Zelflerende Systemen

Steven Bronsveld en Thijs van Loenhout

13 november 2017

Inhoud

1 Inleiding

Al lange tijd interesseerden wij ons in onderwerpen als *computers*, *programmeren* en ook *AI*, maar de laatste bracht nog erg veel vragen met zich mee. Het leek ons als een mysterieus verschijnsel dat een programma zichzelf kon verbeteren. Onze vragen bleven enige tijd onbeantwoord terwijl we met de alledaagse schooltaken bezig waren, maar toen kwam het profielwerkstuk. We hadden lichte keuzestress over het onderwerp, maar terugkijkend was een AI-gerelateerd onderwerp niet te vermijden. We kregen de kans ons te verdiepen in dit mysterieuze onderwerp en grepen deze vol enthousiasme.

Boven alles wilden wij zelf iets leren van dit verslag en zelf met het onderwerp bezig zijn, in plaats van klakkeloos de informatie over te nemen van het internet. Dit zou uiteindelijk betekenen dat we verschillende programma's zouden schrijven, ter illustratie bij de tekst, om zelf de stof beter te begrijpen of simpelweg omdat het leuk was ermee bezig te zijn. Ook hebben wij twee grote programma's, één voor *Digit Recognition* en één voor het spelen van een spel, geschreven die ons moesten helpen bij het beantwoorden van de vraag die wij ons hebben gesteld: *in welke aspecten verschillen diverse zelflerende computersystemen, ontworpen voor één specifieke taak, van elkaar?*

In het oriëntatieproces liepen wij vrij natuurlijk van de ene vraag op de andere. Het beantwoorden van al deze vragen hebben wij in de eerste 7 hoofdstukken beschreven, in het theoriedeel van dit verslag. De kennis die wij hier hebben opgedaan, zouden we later gebruiken voor de twee grote programma's. Dit staat beschreven in de hoofdstukken 8 en 9.

2 Reguliere zoekalgoritmes

2.1 Inleiding

Voordat we onderzoek kunnen doen naar zelflerende algoritmes moeten we eerst een beeld krijgen van niet-zelflerende algoritmes. Dit noemen we in dit verslag *reguliere algoritmes*. We behandelen in dit hoofdstuk de vraag: *wat zijn voorbeelden van reguliere algoritmes en hoe werken ze?* We nemen twee bekende zoekalgoritmes als voorbeeld.

2.2 Verschillende algoritmes

Computers hebben geen bewustzijn. Om deze reden kunnen ze niet zelf bepalen iets te doen. Waar computers wel in uitblinken, is het uitvoeren van taken die hun zijn opgelegd. Deze taken komen in de vorm van code. Via code kun je computers opdrachten geven, bijvoorbeeld: *bereken 7 * 6*. De boodschap valt echter niet op deze manier over te brengen. Afhankelijk van de taal waarin je programmeert zijn er vaste commando's waar de computer op zal reageren. Naarmate de opdracht die je een computer wil laten uitvoeren complexer wordt, zal ook het gebruik van deze commando's ingewikkelder worden. Hier komen algoritmes in het spel. Een algoritme is een stappenplan, waarin een complexere handeling in duidelijke opdrachten weergegeven wordt. De volgende definitie legt het woord „algoritme" als volgt uit: *„Een algoritme is een eindige reeks instructies om vanaf een beginpunt een bepaald doel te bereiken."* [1]

Een toegankelijke vergelijking is koken. Er is een **input** van voedsel waar uiteindelijk een gerecht uit moet komen, de **output**. Voor het tot stand komen van dit gerecht gebruik je een recept. Dit recept is als het ware het algoritme. Het aantal mogelijke algoritmes is enorm groot. Niet alleen is het een ruim begrip, ook kan het desbetreffende doel waarschijnlijk op meerdere manieren bereikt worden. Het ene algoritme zal misschien beter zijn dan het andere doordat het bijvoorbeeld efficiënter werkt.

2.2.1 Breadth-first search (BFS)

Dit algoritme, bedacht in de jaren vijftig van de vorige eeuw door E.F. Moore [2], een Amerikaans professor in de wiskunde en computer sciences en een voortrekker in kunstmatig leven, is een zoekalgoritme voor datasets in de vorm van grafieken of „boom"-structuren. In figuur 31 is een dataset, een (grote) verzameling data, te zien. Eén **node**, een item uit de dataset, wordt als oorsprong benoemd, de **root**. Ook wordt een bepaalde uitkomst als doel gesteld. Vervolgens krijgt elke node drie waardes aangewezen:

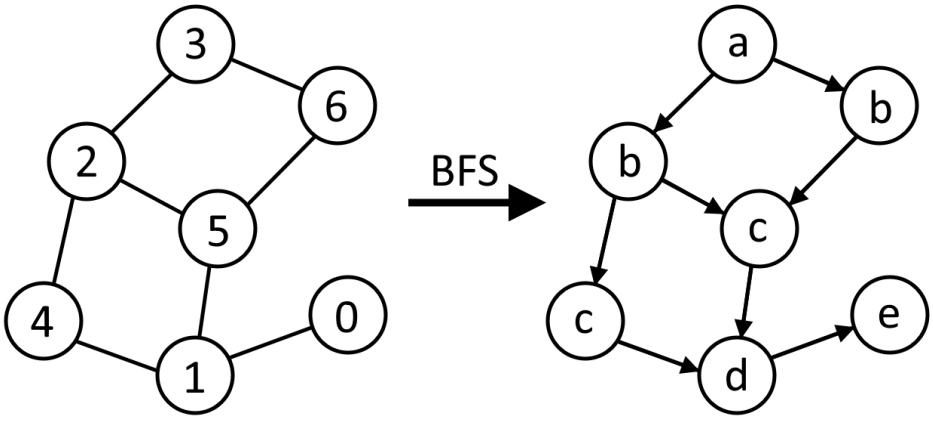
- De afstand van de huidige node naar de root. Dit is het aantal stappen dat gezet moet worden om bij de root te komen.
- De node die vóór de huidige node kwam, de **predecessor**. Anders gezegd: de node waar je uitkomt als je een enkele stap terug zet.
- Een **state**. De state houdt bij of de node al gecontroleerd is.

Bij Breadth-first search wordt gebruik gemaakt van een queue. Dit is een lijst waar nodes toegevoegd en uitgehaald kunnen worden. Net zoals een daadwerkelijke wachtrij wordt het „*eerste erin, als eerste eruit*"-principe toegepast. Het Breadth-first search algoritme ziet er als volgt uit:

1. Maak een lege lijst S voor bezochte nodes.
2. Maak een lege lijst Q met de queue.
3. Benoem één node als root en voeg deze toe aan S.

4. Voeg de root toe aan Q.
5. Zolang Q niet leeg is:
   (a) Haal de voorste node uit de queue. Dit is de **current** node.
   (b) Als current het doel is:
      i. Return current.
   (c) Voor elke node die grenst* aan current:
      i. Als deze node nog niet bezocht is en dus niet in S zit:
         A. Voeg de node toe aan S.
         B. Zeg dat de predecessor van de node de current node is.
         C. Haal de node uit de queue.

**„Grenzen aan" betekent hier: „in directe verbinding staan met"*



Figuur 1: Schematische weergave van een willekeurige dataset, waarop het Breadth-first search algoritme wordt toegepast

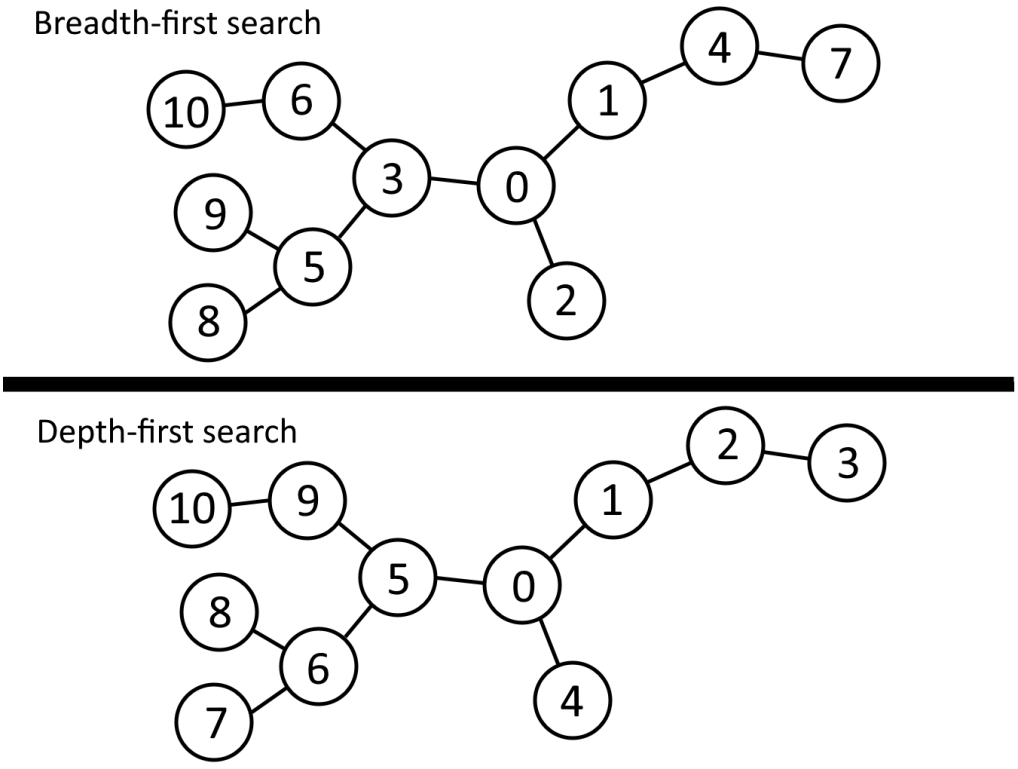
Hierboven is een voorbeeld van een simpele dataset weergegeven (zie figuur 31), genummerd van node 0 tot en met node 6. Node 3 is de root en node 0 het doel. Om bij het doel te komen wordt het Breadth-first search algoritme toegepast. Node 3, de root, wordt toegevoegd aan de lijsten Q en S. Node 3 wordt weer uit de queue gehaald en één voor één worden de aangrenzende nodes bekeken. Hierbij worden ze toegevoegd aan de stack. Omdat zowel node 2 als node 6 het doel niet is, herhaalt het algoritme zich. Nu wordt node 2 bekeken. Het doel is niet gevonden in de aangrenzende nodes. Daarna komt node 6, ook zonder succes. (Let hierbij op dat node 5 niet nogmaals bekeken wordt, dit is namelijk al bij node 2 gedaan en node 5 is dus al aanwezig in lijst S) Intussen zijn node 4 en node 5 toegevoegd aan de queue, ze zijn immers verbonden met node 2. Ook hier wordt het proces herhaald, node 1 zit nu in de queue. Uiteindelijk wordt node 1 bekeken en wordt het doel, node 0, gevonden.

Met BFS kan je zo de weg van de root naar het doel achterhalen. Dit is nuttig als je bijvoorbeeld een wegennetwerk hebt en wil weten wat de kortste weg van de ene naar de andere stad is.

2.2.2 Depth-first search (DFS)

Evenals Breadth-first search is Depth-first search een algoritme voor het doorlopen van datasets in de vorm van grafieken of „boom"-structuren. DFS verschilt echter op twee manieren van BFS:

- Depth-first search gebruikt een stack in plaats van een queue. Waar nodes in een BFS systeem in een wachtrij worden geplaatst met een „als eerst erin, als eerst eruit"-principe, handhaaft een DFS systeem een wachtrij die meer vergelijkbaar is met een stapel papieren. Telkens pak je het bovenste element van de stapel om mee te werken, maar als je iets in de wachtrij stopt, komt dit ook weer bovenop de stapel te liggen. De meest recente toevoeging zal dus als eerste weer eruit gehaald worden.
- Breadth-first search begint bij een root. Vervolgens wordt gekeken naar alle neighbours. Als de gewenste uitkomst niet tussen deze neighbours zit, worden de neighbours van deze neighbours gecontroleerd. Dit proces herhaalt zich totdat het doel gevonden is. Depth-first search begint ook bij een root, maar kijkt direct naar een weg tot een node bereikt is die geen neighbours meer heeft. Als het doel dan niet bereikt is wordt een andere weg geprobeerd. Hiervoor wordt gebruik gemaakt van **recursive backtracking**.



Figuur 2: Schematische weergave van een willekeurige dataset

In figuur 2 is de werking van BFS en DFS weergegeven. Het getal in elke node geeft aan als hoeveelste het bereikt wordt.

