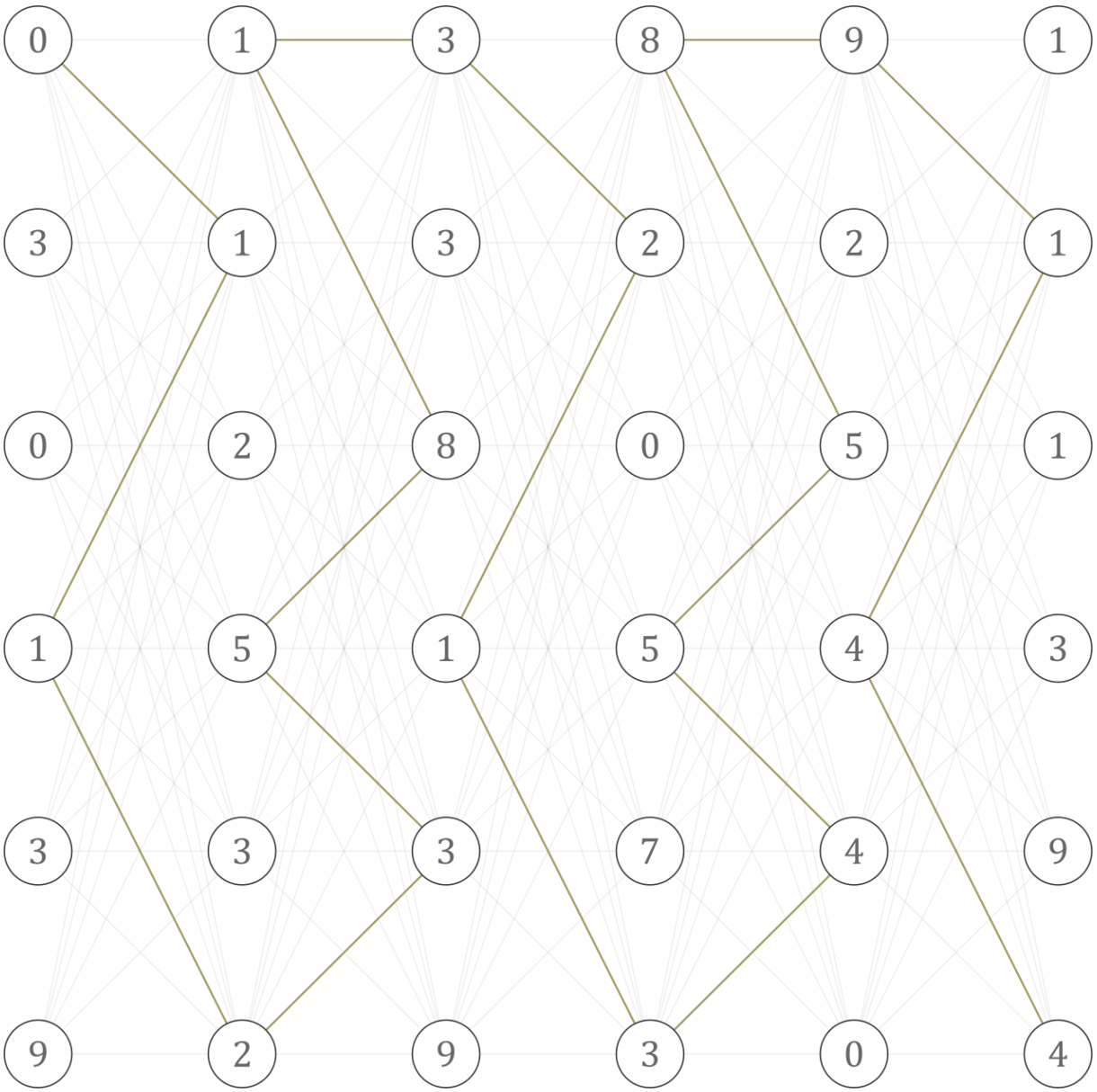

0 1 3 8 9 1
3 1 3 2 2 1
0 2 8 0 5 1
1 5 1 5 4 3
3 3 3 7 4 9
9 2 9 3 0 4

HOE MAAK JE KUNSTMATIGE INTELLIGENTIE?

Een wiskundige uiteenzetting in vervulling van het examenonderdeel: Profielwerkstuk 2021

Door
Damian Liu,

Leerling Stedelijk Gymnasium Arnhem, klas 6C

Vakgebied: Machine Learning

21-12-2021

INLEIDING

Artificial Intelligence, je bent er voor of je bent er tegen. We kunnen er als samenleving in ieder geval niet omheen. Artificial Intelligence zal hoe dan ook de komende 100 jaar van de mens vormgeven. In dit PWS zal ik u echter niet inlichten over de voordelen van AI of juist de gevaren ervan. Ik zal uw gevoelens en opvattingen over AI niet proberen te beïnvloeden. Dit PWS gaat enkel over de wiskundige implementatie van Artificial Intelligence. U zult zien dat er in elk geval geen magie bij betrokken is, hoewel het niet lang zal duren voordat sciencefiction en werkelijkheid vrijwel niet meer te onderscheiden zijn.

Het eindproduct van dit PWS zal een AI zijn die verschillende handgeschreven cijfers kan herkennen. Als je in acht neemt dat AI de beste spelers in het eeuwenoude Chinese spel Go (een spel met meer dan 10^{360} mogelijke zetten!) kan verslaan, is het een vrij simpel voorbeeld. Toch brengt het vrij veel complexiteit met zich mee en vormt het de basis van alle andere typen AI. Als wij mensen een handgeschreven drie zien, herkennen wij die meteen. Toen we kinderen waren, hebben we namelijk jaren lang moeten oefenen door middel van lezen, schrijven en rekenen om verschillende verbindingen tussen zenuwcellen in ons brein te versterken. Het enige wat een computer ziet, is de intensiteit van de pixels op je beeldscherm. De achtergrond van dit Word bestand is bijvoorbeeld wit, dit komt dus overeen met een hogere intensiteit. De letters zijn zwart, een lagere intensiteit. Waar wij mensen jaren over moeten doen, zal een computer met behulp van AI na enkele seconden oefenen al in staat zijn om een drie van een negen te onderscheiden.

De hoofdvraag van dit PWS is:

- Hoe maak je Kunstmatige Intelligentie?

Om deze hoofdvraag te beantwoorden vragen we ons ook het volgende af:

- Wat is Artificial Intelligence (Kunstmatige Intelligentie)?
  - Welke soorten Artificial Intelligence bestaan er?
- Hoe maak je een model dat kan voorspellen of leerlingen voor de middelbare school slagen?
- Hoe maak je een model dat handgeschreven cijfers kan onderscheiden?
  - Hoe maken we dit model nog beter/sneller?
  - Is dit model beter in het herkennen van handgeschreven cijfers dat een mens?

INHOUDSOPGAVE

WAT IS ARTIFICIAL INTELLIGENCE?

> "… It is the science and engineering of making intelligent machines [where] intelligence is the computational part of the ability to achieve goals in the world. …"
>
> John McCarthy in een interview in 2007 [1]

In dit hoofdstuk ga ik de volgende deelvragen beantwoorden:

- Wat is Artificial Intelligence (Kunstmatige Intelligentie)?
  - Welke soorten Artificial Intelligence bestaan er?

De quote van John McCarthy is de breed omvattende definitie van Artificial Intelligence. John McCarthy was een van de pioniers op het gebied van Artificial Intelligence (AI). Dit PWS gaat echter over Neural Networks (NN) wat een onderdeel is van Machine Learning wat op zijn beurt weer een onderdeel is van AI. De echte formuleringen van de deelvragen zijn:

- Wat zijn neural networks?
  - Welke soorten neural networks bestaan er?

Later in dit PWS zal ik zorgvuldig uitleggen wat neural networks nou precies zijn. Voor nu volstaat de volgende definitie: een neural network verwerkt een hele hoop data. Na dit zogenoemde trainingsproces kan een neural network, gebaseerd op de reeds geziene data, voorspellingen maken op data die deze nog niet heeft gezien. Hoewel we dus in feite een neural network gaan maken, is de titel van dit PWS toch "Hoe maak je Kunstmatige Intelligentie". De drie termen Neural Networks, Machine Learning en Artificial Intelligence worden in de volksmond namelijk uitwisselbaar gebruikt. Bovendien brengt de term Artificial Intelligence voor veel mensen verschillende connotaties met zich mee. Neural Networks waarschijnlijk niet.

Er zijn veel verschillende typen Neural Networks. De typen die het meeste voorkomen zijn: Artificial Neural Networks (ANNs), Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) en Generative Adversarial Networks (GANs). In latere hoofdstukken behandelen we het simpelste type Neural Network, de ANN. Hieronder volgt een korte samenvatting van elk soort Neural Network:

1. Artificial Neural Networks (ANNs)

   Aan de basis alle Neural Networks ligt de Perceptron. Deze is in 1958 uitgevonden door Frank Rosenblatt, onderzoeker aan de Cornell Aeronautical Laboratory [2]. De Perceptron werd voor het eerst gebruikt in een fysieke machine met het doel om afbeeldingen te herkennen. Net zoals neuronen in ons brein, kan een perceptron signalen ontvangen. Met deze signalen berekent een perceptron dan een output. Deze output kan de perceptron vervolgens naar andere perceptrons kan sturen. Zo krijgt men een model met veel perceptrons, of neurons, geordend in verschillende lagen die gezamenlijk 1 output berekenen.

2. Deep Neural Networks (DNNs)

   Normale ANNs kunnen alleen simpele problemen oplossen. Om computers complexere problemen op te laten lossen, zoals het begrijpen van menselijke taal en het herkennen van

objecten, gebruiken onderzoekers veel meer lagen dan in de normale ANNs. Iedere laag heeft een bepaald aantal neurons en elk neuron is verboden met alle neurons in de vorige laag en in de volgende laag. (Elke verbinding telt op zijn beurt weer voor een parameter. Het algoritme DALL·E van het onderzoeksbedrijf OpenAI kan bijvoorbeeld hyperrealistische afbeeldingen genereren die gebaseerd zijn op een willekeurige zin geschreven door een mens. Het algoritme van DALL·E gebruikt meer dan 12 miljard parameters om dit te doen.)

3. Convolutional Neural Networks (CNNs)

CNNs worden in een ander deelgebied van AI gebruikt. Namelijk in de computer visie. Dit gebied heeft te maken met het classificeren van afbeeldingen en het detecteren van objecten. CNNs zijn misschien wel de bekendste vordering op normale neural networks. Dit komt omdat men deze vorm van neural networks in het dagelijkse leven het meest gebruikt. Denk bij dit gebruik bijvoorbeeld aan het ontgrendelen van de telefoon door middel van gezichtsherkenning en de autonome voertuigen van Tesla die met camera's alles om hun heen in de gaten houden. Naast volledig verbonden lagen, zoals ze in DNNs voorkomen, bevat een CNN ook convolutional layers (en vaak ook pooling layers) [3]. Deze convolutional layers fungeren als een filter voor de afbeeldingen. CNNs hebben net zoals DNNs vaak veel lagen. Iedere convolutional layer kan steeds complexere patronen waarnemen. Bij het herkennen van dieren bijvoorbeeld, zouden lagen die zich eerder in de structuur van het netwerk bevinden net lijnen en randen kunnen waarnemen. Later in het netwerk nemen de convolutional layers misschien vacht of schubben waar. Nog later kunnen ze koppen van verschillende dieren onderscheiden [4]. Achter de convolutional layers zit een wereld van wiskunde anders dan de wiskunde die ik in dit PWS zal bespreken. Zowel CNNs als de volgende twee typen neural networks, verdienen onder andere door die zeer interessante wiskunde een heel PWS op zich.

4. Recurrent Neural Networks (RNNs)

RNNs onderscheiden zich van andere typen neural networks, door het feit dat ze als het ware een geheugen bezitten. Grafisch gezien zit er een soort lus in het netwerk [5]. Ze houden namelijk rekening met voorspellingen die het netwerk al eerder heeft gemaakt bij het berekenen van de huidige outputs door de voorgaande outputs als nieuwe inputs te beschouwen. RNNs worden daarom veel gebruikt bij problemen waar een bepaalde volgorde of sequentie in zit. Denk hierbij aan spraakherkenning (bijvoorbeeld in Siri en Amazon Alexa) of vertaling (google translate) [6]. Beide zijn domeinen in natural language processing (nlp), een ander deelgebied van AI [7]. RNNs worden ook gebruikt bij het voorspellen van bijvoorbeeld aandelenkoersen, aangezien de prijzen van voorgaande dagen, ofwel de trends, erg belangrijk zijn voor de prijs in de toekomst.

De Long Short-Term Memory (LSTM) architectuur is een verbetering op de normale RNNs. Deze verbetering is teweeggebracht door het feit dat RNNs niet geschikt zijn met problemen, waar een lange termijn geheugen van belang is, LSTMs zijn daar wel voor geschikt. De meeste LSTMs bevatten vier neural networks [8]. De eerste is het normale netwerk, deze produceert een aantal voorspellingen. Het volgende neural network is getraind om onbelangrijke informatie te vergeten. Vervolgens wordt er een kopie gemaakt

van de overige informatie. Deze informatie wordt opgeslagen in het geheugen. Het 3e neural network is getraind om belangrijke informatie in het geheugen te onthouden en de rest op den duur te vergeten. Het laatste neural network krijgt als input zowel de werkelijke voorspellingen, als de informatie die in het geheugen is gegaan. Het geheugen moet dus uit de uiteindelijke voorspellingen gefilterd worden. Hier is het vierde neural network voor getraind.

5. Generative Adversarial Networks (GANs)

GANs zijn in de AI wereld de nieuwste rage. Ze zijn in 2010-2014 ontwikkeld door Ian Goodfellow tijdens zijn PhD onderzoek doctoraat [9]. Het grootste verschil tussen GANs en de andere genoemde typen neural networks, is het feit dat GANs ongestructureerde data gebruiken [10]. Dit noemt men unsupervised learning, de andere typen neural networks vallen onder supervised learning er wordt namelijk gestructureerde data gebruikt. Dat wil zeggen dat er van tevoren al bepaald is wat juiste en onjuiste voorspellingen zijn. GANs proberen data te synthetiseren die niet te onderscheiden zijn van echte data. Ze proberen dus de gegeven data zo nauwkeurig mogelijk te imiteren. Dit werk als volgt, er zijn twee neural networks betrokken. De een heet de generator en de ander de discriminator [10]. De discriminator traint op een dataset bestaande uit echte data en data gegenereerd door de generator. De discriminator wordt vervolgens getraind om de echte data van de gegenereerde data te onderscheiden. De generator probeert data te synthetiseren die de discriminator als echt classificeert. De generator probeert in wezen de discriminator voor de gek te houden. Formeel gezien, concurreren beide neural networks in een Zero-sum game [11], vandaar de term *adversarial* in de naam van het netwerk, wat in het Nederlands *tegenwerkend* betekend.

GANs brengen de laatste tijd ook veel controverse teweeg bij het gebruik in deepfakes. Bij deepfakes wordt er een afbeelding van een (vaak bekend) persoon over het hoofd van een acteur geprojecteerd [12]. Ook de stem kan erg nauwkeurig worden gereproduceerd. Dit proces is zo geavanceerd geworden dat de meeste mensen geen onderscheid meer kunnen maken tussen een echte video en een deepfake.

CNNs, RNNs en vooral GANs komen steeds dichter in de buurt van AI zoals menigeen die voorstelt. Deze voorstellingen zijn ontstaan in de fictie. In de Terminator films zijn er bijvoorbeeld machines gecreëerd die dezelfde capaciteiten bezitten als een mens (later overtreffen deze machines onze nietige capaciteiten). Het meest fascinerende aan GANs is het feit dat ze nu al in staat zijn creatief te zijn. Het is fascinerend omdat creativiteit namelijk als hét definiërende kenmerk van een mens wordt gezien [13].

LOGISTIC REGRESSION

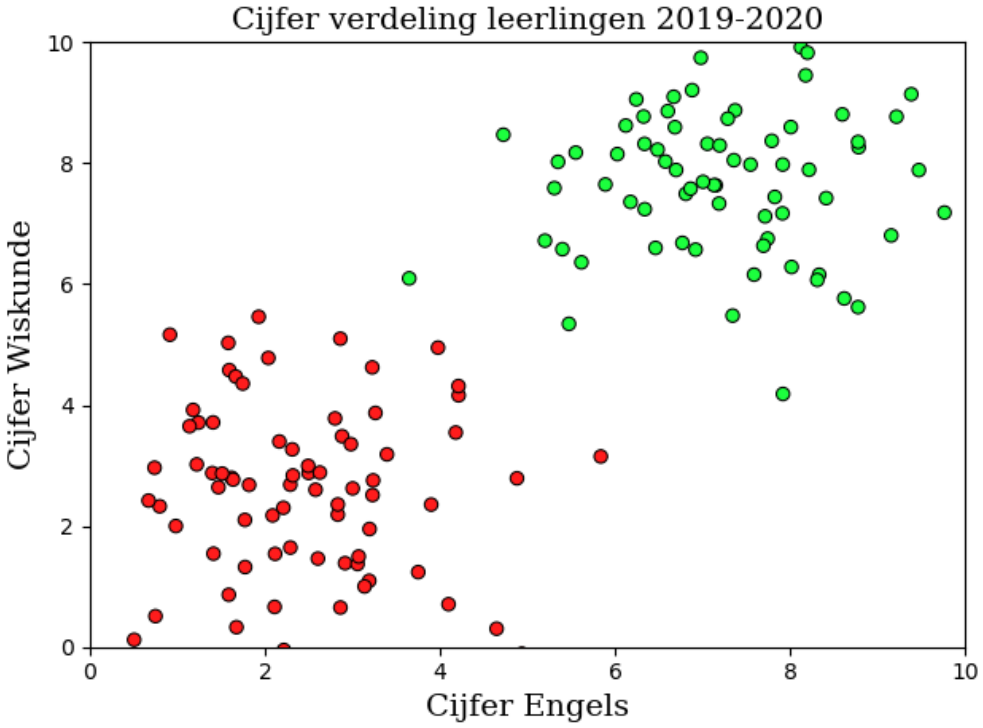
INTRODUCTIE

A. SLAGEN VOOR DE MIDDELBARE SCHOOL

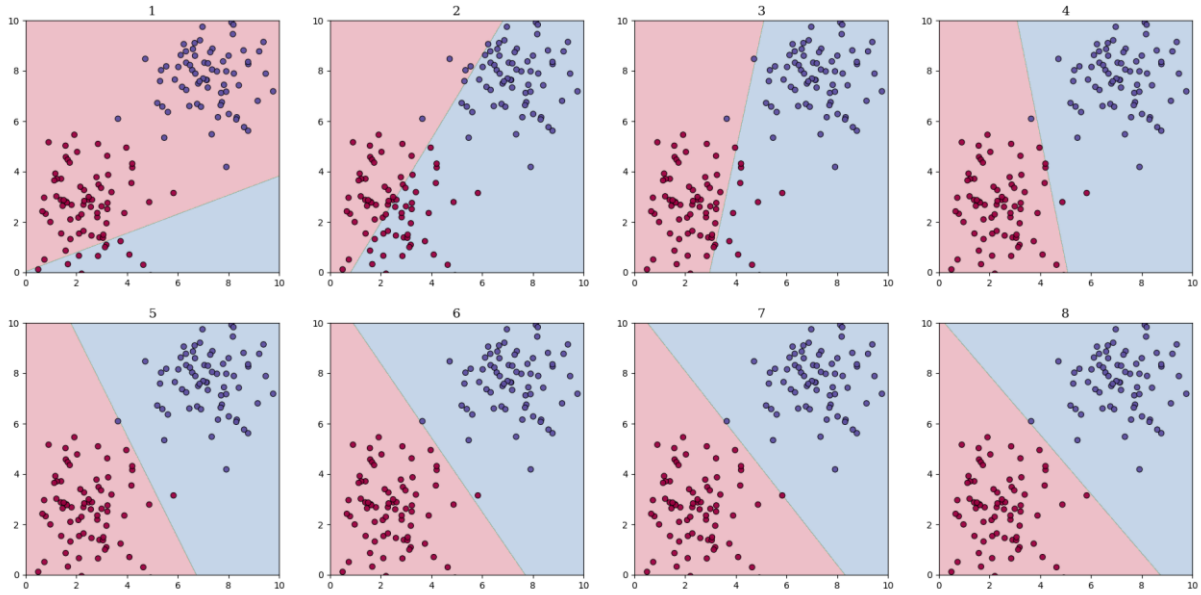
In dit hoofdstuk ga ik de volgende deelvraag beantwoorden.

- Hoe maak je een model dat kan voorspellen of leerlingen voor de middelbare school slagen?

We beschouwen dit hoofdstuk een voorbeeld waarbij de docenten bij ons op school het zat zijn om te bepalen of leerlingen geslaagd zijn voor hun middelbare schooloplleiding of niet. Ze baseren deze keuze op het behaalde eindexamencijfer voor het vak wiskunde en het vak Engels. De docenten willen dat een computer voortaan deze taak zal overnemen. De beschikbare data bestaan uit de examencijfers voor wiskunde en voor Engels die leerlingen in de 6^{e} klas uit het voorgaande jaar hebben behaald. Deze data noemen we de training data. Elk cijfer dat een individuele leerling behaald heeft, noemen we een feature (=kenmerk). Deze features zijn het enige wat een leerling in deze probleemstelling namelijk definieren. Er zijn in dit voorbeeld twee features, namelijk het cijfer voor het vak wiskunde en het cijfer voor het vak Engels. Zou het behaalde cijfer voor het vak Nederlands ook meetellen in het oordeel, dan zijn er in totaal drie features. Verder is per leerling het oordeel van de docenten gegeven. Er zijn twee oordelen mogelijk, de leerling is geslaagd of de leerling is niet geslaagd. We zeggen dat er in totaal twee klassen of labels zijn. Stel dat we het oordeel ‘twijfel geval’ ook willen toevoegen, dan zouden er drie klassen zijn. In totaal hebben er vorig jaar 150 leerlingen de toetsen gemaakt, we zeggen dat er 150 training examples zijn. Van deze data kunnen we een plot maken met op de verticale as het cijfer voor wiskunde en op de horizontale as het cijfer voor Engels. Ook zijn er met de kleuren rood (= niet geslaagd) en groen (=geslaagd) aangegeven of de leerlingen voor de middelbare school waren geslaagd of niet. Ieder bolletje staat voor 1 leerling.



Met een beetje goede wil kan je je een lijn voorstellen die precies de rode bolletjes van de groene bolletjes scheidt. We beginnen met een willekeurige lijn, zie plot 1 op de volgende pagina (in de plots heb ik de groene bolletjes blauw gemaakt voor een mooier kleurcontrast).

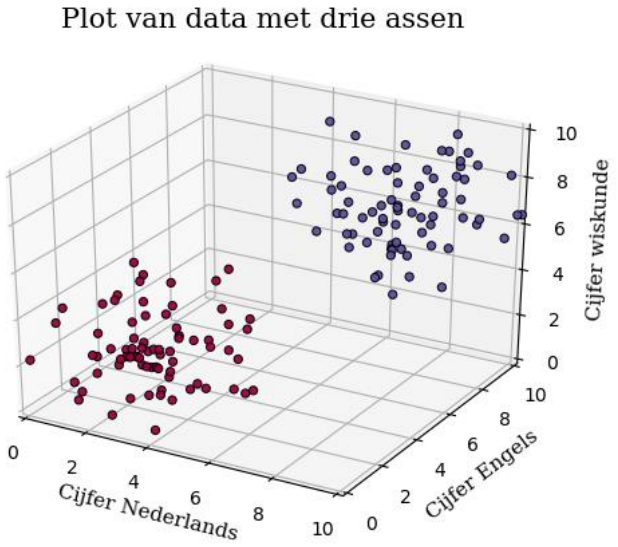


Bij elke combinatie van cijfers in het rode gebied denkt de computer dat de leerling is gezakt. In elk punt in het blauwe gebied denkt de computer momenteel dat je bent geslaagd. Stel dat je dus een 7 voor wiskunde zou hebben en een 9 voor Engels ben je volgens de computer in deze plot gezakt. Dit zou volgens niemand waar kunnen zijn.

Het was dus helaas een verkeerde gok van de computer. Dat maakt voor nu echter niet veel uit. We gaan eerst kijken wat de computer verkeerd heeft gedaan en hoe hij het de volgende keer beter kan doen. Om dit te doen, bepalen we voor ieder bolletje of de computer deze goed of fout heeft geclassificeerd. Als de computer een bolletje juist heeft geclassificeerd, gebeurt er niets. Wanneer de computer een bolletje onjuist heeft geclassificeerd, geven we de computer een strafpunt. Vervolgens tellen we alle strafpunten bij elkaar op. Deze som is de score van de lijn die de computer bedacht heeft. Nu willen we er natuurlijk voor zorgen dat de computer de volgende keer minder strafpunten krijgt.

