nnRacer.

02366 – Neurale Netværk.
Opgave: **Slut projekt.**

Afleveringsfrist: Mandag den 24/5 - 2004

Hold nr: 8

Holddeltagere:
Studie nr, Efternavn, Fornavn, Underskrift


s022307, Bentsen, Martin Atke _____


s012305, Togeskov, Adrian Peter _____


s022591, Hayeem, Adam Frosch _____

Denne rapport er modtaget af: _____ den ___/___ kl: ___:___
(Udfyldes af instituttet)

Denne opgave indeholder 43 sider inkl. denne side, samt 27 siders bilag.

Forord

Velkommen til denne rapport, der vil gennemgå hvordan vi har forsøgt at skabe den perfekte racerkører i et kunstigt neuralt netværk. Da fokus er på principperne i oplæring af det neurale netværk, har vi implementeret et lillet og simpelt racerbilspil, som det neurale netværk kan trænes i. Det har været meget underholdende at lave, og vi håber at du vil finde det interessant at læse om vores erfaringer.

Det færdige program kan downloades fra internettet, og køres på en windows-maskine. Det vil være en fordel at have OpenGL-drivere installeret, men det er ikke noget krav. Du kan hente programmet fra:

http://nnRacer.atkeland.dk

Alle tre deltagere i projektet har deltaget ligeligt, og ønsker at blive ligeligt vurderet.

Indledning

Den biologiske hjerne giver alle dyr her på kloden evnen til at tage beslutninger. Den består af en stor mængde neuroner der sender signaler til hinanden. Sammenspillet af alle disse signaler resulterer i vores forskellige handlinger. Dette system med et netværk af neuroner er grundlæggende det samme hos små insekter som myrer og støvmider, som hos os mennesker som vi oftest betragter som de klogeste dyr i verden. Og det er det der adskiller vores kreative tankegang fra en computers 'dumme' logik.

Det interessante kommer så af, at man kan simulere det biologiske neurale netværk i en computer, og derved opnå en form for kunstig intelligens i computeren, der minder om den biologiske intelligens.

Computere idag er dog langtfra så kraftige at de kan simulere en komplet menneskehjerne, så vi må nøjes med at lave kunstige hjerner der får meget begrænsede opgaver, for at have computerkraft nok til at løse dem.

Så hvad kan vi så bruge det til

Fordelen ved de biologiske hjerner er at de kan lære af deres egne erfaringer, hvorimod en computer kun kan tage de beslutninger som den er programmeret til. Nogen problemer kan være nemme at løse ud fra sine erfaringer, men være svære at finde logikken i, så den kan programmeres i en computer. Og det er i disse tilfælde at et kunstigt neuralt netværk måske kan være en løsning.

Når jeg siger at et neuralt netværk kun måske er en løsning, er det fordi at den kunstige intelligens også har arvet en vis uforudsigelighed fra den biologiske intelligens, så det er ikke altid man får hvad man havde forventet.

Endelig er der det tiltalende perspektiv i det for den dovne programmør, at man ikke selv skal programmere selve kernen af et kompliceret system, men kun træne et neuralt netværk til at løse problemet. Erfaringerne fra dette projekt, siger os dog at det nemt bliver endnu mere besværligt at træne netværket op perfekt, end det ville være bare at løse problemet selv fra starten af.

Opgaven

Ideen med dette projekt, er at lade kunstige intelligenser konkurrere imod hinanden, og imod den menneskelige intelligens. Disciplinen er valgt til racerløb. En simpel konkurrence hvor det bare handler om at komme først. Da det er intelligensen det drejer sig om, få alle ens biler med samme køreegenskaber (topfart og vejgreb).

Det neurale netværk der skal køre bilen er valgt til et feedforward netværk. Dimensionering af antal inputs og antal neuroner i skjulte lag vil vi optimere ved at forsøge os frem. Selve indlæring implementeres på tre forskellige måder:

Adaline:
: Man optager reaktionerne fra en kører der kører banen korrekt igennem. Denne optagelse bliver træningsdata til at træne et netværk ud fra den simple Adaline-algoritme, der retter vægtene i netværket et lille smule ind for hvert træningsdata-sæt.

Back-Propagation:
: Ligesom ved Adaline skal reaktionerne fra en kører der kører banen korrekt igennem optages. Denne optagelse bliver træningsdata til at træne netværket ud fra back-propagation algoritmen, der udmærker sig ved bedre at justere vægtene når der er skjulte lag.

Evolution:
: Her trænes et enkelt netværk ikke. Derimod oprettes en stor mængde netværk med tilfældigt satte vægte. De bedste udvælges og formere sig mens de dårlige uddør. Dette gentages indtil man har fundet et netværk der tilfældigvis kører godt.

Problemformulering

- Lav en styring af en bil i et bilspil med neurale netværk.
- Hvor godt kan systemet styre bilen.
- Kan netværket bliver konkurrencedygtigt imod en menneskelig spiller.

- Hvilken type indlæring er bedst til opgaven.
- Hvilken dimensionering af netværk er bedst til opgaven.

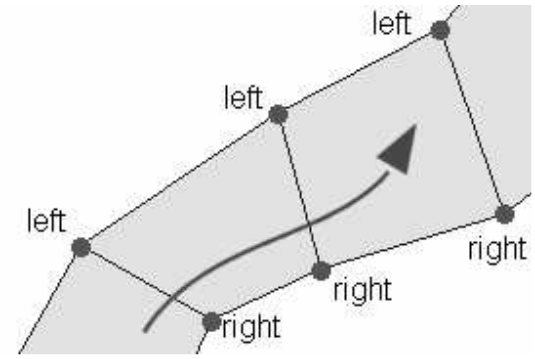
Spillet

Spillet er implementeret i C++, og benytter sig af OpenGL til at lave den grafiske præsentation. Grafikkens holdes i 2D, og man ser banen oppefra. Alle punkter i spillet er således repræsenteret i et 2D koordinatsystem med (0,0) i midten, er 16 bredt, og 12 højt. Således er (-8,-6) i nederste venstre hjørne.

Der vil her blive gennemgået de vigtigeste aspekter af spillet, for at give et overblik over implementationen. Dette er ikke ment som en komplet beskrivelse af program-koden, idet mange detaljer ikke har relevans for indlæringen af de neurale netværk. Det er dog vigtigt at forstå hvordan verden er bygget op.

Banen (Track)

Banen er repræsenteret af en række punkt-par der specificerer venstre og højre side af banen. Det giver et række firkantede felter som man kører på. Ved at holde styr på hvilket felt man er på, ved man hele tiden hvilken retning man skal for at komme videre ind i det næste felt. For at kunne se om bilerne ved hvilket felt de er på, er banen implementeret således at feltet under bilen får en lidt lysere farve.



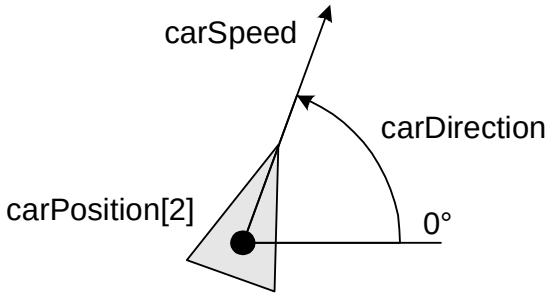
Banens punkter gemmes i et array af struct'er bestående af left og right som begge er float-arrays, hvor index 0 er x-værdien og index 1 er y-værdien. Fidusen ved at have x og y i et lillet array er at det er nemme at videre give til OpenGL når banen skal tegnes på skærmen.

```
struct TrackLine    //ligger i helpers.h
{
        float left[2];
        float right[2];
};
```

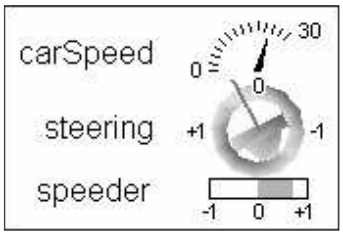
Banen er implementeret i Track.cpp, og giver desuden mulighed for at kunne gemme bane-punkterne i en tekstfil, og indlæse dem igen, så man kan have flere forskellige baner. Der er desuden en simpel bane-editor, så man nemt kan lave nye baner.

Bilen (Car)

Bilen repræsenteres af et enkelt punkt, en reting og en hastighed. Selvom den tegnes som en lille trekant, er det kun midterpunktet der bruges til at beregne hvor på banen bilen er placeret. Retningen er en vinkel mellem 0 og 360 grader, hvor retning 0 er stik øst og retning 90 er stik nord, som den matematiske gradskala. Dette valg af grader fremfor kompassets med 0 grader stik nord og 90 grader stik øst, er at det matematiske passer direkte ind i trigonometriens formler som vi benytter til at flytte rundt på bilens position. Valget af 360 grader frem for 2 &PI (radianer) er at OpenGL bruger 360 grader. Hastigheden er et tal mellem 0 og 30. Kan ændres hvis man vil have anderledes topfart.



Bilen styres via rat og pedaler. Rattet er repræsenteret ved en float hvor 0 er ligeud, +1.0 er fuldt ud til venstre og -1.0 er fuldt ud til højre. Pedalerne repræsenteres af en enkelt float, hvor -1.0 er fuld opbremsning, og +1.0 er speederen i bund. Det er naturligvis muligt at dreje lidt på rettet og træde lidt på pedalerne, men det får man ikke den store gavn af når man styrer med tastatur. De neurale netværk bør dog kunne bruge det. Bilens styring kan se i nederste højre hjørne af skærmbilledet, så man kan se hvordan computeren styrer.

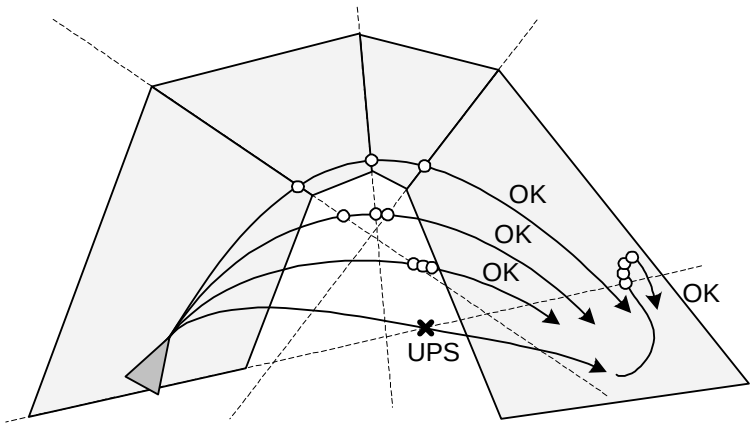


Bilen flyttes rundt på banen ved at der er sat en timer op i OpenGL der hvert 40 millisekund (25 Hz) flytter bilen, op opdaterer skærmen. Bilen flyttes med funktionen Car::move(...) og tegnes med Car::drawCar(...). Disse to er adskilt for at kunne køre med bilen uden at skulle tegne den på skærmen, hvilket der er god brug for i evolutions-indlæringen, hvor bilerne skal testkøres for at finde en vinder. Der er ingen grund til at spilde ressourcer på at vise disse mange testkørsler.

move(...) er altså hvor bilens opførsel er defineret. Det er her carPosition, carDirection og carSpeed opdateres efter hvordan steering og speeder er sat af køreren. For at give lidt øget realisme, mister bilen fart af at man drejer skarpt, og man kan ligeledes ikke dreje så

skarp med speederen i bund, som hvis man bremser. Måden man drejer på er dog mere som et rumskib end en bil, idet man kan dreje lige meget i forhold til tid uanset hvor hurtigt man kører. Man kan således også dreje mens bilen står stille. Der kunne optimeres på styringen så den blev mere realistisk, men da computeren ikke ved hvad der er realistisk for en bil, spiller det ikke den store rolle for indlæringen.

Det er muligt at køre udenfor banens felter. Det giver dog en stor straf i hastigheden. Men ud over det giver det en måde at skære hjørner meget af, for man kan godt komme til næste felt uden rent faktisk at komme ind på det. Om man er kommet forbi næste linie beregnes med linies ligning, så linien går ud over sine punkter. Herunder er illustreret fire måder at skære igennem et sving på.

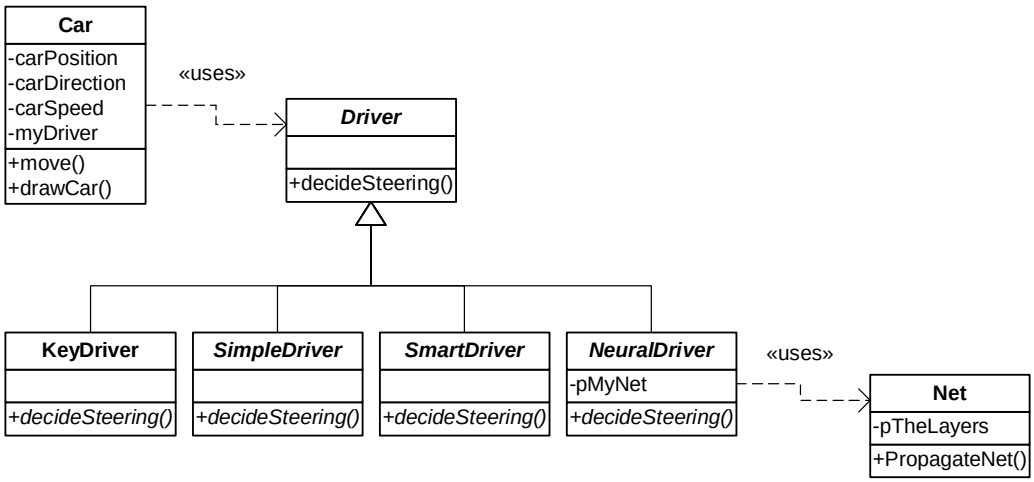


Den øverste er oplagt iorden. Nummer to er også iorden, når man ved at linierne strækker sig ud over banens punkter. Den tredje er lidt mere kryptisk, for linierne passeres i den forkerte rækkefølge. Men efter den første er passeret, er de næste to jo også passeret, og det vil blive registreret de efterfølgende to game-cycles. I den nederste rute passeres dog den bagvedliggende linie først, og man rykker derfor et felt tilbage. Man skal derfor op og forbi linien igen for at kunne komme videre.

