Kennismaking met neuronen in neurale netwerken (deel 2)

Logische Neuronen

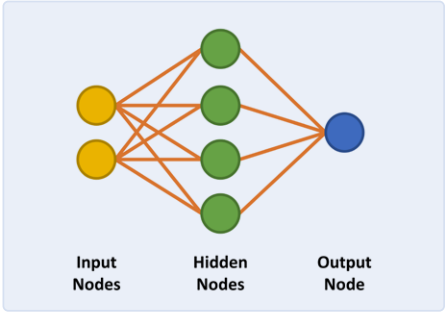
Stuart Cording (Elektor)

In het eerste deel van deze artikelenserie ontdekten we hoe onderzoekers de functionaliteit van het neuron langzaam benaderden. De echte doorbraak op het gebied van kunstmatige neuronen kwam met het multilayer perceptron (MLP) en het gebruik van backpropagation om het te leren hoe het inputs moet classificeren. Met behulp van een volledig nieuwe implementatie van een MLP in Processing, toonden we hoe het werkte en pasten we de gewichten aan om te leren. Hier gaan we terug naar de experimenten uit het verleden, om ons neuraal netwerk te leren hoe logische poorten werken en te controleren of ons MLP in staat is de XOR-functie te leren.

We hebben een flexibele Neurale klasse om een MLP te implementeren die in elk Processing-project kan worden opgenomen. Maar de tot nu toe besproken voorbeelden hebben niet veel bereikt. Zij bevestigen slechts de correcte berekening van de forward-pass en hoe backpropagation de gewichten van het netwerk aanpast om een gegeven taak te leren.
Nu is het tijd om deze kennis toe te passen op een echte taak, dezelfde taak die werd onderzocht tijdens de vroege studies van de McCulloch-Pitts Threshold Logic Units (TLU): het uitvoeren van logica. Zoals wij reeds hebben ontdekt, kan ons MLP met gemak lineair scheidbare problemen zoals AND en OR oplossen. Bovendien moet het ook in staat zijn de XOR-functie op te lossen, iets wat de TLU en andere vroege kunstmatige neuronen niet konden. Ondertussen zullen we ook onderzoeken hoe deze netwerken leren, dankzij een visuele implementatie van het neurale netwerk. En we zullen kijken naar de invloed die de gekozen leersnelheid heeft op de outputfout tijdens het leren.

AND

Hoewel een neuraal netwerk kan leren de AND-functie na te bootsen, functioneert het niet helemaal op dezelfde manier. Wat we eigenlijk doen is inputs toepassen op een netwerk dat de AND-functie heeft geleerd en het vragen: "Hoe zeker ben je ervan dat deze input-combinatie het patroon is waaraan we een 1 toekennen?"
Om dit te demonstreren is een voorbeeldproject gemaakt dat beschikbaar is in de GitHub repository in de map */processing/and/and.pde*. Dit moet worden geopend met Processing[2].
Ons neuraal netwerk heeft met een twee-ingangs AND-poort twee ingangen en een enkele uitgang. Tussen de input- en output-nodes implementeren we vier verborgen nodes (**figuur 1**). We zullen later bespreken hoe we het aantal benodigde verborgen nodes kunnen bepalen. De code om het netwerk voor te bereiden is weergegeven in **Listing 1**.
Het doel is het netwerk te trainen om het patroon '11' bij de ingangen te herkennen. We willen er ook zeker van zijn dat de alternatieven '00', '01', en '10' onze classificatiedrempel niet overschrijden. Bij het aanleren van het netwerk zullen we de stimuli en verwachte resultaten uit **tabel 1** toepassen.



Figuur 1: MLP-configuratie gebruikt om AND-functie te leren (biases niet weergegeven).

Input		Expected Output
i_1	i_2	o_1
0,01	0,01	0,01
0,01	0,99	0,01
0,99	0,01	0,01
0,99	0,99	0,99

Tabel 1: Inputs en verwachte output van MLP wanneer 'AND' volledig is aangeleerd.

