REPRESENTATION OF SPATIAL KNOWLEDGE IN AN ARTIFICIAL SYSTEM

Universiteit Twente

Bachelor These HFM

1e Begeleider: Prof. Dr. Frank van der Velde
2e Begeleider: Dr. Martin Schmettow

Jurre Stienen (s0209805)
J.Stienen@student.utwente.nl

Samenvatting

In deze bachelor scriptie is onderzocht hoe spatiale kennis gerepresenteerd dient te worden in een kunstmatig systeem. Hierbij is vooral gekeken naar hoe dit werkt in het menselijk lichaam. Uit literatuuronderzoek blijkt dat mensen leren door gebruik te maken van een interactie tussen de hippocampus en de neocortex, waarbij verschillende signalen bij elkaar opgeslagen worden als 'bij elkaar horend'. Ook blijkt dat naarmate deze signalen vaker samen tegen gekomen worden de onderlinge connectie versterkt wordt. Dit paradigma wordt in de wetenschap *reinforcement learning* genoemd en is tevens het leermechanisme dat in dit onderzoek wordt toegepast. Voor de opslag van de data is een reservoir opgebouwd volgens de principes van Reservoir Computing, we geven een input en een output laag, de computer genereert zelf een random reservoir van neuronen om de onderliggende verbanden op te kunnen slaan. Dit alles is geschreven in python met behulp van de *PyBrain library*. Om te kunnen verifiëren dat het systeem daadwerkelijk geleerd heeft zijn enkele simulaties uitgevoerd met drie verschillende leeralgoritmes die gebaseerd zijn op *reinforcement learning*. Er werd voor alle drie algoritmes een significant effect gevonden van het aantal interacties op de leerprestaties van het systeem waarmee is aangetoond dat het systeem daadwerkelijk ook geleerd heeft. Deze effecten werden ook geïllustreerd in de visuele representaties die het programma ook produceert.

Trefwoorden: Reservoir Computing, Reinforcement leren, PyBrain, Machine leren, spatiale kennis

Abstract

In this bachelor thesis research was conducted as to how spatial knowledge should be represented in an artificial system. This representation was to be inspired on the way this works in the human body. Literature studies pointed out that when people learn, they make use of an interaction between the hippocampus and the neocortex, where several input signals will be labeled as 'being connected'. They also pointed out that as these signals would appear together more often, they would get an even stronger connection. This paradigm is called *reinforcement learning* in the scientific world and is also the learning mechanism being used in this research. For data storage this study is using a reservoir of neurons which is constructed according to the principles of *reservoir computing*; the user specifies an input and an output layer, the system itself will generate a random reservoir in which it will store the relationships between different signals. This program is written in the python programming language using the *PyBrain Library*. To be able to verify the system had actually learned something simulations were conducted using three different learning algorithms based on *reinforcement learning.* For all three conditions a significant effect was found for the effect the number of interactions had on the learning capabilities of the system. This shows our system actually learned its way around the maze, which is also illustrated in the visual representations the program also produces.

Keywords: Reservoir Computing, Reinforcement Learning, Pybrain, Machine Learning, Spatial Knowledge

Inhoudsopgave

1. Inleiding

De universiteit Twente is op dit moment aan het werk om een zogenoemde iCub te bemachtigen. Deze iCub is een zeer geavanceerde mensachtige robot en vervaagt de grenzen tussen mens en machine. De iCub beschikt over een huid, vingers en andere menselijke kenmerken waardoor hij dingen kan aftasten en druk kan voelen. Dit zorgt voor een natuurlijkere interactie met mensen (Utwente, 2013).

Bij een nieuwe robot die beweegt en er uit ziet als een mens, zou het natuurlijk ook zeer mooi zijn als hij zou kunnen leren als een mens. Als een robot kan leren en begrijpen hoe een mens zich gedraagt, is dit een zeer interessante ontwikkeling voor de zorg-branche. Denk bijvoorbeeld aan een iets blindengeleidehond of een robot die zelfstandig mensen in en uit bed zou kunnen helpen in het ziekenhuis (Utwente, 2013).

Dit klinkt nog voor veel mensen als toekomstmuziek in de oren. Echter ziet ook een instantie als de Europese Unie het belang in van onderzoek in deze richting. In een recent project van de EU genaamd het *Human Brain Project* worden wetenschappers uitgedaagd om het menselijke brein te doorgronden. Aan de hand van deze informatie willen ze het mogelijk maken een digitale versie te creëren van dit menselijke brein, om dit te kunnen gebruiken in de robotica (Human Brain Project, 2013). Echter snijdt het mes aan twee kanten. Door de menselijke kenmerken te simuleren op computersystemen komt men steeds meer te weten over de hersenen zelf. De urgentie van het gebruik van de simulaties is terug te vinden in de begroting van het Human Brain Project. Van de 8.2 Miljoen euro die is uitgetrokken voor expertise van buitenaf is 2.5 Miljoen euro uitgetrokken voor het gebied van virtuele robotische omgevingen, agents, sensor- en motorsystemen. Ter indicatie; de andere 6 onderzoeksgebieden moeten het doen met een budget tussen de 581.250 € en 937.500 € (Human Brain Project, 2013).

Het gebied van `denken en leren als een mens´ is natuurlijk erg breed, maar we moeten ergens beginnen. Voor dit onderzoek is gekozen om in te gaan op de manier waarop een robot kan weten waar hij is. We willen graag zo dicht mogelijk bij de menselijke manier van denken en leren blijven, daarom zal eerst gekeken worden naar hoe een mens denkt en leert. Vervolgens moet een gepaste technologie gevonden worden om dit op te kunnen modeleren. Hier zullen nog enige haken en ogen aan vast zitten, die opgelost moeten worden. Aan het einde van dit onderzoek is het de bedoeling een systeem gecreëerd te hebben dat in een compleet onbekende omgeving geplaatst kan worden, en dan zelf zijn weg kan vinden naar een bepaald doel. Dit terwijl er gebruik gemaakt wordt van een manier van dataverwerking die sterk lijkt op die van een mens.

1.1 Kennis representatie

Leren is een vorm van kennis opslaan. Als de menselijke manier van leren nagebootst moet worden moet eerst gekeken worden naar hoe menselijke kennis in de hersenen opgeslagen wordt. De filosoof Heil beschrijft in zijn boek Philosophy of Mind (2012) een aantal paradigma's met betrekking tot het 'denken' van mensen. Niet elk paradigma past even goed meer in onze case. Heil heeft het bijvoorbeeld over William Lycan die beweert dat mensen niets anders zijn dan een collectie "scanners" die allemaal reageren op een signaal waar ze voor geprogrammeerd zijn (Heil, 2012). De behavioristen beweren dat eigen wil niets meer dan een illusie is en dat we gewoon geconditioneerd zijn om op een bepaalde manier te reageren op bepaalde stimuli (Heil, 2012). Ook zijn er zogeheten eigenschapsdualisten die denken dat denken en het bewustzijn niets anders is dan een bijverschijnsel van hersenactiviteit (Heil, 2012). Een relatief nieuwe stroming in dit gebied is de 'Global Workspace Theorie'. Deze theorie beweert dat de hersenen een gebied bevatten dat alle data bevat en een ander gebied interpreteert informatie uit de verschillende zintuigen, slaat de nuttige data op en kan ook de data weer uitlezen (Heil, 2012).

Deze laatste theorie vertoont veel overeenkomsten met de manier waarop een computer werkt en is dus zeer interessant als de kennisopslag van mensen in robots gemodelleerd gaan worden. Een computer heeft een centrale harde schijf die alle opgeslagen data bevat en een werkgeheugen dat de berekeningen op zich neemt. Echter in recentere studies blijkt dat deze analogie niet helemaal opgaat. Om te beginnen is het nog niet helemaal duidelijk waar onze kennis precies opgeslagen wordt. Onderzoekers zijn al redelijk lang bezig deze zogenoemde *Engram* te zoeken, zonder al te veel succes (Kalat, 2009).

In de menselijke hersenen zijn verschillende gebieden verantwoordelijk voor verschillende manieren van kennis opslaan. (Yonelinas, 2002). De manier van opslaan hangt sterk af van de manier waarop de kennis weer opgehaald dient te worden. Yonelinas spreekt over herkenning (*recognition*) en een algemene bekendheid (*familiarity*). Herkenning kun je omschrijven als het precies herkennen wat er staat, en ook het precies weten in welke context het geleerd is. Terwijl algemene bekendheid betekent dat gevoel het te kennen, ook weten waar het mee te maken heeft maar niet helemaal precies kunnen plaatsen waar het precies vandaan komt. In de praktijk complementeren deze soorten kennis elkaar en is het mogelijk lastig om een voorstelling te maken van elk afzonderlijk (Kalat, 2009).

Beiden manieren van kennis vergaren veronderstellen een verschillende manier van informatie verwerking (O'Reilly & Norman, 2002). Hierdoor is beredeneerd dat de verschillende manieren onder gebracht zijn in zeer verschillende, maar ook zeer interactieve, hersengebieden (Norman & O'Reilly, 2003). De hippocampus is gespecialiseerd in het snel en automatisch coderen van arbitraire combinaties van bestaande representaties uit de hersenschors. Terwijl de neocortex gespecialiseerd is in het langzaam ontwikkelen van een statistische representatie van de werkelijkheid. De hippocampus wijst een specifieke combinatie van input patronen toe aan

verschillende representaties om interferentie tussen data te voorkomen terwijl de neocortex overlappende representaties gebruikt om algemene kennis te vergaren. (O'Reilly & Norman, 2002).