Vi har udeladt enhver form for collision-detection imellem bilerne. Vi regnede ikke med at de neurale netværk ville have nogen chance for at lære at undgå sammenstød, så dette er afgrænset. Bilerne kan derfor aldrig se hinanden, og vil altid opfatte sig selv som eneste bil på banen.

Kører (Driver)

Der findes kun een slags bil, men forskellige racerkørere som kan sættes i bilen. Hver gang Car::move(...) kaldes, kalder den Driver::decideSteering(...), og racerkøreren sætter steering og speeder ud fra den aktuelle situation. Det er denne racerkører de er lavet i forskellige versione, så man både kan styre med taster og lade computeren styre.



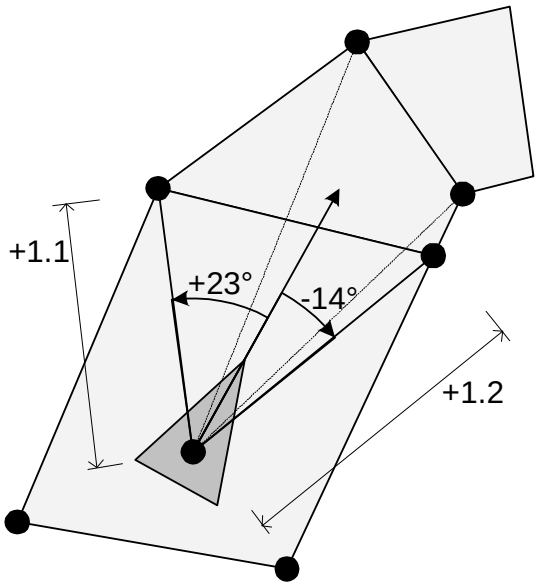
KeyDriver styrer ud fra tasterne. Der er to af disse, en til piletasterne, og en til tasterne A, S, D og W, så to personer kan tage en dyst. Denne Driver bruges til at sikre at bilens køreegenskaber opfører sig som man forventer Den kan desuden bruges til at optage en menneskelig kørsel med, som kan bruges til indlæring af sit netværk med Adaline eller back-propagation.

SimpleDriver og SmartDriver er hard-codede beslutningsmønstre, som er lavet for at se om et neuralt netværk kan slå en bil der kører efter et sæt regler specificeret af en programmør. SimpleDriver gør ikke andet end at rette sin retning ind, efter midterpunktet af den næste tværliggende linie på banen. SmartDriver derimod kigger så langt frem den kan se et ret linie der ikke kommer udenfor banen pga sving. Det har vist sig at SimpleDriver er relativt nem at slå med et neuralt netværk, og SmartDriver kan i enkelte tilfælde slås. Som menneske med en KeyDriver skal man koncentrere sig for at slå SmartDriveren.

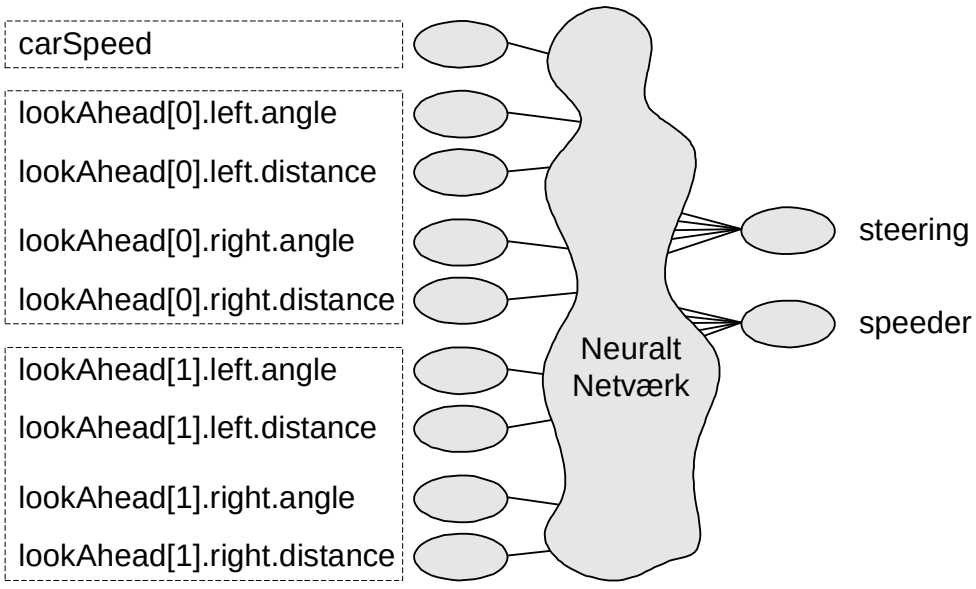
NeuralDriver'en er den interessante for det neurale netværk. Men den er for så vidt ganske simpel, og det er den samme der bruges til alle vores former for indlæring. Den består kun af pre-processering og post-processering for netværket. Selve netværket er implementeret i sin egen klasse.

Pre-processering

Som input får netværket bilens nuværende hastighed, og retning og afstand til de næste punkter på vejen i forhold til bilen selv. Alle vinkler er i forhold til bilens egen. Dvs, hvis et punkt er lige fremme foran bilen er vinklen 0 grader - og er den stik til venstre for at vinklen +90 grader. Alle vinkler skaleres ned, så +180 grader er +1.0 og -180 grader er -1.0. Netværket ved således ikke om den kører opad på skærmen eller nedad - den ser kun punkter foran sig. Afstanden skaleres således at 5 i koordinatsystemet (skærmbilledets dimensioner er 16x12) giver et input på +1.0. Er punkterne længere væk ses de stadig som værende 5 væk. En afstand kan ikke være negativ.



Man kan angive hvor langt der skal ses frem, og altså hvor mange punkter frem der skal beregnes afstande og vinkler til. Man kunne have forsøgt at udlade afstandene og se om bilen stadig kan finde rundt, men det fik vi ikke gjort. Der er altså 1 input til den aktuelle hastighed, og 4 inputs til hvert felt man vil se frem.



Post-processering

Outputtet fra netværket er stilling for rattet, og speeder/bremse. De kommer ud fra netværket som floats mellem -1.0 og +1.0 . Disse returneres direkte tilbage til bilens move-funktion som skal bruge værdierne netop indenfor disse intervaller.

Opsætning (Main)

Vi har fået implementeret en popup-menu der kommer frem når man trykker på højre musetast i spillet. Herfra kan man vælge hvilke kørere der skal køre de enkelte biler, og man kan skifte banen ud. Vil man ændre på eksempelvis netværkets dimensioner, skal man dog ind i source-koden og rette nogle værdier.

Alt starter naturligvis i main(...). Herfra opsættes banen, spillerne, grafikken og den interaktive del (menu og keys) i hver init-funktion. Derefter startes OpenGL's main-loop. En timer sørger for med minimum 40 millisekunders mellemrum at kalde en funktion der opdaterer bilernes position og opdaterer skærmbilledet.

Init-funktionerne:

initWorld()
: Her oprettes banen. Der er hardcoded koordinater til en enkelt bane i exe-filen, og det er den der initialiseres med "`theTrack = new Track();`". Vil man starte på en anden bane (for at træne sin bil) kan man her loade en ny med "`theTrack->loadFromFile(7);`", hvor 7-tallet kan skiftes ud med det nummer fil fra tracks-mappen man vil loade. De numereres ud fra windows alfabetiske sortering af filnavnet.

initPlayers()
: Her oprettes de biler man vi have med i løbet. Der er en hjælpe-funktion (`addNewCarToGame(Driver*)`), der sørger for korrekt at tilføje den nye bil i array'et af biler. Det er her man bør specificere parametrene for hvordan man vil træne et netværk ved opstart af spillet, hvilket er en fordel mens man udvikler på indlæringsalgoritmerne. Grænsen for hvor mange biler der kan være i spillet er sat som en konstant i "Constants.h".

timer(...)
: Denne funktion er selve game-loopet. Den bliver kaldt af en timer i OpenGL med min. 40 millisekunders mellemrum, hvilket er hutigt nok til give en illusion af at spillet kører flydende. Det eneste den gør er at fortælle alle bilerne at de skal flytte sig en tand, og fortælle OpenGL at grafikken skal opdateres. Grafikken opdateres ved at OpenGL kalder display() som gentegner samtlige elementer på skærmbilledet igen.

Hot-keys i spillet.

Ud over menuen der aktiveres på højre musetast er der specificeret en række taster på tastaturet. Det er værd at bemærke at ikke alle disse er med i alle versioner af programmet. De er specificeret i `InputHandlers.cpp`

Esc
: Lukker spillet.

F11
: Skifter mellem fullscreen og windowed mode.

Piletasterne
: Styring af KeyDriver One

a, s, d og w
: Styring af KeyDriver Two

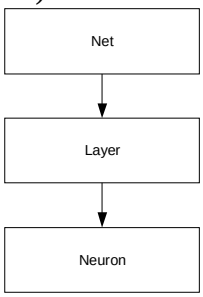
1, 2, 3, 4, 5, 6, 7, 8, 9
: Skifter for hvad der skal vises for de enkelte biler. Der kan vises et spor af hvor bilen har været de sidste 1000 game-ticks, og der kan vises streger hen til de punkter som bilens kører bruger til at bestemme sig ud fra. Det er dog ikke til at overskue hvis man viser disse ting for mere end en enkelt bil af gangen.

Insert
: Laver et lille jordskælv og flytter alle biller et lille stykke i hver sin tilfældige retning. Er meget anvendelig hvis en bil har sat sig fast i en situation den ikke kan komme ud af. Eller hvis flere biler kører nøjagtig oveni hinanden, kan man skille dem ad med denne knap.

Page Up / Page Down
: Zoomer henholdsvist ud og ind. Nyttigt hvis en bil kører udenfor det synlige område, så kan man zoome ud og se hvor den kører hen.

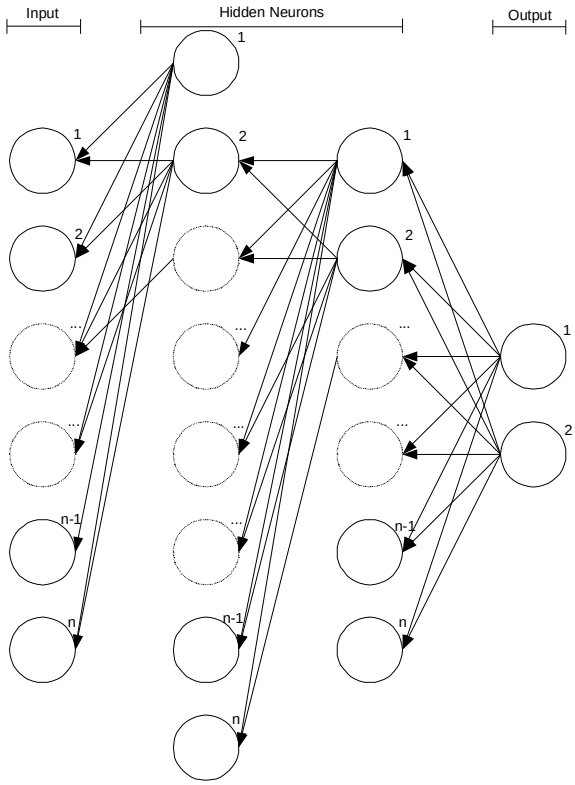
Neurale net generelt

Selve den basale struktur for alle 3 typer netværk vi har implementeret er bygget op af den samme struktur (se Bilag 2, sige 51)



Det er en kode som er skrevet i C++ og er objektorienteret. Klassen Net er selve netværket med alle de funktioner som skal tilgåes fra omverdenen, klassen Layer er som ordet siger, selve lagene i netværket, og til sidst klassen Neuron er selve neuronerne i netværket. Begge af disse klasser er skjult fra omverdenen.
Grunden til at vi valgte at lave vores eget netværk fra bunden af var fordi vi mente at rent struktur mæssigt og forståelsesmæssigt ville en objektorienteret model være mere passende, da man kan indkapsle og definere de enkelte dele for sig.

Strukturen på netværkerne ser således ud:



Vi har valgt at se vægtene som gående fra udgangs neuron til indgangs neuron, hvilket er nemmere at programmere.

