebruiken voor de feature extraction.

[10] Merk op dat beide groepen soms een andere benaming krijgen (maar dezelfde features representeren).

We zullen nu beschrijven hoe *pyAudioAnalysis* gebruikt kan worden om de feature extraction te realiseren voor een muziekbestand. Aangezien *pyAudioAnalysis* alleen met WAV-bestanden overweg kan, converteren we allereerst de muziekbestanden in de GTZAN dataset naar dit bestandsformaat. Hiervoor zijn vele gratis tools beschikbaar op het Web.

De features in tabel 37 opgesomd, worden voor een korte tijdspanne van het audiosignaal ('frame') geëxtraheerd. Volgens Giannakopoulos zelf is 0.1 seconde een geschikte waarde hiervoor [28]. Deze 34 features worden voor ieder opeenvolgend frame berekend. Het resultaat is dus een lange lijst van feature vectors. Meerdere opeenvolgende frames vormen samen een 'window'. We kiezen voor 0.5 seconden als lengte van het window, wat dus met 5 opeenvolgende frames overeenkomt. De motivatie hiervoor volgt nog. Voor iedere feature worden vervolgens het gemiddelde en de standaarddeviatie berekend over de 5 opeenvolgende frames heen. Dat gebeurt natuurlijk voor ieder window. Het resultaat hiervan is bijgevolg een (kortere) lijst van feature vectors die ieder uit 68 features bestaan. Ten slotte wordt per feature het gemiddelde berekend over alle windows heen. Het resultaat is nu 1 feature vector die uit 68 features bestaat.

Het bovenstaande wordt geïmplementeerd door de functie *dirWavFeatureExtraction* in 'audioFeatureExtraction.py' (deel van *pyAudioAnalysis*). Meer bepaald voert deze functie het bovenstaande uit voor ieder WAV-bestand in de opgegeven folder. Zo krijgen we een lijst van feature vectors terug; 1 feature vector per WAV-bestand. In het zelfgeschreven Python-script 'featureExtractor.py' roepen we deze functie op voor de folder 'blues' binnen de 'Train songs' folder, die 95 muziekbestanden van het genre 'blues' bevat. De lijst met 95 feature vectors wordt vervolgens naar het bestand 'blues.txt' weggeschreven. Dat doen we natuurlijk voor ieder van de 10 genres. De reden dat we de feature vectors per genre opslaan, en dus niet allemaal in éénzelfde bestand, is flexibiliteit. Voor het trainen kunnen we op die manier gemakkelijk de bestanden gebruiken, overeenkomstig met de genres die de gebruiker had aangeduid in de toepassing.

13.4 Proof of concept

In deze paragraaf zullen we een neuraal netwerk aanmaken, trainen en evalueren. We zullen aantonen dat het netwerk, in het algemeen, een hoge classificatie nauwkeurigheid haalt. Hierdoor weten we dat de uiteindelijke toepassing, in paragraaf 13.1 beschreven, bruikbaar zal zijn. Dit geldt dus als 'proof of concept'.

13.4.1 Voorbereiding

We maken gebruik van MATLAB [29] en de Neural Network Toolbox (NNT) [30] om met neurale netwerken te werken. MATLAB is in principe een enorm krachtige toepassing die voor allerlei wiskundige berekeningen gebruikt kan worden. Verder kunnen verschillende toolboxes geïntegreerd worden, waaronder de NNT. Deze toolbox laat toe neurale netwerken te creëren, te trainen en te evalueren. Het is een zeer geavanceerde toolbox die tegelijkertijd enorm makkelijk is in gebruik. Men kan uit vele kostfuncties, geavanceerde leeralgoritmes en activatiefuncties kiezen. Het spreekt voor zich dat alle functies die de NNT voorziet, waaronder het trainen van een netwerk, enorm snel zijn. Ook worden de meeste hyper-parameters automatisch ingevuld met geschikte waarden, maar kan men ze zelf instellen indien gewenst.

Feature ID	Feature Name	Description
1	Zero Crossing Rate	The rate of sign-changes of the signal during the duration of a particular frame.
2	Energy	The sum of squares of the signal values, normalized by the respective frame length.
3	Entropy of Energy	The entropy of sub-frames' normalized energies. It can be interpreted as a measure of abrupt changes.
4	Spectral Centroid	The center of gravity of the spectrum.
5	Spectral Spread	The second central moment of the spectrum.
6	Spectral Entropy	Entropy of the normalized spectral energies for a set of sub-frames.
7	Spectral Flux	The squared difference between the normalized magnitudes of the spectra of the two successive frames.
8	Spectral Rolloff	The frequency below which 90% of the magnitude distribution of the spectrum is concentrated.
9-21	MFCCs	Mel Frequency Cepstral Coefficients form a cepstral representation where the frequency bands are not linear but distributed according to the mel-scale.
22-33	Chroma Vector	A 12-element representation of the spectral energy where the bins represent the 12 equal-tempered pitch classes of western-type music (semitone spacing).
34	Chroma Deviation	The standard deviation of the 12 chroma coefficients.

Figuur 37: De 34 features die *pyAudioAnalysis* kan extraheren [27].

Voor deze test zullen we alle muziekbestanden gebruiken. De feature vectors voor de muziekbestanden in de folder 'Train songs' hebben we al gegenereerd (zie voorgaande paragraaf). Die voor de 'Test songs' extraheren door het script 'featureExtractor.py' uit te voeren op die folder.

13.4.2 Testing script

Om de test eenvoudig te kunnen herhalen, schrijven we het MATLAB-script genaamd 'PoC.m'.

Ten eerste stellen we een lijst op met de namen van de te beschouwen muziekgenres. Deze lijst kunnen we makkelijk aanpassen om zo de invloed van het aantal beschouwde genres vast te kunnen stellen bijvoorbeeld. Vervolgens laden we de tekstbestanden met de feature vectors van de te beschouwen genres in (zowel die in 'Train songs' als die in 'Test songs'). Al deze feature vectors worden in de lijst x bijgehouden, die de inputs voor het netwerk voorstelt. Natuurlijk hebben we ook een lijst van targets, t, nodig die de gewenste overeenkomstige outputs voorstelt. Stel dat we n genres beschouwen en een bepaalde input tot het i-de genre behoort, dan is de bijhorende target een lijst van lengte n, bestaande uit allemaal 0'en en op de i-de index een 1. Merk op dat x en t samen de gelabelde dataset vormen.

Ten tweede maken we een nieuw neuraal netwerk aan. Het netwerk gebruikt 1 hidden layer, bestaande uit 50 hidden neuronen. Het aantal hidden neuronen werd experimenteel bepaald (zie volgende paragraaf). De default kostunctie is de cross-entropy kostfunctie, die we in paragraaf 7.1 hebben besproken. De default activatiefunctie voor de hidden layer is de tanh, in paragraaf 9.1 toegelicht.[11] Voor de outputlaag is dat de softmax, in paragraaf 2.4.1 uitgelegd.

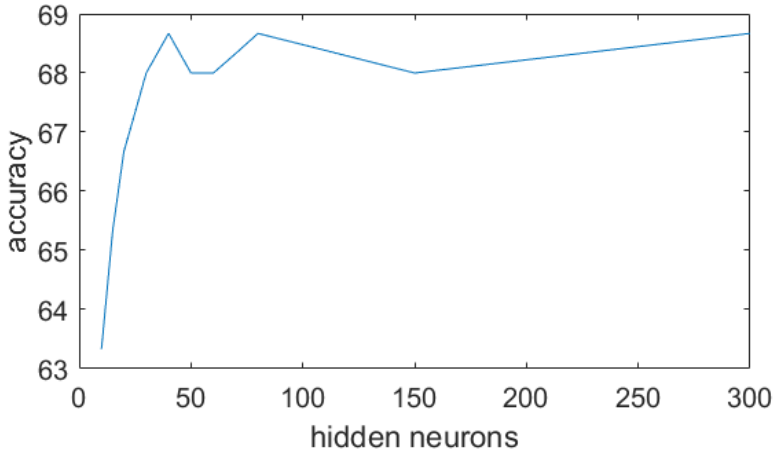
Ten derde trainen we het gecreëerde netwerk. Standaard wordt de gelabelde data willekeurig opgedeeld in 70% training data, 15% validatie data en 15% test data. Het default leeralgoritme heet 'Scaled Conjugate Gradient Descent'. Ook dit geavanceerde leeralgoritme is gebaseerd op de Hessian techniek, zoals in hoofdstuk 12 uitgelegd, en is enorm snel. Tijdens het trainen wordt bovendien 'early stopping' toegepast, wat in paragraaf 11.2.1 werd toegelicht. Meer bepaald stopt het trainen als er gedurende 6 epochs geen verbetering is van de kost voor de validatie data.

Ten slotte evalueren we het getrainde netwerk op basis van de test data, en plotten we de bijhorende 'confusion matrix'. Wat dit is, wordt in volgende paragraaf besproken. Het is dus duidelijk dat we vertrouwd zijn met de basisprincipes die de NNT hanteert. De NTT voegt alleen optimalisaties toe t.o.v. onze bestaande kennis, en voorziet natuurlijk een zeer snelle implementatie.

13.4.3 Experimenten en conclusies

Met 'PoC.m' kunnen we eenvoudig een aantal experimenten uitvoeren. Laat ons om te beginnen met het aantal hidden neuronen experimenteren. De plot in figuur 38 toont de behaalde classificatie nauwkeurigheid in functie van het aantal hidden neuronen dat gebruikt werd, waarbij alle 10 genres beschouwd werden. De nauwkeurigheid werd bepaald voor 10, 15, 20, 25, 30, 40, 50, 60,

[11] Eigenlijk wordt de MATLAB-functie *tansig* gebruikt, die een snelle benadering is van de tanh.



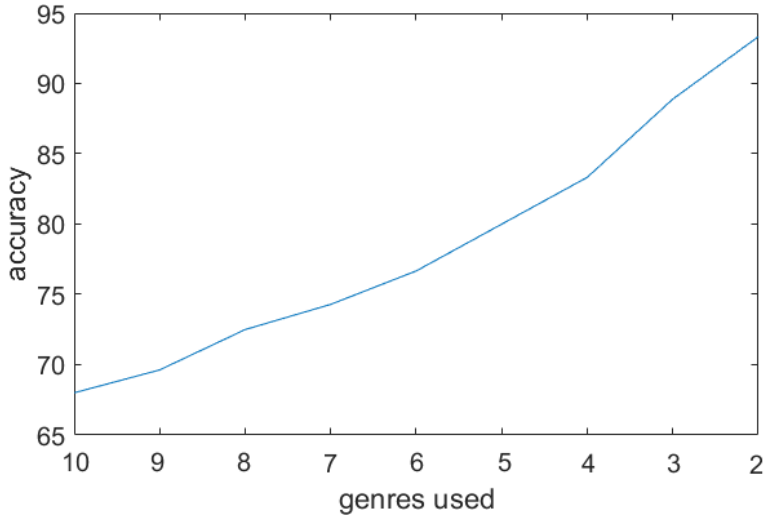
Figuur 38: Plot van de behaalde classificatie nauwkeurigheid in functie van het aantal hidden neuronen dat gebruikt werd.

70, 80, 150 en 300 hidden neuronen.[12] Het is zeer waarschijnlijk dat de kleine onregelmatigheden in het verloop van de nauwkeurigheid te wijten zijn aan de willekeurige initialisatie van de parameters. We mogen daarom concluderen dat 50 hidden neuronen een geschikte keuze is: door er minder te gebruiken, daalt de nauwkeurigheid sterk, en door er meer te gebruiken, stijgt de nauwkeurigheid amper of niet. De bijhorende classificatie nauwkeurigheid is 68.00%.

We zullen nu experimenteren met het aantal te beschouwen genres. We verwachten dat de nauwkeurigheid toeneemt naarmate het netwerk minder genres moet beschouwen. Zoals we reeds weten, halen we voor 10 genres een nauwkeurigheid van 68.00%. Vervolgens negeren we één willekeurig genre, zodat er nog 9 beschouwd worden. De nauwkeurigheid stijgt naar 69.63%. Dit experiment herhalen we voor 8, 7, 6, 5, 4, 3 en 2 beschouwde genres. Figuur 39 plot de behaalde classificatie nauwkeurigheden. Zoals verwacht neemt de nauwkeurigheid toe naarmate er minder genres beschouwd worden. Een interessante observatie is het (bij benadering) lineaire verband; er is geen extra toename vanaf een bepaald aantal beschouwde genres. Vermits de laagst mogelijke nauwkeurigheid (gemiddeld) 68.00% is, is deze 'proof of concept' geslaagd en mogen we concluderen dat onze toepassing bruikbaar zal zijn.

Laat ons ten slotte de classificatie nauwkeurigheden van de verschillende genres met elkaar vergelijken. We beschouwen alle 10 genres. Figuur 40 toont de bijhorende 'confusion matrix'. De cel op rij i en kolom j bevat het aantal (absoluut en procentueel) liedjes dat tot het j-de genre behoort en als i-de genre geclassificeerd werd. De cel helemaal rechts beneden geeft aan dat 68.0% van de liedjes tot het correcte genre geclassificeerd werden. De laatste rij van de confusion matrix toont het procentueel aantal correct geclassificeerde liedjes per genre. Door dit experiment te herhalen, valt het op dat de classificatie nauwkeurigheid voor het tweede genre, 'klassiek', steeds de hoogste is van alle genres. Dat concludeerden we ook uit de literatuurstudie (zie paragraaf 13.2).

[12] In al deze experimenten werd 'PoC.m' 200 keer uitgevoerd. De resulterende nauwkeurigheid is de mediaan van de 200 bijhorende nauwkeurigheden.



Figuur 39: Plot van de behaalde classificatie nauwkeurigheid in functie van het aantal beschouwde genres.

Voor de andere genres verschillen de classificatie nauwkeurigheden te sterk van uitvoering tot uitvoering, waardoor we geen verdere conclusies kunnen maken.

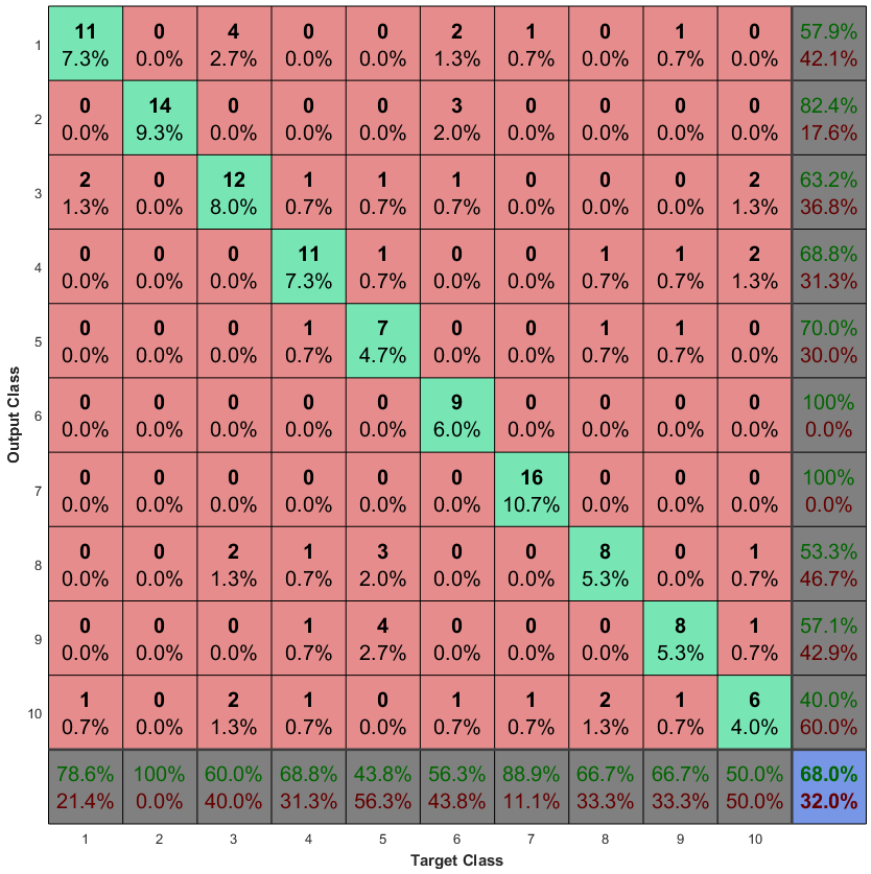
13.5 Feature selection

Herinner dat ons neuraal netwerk maar liefst 68 features (het gemiddelde en de standaarddeviatie van de 34 features in tabel 37 opgelijst) van het muziekbestand beschouwd om het te classificeren. We kunnen ons nu terecht afvragen of ál deze features relevant zijn. In deze paragraaf trachten we te bepalen welke features van belang zijn om een goed presterend netwerk te bekomen, wat men 'feature selection' noemt [5]. Concreet betekent dit dat we op zoek gaan naar een deelverzameling van features die voor een hoge classificatie nauwkeurigheid (of lage kost) zorgt. Door alle redundante en onbelangrijke features te negeren, vermindert de dimensionaliteit van de inputvectoren en zal het netwerk bovendien sneller getraind kunnen worden.