Een interessante case studie in deze context is het leven van de enigszins beroemde patiënt H.M. Deze man heeft vanwege medische redenen ervoor gekozen om zijn hippocampus te laten verwijderen. Nadat deze operatie was uitgevoerd bleken er een aantal dingen gebeurd te zijn. Hoewel hij nog steeds beschikte over een normaal werkgeheugen had hij erg veel moeite met het vormen van nieuwe herinneringen. Hij kon geen enkele gebeurtenis omschrijven die na zijn operatie was gebeurd (Kalat, 2009). Echter hadden de overgebleven delen van zijn brein wel een erg zwak semantisch geheugen gevormd (Corkin, 2002). Hij wist de achternamen bij de voornamen te noemen van mensen die regelmatig in het nieuws kwamen na zijn operatie zoals: Presley(Elvis), King(Martin Luther) en Castro(Fidel) (Corkin, 2002). Ook zijn er nog andere gevallen bekend van mensen met schade aan de hippocampus die wel nieuwe vaardigheden hebben ontwikkeld maar vervolgens geen idee hebben hoe ze die kennis hebben opgepikt (Kalat, 2009). Kort samengevat neemt de hippocampus het declaratieve geheugen voor zijn rekening en zorgt de neocortex voor het procedurele geheugen. In lekentermen is dit waarschijnlijk het beste uit te leggen door te stellen dat de hippocampus met zijn herkenning gespecialiseerd is in het ophalen van ‘Wat-kennis’ en de neocortex met zijn algemene herkenning is gespecialiseerd in het ophalen van ‘Hoe-kennis’.

Een geheel andere vorm van geheugen is het zogenaamde spatiele geheugen. Kennis dat in het spatiele geheugen zit is kennis dat te maken heeft met de omgeving waarin men zich bevindt. Uit onderzoek blijkt dat de hippocampus een grote rol speelt bij het spatiale geheugen (Kalat, 2009). Kalat haalt een onderzoek aan van Maguire et al (Maguire, et al., 2000) waarin Londense taxichauffeurs in een PET-scan gelegd werden. Zij kregen vragen gesteld om zich een route voor te stellen in Londen en andere niet spatiele controle vragen. Bij het beantwoorden van de spatiele

vragen werd de hippocampus vele malen sterker geactiveerd dan bij het beantwoorden van de controle vragen. MRI scans hebben ook aangetoond dat taxichauffeurs een groter dan gemiddelde posterior hippocampus hadden dan andere mensen en bovendien dat hoe langer de chauffeurs in het vak zaten hoe groter deze posterior hippocampus was (Kalat, 2009; Maguire, et al., 2000). Verder haalt Kalat verschillende andere onderzoeken aan met knaagdieren waarbij de hypothese dat de hippocampus belangrijk is voor het spatiele geheugen getest en bevestigd wordt (Kalat, 2009).

Nu weer terug naar het oorspronkelijke onderwerp van onze robots. Een belangrijke eigenschap van een dergelijke robot is dat hij 'weet waar hij is' en zo zelfstandig zijn weg kan zoeken door een 'doolhof' van de dagelijkse wereld. Dit doolhof kan niet van tevoren in het geheugen van de robot geladen worden omdat dit doolhof steeds veranderd. Een robot zal dus moeten leren hoe hij met verschillende obstakels om zal moeten gaan. Hiermee komen we uit bij de hoofdvragen van deze scriptie:

Hoe dient de spatiale kennis representatie van een robot eruit te zien zodat hij in een doolhof kan leren navigeren?

Wat kunnen we leren van de manier hoe dit werkt bij de mens?

De literatuur beschrijft Reservoir Computing (RC) als een ideale techniek om van gebruik te maken bij het modeleren van de kennis opslag, de zogeheten architectuur. Dat komt omdat RC de niet volledige verbonden structuur van de hippocampus goed kan nabootsen, dit wordt *sparse connectivity* genoemd. (Schrauwen, Verstraeten, & Campenhout, 2007). Echter zijn we er dan nog niet want het systeem zal ook moeten kunnen leren. Hoewel er verschillende leermechanismen mogelijk zijn lijkt reinforcement learning een goede plek om te beginnen.

Om een antwoord te kunnen vinden op de hoofdvragen moeten eerst nog de volgende vragen beantwoord worden:

Wat is reservoir computing en hoe is het ontstaan?

Welke vragen moet de robot kunnen beantwoorden om te weten waar hij is?

Wat is reinforcement learning en welk algoritme past het best in ons systeem?

Hoe vertaal je van real-time naar computertermen?

1.2 Wat is Reservoir Computing?

1.2.1 Neurale Netwerken

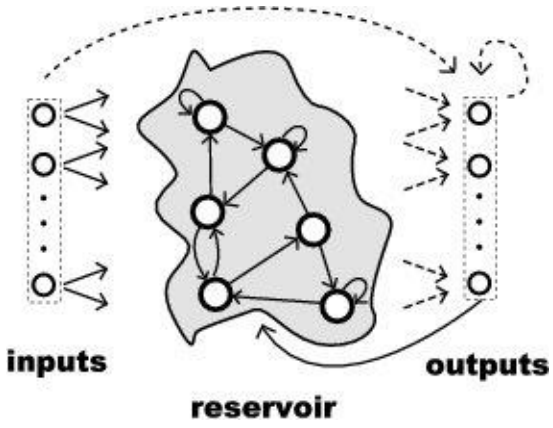
Reservoir computing is een vorm van een neuraal netwerk. Neurale netwerken zijn in principe niet iets nieuws. Tegen 1994 waren er al verschillende systemen met veel succes geïmplementeerd (Widrow, Rumelhart, & Lehr, 1994). Ze werden gebruikt voor verschillende doeleinden: 1) patroononderscheiding, 2) voorspelling en financiële analyses, en 3) besturing en optimalisatie (Widrow, Rumelhart, & Lehr, 1994). Hoewel de netwerken er zeer verschillend uit kunnen zien volgen ze meestal het principe van een *feedforward* netwerk dat ontwikkeld is door Rumelhart. Dit netwerk bestaat uit een laag waar de gebruiker zijn input kan geven. In deze laag zitten neuronen die een bepaalde activatiewaarde hebben gekregen. Als deze neuronen worden geactiveerd verspreiden ze deze activatie naar neuronen die in een tussenlaag zitten. Deze laag is vaak niet zichtbaar en wordt meestal de *hidden layer* genoemd. Deze verborgen neuronen zijn ook weer verbonden met een output laag van neuronen die ook activeren als de activatiewaarde is bereikt. Omdat het een feedforward is gaat de stroom maar 1 richting op (Bechtel & Abrahamsen,

2002). Hoewel deze netwerken zeer krachtig zijn hebben ze een groot nadeel. De verborgen laag moet getraind worden en dat kost enorm veel tijd en moeite (Lukosevicius & Jaeger, 2007; Schrauwen, Verstraeten, & Campenhout, 2007; Lukosevicus, Jaeger, & Schrauwen, 2012). Daarom gaat er de laatste jaren steeds meer aandacht uit naar een relatief nieuwe technologie die *echo state networking, liquid state computing, reservoir computing* of een combinatie van deze termen wordt genoemd.

1.2.2 Reservoir Computing

Reservoir computing is een technologie die pas de laatste 10 jaar echt van de grond is gekomen. Voorheen moest elke neuron in de verborgen laag van het neurale netwerk handmatig verbonden worden met de neuronen in de input en output laag en moest daarbij een gepaste activatiewaarde gevonden worden. Dit is een zeer tijdrovend proces (Ferreira, Ludermir, & Aquino, 2013). Het basisconcept van RC is om de computer random een laag van neuronen te laten genereren met een random activatiewaarde. In plaats van de handmatige training maakt de computer gebruik van lineaire regressie om de gewichten en activatiewaarden bij te stellen (Ferreira, Ludermir, & Aquino, 2013). Dit gebeurt in zogeheten opeenvolgende *cycles* wat een aantal gevolgen heeft voor het netwerk. Door deze *cycles* kan het netwerk *self-sustained temporal activation dynamics* ontwikkelen tussen zijn verschillende neuronen, zelfs als het even geen input heeft. In lekentermen betekent dit dat het netwerk de notie van tijd ontwikkeld, waarmee het wiskundig gezien een dynamisch systeem wordt. *Feedforward networks* daarentegen zijn in principe niets meer dan wiskundige functies (Lukosevicius & Jaeger, 2009). Mocht het netwerk wel input ontvangen dan heeft het eerdere input signalen al als een non-lineaire transformatie opgeslagen in zijn interne staat en hiermee beschikt het over een dynamisch geheugen

(Lukosevicius & Jaeger, 2009). Voor een schematische representatie van een RC-netwerk zie Figuur 1.



Figuur 1 Schematische representatie Reservoir Computing (Reservoir Lab, 2014)

Reservoir Computing wordt in een aantal grote wetenschapsvelden toegepast. Om te beginnen kun je met RC biologische breinen modeleren waardoor deze technologie erg interessant is voor de bio- en neurowetenschappen. Ook wordt RC vaak toegepast als hulpmiddel om technische applicaties mee aan te sturen (Lukosevicius & Jaeger, 2009). Omdat ons onderwerp in beide velden vertegenwoordigd is, is het dus wel duidelijk waarom dit erg mooi past. Hoewel de velden in den beginne erg op elkaar lijken ligt het grote verschil voornamelijk in de manier van output. In het eerste geval gaat het vaak om patroonherkenning en datareductie en ligt de nadruk op het komen tot een conclusie of balans. De hierbij gebruikte technieken leunen sterk op de statistische wetenschap (Lukosevicius & Jaeger, 2009). Het tweede systeem gaat meestal uit van een update mechanisme en gestuurde connecties. Hierbij wordt uitgegaan van non-lineaire filters die een input serie transformeren in een output serie en maakt men vooral gebruik van non-lineaire dynamische systemen.