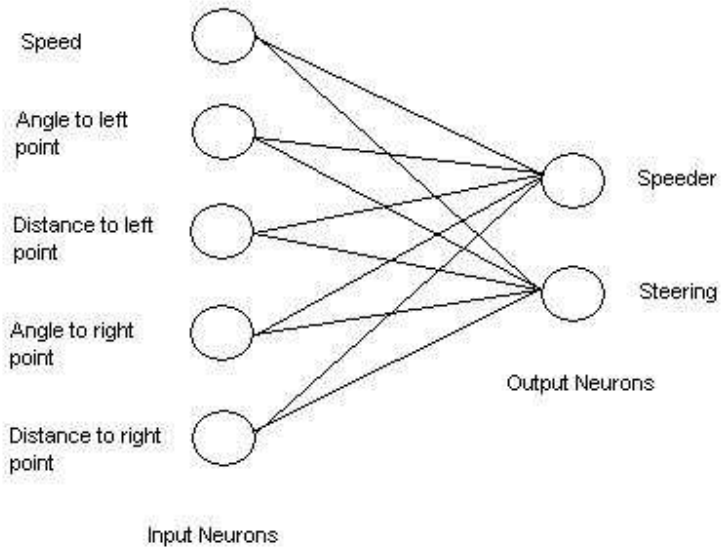
Som det kan ses i koden så er det et meget alsidigt netværk da man frit kan definere et hvilket som helst antal indgange til netværket, et antal (eller ikke nogen) skjulte lag og antal neuroner i disse lag og til sidst hvor mange udgange der skal være. Dog skal der for at kunne bruges i vores spil, altid være 2 output neurons og 5-9-13-15 osv. input neurons. Grunden til disse restriktioner er for at spillet skal have at vide hvilken retning det skal dreje bilen og hvor meget speeder bilen skal have. Netværket skal kende hastigheden på bilen, vinkelen mellem bilen og de to punkter foran den og afstanden til de to punkter foran bilen dvs. 1 til hastighed 2 til vinkler og 2 til afstande = 5 input og hvis netværket skal kunne se længere foran sig end bare til de første punkter skal der bruges yderligere 4 input for hver gang netværkets ”synsfelt” bliver forstørret.

Alle netværkerne har samme algoritme til at propagate netværket igennem, gemme og loade et netværk, men ellers er der blevet implementeret forskellige funktioner til at kunne beregne fejl og ikke mindst rette vægte i netværket når det skal indlæres.

Adaline Net Indledning

Det første, og mest simple af netværkerne som skal lære at køre bilen er adaline netværket. Da adaline netværket er et simpelt klassifikations netværk, som kræver predefineret test-data og resultater for at lære, er det vigtigt at klassificere hvordan bilen skal køre under givne omstændigheder. Dette gøres ved at lade spilleren tage en test kørsel med en varighed af nogle bane omgange, hvor data omkring kørselen bliver gemt. Efter disse iterationerne er det op til adaline netværket, at lære, hvordan man kører ud fra test dataen. Som input til adaline netværket for netværket kendskab til bilens vinkel til vejens nærmeste punkter, bilens afstand til disse punkter, og bilens hastighed. Bilens vinkel og afstand til nærmeste punkter hjælper netværket med at placerer bilen, mens hastigheden fortæller netværket hvor hurtigt bilen kører. Ved hjælp af disse data er det op til netværket at udregn den korrekte speeder værdi samt retning, dvs "speeder" og "steering". Disse informationer er vigtigt at gemme under test kørslen da netværket skal bruge dem til sin indlæring.

Skitse over netværket:



Algoritmen i Pseudokode:

```
while (errorInOutput > stopError)
{
	Indlæs_data_fra_fil();
	while (der stadig er noget at indlæse fra filen)
	{
		propagate_network();
		get_output_of_network();
		findErrorInOutput();
		//delta regel
		vægt += deltafejl * learning rate * output;
	}
}
```

Algoritmen foroven kører så længe deltafejlen (forskellen mellem ønskede output og output) er større end ønskede stopfejl. Alt test data løbes igennem hver gang så netværket lærer hvad den skal gøre i alle de mulige situationer. Indlæringen sker ved hjælp af delta reglen.

Implementationen

Da netværket lærer udfra bil nummer første kørsel er det vigtigt at den kører godt hvis man forventer det samme af netværket. Knappen “r” på keyboardet er reserveret til at man kan optage de næste 500 spil iterationer. Optagelsen sker ved at der bliver oprettet og skrevet til en fil som hedder “carDump.txt”. Trykker man på tasten “r” sættes en tæller til 500 i car.cpp filen. Da car::move funktionen kontrollere om denne tæller er over nul, ved den om der skal oprettes en tekst fil til output. Sker dette, sørger move funktionen også for at recordCar function bliver kaldt som udskriver data til filen. Koden ser således ud:

```
switch (key)
	{
		case 'R' :
		case 'r' : if(theCars[0] != NULL) theCars[0]-
	>recordDecisions("carDump.txt", 500);		break;
	}
```

Denne kode findes i input handlers filen og kalder recordDecisions funktionen så snart “r” er trykket. Argumenterne består af fil navnet og antal game ticks som skal optages til filen.

recordDecisions(char *fileName, int numberOfGameTicksToRecord)

recordDecisions findes i Car.cpp og den åbne filen for at udskrive til, og sætter en outCountDown parameter (som starter med at være -1) til antallet af game ticks.

```
if(outCountDown > 0)  // save recorded steering to file, for training
Adaline
{
	recordCar(theTrack, currentTrackField);
```

```
	outCountDown--;
}
else if(outCountDown == 0)
{
	outfile.close();
	outCountDown--;
	cout<<"Output file closed"<<endl;
}
```

Denne kode befinder sig i move funktionen i car.cpp hvor alt bevægelse af bilen bestemmes. Det er vigtigt at koden findes her da bevægelses dataen udregnes og redigeres her.

Car::recordCar(Track *theTrack, int nextLine)

recordCar funktionen udskriver til filen i en sådan orden at bilens nuværende hastighed udskrives først efterfuldt af vinklen og afstanden til venstre punkt af vejen, herefter udskrives vinklen og afstanden til højre punkt af vejen, og endeligt den korrekte speeder og steering af bilen som bruges til supervised learning algoritmen.

Fintuning af parametre

Mindre eller Mere test data:
Mængden af trænings data kan justeres via InputHandlers.cpp. Funktionen “keyboard” kalder, ved trykket på “r”, funktionen recordDecisions som tager antallet af game ticks med som argument.

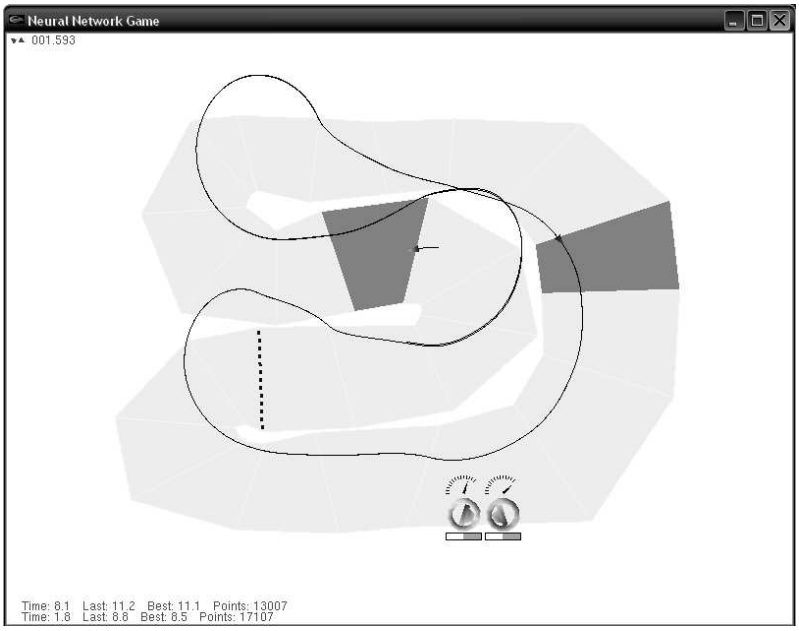
Learning rate:
Learning rate justeres ved hjælp af learnRate variablen som findes i Net_AdaXtras.cpp. LearnRate er konstant i gennem trænings forløbet, og adaline algoritmens performance er baseret meget på learning raten. Hvis LearnRate er for stor er der chance for at algorithmen oscillire, og hvis LearnRate er for lille, tager det for længe for algoritmen at convergere.

Max Error Rate:
Max Error Rate er også gemt som en variabel i Net_AdaXtras.cpp, og den hedder “errorMax”. Ved at justere på denne varibel juster man på hvor meget netværket kan generaliser. Jo højere errorMax er, jo bedre er netværket til at generaliser.

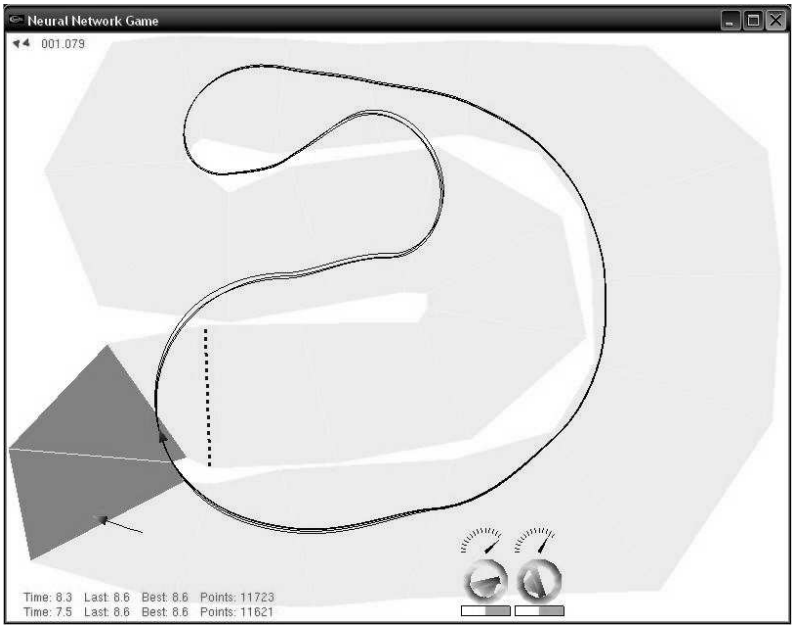
LookAheadLength:
Når vi som mennesker kører bilen i spillet, kigger vi oftest længere fremme på banen end bilen egentligt er nået til, så vi bedre kan styre bilen. LookAheadLength bestemmer antallet af vej dele som bilen skal kunne se frem, og er med til at forøge inputtet.

LookAheadLength justeres via recordCar() i Car_AdaXtras.cpp hvor LookAheadLength sættes til den ønskede værdi og i layerDefinition[2] (husk at for hver extra LookAheadLength skal der 4 extra input neuroner til) som findes i initPlayers funktionen

i main.cpp, og endelig i CreateAdalineNet hvor længden bliver sendt som sidste parameter til TrainAdaNet() funktionen.



Ved LookAheadLength af 1.



Ved LookAheadLength af 2.

Konklusion

Adaline netværket er det mest simple af alle netværkene og selvom dette er tilfældet ses det tydeligt at den sagtens kan få bilen til at køre forholdsvis pænt på banen (og til tider endda slå simpel driveren). Bilen kører bedre og bedre salt efter hvor langt den må kigge frem og hvor meget man fin justere sin learning rate. Adaline netværket er meget afhængig af banens opbyggelse, og dataen som den bruger til indlæringen. Netværket er i stand til at køre på alle baner, hvilket tyder på at netværket ikke er blevet overindlært. Adaline netværket har sværere ved at slå simpel driveren i forhold til de andre type netværk, og dette kan måske skyldes den/de manglende hidden neuroner son de andre netværk indeholder. For få hidden neurons giver en tendens for at netværket har for lidt brain at arbejde med, mens for mange har en tendens til at netværket huske i stedet for at lærer.

Backpropagation indledning

Vi har valgt at benytte BackPropagete netværket da vi har set eksempler på at det er meget velegnet til at kunne ”forudsige” hvilket output det skal komme med efter det har været trænet med nogle forudgående inputs. Denne egenskab er meget velegnet i vores bil spil da netværket skal kunne forudsige hvilke outputs det skal give til nye inputs. Grunden til vi kan være sikker på at det er nye inputs der vil komme også selvom det netværket er blevet trænet på det selv samme track er fordi der skal utroligt lille afvigelse af bilens test kørsel før input vil blive noget ”helt nyt”.

Teori

En kort beskrivelse af anvendelsen af backpropagations algoritmen, som består af to faser.

1. Det sættes et mønster på indgangen, i vores tilfælde vil dette være hastighed og de vinkler og afstande til de felter som ligger foran bilen. Signalerne bevæger sig igennem netværket til udgangen, som giver værdier til styring af rattet og speederen i bilen. Output værdierne bliver sammenlignet med de værdier de skulle have været og en fejl bliver beregnet på basis af dette.
2. I denne fase bliver fejlene tilbage ført igennem netværket fra udgang til indgang, igennem alle neuronerne, og vægt ændringerne bliver beregnet.

Netværkets algoritmer

Her ses de netværks algoritmer vi benytter os af i netværket. Disse algoritmer er taget fra Backpropagation teorien.

Indlærings algoritme:

Find fejlen:
```
Err = Target[i-1]-Out;
```
Først findes fejlen på udgang neuronen

```
For each layer
      For each node{
            Err += Upper->Weight[j][i] * Upper->Error[j];
      }
      Lower->Error[i] = Net->Gain * Out * (1-Out) * Err;
}
```
Herefter tilbageføres fejlen, ved at dele netværket op i forskellige lag. Det øvere lag påvirke det nedre lag. Der tages hensyn til den sinoide overføringsfuntion ved at gange med (1-Out).

Find vægtændringen ud fra fejlen:
```
Net->Layer[l]->dWeight[i][j] = Net->Eta * Err * Out;
```
Delta vægt = lærning rate * error * outputtet fra det foregående lag

Juster vægte:
```
Net->Layer[l]->Weight[i][j] += Net->Eta * Err * Out + Net->Alpha *
dWeight;
```

Den nye vægt beregnes ved hjælp af lærning rate * error * outputtet fra det foregående lag (eller input til dette lag) og momentum tillægges da lærning rate kan være så stor at der oscilleres.