Merk op dat, in plaats van te werken met logische niveaus, dergelijke netwerken werken met decimale waarden. Een 1 in dit geval, wordt ingevoerd als 0,99 (bijna 1), terwijl een 0 wordt ingevoerd als 0,01 (bijna 0). De output zal ook tussen 0,0 en 1,0 liggen. Dit moet worden geëvalueerd als een waarschijnlijkheidsniveau dat de inputs overeenkomen met de geleerde classificatie, b.v. 96,7% waarschijnlijkheid dat beide inputs '1' zijn, in plaats van de duidelijke 0/1 logische output van een echte AND-poort.
We kunnen beginnen met te bepalen of dit nieuw gecreëerde netwerk iets 'weet', door het enkele inputs te geven en het naar zijn output te vragen. Onthoud dat de gegenereerde resultaten elke keer anders zullen zijn, omdat de constructor willekeurige waarden toepast op de gewichten.
De onderstaande code geeft de respons van het netwerk op '11' en '00' bij de inputs. Het is zeer waarschijnlijk dat het resultaat voor '11' dicht bij 0,99 zal liggen, terwijl het resultaat voor '00' veel groter zal zijn dan de gehoopte 0,01:

```
// Check output of AND function
   for 00 input
network.setInputNode(0, 0.01);
network.setInputNode(1, 0.01);
network.calculateOutput();
println("For 00 input, output is: ",
network.getOutputNode(0));
// Check output of AND function
   for 11 input
```

Listing 1: Sectie van and.pde die het neurale netwerk configureert om de AND-functie te leren.

```
// We'll use two inputs, four hidden nodes, and one output node.
network = new Neural(2,4,1);

// Set learning rate here
network.setLearningRate(0.5);

println(network.getNoOfInputNodes(), " ", network.
getNoOfHiddenNodes(), " ", network.getNoOfOutputNodes());

// Set network biasing here
network.setBiasInputToHidden(0.25);
network.setBiasHiddenToOutput(0.3);

network.displayOutputNodes();

println(network.getTotalNetworkError());

network.turnLearningOn();
```

```
network.setInputNode(0, 0.99);
network.setInputNode(1, 0.99);
network.calculateOutput();
println("For 11 input, output is: ",
network.getOutputNode(0));
```

Dit leverde tijdens het testen de volgende uitvoer op:

```
For 00 input, output is: 0.7490413
For 11 input, output is: 0.80063045
```

We zien hier dat wanneer we 0,99 toepassen op beide ingangen, het neurale netwerk denkt dat de ingang '1 AND 1' is met een waarschijnlijkheid van 0,8006, wat omgerekend 80,06% is. Dit is in dit stadium niet slecht. Wanneer echter 0,01 op beide ingangen wordt toegepast, heeft het een waarschijnlijkheid van 0,7490 (74,90%) dat de ingang '1 AND 1' is. Dit is ver verwijderd van wat we willen, namelijk iets dat dicht bij 0% ligt.

Om het netwerk te leren hoe een AND-functie werkt, moet het getraind worden. Dit wordt bereikt door de inputs en de gewenste output voor alle vier de gevallen (respectievelijk 00, 01, 10, en 11, en 0, 0, 0, en 1) op de juiste manier in te stellen en na elke verandering de `calculateOutput()`-methode aan te roepen met 'learning' ingeschakeld. Dit wordt als volgt in een lus uitgevoerd:

```
while (/* learning the AND function
      */) {
// Learn 0 AND 0 = 0
network.setInputNode(0, 0.01);
network.setInputNode(1, 0.01);
network.setOutputNodeDesired(0, 0.01);
network.calculateOutput();
// Learn 0 AND 1 = 0
...
// Learn 1 AND 0 = 0
...
// Learn 1 AND 1 = 1
network.setInputNode(0, 0.99);
network.setInputNode(1, 0.99);
network.setOutputNodeDesired(0, 0.99);
network.calculateOutput();
}
network.turnLearningOff();
```

De beslissing om de training van het netwerk te stoppen kan op een aantal manieren worden genomen. Elke leercyclus wordt in dit voorbeeld beschouwd als een *epoch* (tijdvak). Het leren kan worden gestopt zodra een bepaald aantal epochs, bijvoorbeeld 10.000, is bereikt. Als alternatief kan de fout in de output worden gebruikt. Zodra deze onder, zeg, 0,01% ligt, kan het netwerk als nauwkeurig genoeg worden beschouwd voor de classificatietaak in kwestie.