We stellen ons nu voor dat we ergens bovenaan een berg staan. Om ons heen zien we allerlei paden. Sommige paden leiden naar boven en sommige naar beneden. Sommige paden gaan steiler naar beneden en sommige paden gaan minder steil. We hebben erg veel haast dus we willen in deze situatie zo snel mogelijk afdalen. Het handigste is dan om het steilste pad omlaag te kiezen. In deze analogie komt de hoogte van de berg overeen met het aantal strafpunten. De strafpunten wilden we minimaliseren, we willen de berg dus af gaan. Het pad dat we kiezen heeft effect op hoe de lijn er de volgende keer uit zal zien. We willen die lijn kiezen, die het aantal strafpunten de volgende keer het meest verminderd. We kiezen dus het steilste pad naar beneden. Nadat we dit pad hebben afgelopen komen we weer bij een splitsing van paden en staan we weer voor een keuze. We kiezen weer het steilste pad naar beneden, enzovoort. Elke stap staat voor een verbetering van de lijn in de plots (2 tot en met 7) hierboven. Op een gegeven moment staan we in het dal, we kunnen niet meer verder afdalen. Deze situatie correspondeert met de 8^{e} plot in de afbeelding hierboven. Met deze lijn bevinden alle rode bolletjes zich namelijk in het rode gebied en de blauwe bolletjes in het blauwe gebied.

Als we het cijfer van Nederlands er toch bij willen betrekken, kan je je voorstellen dat we een plot krijgen met drie assen, een voor Wiskunde, een voor Engels en een voor Nederlands. We krijgen dus een driedimensionale plot. In deze driedimensionale ruimte kunnen we vlakken maken die de twee groeperingen van bolletjes kunnen splitsen. Als we meer dan drie schoolvakken willen betrekken, is het voor ons driedimensionaal ingestelde brein onmogelijk om dit voor te stellen. De computer heeft er echter geen moeite mee. Bij de plot op de volgende pagina hebben we nog een idee het vlak er uit zou kunnen zien. In vier dimensies zouden we ons echter geen raad meer weten.



Als we willen dat een computer afbeeldingen van honden en katten kan onderscheiden, kunnen we iedere pixel van een afbeelding van een hond of kat zien als een schoolvak. De klasse hond kunnen we zien als de in het voorbeeld genoemde klasse geslaagd. De klasse kat kunnen we zien als de klasse niet geslaagd. Elk vak is essentieel in de bepaling of een leerling geslaagd is of niet. Net zo is iedere pixel cruciaal in de bepaling of de computer naar een foto van een hond of een kat kijkt.

Het principe in bovenstaand voorbeeld is in essentie de basis van alle neural networks.

B. STAPPENPLAN EN TECHNISCHE INTRODUCTIE

Onder Supervised Learning vallen twee categorieën:

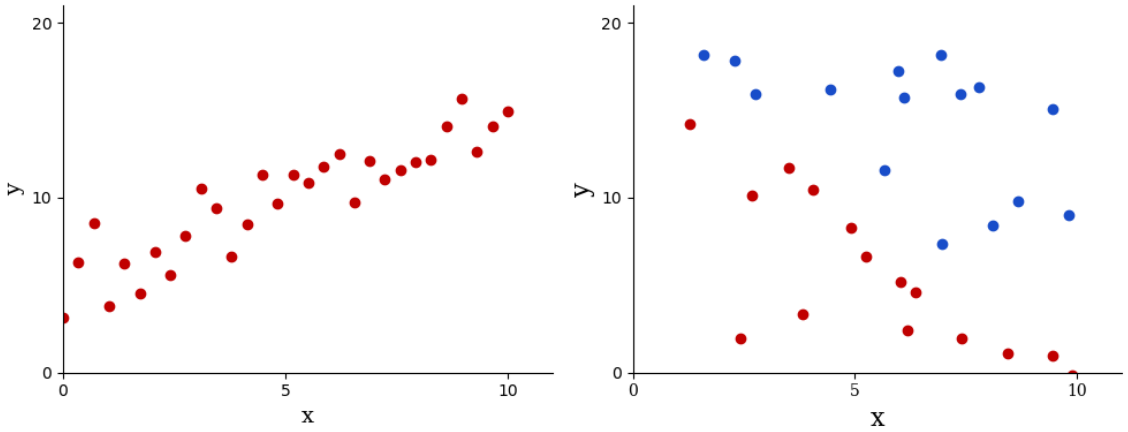
- Classification, het onderscheiden van twee of meer groepen.
- Regression, het voorspellen van toekomstige waarden.

We gaan uiteindelijk een algoritme maken dat verschillende handgeschreven cijfers van elkaar kan onderscheiden. Ik ga het dus vooral hebben over Classification. Het doel van een Classification model is het onderscheiden van twee of meer klassen (geslaagd of niet geslaagd in de vorige paragraaf). In de vorige paragraaf is gebleken dat we wiskundig gezien een aantal dingen nodig hebben.

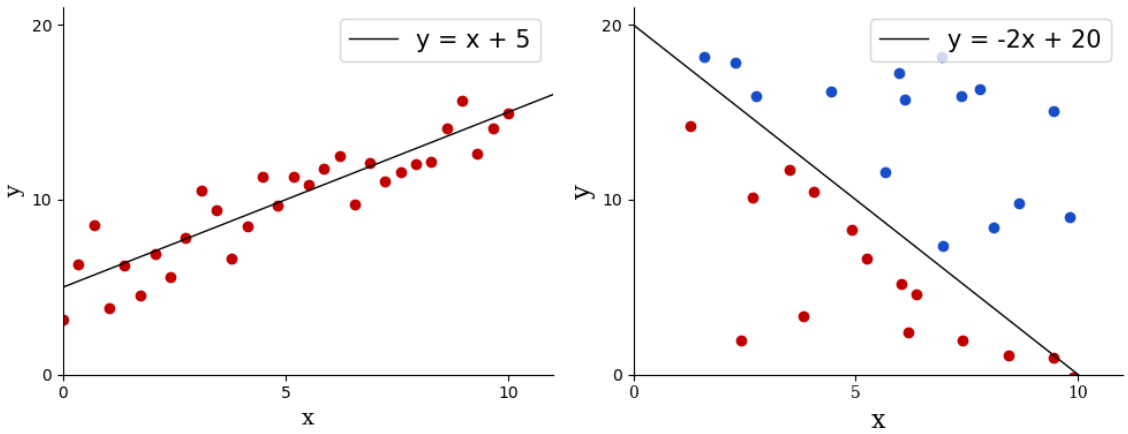
1. Een handige manier om een training dataset (data met cijfers van leerlingen) te organiseren.

2. Een functie die op basis van de training kan voorspellen of een training example bij de ene klasse of de andere klasse hoort (voorspelling van de computer).
3. Een functie die aangeeft hoe ver de voorspelling van functie 2. van het werkelijke antwoord was (het optellen van strafpunten).
4. Een proces dat er voor zorgt dat de waarde van functie 3. Zo klein mogelijk (geminimaliseerd) wordt (het afdalen van de berg).

Om het verschil tussen Classification en Regression duidelijker te maken, gaan we het kort hebben over Linear Regression. De methodes namelijk lijken enigszins op elkaar.



Stel dat we een dataset hebben met datapunten zoals die links zijn aangegeven in bovenstaande afbeelding. Met een beetje fantasie kunnen we ons voorstellen dat deze dataset de prijs van avocado's plot tegen de grootte van de avocado's. We zien dat hoe groter een avocado is, des te groter is de gemiddelde prijs. Stel vervolgens dat we een nieuwe avocado geteeld hebben en dat we graag de prijs willen weten waarvoor we deze avocado kunnen verkopen. Het doel van Linear regression is dan een functie vinden die als input een x (grootte avocado) waarde heeft en als output probeert te voorspellen wat de corresponderende y (prijs avocado) waarde zou moeten zijn. Het model zal deze functie afleiden door middel van de datapunten die er al zijn, de training data. We willen dus eigenlijk een lijn vinden, die precies tussen alle datapunten past. Een functie in de vorm van $h_{a,b}(x) = ax + b$ zou goed door de datapunten passen. Linear regression probeert de juiste waarden van de parameters a en b te vinden, zodat de functie het beste door de datapunten past. De linker grafiek in de afbeelding op de volgende pagina zou een goede benadering zijn voor de functie $h_{a,b}(x)$. De gevonden parameter waarden voor a en b zijn dan $a = 1$ en $b = 5$. Een avocado met grootte 10 zou dus: 10 + 5 = 15 euro kosten.



Classification hebben we al in het voorbeeld besproken. Hier willen we dat de voorspellingsfunctie (nummer 2. in stappenplan) een waarde tussen 0 en 1 aanneemt om aan te geven of een datapunt bij de groep geslaagd (=1) of bij de groep niet geslaagd (=0) hoort. In de afbeelding rechtsboven zijn deze twee groepen de rode- en blauwe puntjes. De lijn die de computer moest leren, noemen we de disicion boundary. Ook deze lijn kan bepaald worden door de juiste waarden van de parameters te achterhalen.

Kort samengevat gaan we het volgende doen. Alle parameters zullen eerst willekeurig gekozen worden (vaak wel met een kleine waarde tussen de -1 en de 1) dit komt overeen met de willkeurig gekozen lijn van de computer. Het model zal, met de willekeurig gekozen parameters vervolgens ieder training example langsgaan om te voorspellen wat zijn klasse is. Deze output is natuurlijk een gok omdat de parameters nog willekeurig en niet getraind zijn. Een andere functie zal dan voor elk training example bepalen hoe ver de gok van de voorspellingsfunctie van de werkelijke klasse zat. Als laatste moeten de willekeurig gekozen parameters dan zo veranderd worden, opdat de output van deze functie bij de volgende iteratie kleiner wordt. Deze stap wordt gradiënt descent genoemt. De iteraties worden vaak duizenden keren herhaald tot de juiste parameters bepaald zijn.

DATASETS, MATRICES EN VECTOREN

Om alle training examples met features op een georganiseerde manier weer te geven, gebruiken we wiskundige objecten genaamd matrices en vectoren. Een matrix en een vector zien er als volgt uit:

$$\begin{matrix} & \leftarrow kolommen \rightarrow & & & & \\ \uparrow & & & & & \\ rijen & & & & & \\ \downarrow & & & & & \end{matrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2m} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nm} \end{bmatrix}$$

Afbeelding van een Matrix

$$\begin{matrix} \uparrow \\ rijen \\ \downarrow \end{matrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}$$

Afbeelding van een Vector

De dimensie van de matrix (dus het aantal rijen en kolommen) kunnen we noteren als: $(\#aantal\ rijen,\ \#aantal\ kolommen)$ of $\mathbb{R}^{\#aantal\ rijen \times \#aantal\ kolommen}$. Matrices worden vaak met een hoofdletter aangegeven en vectoren met een kleine letter.

In de wiskunde worden zowel matrices als vectoren gebruikt in de lineaire algebra. In de natuurkunde kennen matrices ook vele toepassingen. Een gebied in de natuurkunde dat mij erg aanspreekt, is quantum mechanics. Hieronder valt de Matrixmechanica.

In de bloei van quantum mechanics stelde Werner Heisenberg een formulering op, die deeltjes op het kleinste niveau beschreef. Het was voor natuurkundigen in die tijd echter erg lastig om berekeningen te doen met deze matrixmechanica [14]. Toen Erwin Schrödinger, een andere natuurkundige in het begin van de 20e eeuw, dus zijn nu iets populairdere Schrödingervergelijking bedacht had, stapten sommigen natuurkundigen over op de manier van Schrödinger [14]. Deze was in die tijd namelijk makkelijker te berekenen. Waarom de manier van Heisenberg zo lastig was, leg ik in de volgende alinea uit. Zowel de methode van Heisenberg als de methode van Schrödinger zijn wiskundig gelijk aan elkaar. Tegenwoordig kan een computer deze matrix vermenigvuldigingen in een fractie van een seconde doen.

Op het gebied van machine learning gebruiken we matrices en vectoren om alles netjes en georganiseerd te houden. Verder gebruiken we ze om hun handige, en tegenwoordig snelle, manier van vermenigvuldigen. Bij machine learning zijn drie operaties van belang: een vermenigvuldig van een matrix met een vector, de vermenigvuldiging van een matrix met een matrix en het optellen van een vector bij een matrix. Deze operaties zijn hieronder afgebeeld.

- Optellen, $matrix\ +\ vector$

$$\begin{bmatrix} \alpha & \beta & \gamma \\ \delta & \varepsilon & \zeta \\ \eta & \theta & \iota \end{bmatrix} + \begin{bmatrix} \kappa \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} \alpha + \kappa & \beta + \kappa & \gamma + \kappa \\ \delta + \lambda & \varepsilon + \lambda & \zeta + \lambda \\ \eta + \mu & \theta + \mu & \iota + \mu \end{bmatrix}$$

- Vermenigvuldigen, $(horizontale)\ vector\ \times\ vector$

$$[\alpha \quad \beta \quad \gamma] \times \begin{bmatrix} \delta \\ \varepsilon \\ \zeta \end{bmatrix} = (\alpha\delta + \beta\varepsilon + \gamma\zeta)$$

❖ Vermenigvuldigen, $matrix \times vector$

$$\begin{bmatrix} \alpha & \beta & \gamma \\ \delta & \varepsilon & \zeta \\ \eta & \theta & \iota \end{bmatrix} \times \begin{bmatrix} \kappa \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} (\alpha\kappa + \beta\lambda + \gamma\mu) \\ (\delta\kappa + \varepsilon\lambda + \zeta\mu) \\ (\eta\kappa + \theta\lambda + \iota\mu) \end{bmatrix}$$

❖ Vermenigvuldigen, $matrix \times matrix$

$$\begin{bmatrix} \alpha & \beta & \gamma \\ \delta & \varepsilon & \zeta \\ \eta & \theta & \iota \end{bmatrix} \times \begin{bmatrix} \kappa & \lambda & \mu \\ \nu & \xi & o \\ \pi & \rho & \sigma \end{bmatrix} = \begin{bmatrix} (\alpha\kappa + \beta\nu + \gamma\pi) & (\alpha\lambda + \beta\xi + \gamma\rho) & (\alpha\mu + \beta o + \gamma\sigma) \\ (\delta\kappa + \varepsilon\nu + \zeta\pi) & (\delta\lambda + \varepsilon\xi + \zeta\rho) & (\delta\mu + \varepsilon o + \zeta\sigma) \\ (\eta\kappa + \theta\nu + \iota\pi) & (\eta\lambda + \theta\xi + \iota\rho) & (\eta\mu + \theta o + \iota\sigma) \end{bmatrix}$$

Nu zie je wel waarom natuurkundigen liever de manier van Schrödinger gebruikten! Je ziet ook dat bij het vermenigvuldigen van matrices de dimensies (het aantal kolommen en rijen) erg van belang zijn. Immers, de dimensie van de eerste matrix, moet precies aansluiten op de dimensies van de tweede matrix. Het heeft mij erg lang geduurd om deze vermenigvuldigingen goed te begrijpen en überhaupt te snappen waarom dit zo werkt. Daarom heb ik de volgende handige vuistregel bedacht (alle zelfde letters hebben dezelfde waarde). Een vermenigvuldiging is mogelijk als:

$$(A, B) \times (B, C) = (A, C)$$

Waarin B het zelfde getal moet zijn. De resulterende matrix heeft de dimensies (A, C). Visueel gezien betekent deze regel dat de kolommen van de eerste matrix en de rijen van de tweede matrix gelijk moet zijn.

Als voorbeeld vermenigvuldigen we een matrix X met de dimensies (3, 2) en een matrix Y met de dimensies (2, 5). Als uitkomst krijgen we een matrix met de dimensies (A, C) = (3, 5), (zie vuistregel). Om te kijken of deze vermenigvuldiging daadwerkelijk mogelijk is, moet het aantal kolommen van matrix X en het aantal rijen van matrix Y gelijk zijn (opnieuw, zie de vuistregel). In dit geval is dat zo, dus deze vermenigvuldiging is mogelijk.

$$X \times Y = \begin{bmatrix} 4 & 7 \\ 2 & 2 \\ 6 & 5 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 & 8 & 9 \\ 5 & 7 & 9 & 7 & 2 \end{bmatrix} = \begin{bmatrix} 35 & 49 & 67 & 81 & 50 \\ 10 & 14 & 20 & 30 & 22 \\ 25 & 35 & 51 & 83 & 64 \end{bmatrix}$$

Zo is bijvoorbeeld $(60000, 5000) \times (300, 60000)$ niet mogelijk maar;

$(300{,}60000) \times (60000, 5000)$ wel. De verkregen matrix heeft dan de dimensies $(300, 5000)$.

In de volgende hoofdstukken is het handig om matrix vermenigvuldigingen voor jezelf op een kladblaadje te checken aan de hand van mijn vuistregel. Dit heeft mij tenminste erg geholpen.

Een laatste toevoeging aan matrices is dat er een aantal transformaties zijn voor matrices. Zo heb je de transpose van X aangegeven met X^T. Wanneer deze transformatie op een matrix wordt uitgevoerd, worden de rijen en kolommen omgewisseld.

$$\begin{bmatrix} 3 & 7 \\ 9 & 1 \\ 3 & 5 \end{bmatrix}^T = \begin{bmatrix} 3 & 9 & 3 \\ 7 & 1 & 5 \end{bmatrix}$$

Zoals ik al eerder gezegd heb, zullen we matrices en vectoren gebruiken om alle data en parameters in op te slaan. In het volgende hoofdstuk zullen we ze in onderstaande vormen

gebruiken. Bij neural networks gebruiken we een iets andere vorm voor de parameters. Dit behandelen we als we zover zijn.

De training data stoppen we in een matrix. Deze wordt vaak met een hoofdletter X aangegeven. Als we het "slagen voor de middelbare school" voorbeeld erbij betrekken, staat iedere kolom in deze matrix voor een leerling. De eerste rij staat voor de behaalde cijfers voor het vak wiskunde en de tweede rij staat voor het cijfer voor Engels. De waarden van m en n zijn dus respectievelijk 150 (leerlingen) en 2 (vakken).

$$X = \begin{matrix} \overset{\leftarrow m \rightarrow}{} \\ \begin{bmatrix} | & | & | & & | \\ X^{(1)} & X^{(2)} & X^{(3)} & \dots & X^{(m)} \\ | & | & | & & | \end{bmatrix} \end{matrix} \begin{matrix} \uparrow \\ n \\ \downarrow \end{matrix}$$

$$met\ m = aantal\ training\ examples;$$

$$n = aantal\ features\ per\ training\ example;$$

$$x_n^{(m)} \epsilon\ \mathbb{R}$$

Voorbeeld:

$$\begin{matrix} & Leerling\ 1 & Leerling\ 2 & Leerling\ 3 & & Leerling\ 150 \\ cijfer\ wiskunde & \begin{bmatrix} 6.6 & 8.5 & 9.0 & \dots & 10 \\ 3.7 & 7.1 & 5.3 & & 9.3 \end{bmatrix} \\ cijfer\ Engels \end{matrix}$$

We onderscheiden klassen aan de hand van een cijfer in de verzameling natuurlijke getallen (0,1,2,3,4,5,…). De waarde van m is weer 150. Stel dat de leerling op nummer 1 niet is geslaagd, in de vector y staat op deze plek dan een 0. Leerling 2 is wel geslaagd op de 2e plek van de vector staat dus een 1. y is een horizontale vector met de werkelijke klasse (true labels) van alle training examples.

$$y = \begin{matrix} \overset{\leftarrow m \rightarrow}{} \\ [y^{(1)} \quad y^{(1)} \quad y^{(1)} \quad \dots \quad y^{(m)}] \end{matrix}$$

$$met\ m = aantal\ training\ examples;$$

$$y^{(m)} \epsilon\ \mathbb{N}$$

Voorbeeld:

$$\begin{matrix} & Leerling\ 1 & Leerling\ 2 & Leerling\ 3 & & Leerling\ 150 \\ Geslaagd? & [0 & 1 & 1 & \cdots & 1] \end{matrix}$$

De parameters plaatsen we in een verticale vector w. In het volgende hoofdstuk gebruiken we net zoveel parameters als het aantal features en dus het aantal rijen van de data matrix X. De rijen in de parameter vector w zijn dus gelijk aan n. In de volgende paragraaf leg ik hoe we deze vector met parameters precies gaan gebruiken.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix} \begin{matrix} \uparrow \\ n \\ \downarrow \end{matrix}$$

$$met\ n\ =\ aantal\ features\ per\ training\ example$$

$$w_n \epsilon\ \mathbb{R}$$

Als laatst voegen we nog een bias term toe. De betekenis hiervan leg ik in een later hoofdstuk uit. De bias parameter is geen vector of een matrix, het is een gewoon reëel getal. In de volgende paragraaf ga ik uitleggen waarvoor we deze parameter gaan gebruiken.

$$b\ \epsilon\ \mathbb{R}$$

LOGISTIC REGRESSION

A. DE VOORSPELLINGSFUNCTIE

In deze paragraaf gaan we zien hoe de computer precies zijn voorspellingen maakt. In het voorbeeld waren er twee features. In het voorbeeld “slagen voor de middelbare school” heeft de computer voor zijn voorspelling elke feature met een parameter vermenigvuldigd. Samen met een constante term b, heeft hij alles bij elkaar opgeteld (het product van de twee features met de parameters en de constante term b). Als er n features zijn, is de algemene vorm van de voorspellingsfunctie op 1 training example (i) tot nu toe:

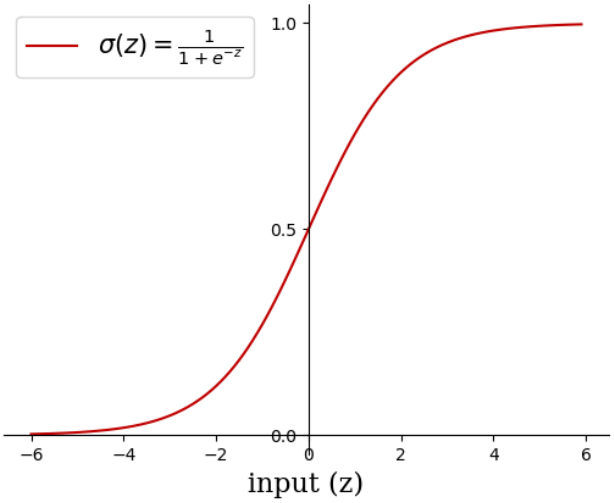
$$z^{(i)} = w_1 {x_1}^{(i)} + w_2 {x_2}^{(i)} + w_3 {x_3}^{(i)} + \cdots + w_n {x_n}^{(i)} + b$$

Dit is het lineaire deel van de voorspellingsfunctie, we noemen deze z. w_n is hierin de parameter die met feature x_n wordt vermenigvuldigd. b is de constante parameter. De parameters w noemen we ook wel weights, het is namelijk net alsof we een gewichtje aan elke feature hangen. De b term wordt de bias genoemd. Deze functie kan dus als een gewogen som worden geïnterpreteerd. Net zoals de weging van examencijfers betekent dit namelijk dat sommige features zwaarder meetellen in de voorspelling dan andere features. Als w een vector is met dimensies $(n, 1)$ en X een matrix is met dimensies (n, m), waarin n het aantal features is en m het aantal training examples, kunnen we z ook noteren als:

$$z = w^T X + b$$

Ga voor jezelf na dat dit resulteert in een vector met de dimensies (1, m), waarbij elk kolom in de vector z staat voor de lineaire output van de voorspellingsfunctie voor elke leerling. Daarom zijn matrix vermenigvuldigingen zo handig! Waar we eerst iedere training example apart met de parameter vector w zouden moeten vermenigvuldigen, hebben we dat nu voor alle training examples, in één regel gedaan.

De output van de functie z kan nog alle waarden aannemen. We willen echter dat z alleen een waarde tussen de 0 en de 1 kan aannemen. De voorspellingsfunctie moet namelijk kunnen aangeven of een training example bij de klasse y=0 of de klasse y=1 hoort. Een functie die een bepaalde input kan veranderen in een waarde tussen de 0 en de 1 is de sigmoid functie. De formule van deze functie en zijn bijbehorende grafiek zien er als volgt uit.