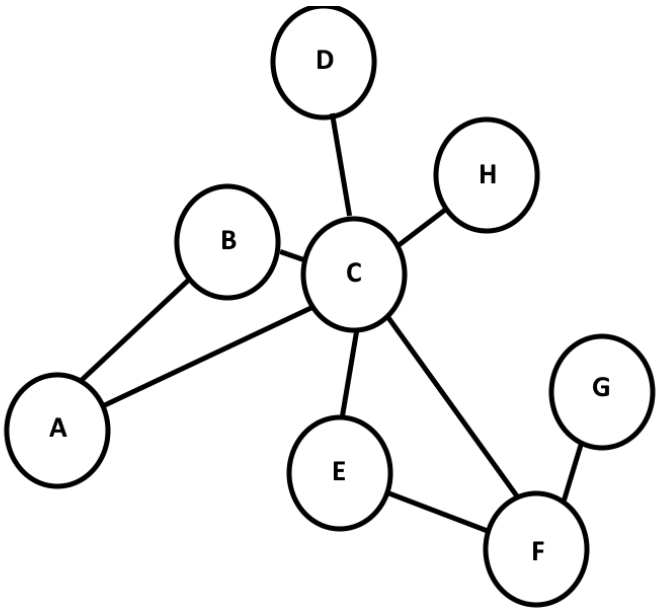
Ook bij DFS hebben de nodes een state: bezocht of niet bezocht. Ten eerste wordt de root gekozen en deze wordt als bezocht opgeslagen. Zoals te zien is, wordt er vanaf de root één (willekeurige) neighbour gekozen om te onderzoeken. Elke bezochte neighbour wordt als bezocht genoteerd. De root wordt in de stack geplaatst. Vanaf deze neighbour wordt weer een nieuwe aanliggende node gekozen, waarvan de state „onbezocht" is. Ook nu wordt de bezochte node in de stack geplaatst. Dit proces herhaalt zich totdat er een node is zonder (onbezochte) neighbours. Op dat moment wordt de bovenste node uit de stack gehaald, dit heet backtracking, en herhaalt het proces zich. Dit blijft doorgaan totdat geen enkele node onbezochte neighbours over heeft of totdat het doel gevonden is.

Als vuistregel kan het volgende gehanteerd worden: Depth-first search wordt gebruikt als je weet dat er maar één uitkomst is, Breadth-first search als je de makkelijkste of snelste uitkomst wil kiezen.

2.3 Praktijk: voorbeelden algoritmes

Algoritmes hebben meestal vele toepassingen. Hier zijn enkele voorbeelden van de eerder genoemde algoritmes.

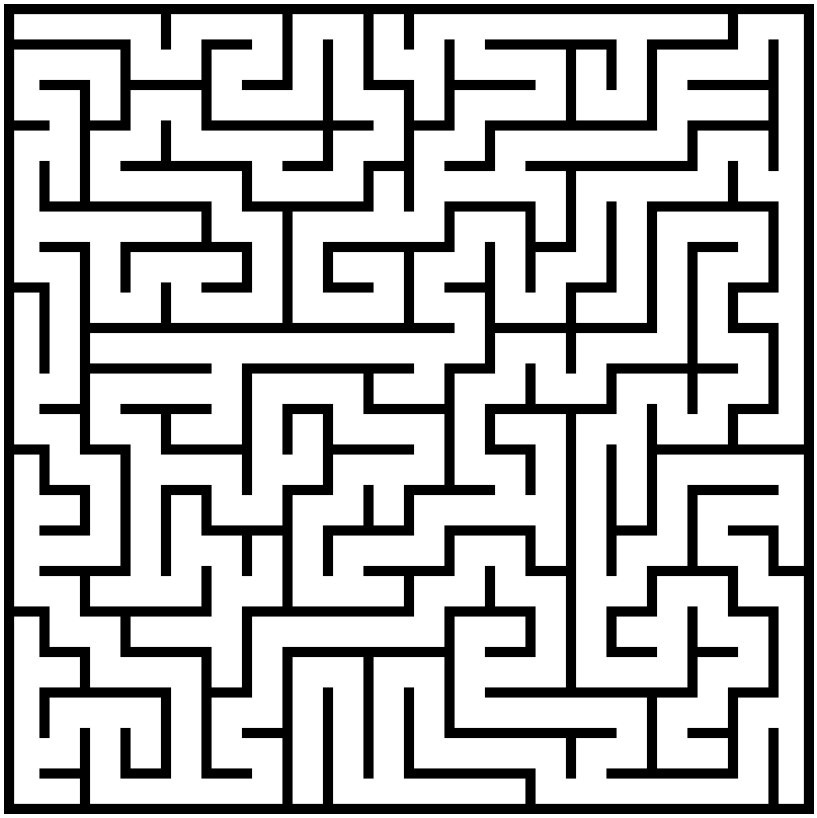
2.3.1 Breadth-first search



Figuur 3: Schematische weergave van een willekeurige dataset

In figuur 3 is een dataset te zien, bijvoorbeeld een telefoonboom. Elke cirkel representeert één persoon. Zo kan persoon A de personen B en C bellen, maar A bezit geen andere telefoonnummers. Toch zou hij een boodschap naar H kunnen sturen: via C. Stel, persoon A wil nu iets tegen F zeggen: in een kleine dataset als deze is het in één oogopslag te zien dat de snelste manier hiervoor A C F is en dat A B C E F veel langer is. Bij grotere datasets is dit echter al snel moeilijk met zekerheid te zeggen. Hiervoor kan Breadth-first search ingezet worden.

2.3.2 Depth-first search



Figuur 4: Een voorbeeld van een automatisch gegenereerd doolhof, gebruik makend van DFS

Depth-first search kan gebruikt worden voor zowel het maken als het oplossen van doolhoven. In figuur 4 is een doolhof te zien dat gemaakt is met behulp van DFS. Wij hebben in het kader van deze deelvraag een doolhof-generator gemaakt. Het algoritme, in de vorm van een stappenplan, is als volgt:

1. Kies een start cel en markeer deze als bezocht. Deze cel noemen we nu de *current* cel.
2. Terwijl er nog niet-bezochte cellen aanwezig zijn:
   (a) Als current neighbours heeft die nog niet bezocht zijn:
      i. Kies willekeurig één van de neighbours.
      ii. Voeg current toe aan de stack.
      iii. Verwijder de muur tussen de huidige cel en de gekozen cel.
      iv. Benoem de gekozen cel als current en zet de state op bezocht.
      v. Kies willekeurig één van de neighbours.
      vi. Voeg current toe aan de stack.
      vii. Verwijder de muur tussen de huidige cel en de gekozen cel.
      viii. Benoem de gekozen cel als current en zet de state op bezocht.
   (b) Anders, als de stack niet leeg is:

i. Haal de laatst toegevoegde cel uit de stack en verwijder deze cel hieruit.
ii. Maak deze cel current.
iii. Haal de laatst toegevoegde cel uit de stack en verwijder deze cel hieruit.
iv. Maak deze cel current.

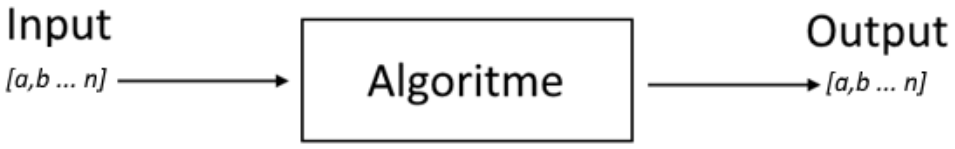
2.4 Conclusie

Twee voorbeelden van reguliere zoek algoritmes zijn „Breadth-first seach" en „Depth-first search". Dit zijn twee algoritmes met vele toepassingen. Beide algoritmes zijn niet zelflerend omdat ze hun manier van zoeken niet zelf verbeteren.

3 Machine learning

3.1 Inleiding

Een zelflerend systeem is een algoritme gebaseerd op machine learning. Machine learning werd door Arthur Samuel, een pionier op dit gebied, gedefinieerd als: *A field of study that gives computers the ability to learn without being explicitly programmed* [3]. In tegenstelling tot de eerder genoemde algoritmes is een zelflerend systeem in staat zichzelf te verbeteren. Hierdoor kan het taken uitvoeren waar reguliere algoritmes tekort schieten. We zullen in dit hoofdstuk de volgende vraag beantwoorden: *wat zijn zelflerende algoritmes en waarin verschillen ze van reguliere algoritmes?*



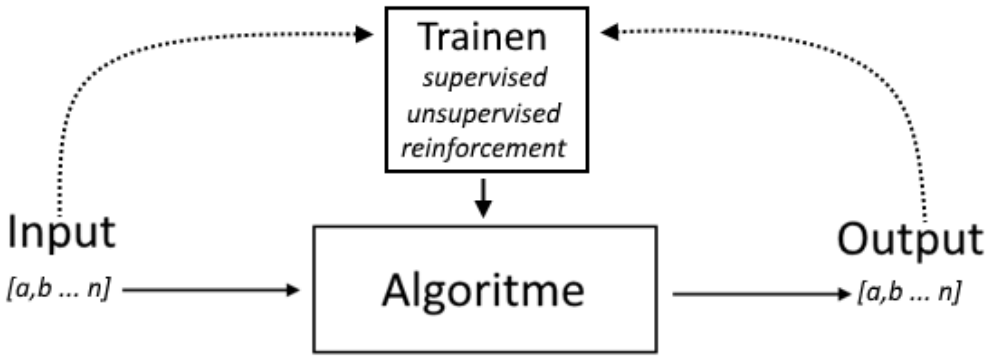
Figuur 5: Schematische weergave van een regulier algoritme

In figuur 5 is een schematische weergave van een regulier algoritme afgebeeld. Bepaalde input data gaat het systeem in en bepaalde output data komt het systeem uit. De input en output data bestaan uit één datatype of uit meerdere datatypen. Als de input simpelweg een reeks getallen betreft, zal dit direct als input gebruikt kunnen worden. In het geval dat de input uit een ander datatype bestaat, zoals een plaatje, zal dit omgezet moeten worden in een reeks getallen voordat het gebruikt kan worden. Het algoritme zal deze getallen bewerken tot de gewenste output. Deze output wordt eveneens in getallen gegeven. Waar nodig zullen deze getallen dus weer moeten worden omgezet tot het gewenste datatype.

Er zijn veel verschillende algoritmes die gebruikt kunnen worden voor een zelflerend systeem. Elk algoritme heeft voor- en nadelen en is geschikt voor andere doeleinden. Een aantal van deze algoritmes zullen we in het volgende hoofdstuk (hoofdstuk 4) bespreken.

3.2 Training

Een zelflerend systeem begint in de meeste gevallen zonder enige kennis van de data. Om de gewenste output te kunnen produceren is het dus nodig om het systeem eerst input data te geven zodat het kan leren. Dit proces wordt het **trainen** genoemd. Voor het trainen van een zelflerend systeem is training data nodig. Deze data moet gelijk of gelijkwaardig zijn aan de echte data. De training data kan in veel verschillende vormen voorkomen en de manier van trainen is afhankelijk van de vorm van de (training) data. In figuur 6 is te zien dat het trainen los staat van het algoritme. Dit verschil zullen we in het volgende hoofdstuk wat duidelijker maken. Er zijn drie prominente manieren waarop een zelflerend systeem kan leren: **supervised**, **unsupervised** en **reinforcement learning**.



Figuur 6: Schematische weergave van een zelflerend systeem

3.2.1 Supervised learning

In het geval van supervised learning heb je te maken met **labeled** training data. Anders gezegd: van een bepaalde input is de gewenste output al bekend. Een klassiek voorbeeld van een labeled dataset is een dataset van huisprijzen en huiseigenschappen (zie figuur 1).

Huisprijs (output)	Huiseigenschappen (input)		
	Woonoppervlakte	Perceeloppervlakte	Aantal Kamers
519.000	124 m^2	311 m^2	4
569.000	133 m^2	309 m^2	5
569.500	170 m^2	310 m^2	6

Tabel 1: Labeled dataset Bron: http://www.funda.nl/koop/huizen/

Bij de training dataset van tabel 1 is de gegeven input de huiseigenschappen en de gewenste output de huisprijs. Het systeem wordt met deze dataset getraind. Hierdoor leert het een output te produceren die steeds dichter bij de gewenste output ligt. Als er een verband bestaat tussen de huiseigenschappen en de huisprijs, wat waarschijnlijk het geval is, zal het zelflerende systeem na genoeg trainen in staat zijn zelf bij nieuwe huiseigenschappen een huisprijs te voorspellen. [4]

3.2.2 Unsupervised learning

Unsupervised learning kan gebruikt worden bij een **unlabeled** dataset, ofwel een dataset waarbij de data niet geclassificeerd is en er geen gewenste output bekend is. Als je een dataset hebt van heel veel niet-geordende foto's is het niet mogelijk om dit te classificeren. Als een deel van de dataset gelabeld wordt, zal met behulp van supervised learning de rest van de dataset geclassificeerd kunnen worden. Dit is echter in veel gevallen niet mogelijk, bijvoorbeeld doordat de dataset enorm groot is of er zodanig veel verschillende groepen bestaan dat het menselijk niet mogelijk is ook maar een procentueel significant deel te labelen. Ook kan het zo zijn dat men niet weet of er een verband aanwezig is. Kortom: unsupervised learning wordt gebruikt voor het classificeren van data, zonder dat er groepen vooraf gedefinieerd zijn. Met behulp van deze vorm van training kunnen in een grote dataset verbanden worden ontdekt, die men misschien niet zonder hulp had kunnen achterhalen. [5]

3.2.3 Reinforcement learning

Reinforcement learning is een zeer specifieke soort van leren. Er is bij deze vorm van learning geen dataset met input data, maar is er een bepaalde **context**. In deze context bevindt zich een **agent**. Een agent is een object dat bepaalde opdrachten kan uitvoeren. De context is een wereld waarin deze agent zich bevindt. Door de agent bij bepaalde acties pluspunten of minpunten te geven kun je bepaald gedrag bevorderen.

