Zelflerende Systemen

Steven Bronsveld en Thijs van Loenhout

24 augustus 2017

Inhoud

1 Inleiding

2 Wat zijn voorbeelden van reguliere algoritmes en hoe werken ze?

2.1 Inleiding

Elk jaar boekt de mens grootschalige vorderingen op het gebied van computers, zowel hardware als software. Iets waar wij echter nog niet in geslaagd zijn te maken is een ware **Artificial Intelligence**, al lukt het steeds beter de schijn van denken te creëren. Voorbeelden zijn de persoonlijke assistenten die inmiddels in elke smartphone gentegreerd zijn. *Siri*, *Google Now* en *Cortana* maken gebruik van spraakherkenning om de gebruiker de gevraagde informatie te tonen, maar denken zoals mensen doen ze hierbij niet. Hoe een computersysteem toch beter kan worden in het imiteren van menselijk gedrag en van zijn eigen fouten kan leren onderzoeken we in deze deelvraag.

2.2 Verschillende algoritmes

Computers hebben geen dus bewustzijn. Om deze reden kunnen ze niet zelf bepalen iets te doen. Waar computers wel in uitblinken, is het uitvoeren van taken die ze zijn opgelegd. Vaak komen deze taken in de vorm van code. Via code kan 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 in deze commando's ingewikkelder worden. Hier komen algoritmes in het spel. Een algoritme is een soort stappenplan, waarin een complexere handeling in duidelijke opdrachten weergegeven wordt. De volgende definitie geeft een betekenis in de meest algemene zin: *„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 misschien een recept. Dit recept is als het ware het algoritme. Uit de gegeven definitie is af te leiden dat het aantal mogelijke algoritmes ontzettend groot is. 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.

Uiteraard zijn er ook vele algoritmes die gebruik maken van toepassingen, zoals een **queue** en een **stack**, die betrekking hebben tot ons onderwerp. Enkele hiervan zullen hier beschreven worden:

2.3 Breadth-first search (BFS)

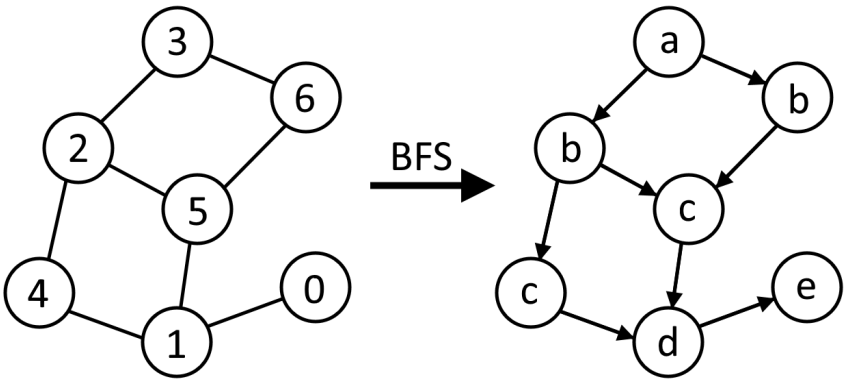
Dit algoritme, bedacht in de jaren vijftig van de vorige eeuw door E.F. Moore [2], een Amerikaans professor in de wiskunde en computer sciences en een voortrekker in kunstmatig leven, is een zoekalgoritme voor datasets in de vorm van grafieken of „boom"-structuren. In deze dataset wordt een **node** 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: bij welke node 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 aan toegevoegd en uitgehaald kunnen worden. Net zoals een daadwerkelijke wachtrij wordt het „*eerste erin, als eerste eruit*" principe toegepast.

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.

**Aangrenzend zijn 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 1), 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 2 als 6 het niet het doel zijn, herhaald het algoritme zich. Nu wordt 2 bekeken. Het doel is niet gevonden. Daarna 6, ook zonder succes. (Let hierbij op dat node 5 niet nogmaals bekeken wordt, dit is namelijk als bij node 2 gedaan en 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.