Een belangrijk punt om op te merken is dat een MLP niet altijd convergeert naar het gewenste resultaat. U kunt pech hebben door de gekozen combinatie van gewichten en biasgewichten. Het kan ook zijn dat de configuratie van het MLP niet in staat is uw taak te leren, waarschijnlijk als gevolg van te veel of te weinig verborgen nodes. Hier zijn geen regels - het juiste aantal nodes, begingewichten en biases kunnen alleen worden bepaald door trial-and-error of ervaring. In dit voorbeeld werden vier verborgen nodes gekozen omdat vier ingangstoestanden moeten worden geleerd. Gehoopt werd dat elke verborgen node één toestand zou leren.

Ook moet worden opgemerkt dat de training in batches moet plaatsvinden, d.w.z. dat de set trainingsgegevens herhaaldelijk van begin tot eind moet worden doorlopen. Als '0 AND 0 = 0' gedurende enkele duizenden cycli herhaaldelijk wordt geleerd, neigt het netwerk naar dit resultaat en wordt het bijna onmogelijk om de resterende gegevens te trainen.

Visualizatie

Nu de basisimplementatie is behandeld, kunnen we het voorbeeld dat het netwerk traint om de AND-functie te leren in meer detail bekijken. Om te laten zien hoe neurale netwerken leren, wordt het netwerk in de applicatie gevisualiseerd tijdens het leren en, later, tijdens de werking.

Klikken op 'Run' zou moeten resulteren in de uitvoer zoals getoond in **Figuur 2** (screenshot). Aanvankelijk staat de toepassing in de leermodus, waarbij het netwerk de verwachte uitvoer voor een AND-functie voor de twee ingangen wordt geleerd. Links staan de invoernodes. Tijdens het leren schakelen de ingangswaarden snel tussen de logische niveaus 0 en 1. Rechts is de enige uitgangsnode. Aanvankelijk is dit 0. De beslissing om 1 uit te voeren wordt alleen genomen als de uitvoernode een betrouwbaarheid van > 90% oplevert dat beide ingangen 1 zijn. Anders wordt een 0 uitgevoerd. Deze beslissing wordt genomen tussen regel 341 en 346 in *and.pde*.

```
// Output Node Text
if (network.getOutputNode(0) > 0.9) {
text("1", 550, 280);
} else {
text("0", 550, 280);
}
```

Aanvankelijk blijft de uitvoer 0 omdat het 90%-waarschijnlijkheidsniveau nog niet is bereikt. Na ongeveer 5.000 epochs zal de outputwaarde heen en weer beginnen te flikkeren tussen 0 en 1, wat aantoont dat het netwerk met succes begint te classificeren dat de '11'-input een '1' moet opleveren. Op dit punt is de totale fout van het netwerk ongeveer 0,15%. Als dit niet gebeurt, is het waarschijnlijk dat het netwerk is vastgelopen en deze keer niet in staat zal zijn om te leren.

Terwijl het netwerk leert, worden de gewichten tussen de knooppunten weergegeven als lijnen van verschillende dikte en kleur. Hoe dikker de lijn, hoe groter de waarde. Zwarte lijnen geven positieve getallen aan, terwijl bruine lijnen negatieve getallen aangeven. Elke keer dat de code wordt uitgevoerd, zullen de lijnen anders zijn. U zult echter merken dat zich een patroon ontwikkelt. Twee verborgen nodes hebben altijd één bruine en één zwarte lijn; één verborgen node heeft twee zwarte lijnen; en één verborgen node heeft twee bruine lijnen. De lijnen tussen de verborgen nodes en de uitvoernode zullen ook een patroon volgen, waarbij de enige zwarte lijn afkomstig is van de node met twee inkomende zwarte gewichtslijnen.