Figuur 7: Pac-Man

In figuur 7 is het spel Pac-Man te zien. Op dit spel zou reinforcement learning toegepast kunnen worden. De agent is hierbij pacman, dit is namelijk een object dat bepaalde opdrachten kan uitvoeren, zoals: beweeg naar links. De context is hierbij het level, ofwel: de positie van de muren (de blauwe obstakels), de posities van de ghosts (de gekleurde vijanden), de posities van de pac-dots (de kleine stipjes) en de posities van de power-pellets (de grotere stipjes). Het eten van de pac-dots is positief, het geraakt worden door de ghosts is negatief. Door reinforcement learing toe te passen op het spel zal de agent steeds beter worden in het spelen van het spel.

3.3 Normaliseren van data

Zoals ook in tabel 1 te zien is, kunnen verschillende inputs erg van elkaar verschillen qua grootte. Zo zal het aantal kamers nooit in de buurt komen van het oppervlak. Uiteindelijk zou dit een probleem kunnen veroorzaken bij de berekeningen van het systeem. Een groot getal zou namelijk een veel groter aandeel kunnen hebben alleen omdat het getal zoveel groter is. Het is daarom gebruikelijk de inputs te normaliseren. Dit houdt in dat de inputs zullen veranderen in een getallen met een waardes binnen een bepaald gebied, zodat alle verschillende inputs eerlijk met elkaar vergeleken kunnen worden. Je zou bijvoorbeeld voor de oppervlaktes kunnen stellen dat alle waardes tussen 100 m^2 en 1000 m^2 zullen liggen. Aan een input van 500 m^2 zou je dan een waarde van 5 kunnen geven.

3.4 Conclusie

Zelflerende computersystemen bevatten algoritmes gebaseerd op machine learning. Een zelflerend systeem verschilt van reguliere algoritmes zoals Breadth-first search en Depth-first search doordat het in staat is zichzelf te verbeteren.

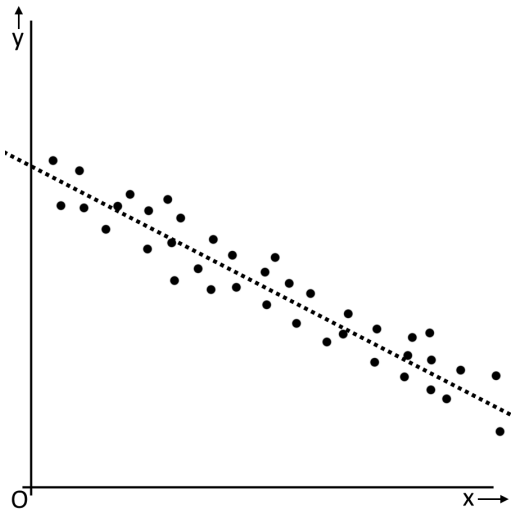
4 Machine learning algoritmes

4.1 Inleiding

In het vorige hoofdstuk hebben we behandeld wat een zelflerend systeem is. In dit hoofdstuk gaan we dieper in op de verschillende soorten zelflerende algoritmes en beantwoorden we de vraag: *wat zijn voorbeelden van zelflerende algoritmes en hoe werken ze?* We zullen in deze deelvraag naar drie verschillende algoritmes kijken: *linear regression, support vector machines en artificial neural networks.* [6]

4.2 Linear regression

Het eerste machine learning algoritme dat we gaan behandelen is **linear regression**. Dit algoritme wordt gebruikt voor het voorspellen van een y-waarde bij (één) gegeven x-waarde(n). Om linear regression te kunnen gebruiken is het belangrijk dat er wel een lineair verband bestaat tussen de waardes van x en y. In figuur 8 is een dergelijk lineair verband te zien. Dit lineaire verband is te beschrijven met een formule in de vorm: $y = ax + b$



Figuur 8: Linear regression

Het doel bij linear regression is het bepalen van de waarde voor a en b in de formule $y = ax + b$. Dit is op verschillende manieren mogelijk. Een statistische manier hiervoor is door gebruik te maken van het **ordinary least squares** algoritme. Dit algoritme bepaalt de best passende lijn door de punten, ook wel bekend als de trendlijn. De waarden voor a en b worden hierbij als volgt bepaald:

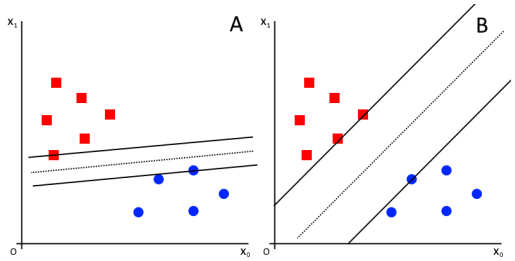
$$a = \frac{\sum_{i=0}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=0}^{n}(x_i - \bar{x})^2}$$
$$b = \bar{y} - (a * \bar{x})$$

In deze formules is $\bar{x}$ het gemiddelde van alle x-waarden en de $\bar{y}$ het gemiddelde van alle y-waarden. Ordinary least squares is echter alleen toepasbaar als er sprake is van één x-waarde als invoer, dus geen multidimensionale invoer waarden zoals $(3, 3)$. Bij meerdere x-waarden is dit algoritme dus niet te gebruiken. Een andere manier om

de waardes voor a en b te vinden is door gebruik te maken van een leerstrategie. In deelvraag 4 bespreken we drie verschillende leerstrategieën.

4.3 Support vector machine

Een support vector machine (SVM) is een machine learning algoritme ontwikkeld door Vladimir Vapnik [7]. Het algoritme kan gebruikt worden voor het classificeren van data. Het algoritme is een vorm van supervised learning [8].



Figuur 9: Support vector machines

Een support vector machine werkt als volgt: het trekt een lijn, een **vector**, tussen twee groepen. Deze vector wordt zó getrokken, dat *de afstand tussen de vector en de dichtstbijzijnde datapunten zo groot mogelijk is* [9]. Deze dichtstbijzijnde datapunten worden de **support vectoren** genoemd. In figuur 9 is twee keer dezelfde dataset weergegeven. In de linker afbeelding is te zien dat de vector de twee groepen scheidt maar de afstand tussen het dichtstbijzijnde datapunt kleiner is dan bij de rechter afbeelding, deze afstand wordt de **marge** genoemd. In de rechter afbeelding is de marge het grootst, dus dit is de betere vector. Het gebied tussen de twee support vectoren wordt de **hyperplane** genoemd.

4.3.1 Het agoritme

Het doel van het algoritme is van een nieuw datapunt bepalen of het tot groep A (de rode vierkantjes) of groep B (de blauwe cirkels) behoort. Als een nieuw datapunt behoort tot groep A, dan willen we dat de output van het algoritme negatief is en als het nieuwe datapunt behoort tot groep B willen we dat de output positief is. Hoe positiever of negatiever de output is, hoe zekerder het is dat dit punt daadwerkelijk tot die groep behoort. Als de output 0 is, dan bevindt het punt zich precies tussen de twee groepen, het ligt dan op de stippellijn van figuur 9. Verder is het zo dat de output tussen -1 en 1 ligt als het binnen de twee support vectoren ligt. In dit gebied is het niet helemaal zeker tot welke groep het punt behoort. We kunnen de drie vectoren als volgt definiëren:

De linker support vector $w * x - b = -1$
De middelste vector $w * x - b = 0$
De rechter support vector $w * x - b = 1$

Bij het trekken van een lijn probeert een support vector machine het volgende te bereiken:

- Alle datapunten moeten buiten de twee support vectoren liggen.
- De afstand tussen de support vectoren moet zo groot mogelijk zijn.

Vanuit de vectoren zijn de volgende twee formules af te leiden:

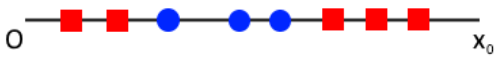
- De formule om te bepalen of een datapunt buiten de twee support vectoren ligt: $y_i(w^T x_i - b) \geq 0$.
- De formule voor de afstand tussen de twee support vectoren: $\frac{2}{||w||}$ ($||w||$ betekent de lengte van de vector).

Een vector die voldoet aan de volgende eisen wordt gekozen:

- Voor alle datapunten moet gelden: $y_i(w^T x_i - b) \geq 0$.
- $\frac{2}{||w||}$ moet zo klein mogelijk zijn, ofwel $||w||$ zo groot mogelijk.

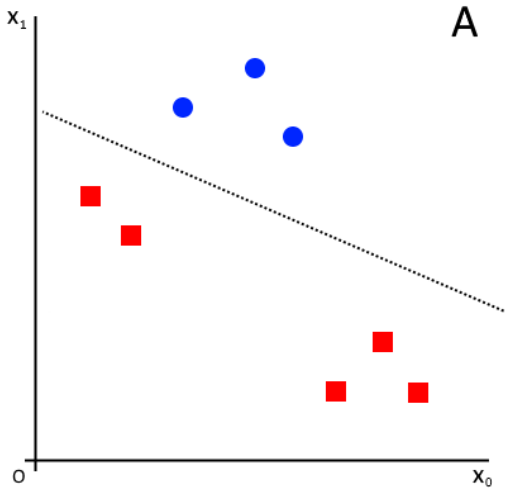
4.3.2 Kernel Methods

In veel gevallen zal de dataset niet zo mooi geordend zijn als in figuur 9. Het is dan niet mogelijk om een rechte lijn te trekken die de twee groepen scheidt. Een support vector machine zou in dit geval dus niet werken. Om toch een support vector machine te kunnen gebruiken is er een techniek genaamd de **kernel krick**.



Figuur 10: Een eendimensionale dataset

In figuur 10 is een eendimensionale dataset te zien. Dit wil zeggen dat er maar één variabele is. Met een support vector machine is het nu niet mogelijk om een lijn te trekken die de twee groepen scheidt. Daarom wordt er een extra variabele bij gemaakt, bijvoorbeeld $X_1 = (X_0)^2$. Het kan zijn dat er dan wel een lijn getrokken kan worden die de groepen scheidt. (zie figuur 11). Deze methode kan ook toegepast worden in situaties met meerdere dimensies.

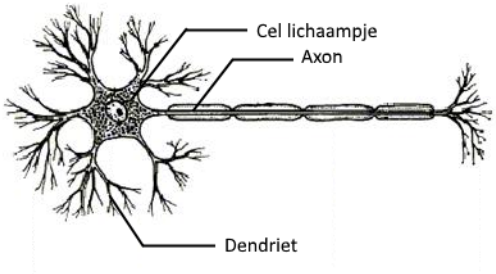


Figuur 11: Een dataset, met een (fictief) uitgevoerde kernel trick

4.4 Artificial neural networks

4.4.1 Biologisch en kunstmatig netwerk

Binnen mensen wordt informatie overgebracht door middel van het zenuwstelsel. Dit zenuwstelsel is opgebouwd uit miljarden zenuwcellen. Een zenuwcel, ook wel een neuron genoemd, is opgebouwd uit drie delen: een cel lichaampje, een aantal dendrieten en één axon. In figuur 12 is een weergave van een biologische neuron te zien.



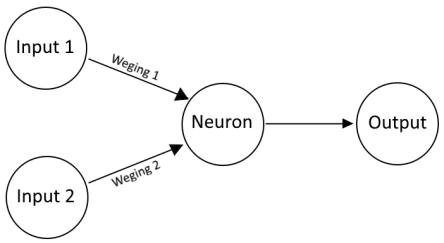
Figuur 12: Een tekening van een biologisch neuron

In de biologie zijn dendrieten verantwoordelijk voor de instroom van informatie. Zij brengen informatie (impulsen) naar het cellichaampje toe. De zenuwcel kan deze informatie vervolgens via een enkele axon doorgeven aan een dendriet van een andere zenuwcel of aan een spier. Het doorgeven van informatie gebeurt in de uiteindes van de axonen en dendrieten, in zogeheten **synapsen**.

Het principe van een neuron kan ook door een computer uitgevoerd worden. Dit is het idee voor een artificial neural network (ANN). Een dergelijk netwerk bestaat uit een verschillend aantal „computerneuronen". Elk van deze neuronen krijgt, net zoals een biologische neuron, informatie binnen. Binnen de neuron vindt een berekening plaats. Vervolgens wordt de berekende waarde doorgegeven aan de volgende neuron of gegeven als output.

