Computer vision met Deep Learning

Profielwerkstuk informatica

Door Brian Groskamp en René Emmaneel

Inleiding

Voor ons profielwerkstuk hebben wij gekozen om het over computervisie te hebben.
Wij hebben voor dit onderwerp gekozen omdat kunstmatige intelligentie en zeker computervisie, een tak van kunstmatige intelligentie, in de afgelopen jaren enorme sprongen heeft gemaakt en het ernaar uitziet dat dit door de ontwikkeling van onder andere zelfrijdende auto's de komende jaren alleen nog maar zal toenemen. In dit profielwerkstuk zullen we eerst de theorie over computer vision en over deep learning behandelen. Daarna gaan we zelf computer vision software schrijven en uitleggen.

Onderzoeksvraag

Hoofdvraag

Kunnen we, zonder gebruik te maken van software-libraries, een computer leren cijfers te herkennen?

Deelvragen

1) Hoe kan een computerprogramma leren zien?
2) Hoe werkt deep learning?
3) Hoe kan een computerprogramma trainen om letters en cijfers te herkennen?
4) Hoe kan een computerprogramma een plaatje als input nemen en als output geven wat er in het plaatje staat?

Hypothese

Wij verwachten een computerprogramma te kunnen schrijven wat minstens 40% van de keren succesvol in staat is om de door de gebruiker geschreven letter of cijfer goed te herkennen.

Theorie

Aangezien de theorie achter computervisie en deep learning vrij uitgebreid is, zullen we deze opbreken in de volgende stukken:

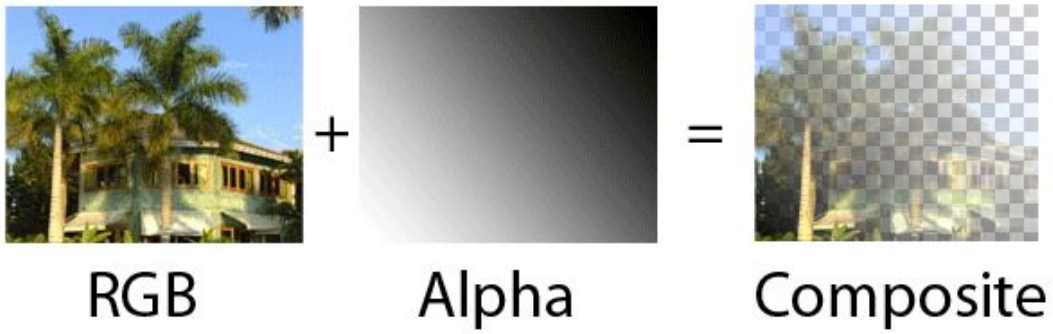
1) Digitale afbeeldingen en hoe een mens en een computer deze zien
2) Patroonherkenning en convoluties
3) Neurale netwerken
4) Deep learning
5) Het trainen van een computermodel om letters en cijfers te herkennen

Het zien van plaatjes

De basis achter digitale afbeeldingen

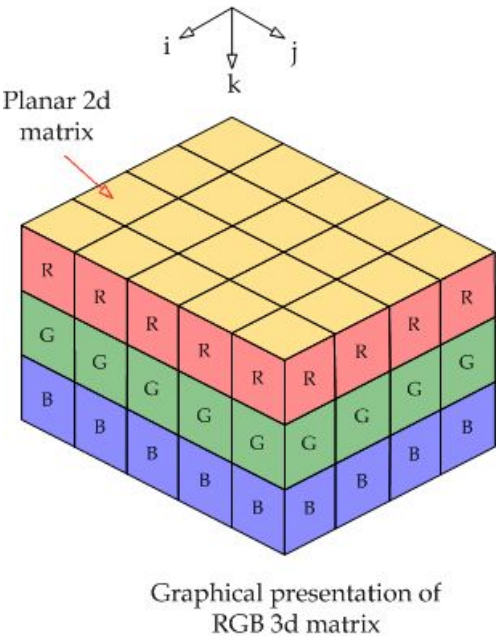
Zoals iedereen weet kan een computer, tablet of smartphone zonder problemen een plaatje weergeven. De computer weet alleen niet wát hij laat zien. Dit komt omdat hij het plaatje compleet anders ziet dan mensen; mensen nemen met hun ogen de kleuren waar en het menselijk brein herkent de patronen en kleuren. Hierdoor kan een mens met gemak herkennen dat in het plaatje hiernaast een kitten te zien is die miauwt en in het gras ligt. Dit herkennen van een object, een handeling en een scène lijkt heel simpel, en dat is het ook voor een mens. Zelfs een peuter kan met gemak in een fractie van een seconde herkennen en beschrijven wat er in het plaatje gebeurt.

Voor een computer ligt dit anders. Een computer ziet een plaatje als een lange rij kleuren die hij in een goede volgorde op het scherm moet weergeven. Hierdoor ontstaat een plaatje wat een mens herkent, maar voor de computer is en blijft het plaatje een lange reeks kleuren. Deze kleuren zijn over het algemeen in het **RGB-kleursysteem**, wat staat voor Rood, Groen, Blauw, of in het **RGBA-kleursysteem**, wat staat voor Rood, Groen, Blauw, Alpha. RGB zijn drie waardes van 0 tot en met 255, die ieder de intensiteit van die kleur aangeven. Zo is de rgb-waarde (255, 0, 0) bijvoorbeeld fel rood en de rgb-waarde (100, 58, 23) oranje. RGBA zijn vier waardes van 0 tot en met 255. De eerste drie waardes geven net als in RGB de kleurintensiteit van van de kleuren rood, groen en blauw aan. De vierde waarde, de Alpha-waarde, geeft de transparantie. Deze waarde zorgt dat de computer weet dat het die kleur mag mengen. Zo kan een een plaatje, zoals in het voorbeeld hieronder, gedeeltelijk (of compleet) transparant gemaakt worden.



Structuur van digitale afbeeldingen

Om het proces van het maken van een systeem dat plaatjes kan herkennen niet nog ingewikkelder te maken dan het al is, zullen we het alpha-kanaal verder buiten beschouwing houden Dit betekent dat we verder werken met plaatjes met RGB waardes. Zoals hierboven al beschreven bestaat een rgb-waarde dus eigenlijk uit drie waardes, in de vorm rgb(Rood, Groen, Blauw) die door de computer in een bepaalde volgorde moeten worden weergegeven. Dit betekent dat een plaatje kan worden gezien als een 3-dimensionale matrix, met de dimensies *breedte, hoogte* en *kleur,* zoals te zien in het plaatje hiernaast. Wanneer we deze 3-dimensionale matrix 'uit elkaar halen' krijgen we drie 2-dimensionale matrices, waarmee we een stuk makkelijker kunnen rekenen.



Graphical presentation of RGB 3d matrix

Net als een mens (of dier) kan de computer niet in 1 keer iets complex herkennen. Denk bijvoorbeeld maar aan een auto: je ziet dat het vier wielen heeft, het een stuur heeft en ramen. Door deze kenmerken (de vier wielen, het stuur en de ramen) te combineren kan je tot de conclusie komen dat hetgeen wat je ziet zeer waarschijnlijk een auto is. Als het ook nog erg groot zou zijn, kan je tot de conclusie komen dat het een bus is en als het maar twee wielen en geen ramen had zou je herkennen dat het een motor is. De mens leert dus niet zozeer wat een auto is, maar leert een simpele beschrijving van hoe een auto eruit moet zien (de vorm, vier wielen, een stuur en ramen).

Een computer laten zien

Convoluties

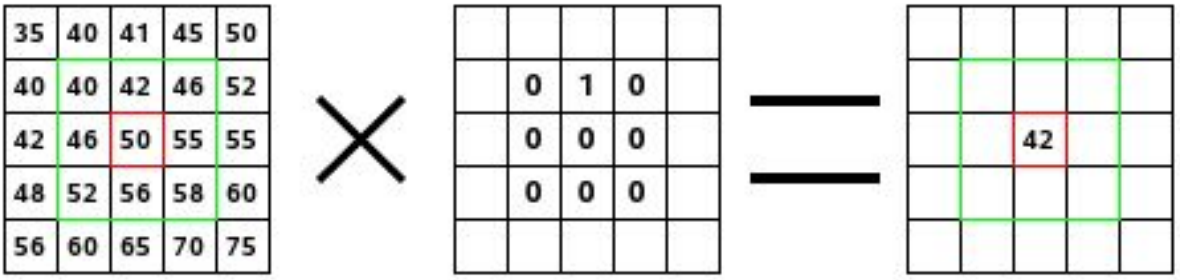
We weten dus hoe een mens ziet en leert om dingen en voorwerpen te herkennen. Aangezien we weten dat deze manier al duizenden jaren perfect werkt, ligt het voor de hand om ons computerprogramma op dezelfde manier te laten werken.

Het doel is dus om algoritmes te schrijven die in de 3-dimensionale matrix van een plaatje patronen, vormen en andere eigenschappen herkent zodat de computer weet wat zich in het plaatje bevindt. Echter, uit 1 pixel kan een computer niks concluderen over de inhoud van de rest van de afbeelding. Om patronen en vormen in een afbeelding te herkennen zal voor elke pixel rekening gehouden moeten worden met de pixels die zich rond deze pixel bevinden. Al sinds de jaren 60 wordt er onderzoek gedaan naar de beste manier op dit te doen, maar op het moment is de best presterende manier het gebruik van convoluties.

Een convolutie is in de kunstmatige intelligentie het samenvoegen van informatie uit meerdere pixels in 1 nieuwe pixel op de feature map. Dit werkt als volgt:

- Je maakt een nieuwe 2-dimensionale matrix van (meestal) 5 bij 5. Deze nieuwe matrix heet ook wel de ***image kernel*** of ***convolution kernel***.
- Je vult deze image kernel met waardes volgens het model dat je gebruikt (over dit model meer in het stuk 'Het trainen van een computermodel').
- Je plaatst de image kernel bovenop een van de drie 2-dimensionale lagen van het plaatje (het rode, groene of blauwe kanaal). Deze laag waar we de image kernel op plaatsen heet de ***receptive field***.
- Je vermenigvuldigd elke waarde van de image kernel met de waarde van de direct daaronder liggende waarde van de receptive field. De som van al deze vermenigvuldigingen is de uitkomst die je zocht. Dit vermenigvuldigen en optellen heet een ***convolutie***.
- Deze uitkomst sla je op in een nog een nieuwe 2-dimensionale matrix. Deze matrix gevuld met uitkomsten van de convoluties heet de ***feature map***. Hoewel dit een matrix van dezelfde grootte is als het bronplaatje en het gevuld is aan de hand van de pixels van het bronplaatje is een feature map zelf geen plaatje. Het is een 2-dimensionale matrix gevuld met waardes die de omgeving van pixels beschrijft.
- Je verplaatst de image kernel (meestal) 1 plekje naar rechts of naar onder, zodat je uiteindelijk elke mogelijke positie van de image kernel op het receptive field gehad hebt. De uitkomst die je op deze plek krijgt zet je ook in de feature map. De afstand waarmee je de image kernel telkens verplaatst heet de ***stride***.

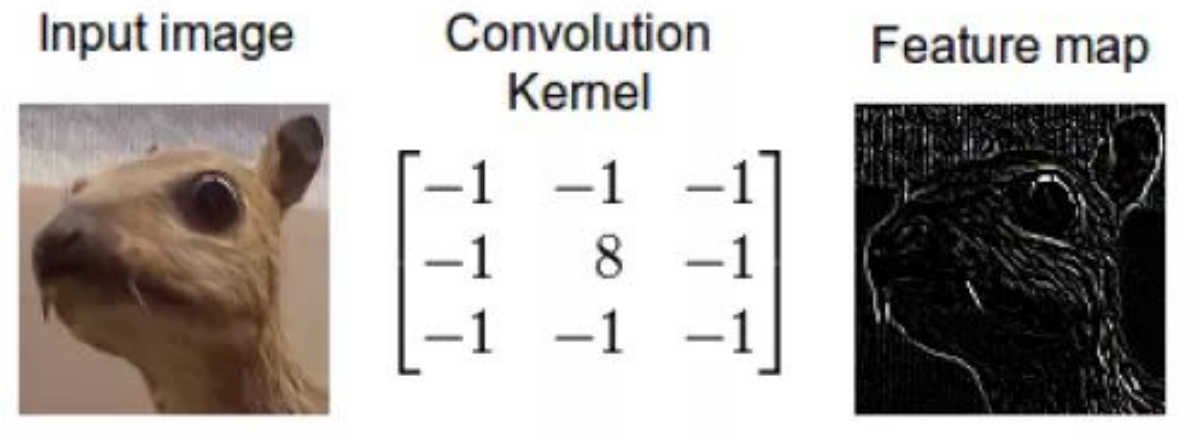
Een convolutie ziet er afgebeeld als volgt uit:



Dit geeft de volgende berekening: *40x0 + 42x1 + 46x0 + 46x0 + 50x0 + 55x0 + 52x0 + 56x0 + 58x0 = 42*.

Hierin is de linker matrix de ***receptive field***, de middelste matrix een 3 bij 3 ***image kernel*** en de rechter matrix een ***feature map***.

In de feature map staat op de middelste positie nu een waarde van 42. Deze waarde betekent nu niet meer hoeveel rood, groen of blauw er op deze positie in het oorspronkelijke plaatje hoort, maar geeft een verband aan tussen de kleur van de oorspronkelijke pixel en de kleur van de 8 pixels daaromheen. In deze waarde in de feature map is nu dus informatie opgeslagen over niet alleen de pixel, maar ook over de omgeving van de pixel.



Wanneer we een betere convolution kernel gebruiken is duidelijker te zien hoe het gebruik hiervan helpt bij het herkennen van vormen en patronen.
In bovenstaand voorbeeld is het resultaat van een convolution kernel op een plaatje te zien. In de feature map zie je dat de randen en lijnen in het plaatje wit (hoge waardes) zijn geworden en de rest zwart (lage waardes). In feite kan de computer hier dus een belangrijke eigenschap, de randen (en daarmee de vorm van het object in het plaatje) zien. Door andere convolution kernels toe te passen zou bijvoorbeeld specifiek een cirkel, een rechte hoek of een rechte lijn herkend kunnen worden en door het gebruiken van meerdere, verschillende convolution kernels kunnen meer eigenschappen van het plaatje uitgelezen worden.

Neurale netwerken

Het verzamelen van zoveel mogelijk eigenschappen

Zoals in het stuk over convoluties beschreven staat is het zeer goed mogelijk om bepaalde eigenschappen uit een plaatje te halen en hieruit een conclusie te trekken. Het uitvoeren van 1 convolutie is alleen (bijna) nooit genoeg om met enige zekerheid te kunnen herkennen wat zich in een plaatje bevindt. Een voorbeeld: wanneer je met een convolutie naar rechte, verticale lijnen zoekt om flatgebouwen te onderscheiden van katten zal dat prima gaan. Een kat heeft immers veel minder van deze lijnen dan een flatgebouw, waardoor er een duidelijk verschil is voor de computer. Maar wat als we de computer een plaatje van een vuurtoren geven? Hoewel dit absoluut niet op een flatgebouw lijkt zal de computer hier toch een flatgebouw in kunnen zien, omdat er verticale rechte lijnen in voorkomen. Om dit soort false-positives te voorkomen zullen we de computer daarom naar meer eigenschappen moeten laten zoeken. De beste en meest geoptimaliseerde manier om dit te kunnen doen is met een **convolutioneel neuraal netwerk (CNN)**. Om een CNN goed te begrijpen is het echter handig om eerst te kijken naar een normaal neuraal netwerk.