Dit is een interessant inzicht omdat het laat zien hoe het netwerk de AND-functie heeft geleerd. '00' aan de ingang wordt gemakkelijk omgezet in een 0 aan de uitgang, net als '11' in een 1. Voor de '01' en '10' combinaties lijkt het erop dat de 0's veel van het zware werk doen om de uitgang in de richting van 0 te duwen.

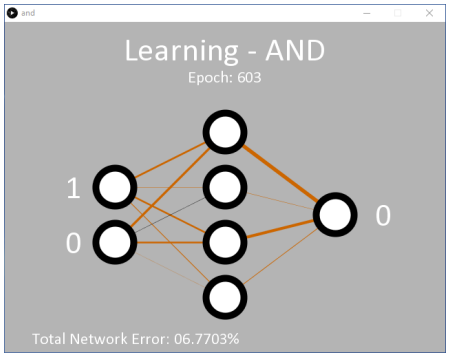
De toepassing is geprogrammeerd om te stoppen met leren zodra de totale fout van het netwerk < 0,05% is op regel 57 in *and.pde*. Als alternatief kan het leren ook worden geprogrammeerd om te stoppen na een bepaald aantal epochs op regel 55. Zodra het leren is voltooid, doorloopt de toepassing gewoon in volgorde de binaire ingangen, zodat het neurale netwerk kan laten zien wat het heeft geleerd (**figuur 3**). In de tekstconsole worden de inputs weergegeven (als een decimale waarde

tussen 0 en 3) met de berekende output in tekstformaat als volgt:

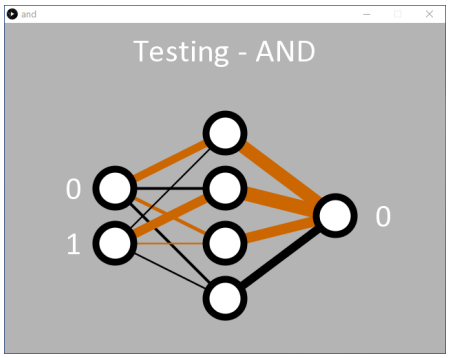
```
0 : 5.2220514E-4
1 : 0.038120847
2 : 0.04245576
3 : 0.94188505
0 : 5.2220514E-4
1 : 0.038120847
2 : 0.04245576
3 : 0.94188505
```

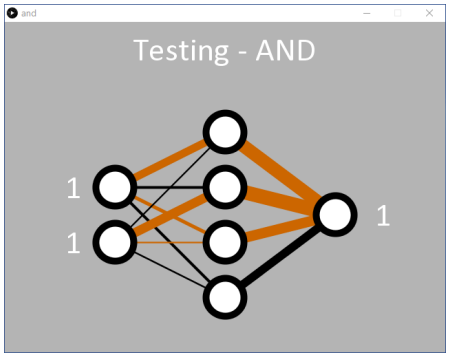
Voor geïnteresseerden: de uitvoerfout voor de toegepaste inputs en de gemiddelde netwerkfout worden om de 50 epochs weggeschreven naar een CSV-bestand met de naam *and-error.csv*. Dit kan in Excel worden geïmporteerd om na te gaan hoe het netwerk naar de oplossing convergeerde (**figuur 4**). De output laat zien hoe de fout heen en weer slingert tussen hoge en lage waarden voor specifieke input/output-combinaties. Zoals we al zagen, heeft de hoge fout waarschijnlijk te maken met de uitvoer voor de patronen '00', '01', en '10' die veel te hoog was tijdens de vroege fase van het leren. De lage fout is waarschijnlijk wanneer het netwerk de invoer '11' evalueert.