Door de manier waarop RC zijn data opslaat lijkt dit de ideale techniek om te gebruiken in ons systeem.

1.3 Waar ben ik?

Als een systeem een weg moet vinden door een doolhof en daarbij zijn een vijftal vragen erg belangrijk (Balakrishnan, Bousquet, & Honavar, 1999):

1. *Waar ben ik? (Localization)*
2. *Waar zijn andere plaatsen ten opzichte van mijzelf? (Spatial map)*
3. *Waar is het doel? (Goal determination)*
4. *Hoe kom ik bij het doel vanaf hier? (Path planning)*
5. *Hoe kan ik plaatsen en doelen verwerven? (Spatial learning)*

De antwoorden op deze vragen hangen zijn onderling veel van elkaar afhankelijk. De eerste vraag houdt zich bezig met het probleem dat het systeem zijn huidige plaats moet herkennen. Dit wordt in de robotica vaak aangeduid met de tem *Localization*. Als de plek waar het systeem is uit te drukken is in sensorische eigenschappen die uniek zijn voor die locatie is dit ideaal om verschillende locaties van elkaar te onderscheiden. In de praktijk echter vaak niet realistisch omdat verschillende locaties te veel op elkaar lijken (Balakrishnan, Bousquet, & Honavar, 1999).

De tweede vraag houdt zich bezig met de representatie van de buitenwereld. Dit wordt doorgaans aangeduid met de term *spatial map*. Deze kaart bevat de onderlinge relatie tussen verschillende plaatsen in de buitenwereld en kan zowel topografische als metrische of directionele informatie bevatten. Het is in de meeste gevallen zelfs nodig om meerdere typen te gebruiken om relaties in verschillende niveaus van abstractie aan te kunnen geven (Balakrishnan, Bousquet, & Honavar, 1999).

Als een systeem zijn uiteindelijke doel niet kan identificeren aan de hand van sensorische informatie is een doel gedreven navigatie niet mogelijk. Als je niet weet waar je moet zijn kun je er ook niet komen. Doelen kunnen gerepresenteerd worden op verschillende manieren zoals bijvoorbeeld visuele hints, relatieve afstand tot herkenningspunten (*landmarks)* of een metrische positie in een coördinatenruimte. Dit heet in de robotica *goal determination* (Balakrishnan, Bousquet, & Honavar, 1999).

Om uiteindelijk zijn weg te vinden zal het systeem een route moeten plannen van het begin- naar het eindpunt. De zogeheten *path planning* hangt sterk af van de informatie die opgeslagen ligt in de *spatial map,* de specificatie van het doel en de verschillende manieren van beredeneren die het systeem tot zijn beschikking heeft. Verder kan deze route ook nog eens beïnvloed worden door optimalisaties als kortste afstand, kortste tijd of minste risico. De manier waarop de informatie opgeslagen ligt in de *spatial map* is ook nog eens van invloed. Als bijvoorbeeld de het doel een plek is in een coördinatenruimte en de robot weet zijn huidige locatie is de kortste route simpelweg het vector-verschil tussen deze twee posities (Balakrishnan, Bousquet, & Honavar, 1999).

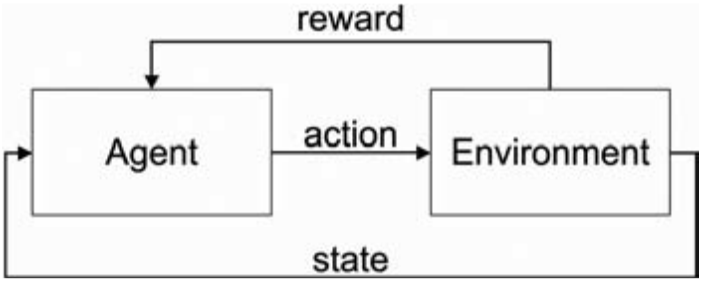
Als de omgeving van het systeem a priori bekend is en bepaald is en zijn de doelen vooraf in geprogrammeerd en statisch dan kan aan de hand van de bovenstaande vier vragen een systeem zijn weg vinden. In de buitenwereld echter is deze informatie hooguit deels bekend. Hierdoor zal het systeem moeten beschikken over een manier om zijn omgeving te leren en zo kan opnemen in zijn *spatial map*. Aan de hand van deze informatie moet het zijn handelen kunnen aanpassen wat *spatial learning* wordt genoemd (Balakrishnan, Bousquet, & Honavar, 1999).

1.4 Leren

Iedereen zal het waarschijnlijk eens zijn dat iets kunnen leren een belangrijk vereiste is voor een vorm van intelligentie. Maar wat is leren nou eenmaal precies? Het boek Cognitive Neuroscience (Purves, et al., 2008) beschrijft leren als volgt. Het begint met een gebeurtenis die door de hersenen dient te worden opgeslagen. Deze gebeurtenis bestaat uit informatie die op verschillende plaatsen opgeslagen dienen te worden. De visuele cortex bijvoorbeeld zou de informatie opslaan over de kleur van de ballonnen van het verjaardagsfeestje, terwijl de auditieve cortex de muziek opslaat en de lay-out van de kamer wordt in pariëtale cortex opgeslagen (Purves, et al., 2008). Al deze gebieden liggen in de neocortex.

Na verloop van tijd vraagt iemand iets over die bewuste verjaardag. Op dat moment sturen je hersenen een zogenaamde *retrieval cue* naar je hippocampus. In de hippocampus ligt een soort index opgeslagen van verschillende gebeurtenissen en welke verschillende eigenschappen daar dan weer bij horen. In ons voorbeeld kunnen we denken aan rode ballonnen, muziek van Justin Bieber en de lay-out was de kamer van je kleine zusje. Bij de eerste keren dat je terugdenkt neemt de hippocampus de hele herinnering voor zijn rekening. Echter wordt er elke keer dat deze informatie samen opgevraagd wordt er ook een verbinding gelegd tussen de verschillende informatie in de neocortex. En iedere keer dat deze herinnering wordt opgevraagd wordt de verbinding tussen de verschillende stukjes informatie van een gebeurtenis steeds sterker. Dit gaat zelfs zo vaak door dat op een gegeven moment de hippocampus helemaal niet meer nodig is en dat de *retrieval cue* direct de desbetreffende gebeurtenis oproept, zonder tussenkomst van de hippocampus. Dit noemt men *consolidation* (Purves, et al., 2008). En op een gegeven moment denk je bij een rode ballon automatisch aan dat feestje bij je zusje waar je de hele avond moest luisteren naar Justin Bieber.

Ons systeem moet zelfstandig kunnen leren hoe het doolhof er uit ziet. Een theorie die in het werkveld van machine leren vaak wordt toegepast is de theorie van *reinforcement learning* (Doya, 2007). Deze theorie is in de praktijk gedefinieerd als een *Markov descision proces* (MDP), zie figuur 2. De actie die de *Agent* onderneemt is van invloed op het *Environment.* Dit levert al naar gelang wel of geen beloning (*reward*) op en kan mogelijk de eigenschappen van het environment veranderen. Het doel van reinforcement learning is om de kans op state verandering of de kans op een beloning, direct of in de toekomst, zo groot mogelijk te maken. Door het toevoegen van een toekomstperspectief is het gebruikt van reinforcement learning erg interessant, maar dit maakt het volledig begrijpen wel een stuk ingewikkelder (Doya, 2007). Reinforcement learning lijkt de ideale leermethode om te gebruiken voor dit onderzoek.



Figuur 2. De schematische weergave van reinforcement learning. (Doya, 2007)

1.5 Vertaling van Real-Time naar Computer

Een computer kan niet direct gesproken taal interpreteren en maakt daarom gebruik van computer code. De code voor dit project is geschreven in de programmeertaal Python. Deze taal heeft voor dit project een aantal voordelen. Python maakt gebruik van hogere orde interpretatie en heeft als voordeel dat het direct en onafhankelijk van het *operating-system* van de gebruiker direct zonder compilatie gelezen kan worden. Dit betekent ook dat het makkelijker is kleine aanpassingen direct te testen. Het leren van Python werkt intuïtiever en simpeler dan C++. Door gebruik te maken van verschillende grote *libraries* is het mogelijk de code klein en overzichtelijk te houden. Het nadeel dat Python heeft is dat het een geïnterpreteerde taal is. Hierdoor heeft het systeem meer

moeite om de code te draaien en daardoor zal de performance van het systeem dalen. Echter zijn de programma's vereist voor dit project niet al te groot en is dit nadeel niet van grote invloed.

In dit project wordt dankbaar gebruik gemaakt van de python library PyBrain (Schaul, et al., 2010). Deze library is ontwikkeld door (ex-) studenten van de prof. Jürgen Schmidhuber aan de Dalle Molle Institute for Artifical Intelligence in Zwitserland en de Technische Universität München in Duitsland. Zonder het pionierswerk van deze mensen had deze scriptie nooit tot stand kunnen komen. In deze library vindt men alle nodige tools om een lerend systeem te maken met Python.

