Danmarks Tekniske Universitet

Kortlægning af Bevægelsesmønstre
Bacheloropgave
09.05.2012

Mads Eiler Hansen, s093297

Forord

Dette er et bachelorprojekt udarbejdet af Mads Eiler Hansen (s093297) i perioden 30.1.2012-9.5.2012, under vejledning af Carsten Witt.
Projektet i softwareteknologi og udarbejdet ved Institut for Matematisk Modellering, på Danmarks Tekniske Universitet.
Projektet har omfang svarende til 15 ETCS point.

Mads Eiler Hansen, s0932972
09.05.2012

Abstract

This project tests 3 predicative machine learning models: artificial neural networks, decision trees and logistic regression. The tests are done to compare the models and to asses if they have any applicability in an agents subroutine, to predict moving patterns of other agents, in a game such as Pac-Man.
A simple version of Pac-Man is implemented, with specific focus on making it a test platform for the three models.
Machine learning and specifically the problems that deals with prediction are assessed, with a focus on neural networks as a, biologically founded, mathematical model.
Different complexities of moving patterns are explored. The three models are tested with respect to how well they perform, at predicting different complexities of moving patterns.
The project deals with whether the models can be used to form such a subroutine, and when which model is more applicable.
Conclusively the two simple models, decision trees and logistic regression, are found not to have any meaningful use. The neural network is only really usable in the case of very simple moving patterns. It is finally suggested that one uses even more complex models if one wants to implement the desired subroutine.

Resumé

I dette projekt testes 3 prædikative maskinlæringsmodeller: neurale netværk, valgtræer og logsitsk reggression. Tests har til formål at sammenligne og vurdere om modellerne kan fungere i en agentdel, der skal forudse bevægelsesmønstre for andre agenter, i spil som Pac-Man.
Der implementeres en forsimplet udgave af Pac-Man, med designfokus på at dette skal være en testplatform for de 3 modeller.
Der gøres rede for prædikationsproblemer og maskinlæring med fokus på det neurale netværk som matematisk model, der har afsæt i en biologisk forståelse af neuroner.
Forskellige sværhedsgrader af bevægelsesmønstre undersøges. Og det testes hvordan de 3 modeller formår at forudse bevægelsesmønstrene for 4 forskellige af sådanne sværhedsgrader.
Projektet søger således svar på om modellerne kan anvedes til formålet, samt hvornår hvilken model er mest meningsfuld at anvende.
Det konkluderes at valgtræer og logistisk regression ikke har nogen anvendelse, og at neurale netværk kan anvendes til simple bevægelsesmønstre. Det foreslås at man anvender mere komplekse modeller, hvis man ønsker at lave en kortlæggende agentdel.

Indhold

1 Indledning **6**
1.1 Motivation for projektet . . . 6
1.2 Problemformulering . . . 7
1.3 Læsevejledning . . . 8

2 A - Introduktion til Neurale Netværk og Pac-Man **10**
2.1 Intro til maskinlæring og prædiktionsproblemer . . . 10
2.2 Neurale Netværk . . . 11
2.2.1 Den biologiske baggrund . . . 12
2.2.2 Neurale netværk som matematisk model . . . 13
2.2.3 Backpropagation . . . 15
2.3 Pac-Man . . . 16
2.3.1 Hvad er Pac-Man . . . 17
2.3.2 Afgrænsning af Pac-Man platformen . . . 18

3 B - Programbeskrivelse og Designvalg **20**
3.1 Programmets centrale elementer . . . 20
3.1.1 Spillets opbygning . . . 22
3.1.2 AIens opgave . . . 22
3.1.3 Det neurale netværk . . . 23
3.2 Spillet . . . 23
3.2.1 Opbygning af spillet . . . 25
3.2.2 Turen . . . 26
3.2.3 Dataopsamling . . . 26
3.3 Det neurale netværks design . . . 27
3.3.1 Design af klasser og kontrol . . . 28
3.3.2 Dataspecifikt design . . . 28
3.3.3 Læring i praksis . . . 30
3.4 Design af bevægelsesmønstre . . . 31
3.4.1 Spøgelsernes AI . . . 31
3.4.2 Design af PMs AI . . . 32

4 C - Behandling af Prædikative Modeller **34**
4.1 Introduktion til statistiske undersøgelser . . . 34
4.1.1 Test og krydsvalidering . . . 34
4.1.2 Forventninger og sammenhænge i data . . . 36
4.2 Alternative løsninger til prædiktionsproblemet . . . 39

4.2.1 Valgtræer 39
4.2.2 Logistisk regression 40
4.2.3 Resultater 41
4.3 Træning af det neurale netværk 42
4.3.1 Praksis for opsætning af et neuralt netværk 43
4.3.2 Skalering af NN til Pac-Man 44
4.4 Sammenligning af resultater fra de 3 modeller 47

5 Konklusion **50**

6 Perspektivering **51**

7 Bilag **53**

A Brættet fra fil **53**

1 Indledning

Dette projekt undersøger, hvordan man kan anvende et neuralt netværk til at forudsige bevægelsesmønstre for simple logikbaserede kunstige intelligenser. Den platform, der bruges til dette, er det populære arkade spil fra Namco 1980, Pac-Man. Her bevæger den spillerstyrede Pac-Man sig rundt blandt fire spøgelser, som har faste, men for spilleren ukendte bevægelsesmønstre. Denne opgave handler om at lave en del af en agent, der kan tage over for den spillerstyrede Pac-Man, og som har til formål at forudsige, hvad spøgelserne har i sinde at gøre.

1.1 Motivation for projektet

Nærværende afsnit er baseret på [Russel & Norvig, 2010], specielt indledning (1.1-2.5) og filosofisk baggrund (26.1-26.4).

Det at skabe en autonom agent med menneskeefterlignet intelligens er noget, der har interesseret mange som har arbejdet med eller læst om kunstig intelligens. Tænk, hvis man havde en agent man blot kunne bede om at løse et problem, hvorefter den selv forstod hvad man mente og fandt ud af hvordan problemet kunne løses. Vi møder mange agenter, der fungerer sådan i dagligdagen, men som kræver at brugerne anvender dem på en mere eller mindre intelligent måde. Google og tale-genkendende smart phones er gode eksempler på sådanne hverdagsteknologier. Teknologier der er ekstremt specialiserede til at løse meget svære, men relativt afgrænsede problemstillinger. Og som vi brugere bliver nødt til at manipulere, for at få dem til at forstå, hvad vi ønsker af dem.
Når man har løst sådanne problemstillinger har det altid været nærliggende at spørge: hvordan gør hjernen? Ud fra disse spørgsmål er der opstået flere hjerne-efterlignende modeller. Disse er interessante, fordi de er blevet meget udbredte og yderst effektive. Og fordi de giver et indblik i hvordan vi måske selv udfører handlinger såsom tankeprocesser og problemløsning, ved at give os indblik hvilke problemer specifikke processer løser.

Dette er den ene form for metakognitiveproblemstilling der har interesseret mig, i forbindelse med formulering af problemstillingen. En anden er ikke bare at adressere sine egne tankeprocesser, men også andres. Det kræver at man først indser, at andre kan have tankeprocesser overhovedet. Dette ville selvfølgelig kunne fortsætte i det uendelige, hvis flere agenter har lige mulig-

heder for at tænke over hinanden.
Det bliver undersøgt meget inden for AI og robotteknologi, hvordan man kan lave denne form for multiagentsystemer. I dette projekt lader jeg én agent have mulighed for at gennemskue, hvad de andre agenter tænker. Og for at muliggøre problemstilling er tankeprocesserne for de andre agenter rationelle og meget simple.

1.2 Problemformulering

Målet for dette projekt er at lave og afprøve en del af en større agent, kaldet PM, der kan spille Pac-Man. Denne specifikke del prøver at kortlægge bevægelsesmønstre for de andre agenter i spillet. Denne kortlægning kan PM bruge til at forudsige, hvad hvor andre agenter befinder sig i fremtiden og derpå optimere de mål PM måtte have. Jeg har valgt at bruge neurale netværk til at lave en kortlægning, i form af en forudsigende model.

Jeg vil redegøre for, hvad et neuralt netværk er og implementere et neuralt netværk.
Jeg vil udforme en simplificeret spilmodel for Pac-Man og et neuralt netværk. Jeg vil gøre rede for de designvalg jeg har taget i forbindelse med denne udformning. Implementering foregår i java.
Jeg vil undersøge *om det kan lade sig gøre* at anvende NN til at lave denne kortlægning. Det bygger på en undersøgelse af om NN har en misklassificeringsrate, der er lav nok, til at anvende i en planlægningsproces. Jeg vil sammenligne NN med to andre mulige, men simplere, modeller til opgaven; valgtræer og logistisk regression.
Desuden vil jeg se på kompleksiteten af de bevægelsesmønstre, der anvendes i Pac-Man. Jeg vil undersøge *hvornår man kan* bruge modellerne til at kortlægge et mønster. Her ses specielt på om det bliver sværere for en model at kortlægge bevægelsesmønsteret, hvis man øger kompleksiteten af mønsteret. I udgangspunktet har jeg en hypotese om at de mindre modeller har mulighed for at forudsige simple bevægelsesmønstre, men at der ved mere komplekse bevægelsesmønstre skal anvendes større modeller så som NN.
Endeligt vil jeg vurdere *om det kan betale sig* at bruge NN, i forhold til de simplere modeller, som er mindre og anvender kortere læringstid.

1.3 Læsevejledning

Opgaven er opdelt i 3 dele, A, B og C.
I første del forklares de elementer som projektet bygger på, så som maskinlæring, neurale netværk og Pac-Man spillet. Har man ingen erfaring med Pac-Man foreslås det, at man læser afsnit 2.3 først - eller googler Pac-Man og tager et spil[1].
I anden del beskrives de java implementeringer, som kan findes i ZIP-filen tilknyttet denne opgave. Her ses der på de design-problemer og -løsninger der er fundet.
I tredje og sidste del analyseres de resultater, der er fundet igennem implementeringen i forhold til om neurale netværk kan anvendes til at løse denne problemstilling.

Til dette projekt er der knyttet en ZIP-fil. Denne indeholder 2 mapper mærket "java code"og "matlab files"med henholdsvis java og matlab kode, som der refereres til løbende i denne opgave.
ZIP-filen indeholder også en text fil mærket "READ ME". Denne indeholder instruktioner til hvordan man kan foretage tests i java koden.

Ordforklaring

Der vil gennemgående i denne opgave være visse forkortelser, disse er beskrevet i dette afsnit. Navne anført i hårde parenteser er henvisninger til bibliografien 6.

NN Neurale Netværk - Her menes kunstige neurale netværk, hvis andet ikke er angivet.

AI Artificial Intelligens - eng. kunstig intelligens. I denne opgaver betegner AI specielt de regler der definerer spøgelsernes bevægelsesmønstre. I java programmet kan disse findes i klassen *ai*.

PM Pac-Man, her refereres til *agenten* Pac-Man ikke til Namcos spil.

TTC TurnTimeControl (eng. Tur-Tid-Kontrol), dette er en af klasserne i java programmet der er lavet i forbindelse med dette projekt.

[1] For den meget interesserede kan jeg henvise til [J. Pittman, 2011] som er en historisk og teknisk gennemgang af hvordan den oprindelige Pac-Man platform virker.

Node Det engelske node anvendes i flæng om punkter i grafer, eller om implementeringen af interfacet *Node*.

2 A - Introduktion til Neurale Netværk og Pac-Man

I denne del af rapporten gør jeg først rede for prædiktionsproblemer, og hvordan maskinlæring fungere i forhold til disse problemer.
Dernæst beskriver jeg neurale netværk, som en matematisk prædiktativmodel. Med udgangspunkt i en bestemt biologisk forståelse af hvad en neuron er.
Endeligt vil Namcos Pac-Man blive beskrevet og jeg vil gøre rede for den afgrænsning jeg har lavet, i forbindelse med designet af mit program.

2.1 Intro til maskinlæring og prædiktionsproblemer

Nærværende afsnit er baseret på [Russel & Norvig, 2010], afsnit 18.1 og 18.2.

Der er mange problemer der kræver at man anvender en eller anden form for planlægning, for at kunne finde en ordenlig løsning. Et godt grundlag for en god planlægning er at kunne forudsige problemer eller scenarier, man vil støde på. I en modellering opstiller vi typisk en verden meget firkantet og siger, at en eller anden mængde af hændelser eller omstændigheder førte til at noget bestemt skete. Vi beskriver dette ved at konkretisere omstændighederne og sige, at en funktion af disse omstændigheder leder til, at en bestemt hændelse sker.
En egentlig planlægning ville nok i høj grad bruge elementer fra traditionel AI. Men i maskinlæring fokuserer man på hvordan man kan bruge statistisk til at modellere verdenen.
Lad os for eksempel antage vi prøver at forudsige længden af græsset på en plæne i morgen kl. 13. Det er klart at dette er en funktion af næsten uendeligt mange parametre, så som gødning, om det er blevet slået, om solen har skinnet meget eller lidt på det sidste og ikke mindst længden af græsset lige nu. Hvis vi havde alle oplysninger om vejr, vind, sol, gartner osv. ville vi måske kunne opstille nogle fuldstændig præcise regler for længden af græs. Det der ligger til grund for græslængden, kaldes den *ægte funktion*. I praksis ville det være en uoverskuelig og sikkert også umulig opgave at finde den, da verdenen ikke altid kan beskrives så formaliseret. Det man typisk gør er at koge omstændighederne ned til nogle helt basale komponenter lad os sige: græslængden lige nu samt tiden til klokken slog 13. Dette indfører unægteligt

en del usikkerhed i den funktion, man prøver at skabe, og gør at vi ikke vil finde den *ægte funktion*, men vi vil prøve at komme så tæt på som muligt. Maskinlæring anvedes typiskt til problemstillinger som klassificering, regression, klyngeanalyse (fra eng. clustering) og anomalie-detektion. Typisk for de første to er at de er lavet med *supervised learning*, mens klyngeanalyse typisk er en statistisk analyse af data og dermed *unsupervised*. I dette projekt bruger jeg *supervised learning* til at prædikterer over et klassificeringsproblem. For at løse dette prædikationsproblem anvender man typisk klassificerende eller regressive maskinlæringsmodeller. Ens for disse er at de, på baggrund af store mængder af observationer, anvender statistik til at udarbejde en efterligning af den *ægte funktion*.