$\sigma(z)$ kan zoals je ziet een waarden met een willekeurig bereik omtoveren naar een waarde met een bereik tussen de 0 en de 1. Dit is dus is dus een geschikte functie voor ons probleem. Je ziet ook dat de sigmoid functie een verloop heeft. Het heeft als output dus niet 1 of 0. Als $\sigma(z)$ als output bijvoorbeeld de waarde 0.78 heeft, kunnen we zeggen dat de functie 78% zeker is dat de bijbehorende training example bij de klasse y=1 hoort. Dit is de volledige vorm van de voorspellingsfunctie. Deze geven we aan met het symbool $\hat{y}$, niet te verwarren met de normale y. $\hat{y}$ is wat de computer denkt dat de klasse van een bepaalde leerling is, y is de werkelijke klasse bepaald door de docenten.

$$\hat{y} = g(W^T X + b) \; waarin \; g = \sigma(z) = \frac{1}{1 + e^{-z}}$$

In bovenstaande formule noemen we g ook wel een activation function. De sigmoid functie is een van de mogelijke activation functions die we kunnen gebruiken. Er zijn er nog een aantal andere activation functions. Hier zal ik echter later pas verder op ingaan. Voor nu hebben we genoeg aan de sigmoid functie. Zoals ik in het voorbeeld in het begin van dit PWS al heb gezegd, kiezen we in het begin een willekeurige lijn. Dit komt overeen met het kiezen van willekeurige parameters. Wanneer we in het dal van de denkbeeldige berg zijn, zijn de juiste waarden voor de parameters bepaald. We zeggen ook wel dat we klaar zijn met trainen van het model. De parameters van een model dat ik met behulp van logistic regression getraind heb zijn:

$$w = \begin{bmatrix} 1.47 \\ 2.26 \end{bmatrix} \; en \; b = -18.3$$

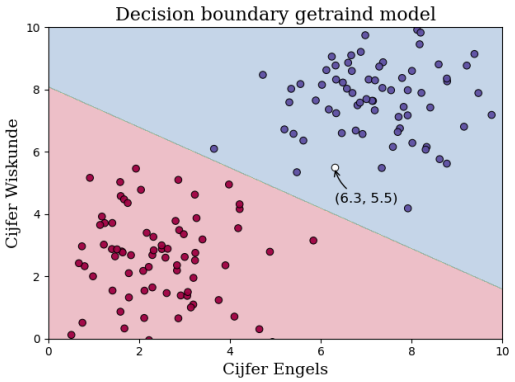
Je kan nu zelf uitrekenen wat de voorspelling van de computer zou zijn bij twee zelfbedachte features. We nemen als voorbeeld een leerling die voor zijn toets Engels een 6.3 en voor zijn toets Wiskunde een 5.5 behaald heeft. De voorspelling van de computer is dan:

$$z = [1.49 \quad 2.26] \begin{bmatrix} 6.3 \\ 5.5 \end{bmatrix} + (-18.2) = 3.41$$

$$\hat{y} = \sigma(z)$$

$$= \frac{1}{1 + e^{-3.41}} = 0.97$$

Als we aannemen dat een leerling is geslaagd als de voorspelling van de computer meer dan 50% is, kunnen we ervan uitgaan dat deze leerling is geslaagd, aangezien 97% > 50%. We kunnen ook zeggen dat de computer er 97% zeker van is dat de leerling is geslaagd. Dit kunnen we ook in onderstaande afbeelding zien. Het punt met de coördinaten (6.3; 5.5) valt immers in het blauwe gebied.

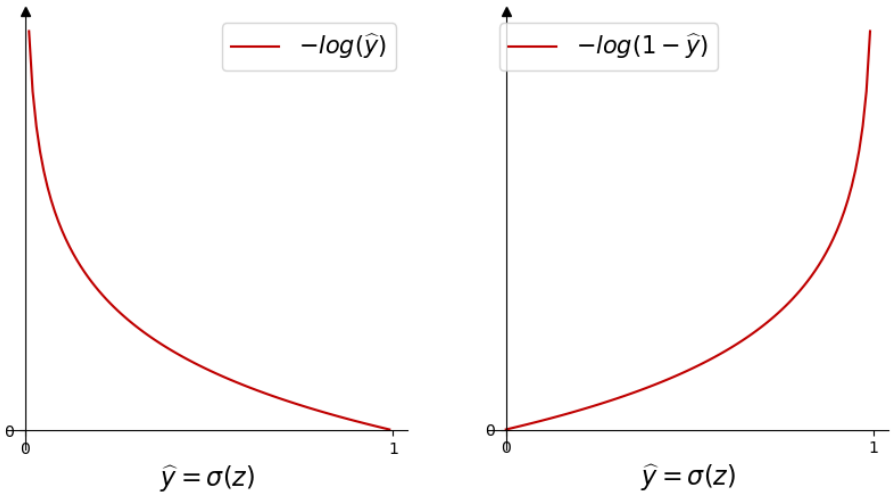


B. LOSS FUNCTION / COST FUNCTION (TOEWIJZEN VAN STRAFPUNTEN)

Volgens het stappenplan hebben we nu nog een maat nodig voor de nauwkeurigheid van de voorspellingsfunctie. We hebben dus een functie nodig die aangeeft hoe goed de parameters van de voorspellingsfunctie zijn. Eigenlijk zeggen we dat deze functie een getal als output moet hebben om aan te geven hoe slecht de parameters van de voorspellingsfunctie zijn. Als de output groot is, is de voorspelling dus slecht. Het is lastig om zo'n soort functie zelf af te leiden, daarom staat hij hieronder gegeven.

$$\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) = -y^{(i)}\, log\left(\hat{y}^{(i)}\right) - \left(1 - y^{(i)}\right) log\left(1 - \hat{y}^{(i)}\right)$$
$$met\ \log(x) = \ln(x)$$
$$en\ met\ \hat{y}^{(i)} = W^T x^{(i)} + b$$

Deze functie heet de Logaritmische Loss[1] functie. Let er op dat met de log() functie de natuurlijke logaritme bedoeld wordt ($log_e(x) = \ln(x)$). In machine learning wordt vaak de aanduiding log() met basis e gebruikt. Ik zal dat dus ook doen. De formule moet je als twee delen zien. Een voor het geval dat y = 1, de hele vergelijking wordt dan versimpeld tot $-log\left(\hat{y}^{(i)}\right)$, omdat de rechter term met $(1 - 1) = 0$ wordt vermenigvuldigt. Voor y = 0 wordt de hele vergelijking: $-log\left(1 - \hat{y}^{(i)}\right)$, omdat de linker term met 0 wordt vermenigvuldigd. De grafieken zien er als volgt uit (links y = 1, rechts y=0):



In het geval dat y = 1 voor een bepaalde training example wordt de linker grafiek dus gebruikt. Als de voorspellingsfunctie in dit geval een waarde dichtbij de 1 voorspelt, is de output van de loss function dichtbij nul, zie grafiek. De voorspellingsfunctie heeft dus een goede voorspelling gedaan en er worden vrijwel geen strafpunten toegekend. Wanneer de voorspellingsfunctie echter een waarde dichtbij de 0 voorspelt, wat dus fout is aangezien y = 1, nadert de grafiek steeds verder naar het oneindige. De voorspellingsfunctie wordt dus steeds zwaarder gestraft voor zijn foute voorspelling. In het geval dat y = 0, is precies het omgekeerde het geval, zie de rechter grafiek.

[1] *Ook wel de Bernoulli Loss function of binary cross-entropy Loss function genoemd.*

Als we de Loss functie op alle training examples in een dataset willen toepassen, kunnen we de volgende vergelijking gebruiken. Deze functie noemen we ook wel de Cost function en geven we aan met de letter $\mathcal{J}$:

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$
$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(\hat{y}^{(i)}\right) + \left(1-y^{(i)}\right)log\left(1-\hat{y}^{(i)}\right)\right]$$
$$= -\frac{1}{m}(ylog(\hat{y}^{T}) + (1-y)log(1-\hat{y}^{T}))$$

In het voorbeeld telden we alle strafpunten bij elkaar op. Nu vermenigvuldigen we echter ook nog met 1/m. In feite berekenen we dus het gemiddelde strafpunt per training example. We tellen namelijk de output van de Loss function van alle training examples bij elkaar op. Deze som wordt vervolgens door het aantal training examples gedeeld. Verder heb ik het min teken uit de Loss function gehaald en deze voor de duidelijkheid vooraan neergezet. De laatste uitdrukking van de cost functie heeft betrekking op de berekening van output door middel van vector of matrix vermenigvuldigingen.

Stel dat we de volgende voorspellingsvector ($\hat{Y}$) voor 2 training examples en de echte klassenvector (Y) hebben:

$$\hat{Y} = [0.95 \;\; 0.39].\; Y = [1 \quad 0]$$

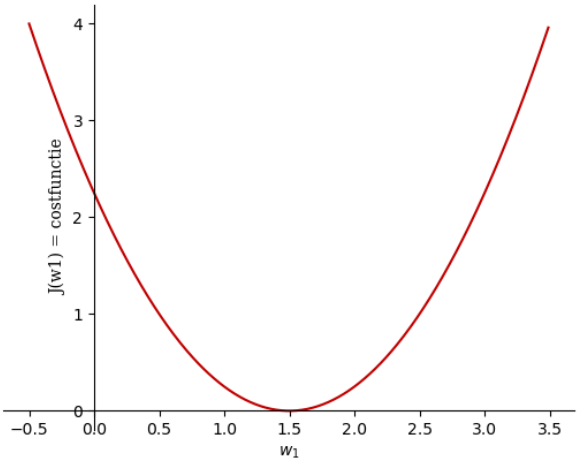
De computer is dus van de eerste training example 95% overtuigd dat deze leerling bij groep y = 1 hoort. Hij heeft dus een goede voorspelling gedaan. Van de tweede training example is de voorspellingsfunctie 39% zeker dat de leerling bij de groep y = 1 hoort (of 61% zeker dat hij/zij bij y = 0 hoort natuurlijk). 61% is meer dan 50% dus het was op zich ook geen verkeerde voorspelling. Toch kan de computer nog nauwkeuriger zijn. De uitkomst van cost function toegepast op beide training examples is:

$$\mathcal{J}([0.95 \;\; 0.39],[1 \quad 0]) = -\frac{1}{2}\left([1 \quad 0]log\left(\begin{bmatrix}0.95\\0.39\end{bmatrix}\right) + (1-[1 \quad 0])log\left(1-\begin{bmatrix}0.95\\0.39\end{bmatrix}\right)\right)$$
$$= -\frac{1}{2}\left([1 \quad 0]\begin{bmatrix}-0.051\\-0.94\end{bmatrix} + [0 \quad 1]\begin{bmatrix}-3.0\\-0.49\end{bmatrix}\right)$$
$$= -\frac{1}{2}\left((-0.051) + (-0.49)\right) = -\frac{1}{2}(-0.54)$$
$$= 0.27$$

Je ziet dat de cost functie het eerste training example relatief licht straft. Deze was immers ook goed voorspeld door de computer. De tweede voorspelling heeft een zwaardere straf gekregen. Deze heeft de computer namelijk met iets minder zelfvertrouwen gemaakt. Uiteindelijk heeft de computer met de huidige parameters een score van 0.27 weten te behalen. In de volgende paragraaf bekijken we hoe we de berg kunnen afdalen en de cost omlaag kunnen krijgen.

C. GRADIENT DESCENT (AFDALEN VAN DE BERG)

In de vorige paragraaf hebben we een functie gevonden die strafpunten aan de voorspellingsfunctie toekent. Deze functie noemden we de cost functie. Nu willen we de cost functie minimaliseren door stapjes in de richting van het dal van een berg te nemen. De hoogte van deze berg is de waarde van de cost functie, wanneer de computer voor het eerst zijn gok doet. Dit betekende dus dat de parameters voor de eerste gok willekeurig gekozen zijn. Het dal van deze berg staat gelijk aan het laagste aantal strafpunten ofwel de laagste waarde die de cost functie kan aannemen. We gaan nu weer in op het voorbeeld in het begin van het PWS. Stel dat we alleen het cijfer voor wiskunde beschouwen, we hebben dus 1 parameter. We negeren namelijk de bias term in dit voorbeeld. Als we de cost function tegen de enige parameter uitzetten, met de cost function op de y-as en de parameter w_1 op de x-as, heeft de grafiek vrijwel altijd een dal met 1 globaal minimum[2]. Er zijn dus geen kleinere dalen hoger op de berg waar we verdwaald kunnen raken.



In het midden van de plot zie je een dal met aan de linkerkant de top van een berg en aan de rechter kant de top van een andere berg. Je ziet dat we w_1 zo kunnen kiezen dat we in het dal van de cost functie terecht kunnen komen. De waarde voor w is in eerste instantie willekeurig gekozen. Deze kan dus zowel links als rechts van het dal beginnen. We kunnen het dal al in de verte zien en willen nu zo snel mogelijk van deze berg afdalen. Om het juiste pad te kiezen en om zo snel mogelijk omlaag te komen kunnen we Gradient Descent (GD) gebruiken. Het algoritme van Gradient Descent ziet er als volgt uit:

$$w_n := w_n - \alpha \frac{d\mathcal{J}(w_n)}{dw_n}$$

$\frac{d\mathcal{J}(w_1)}{dw_1}$ is de afgeleide van de Cost functie ten opzichte van de parameter w_1. Dit kunnen we zien als de steilheid van de cost function op de x-coördinaat w_1. We gaan nu kijken wat er gebeurt als de eerste keuze voor w links of recht van het dal ligt.

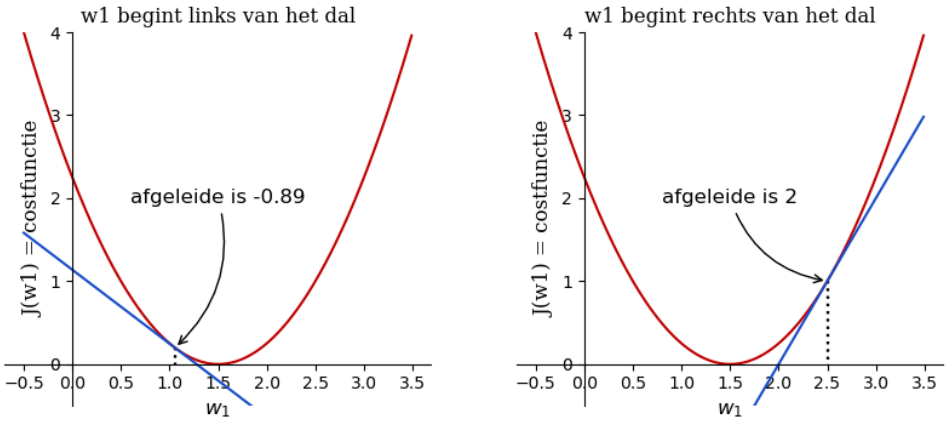
[2] De reden dat we de Logaritmische Loss functie gebruiken, is omdat deze functie convex is. Dit betekent dus dat men voor elke waarde van w_1 in principe op het globale minimum moet komen.

1. w_1 ligt rechts van het dal.

Als w rechts van het dal ligt, heeft de afgeleide functie in het bijbehorende punt een positieve waarde. De afgeleide zal volgens het algoritme van GD van de parameter w worden afgetrokken. Dit betekent dat de nieuwe waarde van w nu iets verder naar links verschoven wordt. We hebben een stapje in de juiste richting genomen en bevinden ons nu iets dichter bij het dal.

2. W ligt links van het dal.

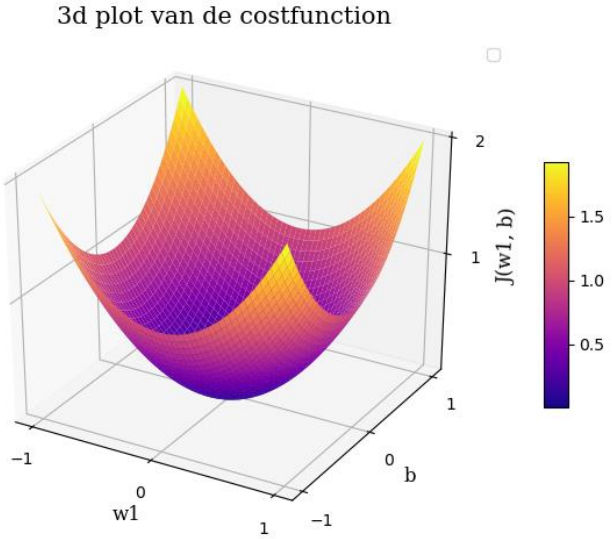
Als w links van het dal ligt, heeft de afgeleide functie in het bijbehorende punt een negatieve waarde. Als $\frac{dJ(w_1)}{dw_1}$ in dat punt bijvoorbeeld -0.89 is, zal er $-(-0.89) = 0.89$ bij w worden opgeteld. Dit betekent dat de nieuwe waarde van w nu iets verder naar recht verschoven wordt, en dus iets dichterbij het dal zal liggen. Ook aan de linkerkant kunnen we dus dichterbij het dal komen.



Dit algoritme zal een aantal iteraties herhaald worden. Zo zetten we steeds kleine stapjes naar het dal toe en zal de parameter w_1 een steeds betere waarde aannemen, de optimale waarde van w_1 is uiteindelijk 1.5 zie de afbeelding hierboven. α is een hyperparameter, dit is dus niet hetzelfde soort parameter als w of b maar een parameter dat ons model in zijn geheel beïnvloed. α noemen we ook wel de learning rate, een maat voor de grootte van de stap die we naar het dal nemen. We zullen zien dat het belangrijk is om en juiste waarde voor α te kiezen. Als deze te klein is, kan het lang duren voordat we in het dal staan. Wanneer α te groot is, kunnen we te grote stappen nemen en het dal de hele tijd voorbijschieten. Bij het implementeren in code zullen we dit concreet zien.

Stel dat we nu de bias term b erbij betrekken. Samen met de parameter w_1 kunnen we een driedimensionale plot maken met de cost function op de verticale-as, en de parameters w_1 en b op de horizontale assen. De cost functie lijkt dan steeds meer op een dal die wij in onze driedimensionale wereld gewend zijn. Om het dal heen bevinden zich allemaal hoge bergtoppen. De cost functie is nu een oppervlak dat boven het horizontale vlak zweeft. We zouden op elk punt op dit oppervlak kunnen starten, w_1 en b worden voor de eerste gok nogmaals willekeurig gekozen. Het doel is wederom om zo dicht mogelijk bij het dal terecht te komen, het meest paarse gedeelte van de plot. In plaats van twee richtingen (omhoog of omlaag) kunnen we nu $360°$ om ons heen bewegen. Er zijn dit keer dus meer paden, er is echter maar 1 pad dat ons het snelst naar dat dal doet verplaatsen. Differentiëren met meerdere variabelen

werkt op een iets andere manier. Dit is echter niet belangrijk om te begrijpen. In de lineaire algebra is de gradient van een multivariabele functie $\mathcal{J}$, $\nabla\mathcal{J}$. Dit is een vector in de richting van de steilste route omhoog. Het eerste element in deze vector is de afgeleide in de w_1 richting en het tweede element is de afgeleide in de b richting. In plaats van $\frac{d\mathcal{J}(w_1)}{dw_1}$ noteren we afgeleiden met meerdere variabelen nu iets anders. Namelijk: $\frac{\partial\mathcal{J}(w_1,b)}{\partial w_1}$ (de d is vervangen door een ∂) dit is alleen een andere vorm van notatie en heeft voor ons verder geen belangrijke implicaties. De richting die we moeten nemen om zo snel mogelijk naar beneden te komen is $-\nabla\mathcal{J}(w_1, b)$.



Multivariabele calculus is voor nu alleen belangrijk bij de notatie van afgeleiden van multivariabele functies. Ondanks het feit dat het voor mensen onmogelijk is om in meer dan drie dimensies te denken, moet je je voorstellen dat de cost functie een n-dimensionaal (met n=aantal features) landschap is, waar GD ervoor zal zorgen dat we steeds dichterbij het n-dimensionale dal komen.

De formele regels die we zullen gebruiken om de juiste waarden voor één parameter w_n en de bias term b te vinden, staan hieronder. Alle training examples krijgen bij bij batch GD (GD waar alle training examples gebruikt worden, voor andere mogelijkheden zie een na laatste paragraaf) zeggenschap over de richting waarin we een stapje moeten nemen. Om de beste richting te kiezen, nemen we het gemiddelde van alle afgeleiden van de training examples. Zie de algoritmen hieronder.

$$w_n := w_n - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial w_n}\mathcal{J}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

$$b := b - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b}\mathcal{J}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

In de bovenste regel staat het subscript n voor een individuele parameter. Voor alle parameters in een vector w geldt:

$$w := w - \nabla\mathcal{J}(w) = w - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial w}\mathcal{J}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Op het eerste gezicht lijkt dit een hele vreemde methode om het dal van een parabolische functie te vinden. Normaal gesproken zou je de afgeleide van de functie eerst op 0 herleiden om de x-coördinaat van een dal te berekenen. Nu is dit alleen bij linear regression mogelijk. We gebruiken dan de Normal equation:

$$w = (X^T X)^{-1} X^T y$$

Vroeger kon deze methode bij kleine datasets makkelijk toegepast worden. In het tijdperk van de BigData is deze methode, zelfs voor een computer zeer lastig uit te rekenen. Dat komt, omdat de inverse van een matrix berekend moet worden. Dat proces kan erg lang duren.

Dit is wat we bedoelen als we zeggen dat een neural network leert. Het berekenen van afgeleiden. In de volgende paragraaf gaan we deze afgeleiden bepalen.

D. AFGELEIDE VAN DE COST FUNCTION

We moeten nu nog de juiste uitdrukking zien te vinden voor $\frac{\partial \mathcal{J}(w_n,b)}{\partial w_n}$, ofwel de afgeleide van de cost functie ten opzichte van de weights w en de bias b. Bedenk dat we de volgende formules hebben om de Loss function[3] te berekenen:

$\mathcal{L}(a,y) = -y\,log(a) - (y-1)log\,(1-a)$

$a^4 = \sigma(z) = \frac{1}{1+e^{-z}}$

$z = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$
$(W^TX + b\ voor\ alle\ features\ en\ training\ examples)$

Met behulp van de kettingregel[5] kunnen we $\frac{\partial \mathcal{L}}{\partial w_n}$ als volgt berekenen:

$$\frac{\partial \mathcal{L}}{\partial w_n} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_n}$$

Mensen zonder wiskundige achtergrond of mensen die gewoon een hekel aan wiskunde hebben :(kunnen dit onderdeel overslaan. Ik raad echter aan om het zelf uit te werken. Hieronder staan de antwoorden. Het is de bedoeling dat we nu $\frac{\partial \mathcal{L}}{\partial a}$, $\frac{\partial a}{\partial z}$ en $\frac{\partial z}{\partial w_n}$ apart berekenen, en ze vervolgens met elkaar vermenigvuldigen om $\frac{\partial \mathcal{L}}{\partial w_n}$ te krijgen.

1. *De afgeleide van $\mathcal{L}(a,y)$ ten opzichte van a*

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial}{\partial a}(-y\,log(a) - (y-1)log\,(1-a)),\ \left(\frac{d}{dx}log(x) = \frac{1}{x}\right)$$
$$\frac{\partial \mathcal{L}}{\partial a} = -y * \frac{1}{a} - (y-1) * \frac{1}{1-a} * -1$$

[3] De Loss function - dus niet de cost function - berekent de strafpunten van een enkele training example.
[4] We gebruiken a i.p.v $\hat{y}$ dit is in het vervolg namelijk iets accurater.
[5] Afgeleide van een ketting functie is de afgeleide van de buitenste schakel keer de afgeleide van de binnenste schakel. De binnenste schakel heeft echter nog een schakel.

$$\frac{\partial \mathcal{L}}{\partial a} = -\frac{y}{a} + \frac{y-1}{1-a}$$
$$\left(\frac{\partial \mathcal{L}}{\partial a} = \frac{-y(1-a) + a(1-y)}{a(1-a)}\right)$$

2. *De afgeleide van a ten opzichte van z*

$$\frac{\partial a}{\partial z} = \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) = \frac{\partial}{\partial z}((1+e^{-z})^{-1})$$
$$\frac{\partial a}{\partial z} = -1 * (1+e^{-z})^{-2} * -e^{-z} = \frac{e^{-z}}{(1+e^{-z})^2}$$
$$\frac{\partial a}{\partial z} = \frac{1}{(1+e^{-z})}\frac{e^{-z}}{(1+e^{-z})}$$
$$\frac{\partial a}{\partial z} = \frac{1}{(1+e^{-z})}\frac{(1+e^{-z}) - 1}{(1+e^{-z})}, \quad (+1 \; -1 \; in\; de\; teller\; blijft\; hetzelfde)$$
$$\frac{\partial a}{\partial z} = \frac{1}{(1+e^{-z})}\left(1 - \frac{1}{(1+e^{-z})}\right)$$
$$a = \frac{1}{(1+e^{-z})}, \qquad (substitutie\; met\; a)$$
$$\frac{\partial a}{\partial z} = a(1-a)$$

3. *De afgeleide van z ten opzichte van w_n*

$$\frac{\partial z}{\partial w_n} = \frac{\partial}{\partial w_n}(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b) = x_n$$

4. *De afgeleide van z ten opzichte van b*

$$\frac{\partial z}{\partial b} = \frac{\partial}{\partial b}(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b) = 1$$

5. *De afgeleide van $\mathcal{L}(a,y)$ ten opzichte van w_j en b*

$$\frac{\partial \mathcal{L}}{\partial w_n} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_n} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w_n}$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b}$$

5.1 (Tussenstap) De afgeleide van $\mathcal{L}(a,y)$ ten opzichte van z

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z}$$
$$\frac{\partial \mathcal{L}}{\partial z} = \frac{-y(1-a) + a(1-y)}{a(1-a)} * a(1-a)$$

$$\frac{\partial \mathcal{L}}{\partial z} = -y(1-a) + a(1-y) = -y + ay + a - ay$$
$$\frac{\partial \mathcal{L}}{\partial z} = a - y$$

5.2 De afgeleide van $\mathcal{L}(a, y)$ ten opzichte van w_n en b

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial z} x_n = (a-y)x_n$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} = (a-y)$$

Voor een parameter w_n en de bias term b betekent GD dus (zie vorige paragraaf):

$$w_n := w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right) x_n{}^{(i)}$$
$$b := b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right)$$

Met matrix vermenigvuldigingen kunnen we dit versimpelen tot:

$$W := W - \alpha \frac{1}{m} X\, dZ^T$$
$$W := W - \alpha \frac{1}{m} \sum_{i=1}^{m} (dZ)$$
$$met\ dZ = \frac{\partial \mathcal{L}}{\partial z}$$

Algoritme 1: gradient descent voor Logistic Regression

Voor i = 1 tot max_herhalingen:

 # bereken de activation A:
$\boldsymbol{Z = W^T X + b}$
$\boldsymbol{A = \sigma(Z)}$

 # bereken de afgeleide van W en b:
$\boldsymbol{dZ = A - Y}$
$\boldsymbol{dW = \frac{1}{m} X dZ^T}$
$\boldsymbol{db = \frac{1}{m} \sum_{i=1}^{m} (dZ)}$

 # update de parameters:
$\boldsymbol{W := W - \alpha dW}$
$\boldsymbol{b := b - \alpha db}$

PYTHON

In dit hoofdstuk zullen we wat verder in gaan op het programmeren en het gebruik van matrices en vectoren om het programma zo snel, en efficiënt mogelijk te laten lopen.