Simulerings algoritme:
```
For each Layer{
      For each Node{
            Sum += Upper->Weight[i][j] * Lower->Output[j];
      }
      Upper->Output[i] = 1 / (1 + exp(-Net->Gain * Sum));
}
Output = sum af vægte * aktivitet + Bias (som findes i output[0])
```

Beskrivelse

Koden

Vi har implementeret selve indlærings algoritmerne fra vores øvelse i backpropagations netværk under semesteret, samt enkelte hjælpe funktioner. De ligger i Net.cpp (se bilag ”Bilag 4, Backpropagation Source-code” side 56).
Yderligere så ligger forskellige algoritmer, som binder selve spillet til netværket i SimpleDriver.cpp (se bilag ”Kode til Backpropagation” side 56) og NeuralDriver.cpp (se bilag ” Bilag 4, Backpropagation Source-code” side 56).

Selve mekanikken til hvornår der skal stoppes for indlæringen har vi været nødt til at prøve os frem til, da vi fandt ud af at det ikke gav så meget mening at stoppe en indlæring ved at kigge på hvornår at fejlen på netværkets output var under en vis tærskel.
Vi skulle derimod hellere have kigget på et længere forløb og have undersøgt hvordan netværket havde opført sig. Grunden til dette er fordi at der ikke findes en bestemt måde at kører korrekt på. At Kører korrekt eller retter sagt pænt skal ses meget mere subjektivt end først antaget. Med andre ord så skulle vi have opstillet en algoritme som kunne tage højde hvor hvornår netværket kører pænt og på de kritierer have stoppet indlæringen.

Test

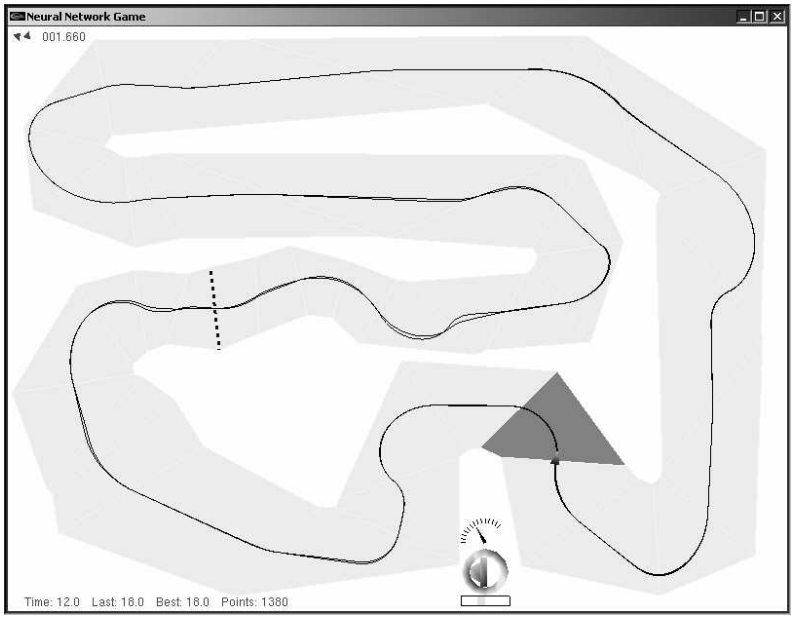
Igennem alle test er måden hvorpå netværket bliver indlært den sammen bortset fra at antallet af data i et dataset afhænger af hvor langt netværket ser frem.

Et dataset består af Speed (bilens hastighed), vinkelerne mellem de to punkter, som er en del af banen, foran bilen, afstandene til punkterne foran bilen og til sidst hvordan ”lærer mesteren” reagerede, hvor meget speeder, steering der bliver givet bilen. Ialt 7 værdier i det dataset når man ser et felt foran sig, for hver gang der bliver tilføjet til felt til bilens synsfelt vil datasettet stige i størrelse med 4 for hvert flet.

I alle tests bliver netværket præsenteret for 300 datasæt under dens indlæring. Datasættene bliver løbet igennem kun en gang, da dette har vist sig give det bedre resultat, det har endog vist sig at netværket har været ude i stand til at kunne kører hvis det blev indlært flere gang.

Yderligere er data’erne i datasættene blevet normaliseret så alle data’erne ligger mellem -1 til 1. Men data’erne er ikke blevet normaliseret i forhold til hinanden med derimod med fast satte værdier såsom speed som vil løbe mellem 0-30 er blevet divideret med 30 dvs. har værdierne mellem 0-1.

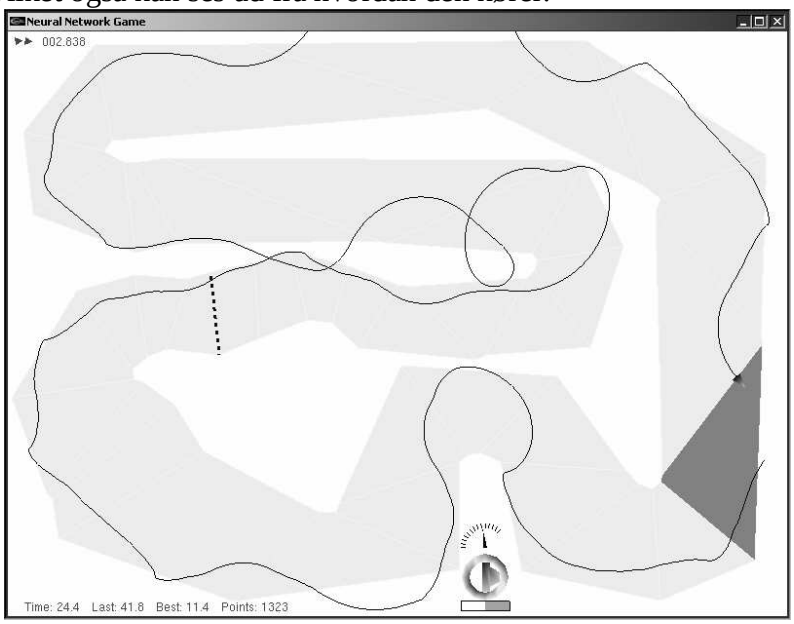
Vi har valgt at det skulle være ”Simple Driver” som skulle være den der lærer netværkene op af forskellige grunde. Nr1, så vil simple driver give test data som vil være stort set være ens for alle netværk. Nr2, vi ville gerne have en lærer, hvor det ville være nemt at se om netværket var i stand til at gøre det bedre. Nr3, ”Simple Driver” er den der kører mest pænt i den forstand at den aldrig vil prøve at skærer hjørner af eller lægge sig meget tæt på siderne på banen, hvilket har vist sig være en positiv egenskab da på grund af usikkerheden i netværkene ellers ville have en tendens til at køre ud af banen.
Her under ses hvordan ”Simple Driver” kører test banen igennem. Som det ses er der tydeligt mulighed for forbedring.



Den sorte linie viser hvordan "Simple Driver" har kørt over et par omgange

Test ser 1 foran

I den første test har vi valgt at lade netværket se et felt foran sig. Dvs. det har ingen mulighed for at kunne bestemme om et givent felt er et sving eller en del af den lige strækning, hvilket også kan ses ud fra hvordan den kører.



Netværket hvor det ser 1 felt foran sig

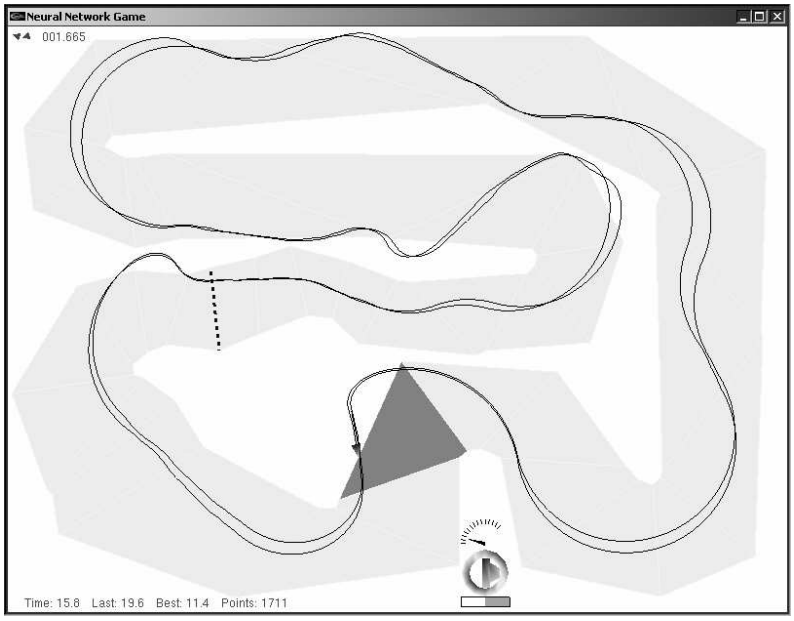
Det ses tydeligt i toppen, at det meget skæve felt bevirker at netværket tror at det er ved at skal i gennem et meget skarpt sving, hvorfor det så styre lige ud af banen.

Man kan se en krølle umiddelbart efter start, denne forekommer fordi at felterne ligger meget tæt på hinanden i skæve vinkler, og fordi at netværket kører ud af banen betyder det at netværket kører på tværs af felterne, og netværket kommer til at køre rundt om sig selv en enkelt gang.

På trods af de synlige fejl, så er netværket trods alt i stand til at kunne navigerer gennem banen uden at blive helt forvirret og gå i stå.

Test ser 2 foran

Her har netværket fået lov til at se 2 felter frem. Dvs. det er også blevet trænet med testdata som indeholder de to felter foran det.

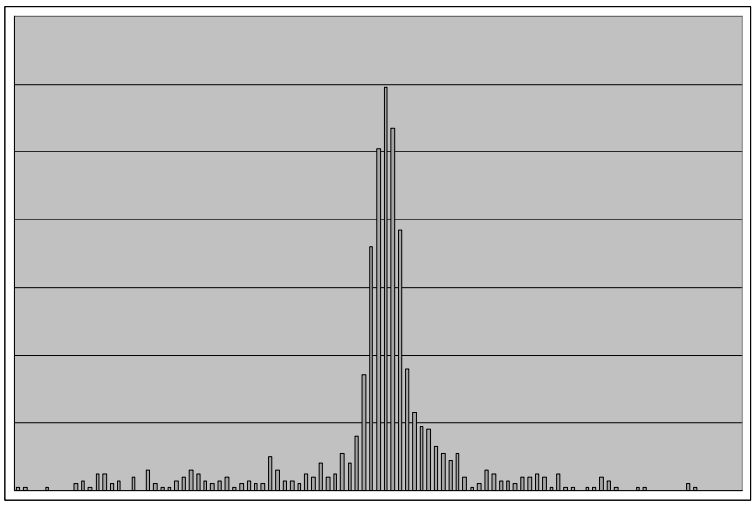


Netværket hvor det ser 2 felter foran sig

Som det kan ses i forhold til hvor det kun så et felt foran sig, er netværket blevet meget bedre til at kunne navigerer banen igennem. Det ses dog at det stadigvæk har problemer hvor det tidligere netværk også havde problemer, men de er blevet væsentlige mindre.

Tiden for et gennemløb er også blevet reduceret fra de ca. 40 sekunder til ca. 20 sekunder for en omgang.

Herunder ses fordelingen af vægtene i netværket.



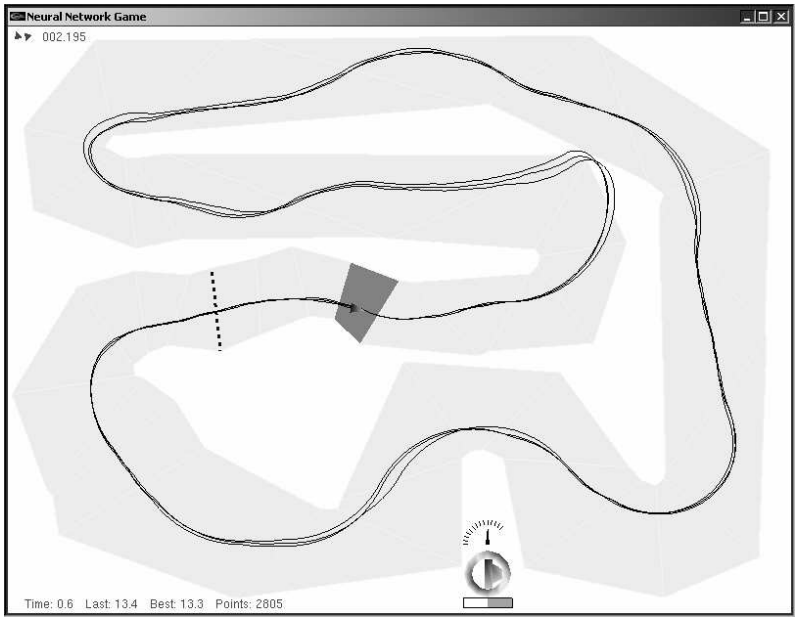
Fordeling af vægte

Ad y-akslen ses antal af neuroner og ad x-akslen ses vægt værdien. Man kan således se hvor meget neuroner har en given vægt. Vægt med værdien 0 ses i midten af spidsen.

Som det kan ses er det en meget spids kurve, som vægt fordelingen giver os. Dette indikerer at vi nok har over indlært netværket, hvilket igen betyder at det vil få svært ved at generaliser. Det ville nok have være mere fordelagtigt at få kurven til at være lidt mindre spids, med andre ord at vægtene var spredt over et større værdi område.

Test ser 3 foran

I denne test ses det tydeligt at netværket kører meget bedre end den ”Simple Driver” som har oplært det. ”Simple driver” kører banen på 18.0 sekunder mens netværket har sin hurtigste omgang på 13.3 sekunder, hvilket er en markant forbedring.
Det ses også tydeligt at netværket har forbedret sig betydeligt i forhold til netværket som kun så 2 felter foran sig.