Deze individuele patroonfouten worden gemiddeld over vier epochs om de gemiddelde netwerkfout te berekenen. Als uw PC geen Engelse locale gebruikt, wilt u misschien de ',' in het CSV-bestand in een teksteditor (zoals Notepad++) vervangen door een ';' als scheidingsteken, en vervolgens de '.' door ',' voordat u de gegevens importeert naar Excel.
Het CSV-bestand is ook interessant voor het bekijken van de impact die de leersnelheid heeft op het netwerk. De voorbeeldcode gebruikt een leersnelheid η van 0,5. Hogere getallen zorgen ervoor dat het

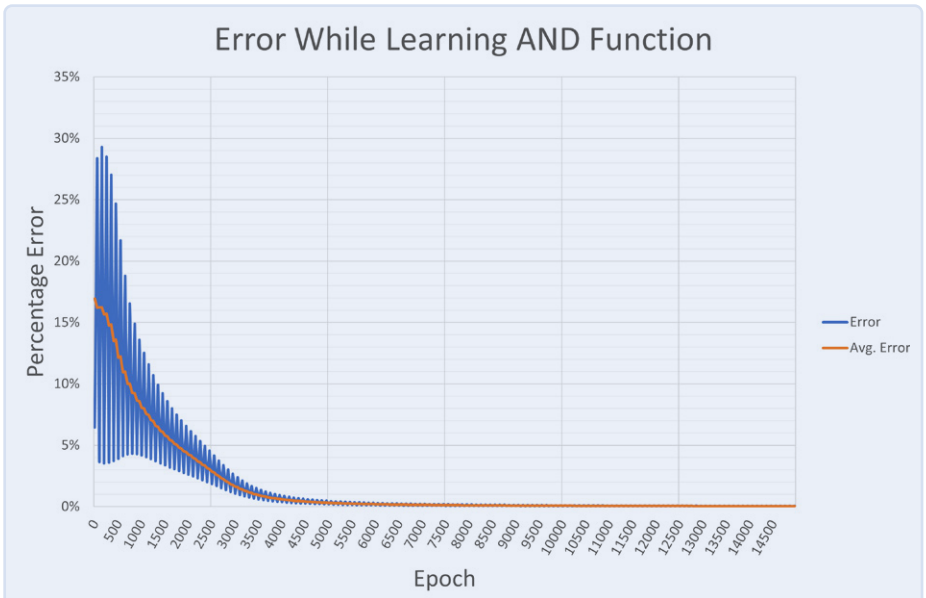


Figuur 2: Screenshot van het neurale netwerk tijdens de leerfase van AND.