4.4.2 De perceptron

De simpelste vorm van een neural network is een netwerk met slechts één neuron. Zo'n ANN, voor het eerst gemaakt door F. Rosenblatt in 1958 [10], wordt een **perceptron** genoemd.



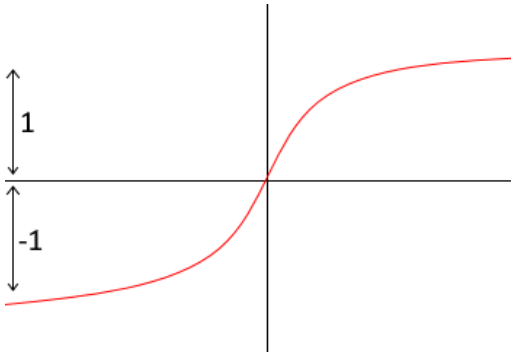
Figuur 13: een schematische weergave van en perceptron

In figuur 13 is te zien dat één neuron twee inputs binnen krijgt en daarna een output geeft. De pijlen naar de neuron toe en er vanaf stellen de synapsen voor. Elke synaps heeft een bepaalde weging. De weging van een synaps bepaalt hoeveel invloed die ene input heeft op het netwerk. Het uiteindelijke doel van een neural network is *het zoeken naar de optimale weging voor alle synapsen binnen het netwerk.* Om tot een output te kunnen komen moet de neuron een berekening uitvoeren. In deze situatie is de berekening nog vrij eenvoudig:

$$totaal = X_1 * W_1 + X_2 * W_2$$

4.4.3 Activation functions

De waarde die uit deze berekening volgt, wordt door een **activation function** gehaald. Een activation function zorgt er voor dat aan deze som een waarde kan worden gehangen, bijvoorbeeld 1 of -1, zonder dat de som absoluut deze waarde heeft. Dit wordt gedaan door te kijken waar het punt op de grafiek van deze functie zich bevindt.



Figuur 14: Een voorbeeld van een algemene activation function en welke waardes hieraan worden gekoppeld

In figuur 14 is een grafiek van een activation function gegeven. In dit voorbeeld worden aan alle positieve y-waardes een 1 verbonden en aan alle negatieve een -1.

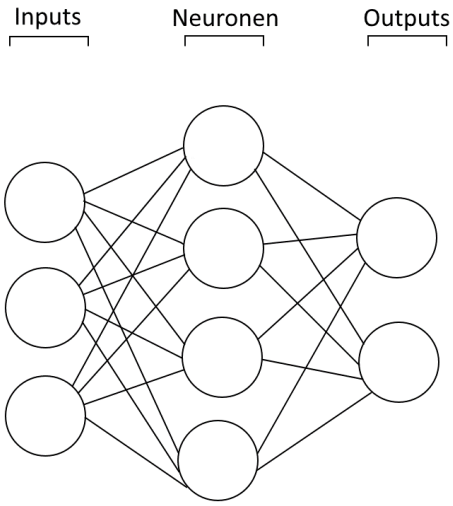
Een ANN is een vorm van supervised learning. Het programma weet dus wat het antwoord zou moeten worden. Als de output correct is zal er weinig gebeuren, maar als de output incorrect is zal het programma zichzelf moeten aanpassen om wél de goede uitkomst te krijgen. Dit gebeurt met behulp van de wegingen van elke synaps. Deze wegingen kunnen namelijk worden aangepast. De invloed van elke input kan ofwel vergroot ofwel verkleind worden. Op deze manier zal uit de berekening in de neuron de volgende keer misschien een andere, betere uitkomst komen. De nieuwe weging van een synaps wordt nu: $w_i = w_i + \Delta w_i$. Hoe Δw_i wordt berekend, wordt bepaald door de leerstrategie van het systeem. Hier wordt in het volgende hoofdstuk verder op in gegaan.

4.4.4 Bias

Met de besproken perceptron is echter een probleem. Wanneer beide inputs gelijk zijn aan nul heeft het aanpassen van wegingen geen effect. Een weging maal nul zal immers altijd in nul resulteren. Om dit probleem tegen te gaan, wordt er een **bias** toegevoegd. Dit is een extra input die standaard gelijk is aan één. De weging van de synaps van de bias wordt niet veranderd. Omdat de neuron nu ook bij inputs van nul een andere uitkomst uit de berekening zal geven, zal er nu toch een getal door de activation function gaan en zullen de wegingen toch worden aangepast.

4.4.5 Een netwerk van perceptrons

Natuurlijk is het ook mogelijk om niet één, maar meerdere perceptrons te hebben. Zo wordt het een echt netwerk van synapsen en neuronen.



Figuur 15: Een schematische weergave van een willekeurig ANN

De laag neuronen noemen we de **hidden layer**. Het is ook mogelijk meerdere lagen neuronen in de hidden layer te hebben. Dit wordt een **deep neural network** genoemd. De tot nu toe besproken ANN's hebben hun informatie allemaal in één richting bewogen: van alle inputs, naar alle neuronen, naar alle outputs. Dit wordt **feedforward** genoemd. Ook zou de output van het netwerk weer terug in het netwerk kunnen worden gestopt, dan noemen we het een **recurrent neural network**.

4.5 Conclusie

Voorbeelden van zelflerende algoritmes zijn: *linear regression*, *support vector machines* en *artificial neural networks*. Alle drie de algoritmes werken verschillend en zijn goed in verschillende scenario's. Linear regression werkt door het opstellen van een lineaire formule door waarden met een lineair verband. Support vector machines werken door een vector te maken die twee groepen zo goed mogelijk scheidt en artificial neural networks werken door een netwerk van neuronen te simuleren.

5 Het verbeteren

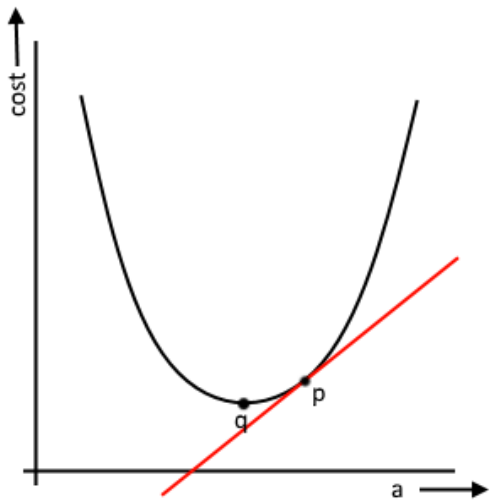
5.1 Inleiding

In de vorige deelvraag hebben we verschillende machine learning algoritmes behandeld. Hierbij hebben we nog niet besproken hoe een algoritme zichzelf kan verbeteren: hoe bepalen we bij linear regression de waarden voor a en b in de formule $y = ax + b$? Hoe bepalen we de waarden voor x en b in de vector $ax - b = 0$ bij een Support Vector Machine? Hoe bepalen we de wegingen van de synapsen in een artificial neural network? Kortom: *op wat voor manieren kunnen zelflerende algoritmes zichzelf verbeteren?* Er zijn verschillende manieren waarop al deze waardes bepaald kunnen worden: gradient descent, Newton's method en evolutionary improvement. Deze drie leerstrategieën zullen we in deze deelvraag behandelen.

5.2 Gradient descent

De eerste leerstrategie die we behandelen is gradient descent. Gradient descent is een algoritme dat functies minimaliseert door het aanpassen van bepaalde parameters. Er wordt geprobeerd de waarde van een bepaalde functie zo laag mogelijk te maken. De functie die we bij een zelflerend systeem proberen te minimaliseren is de **cost function**, ook wel loss function genoemd. Dit is een functie die bepaalt hoe goed het systeem op dat moment werkt. Er wordt bepaald hoeveel de huidige outputs afwijken van de gewenste outputs. Het is hierbij dus nodig dat je de gewenste outputs weet bij gegeven inputs. Er is bij gradient descent dus altijd sprake van supervised learning.

5.2.1 Het algoritme



Figuur 16: De cost function

In figuur 16 is de cost van een bepaalde situatie uitgezet tegen een variabele a. Dit kan bijvoorbeeld de a uit de formule $y = ax + b$ bij linear regression zijn. Het is te zien dat de cost minimaal is in punt q. We willen dus dat a gelijk wordt aan de waarde van a in punt q. Nu is dit punt in deze grafiek erg makkelijk te vinden, maar zodra er gebruik wordt gemaakt van een ingewikkelder algoritme, zoals een ANN, wordt dit punt moeilijker te bepalen.

Op een gegeven moment in het trainingsproces is de a gelijk aan het punt p. Het gradient descent algoritme doet dan het volgende:

- De afgeleide op het huidige punt wordt bepaald (de rode lijn in figuur 16).
- De a wordt zodanig aangepast dat het meer in de richting komt van de q. (Dit wordt gedaan door de afgeleide bij de variabele op te tellen)

Wanneer gradient descent wordt toegepast zal een bepaalde variabele in een zelflerend systeem zo aangepast worden dat de cost als gevolg van die variabele het laagst wordt.

5.2.2 De wiskunde achter gradient descent

Het machine learning algoritme produceert met een bepaalde input een bepaalde output, dit noemen we de **guess**. Omdat we weten wat de goede ouput is kunnen we de **error** bepalen voor die input. De goede output in de volgende formule is y.

$$error_i = y_i - guess_i$$

De vorige formule geldt dus voor de individuele datapunten. De totale error, de som van alle individuele error waardes, ook wel cost of loss genoemd kan als volgt beschreven worden:

$$cost = \sum_{i=0}^{n} (error_i)^2$$

Zoals bekend uit de wiskunde is het mogelijk om hiervan de laagste waarde te bepalen door de afgeleide op nul te herleiden. Voor elk individueel datapunt is de afgeleide van de error:

$$cost'_i = 2(error_i) * error'_i$$

Bij het differentiëren wordt gebruik gemaakt van de kettingregel.

5.2.3 Linear regression met gradient descent

Om het principe van gradient descent beter te begrijpen gaan we nu door middel van gradient descent linear regression uitvoeren.

De guess is hier dus de huidige uitkomst van $y = ax + b$.

$$error_i = y_i - (xa + b)_i$$

De waarde van b_i, x_i en y_i zijn hier constant. De x_i en y_i zijn namelijk bekend uit de training data en b_i verandert wel, maar niet hierbij. De afgeleide van de error functie is dan:

$$error'_i = x_i$$

De afgeleide van de cost function is dan:

$$cost'_i = 2(error_i) * x_i$$
$$cost'_i = 2(y_i - (xa + b)_i) * x_i$$

Met deze afgeleide is de helling van de cost function te bepalen. Hiermee is dus te bepalen welke richting we de variabele a in moeten veranderen. Het aanpassen van de a bij Linear Regression gebeurt dus als volgt:

$$a = a + (2 * error_i * x_i)$$

5.2.4 Learning rate

Een zelflerend systeem bereikt niet in een keer de gewenste output. Er wordt langzaam in de richting van de goede output gewerkt. De formule voor het aanpassen van de a waarde uit het vorige kopje is daarom iets anders. Er wordt een **learning rate** geïntroduceerd:

$$a = a + (error_i * x_i * learningrate)$$

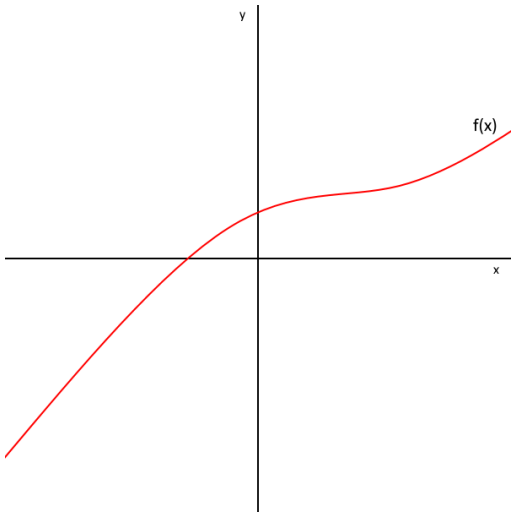
Het kiezen van een goede learning rate is heel belangrijk. Een te lage learning rate zorgt ervoor dat het heel lang duurt voordat de goede output bereikt wordt. Een te hoge learning rate zorgt ervoor dat de gewenste output voorbij wordt geschoten. De gewenste output wordt dan nooit bereikt omdat de variabele net te groot of te klein wordt gemaakt. [11][12]

5.3 Newton's method

Net zoals gradient decent is Newton's method, vernoemd naar Isaac Newton, een manier om de laagste waarde van een bepaalde functie te bepalen. Hiervoor maakt gradient descent gebruik van het gegeven dat een extreme waarde van een grafiek een richtingscoëfficiënt van nul heeft en de afgeleide op dat punt dus gelijk is aan nul. Newton's method gebruik voor het bepalen van de laagste waarde de tweede afgeleide. Er wordt dan gekeken op welke punten deze lijn de x-as snijdt, dit zijn namelijk de toppen van de grafiek van de eerste afgeleide. Door gebruik te maken van Newton's method, zul je een schatting krijgen van het snijpunt met de x-as, maar waarschijnlijk zal je dit punt niet exact kunnen vinden. De nauwkeurigheid van de schatting hangt af van de hoeveelheid waarmee je de stappen herhaalt.