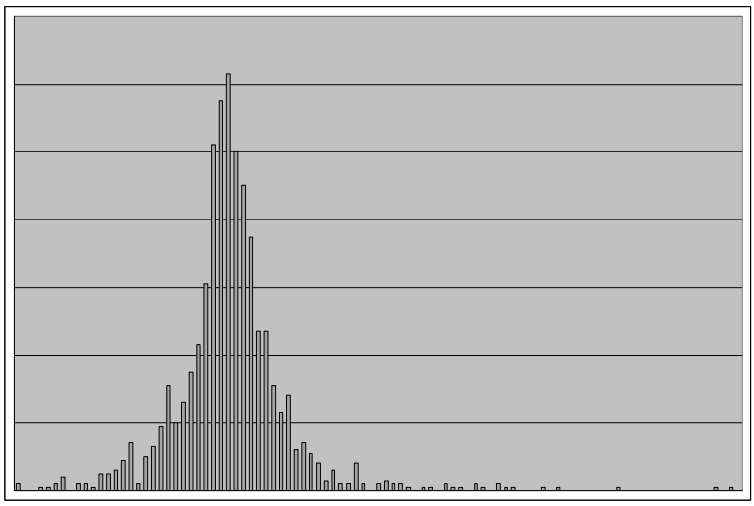


Netværket hvor det ser 3frem foran sig

Yderligere ses det at netværket her viser en evne til at kunne bremse i skarpe sving, hvilket kan ses tydeligt i svinget i øverste venstre hjørne.
Dette har netværket ellers ikke har vist nogen særlig tendens til. Netværket tager også generelt kurverne meget bedre en "Simple Driver" kan.
Dog kan det ses at netværket har problemer med det første sving efter start, hvor det stort set altid kører uden for banen. Svinget i toppen viser netværket også en tendens til at ville køre uden for banen, men i denne version holder det sig dog inde for banens sider i forhold til de andre versioner.

Stadigvæk kan man se at netværket tror at skæve felter er sving, hvilket kan ses ved at det svinger fra side til side som om det skal til at lave et sving på disse skæve felter, dog skal det siges at det ikke kører uden for banen disse steder.

Herunder ses fordelingen af vægtene i netværket.



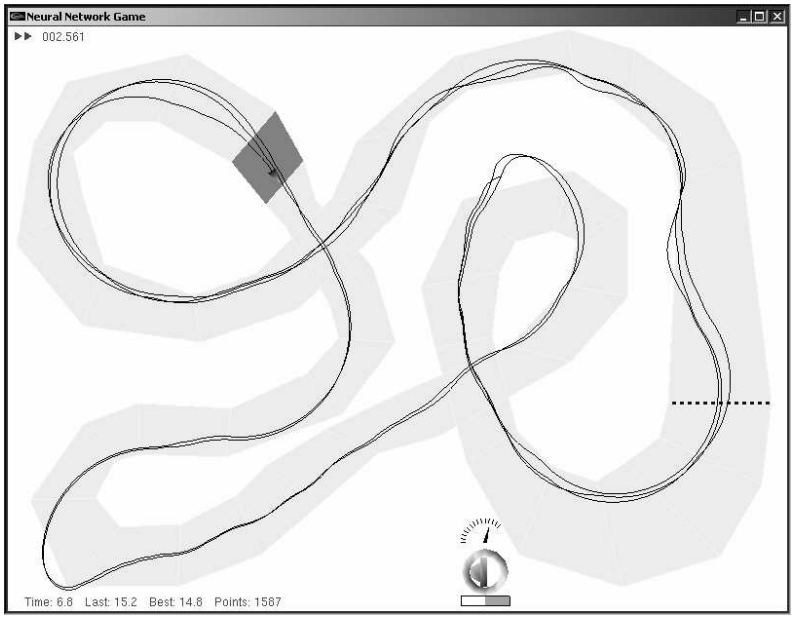
Fordeling af vægte

Ad y-akslen ses antal af neuroner og ad x-akslen ses vægt værdien. Man kan således se hvor meget neuroner har en given vægt. Vægt med værdien 0 ses i midten af spidsen.

Igen ses det at netværket er nok lidt for indlært, det har mistede noget af sin evne til at kunne generaliserer. Dog er den noget mindre spids end det netværk som så 2 felter foran sig.

Netværkets evne til at kører andre baner

Herunder er der vedlagt et eksempel på en kørsel på en anden bane end den som netværket ellers er blevet oplært på, med andre ord så skal netværket kører på en banen som det ikke er blevet oplært til. Det burde ikke have nogen indflydelse hvilken bane det kører på, da netværket meget gerne kun skulle have lært hvordan det skal behandle forskellige typer af kurver og ikke en hel bane som sådan.



Netværket ser 3 felter foran sig. På ukendt bane

Som det kan ses klarer netværket denne opgave helt fint. Det skal dog lige nævnes at lige når det starter på denne bane har det en ”lille skovtur” fra start til det første kryds, hvorefter det finder banen og så ellers kører korrekt derefter.
Det har lidt problemer med de lidt skarpe sving på banen, men ikke større problemer end man kunne forstille sig.
Med andre ord kan man sige at, det viser at netværket har lært forskellige kurver og ikke baner som helhed, for hvis det var selve banen det kunne huske ville det ikke være i stand til at kunne kører denne ukendte bane så godt.

Backpropagation Konklusion

Som det kan ses er det muligt at få et backpropagation netværk til at kunne køre en bil igennem en bane som enten er ”kendt eller ukendt”. Det har tilligemed vist sig muligt at lave et netværk som er i sand til at kunne kører bedre end det mekanisme som har lært det at kører (Simple Driver – mekanismen).

Det har dog vist sig temmelig besværligt at få præcenteret data’en på en sådan måde at netværket ville være i stand til at kunne få noget fornuftigt ud af det. Her tænktes ikke på at indgang data’en skulle have været anderledes men at selve måden hvorpå netværket blev indlært. Nogle af de første forsøg på at indlærer netværket indeholdte kode som ville lærer netværket op af flere gange, hvor det blev præsenteret for en strækning af banen, som så blev indlært X-antal gange før netværket blev præsenteret for den næste strækning. Dette viste sig yderst uhensigtsmæssigt.
Den bedste metode der blev fundet var at lade netværket få dataset for en hel omgang (ca. 300 dataset), som det så blev præsenteret for hvert dataset kun en gang. Denne metode

var den som gav de meste lovende resultater. Typisk ville de andre metoder lave netværk som var ude i stand til at kunne kører bilen på en fornuftig måde.

En af hoved årsagerne til at vi har været nødt til at prøve os frem til hvilken indlærings metode, eller retter sagt netværkets overfølsomhed overfor hvilken metode der bliver brugt, ligger i at vi ikke har nogen rigtig måde at kunne bestemme hvornår netværket har lært ”nok”. En af de typiske fejl var, at vi indlærte netværket for meget, som gjorde det ”ligeglad” med sving, det ville med andre ord bare kører lige ud. En anden fejl var, for lidt indlæring som gjorde at netværket ville kører rundt i en cirkel uden at komme nogen veje.
Vi forsøgte at efter hvert dataset at se på om fejlen var under en vis tærskel, men denne metode viste sig ubrugelig, da man ikke som sådan kan bestemme hvor godt netværket er ved kun at kigge på en lille given situation på banen. Vi skulle derimod have lavet en algoritme der kiggede over et længere forløb og ud fra resultatet burde lave en form for subjektiv vurdering som det var blevet bedre og ikke mindst godt nok så en over indlæring ikke ville forekomme.

Evolution

Evolution er en væsentlig anderledes måde at træne et netværk på end indlæringsalgoritmerne. I evolution fortæller man ikke hvad netværket skal gøre. Man udvælger bare den der tilfældigvis er bedst. Algoritmen i pseudo-kode:

```
opret_stor_population_af_tilfældigt_genererede_netværk();
test_alle_i_populationen();

while( leder_stadig )
{
        lav_ny_generation_ud_fra_de_bedste_individer();
        test_alle_i_populationen();
}

vælg_den_bedste();
```

Dette giver den frihed at man ikke skal komme med indlæringsdata, der ikke altid er til at få fat i. Til gengæld skal man kunne afgøre hvor gode individerne er til at løse en opgave, hvilket heller ikke altid er muligt. Dette bilspil er dog meget velegnet til evolution, da det hele handler om at komme hurtigst frem og tiden kan måles.

Implementationen

Systemet er implementeret ud fra pseudokoden ovenfor. Funktionerne hedder blot lidt noget andet. De er implementeret i EvolutionLearner.cpp og kaldes således hvis man eksempelvis kun vil udvikle en enkelt generation efter udgangs-populationen:

```
        CreateRandomizedPopulation( size, rSeedValue );
        EvaluateGeneration( timeTicksToLive );
        CreateNextGeneration( newPopulationSize, survivors, mutants,
mutationProb);
        EvaluateGeneration( timeTicksToLive );
        UseBestCarsInGame( numberOfCarsToUse );
```

Det er vigtigt at man har kørt `EvaluateGeneration(...)` før man kører `CreateNextGeneration(...)` eller `UseBestCarsInGame(...)`, for de er afhænguge af at populationen er prøvekørt for at kunne vælge de bedste. Som det fremgår, er der en del parametre der gives med som argumenter, så man nemt kan afprøve forskellige.

CreateRandomizedPopulation(size, rSeedValue)

Opretter et array af biler der får hver sin NeuralDriver med hvert sit nyoprettede (neurale) Net med alle vægte sat til tilfældige værdier. Antallet af biler gives med som argument. Ved små populationer har det også stor betydning hvilken random-seed der bruges.

Eneste anden parameter der har betydning her er hvor store vægtene bliver sat til i det nyoprettede net. Da evolution primært bygger på at blande vægte fra to gode forældre vil

vægtene aldrig blive større (abs) end de startede ud med. Hvis alle vægte initialiseres til at ligge tilfældigt mellem -0.5 og +0.5 vil deres børn aldrig kunne komme ud over disse grænser, men den implementerede CreateNetFromParents()-funktion. Grænserne for de tilfældige initial-vægte kan ændres i kode til Net::Layer::Neuron::Setup(...).

EvaluateGeneration(timeTicksToLive)

Løber bilerne igennem en for en. Placerer en bil på banen af gangen, og lader den køre det givne antal gameticks. Det er så bilen selv der holder styr på hvor godt den har kørt.

Da bilerne er tilfældigt genereret, er der mange der ikke kan finde ud af at køre hele vejen rundt på banen. Man kan derfor ikke bare teste på hvor lang tid det tog at komme rundt. Der er istedet implementeret et point-system der giver flest point for god kørsel, og det er denne point-score der er evalueringen af hvor god bilen er.

Hver gang en bil krydser en linie og kommer videre til næste felt på banen, får den point. Jo længere tid den er om at komme til næste linie afgør hvor mange point bilen får. Der gives ekstra straf-point for at køre udenfor banen, da de ellers havde kraftig tendens til at "snyde".

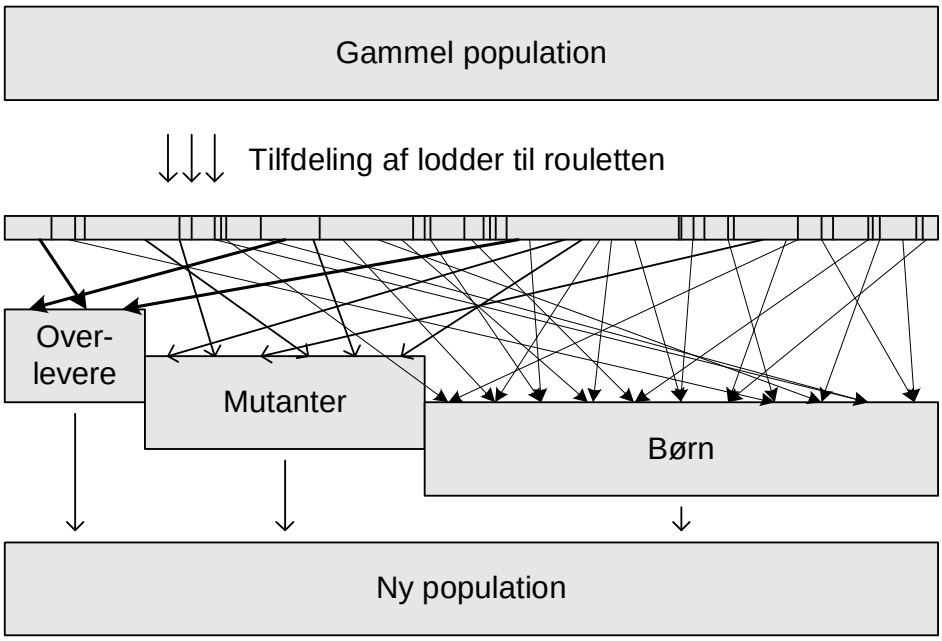
CreateNextGeneration(newPopulationSize, survivors, mutants, mutationProb);

Dette er hjertet i evolutionen. Man kan ændre meget i opbygningen af denne funktion for at optimere forbedringen ef den nye generation. Hej beskrives hvordan den er implementeret.

Udvælgelse af gode individer sker ved lodtrækning, hvor antallet af lodder til de enkelte individer afgøres af deres point fra evalueringen. Deres points lægges i et array af accumulerede point. Der vælges et tilfældigt tal indenfor summen af alle point, og vinderen søges i det accumulerede array. Der var en kedelig tendens til at de bedste ikke blev valgt fordi de dårlige fik for mange lodder. Som løsning blev det rettet, så man kun fik lodder for de points man havde over gennemsnittet. Alle dem med færre points end gennemsnittet kan således ikke overleve.

Et fastsat antal individer får lov til at overleve uden ændringer, for det er ikke sikkert at der kommer gode ændringer. Et andet fastsat antal individer overlever, men bliver muteret med Net::mutate(mutationProbability). De bliver alle udvalgt ved samme slags lodtrækning.