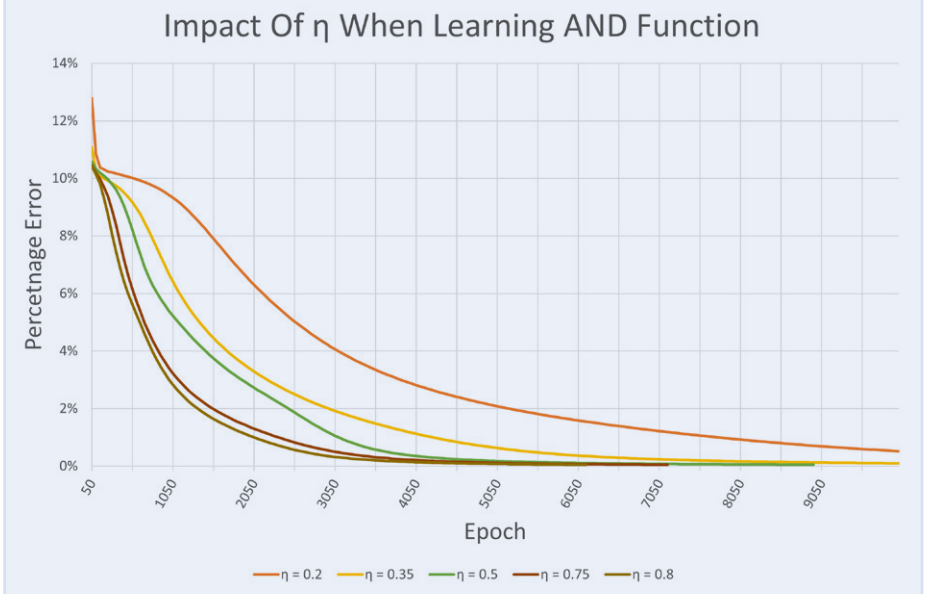




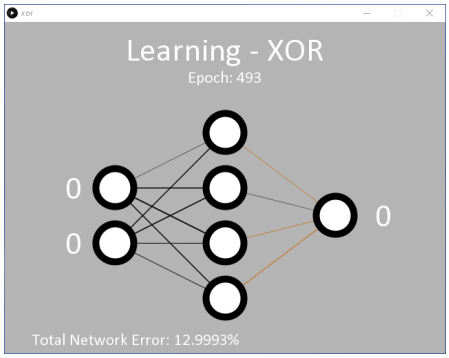
Figuur 3: Zodra het AND-patroon is aangeleerd, doorloopt de toepassing de vier binaire ingangscombinaties en geeft zij de uitvoerrespons van het netwerk weer.



Figuur 4: Fout om de 50 epochs en gemiddelde fout tijdens het leren van AND.



Figuur 5: Effect van leersnelheid η op de fout tijdens leren (eerste 10.000 epochs).

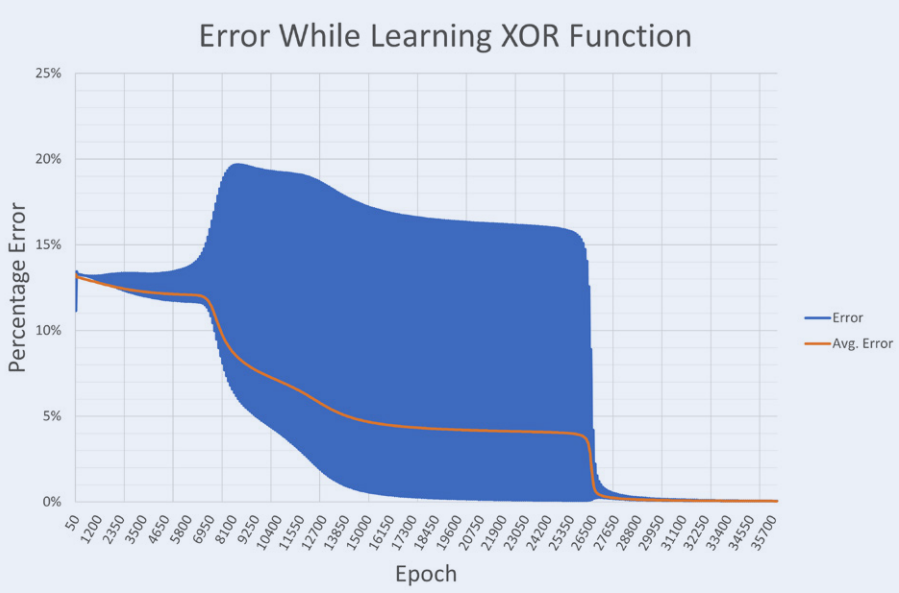


Figuur 6: De MLP heeft tijdens het leren van XOR moeite om de juiste gewichten te vinden.

netwerk sneller leert, zoals te zien is in **figuur 5**. Ze kunnen echter ook oscillaties veroorzaken en resulteren in een netwerk dat nooit convergeert naar het gewenste leerresultaat. Alle hier geteste leersnelheden resulteerden in een correct functionerend netwerk dat AND had geleerd. Er zij echter op gewezen dat de begingewichten telkens willekeurig werden gekozen.

Maar Kan Het XOR Leren?

De repository bevat ook voorbeelden voor een OR-functie in *processing/or/or.pde*. Omdat deze functie lineair scheidbaar is, heeft het MLP ook geen moeite met het leren van deze functie. De lezer kan het nuttig vinden om het verschil in de gewichten na het leren te onderzoeken in vergelijking met het AND-voorbeeld. Zowel *or.pde* als *and.pde* kunnen gemakkelijk worden aangepast om het netwerk de NAND- en NOR-functies te leren. Het moment van de waarheid komt echter met de XOR-functie. Een voorbeeld hiervan wordt gegeven in *processing/xor/xor* dat net zo werkt als de vorige code en gebruik maakt van dezelfde 2/4/1 (input/verborgen/output) MLP-nodeconfiguratie (**figuur 6**). Bij de gebruikte leersnelheid (η = 0,5) zal het waarschijnlijk 15.000 epochs of meer vergen voordat de output begint te veranderen. Ongeveer 35.000 epochs zijn nodig voordat de beoogde gemiddelde fout van 0,05% is bereikt.



