Samenvatting Computational Intelligence – A.B. van Ramshorst

Computational Intelligence Technieken

- Particle Swarm Optimisation (PSO)
- Evolutionary Strategies/ Genetische Algoritmes
- Artificial Neural Networks(ANN)
- Associative memory (BAM/Hopfield)
- Reinforcement Learning

ANN

Een artificial neural network heeft vaak een aantal inputs, een hidden layer die naarmate de input/output computationi complexer wordt groter wordt en een aantal outputs. I/O zijn afhankelijk van je probleem, en de hidden layer-grootte van de complexiteit van het probleem. Vaak gebruik je **Backpropagation** en de **delta regel** om de weigths te trainen:

$$\Delta w_{jk}(p) = \beta * \Delta w_{jk}(p-1) + \alpha * \ y_j(p) * \delta_k(p)$$

Hierbij zijn w de weights, p staat voor de p-de epoch, Δw de weight update, alpha de learning rate en beta is het momentum. Hoe lager beta hoe minder momentum en maximaal als b = 1. δ_k staat voor de error-gradiënt. De error gradiënt is de afgeleide van de activatie functie maal de error. Voor een Sigmoid functie is de error gradient dus als volgt gedefinieerd:

$$\delta_k(p) = \ y_k(p) * \big(1 - \ y_k(p)\big) * \ e_k(p)$$

Hier is e de error (gemeten – gewenst = error) en y is Sigmoid(X)

Je kan ook **Hebb Learning** toe passen. Dit kan alleen voor netwerken waarin binaire signalen lopen. Wat er dan gebeurt, is dat een positieve weight update alleen optreedt als een rij neurons afgevuurd wordt van begin tot eind. Deze verbinding(en) worden dan versterkt met de volgende leer regel:

$$\Delta w_{jk}(p) = \alpha * y_k(p) * x_j(p) - +\emptyset * y_k(p) * w_{jk}(p)$$

Waarbij y en x [0,1] zijn en respectievelijk de outputs van de tweede en eerste laag. $\emptyset$ is de **forgetting factor** die ervoor zorgt dat een weinig gebruikte verbinding ook afzwakt als de y neuron door een andere oorzaak gevuurd wordt.. Let op het verschil tussen de weight rechts en de delta weight links.

In deze netwerken kan ook een winner takes all functionaliteit opgenomen worden die ervoor zorgt dat maar 1 rij tegelijk kan vuren, de sterkste. Dit is een **unsupervised learning** techniek. Een toepassing kan bv een **mexican-hat** functie zijn, dan vuurt de sterkste en heeft die invloed op omringende neuronen. Een **Self-Organising Map (SOM)** kan hiermee gemaakt worden met behulp van een **Kohonen** netwerk. Een Kohonen netwerk heeft op de uitvoer niet een 1d rij neuronen maar 2 dimensionaal, een vlak dus, en bestaat uit twee lagen.

Er zijn ook ANN's die werken met een **Jordan** of **Elman** opbouw. Deze netwerken worden recursief genoemd vanwege het feit dat ze feedback/terugkoppeling hebben tussen de lagen.

PSO

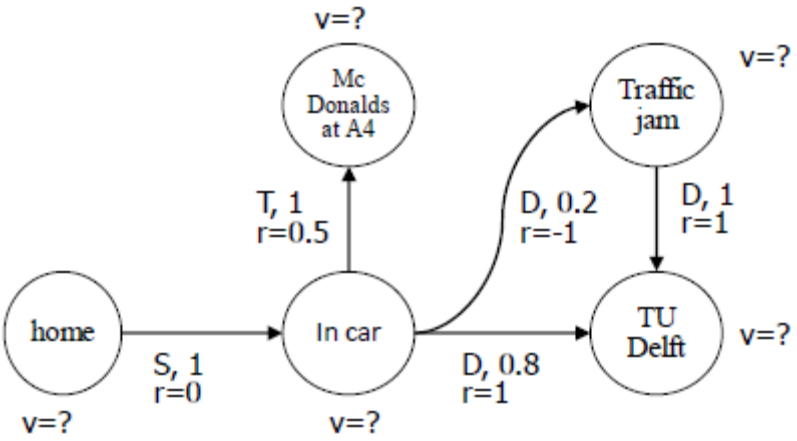
Particle swarm optimization is gebaseerd op een hoeveelheid particles die door de probleemruimte bewegen aan de hand van een aantal formules en op die manier op zoek zijn naar een goede oplossing. de probleem ruimte heeft een j aantal dimensies met een i aantal deeltjes die daar doorheen bewegen. De formule voor de snelheid op de verscheidene dimensie vlakken wordt dan als volgt berekent:

$$v_{ij}(t+1) = \theta v_{ij}(t) + c_1 r_{1j}\left[y_{ij}(t) - x_{ij}(t)\right] + c_2 r_{2j}[\hat{y}_j(t) - x_{ij}(t)]$$

De eerste term aan de linkerhand is de snelheid van het deeltje i in vlak j. Dit zorgt ervoor dat het deeltje een logisch pad kan volgen in plaats van over de hele ruimte heen en weer springen. De eerste term tussen de haakjes $[y_{ij}(t) - x_{ij}(t)]$ heet het cognitieve/nostalgische component. Deze term geeft het verschil aan tussen de huidige fittheid x en de maximale fitness y die het deeltje gevonden heeft. Bij een groot verschil zal het deeltje de neiging hebben terug te keren naar zijn maximum. De tweede $[\hat{y}_j(t) - x_{ij}(t)]$ term is het sociale component. Dit is het maximum gevonden door de gehele populatie deeltjes. Deze term creëert de neiging om naar dat optimum te bewegen. De parameters c1 en c2 zijn de mate waarin het deeltje waarde hecht aan enerzijds zijn eigen ervaringen en anderzijds de ervaringen van de groep. Meestal geldt dat c1,c2 ongeveer gelijk zijn. De parameters r1 en r2 zijn willekeurige waardes op het interval [0,1]. Ze zorgen ervoor dat door middel van hun willekeurigheid niet alle particles hetzelfde gedrag vertonen en op die manier wordt het hele probleem gebied doorzocht/ontdekt. Er kan eventueel in de v(t) term ook nog een inertia factor toegevoegd worden. Die zorgt er in dat geval voor wat belangrijker is: zijn vorige snelheid en richting of de ervaringen van de particle zelf/ de groep.

Reinforcement Learning

Het zogeheten **Markov Decision Process (MDP)** is een manier om de aantrekkelijkheid van bepaalde keuzes in een geheel proces te bepalen. Dit doe je aan de hand van een keuze-functie, rewards voor een handeling (Het is namelijk Reinforcement Learning) en het gegeven dat bij een bepaalde keuze een kansverdeling is wat betreft de uitkomst. Als je een bepaalde route kiest bijvorbeeld hoeft eentje niet *altijd* de snelste te zijn. Er kan file staan oid. Dit wordt in een MDP ook meegenomen. Als voorbeeld de volgende route:



De waardes voor V en de waardes voor sas' worden als volgt bepaald (**Bellman equation**):

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

Hierin is de functie π een manier van bepalen welke routes gekozen worden, alleen de beste of met een roulette wiel etc. De s staat voor de huidige state of node waar je bent en s' staat voor de state waar je naartoe kan. Als s' niet bestaat zoals bij de node TUDelft dan geldt $V^{pi}(s)=0$. De P term is de kans op de route na het maken van de keuze. Vanuit de node incar naar Mcdonalds geldt P=1, vanuit incar naar TUDelft geldt P=0.8 enzovoorts. R is de reward toegekend op de route. Deze kan zowel positief als negatief zijn en is aangegeven met r={-1,1}. De laatste is gamma. Gamma staat voor de zogeheten discount factor. Deze bepaalt in hoeverre een State Value (V(s')) van de bestemmings-node invloed heeft op de Value van de huidige node.