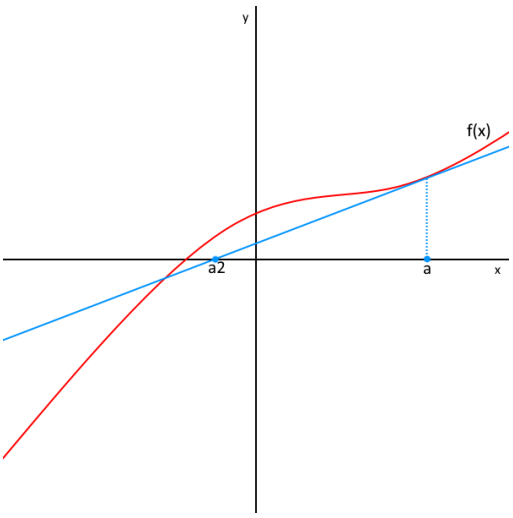
5.3.1 Het proces

In figuur 17 is een willekeurige grafiek getekend.



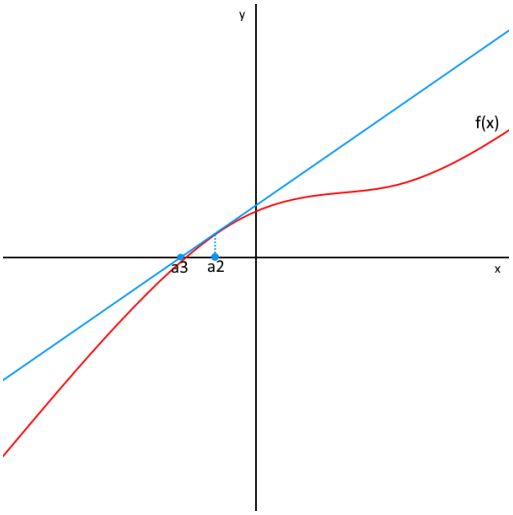
Figuur 17: De grafiek van een willekeurige functie $f(x)$

Bij het gebruik van Newton's method, waarbij we dus zoeken naar een snijpunt met de x-as, wordt eerst een gok gedaan. Deze gok, op punt a, correspondeert met een waarde op de grafiek van $f(x)$. Aan dit punt wordt een raaklijn getekend.



Figuur 18: Een raaklijn aan $f(x)$ op punt $x = a$

De raaklijn van $f(x)$ op $f(a)$ snijdt de x-as op een bepaald punt a_2. Te zien is dat dit punt al aanzienlijk dichter bij het doel ligt dan de originele schatting. Ook a_2 correspondeert met een waarde van $f(x)$ en ook op dit punt kan weer een raaklijn getekend worden (figuur 19).



Figuur 19: Een raaklijn aan $f(x)$ op punt $x = a_2$

Na slechts twee raaklijnen getekend te hebben, ligt het punt a_3 al erg dicht bij het doel. Om een nauwkeurigere benadering van dit doel te bereiken kan je vaker een raaklijn tekenen en het nieuwe snijpunt bepalen. Hoe nauwkeurig je een benadering wil hebben verschilt per situatie.

5.3.2 De wiskunde

Uiteraard zijn de waardes van de punten a_2, a_3 ... a_n te berekenen, we willen immers de waarde van het nulpunt bepalen. Dit gebeurt als volgt:

We weten dat de afgeleide de helling van de grafiek aangeeft: $\frac{\Delta y}{\Delta x}$. Het idee is dat je de helling berekent tussen twee punten die oneindig dicht bij elkaar liggen, hier aangegeven met d: $\frac{dy}{dx}$. Voor het gemak noemen we deze punten x en c. Dit geeft voor de afgeleide:

$$f'(c) = \frac{f(x)-f(c)}{x-c}$$

Dit kan omgeschreven worden tot de formule voor een raaklijn:

$$\begin{aligned} f'(c)(x-c) &= f(x) - f(c) \\ f'(c)(x-c) + f(c) &= f(x) \end{aligned}$$

Om het nulpunt te berekenen, moet gelden $f(x) = 0$. Omdat c slechts een andere waarde voor x aanduidde, kunnen we deze vervangen door het volgende: $c = x_n$ en $x = x_{n+1}$

$$\begin{aligned} f'(x_n)(x_{n+1} - x_n) + f(x_n) &= 0 \\ f'(x_n) * x_{n+1} - f'(x_n) * x_n + f(x_n) &= 0 \\ f'(x_n) * x_{n+1} &= f'(x_n) * x_n - f(x_n) \\ \frac{f'(x_n) * x_{n+1}}{f'(x_n)} &= \frac{f'(x_n) * x_n - f(x_n)}{f'(x_n)} \\ x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \end{aligned}$$

Met deze formule kan de volgende waarde voor x berekend worden.

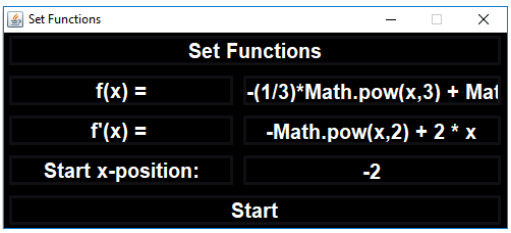
5.3.3 Het gebruik

Er zijn vele situaties te bedenken waarin je de nulpunten van een functie zou willen weten. In het gebied van machine learning wordt het gebruikt om te berekenen waar de cost functie minimaal is. Onder het kopje gradient descent staat al beschreven hoe we aan de cost functie komen en wat de afgeleide hier van is. De grafiek die afgebeeld staat, zou de afgeleide van deze cost functie zijn. Dit betekent namelijk dat de tweede afgeleide van de cost functie wordt genomen wanneer je een raaklijn aan de grafiek berekent.

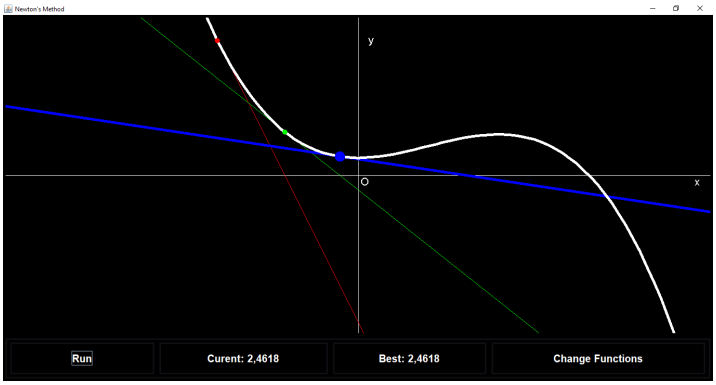
Gradient descent kan goed worden gebruikt bij grafiek met slechts één minimum. Zodra dit niet het geval is, kan het makkelijk in een dal vast blijven hangen, denkend dat het de minimale waarde gevonden heeft, terwijl er misschien nog een lager punt te vinden is. Bij zulke gevallen kan Newton's method ingezet worden, want al deze punten zullen wel op de x-as liggen en dus zullen ze allemaal te vinden zijn met Newton's method.

5.4 Praktijk: Newton's Method

Om Newton's method beter te begijpen en om op een andere manier met de stof bezig te zijn, hebben een programma geschreven in Java dat de manier waarop Newton's method werkt, uitbeeldt. Zodra je dit programma opstart, krijg je een scherm te zien waarin je de functie die je wil onderzoeken kan invullen. Dit is te zien in figuur 20. Als je vervolgens op *Start* drukt, krijg je de door jouw ingevoerde formule te zien in een assenstelsel. Druk nu op *Run* om Newton's method uit te voeren (zie figuur 21). In de vakjes *Current* en *Best* staan de huidige schatting en de beste schatting van het nulpunt van de lijn weergegeven. Ten slotte heb je een knop om de ingevoerde functie te veranderen.



Figuur 20: Het invullen van een formule in ons programma



Figuur 21: Newton's method uitgebeeld

5.5 Evolutionary improvement

Het leerproces van een systeem zou de evolutie van het systeem genoemd kunnen worden: het leert zichzelf beter te functioneren in een bepaalde omgeving. Net zoals evolutie in de biologie, gaat evolutionary improvement in generaties van systemen. Deze manier van leren maakt gebruik van het doorgeven van informatie tussen deze generaties om het algemene niveau van presteren te verhogen.

5.5.1 DNA

Wanneer evolutionary improvement wordt toegepast, is er altijd sprake van een bepaald DNA. In dit DNA staan een aantal waardes. Deze waardes kunnen worden doorgegeven aan de volgende generatie.

5.5.2 Een populatie

Wanneer de informatie van een enkel individu telkens wordt doorgegeven aan een volgende generatie die ook bestaat uit slecht één individu, zal de verbetering van een systeem niet zo groot of zelfs afwezig zijn. Het systeem weet niet of het DNA dat doorgegeven wordt goed of slecht presteert, want er is maar één individu per generatie. Om deze reden bestaat een generatie meestal uit meerdere individuen. Ze zullen niet allemaal even goed presteren en dus zal er onderscheid gemaakt kunnen worden tussen beter en slechter presterende individuen.

De overgave van DNA kan op veel verschillende manieren gebeuren en is afhankelijk van het soort programma en de voorkeur van de programmeur. Je zou bijvoorbeeld de individuen uit een populatie kunnen rangschikken op volgorde van prestatie (hoe prestatie wordt gemeten is natuurlijk ook geheel afhankelijk van het soort programma) en een

bepaald percentage van het slechtst presterende deel laten afvallen [13]. Vervolgens vul je dit deel weer op met individuen met een willekeurig DNA, de rest van de populatie blijft gelijk. Het idee is dat je door telkens het slechte DNA weg te filteren, uiteindelijk een populatie krijgt die gemiddeld steeds beter presteert.

Een andere manier voor evolutie is stellen dat na een bepaalde tijd elk individu een „kind" krijgt. Dit zou goed kunnen werken in een simulatie waarin individuen een rivaliserend verband met elkaar hebben, bijvoorbeeld doordat ze dezelfde voeding nodig hebben. De individuen die langer overleven zullen meer kinderen krijgen en hun DNA dus vaker doorgeven, terwijl de individuen met slechte eigenschappen snel doodgaan. Natuurlijk kan je er ook voor kiezen het DNA van meerdere individuen te combineren voor een volgende generatie.

5.5.3 Mutaties

In de biologie kan in het DNA een **mutatie** plaatsvinden. Een mutatie is een willekeurige verandering zonder echte reden. Nu kunnen deze mutaties nadelig zijn door bijvoorbeeld ziektes te veroorzaken, maar voor evolutie zijn ze erg nuttig. Zonder mutaties zou het DNA altijd gebonden blijven aan wat er al bestaat omdat het telkens wordt doorgegeven. Zo zou er niets nieuws kunnen ontstaan en zou het systeem misschien vast komen te zitten. Als je bijvoorbeeld een programma hebt waarin een systeem leert een hindernisbaan over te gaan, maar geen enkel individu heeft in zijn DNA staan hoe je moet springen, dan kan het programma nooit over een horde heen komen. Mutaties dienen ervoor zulke problemen te voorkomen. Je voegt een bepaalde mutatiefactor toe, een kleine kans die ervoor zorgt dat het programma soms een willekeurige verandering aanbrengt waardoor nieuwe mogelijkheden voor de individuen kunnen ontstaan.

5.6 Conclusie

Zelflerende algoritmes kunnen zichzelf verbeteren door gebruik te maken van leerstrategieën. Drie veel gebruikte leerstrategieën zijn: *gradient descent*, *Newton's method* en *evolutionary improvement*. De eerste twee gebruiken een wiskundige aanpak terwijl evolutionary improvement vooral op kans gebaseerd is.