Når en prædikativ model laves, træner man den på datapunkter. I tilfælde hvor en agent bevæger sig rundt i verdenen og ser værdierne af alle de parametre, den har brug for, vil den så også se resultaterne af de bestemte parametre umiddelbart efter. Det er altså muligt at lave, det man kalder for *supervised learning*. Som betyder at man har en mængde af datapunkter, hvor man allerede kender resultatet af den *ægte funktion* og så bruger dem til at undervise en model i at efterligne den *ægte funktion*.
Den måde man sammenligner modeller på er ved at opdele den data man har i test og træningssæt, modellen får så lov til at træne på træningssættet, men får ikke lov til at se løsningerne af testsættet. Man lader så modellen udføre forudsigelser for alle punkter i testsættet og sammenligne mængden af fejl de forskellige modeller laver. Denne metode kaldes for kryds-validering (fra eng. *cross validation*). Hvordan jeg anvender kryds-validering forklares mere gennemgående i 4.1.

2.2 Neurale Netværk

I dette afsnit forklares den biologiske baggrund for det neurale netværk. Desuden indeholder afsnittet en beskrivelse af et neuralt netværk som en matematisk model, med udgangspunkt i den biologiske forståelse. Der beskrives dog ikke de praktiske ændringer, der laves i forbindelse med en implementering. Her henvises i stedet til afsnit 3.3.

2.2.1 Den biologiske baggrund

Det nærværende afsnit er baseret på introduktionen (1.2.4) af [Russel & Norvig, 2010] og introduktionen (1.1-1.2) af [R. Rojas, 1996].

Et neuralt netværk er egentligt navnet på en matematisk model, man kan bruge til at modellere forskellige funktioner. Modellen prøver at fungerer som en samling af neuroner, eller efterligninger af neuroner. Det er derfor interessant først at se på, hvad en neuron er og hvordan den fungerer. At beskrive processer i neuroner er noget der, inden for medicin og biologi, kan beskrives langt mere dybdegående, end nødvendigt her. Der fokuseres derfor på den simplificering, som ligger til grund for den matematiske model.
Neuroner er de celler, som vores hjerne og nervesystem hovedsagligt består af. Det specielle ved disse celler er, at de har aksoner som er en aflang celledel der kan sende elektriske signaler. Det at en celle sender et signal kaldes, at den 'fyrer'. Når cellen fyrer, vil der blive udsendt transmitterstoffer for enden af aksonet. Det er koncentrationen af forskellige transmitter stoffer omkring cellekernen af en neuron, der afgør om denne fyrer eller ej[2]. En celle der fyrer øger altså chancen for, at celler i nærheden af dens aksons udgange også fyrer.
Det har længe været den generelle opfattelse, at hjernen er stedet, hvor vores tanker og bevidsthed opstår. Disse konstellationer af neuroner formår altså at bygge modeller af verden, som indeholder en enorm præcision og kompleksitet. Hvis vi for eksempel skal krydse en vej, skal vi på baggrund af den verden vi ser, bestemme om det er sikkert at gå over, eller om vi bliver kørt ned. Hvis vi går endnu længere ind i hvad der sker, er øjnene en samling af celler, som på baggrund af forskellige lysinputs, giver et input til hjernen. Denne skal så på baggrund af disse inputs forme scenariet, genkende at der er en bil, tage et valg og enten fortælle benene om de skal gå, eller ej. Dette formår hjernen at gøre, mens vi går og tænker på, om der stadig var mælk i køleskabet, eller om der skal købes noget mere ind.

Det er en kompleksitet, der har forundret og interesseret mange. Og det har derfor været nærliggende at lave intelligente eller lærende modeller, der prøver at efterligne hjernen. De første neurale netværk blev lavet i 1943 af

[2]Dette er en klar oversimplificering, da forskellige transmitter hver i sær kan aktiverer forskellige receptorer som afføder bestemte fyringssekvenser. Men denne kompleksitet er dårligt kortlagt og er irrelevant i forhold til den matematiske model beskrevet i denne opgave.

Warren McCulloch og Walther Pitts. Siden hen har man lavet mange forskellige former for neurale netværk, og neurale netværk er nu en ofte anvendt til at løse prædiktions problemer.

2.2.2 Neurale netværk som matematisk model

Det nærværende og næste afsnit er baseret på beskrivelser af netværk fra [R. Rojas, 1996], og på *bacpropagation* algoritmen som beskrevet i [C. M. Bishop, 2006], kapitel 5.

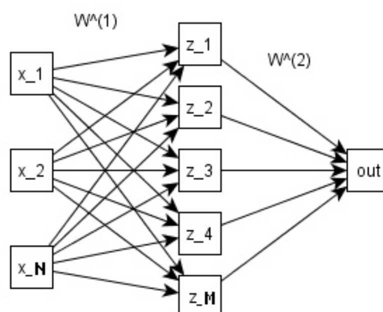
I dette afsnit belyses den matematiske model med udgangspunkt i biologiske forståelse. Den grundlæggende idé bag det matematiske neurale netværk er at sammensætte flere regressive enheder til en større. Hver af disse enheder kaldes for en *perceptron*. Perceptronen svarer til cellekroppen i det biologiske system. Den modtager en række af inputs, som den summerer og tager en aktiveringsfunktion på summen. Denne aktivering svarer til hvilken grad cellen fyrer med. Det er derfor typisk en stepfunktion fra -1 til 1, eller fra 0 til 1. Det at videregive en aktiveringsværdi kaldes for at *feede* (eng. fodre).
Et neuralt netværk har derfor struktur som en graf, hvor perceptronerne er noder. Kanterne videregiver information mellem perceptronerne. På den måde kan man forstå en perceptron og de kanter, der går ud af den som henholdsvis cellekroppen og aksonet på en neuron.
Den konstellation af celler, der er opbygget repræsenteres ved vægtningen af værdier, der bevæger sig gennem kanterne til perceptroner. Der ses typisk på netværk, som er ensrettede, hvor der først er et lag af inputenheder, som ikke gør andet end at feede værdier ind i netværket. Dette lag er så efterfulgt af et lag af *skjulte perceptroner*. Ved efterfulgt forstås, at hver inputenhed er forbundet til hver af det første skjulte lags enheder. Herpå følger en vilkårlig række af skjulte lag, hvorpå det afsluttes med et output lag, se figur 1.

Matematisk kan et netværk med 1 skjult lag altså skrives som følger:

$$z_j = f_1 \sum_{k}^{N} (x_k w_{kj}^{(1)})$$

$$y_i = f_2 \sum_{j}^{M} (\mathbf{z}_j \mathbf{w}_{ji}^{(2)})$$

N er antallet af inputs, M er antallet af skjulte enheder, z_j er outputtet af den j'te skjulte enhed, y_i er outputtet af den i'te outputenhed, x_k er det k'te



Figur 1: Et simpelt neuralt netværk.

input og $w_{kj}^{(1)}$ betegner vægten mellem input n og skjulte enhed j. Tilsvarende betegner $w_{ji}^{(2)}$ vægten mellem skjulte enheder og output enheder. Til funktionerne f_1 og f_2 kan man anvende den samme, men for et klassificeringsproblem med flere end 2 output en funktion i output enhederne der tager højde for det. Valg af funktioner er beskrevet i 3.3. Det er essentielt at funktionerne kan differentieres, da dette kræves af den algoritme der bruges til læring af netværket.
Desmere har man en *bias* enhed i hvert lag (på nær output laget), som virker som en grænse systemet kan flytte, da den altid giver inputtet 1 til næste lag. Det betyder, at der for N inputs vil blive lavet en inputvektor af længde N+1:

$$\mathbf{x}' = (x_1, x_2, ..., x_N, 1)$$

Og tilsvarende for M skjulte enheder.

En prediktion går ud på at give et input til netværket og se, hvilket output det giver, mens læring består i at opsætte netværket, så det giver det rigtige output. En prediktion løber netværket igennem forfra og bliver kaldt *feed forward*. Læring sker ved, at netværket løbes baglæns og man ændrer vægtene, denne process kaldes *backpropagation*. Problemet at finde et godt netværk kan ses som at optimere en fejl funktion. Her bruges typisk summen af kvadrerede fejl. For en samling af B datapunkter, findes fejlen ved:

$$E(\mathbf{w}, \mathbf{x}) = \sum_{b}^{B} (y_b - t_b)^2$$

Da denne fejl beskrives ved inputtet $\mathbf{x}$ og vægtene $\mathbf{w}$. Da inputtet er uændret for de B datapunkter, kan fejlen for datasættet beskrives udelukkende af $\mathbf{w}$,

altså ved $E(\mathbf{w})$. Det betyder at for at opnå de bedste resultater skal vi optimere $E(\mathbf{w})$, hvilket gøres ved at ændre vægtene i retningen af gradienten, $\nabla E(\mathbf{w})$ (på eng. *gradient descent*).

Man har udviklet denne model så længe at den er blevet mere effektiv på trods af, at den så ikke minder så meget om neuroner længere. En af de store aspekter der former modellen, som ikke bygger så meget på den biologiske baggrund, er valget af funktionerne f_1 og f_2. Her anvendes typisk ikke-lineære funktioner, der måske kan være lokalt lineære. Det gør at NN kan bruges til at modellere problemer af meget forskellig artet kompleksitet. I denne opgave beskrives valget af disse funktioner i 3.3.

2.2.3 Backpropagation

Det nærværende afsnit er baseret på kapitel 5 i [C. M. Bishop, 2006], om neurale netværk og kapitel 7 i [R. Rojas, 1996].

Her følger en beskrivelse af gradient metoden. Der findes andre og bedre metoder til træning af NN, men gradient metoden er smart i en implementering fordi den kan distribueres ud i perceptronerne, så hele optimeringsprocessen ikke skal foregå centralt. Gradienten af $E(\mathbf{w})$ er som bekendt betegnet ved:

$$\nabla E(\mathbf{w}) = \left(\frac{\delta E}{\delta w_1}, \frac{\delta E}{\delta w_2}, ..., \frac{\delta E}{\delta w_k} \right)$$

For at finde denne ser vi på hvordan man differentierer hver del af netværket. Feed forward er en sammensat funktion og kædereglen benyttes til differentiering. Det udtryk vi ønsker at differentiere er sammensætningen af udtrykkene for y_i og z_j:

$$y_i = f_2 \left(\sum_j \mathbf{w}_{ji}^{(2)} f_1 \left(\sum_k (\mathbf{x}_k \mathbf{w}_{kj}^{(1)})) \right) \right)$$

Vi kan beskrive dette mere kompakt ved at definere aktiveringen ved:

$$a_j = \sum_k \mathbf{w}_{kj} \mathbf{z}_k$$

Hvorved z betegner output af enheder:

$$z_j = f(a_j)$$

Denne notation simplificerer differentieringen af udtrykket til:

$$\frac{\delta y_i}{\delta w} = f_2'(a_j) f_1'(a_k)$$

Her ses, at det er trivielt at finde $f_1'(a_k)$ da aktiveringen allerede er fundet ved feed forward udregning. $f_2'(a_j)$ derimod kan findes som en sum af differentialkvotienter fra højere oppe i netværket. Hvis differentialkvotienten for en senere enhed er δ_i, kan den for enhed j, findes ved:

$$\delta_j = f'(a_j) \sum_i w_{ik} \delta_k$$

Hvor $f'(a_j)$ er differentieringen af den indrefunktion i enheden. Se udledning af formel 5.55 i [C. M. Bishop, 2006].
Differentialkvotienten skal altså blot findes for output enhederne for at kunne backpropagate over hele netværket. Man kan for en simpelheds skyld skrive fejlen om:

$$E'(w) = \left(\frac{1}{2} \sum_i (y_i - t_i)^2 \right)' = \sum_i y_i - t_i$$

Denne er så differentialkvotient inputtet δ til outputenhederne.
Når denne er fundet for outputenheder, kan differentialkvotienten i hele netværket bestemmes, ved at backpropagate disse differentialkvotienter. Når det er gjort skal vi ændre vægtene i retningen af gradienten. Det gøres ved at tage den afledte i hver enhed og ændre alle input noder i den modsatte retning af den afledte. For alle inputs til en enhed j, ændres vægten ved:

$$w_{ij} \equiv w_{ij} + d_j \times l$$

Hvor l er læringsraten (fra eng. learning rate). Læringsraten er indført for at sørge for at algoritmen hurtigere når til lokale minima, og øger chancen for at den stopper. Læringsraten er ændres typisk efter hver gennemgang af backpropagation-algoritmen.
Valg af funktioner og designløsninger i forbindelse med implementering af NN er beskrevet i kapitel 3.3.