13.5.1 Methoden

Om precies te bepalen welke features van belang zijn, zouden we alle 2^{68} mogelijke combinaties van features moeten beschouwen. Vervolgens zouden we bepalen hoe goed het netwerk presteert (classificatie nauwkeurigheid of kost) voor ieder van die deelverzamelingen van de features, en die resultaten met elkaar vergelijken. Deze methode zou enorm tijdrovend zijn, zeker als we het experiment meermaals zouden uitvoeren om statisch verantwoorde conclusies te kunnen maken. Daarom stappen we over naar heuristieken, die sneller zijn maar een suboptimale oplossing geven.

Zo zouden we het netwerk op iedere individuele feature kunnen evalueren, wat slechts 68 mogelijkheden zijn, om zo de relevantie van iedere feature apart te bepalen. Dit is echter geen goede methode: twee features die individueel



Figuur 40: Confusion matrix wanneer alle 10 genres beschouwd worden.

onbelangrijk zijn, kunnen dat tezamen soms wel zijn [5]. Deze methode houdt hier vanzelfsprekend geen rekening mee.

Een populaire heuristiek om aan feature selection te doen, is 'sequential forward selection' (SFS) [5]. Deze methode vertrekt van een lege lijst beschouwde features en gaat er herhaaldelijk de meest relevante, niet eerder beschouwde feature aan toevoegen. De precieze te doorlopen stappen zijn:

1. Beschouw de lege verzameling als huidige selectie S. Beschouw alle 68 features als op dit moment beschikbare features F.
2. Stel de $|F|$[13] mogelijke deelverzamelingen op, bestaande uit S en één feature f uit F.
3. Bepaal de performantie (classificatie nauwkeurigheid of kost) van het netwerk voor ieder van die deelverzamelingen. Bepaal de verzameling P waarvoor het netwerk het best presteerde. Sla P op met de bijhorende performantie.
4. Beschouw de feature f in P. Voeg deze toe aan S, en verwijder deze uit F.
5. Herhaal stappen 2 t.e.m. 4 tot F leeg is.

Uit de opgeslagen resultaten kunnen we eenvoudig afleiden hoe de performantie verloopt in functie van het aantal beschouwde features. Ook kan telkens opgevraagd worden welke features hierbij beschouwd werden. Het nadeel van deze aanpak is (opnieuw) dat naar individuele bijdrages gekeken wordt, waardoor twee features die individueel onbelangrijk zijn, maar dat tezamen wel zijn, niet geselecteerd zullen worden.

Nu bovenstaande methode werd toegelicht, kan 'sequential backward selection' (SBS) worden uitgelegd [5]. Deze heuristiek gaat omgekeerd te werk: vertrekkend van alle features wordt herhaaldelijk de minst relevante feature verwijderd.

1. Beschouw de verzameling van alle 68 features als huidige selectie S.
2. Stel de $|S|$ mogelijke deelverzamelingen op, bestaande uit S waaruit één feature f verwijderd werd.
3. Bepaal de performantie (classificatie nauwkeurigheid of kost) van het netwerk voor ieder van die deelverzamelingen. Bepaal de verzameling P waarvoor het netwerk het best presteerde. Sla P op met de bijhorende performantie.
4. Beschouw de feature f in P. Verwijder deze uit S.
5. Herhaal stappen 2 t.e.m. 4 tot S leeg is.

Deze aanpak heeft als voordeel tegenover SFS dat features die tezamen wel belangrijk zijn, geselecteerd zullen worden. Het nadeel is echter dat deze aanpak computationeel zwaarder is: het netwerk moet de verzamelingen bestaande uit vele features vaak evalueren, terwijl dat bij SFS de verzamelingen zijn die uit weinig features bestaan.

[13] $|V|$ staat voor de kardinaliteit van verzameling V; het aantal elementen in V.

13.5.2 Implementatie en basistest

Om zelf aan feature selection te doen, implementeren we SBS, met het bovenstaande als motivatie. We noemen het bijhorende MATLAB-script 'FeatureSelection.m'. Voor het bepalen van een performantie (in stap 3) gaan we "voorzichtig" te werk. Herinner namelijk dat de performantie van het netwerk ook zal afhangen van de willekeurige initialisatie van de gewichten en biases. Daarom bepalen we de performantie 11 keer.[14] We beschouwen de mediaan daarvan als uiteindelijke performantie. Zo zijn de resultaten statistisch gezien meer verantwoord. We gebruiken hetzelfde netwerk als in paragraaf 13.4.2 geïntroduceerd werd. De resultaten van de feature selection, zijnde de lijst met performanties en bijhorende deelverzamelingen van features, worden opgeslagen om ze later eenvoudig te kunnen analyseren. Als maat voor de performantie van het netwerk gebruiken we het procentuele aantal misclassificaties. We kijken naar het aantal misclassificaties omdat we dit aantal, net als de kost, willen minimaliseren. Hierdoor kunnen we eenvoudig bepalen of de gebruikte maat voor verschillende resultaten zorgt.

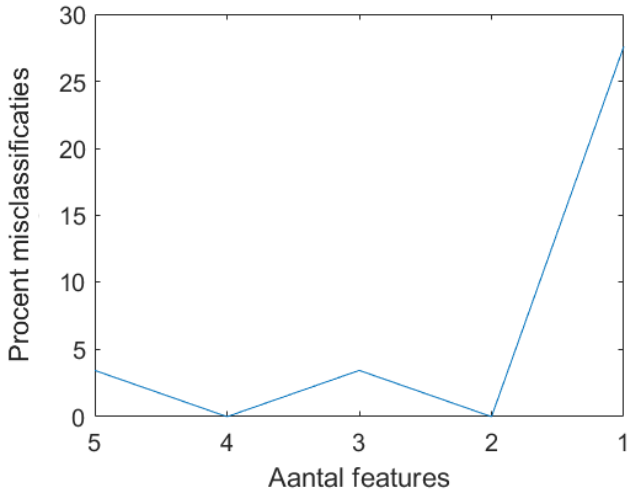
Vooraleer we trachten te bepalen welke van de 68 features van belang zijn in de context van muziekgenres, voeren we een basistest uit. We zullen deze test niet alleen gebruiken om vertrouwd te raken met feature selection, maar vooral om de observaties bij de eigenlijke experimenten (zie paragraaf 13.5.3) te kunnen kaderen. De basistest omvat het toepassen van feature selection in volgende triviale situatie. Een inputvector voor het netwerk bestaat uit 5 getallen, $(x_1, x_2, x_3, x_4, x_5)$, tussen 0 en 1 gelegen. We zullen het netwerk trainen om te bepalen (classificeren) of het eerste getal kleiner is dan het tweede. De gewenste output van het netwerk bestaat bijgevolg uit twee componenten: de eerste is de logische waarde $(x_1 < x_2)$, en de tweede is de logische waarde $(x_1 \geq x_2)$. We benadrukken dat x_3, x_4 en x_5 nutteloos zijn. Dit verwachten we dan ook vast te kunnen stellen bij het analyseren van de resultaten van de feature selection. De training set bestaat uit 200 willekeurig gegenereerde samples.

Ten eerste plotten we het verloop van het procentueel aantal misclassificaties in functie van het aantal beschouwde features, wat figuur 41 illustreert. Bij iedere herhaling van deze basistest krijgen we gelijkaardige resultaten. Dit geldt ook als we de kost gebruiken als maat voor de performantie van het netwerk. Steeds neemt het procentueel aantal misclassificaties toe wanneer slechts 1 feature beschouwd wordt, wat te verwachten is aangezien er 2 features wél van belang zijn. Ten tweede vragen we op welke features beschouwd werden, en dus relevant zijn, wanneer er 2 features beschouwd werden. Dit zijn steeds de verwachte features x_1 en x_2.

13.5.3 Experiment en resultaten

Nu we vertrouwd zijn met feature selection en het interpreteren van de bijhorende resultaten, kunnen we overgaan tot het echte experiment: bepalen welke van de 68 features relevant zijn voor het classificeren van muziekbestanden in muziekgenres. Bij dit experiment zullen we op dezelfde manier te werk gaan als voor de basistest. We voeren 'FeatureSelection.m' uit waarbij alle muziekbestanden (zowel die in 'Train songs' als die in 'Test songs') van alle 10 genres

[14] De motivatie voor die '11 keer' volgt in volgende paragraaf.



Figuur 41: Het verloop van het procentueel aantal misclassificaties in functie van het aantal beschouwde features voor de basistest.

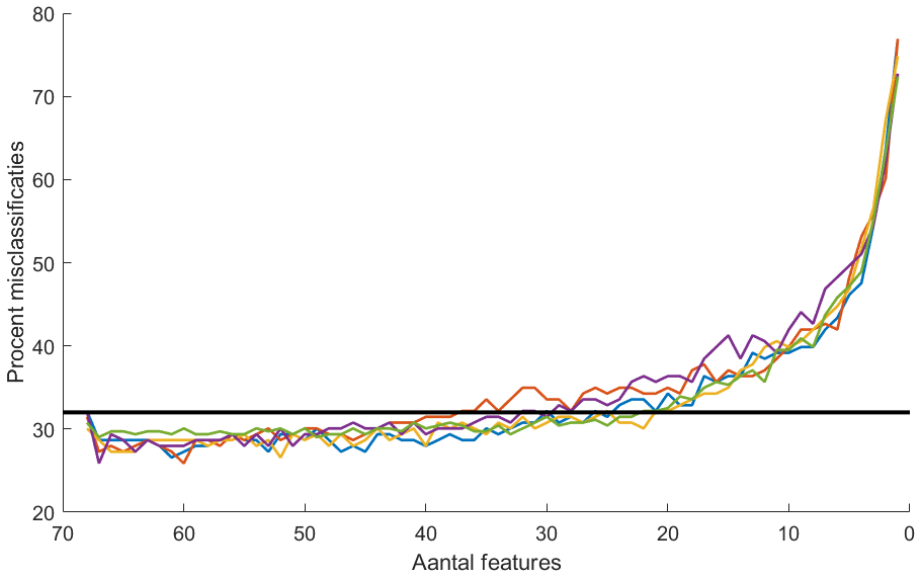
beschouwd worden. We blijven het netwerk uit paragraaf 13.4.2 gebruiken. Het eenmalig uitvoeren van dit experiment duurt zo'n 5 uur, wat de motivatie is achter het "slechts" 11 keer bepalen van de performantie (zie paragraaf 13.5.2).

Dit experiment werd 5 keer herhaald. Figuur 42 plot het verloop van het procentueel aantal misclassificaties in functie van het aantal beschouwde features voor de 5 uitvoeringen. We zien duidelijk dat de performanties kort bij elkaar liggen. Herinner dat we een classificatie nauwkeurigheid van 68.00% haalden in de proof of concept, en het procentueel aantal misclassificaties dus 32.00% is. De horizontale lijn in de figuur visualiseert dit. We concluderen dat het aantal misclassificaties vermindert als we slechts 60 tot 40 features zouden beschouwen. Een andere opmerkelijke conclusie is dat het netwerk nog steeds goed zou presteren indien slechts 20 van de 68 features beschouwd zouden worden. Daarna neemt de performantie van het netwerk wel sterk af. Het ziet er naar uit dat de feature selection een grote impact zal hebben op de features die we uiteindelijk zullen beschouwen bij de implementatie van de toepassing.

We zijn natuurlijk geïnteresseerd in welke features beschouwd werden. Daarom zullen we nagaan of bij iedere uitvoering van het experiment ongeveer dezelfde features beschouwd werden. Zo ja, weten we dat die features de belangrijkste zijn. De geselecteerde features worden gerepresenteerd door een binaire vector (van lengte 68), waarbij een '1' aanduidt dat de feature geselecteerd is en een '0' aanduidt dat de feature niet geselecteerd is. Om de mate van gelijkheid van twee binaire vectoren A en B te meten, gebruiken we de 'Jaccard similarity' [31], gedefinieerd als:

$$J = \frac{|A \wedge B|}{|A \vee B|}.$$

Wanneer vele features beschouwd worden, verwachten we een kleine Jaccard similarity (≈ 0). Uit figuur 42 bleek immers dat er vele onbelangrijke features zijn, waardoor de volgorde van de keuze van die onbelangrijke features niet al-



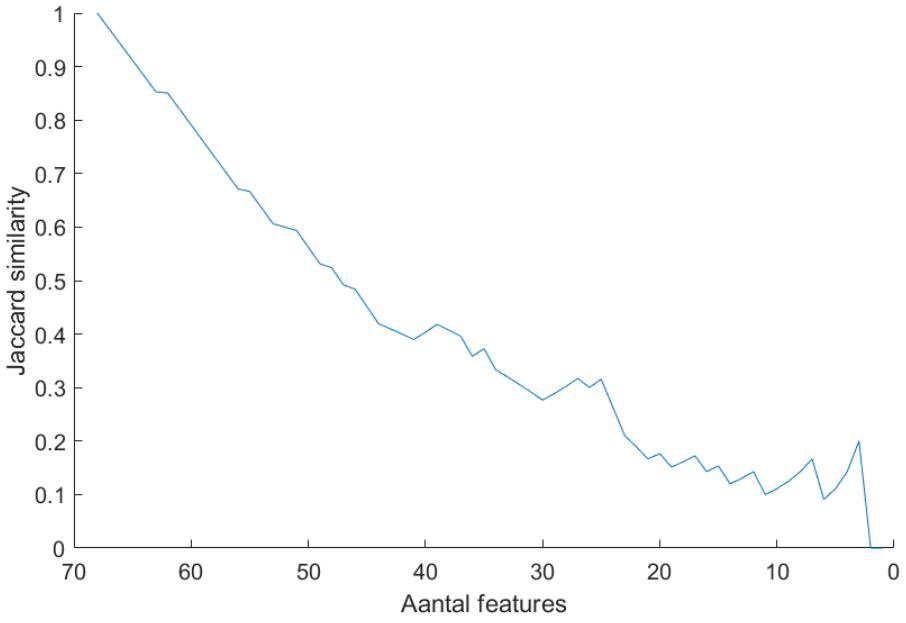
Figuur 42: Het verloop van het procentueel aantal misclassificaties in functie van het aantal beschouwde features voor iedere herhaling van het experiment. De horizontale lijn staat op 32.

tijd dezelfde zal zijn. Wanneer weinig features beschouwd worden, verwachten we een grote Jaccard similarity (≈ 1). Dan worden immers alleen de belangrijke features beschouwd. We verwachten dat die features bij iedere uitvoering ongeveer dezelfde zullen zijn. Figuur 43 toont het verloop van de Jaccard similarity in functie van het aantal beschouwde features. Voor ieder beschouwd paar experimenten verkrijgen we een analoog verloop. In tegenstelling tot de verwachtingen, verkrijgen we een kleine Jaccard similarity wanneer weinig features beschouwd worden. Dit betekent dat de beschouwde features sterk van elkaar verschillen over de verschillende uitvoeringen. Hierdoor kunnen we geen conclusie maken over welke features de belangrijke zijn en zullen we bij de implementatie van de toepassing alle 68 features blijven beschouwen.