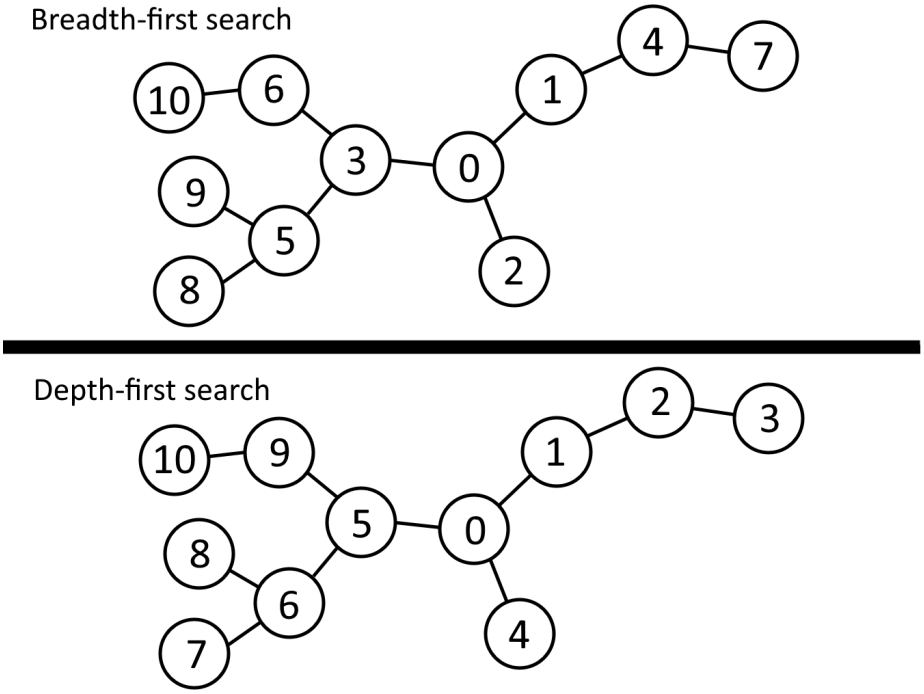
Met BFS kan je zo de weg van de root naar het doel achterhalen. Dit is nuttig als je bijvoorbeeld een wegennetwerk hebt en wil weten wat de kortste weg van de ene naar de andere stad is.

2.4 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 trees. 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 werden geplaats met een „als eerst erin, als eerst eruit" principe, handhaaft een DFS systeem een wachtrij meer vergelijkbaar 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 begon bij een root. Vervolgens werd gekeken naar alle neighbors. Als de gewenste uitkomst niet tussen deze neighbors zit, worden de neighbors van deze neighbors 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 neighbors 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 wordt deze als bezocht opgeslagen. Zoals te zien wordt er vanaf de root één (willekeurige) neighbor gekozen om te onderzoeken. Elke bezochte neighbor wordt als bezocht genoteerd. De root wordt in de stack geplaats. Als de gekozen neighbor niet het doel is, wordt de meest recentelijk toegevoegde node, de root, gehaald, de eerst bezochte had immers geen aanliggende nodes om te onderzoeken. Deze tweede neighbor heeft wel weer een neighbor. Deze wordt gecontroleerd, evenals diens neighbors. Telkens wordt de huidige node toegevoegd aan de stack.

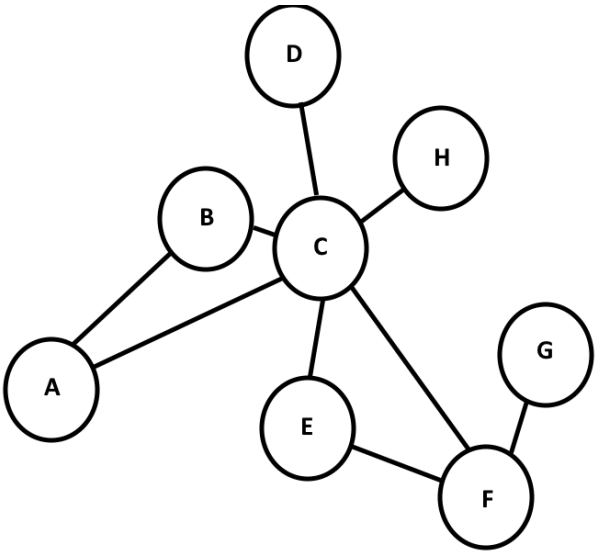
Als het programma de laatste in een reeks nodes bereikt heeft, wordt de bovenste node uit de stack gepakt en gekeken of daar nog niet bezochte nodes aan grenzen. Dit wordt backtracking genoemd. Dit proces wordt herhaald totdat het doel gevonden is of totdat alle nodes bekeken zijn.