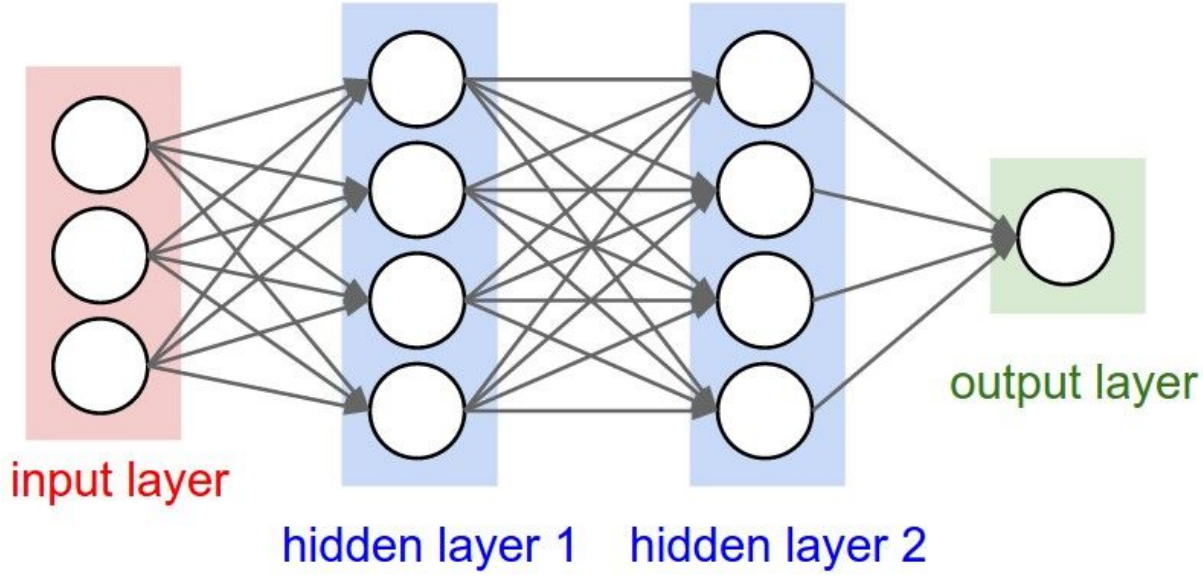
Neurale netwerken

Een neuraal netwerk is een grote functie die probeert de werking van een brein te simuleren (vandaar ook de naam neuraal netwerk). In het neuraal netwerk zijn daarom, net als in een brein, neuronen en synapsen (verbindingen tussen de neuronen). Elke neuron is een waarde en welke verbinding tussen de neuronen is een vermenigvuldiging met een unieke **weight** (W) en een eventuele optelling/aftrekking met de **bias** (B). De werking van dit netwerk is als volgt:

- het neurale netwerk krijgt een grote set input-data en weights en biases binnen
- de input-data wordt via een vrij complexe set aan berekeningen verwerkt
- het neurale netwerk geeft een antwoord van een of meerdere waardes; vaak zijn dit scores die aangeven hoe groot de kans ergens op is (bijvoorbeeld: *er is 87% kans dat in dit plaatje een banaan te zien is*).

Dit is een neuraal netwerk

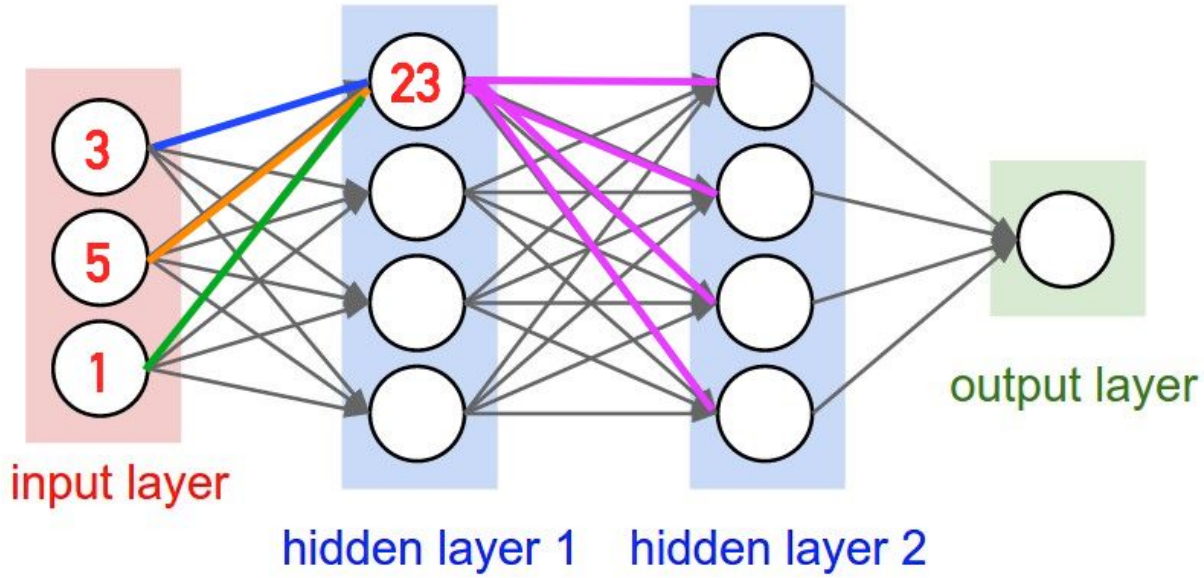
Een standaard neuraal netwerk kan je op de volgende manier weergeven:



Hierin is elke witte cirkel een waarde en is elke pijl een verbinding met een functie in de vorm van F(x) = Wx + B. Elke witte cirkel krijgt de waarde van de som van alle functies die er naar toe lopen, waardoor dit een behoorlijk complexe berekening kan worden. Een hidden layer is

elke laag die tussen de input layer en output layer zit en elke hidden layer is als het ware een tussenstap die kan helpen om complexere dingen te berekenen. Met elke extra hidden layer wordt het netwerk dus ook complexer en kunnen er ingewikkeldere dingen berekend worden.

Om het wat duidelijker te maken zullen we een voorbeeld geven met een van de simpelst mogelijke neurale netwerken:



De input van het neurale netwerk zijn de waardes 3, 5, en 1. Elk van deze waardes heeft precies vier verbindingen, aangezien hidden layer 1 vier neuronen heeft. Elk van deze verbindingen is een functie met een weight en een bias in de vorm van F(x) = Wx + B.
Om het wat eenvoudiger te houden zullen we het voorbeeld houden over alleen de bovenste neuron van hidden layer 1. Naar deze neuron lopen 3 verbindingen, 1 voor elke neuron in de laag ervoor. Voor ons voorbeeld gebruiken we de volgende functies:

- F(x) = 2x + 1
- G(x) = 4x - 3
- H(x) = -x + 0

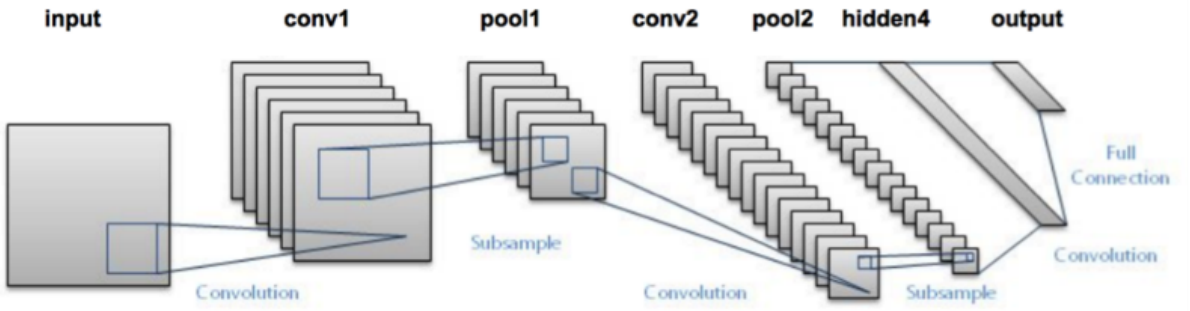
Alle drie de waardes in de input gaan dus via hun eigen functie naar de bovenste neuron in hidden layer 1. De waarde van deze neuron wordt dus:

F(3) + G(5) + H(1) =
(2*3 + 1) + (4*5 - 3) + (-1 + 0) = 23

In de bovenste neuron van hidden layer 1 komt dus de waarde 23. Deze neuron geeft deze waarde ook weer door aan de volgende vier neuronen in hidden layer 2, via verbindingen die ook weer functies met eigen weights hebben.

Convolutioneel neuraal netwerk

Een convolutioneel neuraal netwerk lijkt qua structuur sterk op een normaal neuraal netwerk, alleen gebruiken we (zoals de naam al suggereert) convoluties en nog een aantal andere tussenstappen. Een bekend CNN is LeNet-5 en deze ziet er als volgt uit:



Hoewel het er heel anders uitziet dan een normaal neuraal netwerk, volgt een CNN hetzelfde basisprincipe van neuronen en synapses. De grootste verschillen zijn dat waar een normaal NN simpele F(x) = Wx + b verbindingen gebruikt een CNN complexere berekeningen gebruikt en dat waar een neuron in een normaal NN 1 waarde bevat, deze in een CNN veel meer waardes kan bevatten, in de vorm van een feature map.
Verder gelden belangrijke principes die voor een normaal NN gelden, zoals dat elke extra hidden layer meer complexiteit kan toevoegen, ook voor een convolutioneel neuraal netwerk.

Om iets meer duidelijkheid te scheppen in de werking van het CNN wat in het plaatje hierboven staat, zullen we alle stappen beschrijven en uitleggen:

- Het CNN leest een plaatje in (en verkleint deze als deze te groot is naar 32x32 pixels).
- Er worden 6 feature maps gecreëerd door middel van convolution kernels van 5x5 met een stride van 1. Hierdoor ontstaan feature maps van 28x28.
- Deze feature maps worden geactiveerd met een **activation function**. Hierover hieronder meer.
- De feature maps worden vervolgens gesubsampled door middel van **max pooling**. Hierdoor worden de feature maps verkleind tot 14x14. Hierover hieronder meer.
- De 6 feature maps worden op elkaar gelegd en samengevoegd en hieruit worden met 16 verschillende convolution kernels van 5x5 met een stride van 1 weer 16 verschillende feature maps gecreëerd. De feature maps zijn hier nog maar 10x10.
- De 16 nieuw ontstane feature maps worden geactiveerd met de activation function.
- De nieuwe feature maps worden ook gesubsampled door middel van max pooling. Hierdoor zijn de featuremaps nog maar 5x5.
- Om van de 16 feature maps van 5x5 worden alle feature maps aan elkaar gelegd om 1 grote feature map van 80x5 te krijgen.
- Over deze grote feature map wordt een convolutie van 5x5 met een stride van 1 uitgevoerd. Hierdoor is de feature map nog maar 76x1.
- Deze laatste grote feature map wordt weer geactiveerd.

- Als laatste stap wordt deze feature map door een **fully connected layer** gehaald, waardoor er een uitkomst uit komt. Hierover hieronder meer.

Een belangrijke functie die niet in het model in dit plaatje te zien is maar die wij zelf wel gebruiken is **dropout**. Ook hierover valt hieronder meer te lezen.

Hoewel een CNN dus enigszins zeker op een normaal NN lijkt, zijn er dus grote verschillen in het aantal berekeningen en wat voor berekeningen. De basis van een NN, allemaal neuronen die allemaal met elkaar verbonden zijn via berekeningen, komt echter wel duidelijk terug in een CNN.

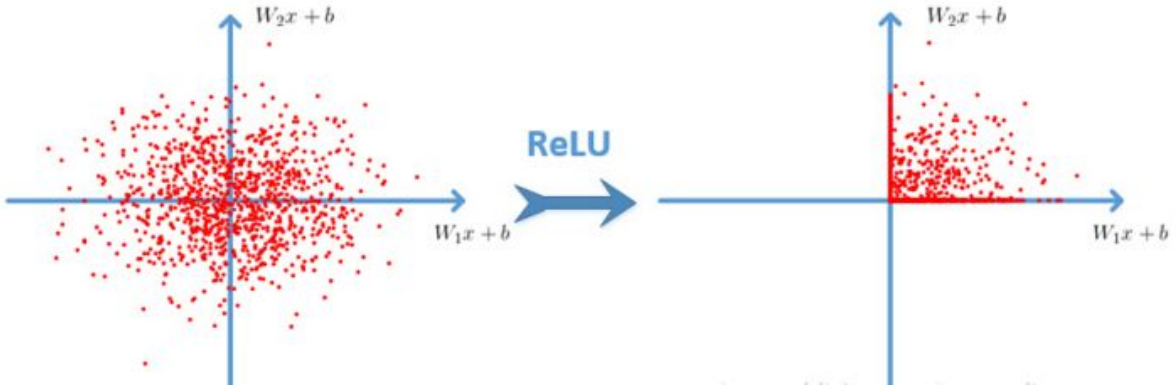
Activation functions

Activation functions zijn in een neuraal netwerk heel belangrijk omdat ze zorgen voor non-lineariteit. Wanneer deze functies niet aanwezig zouden zijn, kan een neuraal netwerk nooit meer dan 1 laag zijn aangezien je de lineaire functies (zoals de convoluties) gewoon als 1 functie zou kunnen schrijven.

Er zijn verschillende activation functions maar aangezien wij de Rectified Linear Unit (ReLU) gebruiken, de op het moment populairste activation function, en dit ook de simpelste is zullen we ons op deze focussen. De ReLU-functie ziet er als volgt uit:

$$\mathbf{F(x) = max(0,x)}$$

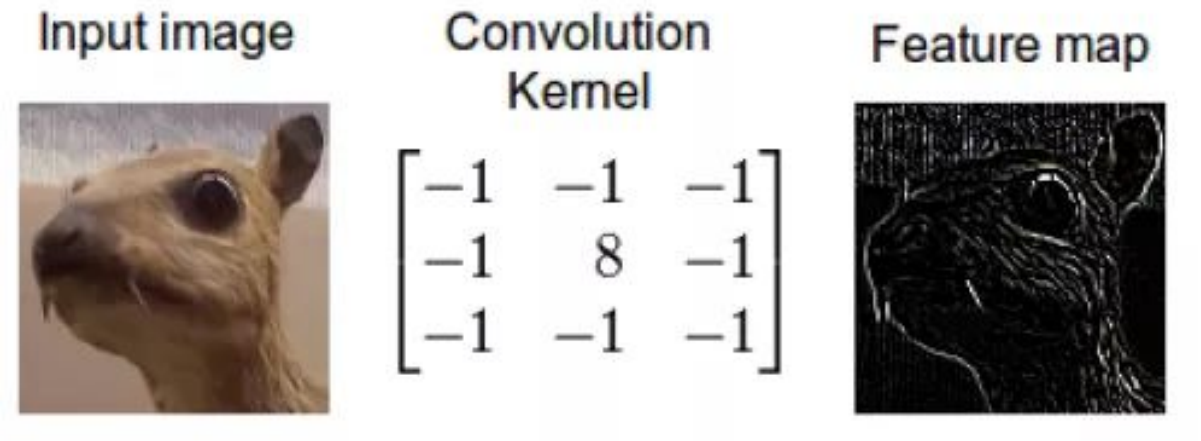
De functie returned dus 0 als de input-waarde lager dan of gelijk is aan 0 en returned de input-waarde wanneer deze groter dan 0 is. Dit heeft het volgende effect op de waardes:



Door deze functie na elke hidden layer toe te passen heeft het door de toegevoegde non-lineariteit dus ook daadwerkelijk nut om meerdere hidden layers in het netwerk te

gebruiken, waardoor het netwerk vele malen complexer kan worden en dus ook vele malen beter kan worden.

Max pooling



Zoals al beschreven in het stuk over convoluties is het herkennen van bepaalde vormen en eigenschappen in een plaatje zeer goed te doen. Zoals in het plaatje hierboven al te zien is, lijkt de feature map die we met deze convolutie gemaakt hebben sprekend op het dier in het originele plaatje. Zo sprekend zelfs dat je aan de feature map direct kan zien dat het uit dit specifieke originele plaatje komt. Als we hiermee de computer al zouden leren hoe het dier in het originele plaatje eruit ziet, zou hij het geweldig doen. Totdat we een nieuw plaatje van hetzelfde dier zouden geven.