13.6 Implementatie

Net zoals voor de 'HandwritingGUI' (uit paragraaf 5.3) beperken we ons tot het bespreken van de voornaamste aspecten op het vlak van implementatie. De toepassing wordt in Python geïmplementeerd, maar zal ook van MATLAB's NTT gebruik maken.

Trainen van het netwerk:
Ten eerste implementeren we het MATLAB-script genaamd 'TrainNetwork.m', met als parameter de lijst van te beschouwen genres. De training set bestaat uit de feature vectors (van die genres) in de folder 'Train songs'. Vermits we geen nood hebben aan test data, wordt 85% van de data voor training gebruikt en 15% voor validatie. Ten slotte wordt het getrainde netwerk lokaal opgeslagen als 'net.mat', zodat het later gebruikt kan worden voor het classificeren van een



Figuur 43: Het verloop van de Jaccard similarity in functie van het aantal beschouwde features voor een willekeurig paar experimenten.

liedje.

Wanneer de gebruiker in de Python-toepassing op de 'Train network'-knop klikt, wordt simpelweg 'TrainNetwork.m' opgeroepen met de door de gebruiker geselecteerde genres als parameter. Het oproepen van een MATLAB-script vanuit Python is mogelijk dankzij de 'MATLAB Engine API for Python' [32], die bij de installatie van MATLAB zit.

Classificeren van een liedje:
Wanneer de gebruiker op de 'Classify song'-knop klikt, wordt ten eerste iteratieve feature extraction gedaan voor het WAV-bestand dat de gebruiker geselecteerd heeft. We gebruiken *pyAudioAnalysis* voor de feature extraction tot op het niveau van een 'window' (zie paragraaf 13.3.2). Herinner dat we als lengte van het window voor 0.5 seconden kozen. Het resultaaat is bijgevolg een lijst van feature vectors, waarbij de eerste vector de features voor de eerste halve seconde van het liedje representeert, de tweede vector de features voor de tweede halve seconde van het liedje representeert, enz.

Ten tweede worden deze feature vectors "iteratief uitgemiddeld". We stellen een lijst op waarvan het i-de element het gemiddelde is van de eerste i feature vectors. Laat ons deze lijst *vecs* noemen. Vervolgens roepen we vanuit Python het MATLAB-script genaamd 'Classify.m' op, met *vecs* als parameter. Dit (zelfgeschreven) script laadt het eerder getrainde netwerk ('net.mat') in, en classificeert ieder van de gemiddeldes in *vecs*. De resulterende lijst met kansdistributies, die we *res* noemen, zullen we gebruiken voor de visualisaties.

Ten derde tonen we rechtsboven het eigenlijke resultaat van de classificatie van het zelfgekozen liedje. Daartoe gebruiken we eenvoudigweg het laatste ele-

ment in res, vermits die classificatie gebeurde op basis van de features over het volledige liedje.

Visualisaties:
Wanneer de gebruiker beneden op de afspeel-knop klikt, begint ten eerste het liedje af te spelen. Hiervoor gebruiken we de Python-library genaamd 'pyglet' [33]. De pauze-knop, stop-knop en slider zijn vanzelfsprekend gekoppeld aan overeenkomstige functies uit pyglet.

Ten tweede initialiseren we de plot en het spider diagram. Bij het initialiseren van de plot wordt simpelweg een lijnplot gemaakt per genre, met als x-coördinaten de opeenvolgende halve seconden $(0.0, 0.5, 1.0, \dots)$ en als y-coördinaten de opeenvolgende kansen in res voor dat genre. Hiervoor gebruiken we de Python-library 'matplotlib' [34], die deel uitmaakt van MATLAB. Bij het initialiseren van het spider diagram wordt de webstructuur geplot. Deze is afhankelijk van het aantal beschouwde genres en vereist een aantal eenvoudige berekeningen met hoeken om de gepaste eindpunten van de benen van het web te bepalen.

Ten derde worden de visualisaties iedere halve seconde geüpdatet. Voor het updaten van de plot wordt alleen de verticale balk verplaatst over een afstand van 0.5 naar rechts. Voor het updaten van het spider diagram visualiseren we telkens de volgende kansdistributie in res, begonnen vanaf de eerste natuurlijk. Laat ons de huidige kansdistributie $curRes$ noemen. Merk op dat de hoekpunten van de eigenlijke figuur op de benen van het web liggen. De afstand van zo'n hoekpunt tot het centrum van het web komt overeen met de kans die $curRes$ aangeeft voor het overeenkomstige genre.

14 Universal approximation theorem

In dit hoofdstuk zullen we aantonen dat een neuraal netwerk eender welke functie kan berekenen. Herinner dat we in paragraaf 1.1 uitlegden dat een neuraal netwerk de 'verborgen functie' in de training set probeert te achterhalen. Dit bewijs toont dus aan dat een neuraal netwerk die functie sowieso kan representeren, wat die ook moge zijn.

14.1 Inleiding

Het bovenstaande was natuurlijk slechts een informele beschrijving. De formele stelling heet het 'Universal Approximation Theorem' (UAT), en stelt dat het om een benadering van de verborgen functie gaat. Wel kan men die benadering steeds nauwkeuriger maken door meer neuronen te gebruiken in het neuraal netwerk. Met 'nauwkeuriger maken' bedoelen we dat het verschil tussen de benadering en de echte functie (de 'error') alsmaar kleiner wordt, voor iedere input. Verder stelt het UAT dat de benadering een continue functie is. Een neuraal netwerk berekent immers steeds een continue functie. Hierdoor zijn ze minder geschikt zijn voor het benaderen van niet-continue functies. Het UAT stelt dat een netwerk met slechts 1 hidden layer volstaat, wat zeer opmerkelijk is. Weet ten slotte dat het UAT geen beperking oplegt aan het aantal inputs en outputs van de functie.

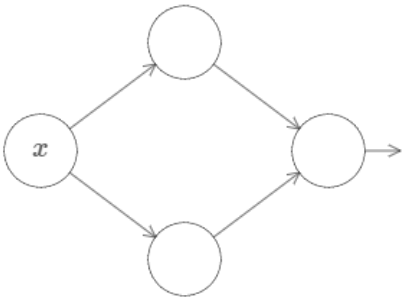
Een formeel bewijs van deze stelling blijkt zeer wiskundig [1]. Vaak is het te moeilijk om te begrijpen, tenzij men een enorme wiskundige kennis heeft. Daarom beschrijft dit hoofdstuk een visueel bewijs, waaruit wél duidelijk zal blijken dat een netwerk eender welke functie kan berekenen. Weet dus dat het onderstaande geen formeel bewijs is. Allereerst bewijzen we het UAT voor een neuraal netwerk dat 2 hidden layers gebruikt. Met de inzichten die we hierbij zullen verwerven, tonen we vervolgens aan dat een neuraal netwerk met slechts 1 hidden layer volstaat.

Merk op dat dit bewijs existentieel is: het toont alleen aan dat een neuraal netwerk in theorie eender welke functie kan berekenen; het beschrijft geen leeralgoritme dat dit ook kan realiseren in de praktijk. We zullen aantonen dat wij, als mens, een neuraal netwerk kunnen opbouwen dat een willekeurige functie benadert. Dit vereist dat de te benaderen functie gekend is, wat voor de eerder besproken 'verborgen functie' niet het geval is. Daarom is de stelling in de praktijk niet erg nuttig. Het verklaart ook de eerder beschreven heuristiek die stelt dat netwerken met meerdere hidden layers ('deep neural networks') complexere beslissingen kunnen nemen. In theorie volstaat 1 hidden layer dus, maar in de praktijk niet.

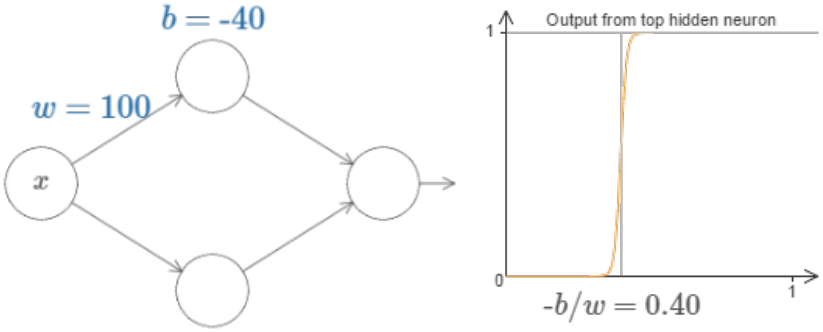
14.2 Eén input – één output

Ten eerste tonen we het specifieke geval 'één input – één output' aan. In volgende paragrafen zullen we dit geleidelijk veralgemenen naar 'meerdere inputs – meerdere outputs'.

Beschouw de eenvoudige netwerkarchitectuur die in figuur 44 getoond wordt. We beschouwen de activatie $a = \sigma(wx + b)$ van het bovenste hidden neuron. Herinner, ten eerste, uit paragraaf 2.3 dat een sigmoid neuron zich in limiet als een perceptron gedraagt. Voor een grote w en b wordt de logistic sigmoid



Figuur 44: Eenvoudige netwerkarchitectuur met 1 input en 1 output.



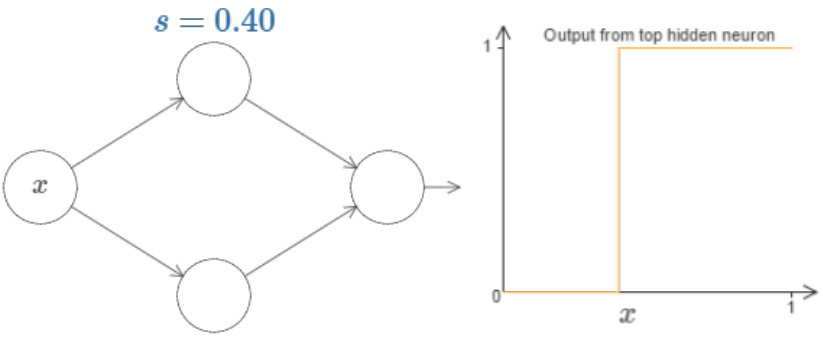
Figuur 45: Creëren van een stapfunctie met de stap op 0.40.

namelijk de stapfunctie, bij benadering. Herinner, ten tweede, dat de positie van de stap zich op $z = 0$ bevindt. Dit komt overeen met $z = wx + b = 0$, en dus met $wx = -b$, en dus met $x = -b/w$. Figuur 45 illustreert dit alles voor de activatie van het bovenste hidden neuron in het eenvoudige netwerk.

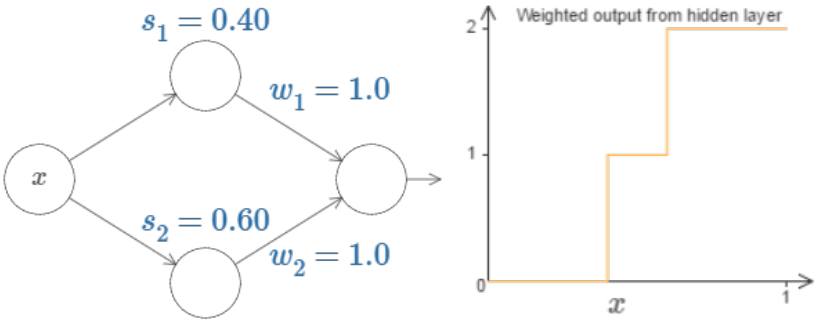
We kunnen dus (bij benadering) een stapfunctie creëren met de stap op de gewenste positie s als volgt. We kiezen een constant groot gewicht, zoals $w = 1000$, en de bias $b = -ws$. Naarmate we w groter maken en b gepast updaten, verkrijgen we een alsmaar nauwkeurigere benadering van de stapfunctie met de stap op positie s. Laat ons daarom in het vervolg de echte stapfunctie gebruiken, en de figuren vereenvoudigen door meteen de positie van de stap s te noteren. Figuur 46 illustreert dit.

Voor het onderste hidden neuron geldt natuurlijk hetzelfde. Laat ons nu de gewogen som in het outputneuron beschouwen; niet de outputactivatie. Voor de bias stellen we $b = 0$. Het is dus de gewogen som van 2 stapfuncties waarvan we de posities van de stap, s_1 en s_2, kunnen instellen. Als beide gewichten 1.0 zijn, verkrijgen we vanzelfsprekend het resultaat dat in figuur 47 getoond wordt. Als de gewichten tegengesteld zijn aan elkaar, $w_1 = -w_2$, krijgen we een bult. Met de absolute waarde van de gewichten kunnen we de stapfuncties schaleren, en de hoogte van de bult dus instellen. Laat ons die hoogte met h noteren en de begin- en eindpositie van de stap binnenin het neuron schrijven. Figuur 48 geeft een voorbeeld. Merk op dat als de hoogte negatief is, de bult naar beneden wijst.

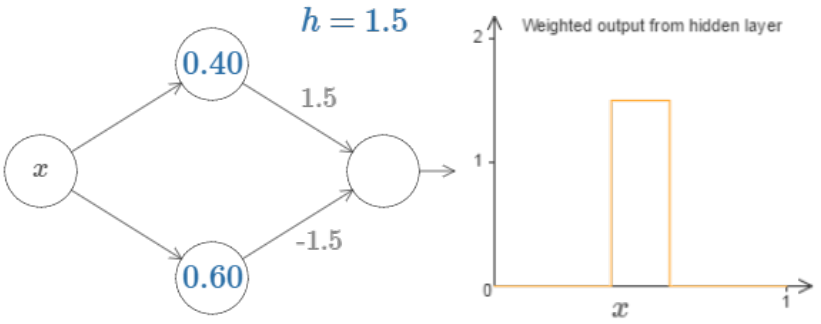
Van één bult kunnen we dus de beginpositie, eindpositie en hoogte instel-



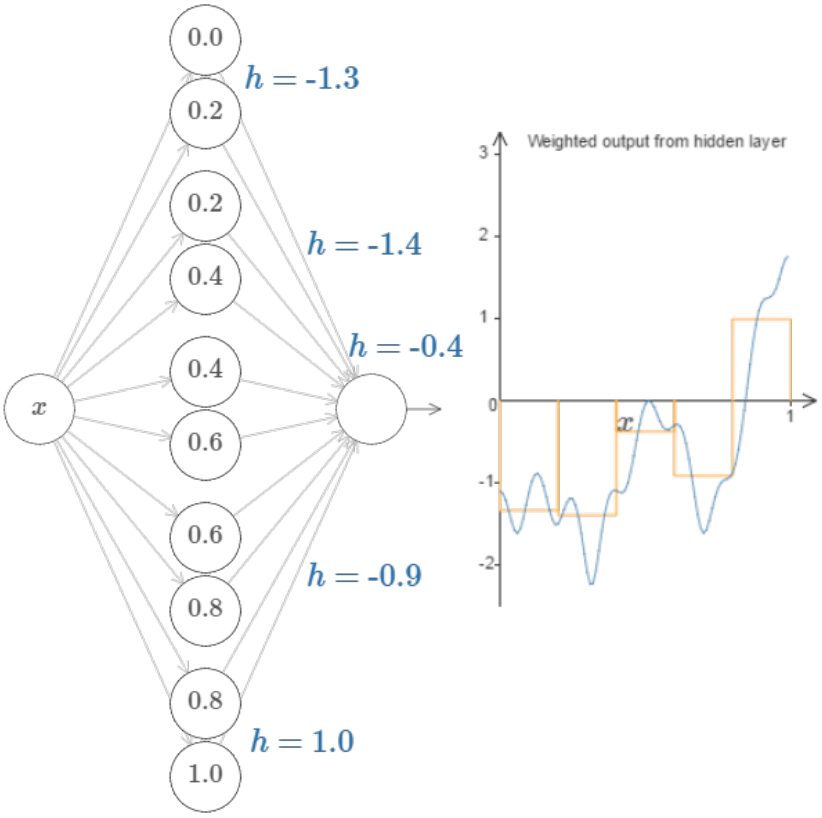
Figuur 46: Vereenvoudigde notatie.