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 kortste weg wil weten.

2.5 Voorbeelden algoritmes

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

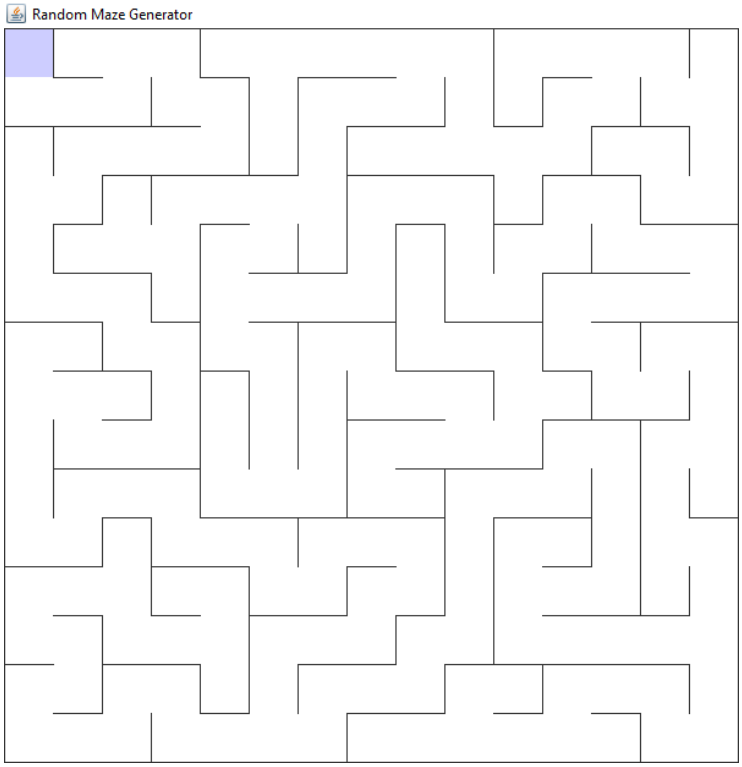
2.5.1 Breadth-first search



Figuur 3: Schematische weergave van een willekeurige dataset.

In figuur 3 is een dataset te zien, bijvoorbeeld een telefoonboom. Elke cirkel representeert een 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 makkelijk met het oog 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.

2.5.2 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 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. Maak de start cel current en markeer deze als bezocht.
2. Terwijl er nog niet bezochte cellen aanwezig zijn:
   (a) Als current neighbors heeft die nog niet bezocht zijn:
      i. Kies willekeurig een van de neighbors
      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 een van de neighbors.
      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 hieruit
ii. Maak deze cel current
iii. Haal de laatst toegevoegde cel uit de stack en verwijder deze hieruit.
iv. Maak deze cel current.

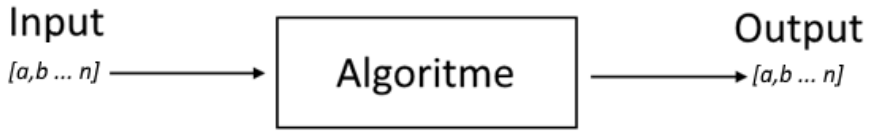
2.6 Zelflerend?

Breadth-first search en Depth-first search zijn beide algoritmes met vele toepassingen. Toch kunnen beide algoritmes niet als zelflerend worden beschouwd, ze verbeteren hun manier van zoeken namelijk niet. Hoe zit een zelflerend systeem dan wel in elkaar? Hoe kan een algoritme zichzelf verbeteren?

3 Wat zijn zelflerende algoritmes en waarin verschillen ze van reguliere algoritmes?

3.1 Machine learning

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 waarbij reguliere algoritmes tekort schieten. Welke taken dit betreft, zullen we in de derde deelvraag behandelen.