Door zulke precieze feature maps te maken lopen we namelijk tegen een probleem aan: ***overfitting***. Overfitting is een term uit de computerwetenschappen en statistieken en betekent dat het model wat je maakt te precies bij je trainingsdata past en daardoor in de problemen raakt wanneer het nieuwe data ziet. In plaats van een grove beschrijving van het dier hebben we nu een heel exacte beschrijving van dit specifieke plaatje. Om dit te voorkomen zullen we de feature map ***max poolen*** (ook wel bekend als ***subsamplen***).

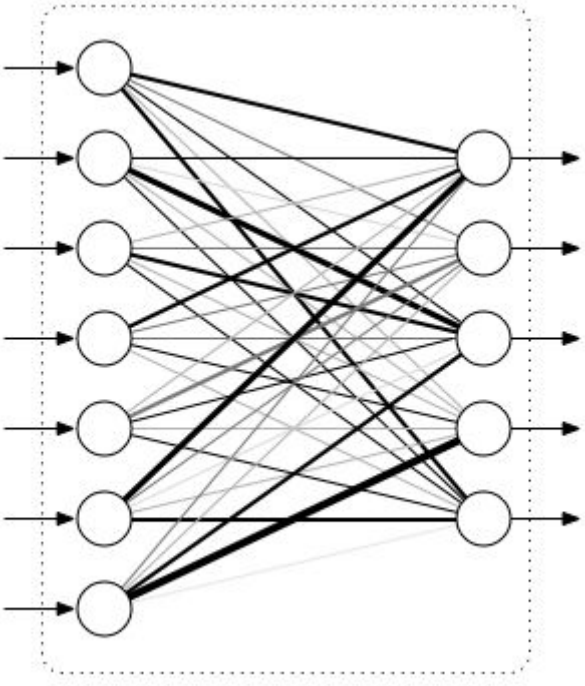
Bij max pooling verwijderen we details zonder de grove eigenschappen zoals vorm kwijt te raken. Dit doen we door ook weer een nieuwe 2-dimensionale matrix van (meestal) 2 bij 2 te maken en deze met een stride van (meestal) 2 over de feature map te verplaatsen. Alleen waar we bij een convolutie zouden vermenigvuldigen en optellen gebruiken we bij max pooling simpelweg de hoogste waarde. Dit ziet er afgebeeld als volgt uit:

Doordat een groot deel van de details hiermee verloren gaan, kijkt de computer als het ware als iemand met een verkeerde bril: een hoop details verdwijnen, maar kleur en ruwe vormen blijven zichtbaar.

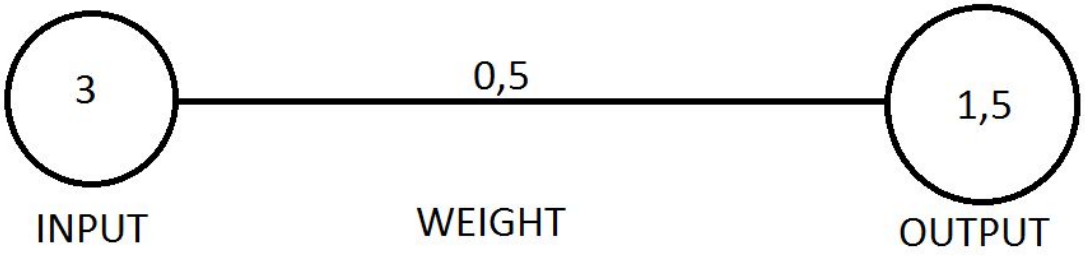
Als extra voordeel heeft max pooling dat de grootte van de feature maps afneemt waardoor het aantal berekeningen die in de rest van het CNN nodig zijn afneemt en het zo de werking van het CNN aanmerkelijk versneld.

Fully connected layer

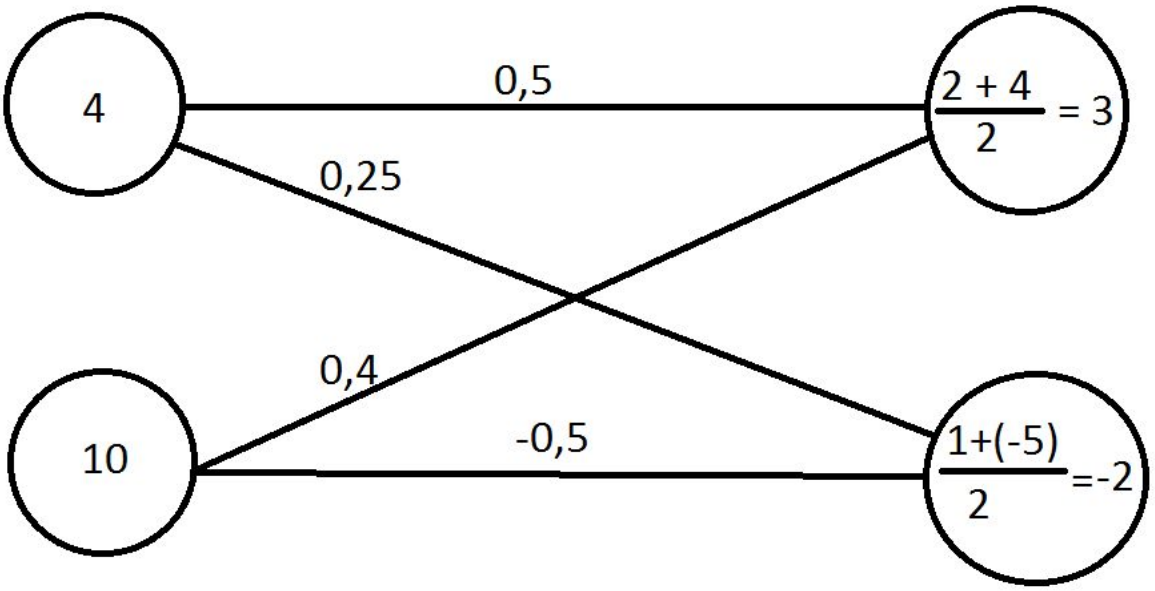
Een **fully connected layer** is de laatste laag in een CNN en is verantwoordelijk voor het omzetten van de resultaten van het CNN naar een antwoord. De fully connected layer is weer een grote functie, die als input een rij getallen neemt en een grotere rij met weights. En als output geeft het een rij getallen met een gewenste grootte.



Hierboven zie je een voorbeeld hoe een fully connected layer eruit ziet.
Een fully connected layer zet dus een array (een rij) met getallen om in een andere array van een gewenste grootte met getallen. De grote van deze arrays zijn als eerste belangrijk. Aan de linkerkant heb je alle pixels van alle plaatjes die uit de convulutie laag komt. Bijvoorbeeld als er 6 plaatjes met elk plaatje 3x3 pixels zijn er 6x3x3 = 54 input getallen. Deze getallen heten nodes.
Aan de rechterkant, de output, staat weer een array met getallen, ook deze getallen heten nodes. Elke input node is verbonden met een output node, en in elk van deze verbindingen staat een weight.



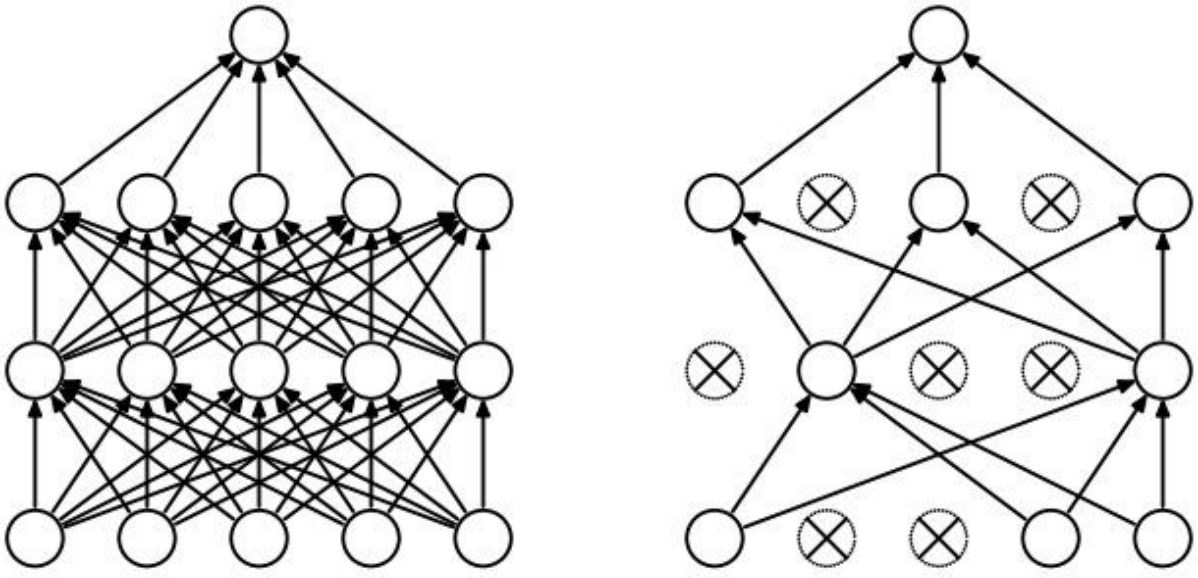
Hier is 1 verbinding uitvergroot, de input staat aan de linkerkant, die wordt vermenigvuldigd met de weight (elke verbinding heeft een unieke weight) en dat wordt in de output node gestopt.
Elke input gaat dus naar elke output. Dit zorgt ervoor dat de outputs meerdere getallen binnenkrijgen. Elke output neemt daarom het gemiddelde van de getallen die hij gekregen heeft. Hieronder staat een simpel voorbeeld met 2 inputs en 2 outputs, en dus 2x2=4 weights.
Een weight kan inprincipe elk getal zijn, maar zit meestal tussen -1 en 1



Hier zijn de inputs dus [4, 10], de weights [0,5 ; 0,25 ; 0,4 ; -0,5] en de outputs [3, 2]
Een fully connected layer verbind dus elke input met elke output, en bij elke verbinding zit dus een weight. Het aantal weights is dus het aantal inputs vermenigvuldigd met het aantal outputs.

Dropout

Dropout is net als max pooling een techniek die gebruikt wordt om overfitting tegen te gaan. Waar max pooling de data als het ware vager maakte, voegt dropout een soort ‘ruis’ toe.



(a) Standard Neural Net (b) After applying dropout.

Dit doet het door op willekeurige plekken op willekeurige plekken de waarde van een neuron op 0 te zetten. Alle functies die vanaf deze neuron lopen zullen dus als input 0 krijgen en elke convolutie die vanaf deze neuren loopt zal dus ook 0 als antwoord krijgen. Het is dus als het ware alsof deze neuron uitgezet is.

Dropout is een erg succesvolle manier om met een kleinere hoeveelheid aan trainingsdata toch nog goede resultaten te krijgen, omdat het door het willekeurig uitzetten van willekeurige neuronen zorgt voor constante variatie in de trainingsdata. Dit zorgt ervoor dat het netwerk zo heel moeilijk 'grip' kan krijgen op de constant veranderende trainingsdata en de kans op overfitting dus veel kleiner wordt.

Deep learning

Nu je weet wat een neuraal netwerk is kunnen we eindelijk naar de deep learning gedeelte gaan. Tot nu toe zijn we vanuit gegaan dat we de weights al hebben, maar dat is natuurlijk niet zo. De weights moeten nauwkeurig uitgekozen worden om te zorgen dat uit de berekeningen in het midden de juiste output komt. Zelf kiezen is onmogelijk in grotere netwerken, omdat er duizenden tot miljoenen weights gekozen kunnen worden en de berekeningen enorm groot zijn. Hoe dit gekozen wordt heet deep learning.

Dit gaat met simpel weg de computer waardes voor de weights laten testen met **training sets**, deze training sets zijn inputs met daarbij al gegeven wat de output moet zijn. Door deze inputs te testen met willekeurige weights en kijken hoe goed de outputs zijn kunnen nieuwe testweights worden bepaald. 1 cyclus van nieuwe weights genereren, weights testen en kijken hoe de weights aangepast moeten worden is 1 **generatie**. Na elke generatie worden de weights in stukje in de juiste richting aangepast, en zal de outputs dichter bij de juiste outputs zitten.

Wanneer er lange tijd geen verbeteringen meer tussen de generaties zitten moet de deep learning process gestopt worden. Hierna geeft de code 1 set met weights, en dit zijn de uiteindelijke weights die je kan gebruiken voor je neuraal netwerk. Wanneer dan een nieuwe input wordt gestuurd met die juiste weights zal het resultaat vrij accuraat moeten zijn.

Deep learning bestaat dus uit de volgende stappen:
1. WIllekeurige **startweights** kiezen
2. Een van de inputs van de input set kiezen.
3. De input en weight door het neurale netwerk halen, zodat je een output krijgt die je kan vergelijken met de van te voren bepaalde outputs
4. De **afgeleide** berekenen doormiddel van **backpropagation**
5. Doormiddel van de afgeleide en hoe goed de gevonden outputs bij de gegeven output passen, de startweights aanpassen
6. Herhaal stap 2 tm 4 totdat je tevreden bent met het resultaat.

Wat houdt een classificatie netwerk in

Een classificatie netwerk is een neuraal netwerk met als doel een **label** te geven aan een input. Dit is bijvoorbeeld een netwerk die kan bepalen welke letter de input is, of welk soort fruit. Een banaan is in dat geval een label. Voordat het neuraal netwerk wordt gemaakt moeten labels gemaakt zijn, bij een netwerk om getallen te herkennen zijn de labels dus 0, 1, 2, ... , 9

Elke label moet een vector krijgen, dat is een array die bestaat uit een 1 voor de correcte waarde en een 0 voor alle andere waardes. Een 0 bestaat dus uit een array met een 1 voor het eerste element en een 0 voor alle andere elementen:

0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
etc.

Dit is nodig om te kunnen bekijken hoe ver de programma van een bepaalde label af zit. Als uit een berekening van een neuraal netwerk [12; 25; 4] komt, zal dat dicht bij de label met als waarde [0, 1, 0] zitten, want het 2e getal is het hoogste. Hoe ver dit precies van elkaar af zit kan worden berekend met een combinatie van **softmax** en **cross entropy**. Dit zijn 2 relatief simpele wiskundige berekeningen die als waarde een getal geven. Dit getal wordt dan dus gebruikt om te kijken hoe goed de weights presteren, waar kleiner beter is.

Softmax

Softmax neemt een array met getallen en geeft als output een nieuwe array met getallen. Dit doet het algoritme met de volgende berekeningen.

orgineelArray = [O1, O2...]
softmaxArray = [S1, S2...]
softmaxArray[i] = e^orgineelArray[i] / (e^orgineelArray[0] + e^orgineelArray[1] + …)
waar n de grote van orgineelArray is

Voorbeeld:
orgineelArray = [2, 5, 3]
softmaxArray[0] = e^2 / (e^2 + e^5 + e^3) = 0,042
softmaxArray[1] = e^5 / (e^2 + e^5 + e^3) = 0,844
softmaxArray[2] = e^3 / (e^2 + e^5 + e^3) = 0,114
softmaxArray = [0,042; 0,844; 0,114]

Het nuttige hiervan is dat de elementen van de softmaxArray bij elkaar precies 1 zijn. Dit is handig voor de cross entropy, omdat dan ook wordt vergeleken met een array met de som van de elementen van 1.

Ook worden de verhoudingen min of meer behouden. Als in orgineelArray het tweede getal het grootste is, zal dat in softmaxArray dat ook zo zijn.