Figuur 47: Eenvoudige gewogen som.



Figuur 48: Een bult waarvan de beginpositie, eindpositie en hoogte ingesteld kunnen worden.

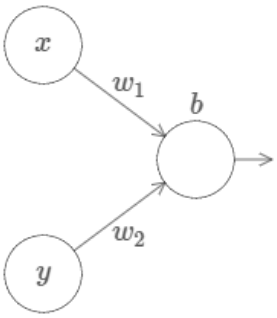


Figuur 49: Functiebenadering door meerdere aaneensluitende bulten.

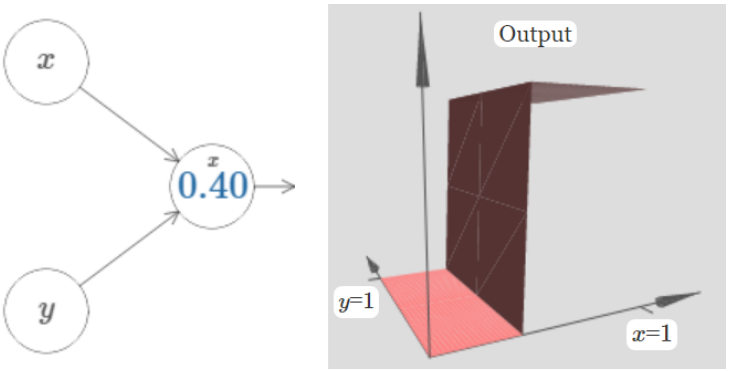
len. Zo'n bult is de basiscomponent voor een functiebenadering. We kunnen deze netwerkarchitectuur namelijk herhalen, zodat de gewogen som (met $b = 0$) meerdere bulten sommeert. Door de bulten op elkaar te laten aansluiten qua positie, ze smal genoeg te maken en gepaste hoogtes in te stellen, kunnen we een benadering voor een functie opstellen. Figuur 49 illustreert dit. Het spreekt voor zich dat de benadering alsmaar nauwkeuriger wordt door meer neuronen, en dus smallere bulten, te gebruiken.

We willen echter dat de output van het netwerk, de outputactivatie dus, de benadering voor de functie $f(x)$ is; niet de gewogen som. Als we de identiteitsfunctie gebruiken als activatiefunctie voor het outputneuron, is $a = z$ en krijgen we bijgevolg het gewenste resultaat. Zoals we in paragraaf 2.4.2 aangaven, gaat men typisch zo te werk in regressie toepassingen. Indien het outputneuron een andere activatiefunctie gebruikt, zoals de logistic sigmoid σ, is dat natuurlijk niet het geval. Om te zorgen dat $\sigma(z) \approx f(x)$, moeten we bepalen welke functie z dan moet benaderen. Dat is $\sigma^{-1}(f(x))$, want dan geldt $\sigma(z) = \sigma(\sigma^{-1}(f(x)) = f(x)$.

We hebben nu het UAT bewezen voor het geval 'één input – één output', waarbij het neuraal netwerk slechts 1 hidden layer gebruikt.



Figuur 50: Eenvoudige netwerkarchitectuur met 2 inputs en 1 output.



Figuur 51: Creëren van een stapfunctie langs de x-as met de stap op 0.40.

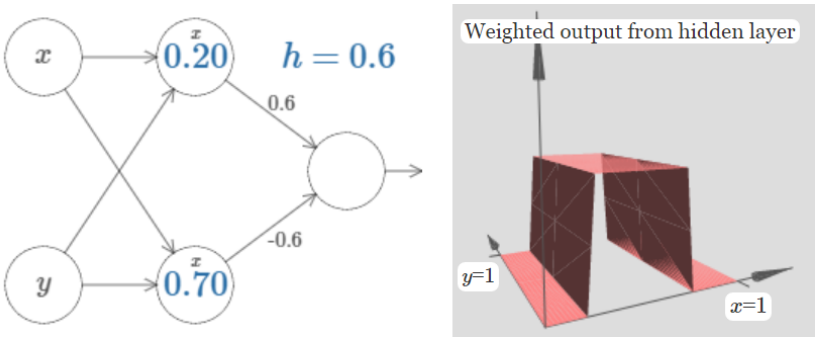
14.3 Twee inputs – één output

We beschouwen nu een netwerk met 2 inputs, x en y, en 1 output. Om het UAT voor dit geval te bewijzen, zullen we de uitleg uit voorgaande paragraaf veralgemenen, van 2D naar 3D dus.

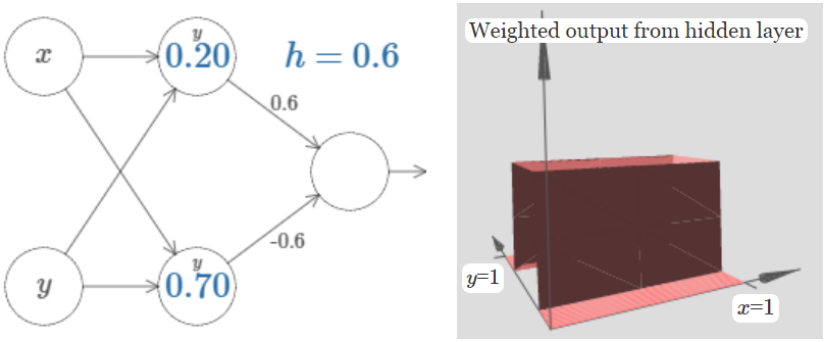
Beschouw de eenvoudige netwerkarchitectuur met 2 inputs, x en y, in figuur 50 getoond. Beschouw de activatie van het outputneuron. Als we $w_2 = 0$ stellen, telt alleen de input x mee. De situatie is nu analoog met die in het begin van voorgaande paragraaf. Met een groot gewicht en een gepast bias, kunnen we dus een stapfunctie creëren langs de x-as met de stap op de gewenste positie, maar nu in 3D natuurlijk. Figuur 51 geeft een voorbeeld. Let op de letter x in het outputneuron, om aan te geven dat de stapfunctie langs de x-as is.

Opnieuw kunnen we twee zulke constructies combineren, gebruik makend van tegengestelde gewichten. De gewogen som (met $b = 0$) wordt nu een rug in x-richting; het 3D-equivalent van de 2D-bult. Opnieuw kunnen we eenvoudig de hoogte h, begin- en eindpositie instellen. Figuur 52 geeft een voorbeeld. Herinner dat de twee gewichten tussen input y en de eerste hidden layer 0.0 zijn.

Dit kunnen we natuurlijk ook, volledig analoog, langs de y-as doen. Figuur 53 illustreert dit. Merk de letter y op in de hidden neurons. Het wordt interessant wanneer de gewogen som een x- en y-rug met dezelfde hoogte h combineert (met $b = 0$). Figuur 54 illustreert dit. Merk op dat de verbindingen waarvoor



Figuur 52: Een rug in x-richting; het 3D equivalent van de 2D bult.

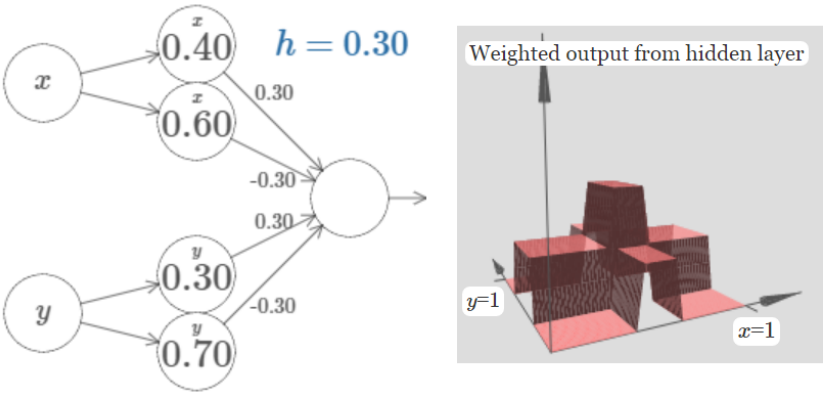


Figuur 53: Een rug in y-richting; het 3D equivalent van de 2D bult.

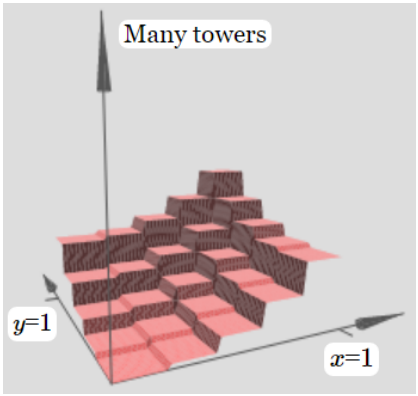
$w = 0$ niet meer getoond worden. Laat ons de gevormde structuur een piramide noemen. Het plateau heeft hoogte h, en het centrale torentje heeft hoogte $2h$. Idealiter zou de gewogen som alleen het centrale torentje vormen. Door vele torentjes te combineren kan dan eenvoudig een 3D-functie benaderd worden. Figuur 55 illustreert dit voor een denkbeeldige functie.

We zullen daarom de piramide herleiden naar een torentje. Beschouw daartoe het gedrag van de stapfunctie als volgt. Als $z = wx+b < 0$, mapt stapfunctie z naar 0. Dus als $wx < -b$, wordt de gewogen input wx naar 0 gemapt. Analoog voor $z > 0$ natuurlijk. De bias b vormt dus de treshold voor de mapping van de gewogen input wx. Beschouw nu opnieuw de piramide, zijnde de gewogen input wx. Laat ons een treshold plaatsen ter hoogte van $3h/2$; in het midden tussen het plateau en de centrale toren. Het plateau en de grond ($<$ treshold) worden dan naar 0 gemapt, en de centrale toren ($>$ treshold) naar 1. Deze mapping (activatie) geeft dus het gewenste gedrag. Herinner dat we grote gewichten en een groot bias moeten gebruiken opdat de sigmoid de stapfunctie zou benaderen. Laat ons daarom een vast groot gewicht gebruiken, wat overeenkomt met een grote waarde voor h. Als treshold kiezen we natuurlijk $b = -3h/2$. De outputactivatie geeft dan een torentje met hoogte 1, op de gewenste positie. Figuur 56 illustreert dit.

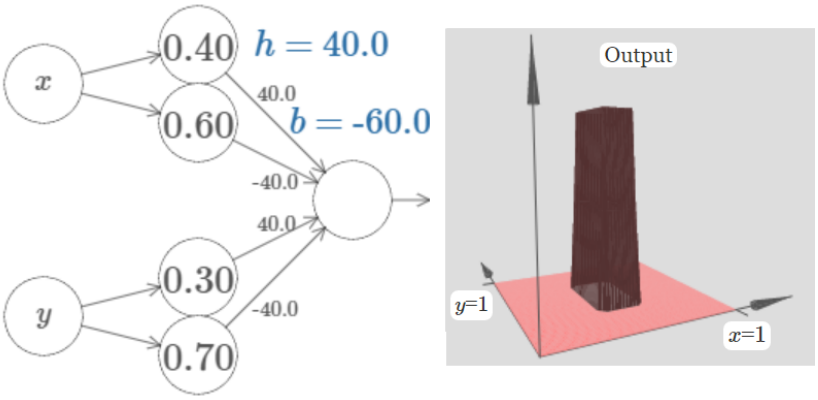
Meerdere zulke torentjes kunnen we combineren (gewogen som). De gewichten bepalen de hoogte van ieder torentje, en als bias kiezen we $b = 0$. Figuur 57 illustreert dit. Het spreekt voor zich dat we een 3D functie kunnen benaderen



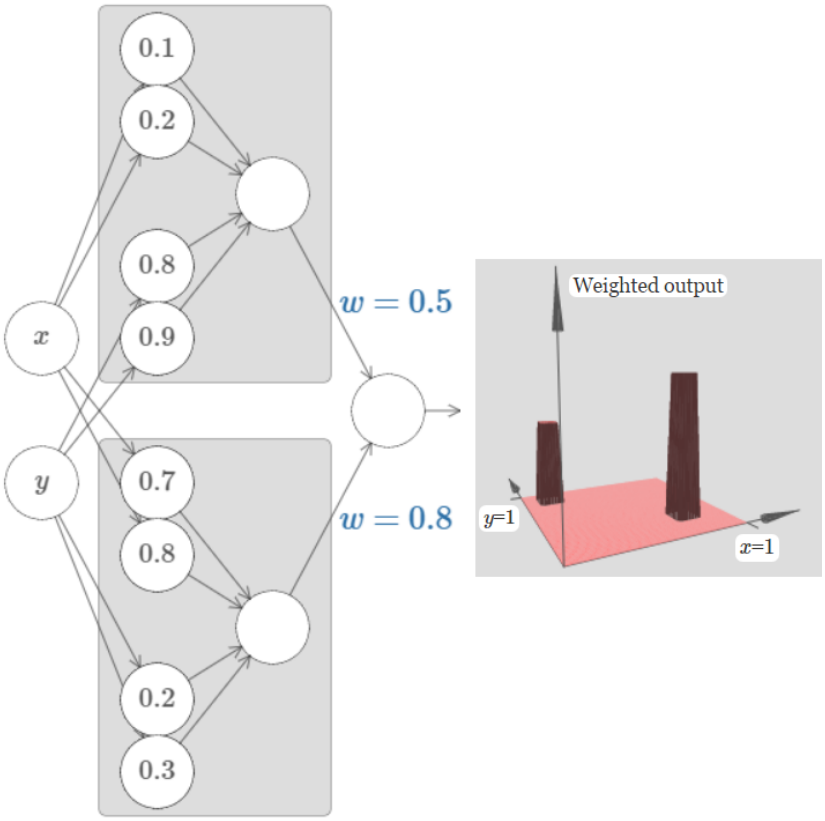
Figuur 54: Gewogen som van een x- en y-rug geeft een piramide.



Figuur 55: Benadering van een denkbeeldige 3D-functie, gebruik makend van torentjes.



Figuur 56: Activatie van outputneuron is een torentje met hoogte 1.



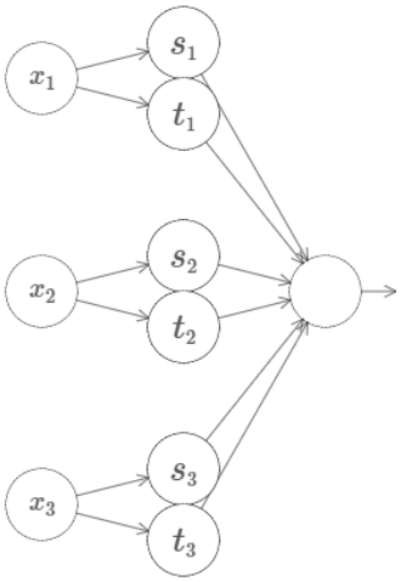
Figuur 57: Meerdere torentjes.

door vele torentjes op elkaar te laten aansluiten, zoals figuur 55 eerder al illustreerde. Hoe meer torentjes we gebruiken, en hoe fijner de torentjes dus, hoe preciezer de benadering. Het verhaal over de activatiefunctie van het outputneuron is analoog aan dat in de vorige paragraaf. Zo kunnen we simpelweg de identiteitsfunctie gebruiken. Als we de sigmoid wensen te gebruiken, moet de gewogen som in het outputneuron de functie $\sigma^{-1}(f(x, y))$ benaderen.