2.3 Pac-Man

I følgende del beskrives først de mest interessante aspekter af spillet Pac-Man. Herpå følger en beskrivelse af hvorfor det er interessant til dette projekts problemstilling, og hvilke aspekter der er udvalgt.

Figur 2: Startopstillingen i Pac-Man. Pac-Man er den gule cirkel lige under midten. Dette billede er taget fra [J. Pittman, 2011].

2.3.1 Hvad er Pac-Man

Nærværende afsnit er bygget på [J. Pittman, 2011].

Ideen bag Pac-Man er relativt simpel. Den lyder: *"Løs det kinesiske postbuds problem*[3]*, samtidig med at der er nogen der jager dig"*. I Pac-Man er der fire spøgelser som jager en menneskestyret gul delcirkel, Pac-Man. Disse bevæger sig rundt i en labyrint, hvor der på hvert felt i labyrinten ligger en prik som PM skal spise. Målet for PM er at spise alle disse prikker. Når dette er gjort starter spillet forfra i et højere tempo. PM har i hvert hjørne et våben mod spøgelserne: en stor prik. Når denne spises giver det ham mulighed for at spise spøgelserne i en bestemt tidsperiode.
Pac-Man kan bevæge sig frit rundt og skifte retning, når spilleren lyster. Spøgelserne derimod følger nogle faste AIer og har kun mulighed for at ændre retning ved labyrintens skilleveje. Desuden må de ikke bevæge sig tilbage ad den vej de kom ad. Spøgelserne kan ikke stå stille, mens PM kun kan hvis han går ind i en væg. Måden bevægelse fungerer på i Pac-Man er relativt

[3] Fra eng. *Chineese postmans problem*, er spørgsmålet om på den smartest mulige måde at bevæge sig ad alle kanter i en graf.

kompliceret, og implementeringen indleder en del varians i form af forskellige timingsvinduer[4] og bugs. Hvilket gør det mere tilfældigt, når det spilles af et menneske, men som en agent kan se bort fra. Kollision og bevægelse foregår på pixel plan, og vil ikke blive beskrevet i detalje her.
I det oprindelige Pac-Man er der fire spøgelser med fire forskellige bevægelsesmønstre. Disse fungerer sådan, at de bestemmer et *målfelt* som spøgelset prøver at nå. Når spøgelset så kommer til en skillevej, vil der for hver af de tilstødende felter blive udregnet den euklidske afstand til målet. Det felt, der har kortest afstand til målet, er så den vej spøgelset vælger. De fire måder at regne målfelter ud på er forskellige fra spøgelse til spøgelse:

Rød Har PMs position som målfelt.

Pink Har feltet fire felter foran PM som målfelt.

Orange Har PM som målfelt når den er mere end 8 felter fra ham, ellers har den et hjørne.

Turkisblå Tegner en vektor fra det røde spøgelse til PM. Den samme vektor lægges så med begyndelses punkt i PM, det punkt den slutter i er dette spøgelses målfelt.

Ovenstående bliver brugt som udgangspunkt for de AI, der anvendes i dette projekt.
Udover disse bevægelsesmønstre varierer det om disse bliver anvendt eller om spøgelserne flygter fra PM ved at gå ud i hver sit hjørne. Tidsrummene hvor spøgelserne flygter bliver kortere og færre som spillet udvikler sig. Spøgelserne flygter altid når PM spiser en af de store prikker og bringer dem i fare.

2.3.2 Afgrænsning af Pac-Man platformen

Der er tidligere lavet agenter, som har adresseret problemet med at lave en computerstyret Pac-Man. Blandt andet er der en årlig konkurrence i at lave spøgelses og Pac-Man AI[5]. Her måles spøgelses AIerne på hvor sjove de gør spillet, eller hvor svært de gør det. Fælles for de PM agenter, der er lavet, er dog at man prøver at optimere antallet af point som agenten kan opnå.

[4]PM har, når han skal runde et hjørne fem pixels hvor han kan angive hvilken vej han vil. Hvis han angiver en retning når han kommer til den første pixel vil han lave den hurtigste vending muligt. Men han kan også vente til den midterste pixel i krydset, hvilket gør at hjørnet har taget ham nogle få millisekunder længere at runde.

[5]Se http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html

Dette er et langt mere ambitiøst og komplekst mål, end det der adresseres i dette projekt. Hér prøves blot at forudse bevægelsesmønstre for spøgelser, hvilket ville være et delproblem for de egentlige PM agenter.
Det betyder at der kan laves en afgrænsning af de aspekter som Pac-Man indeholder til den prædikative model. Og man kan så generalisere de mønstre der findes, til brug i det egentlige Pac-Man. Der fjernes altså dele af Pac-Man med udgangspunkt i den forudsigende model, der skal bygges. Modellen ser på opstillingen af brættet og skal så komme med prædiktioner om, hvad spøgelser vil gøre. Den prædiktionsmodel der anvendes i dette projekt kigger ikke på tidsbaseret sammenhæng og det giver derfor ikke mening at have regler der involverer tidligere datapunkter. Så som at spøgelser ikke kan vende om i skilleveje, eller at have forskellige hastigheder på agenterne. Fordi modellerne udelukkende ser på ét datapunkt af gangen og derfor ikke ville kunne forstå denne slags regler.
Desuden fjernes pointsystemet, da det ikke har nogen indflydelse på spøgelsernes AI.
Antallet af spøgelser er også reduceret til ét. Hvilket gør at spøgelserne ikke kan have bevægelsesmønstre som det turkisblå, hvor målfeltet er baseret på positionen af et andet spøgelse. Det at der kun er én agent er egentlig blot en simplificering, da der testes på flere forskellige AI. Hvis man skulle anvende dette program til at lave en større agent ville man kunne lave et neuralt netværk pr. spøgelse og sammensætte en agent der anvendte de 4 NN til at planlægge sin færden.

3 B - Programbeskrivelse og Designvalg

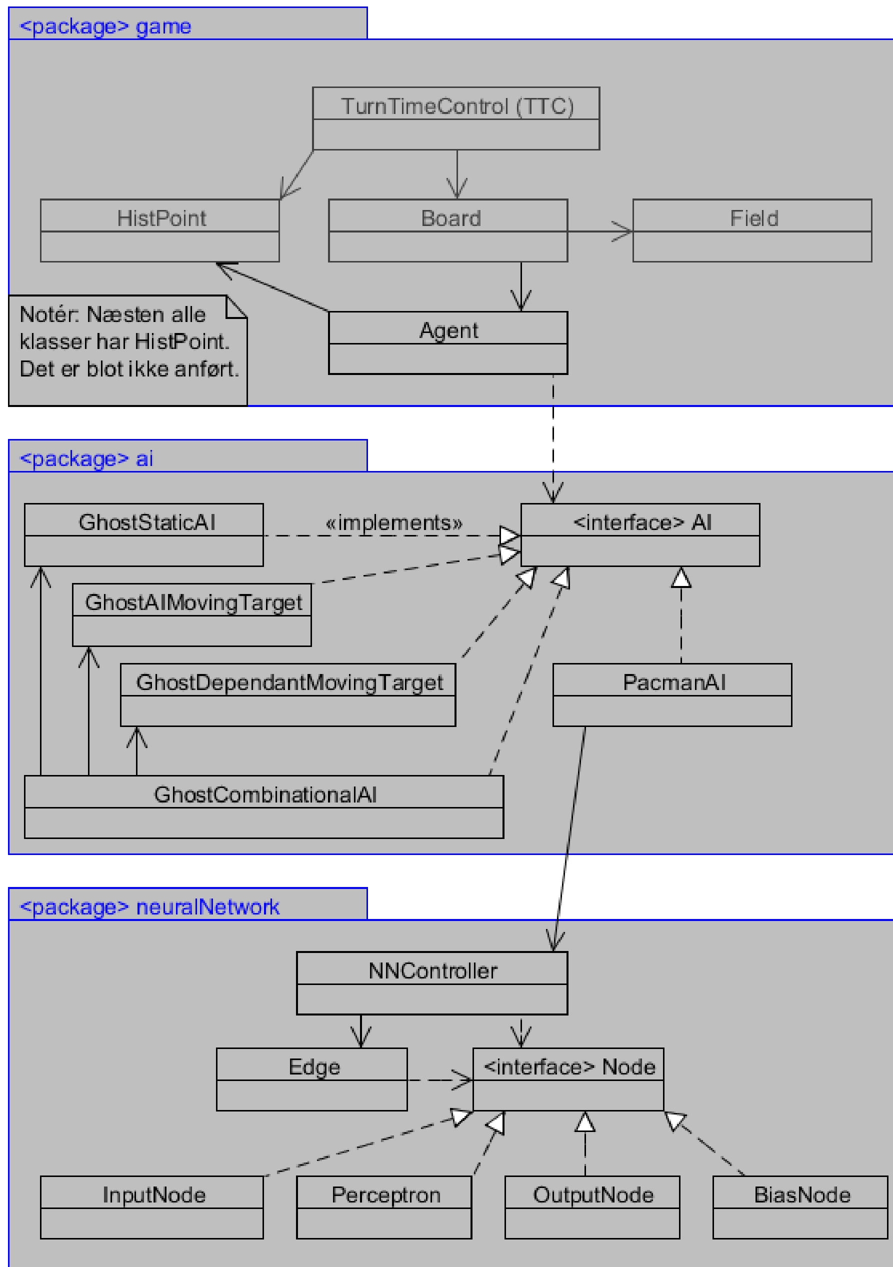
Det nærværende java program er opdelt i flere pakker (fra eng. package), disse grunder i det designmæssige planlægningsarbejde og i den process der ligger til baggrund for udviklingen af programmet. Processen har været en iterativ udvikling hvor der i hver fase har været design, udvikling og fejlfinding. Dette har givet anledning til følgende 3 grundsten i programmet, som hver har været grundlag for en iteration:

- Spilmodellen for Pac-Man
- AI
- Neuralt Netværk

Disse er i følgende kapitel beskrevet. Først i overblik i afsnit 3.1 og senere mere gennemgående, i hver sit afsnit. Der refereres med danske betegnelser til disse tre pakker, som også er beskrevet i figur 3. Hvis der refereres til pakken *game* skrives der *spil* eller *spillet*. Mens arkadespillet Pac-Man, betegnes Pac-Man. *ai* henviser til interfacet, mens AI henviser til implementeringer af *ai*. Det tydeliggøres hvis NN ikke henviser til det neurale netværk der er implementeret i pakken *neuralNetwork*.
I de mere gennemgående beskrivelser af delene lægges fokus på den problemstilling, som netop denne programdel skal løse. Spillet er således beskrevet i teknisk analytisk stil, med fokus på kontrol og sammenhæng mellem alle klasser i programmet. De simple AIers design har derimod taget udgangspunkt i den statistiske udfordring det skal være at gennemskue AIernes mønster. Det neurale netværk er beskrevet ud fra de designvalg der er taget for at implementere den matematiske struktur.

3.1 Programmets centrale elementer

Mellem de tre dele foregår der udveksling af information, som kan indgå i de beregninger, som foretages centralt i delene. Det betyder, at det har været muligt at strukturere (vha. interfaces) de forbindelser, der er fra del til del. Man kan således foretage store ændringer internt i en del, sålænge de ud- og indgående informationer har samme format. Ud over det har programmet en overordnet hierarkisk struktur, således at en kontrolklasse har et bræt som indeholder agenter med AI. Den AI PM har, indeholder et neuralt netværk. I dette afsnit ses der på de sammenhænge der kan ses i figur 3, mens de senere afsnit 3.2 - 3.4, beskriver hver af de tre store pakker mere indgående.



Figur 3: En simpel oversigt over alle klasser i spillet. Ikke alle sammenhænge er anført, blot de mest essentielle.

3.1.1 Spillets opbygning

Øverst i hierarkiet findes spillet. Det sørger for at de ting der foregår på brættet følger spillets regler. Det skal her bemærkes at spillet ignorerer eventuelle ryk som ikke er tilladt, altså ryk der er fremkommet af fejlberegninger eller misinformation i andre dele af programmet. Dette sker for at forhindre agenterne i at snyde.
Selve spillets opbygning er beskrevet i de tre klasser *Field*, *Board* og *Agent*. *TurnTimeControl* (TTC) er brugt til at holde styr på hvor der foregår beregning.
En anden af spillets opgaver er, at fortælle de agenter der måtte være på brættet, hvad der foregår. En bestemt opsætning af brættet kan gemmes i klassen HistPoint, som indeholder agenternes positioner, retninger og afstande mellem dem. Alle forrige HistPoints bliver så gemt i TTC. Det er på baggrund af disse at spøgelsernes AI tager deres valg og det er med disse parametre det neurale netværk er trænet.
Agenten er mest en transmitterklasse mellem den AI, der definerer agentens bevægelsesmønster og spillet. Det vil sige at al form for information mellem spillet og de øvrige pakker går gennem instanser af agentklassen (I visse tests er det neurale netværk tættere på driveren, da det gør det nemmere at lave statistisk arbejde).