Eerst zal ik het hebben over programmeren. Voordat ik ooit ben begonnen met Artificial Intelligence, was ik al erg geïnteresseerd in programmeren. Er bestaan verschillende programmeertalen. De programmeertaal die in de wetenschap en vooral in machine learning het meest wordt gebruikt, is Python. Dit komt omdat Python makkelijk te leren is, en je veel libraries kan downloaden. Dit zijn extensies met code van andere mensen zodat je dat zelf niet allemaal hoeft te doen. Voor de implementatie in code is er echter geen voorkennis van Python nodig, aangezien ik vooral in zal gaan op de wiskunde. Het is wel handig om een aantal dingen te weten:

- Een “for” loop wordt gebruikt voor het herhalen van de code dat in het blok van de for loop staat. Hierbij neemt “i” steeds de volgende waarde aan in de reeks, in dit geval een reeks van 0 tot x met een stapgrootte van 1.

```
for i in range(x):
      #<Code die x aantal keer herhaald moet worden>
```

- Het trefwoord “def” wordt gebruikt om een functie aan te geven. Een functie heeft input parameters, deze staan tussen de haakjes rechts van de naam van de functie gedefinieerd. Verder heeft een functie vaak ook een output, deze staat achter het trefwoord “return”.

```
def naam_functie(input):
      output = #<code voor output die berekend moet
worden>
      return output
```

Een van de extensies die we zullen gebruiken is Numpy. Een operatie met behulp van Numpy wordt in de code aangegeven als np.<operatie>. Dit is een library die het vermenigvuldigen van matrices en vectoren zo efficiënt mogelijk maakt. Daarnaast heeft het een aantal wiskunde functies die erg van te pas zullen komen. Zo zullen de volgende functies vaak gebruikt worden:

1. np.exp()

Deze functie neemt elk element van de matrix of vector tussen de haakjes en zet deze in de macht van het grondtal e.

```
np.exp(x)
```

2. np.log()

Deze functie neemt elk element van de matrix of vector tussen de haakjes en neemt er de natuurlijke logaritme van.

```
np.log(x)
```

3. np.array()

Deze functie maakt van een list (een object in standaard Python) een matrix of vector (Numpy array, een object in numpy).

```
np.array(object)
```

4. np.zeros()

Deze functie maakt een matrix met alleen het cijfer “0” in elke rij en in elke kolom.

```
np.zeros((dimensies))
```

5. np.dot()

Deze functie neemt het dot product van twee matrices.

```
np.dot(a, b)
```

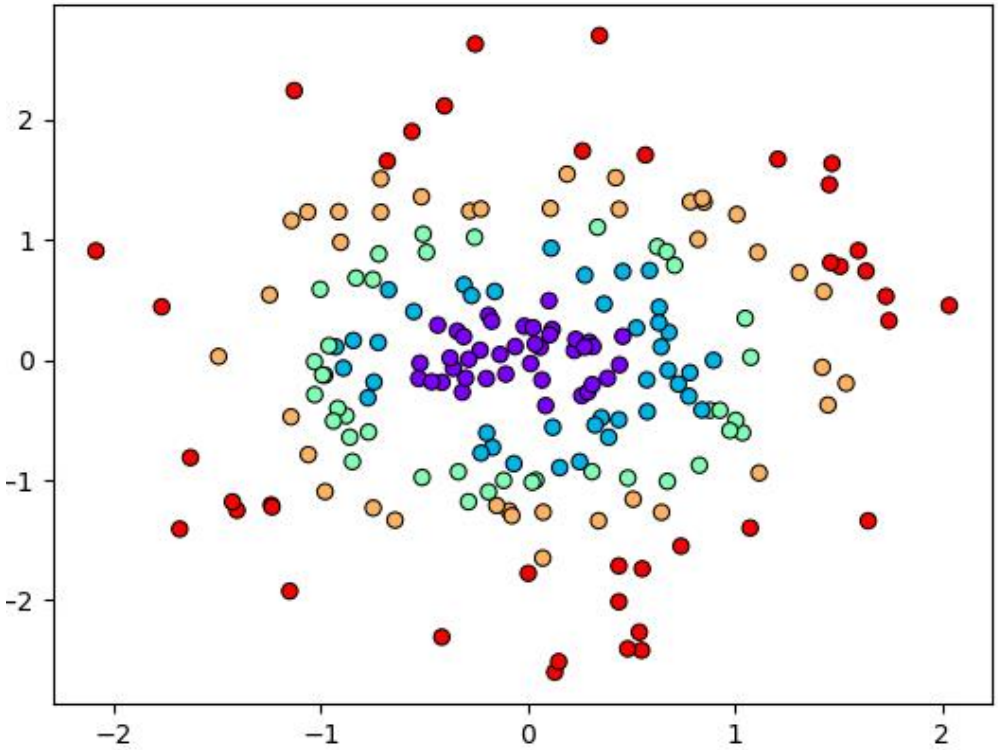
Naast numpy zijn libraries als Tensorflow, Keras en bijvoorbeeld Scikit-learn bekende extensies in de machine learning wereld. In het volgende hoofdstuk zullen we zien dat we qua wiskunde nog een aantal dingen moeten uitwerken voordat we een werkend neural network kunnen maken. In de zojuist benoemde extensies is deze wiskunde al ingebouwd. Het enige wat je moet specificeren is de cost function die je gaat gebruiken, het aantal lagen / aantal neurons per laag en de activation functies per laag waarvan je gebruik zult maken (zie voor de laatste twee het volgende hoofdstuk). Verder moet je gestructureerde training data aanleveren. Dit betekent dat er een waarde of klasse moet zijn die het neural network kan voorspellen. Uit random data zal het neural network niets leren. Als je bijvoorbeeld wilt dat een neural network de temperatuur van een bepaalde dag kan voorspellen, moet ten eerste de data bestaan uit heel veel training examples. In dit voorbeeld zouden de training examples kunnen bestaan uit de dagen in een bepaalde periode (jaar of meer). Ten tweede moet de training data ook veel features hebben. Per training example zouden de features bijvoorbeeld de temperatuur, de luchtvochtigheid, de lengtegraad, de breedtegraad, windsnelheid, de zonneconstante, etc. kunnen zijn. Het is de temperatuur die we willen voorspellen. Deze halen we dus uit de training data en wordt onze klassen vector. De waarden in deze vector zijn nog reële getallen. We zouden kunnen zeggen dat een temperatuur $< 15℃$ de klasse ‘koud’ wordt en een temperatuur $\geq 15℃$ de klasse ‘warm’ wordt. We zouden natuurlijk ook een andere feature als label kunnen nemen. Stel dat we de feature windsnelheid als label nemen. Zo kunnen we dus ook de windsnelheid van een bepaalde dag proberen te voorspellen.

NEURAL NETWORKS

DE BEHOEFTE AAN EEN COMPLEXER MODEL

A. TWEE PROBLEMEN

We hebben we een veel complexer model nodig om een computer de taak te geven, afbeeldingen van verschillende handgeschreven cijfers te onderscheiden. Logistic regression heeft ons in het vorige hoofdstuk al heel ver gebracht. Bij het plotten van een decision boundary, krijgen we een lineaire lijn. Als de distributie van de data punten er echter als onderstaande figuur uit had gezien, zou een simpele logistic regression classifier nooit een hoge nauwkeurigheid kunnen behalen. Immers, LG kan alleen lineaire decision boundaries beschrijven. In de afbeelding hieronder zouden elliptische decision boundaries echter het beste passen.



Bovendien kan een Logistic Regression classifier alleen onderscheid maken tussen twee klassen, Als we meerdere klassen willen classificeren, in de afbeelding hierboven onderscheden door verschillende kleuren, hebben we aan Logistic Regression dus nogmaals niet genoeg.

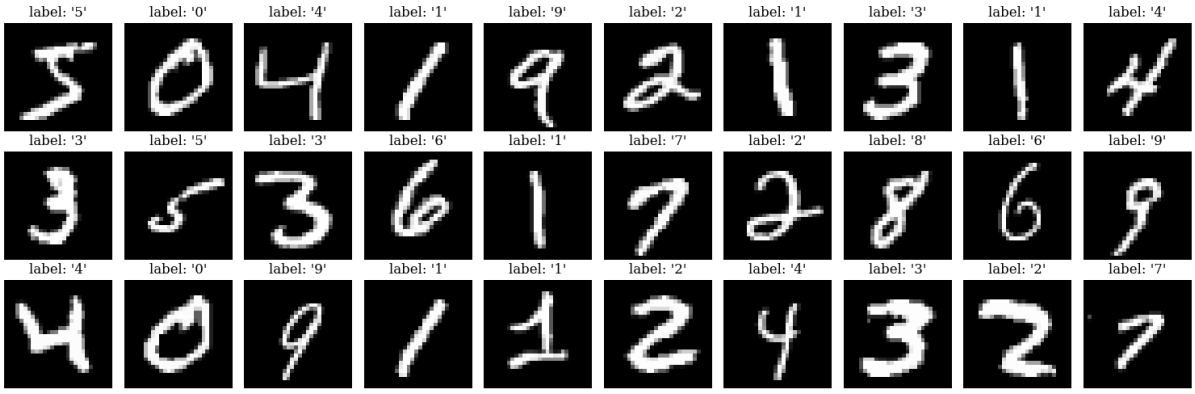
Neural Networks (NN) bieden de oplossing. Ze zijn de heilige Graal van artificial intelligence in de 21^{e} eeuw. Zoals ik in het begin van dit PWS al heb besproken, worden NN's in bijna alle toepassingen van artificial intelligence gebruikt (zelfrijdende auto's, classificatie, computer visie, enz.). In dit hoofdstuk zal de wiskundige strekking uitleggen, evenals een aantal manieren bespreken hoe we dit model beter en sneller kunnen maken. We zullen zien dat Neural Networks de oplossing bieden voor de twee bovenstaande problemen (complexere decision boundary en meerdere klassen).

B. TRAINING DATA

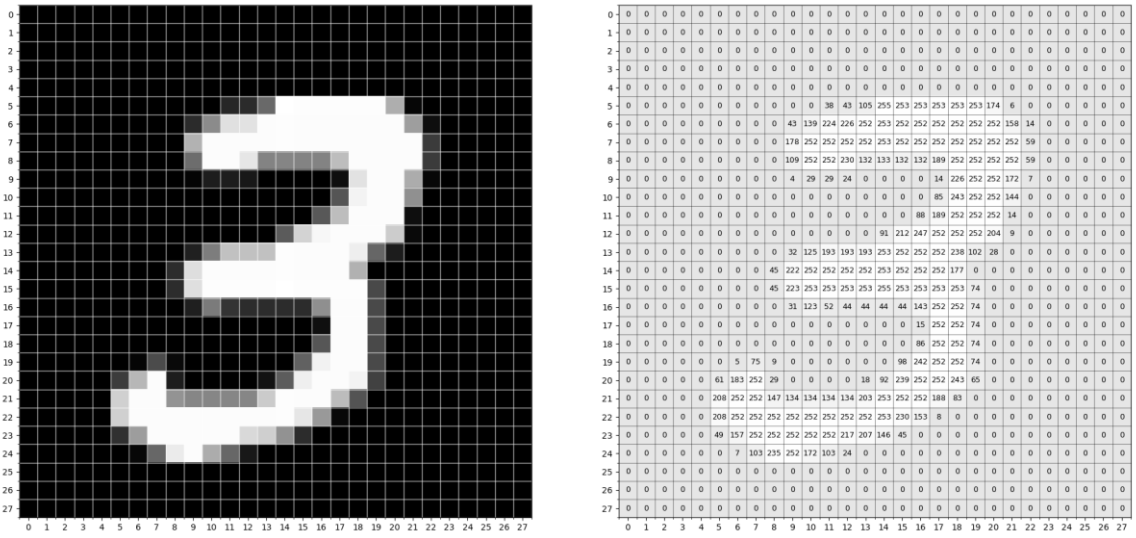
In dit hoofdstuk behandelen we de deelvraag:

- Hoe maak je een model dat handgeschreven cijfers kan onderscheiden?

Het model dat we gaan gebruiken is een Neural Network. Eerst is er een dataset nodig met genoeg afbeeldingen zodat het model genoeg training examples heeft om te leren. De dataset die we gaan gebruiken heet de MNIST dataset. Deze dataset bestaat uit 60000 afbeeldingen van handgeschreven cijfers $\epsilon\ \{0, 9\}$. Hieronder zie je een aantal voorbeelden.



De training data slaan we weer op in een input matrix X met in elke kolom een andere afbeelding (training example). In de rijen zitten de verschillende pixelwaarden (features, in totaal $28 \times 28 = 784$) opgeslagen. De dimensie van de matrix X is dus (784, 60000). Wij mensen zien een drie zoals die links in onderstaande afbeelding is afgebeeld. Een computer ziet de drie zoals die rechts is afgebeeld. Witte pixels hebben een waarde van 255 en zwarte pixels een waarde van 0. Alles ertussen heeft een waarde tussen de 0 en de 255. Voordat we de features van de afbeelding van de drie aan de matrix X kunnen toevoegen, moeten we eerst alle pixelwaarden in de rechterafbeelding als het ware uitrollen in een vector met de dimensie (1, 784). Dit doen we door elke rij van de matrix in de rechterkant van de afbeelding te nemen en deze achterelkaar te plakken. Het NN leert het beste als alle input features een bereik tussen de 0 en de 1 hebben. We delen alle waarden dus nog door het grootst aanwezige getal, in dit geval 255. Als laatste voegen we dit training example toe aan de training dataset X.



In plaats van twee klassen hebben we er nu 10 (0 tot 9). In het vorige hoofdstuk gebruikte we een horizontale vector met de dimensie (1, m) om de klassen in op te slaan. Met twee klassen codeerden we de twee verschillende groepen met een 0 of met een 1. Nu we 10 klassen hebben, moet de vector Y deze extra groepen ook kunnen onderscheiden. De derde klasse kunnen we in de vector Y bijvoorbeeld aanduiden met een 2, de vierde klasse kunnen we aanduiden met een 3 et cetera. In plaats van gehele getallen is het handiger om een zogeheten “One-Hot Encoding”

te gebruiken. Hierbij wordt Y uitgebreid tot een matrix en blijven we een binaire codering gebruiken. Het aantal kolommen staan nu nog steeds voor het aantal training examples. De rijen staan nu echter voor de verschillende klassen.

Stel dat we willen dat een Neural Network leert om onderscheid te maken tussen een kat, een hond, een vogel, een konijn en een vis. De label matrix Y heeft dus 5 rijen. Als we zeven training examples hebben waarvan de labels respectievelijk een [Kat, Vogel, Vis, Kat, Hond, Konijn, Vogel] zijn, dan ziet de matrix Y er als volgt uit:

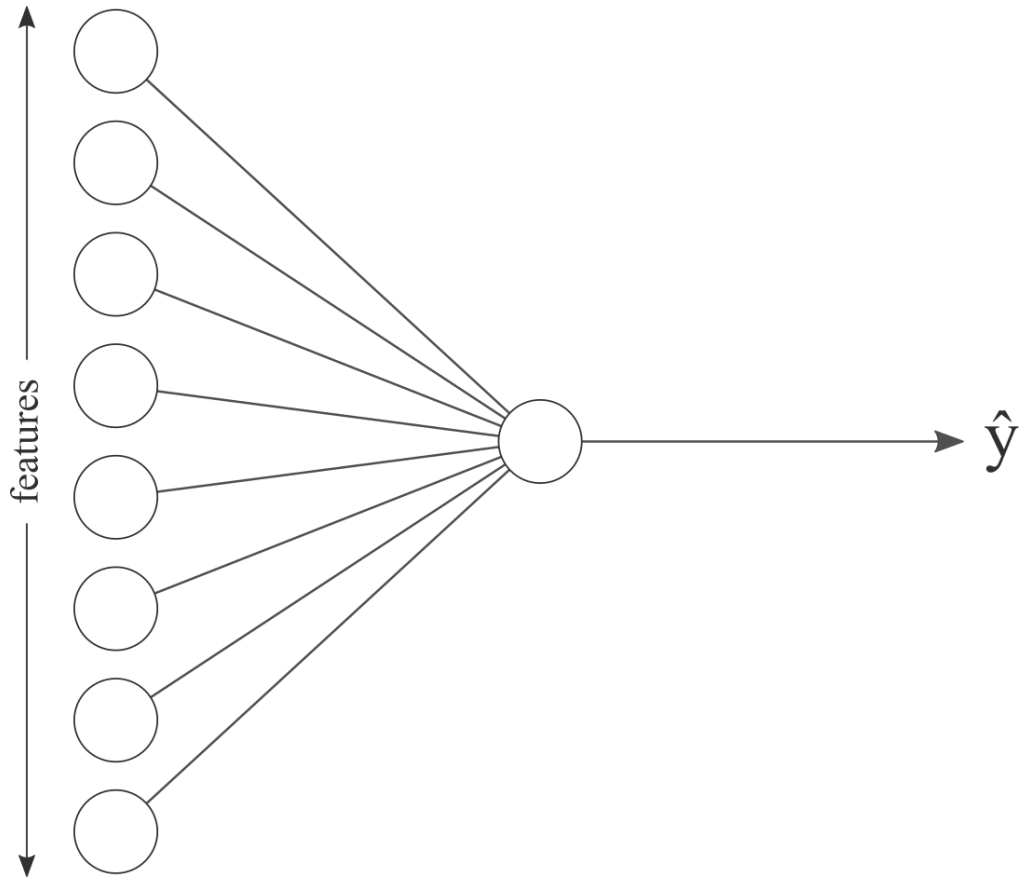
$$\begin{matrix} Kat \\ Hond \\ Vogel \\ Konijn \\ Vis \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Dit is een voorbeeld van een one-hot matrix. Waarom het handig is om de label matrix Y zo weer te geven, wordt later in de volgende paragraaf uitgelegd.

NEURAL NETWORKS

A. VOORSTELLING MODEL

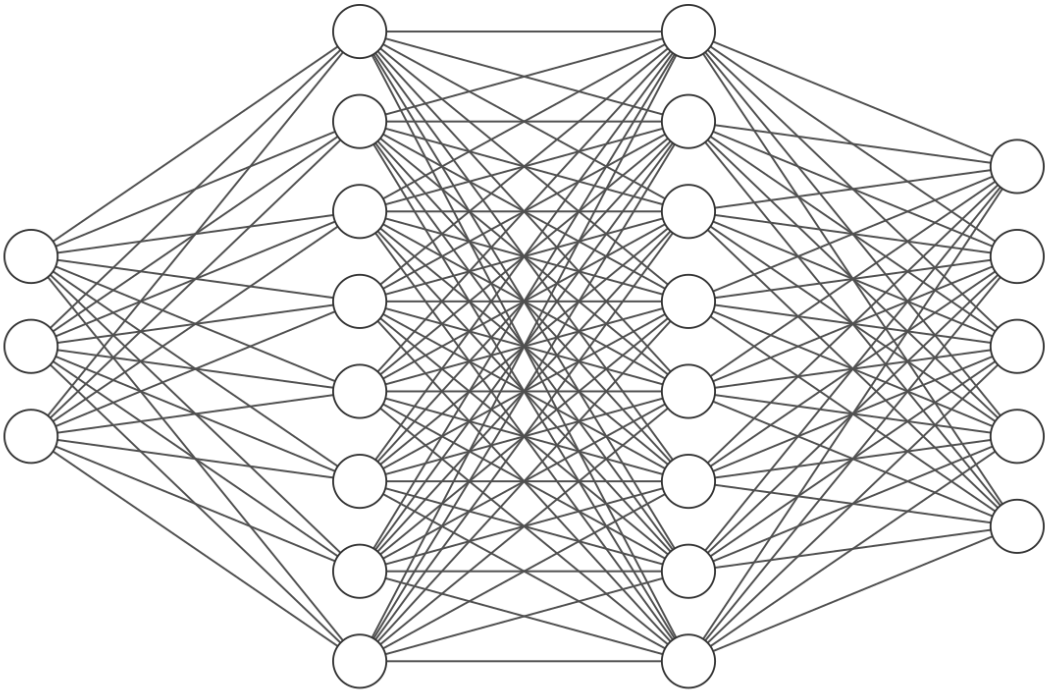
Het Logistic regression model uit het vorige hoofdstuk zouden we als één neuron kunnen voorstellen met als input de verschillende features en als output zijn voorspelling $\hat{y}$:



Ieder bolletje aan de linker kant staat voor een individuele feature. Onderweg naar de neuron wordt de waarde van een feature x vermenigvuldigt met zijn parameter (weight) w. In de middelste neuron worden alle waarden van de features, die vermenigvuldigd zijn met de parameters, bij elkaar opgeteld. Vervolgens wordt de bias term b ook nog hierbij opgeteld. Als laatste passen we sigmoid functie toe op de lineaire output. De output van dit neuron heeft dus een waarde tussen de 0 en de 1. Het beschreven model is in zekere zin een biologische analogie van zenuw cellen in ons brein. Je zult zien dat deze analogie handig is bij Neural Networks.

Net zoals zenuwcellen (neurons) in ons brein, kunnen we heel veel van deze neurons namelijk met elkaar verbinden. In plaats van de wirwar aan neuronverbindingen in ons brein, ordenen we de neurons in een neural network echter netjes in lagen. Als we dat doen dan hebben we in essentie een neural network (Deep Neural Networks zijn Neural Networks met heel veel lagen). Elk neuron in een laag $[\ell]$ is verbonden met de output van alle neurons in de vorige laag - laag $[\ell - 1]$ (de feature neurons beschouwen we nu dus ook als een laag). De output van dit neuron is weer verbonden met de input van alle neurons in de volgende laag - laag $[\ell + 1]$ - zie de

afbeelding hieronder. Het aantal lagen en neurons per laag zijn in het voorbeeld hieronder willekeurig gekozen.



De allereerste laag, de features, noemen we de input layer. Het aantal neurons in deze laag is gelijk aan het aantal features (net als LG). Daarna komen de hidden layers, het aantal neurons in deze lagen kunnen willekeurig gekozen worden. Sommige configuraties werken wel beter dan andere. Als laatst hebben we de output layer, het aantal neurons in deze laag is gelijk aan het aantal klassen. De hidden layers noemen we zo omdat deze als het ware verstopt blijven, we kunnen namelijk wel zien wat er in het NN gaat en wat er uit komt. Niet wat er binnenin gebeurt. We beginnen met tellen van het aantal lagen bij de eerste hidden layer. De input layer tellen we dus niet mee. Het Neural Network uit de afbeelding bovenaan deze pagina noemt men dus een Neural Network met drie lagen.

Met zo veel lagen die weer neurons bevatten, kan men snel de draad verliezen. Daarom is de notatie nu extra van belang. Een individueel neuron geven we aan met de notatie:

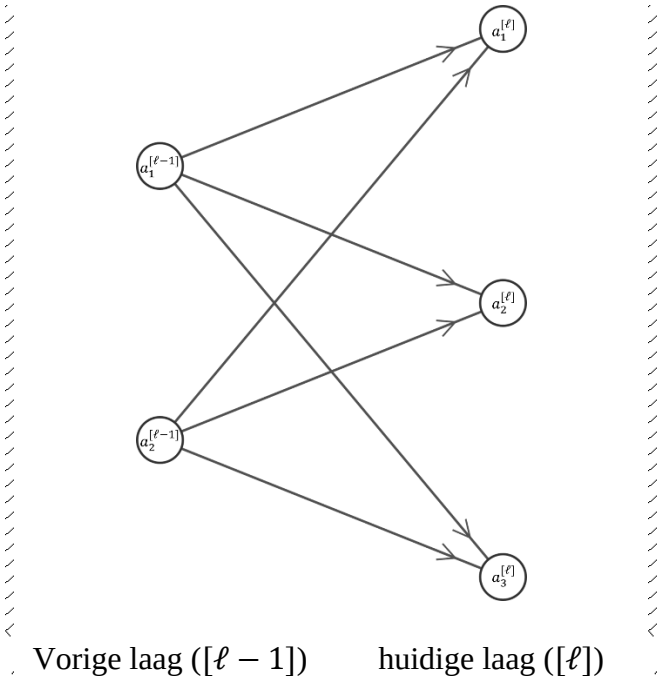
$$a_n^{[\ell]}$$

Met ℓ de laag waar het neuron zich in bevindt, en n de specifieke neuron in laag ℓ. Let op dat het getal tussen de vierkante haakjes nu staat voor de laag en niet voor een training example zoals we dat eerder noteerden. Het superscript $^{(i)}$ in ronde haakjes is namelijk de index voor een training example. Beschouw het voorbeeld hierboven, het zevende neuron van boven in de tweede hidden layer geven we aan met:

$$a_7^{[2]}$$

Merk op dat er 8 verbindingen bij dit neuron aankomen en er 5 verbindingen vertrekken.