We hebben nu het UAT bewezen voor het geval 'twee inputs – één output', althans voor een neuraal netwerk dat twee hidden layers gebruikt. Later zullen we aantonen dat 1 hidden layer volstaat.

14.4 Meerdere inputs – meerdere outputs

De veralgemening van '2 inputs – 1 output' naar '3 inputs – 1 output' (4D) is eenvoudig. Figuur 58 toont een eerste stap in de gepaste netwerkarchitectuur. Opnieuw kunnen we, m.b.v. een groot gewicht en gepast bias, zorgen dat een activatie in de eerste hidden layer een 4D-stapfunctie geeft. De gewogen som (met tegengestelde gewichten en $b = 0$) van één paar stapfuncties geeft het 4D-equivalent van een rug met hoogte h. Dit kunnen we natuurlijk voor ieder van de 3 inputvariabelen doen. De gewogen som van de 3 ruggen met hoogte h geeft (het equivalent van) een piramide. Het equivalent van de centrale toren



Figuur 58: De outputactivatie geeft het 4D equivalent van de 3D tofen.

heeft nu een hoogte van $3h$ en de overige structuur heeft hoogtes h en $2h$. Door vervolgens een grote waarde voor h te kiezen en als bias $b = -5h/2$, wordt de activatie (het equivalent van) een toren. Merk op dat $5h/2$ midden tussen $2h$ en $3h$ in ligt. Om vertrekkende van deze basiscomponent een functie te benaderen, gaat men volledig analoog aan voorgaande paragraaf te werk.

Het bovenstaande kan natuurlijk eenvoudig veralgemeend worden naar 'meerdere inputs – één output'. Het enige wat verandert, is de bias. De algemene formule voor de bias is: $b = -(m - 1/2)h$, met m het aantal inputvariabelen.

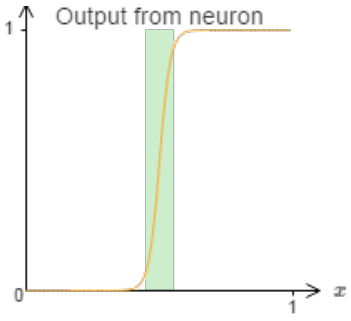
De veralgemening naar meerdere, n, outputs is nog eenvoudiger. We kunnen de functie namelijk simpelweg als n aparte functies zien. Het komt er dus op neer de uitleg over 'meerdere inputs – één output' n keer te herhalen en in één groot netwerk te combineren.

We hebben het UAT nu voor een willekeurige functie bewezen, althans voor een neuraal netwerk dat twee hidden layers gebruikt. In paragraaf 14.6 tonen we aan dat het ook met slechts 1 hidden layer kan.

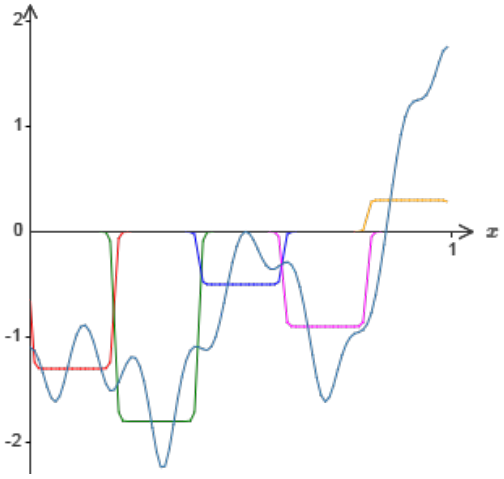
14.5 Stapfunctie verbeteren en vereisten van de activatiefunctie

14.5.1 Stapfunctie verbeteren

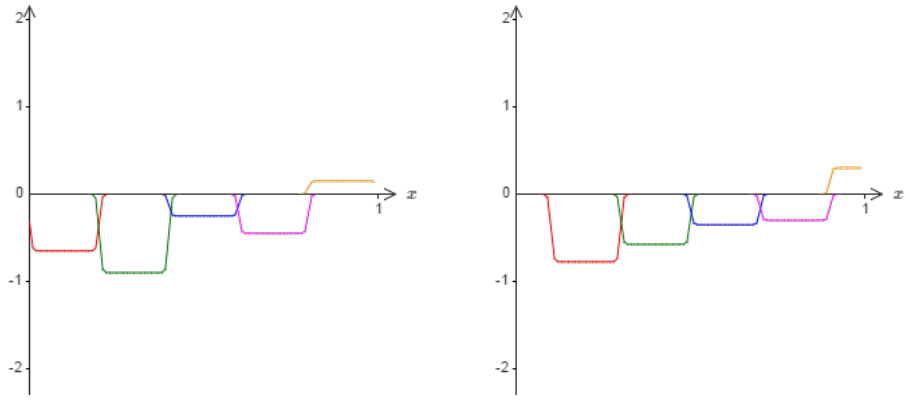
Herinner dat we in voorgaande paragrafen steeds de echte stapfunctie gebruikten. De motivatie hierachter was dat de sigmoid een goede benadering voor de stapfunctie is, op voorwaarde dat een groot gewicht en bias gebruikt worden. Vermits het een benadering is, is er een foutmarge t.o.v. de echte stapfunctie. Figuur 59 illustreert dit. Laat ons het gekleurde gebied de 'window of failure' noemen, vermits de benadering daar incorrect is. Figuur 60 illustreert de rol van de window of failure in de context van een functiebenadering.



Figuur 59: Window of failure van de sigmoid t.o.v. de stapfunctie.



Figuur 60: Illustratie van 'window of failure' voor een functiebenadering.



Figuur 61: Halvering van f_1, die we f_1' noemen (links); Halvering van f_2, die we f_2' noemen (rechts).

Door grotere gewichten en biases te gebruiken, vermindert het probleem uiteraard. De volgende techniek biedt echter een betere aanpak.

Beschouw een eerste benadering van de functie f, die we f_1 noemen en in figuur 60 wordt afgebeeld. De bulten beginnen op 0.0 en zijn 1.0 breed. Laat ons f_1 nu halveren. Die functie noemen we f_1' en wordt aan de linkerkant in figuur 61 afgebeeld. Beschouw nu een tweede benadering van f, genaamd f_2, waarbij de bulten op 0.5 beginnen (en 1.0 breed zijn). Laat ons f_2 nu halveren. Die functie noemen we f_2' en wordt aan de rechterkant in figuur 61 afgebeeld. De functie f wordt dan benaderd door $f_1' + f_2'$. Het idee achter deze aanpak is dat een punt dat binnen een window of failure van de ene functie valt, niet binnen een window of failure van de andere functie valt.

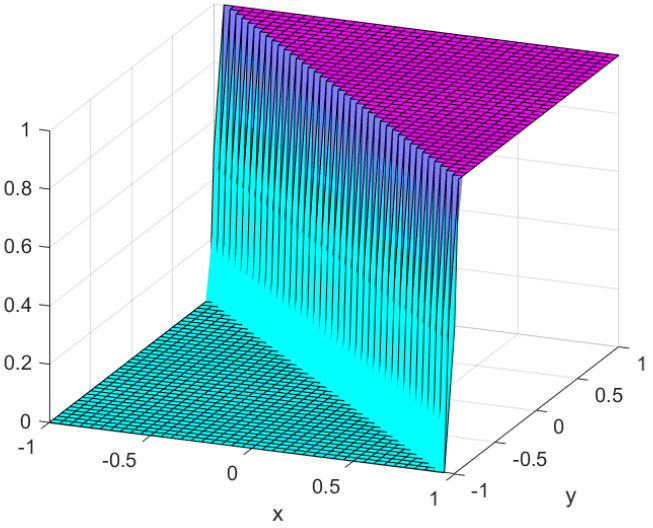
Dit idee kan veralgemeend worden naar het gebruik van n benaderingen, i.p.v. slechts twee. Laat ons stellen dat de breedte van een bult 1.0 is. De bulten van benadering f_i (met $i \in [0; n[$) beginnen dan op i/n. De benadering die uiteindelijk gebruikt wordt, is natuurlijk $f_i' = f_i/n$. Iedere mogelijke input valt in hoogstens 1 window of failure[15], waardoor de benadering nu zeer nauwkeurig wordt. Het spreekt voor zich dat deze aanpak ook voor het algemene geval, 'meerdere inputs – meerdere outputs', bruikbaar is.

14.5.2 Vereisten van de activatiefunctie

In het hele bewijs over het UAT gebruikten we de logistic sigmoid als activatiefunctie, maar dat had ook een andere kunnen zijn. De enige vereiste van de activatiefunctie $f(z)$ is namelijk dat deze omgevormd kan worden naar een stapfunctie. Daartoe moeten $\lim_{z\to-\infty} f(z)$ en $\lim_{z\to+\infty} f(z)$ gedefinieerd zijn (en dus $\neq \pm\infty$), en verschillend zijn van elkaar.

De beide sigmoids die we in deze tekst beschouwd hebben, zijnde de logistic sigmoid en de tanh, voldoen hier uiteraard aan. De softmax beschouwen we niet, vermits deze alleen in outputlagen gebruikt wordt. De identiteitsfunctie voldoet niet aan de vereiste: $\lim_{z\to-\infty} f(z) = -\infty$; analoog voor $+\infty$. Zoals we in paragraaf 2.4.2 uitlegden, kan een netwerk dat alleen de lineaire activatiefunctie

[15] Voor een relatief brede window of failure geldt dit natuurlijk niet.



Figuur 62: Schuine stapfunctie voor $z = 200x + 200y$.

gebruikt, alleen een lineaire functie voorstellen; niet eender welke functie. Desondanks de ReLU niet aan de vereiste voldoet ($\lim_{z\to+\infty} f(z) = +\infty$), kan deze toch gebruikt worden[35]. Daar gaan we niet verder op in.

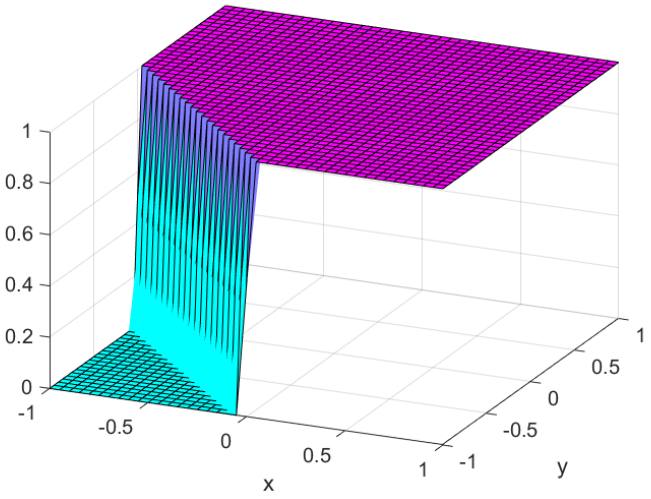
14.6 Gebruik van 1 hidden layer

Na het bewijs van het UAT voor een netwerk dat twee hidden layers gebruikt, zullen we nu aantonen dat 1 hidden layer volstaat. Hiervoor maken we gebruik van de ideeën in voorgaande paragrafen besproken. We beschouwen het geval met 2 inputs x en y, en één output.

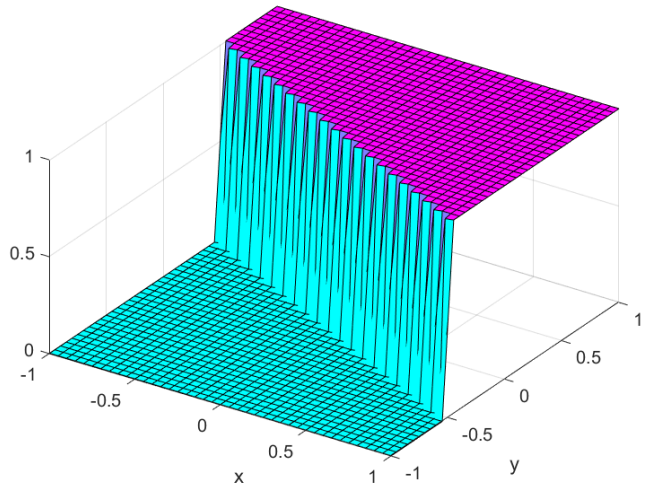
We kijken opnieuw naar de activatie van een neuron in de eerste hidden layer, zoals in figuur 50 afgebeeld. Als we $w_2 = 0.0$ kozen, kregen we een stapfunctie in x-richting; analoog voor y-richting. Het zou daarom logisch moeten lijken dat als beide gewichten $\neq 0.0$, we een geroteerde stapfunctie in 3D krijgen. Figuur 62 illustreert dit voor $w_1 = w_2 = 200$. We zien dat de stapfunctie nu een hoek van 45° maakt met de x-as. Met de bias kunnen we opnieuw de positie van de stap verschuiven; de hoek blijft ongewijzigd. Figuur 63 illustreert dit voor $b = 200$. Door de verhouding tussen w_1 en w_2 te wijzigen, kan de hoek van de stapfunctie ingesteld worden. Figuur 64 illustreert dit voor $w_1 = 100$ en $w_2 = 200$. Dat de hoek met de x-as verkleind is, mag niet verbazen. Vermits $w_1 = 100$ het kleinste gewicht is, en dus korter bij 0 ligt, gaat de stapfunctie namelijk richting 'een stapfunctie langs de y-as' (waarvoor $w_1 = 0$ en de hoek met de x-as 0° is).

Laat ons nu de gewogen som in de output layer beschouwen. Met één paar stapfuncties (met tegengestelde gewichten) stelt de gewogen som een geroteerde rug voor. Als we de rug zeer smal maken, wordt de rug in limiet een lijn, waarvan we de richting, positie en hoogte kunnen instellen.

Stel dat we vele lijnen gebruiken, laat ons zeggen n, die allemaal door



Figuur 63: Schuine stapfunctie voor $z = 200x + 200y + 200$.



Figuur 64: Schuine stapfunctie voor $z = 100x + 200y$.

éénzelfde punt gaan, een verschillende hoek hebben en eenzelfde hoogte h hebben. De gewogen som is dan, bij benadering, een plateau van hoogte h met in het midden een ronde toren van hoogte $n \cdot h$. Indien we nu in de gewogen som als bias $b = -h$ kiezen, valt het plateau weg, en houden we alleen een ronde toren van hoogte $(n-1) \cdot h$ over. Zo kunnen we dus een ronde toren creëren met de gewenste positie en hoogte.

Met de gewogen som kunnen we vele ronde torens op elkaar laten aansluiten qua positie. Als bias kiezen we dan de som van de plateauhoogtes van ieder van de torens, maal -1. Op die manier krijgen we uiteraard een benadering voor de functie. Dit hele idee kan (opnieuw) veralgemeend worden naar het algemene geval ('meerdere inputs – meerdere outputs'). Het probleem van de 'window of failure' bij een rechthoekige toren kennen we. Bij ronde torens manifesteert dit zich vanzelfsprekend sterker, precies door de ronde vorm. Maar we kunnen ook dezelfde aanpak gebruiken om het probleem in te perken.