3.1.2 AIens opgave

Den information, der bevæger sig ind i spillet, er de valg den enkelte AI foretager. Agenten kan sætte sin retning til nord, syd, øst eller vest, som spillet så tager som dens valg af hvilken vej agenten ønsker at bevæge sig. De simple AIer som spøgelserne bruger, baserer deres valg på hvordan spillet ser ud i det nyeste scenarie, de er blevet informeret om. PM husker gamle scenarier som den bruger til at træne det neurale netværk og vil, ved brug af NN, tage beslutninger på baggrund af flere scenarier.
Forholdet mellem agent-klassen og AIen er det sted, hvor der egentligt foregår udveksling. Agenten indeholder al den information som brættet giver den, om dens position og den indeholder de valg som AIen tager, på baggrund af informationen. Den fungerer altså som en klasse, der sender information fra bræt til AI og den anden vej.

Selve AI pakken består egentligt blot af et interface for kommunikation med agent-klassen og så nogle forskellige implementeringer af dette interface: Fire

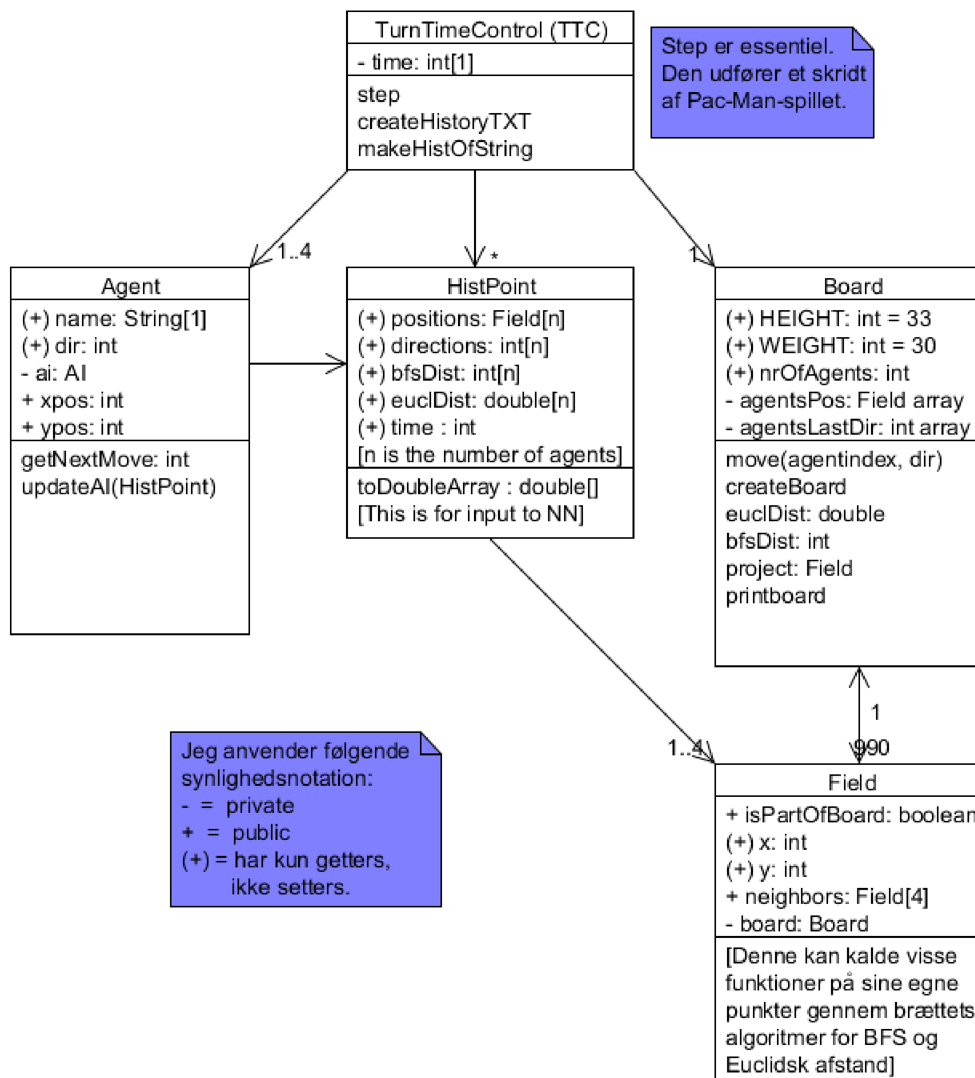
forskellige sværhedsgrader af spøgelser og én AI til PM, som er ret anderledes da det indeholder et NN.

3.1.3 Det neurale netværk

Det neurale netværk ligger i Pac-Mans AI, og har som sådan samme indputs og outputs som en AI. Men hvor interfacet til det normale AI skal kunne besvare ét spørgsmål; "Hvilken retning vil du bevæge dig i?", skal det neurale netværk ikke sige noget om PM. Derimod om de agenter som PM bevæger sig rundt sammen med. Det bliver altså med det neurale netværk også muligt at svare 'hypotetiske' spørgsmål, så som; "Hvis vi befandt os i denne situation, hvad ville spøgelset så gøre?".
Dette gør at de informationer der bevæger sig fra AI til NN også er af klassen HistPoint, og de informationer de bevæger sig tilbage er retninger. Præcis som i forholdet mellem spil og AI. Dog er fundamentet for disse udvekslinger af en anden karakter. Desuden har NN *public* metoder til at reinitialisere sig selv, eller træne på nye punkter.
Opbygningen af det neurale netværk kan godt se lidt kompliceret ud men den er meget ligefrem. Den indeholder et netværk bestående af fire forskellige slags noder og nogle kanter i mellem noderne. Og så indeholder den *NNController*, en klasse der bygger, træner og vedligeholder netværket, ved blandt andet at fortælle de forskellige dele, hvornår de skal være aktive.

3.2 Spillet

Pakken *game* har visse ret essentielle opgaver at løse for at spillet kan fungere. Den skal først og fremmest opsætte spillet, så det er klar til at blive spillet. Så skal den muliggøre at man kan tage visse valg som spiller. Og sidst, men ikke mindst, skal den sørge for at reglerne for spillet bliver overholdt. Dette afsnit beskriver først hvordan spillet er struktureret, hvordan forskellige elementer af Pac-Man er modelleret osv. Derefter ses der på hvordan spillets ture fungerer, og hvordan spillet opretholder reglerne for de enkelte agenter. Ydermere er spillets rolle som dataopsamler beskrevet i sidste delafsnit. Denne rolle er essentiel i dette projekt, da det er historikken af spillet det neurale netværk bygger på. Den data der bliver opsamlet og sendt ud er også den eneste måde alle AIerne kan se brættet. De har ingen forudbestemt forståelse for hvordan brættet ser ud, blot regler for hvordan de bør reagere på forskellige datapunkter, der følger af bestemte opsætninger.



Figur 4: Oversigt over indholdet i pakken *game*. Læg mærke til at Agent indeholder AI, hvori resten af programmet er indeholdt.

Det er også dataopsamlingen der muliggør en ekstern statistisk bearbejdning som er beskrevet i afsnit 4.2.

3.2.1 Opbygning af spillet

Som det også kan ses i figur 4 består spillet grundlæggende af tre ting. Et **bræt**, som består af nogle **felter**, hvorpå der kan stå nogle **agenter**. Brættet er derfor en klasse der indeholder en samling af felter som den sætter sammen på en bestemt måde, for at opnå en bestemt labyrint. Derudover har brættet et ansvar for at man kan kalde visse grundlæggende algoritmer der kan udregne afstande på brættet (*bfsDist* og *euclDist*), projektere felter uden for brættet ind på det (*project*) eller skrive brættet til en tekststreng (*printboard*).
Selve felterne er sat sammen ved at har 0 til 4 naboer: Ingen eller en kombination af Nord, Syd, Øst og Vest. Hvis et felt ikke har nogle naboer, betyder det at agenter ikke kan bevæge sig til eller fra det felt. Feltet betegnes som ikke værende en del af spillebrættet, men det er dog med da det stadig kan virke som målfelt for en agent (se afsnit 3.4). Ellers har felter typisk 2, 3 eller 4 naboer.
Brættet opbygger felterne ud fra en tekst fil, denne er inkluderet i bilag A (for at se strukturen af brættet bedre henvises til figur 7). Dette er gjort for at gøre det nemmere at bygge om på labyrinten. Det er en også en simpel måde at lave brættet, så man ikke behøver at hardcode en stor mængde forbindelser mellem felter.

Felterne er identificeret ved et x og y koordinat og er den forbindelse en AI har til brættet. Felterne kan tilgå afstandsalgoritmerne i brættet. På den måde har alle klasser, med adgang til felter, mulighed for at finde basale afstande på brættet. Dette bliver mest brugt af AIerne.
Agenternes primære opgave er at videregive information til AIer, for så at indhente hvilke ryk de ønsker at lave ved at angive hvilken retning de peger. Disse retninger er beskrevet som enten Nord (0), Syd (1), Øst (2) eller Vest (3).

Det er brættets metode *move*, der foretager ryk på brættet. Når brættet får at vide oppefra at en bestemt agent som ønsker at bevæge sig i en bestemt retning, vil den undersøge om dette ryk er legalt. Det betyder at den undersøger om agenten prøver at bevæge sig i en retning, hvor der ikke er felter, i så fald vil brættet rykke agenten den retning som den bevægede sig

ved sidste ryk. Hvis rykket derimod er lovligt bevæges agenten den ønskede retning.
Både opdateringen af agenter og bræt foregår ikke lokalt i brættet men i kontroldelen, TTC.

3.2.2 Turen

Spillet, som det kan ses i figur 4, er bygget op så alle klasser adlyder TTC. Det er TTC der bestemmer hvad der skal ske. Det er her der defineres hvordan en tur udformes. Alle andre klasser fungerer som et brætspil, der ikke har mulighed for at blive flyttet på uden TTC. Turen er delt ind i følgende faser:

1. **Få agenters retninger**
   Den starter med for hver agent at indsamle de ryk som agenternes AI har produceret på baggrund af hvordan brættet ser ud på nuværende tidspunkt. Agenternes retninger bliver gemt i TTC, da disse skal bruges til at undervise det neurale netværk.
2. **Opdatér brættet**
   Når information om hvad agenterne vil gøre er indhentet, ændres brættet så agenterne er rykket 1 felt i den retning de peger.
3. **Få brættets nye udseende**
   Nu bliver brættets opsætning så gemt i et HistPoint (beskrevet yderligere i afsnit 3.2.3), som bliver gemt i TTCs oversigt over alle tidligere konstellationer af brættet.
4. **Fortæl agenter hvordan brættet ser ud**
   Alle agenter bliver så givet dette nye HistPoint, hvorpå deres AI kan tage en beslutning.

Hvis en spilsession skal gemmes er det følgen af disse HistPoints, kombineret med hvilke valg der var taget på baggrund af dem, som bliver gemt.

3.2.3 Dataopsamling

Historikken af brættet er repræsenteret punktvis, i klassen HistPoint. Det betyder at en konstellation af brættet på et bestemt tidspunkt er givet ved ét HistPoint. Da brættet ikke ændrer udsende, men kun flytter rundt på de

ting der står på det, er det kun agenters information, og forholdet mellem som HistPoint opsamler. HistPoints indeholder følgende:

- Alle agenters positioner, som felter.
- Alle agenters retninger, som heltal 0 til 3.
- Afstanden med BFS, fra alle agenter til PM, som heltal.
- Den euclidske afstand fra alle agenter til PM, som rationelt tal (double).
- Tiden, som er givet ved turens nummer. TTC skriver den til HistPoints i turens 3. fase.

HistPoints er meget essentielle i resten af programmet, hvilket tydeliggøres i at næsten alle klasser indeholder HistPoints. Fordi det er i denne form at AI og NN er informeret ombrættets opsætning.
Dataopsamling foregår af to forskellige årsager. Der foregår dataopsamling i løbet af hver tur, hvor der bliver overført HistPoints fra bræt, gennem TTC og agenter, til AI. Og som afslutning af en kørsel kan data gemmes som .csv (*comma seperated values*). Den afsluttende dataopsamling foregår, tildels for at man kan lave ekstern statistisk bearbejdning af dataet, eller for at man kan starte med at træne det neurale netværk på opsamlet data fra en gammel session.

3.3 Det neurale netværks design

I afsnit 2.2 er den matematiske struktur bag et neuralt netværk beskrevet. I dette afsnit uddybes det hvordan man implementerer et NN og lærings algoritmen. Først gøres der dog rede for de forskellige klasser der er indeholdt i implementeringen. Selve opsætningen af netværket sker på baggrund af den data man træner over og er derfor beskrevet i afsnit 4.3.
De andre dele af programmet er meget specifikt lavet til, at man skal kunne spille Pac-Man på et bræt. Der er meget få steder, hvor det giver mening at tale om at lave dynamiske strukturer. Det er muligt at lave om på hvordan brættet ser ud, eller man kan lave nye og flere AIer. Men udover det er der ikke steder, hvor man kan genbruge kode til andre projekter.
NN er derimod et fantastisk eksempel på en struktur, der både er dynamisk og giver menig at genbruge. Modellen er, som beskrevet i afsnit 2.2, lavet til at kunne implementere mange forskellige funktioner, og den er derfor bygget til at skulle kunne skaleres. Ved skalering forstås det at lave om på antallet af gemte noder og antallet af lag med gemte noder. I denne implementering

er det også muligt at lave om på datatypen, ved at bestemme antallet af in- og outputs.