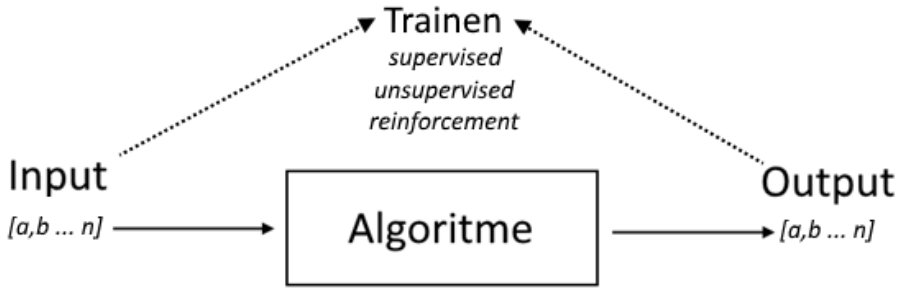
Figuur 5: Schematische weergave van een zelflerend systeem

In figuur 5 is een schematische weergave van een zelflerend systeem afgebeeld. Bepaalde input data gaat het systeem in en bepaalde output data komt het systeem uit. De input en output data bestaat uit één of meerdere getallen. 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 in een zelflerend systeem 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 de tweede deelvraag behandelen.

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 de volgende deelvraag wat duidelijker maken. Er zijn drie prominente manieren waarop een zelflerend systeem getraind kan worden: **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	311 m	4
569.000	133 m	309 m	5
569.500	170 m	310 m	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 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 zullen in een grote

dataset verbanden kunnen 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: Pacman

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). [4] 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 getal met een waarde 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 en 1000 m zullen liggen. Aan een input van 500 m zou je dan een waarde van 5 kunnen geven.

3.4 Conclusie

Zelflerende computersystemen zijn algoritmes gebaseerd op machine learning. Een zelflerend systeem verschilt van reguliere algoritmes zoals breadth-first search en depth-first search doordat ze in staat zijn zichzelf te verbeteren.

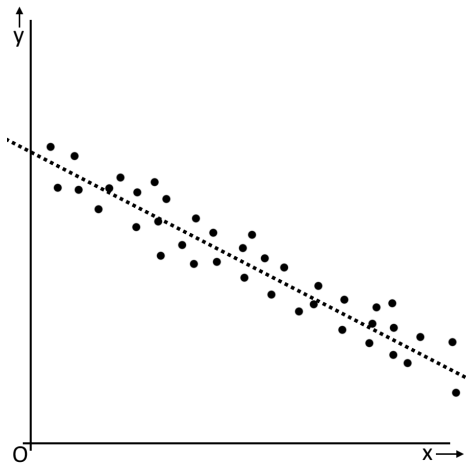
4 Wat zijn voorbeelden van zelflerende algoritmes en hoe werken ze?

4.1 Inleiding

In de vorige deelvraag hebben we behandeld wat een zelflerend systeem is. In deze deelvraag gaan wij dieper in op de verschillende soorten zelflerende algoritmes. 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 gegeven x-waarde(n). Om linear regression te kunnen gebruiken is het belangrijk dat er wel een lineair verband bestaat tussen de x-waarde(n) en de y-waarde. In figuur 8 is een dergelijk linear verband te zien. Dit lineaire verband is te beschrijven met de formule: $y = ax + b$



Figuur 8: Linear regression

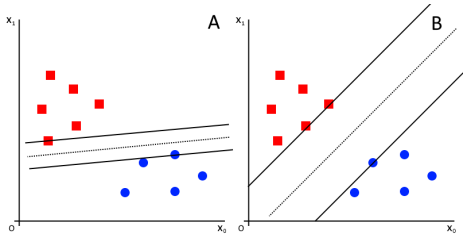
Het doel bij linear regression is het bepalen van de waarde voor a en b. Dit is op verschillende manieren mogelijk. Een statistische manier hiervoor is door gebruik te maken van het **ordinary least squares** algoritme. Dit algoritme bepaalt de best passende lijn door de punten, ook wel bekend als de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

$$\frac{\sum_i^n (x_i - x)(y_i - y)}{\sum_i^n (x_i - x)^2}$$

In deze formules is de x zonder subscript het gemiddelde van alle x waarden en de y zonder subscript het gemiddelde van alle y waarden. Dit algoritme is echter alleen toepasbaar als er sprake is van n x-waarde als invoer. Bij meerdere x-waarden is het dus niet te gebruiken. Een andere manier om de a en b waarde te vinden is door gebruik te maken van een leerstrategie. In deelvraag 4 bespreken we drie verschillende leer strategien.