Figuur 7: De uitdagingen die het MLP ondervond bij het leren van XOR worden weerspiegeld in de output-fout.

Het is duidelijk dat het netwerk moeite heeft om de XOR-functie te leren. Dit wordt weerspiegeld in de weergegeven gewichten die flikkeren tussen positief en negatief alvorens een richting te kiezen, en de netwerkfout daalt zeer geleidelijk. Dit komt doordat '00' en '11' (voorgesteld als 0,01 en 0,01, en 0,99 en 0,99 bij de ingangen) beide de output 0,01 moeten opleveren. Wiskundig gezien resulteren invoerwaarden van 0,99 in hoge uitvoerwaarden totdat het netwerk tijdens het leren in staat is het resultaat omlaag te duwen in de richting van 0,01. Dit is te zien aan de uitvoerfout die tijdens het leren in *xor-error.csv* wordt opgeslagen (**figuur 7**).

Ondanks de uitdagingen van de taak, leert het netwerk de XOR-functie zoals gevraagd. Zodra de fout onder 0,05% ligt, past de Processingcode plichtsgetrouw de binaire inputs toe op het netwerk, en de output reageert, waarbij de patronen '01' en '10' correct worden gedetecteerd. De code geeft dan een 1 weer bij de uitvoernode (**figuur 8**). Net als bij de AND-code kunnen we zien hoe het netwerk de XOR-functie heeft geleerd. Twee verborgen nodes hebben een binnenkomende zwarte en een bruine lijn en een dikke zwarte lijn die ze verlaat (de twee middelste nodes van figuur 8). Deze lijken verantwoordelijk te zijn voor de '01' en '10' classificatie. Het netwerk loste ook vrij goed '00' aan de ingangen op in 0 aan de uitgang (bovenste verborgen node). In dit geval lijkt de '11'-ingang te worden verwerkt door de onderste verborgen node, maar dit is mogelijk tijdens het leren niet goed opgelost, wat resulteerde in een hogere fout dan gewenst voor de ingang '11'. Als de code

opnieuw wordt uitgevoerd, zal waarschijnlijk één verborgen node de '11' verwerken, met twee zwarte lijnen die één van de verborgen nodes binnenkomen (**figuur 9**).

Voor de volgende keer

Een van de belangrijkste dingen om mee te nemen is, dat er met neurale netwerken geen goed of fout antwoord is. Het netwerk zelf classificeert slechts hoe waarschijnlijk de inputs zijn die u hebt gegeven. Als u het hebt geconfigureerd, getraind en het levert het gewenste resultaat, dan is het waarschijnlijk juist. Idealiter wil je dit ook bereiken met een minimum aantal nodes om geheugen en rekentijd te besparen. Hoewel de visualisatie leuk is, is het niet helemaal noodzakelijk. Als u meer wilt weten, kunt u het *processing/fsxor/fsxor.pde* project aanpassen, zodat de visualisaties wegvallen. Het verwijderen van de code die de foutwaarden naar een CSV-bestand schrijft, versnelt de code ook aanzienlijk. U kunt dan uw eigen code schrijven met behulp van de `Neural`-klasse om het volgende te onderzoeken:

- Welke invloed heeft de leersnelheid op het netwerk bij het leren van XOR? Misschien ook telkens dezelfde startgewichten proberen.
- Kun je de gewichten initialiseren met waarden die het netwerk aanmoedigen om in minder epochs op te lossen? Herzie misschien de outputgewichten van een eerdere run.
- Hoe weinig verborgen nodes heb je nodig om AND te leren? Hoe weinig om XOR te leren? Kun je te veel verborgen nodes hebben?
- Heeft het zin om twee uitgangsnodes te hebben? De ene zou de ongewenste patronen kunnen classificeren als 0,99 (voor XOR, '00' en '11'), terwijl de tweede wordt gebruikt om de gewenste patronen te classificeren als 0,99 (voor XOR, '10' en '01').