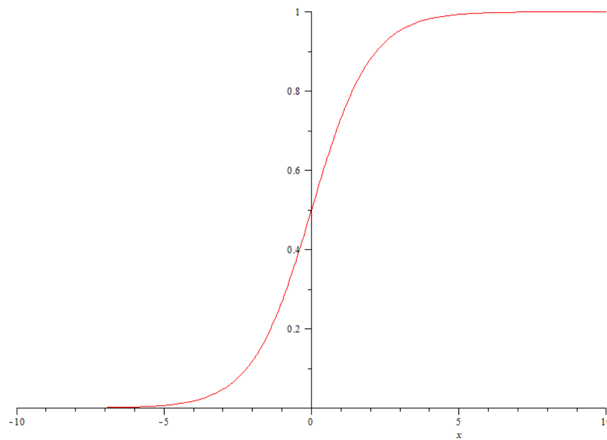
3.3.1 Design af klasser og kontrol

Det neurale netværk består grundlæggende af 2 klasser: kanter og noder. Noderne indeholder værdier for deres input og output. Desuden er det angivet, hvad de bør gøre ved *feed forward* udregning, og *backpropagation* (se 2.2). Kanterne sørger for at få et input, fra den node de har fået anført som deres input node. Og skrive til den node de har som output. Ved backpropagation, bruges andre datafelter end dem ved feed forward.
Problemet for et neuralt netværk er at man ikke bare kan lade noderne regne og sende videre selv, de bliver nødt til at vente på, at alle noder i forrige lag er færdige med at udregne, før de kan indhente alle deres inputværdier. Det betyder at der her må anvendes en kontrolklasse. Dette er klassen *NN-Controller*. Den sørger for at rækkefølgen af udregninger foregår som den skal.

Noderne er et interface som implementeres af input-, output-, perceptron- og biasnoder. Perceptronen er grundstenen for netværket. Det er her der foregår noget interessant. Input- og biasnoden er blot til for at klargøre data, mens outputnoden minder meget om perceptronen. Input sender bare de datapunkter videre den bliver givet. Bias sender altid 1 videre som output.
Perceptronen derimod summerer først over dens inputs, hvorefter den udregner sin aktivering og sætter den til at være sit output. Outputnoden gør det samme i feed forward udregning, men har mulighed for at få input fra kontrollen ved backpropagation.

3.3.2 Dataspecifikt design

Selve opbygningen af det neurale netværk, foregår med udgangspunkt i datasættet. Netværket får ligeså mange input units som der er inputs. Dvs. 8, da disse kommer fra HistPoints og tiden ikke gives med. Selve opsætningen af netværket foregår i NNController, og det er også her at dataet præformateres. Dette sker ved at alle datapunkter laves om til rationelle tal og standardiseres, så det har middelværdi 0 og varians 1. Standardiseringen gør at inputtet er i samme format som aktiveringer. Det bliver derfor nemmere for netværket at kontrollere størrelsen af de værdier, der bevæger sig igennem det.



Figur 5: Her ses den anvendte aktiveringsfunktion: $s(x) = \frac{1}{1+exp(-x)}$

Når man skal designe en outputnode bliver man nødt til også at se på aktiveringsfunktionen, da man skal omsætte bestemte rationelle aktiveringer til 4 diskrete klasser. Måden hvor på den anvendte aktiveringsfunktion virker er at den giver en aktivering mellem 0 og 1. Det virker godt hvis den klassifikation man laver er binær. Løsningen for et 4 klasses problem er det man kalder 1-af-4 lagring (fra eng. 1-of-K coding). Her omsættes en diskret værdi der har omfang 0 til 3, til en vektor med 4 binære inputs, men hvor kun én af dem må være 1 afgangen. På den måde vil positionen af ettallet svarer til, hvilken af de 4 diskrete værdier den har.
Der er lavet 4 outputnoder, en for hver klasse. Deres aktivering sammenlignes for at bestemme, hvilken af de fire retninger NN tror spøgelset vil gå.

Man kan anvende flere forskellige aktiveringsfunktioner, det eneste egentlige krav er at den skal kunne differentieres. Desuden bruger man ofte funktioner der minder om stepfunktioner[6]. Det skyldes at man laver binær klassificering eller logistisk regression, hvor man prøver at bestemme sandsynligheden for at et datapunkt, tilhører en bestemt klasse. I dette projekt anvendes:

$$s(x) = \frac{1}{1 + exp(-x)}$$

$$s'(x) = \frac{exp(-x)}{(1 + exp(-x))^2}$$

[6]En stepfunktion er en funktion der har output 0 hvis input er lavere end en bestemt værdi, og output 1 hvis input er større.

Som har værdimængde $]0;1[$. Det at anvende en ikke lineær funktion muliggør at det neurale netværk kan modellere ikke lineærer problemer bedre. En anden fordel er at s(x), hvis man øger vægtene meget, fungerer som en stepfunktion, hvor biasnodens vægt udgør grænsen hvornår outputtet skal være henholdsvis 0 eller 1.
For sammenligning af de fire outputenheder bruges så *softmax* funktionen. For N outputs $z_1..z_N$ tages der softmax ved:

$$softmax(z_j) = \frac{exp(-z_j)}{\sum_n exp(-z_n)}$$

Som for N = 2 svarer til bare at bruge s(x). Den ouputenhed der får højest aktivering i softmax, bestemmer udseendet af outputvektoren (ved anvendelse af 1-of-K coding). Er den tredje outputenhed aktiveret af softmax, fås outputvektoren $[0, 0, 0, 1]$, svarende til vest.

3.3.3 Læring i praksis

Når man skal træne det neurale netværk bruger man batch metoder. Det vil sige at man for en stor mængde af data finder gradienten for hvert datapunkt. Differentialkvotienten, der bevæger sig ind i noderne, gemmes for hvert datapunkt. Når gradienterne for alle datapunkter er udregnet, summeres de og vægtene kan ændres. Vægtene ændres med en bestemt læringsrate, som beskrevet i 2.2.
Efter hver iteration af disse vægtopdateringer undersøges det om misklassificeringsfejlen stiger eller falder. Målet er at den skal nå et minimum hurtigt, men den skal også sørge for at stoppe, når man når et minimum. Hvis fejlen falder, fordobles læringsraten så man nærmer sig minimum hurtigere. Men når fejlen øges, ændres læringsraten til en femtedel af, hvad den var før. På den måde vil den gradvist nærme sig et minimum. Når fejlen ikke længere ændrer sig fra iteration til iteration, termineres læringsprocessen. Det betyder at man har nået et lokalt minimum.
Som udgangspunkt er vægtene valgt tilfældigt, mellem -1 og 1. Det vil altså være muligt at opnå bedre resultater ved yderligere kørsler på samme data. Det at vægtene vælges lavt gør at aktiveringsfunktionerne ikke modellerer fuldstændigt som en stepfunktion. På den måde sikrer man sig, at de kun begynder deres ikke-linearitet, hvis det hjælper modellen.
Det der dog skal passes på med denne træningsmodel er at en høj sum af gradienter kan ændre vægtene drastisk. Da det i praksis aldrig giver mening at ændre vægtene særlig meget, fordi $s(x) \approx 0$ for $x < -10$ og tilsvarende er

$s(x) \approx 1$ for $x > 10$ (se figur 5). Derfor er der indført en begrænsning på hvor stor ændring man kan lave på vægtene, så de aldrig kan ændre sig mere end ± 1 for hver iteration. Det gør også alle former for *overflow* praktisk umulige.

3.4 Design af bevægelsesmønstre

Designmæssigt er der ikke meget at nævne om den kunstige intelligens. Der er fem forskellige udgaver af det samme interface, som på baggrund af en række HistPoints kan bestemme, hvad retning en agent bør bevæge sig i. Der er dog det interessante, at desto flere af elementerne i HistPoint, der bliver brugt til at bestemme retningen af en agent, desto sværere bør det være for en prædikativ model at finde et system i dens færden. Det følgende afsnit gør rede for, hvilke bevægelses mønstre jeg har valg for spøgelserne, for at kunne undersøge det forhold, der mellem kompleksitet af AI og af model.
Der er meget lidt at notere om designet af en AI, som ikke allerede er nævnt i 3.1. Grundlæggende skal det have to funktioner:

- Man skal kunne give AIen det nyeste HistPoint (*addHistPoint*)
- Man skal kunne få det næste træk AIen ønsker at lave for sin agent (*getNextMove*)

Disse kan findes i interfacet *ai*.

3.4.1 Spøgelsernes AI

I det oprindelige Pac-Man er der nogle forskellige AIer for de fire spøgelser, se afsnit 2.3. Disse bruger kombinationer af følgende parametre til at bestemme hvilket målfelt de har:

1. Forudbestemt stationært felt.
2. Position af spøgelse.
3. Position af PM.
4. Afstand til PM.
5. Retningen PM peger.
6. Vektor til PM fra andet spøgelse.
7. Positionen af andre spøgelser.

8. Tiden.

Vektorer til PM og positionen af andre spøgelser er ikke en mulighed her, da disse begge involverer flere end én agent. Tiden kan kan ikke bruges direkte da modellen NN ikke håndterer, at datapunkter har en bestemt rækkefølge. Dog kan modulus af tiden anvendes.
Til at implementere spøgelses AI i dette projet kan der altså anvendes de fem første parametre. Dette lægger grundlaget for de fire forskellige AIer. De er bygget delvist på de oprindelige AIer fra Pac-Man. Nedenfor er anført deres sværhedsgrad og den farve spøgelset med samme AI havde i det oprindelige Pac-Man. Implementeringerne har alle *GhostAI-* som præfiks, resten af navnet er angivet i kursiv.

Simpel AI, ingen (*Static*)
: Denne går efter ét bestemt punkt, fuldstændig uafhængigt af spillets tilstand. Det svarer til at den udelukkende bruger **sin egen position**.

Medium AI, Rød, (*MovingTarget*)
: Denne går konstant efter PM, altså anvender den **sin egen** og **PMs position**.

Svær AI, Pink, (*DependantMovingTarget*)
: Denne tager feltet n foran PM, hvor n er et heltal. Dvs. den bruger **sin egen** og **PMs position** og den bruger **PMs retning**.

Meget svær AI, Orange, Pink og Rød, (*Combinational*)
: Denne anvender alle af de tre ovenstående, afhængigt af afstanden til PM.

Her kan man se en klar udvikling i mængden af parametre der anvendes til at afgøre retningen. Dette kan ses, ikke bare på mængden af parametre, men også på at agenterne bygger på hinanden. Den første er statisk, næste afhænger af et punkt i bevægelse og den næste af et punkt i bevægelse med en retning. Og endeligt er de tre bygget samen.
Det er dog interessant at notere at ingen af dem behøver en hukommelse, hvilket betyder man ikke behøver en tidslig dimension for at gennemskue dem.

3.4.2 Design af PMs AI

Man ville kunne gå meget ind i at lave en optimerende AI til PM som havde forskellige mål i Pac-Man. Enten mål som at overleve eller indsamle flest

point, evt. på tid. Som det også er beskrevet i 1.2 arbejder dette projekt dog kun med en undersøgelser af om man kan bruge NN til at lave en sådan planlægning.
For at besvare det spørgsmål stilles der lidt andre krav til hvordan PM bevæger sig. Det er vigtigt at han ser så mange mulige scenarier som muligt, for at give netværket den mest varierede træning. PMs AI er derfor lavet som den statiske spøgelses AI. Dog med den ret vigtige ændring at hvis Pac-Man kommer til sit mål laver han et nyt mål.
PM opdaterer også det neurale netværk hver gang AIen bliver informeret, om nye HistPoints. Det neurale netværk bliver imidlertid ikke anvendt til egentlig optimering af faktorer i denne implementering.
Det ville være en interessant udvidelse at lade Pac-Man anvende NN til at finde scenarier som han ikke havde set før og derved gavne træningen af NN. På den måde kunne man lave en hypotesetestende agent, der kunne forkaste bevægelsesmønstre den havde fundet.

4 C - Behandling af Prædikative Modeller

Følgende kapitel introduceres med en forklaring af de statistiske værktøjer, der er opsat for at kunne evaluere en prædikativ model. Derpå følger beskrivelsen af to andre prædikative modeller: valg træer og logistisk regression. Evalueringsværktøjerne er anvendt på disse to modeller til senere sammenligning. Derefter følger der en beskrivelse af hvordan NN er trænet og evalueret. Endeligt ses der på en sammenligning af de 3 prædikative modeller. I konklusionen (kapitel 5) anvendes disse resultater til at finde den agent der ville være bedst til en fuldbyrdet Pac-Man agent.

4.1 Introduktion til statistiske undersøgelser

I det følgende afsnit ses der først på hvordan man sammenligner prædikative modeller, og hvordan man træner dem på en smart måde. Derefter følger et indblik ind i de sammenhænge som uden modeller kan visualiseres i det genererede datasæt.