6 Toepassingen

6.1 Inleiding

Kunstmatige intelligentie klinkt misschien als iets dat uitsluitend voorkomt in sciencefiction, maar in werkelijkheid kent het al vele toepassingen in de hedendaagse wereld. We beantwoorden in dit hoofdstuk de vraag: *welke toepassingen hebben systemen die gebruik maken van een zelflerend algoritme?*

6.2 Weak AI

Er kan onderscheid gemaakt worden tussen weak AI en strong AI. Dit zegt niet zozeer iets over de denkkracht van het systeem, maar eerder over de manier waarop het met informatie omgaat. De meeste voorbeelden van hedendaagse AI vallen in de eerste categorie. Weak AI is ontworpen voor een specifieke taak. Persoonlijke assistenten als Siri en Cortana zijn hier goede voorbeelden van. Ze zijn ontworpen om te functioneren binnen een van tevoren bepaald gebied. Zodra je iets tegen Siri zegt wat niet vergelijkbaar is met de dingen binnen dit gebied, dan zal het niet in staat zijn goed te reageren. Er is geen sprake van echte intelligentie of bewustzijn. Persoonlijke assistenten werken met voice recognition. Wanneer de gebruiker iets zegt, vergelijken de assistenten dit met dingen die ze kennen. Ze kiezen uit wat het meest vergelijkbaar is en geven op basis van deze vergelijking een reactie. Deze vergelijking maken is kenmerkend voor weak AI. Deze soort AI werkt dus met supervised learning.

6.3 Strong AI

De AI die dichter in de buurt komt van de robots uit de sciencefiction is strong AI. Het streven hierbij is een programma te maken vergelijkbaar met een menselijk brein [14]. Een AI als deze zou nieuwe informatie moeten kunnen interpreteren. Hiervoor moet het, naast vergelijken, kunnen associëren. Een simpel voorbeeld is zeggen dat je de volgende dag om acht uur op wilt staan. Een weak AI zal waarschijnlijk niks met deze informatie doen, terwijl een strong AI het initiatief zou kunnen nemen om de wekker op acht uur te zetten. Strong AI maakt dus gebruik van unsupervised learning. Deze vorm van artificial intelligence vereist echter nog veel onderzoek.

6.4 Artificial general intelligence (AGI)

Natuurlijk kan een programma ook voor meerdere taken toepasbaar zijn zonder dat het bewustzijn heeft. In dit geval wordt gesproken van artificial general intelligence. Dit is een AI die zichzelf kan leren verschillende dingen te doen. Een voorbeeld is Deep Q, een deep artificial neural network. Deep Q leerde zichzelf een bepaald Atari 2600 spel te spelen. Toen dit lukte en de onderzoekers 48 andere spellen aan het ANN gaven, was Deep Q in staat ook deze spellen, waarvoor hij niet geprogrammeerd was, te kunnen spelen [15]. AGI is al een stuk verder dan Strong AI is, zeker met de recente ontwikkeling van Google DeepMind [16].

6.5 Verdere toepassingen

Machine learning kan hulp bieden bij vele taken op heel veel gebieden. Dit komt doordat sommige vraagstukken te groot zijn voor een menselijk brein of te veel berekeningen vereisen. Denk bijvoorbeeld aan het maken van schoolroosters. De optimale roosters vinden voor honderden leerlingen en docenten is een opdracht die voor een mens, zonder hulp van een computer, haast niet te doen is. De computer echter kan veel sneller

de opties langsgaan om te zoeken naar het optimum. De mens moet dan slechts nog aangeven waardoor dit optimum wordt bepaald. Ook in de financiële sector zijn vele toepassingen te noemen. Neem bijvoorbeeld de aandelenmarkt. Continu vinden stijgingen en dalingen plaats van bepaalde waardes en na een tijdje kan het teveel worden voor een mens. Een computer is echter in staat veel meer waardes te interpreteren en te vergelijken. Daarom worden er programma's getraind om het verloop van deze markt te voorspellen. Het aantal voorbeelden dat hier gegeven kan worden is ontzettend groot. In vrijwel elk gebied is wel iets te bedenken waarin een AI hulp kan bieden.

6.6 Conclusie

Zelflerende systemen worden ingezet voor taken die voor de mens te groot, te moeilijk of te intensief worden. Er kan onderscheid gemaakt worden tussen zulke systemen op basis van toepasbaarheid (voor een enkele taak of voor een onbepaald aantal taken) en de manier waarop het met informatie omgaat.

7 Limitaties

7.1 Inleiding

In het vorige hoofdstuk is te lezen waar zelflerende systemen allemaal voor gebruikt kunnen worden. Toch zijn zelflerende systemen niet altijd toepasbaar. We gaan in dit hoofdstuk de vraag beantwoorden: *welke factoren zorgen ervoor dat zelflerende systemen in de praktijk niet altijd toepasbaar zijn?* We behandelen factoren die het gebruik van machine learning limiteren.

7.2 Training Data

In veel gevallen heeft een zelflerend systeem training data nodig om beter te kunnen presteren. Een zelflerend systeem moet een bepaald niveau bereiken voordat het in de praktijk kan worden toegepast. Denk hierbij bijvoorbeeld aan een op machine learning gebaseerde zelfrijdende auto. Deze moet een bepaalde afstand kunnen rijden zonder gevaarlijke situaties te veroorzaken. Om dit bepaalde niveau te kunnen bereiken is er veel training data nodig waarmee het systeem verbeterd kan worden. Deze training data is niet altijd voldoende en in goede kwaliteit beschikbaar. De training data moet gelijkwaardig zijn aan de data die het zelflerende systeem in de praktijk tegenkomt. Hoe complexer de data is, hoe meer data er ook nodig is om een goed niveau te bereiken. Om bijvoorbeeld *image classifiction* te kunnen uitvoeren, zijn er duizenden voorbeeldplaatjes, met de goede classificatie, nodig om het algoritme te trainen. Als men niet over die data beschikt, kan het algoritme niet verbeteren.

7.2.1 Semi-supervised learning

Semi-supervised learning is een techniek die het bovengenoemde probleem beperkt. Er wordt gebruik gemaakt van twee groepen data, een grote unlabeled dataset en een kleine labeled dataset. De labeled data zal vaak door een mens van een label moeten worden voorzien en is daardoor moeilijker te verkrijgen, terwijl er vaak genoeg unlabeled data is [17].

7.3 Grootte

Een andere limitatie die men vaak tegenkomt is die van de computersnelheid. Voor complexe taken zijn grotere zelflerende systemen nodig. Op een gegeven moment loop je tegen de limieten van de computer aan. Een complexe taak als het spelen van het spel Go vereist enorm veel computer capaciteit. *Google's DeepMind-project* gebruikte hiervoor 1202 CPUs and 176 GPUs [18]. Voor iemand die niet in het bezit is van evenveel computercapaciteit als Google, zou dit dus onmogelijk zijn geweest. De capaciteit van de computer limiteert de haalbaarheid van bepaalde doelen enorm. Het enige wat hieraan gedaan kan worden, is het verbeteren van de computers en het slimmer schrijven van het zelflerende systeem.

7.4 Specifiek

Er is nog een limitatie die die het gebruik van machine learning belemmert: een systeem is specifiek getraind voor een bepaalde taak. Als het algoritme getraind is voor een specifiek doel, kan het niet zomaar een ander doel krijgen. Als er bijvoorbeeld een zelflerend systeem is getraind op het spelen van schaak, zal het niet ook zomaar andere spellen kunnen spelen. Er wordt in de machine learning gestreefd naar het crëren van een *general intelligence*, ofwel een AI die meerdere taken kan vervullen.

7.4.1 Transfer learning

Transfer learning is het toepassen van de kennis van het zelflerende systeem van één probleem op een ander probleem. Dit is erg goed toepasbaar met image classification. Het algoritme moet hierbij namelijk eerst leren hoe een plaatje in elkaar zit en kan daarna pas specifieke plaatjes sorteren. Door alleen een bepaald deel van het zelflerende systeem opnieuw te trainen hoef je niet het hele systeem opnieuw te laten leren, maar alleen een bepaald deel.

7.5 Conclusie

Er zijn factoren die het gebruik van zelflerende systemen in de praktijk beperken. Hoewel er veel onderzoek wordt verricht naar manieren om de limieten van zelflerende systemen te omzeilen, zullen onder andere een gebrek aan goede training data, een beperkte computer capaciteit en het feit dat een zelflerend systeem slechts een enkele taak kan uitvoeren, in de nabije toekomst iets blijven om rekening mee te houden.

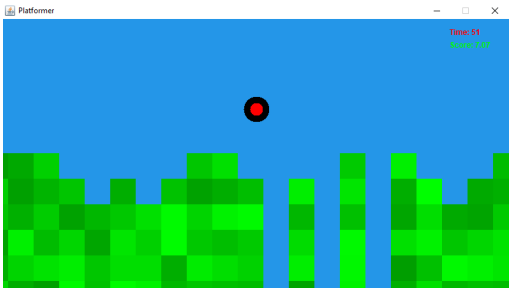
8 Praktijk: evolutionary improvement

8.1 Inleiding

Nu wij in de vorige hoofdstukken de nodige achterliggende theorie hebben onderzocht, gaan we onze kennis inzetten voor het maken van twee praktijkopdrachten: een spel dat gespeeld moet worden door een zelflerend programma en een zelflerend programma om handgeschreven nummers te herkennen. In dit hoofdstuk zijn wij bezig met de vraag: *welke onderdelen zijn nodig voor een zelflerend computersysteem, in de vorm van een door ons ontworpen computerprogramma, dat in staat is een eenvoudig, zelfgemaakt computerspel, met daarin meerdere obstakels, te spelen?*

8.2 Het spel

Voordat we het zelflerende deel van het programma kunnen maken, moet er eerst een spel komen. Hiervoor kiezen wij een *platformer*. Er is een wereld waar de speler doorheen kan springen. Er zitten heuvels, kuilen en gaten in. Wanneer de speler in zo'n gat valt, gaat hij dood. Het doel van het spel is simpelweg zo ver mogelijk te komen. Lopen doe je automatisch, dus het komt echt aan op de timing van je sprong. Voor een mens is dit makkelijk te bevatten, maar voor een computer kan dit een groot obstakel blijken te zijn. In figuur 23 is goed te zien hoe het spel is opgebouwd uit „blokjes". Op sommige blokjes, de groene, kan de speler staan. Alles wat blauw is, is niet solide en de speler valt er dus doorheen (dit is een eerdere versie van het spel).



Figuur 22: De eerste versie van het spel

8.3 Evolutionary improvement toepassen

Nu is het tijd om een zelflerend systeem te maken die dit spel kan leren spelen. We doen dit met evolutionary improvement (zie hoofstuk 5). Eigenlijk moet het programma een enkele vraag kunnen beantwoorden: wat is het beste moment om te springen?

Elk individu dat het spel speelt heeft DNA. In dit DNA staat beschreven hoe lang een speler telkens moet wachten voordat hij springt. (Ook staan de kleuren van het blokje en van de ogen hierin beschreven, maar dat heeft geen invloed op het spelen van het spel). Het DNA van de best presterende individuen kan worden doorgegeven tussen generaties. Het idee is dat het programma steeds beter wordt doordat het steeds „beter" DNA voor zijn spelers heeft. Ook kunnen er willekeurige mutaties en **crossovers** plaatsvinden. Een mutatie is een willekeurige verandering in het DNA en een crossover is het op twee verschillende plekken wisselen van de informatie op het DNA. Deze acties worden ondernomen om diversiteit in de populatie te vergroten. Stel je voor dat je op tijdstip 4, moet springen, maar geen enkel individu heeft hier de code voor, dan zal er nooit vordering kunnen komen.

De *fitness* van een bepaald individu wordt bepaald door zijn horizontale positie. Komt het individu verder, dan is hij meer geschikt en zal hij dus een hogere fitness hebben. Om tot het getal te komen dat weergegeven wordt als de fitness, wordt de positie echter in het kwadraat gedaan. Dit is niet zonder reden: op het moment dat een bepaald individu wél over een gat springt, maar een ander individu niet, dan is het verschil in x-positie niet zo groot, een gat is immers maar 1 of 2 blokjes breed. Om een kleine vooruitgang meer impact te geven op de fitness, kwadrateren we.

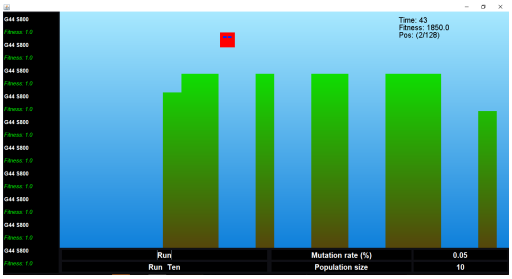
8.4 Het programma

Zodra je het programma opstart, kan je eerst een keer zelf het spel spelen (springen doe je door op de „w"-toets te drukken). Op het scherm is nu echter een stuk meer te zien dan alleen het spel. Rechtsonder zijn de *Mutation rate* en de *Population size* weergegeven, deze getallen geven respectievelijk aan hoe groot de kans is op een mutatie en de grootte van n generatie aan individuen.

Er zijn ook nog twee knoppen te zien: *Run* en *Run Ten*. Wanneer je op *Run* drukt, maakt het programma één generatie individuen aan en laat al deze individuen het spel doorlopen. De generatie staat links weergegeven, op volgorde van hoogste naar laagste fitness. De naamgeving is als volgt: eerst een generatienummer (bijvoorbeeld G10 of G16) en daarna het nummer van het individu uit de generatie (S102, S300, enzovoorts). Als je op een bepaald individu klikt, wordt er afgespeeld hoe hij het spel speelt.