Aanpak:
Je begint bij de eind nodes, deze hebben allen de value 0. Hier kan immers geen keuze of reinforcement meer verkregen worden. Dan ga je terugrekenen: een node waarvan alle richtingen naar bekende nodes leiden kunnen ook berekend worden. Voor elke keuze (vanuit in car bv. T en D) bereken je dan de tweede sommatie. De eerste term in die sommatie is 0.5, er zijn twee keuzes die even zwaar mee wegen. Er kan ook gekozen worden voor een MAX() functie. In dat geval wordt alleen de keuze met de hoogste Value returnwaarde gebruikt om V(s) te bepalen. Op deze manier loop je het hele pad af totdat je voor elke node de Value berekend hebt.

Genetische Algoritmes/Evolutionaire Strategie

Genetische algoritmes werken aan de hand van het principe dat er een generatie Chromosomen is. Deze worden getoetst op fitness en vervolgens wordt er met bepaalde functies een nieuwe generatie gecreëerd van de chromosomen die het best waren. Dit kan door middel van:

- Klonen
- Cross-over
- Mutaties

Volgens de **schema theorie** is (schema als in een patroon) zal het aantal schemata/chromosomen met een boven gemiddelde fitness toe nemen met elke generatie.

Klonen is het 1 op 1 kopiëren van een chromosoom naar een nieuwe generatie. Voordeel is dat je zeker weet dat hij een bepaalde fitness heeft, een nadeel is dat je met de kloon techniek geneigd ben te blijven steken op een lokaal maximum/minimum. Deze techniek wordt vaak gebruikt bij elitisme: het niet afsterven van de fitste deeltjes.

Bij cross-over wordt er een aantal genen van een fit chromosoom gecombineerd met een aantal genen van een ander fit chromosoom. Deze twee worden de *parents* genoemd en samen door te combineren kan je van de genen uit deze twee parents een setje van twee *childs* maken.

Bij mutatie wordt er een bewerking toegepast op een chromosoom door simpelweg 1 bit te *flippen*. Dit moet uiteraard wel toegestaan zijn voor de geldigheid van het chromosoom. Een andere voor van mutatie kan ook het omwisselen van plek van twee genen of een genen sequence te inverteren.

De mate waarin deze functies aanpassingen maken kan ook gevarieerd worden. Dit heet **self adaption**.

Associative Memory

Associative memory wordt gebruikt om bekende input te matchen met opgegeven output. Het doel is het herkennen van beelden of patterns, ook al zijn deze incompleet of gecorrumpeerd. Dit maakt de techniek geschikt voor patroon herkenning en error correctie.

Een voorbeeld van zo'n network is een **Hopfield** network. Omdat het netwerk een gegeven patroon *x* moet herkennen en reproduceren, zal het aantal inputs gelijk zijn aan het aantal outputs. Het is een netwerk met maar 1 layer waarin alles aan elkaar gelinkt is, zonder feedback. De gebruikte activatie functie geeft {-1,0,1} en is afhankelijk of de input kleiner, groter of gelijk aan nul is.

Een hopfield network heeft echter maar weinig capaciteit om patterns op te slaan. Het maximale aantal patterns die opgeslagen kunnen worden is afhankelijk van het aantal neutronen in het netwerk. Dit geldt ook voor het aantal succesvol te herkennen patterns, dit aantal is lager dan het aantal opgeslagen patterns! Maar de complexiteit van het hopfield network neemt zeer snel toe naarmate er meer neuronen toegevoegd worden.

Bidirectional Associative Memory (BAM) is een ander soort associative. Deze heeft twee volledig gelinkte Layers met een n-aantal inputs en een m-aantal outputs. Deze linkt dus sets aan elkaar die niet hetzelfde hoeven te zijn. De input vector kan ook ongelijk zijn in grootte vergeleken met de output vector.