4.1.1 Test og krydsvalidering

Dette afsnit er baseret på afsnit om at evaluering af en models effektivitet (4.4-4.6) i [Tan, Steiback & Kumar, 2006].

For at kunne sammenligne forskellige prædikative modeller, bør man have nogle undersøgelser der ikke favoriserer bestemte modeller. Man skal altså træne og teste modellerne på lige fod. Dette gøres ved at man ud af det tilgængelige data, inddeler dataet i trænings- og testsæt. Modellerne får så mulighed for at træne på træningssættet.
Derefter lader man de prædikative modeller forudsige over testsættet, og sammenligner deres prædiktioner med hvad svaret var i testsættet. Dette leder til en fejlrate som kan sammenligne modellerne.
Hvis i et datasæt med forskellige ægte funktioner, ønsker at sammenligne hvor svært modellerne har ved at modellere disse funktioner, kan man sammenligne med majoritetsmodellen. Denne finder den klasse der er bredest repræsenteret i problemet, og gætter på den for alle datapunkter i testsættet[7].

[7] Hvis der er lige stor repræsentation af alle klasser, vil dette svare til at gætte tilfældigt.

Alle modeller med bedre klassificeringsrate end majoritetsmodellen, har kunnet forstå egentlige sammenhænge i dataet. Hvor stor denne forståelse (eller misforståelse) er kan ses på afvigelsen fra majoritetsmodellens fejlrate. På den måde kan man sammenligne forståelsen af forskellige ægte funktioner.

En af de store problemer ved træning af modeller i maskinlæring er *overfitting*. Overfitting opstår ved at en model træner alt for meget på et bestemt datasæt og tror at det datasæt indeholder varians nok, til at kortlægge den ægte funktion. Det gør at modellen laver specifikke regler hvor den burde lave generelle.
For at komme dette til livs laves der krydsvalidering. Her deler man datasættet op i K dele, hvor K er et heltal. Man laver så K udgaver af den model man er i gang med at træne. Man lader så de K modeller træne på træningssættet, men hvor man for hver af dem har udeladt en K'te del af datasættet. På den måde er ingen af dem er trænet på fuldstændig den samme data. Derpå testes hver af de K modeller på den del af træningssættet de ikke har set før. Nu vil det blive klart hvilke modeller der har overfittet dataet for nært, og man vælger så den der har lavest fejl i denne træningstest.
Man siger at krydsvalideringen har K fold. Generelt bliver modellerne bedre desto flere fold en validering har, med den ultimative værende lige så mange fold som der er datapunkter i træningsdataet. Problemet er dog at man også skal lave lige så mange modeller som der er fold, hvilket kan være utroligt tidskrævende.

Den vigtigste ting i forbindelse med træning af en model er mængden af data man har til at træne den. Hvis datamængden er for lille er der en stor usikkerhed forbundet med den model der bygges. Man sammenligner derfor ofte hvordan forskellige prædikative modeller klare sig ved forskellige datastørrelser.

I dette projekt er der genereret 1500 datapunkter for hver af de 4 AIer. De første 1000 datapunkter er dedikeret til træning af de prædikative modeller, mens de sidste 500 bruges til sammenlignende tests.
Alle modeller er trænet ved krydsvaliderings fold på 10.
Der trænes på flere datastørrelser så træningsdataet bliver ikke brugt fuldt ud hver gang. Datastørrelserne er 10, 100 og 1000.

4.1.2 Forventninger og sammenhænge i data

I de følgende beskrives der hvordan eller om det kan blive problematisk for simple modeller at gennemskue mønstret i den data der er produceret. Først bør der derfor ses på om der er muligt at se mønstre i dataet overhovedet.
Dataet er lavet ved tilfældigt at placere agenter, og lade dem bevæge sig et

AI difficulty

PMpos		GHpos		Directions		Distance	
15.1567	16.0087	11.8820	13.5760	1.0727	1.5680	19.8127	14.2359
15.1667	13.9880	14.2513	14.9527	1.0687	1.3580	13.5093	10.2110
15.0820	15.2647	15.8933	14.0680	1.0927	1.2113	14.7760	11.1494
15.0820	15.2647	15.8933	14.0680	1.0927	1.2113	14.7760	11.1494

Figur 6: Middelværdier

antal gange. Hvorefter de er blevet placeret tilfældigt på ny. Derfor vil der være visse mønstre som er helt oplagte, specielt når datasættet er så relativt stort som det er. For eksempel vil middelværdien af positioner være givet ved størrelsen af brættet, se figur 6. Det eneste egentligt interessante man kan se ud fra middelværdier er at den simple AI, som ikke afhænger af PM har højere middelværdi for afstanden til PM, og lavere for spøgelses position. Afstanden til PM bliver øget da spøgelset ikke følger efter ham på nogen måde, mens den lavere positionsværdi kommer af positionen af målfeltet.

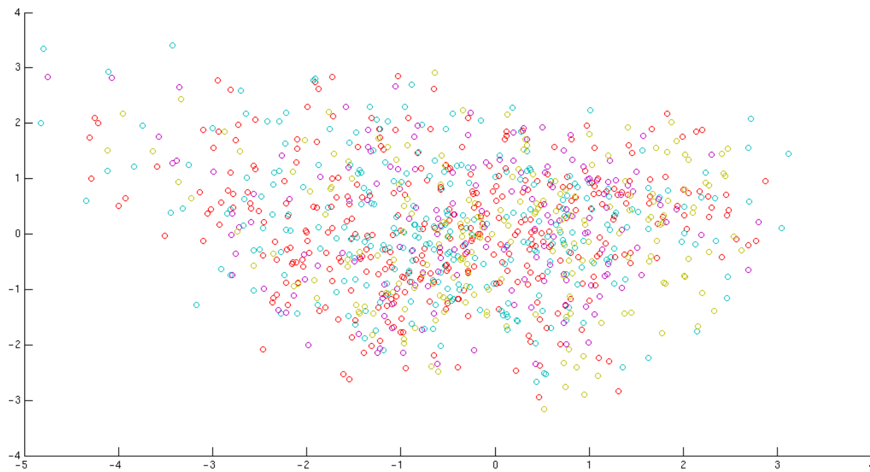
En anden måde at beskrive dataet på er forskellige former for visualiseringer. Den simpleste af disse er helt klart at udskrive brættet og så se hvad en agent ville gøre i den situation brættet er i. Problemet er at der kan være uoverskueligt mange konstellationer. På trods af at dette er tilfældet for størstedelen af de anvendte AI'er er det ikke tilfældet for den simpleste. Den kan godt nok antage alle felter som målfelter, men i den data der er lavet har der altid været anvendt det samme felt. Det gør at der for hvert et felt på brættet er ét, og kun ét, bestemt træk som AI'en vil lave. Dette er beskrevet i figur 7, hvor der ses visse sammenhænge, men det ikke er ligefremt at lave gode områdebaserede opdelinger.

En anden form for visualisering er det det der hedder principiel komponent analyse[8]. Her projekteres højdimensionel data på en sådan måde at variansen

[8] PCA - fra eng. principle component analysis. Denne kan også findes i kapitel 3.3 om visualisering i [Tan, Steiback & Kumar, 2006]

```
S E E E E S W W E E E S     S W W E E E S W W W W S
S         S           S     S           S         S
S         S           S     S           S         S
S         S           S     S           S         S
E E E E E E E E S W W W W W E E E S W W W W W W W W
N         N     S                 S     N         N
N         N     S                 S     N         N
E E E E E S     E E E S     S W W W     N W W W W N
          S           S     S           S
          S           S     S           S
          S     E E E W W W W W W W     S
          S     N                 N     S
          S     N                 N     S
          E E E N                 N W W W
          N     N                 N     N
          N     N                 N     N
          N     N W W W W W W E E N     N
          N     N                 N     N
          N     N                 N     N
E E E E E N E E N W W W     E E E N W W N W W W W W
N         N           N     N           N         N
N         N           N     N           N         N
N W W     N W W E E E N W W W W W W E E N     E E N
    N     N     N                 N     N     N
    S     N     N                 N     N     S
E E E E E N     N W W W     E E E N     N W W W W W
N                     N     N                     N
N                     N     N                     N
N W W W W W E E E E E N W W N W W W W W E E E E E N
```

Figur 7: Her ses de valg den simpleste AI ville tage for hver position, hvis den havde det blå 'E' som mål. (Bemærk der benyttes engelske forkortelser for verdenshjørnerne)



Figur 8: Her ses de to første principielle komponenter plottet, altså de to der indeholder størstedelen af variansen for datasættet. De fire klasser har hver sin farve.

kommer være koncentreret i de første komponenter. Det er her normalt ret nemt at se om man kan adskille dataet på en simpel måde eller ej. Altså om modellen vil have nemt ved at finde regler i forskelligheden af datapunkterne. Denne projektion kan ses i figur 8.
Det fremstår af denne figur at det kunne blive meget kompliceret at lave en adskillelse. Det er ikke til at sige om der er visse områder hvor en form for klasse er i overtal. Det betyder også at hvis modellerne får gode resultater kan det være yderst svært at fortolke de sammenhænge de har fundet, da disse må være meget komplekse.
PCA vil ikke blive forklaret yderligere her, men kan findes i [Tan, Steiback & Kumar, 2006].

HistPoints lidt ukonkrete punktform er det eneste en model får at vide om universet agenterne bevæger sig rundt i betyder det også at agenten ikke har nogen forståelse for universet. Den ved ikke hvordan brættet ser ud og har ved ikke at der er steder man ikke må være. Den ved heller ikke at der er visse attributter der bestemmer position, mens andre afstand eller retning. Den kan dog tegne brættet med de store mængder af data den har fået. Og kan gennem store datamængder få en forståelse for hvordan forskellige attributter hænger sammen. For at give et eksempel kan den euklidiske afstand afledes af PM og spøgelsets positioner.

En model kan derfor forventes at bruge en vis mængde af dataet på at træne grundreglerne i spillet, mere end at gennemskue de egentlige bevægelsesmønstre en anden agent måtte have. Det gør at man klart må forvente at de små data sæt vil klare sig meget dårligt da de ikke har set nok eksempler til at have tegnet brættet ordenligt.

4.2 Alternative løsninger til prædiktionsproblemet

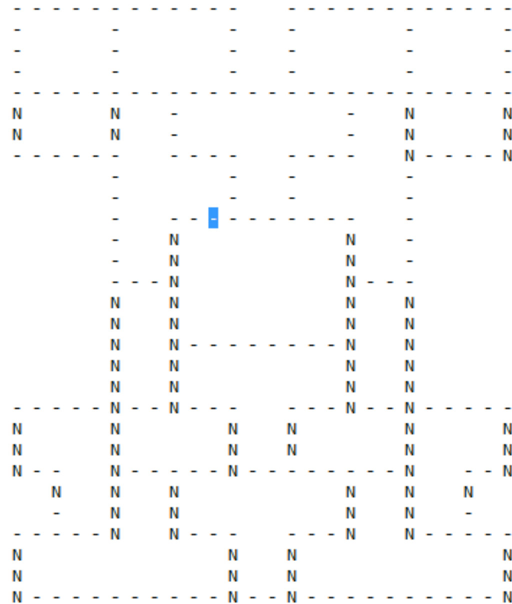
I det følgende beskrives to forskellige løsninger til det at prædikterer hvad et spøgelse gør. Ideen er her at undersøge om et NN er for kompleks en model til at modellerer problemet ordenligt. Derfor er der valgt to simplere modeller, med forskellige træk.
For at gøre undersøgelsen mere dybdegående undersøges det også om der er nogle af AIerne der med fordel kunne modelleres af simplere modeller end ved NN. Alle implementeringer er sket i matlab, koden kan findes i samme ZIP-fil som javaprogrammet.
Hypotesen er at de komplekse problemer giver flere fejl for simple modeller, men bedre kan modelleres af de større modeller.
Et andet spørgsmål kunne være en undersøgelse af om NN var for simpel en model, men det vil ikke blive adresseret her da det at anvende mere komplicerede modeller er for tidskrævende.

I det følgende afsnit er der først beskrevet decision trees og logistisk regression, hvorefter de statistiske resultater for disse vises. Sammenligningen med neurale netværk sker i 4.4.

4.2.1 Valgtræer

Følgende afsnit er bygget på afsnit 4.3 i [Tan, Steiback & Kumar, 2006].

Valgtræer er en meget simpel struktur der bygger på en meget binær opdeling af verdenen. Træet består af noder med 2 børn, og har altså struktur som et binært træ. Ideen bag dem er meget simpel: Hver node opstiller et spørgsmål der kan besvares binært (med sandt eller falsk). Afhængigt af svaret bevæger man sig så den ene eller den anden vej. Sådan fortsættes der til et blad nås. Hvert blad bestemmer så en af klasserne i klassificeringen.
Spørgsmålene er egentlig en grænse på en af de indgående datapunkters parametre. Det vil sige i Pac-Man tilfældet kunne det være ting som: "Er af-



Figur 9: Her ses igen på den simple AI fra figur 7, dog hvor der er fra filtreret alle retninger på nær nord.

standen mellem PM og spøgelset højere end 5.3?".
Det er selvfølgelig opbygningen af disse spørgsmål som former de forskellige modeller som træet kan udgøre. Dette er en relativ simpel model som er god at anvende i meget diskrete scenarier med relativt bestemte mønstre. Det kan være en fordel, da visse af de simplere AI kan beskrives ved få regler.
Træningen af træerne består i at man finder og opdeler den attribut, der deler dataet bedst muligt. Med den opdeling laves der en node, for hvis børn der foretages den samme form for opdeling for det data er kommer ud i dem. Kriterierne for hvad en god opdeling er defineres af entropi eller klassificeringsfejl.