Cross Entropy

Cross entropy vergelijkt de softmaxArray met de array van de bijbehorende label
Een cross entropy functie heeft de volgende algoritme.

softmaxArray = [O1, O2...]
labelArray = [L1, L2...]
crossEntropy = -log(softmaxArray[0]) * labelArray[0] + log(softmaxArray[1]) * labelArray[1]+...

Voorbeeld:
softmaxArray = [0,042; 0,844; 0,114]
labelArray = [0 , 1, 0]
crossEntropy = -log(0,844) = 0,07365

Hier is enkel log(0,844) berekend, de andere logs worden namelijk met 0 vermenigvuldigd dus die zijn niet belangrijk om te weten.

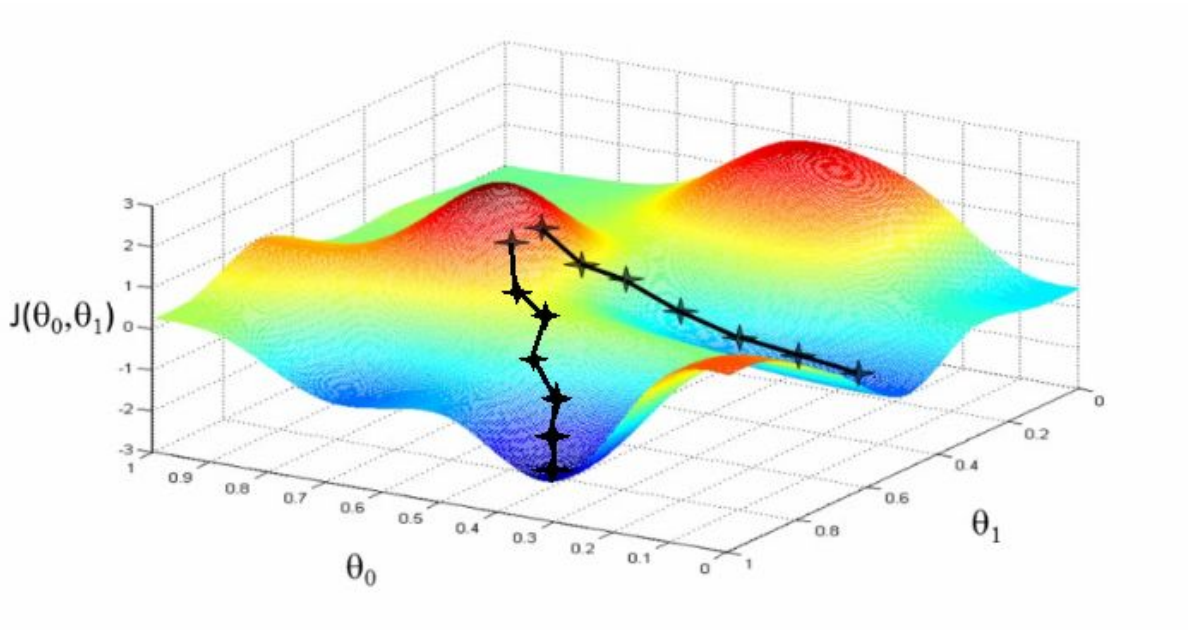
Als er willekeurige weights worden gebruikt, is de gemiddelde cross entropy dus -log(1/n) waar n het aantal labels zijn. Bij een netwerk met 10 labels zal de gemiddelde cross entropy dus ongeveer -log(0,1) = 1 zijn.

Uit het neurale netwerk komt dus een array met getallen, wat omgezet wordt tot een cross entropy score, hiermee worden de weights voor de volgende generatie gegenereerd.

Local minimum

Stel, je staat op een willekeurig punt op een berg en je wilt naar beneden lopen, hoe kan je dit doen? 1 manier om dit te doen klinkt heel stom maar het werkt. Je kijkt welke kant direct voor je omlaag gaat. Daarna zet je een stap in die richting. Dit herhaal je een paar honderd keer totdat je op het laagste punt bent, je bent dan op een **local minimum**. Een local minimum is dus een punt waar alle punten die er omheen liggen hoger liggen dan dat local

minimum.



Hierboven staat een wiskundig voorbeeld van een berg, de rode plekken zijn hoog en de blauwe plekken zijn laag. Door elke keer om je heen te kijken en een stap te zetten naar een lager gelegen gebied kom je uiteindelijk dus uit in een local minimum. Er kunnen meerdere local minimums zijn, en het ligt er ook aan waar je begint voor in welk local minimum je terecht komt. In dit plaatje zie je 2 startpunten die dicht bij elkaar liggen, maar de 2 punten waar ze naar toe gaan liggen ver uit elkaar.

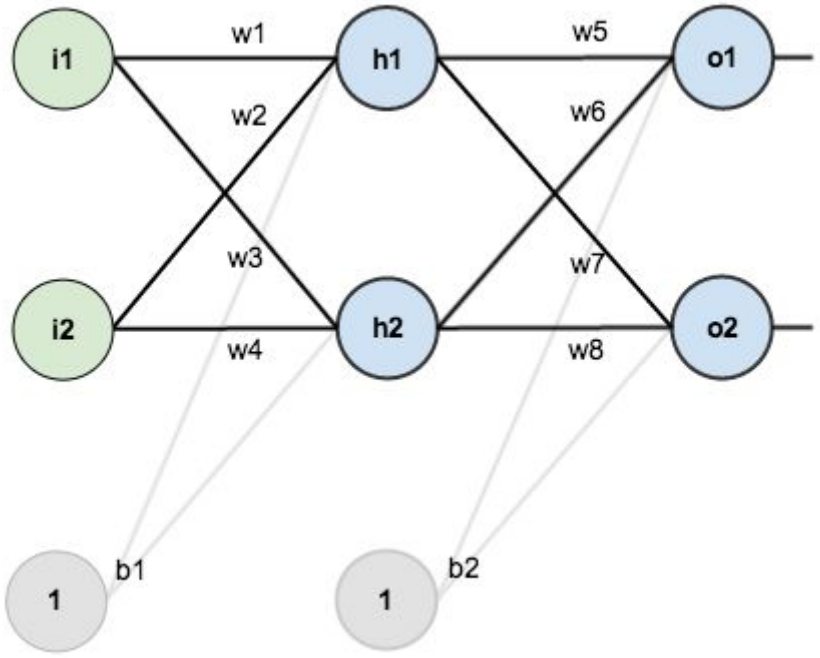
Het plaatje hierboven is eigenlijk een voorstelling van een simpel neuraal netwerk. Er zijn 2 weights θ0 en θ1. Deze zijn aangegeven aan de 2 assen. Er is ook een 3e as, J(θ0,θ1), dit is de cross entropy score van het model wanneer de 2 weights aangegeven op de assen worden gebruikt. Hier kan je zien dat sommige weights een grote cross entropy score geeft en andere weights een lage cross entropy score geeft. De bedoeling van deep learning is de weights zo bepalen zodat de cross entropy score zo laag mogelijk is, en dit gebeurd door **gradient descent**. Gradient descent is dus de weights elke keer een klein beetje aanpassen in de richting van de helling van deze zogenaamde berg. De afgeleide van de cross entropy score ten opzichte van de weights moet dus berekent worden. Hoe dit gedaan wordt wordt in het volgende deelonderwerp uitgelegd.

In het plaatje is het makkelijk te zien waar de local minimums zijn, maar dit is niet altijd het geval, in grote neurale netwerken kunnen duizenden tot miljoenen weights gebruikt worden. Maar de manier om een local minimum te vinden is alsnog hetzelfde. Je begint op een bepaald punt, je kijkt om je heen en je zet een stap richting het laagste punt. Hieronder is elke stap met wat meer details uitgelegd.

Backpropagation

Zoals net uitgelegd moet dus de afgeleide van de cross entropy score ten opzichte van de weights worden berekend. Omdat er vele duizenden weights zijn en een neuraal netwerk

meestal een erg ingewikkeld wiskundig systeem is kan dit niet altijd makkelijk worden berekend. Het wordt berekend door middel van backpropagation.



Hier zie je een voorbeeld van een erg simpel neuraal netwerk. Dit netwerk heeft 2 input nodes, 2 hidden nodes en 2 output nodes. Bij elke laag zit er ook 2 bias nodes die altijd een waarde van 1 hebben. Met deze 2 output nodes kan de cross entropy score worden berekend. Nadat je inputs hebt gegeven, weights hebt gegeven, alle nodes hebt berekent en daarna de cross entropy score hebt berekend kan backpropagation worden toegepast. Dit werkt van achteren naar voren. Eerst wordt bepaald welke verandering w5, de eerste weight van de laatste layer, de totale (cross entropy) score zal veranderen. Gevraagd is dus de afgeleide van de totale score ten opzichte van w5.

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

Fig x. De kettingregel

De kettingregel zegt dat de afgeleide van de totale score ten opzichte van w5 gelijk is aan het product van alle tussengelegen afgeleiden. Deze afgeleides zijn individueel te berekenen. Als deze individuele afgeleides berekent zijn, is de totale afgeleide niet moeilijk meer. Hoe de individuele wiskundige berekingen moeten hangt erg van het neuraal netwerk zelf af.

De berekeningen voor de weights die eerder in het neuraal netwerk worden gecompliceerder. Dezelfde kettingregel geldt, maar de eerste term van de kettingregel wordt lastiger te berekenen. Normaal moet de afgeleide van het totaal ten opzichte van de waarde van de laatste node berekend worden, maar nu wordt de afgeleide van het totaal ten opzichte van de eerst opvolgende hidden node. Omdat de hidden node effect heeft op elke laatste node,

moet het individuele effect van de hidden node op eind nodes eerst berekend worden. Oftewel de afgeleide van het totaal ten opzichte van de hidden node is gelijk aan de afgeleides van elke eindnode ten opzichte van de hidden node.

$$\frac{\partial E_{totaI}}{\partial w_1} = \frac{\partial E_{totaI}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$
$$\frac{\partial E_{totaI}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Fig x. De berekening voor eerdere weights

Voor berekeningen voor eerdere lagen moet zowel het effect dat de hidden node op een hidden layer heeft berekenen, als het effect dat de verandering van die hidden layer op de eindnodes heeft berekenen. Dit zorgt ervoor dat het systeem al heel snel heel veel berekeningen moet doen.

De stappen van een deep learning systeem

Zoals eerder gezegd bestaat een deep learning systeem uit de volgende stappen:
1. WIllekeurige startweights kiezen
2. Een van de inputs van de input set kiezen.
3. De input en weight door het neurale netwerk halen, zodat je een output krijgt die je kan vergelijken met de van te voren bepaalde outputs
4. De afgeleide berekenen doormiddel van backpropagation
5. Doormiddel van de afgeleide en hoe goed de gevonden outputs bij de gegeven output passen, de startweights aanpassen
6. Herhaal stap 2 tm 4 totdat je tevreden bent met het resultaat.

Hieronder staat elke stap in meer detail uitgelegd.

Stap 1 in veel deep learning systemen is een (willekeurig) model genereren. Een model is dus een set met veel weights bij elkaar. Voor elke weight die je moet gebruiken voor de input van je neuraal netwerk moet je een getal kiezen. Dit kan bijvoorbeeld altijd 0 zijn, of een willekeurig getalen tussen -1 en 1. Meestal is het niet super belangrijk om hier veel aandacht aan te besteden, omdat een deep learning systeem dit vanzichzelf zou moeten oplossen, maar je kan het systeem altijd een handje helpen door al een redelijk model als start punt te geven.
Je kan dit (willekeurig) model vergelijken met een willekeurig punt op de berg waar je van af wilt. Dit punt waarmee je start heet het **start model**

Stap 2 is het kiezen van een input en de daarbij horende output uit de van te voren bepaalde trainingsset. Bij elke input in de trainingsset is al de juiste output bekend. Dit is bijvoorbeeld een plaatje van een 0 en de output is dat het ook daadwerkelijk een 0 is.

Stap 3 is om het neurale netwerk uit te voeren. Als input wordt dus de input die bij stap 2 bepaald is genomen en als weights wordt het start model gebruikt. Na het uitvoeren van het neurale netwerk en het toepassen van een softmax functie en een cross entropy functie kan

er worden berekend hoe goed de willekeurige weights zijn. Dit wordt gebruikt bij de volgende stap, de backpropagation.

Stap 4 is de waardes waarmee je elke individuele weight mee gaat veranderen berekenen door middel van backpropagation. De wiskunde achter backpropagation verschilt per neuraal netwerk, en kan voor ingewikkelde netwerken erg lastig worden. Backpropagation is voor een neuraal netwerk met veel hidden layers meestal de stap die het langst duurt om door een computer berekend te worden.

Stap 5 is het toepassen van de waardes die in stap 4 gevonden zijn. Voor elke weight is nu de afgeleide gevonden, waarmee het model verbeterd kan worden. Hierna wordt de afgeleide nog met de **alpha** vermenigvuldigt, de alpha is een waarde bedacht door de makers van het deep learning model. Dit kan dus elke waarde hebben die je maar wilt. Hoe hoger de alpha hoe groter de stappen. Het is nuttig om de alpha zelf aan te kunnen passen omdat bepaalde modellen grover of juist minder grof getrained moeten worden

Dus de formule voor het start model van elke weight van dat model is:

weight := -afgeleide * alpha

:= betekent zet gelijk aan. Het start model wordt dus gelijk gesteld aan het start model + stap grote * vector. Dit houdt dus in dat elke weight aangepast wordt

Nu je een nieuw start model hebt berekend heb je 1 **generatie** berekend. Hopelijk zit je nu al in de goede richting, maar er moeten nog vele generaties meer komen voordat je het beste model hebt. Je moet stap 2 en 3 dus misschien wel honderden of duizenden keren herhalen. Je stopt pas als het start model zo goed is dat het nauwlijks meer verbeterd. Het eindresultaat is dan het model dat het deep learning systeem heeft berekend.

Het trainen van een computermodel

Zoals in de theorie over convoluties al vermeld is, is er een computermodel nodig voor het herkennen van wat zich in plaatjes bevindt. Dit model is gevuld met alle waardes voor de convoluties en de fully connected layer waarmee de best mogelijke resultaten behaald kunnen worden. Een model als dit is ingewikkeld: zo ingewikkeld dat geen mens dit goed kan voorprogrammeren in de computer. De oplossing voor het maken van dit model is het gebruik van deep learning om het model te trainen. Er zijn twee technieken om hiermee een computermodel te trainen: **supervised learning** en unsupervised learning. Deze laatste manier is in een stuk minder situaties toepasbaar en is ook niet van toepassing op een CNN, dus deze zullen we buiten beschouwing laten.

Supervised learning

Supervised learning houdt in dat er een grote, gelabelde dataset wordt gevoerd aan de computer die hierin door middel van deep learning zelf de verbanden gaat zoeken en hiermee het model creëert om nieuwe afbeeldingen zelf te kunnen classificeren.

Dit is de meest gebruikte leertechniek omdat dit ook de meeste toepassingen heeft. Zo willen we dat een zelfrijdende auto voetgangers als voetganger herkent en zodat de auto weet hoe hij moet handelen en willen we dat toepassingen als Google Foto's onze vakantiekiekjes automatisch kunnen labelen met wat er in het plaatje te zien is. Bij supervised learning leert de computer dus wel zelf, maar wijst de programmeur aan *wat* de computer moet leren.

Wanneer we een computer dus willen leren hoe het alfabet eruit ziet, koppelen we onze deep learning-code aan onze CNN-code en geven we hem een grote trainingsdataset met afbeeldingen van letters en een label met welke letter het is. Een voorbeeld van wat we de computer voeren is dit:

dit is een w

Een computer laten leren een getal of letter te herkennen gaat dus door middel van deep learning zoals dat in het vorige deelonderwerp uitgelegd is.

Trainingsdataset

Voor het trainen van een convolutioneel neuraal netwerk is dus een hoop input-data (letters en labels) nodig. Het maken van zo'n dataset is een hoop werk en daarom stellen veel onderzoekers dit soort datasets gratis online beschikbaar. Voor het trainen van ons model gebruiken wij de Chars74K-dataset, oorspronkelijk gemaakt door Microsoft Research India.