Elk neuron heeft zijn eigen weights (parameters) en zijn eigen bias. Bij Logistic Regression zaten de weights van een neuron in een verticale vector en was de bias een reëel getal. Bij Neural Networks worden de weights van alle neurons in een laag $[\ell]$, horizontaal georiënteerd, in een matrix gezet.



laag ($[\ell-2]$) Vorige laag ($[\ell-1]$) huidige laag ($[\ell]$) laag ($[\ell+1]$)

Neem de weights van laag $[\ell]$ met drie neurons in bovenstaande afbeelding. De weight matrix van deze laag is:

$$W^{[\ell]} = \begin{matrix} a_1^{[\ell]} \\ a_2^{[\ell]} \\ a_3^{[\ell]} \end{matrix} \begin{bmatrix} - & {w_1^{[\ell]}}^T & - \\ - & {w_2^{[\ell]}}^T & - \\ - & {w_3^{[\ell]}}^T & - \end{bmatrix} = \begin{bmatrix} w_{11}^{[\ell]} & w_{12}^{[\ell]} \\ w_{21}^{[\ell]} & w_{22}^{[\ell]} \\ w_{31}^{[\ell]} & w_{32}^{[\ell]} \end{bmatrix}$$

(de waarden van de neurons worden hier niet met de weights vermenigvuldigd, ze staan er enkel om duidelijke te maken dat elke rij de parameters van elke neuron bevat)

Je ziet dat er twee lijntjes naar elk neuron in laag $[\ell]$ gaan. Dit betekent alle drie de neurons in laag $[\ell]$ twee input neurons hebben. Er zijn per neuron in laag $[\ell]$ dus twee parameters nodig. De dimensie van de weight matrix kunnen we berekenen met de formule: $(\#aantal\ neurons\ laag\ [\ell],\ \#aantal\ neurons\ laag\ [\ell-1])$. In bovenstaand voorbeeld dus: (2, 3).

De biases van elke neuron worden in een verticale vector gezet. Voor laag $[\ell]$ betekent dit dus:

$$b^{[\ell]} = \begin{matrix} a_1^{[\ell]} \\ a_2^{[\ell]} \\ a_3^{[\ell]} \end{matrix} \begin{bmatrix} b_1^{[\ell]} \\ b_2^{[\ell]} \\ b_3^{[\ell]} \end{bmatrix}$$

Het neural network in de afbeelding op pagina 35 heeft 3 lagen. Er zijn dus drie weight matrices nodig:

$W^{[1]} \in \mathbb{R}^{8 \times 3}, W^{[2]} \in \mathbb{R}^{8 \times 8}$ *en* $W^{[3]} \in \mathbb{R}^{5 \times 8}$

En drie bias vectoren:

$b^{[1]} \in \mathbb{R}^{8}, b^{[2]} \in \mathbb{R}^{8}$ *en* $b^{[3]} \in \mathbb{R}^{5}$

In dit hoofdstuk moeten we, net als bij logistic regression, de afgeleiden van alle parameters zien te vinden. Dit doen we door eerst de output van het hele neural network te berekenen. Feitelijk gaan we eerst de output van een aantal voorspellingsfuncties uit het vorige hoofdstuk berekenen. Het aantal voorspellingsfuncties staat hier voor het aantal neurons in de eerste laag. Daarna berekenen we de output van de tweede laag voorspellingsfuncties, die als input de output van de voorspellingsfuncties in de eerste laag hebben. We blijven dit doen voor alle lagen tot we bij de laatste laag zijn. De neurons in deze laag zijn uiteindelijk de echte voorspelling. Een training example begint als het ware bij de eerste laag. Daarna wordt de output van de eerste laag berekend. Vervolgens worden deze waarden door het hele neural network heen gepropageerd. We noemen deze stap dus ook wel forward propagation.

Als we de output van het hele neural network hebben berekend, kunnen we de afgeleiden van alle parameters berekenen. Dit doen we achterstevoren. We beginnen in de laatste laag en berekenen de afgeleiden van de parameters in deze laag. Dit verschilt niet veel van logistic regression. Vervolgens moeten we een tussenstap berekenen waarmee in de een na laatste laag de afgeleiden berekend kunnen worden. Dit blijven we doen tot we weer bij het begin van het neural network zijn. Het berekenen van afgeleiden noemen we ook wel backpropagation.

B. EEN VOORBEELD MET LOGIC GATES

In de vorige paragraaf heb ik beschreven dat we neural networks nodig hebben om complexere decision boundaries te maken. Ik zal nu aan de hand van logic gates toelichten waarom we hiervoor meerdere lagen moeten gebruiken. Zoals je misschien weet kunnen computers alleen werken op binaire gegevens. Ze kunnen dus alleen werken met bits oftewel een 0 of een 1. Op het allerlaagste niveau bestaat je computer uit allemaal transistors. Transistors laten elektrische stroom door als ze ‘aan staan’. Als ze uit staan, laten transistors geen stroom door. Van transistors kan je logic gates maken. Dit zijn schakelingen die een output op twee bits kunnen berekenen. (In je computer is de NAND gate de basis logic gate. Van de NAND gates worden er allerlei complexere circuits gemaakt.) De logic gate die we met behulp van neural networks proberen te maken heet de XNOR gate. Deze berekent voor elke combinatie van de waarden 0 en 1 een bepaalde output. Zowel de combinaties van inputs en de bijbehorende outputs zijn hieronder in een zogenaamde truth table weergegeven.

XNOR

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	1

In een machine learning context zijn de x combinaties onze features, er zijn twee features. Er zijn vier training examples (m=4). De output is de klassen vector (y), er zijn twee klassen.

De tabel zouden we ook als de volgende feature matrix X en de label vector y kunnen interpreteren.

$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \quad y = [1 \quad 0 \quad 0 \quad 1]$$

Dit betekent dus dat wanneer de computer de vector $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ als input krijgt, hij een output van 1 moet voorspellen. Bij de vector $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ zou de computer de output 0 moeten voorspellen, etc.

De XNOR gate kan in principe door drie andere logic gates gemaakt worden. Namelijk de AND, NOR en OR gate. De AND gate geeft als output een 1 als zowel x_1 als x_2 ook een 1 zijn. De NOR gate geeft als output alleen een 1 als zowel x_1 als x_2 een 0 zijn. De OR gate heeft als output in elk geval een 1, behalve als zowel x_1 als x_2 een 0 zijn. Hieronder zie je de truth tables van de AND, NOR en OR gate.

AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

NOR

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	0

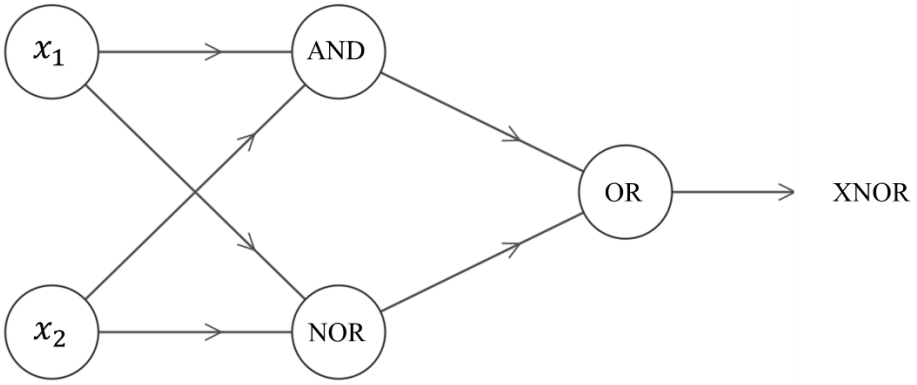
OR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Als we nu de output van de AND gate en NOR gate als input van de OR gate zien, krijgen we de juiste output van de XNOR gate. Volgens de truth table van de OR gate geeft deze gate namelijk een 1 als output wanneer een van de input waarden een 1 is en de ander 0. De OR gate geeft als output 0 als beide input waarden 0 zijn, zie tabel hieronder.

x_1	x_2	x_1 AND x_2	x_1 NOR x_2	(x_1 AND x_2) OR (x_1 NOR x_2)
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

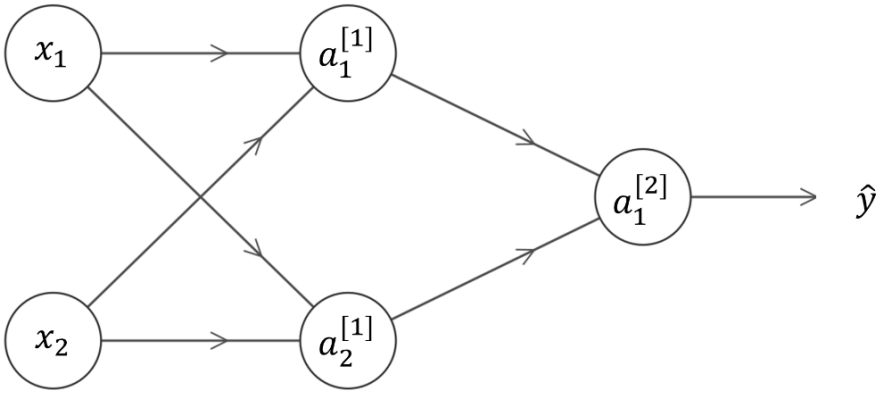
Grafisch gezien kunnen we de drie logic gates als volgt oriënteren om de XNOR gate te maken.



Dit begint al steeds meer op een neural network te lijken. Als we alle logic gates nu vervangen met neurons kan je je de volgende tabel voorstellen.

x_1	x_2	$a_1^{[1]}$	$a_2^{[1]}$	$a_1^{[2]} = \hat{y}$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Grafisch gezien:



Nu moeten we alleen nog de juiste parameters vinden en we hebben een neural network. De intuïtie die je uit dit voorbeeld moet meenemen, is het feit dat de AND en NOR gates als het ware nieuwe features maken, waarmee de OR gate de juiste output kan berekenen. Met 1 neuron hadden we op geen enkele manier de juiste output kunnen krijgen.

C. FORWARD PROPAGATION

Om de output van het Neural Network te berekenen, propageren we de features (input layer) van links naar rechts door het netwerk. De features, weights en biases zijn niet voor niets, respectievelijk, netjes in de matrices X, W en de vector b geplaatst. De output van een laag $[\ell]$ kan nu simpelweg berekend worden met de volgende matrix berekeningen:

Formules:

$$Z^{[\ell]} = W^{[\ell]} A^{[\ell-1]} + b^{[\ell]} \; met \; A^{[0]} = X$$

$$A^{[\ell]} = g(Z^{[\ell]})$$

Uitgeschreven matrices:

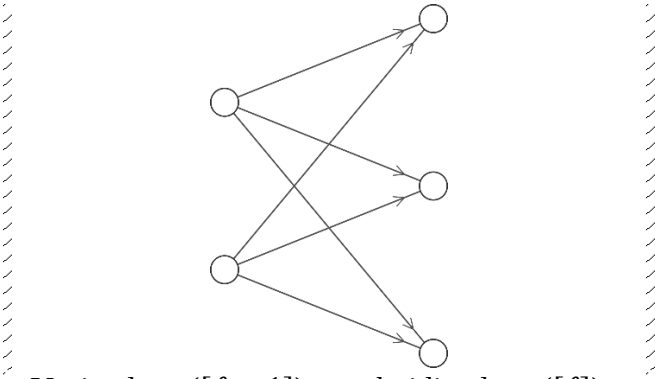
$$\begin{bmatrix} | & | & & | \\ Z^{[\ell](1)} & Z^{[\ell](2)} & \dots & Z^{[\ell](m)} \\ | & | & & | \end{bmatrix} = \begin{bmatrix} - & W_1^{[\ell]T} & - \\ - & W_2^{[\ell]T} & - \\ & \vdots & \\ - & W_n^{[\ell]T} & - \end{bmatrix} \begin{bmatrix} | & | & & | \\ A^{[\ell](1)} & A^{[\ell](2)} & \dots & A^{[\ell](m)} \\ | & | & & | \end{bmatrix} + \begin{bmatrix} b_1^{[\ell]} \\ b_2^{[\ell]} \\ \vdots \\ b_3^{[\ell]} \end{bmatrix}$$

$$\begin{bmatrix} | & | & & | \\ A^{[\ell](1)} & A^{[\ell](2)} & \cdots & A^{[\ell](m)} \\ | & | & & | \end{bmatrix} = g\left(\begin{bmatrix} | & | & & | \\ Z^{[\ell](1)} & Z^{[\ell](2)} & \cdots & Z^{[\ell](m)} \\ | & | & & | \end{bmatrix}\right)$$

Met g wordt de activation functie bedoeld. In het geval van Logistic Regression is dit de sigmoid functie. Je ziet dat in de matrix Z, waar de lineaire output van alle neurons zitten, de training examples weer netjes in kolommen geordend zijn. De eerste rij van deze matrix staat voor de output van de bovenste neurons in laag $[\ell]$. De tweede rij in de matrix staat voor de tweede rij neurons etc. Deze formules werken met een willekeurig aantal training examples. Immers,

$$Z^{[\ell]} = W^{[\ell]}A^{[\ell-1]} + b^{[\ell]} = (n_\ell, n_{\ell-1}) \times (n_{\ell-1}, m) + (n_\ell) = (n_\ell, m)$$

Met n_ℓ het aantal neurons in laag $[\ell]$, $n_{\ell-1}$ het aantal neurons in laag $[\ell-1]$ en m het aantal training examples. Aangezien de activation functie wordt toegepast op elk individueel element in de matrix Z, is de dimensie van $A^{[\ell]}$ hetzelfde als die van $Z^{[\ell]}$. De matrix $A^{[\ell]}$ is na het toepassen van de formules dus ook een matrix met in de verticale richting de outputs van de verschillende neurons in laag $[\ell]$ en in de horizontale richting alle training examples. Om dit concreet te zien volgt er nu een voorbeeld:



laag ($[\ell-2]$) Vorige laag ($[\ell-1]$) huidige laag ($[\ell]$) laag ($[\ell+1]$)

De huidige laag, die zich in een willekeurig neural network bevindt, heeft drie neurons, Zie afbeelding.

$$A^{[\ell-1]} = \begin{bmatrix} 7 & 9 & 3 \\ 1 & 8 & 1 \end{bmatrix}, \qquad W^{[\ell]} = \begin{bmatrix} 0.6 & -0.2 \\ 0.4 & 0.2 \\ -0.4 & 0.6 \end{bmatrix}, \qquad b^{[\ell]} = \begin{bmatrix} 0.2 \\ -1.6 \\ -0.9 \end{bmatrix}$$

Je kan dit ook zien aan de rijen van de bovenstaande weight matrix $W^{[\ell]}$ en de bias vector $b^{[\ell]}$. Er zijn 2 neurons in de vorige laag, zie de afbeelding, rijen van de $A^{[\ell-1]}$ matrix of kolommen van de $W^{[\ell]}$ matrix. Er zijn 3 training examples, zie kolommen van de $A^{[\ell-1]}$ matrix. De output van de drie neurons kunnen we nu als volgt berekenen.

$$Z^{[\ell]} = \begin{bmatrix} 0.6 & -0.2 \\ 0.4 & 0.2 \\ -0.4 & 0.6 \end{bmatrix}\begin{bmatrix} 7 & 9 & 3 \\ 1 & 8 & 1 \end{bmatrix} + \begin{bmatrix} 0.2 \\ -1.6 \\ -0.9 \end{bmatrix} = \begin{bmatrix} 4.2 & 4 & 1.8 \\ 1.4 & 3.6 & -0.2 \\ -3.1 & 0.3 & -1.5 \end{bmatrix}$$

$$A^{[\ell]} = \sigma\left(\begin{bmatrix} 4.2 & 4 & 1.8 \\ 1.4 & 3.6 & -0.2 \\ -3.1 & 0.3 & -1.5 \end{bmatrix}\right) = \begin{bmatrix} 0.99 & 0.98 & 0.86 \\ 0.8 & 0.97 & 0.45 \\ 0.04 & 0.57 & 0.18 \end{bmatrix}$$

De bovenstaande output is een drie bij drie matrix van laag $[\ell]$. Deze output vormt de input van de neurons in laag $[\ell + 1]$. Het kan niet de laatste laag zijn. Als het de laatste laag zou zijn, zou er een ander soort activation functie gebruikt moeten worden. Meer hierover in de volgende paragraaf.

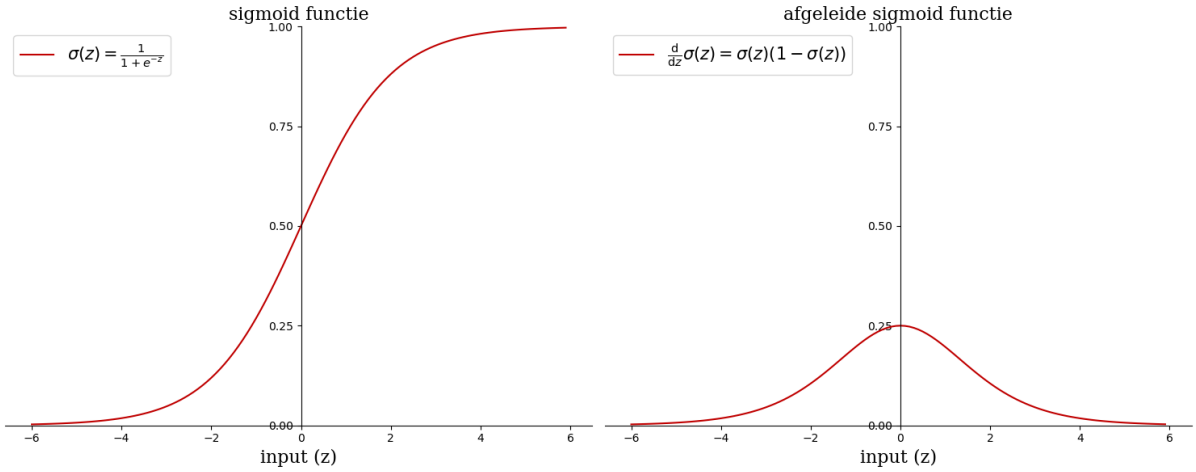
D. ACTIVATION FUNCTIONS

De theorie achter neural networks is al meer dan 50 jaar oud. Neural Networks zijn in de laatste tien jaar pas razend populair geworden. De oude theorie wordt in ieder neural network toegepast. We kunnen nu de eerste hyperparameters bepalen. We beginnen met het type activation functie. Dat wil zeggen hoe een neuron de lineaire input veranderd naar een betekenisvolle output. De literatuur biedt ons een aantal interessante opties.

1. Als eerst is er de reeds bekende sigmoid functie [15]

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{d}{dz}\sigma(z) = \sigma(z)\,(1 - \sigma(z))$$



2. Hyperbolic Tangent (tanh) functie [15]

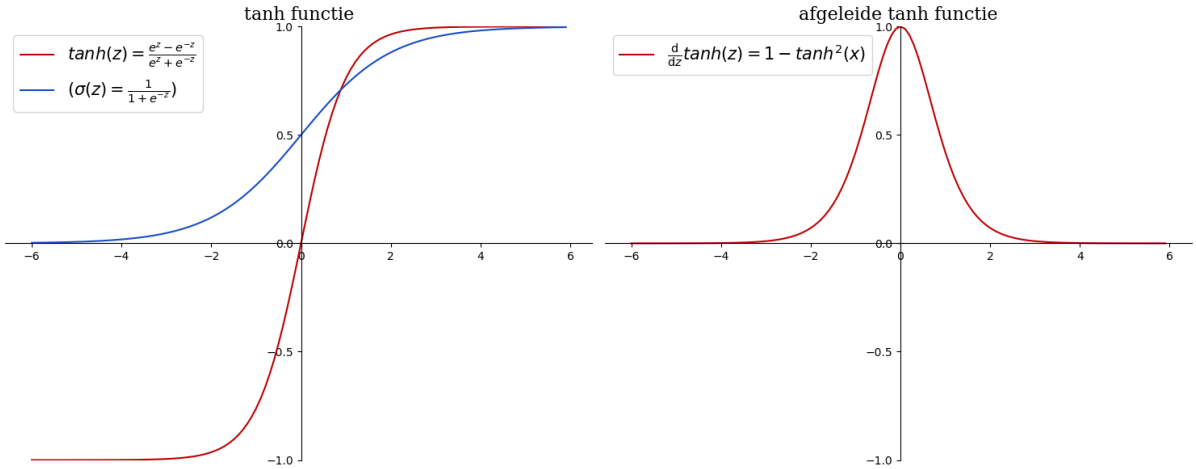
$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{d}{dz}tanh(z) = \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2}$$

$$= \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

$$= 1 - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

$$= 1 - tanh^2(z)$$



De Tanh functie werkt beter dan de sigmoid functie en maakt het leerproces sneller.

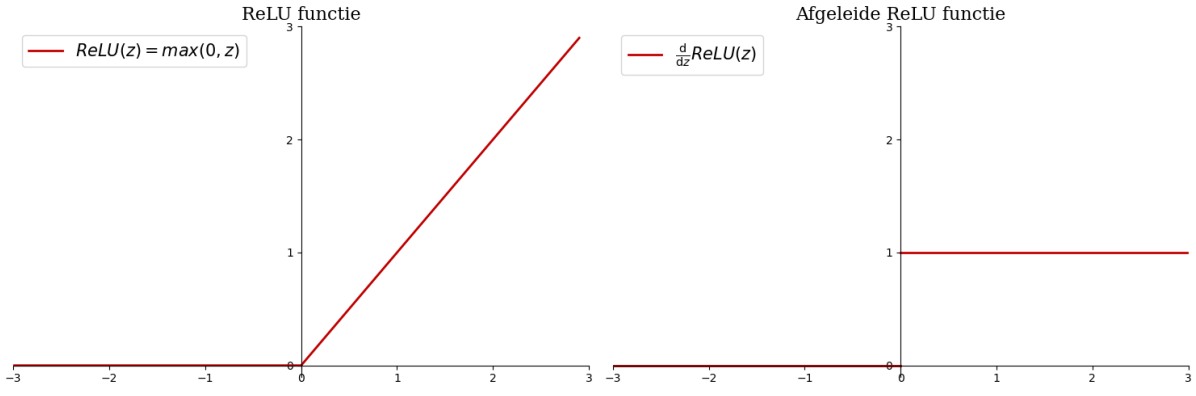
3. ReLU

De Rectified Linear Unit is veruit de beste activation functie voor hidden layers [15]. Deze activation functie wordt toegepast op de lineaire output van alle hidden layers. Het zorgt voor het snelste leerproces en wordt tegenwoordig in bijna alle neural networks gebruikt.