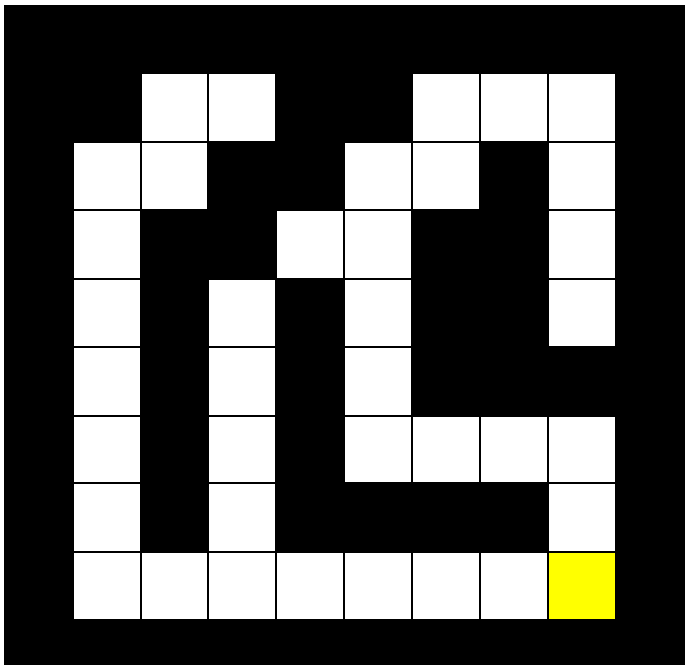
2. Methode

In deze scriptie wordt het leren van spatiale kennis gesimuleerd met behulp van het reinforcement learning model, zie figuur 2. Elk gedeelte van dit model zal dus gedefinieerd moeten worden in het geschreven programma. Hieronder volgt een korte omschrijvingen van de verschillende onderdelen. Voor de complete python-code zie de bijlagen.

2.1 Environment

Het environment is de omgeving waarin het experiment draait. Omdat deze scriptie gaat over het modeleren van spatiele kennis wordt er gebruik gemaakt van een doolhof. Dit doolhof wordt beschreven aan de hand van een tabel, die de vorm heeft van een array. Een '1' stelt een muur voor en een '0' is een veld waarover onze agent zich kan bewegen. In het plaatje zijn ter verduidelijking de muren zwart gemaakt. Verder is vastgesteld dat ons *Goal-Field* ligt op de coördinaten (8,8) wat voorgesteld wordt door het gele blokje. Zie Figuur 3.

```
envmatrix = array      ([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                         [1, 1, 0, 0, 1, 1, 0, 0, 0, 1],
                         [1, 0, 0, 1, 1, 0, 0, 1, 0, 1],
                         [1, 0, 1, 1, 0, 0, 0, 1, 0, 1],
                         [1, 0, 1, 0, 1, 0, 1, 1, 0, 1],
                         [1, 0, 1, 0, 1, 0, 1, 1, 1, 1],
                         [1, 0, 1, 0, 1, 0, 0, 0, 0, 1],
                         [1, 0, 1, 0, 1, 1, 1, 1, 0, 1],
                         [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                         [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
```



Figuur 3 Schematische weergave doolhof met goal field geel aangegeven

2.2 Agent

De agent is de entiteit die de acties moet gaan uitvoeren. In dit experiment moet men

denken aan een persoon of robot die in het doolhof staat en de keuze heeft om in een van vier richtingen te lopen (Noord, West, Zuid, Oost). De agent kan onderscheiden of er een muur of een leeg vakje naast zich bevindt en kan aan de hand daarvan de keuze maken welke kant het op loopt, niet tegen een muur want daar schiet je niks mee op. Het doel van deze agent is om over de lege vakjes te lopen en hiermee het goal field te bereiken. Er wordt random een start locatie gekozen voor de agent om te beginnen. De code om dit te bewerkstelligen is deels ondergebracht in de *environment* en deels in de *LearningAgent()*. De actie die een agent moet uitvoeren wordt doorgaans een *task (*taak) genoemd. Deze taak is meestal het doel van het experiment. Het doel van dit experiment is het bereiken van het goal field waarop het systeem een beloning krijgt. Dit stuk van de code ligt vastgelegd in de *MDPMazeTask*().

2.3 Action, State en Reward

De action die de agent uitvoert is het bewegen over de vrije paden door het doolhof. Hij probeert bij het einde te komen. In het programma is een stuk code opgenomen wat ervoor zorgt dat het programma leert. Lukt het de agent om het doel te bereiken registreert de *LearningAgent*() dit als een succes, waardoor het zijn state aanpast. Deze state wordt in de code gerepresenteerd door een random gegenereerd netwerk, het *ActionValueNetwerk*(). Dit netwerk is opgebouwd volgens de techniek van reservoir computing en gaat dus uit van een input laag, output laag en een verborgen laag waaraan het systeem zelf zijn eigen activatiewaarden heeft gehangen. Het systeem moet de lay-out van het doolhof kunnen opslaan in zijn geheugen. Het doet dit door aan bepaalde

neuronen een activatiewaarde mee te geven, en tussen verschillende neuronen ontstaat een verbinding als het systeem in de gaten heeft dat het van de positie van de ene neuron naar de andere kan bewegen. En de volgende neuron kan ook weer verbonden worden met een select aantal neuronen waar naartoe bewogen kan worden. Als je al deze neuronen aan elkaar schakelt krijg je een weergave van het doolhof, gerepresenteerd zoals dat ook in het menselijk brein gebeurt. Voor een computer maakt de fysieke locatie van deze neuron niets uit, deze onthoudt toch alleen de onderlinge relaties. De activatiewaarden worden aangepast door de *LearningAgent*() en zo leert het systeem zichzelf.

2.4 Learning

Het is de bedoeling dat ons systeem zichzelf dingen kan leren. Om dit te bewerkstelligen is de *LearningAgent*() in het leven geroepen. Deze *LearningAgent*() bestaat uit een drietal onderdelen. Om te beginnen is er de module waarin het gedrag van het programma wordt vastgelegd. Vervolgens heeft het een *learner*, een deel van het programma dat de module kan modificeren aan de hand van wat het geleerd heeft, en een *explorer*, het deel van de programma dat verantwoordelijk is voor het uitvoeren van de verschillende acties. Een voorbeeld van deze drie dingen samen zien we ook terug komen in de natuur waar bijvoorbeeld een mens een set idealen heeft, een geweten om na te denken over en eventueel zijn idealen te veranderen en vervolgens ook beschikt over een lichaam met mogelijkheden om zijn verschillende ideeën en idealen naar buiten toe uit te dragen.

Het *learner* gedeelte van het programma is een algoritme waar de magie vandaan komt. Hier zijn in principe vele verschillende mogelijkheden aan te dragen maar in deze scriptie wordt gewerkt met *reinforcement learning* en een drietal verschillende leeralgoritmes die werken volgens dit principe. De wiskundige functies proberen een optimale ratio te vinden tussen een *reward* pakken en een nieuwe optie uitproberen om een eventueel hogere *reward* te krijgen, de manier

waarop dit gebeurt echter verschilt per algoritme. *Q-learning* wijst aan elke mogelijke actie voor een *state* in zijn netwerk een waarde toe, de acties met een hogere directere *reward* krijgen een hogere waarde, dit noemt men de *actionvalues*. Aan het eind van het programma koppelt het terug wat de beste actie was voor elke state. *Q-Lambda learning* werkt hetzelfde, echter is er een stuk formule toegevoegd waardoor het systeem niet alleen kijkt naar de directe beloning, maar ook naar eventuele vertraagde beloningen door een stap terug in de tijd te kijken. *SARSA learning* kijkt naar *(S)tate (A)ction (R)eward (S)tate (A)ction* en past aan de hand daarvan zijn *actionvalues* aan. Het verschil kan uitgelegd worden door de stellen dat *Q-learning* een vooraf ingestelde verhouding van verkennen en zichzelf belonen volgt, terwijl *SARSA* dit voor zichzelf bepaald terwijl het programma draait. In het programma moet aan de agent duidelijk gemaakt worden welk leeralgoritme het moet gebruiken, hiervoor vult men op de plek van **A** *Q(), QLambda()* of *SARSA()* in.

```
learner = A

agent = LearningAgent(table, learner)
```

Het oorspronkelijke programma is geschreven met behulp van het zogeheten *ActionValueNetwork()*. Dit netwerk stelt de spatiele kennisrepresentatie voor van de ons systeem. Dit programma draait en is in principe een *proof of concept*. Echter omdat dit erg moeilijk is om visueel weer te geven is ervoor gekozen om de neuronen te ordenen in een tabel zodat wij zelf ook kunnen zien wat er daadwerkelijk met de neuronen gebeurt. Verder zijn ze ook nog zo geordend dat ze precies overeenkomen met de locatie op het doolhof, dit allemaal om de analyse makkelijker

te maken. In bijlage A staat de oorspronkelijke code. Op bijlage B vindt u de code gebruikt voor de verschillende simulaties.

2.5 Algemene simulatiemethode

Om vast te kunnen stellen dat ons systeem leert worden er een aantal experimenten uitgevoerd waarbij onze agent (*LearningAgent()*) een taak (*MDPMazeTask()*) uit moet voeren.

```
experiment = Experiment(task, agent)

for i in range(1000):
        experiment.doInteractions (N)
        agent.learn ()
        agent.reset ()
```

Het programma draait iedere keer voor 1000 cycli. In die 1000 cycli wordt het experiment **N** maal uitgevoerd. Nadat het experiment N maal is uitgevoerd gaat de agent leren, dit wil zeggen dat hij zijn *ActionValues* aan gaat passen. Daarna reset de agent zijn uitkomstgeschiedenis om overtollig gebruik van geheugen tegen te gaan.