15 Deep neural networks

15.1 Inleiding

We hebben ondertussen uitgebreid kennis opgedaan rond neurale netwerken met één hidden layer, zgn. shallow neural networks. Uit het vorige hoofdstuk weten we dat zo'n netwerk eender welke functie kan voorstellen. Het probleem is echter dat het netwerk de verborgen functie in de training data niet altijd kan ontdekken. Een 'deep neural network' (DNN) kan hier een oplossing voor bieden. Zulke netwerken bestaan uit meerdere hidden layers, wat als meerdere abstractieniveaus gezien kan worden. Doordat iedere laag alsmaar abstractere beslissingen neemt, wordt het voor het neuraal netwerk makkelijker om ook een complexe verborgen functie te ontdekken in de training data.

In paragraaf 2.2.2 werd dit geïllustreerd met het voorbeeld over het herkennen van een auto op een afbeelding. Een andere analogie is het programmeren van een ingewikkelde functie. Door vele hulpfuncties te gebruiken, kan het implementeren van de functie op een hoog (abstract) niveau gebeuren. De hulpfuncties nemen minder abstracte beslissingen en kunnen, indien nodig, nog verder worden opgedeeld in hulpfuncties. Uiteindelijk kan iedere hulpfunctie makkelijk geïmplementeerd worden doordat deze erg low-level beslissingen neemt.

DNNs zouden dus veel krachtiger moeten zijn dan shallow neural networks. We zullen echter vast kunnen stellen dat het trainen van zo'n netwerk problematisch is. Vervolgens bespreken we 'Convolutional neural networks', een zeer populair type van DNNs. Ten slotte beschrijven we kort het idee achter 'Recurrent neural networks', een ander populair type DNNs.

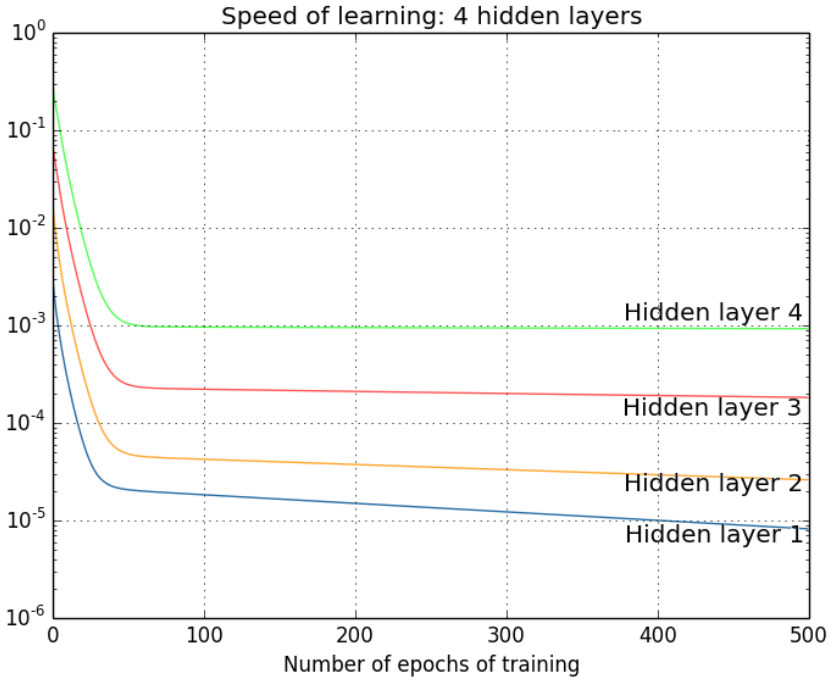
15.2 Vanishing gradient problem

Het trainen van een DNN is niet eenvoudig. Om het probleem te illustreren, gebruiken we de voorbeeldtoepassing over het classificeren van handgeschreven cijfers. Herinner dat we een classificatie nauwkeurigheid van zo'n 96.50% haalden toen we één hidden layer gebruikten. Door een tweede hidden layer toe te voegen, die ook uit 30 neuronen bestaat, haalt het netwerk zo'n 96.95%. Zoals verwacht nam de classificatie nauwkeurigheid toe. Laat ons daarom een derde hidden layer toevoegen. Nu daalt de nauwkeurigheid echter tot zo'n 96.64%. Door een vierde hidden layer toe te voegen, daalt de nauwkeurigheid nog verder tot zo'n 96.38%. Er loopt duidelijk iets mis bij het trainen van zo'n DNN.

15.2.1 Het probleem

Herinner het 'traag leren' uit paragraaf 7.1.1. We zagen dat de groottes van de partiële afgeleiden (tegenover de gewichten en biases) bepalend zijn voor de snelheid waar het netwerk mee leert. Zo zorgen kleine partiële afgeleiden ervoor dat het netwerk traag leert. Voor DNNs blijkt iets gelijkaardigs aan de hand te zijn.

De gradiënt voor een neuron is de vector bestaande uit de partiële afgeleiden van de kost tegenover de bias en alle gewichten van dat neuron. De leersnelheid van een neuron hangt dus af van de gradiënt van dat neuron. Herinner vergelijkingen 3 en 4 van backpropagation (zie paragraaf 6.3), voor het bepalen van de partieel afgeleide van een bias en gewicht respectievelijk. Beide vergelijkingen gebruiken de error, δ, waardoor we de error van een neuron als een



Figuur 65: Verloop van de leersnelheid van de 4 hidden layers tegenover het aantal getrainde epochs.

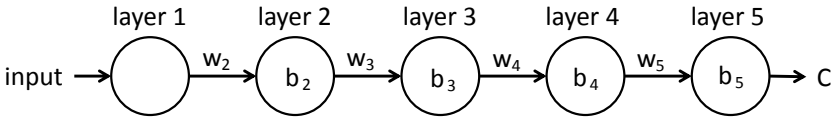
veralgemening van de gradiënt voor dat neuron kunnen zien. Hoe groter die (veralgemeende) gradiënt, hoe sneller het neuron dus leert. Beschouw de vector δ^l, bestaande uit de gradiënt/error voor ieder neuron in laag l. De lengte (of norm) van deze vector, $\|\delta^l\|$, zegt ons dan iets over de gradiënten van de neuronen in laag l.[16] Dit maakt $\|\delta^l\|$ een maat voor de leersnelheid van die laag.

Figuur 65 toont het verloop van de leersnelheid van de 4 hidden layers tegenover het aantal getrainde epochs. We zien duidelijk dat eerdere hidden layers trager leren dan latere hidden layers, wat betekent dat de gradiënten van hun neuronen kleiner zijn. Deze trend doet zich niet alleen voor in ons voorbeeld, maar ook in het algemeen [1]. Hoe meer hidden layers, hoe kleiner de gradiënten dus worden in eerdere lagen. Uiteindelijk zou de gradiënt "verdwijnen" (naar 0 gaan), waardoor men van het 'vanishing gradient problem' spreekt. Doordat de eerste lagen traag leren, krijgen de volgende lagen weinig nuttige informatie over de inputs, waardoor geen goede abstracties gemaakt kunnen worden. Daarom presteert zo'n DNN vaak minder goed dan een shallow neural network.

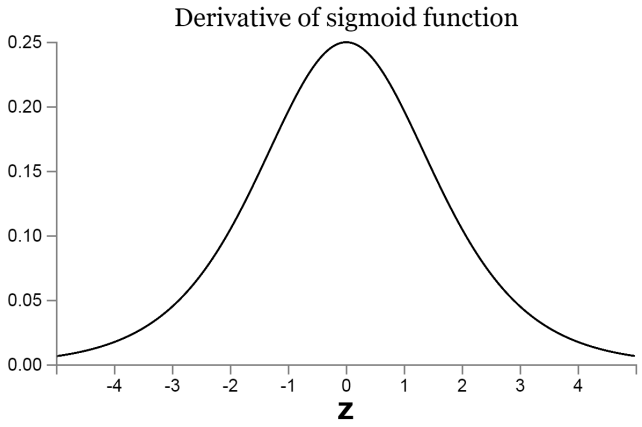
15.2.2 De oorzaak

We zijn natuurlijk geïnteresseerd in de precieze oorzaak van het vanishing gradient problem. Beschouw daartoe het eenvoudige netwerk in figuur 66, bestaande uit 5 lagen van elk één neuron. Laat ons δ^2, de error van het neuron

[16] Herinner $\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$, met $v = (v_1, v_2, \ldots, v_n)$.



Figuur 66: Eenvoudig deep neural network.



Figuur 67: De afgeleide van de logistic sigmoid; $\sigma'(z)$.

in de eerste hidden layer, volledig uitwerken m.b.v. vergelijkingen 1 en 2 van backpropagation (zie paragraaf 6.3):

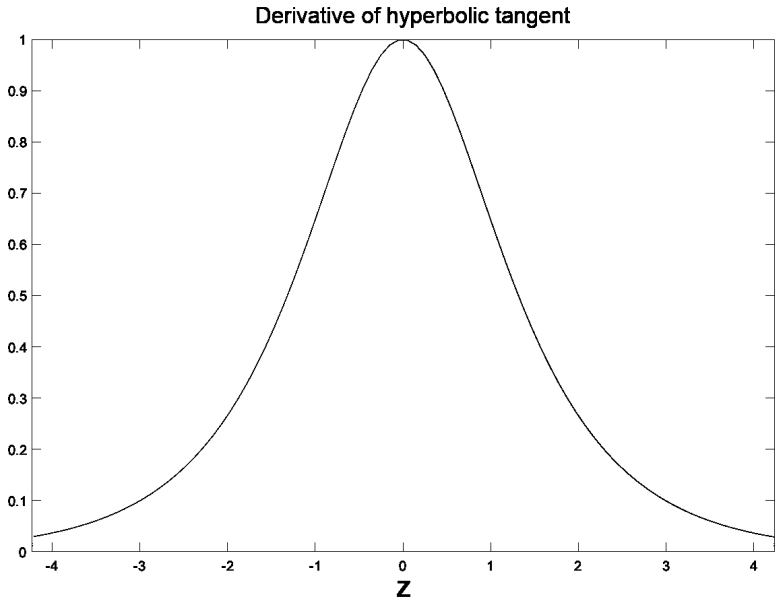
$$\delta^2 = \sigma'(z^2) \cdot w^3 \cdot \sigma'(z^3) \cdot w^4 \cdot \sigma'(z^4) \cdot w^5 \cdot \sigma'(z^5) \cdot \frac{\partial C}{\partial a^5}$$

We negeren de factor $\partial C/\partial a^5$. Herinner dat de gewichten geïnitialiseerd worden volgens de normale distributie met een gemiddelde van 0, en standaarddeviatie van 1 (of minder indien de initialisatie uit hoofdstuk 8 gebruikt wordt). Hierdoor geldt bij de initialisatie van een gewicht w_j meestal dat $|w_j| < 1$. Beschouw de afgeleide van de logistic sigmoid, $\sigma'(z)$, in figuur 67 afgebeeld. Deze bereikt duidelijk een maximum van 0.25 (in $z = 0$), waardoor $\sigma'(z) \leq 0.25$. Bijgevolg geldt (kort) na de initialisatie typisch dat $|w_j \cdot \sigma'(z_j)| < 0.25 < 1$.

Voor iedere extra hidden layer komt er zo'n factor $w_j \cdot \sigma'(z_j)$ bij, waardoor de error alsmaar afneemt in eerdere lagen. Herinner dat die error de veralgemeende gradiënt voorstelt, die de leersnelheid bepaalt. Hierdoor is het duidelijk dat de leersnelheid kort na de initialisatie steeds lager wordt van de laatste hidden layer naar de eerste hidden layer toe.

15.3 Instabiele gradiënten

Zoals meermaals benadrukt, kan het vanishing gradient problem kort na de initialisatie zeker optreden. Het leeralgoritme kan er daarna echter voor zorgen dat de gewichten toenemen, zodat $|w \cdot \sigma'(z)| \geq 1$ en het vanishing gradient problem niet meer optreedt. Vermits $\sigma'(z) \leq 0.25$, moet $|w| \geq 4$ opdat $|w \cdot \sigma'(z)| \geq 1$. Indien echter iedere factor $|w \cdot \sigma'(z)| > 1$ is, nemen de gradiënten in eerdere lagen steeds toe (exponentieel). Hierdoor worden de gradiënten zodanig



Figuur 68: De afgeleide van de tangens hyperbolicus; $\tanh'(z)$.

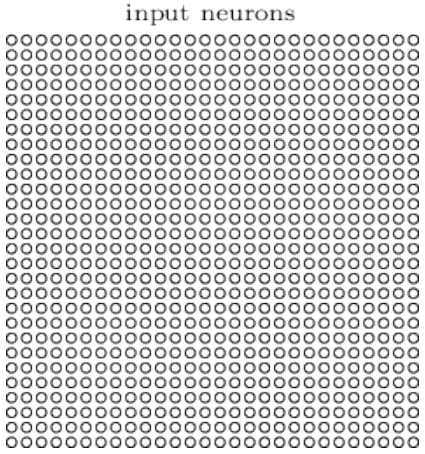
groot dat in de update-regels te grote aanpassingen gemaakt worden en het netwerk niet kan leren. Dit noemt men het 'exploding gradient problem'.

Het vanishing gradient problem doet zich het vaakst voor van beide problemen. Opdat het exploding gradient problem zich kan voordoen, moet het gewicht w groot zijn. Maar aangezien $z = w \cdot a + b$ die w gebruikt, wordt ook z groter. Hierdoor wordt $\sigma'(z)$ kleiner, wat het grotere gewicht dan weer compenseert. Opdat ook $\sigma'(z)$ groot is, moet de activatie a in een relatief klein interval vallen. Net omdat het om een klein interval gaat, doet het exploding gradient problem zich slechts zelden voor.

In de context van het vanishing gradient problem heeft de tanh een heuristiek voordeel (over de logistic sigmoid) als activatiefunctie. Figuur 68 toont de afgeleide van de tanh; $\tanh'(z)$. Deze heeft dezelfde vorm als $\sigma'(z)$, maar met 1.0 als maximum, waardoor $|w \cdot \tanh'(z)| \leq 1.0$. Door met figuur 67 te vergelijken, zien we dat de $\tanh'(z)$ meestal groter is dan de logistic sigmoid. Hierdoor zal de error minder klein zijn, en het vanishing gradient problem minder prominent aanwezig zijn. Bijgevolg zal het netwerk sneller leren.

Het vanishing gradient problem en het exploding gradient problem geven aan dat de gradiënten erg instabiel/gevoelig zijn: als de factoren $w \cdot \sigma'(z)$ een beetje te klein zijn, "verdwijnen" de gradiënten; als ze een beetje te groot zijn, "ontploffen" de gradiënten. Dit maakt het trainen van een DNN erg moeilijk, maar niet onmogelijk zoals uit de volgende paragrafen zal blijken.

We hebben het probleem van de instabiele gradiënten besproken in de context van het eenvoudige DNN in figuur 66. Uiteraard geldt dit ook voor complexere DNNs, bestaande uit meerdere neuronen per laag.



Figuur 69: Conceptuele ordening van de inputneuronen.

15.4 Convolutional neural networks

Herinner het "traditionele" DNN uit paragraaf 15.2, waar gewoon extra hidden layers toegevoegd werden aan een shallow neural network. Herinner ook de voorbeeldtoepassing over het classificeren van handgeschreven cijfers. Het "traditionele" DNN beschouwt alleen de intensiteiten van de 784 pixels. Het spreekt echter voor zich dat hun positie binnen de afbeelding cruciaal is voor een correcte classificatie. Het netwerk moest die spatiale structuur zelf bepalen/afleiden. Inherent aan de architectuur van een 'convolutional neural network' (CNN) is dat daar ondersteuning aan geboden wordt. Hierdoor kan het netwerk makkelijker leren en presteert het beter dan het "traditionele" DNN. Hieronder wordt de architectuur van een CNN besproken.