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

$$of\ ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$\frac{d}{dz} ReLU(z) = \begin{cases} 0, & z < 0 \\ 1, & z > 0 \end{cases} \quad hierbij\ is\ x = 0\ ongedefiniërd$$



Andere veel gebuikte varianten van de ReLU activation functie zijn Leaky ReLU en GELU [16] (Gaussian Error Linear Unit).

$$Leaky\ ReLU(z) = max(0, 0.01z)$$

$$of\ Leaky\ ReLU(z) = \begin{cases} 0.01z, & z < 0 \\ z, & z \geq 0 \end{cases}$$

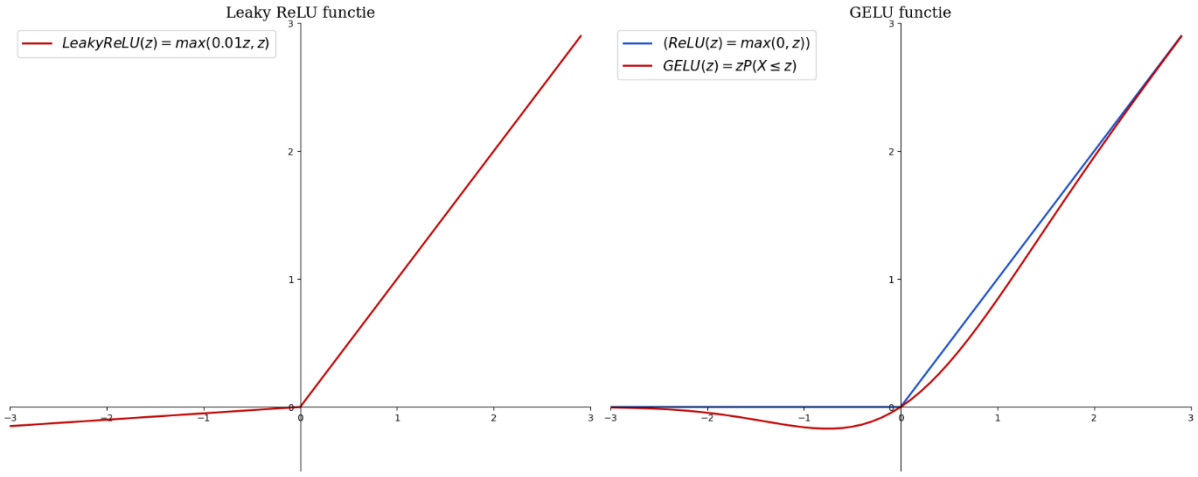
$$GELU(z) = zP(X \leq z) = z\Phi(z)$$

$$met\ \Phi(z)\ =\ de\ standaard\ Gaussverdelingsfunctie$$

$$en\ met\ X \sim \mathcal{N}(0,1)$$

Bovenstaande uitdrukking ziet er erg lastig uit. Gelukkig kan deze functie ook benaderd worden met:

$$GELU(z)\ = \frac{1}{2} z \left(1 + tanh\left[\sqrt{2/\pi}\,(x + 0.044715z^3)\right]\right)$$



4. Softmax [15] [17]

$$softmax(z_i)\ = \frac{e^{z_i}}{\sum_{j=1}^{K} e_j}$$

$$\frac{\partial}{\partial z_i} softmax(z_i)\ = \begin{cases} S_i\,(1\ -\ S_i\,), & i = j \\ -S_i\,S_i\,, & i \neq j \end{cases}$$

$$met\ S_i\ =\ softmax(z_i)$$

Van de softmax activation functie kan voor de meeste gevallen geen plot worden gemaakt. Deze activation functie heeft als input namelijk een vector en als output ook een vector met dezelfde dimensies. Deze activation functie zullen we gebruiken voor een neural network die meerdere klassen kan onderscheiden. Als input geven we de softmax functie namelijk de laatste laag neurons in het neural network voordat er een activation functie op toegepast wordt. Dit is dus de lineaire output Z. Vervolgens berekent de softmax functie voor elke kolom (elke kolom staat voor een training example) in de matrix Z een kansverdeling. Grote waarden krijgen een grotere kans om de juiste klasse in de voorspelling te zijn. Alle waarden in één kolom moeten samen 1 (=100%) zijn. Stel dat we het neural network dat verschillende dieren kan onderscheiden uit het one-hot encoding voorbeeld weer als voorbeeld nemen. We geven dit neural network een afbeelding van een hond. De lineaire output en de one-hot klassen vector van dit training example zijn:

$$Z = \begin{matrix} Kat \\ Hond \\ Vogel \\ Konijn \\ Vis \end{matrix} \begin{bmatrix} 6.47 \\ 12.6 \\ 0.21 \\ 2.32 \\ 0.11 \end{bmatrix}, \quad Y = \begin{matrix} Kat \\ Hond \\ Vogel \\ Konijn \\ Vis \end{matrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Als we de softmax functie op deze vector toepassen krijgen we de output:

$$\hat{Y} = softmax(Z) = softmax\left(\begin{bmatrix} 6.47 \\ 12.6 \\ 0.21 \\ 2.32 \\ 0.11 \end{bmatrix}\right) = \begin{bmatrix} 2.17 \times 10^{-3} \\ 0.978 \\ 4.15 \times 10^{-6} \\ 3.42 \times 10^{-5} \\ 3.76 \times 10^{-6} \end{bmatrix}$$

De voorspelling voor de klasse hond is het grootst, deze heeft het neural network dus goed voorspeld. Omdat de som van de vector 1 is (in het voorbeeld niet helemaal vanwege afronding), kunnen we zeggen dat het neural network 97.8% zeker is dat hij naar een afbeelding van een hond kijkt. Dit lijkt erg op de output van de sigmoid functie, nu kunnen we echter meerdere klassen betrekken. Met meerdere klassen volstaat de cost functie zoals we die bij LG hebben gezien niet meer. Deze werkte alleen als er twee klassen zijn, namelijk y=0 of y=1. In plaats van de binary cross-entropy loss functie gebruiken we nu de categorical cross-entropy loss functie. Deze ziet er als volgt uit:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{K} y_j log(\hat{y}_j)$$
$$met\ K = totaal\ aantal\ klassen$$

De categorical cross-entropy cost functie ziet er dan zo uit:

$$\begin{aligned} \mathcal{J}(W, b) &= \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} {y_j}^{(i)} log\left({\hat{y}_j}^{(i)}\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} Y \odot log\left(\hat{Y}\right) \end{aligned}$$

Het symbool: $\odot$ betekent dat het hier gaat om een element-gewijze multiplicatie. Dit is mogelijk omdat Y dezelfde dimensies heeft als $\hat{Y}$. De laatste uitdrukking van de categorical cost functie heeft weer betrekking op het berekenen van de cost functie met behulp van matrices. De eerder geïntroduceerde one-hot encoding komt nu eindelijk goed van pas. De cost functie van het dieren onderscheidende voorbeeld kunnen we namelijk simpelweg berekenen met:

$$\mathcal{J}(W, b) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K} Y log\left(\hat{Y}\right)$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K}\left[\begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix}\odot log\left(\begin{bmatrix}2.17\times10^{-3}\\0.978\\4.15\times10^{-6}\\3.42\times10^{-5}\\3.76\times10^{-6}\end{bmatrix}\right)\right]$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K}\begin{bmatrix}0\\-0.0222\\0\\0\\0\end{bmatrix}$$

$$= -\frac{1}{1}\times -0.0222 = 0.0222$$

Het aantal strafpunten dat dit neural network krijgt voor zijn voorspelling is dus 0.0222. Dit is een aardig goede score. De voorspelling was immers ook erg goed.

E. BACKPROPAGATION

In deze paragraaf gaan we het algoritme van backpropagation afleiden. Het is weer een zeer wiskundig verhaal, als je dit wil overslaan, kan je gelijk naar het algoritme op de volgende pagina kijken. We moeten nu de afgeleiden berekenen van alle weights van alle lagen. Gelukkig zijn alle lagen in een neural network mooi achter elkaar geschakeld. Met behulp van de kettingregel kunnen we dus alle afgeleiden van achter, naar voren berekenen. We beschouwen een neural network met 3 lagen. Forward propagation berekent de output van het neural network.

Forward Propagation:

	Forward propagation	*Afgeleiden*
Laag 1	$Z^{[1]} = W^{[1]}X + b^{[1]}$ $A^{[1]} = g(Z^{[1]})$	$\frac{\partial Z^{[1]}}{\partial W^{[1]}} = X$ $\frac{\partial Z^{[1]}}{\partial b^{[1]}} = 1$ $\frac{\partial A^{[1]}}{\partial Z^{[1]}} = g'(Z^{[1]})$
Laag 2	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$ $A^{[2]} = g(Z^{[2]})$	$\frac{\partial Z^{[2]}}{\partial A^{[1]}} = W^{[2]}$ $\frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]}$ $\frac{\partial Z^{[2]}}{\partial b^{[2]}} = 1$ $\frac{\partial A^{[2]}}{\partial Z^{[2]}} = g'(Z^{[2]})$
Laag 3	$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$ $A^{[3]} = \sigma(Z^{[3]})$	$\frac{\partial Z^{[3]}}{\partial A^{[2]}} = W^{[3]}$ $\frac{\partial Z^{[3]}}{\partial W^{[3]}} = A^{[2]}$ $\frac{\partial Z^{[3]}}{\partial b^{[3]}} = 1$ $\frac{\partial A^{[3]}}{\partial Z^{[3]}} = A^{[3]}(1 - A^{[3]})$
Loss	$\mathcal{L}(A^{[3]}, Y) = -y\,log(A^{[3]}) - (Y-1)log\,(1 - A^{[3]})$	$\frac{\partial \mathcal{L}}{\partial A^{[3]}} = -\frac{y}{A} + \frac{y-1}{1-A}$

Backpropagation:

	Backpropagation	
Laag 3.	$\frac{\partial \mathcal{L}}{\partial Z^{[3]}} = \frac{\partial \mathcal{L}}{\partial A^{[3]}} \frac{\partial A^{[3]}}{\partial Z^{[3]}}$ $(tussenstap)$	$= \left(A^{[3]} - Y\right)$
	$\frac{\partial \mathcal{L}}{\partial W^{[3]}} = \frac{\partial \mathcal{L}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial W^{[3]}}$	$= \left(A^{[3]} - Y\right) A^{[2]}$
	$\frac{\partial \mathcal{L}}{\partial b^{[3]}} = \frac{\partial \mathcal{L}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial b^{[3]}}$	$= \left(A^{[3]} - Y\right)$
	$\frac{\partial \mathcal{L}}{\partial A^{[2]}} = \frac{\partial \mathcal{L}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial A^{[2]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[3]}} W^{[3]}$
Laag 2.	$\frac{\partial \mathcal{L}}{\partial Z^{[2]}} = \frac{\partial \mathcal{L}}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}}$ $(tussenstap)$	$= \frac{\partial \mathcal{L}}{\partial A^{[2]}} g'(Z^{[2]})$
	$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial W^{[2]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[2]}} A^{[1]}$
	$\frac{\partial \mathcal{L}}{\partial b^{[2]}} = \frac{\partial \mathcal{L}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial b^{[2]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[2]}}$
	$\frac{\partial \mathcal{L}}{\partial A^{[1]}} = \frac{\partial \mathcal{L}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial A^{[1]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[2]}} W^{[2]}$
Laag 1.	$\frac{\partial \mathcal{L}}{\partial Z^{[1]}} = \frac{\partial \mathcal{L}}{\partial A^{[1]}} \frac{\partial A^{[1]}}{\partial Z^{[1]}}$ $(tussenstap)$	$= \frac{\partial \mathcal{L}}{\partial A^{[1]}} g'(Z^{[1]})$
	$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{\partial \mathcal{L}}{\partial Z^{[1]}} \frac{\partial Z^{[1]}}{\partial W^{[1]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[1]}} X$
	$\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{\partial \mathcal{L}}{\partial Z^{[1]}} \frac{\partial Z^{[1]}}{\partial b^{[1]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[1]}}$

In zijn algemeenheid geldt dus voor een laag $[\ell]$ die **niet** de laatste laag is:

Laag $[\boldsymbol{\ell}]$	$\boldsymbol{\frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} = \frac{\partial \mathcal{L}}{\partial A^{[\ell]}} \frac{\partial A^{[\ell]}}{\partial Z^{[\ell]}}}$ $\boldsymbol{(tussenstap)}$	$\boldsymbol{= \frac{\partial \mathcal{L}}{\partial A^{[\ell]}} g'(Z^{[\ell]})}$
	$\frac{\partial \mathcal{L}}{\partial W^{[\ell]}} = \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} \frac{\partial Z^{[\ell]}}{\partial W^{[\ell]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} A^{[\ell-1]}$
	$\frac{\partial \mathcal{L}}{\partial b^{[\ell]}} = \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} \frac{\partial Z^{[\ell]}}{\partial b^{[\ell]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}}$
	$\frac{\partial \mathcal{L}}{\partial A^{[\ell-1]}} = \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} \frac{\partial Z^{[\ell]}}{\partial A^{[\ell-1]}}$	$= \frac{\partial \mathcal{L}}{\partial Z^{[\ell]}} W^{[\ell]}$

$g'(Z^{[\ell]})$ is hierin de afgeleide van de activation functie. zie de vorige paragraaf voor de afgeleiden van de activtion functies.

Algoritme 2: backpropagation

Van i=1 tot max_herhalingen:

#Forwardpropagation:
Van $\boldsymbol{\ell = 1}$ tot $\boldsymbol{\ell = aantal\_lagen}$:

$$\boldsymbol{Z^{[\ell]} = W^{[\ell]} A^{[\ell-1]} + b^{[\ell]}}$$
$$\boldsymbol{A^{[\ell]} = g(Z^{[\ell]})}$$

#Backpropagation:
Van $\boldsymbol{\ell = aantal\_lagen}$ tot $\boldsymbol{\ell = 1}$:

$$\boldsymbol{dZ^{[\ell]} = dA^{[\ell]} * g'\left(Z^{[\ell]}\right)}$$
$$\boldsymbol{dW^{[\ell]} = \frac{1}{m} dZ^{[\ell]} A^{[\ell-1]^T}}$$
$$\boldsymbol{db^{[\ell]} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[\ell]}}$$
$$\boldsymbol{dA^{[\ell-1]} = W^{[\ell]^T} dZ^{[\ell]}}$$

#Update de parameters:
$$\boldsymbol{W^{[\ell]} := W^{[\ell]} - \alpha dW^{[\ell]}}$$
$$\boldsymbol{b^{[\ell]} := b^{[\ell]} - \alpha db^{[\ell]}}$$

VERBETERINGEN

Het model zoals het nu is, leert erg traag en is bijvoorbeeld geneigd om de training data te overfitten. Daarom ga ik in dit hoofdstuk met behulp van de literatuur uitleggen hoe deze problemen tegenwoordig worden opgelost. We bespreken in dit hoofdstuk de deelvraag:

- Hoe maken we het model nog beter/sneller?

Ik duik in dit hoofdstuk niet al te diep in de technische aspecten. Ik zal echter een kort overzicht geven.

❖ Mini-Batch Gradient Descent [18]

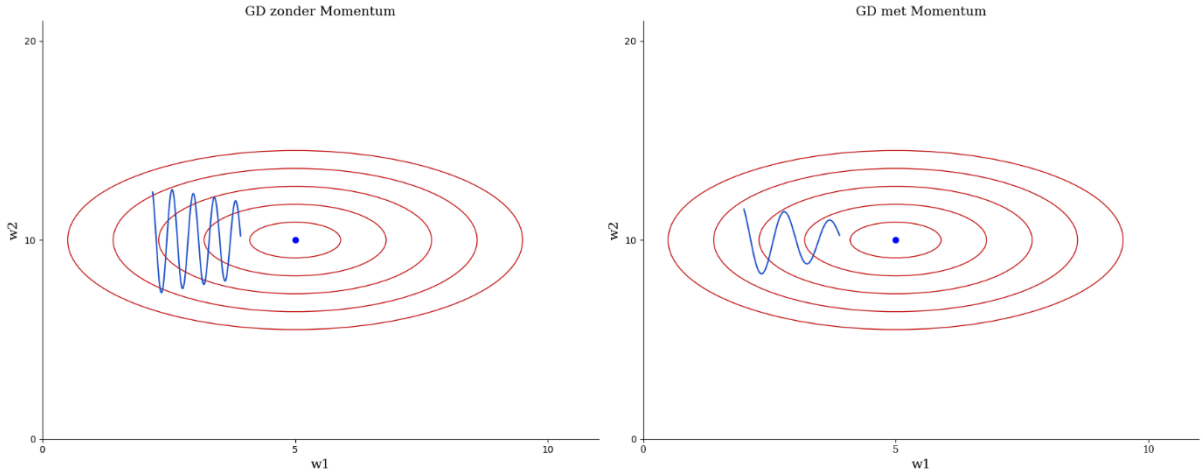
Rekening houdend met het feit dat de meeste machine learning problemen veel data nodig hebben om te trainen en het feit dat in dit big data tijdperk training data in overvloed beschikbaar is, kan je begrijpen dat training sets tegenwoordig uit miljoenen training examples kunnen bestaan. In iedere iteratie (epoch) in ons neural network wordt er gerekend met de hele training set. Het kan vanwege alle matrix vermenigvuldigingen dus erg lang duren voordat het neural network iets nuttigs heeft geleerd. Daarom splitst men de training data tegenwoordig vaak op in zogenaamde mini-batches. Alle mini-batches bevatten hetzelfde aantal training examples (behalve de laatste, deze kan er meer of minder bevatten). Klein detail: de lengte van een mini-batch is vaak een macht van 2 (1,2,4,8,16,32,…) in verband met de RAM structuur in een computer. Gradient Descent wordt dus eerst op elke mini-batch in de training data uitgevoerd voordat de volgende iteratie (epoch) van het leren begint. De vorm van Gradient Descent die we voorheen gebruikten noemt men Batch Gradient Descent. Aan het andere kant van het spectrum bevind zich Stochastic Gradient Descent, hierbij gebruikt men steeds 1 training example per batch. Mini-batch gradient descent is het beste van beide werelden en leert vaak het snelst.

❖ Weight Initialization [19]

Zoals we al eerder besproken hebben beginnen de parameters met willekeurige waarden tussen de -1 en 1. Z (de lineaire output, voordat de activation functie erop wordt toegepast) wordt berekend als een gewogen som met deze parameters. Het aantal termen van de gewogen som wordt bepaald door het aantal neurons in de laag voor de neuron met output Z. Wanneer er in laag $[l-1]$ dus veel neurons zijn, kan Z aardig groot worden. Gradient descent leert echter sneller als Z een bereik aanneemt dat dichter bij de 0 zit. Daarom gebruikt men vaak Kaiming / He Initialization, vernoemd naar een van de auteurs (Kaiming He) van een in 2015 gepubliceerd onderzoek. Om aan willekeurige parameter waarden te komen, gebruikten we voorheen een standaard Gaussian verdeling. He Initialization houdt in dat de parameters een standaard deviatie van $\sqrt{2/n}$ krijgen. Met n = het aantal termen in de som van Z (dit komt dus overeen met: $w_l \sim \mathcal{N}(0,\ 2/n_l)$). Deze parameter initialisatie werkt het beste bij de eerder benoemde ReLU activation functie. Voor de tanh activation functie gebruikt men vaak Xavier initialization. Hierbij wordt een standaard deviatie van $\sqrt{1/n}$ gebruikt.

❖ ADAM [20]

In de afbeelding hieronder is twee keer dezelfde cost functie geplot. De afbeeldingen hieronder zijn contour plots. Dit is een manier om een driedimensionale plot, zoals afbeelding <3d plot> op een tweedimensionaal vlak te projecteren. De buitenste ringen komen overeen met de hoogste waarden van de cost functie. De blauwe punt komt overeen met het laagste deel van de cost functie. Gradient Descent zorgt er nogmaals voor dat we dit dal (globale maximum) bereiken. De linker afbeelding is een plot met het algoritme voor Gradient Descent dat we nu gebruiken. We zien dat het erg lang duurt voordat het dal is bereikt, dit komt omdat er veel schommelingen zijn ten opzichte van w_1 (de x-as). De rechter plot ziet er veel beter uit, er zijn minder schommelingen dus het dal wordt eerder bereikt. Het bereik van ieder punt op de blauwe plot zou in de x richting dus groter moeten worden en in de y richting kleiner, zo komen we namelijk eerder in het dal terecht.



Een van de manieren om dit te doen, is het berekenen van moving averages of exponentieel gewogen gemiddelden. Een exponentieel gewogen gemiddelde (EWMA) houdt rekening met afgeleiden die al t aantal stappen eerder waren berekend. Door het toepassen van een EWMA worden de punten als het ware gladgestreken. Het is voor de computer duur om alle afgeleiden per tijdstap t in het geheugen op te slaan. Daarom wordt een algoritme gebruikt dat een EWMA benaderd. Dit algoritme heeft wel twee nieuwe hulp matrices nodig.

Het algoritme dat we voortaan zullen gebruiken heet ADAptive Moment Estimation (ADAM). Feitelijk bestaat dit algoritme uit twee andere algoritmen, namelijk Momentum en RMSprop. Momentum heb ik hierboven al kort besproken. Momentum houdt nogmaals rekening met de waarden van afgeleiden die t stappen geleden waren berekend. Er wordt vaak een fysische analogie gebruikt om Momentum uit te leggen. Je moet denken aan een bal die van een berg afrolt. Wanneer dit gebeurd wordt de bal geaccelereerd, de snelheid wordt dus steeds groter. We kunnen dus ook zeggen dat de bal steeds meer momentum krijgt. Bij het berekenen van EWMAs werkt Root Mean Square Propagation (RMSprop) ongeveer hetzelfde als Momentum. Zoals de naam al doet vermoeden wordt er gebruik gemaakt van de afgeleiden in het kwadraat. Bij de update regel van GD wordt er bij RMSprop nog gedeeld door de wortel van de bias gecorrigeerde afgeleiden. Om het leren in het begin ook snel te laten verlopen moeten we rekening houden met de hierboven genoemde bias gecorrigeerde afgeleiden. Dit is voor nu echter niet belangrijk. ADAM

gebruikt beide algoritmen om het trainen van een neural network zo snel mogelijk te laten verlopen.

❖ Reguralization [21]

Het neural network dat we nu hebben, met Mini-Batch GD, He Initialization en ADAM, is nog steeds vatbaar voor overfitten. Dit betekent dat het model geneigd is om zo goed mogelijk bij de training data te passen. Dit is echter niet de bedoeling. Bij metingen in het echte leven zijn er altijd datapunten die door mensen onjuist geclassificeerd zijn. Het model moet deze uitschieters zien te ontwijken en zo goed mogelijk proberen te generaliseren. Zou aan ons neural network dus een nieuw datapunt gepresenteerd worden, moet het deze goed juist kunnen classificeren. Hieronder zie je drie plots van drie neural network die ik getraind heb. De training data bestaat uit rode- en blauwe puntjes, ze behoren respectievelijk tot de klassen y = 1 en tot y = 0. Er zijn in totaal 200 punten geplot (zowel rode als blauwe) dus m = 200. Verder heeft ieder puntje 2 features (namelijk de x-coördinaat en de y-coördinaat). Het rode gebied zijn de features waarvoor het neural network denkt dat de klasse y = 1 is. In het blauwe gebied voorspelt het neural network y = 0. De decision boundary is de lijn die het rode gebied van het blauwe gebied scheidt.

We zien dat de meest linker plot te simpel is. Het neural network in deze plot kan de training data namelijk maar voor 86.5% juist classificeren. Men zegt ook wel dat dit neural network lijdt aan te hoge Bias. Het neural network in de meest rechter plot is veel te complex. Hoewel het de training data 99.0% juist classificeert, zou het nooit goed kunnen generaliseren op data die het neural network nog niet gezien heeft. Dit neural network lijdt aan te hoge Variance. Het neural network in de middelste plot is simpel en net complex genoeg. Het heeft door, dat de blauwe punten in het rode gebied en de rode punten in het blauwe gebied, verkeerd geclassificeerd zijn door mensen. Hoewel het dus voor 91.0% bij de training data past, zou het veel beter generaliseren en test data juist classificeren. Het middelste neural network heeft dus beter kunnen generaliseren dan de twee neural networks aan de linker en rechter kant.

Het verminderen van variance wordt gedaan met behulp van L_2 reguralization (weight decay). Door nog een term aan de cost function toe te voegen (en dus de afgeleide van deze term aan de afgeleiden van de parameters), kunnen we ervoor zorgen dat parameters gestraft worden als ze te groot worden. Als een neuron veel parameters heeft die vrijwel 0 zijn, heeft dit als effect dat de output van dit neuron nauwelijks impact heeft op het geheel van het

neural network. Een neural network met minder neurons is simpler en produceert dus ook een simpelere decision boundary.

De mensen die toch geïnteresseerd zijn in de technische strekking van de verbeteringen, kunnen de volgende algoritmen bestuderen en/of de werkelijke publicaties bekijken.

ADAM:

Algoritme 3: ADAM

#Initialiseer hulpmatrices ADAM
$\boldsymbol{M_0 = 0}$ (eerste hulpmatrix)
$\boldsymbol{V_0 = 0}$ (tweede hulpmatrix)
 # Initialiseer de tijd stap
$\boldsymbol{t = 0}$

Van i=1 tot max_epochs:

Van j = 1 tot max_aantal_minibatches:

Algoritme Forwardpropagation (Zie algoritme 2: Backpropagation)

Algoritme Backpropagation (Zie algoritme 2: Backpropagation)

#verhoog de tijdstap met 1
$\boldsymbol{t = t + 1}$

Van $\boldsymbol{\ell = 1}$ tot $\boldsymbol{\ell = aantal\_lagen}$:

$\boldsymbol{M_t := \beta_1 M_{t-1} + (1 - \beta_1) dPar^{[\ell]}}$ (dPar = afgeleiden parameters)
$\boldsymbol{V_t := \beta_2 V_{t-1} + (1 - \beta_2) dPar^{[\ell]^2}}$ (dPar = afgeleiden parameters)

$\boldsymbol{\widehat{M_t} := \frac{M_t}{(1-\beta_1{}^t)}}$ (eerste bias gecorrigeerde hulpmatrix)
$\boldsymbol{\widehat{V_t} := \frac{M_t}{(1-\beta_2{}^t)}}$ (tweede bias gecorrigeerde hulpmatrix)

#Update de parameters
$\boldsymbol{Par := Par^{[\ell]} - \alpha \frac{\widehat{M_t}}{\sqrt{\widehat{V_t}}+\varepsilon}}$ ($\boldsymbol{\varepsilon = 10^{-8}}$ delen door 0 onmogelijk)

https://arxiv.org/pdf/1412.6980.pdf

L2 Reguralization:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{K} y_j log(\hat{y}_j) \, (categorical\ cross - entropy\ loss)$$

$$\left\|W^{[\ell]}\right\|_F^2 = \sum_{i=1}^{n^{[\ell]}} \sum_{j=1}^{n^{[\ell-1]}} \left(W_{ij}{}^{[\ell]}\right)^2 \ (Frobenius\ norm\ van\ laag\ [\ell])$$

$$\mathcal{J}(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m}\sum_{\ell=1}^{L}\left\|W^{[\ell]}\right\|_F^2$$

Met een andere cost functie zijn de afgeleiden van de parameters natuurlijk ook anders. Gelukkig wordt de regularization term bij de normale categorical cross-entropy toegevoegd. Geen lastige toestanden met differentiëren dus.

$$\frac{\partial \mathcal{J}}{\partial W^{[\ell]}} = \frac{\partial \mathcal{J}}{\partial Z^{[\ell]}} A^{[\ell-1]} + \frac{\lambda}{m} W^{[\ell]}$$

We gebruiken geen L2 regularization voor de bias parameters.