In het volgende artikel over neurale netwerken zullen we het neurale netwerk trainen om kleuren te herkennen van een webcam die op onze PC is aangesloten. Maar waarom ontwikkel en test u niet zelf een MLP-nodeconfiguratie die volgens u geschikt is voor deze taak?

210160-B-03

Vragen of opmerkingen?

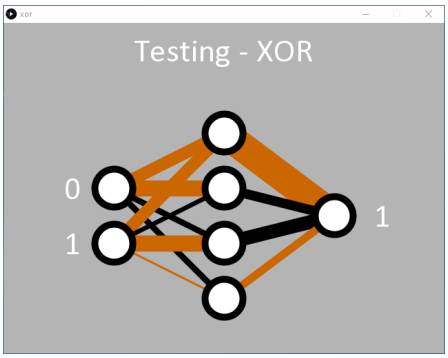
Heeft u vragen of opmerkingen over dit artikel? Stuur dan een e-mail naar de auteur op stuart.cording@elektor.com.

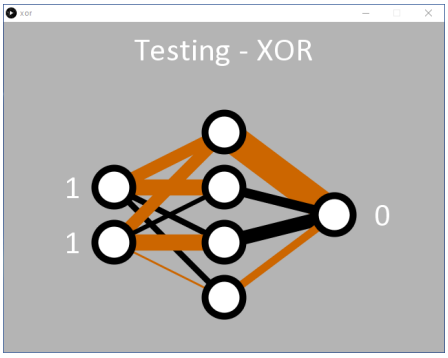
Een bijdrage van

Idee, tekst en afbeeldingen: **Stuart Cording**
Redactie: **Jens Nickel** and **C. J. Abate**
Illustraties: **Patrick Wielders**
Layout: **Harmen Heida**
Vertaling: **Jelle Aarnoudse**

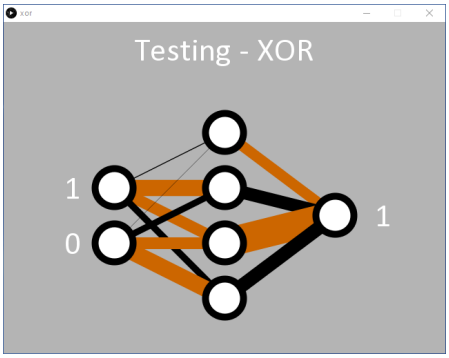
Creatief leren

AI en ML worden vaak toegepast om afbeeldingen en complexe data-verzamelingen te classificeren, maar kan het ook creatief zijn? Het FlowMachines-project van SONY CSL Research Laboratory zou suggereren van wel. Met 'Daddy's Car' lijkt hun AI een Beatles-achtige te hebben geschreven. Maar als je wat dieper graaft, blijkt dat er veel menselijke inbreng nodig is geweest om dit nummer te arrangeren, te produceren en de tekst te schrijven. Creatief of niet, het proces is innovatief en spannend; dergelijke AI kan een waardevolle sparring partner zijn als de menselijke inspiratie het plotseling laat afweten.





Figuur 8: Het netwerk toont dat het met succes de functionaliteit van XOR heeft geleerd.



Figuur 9: Na het opnieuw doorlopen van de XOR-code is de bovenste verborgen node duidelijk verantwoordelijk voor de '11'-classificatie, terwijl de derde verborgen node verantwoordelijk is voor '00.'

WEBLINKS

[1] GitHub repository - simple-neural-network: http://bit.ly/2ZHLv9p
[2] Processing IDE: https://processing.org/
[3] J. Brownlee, "How to Configure the Number of Layers and Nodes in a Neural Network," July 2018: http://bit.ly/3aMHqam
[4] T. Lee, "'Daddy's Car' Is A Pop Song Composed By Artificial Intelligence," Übergizmo, September 2016: http://bit.ly/3shyfo6