Wanneer je nogmaals op de knop drukt, wordt de volgende generatie aangemaakt. De individuen uit de nieuwe generatie kunnen dus wat DNA „overerven" van de vorige. Links wordt de nieuwe populatie weergegeven. We verwachten hier een hogere fitness te zien, doordat het programma steeds zal proberen het beste DNA door te geven.

Wanneer je op *Run Ten* drukt, loopt de simulatie tien maal en je krijgt de resultaten te zien van de laatste generatie.

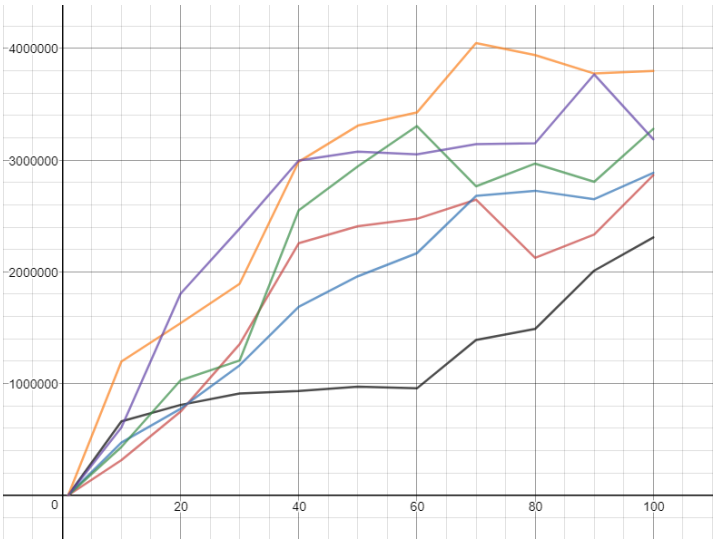


Figuur 23: Een bepaald individu dat, aangestuurd door het programma, het spel speelt

8.5 De resultaten

Het is de bedoeling dat het programma beter wordt in het spelen van het spel, naarmate er meer generaties zijn geweest. En ja hoor! Wanneer je één run uitvoert, zie je al een grote verandering in de fitness. Druk je vervolgens nog een aantal keer op *Run Ten*, dan zie je de getallen nog groter worden. In figuur 24 is de fitness te zien van het beste individu van 10 generaties uit 6 verschillende tests (langs de horizontale as staat het generatienummer, langs de verticale staat de fitness). Ook hier is goed te zien dat het programma alsmaar beter wordt. Soms lijkt er generaties lang echter geen vordering te zijn, soms neemt de fitness zelf af, maar vaak komt het programma ook zo'n dipje uit. Dit kan als volgt verklaart worden: het is waarschijnlijk dat er op een bepaalde plek in de wereld, waar de individuen doorheen moeten, een moeilijk obstakel zit, waar nog geen enkel individu het DNA voor heeft om overheen te komen. Nu moet het programma

geluk hebben dat er een mutatie plaatsvind, die helpt dit obstakel te overkomen. Bij de rode lijn is bijvoorbeeld te zien dat er rond generatie 80 een daling in fitness plaatsvond, maar ook dat er daarna ook weer vordering kwam.



Figuur 24: De hoogste fitness per 10 generaties voor 6 verschillende tests. Op de x-as staat het aantal generaties, op de y-as staat de hoogste fitness van die generatie. (Grafiek gemaakt m.b.v Desmos [19])

8.6 Conclusie

Een computerprogramma blijkt niet erg intelligent te hoeven zijn om zichzelf een spel als deze te kunnen leren spelen. Door middel van willekeurige trial-and-error kan het programma steeds beter worden. Wij hebben gekozen dit evolutionair te doen, maar we denken dat dit ook op de andere in hoofdstuk 5 genoemde manieren kan. We hebben gemerkt dat de speler, aangestuurd door het programma, zich niet „bewust" hoeft te zijn om de wereld waarin hij loopt, hij hoeft immers alleen te weten wanneer hij over een gat heen moet springen.

9 Praktijk: neural network

9.1 Inleiding

Met de kennis die we hebben opgedaan bij het maken van het theorieverslag gaan we zelf een tweede praktijkopdracht uitvoeren. We dagen onszelf uit tot het maken van een computerprogramma dat de volgende vraag beantwoordt: *welke onderdelen zijn nodig voor een zelflerend computersysteem, in de vorm van een door ons ontworpen computerprogramma, dat in staat is afbeeldingen te classificeren binnen vastgestelde categorieën met een precisie van meer dan 80 procent?*

9.2 Werkwijze

Voordat we kunnen beginnen aan het ontwerp van het systeem, moeten we eerst duidelijk maken wat we precies willen bereiken en hoe we dat willen bereiken. We moesten eerst een aantal vragen beantwoorden:

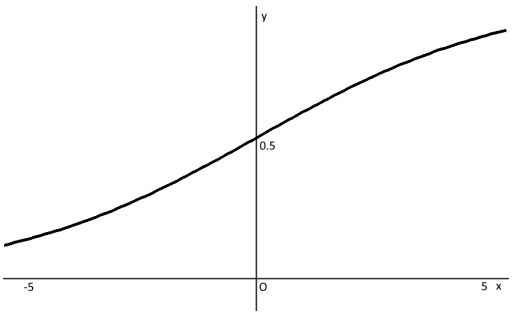
- **Welke afbeeldingen gaan we proberen te classificeren?**
  We gaan proberen handgeschreven cijfers te classificeren. We gebruiken hiervoor de MNIST dataset [20]. Dit is een dataset van een groot aantal afbeeldingen van handgeschreven cijfers met het bijbehorende label. De afbeeldingen zijn elk 28 x 28 pixels groot. Deze dataset is erg geschikt om machine learning op toe te passen.
- **Welk algoritme gebruiken we voor het classificeren van de afbeeldingen?**
  Wij gebruiken een neural network (zie 4.4) om deze afbeeldingen te classificeren. Het scheen ons toe dat dit het beste algoritme is voor het classificeren van afbeeldingen. Er zijn 28 * 28 = 784 input neurons en 10 output neurons. We kiezen ervoor om één hidden layer te maken.
- **Welke manier van leren gebruiken we voor het verbeteren voor het algoritme?**
  We gaan gebruik maken van gradient descent (5.2).

9.3 Netwerk

Het aantal input neurons en het aantal output neurons staat vast (respectievelijk 784 en 10). Het aantal hidden neurons kunnen we zelf bepalen. Ook moet aan elke laag nog een bias toegevoegd worden (4.4.4). Zoals uitgelegd in 4.4.3 moeten we ook een *activation function* kiezen. Hiervoor gebruiken we een sigmoid function (zie figuur 26). De lagen zijn onderling volledig verbonden. Elke verbinding heeft zijn eigen weging die op het begin naar een willekeurige waarde wordt gezet.

$$f(x) = \frac{1}{1 + e^{-u}}$$

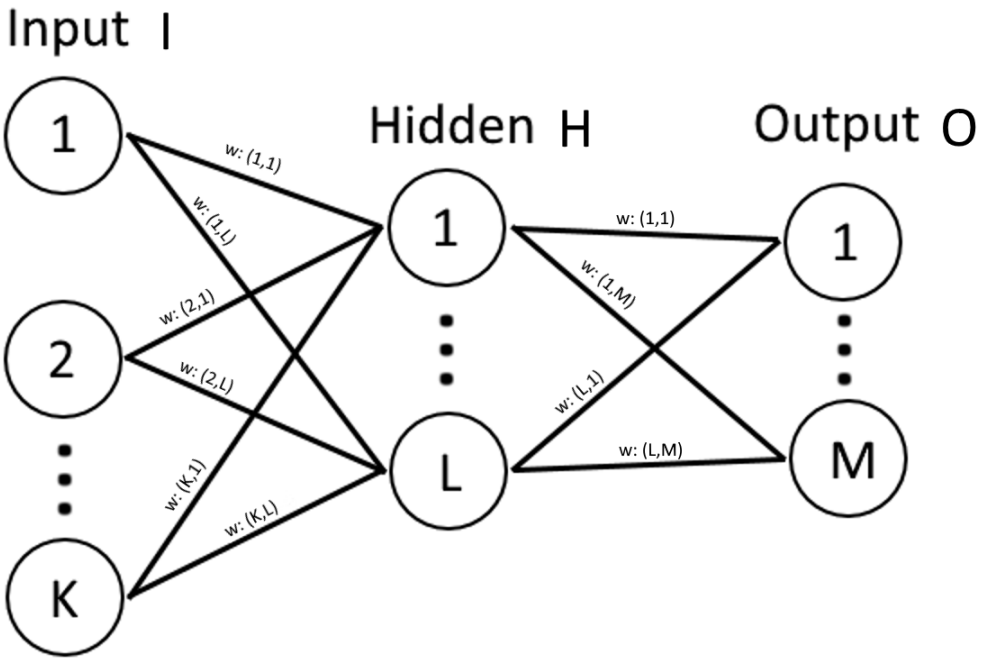
Figuur 25: De sigmoid function



Figuur 26: De grafiek van de sigmoid function

9.4 Gradient descent

Nu we een werkend netwerk hebben kunnen we naar het belangrijkste deel kijken: het vinden van de juiste wegingen voor de neurons. Zoals gezegd gaan we dit doen door middel van gradient descent. Wat we gaan doen is: *het bepalen van de afgeleide van de cost function relatief tot de te veranderen variable.*



Figuur 27: Een neural network met aangegeven weights

De cost function is:

$$C(x) = \frac{1}{2} \sum_{m=1}^{M} (y_m - t_m)^2$$

Let op dat $\frac{1}{2}$ gewoon wordt toegevoegd om later makkelijker de afgeleide te kunnen bepalen. We moeten ook de afgeleide van de sigmoid function weten, deze is gelukkig erg simpel, namelijk:

$$f'(x) = f(u)(1 - f(u)))$$

Wat we nu doen is het bepalen van de afgeleide van de cost function relatief tot de weging van de hidden neurons naar de output neurons (w_{ho}). Om tot dit resultaat te komen, moeten we een aantal keer de kettingregel toepassen:

$$\begin{aligned} c'(x) &= (O_m - t_m) * O' \\ O' &= O_m(1 - O_m) * H_l \\ c'(x) &= (O_m - t_m) * O_m(1 - O_m) * H_l \end{aligned}$$

Nu we de afgeleide berekend hebben, kunnen we de weight een beetje in de goede richting verplaatsen. Hierin staat η voor de learning rate (5.2.4).

$$w_{lm} = w_{lm} - \eta * (O_m - t_m) * O_m(1 - O_m) * H_l$$

Om de afgeleide van de cost function te bepalen relatief tot de weging van de de input neurons naar de hidden neurons moeten we nog een aantal keer de kettingregel toepassen:

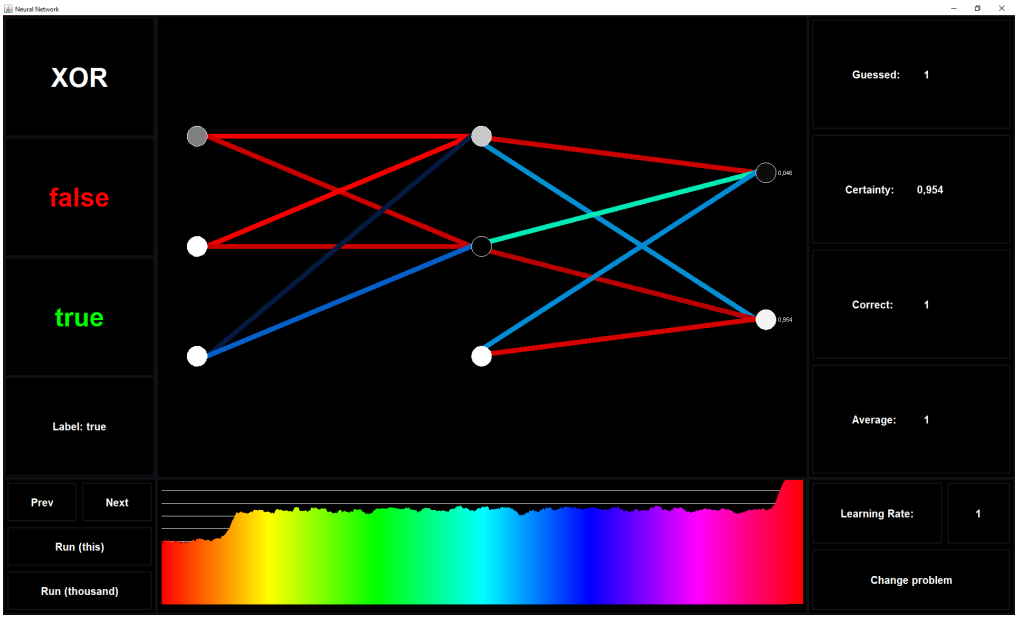
$$\begin{aligned} C'(x) &= \sum_{m=1}^{M} [(O_m - t_m) * O'] \\ O' &= O_m(1 - O_m) * H_l * H' \\ H' &= H_l(1 - H_l) * I_k \\ C'(x) &= \sum_{m=1}^{M} [(O_m - t_m) * O_m(1 - O_m) * w_{ho}] * H_l(1 - H_l) * I_k \end{aligned}$$

De weging w_{lm} wordt dan:

$$w_{lm} = w_{lm} - \eta * \sum_{m=1}^{M} [(O_m - t_m) * O_m(1 - O_m) * w_{ij}] * H_l(1 - H_l) * I_k$$

9.5 Het programma

Bij het opstarten van het programma verschijnt een scherm waarin de beginwaardes van het neural network bepaald kunnen worden en het probleem gekozen kan worden. Als je met de muis op een onderdeel staat, verschijnt er meer informatie over het desbetreffende onderdeel. Verder is een weergave te zien van het neural network. Hierin geeft een rode verbinding een negatieve weging aan, en een blauwe verbinding een positieve weging aan. Om uit te testen of het programma goed functioneert, hebben we eerst een simpel probleem uitgetest: XOR. Het doel van een XOR poort is als volgt: als één van beide input waardes "1" is, moet het antwoord "1" worden, anders "0". Na even trainen leert het neural network dit probleem op te lossen! Dit betekent dat het programma goed functioneert, dus kunnen we naar het *echte* probleem.