4.2.2 Logistisk regression

Nærværende afsnit bygger på tidligere refereret litteratur om NN og på afsnit 5.6 om sammensatte modeller fra [Tan, Steiback & Kumar, 2006].

Den anden form for modellering er logistisk regression. Denne minder meget om et neuralt netværk, da den faktisk bare er én perceptron, som man bruger til at modellere. Det smarte ved det er at den kan bruges til at sammenligne

Datasæt størrelse	Simpel AI	Medium AI	Svær AI	Meget Svær AI
10	0.6960	0.7160	0.7360	0.7360
100	0.7460	0.7660	0.7440	0.7160
1000	0.7660	0.7600	0.7320	0.7400

Tabel 1: Her ses test fejlen ved brug af valgtræer.

dette projekts implementering med en mere gennemarbejdet implementering, hvor der findes lidt flere kontrol elementer end i det neurale netværk der blev beskrevet i kapitel 3.
Ligesom perceptronen bruges der en funktion med værdimængde mellem 0 og 1 som output. Det betyder at man kan se outputtet som sandsynligheden for at et datapunkt tilhører en klasse. Dog kan man ikke modellere flerklassesproblemer. Så for at kunne anvende logistisk regression på det datasæt der arbejdes med her er der lavet fire modeller. De klasser de inddeler i er så blevet sammensat af flere. Den første model prøver at beskrive sandsynligheden for at nord er outputtet af AIen. Den anden syd, osv.
Ved bestemmelse af nye datapunkter bestemmes der for hver af de fire modeller et tilhørsforhold. Der klassificeres dermed til den klasse hvor tilhørsforholdet er størst. Dette kan gøres på mere komplekse måder så man blive nødt til at lave endnu flere modeller end fire.
Det bør også bemærkes at det simplificerer problemet for hver af de fire modeller at problemet gøres binært. Hvilket er noget der kan lade sig gøre her, da der laves *supervised learning*. Det kan altså ikke forventes at et neuralt netværk med 4 noder svarer til denne opstilling af 4 perceptroner.
Hvis man ser på eksemplet med den simple AI fra figur 7, vil det svare til at en af dem blot skal se på en del figur nemlig figur 9. I denne figur er det markant nemmere at skelne hvordan og hvornår agenten bevæger sig mod nord.

4.2.3 Resultater

Effektiviteten af de to modeller kan aflæses af deres fejlrater i kombination af hvilken AI og hvilken datastørrelse der er brugt. I det følgende afsnit beskrives de fejlrater der kan findes i tabel 1 og 2.
Det umiddelbart første der observeres er at der er meget lille forskel i fejlene fra de to modeller, og at de er meget høje for begge modeller. Der vil blive

Datasæt størrelse	Simpel AI	Medium AI	Svær AI	Meget Svær AI
10	0.7960	0.7280	0.7640	0.7420
100	0.7340	0.7640	0.7620	0.7560
1000	0.7260	0.7460	0.7120	0.7200

Tabel 2: Her ses test fejlen ved brug af logistisk regression.

beskrevet yderligere om modellerne er brugbare eller ej i afsnit 4.4.
Fra fejlraten af træerne ses der flere ting. En af de store tendenser er at fejlraten ikke nødvendigvis forbedres i takt med at størrelsen af datasættet stiger. Det kommer sig af at træerne prøver at tilpasse sig mere data ved at blive større og mere komplekse. Men de tilpasninger de laver bygger på meget specielle tilfælde og er højst sandsynligt ikke beskrivende for den bagvedliggende AI. Dette kaldes at træerne over tilpasser træningsdataet.
Der kan anvendes visse kontrollerende elementer for at stoppe en sådan over tilpasning, som ikke er taget i brug her. Dog ville effektiviteten af disse midler være begrænset når træet tilpasser dataet så dårligt.

Logistisk regression kan man se bliver bedre som datastørrelsen øges. Den har altså bedre mulighed for at tilpasse sig den type af data som der arbejdes med. Det giver større håb for NN da disse modeller minder en del om hinanden. Dog er fejlen stadig meget stor.
Det er sandt for begge af de to modeller at det næsten ikke er muligt at skelne de tre AI typer fra hinanden. Når datasættet først bliver stort virker det ikke som om der er en egentlig forskellighed i kompleksitet at spore. Det kan dog være svært at sige noget om når modellerne har så høj en fejlrate.

4.3 Træning af det neurale netværk

I det følgende afsnit beskrives der først den normale praksis for at skalere et netværk. Derefter ses der på hvordan opsætningen af dette projekts netværk er lavet. Afslutningsvis ses der på resultaterne af det neurale netværk og effektiviteten af de forskellige læringselementer.

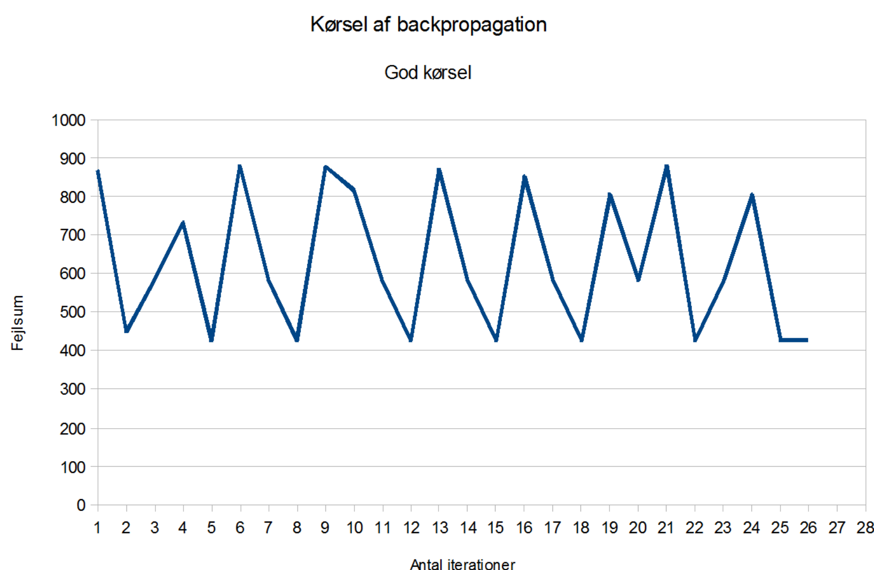
4.3.1 Praksis for opsætning af et neuralt netværk

Når man skal sætte et NN op er der tre grundsten som definerer netværket.

Først og fremmest er der den algoritme der bruges til træne det. Denne er allerede beskrevet i afsnit 3.3.

Det andet er antallet og sammensætningen af skjulte noder. Disse former det neurale netværk så den kan modellerer mindre eller mere komplekse funktioner. Det kan være relativt svært at gennemskue hvor mange noder man behøver for at modellere et bestemt problem og man foretager derfor typisk en iterativ valgprocess, hvor man starter med 1 og så tilføjer flere så længe at antallet af fejlklassificeringer falder. Det skyldes at backpropagation er en ret tidskrævende algoritme, som tager længere tid at træne i takt med at netværkets størrelse stiger.
Der er her hovedsagligt fokus på at lave ét lag af skjulte noder. Tildels fordi de fundne beskrivelser af hvad flere lag gør har været få. Men også fordi det ikke har givet bedre resultater i praksis.

Det tredje er kontrolparametre. Dette er fællesbetegnelse for læringsraten og stopkriteriet for algoritmen. Stopkriteriet er typisk den ændringen i fejl fra to på hinanden følgende iterationer af backpropagation. Det er altså den afledte fejl, ikke en fejlklassificeringsrate. Det er dog absurd at vente på at denne fejl går i nul, da de sidste mange iterationer i så fald ikke giver nogen egentlig ændring i klassificeringsraten.
I dette projekt er der desuden indført en lidt speciel kontrolparameter, nemlig at fejlen ved termination, skal være den laveste der er set. Det betyder at algoritmen ikke altid terminerer, hvilket der tages højde for ved at man ikke skal træne mere end 100 iterationer. Grunden til denne parameter er at algoritmen nogen gange terminerede i et forkert minimum. Det på trods af at læringsraten var høj nok til at få den tilbage til et bedre minimum. Men da fejlen regnes diskret bliver forskellen i fejl meget nemt 0. Det gjorde at man enten kunne bruge læringsraten til at opsætte et stop kriterium, men det er lidt mere oplagt bare at anvende fejlen.
Læringsraten er interessant fordi det er den der skal sørge for at der nås frem til et lokalt minimum. Hvis den er for høj når den når minimumet kan den meget nemt komme til at hoppe videre til et nyt. Hvilket både kan være godt og skidt, afhængigt af om det nye minimum er mindre elle større end det gamle. Hvordan man normalt initialisere denne rate har ikke været beskrevet i nærværende litteratur, men der er fundet en metode som kan bruges.

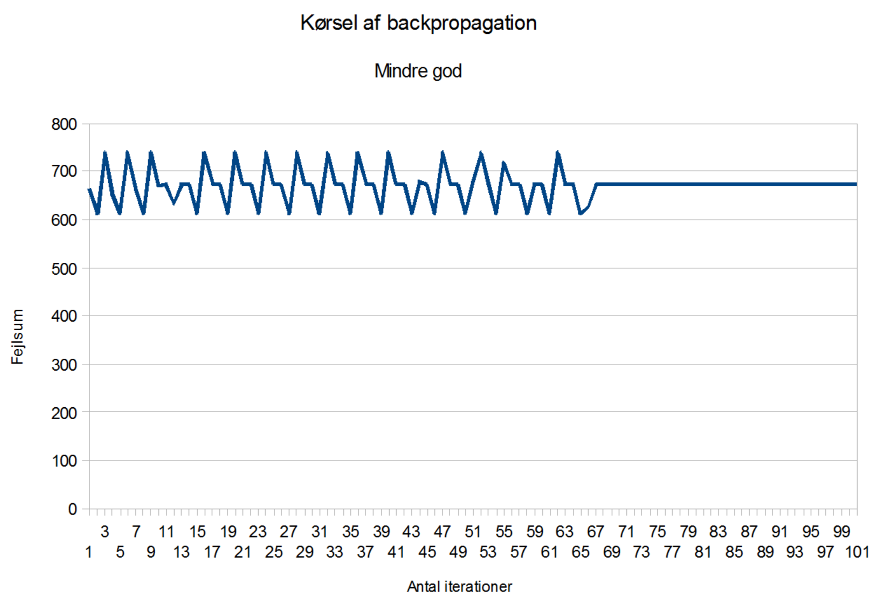


Figur 10: Her ses en kørsel af NN på svær AI, som foretager en o.k. optimering.

Her bliver læringsraten initialiseret til at være 0.5 og multipliceres med 2 hvis sidste gradientskridt var mod minimumet, men divideres med 5 hvis det var væk fra.
Man kan også indføre yderligere vægtkontrol, som i højere grad afstemmer vægtene ifht. den bias de måtte have overfor træningsdataet. Dette er dog ikke implementeret her, men er det typisk i anvendte neurale netværk.

4.3.2 Skalering af NN til Pac-Man

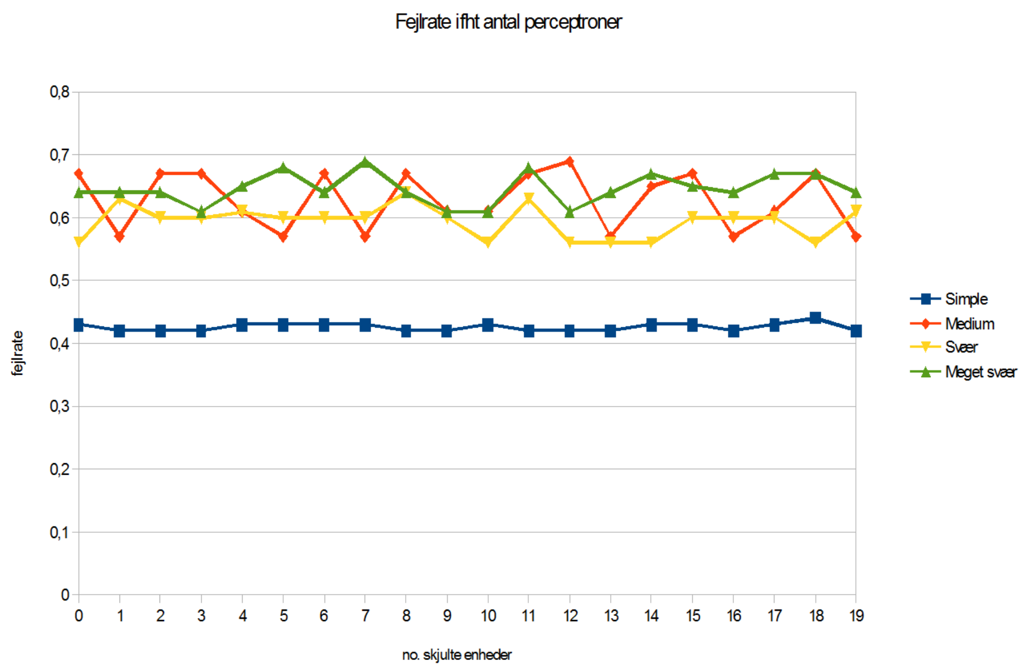
Det første og mest essentielle at slå fast for NN som model, er at undersøge om der foregår en egentlig læring, ved kørslen af backpropagation. Effektiviteten af denne læring ses der på senere, men i første omgang handler det om at undersøge om netværket overhovedet virker. Figur 10 og 11 viser to forskellige læringsscenarier. Ved den *gode kørsel* når algoritmen ned i det bedste minimum den har fundet, mens den i *den dårlige* finder et andet og ringere minimum.
Der kan udledes mange ting fra disse to grafer. Men det første og mest basale er at det kan ses at der foregår en læring. Men at den ikke altid stopper ved



Figur 11: Her ses en kørsel af NN på svær AI, som ikke rammer det bedste lokale minimum den havde fundet.

det optimale minimum. At det så virker som om at læringen tiltider bliver glemt og genlært, kommer af at det ikke har været muligt at afstemme kontrolparametre ordenligt.
Disse grafer er lavet af træningsdataet og der kan derfor ikke siges så meget om effektiviteten af modellen. Dette bliver behandlet ved sammenligning af de 3 implementerede modeller, i afsnit 4.4.