Resten af den nye population fyldes op med børn af hver to forældre der igen udvælges ved samme lodtrækning. Selve oprettelsen af barnet ud fra forældrene sker i Net::CreateNetFromParents(motherNet, fatherNet).



UseBestCarsInGame(numberOfCarsToUse)

Denne funktion laver en lille hitliste over hvem der har scoret flest point i evalueringen, og tilføjer dem til spillet, så man kan se hvordan de kører. Vil man vurdere de udviklede net bør man se på bilernes omgangstider, efter de har kørt et par runder. Points'ne bilerne har scoret er afhængige af hvor lang testperioden har været, og da man kan ændre på den, kan de ikke bruges som indikation på bilens færdigheder.

Net::mutate(float mutationProbability)

Funktionen mutate løber alle vægtene igennem for alle neuronerne i alle lagene, og muterer dem, hvis et tilfældigt trukket tal er mindre en den mutations-sandsynlighed der er givet med som argument

En mutation af en vægt er implementeret ved at give den en ny tilfældig værdi. Man kunne også bare forstærke en vægt ved at gange den med ti, så den stadig er i samme retning, men det har vi ikke særlig gode erfaringer med. Man kunne også bare addere en mindre tilfældige (+/-) værdi til den aktuelle vægt. Det vigtigeste er at man får ændret et lille antal af vægtene tilfældigt.

Net::CreateNetFromParents(motherNet, fatherNet)

Tager to net og opretter et nyt net, hvor alle vægtene sættes til en mellemting mellem fader-nettets og moder-nettets. Vi havde i første omgang implementeret det således at hver vægt blev enten moderens eller faderens, valgt tilfældigt. Men det giver et begrænset antal mulige nye børn, så det blev ændret til at give barnet en tilfældig procentdel af hver

forældres. Eksempelvis hvis en vægt hos moderen er +1 og hos faderen -1 og tilfældigheden siger 75% af faderens, bliver barnets vægt (+1)*0.25 + (-1)*0.75 = -0.5

Dette resulterer i at efter mange generationer af børn kommer vægtene tættere og tættere på 0.0, da de jo ikke kan overgå begge forældre. Man får derfor en degenerering. Dette kan løses ved at multiplicere barnets nye vægt med eksempelvis 2 eller 1.5. Præcis hvad der skal multipliceres med må man prøve sig frem med, for en for stor multiplikator giver for ekstreme vægte efter mange generationer, og det ønskes heller ikke.

Fintuning af parametre.

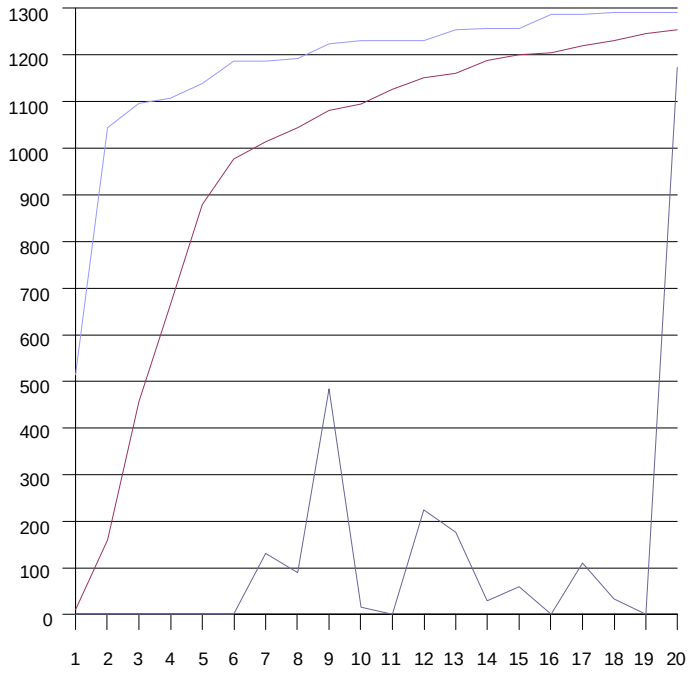
I dette afsnit vil jeg systematisk gennemgå de forskellige parametre der kan sættes, for at finde deres effekt, og dokumentere med testdata. Der vil blive kigget på:

- Antal generationer.
- Effekten af random-seed.
- Evalueringstid.
- Antal individer i population.
- Antal survivers, mutants og children i population.
- Mutations-sandsynligheden.
- Antal neuroner i indput-laget (Hvor langt frem på banen kan netværket se).
- Antal skjulte lag.
- Antal neuroner i skjulte lag.
- Overførings-funktionen.
- Initial-vægtenes størrelse.

De forskellige test er udført på den bane der hedder: "Teach Me To Drive 2.track" som blev designet til at tvinge bilerne til ikke at skære hjørner af, og derfor blive på banen.

Antal generationer

Hvor mange generationer der skal gå før man finder et rigtig godt net varierer efter de mange indstillinger. Jo flere generationer der går des mindre ændringer sker der mellem generationerne. Generelt skal et lille netværk udvikles over få generationer, og store populationer kræver færre generationer. Herunder vises grafen over point for bedste bil, gennemsnittet og dårligste bil i en population over 20 generationer.



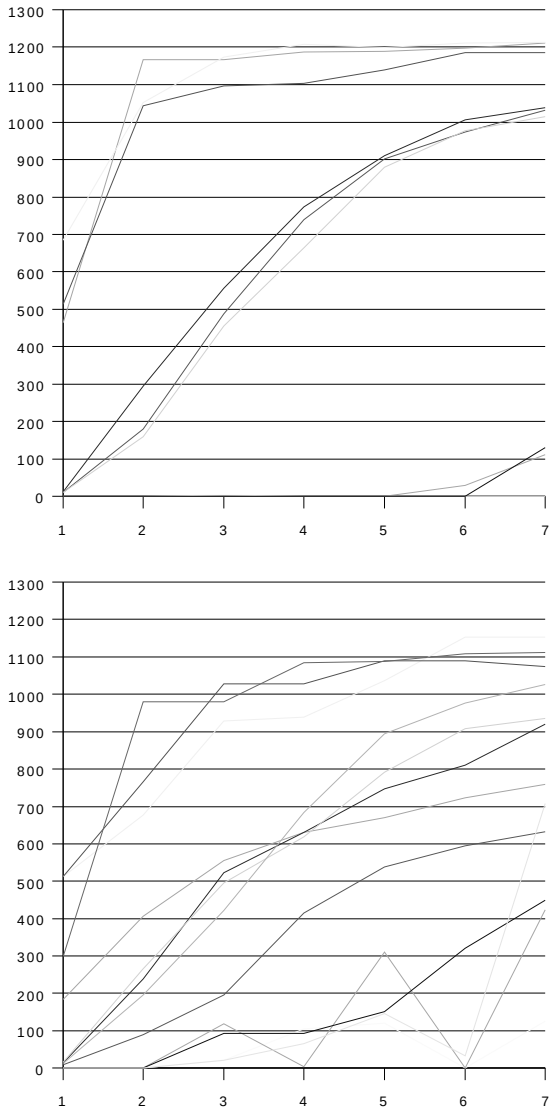
Net: {13, 2} Population: 500
RandomSeed: 9
Evaluering: 800
Overlevere: 100
Mutanter: 100
Mutations-sandsynlighed: 0.001

Som det kan ses, bliver bedste bil hurtigt rigtig god, og så bliver den ikke meget bedre. Gennemsnittet stiger støt, men det betyder bare at bilerne bliver mere og mere ens, og der bliver mindre chance for at få et barn med forbedret kombination.

Når bedste bil bliver dårligere, betyder det at den der var bedst ikke overlever lodtrækningen. Når mange biler bliver gode, blive det mere tilfældigt hvilke der overlever, da de så allesammen får mange lodder.

Effekten af random-seed.

Da meget i evolutionen er baseret på tilfældigheder, har random-seed en vis betydning. Ved store populationer, bør det ikke betyde så meget. Her vises som før bedste, gennemsnit og dårliste pointscore, men med forskellig random-seed. Populationen er på henholdsvis 500 og 100 i eksemplerne:



Med populationer over 500 har random-seed tilsyneladende ikke den helt store betydning.

Evalueringstid.

Hvis man kun evaluerer på en kort tid, når bilen ikke banen rundt under testen, og man udvælger derfor kun på hvor godt bilen klarer første del af banen. Jo længere tid man tester, des mere viser de enkelte biler deres overlegenhed, men det tager lang tid at evaluere. Her er en tabel der viser den bedste bils (i points) omgangtid i forhold til hvor lang evalueringen var under træning. populationen er 500, 7. generation på "Teach Me To Drive 2.track". (bemærk at tiden tælles 40ms op for hver game-cycle, så en langsom computer vil ikke give langsommere omgangstider. Tiden vil bare gå langsommere).

Evalueringstid:	Antal omgange:	Bedste omgangstid:
5000	ca. 15.0	15.6 s
1000	ca. 3.1	12.8 s
600	ca. 1.9	13.3
300	ca. 0.9	13.4 s
100	ca. 0.3	16.4

Mærkeligt nok, ser det ud til at tester man dem for grundigt, bliver dårligere valgt til næste generation. Det må være en fejl eller uhensigtsmæssighed i lotteriet for overlevelse.

Antal individer i population.

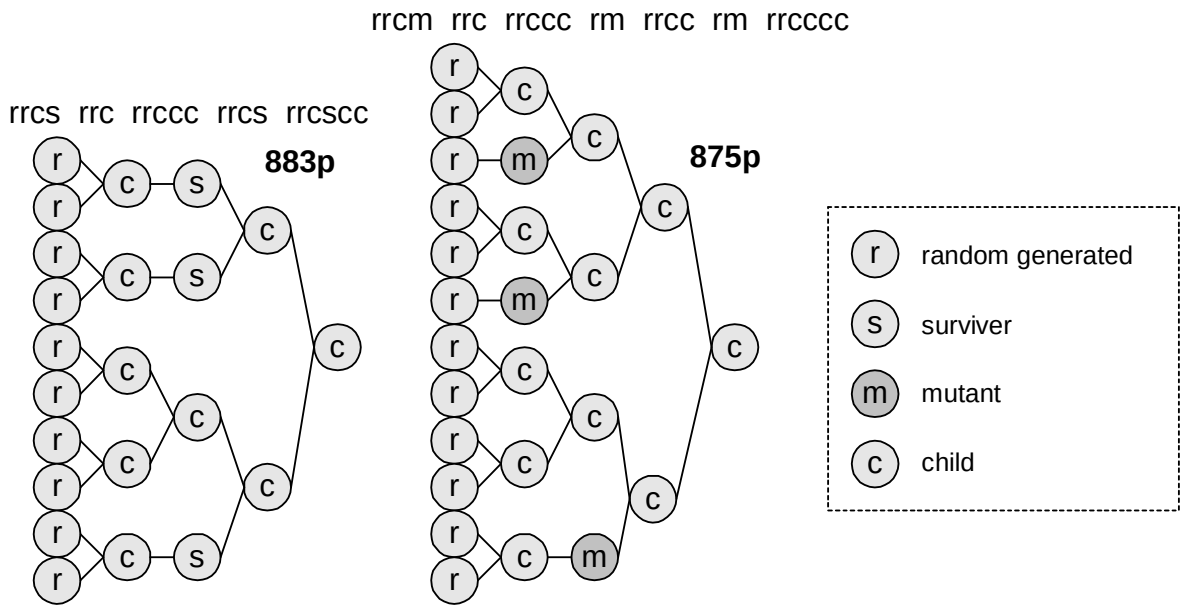
Man kan være heldig at få den perfekte bil ud af tilfældighederne i første forsøg. Men jo flere tilfældige biler der laves, des større sandsynlighed er der naturligvis også for at der findes en rigtig god imellem. Testen herunder er lavet med 25% overlevere og 25% mutanter, 600 ticks og 7. generation.

Individer i population	Bedste point 1. gen:	Bedste point 7. gen:	Bedste omgangstid:
25	55	328	24.2
100	345	686	15.0
400	345	817	13.0
1600	655	819	13.2
6400	664	814	13.2

Igen her strider test-resultaterne imod teorien, idet bilerne bliver dårligere når populationerne bliver meget store. Igen må fejlen ligge i udvælgelsen til næste generation.

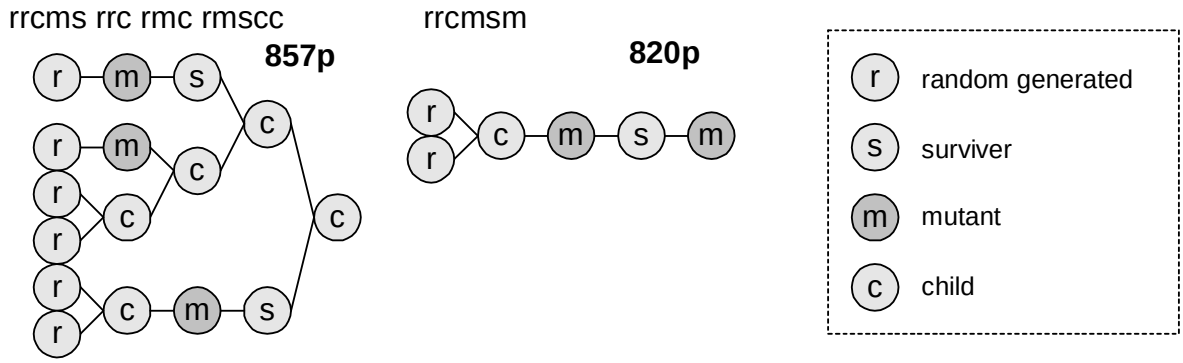
Antal survivers, mutants og children i population.

Til at vurdere hvordan hvilken af de tre muligheder der er meste effektiv for forbedring af bilerne, gemmes hver bils stamtræ i en hægtet liste. De vises som en enkelt tekststreng når bilerne udvælges til spillet, og kan dekodes bagfra til et træ. Her vises de to bedste biler fra 7. generation af population på 500.