4.3 Support vector machine

Een support vector machine (SVM) is een machine learning algoritme ontwikkeld door vladimir vapnik. Het algoritme kan gebruikt worden voor het classificeren van data. Het algoritme heeft training data nodig en is dus een vorm van supervised learning. [7]



Figuur 9: Support vector machines

Een support vector machine werkt als volgt: Het trekt een lijn, een textbf-vector, tussen de twee groepen. Deze vector is *zó dat de afstand tussen de vector en het dichtstbijzijnde datapunten zo groot mogelijk is.* [8] Deze dichtstbijzijnde datapunten worden de **support vectoren** genoemd In figuur 11 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. De in de rechter afbeelding is de marge het grootst, dus dit is de betere vector. Het gebied tussen de twee support vectoren wordt het **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 11. 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 definiren:

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

Een support vector machine probeert het volgende:

- 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:

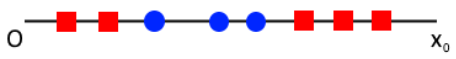
- De formule voor of een datapunt buiten de twee support vectoren ligt: $y_i(w^t x_i - b) \geq 0$
- De formule voor de afstand tussen de twee support vectoren: $\frac{2}{||w||}$

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

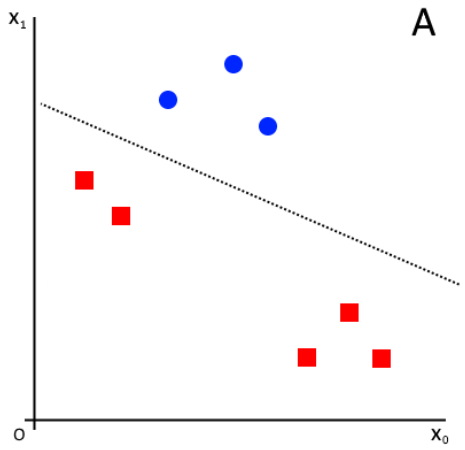
4.3.2 Kernel Methods

In veel gevallen zal de dataset niet zo mooi geordend zijn als in figuur 11. 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 iets genaamd de **Kernel Trick**.



Figuur 10: Een n dimensionale dataset

In figuur **??** is een n dimensionale dataset te zien, dit wil zeggen, er is maar n variabele. 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 $X_1 = (X_0)^2$. Nu is het wel mogelijk een lijn te trekken door de dataset die de twee groepen opdeelt:



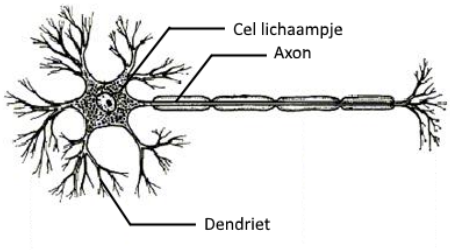
Figuur 11: Een dataset, gescheiden door een lijn

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 n axon. In firguur 12 is een weergave van een biologische neuron te zien.



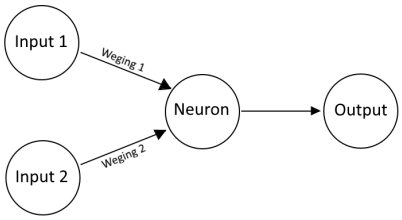
Figuur 12: Een tekening van een biologisch neuron

In de biologie zijn dendrieten verantwoordelijk voor de instroom van informatie, zij brengen informatie (impulsen) naar het cel lichaampje 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 het uiteinde van de axon 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 deze waarde doorgegeven of gegeven als output.

4.4.2 De perceptron

De simpelste vorm van een neural network is een netwerk met slechts n neuron. Zon ANN, voor het eerst gemaakt door F. Rosenblatt in 1958 [9], wordt een **perceptron** genoemd.



Figuur 13: een schematische weergave van en perceptron

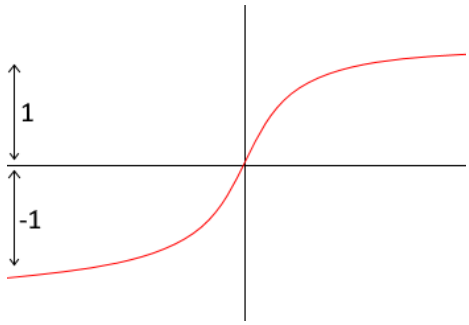
In figuur 13 is te zien dat een neuron twee inputs binnen krijgt en daarna een output geeft. De pijlen naar de neuron toe en er vanaf stellen de synapsen voor. Elke synaps heeft een bepaalde weging. De weging van een synaps bepaald

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:

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

4.4.3 Activation functions

De waarde die uit deze berekening volgt, wordt door een **activation function** gehaald. Een activation function zorgt ervoor 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 bevind.