Ved skalleringen af netværket bruges den metode der er beskrevet i forrige afsnit. I figur 12 er udviklingen i fejlrate for hver af de 4 datasæt plottet, som antallet af skjulte noder stiger. Hvad der er virkelig interessant ved disse grafer er at der, i modsætning til normalen af NN træning, ikke er nogen forbedring at spore når antallet øges.
Grunden til denne mangel på tilpasning skyldes at de ting der kan gemmes i netværket bliver gemt så godt som de kan, men at der er ikke forbedring i netværkets analyse som det vokser sig større. Det skyldes måske at netværket ikke er sat ordentligt op til den slags funktion der ligger bagved, eller at læringen ikke er præcis nok. Hvilket igen falder tilbage på kontrolparametrene, da det er vist at der foregår læring i netværket. Den mest modellerende kontrolparameter er selvfølgelig den funktion som perceptronerne bruger. Den



Figur 12: Her ses hvordan test fejlen ændrer sig som funktion af antallet af skjulte noder.

AI:	Simpel	Medium	Svær	Meget svær
Middelværdier:	0,4255	0,628	0,594	0,646

Tabel 3: Tabel af middelværdier for figur 12.

Datasæt størrelse	Simpel AI	Medium AI	Svær AI	Meget Svær AI
10	0.668	0.752	0.812	0.676
100	0.466	0.79	0.804	0.78
1000	0.466	0.744	0.58	0.64

Tabel 4: Her ses test fejlen af en model med 4 skjulte noder. Bemærk at denne er valgt ved krydsvalidering og derfor er den bedste af 10, trænet på samme data.

der anvendes her er meget generel. Det ville kræve yderligere analyse for bedre at kunne modellere lige præcis dette problem.

En interessant ting man dog kan udlede fra figur 12, er at der er en klar forskel mellem de forskellige AIers kompleksitet. I tabel 3 kan man se middelværdierne fra figur 12. Her ses det at den simple AI markant nemmere kan kortlægges, end de tre andre AIer.
Rækkefølgen af hvilke AIer der var mest komplicerede er lidt anderledes end forventet. Fejlene er imidlertid så tæt på hinanden at det kan være svært at sige, om dette grunder i egentlig forskel i kompleksitet eller blot tilfældighed. Ser man på hvordan datastørrelserne har indflydelse på læringen, vist i tabel 4, som viser testfejl. Er det interessant at bemærke at et datasæt på 10 i alle tilfælde er for lidt, mens det på 50 varierer fra model til model. Den kan dog ses at hvis datasættet forstørres vil vi for det meste få bedre modeller. Præcis som antaget.

4.4 Sammenligning af resultater fra de 3 modeller

Det kan være svært at sige om en model er god eller ej. Det er klart at hvis en model har en test fejlrate over et relativt stort testsæt på 0% eller lignende vil man kunne konkludere at denne modellere sættet godt. Men når man er i

den anden del af spektret bliver man nødt til at have et anderledes sammenligningsgrundlag, for hvordan kan en forskel på 75% og 60% sammenlignes? En god ting man altid kan sammenligne med i klassificeringsproblemer er en model der bare tilknytter alle datapunkter til den klasse der var størst i træningssættet. Altså en majoritetsmodel. Hvis klasserne er nogenlunde lige store vil dette svare til en model der bare gætter på hvilken klasse et datapunkt tilhører. Fejlraten for denne model findes i tabel 5. Dette giver en god kvalitativ måde at fortolke en models effektivitet.
Her ses det meget tydeligt hvordan de to simple modeller ikke har nogen praktisk anvendelse. De regler de har fundet modellere dataet helt skævt så de faktisk har et værre resultat end bare at gætte på den største klasse. Det viser at de er alt for simple til at kunne anvendes og har ikke megen praktisk betydning hvis man skulle lave en større agent til Pac-Man.
Det gør også at det kan være svært at adressere forskellene i kompleksitet for de forskellige AIer da der ikke kan sammenlignes med de simplere modellers svar.

Der kan derimod ses en klar forbedring i forskellene til NN. Her er der i bedste fald en forbedring på ca. 20%, hvilket er godt når det tages i betragtning at selv logistisk regression som ellers minder om NN, ikke har kunne prædikterer på datasættet overhovedet. Hvilket betyder at, hvis valget udelukkende står mellem disse tre modeller, står det klart at NN er at foretrække. Der også ses hvordan forskellen i de 4 AIer er betydelig her i forhold til hvad der har kunnet ses før. Dog er den forventede række følge ikke som forventet da den sværeste faktisk var den der tog den hurtigste vej til PM. Det giver os indblik i hvordan NN har modelleret, nemlig ikke ved forståelse for AIen, men ved at prøve at finde regler og grænser i dataet.

For at opsummerer bør det også siges at disse ekstremt høje fejlrater der findes i tabellerne 1, 2 og 4 er uacceptable i forbindelse med at skabe en agent der skal kunne bruge disse modeller til noget. Det kræver i alt fald at agenten skal kunne arbejde med meget store mængder af støj i prædiktionerne, og så enten skal lave fleksible planer eller opdatere sine planer ualmindelig ofte.

Model	Simpel AI	Medium AI	Svær AI	Meget Svær AI
Majoritets Fejlrate	0.6780	0.7187	0.7013	0.7013
Forskel til Træer	-0,088	-0,0413	-0,0307	-0,0387
Forskel til Logreg	-0,048	-0,0273	-0,0107	-0,0187
Forskel til NN	0,212	-0,0253	0,1213	0,0613

Tabel 5: Her ses fejlraten ved at gætte på den største klasse og forskellen mellem den og de andre modeller. Bemærk at den er ikke er helt ligelig fordelt mellem de fire klasser, da det ville give en fejl på 75%.

5 Konklusion

Målet for mit projekt var at lave og afprøve en del af en agent, der kunne spille Pac-Man. Denne del skulle kunne kortlægge bevægelsesmønstre. Jeg ville sammenligne og vurdere tre modeller; NN, valgtræer og logistisk regression, til dette formål.

Jeg har undersøgt hvilke af de 3 modeller, der kan anvendes til en Pac-Man spillende agent. Her må det klart konkluderes, at NN er den eneste model af de tre, der med fordel kan anvendes. På trods af at denne fordel er meget spinkel.

Jeg har undersøgt, hvilke modeller, der meningsfyldt kan anvendes til de forskellige mønsterkompleksiteter. Det er klart, at besvarelsen af dette spørgsmål bliver en smule ensidigt, da de simple modeller (valgtræer og logistisk regression), mod forventning, ikke kunne kortlægge nogle af mønsterene overhovedet. Jeg vurderer at NN kan anvendes til tre af de implementerede bevægelsesmønstre. Det er dog ikke muligt at generalisere, hvornår NN kan bruges, da effektiviteten af NN er meget varierende over de 4 mønstre. Hvis man ser bort fra den medium svære AI, kan man dog se en klar stigning i sværhedsgrad for NN som mønsterkompleksiteten øges.
Min hypotese om at de simple modeller kunne bruges til simplere mønstre og NN til mere komplekse, er klart forkastet. Selve ideen om at bruge forskellige modeller til forskellig kompleksitet af mønstre behøves dog ikke forkastes. Hvis blot man antager NN som en simpel model, og så ser på endnu mere komplekse modeller end NN.

Endeligt ønskede jeg at lave en vurdering af hvilke af de tre modeller det kan betale sig at anvende til hvilke bevægelsesmønstre. Igen må jeg forkaste de to simple modeller og udelukkende vurdere om NN kan bruges, til at lave en Pac-Man spillende agent.
Her ville jeg klart anbefale ikke at anvende NN til løsning af denne problemstilling. Hvis man ser på mængden af data, der skal til at træne det neurale netværk, i forhold til hvor stor en misklassificeringsrate modellen har, er der et klart misforhold. Hvis man skulle lave en planlæggende agent, der skulle kunne tænke bare en smule frem, ville den konstant skulle genplanlægge fordi NN så ofte giver forkerte forudsigelser, selv for det helt simple bevægelsesmønster.

6 Perspektivering

Da neurale netværk og de simplere modeller ikke kunne få den store anvendelighed til denne problemstilling, ligger det ligefor at spørge, hvordan og med hvilke midler man så kunne løse problemet. For nok fremstiller hypotesen en idé om at simple modeller kan bruges til simple AIer og komplekse til mere komplicerede, som ikke behøves at forkastes hvis man blot ser NN som en simpel model.
Der er mere komplicerede modeller end det neurale netværk, som for eksempel tager højde for tiden, ved at se på sekvenser af datapunkter i stedet for at se på data som enkeltstående begivenheder. Heriblandt kan nævnes *hiddem Markov models* som er meget udbredt til at klassificere fonemer til talegenkendende algoritmer. Denne kunne være yderst interessant at anvende, hvis man ville holde sig til maskinlæringstilgangen.
Man kunne også bruge mere traditionel AI og prøve at opsætte logiske regler for hver datapunkt man havde set. Dette ville dog kræve at man gav PM agenten noget grundlæggende forståelse for, hvordan Pac-Man fungerer, hvilket ikke var udgangspunktet i dette projekt.

En anden tilgang kunne være at forbedre de løsninger som er implementeret. I afsnit 4.3 beskrives der hvordan de forskellige kontrolparametre ikke var helt afpassede, hvilket gav nogle lidt upræcise læringsmønstre. Man ville kunne gå ind i at lave flere og mere gennemtænkte kontrolparametre.
En anden ting der kunne gavne alle modellerne i dette projekt, er at manipulere de inputs de fik. Jeg har valgt at lade dem alle få hele inputtet fra HistPoints. Man kunne guide dem ved kun at give dem de inputs, der var brugbare i analysen af den AI de blev præsenteret for. Dette kunne gøres testbaseret, så modellen bare afprøvede forskellige konstellationer af inputs. Ellers ville det kræve, at man gav modellen viden som man havde fordi man selv kendte den ægte funktion.

Litteratur

[Russel & Norvig, 2010] Stuart Russell og Peter Norvig, **Artificial Intelligens - a modern approach**. Pearson Education, 3. Udgave, 2010.

[R. Rojas, 1996] Raúl Rojas, **Neural Networks - a systematic introduction**. Springer-Verlag, 1996.

[C. M. Bishop, 2006] Christopher M. Bishop, **Pattern Recognition and Machine Learning**. Springer Science, 2006.

[Tan, Steiback & Kumar, 2006] Pang-Ning Tan, Micheal Steinback og Vipin Kumar, **Introduction to Data Mining** Pearson Education, International Udgave, 2006.

[J. Pittman, 2011] Jamey Pittman, **The Pac-Man Dossier** er en udførlig beskrivelse af Pac-Mans historie og systemet bag, http://home.comcast.net/ jpittman2/pacman/pacmandossier.html, Version 1.0.26, 2011.

7 Bilag

A Brættet fra fil

Her betyder 1 at feltet er en del af brættet, mens 0 angiver at det ikke er med. Alle felterne bliver oprettet men kun felter der er en del af brættet har naboer.

```
000000000000000000000000000000
000000000000000000000000000000
000000000000000000000000000000
001111111111110011111111111100
001000010000010010000010000100
001000010000010010000010000100
001111111111111111111111111100
001000010010000000010010000100
001000010010000000010010000100
001111110011110011110011111100
000000010000010010000010000000
000000010000010010000010000000
000000010011111111110010000000
000000010010000000010010000000
000000010010000000010010000000
000000011110000000011110000000
000000010010000000010010000000
000000010010000000010010000000
000000010011111111110010000000
000000010010000000010010000000
000000010010000000010010000000
001111111111110011111111111100
001000010000010010000010000100
001000010000010010000010000100
001110011111111111111110011100
000010010010000000010010010000
000010010010000000010010010000
001111110011110011110011111100
001000000000010010000000000100
001000000000010010000000000100
001111111111111111111111111100
000000000000000000000000000000
000000000000000000000000000000
```