Om vervolgens te controleren of het systeem ook daadwerkelijk leert is een tweetal methoden bedacht. Ten eerste wordt er een plaatje geschetst van plaatsen in het doolhof waarvan het systeem zeker is dat het de weg naar het goal field kan vinden. Als het contrast tussen een muur en een leeg veld groter is, is het systeem zekerder dat het de weg kan vinden. Dus als het programma langer draait, zullen alle lege velden van het doolhof oplichten.

```
plt.pcolor (table.params.reshape (100, 4).max (axis=1).reshape (10, 10))
plt.gcf().canvas.draw
```

Hoewel dit een methode is waarmee men snel grote verschillen kan vaststellen werkt dit echter minder goed om wetenschappelijk bewijs te leveren aangezien deze methode uitgaat van een subjectieve observatie. Daarom is er een tweede methode bedacht die numerieke waarden moet opleveren. De *ActionValues* worden bij deze methode weggeschreven naar een bestand, gespecificeerd door **Z.** Op deze waarden kan vervolgens een statistische analyse worden uitgevoerd.

```
ofile = open ( 'Z.csv' , "wb" )
writer = csv.writer ( ofile , delimiter = ' ; ' )

for x in range (100):
        writer.writerow (table.getActionValues(x))

ofile.close()
```

3. Simulatie

De simulatie is in totaal 20 maal uitgevoerd per leeralgoritme. In het programma werd **N** vervangen door 1, 2, 5, 10 en 25, en deze simulatie werd steeds 4 maal gedraaid. Vervolgens werden de neuronen geteld die een activatiewaarde hadden van meer dan 50%. Dit gaf voor de verschillende algoritmes de volgende resultaten, weergegeven in Tabel 1 en Figuren 4, 5 en 6:

Tabel 1

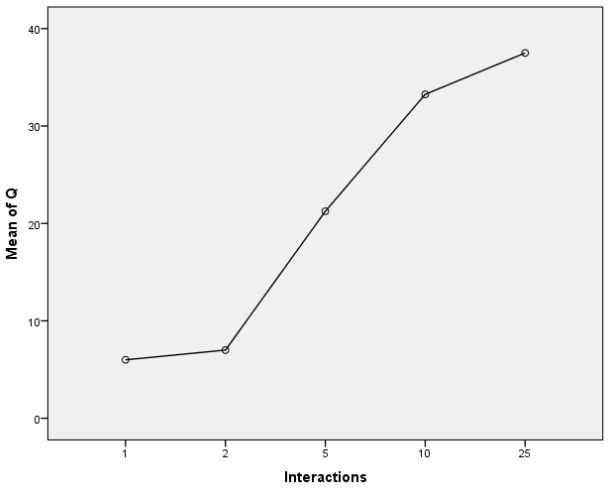
Het aantal velden waarvan de activatiewaarden boven de 50% lag, opgesplitst naar verschillende waarden voor N.

Interacties voor N	Q-learning: Velden met actievatiewaarden boven de 50%	QLambda-learning: Velden met actievatiewaarden boven de 50%	SARSA-learning: Velden met actievatiewaarden boven de 50%
1	4, 6, 7, 7	0, 0, 0, 0	8, 7, 9, 7
2	9, 3, 8, 8	5, 6, 3, 5	12, 7, 14, 11
5	20, 23, 18, 24	17, 20, 19, 21	23, 23, 24, 27
10	35, 32, 34, 32	34, 33, 34, 33	32, 32, 26, 31
25	38, 38, 37, 37	34, 35, 32, 33	30, 21, 28,16

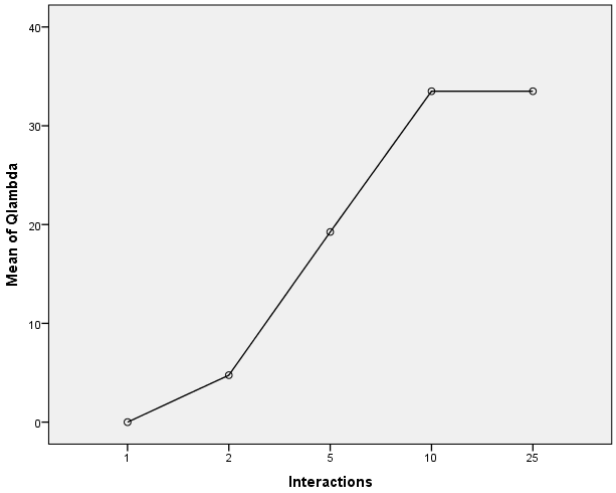
Bij deze gegevens vallen een aantal dingen op. Om te beginnen de vier nullen bij *QLambda*. Dit wordt verklaard door het feit dat dat algoritme ook terugkijkt naar de vorige actie die het heeft uitgevoerd, echter is dat bij één interactie niet mogelijk. Hierdoor heeft het programma nog geen kans gehad zijn *Actionvalues* aan te passen.

Aan de curve van *SARSA-learning* vallen drie dingen op. Om te beginnen begint de curve een stuk hoger dan *Q-learning*. Dit betekent dat *SARSA* prima uit de voeten kan met weinig data. Echter bereikt het hoogste punt van de curve nooit het niveau van dat van *Q-* en *QLambda-learning*, wat betekent dat *SARSA* nooit het hele doolhof zal doorgronden. Verder is de dip aan het einde van de curve ook eigenaardig. Dit zou eventueel verklaard kunnen worden door te stellen dat het *SARSA* algoritme altijd gaat zoeken naar een betere optie dan die het al heeft, en zich daarmee misschien in de vingers snijdt.

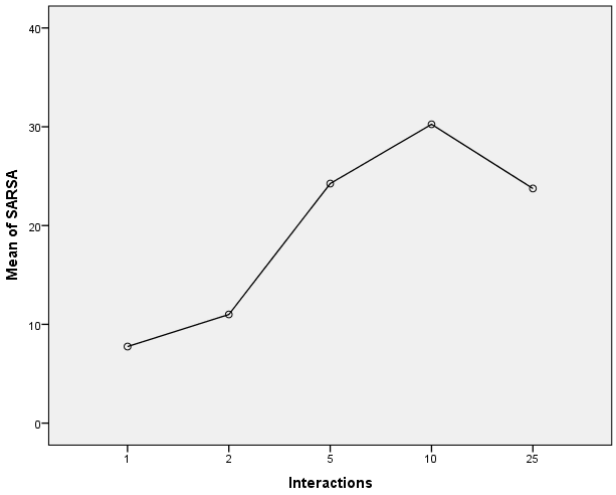
Op deze gegevens is vervolgens een ANOVA variantieanalyse uitgevoerd om het effect te bepalen wat het aantal interacties heeft op het aantal velden dat een grotere activatie laat zien per leeralgoritme. Er werden significante effecten gevonden van de invloed van het aantal interacties op de aantal velden, [$F(4,15) = 216.25$, $p < 0.001$] voor *Q-learning*. [$F(4,15) = 754.94$, $p < 0.001$] voor *QLamda-learning* en [$F(4,15) = 29.08$, $p < 0.001$] voor SARSA-learning. Dit betekent in gewone woorden dat voor elk leeralgoritme geldt dat wanneer het programma meer interacties uitvoert, het



Figuur 4 Q-learning

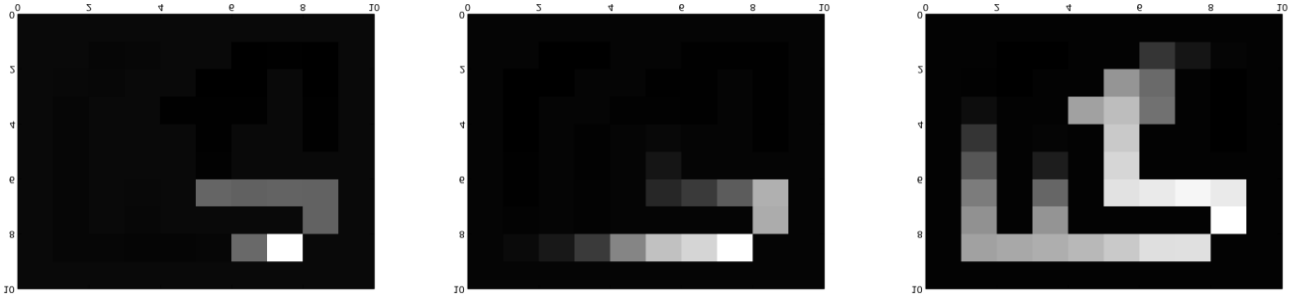


Figuur 5 QLambda-learning



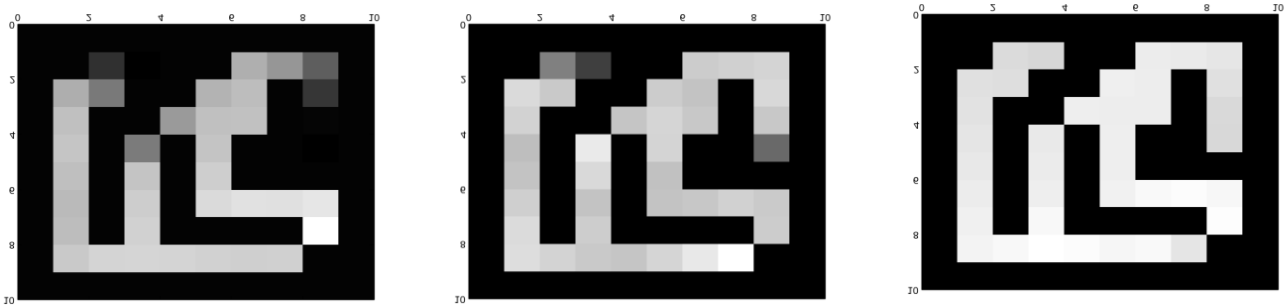
Figuur 6 SARSA-learning

een beter beeld krijgt van het doolhof. Het systeem heeft geleerd. Dit effect wordt ook geïllustreerd in de uitkomstplaatjes van het Q-learning algoritme. Zie Figuur 7 van links naar rechts: 1000x 1, 1000x 2 en 1000x 5 interacties met Q-learning.