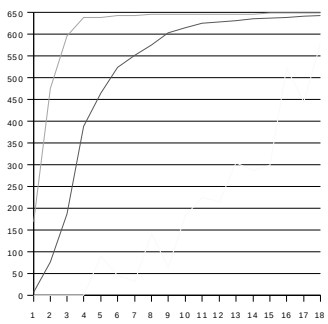
Mutations-sandsynligheden.

Der var tilsyneladende ikke særlig meget gavn af mutationerne. De muterede indgik sjældent i stamtræerne. Mutations-sandsynligheden var også kun sat til 0.001 (1 promille). Ved at hæve den til 0.02 (2 procent) og sætte antallet af mutanter op fra 100 til 200, kommer de med i udviklingen. Sættes mutations-sandsynligheden til 0.5 (50 procent), er der ingen mutationer i top-3 stamtræerne. I et forsøg med mutations-sandsynligheden til 0.1 (10 procent), blev familie-træet til højre genereret.

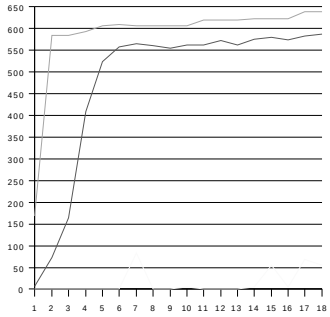


Det er svært at fastsætte en perfekt mutations-rate, da den jo baserer sig på at populationen får en ekstra chance for at få en løbende udvikling, og ikke lide for meget af indavl. Herunder ses bedste, gennemsnit og dårligste bil over 30 generationer med henholdsvis ingen mutation, 5% og 1% mutation (400 individer, 100 overlevere, 100 mutanter, 400 evaluerings-ticks)

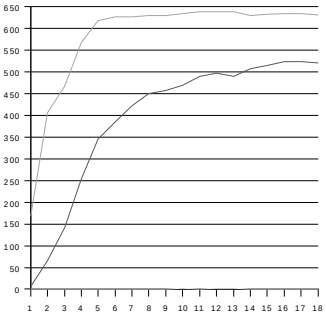
0%



1%



5%



Det ser ikke på disse grafer ud til at mutationen hjælper noget. Det ser mere ud til at de er en hæmsko for eliten, hvilket de jo også tit er i naturen. Jo mindre de er muteret, des mindre skade gør de åbenbart. En grov mutation har meget svært ved at konkurrere med de andre og bliver udryddet inden deres alternative gener blandes ind i mængden og finder anvendelse.

Antal neuroner i indput-laget (Hvor langt frem på banen kan netværket se).

Jo længere man kan se frem på banen, jo bedre kan man planlægge en perfekt route. Problemet er så at netværket skal have kapacitet til at behandle det større antal situationer, så det kan være nødvendigt med flere neuroner i skjulte lag, hvilket igen gør det mere krævende at oplære.

I denne test laves netværk der kun har input- og output-lag, med forskellig antal input, så de kan se forskellig længde frem. Der er udviklet så det bedste point-tal er opnået indenfor de første 10 generationer på 500 individer, evalueret 500 ticks. Omgangstid 2 er på en anden bane end den trænede, for at se generalisationsevnen. De to tider tages for samme individ.

Se felter frem	Net	Points (gen.)	Omgangstid	Omgangstid 2	Kommentar
1	{5,2}	484 (8)	17.4	14.3	kører meget side til side.
2	{9,2}	683 (7)	13.5	10.1	
3	{13,2}	692 (10)	13.1	9.0	Slingrer lidt
4	{17,2}	767 (8)	12.1	8.7	
5	{21,2}	762 (7)	12.1	9.5	
6	{25,2}	713 (10)	12.0	10.9	9.0 i omgangstid 2 for søskende
7	{29,2}	771 (10)	11.9	8.6	11.8 for Car 4 og 6

8	{33,2}	757 (8)	11.9	8.7	
9	{37,2}	768 (10)	12.0	8.7	
-	-	-	11.7	7.8	SmartDriver...

Det ses at netværket bliver bedre, når det kan se langt frem. Men det ser ikke ud til at give særlig meget ekstra, når der ses ud over 4 felter frem. Medmindre man skal optimere de sidste millisekunder, kan man nøjes med at se 4 - 5 felter frem.

Antal skjulte lag.

I testen vælges at se 5 felter frem, og lave forskellige antal skjulte lag. Ellers er reglerne som de forudgående test. Det formodes at 3 skjulte lag og derover ikke er nogen fordel.

Skjulte lag	Net	Points (gen.)	Omgangstid	Omgangstid 2	Kommentar
0	{21,2}	762 (7)	12.1s	9.5s	
1	{21, 11, 2}	581 (10)	15.2s	10.2s	Wobbler meget på rattet.
2	{21, 15, 7, 2}	500 (10)	17.5s	11.0s	Kan ikke lave glidende styring.
3	{21, 16, 11, 6, 2}	000 (0)	00.0	0.0	

Tendensen er klar. Det er ikke nogen fordel med mange lag, når der er veldimensioneret antal input. Det kan muligvis lykkes at træne disse dybe net med meget store populationer og mange generationer, men det vil blive svært, og sandsynligvis ikke besværet værd. Med få input er det måske en fordel.

Skjulte lag	Net	Points (gen.)	Omgangstid	Omgangstid 2	Kommentar
0	{5, 2}	484 (8)	17.4s	14.3s	
1	{5, 3, 2}	468 (10)	17.8s	14.6s	
2	{5, 4, 3, 2}	318 (9)	20.6s	15.5s	
3	{5, 5, 4, 3, 2}	250 (10)	21.2s	14.4s	

Heller ikke her ser det ud til at være en fordel med skjulte lag.

Antal neuroner i skjulte lag.

Det ser jo godt nok ikke ud til at skjulte lag er nogen fordel, ud fra den foregående test. Men lad os alligevel prøve at se tre felter frem og have et enkelt skjult lag med forskelligt antal neuroner.

Net	Points (gen.)	Omgangstid	Omgangstid 2	Kommentar
{13, 1, 2}	111 (10)	--.-s	-.-s	Kan ikke finde rundt.
{13, 2, 2}	672 (10)	13.4s	9.4s	
{13, 4, 2}	456 (10)	14.0s	9.2s	
{13, 6, 2}	632 (10)	14.7s	10.4s	
{13, 9, 2}	584 (10)	15.6s	11.1s	
{13, 13, 2}	586 (10)	15.5s	11.4s	
{13, 80, 2}	555 (10)	15.9s	11.0s	

Når man ser den køre, ses det at de ikke rigtig kan finde ud af at dreje lidt på rettet. Det er helt ud til den ene eller anden side. De kan dog sagtens finde rundt alligevel, de drejer bare meget hurtigt til den ene og den anden side, så de kommer til at køre nogenlunde lige ud i den rigtige retning. Der er indført straf i hastigheden når man drejer skarpt, og det koster dyrt for disse biler. Man kunne måske lave noget postprocessering, der konvereterer det pulsmodulerede retningssignal som netværket leverer, og lave et flydende output ud af det.

Overførings-funktionen.

Vi har som standard til evolutions netværket valgt en lineær overføringsfunktion. Lad os se om nettet bliver bedre med en sigmoid overføringsfunktion. Ændringen laves i `void Net::Layer::Neuron::PropagateNeuron()`, og testen består i bedste omgangstid for forkellige net-dimensioner.

lineær: 12.1 sigmoid: 12.4

Net	Lineær	Sigmoid
{ 5, 2}	17.1s	17.6s
{13, 2}	13.1s	13.0s
{17, 2}	12.1s	12.4s
{17, 7, 2}	14.8s	13.1s
{21, 10, 2}	15.2s	14.5s

Det kan tyde på at den sigmoide overføringsfunktion er en fordel i større net, med tendens til at glemme mellemværdierne for outputs. Men til mindre simple net er den ikke nødvendigvis en fordel.

Initial-vægtenes størrelse.

Det har betydning hvor stort et interval vægtene sættes indenfor, når der oprettes nye net med random vægte. Er der for lidt variation (ex. -0.5 til +0.5), kan bilerne ikke rigtig lære nok, da de ikke kan øge vægtene ud over de oprindelige rammer. Er start-intervallet for

stort (ex. -32 tol +32) får bilerne tendens til at miste evnen til at give mellemværdier for outputs. Det giver en meget urolig kørsel. Alle forgående tests har haft initial-vægte mellem -16 og +16.

Forslag til forbedringer for evolution

Hvis ikke lotteriet fordeler de gode meget, som i implementationen, ser man tit at de stærkeste risikerer at tabe i lotteriet, og toppen af generationen bliver dårligere jo flere generationer vi kommer frem. Man kunne lave en top-10 så de allerbedste var sikret en plads. Så ville top-10 i det mindste aldrig degenerere, om man kunne roligt lade generationerene løbe derudaf.

Det ender ofte med at en population bliver meget ensartet efter mange generationer. Hvis man delt hele populationen op i mindre flokke som hver kunne udvikle deres egne forbedrede udgaver ud fra deres gener, og så kunne man blande flokkene engang imellem. Det ville muligvis bane vejen for mere innovativ dannelse af nye medlemmer i populationen.

Konklusion på Evolution

Det virker ganske udemærket at få udklækket et neuralt netværk til styring af en simpel racerbil, ved evolution. Der er mange parametre der kan sættes for at få det optimale resultat, og det er ikke lykkedes at give en endegyldig guide. Kun en række retningslinier.

Det ser ud til at der stadig er lidt programmeringsfejl i udvælgelsen eller evalueringen, og det kan være skyld i at man ikke kan skalere evolutionen op. Teorien siger at, jo større population og jo flere generationer, des bedre bil får man. Men testresultaterne siger at man hurtigt møder et loft, og bilerne bliver dårligere.

En meget sandsynlig kandidat til problemet er den random-funktion vi bruger. Den laver et pseudo-random nummer mellem 0 og 32.767, og hvis man laver en stor population kommer der flere lodder i lotteriet end der er numre at udtrække og nogen biler vil ikke kunne blive trukket. Det kunne løses ved at trække to random-numre og shifte det ene op:

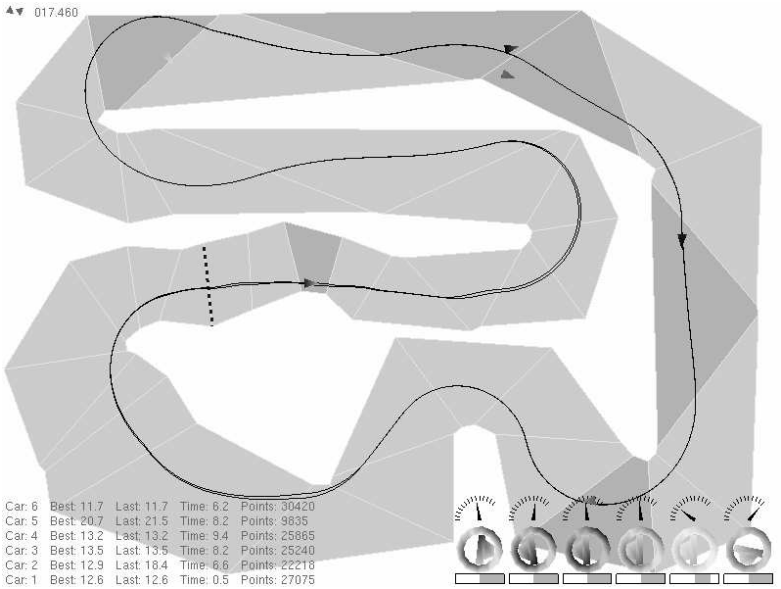
```
long bigRand()
{
      return rand() * RAND_MAX + rand();
}
```

Det giver 1.073.676.289 forskellige, som egentlig kan være i en int og ikke behøver en long. Den kan udvides yderligere hvis den ene milliard ikke er nok.

Man løber også ind i et RAM-problem, når populationen og antal generationer forøges meget. Vi får desværre ikke håndteret oprydning i RAM helt pænt, så efter omkring 20 generationer har programmet allokeret mere RAM end en almindelig windows-box har idag.

Kunne man stille alle parametrene optimalt og der ikke var nogen programmeringsfejl, burde generationerne ikke blive dårligere med tid, og mutatione burde kunne tilføre nye gener, så man i teorien kunne lade evolutionen køre en hel nat, og vågne op til en perfekt Driver.

Det er dog stadig lykkedes at udklække Drivere med evolution der er hurtigere end med de to andre indlæringsformer, så konklusione må være at det virker udemærket. Som det ses på billedet nedenfor, bliver bilen rimelig god til at skære hjørner af pænt. Det er enkelte gange lykkedes at slå SmartDriver på enkelte baner, men det er sjældent, og endnu ikke sket på "Teach Me To Drive 2.track" som er vist herunder.



Konklusion

Det er lykkedes os at implementere alle tre netværk og få bilen til at køre på banen ved hjælp af disse netværk. Bilens opførsel på banen afhænger meget af hvilke indlæring der bliver benyttet, og alle tests tyder på at evolutions netværket er den bedste til at løse en sådan type opgave. Alle netværk er begrænset af vejens udformning da afstanden til punkterne på vejen samt vinklerne og speed kan være det samme i forskellige situationer. Dvs. at bilen i nogle tilfælde kan tro at den skal dreje, mens den faktisk burde køre lig ud. Dette opdager bilen først efter den har passeret stregen, hvis ikke den kigger langt nok frem.