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 wel 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 verkleint 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:$w_0 = w_1 + w$. Hoe w berekend wordt, wordt bepaald door de leerstrategie van het systeem. Hier wordt in de volgende deelvraag verder op in gegaan.

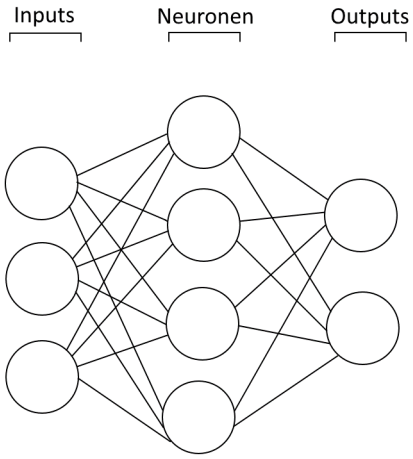
4.4.4 Bias

Met de besproken perceptron is echter n probleem. Wanneer beide inputs gelijk zijn aan 0, heeft het aanpassen van wegingen geeneffect. Een weging maal 0 zal immers altijd in 0 resulteren. Om dit probleem tegen te gaan, wordt er een **bias** toegevoegd. Dit is een extra input die standaard gelijk is aan 1. De weging van de synaps van de bias wordt niet veranderd. Omdat de neuron nu ook bij inputs

van 0 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 van niet n, maar meerdere perceptrons te hebben. Zo wordt het een echt netwerk van synapsen en neuronen.



Figuur 15: Een schematische weergave van een willekeurig ANN.

Ook is het mogelijk meerdere lagen neuronen te hebben. Dit wordt een **deep neural network** genoemd, of simpelweg **deep learning**. De tot nu toe besproken ANNs hebben hun informatie allemaal in n richting bewogen: alle inputs, alle neuronen, alle outputs. Dit wordt **feedforward** genoemd. Ook zou je een neural network kunnen hebben waarin de informatie ook nog tussen de neuronen in dezelfde laag beweegt: een **recurrent neural network**.

4.5 Conclusie

5 Op wat voor manieren kunnen zelflerende algoritmes zichzelf verbeteren?

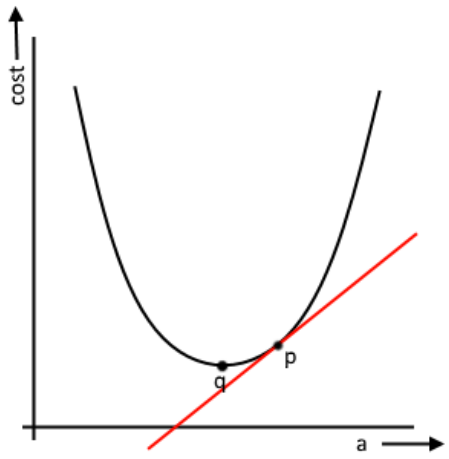
5.1 Inleiding

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 $wx - b = 0$ bij een Support Vector Machine? Hoe bepalen we de wegingen van de synapsen in een Artificial Neural Network? Er zijn verschillende manieren waarop al deze waarden bepaald kunnen worden: Gradient Descent, Newton's method en evolutionary improvement. Deze drie leerstrategien 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 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 dus bij gradient descent 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 dus 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 ingewikkeldere algoritmen, 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 waarden, 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:

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

Bij het differentiren 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 doormiddel 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 veranderd wel, maar niet hierbij. De afgeleide van de error functie is dan:

$$error_i' = x$$

De afgeleide van de cost function is dan:

$$cost' = 2(error) * x \; cost' = 2(y_i - (xa + b)_i) * x$$

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

$$a = a + (2 * error * x)$$

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** gentroduceerd:

$$a = a + (error * x * learningrate)$$

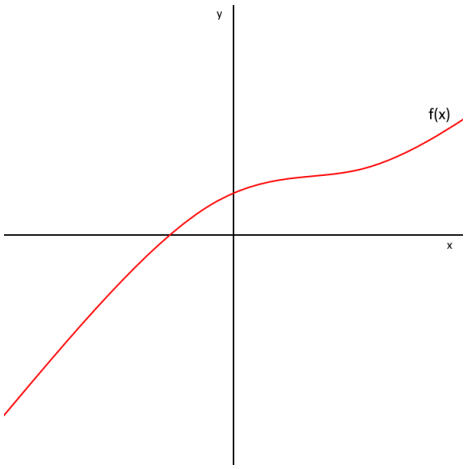
Het kiezen van een goede learning rate is heel belangrijk. Een te lage learning rate zorgt ervoor dat het heel lang duurt voordat de goede output bereikt wordt. Een te hoge learning rate zorgt ervoor dat de gewenste output voorbij wordt geschoten. De gewenste output wordt dan nooit bereikt omdat de variabele net te groot of te klein wordt gemaakt. [10][11]

5.3 Newton's method

Net zoals gradient decent is Newtons 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 0 heeft en de afgeleide op dat punt dus gelijk is aan 0. Newtons 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 Newtons method, zal 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.

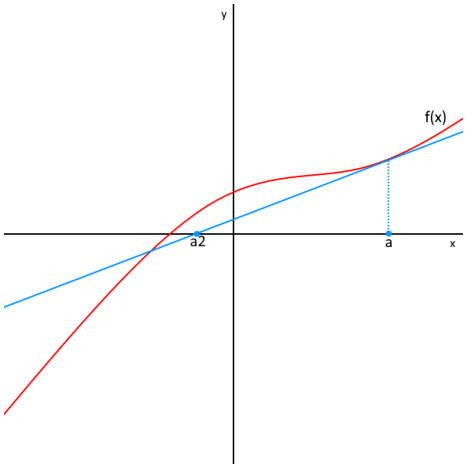
5.3.1 Het proces

In figuur 17 is een willekeurige grafiek getekend.



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

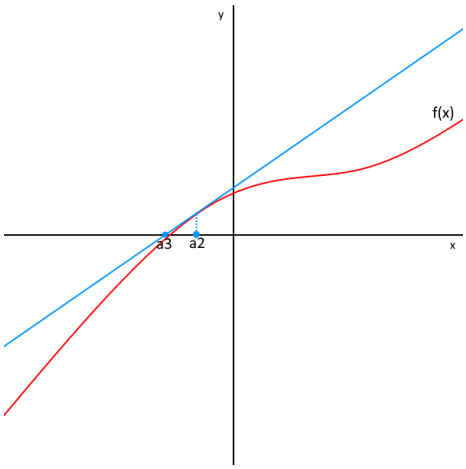
Bij het gebruik van Newtons method, waarbij we dus zoeken naar een snijpunt met de x-as, wordt eerst een gok gedaan. Deze gok, op punt a, correspondeert met een waarde op de grafiek van $f(x)$. Aan dit punt wordt een raaklijn getekend.



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 dichten 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.

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.



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

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:

$$f'(c)(x-c) = f(x) - f(c)$$
$$f'(c)(x-c) + f(c) = f(x)$$

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 $x = x_{n+1}$

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

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 Newtons method ingezet worden, want al deze punten zullen wel op de x-as liggen en dus zullen ze allemaal te vinden zijn met Newtons method.

5.4 Evolutionary improvement

Het leerproces van een systeem zou de evolutie van het systeem genoemd kunnen worden, het leert zichzelf beter te functioneren in een 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.

5.4.1 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.

5.4.2 Een populatie

Wanneer de informatie van een enkel individu telkens wordt doorgegeven aan een volgende generatie die ook bestaat uit slecht n 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 n 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 goed en slecht.

Hoe de overgave van DNA wordt geregeld kan op veel verschillende manieren 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 [12]. 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.

5.4.3 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 hindernis baan over te gaan, maar geen enkel individu heeft in zijn DNA de informatie om te springen staan, 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.

6 Voor welke toepassingen kunnen computerprogramma's, waarin een zelflerend algoritme geïmplementeerd is, gebruikt worden?

7 Welke limitaties hebben zelflerende algoritmes?

8 Bronnen

8.1 Deelvraag 1

[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).

8.2 Deelvraag 2

[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.

8.3 Deelvraag 3

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

[7] 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.

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

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

8.4 Deelvraag 4

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

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

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