Figuur 7 Uitkomstplaatjes Q-learning 1x 2x en 5x

Zie Figuur 8 van links naar rechts: 1000x 10, 1000x 25 en 1000x 100 interacties met Q-learning.



Figuur 8 Uitkomstplaatjes Q-learning 10x 25x en 100x

4. Conclusie en Discussie

Deze scriptie had als onderwerp de spatiale kennisrepresentatie in een kunstmatig systeem. In gewone termen betekent dit dat er onderzoek gedaan is naar hoe een robot kennis op kan slaan over waar het is en waar het heen moet. Hierbij was het erg interessant om te kijken naar hoe dit geregeld is bij de mens en daar kwamen een aantal conclusies uit naar voren. Uit de literatuur bleek dat er verschillende hersenstructuren samen verantwoordelijk waren voor de opslag en recollectie van herinneringen. Hierdoor werd duidelijk dat er gewerkt kon worden met verschillende modules van programma's die samen de totale werking voor hun rekening konden nemen. Ook kon er uit de manier waarop de hippocampus functioneert worden opgemaakt, dat een data architectuur waarbij gebruik gemaakt werd van een *sparse* connectiviteit de voorkeur had. Door dit gegeven is er gekozen voor de interactie tussen de hippocampus en de neocortex zoals herinneren werkt bij de mens wordt het beste gevat door de theorie van *reinforcement learning*. Deze leertheorie is ook nog eens goed te modeleren door het gebruik van verschillende leeralgoritmes, in deze scriptie werd gebruik gemaakt van *Q-Learning*. Dit alles samenvoegend geeft ons consolidatie, de manier waarop mensen zelf ook leren.

Het systeem werd geplaatst in een omgeving die vooraf nog niet bekend was. Belangrijke vragen die beantwoord moesten worden hadden dus betrekking tot de *path planning en spatial learning*. Gaandeweg heeft het zijn weg moeten vinden door het a priori onbekende doolhof. Uit het feit dat na een aantal interacties het doolhof verkend werd en dat na meer interacties meer wegen naar het *goal field* bekend werden betekent dat het systeem de weg naar buiten leerde, terwijl het vooraf geen idee had hoe het doolhof eruit zag. Het systeem heeft dus geleerd wat de juiste wegen waren en door deze wegen aan elkaar te knopen heeft het een route kunnen vinden naar het

goal field waarmee we antwoorden hebben gevonden op de vragen over de *path planning* en *spatial learning*.

In dit onderzoek is gebruik gemaakt van een drietal verschillende leeralgoritmes. Elk bleken ze verschillende voor en nadelen te hebben, wat consequenties heeft voor de keuze van het ideale algoritme. SARSA-learning bleek goed uit de voeten te kunnen met relatief weinig interacties. Ook had het vrij snel een algemeen beeld van het doolhof, echter bleek het algoritme niet in staat het hele doolhof te doorgronden. Deze gegevens maakt het algoritme echter niet onbruikbaar. Mocht er behoefte zijn aan een systeem dat snel leert en niet al te vaak in dezelfde omgeving opereert dan zou het SARSA leeralgoritme best eens de ideale keuze kunnen zijn. QLambda-learning bleek met weinig interacties in het *environment* minder goed in staat om het doolhof te leren. Echter naar mate er meer interacties volgden haalde het QLambda algoritme de beide anderen snel in. QLambda lijkt dus ideaal in een omgeving waar het systeem iets meer tijd krijgt om zich aan te passen en zo steeds een beter beeld krijgt van het *environment.* Echter de koning van de lange termijn bleek toch Q-learning te zijn. Dit algoritme gaf de beste prestaties met een hoog aantal interacties, daarom zal Q-learning het beste tot zijn recht komen als het systeem voor een lange termijn in een weinig veranderende omgeving geplaatst zal worden.

Een enkele kanttekening die bij deze scriptie gemaakt moet worden is het feit dat de auteur graag nog wat meer gestoeid had met de *sparseness* van het reservoir. Echter bleek tijdens de laatste fase dat de voor dit project aangeleerde python programmeerkennis toch enigszins tekortschoot. Bij pogingen om met de sparseness van het reservoir te spelen bleek het programma niet meer te draaien en kon helaas de bug(s) niet worden achterhaald. Bij een eventueel vervolgonderzoek is het zeer sterk aan te raden dat om de programmeerkennis te verbreden alvorens nog een poging te wagen om met een van de meest interessante onderwerpen van deze scriptie verder te gaan. Een

andere mogelijk interessante variatie op dit onderzoek zou zijn om te gaan werken met dynamische omgevingen en kijken hoe een systeem zich daarin zou redden.

Referenties

Balakrishnan, K., Bousquet, O., & Honavar, V. (1999). Spatial Learning and Localization in Rodents: A Computational Model of the Hippocampus and its Implications for Mobile Robots. *Adaptive Behavior*, 173-216.

Bechtel, W., & Abrahamsen, A. (2002). *Connectionism and the mind: Parralel processing, dynamics and evolution in networks.* Oxford: Blackwell Publishing.

Corkin, S. (2002). Whats new with amnesic patiënt H.M.? *Nature Reviews Neuroscience*, 153-159.

Doya, K. (2007). Reinforcement learning: Computational theory and biological mechanisms. *HFSP Journal*, 30-40.

Ferreira, A. A., Ludermir, T. B., & Aquino, R. R. (2013). An approach to reservoir computing design and training. *Expert Systems with Applications*, 4172-4182.

Heil, J. (2012). *Philosophy of Mind: A Contemporary Introduction.* (3rd ed.). Routledge.

Human Brain Project. (2013). Retrieved from The Human Brain Project: https://www.humanbrainproject.eu/nl/home

Kalat, J. W. (2009). *Biological Psychology.* Belmont: Wadsworth, Cengage Learning.

Lukosevicius, M., & Jaeger, H. (2007). *Overview of Reservoir Recipes.* School of Engineering and Science, Jacobs University.

Lukosevicius, M., & Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 127-149.

Lukosevicus, M., Jaeger, H., & Schrauwen, B. (2012). Reservoir Computing Trends. *KI-Künstliche Intelligenz*, 1-7.

Maguire, E. A., Gadian, D. G., Johnsrude, I. S., Good, C. D., Ashburner, J., & Frackowiak, R. S. (2000). Navigation-related structural change in the hippocampi of taxi drivers. *Proceedings of the National Acadamy of Sciences, USA*, 4398-4403.

Norman, K. A., & O'Reilly, R. C. (2003). Modeling Hippocampal and Neocortical Contributions to Recognition Memory: A Complementary-Learning-Systems Approach. *Psychological Review*, 611-646.

O'Reilly, R. C., & Norman, K. A. (2002). Hippocampal and neocortical contributions to memory: advances in the complementary learning systems framework. *TRENDS in Cognitive Sciences*, 505-510.

Purves, D., Brannon, E. M., Cabeza, R., Heuttel, S. A., LaBar, K. S., Platt, M. L., & Woldorff, M. G. (2008). *Principles of Cognitive Neuroscience.* Sunderland: Sinauer Associates.

Reservoir Lab. (2014). *Reservoir Lab*. Retrieved from Reservoir Computing: http://old.reslab.elis.ugent.be/rcbook

Schaul, T., Bayer, J., Wierstra, D., Yi, S., Felder, M., Sehnke, F., . . . Schmidhuber, J. (2010). Retrieved from PyBrain. The Python Machine Learning Library: http://www.pybrain.org

Schrauwen, B., Verstraeten, D., & Campenhout, J. v. (2007). *An overview of reservoir computing: theory, applications and implementations.* Electronics and Information Systems Department, Ghent University, Belgium.

Utwente. (2013, September 17). *MENSELIJKE ROBOT STEEDS DICHTERBIJ*. Retrieved from Universiteit Twente: http://www.utwente.nl/archief/2013/09/menselijke-robot-steeds-dichterbij/

Widrow, B., Rumelhart, D. E., & Lehr, M. A. (1994). Neural Networks: Applications in Industry Business and Science. *Communications of the ACM*, 93-105.

Yonelinas, A. P. (2002). The Nature of Recollection and Familiarity: A Review of 30 Years of Research. *Journal of Memory and Language*, 441-517.