We hebben dit hoofdstuk kennis gemaakt met een aantal nieuwe hyperparameters. In de volgende paragraaf gaan we kijken hoe we alle hyperparameters optimaal kunnen aanpassen.

https://proceedings.neurips.cc/paper/1991/file/8eefcfdf5990e441f0fb6f3fad709e21-Paper.pdf

TESTEN HYPERPARAMETERS MODEL

Hyperparameters zijn nogmaals parameters die het model in zijn geheel beïnvloeden. De waarden van de hyperparameters kunnen we zelf kiezen. Samen met de in dit hoofdstuk behandelde hyperparameters hebben we nu de volgende hyperparameters die we kunnen aanpassen:

1. α, de learning rate of leersnelheid;
2. De activation functies voor iedere laag;
3. De lengte van de mini-batches;
4. He/Xavier initialization;
5. β_1, eerste hyperparameter ADAM;
6. β_2, tweede hyperparameter ADAM;
7. λ, regularization parameter.

Bij ieder probleem waar neural networks bij betrokken zijn, worden zogenoemde development datasets gebruikt om verschillende waarden hyperparameters op te testen. Een klein deel van de MNIST dataset is uit de training set gehaald. Deze training examples zal een getraind neural network dus nog niet hebben gezien. Hier kunnen we dus verschillende waarden van de hyperparameters op testen. Naar de hyperparameters 2., 4., 5. en 6. is in de literatuur al veel onderzoek gedaan. Daarom beschouwt iedere machine learning onderzoeker de volgende waarden als standaard.

2. Activation functie hidden layers: ReLU, activation functie laatste laag: ligt aan het probleem, bij classificatie probleem meestal softmax.

4. Laag met ReLU: He initialization, laag met tanh: Xavier initialization.

5. $\beta_1 = 0.9$

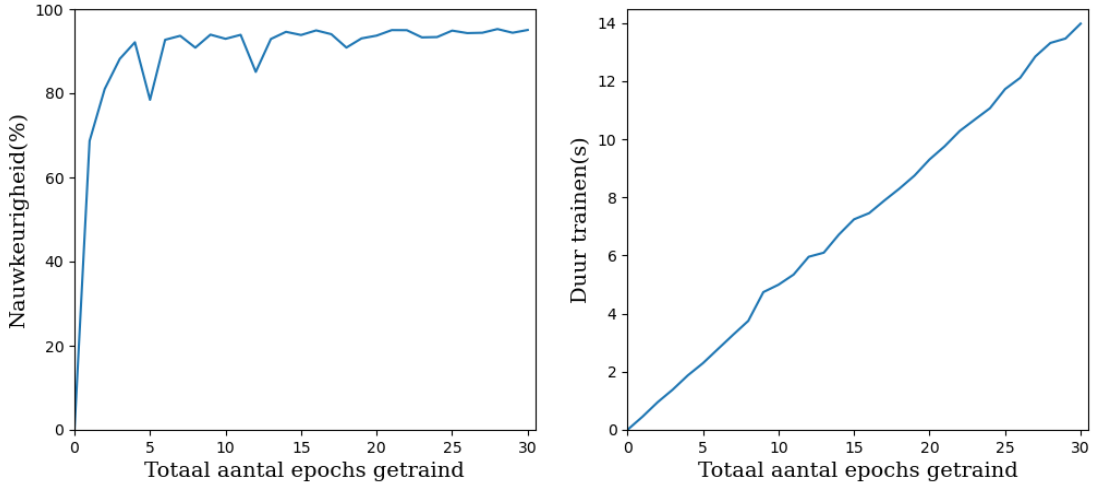
6. $\beta_2 = 0.999$

Ik zal deze waarden dus ook als standaard gebruiken. Als laatste kunnen we natuurlijk ook bepalen hoelang we het neural network laten trainen. Hoe meer stapjes we immers zetten, hoe dichter we bij het dal komen.

❖ Aantal epochs (iteraties)

Om de onderstaande afbeelding te maken heb ik 30 neural networks getraind. Het eerste neural network heeft 1 epoch geleerd, de tweede twee epochs etc. tot het 30^e neural network. Na het trainen heb ik de voorspellingsgraad van elk neural network op de Dev/Test set bepaald. Je ziet dat er wat schommelingen in de lijn van de linker plot zitten. Dit komt omdat de beginwaarden van parameters voor ieder neural network willekeurig zijn gekozen. Dit betekent dat sommige neural networks al dichterbij het dal waren en dus een voorsprong hadden op de andere neural networks. Verder zie je dat de linker plot steeds langzamer stijgt. Het zal dus heel lang duren voordat het model de 100% nadert. Sterker nog, het model zal waarschijnlijk nooit de 100% behalen. Dit komt omdat we nog steeds een te simpel neural network gebruiken. Zoals in een van de eerste hoofdstukken al zei, gebruikt men tegenwoordig Convolutional Neural Networks om afbeeldingen te classificeren. Deze zijn

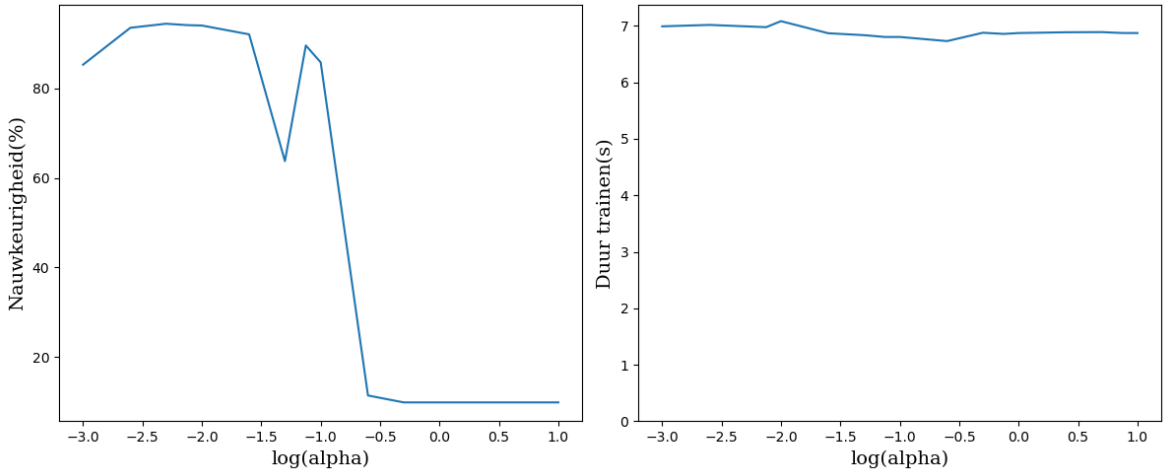
namelijk veel beter in het herkennen van patronen. Bij het cijfer “9” kunnen CNN’s bijvoorbeeld filters hebben die het ronde stuk van de negen herkennen en het verticale stokje. Ons neural network kan niet zulke onderscheidingen maken. Verder heb ik in de rechter plot de tijd uitgezet die het duurde voordat elk neural network klaar was met trainen. We zien dat er een evenredig verband bestaat tussen het aantal epochs en de trainingstijd. Per vijf epochs meer trainen, duurt het trainen ongeveer 2 seconden meer. Met linear regression zouden we aan de hand van de plot de leersnelheid precies kunnen bepalen :). Als laatste moet er wel rekening gehouden worden met de snelheid van de computer. Mijn processor (CPU) is vrij snel. Verder heb ik mijn computer zo ingesteld dat berekeningen ook op de videokaart (GPU) gedaan kunnen worden. Deze is over het algemeen sneller dan de CPU. Op een andere computer kunnen deze tijden dus verschillen.



α = 0.0075 | Lengte minibatch= 2048 | $\beta_1 = 0.9$ | $\beta_2 = 0.999$ | $\lambda = 0.7$ | Aantal epochs = Variabel

De learning rate (α)

De learning rate bepaald nogmaals hoe snel ons model leert. Als we een te kleine waarde voor α nemen, duurt het leren erg lang. Als we een te groote waarde voor α nemen, lopen we het risico dat we het dal voorbijschieten. De waarden voor alpha die ik heb onderzocht komen uit de rij {0.001, 0.0025, 0.0075, 0.001, …, 1, 2.5, 5, 7.5, 10}. Om de waarden mooi op een plot te krijgen heb ik een logaritmische schaal gebruikt. De grafiek daalt vanaf $10^{-1.0}$ dus vanaf een waarde van $\alpha = 0.1$ erg snel naar beneden. Vanaf dit punt zijn de stappen dus te groot en schieten we het dal voorbij. De nauwkeurigheid is ongeveer 10%. Er zijn 10 klassen, dit betekent dus dat het model net zo goed aan het gokken zou kunnen zijn. In de linker plot is er een top te zien. Deze komt ongeveer overeen met $log_{10}(alpha) = -2.3$. De waarde voor alpha waar we met hetzelfde aantal epochs de hoogste nauwkeurigheid kunnen behalen is dus: $10^{-2.3} \approx 0.005$. Deze waarde zullen we dus voortaan gebruiken.



α = variabel | Lengte minibatch= 2048 | $\beta_1 = 0.9$ | $\beta_2 = 0.999$ | $\lambda = 0.7$ | Aantal epochs = 15

❖ Lengte van de minibatches

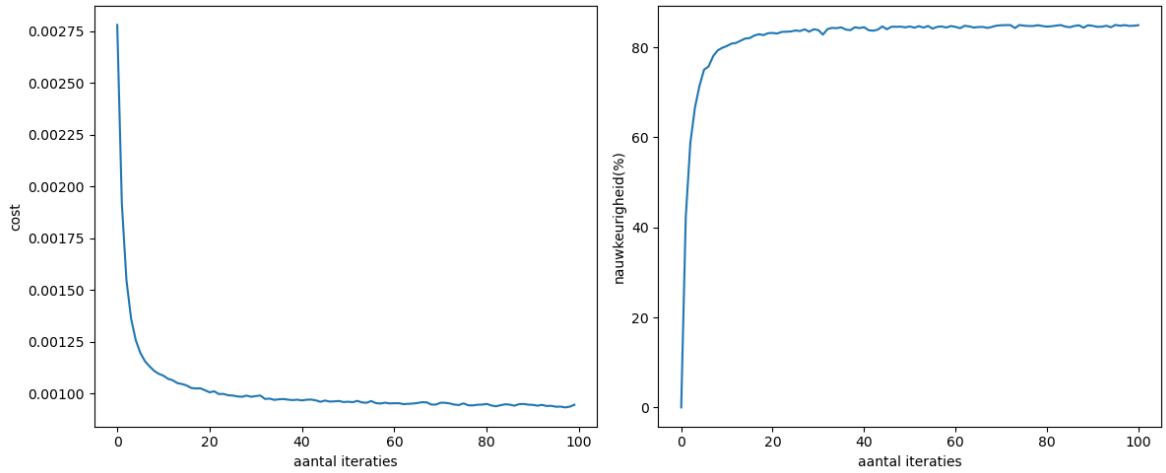
Bij GD met behulp van minibatches verdeelden we de hele training set in kleinere subsets, de minibatches. Dit is een proces van uitproberen. GD met kleine minibatch lengten duurt namelijk heel erg lang. De eerste en beste lengte die ik gevonden heb is $2^{11} = 2048$. Hier was het algoritme snel genoeg en had binnen een aantal epochs al een hoge nauwkeurigheid weten te behalen.

α = 0.005 | Lengte minibatch= variabel | $\beta_1 = 0.9$ | $\beta_2 = 0.999$ | $\lambda = 0.7$ | Aantal epochs = 15

❖ Configuratie lagen

Er is nog één onderdeel dat ik niet besproken heb. Dat is de configuratie van de hidden layers in het neural network. Dit zijn dus het aantal lagen en het aantal neurons per hidden layer. Ik heb drie keer een ander soort configuratie geprobeerd. Als eerste hebben we een neural network met een oplopend aantal neurons per laag, vervolgens een neural network met neurons die per laag eerst oplopen en daarna weer aflopen. Als laatste hebben we een neural network met een aflopend aantal neurons per laag.

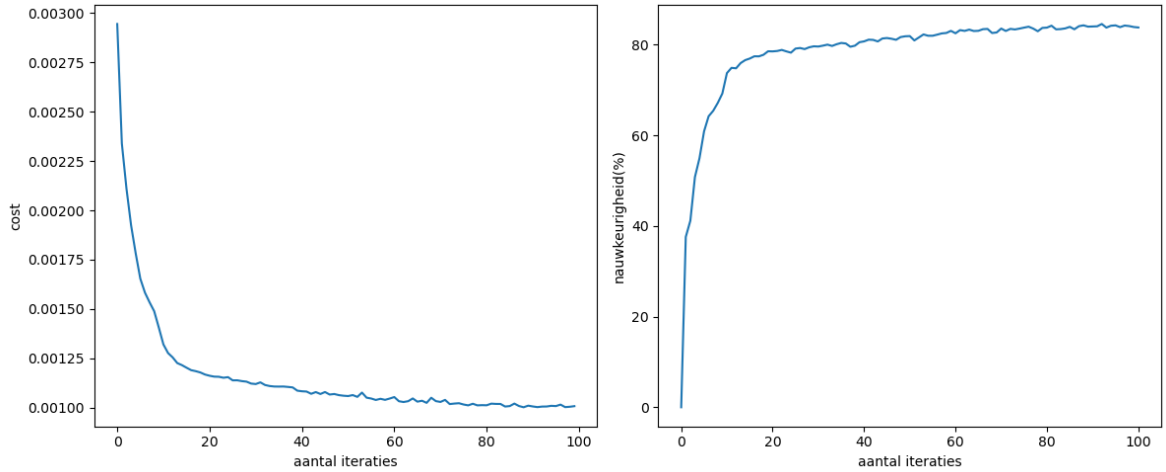
Configuratie hiddel layers: 5 – 10 – 15 – 20.



Nauwkeurigheid test set: 84.05%

α = 0.0075 | Lengte minibatch= 2048 | $\beta_1 = 0.9$ | $\beta_2 = 0.999$ | $\lambda = 0.3$ | Aantal epochs = 100

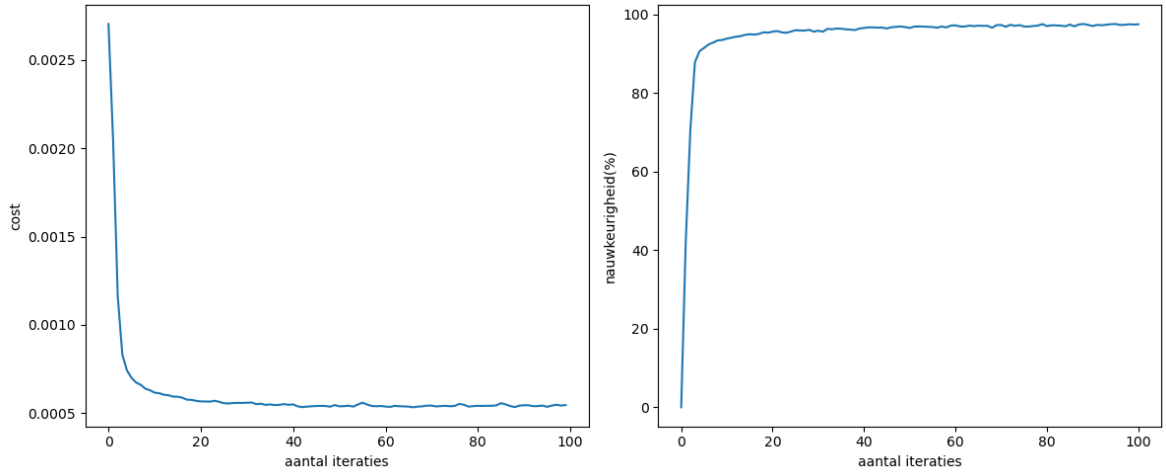
Configuratie hiddel layers: 5 – 10 – 15 – 20–20–15–10–5.



Nauwkeurigheid test set: 82.68%

α = 0.0075 | Lengte minibatch= 2048 | $\beta_1 = 0.9$ | $\beta_2 = 0.999$ | $\lambda = 0.3$ | Aantal epochs = 100

Configuratie hiddel layers: 20–15–10–5.



Nauwkeurigheid test set: 95.27%

α = 0.0075 | Lengte minibatch= 2048 | $\beta_1 = 0.9$ | $\beta_2 = 0.999$ | $\lambda = 0.3$ | Aantal epochs = 100

Er is een duidelijke winnaar, het neural network met de configuratie 20–15–10–5 heeft de hoogste nauwkeurigheid op de testset behaald. Opmerkelijk is dat het neural network met de configuratie 5 – 10 – 15 – 20–20–15–10–5 een lagere nauwkeurigheid heeft behaald ondanks het feit dat dit neural network 4 lagen meer heeft dan de rest van de neural networks. Blijkbaar werkt een neural network met een aflopende configuratie het beste.

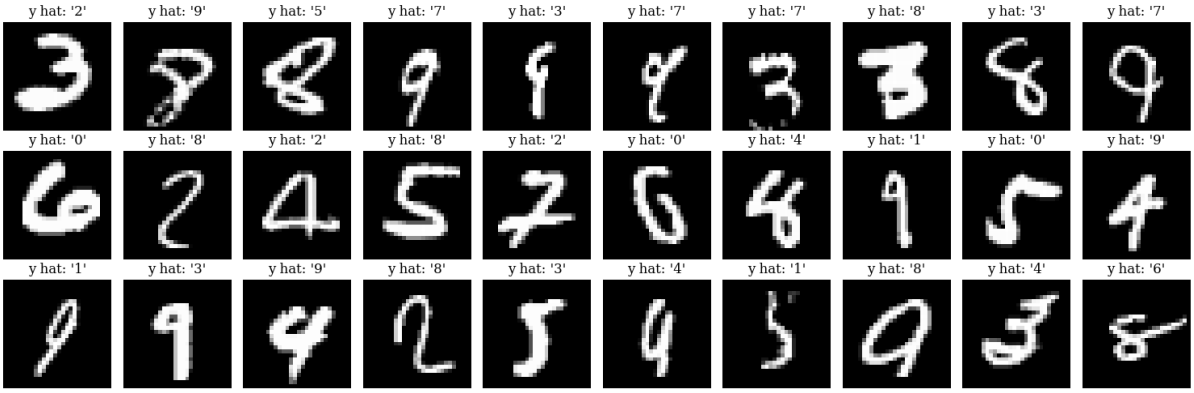
We zijn nu aan het einde van mijn PWS gekomen. Als uitsmijter proberen we de ultieme vraag te beantwoorden:

- Is een neural network beter in het herkennen van handgeschreven cijfers dat een mens?

Het antwoord op deze vraag is: Het ligt aan u. In de test set zitten 10000 training examples. Het getrainde neural network kon een nauwkeurigheid van 95.19% op deze testset behalen. Hier onder heb ik de eerste 30 van de 481 handgeschreven cijfers geplot die het neural network niet juist kon classificeren. In de plot zelf staat na “y hat” de voorspelling van de computer. Na de

afbeelding staat een tabel met de juiste labels. Is het u gelukt om ze allemaal juist te classificeren? Zo ja, gelukkig, de wereld is nog van ons. Voor nu zijn we nog beter in het herkennen van cijfers dan neural networks.

Voorspellingen neural network:



Echte labels:

3	8	8	9	9	9	3	3	8	9
6	2	4	5	7	6	8	9	5	4
9	9	4	2	5	9	5	9	3	8

AFKORTINGEN EN NOTATIE

Afkortingen

AI: Artificial Intelligence (in het nederlands: Kunstmatige Intelligentie (KI))

NN: Neural Network

ANN: Artificial Neural Network

DNN: Deep Neural Network

CNN: Convolutional Neural Network

RNN: Recurrent Neural Network

GAN: Generative Adversarial Network

LSTM: Long Short-Term Memory

LG: Logistic regression

GD: Gradient descent

ADAM: Adaptive Moment Estimation

Vertalingen meest voorkomende Engelse woorden

Training example: Een voorbeeld waar een NN op leert.

Feature: Kenmerk van een training example.

Supervised learning: de klassen (labels) van de data zijn bekend.

Unsupervised learning: de klassen (labels) van de data zijn onbekend.

Algemeen

$x^{(i)}$

- (i) staat voor het i^{ste} voorbeeld.

$x^{[l]}$

- $[l]$ staat voor de l$^{\text{ste}}$ layer.

Formaat notatie

- m : Het aantal voorbeelden in de dataset.
- n_x : Aantal units (neurons) in de 1$^{\text{e}}$ (input) layer.
- n_y : Aantal units in de laatste (output) layer.
- $n_h^{[l]}$: Het aantal hidden units in de l^{ste} laag.