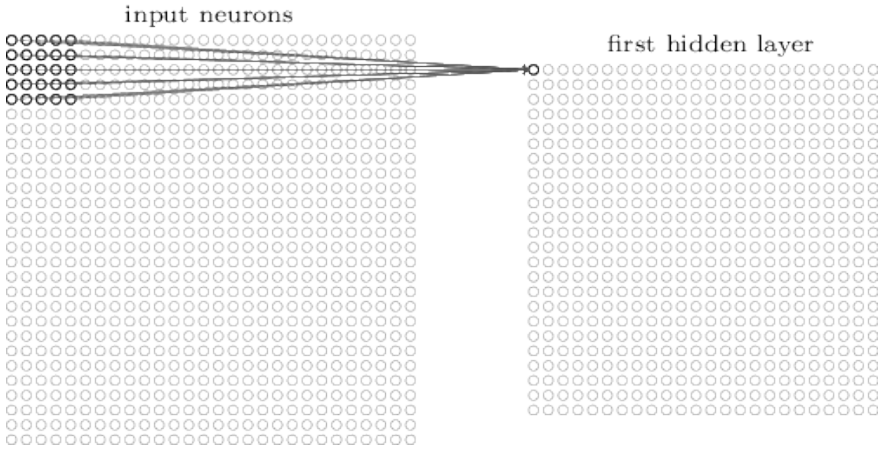
15.4.1 Convolutional layer

Local receptive field

Om duidelijk te maken hoe een CNN rekening houdt met de spatiale structuur van de intensiteiten, ordenen we de 784 inputneuronen conceptueel in een vierkant van 28×28 inputneuronen. Figuur 69 illustreert dit. Een hidden neuron staat nu niet met alle inputneuronen in verbinding, maar slechts met een klein gebied van bv. 5×5 inputneuronen. Zo'n gebied noemt men een 'local receptive field' en wordt door figuur 70 geïllustreerd. Merk op dat het hidden neuron 25 gewichten en 1 bias gebruikt.

Dit veld verschuiven we vervolgens over een afstand van 1 neuron/pixel naar rechts. We verbinden een ander hidden neuron, dat conceptueel rechts van het vorige staat, met dit local receptive field. Dit herhalen we voor de overige local receptive fields, waardoor de eerste hidden layer uiteindelijk uit 24×24 neuronen bestaat.

Meer algemeen wordt het veld telkens over s neuronen/pixels verschoven, wat men de 'stride length' noemt. Deze kan men experimenteel bepalen (op basis van de validatie data). De breedte en hoogte van het local receptive field laat men afhangen van de dimensies van de afbeelding: hoe breder de afbeelding, hoe breder het veld; analoog voor de hoogte.



Figuur 70: Local receptive field.

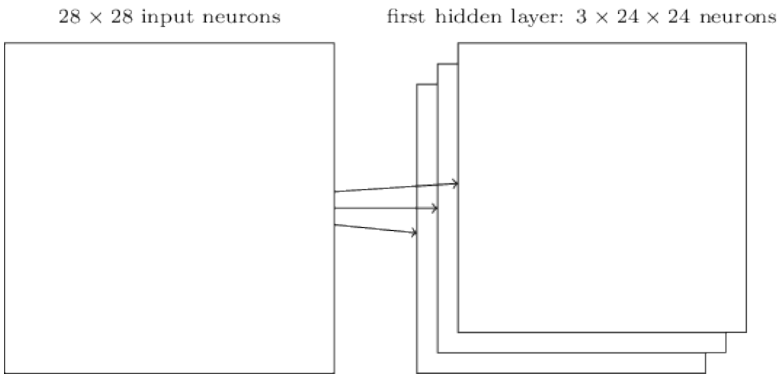
Feature map
Uit voorgaande uitleg zou men kunnen afleiden dat ieder hidden neuron zijn eigen 25 gewichten en ene bias gebruikt. Echter, al deze hidden neuronen gebruiken dezelfde 26 parameters; ze delen hun gewichten en bias. De berekening van de activatie van het hidden neuron op positie j, k wordt dan gegeven door:

$$a_{j,k} = \sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} \cdot i_{j+l,k+m}\right)$$

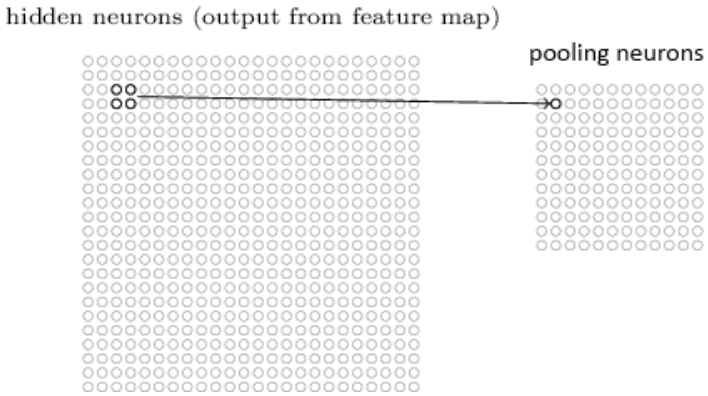
met w de 5×5 gewichtenmatrix, b de bias, en $i_{x,y}$ de input op positie x, y. Bovenstaande operatie wordt een convolutie genoemd in de beeldverwerking. De 26 parameters vormen een 'filter' (of 'kernel'), waarmee een bepaalde feature gedetecteerd kan worden. Zo'n feature is een eenvoudig patroon, zoals een verticale lijn. Een convolutie bestaat er dus in de filter toe te passen op ieder local receptive field.

Iedere activatie in de hidden layer geeft bijgevolg aan in welke mate de feature gedetecteerd werd in het overeenkomstig local receptive field. Daarom noemt men al die activaties samen een 'feature map'. Men zegt ook dat het detecteren van een feature translatie-invariant is: het netwerk zal de feature kunnen detecteren, zelfs als de afbeelding verschoven (getransleerd) werd.

Convolutional layer
Met een feature map kunnen we dus één feature detecteren in de afbeelding. Natuurlijk kunnen we de hidden layer ook uitbreiden naar bv. 3 feature maps, zodat 3 features gedetecteerd kunnen worden. Het spreekt voor zich dat iedere feature map zijn eigen 26 parameters gebruikt. De hidden layer bestaat dan uit $3 \times 24 \times 24$ neuronen, zoals figuur 71 illustreert. Een 'convolutional layer' is een (hidden) layer die uit een aantal feature maps is opgebouwd.



Figuur 71: Input layer (links) en convolutional layer bestaande uit 3 feature maps (rechts).

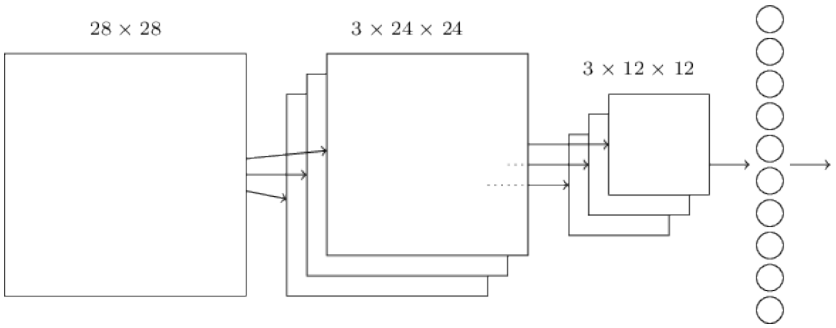


Figuur 72: Pooling neuronen voor één feature map.

15.4.2 Pooling layer

Een pooling layer is een hidden layer die typisch na een convolutional layer geplaatst wordt. Deze laag beschouwt iedere feature map apart en maakt abstracties van de posities van de gedetecteerde feature. De pooling layer geeft dus, per feature, aan of de feature aanwezig is in een grootschaliger gebied. Zo zou de pooling layer aaneensluitende (niet-overlappende) gebieden van bv. 2×2 neuronen uit de feature map kunnen beschouwen. Bijgevolg bestaat de pooling layer uit 12×12 neuronen per feature map, zoals figuur 72 illustreert. Hoe die abstractie precies gebeurt, hangt af van de gebruikte pooling-techniek.

De bekendste techniek is 'max-pooling'. Hier is de output van het neuron in de pooling layer ('pooling neuron') het maximum van de 4 beschouwde activaties uit de feature map. Nog een populaire pooling-techniek is 'L2-pooling', wiens output de wortel is van de som van de kwadraten van de 4 beschouwde activaties. Merk op dat in een pooling-layer geen parameters (gewichten en biases) aanwezig zijn. Laat ons de opeenvolging van een convolutional layer en een pooling layer een convolutional-pooling layer noemen.



Figuur 73: Eenvoudig convolutional neural network.

15.4.3 Opbouw van een CNN

Om tot een volwaardig CNN te komen, moeten we alleen nog een outputlaag (bestaande uit 10 neuronen) toevoegen. Deze laag combineert de info over de aanwezigheid van 3 features en hun posities om te bepalen welk cijfer de input representeert. Zulke cruciale info zou een "traditioneel" deep neural network helemaal zelf moeten afleiden vanaf pixelniveau, waardoor het (intuïtief) minder goed presteert dan een CNN.

Figuur 73 illusteert een eenvoudig CNN. Ieder pooling neuron is verbonden met alle outputneuronen. We kunnen extra hidden layers toevoegen om een dieper CNN te verkrijgen, dat hopelijk nog beter presteert. Ten eerste kan dit een gewone hidden layer zijn, vóór de outputlaag, bestaande uit bv. 100 sigmoid neuronen. Ten tweede kunnen we het CNN uitbreiden door een extra convolutional-pooling layer toe te voegen achter de vorige. In paragraaf 15.6 experimenteren we hiermee, en gaan we in op verdere details.

15.5 Backpropagation voor CNNs

Wanneer men gebruik maakt van een CNN, moet het backpropagation algoritme (zie hoofdstuk 6) aangepast worden. Voor deze uitleg gebruiken we het eenvoudige CNN als voorbeeld, bestaande uit een input layer (van 28×28 neuronen), een convolutional layer (van 24×24 neuronen), een max-pooling layer (van 12×12 neuronen) en een output layer (van 10 neuronen).

Output layer
Voor de output layer wijzigt er niets t.o.v. het backpropagation algoritme zoals voorgesteld in hoofdstuk 6. We gebruiken vergelijking 1 om de error te bepalen in ieder van de outputneuronen, en gebruiken vervolgens vergelijkingen 3 en 4 voor het bepalen van de partieel afgeleiden van de kost tegenover de biases en gewichten.

Errors in de convolutional layer
De berekening van de errors in de convolutional layer volgt vergelijking 2. De errors die de convolutional layer ontvangt van de volgende laag, zijn afkomstig uit de output layer aangezien een pooling layer zelf geen parameters (en dus errors) heeft. Hierbij bepaalt de pooling layer hoe die errors precies in de convolutional layer terecht komen. In ons voorbeeld staat één pooling neuron in

verbinding met 2×2 neuronen uit de convolutional layer. Alleen het neuron met de maximale activatie van de 4 neuronen ontvangt de error uit de outputlaag. Hiervoor blijft dus vergelijking 2 gelden, waarbij laag $l+1$ de outputlaag is. Voor de overige 3 neuronen is het alsof de $\delta^{l+1} = 0$ is, en hun eigen error bijgevolg ook 0 is.

Partieel afgeleide tegenover een bias
Stel dat laag l een convolutional layer is. Voor het berekenen van de partieel afgeleide van de kost tegenover een bias b^l, steunen we op vergelijking 3. We moeten er echter rekening mee houden dat alle neuronen in deze laag dezelfde bias gebruiken. Die ene bias beïnvloedt dus de gewogen som in alle 24×24 neuronen. Bijgevolg geldt:

$$\frac{\partial C_x}{\partial b^l} = \sum_j \sum_k \frac{\partial C_x}{\partial z^l_{j,k}} \frac{\partial z^l_{j,k}}{\partial b^l} = \sum_j \sum_k \delta^l_{j,k}$$

met j, k gaande over alle posities in de convolutional layer (dus $j \in [1; 24]$ en $k \in [1; 24]$) en $\delta^l_{j,k}$ de error voor het convolutional neuron op positie j, k.

Partieel afgeleide tegenover een gewicht
Voor het berekenen van de partieel afgeleide van de kost tegenover een gewicht $w^l_{m,n}$ uit de gewichtenmatrix (filter), geldt een analoge redenering. We steunen hier op vergelijking 4. Op iedere positie j, k waar we de (linkerbovenhoek van) de filter kunnen plaatsen op de inputs/pixels, heeft het gewicht een invloed op de gewogen som. Bijgevolg geldt:

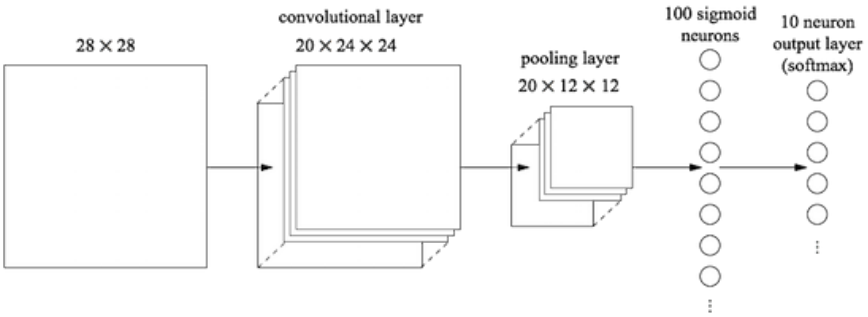
$$\frac{\partial C_x}{\partial w^l_{m,n}} = \sum_j \sum_k \frac{\partial C_x}{\partial z^l_{j,k}} \cdot \frac{\partial z^l_{j,k}}{\partial w^l_{m,n}} = \sum_j \sum_k \delta^l_{j,k} \cdot i_{j+m,k+n}$$

met $i_{x,y}$ de intensiteit van pixel x, y. Deze formule moeten we op ieder van de 5×5 gewichten toepassen om de partieel afgeleide van de kost tegenover ieder van de gewichten te kennen. Dit kan echter efficiënt geïmplementeerd worden, als volgt. We kunnen bovenstaande formule zien als deel van de convolutie-operatie, waarbij de 24×24 matrix met errors als filter beschouwd wordt. Bijgevolg kunnen de partieel afgeleiden van de kost tegenover de gewichten met één convolutie-operatie worden berekend, wat enorm snel kan op een GPU.

15.6 Experimenten met CNNs

Om met convolutional neural networks te experimenteren, gebruiken we Nielsens 'network3.py' [13]. Hiermee kunnen we eenvoudig een CNN opbouwen met de gewenste architectuur. Achterliggend maakt 'network3.py' gebruik van Theano [36], wat een krachtige wiskunde library voor Python is, om een efficiëntere implementatie te verkrijgen.

Vermits de implementatie van 'network3.py' fundamenteel veranderd is (t.o.v. 'network2.py'), voeren we eerst een basistest uit. Hier trainen we een shallow neural network met 100 hidden neuronen gedurende 60 epochs. De learning rate η is 0.1 en de mini-batchgrootte m is 10. We gebruiken een softmax outputlaag. De behaalde classificatie nauwkeurigheid is 97.80%. Herinner dat we met



Figuur 74: Architectuur voor het eerste experiment.

'network2.py' slechts zo'n 96.50% haalden, gebruik makend van regularisatie.