Toepassingen en libraries

Deep learning en computer vision is al gedaan door andere bedrijven. Zij hebben hier vaak enorme grote datasets en goede computers tot hun beschikking. Ook hebben zij erg geavanceerde code. Hierdoor halen zijn vaak erg hoge succespercentages. Als men een deep learning applicatie maakt wordt vaak gebruik gemaakt van libraries. Hoewel wij daar geen gebruik van maken in onze deep learning programma is het nuttig om te weten dat ze wel bestaan.

Tensorflow

De populairste deep learning library is ongetwijfeld tensorflow. Het is gemaakt door Google en wordt gebruik in veel Google producten zoals gmail maar ook in de spraakherkenningssoftware van Google. Het is een open source en wordt tegenwoordig veel gebruikt.

Toepassingen

Machine learning en Computer vision wordt in veel huidige en toekomstige applicaties gebruikt. Het herkennen van tekst kan bijvoorbeeld heel nuttig zijn. Maar ook in meer onverwachte hoeken duikt machine learning op, zoals in het advertentiesysteem van grote bedrijven zoals Google en Facebook, of in de Self Driving Car van Google. Doordat huidige computers steeds sneller worden wordt het makkelijker om vele generaties te trainen. Hierdoor kunnen steeds grotere en complexere netwerken sneller en nauwkeuriger getrained worden.

Hardware

’n dataset is een hoop werk en daarom stellen veel onderzoekers dit soort datasets gr
De meest ideale hardware voor het kunstmatige intelligentie is een server met een hoop grafisch geheugen. Aangezien dit veel meer kost dan een supermarkt baantje oplevert gebruiken wij een **Raspberry Pi model 3B** draaiend op **Raspbian Jessie**.

Qua prestaties is dit absoluut geen ideale computer maar de lage kosten en het lage stroomverbruik maken het voor ons toch de beste optie.

Software

Aangezien ons onderzoek over de werking van een convolutioneel neuraal netwerk gaat, hebben we het aantal software-libraries van derden zoveel mogelijk beperkt.
Ons project is geschreven in **Python 2.7.9** met de volgende python libraries:

- **NumPy 1.8.2**
- **Pillow 2.6.1**

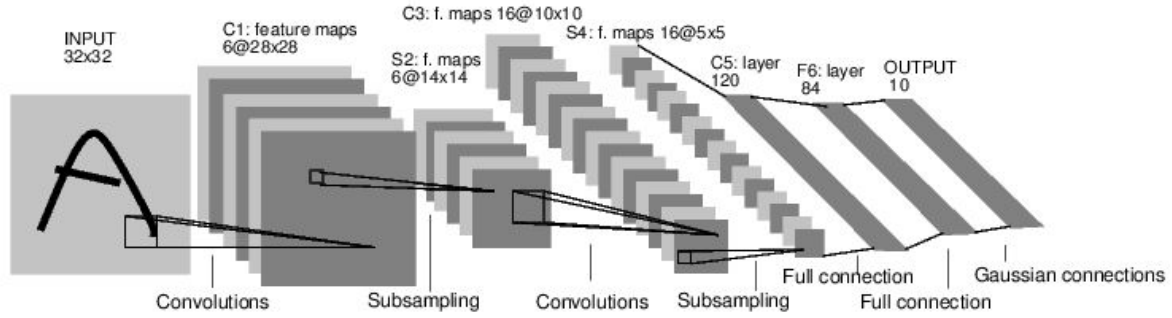
Onze software-implementatie van een convolutioneel neuraal netwerk

Wij hebben voor ons onderzoek een eigen convolutioneel neuraal netwerk en deep learning systeem geschreven in python. Om het overzicht te bewaren zullen we de architectuur van ons CNN, onze preprocessing code, de code van ons CNN en de code voor ons deep learning systeem apart behandelen. We zullen de code zo goed mogelijk uitleggen, maar enige kennis van python wordt dus wel verwacht.

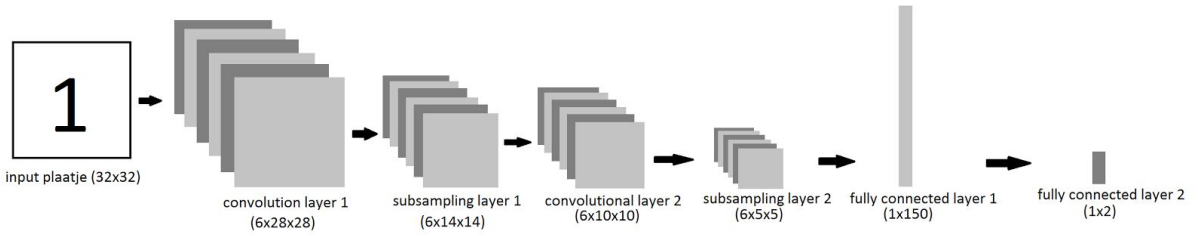
Het deep learning-systeem en en CNN zullen we in stukken uitleggen gezien de lengte van deze scripts.

Architectuur

De architectuur van ons convolutioneel neuraal netwerk is gebaseerd op het bekende LeNet-5, een architectuur ontworpen in 1998 door Yann Lecun. Dit netwerk is eind jaren 90 bekend geworden door de goede resultaten in het herkennen van handgeschreven letters en cijfers en sluit dus perfect aan op ons doel.
LeNet-5 ziet er als volgt uit:



Ons netwerk is een wat versimpelde vorm van LeNet-5 aangezien we ons alleen richten op het herkennen van een 0 en een 1 en ziet er zo uit:



Ons netwerk is dus vooral wat vereenvoudigd vanaf convolution layer 3. Dit hebben wij gedaan omdat ons doel, het onderscheiden van 0en en 1en, minder complex is dan het herkennen van het hele alfabet. Aangezien zoals al beschreven in het stuk over neurale netwerken extra (grote) hidden layers het netwerk in staat stellen complexere dingen te herkennen en wij dit dus niet nodig hebben, hebben wij het aantal feature maps in convolutional layer beperkt van 16 naar 6.

Verder is een heel belangrijk verschil tussen LeNet-5 en onze architectuur de image kernels. Waar bij eigenlijk elke CNN-architectuur zowel de image kernels als de fully-connected layer

getraind worden door deep learning, hebben wij het met de schaarse op internet verkrijgbare informatie en niet genoeg wiskundige kennis het niet voor elkaar gekregen om de image kernels te trainen. Om de code toch werkend te krijgen hebben wij voor de image kernels handmatig waardes ingevuld die volgens ons een kans zouden hebben om te werken en hebben we alleen de fully connected layer getraind.

De verschillende bestanden

Het project bestaat uit verschillende bestanden:

- Een folder met de dataset, dat zijn honderden plaatjes die gebruikt wordt voor het trainen van het neurale netwerk.
- preprocessing.py, het bestand om plaatjes te veranderen voor het neurale netwerk
- cnn.py, het bestand dat het architectuur van het neurale netwerk bevat
- training.py, het bestand dat het deep learning systeem bevat
- controller.py, het bestand die training.py aanroept om het deep learning systeem aan te zetten
- metadata.doc, een document waarin gegevens over hoe goed het neurale netwerk presteert staat opgeslagen
- model.doc, een document waarin het huidige model staat opgeslagen

De trainingsset

De trainingsset is een grote groep plaatjes met de bijbehorende labels er al bij gezet. We hebben 260 plaatjes van een 0 en 260 plaatjes van een 1 gebruikt in onze deep learning netwerk. Deze plaatjes zijn afkomstig van de The Chars74K image database. De link naar deze database is bij onze bronnenlijst opgenomen.

De preprocessing

```
1  import numpy as np
2  from PIL import Image
3
4  def preprocess(imgPath):
5      img = Image.open(imgPath)
6      img.thumbnail((32, 32), Image.ANTIALIAS)
7      container = Image.new('RGB', (32, 32), (255, 255, 255))
8      container.paste(img, (int((32 - img.size[0]) / 2), int((32 - img.size[1]) / 2)))
9      container.save(imgPath)
```

Om plaatjes voor te bereiden voor het CNN halen we de plaatjes eerst door onze preprocessing-functie. Deze functie zorgt ervoor dat elk plaatje netjes binnen een vierkantje van 32x32 pixels past. De functie loopt als volgt:

- op **regel 5** openen we het plaatje en slaan we deze op in de variabele *img*

- op **regel 6** gebruiken we de Pillow-library (PIL) om het plaatje zonder te vervormen te laten passen in een 32x32 vierkantje (maar dit betekent niet dat het plaatje 32x32 is, als het plaatje hiervoor niet vierkant was is dat het nu ook niet)
- op **regel 7** creëren we een wit 32x32 plaatje genaamd *container*
- op **regel 8** plakken we *img* op *container*
- op **regel 9** slaan we het resultaat op met de originele naam en op de originele plek van het input-plaatje

Het convolutioneel neuraal netwerk

```
1  import numpy as np
2  from PIL import Image
3  import time
4  import math
5
6  def cnn(imagePath, model, label, training=False):
7
8      """als het netwerk getrained word de starttijd vastleggen"""
9      if(training):
10         starttime = time.time()
11
12     """plaatje openen"""
13     img = np.asarray(Image.open(imagePath))
14
15     """3d plaatje omzetten naar een 2d array met alleen de kleurintensiteit (zwart-wit)"""
16     input = []
17     for y in range(0, 32):
18         row = []
19         for x in range(0, 32):
20             row.append(max(img[y][x]))
21         input.append(row)
22     inputlayer = np.asarray(input)
23
24     """eerste featuremaps creeeren. activatiefuncties worden ook meteen toegepast"""
25     featuremap1 = [ [], [], [], [], [], [] ]
26     convmodel1 = model[0]
27     for y in range(2, 30):
28         rows = [ [], [], [], [], [], [] ]
29         for x in range(2, 30):
30             receptivefield = np.asarray([
31                 inputlayer[y-2][x-2], inputlayer[y-2][x-1], inputlayer[y-2][x], inputlayer[y-2][x+1], inputlayer[y-2][x+2],
32                 inputlayer[y-1][x-2], inputlayer[y-1][x-1], inputlayer[y-1][x], inputlayer[y-1][x+1], inputlayer[y-1][x+2],
33                   inputlayer[y][x-2],   inputlayer[y][x-1],   inputlayer[y][x],   inputlayer[y][x+1], inputlayer[y][x+2],
34                 inputlayer[y+1][x-2], inputlayer[y+1][x-1], inputlayer[y+1][x], inputlayer[y+1][x+1], inputlayer[y+1][x+2],
35                 inputlayer[y+2][x-2], inputlayer[y+2][x-1], inputlayer[y+2][x], inputlayer[y+2][x+1], inputlayer[y+2][x+2],
36             ])
37             for i in range(0, 6):
38                 rows[i].append(max(0, np.sum(np.multiply(receptivefield, convmodel1[i]))))
39         for h in range(0, 6):
40             featuremap1[h].append(rows[h])
41     featuremap1 = np.asarray(featuremap1)
```

In dit eerste deel van het CNN openen we het plaatje en creëren we de eerste featuremaps

- op **regel 12** openen we het plaatje en slaan we deze op in de variabele *img*
- op **regel 16 t/m 21** lopen we elke RGB-waarde van het plaatje na en slaan we de hoogste kleurwaarde (rood, groen of blauw) op in de python-array *input*
- op **regel 22** zetten we de python-array *input* om in de numpy-array *inputlayer*
- op **regel 27 t/m 40** berekenen we de convoluties. Eerst gaan we alle mogelijke posities voor de image kernel na (met de twee eerste for-loops) en berekenen we bij elke positie de receptive field. Elke receptive field wordt vervolgens in regel 37 en 38 geconvolueerd met alle 6 convolutiekernels die in ons model zijn opgeslagen en opgeslagen op de goede plek in de *row* - array. Op alle waardes wordt op regel 38 voor het opslaan al de activatiefunctie toegepast.

```
42
43      """maxpooling toepassen op alle 6 featuremaps"""
44      maxpooled1 = [ [], [], [], [], [], [] ]
45      for z in range(0, 6):
46          collumn = []
47          for y in range(0, 28, 2):
48              row = []
49              for x in range(0, 28, 2):
50                  row.append(max([featuremap1[z][y][x], featuremap1[z][y][x+1], featuremap1[z][y+1][x], featuremap1[z][y+1][x+1]]))
51              collumn.append(row)
52          maxpooled1[z] = np.asarray(collumn)
53      maxpooled1 = np.asarray(maxpooled1)
54
55
56      """dropout 1 toepassen"""
57      if(training):
58          maxpooled1.flags.writeable = True
59          for z in range(0, 6):
60              for y in range(0, 14):
61                  for x in range(0, 14):
62                      if np.random.rand() < 0.15 :
63                          maxpooled1[z][y][x] = 0
```

In dit tweede deel van het CNN passen we maxpooling toe op de hiervoor gecreerde 6 featuremaps en passen we, als het netwerk aan het trainen is, dropout toe.