Bijlagen

A. MazeTaskNetwork

```
# Imports
from numpy import *
from matplotlib import pyplot as plt
from pybrain.rl.environments.mazes import Maze, MDPMazeTask
from pybrain.rl.agents import LearningAgent
from pybrain.rl.learners import NFQ, ActionValueNetwork
from pybrain.rl.experiments import Experiment

# Creating our maze
envmatrix = array    ([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                       [1, 1, 0, 0, 1, 1, 0, 0, 0, 1],
                       [1, 0, 0, 1, 1, 0, 0, 1, 0, 1],
                       [1, 0, 1, 1, 0, 0, 0, 1, 0, 1],
                       [1, 0, 1, 0, 1, 0, 1, 1, 0, 1],
                       [1, 0, 1, 0, 1, 0, 1, 1, 1, 1],
                       [1, 0, 1, 0, 1, 0, 0, 0, 0, 1],
                       [1, 0, 1, 0, 1, 1, 1, 1, 0, 1],
                       [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

# Declaring the environment
environment = Maze(envmatrix, (8, 8))

# Declaring the task as a MDPMazeTask
task = MDPMazeTask(environment)

# Declaring that the module for our LearningAgent is an ActionValueNetwork.
module = ActionValueNetwork(1,4)

# Tell the program wich learning algorithm to use
learner = NFQ()

# Declaring our agent that should use the module and a learner
agent = LearningAgent(module, learner)

# Our experiment consists of a task that has to be completed by our agent
experiment = Experiment(task, agent)

# We tell our system to do 1000 cycles. Every cycle it runs it will do 100 interactions after which
# it will learn and reset the previous data, but keeps what it has learned.
for i in range(1000):
        experiment.doInteractions(100)
        agent.learn()
        agent.reset()

# Show ActionValues in console
for x in range(100):
        print module.getActionValues(x)
```

B. MazeTaskTableQ

```
# imports
Import csv
from numpy import *
from matplotlib import pyplot as plt
from pybrain.rl.environments.mazes import Maze, MDPMazeTask
from pybrain.rl.agents import LearningAgent
from pybrain.rl.learners import Q, QLambda, SARSA, ActionValueTable
from pybrain.rl.experiments import Experiment

# Creating our maze
envmatrix = array        ([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                           [1, 1, 0, 0, 1, 1, 0, 0, 0, 1],
                           [1, 0, 0, 1, 1, 0, 0, 1, 0, 1],
                           [1, 0, 1, 1, 0, 0, 0, 1, 0, 1],
                           [1, 0, 1, 0, 1, 0, 1, 1, 0, 1],
                           [1, 0, 1, 0, 1, 0, 1, 1, 1, 1],
                           [1, 0, 1, 0, 1, 0, 0, 0, 0, 1],
                           [1, 0, 1, 0, 1, 1, 1, 1, 0, 1],
                           [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

# Declaring the environment
environment = Maze(envmatrix, (8, 8))

# Declaring the task as a MDPMazeTask
task = MDPMazeTask(environment)

# Declaring our table as an ActionValueTable and initialize them with ones
table = ActionValueTable(100, 4)
table.initialize(1.)

# Tell the program wich learning algorithm to use, instead of Q() you could also use Qlambda() or SARSA()
learner = Q()

# Declaring our agent that should use the table as a module and a learner
agent = LearningAgent(table, learner)

# Our experiment consists of a task that has to be completed by our agent
experiment = Experiment(task, agent)

# Prepare the plotting
plt.ion()
plt.gray()

# We tell our system to do 1000 cycles. Every cycle it runs it will do N interactions after which
# it will learn and reset the previous data, but keeps what it has learned.
for i in range(1000):
        experiment.doInteractions(N)
        agent.learn()
        agent.reset()
```

```
# We plot the results of our program in our prepared figure
      plt.pcolor(table.params.reshape(100,4).max(axis=1).reshape(10, 10))
            plt.gcf().canvas.draw


# We open a file for the program to write its data to
ofile = open ( ‘ Z . csv ’ , “ wb ” )

# Call the writer program, tell it to write to ofile and separate multiple pieces of data with a semicolon
writer = csv.writer ( ofile, delimiter= ‘ ; ’ )

# Tell our program to write down all the activation values of our system
for x in range (100) :
            writer.writerow(table.getActivationValues(x))

# Close the file so another program could access it
ofile.close()
```

C. Pybrain Maze

```
__author__ = 'Tom Schaul, tom@idsia.ch'

from random import random, choice
from scipy import zeros

from pybrain.utilities import Named
from pybrain.rl.environments.environment import Environment

# TODO: mazes can have any number of dimensions?


class Maze(Environment, Named):
    """ 2D mazes, with actions being the direction of movement (N,E,S,W)
    and observations being the presence of walls in those directions.

    It has a finite number of states, a subset of which are potential starting states (default: all except goal states).
    A maze can have absorbing states, which, when reached end the episode (default: there is one, the goal).

    There is a single agent walking around in the maze (Theseus).
    The movement can succeed or not, or be stochastically determined.
    Running against a wall does not get you anywhere.

    Every state can have an an associated reward (default: 1 on goal, 0 elsewhere).
    The observations can be noisy.
    """

    # table of booleans
    mazeTable = None

    # single goal
    goal = None

    # current state
    perseus = None

    # list of possible initial states
    initPos = None

    # directions
    N = (1, 0)
    S = (-1, 0)
    E = (0, 1)
    W = (0, -1)

    allActions = [N, E, S, W]

    # stochasticity
    stochAction = 0.
    stochObs = 0.

    def __init__(self, topology, goal, **args):
        self.setArgs(**args)
        self.mazeTable = topology
        self.goal = goal
        if self.initPos == None:
            self.initPos = self._freePos()
            self.initPos.remove(self.goal)
        self.reset()

    def reset(self):
        """ return to initial position (stochastically): """
        self.bang = False
        self.perseus = choice(self.initPos)

    def _freePos(self):
```

```
    """ produce a list of the free positions. """
    res = []
    for i, row in enumerate(self.mazeTable):
        for j, p in enumerate(row):
            if p == False:
                res.append((i, j))
    return res

def _moveInDir(self, pos, dir):
    """ the new state after the movement in one direction. """
    return (pos[0] + dir[0], pos[1] + dir[1])

def performAction(self, action):
    if self.stochAction > 0:
        if random() < self.stochAction:
            action = choice(range(len(self.allActions)))
    tmp = self._moveInDir(self.perseus, self.allActions[action])
    if self.mazeTable[tmp] == False:
        self.perseus = tmp
        self.bang = False
    else:
        self.bang = True

def getSensors(self):
    obs = zeros(4)
    for i, a in enumerate(Maze.allActions):
        obs[i] = self.mazeTable[self._moveInDir(self.perseus, a)]
    if self.stochObs > 0:
        for i in range(len(obs)):
            if random() < self.stochObs:
                obs[i] = not obs[i]
    return obs

def __str__(self):
    """ Ascii representation of the maze, with the current state """
    s = ''
    for r, row in reversed(list(enumerate(self.mazeTable))):
        for c, p in enumerate(row):
            if (r, c) == self.goal:
                s += '*'
            elif (r, c) == self.perseus:
                s += '@'
            elif p == True:
                s += '#'
            else:
                s += ' '
        s += '\n'
    return s
```

D. Pybrain MDPMazeTask

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de'

from pybrain.rl.environments import Task
from scipy import array

class MDPMazeTask(Task):
    """ This is a MDP task for the MazeEnvironment. The state is fully observable,
        giving the agent the current position of perseus. Reward is given on reaching
        the goal, otherwise no reward. """

    def getReward(self):
        """ compute and return the current reward (i.e. corresponding to the last action performed) """
        if self.env.goal == self.env.perseus:
            self.env.reset()
            reward = 1.
        else:
            reward = 0.
        return reward

    def performAction(self, action):
        """ The action vector is stripped and the only element is cast to integer and given
            to the super class.
        """
        Task.performAction(self, int(action[0]))


    def getObservation(self):
        """ The agent receives its position in the maze, to make this a fully observable
            MDP problem.
        """
        obs = array([self.env.perseus[0] * self.env.mazeTable.shape[0] + self.env.perseus[1]])
        return obs
```

E. Pybrain LearningAgent

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de'

from pybrain.rl.agents.logging import LoggingAgent


class LearningAgent(LoggingAgent):
    """ LearningAgent has a module, a learner, that modifies the module, and an explorer,
        which perturbs the actions. It can have learning enabled or disabled and can be
        used continuously or with episodes.
    """

    def __init__(self, module, learner = None):
        """
        :key module: the acting module
        :key learner: the learner (optional) """

        LoggingAgent.__init__(self, module.indim, module.outdim)

        self.module = module
        self.learner = learner

        # if learner is available, tell it the module and data
        if self.learner is not None:
            self.learner.module = self.module
            self.learner.dataset = self.history

        self.learning = True


    def _getLearning(self):
        """ Return whether the agent currently learns from experience or not. """
        return self.__learning


    def _setLearning(self, flag):
        """ Set whether or not the agent should learn from its experience """
        if self.learner is not None:
            self.__learning = flag
        else:
            self.__learning = False

    learning = property(_getLearning, _setLearning)


    def getAction(self):
        """ Activate the module with the last observation, add the exploration from
            the explorer object and store the result as last action. """
        LoggingAgent.getAction(self)

        self.lastaction = self.module.activate(self.lastobs)

        if self.learning:
            self.lastaction = self.learner.explore(self.lastobs, self.lastaction)

        return self.lastaction


    def newEpisode(self):
        """ Indicate the beginning of a new episode in the training cycle. """
        # reset the module when a new episode starts.
        self.module.reset()

        if self.logging:
            self.history.newSequence()

        # inform learner about the start of a new episode
        if self.learning:
```

```
        self.learner.newEpisode()

    def reset(self):
        """ Clear the history of the agent and resets the module and learner. """
        LoggingAgent.reset(self)
        self.module.reset()
        if self.learning:
            self.learner.reset()


    def learn(self, episodes=1):
        """ Call the learner's learn method, which has access to both module and history. """
        if self.learning:
            self.learner.learnEpisodes(episodes)
```

F. Pybrain NFQ

```
from scipy import r_

from pybrain.rl.learners.valuebased.valuebased import ValueBasedLearner
from pybrain.datasets import SupervisedDataSet
from pybrain.supervised.trainers.rprop import RPropMinusTrainer
from pybrain.supervised.trainers import BackpropTrainer
from pybrain.utilities import one_to_n


class NFQ(ValueBasedLearner):
    """ Neuro-fitted Q-learning"""

    def __init__(self, maxEpochs=20):
        ValueBasedLearner.__init__(self)
        self.gamma = 0.9
        self.maxEpochs = maxEpochs

    def learn(self):
        # convert reinforcement dataset to NFQ supervised dataset
        supervised = SupervisedDataSet(self.module.network.indim, 1)

        for seq in self.dataset:
            lastexperience = None
            for state, action, reward in seq:
                if not lastexperience:
                    # delay each experience in sequence by one
                    lastexperience = (state, action, reward)
                    continue

                # use experience from last timestep to do Q update
                (state_, action_, reward_) = lastexperience

                Q = self.module.getValue(state_, action_[0])

                inp = r_[state_, one_to_n(action_[0], self.module.numActions)]
                tgt = Q + 0.5*(reward_ + self.gamma * max(self.module.getActionValues(state)) - Q)
                supervised.addSample(inp, tgt)

                # update last experience with current one
                lastexperience = (state, action, reward)

        # train module with backprop/rprop on dataset
        trainer = RPropMinusTrainer(self.module.network, dataset=supervised, batchlearning=True, verbose=False)
        trainer.trainUntilConvergence(maxEpochs=self.maxEpochs)

        # alternative: backprop, was not as stable as rprop
        # trainer = BackpropTrainer(self.module.network, dataset=supervised, learningrate=0.005, batchlearning=True, verbose=True)
        # trainer.trainUntilConvergence(maxEpochs=self.maxEpochs)
```

G. Pybrain SARSA

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de'

from pybrain.rl.learners.valuebased.valuebased import ValueBasedLearner


class SARSA(ValueBasedLearner):
    """ State-Action-Reward-State-Action (SARSA) algorithm.