Objecten neural networks

- $X \in \mathbb{R}^{n_x \times m}$: Matrix met alle input voorbeelden.
- $x^{(i)} \in \mathbb{R}^{n_x}$: Het i^{ste} voorbeeld in de vorm van een kolom vector.
- $Y \in \mathbb{R}^{n_y \times m}$: Matrix met alle juiste klassen.
- $y^{(i)} \in \mathbb{R}^{n_y}$: De output voor de i^{ste} voorbeeld.
- $W^{[l]} \in \mathbb{R}^{aantal\ units\ volgende\ layer \times aantal\ units\ vorige\ layer}$: Matrix met alle weights (parameters).
- $b^{[l]} \in \mathbb{R}^{aantal\ units\ in\ de\ volgende\ layer}$: bias vector in de l^{ste} layer.
- $\hat{y} \in \mathbb{R}^{n_y}$: Vector met alle voorspellingen

BIJLAGE 1: CODE NEURAL NETWORK

```
#Importeer libraries
import matplotlib.pyplot as plt
import numpy as np
from sklearn.utils import shuffle
import sklearn.datasets
import tensorflow as tf
import pickle

#De MNIST dataset bestaat uit handgescreven getallen van 0 tot 9
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#Bereidt de MNIST data voor
def maak_data(features, labels):

        #De onderstaande regels mixen de de training/test data
        # en transformeert ze in de juiste dimensies
        features_gemixt, labels_gemixt = shuffle(features, labels)
        features_gemixt = features_gemixt.reshape(features_gemixt.shape[0], -1).T
        features_gemixt = features_gemixt / 255

        return features_gemixt, labels_gemixt

#Maakt mini batches uit alle training examples
def maak_mini_batches(X, Y, len_mini_batch = 64):
        #m = aantal training examples
        m = Y.shape[1]
        #Lijst om mini_batches in op te slaan
        mini_batches = []

        #Omdat het mogelijk is dat de training data gestructureerd is,
        #moeten we het eerst husselen
        verschuiving = list(np.random.permutation(m))
        X_gehusseld = X[:, verschuiving]
        Y_gehusseld = Y[:, verschuiving]

        #Bereken hoeveel hele mini_batches er gemaakt kunnen worden
        hele_mini_batches = int(np.floor(m/len_mini_batch))

        #Deel de data op in mini batches
        for i in range(hele_mini_batches):
                #neemt alle rijen en slice
                X_mini_batch = X_gehusseld[:, len_mini_batch*i:len_mini_batch*(i+1)]
                Y_mini_batch = Y_gehusseld[:, len_mini_batch*i:len_mini_batch*(i+1)]
                #Zip is maakt een object van 2 iterables
                mini_batches.append((X_mini_batch, Y_mini_batch))

        #Als len_mini_batch niet een geheel aantal keer in m past,
        #moet de rest ook een mini batch worden
        #Bug wanneer overige batch = 0
        if not m % hele_mini_batches:
                #Hoeveel training examples blijven er over?
                m_geheel = len_mini_batch * hele_mini_batches

                X_mini_batch = X_gehusseld[:, m_geheel:]
                Y_mini_batch = Y_gehusseld[:, m_geheel:]

                mini_batches.append((X_mini_batch, Y_mini_batch))

        return mini_batches
```

```
#Verandert een vector met integers naar een onehot matrix
def maak_onehot(labels):
        #m = aantal training examples, n = klassen
        m = labels.shape[1]
        n = np.max(labels) + 1

        Y = np.zeros((n, m))
        Y[labels, np.arange(m)] = 1

        return Y

#Maakt de parameter matrices aan en zet ze in een dict
def maak_parameters(dimensies_lagen):
        #L = totaal aantal lagen
        parameters = {}
        L = len(dimensies_lagen)

        for l in range(1, L):
                #W parameters hebben waarden tussen <~-1, ~1>,
                #He Initialisation (Var = 2/n[l-1])
                parameters[f'W{l}'] = np.random.randn(dimensies_lagen[l],
                                        dimensies_lagen[l-1]) *
float(np.sqrt(2/dimensies_lagen[l-1]))
                parameters[f'b{l}'] = np.zeros((dimensies_lagen[l], 1))

        return parameters

#Maakt matrices om weighted averages voor ADAM bij te houden
def maak_ADAM_matrices(parameters):

        L = len(parameters) // 2
        m = {}
        v = {}

        for l in range(1, L+1):
                #Alle matrices hebben in elke rij en kolom eerst een 0
                m[f'dW{l}'] = np.zeros_like(parameters[f'W{l}'])
                m[f'db{l}'] = np.zeros_like(parameters[f'b{l}'])
                v[f'dW{l}'] = np.zeros_like(parameters[f'W{l}'])
                v[f'db{l}'] = np.zeros_like(parameters[f'b{l}'])

        return m, v

#Sigmoid activation functie
def sigmoid(x):

        a = 1 / (1 + np.exp(-x))

        return a

#Afgeleide sigmoid activaiton functie
def dsigmoid(x):  #input is Z
        a = sigmoid(x)
        da = a * (1 - a)

        return da

#Relu activation functie
def relu(x):

        a = np.maximum(0, x)

        return a

#Afgeleide Relu activation functie
def drelu(x):     #input is Z
```

```
        #*1 om boolean values naar integers te veranderen
        da = (x > 0) * 1

        return da

#Softmax activation functie
def softmax(x):
        #constante om NaN's te voorkomen, np.max(x) deelt namelijk weg
    exp_x = np.exp(x - np.max(x))

    a = exp_x / np.sum(exp_x, axis=0, keepdims=True)

    return a

#Het forward propagation algoritme
def forward_propagation(X, parameters):
        #De dictionary geheugen slaat de gegevens op die
        #bij backpropagation nodig zijn
        geheugen = {}

        A = X
        L = len(parameters) // 2

        #Loopt over alle lagen
        for l in range(1, L+1):
                vorige_A = A

                W = parameters[f'W{l}']
                b = parameters[f'b{l}']

                #Z[l] = W[l]A[l-1] + b[l]
                Z = np.dot(W, vorige_A) + b

                #Als we in de laatste laag zijn,
                #gebruiken we de softmax activation functie i.p.v. Relu
                if l == L:
                        #A = sigmoid(Z)
                        A = softmax(Z)
                else:
                        A = relu(Z)

                geheugen[f'A{l-1}'] = vorige_A
                geheugen[f'W{l}'] = W
                geheugen[f'b{l}'] = b
                geheugen[f'Z{l}'] = Z

        return A, geheugen

#Het backward propagation algoritme
def backward_propagation(Y_hat, Y, geheugen, num_lagen, lambd):
        #De dictionary afgeleiden slaat alle afgeleiden op
        afgeleiden = {}
        #L = aantal lagen, m = aantal training examples
        L = num_lagen
        m = Y.shape[1]

        #alleen bij sigmoid
        #dLdA = -(np.divide(Y, Y_hat) - np.divide(1-Y, 1-Y_hat))

        for l in reversed(range(1, L+1)):
                #In de laatste laag is de afgeleide van de softmax activation functie
                if l == L:
                        #alleen bij sigmoid
                        #dAdZ = dsigmoid(geheugen[f'Z{l}'])

                        #Y_hat - Y is zowel de afgeleide van L tot Z via sigmoid,
```

```
                        # als via softmax
                        dLdZ = Y_hat - Y
                else:
                        #dL/dZ[l] = dL/dA[l]*g'(Z[l])
                        dAdZ = drelu(geheugen[f'Z{l}'])
                        dLdZ = afgeleiden[f'dA{l}']*dAdZ

                #dL/dW[l] = dL/dZ[l]*A[l-1]
                dLdW = (1/m)*np.dot(dLdZ, geheugen[f'A{l-1}'].T)+
                                (lambd/m)*geheugen[f'W{l}']
                dLdb = (1/m)*np.sum(dLdZ, axis=1, keepdims=True)

                #dL/dA[l-1] = dL/dZ[l]*W[l]
                dLdA = np.dot(geheugen[f'W{l}'].T, dLdZ)

                #afgeleiden worden in de dictionary opgeslagen
                afgeleiden[f'dW{l}'] = dLdW
                afgeleiden[f'db{l}'] = dLdb
                afgeleiden[f'dA{l-1}'] = dLdA

        return afgeleiden

#Berekent de cost
def bereken_cost(A, Y, parameters, lambd):

        m = Y.shape[1]

        #Cost softmax, np.mean berekent het gemmiddelde =
        #1/m * sum(<alle elementen matrix>)
        cross_entropy_cost = -1*np.mean(Y*np.log(A+1e-8))

        #Frobenius Norm
        frobenius_norm = [np.sum(np.square(parameters[f'W{l}']))
                          for l in range(1,len(parameters)//2+1)]

        #totale cost
        cost = np.squeeze(cross_entropy_cost+(lambd/(2*m))*np.sum(frobenius_norm))

        return cost

#Updates de parameters, gradient descent
def ADAM(parameters, afgeleiden, learning_rate, m, v, t, beta1=0.9, beta2=0.999,
                epsilon=1e-8):
        '''
        x = HINT ADAM
        x - 602 = (Locke + Reyes + Ford + Jarrah + Shephard * Kwon)
        '''
        L = len(parameters) // 2
        m_bias_gecorrigeerd = {}
        v_bias_gecorrigeerd = {}

        for l in range(1, L+1):

                m[f'dW{l}'] = beta1*m[f'dW{l}']+(1-beta1)*afgeleiden[f'dW{l}']
                m[f'db{l}'] = beta1*m[f'db{l}']+(1-beta1)*afgeleiden[f'db{l}']

                v[f'dW{l}'] = beta2*v[f'dW{l}']+(1-beta2)*np.square(afgeleiden[f'dW{l}'])
                v[f'db{l}'] = beta2*v[f'db{l}']+(1-beta2)*np.square(afgeleiden[f'db{l}'])

                m_bias_gecorrigeerd[f'dW{l}'] = m[f'dW{l}'] / (1-beta1**t)
                m_bias_gecorrigeerd[f'db{l}'] = m[f'db{l}'] / (1-beta1**t)

                v_bias_gecorrigeerd[f'dW{l}'] = v[f'dW{l}'] / (1-beta2**t)
                v_bias_gecorrigeerd[f'db{l}'] = v[f'db{l}'] / (1-beta2**t)

                parameters[f'W{l}'] -= learning_rate*m_bias_gecorrigeerd[f'dW{l}']/
```

```
                    (np.sqrt(v_bias_gecorrigeerd[f'dW{l}'])+epsilon)
                    parameters[f'b{l}'] -= learning_rate*m_bias_gecorrigeerd[f'db{l}']/

          (np.sqrt(v_bias_gecorrigeerd[f'db{l}'])+epsilon)

          return parameters, m, v

#Voospelt de labels van test data
def voorspel(parameters, X, Y):
          #Eerst de voorspellingen uit fp halen
          Y_hat, _ = forward_propagation(X, parameters)

          #De klasse met de grootste waarde wordt voor elke training example een 1,
          #de rest 0
          voorspellingen = np.zeros_like(Y_hat)
          voorspellingen[Y_hat.argmax(0), np.arange(Y_hat.shape[1])] = 1

          #Bereken de nauwkeurigheid (aantal goed geraden voorbeelden/
          #totaal aantal voorbeelden)
          nauwkeurigheid = float((np.sum(np.sum(abs(Y*voorspellingen)))/
                              float(X.shape[1])*100))

          return voorspellingen, nauwkeurigheid

#Traint alle parameters van het neural network (fit functie)
def train_neural_network(X, Y, dimensies_lagen, learning_rate=0.0075,
                              aantal_epochs = 1000, cost_interval=100, lambd=0.3,
                              len_mini_batch = 2048, beta1=0.9, beta2=0.999, epsilon=1e-8):

          costs = []
          nauwkeurigheden = []
          #Aantal training examples
          m_train = Y.shape[1]
          #Begin tijd, voor ADAM
          t = 0
          #Maak de parameter matrices aan
          parameters = maak_parameters(dimensies_lagen)
          #Maak de matrices aan die nodig zijn voor ADAM
          m, v = maak_ADAM_matrices(parameters)

          #Voer gradient descent <aantal_epochs> aantal keer uit
          for i in range(0, aantal_epochs):

                    mini_batches = maak_mini_batches(X, Y, len_mini_batch)
                    epoch_cost = 0

                    for mini_batch in mini_batches:
                              #Haal minibatches uit de tuple
                              X_mini_batch, Y_mini_batch = mini_batch
                              #forward propagation
                              Y_hat, geheugen = forward_propagation(X_mini_batch, parameters)
                              #bereken de cost van deze mini batch
                              epoch_cost += bereken_cost(Y_hat, Y_mini_batch, parameters,
                              lambd)
                              #backward propagation
                              afgeleiden = backward_propagation(Y_hat, Y_mini_batch,
                              geheugen, len(dimensies_lagen)-1, lambd)
                              #Verhoog de tijd voor ADAM
                              t += 1
                              #update parameters
                              parameters, m, v = ADAM(parameters, afgeleiden, learning_rate,
                                                       m, v, t, beta1, beta2, epsilon)

                    #Bereken gemmiddelde cost van deze epoch
                    gemmiddelde_cost = epoch_cost/(m_train/len(mini_batches))
```

```
                #sla per <cost_interval> aantal iteraties de cost en nauwkeurigheid op
                if not i % cost_interval:
                        #bereken de cost
                        costs.append(gemmiddelde_cost)

                        #bereken nauwkeurigheid
                        _, nauwkeurigheid = voorspel(parameters, X, Y)
                        nauwkeurigheden.append(nauwkeurigheid)

                        print(f"Nauwkeurigheid en cost na epoch {i}: ", end='')
                        print(f"{round(nauwkeurigheid, 3)}%, {round(gemmiddelde_cost,
                        10)}")

        return parameters, costs, nauwkeurigheden

#Plot de decision boundary
def plot_decision_boundary(X, Y, parameters, axes):
    #Minimale en maximale dimensies plot (+opvulling)
    x_min, x_max = X[0, :].min() - 0.5, X[0, :].max() + 0.5
    y_min, y_max = X[1, :].min() - 0.5, X[1, :].max() + 0.5
    #Stapgrootte
    h = 0.01

        #Punten op plot die geclassificeerd moeten worden
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

        #Voorspel de label van de punten (xx, yy)
    Z, _ = voorspel(parameters, np.c_[xx.ravel(), yy.ravel()].T, 0)
    Z = Z.argmax(0)
    Z = Z.reshape(xx.shape)
    print(Z.shape)

        #Plot punten + contour
    axes.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.3)
    axes.set_xlabel('x1')
    axes.set_ylabel('x2')
    axes.scatter(X[0, :], X[1, :], marker='.', s=150, c=Y, linewidth=0.75,
                alpha=0.95, edgecolor='black', cmap=plt.cm.Spectral)


def fout_geclassificeerd(X, Y, voorspellingen, ax):
        #Goede voorspellingen
        goed = (Y*voorspellingen).max(0)
        #Indexes foute voorspellingen
        fout = np.where(goed==0)[0]
        #plot foute voorspelling
        ax.imshow()


x_train, y_train = maak_data(x_train, y_train)
x_test, y_test = maak_data(x_test, y_test)
y_train = maak_onehot(y_train.reshape(1,len(y_train)))
y_test = maak_onehot(y_test.reshape(1,len(y_test)))

lagen_config = [x_train.shape[0], 20, 15, 10, 5, y_train.shape[0]]
parameters, costs, nauwkeurigheden = train_neural_network(x_train, y_train,
                                            lagen_config, cost_interval=1, lambd=0.3,
                                            aantal_epochs=100)

voorspellingen, nauwkeurigheid = voorspel(parameters, x_test, y_test)

print (f'Nauwkeurigheid: {nauwkeurigheid}%')
```

LOGBOEK

DATUM	TIJD	VERRICHTE WERKZAAMHEDEN	OPMERKINGEN
15-07-2021	5.0 uur	Mediatheek opdracht	
23-07-2021	5.5 uur	Brainstormen over onderzoek, uiteindelijk eindproduct en geschreven versie PWS.	
Zomer vakantie	10 uur	Voordat het hele PWS proces was begonnen had ik al onderzoek gedaan naar AI en de wiskundige implementatie ervan. In de zomervakantie heb ik alles opgefrist.	Afgesproken is dat de uren besteed aan onderzoek vóór het PWS begon ook meegeteld mochten worden.
10-10-2021	2.5 uur	Stappenplan Logistic Regression, begin Datasets, matrices en vectoren, begin Notatie	Afbeeldingen tijdelijk in paint gemaakt
11-10-2021	2.0 uur	Begin voorspellingsfunctie, begin Loss Function / Costfunction	
12-10-2021	1.5 uur	Verder Datasets, matrices en vectoren	
15-10-2021	1.0 uur	Begin Gradient Descent	
16-10-2021	0.5 uur	Begin afleiding cost function, vectorization	
18-10-2021	0.5 uur	Verder Vectorization, uitleg python	
30-10-2021	1.0 uur	Begin Neural Networks	
31-10-2021	0.5 uur	Uitleg neurons	Weer tijdelijke schetsen in paint
1-11-2021	2.0 uur	One-hot encoding, matrices neural networks, Forward propagation, Activation functions	
2-11-2021	1.0 uur	Uitwerken afgeleiden parameters logistic regression	
3-11-2021	0.5 uur	Weer verder uitwerken afgeleiden en algoritme GD	
	1.5 uur	Uitwerken afgeleiden en algoritme Neural Networks, dus eerst Forward Prop en dan Backprop	
4-11-2021	5.0 uur	Werken aan LG in code. Eindproduct is een geprogrammeerd algoritme.	Logistic regression is niet ingeleverd als eindproduct.
5-11-2021	-	Meneer Schoorl gevraagd of hij kon helpen met uitwerking afgeleiden LG/NN	SE-week nadert, leren voor toetsen dus minder tijd PWS
6-11-2021	5.5 uur	Werken aan NN in code. LG was een goede basis hiervoor (algoritme Backpropagation). Basis algoritme af, maakt voorspellingen op gegenereerde data. Voorspellingen handgeschreven cijfers nog implementeren.	Het algoritme voor NN is wel als eindproduct ingeleverd en is als bijlage aan het PWS toegevoegd.
22-11-2021	1.5 uur	Datasets, matrices en vectoren verschillende formules uitwerken. Begin maken afbeeldingen met Matplotlib in Python	Plots zijn geprogrammeerd
23-11-2021	2.0 uur	GD plots met Matplotlib maken en in PWS uitleggen. Ook 3d plot gemaakt met Matplotlib.	
	1.0 uur	Vervangen tijdelijke afbeeldingen Stappenplan Logistic Regression met plots gemaakt met Matplotlib	

29-11-2021	3.5 uur	Uitleg Neurons gevisualiseerd met afbeeldingen van: http://alexlenail.me/NN-SVG/index.html, afbeeldingen alle activation functies gemaakt	Rest klas 6 naar Maastricht, ik in isolatie vanwege mogelijke blootstelling covid :(
2-11-2021	2.0 uur	Begin Wat is AI? Bron onderzoek gebruiken voor uitleg over ANNs, DNNs	Tevens versie die is ingeleverd voor beoordeling 2
7-12-2021	1.0 uur	Bron onderzoek gebruiken voor uitleg over CNNs, RNNs en GANs	
9-12-2021	0.5 uur	Begin Verbeteringen	
13-12-2021	5.0 uur	Bron onderzoek gebruiken voor uitleg over Minibatches, Weight Initialization, ADAM en Reguralization. Afbeelding ADAM gemaakt met Matplotlib	
14-12-2021	2.0 uur	Uitwerken verscheidene voorbeelden voor LG, Verder uitleg Python en afbeelding complexere datapunten. Programma Neural Network is klaar	Morgen gesprek meneer Kramer
15-12-2021	-	Gesprek meneer Kramer, nog veel te doen in verband met duidelijker uitleggen en simpeler maken.	Nog veel te doen, komende dagen harder werken
16-12-2021	2.0 uur	Begin uitwerken voorbeeld slagen van de middelbare school, trainen neural networks en maken plots	
17-12-2021	3.0 uur	Aanpassen/versimpelen hoofdstuk LG, voorspellingsfunctie, cost functie en GD. Onderzoek hyperparameters	
18-12-2021	3.0 uur	Uitwerken voorbeeld Logic gates (XNOR), toevoegen voorbeeld slagen middelbare school aan LG, Uitleg Tensorflow, Keras of Scikit-learn	
20-12-2021	5.0 uur	Begin Inleiding, uitleg training data handgeschreven cijfers, voorbeeld Forward Propagation. Layout verder uitwerken en aesthetische elementen toevoegen, bronnen bijwerken	
21-12-2021	8.0 uur	Laatste dingen verbeterd/toegevoegd, onderzoek hyperparameters op papier zetten. Uitwerking ADAM/ regularization voor geïnteresseerden.	10:00 inleveren, hele avond wakker gebleven om alle laatste dingen toe te voegen en te verbeteren.
21-12-2021	-	Definitieve versie PWS ingeleverd.	
TOTAAL:	85 uur		

BRONNEN

Visualisaties neural networks zijn gemaakt met:

http://alexlenail.me/NN-SVG/index.html

Verder heb ik alles wat ik heb geleerd over neural networks te danken aan Andrew Ng met zijn geweldige cursussen op Coursera.

https://www.coursera.org/

[1] -, „Basic Questions,” (2007-11-12). [Online]. Available: http://www-formal.stanford.edu/jmc/whatisai/node1.html. [Geopend 2021].

[2] F. Rosenblatt, „The Perceptron—a perceiving and recognizing automaton,” Cornell Aeronautical Laboratory., 1957. [Online]. [Geopend 2021].

[3] „Convolutional Neural Networks (CNNs / ConvNets),” [Online]. Available: https://cs231n.github.io/convolutional-networks/. [Geopend 2021].

[4] H. N. e. al., „Animal Recognition and Identification with Deep Convolutional Neural Networks for Automated Wildlife Monitoring,” IEEE International Conference on Data Science and Advanced Analytics (DSAA), 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8259762. [Geopend 2021].

[5] W. Koehrsen, „Recurrent Neural Networks by Example in Python,” (5 11 2018). [Online]. Available: https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470. [Geopend 2021].

[6] D. Mishra, „Applications of Recurrent Neural Networks (RNNs),” [Online]. Available: https://iq.opengenus.org/applications-of-rnn/. [Geopend 2021].

[7] „Natural Language Processing (NLP) What it is and why it matters,” [Online]. Available: https://www.sas.com/en_us/insights/analytics/what-is-natural-language-processing-nlp.html. [Geopend 2021].

[8] „Understanding LSTM Networks,” (27-08-2015). [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Geopend 2021].

[I. J. G. e. al., „Generative Adversarial Nets,” arXiv:1406.2661 [stat.ML], (10-06-2014).
9 [Online]. Available: https://arxiv.org/pdf/1406.2661.pdf. [Geopend 2021].
]

[J. Brownlee, „A Gentle Introduction to Generative Adversarial Networks (GANs),” (17-
1 06-2019). [Online]. Available: https://machinelearningmastery.com/what-are-generative-
0 adversarial-networks-gans/. [Geopend 2021].
]

[„Zero Sum Game (and Non Zero Sum),” Corporate Finance Institute, [Online]. Available:
1 https://corporatefinanceinstitute.com/resources/knowledge/economics/zero-sum-game-
1 non-zero-sum/. [Geopend 2021].
]

[T. S. e. al., „“Deep Fakes” using Generative Adversarial Networks (GAN),” (2018).
1 [Online]. Available: http://noiselab.ucsd.edu/ECE228_2018/Reports/Report16.pdf.
2 [Geopend 2021].
]

[„De creatieve mens,” Lezing en gesprek met neurowetenschapper Roshan Cools en
1 socioloog Kim van Broekhoven, (21-05-2018). [Online]. Available:
3 https://www.ru.nl/radboudreflects/terugblik/terugblik-2021/terugblik-2021/21-05-18-
] creatieve-mens-lezing-gesprek/. [Geopend 2021].

[M. Kumar, Quantum, Icon books Ltd., 2008.
1
4
]

[C. E. N. e. al., „Activation Functions: Comparison of Trends in Practice and Research for
1 Deep Learning,” arXiv:1811.03378v1 [cs.LG], (8-11-2018). [Online]. Available:
5 https://arxiv.org/pdf/1811.03378.pdf. [Geopend 2021].
]

[A. A. Latif, „On the GELU Activation Function,” (11-04-2019). [Online]. Available:
1 https://alaaalatif.github.io/2019-04-11-gelu/. [Geopend 2021].
6
]

[S. Oman, „The Softmax Function Derivative (Part 1),” (17-06-2019). [Online]. Available:
1 https://aimatters.wordpress.com/2019/06/17/the-softmax-function-derivative/. [Geopend
7 2021].
]

[S. Ruder, „An overview of gradient descent optimization algorithms,” arXiv:1609.04747v2
1 [cs.LG], (15-06-2017). [Online]. Available: https://arxiv.org/pdf/1609.04747.pdf.
8 [Geopend 2021].
]

[19] K. He, „Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” arXiv:1502.01852v1 [cs.CV], (06-02-2015). [Online]. Available: https://arxiv.org/pdf/1502.01852.pdf. [Geopend 2021].

[20] D. P. Kingma, „ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION,” arXiv:1412.6980v9 [cs.LG], (30-01-2017). [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf. [Geopend 2021].

[21] A. Krogh, „A Simple Weight Decay Can Improve Generalization,” (1992). [Online]. Available: https://proceedings.neurips.cc/paper/1991/file/8eefcfdf5990e441f0fb6f3fad709e21-Paper.pdf. [Geopend 2021].

[22] I. Jahan, „STOCK PRICE PREDICTION USING RECURRENT NEURAL,” (2018). [Online]. Available: https://library.ndsu.edu/ir/bitstream/handle/10365/28797/Stock%20Price%20Prediction%20Using%20Recurrent%20Neural%20Networks.pdf?sequence=1&isAllowed=y#:~:text=In%20this%20work%2C%20the%20closing,with%20the%20true%20closing%20price.. [Geopend 2021].

Creation of Adam:

https://commons.wikimedia.org/wiki/File:Creation_of_Adam_(Michelangelo)_Detail.jpg

0100001101101111011011100110011101110010011000010111010001110101
0101101100011000001011101000110100101101111011011100111001100101
1100001000000011110010110111011101010010000001101000011000010 1
1101100110010100100000011001000110010101100011011010010111000 0
0110100001100101011100100110010101100100000100000011101000110 10
0001100101001000000110011001101001011100100111001101110100001 0
0000011100000011000010111001001110100001000000110111101100110 00
1000000111010001101000011010010111001100100000011011010110010 1
0111001101110011011000010110011101100101001011100010000001000 1
0001100101011000110110100101110000011010000011001010111001000 10
0000011101000110100001100101001000000110011001101111011011000 1
1011000110111101110111011010010110111001100111001000000111010 0
0110111100100000011101010110111001100011011011101110111001100 1
0101110010001000000111010001101000011000101001000000110111001 10
0101011110000111010000100000011100000011000010111001001110100 00
1000000110111101100110001000000111010001101000011001010010000 0
01101101 01100101
01110011 01110011
01100001 01100111
01100101 00111010
00100000 00111100
01100110 11001011
00010010 00011001
00001011 01100000
10010100 00110011
10110110 00010111
01110001 10111001
11001101 11101110
00011110 01110011
01101110 11110011
11010100 11011010
10010101 01101000
11011110 01000110
10001111 10110001

1110101100010000001101111100110000000010001101111011001100110 11
0110000111100111011011110101000000001100101100011001001011101 0
0100010000010011110100011101000001110011110101111001000011100 10
0110101110010000000100001110010010101011010111000001000110000 1
1001101000010000000011101111101101100111001101010001110110100 00
1010000011001010000001001100110101011000010111001001001001000 10
1010000101001100100111111010001101000101010010010000000010110 110
1011001100101101000100010011111011001011001101110011001010100 100
1001000010101000101010101010010010000000000100111100110001010 10
0100111110001000000100001101101100011101010110010100100000001 0
0011001100010011101000100000010100100100101001000100010010100 1
0001000101011101001100010110000010010000101010101001100101001 000
0100001101001101010001100101010101010101100100101001010010010 010
1100101110001000000100001101101100011101010110010100100000001 0
0011001100100011101000100000001010000101100001011000001011000 0
1000000101100001011000001010010010110