- op **regel 45 t/m 52** lopen we van alle 6 featuremaps alle waardes na in setjes van 4. Vervolgens selecteren we de hoogste van deze 4 en slaan deze op in een nieuwe numpy-array *maxpooled1*
- op **regel 57 t/m 63** gaan we, als de functie-parameter *training* aan staat, de *maxpooled1* numpy-array volledig langs. Bij elke waarde is er een 15% kans dat er dropout wordt toegepast op die waarde en wordt deze dus op 0 gezet.

```
65      """tweede featuremaps creeeren, activatiefuncties worden ook meteen toegepast"""
66      featuremap2 = [ [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [] ]
67      convmodel2 = model[1]
68      for y in range(2, 12):
69          rows = [ [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [] ]
70          for x in range(2, 12):
71              receptivefield = np.asarray([
72                  maxpooled1[0][y-2][x-2], maxpooled1[0][y-2][x-1], maxpooled1[0][y-2][x], maxpooled1[0][y-2][x+1], maxpooled1[0][y-2][x+2],
73                  maxpooled1[0][y-1][x-2], maxpooled1[0][y-1][x-1], maxpooled1[0][y-1][x], maxpooled1[0][y-1][x+1], maxpooled1[0][y-1][x+2],
74                    maxpooled1[0][y][x-2],   maxpooled1[0][y][x-1],   maxpooled1[0][y][x],   maxpooled1[0][y][x+1],   maxpooled1[0][y][x+2],
75                  maxpooled1[0][y+1][x-2], maxpooled1[0][y+1][x-1], maxpooled1[0][y+1][x], maxpooled1[0][y+1][x+1], maxpooled1[0][y+1][x+2],
76                  maxpooled1[0][y+2][x-2], maxpooled1[0][y+2][x-1], maxpooled1[0][y+2][x], maxpooled1[0][y+2][x+1], maxpooled1[0][y+2][x+2],
77
78                  maxpooled1[1][y-2][x-2], maxpooled1[1][y-2][x-1], maxpooled1[1][y-2][x], maxpooled1[1][y-2][x+1], maxpooled1[1][y-2][x+2],
79                  maxpooled1[1][y-1][x-2], maxpooled1[1][y-1][x-1], maxpooled1[1][y-1][x], maxpooled1[1][y-1][x+1], maxpooled1[1][y-1][x+2],
80                    maxpooled1[1][y][x-2],   maxpooled1[1][y][x-1],   maxpooled1[1][y][x],   maxpooled1[1][y][x+1],   maxpooled1[1][y][x+2],
81                  maxpooled1[1][y+1][x-2], maxpooled1[1][y+1][x-1], maxpooled1[1][y+1][x], maxpooled1[1][y+1][x+1], maxpooled1[1][y+1][x+2],
82                  maxpooled1[1][y+2][x-2], maxpooled1[1][y+2][x-1], maxpooled1[1][y+2][x], maxpooled1[1][y+2][x+1], maxpooled1[1][y+2][x+2],
83
84                  maxpooled1[2][y-2][x-2], maxpooled1[2][y-2][x-1], maxpooled1[2][y-2][x], maxpooled1[2][y-2][x+1], maxpooled1[2][y-2][x+2],
85                  maxpooled1[2][y-1][x-2], maxpooled1[2][y-1][x-1], maxpooled1[2][y-1][x], maxpooled1[2][y-1][x+1], maxpooled1[2][y-1][x+2],
86                    maxpooled1[2][y][x-2],   maxpooled1[2][y][x-1],   maxpooled1[2][y][x],   maxpooled1[2][y][x+1],   maxpooled1[2][y][x+2],
87                  maxpooled1[2][y+1][x-2], maxpooled1[2][y+1][x-1], maxpooled1[2][y+1][x], maxpooled1[2][y+1][x+1], maxpooled1[2][y+1][x+2],
88                  maxpooled1[2][y+2][x-2], maxpooled1[2][y+2][x-1], maxpooled1[2][y+2][x], maxpooled1[2][y+2][x+1], maxpooled1[2][y+2][x+2],
89
90                  maxpooled1[3][y-2][x-2], maxpooled1[3][y-2][x-1], maxpooled1[3][y-2][x], maxpooled1[3][y-2][x+1], maxpooled1[3][y-2][x+2],
91                  maxpooled1[3][y-1][x-2], maxpooled1[3][y-1][x-1], maxpooled1[3][y-1][x], maxpooled1[3][y-1][x+1], maxpooled1[3][y-1][x+2],
92                    maxpooled1[3][y][x-2],   maxpooled1[3][y][x-1],   maxpooled1[3][y][x],   maxpooled1[3][y][x+1],   maxpooled1[3][y][x+2],
93                  maxpooled1[3][y+1][x-2], maxpooled1[3][y+1][x-1], maxpooled1[3][y+1][x], maxpooled1[3][y+1][x+1], maxpooled1[3][y+1][x+2],
94                  maxpooled1[3][y+2][x-2], maxpooled1[3][y+2][x-1], maxpooled1[3][y+2][x], maxpooled1[3][y+2][x+1], maxpooled1[3][y+2][x+2],
95
96                  maxpooled1[4][y-2][x-2], maxpooled1[4][y-2][x-1], maxpooled1[4][y-2][x], maxpooled1[4][y-2][x+1], maxpooled1[4][y-2][x+2],
97                  maxpooled1[4][y-1][x-2], maxpooled1[4][y-1][x-1], maxpooled1[4][y-1][x], maxpooled1[4][y-1][x+1], maxpooled1[4][y-1][x+2],
98                    maxpooled1[4][y][x-2],   maxpooled1[4][y][x-1],   maxpooled1[4][y][x],   maxpooled1[4][y][x+1],   maxpooled1[4][y][x+2],
99                  maxpooled1[4][y+1][x-2], maxpooled1[4][y+1][x-1], maxpooled1[4][y+1][x], maxpooled1[4][y+1][x+1], maxpooled1[4][y+1][x+2],
100                 maxpooled1[4][y+2][x-2], maxpooled1[4][y+2][x-1], maxpooled1[4][y+2][x], maxpooled1[4][y+2][x+1], maxpooled1[4][y+2][x+2],
101
102                 maxpooled1[5][y-2][x-2], maxpooled1[5][y-2][x-1], maxpooled1[5][y-2][x], maxpooled1[5][y-2][x+1], maxpooled1[5][y-2][x+2],
103                 maxpooled1[5][y-1][x-2], maxpooled1[5][y-1][x-1], maxpooled1[5][y-1][x], maxpooled1[5][y-1][x+1], maxpooled1[5][y-1][x+2],
104                   maxpooled1[5][y][x-2],   maxpooled1[5][y][x-1],   maxpooled1[5][y][x],   maxpooled1[5][y][x+1],   maxpooled1[5][y][x+2],
105                 maxpooled1[5][y+1][x-2], maxpooled1[5][y+1][x-1], maxpooled1[5][y+1][x], maxpooled1[5][y+1][x+1], maxpooled1[5][y+1][x+2],
106                 maxpooled1[5][y+2][x-2], maxpooled1[5][y+2][x-1], maxpooled1[5][y+2][x], maxpooled1[5][y+2][x+1], maxpooled1[5][y+2][x+2],
107             ])
108             for i in range(0, 6):
109                 rows[i].append(max(0, np.sum(np.multiply(receptivefield, convmodel2[i]))))
110         for h in range(0, 6):
111             featuremap2[h].append(rows[h])
112     featuremap2 = np.asarray(featuremap2)
```

In dit derde deel van het CNN creëren we de tweede convolutielaag.

- op **regel 68 t/m 112** lopen we alle mogelijke posities van de imagekernel na en halen we de receptivefields van alle 6 lagen uit de eerste convolutielaag op en slaan deze gezamelijk op in de variabele *receptivefield*. Op regel 108 en 109 worden alle receptivefields weer geconvolueerd met alle 6 imagekernels uit het model en worden de activatiefuncties toegepast waarna het resultaat wordt opgeslagen.

```
114     """maxpooling toepassen op alle 16 featuremaps"""
115     maxpooled2 = [ [], [], [], [], [], [] ]
116     for z in range(0, 6):
117         collumn = []
118         for y in range(0, 10, 2):
119             row = []
120             for x in range(0, 10, 2):
121                 row.append(max([featuremap2[z][y][x], featuremap2[z][y][x+1], featuremap2[z][y+1][x], featuremap2[z][y+1][x+1]]))
122             collumn.append(row)
123         maxpooled2[z] = np.asarray(collumn)
124     maxpooled2 = np.asarray(maxpooled2)
125 
126     """dropout 2 toepassen"""
127     if(training):
128         maxpooled2.flags.writeable = True
129         for z in range(0, 6):
130             for y in range(0, 5):
131                 for x in range(0, 5):
132                     if np.random.rand() < 0.2 :
133                         maxpooled2[z][y][x] = 0
```

In dit vierde deel van het CNN passen we maxpooling en, als het netwerk aan het trainen is, ook dropout.

- op **regel 116 t/m 123** lopen we langs alle 6 featuremaps die zojuist zijn gecreerd na in setjes van 4 en slaan we de hoogste waarde van deze 4 op in een nieuwe array genaamd *maxpooling2*
- op **regel 124** veranderen we de python-array *maxpooled2* in de numpy-array *maxpooling2*
- op **regel 127 t/m 133** lopen we, als de functie-parameter *training* true is, alle waardes van *maxpooling2* langs. Elke waarde heeft een 20% kans om veranderd te worden in een 0 en zo als het ware uitgezet te worden.

```
134 
135     """3 dimensies terugbrengen naar 1 dimensie"""
136     flattened = maxpooled2.reshape(150)
137     flattened = np.append(flattened, 1) #bias
138 
139     fullyconnectedweights = model[2][0]
140 
141     fullyconnected1 = np.zeros(len(fullyconnectedweights) / len(flattened))
142     """fully connected layer 1 berekenen"""
143 
144     for i in range(len(fullyconnectedweights)):
145         fullyconnected1[math.floor((i / len(flattened)))] += flattened[i%len(flattened)] * fullyconnectedweights[i]
146 
147     for i in range(len(fullyconnected1)):
148         fullyconnected1[i] = abs(fullyconnected1[i])
```

- op **regel 136** wordt de array *maxpooling2* naar een array van 1 dimensie gebracht en in de array genaamd *flattened* gestopt
- op **regel 137** wordt aan de array een bias van 1 toegevoegd
- op **regel 139** wordt de nuttige weights van het model gehaald
- op **regel 141** wordt de array *fullyconnected1* gemaakt, deze array zal de waardes van de output nodes gaan bevatten
- op **regel 144 en 145** wordt de output nodes berekend en in *fullyconnected1* gestopt

- op **regel 147 en 148** wordt elke negatieve waarde van de output nodes positief gemaakt, om te zorgen dat de softmax functie goed werkt

```
150     #normalize naar 100
151     total = 0
152     for i in range(len(fullyconnected1)):
153         total += abs(fullyconnected1[i])
154
155     if total > 100:
156         for i in range(len(fullyconnected1)):
157             fullyconnected1[i] = fullyconnected1[i] / total * 10
158
159     def softmax(x):
160         return np.exp(x) / np.sum(np.exp(x))
161
162     def crossEntropy(s,l):
163         return -1 * np.sum([l] * np.log10([s]))
164
165     softmaxScore = softmax(fullyconnected1)
166
167     crossentropyscore = crossEntropy(softmaxScore, label)
168
169     if training:
170         returnarray = []
171         returnarray.append(crossentropyscore)
172         returnarray.append(label)
173         returnarray.append(softmaxScore)
174         returnarray.append(flattened)
175         return returnarray
176     else:
177         return softmaxScore
```

- op **regel 151 t/m 157** wordt de totale som van de output nodes zo nodig verlaagd naar 100, zodat de softmax functie de waardes aan kan.
- op **regel 159** wordt de softmax functie gedefinieerd
- op **regel 160** wordt de cross entropy functie gedefinieerd
- op **regel 165** wordt de softmax score berekend
- op **regel 167** wordt de cross entropy score berekend
- op **regel 169 t/m 177** wordt de output teruggestuurd, als het gebruikt wordt voor training worden er meer waardes teruggestuurd dan als er niet getraind wordt.

De deep learning code

Dit kopje beschrijft training.py.
Dit document bestaat uit 1 functie: train()
Deze functie neemt als parameter de alpha (stapgrote), het aantal generaties dat het model getraind moet worden en in welk document het model en de metadata moet worden opgeslagen.

```python
import numpy as np
import cnn as cnn
import math as math
import random as random
import time as time
import os.path
import random

#training functie
def train(alpha, totalGeneration, modelfile, metadatafile):

    #zorgen dat data goed wordt weggeschreven
    with open(metadatafile, "w") as myfile:
        myfile.write('\n\n\n\n')

    #tijd en generatie bijhouden
    startTime = int(time.time())

    vervangTekst(metadatafile, 0, "Training gestart op " + str(int(time.time())) + "..\n")

```

- Op **regel 13** wordt de metadata file leeggehaald wordt voordat het model getraind gaat worden. Wat precies in de metadata file komt te staan wordt later verder uitgelegd.
- Op **regel 17 t/m 19** wordt het eerste gegeven in de metadata file opgeslagen, namelijk wanneer de trainig gestart is.

```python
    #alle labels
    lettersArray = ['0','1']

    #vectors voor classificatie systeem genereren
    AllLabels = np.zeros((len(lettersArray), len(lettersArray)))
    for x in xrange(len(lettersArray)):
        for y in xrange(len(lettersArray)):
            if x == y:
                AllLabels[x][y] = 1

    #het maken van de startweights
    hiddenSize = 150
    #hiddenSize zijn het aantal nodes aan het einde van de 2e hidden layer
    model = np.asarray([

        np.asarray([[-1, 0, 2, 0, -1, 0, -1, 3, -1, 0, 2, 3, 4, 3, 2, 0, -1, 3, -1, 0, -1, 0, 2, 0, -1],
         [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 3, -5, 4, 1, 0, -1, 3, -1, 0, 0, -2, 1, -2, 0],
         [0, -1, 3, -1, 0, 0, -1, 3, -1, 0, 0, -1, 3, -1, 0, 0, -1, 3, -1, 0, 0, -1, 3, -1, 0],
         [0, 0, 0, 0, 0, -1, -1, -1, -1, -1, 3, 3, 3, 3, 3, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0],
         [0, -2, 1, -2, 0, 0, -1, 3, -1, 0, 1, 3, -5, 3, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
         [2, 1, -1, 1, 2, 1, 3, 1, 3, 1, -1, 1, 4, 1, -1, 1, 3, 1, 3, 1, 2, 1, -1, 1, 2]]),

        np.asarray([[1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 4, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1,
         [4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0, 2, 4, 4, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0, 2
         [0, 0, 0, 0, 0, -1, -1, 1, -1, -1, 1, 1, 2, 1, 1, -1, -1, 1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
         [0, 0, 1, 0, 0, 0, 1, 2, 1, 0, 1, 2, 4, 2, 1, 0, 1, 2, 1, 0, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 0, -1, -1, -1, 0, -1, -1
         [1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 4, 2, 0, 2, 4, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2, 3, 2, 0, 1, 3, 4, 3
         [0, 0, 2, 0, 0, 0, 1, 2, 1, 0, 2, 2, 3, 2, 2, 0, 1, 2, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, 0, 0, -1,
        ]),

        [np.asarray(np.random.rand((hiddenSize + 1) * len(lettersArray)))]

    ])
```

- Op **regel 22** wordt een array gemaakt met alle labels waarop je wilt trainen, in dit geval de 0 en 1.
- Op **regel 24 t/m 29** wordt een 2 dimensionaal array gemaakt waar de vectors voor elke label wordt gemaakt, zoals uitgelegd onder het kopje ‘Wat houdt een classificatie netwerk in’. In dit geval is het relatief simpel:
  AllLabels = [[1,0],[0,1]] is het resultaat
- Op **regel 32** staat de hiddensize gegeven, dat zijn de hoeveelheidnodes aan het einde van de 2e hidden layer

- Op **regel 34 t/m 53** wordt het startmodel gemaakt.
- Op **regel 36 t/m 41** staan de weights voor de 6 kernels in de eerste laag.
- Op **regel 43 t/m 49** staan de weights voor de 6 kernels in de tweede laag (de regels loopt aan de rechterkant door). Deze weights zijn zoals eerder besproken van te voren al gegeven.
    - Alle getallen die hiervoor gebruikt zijn staan in de bijlage ‘model’ gegeven.
- Op **regel 51** staan de weights voor de fully connected layer. In dit geval zijn dat 302

```
55      #importen van data
56      labelSet = []
57      dataSet = []
58
59      for x in xrange(len(lettersArray)):
60          y = 0
61          while os.path.isfile("dataset/" + lettersArray[x] + "/img" + str(x + 1).zfill(3)  + "-" + str(y + 1).zfill(3) + ".png"):
62              dataSet.append(str("dataset/" + lettersArray[x] + "/img" + str(x + 1).zfill(3)  + "-" + str(y + 1).zfill(3) + ".png"))
63              labelSet.append(int(x))
64              y += 1
65          y = 0
66          while os.path.isfile("dataset/" + lettersArray[x] + "/img" + str(x + 1).zfill(3)  + "-" + str(y + 1).zfill(5) + ".png"):
67              dataSet.append(str("dataset/" + lettersArray[x] + "/img" + str(x + 1).zfill(3)  + "-" + str(y + 1).zfill(5) + ".png"))
68              labelSet.append(int(x))
69              y += 1
70      dataSet = np.asarray(dataSet)
71
72      #label koppelen aan data
73      allData = []
74      i = 0
75      for data in dataSet:
76          allData.append([data, labelSet[i]])
77          i += 1
78
```

getallen die willekeurig tussen de 0 en de 1 gegenereerd zijn.