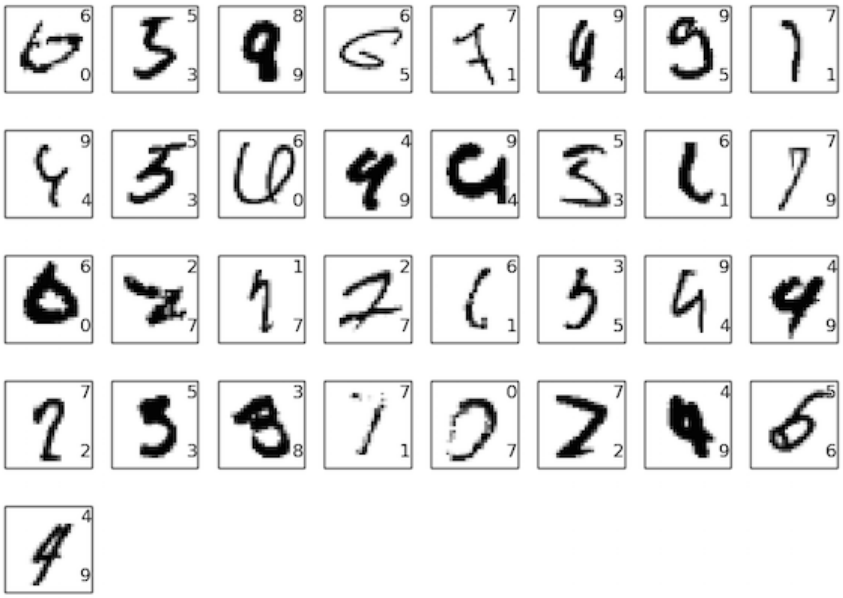
Vanaf nu zullen we met CNNs experimenteren. We gebruiken de architectuur in figuur 74 afgebeeld. Merk op dat er 20 feature maps gebruikt worden, en 1 gewone hidden layer. De behaalde classificatie nauwkeurigheid is 98.78%, wat een relatief grote verbetering is t.o.v. het shallow neural network.

Laat ons een tweede convolutional-pooling layer toevoegen (achter de eerste). Deze gebruikt de activaties van de $20 \times 12 \times 12$ pooling neurons uit de vorige convolutional-pooling layer als input. Een receptive field beschouwt nu alle 20 lagen in hetzelfde gebied (van 5×5), en bestaat dus uit $20 \times 5 \times 5$ neuronen. Voor het aantal feature maps kiezen we 40, waardoor de convolutional layer uit $40 \times 8 \times 8$ neuronen bestaat. De pooling layer beschouwt opnieuw 2×2 gebieden, waardoor deze uit $40 \times 4 \times 4$ neuronen bestaat. De classificatie nauwkeurigheid stijgt naar 99.06%, wat opnieuw significant is. De intuïtie hierachter is de volgende. Een local receptive field combineert de info over de aanwezigheid van 20 eenvoudige features in een groot gebied, om zo de aanwezigheid van een complexere feature te bepalen in dat gebied. Door deze verdere abstractie wordt het voor het netwerk eenvoudiger om te bepalen welk cijfer de input voorstelt.

Door het gebruik van twee hidden layers (i.p.v. 1) van ieder 1000 neuronen, de ReLU als activatiefunctie[17], dropout, en het artificieel uitbreiden van de dataset met translaties over één pixel[18], haalt het CNN een classificatie nauwkeurigheid van maar liefst 99.67%! Dat betekent dat het netwerk slechts 33 van de 10 000 afbeeldingen foutief classificeert. Figuur 75 toont deze. Voor ieder cijfer staat rechtsboven de correcte classificatie (target) en rechtsbeneden de output van het netwerk. Het valt op dat we niet al deze fouten effectief als een fout van het netwerk mogen beschouwen. Zo zijn sommige cijfers foutief

[17] Voor de meerwaarde van de ReLU kunnen we geen motivatie geven, zoals in paragraaf 9.2 werd uitgelegd.

[18] Het lijkt misschien vreemd dat deze techniek hier nuttig is. In paragraaf 15.4.1 werd namelijk gezegd dat het detecteren van een feature translatie-invariant is. Deze translatie heeft geen invloed op de feature maps: aangezien naburige local receptive fields één pixel verschoven zijn t.o.v. elkaar, kan de feature nog steeds gedetecteerd worden. Maar de gebieden van 6×6 pixels die naburige pooling neuronen bestrijken, zijn twee pixels verschoven t.o.v. elkaar. Daardoor is het mogelijk dat het ene pooling neuron een feature wel detecteert, en het naburige pooling neuron nu ook. De feature kwam dan in hun overlappend gebied terecht t.g.v. de translatie. Analoog kan de feature buiten hun overlappend gebied terecht komen, waardoor het ene pooling neuron de feature nog detecteert maar het naburige pooling neuron niet meer.



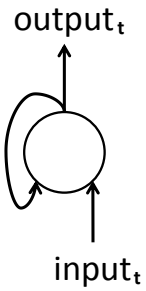
Figuur 75: De 33 cijfers die foutief geclassificeerd werden. Voor ieder cijfer staat rechtsboven de correcte classificatie (target) en rechtsbeneden de output van het netwerk.

gelabeld (bv. dat op rij 1 - kolom 3) en andere zelfs voor een mens onherkenbaar (bv. dat op rij 2 - kolom 3). Daarom mogen we concluderen dat het netwerk (minstens) even goed presteert als een mens.

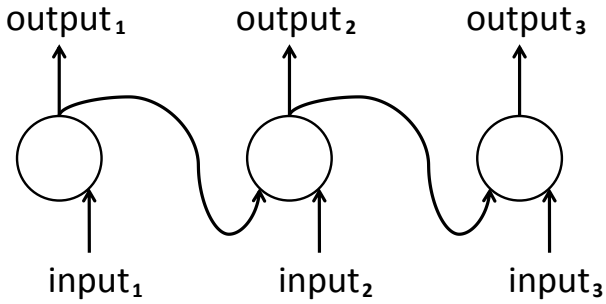
15.7 Recurrent neural networks

De neuronen in de neurale netwerken die we tot nog toe besproken hebben, gebruiken eenvoudigweg hun input om de bijhorende activatie (output) te berekenen en propageren deze naar neuronen in de volgende laag. In dat opzicht bezitten ze geen staat ('stateless'). Een neuron in een 'recurrent neural network' (RNN) daarentegen gebruikt niet alleen de input, maar ook zijn vorige activatie om de activatie te berekenen. In een RNN zijn dus conceptueel lussen aanwezig, zoals figuur 76 illustreert. Zo houdt het neuron wel een staat bij ('stateful'): de vorige activatie kan beschouwd worden als de geschiedenis van dat neuron.

Doordat RNNs rekening houden met hun voorgaande activaties, zijn ze erg geschikt voor inputs waarbij de tijdsdimensie van belang is. Een eenvoudig voorbeeld hiervan is een netwerk dat het volgende woord in een zin moet voorspellen. De individuele reeds geschreven woorden vormen de opeenvolging van inputs. Net zoals een mens houdt het netwerk niet alleen rekening met het laatst beschouwde woord, maar ook met de woorden die voordien beschouwd werden, om zo de correcte context te bepalen en een degelijke voorspelling te kunnen maken. Figuur 77 verduidelijkt dit, door het RNN uit figuur 76 uit te vouwen voor de eerste 3 beschouwde inputs (woorden). In de context van het voorbeeld zou het netwerk het vierde woord voorspellen op basis van de eerste



Figuur 76: Een neuron in een RNN. De lus is kenmerkend.



Figuur 77: Een neuron in een RNN dat 3 maal werd ontvouwd.

3 geschreven woorden.

Het rekening houden met de tijdsdimensie maakt RNNs erg populair in toepassingen rond taal. Zo maakt het systeem voor spraakherkenning in Android [37] gebruik van een LSTM RNN ('Long Short-Term Memory RNN'), wat makkelijker getraind kan worden dan een gewoon RNN. We gaan niet verder in op (LSTM) RNNs.

16 Conclusies en toekomstig werk

In deze bachelorproef heb ik meerdere concepten en technieken bestudeerd voor het trainen van (deep) neural networks. Hun invloed op de classificiatie nauwkeurigheid wordt bepaald voor 'The MNIST database of handwritten digits'. We gebruiken een shallow neural network, bestaande uit een inputlaag van 784 neuronen, een hidden layer van 30 neuronen en een outputlaag van 10 neuronen.

De mean squared error en de cross-entropy kostfunctie blijken even geschikt voor dit experiment. Met beide kostfuncties halen we een classificatie nauwkeurigheid van 95.50%. We concluderen ook dat het initialiseren van de gewichten volgens de normale distributie met gemiddelde 0 en standaarddeviatie $1/\sqrt{n}$ (met n het aantal inputs) er voor zorgt dat het netwerk ongeveer 6 epochs eerder convergeert naar zijn uiteindelijke classificatie nauwkeurigheid, dan wanneer de standaarddeviatie 1 is. Momentum-based gradient descent, een geavanceerd leeralgoritme, zorgt dat het netwerk ongeveer 7 epochs eerder convergeert dan wanneer stochastic gradient descent gebruikt wordt.

Deze bachelorproef bespreekt ook een aantal regularisatie-technieken. L2 en L1 regularisatie hebben dezelfde impact binnen dit experiment: de classificatie nauwkeurigheid stijgt van 95.50% naar 96.50%. Het artificieel uitbreiden van de dataset blijkt een zeer eenvoudige maar zeer doeltreffende regularisatie-techniek: de classificatie nauwkeurigheid stijgt naar 97.50%.

Hoewel een shallow neural network goed presteert voor dit experiment, stellen we vast dat een convolutional neural network (CNN) nog beter doet. Het gebruikte netwerk bestaat uit een inputlaag van 784 neuronen, twee convolutional layers van respectievelijk 20 en 40 feature maps gevolgd door een max-pooling layer, twee hidden layers van 1000 neuronen en een outputlaag van 10 neuronen. Met dit CNN halen we een classificatie nauwkeurigheid van 99.67%.

Deze bachelorproef realiseert twee toepassingen. De eerste laat toe een cijfer te tekenen in een canvas en het door een neuraal netwerk te laten classificeren. De training data komen uit de MNIST database. Door het gebruik van deze toepassing concluderen we dat de positie van het cijfer binnen het canvas en de schrijfstijl twee cruciale factoren zijn voor een correcte classificatie. Dit toont het belang van geschikte training data aan. De tweede toepassing laat toe een muziekbestand te classificeren op basis van het muziekgenre. De training data komen uit de 'GTZAN Genre Collection'. We concluderen dat de classificatie nauwkeurigheid afhangt van het aantal beschouwde genres, en het netwerk vooral liedjes van het genre 'klassiek' correct identificeert. Deze resultaten zijn tevens in overeenstemming met de conclusies uit de literatuurstudie. Om een liedje te classificeren, beschouwt het netwerk 68 features van dat liedje. Er werd aan feature selection gedaan om te bepalen welke features de belangrijkste zijn. Hier kwamen echter geen consistente resultaten uit waardoor we er geen conclusies uit kunnen trekken.

In de toekomst zou men verder onderzoek kunnen doen naar de werking van de vele toegelichte concepten waarvoor men slechts een intuïtieve verklaring heeft tot nog toe. In het bijzonder zou men de tanh als activatiefunctie verder kunnen onderzoeken: ondanks de gekende intuïtieve voordelen van de tanh over de logistic sigmoid, presteren ze beide even goed. Men zou ook verder onderzoek kunnen doen naar de werking van de ReLU als activatiefunctie: hoewel de ReLU ervoor kan zorgen dat een partieel afgeleide 0 kan worden, heeft deze activatiefunctie zijn nut al meermaals bewezen.

17 Referenties

[1] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[2] Michael Negnevitsky. *Artificial Intelligence: A Guide to Intelligent Systems (2nd Edition).* Addison-Wesley, 2005.

[3] Neha Yadav et al. *An Introduction to Neural Network Methods for Differential Equations.* Springer, 2015.

[4] S. N. Sivanandam, S. Sumathi, and S. N. Deepa. *Introduction to Neural Networks Using MATLAB 6.0.* Tata McGraw-Hill, 2005.

[5] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995.

[6] Simon Haykin. *Neural Networks: A Comprehensive Foundation (2nd edition).* Pearson Education (Singapore), 2005.

[7] Edward J. Rzempoluck. *Neural Network Data Analysis Using Simulnet.* Springer-Verlag, 1998.

[8] M. Lichman. UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml` [Geraadpleegd op 18 februari 2016], 2013.

[9] Kevin L. Priddy and Paul E. Keller. *Artificial Neural Networks: An Introduction.* SPIE Press, 2005.

[10] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/` [Geraadpleegd op 23 februari 2016].

[11] Python. `https://www.python.org/` [Geraadpleegd op 10 maart 2016].

[12] NumPy. `http://www.numpy.org/` [Geraadpleegd op 10 maart 2016].

[13] M. Nielsen. Github: neural-networks-and-deep-learning. `https://github.com/mnielsen/neural-networks-and-deep-learning` [Geraadpleegd op 10 maart 2016].

[14] Python Imaging Library (PIL). `http://www.pythonware.com/products/pil/` [Geraadpleegd op 21 maart 2016].

[15] Tkinter. `https://wiki.python.org/moin/TkInter` [Geraadpleegd op 21 maart 2016].

[16] Yves Chauvin and David E. Rumelhart. *Backpropagation: Theory, Architectures, and Applications.* Springer, 2015.

[17] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[18] Nitish Srivastava et al. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[19] Geoffrey E. Hinton et al. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[20] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. *Document Analysis and Recognition*, pages 958–963, 2003.

[21] Yann LeCun et al. Efficient backprop. *Neural Networks: tricks of the trade, Springer*, 1998.

[22] G. Tzanetakis and P. Cook. Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing*, 10(5):293–302, 2002.

[23] Paolo Annesi, Roberto Basili, Raffaele Gitto, Alessandro Moschitti, and Riccardo Petitti. Audio feature engineering for automatic music genre classification. *RIAO '07*, pages 207–211, 2007.

[24] M. Haggblade, Y. Hong, and K. Kao. Student Project for CS229 Machine Learning: Music Genre Classification. *Stanford Computer Science*, 2011.

[25] D. Chathuranga and L. Jayaratne. Automatic music genre classification of audio signals with machine learning approaches. *GSTF Journal on Computing*, 3(2):1–12, 2013.

[26] George Tzanetakis - GTZAN Genre Collection. `http://marsyas.info/downloads/datasets.html` [Geraadpleegd op 15 maart 2016].

[27] Theodoros Giannakopoulos - pyAudioAnalysis. `https://github.com/tyiannak/pyAudioAnalysis` [Geraadpleegd op 15 maart 2016].

[28] T. Giannakopoulos. pyaudioanalysis: An open-source python library for audio signal analysis. *PLoS ONE*, 10(12), 2015.

[29] MATLAB. `http://www.mathworks.com/index.html` [Geraadpleegd op 11 februari 2016].

[30] Neural Network Toolbox. `http://www.mathworks.com/products/neural-network/index.html` [Geraadpleegd op 11 februari 2016].

[31] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234:34–35, 1971.

[32] MATLAB Engine API for Python. `http://nl.mathworks.com/help/matlab/matlab-engine-for-python.html` [Geraadpleegd op 2 april 2016].

[33] pyglet. `https://bitbucket.org/pyglet/pyglet/wiki/Home` [Geraadpleegd op 4 april 2016].

[34] matplotlib. `http://matplotlib.org/` [Geraadpleegd op 2 april 2016].

[35] Sho Sonoda and N. Murata. Neural network with unbounded activations is universal approximator. *CoRR*, 2015.

[36] Theano. `http://deeplearning.net/software/theano/` [Geraadpleegd op 22 mei 2016].

[37] Françoise Beaufays. The neural networks behind Google Voice transcription. http://googleresearch.blogspot.be/2015/08/the-neural-networks-behind-google-voice.html [Geraadpleegd op 19 mei 2016], 2015.