    In batchMode, the algorithm goes through all the samples in the
    history and performs an update on each of them. if batchMode is
    False, only the last data sample is considered. The user himself
    has to make sure to keep the dataset consistent with the agent's
    history."""

    offPolicy = False
    batchMode = True

    def __init__(self, alpha=0.5, gamma=0.99):
        ValueBasedLearner.__init__(self)

        self.alpha = alpha
        self.gamma = gamma

        self.laststate = None
        self.lastaction = None

    def learn(self):
        if self.batchMode:
            samples = self.dataset
        else:
            samples = [[self.dataset.getSample()]]

        for seq in samples:
            for state, action, reward in seq:

                state = int(state)
                action = int(action)

                # first learning call has no last state: skip
                if self.laststate == None:
                    self.lastaction = action
                    self.laststate = state
                    self.lastreward = reward
                    continue

                qvalue = self.module.getValue(self.laststate, self.lastaction)
                qnext = self.module.getValue(state, action)
                self.module.updateValue(self.laststate, self.lastaction, qvalue + self.alpha * (self.lastreward + self.gamma * qnext - qvalue))

                # move state to oldstate
                self.laststate = state
                self.lastaction = action
                self.lastreward = reward
```

H. Pybrain QLambda

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de'

from pybrain.rl.learners.valuebased.valuebased import ValueBasedLearner


class QLambda(ValueBasedLearner):
    """ Q-lambda is a variation of Q-learning that uses an eligibility trace. """

    offPolicy = True
    batchMode = False

    def __init__(self, alpha=0.5, gamma=0.99, qlambda=0.9):
        ValueBasedLearner.__init__(self)

        self.alpha = alpha
        self.gamma = gamma
        self.qlambda = qlambda

        self.laststate = None
        self.lastaction = None


    def learn(self):
        states = self.dataset['state']
        actions = self.dataset['action']
        rewards = self.dataset['reward']

        for i in range(states.shape[0] - 1, 0, -1):
            lbda = self.qlambda ** (states.shape[0] - 1 - i)
            # if eligibility trace gets too long, break
            if lbda < 0.0001:
                break

            state = int(states[i])
            laststate = int(states[i - 1])
            # action = int(actions[i])
            lastaction = int(actions[i - 1])
            lastreward = int(rewards[i - 1])

            qvalue = self.module.getValue(laststate, lastaction)
            maxnext = self.module.getValue(state, self.module.getMaxAction(state))
            self.module.updateValue(laststate, lastaction, qvalue + self.alpha * lbda * (lastreward + self.gamma * maxnext - qvalue))
```

I. Pybrain Q

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de'

from pybrain.rl.learners.valuebased.valuebased import ValueBasedLearner


class Q(ValueBasedLearner):

    offPolicy = True
    batchMode = True

    def __init__(self, alpha=0.5, gamma=0.99):
        ValueBasedLearner.__init__(self)

        self.alpha = alpha
        self.gamma = gamma

        self.laststate = None
        self.lastaction = None

    def learn(self):
        """ Learn on the current dataset, either for many timesteps and
            even episodes (batchMode = True) or for a single timestep
            (batchMode = False). Batch mode is possible, because Q-Learning
            is an off-policy method.

            In batchMode, the algorithm goes through all the samples in the
            history and performs an update on each of them. if batchMode is
            False, only the last data sample is considered. The user himself
            has to make sure to keep the dataset consistent with the agent's
            history.
        """
        if self.batchMode:
            samples = self.dataset
        else:
            samples = [[self.dataset.getSample()]]

        for seq in samples:
            for state, action, reward in seq:

                state = int(state)
                action = int(action)

                # first learning call has no last state: skip
                if self.laststate == None:
                    self.lastaction = action
                    self.laststate = state
                    self.lastreward = reward
                    continue

                qvalue = self.module.getValue(self.laststate, self.lastaction)
                maxnext = self.module.getValue(state, self.module.getMaxAction(state))
                self.module.updateValue(self.laststate, self.lastaction, qvalue + self.alpha * (self.lastreward + self.gamma * maxnext - qvalue))

                # move state to oldstate
                self.laststate = state
                self.lastaction = action
                self.lastreward = reward
```

J. Pybrain ActionValueNetwork & Pybrain ActionValueTable

```
__author__ = 'Thomas Rueckstiess, ruecksti@in.tum.de'

from pybrain.utilities import abstractMethod
from pybrain.structure.modules import Table, Module, TanhLayer, LinearLayer, BiasUnit
from pybrain.structure.connections import FullConnection
from pybrain.structure.networks import FeedForwardNetwork
from pybrain.structure.parametercontainer import ParameterContainer
from pybrain.tools.shortcuts import buildNetwork
from pybrain.utilities import one_to_n

from scipy import argmax, array, r_, asarray, where
from random import choice


class ActionValueInterface(object):
    """ Interface for different ActionValue modules, like the
        ActionValueTable or the ActionValueNetwork.
    """

    numActions = None

    def getMaxAction(self, state):
        abstractMethod()

    def getActionValues(self, state):
        abstractMethod()


class ActionValueTable(Table, ActionValueInterface):
    """ A special table that is used for Value Estimation methods
        in Reinforcement Learning. This table is used for value-based
        TD algorithms like Q or SARSA.
    """

    def __init__(self, numStates, numActions, name=None):
        Module.__init__(self, 1, 1, name)
        ParameterContainer.__init__(self, numStates * numActions)
        self.numRows = numStates
        self.numColumns = numActions

    @property
    def numActions(self):
        return self.numColumns

    def _forwardImplementation(self, inbuf, outbuf):
        """ Take a vector of length 1 (the state coordinate) and return
            the action with the maximum value over all actions for this state.
        """
        outbuf[0] = self.getMaxAction(inbuf[0])

    def getMaxAction(self, state):
        """ Return the action with the maximal value for the given state. """
        values = self.params.reshape(self.numRows, self.numColumns)[state, :].flatten()
        action = where(values == max(values))[0]
        action = choice(action)
        return action

    def getActionValues(self, state):
        return self.params.reshape(self.numRows, self.numColumns)[state, :].flatten()

    def initialize(self, value=0.0):
        """ Initialize the whole table with the given value. """
        self._params[:] = value


class ActionValueNetwork(Module, ActionValueInterface):
    """ A network that approximates action values for continuous state /
```

```
    discrete action RL environments. To receive the maximum action
    for a given state, a forward pass is executed for all discrete
    actions, and the maximal action is returned. This network is used
    for the NFQ algorithm. """

def __init__(self, dimState, numActions, name=None):
    Module.__init__(self, dimState, 1, name)
    self.network = buildNetwork(dimState + numActions, dimState + numActions, 1)
    self.numActions = numActions

def _forwardImplementation(self, inbuf, outbuf):
    """ takes the state vector and return the discrete action with
        the maximum value over all actions for this state.
    """
    outbuf[0] = self.getMaxAction(asarray(inbuf))

def getMaxAction(self, state):
    """ Return the action with the maximal value for the given state. """
    return argmax(self.getActionValues(state))

def getActionValues(self, state):
    """ Run forward activation for each of the actions and returns all values. """
    values = array([self.network.activate(r_[state, one_to_n(i, self.numActions)]) for i in range(self.numActions)])
    return values

def getValue(self, state, action):
    return self.network.activate(r_[state, one_to_n(action, self.numActions)])
```

K. Pybrain Experiment

```
__author__ = 'Tom Schaul, tom@idsia.ch'


class Experiment(object):
    """ An experiment matches up a task with an agent and handles their interactions.
    """

    def __init__(self, task, agent):
        self.task = task
        self.agent = agent
        self.stepid = 0

    def doInteractions(self, number = 1):
        """ The default implementation directly maps the methods of the agent and the task.
            Returns the number of interactions done.
        """
        for _ in range(number):
            self._oneInteraction()
        return self.stepid

    def _oneInteraction(self):
        """ Give the observation to the agent, takes its resulting action and returns
            it to the task. Then gives the reward to the agent again and returns it.
        """
        self.stepid += 1
        self.agent.integrateObservation(self.task.getObservation())
        self.task.performAction(self.agent.getAction())
        reward = self.task.getReward()
        self.agent.giveReward(reward)
        return reward
```