-
- Op **regel 59 t/m 70** staat de functie om het pad naar elk plaatje in een array te zetten. Zie het kopje ‘De trainingsset’ hoe het opgeslagen staat.
- Op **regel 72 t/m 77** wordt elk plaatje met de bijbehorende label gekoppeld.
  allData = [[‘dataset/0/img001-001.png’,0],[‘dataset/0/img001-002.png’,0]... etc

```
79      #start van nieuwe generatie
80      for numLoop in xrange(totalGeneration):
81          #metadatafile aanpassen
82          vervangTekst(metadatafile, 1, "Bezig met generatie " + str(numLoop) + "..\n")
83
84          #willekeurige data kiezen
85          gebruikteData = allData[random.randint(0, len(allData) - 1)]
86          print gebruikteData[1]
87
88          #neurale netwerk uitvoeren
89          outputData = cnn.cnn(gebruikteData[0], model, AllLabels[gebruikteData[1]], True)
90
```

- Op **regel 80** begint de grote loop van de code. Elke loop staat voor 1 generatie.
- Op **regel 82** wordt de eerste regel van de metadata document aangepast zodat er komt te staan welke generatie wordt berekend.
- Op **regel 85 en 86** wordt een willekeurig plaatje van de trainingsset gekozen
- Op **regel 89** wordt cnn.cnn() aangeroepen, hier roept de code dus de functie die hiervoor is uitgelegd aan. Het neurale netwerk wordt hier dus uitgevoerd, met als inputs het plaatje dat wordt gebruikt, het huidige model, de label van het uitgekozen plaatje en als laatste input wordt gezegd dat dropout toegepast moet worden.
  De output van cnn.cnn() wordt opgeslagen in outputData.

```
91          #backpropagation
92          for i in range(len(model[2][0])):
93              x = i%(hiddenSize + 1)
94              y = math.floor(i/(hiddenSize + 1))
95              derivitive = (outputData[2][y] - outputData[1][y]) * outputData[3][x]
96              model[2][0][i] -= alpha * derivitive
```

- Op **regel 92** begint de backpropagation voor elke weight van de fully connected layer (oftewel model[2][0])
- Op **regel 93** wordt berekend vanaf welke node de weight komt. Elke weight verbind namelijk een input node en output node. Deze x geeft dus aan welke input node er gebruikt moet worden.
- Op **regel 94** wordt berekend welke output node gebruikt moet worden.
- Op **regel 95** wordt de afgeleide berekend. Dit gaat met de formule (softmaxScore - labelScore) * inputScore
- Op **regel 96** wordt het model aangepast. Hier wordt ook nog met de alpha vermenigvuldigd.

```
98          #gegevens in de documenten aanpassen
99          with open(modelfile, "w") as myfile:
100             myfile.write(str(model[0]) + ',\n')
101             myfile.write(str(model[1]) + ',\n')
102             myfile.write(str(np.asarray(model[2])) + ',\n')
103
104         with open(metadatafile, "a") as myfile:
105             myfile.write(str(numLoop) + ' ' + str(outputData[1][1]) + ' ' + str(outputData[0]) + ' ' + str(outputData[2][0]) + '\n')
106
107 def vervangTekst(file, line, text):
108     lines = open(file, 'r').readlines()
109     lines[line] = text
110     out = open(file, 'w')
111     out.writelines(lines)
112     out.close()
```

- Op **regel 99 t/m 102** wordt het model aangepast in het model document
- Op **regel 104 en 105** wordt aan het metadata document extra data toegevoegd: de generatie, de gebruikte label, de cross entropy score, hoeveel kans er is dat het plaatje een 0 was volgens het neurale netwerk.
- Op **regel 107 t/m 112** wordt de functie vervangTekst gemaakt die tekst vervangt in bepaalde documenten. Deze functie is 2 keer aangeroepen in training.py

Het control document

Het laatste document is controller.py. Dit is een klein document die training.train() aanroept.

```
1  import cnn as cnn
2  import training as training
3  import time
4  import numpy
5
6  numpy.set_printoptions(threshold=numpy.nan)
7  training.train(0.00000001, 30000, "model.doc", "metadata.doc")
```

- Op **regel 6** wordt de printopties veranderd waardoor het volledige model in model.doc wordt geprint in plaats van enkel de eerste paar weights.

- Op **regel 7** wordt training.train() aangeroepen, met de alpha, hoeveel generaties, het model document en metadata document.

Dit document wordt aangezet als je het model wilt trainen.

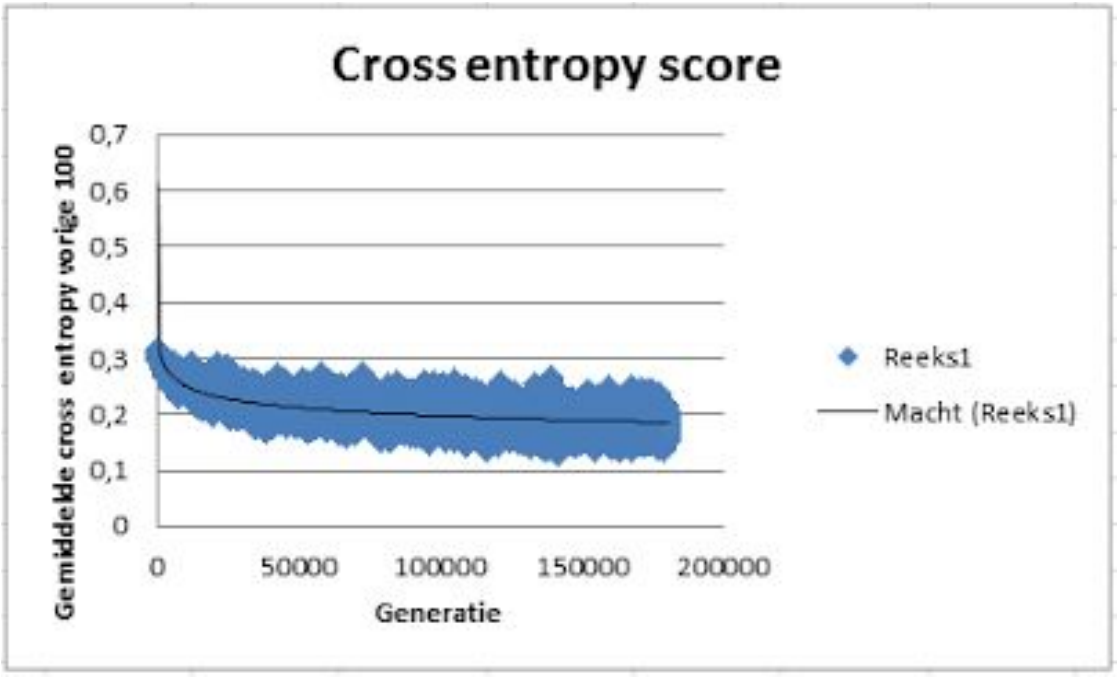
Resultaten

Gebruike waardes gedurende de training

We hebben gekozen om een convolutional neuraal netwerk te trainen die het verschil tussen het getal 0 en het getal 1 kan zien. Normaal gesproken moet een deep learning systeem het hele model trainen, maar omdat dit behoorlijk complex is hebben we besloten de eerste twee lagen met convoluties met de hand te schrijven en enkel de fully connected layer te schrijven. De gekozen weights kan gevonden worden in de bijlage met het model.
Voor de trainingsset hebben we 260 verschillende plaatjes van een 0 en 260 verschillende plaatjes van een 1 gebruikt.
Voor de alpha hebben we 0.000000005 gebruikt, deze waarde moet zo laag zijn omdat de weights van de fully connected anders te veel schommelde.
We hebben in totaal ruim 180.834 generaties berekend

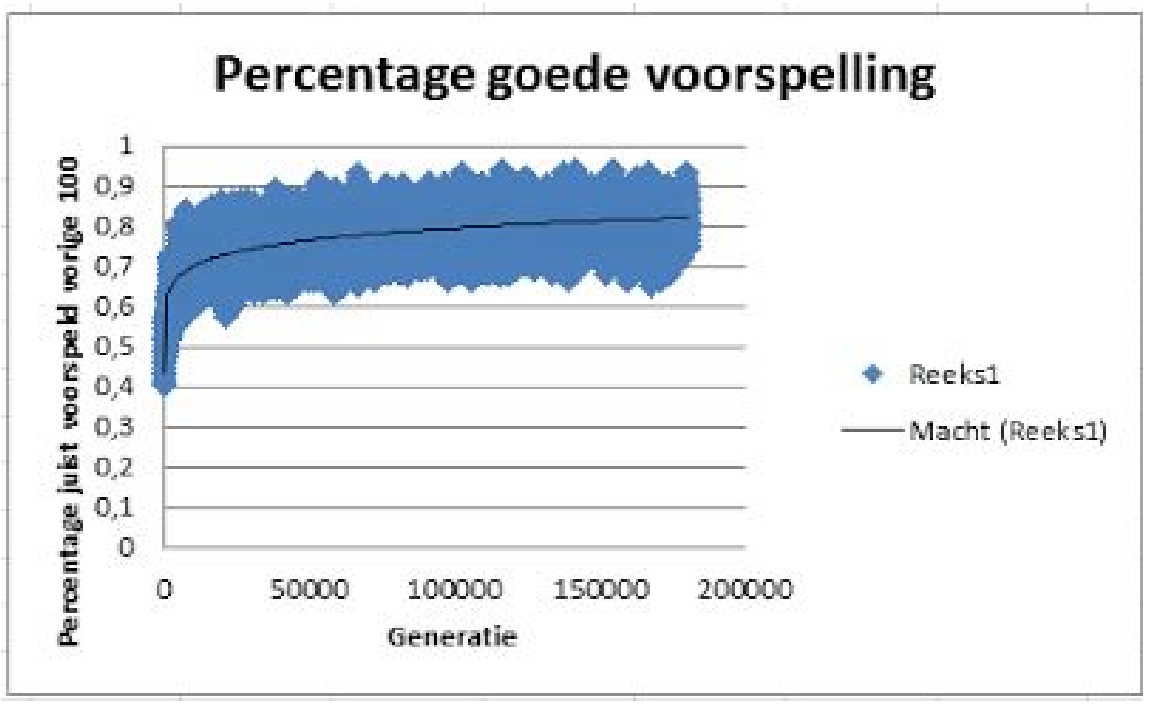
Resultaat gedurende de training

Gedurende de training was er een duidelijke verbetering aan het begin maar werd dat al snel stabieler. We hebben na elke generatie de gemiddelde cross entropy score berekend van de 100 voorgaande generaties. De volgende grafiek is het resultaat



Elk blauwe punt is een waarde die gevonden is. De zwarte lijn is de trendlijn. De gemiddelde cross entropy van de laatste 100 generaties is 0,1728. 10^(-0,1728) = 0,6717, dus het model heeft gemiddeld genomen een softmax score voor de goede label van 0,6717.

Het tweede resultaat waar we naar hebben gekeken is hoeveel procent van de keer heeft het model de juiste label gekozen, dus hoeveel procent van de keer is de softmax score van de juiste label hoger dan 0,5. Hiervoor hebben we na elke generatie de percentage juiste resultaten van de vorige 100 generaties berekend. De volgende grafiek is het resultaat:



Elk blauwe punt is een waarde die gevonden is. De zwarte lijn is de trendlijn. De percentage goede voorspellingen van de laatste 100 generaties is 85,15%.

Het uiteindelijke model kan de trainingsset dus met ongeveer 85% zekerheid goed classificeren.

Live demo op internet

Om het makkelijker te maken het resultaat van onze code te zien, hebben wij een live demo gemaakt op https://www.pws.brian.party. Op deze pagina kan de gebruiker een 0 of een 1 tekenen en die vervolgens naar de server sturen. Op deze server draait onze code met ons model en deze zal het resultaat terugsturen naar de browser.

Conclusie

Onze computerprogramma kan de cijfer 0 en 1 herkennen met behulp van een convolutionaal neuraal netwerk, dit netwerk wordt met behulp van deep learning getrained. Het model kan met ongeveer 85% zekerheid het verschil tussen een 0 en een 1 herkennen. Onze hypothese ging ervan uit dat we een computerprogramma zouden kunnen schrijven die elke letter of getal zou kunnen herkennen, maar dit is ons niet gelukt. Door problemen waar we tijdens ons onderzoek tegenop liepen (meer hierover in de discussie) presteerde onze code niet zoals we hadden gehoopt. Desalniettemin bewijzen de grafieken en het uiteindelijke succespercentage van 85,15% dat, hoewel niet optimaal, ons convolutioneel neuraal netwerk wel degelijk werkt

Discussie

Hoewel de door onze software behaalde resultaten op de (vrij simpele) taak van het onderscheiden van nullen en enen een flink hoger succespercentage hebben dan bij willekeurig gokken, is wel duidelijk dat onze code niet te vergelijken valt met de software van grote bedrijven als Google en universiteiten als de University of Toronto. Dit komt voor een deel doordat we niet de benodigde wiskundige vaardigheden bevatten maar een groot probleem wat we gemerkt hebben tijdens ons onderzoek is de vrij beperkte informatie op het internet over dit onderwerp. Er is meer dan genoeg informatie over het maken van een programma voor computervisie met behulp van een van de grote software-libraries, maar de informatie over hoe computervisie in details werkt is vaak gebrekkig. Dit gebrek aan goede informatie heeft ervoor gezorgd dat we sommige gaten in onze kennis zelf moesten opvullen door een oplossing te verzinnen. Zo hebben we niet kunnen vinden hoe we de imagekernels moeten trainen en is het ons ook nog steeds niet duidelijk of de architectuur van ons netwerk optimaal is. Voor deze problemen hebben we desalniettemin een (geïmproviseerde) oplossing gevonden waardoor de code wel werkt, maar dit soort oplossingen zorgen uiteraard voor slechtere prestaties.

Bronnenlijst

ujjwalkarn (2016). *An Intuive Explanation of Convolutional Neural Networks.* Geraadpleegd op 7 oktober 2016, van https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

Convolutional Neural Networks (CNNs / ConvNets). Geraadpleegd op 2 september 2016, van http://cs231n.github.io/convolutional-networks/.

Convolutional Neural Networks (LeNet). Geraadpleegd op 2 september 2016, van http://deeplearning.net/tutorial/lenet.html.

Vanhoucke, Vincent. *Deep Learning*. Geraadpleegd op 17 oktober 2016, van https://www.udacity.com/course/deep-learning--ud730.

Dan Fleisch (2011). *What's a Tensor?.* Geraadpleegd op 9 september 2016, van https://www.youtube.com/watch?v=f5liqUk0ZTw.

University of Surrey. *The Chars74K dataset.* Geraadpleegd op 3 november 2016, van http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/

Matt Mazur. *A Step by Step Backpropagation Example.* Geraadpleegd op 28 december 2016, van https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

Long, Jonathan; Shelhamer, Evan; Darrell, Trevor (2015). *Fully Convolutional Networks for Semantic Segmentation* (academic paper). UC Berkeley, Berkeley. Van https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf.

LeCunn, Yann; Botou, Léon; Bengio, Yoshua; Haffner, Patrick (1998). *Gradient-Based Learning Applied to Document Recognition* (academic paper). New York University, New York. Van http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

Sundaresan, Vishnu; Lin, Jasper (2015). *Recognizing Handwritten Digits and Characters*. Stanford University, Stanford.

Srivasta, Nitish; Hinton, Geoffrey; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan (2014). *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* University of Toronto, Toronto.