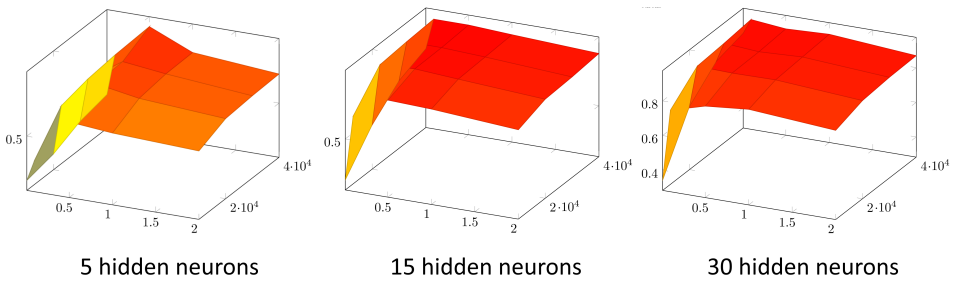
Figuur 28: Het door ons geschreven neural network programma dat zelf heeft geleerd XOR-probleem op te lossen

9.6 Resultaten

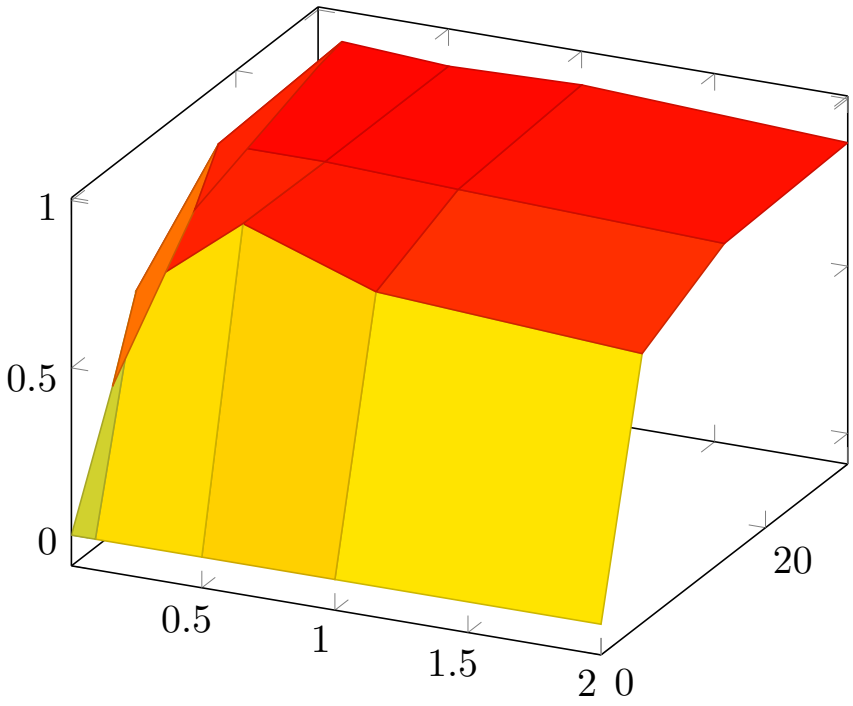
De prestaties van het neural network worden door een aantal variablen beeïnvloed:

- Het aantal hidden neurons
- De learnig rate
- Het aantal training plaatjes

In figuur 29 is te zien welk effect het veranderen van de variablen heeft op de prestaties van het netwerk.



Figuur 29: De resultaten van ons neural network. De y-as duidt de prestatie van het netwerk aan. De x-as duidt de de learning rate aan. De z-as duidt het aantal plaatjes aan.



Figuur 30: De resultaten van ons neural network getraind op **40000** afbeeldingen. De y-as duidt de prestatie van het netwerk aan. De x-as duidt de de learning rate aan. De z-as duidt het aantal hidden neurons aan.

Het hoogst behaalde resultaat is een nauwkeurigheid van 91,7%, bij een learning rate van 0,1 en 30 hidden neurons getraind op 40000 afbeeldingen.

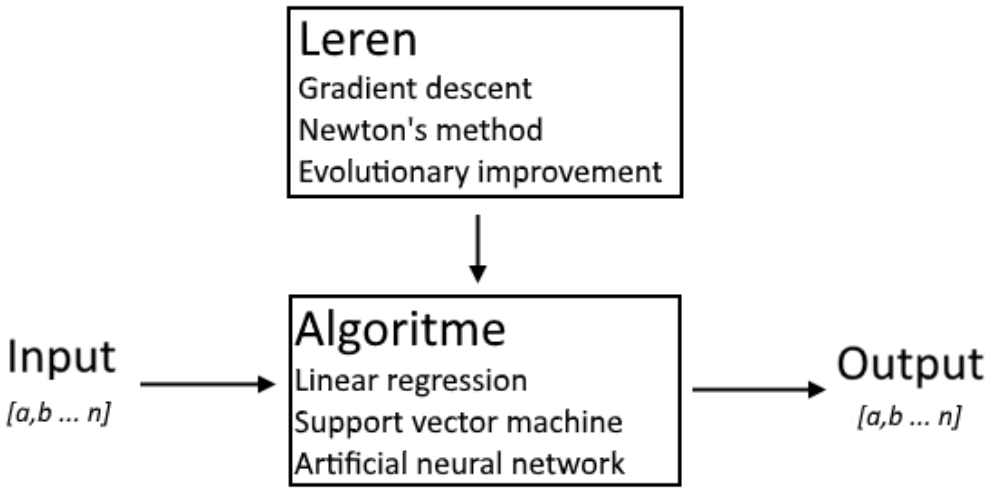
9.7 Conclusie

Eén manier voor het maken van een computerprogramma dat in staat is afbeeldingen te categoriseren, is door gebruik te maken van een neural network dat verbeterd wordt door middel van gradient descent.

10 Conclusie

De vraag die wij onszelf hebben gesteld aan het begin van dit verslag is: *in welke aspecten verschillen diverse zelflerende computersystemen, ontworpen voor één specifieke taak, van elkaar?* Deze aspecten zijn:

- Het algoritme dat gebruikt wordt voor het zelflerende systeem. Hierbij hebben we linear regression, support vector machines en artificial neural networks verder toegelicht.
- De manier van leren. Hierbij hebben we gradient descent, Newton's method en evolutionary improvement verder toegelicht.



Figuur 31: Schematische weergave van een zelflerend systeem

11 Bronnen

11.1 Hoofdstuk 2

[1] Onbekend. „http://www.woorden.org". In: (). DOI: https://www.cims.nyu.edu/~munoz/files/ml_optimization.pdf.

[2] Olvi Mangasarian (chair) Jin-Yi Cai Larry Landweber. „MEMORIAL RESOLUTION OF THE FACULTY OF THE UNIVERSITY OF WISCONSIN-MADISON". In: (1727).

[21] In: (). DOI: https://beebom.com/examples-of-artificial-intelligence/.

[22] In: (). DOI: http://study.com/academy/lesson/what-is-an-algorithm-in-programming-definition-examples-analysis.html.

[23] In: (). DOI: http://www.secfac.wisc.edu/senate/2003/0929/1727(mem_res).pdf.

[24] In: (). DOI: https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm.

[25] In: (). DOI: http://www.graph500.org/specifications#sec-5.

[26] In: (). DOI: https://www.kirupa.com/developer/actionscript/depth_breadth_search.html.

[27] In: (). DOI: http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first_search.

[28] In: (). DOI: https://en.wikipedia.org/wiki/Maze_generation_algorithm#Recursive_backtracker.

[29] In: (). DOI: https://www.khanacademy.org/computer-programming/depth-first-traversals-of-binary-trees/934024358.

11.2 Hoofdstuk 3

[3] Arthur Samuel. „Machine Learning and Optimization". In: *Machine Learning and Optimization* (). DOI: https://www.cims.nyu.edu/~munoz/files/ml_optimization.pdf.

[4] Andrew Ng. „www.coursera.org". In: (). DOI: https://www.coursera.org/learn/machine-learning/lecture/1VkCb/supervised-learning.

[5] Andrew Ng. „www.coursera.org". In: (). DOI: https://www.coursera.org/learn/machine-learning/lecture/olRZo/unsupervised-learning.

11.3 Hoofdstuk 4

[6] Sunil Ray. „www.analyticsvidhya.com". In: (). DOI: https://www.analyticsvidhya.com/blog/2015/08/common-machine-learning-algorithms/.

[7] Vladimir Vapnik. „www.analyticsvidhya.com". In: (). DOI: https://www.analyticsvidhya.com/blog/2015/08/common-machine-learning-algorithms/.

[8] Andreas Christmann Ingo Steinwart. „Support Vector Machines". In: (). DOI: https://books.google.nl/books?hl=nl&lr=&id=HUnqnrpYt4IC&oi=fnd&pg=PP7&dq=support+vector+machines&ots=g8lIEB0rSi&sig=FTLWxhxAwcf95E1xLoWZ8WYFZ4k#v=onepage&q=support%20vector%20machines&f=false.

[9] Onbekend. „www.saedsayad.com". In: (). DOI: http://www.saedsayad.com/support_vector_machine.html.

[10] Onbekend. „psycnet.apa.org”. In: (). DOI: http://psycnet.apa.org/journals/rev/65/6/386/.

[30] In: (). DOI: http://biologiepagina.nl/Vwo5/Zenuwstelsel/inleiding.html.

[31] „Afbeelding biologisch neuron”. In: (). DOI: https://www.evolvingsciences.com/Neuron%20.html.

11.4 Hoofdstuk 5

[11] Andrew Na. „www.coursera.org”. In: (). DOI: https://www.coursera.org/learn/machine-learning/lecture/kCvQc/gradient-descent-for-linear-regression.

[12] Matt Nedrich. „spin.atomicobject.com”. In: (). DOI: https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/.

[13] carrykh. „evolution simulations”. In: (). DOI: https://www.youtube.com/playlist?list=PLrUdxfaFpuuK0rj55Rhc187Tn9vvxck7t.

11.5 Hoofdstuk 6

[14] John Searle. „cogprints.org”. In: (). DOI: http://cogprints.org/7150/1/10.1.1.83.5248.pdf.

[15] Shalini Saxena. „arstechnica.com”. In: (). DOI: https://arstechnica.com/science/2015/02/ai-masters-49-atari-2600-games-without-instructions/.

[16] Matthew Griffin. „www.globalfuturist.org”. In: (). DOI: http://www.globalfuturist.org/2017/03/bad-news-for-jobs-fabled-artificial-general-intelligence-could-arrive-much-earlier-than-expected/.

[32] Richard Evans en Jim Gao. In: (). DOI: https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.

[33] Jeff Kerns. In: (). DOI: http://www.machinedesign.com/robotics/what-s-difference-between-weak-and-strong-ai.

11.6 Hoofdstuk 7

[17] Piyush Rai. „Semi-supervised Learning”. In: (). DOI: https://www.cs.utah.edu/~piyush/teaching/8-11-slides.pdf.

[18] Sam Shead. „Semi-supervised Learning”. In: (). DOI: http://uk.businessinsider.com/heres-how-much-computing-power-google-deepmind-needed-to-beat-lee-sedol-2016-3?international=true&r=UK&IR=T.

11.7 Hoofdstuk 8

[19] „Desmos”. In: (). DOI: www.desmos.com.

11.8 Hoofdstuk 9

[20] „THE MNIST DATABASE of handwritten digits”. In: (). DOI: http://yann.lecun.com/exdb/mnist/.