Logboek

7-9	René en Brian	Oriëntatie, bepalen wat we willen doen en opzoeken wat deep learning inhoudt	2 uur
14-9	René en Brian	Beginnen met python leren, dieper ingaan op de wiskunde achter deep learning	2 uur

16-9	René	Verder gegaan met udacity courses	1 uur
17-9	Brian	Code gemaakt voor importen en voorbewerken van plaatjes	2 uur
17-9	René	Softmax en cross entropy functies geschreven, numpy library leren en begrijpen	2 uur
21-9	René en Brian	Verder gegaan met udacity courses Verder gegaan met structuur van plaatjes bestuderen en net-5 structuur begrijpen	2 uur
23-9	René	Begin gemaakt met code van een simpel neuraal netwerk dat opgelost kan worden met deep learning (matrix multiplication)	2 uur
23-9	Brian	Code geschreven voor convoluties	2 uur
24-9	René	basisstructuur voor deep learning maken, zodat avarage loss per model kan worden berekend	2 uur
28-9	Brian	Verder gegaan met programmeren, plaatjes kunnen nu geimport worden, geresized en er kunnen convoluties uitgevoerd worden	2 uur
28-9	René	Hidden layers begrijpen	2 uur
2-10	Brian	Verschillende convoluties uitvoeren, begrijpen waar convoluties precies voor zijn. Verder leren over lenet-5	2 uur
2-10	René	softmax en cross entropy samen met labels en test data in 1 model structuur zetten. Leren hoe het leren moet	2 uur
5-10	René en Brian	Vooruitgang met elkaar vergeleken, kijken hoe we onze beiden delen samen kunnen voegen	2 uur
9-10	Brian	Tensors bestudeerd	2 uur
10-10	René	Geleerd over gradient descent	1 uur
12-10	René	Genereren van weights veranderd, gradient descent geimplementeerd	2 uur
12-10	Brian	Alle code georderd, meerdere files gemaakt die als libraries dienen.	3 uur
16-10	René	Hidden layers toegevoegd, begonnen met het mogelijk maken om grotere datasets te gebruiken	3 uur

21-10	René en Brian	Middag bij elkaar gekomen om deep learning die René geschreven heeft uit te oefenen op en cijfer dataset, met behulp van de code van het neurale netwerk dat Brian geschreven heeft	4 uur
28-10	Brian	Verder gegaan met pws zelf schrijven	2 uur
30-10	René en Brian	Verder gegaan met pws zelf schrijven, veel structuur aangelegd en verder met theorie	2 uur
5-11	Brian	Theorie over convoluties geschreven	1 uur
8-11	René	Begin met deep learning deel van theorie	1 uur
12-11	René	Herschrijven deel van de code	1 uur
18-11	Brian	Code voor convoluties verbeterd	1 uur
19-11	René	Verder met theorie	1 uur
23-11	René en Brian	Code geschreven voor nieuwe dataset, starten met laatste delen van de code schrijven voor het trainen	2 uur
27-11	René en Brian	Verder met de code, begonnen met de rasberry pi in te schakelen en voortgangssite maken	1 uur
28-11	René en Brian	Code laten uitvoeren op de rasberry pi, starten met trainen, verder met verslag	2 uur
29-11	Brian	afmaken uitleg convulutionaal netwerk	2 uur
29-11	René	Verder met uitleg deep learning	2 uur
30-11	René en Brian	Theorie helemaal afmaken, verslag afmaken om als voorlopige versie in te leveren	2 uur
02-12	Brian	Verbeteringen gemaakt aan de code voor het convolutioneel neuraal netwerk	1 uur
03-12	René	Geprobeerd deep learning code goed werkend te krijgen	1,5 uur
9-12	René en Brian	Laatste poging om huidige code werkend te krijgen	4 uur
25-12	Brian	Compleet nieuwe code geschreven voor convolutioneel neuraal netwerk	4 uur
28-12	René	Verder onderzoek naar hoe deep learning code geschreven moet worden	2 uur
03-01	Brian	Snelheidsverbeteringen toegevoegd aan code	2,5 uur

		voor convolutioneel neuraal netwerk	
05-01	René	Meer informatie gezocht over deep learning in combinatie met een convolutioneel neuraal netwerk	1,5 uur
06-01	Brian	Snelheidsverbeteringen toegevoegd aan code voor convolutioneel neuraal netwerk	2,5 uur
06-01	René	Meer informatie gezocht over deep learning in combinatie met een convolutioneel neuraal netwerk	2 uur
07-01	Brian	Werkstuk iets aangepast om beter bij de nieuwe convolutioneel neural netwerk-code te passen	1 uur
20-01	René	Meer informatie gezocht over deep learning in combinatie met een convolutioneel neuraal netwerk	2 uur
22-01	René en Brian	Samen nieuwe code geschreven voor deep learning en samengevoegd met code voor convolutioneel neuraal netwerk	4 uur
24-01	René	Meer informatie gezocht over deep learning in combinatie met een convolutioneel neuraal netwerk	1 uur
27-01	René	Meer informatie gezocht over deep learning in combinatie met een convolutioneel neuraal netwerk	1 uur
28-01	René	Meer informatie gezocht over deep learning in combinatie met een convolutioneel neuraal netwerk	2 uur
30-01	René en Brian	Alternatieve oplossing voor het trainen van imagekernels bedacht en code geschreven	3,5 uur
31-01	René en Brian	Code getest en verslag afgemaakt	2 uur
31-01	Brian	Code voor convolutioneel neuraal netwerk op basis van de tests iets aangepast	0,5 uur
01-02	Brian	Demo-website gemaakt om via de browser de code te kunnen testen	3,5 uur
01-02	René en Brian	Profielwerkstuk laatste keer gecontroleerd, fouten verbeterd	2 uur

Totaal

René: 71,5 uur
Brian: 68,5 uur

Bijlage

Het model

Het model

Hieronder staat het model, dit is de code om een numpy array te krijgen in python met het uiteindelijke model erin
model[0] zijn de weights voor de eerste convolutie laag (van te voren bepaald)
model[1] zijn de weights voor de tweede convolutie laag (van te voren bepaald)
model[2] zijn de weights voor de laatste fully connected laag (bepaald door deep learning)
zorg ervoor dat numpy geimport is met de volgende code: import numpy as np

model = np.asarray([

np.asarray([[-1, 0, 2, 0, -1, 0, -1, 3, -1, 0, 2, 3, 4, 3, 2, 0, -1, 3, -1, 0, -1, 0, 2, 0, -1],
[0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 3, -5, 4, 1, 0, -1, 3, -1, 0, 0, -2, 1, -2, 0],
[0, -1, 3, -1, 0, 0, -1, 3, -1, 0, 0, -1, 3, -1, 0, 0, -1, 3, -1, 0, 0, -1, 3, -1, 0],
[0, 0, 0, 0, 0, -1, -1, -1, -1, -1, 3, 3, 3, 3, 3, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0],
[0, -2, 1, -2, 0, 0, -1, 3, -1, 0, 1, 3, -5, 3, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
[2, 1, -1, 1, 2, 1, 3, 1, 3, 1, -1, 1, 4, 1, -1, 1, 3, 1, 3, 1, 2, 1, -1, 1, 2]]),

np.asarray([[1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 4, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2, 1, 1, 2, 4, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 4, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 4, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 4, 2, 1,
1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 4, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1],
[4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0, 2, 4, 4, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0,
2, 4, 4, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0, 2, 4, 4, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 2, 2, 2, 4, 4, 2, 0, 2, 4, 4, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0, 2, 4, 4, 2, 2, 2, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 4, 4, 2, 0, 2, 4, 4, 2, 2, 2, 4, 4, 4, 4, 4],
[0, 0, 0, 0, 0, -1, -1, 1, -1, -1, 1, 1, 2, 1, 1, -1, -1, 1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
1, 4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, 0, 0, -2, -2, -2, 0, 0, -1, -1, -1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 2, 1, 2, 0, 1, 1, 2, 1, 1, 0, 2, 1, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 4, 1, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 2, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 1, 2, 1, 0, 1, 2, 4, 2, 1, 0, 1, 2, 1, 0, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 0, -1, -1, -1, 0, -1,
-1, -2, -1, -1, 0, -1, -1, -1, 0, 0, 0, -1, 0, 0, -1, 0, 0, 0, -1, -1, 0, 0, -1, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1, -1, -1, 0, -1,
-1, -2, -1, -1, 0, -1, -1, -1, 0, 0, 0, -1, 0, 0, 0, 0, 1, 0, 0, 0, 2, 1, 2, 0, 1, 1, 4, 1, 1, 0, 2, 1, 2, 0, 0, 0, 1, 0,
0],
[1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 4, 2, 0, 2, 4, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2, 3, 2, 0, 1, 3, 4,
3, 1, 0, 2, 3, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, 0, 0, -1, -2, -1, 0, 0, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 2, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -2, -3, -2, 0, -1, -3, -4, -3,
-1, 0, -2, -3, -2, 0, 0, 0, -1, 0, 0, 0, -1, 1, 3, 4, -1, -1, 1, 2, 3, 1, 1, 1, 1, 1, 3, 2, 1, -1, -1, 4, 3, 1, -1, 0],
[0, 0, 2, 0, 0, 0, 1, 2, 1, 0, 2, 2, 3, 2, 2, 0, 1, 2, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, 0, 0, -1,
-2, -1, 0, 0, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 3, 1, 0, 0, 1, 4, 1, 0, 0, 1, 3, 1, 0, 0, 0, 2, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 2, 3, 4, 3, 2, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, 0, 0, -1, -2, -1, 0,
0, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 3, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]]),

[np.asarray([1.27299680e+00, 1.45228768e-02, 1.93963504e+00, 1.00190291e+00

, 5.38487840e-01, 5.17085489e-01, -8.83821189e-01, 2.95710787e+00
, 1.28133756e+00, -7.19235330e-01, -1.25435799e+00, -9.37270590e-01
, 2.23466991e+00, 6.59730349e-01, -8.22655630e-01, -1.25549664e+00
, -1.03257380e+00, 2.80663720e+00, 2.52601688e-01, 3.00975150e-01
, -6.14360816e-03, -6.71694962e-01, 1.75511651e+00, 1.52693658e+00
, 1.09915468e+00, 1.08290929e+00, 4.34371884e-01, 7.74429629e-01
, 9.43417195e-01, 6.18881785e-01, -1.55311531e-02, 3.19015809e-01
, 1.32216369e+00, 1.09555071e+00, 4.88535493e-01, -2.22839139e-01
, 1.95442914e-01, 1.18723528e+00, 6.19003599e-01, 2.32232478e-01
, -8.01402820e-02, 3.22259469e-01, 6.33486554e-01, 4.76349937e-01
, 3.50656749e-01, 3.26758328e-01, 7.14452058e-01, 8.92512272e-01
, 9.96903934e-01, 9.25236733e-01, 1.21322740e+00, -3.39923370e-01
, 1.80257458e+00, 1.48404770e+00, 5.72223977e-01, 1.07306802e-01
, -7.73128507e-01, 2.65795455e+00, 7.72790416e-01, -9.80936206e-01
, -1.25663073e+00, -8.36762975e-01, 2.41539217e+00, -3.38806177e-02
, -4.40484836e-01, -1.19467642e+00, -6.89676048e-02, 2.68574279e+00
, 1.59361044e-01, -3.51978144e-01, 7.09905942e-02, 5.56881163e-02
, 1.66525121e+00, -1.85481263e-01, 1.35843519e+00, 1.78206027e+00
, -7.27530574e-01, 1.65298292e+00, 6.64001151e-01, 2.95348135e-01
, -2.69057898e-01, -2.17330501e+00, 3.87428408e+00, 1.61181827e+00
, -1.35460499e+00, -1.29274364e+00, -1.90187578e+00, 3.43858736e+00
, 5.60787848e-01, -1.40562629e+00, -1.85711486e+00, -1.52418622e+00
, 4.11649635e+00, -8.84535513e-01, -2.21573536e-01, -1.74173483e-01
, -1.08681789e+00, 2.45059473e+00, -3.47362428e-01, 1.55636815e+00
, 2.06941255e+00, 4.29266056e-02, 1.11132903e-01, 3.66375753e+00
, 8.35214025e-02, -2.40570104e-03, -6.37945743e-01, 1.15625938e+00
, 2.49973955e+00, -6.56648082e-01, -1.29526025e+00, -1.14337296e+00
, 1.31816023e+00, 1.15751969e+00, -1.62399798e+00, -1.52903264e+00
, 3.10092369e-01, 1.56103377e+00, 2.02603255e-02, -9.65594387e-01
, -9.18236811e-01, 2.19183146e+00, 7.89815495e-01, 2.38070448e-01
, 1.96179577e+00, 1.15713806e+00, -8.39468345e-01, 1.21498895e+00
, 1.50376514e+00, 6.68856849e-01, -4.16224116e-02, -1.10147299e+00
, 3.24268489e+00, 1.83256554e+00, -8.79689092e-01, -7.62865758e-01
, -8.69241045e-01, 3.51342679e+00, 1.03292234e+00, -9.26432085e-01
, -5.65327684e-01, -1.09229181e+00, 3.20376961e+00, -1.56068215e-01
, -1.65617055e-01, 4.17086502e-01, -6.77609349e-01, 1.75225246e+00
, -5.32547176e-01, 1.30314551e+00, 2.64177549e-01, -7.63144086e-01
, 5.00096965e-01, -4.92418959e-01, -5.90088786e-01, -1.06012263e-01
, 1.09017439e+00, 1.89085689e+00, -1.83022505e+00, -4.24065638e-01
, 1.62545939e+00, 1.72574032e+00, 2.32534530e+00, -2.12856346e+00
, 9.54466586e-02, 1.73309572e+00, 1.85153079e+00, 1.71354863e+00
, -1.75795802e+00, 6.54261550e-01, 9.65466414e-01, 3.83415658e-02
, 1.21705106e+00, -1.03721312e+00, 4.02868126e-02, -8.29993013e-01
, 4.84741898e-01, 1.58381124e-01, -1.33586587e-01, -2.28950226e-01
, 9.64399615e-02, 7.94190293e-01, 8.00472472e-01, 1.15048868e-01
, -2.11288045e-01, 9.43118547e-01, 9.20435859e-01, 1.03473796e+00

, 4.81629909e-01, 2.28340527e-02, 1.08672431e+00, 9.80992938e-01
, 1.16079306e+00, 2.31113308e-01, -1.45638351e-01, 7.28399278e-01
, 2.66508434e-01, 2.95232845e-01, 1.45621501e-01, -3.42063771e-01
, 3.68154284e-01, -3.08326111e-01, 8.07627438e-01, -2.04034538e-01
, -3.98101307e-01, 6.16449073e-01, 6.41427916e-01, 1.36909138e+00
, -1.28151456e+00, -3.84942979e-01, 1.82922878e+00, 1.51436484e+00
, 1.75569719e+00, -1.44607211e+00, 9.42831695e-01, 1.25711253e+00
, 1.57347694e+00, 1.74176362e+00, -1.79957228e+00, 1.18676917e+00
, 5.11967633e-01, 1.28159856e+00, 1.68257140e+00, -1.45429579e+00
, 7.37992814e-01, -3.13833775e-01, -2.36384065e-01, 2.39000231e+00
, -1.13071734e+00, -2.15551674e-01, 7.06773267e-01, 1.24026000e+00
, 3.31196233e+00, -2.80934925e+00, -2.70676026e-01, 2.51773927e+00
, 2.75172941e+00, 3.27117161e+00, -3.15860714e+00, 8.73311068e-01
, 2.48942632e+00, 2.41422698e+00, 2.17283531e+00, -2.89391240e+00
, 1.93921713e+00, 1.73801437e+00, 1.49028309e+00, 2.74509100e+00
, -1.78934362e+00, 1.46492456e+00, -3.18648174e-01, -7.67231784e-01
, 5.90848896e-01, 1.28089161e+00, -2.59931874e+00, 5.69308542e-02
, 5.24040403e-01, 2.13501819e+00, -2.28464136e-01, -1.39081488e+00
, 1.76397903e+00, 2.57345867e+00, 2.07920380e+00, -4.70071076e-01
, -6.50439517e-01, 2.78756990e+00, 2.37350360e+00, 1.21889865e+00
, 3.96533847e-02, 2.07270976e-01, 1.62793631e+00, 1.72390425e+00
, -1.17826976e+00, -6.05697727e-02, 5.26893235e-01, -7.59343269e-01
, -3.02901884e-01, 1.89771149e+00, -1.01298394e+00, 1.25809902e-01
, 3.86857040e-01, 9.22495501e-01, 2.89913946e+00, -2.79842942e+00
, -2.35924799e-01, 1.78741791e+00, 1.98098145e+00, 2.45745910e+00
, -1.92949099e+00, 4.40071032e-01, 1.67750682e+00, 2.29228560e+00
, 1.91505982e+00, -2.21207919e+00, 1.56207029e+00, 1.21830856e+00
, 7.31777196e-01, 1.89957659e+00, -1.55168259e+00, 1.53547002e+00
, 2.96527969e-02, 3.32278097e-01])]
])