Målet var at lave den perfekte kører, der ikke var til at slå som menneske. Dette er ikke lykkedes, men var også et højt krav. Det er svært at slå SmartDriver som er en logisk implementeret kører. Den kan man slå som menneske hvis man virkelig koncentrerer sig, så mennesker kan altid slå NeuralDriver. I alsidighed har vores NeuralDriver heller ikke kunnet slå SmartDriver. NeuralDriver bliver altid optimeret til den bane den er trænet på, og kan så finde på at lave mærkelige ting på andre baner. De finder dog som regel rundt, så generalisationsevnen kan også ses.

Generel succes – men der er som altid flere optimeringer at hente, hvis man har tiden til det.

Bilag 1, Spil Source-code

Main.cpp

```
#include <GL/glut.h>
#include "Constants.h"
#include "Helpers.h"
#include "MainGraphics.h"
#include "MenuBuilder.h"
#include "InputHandlers.h"
#include "Car.h"
#include "Track.h"
#include "SimpleDriver.h"
#include "SmartDriver.h"
#include "NeuralDriver.h"
#include "EvolutionLearner.h"
#include <stdio.h>


////////////////////////////////////////////


float viewPortLeft   = -8.0f; //-1.0f;
float viewPortRight  =  8.0f; //14.0f;
float viewPortBottom = -6.0f; //-3.0f;
float viewPortTop    =  6.0f; // 9.0f;

Track *theTrack              =   NULL;
Car   *theCars[maxNumOfCars] = { NULL };

EvolutionLearner *evolutionLearner;

Driver *keyDriverOne = NULL;
Driver *keyDriverTwo = NULL;

bool gamePaused = false;
int gameTicksCount = 0;


////////////////////////////////////////////


void display()
{
            //placeCamera();  //no need to place each time !!!

            glMatrixMode( GL_MODELVIEW );
            glClear( GL_COLOR_BUFFER_BIT );
            glLoadIdentity();


            theTrack->drawTrack();

            drawDummyThingy();
            for(int i=0; i<maxNumOfCars; i++)
            {
                        if(theCars[i] != NULL) //found an existing Car
                        {
                                    theCars[i]->drawDecisionLines();
                                    theCars[i]->drawTrail();
                                    theCars[i]->drawCar();
                                    theCars[i]->drawSteeringWheel();
                        }
            }

            if(gamePaused) drawPausedNote();

            glutSwapBuffers(); //DoubleBuffer update screen
}


void timer(int value)
{
            glutTimerFunc(40, timer, 0);  // register next timer-tick

            if(!gamePaused)
            {
                        gameTicksCount++;

                        for(int i=0; i<maxNumOfCars; i++)
                        {
                                    if(theCars[i] != NULL) //found an existing Car
                                    {
                                                theCars[i]->move();
                                    }
                        }
            }

            glutPostRedisplay(); //aka. display();
}




void initWorld()
{
```

```
            theTrack = new Track();

            theTrack->loadFromFile(7);       ////////////////////////////////////////////////////

            keyDriverOne = new KeyDriver(); // must be created, even if not used!!!
            keyDriverTwo = new KeyDriver(); // must be created, even if not used!!!
}



int addNewCarToGame(Driver* driver)
{
            for(int i=0; i<maxNumOfCars; i++)
            {
                        if(theCars[i] == NULL) //found an available-slot
                        {
                                    theCars[i] = new Car(i);
                                    theCars[i]->setDriver(driver);
                                    return i;
                        }
            }
            printf("Unable to fit another Car\n");
            return -1;
}


void initPlayers() {
//          addNewCarToGame(keyDriverOne);
//          addNewCarToGame(keyDriverTwo);
//          addNewCarToGame(new SimpleDriver());
//          addNewCarToGame(new SmartDriver());

//          int layerDefinition[2] = {5, 2};
//          adalineLearner = new Adaline(2, layerDefinition);
//          addNewCarToGame(new NeuralDriver(adalineLearner->CreateAdalineNet()));

            int layerDefinition[] = {17, 2};
            evolutionLearner = new EvolutionLearner(2, layerDefinition);
            evolutionLearner->CreateRandomizedPopulation(500, 7);
            evolutionLearner->loopInEvolution(3, 500, 500, 100, 100, 0.01);

            evolutionLearner->EvaluateGeneration(500);
            evolutionLearner->UseBestCarsInGame(4);

            addNewCarToGame(new SimpleDriver());
            addNewCarToGame(new SmartDriver());
}



void main(int argc, char** argv)
{
            glutInit(&argc, argv);

            glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
            glutInitWindowSize(800,600);
            glutInitWindowPosition(200, 100);
            glutCreateWindow("Neural Network Game");

            initWorld();   // load first track
            initPlayers(); // add players to track

            initGraphics();
            initCallbackFuncs();
            initMenu();  //after initPlayers

            glutMainLoop();
}
```

Car.cpp

```
#include "Car.h"
#include "CarTrail.h"
#include "Driver.h"
#include "Track.h"
#include "Constants.h"
#include "Helpers.h"
#include <iostream.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <strstrea.h>
#include <iomanip.h>

float carDrawPoints[3][2] = {
            { 0.0f, 0.2f},   //front
            {-0.1f,-0.1f},   //back-left
            { 0.1f,-0.1f} }; //back-right


Car::Car(int carsIndex)
{
            myDriver     = NULL;
            myDevHistory = NULL;
            myCarTrail   = new CarTrail(1000);  //4
            restartOnTrack(carsIndex);

            showDecisionLines = false;
```

```
            showTrail         = false;
}

Car::Car(Car *carToCopy)  //Copy-Constructor
{
            myDriver     = carToCopy->myDriver->clone();
            myDevHistory = NULL;
            myCarTrail   = new CarTrail(carToCopy->myCarTrail->size());
            restartOnTrack(carToCopy->myCarsIndex);

            showDecisionLines = false;
            showTrail         = false;

            DevHistoryElement *temp = carToCopy->myDevHistory;
            while(temp != NULL)
            {
                        addToDevHistory(temp->creation);
                        temp = temp->next;
            }
}


Car::~Car() {
            deleteMyDriver();
            delete myCarTrail;

            DevHistoryElement *temp;
            while(myDevHistory != NULL)
            {
                        temp = myDevHistory->next;
                        delete myDevHistory;
                        myDevHistory = temp;
            }
}



void Car::restartOnTrack(int carsIndex)
{
            extern Track *theTrack;

            myCarsIndex = carsIndex;     //only used for drawing steeringwheel and statistics-text in different locations

            speeder  = 0;
            steering = 0;

            maxCarSpeed = 30;

            carPosition[0] = (theTrack->trackLines[0].left[0] + theTrack->trackLines[0].right[0]) / 2;
            carPosition[1] = (theTrack->trackLines[0].left[1] + theTrack->trackLines[0].right[1]) / 2;

            carDirection = angleBetweenTwoPoints(theTrack->trackLines[0].left, theTrack->trackLines[0].right) + 90;

            carSpeed = 0;

            currentTrackField = 0;
            trackFieldAdvance = 0;

            myCarTrail->clear();

            lapTimeCount = 0;
            lapTimeLast  = 0;
            lapTimeBest  = 0;

            gatheredPoints  = 0;
            pointsCountDown = 20;

            switch(myCarsIndex)
            {
                        case 0  : setColors(COLOR_RED,          COLOR_YELLOW      ); break;
                        case 1  : setColors(COLOR_GREEN,        COLOR_WHITE       ); break;
                        case 2  : setColors(COLOR_WHITE,        COLOR_YELLOW      ); break;
                        case 3  : setColors(COLOR_PURPLE,       COLOR_RED_BRIGHT  ); break;
                        case 4  : setColors(COLOR_BLUE_DARK,    COLOR_BLUE_BRIGHT ); break;
                        case 5  : setColors(COLOR_RED_BRIGHT,   COLOR_GREEN       ); break;
                        default : setColors(COLOR_WHITE,        COLOR_GRAY_BRIGHT ); break;
            }
}



Driver* Car::getDriver()
{
            return myDriver;
}

void Car::setDriver(Driver* newDriver)
{
            deleteMyDriver();

            steering = 0;
            speeder  = 0;
            carSpeed = 0;

            myDriver = newDriver;
}



void Car::deleteMyDriver()
{
            extern Driver *keyDriverOne;
            extern Driver *keyDriverTwo;

            if(myDriver == NULL || myDriver == keyDriverOne || myDriver == keyDriverTwo)
            {
                        //Dont Delete It
            }
            else
            {
                        delete myDriver;
```

```
		}
}




void Car::drawCar()
{
		if(myDriver == NULL) return; // dont draw car without driver

		glLoadIdentity();

		extern float viewPortLeft, viewPortBottom;

		ostrstream oss;
		oss << setw(5) << setprecision(1) << setiosflags(ios::fixed);
		oss << "Car: " << myCarsIndex + 1;
//		oss << "   Speed: " << carSpeed;
//		oss << "   Steering: " << steering;
//		oss << "   Track: " << currentTrackField;
		oss << "   Best: " << (float)(lapTimeBest)  / 1000;
		oss << "   Last: " << (float)(lapTimeLast)  / 1000;
		oss << "   Time: " << (float)(lapTimeCount) / 1000;
		oss << "   Points: " << gatheredPoints;
		char *str = oss.str();

		//draw Statis-String
		glColor3fv(COLOR_GREEN);
		glRasterPos2f(viewPortLeft + 0.1f, viewPortBottom + 0.1f + (myCarsIndex*0.3f));
		int len, i;
		len = (int)strlen(str) - 1;
		for (i = 0; i < len; i++)
		{
			glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[i]);
		}
		delete str;


		//Draw Car
		glTranslatef(carPosition[0], carPosition[1], 0.0f);
		glRotatef(carDirection-90, 0.0, 0.0, 1.0);

		glPolygonMode(GL_FRONT, GL_FILL);
		glBegin(GL_POLYGON);
			glColor3fv(colorCarFront);
			glVertex2fv(carDrawPoints[0]);
			glColor3fv(colorCarRear);
			glVertex2fv(carDrawPoints[1]);
			glVertex2fv(carDrawPoints[2]);
		glEnd();
}




void Car::setColors(const float *frontColor, const float *rearColor)
{
		arrayCopyF(frontColor, colorCarFront, 3);
		arrayCopyF(rearColor,  colorCarRear,  3);
}




void Car::move()
{
		extern Track *theTrack;

		lapTimeCount += 40; // 40 milliseconds between screen-update (approx)
		pointsCountDown--;

		if(myDriver == NULL) return; // dont move car without driver


		myDriver->decideSteering(&speeder, &steering, theTrack, currentTrackField, carPosition, carDirection, carSpeed/maxCarSpeed);


		if( speeder >  1) speeder  =  1;
		if( speeder < -1) speeder  = -1;
		if(steering >  1) steering =  1;
		if(steering < -1) steering = -1;

		carDirection += 8*(steering * (1.0 - (speeder+1) / 8)); //incl. steering-penalty for accelerating
		while(carDirection > 360.0f) carDirection -= 360.0f;
		while(carDirection <   0.0f) carDirection += 360.0f;

		carSpeed     += speeder;
		if(carSpeed > maxCarSpeed) carSpeed = maxCarSpeed;
		if(carSpeed <      0     ) carSpeed = 0;

		carSpeed     *= 1.0f - ((steering>0.0f)? steering: -steering) / 20.0f; //penalty in speed for turning
		carSpeed     *= 0.98f;                          //penalty in speed for wind-resistance

		carPosition[0] = (float)( carPosition[0] + cos(carDirection*2*PI/360)*carSpeed/100 );
		carPosition[1] = (float)( carPosition[1] + sin(carDirection*2*PI/360)*carSpeed/100 );

		myCarTrail->addTrailPoint(carPosition);

		int nextLine = (currentTrackField+1) % theTrack->length;
		//Check Backwards
		if(beyondLine(theTrack->trackLines[currentTrackField].right, theTrack->trackLines[currentTrackField].left, carPosition))
		{
			currentTrackField--;
			nextLine--;
			if(currentTrackField < 0) currentTrackField = theTrack->length - 1;
			if(nextLine          < 0) nextLine          = theTrack->length - 1;
		}
		else //Check Forwards
		if(beyondLine(theTrack->trackLines[nextLine].left, theTrack->trackLines[nextLine].right, carPosition))
```

```
	{
		currentTrackField++;
		nextLine++;

		if(trackFieldAdvance == currentTrackField - 1)
		{
			trackFieldAdvance++;

			if(pointsCountDown < 0) pointsCountDown = 0;

			gatheredPoints += pointsCountDown;		//add points
			pointsCountDown = 20;
		}

		if(currentTrackField >= theTrack->length) currentTrackField = 0;
		if(nextLine          >= theTrack->length) nextLine          = 0;
		if(trackFieldAdvance >= theTrack->length)
		{
			trackFieldAdvance = 0;

			lapTimeLast = lapTimeCount;
			if(lapTimeBest == 0 || lapTimeLast < lapTimeBest) lapTimeBest = lapTimeLast;
			lapTimeCount = 0;
		}
	}
	//Check Left and right
	if( beyondLine(theTrack->trackLines[currentTrackField].left,  theTrack->trackLines[nextLine].left,  carPosition) ||
		beyondLine(theTrack->trackLines[nextLine].right, theTrack->trackLines[currentTrackField].right, carPosition) )
	{
		carSpeed     *= 0.80f;			//penalty in speed being outside Track
		pointsCountDown -= 4; //penalty in points being outside Track
	}

	theTrack->setHighlightTrackElement(currentTrackField);
}



void Car::drawTrail()
{
	if(showTrail && myDriver != NULL)
	{
		myCarTrail->drawTrail();
	}
}


void Car::earthQuake()
{
	carPosition[0] += (float)rand() / RAND_MAX - 0.5f;
	carPosition[1] += (float)rand() / RAND_MAX - 0.5f;
}



void Car::addToDevHistory(char acCreation) //add to end to make copy easier!
{
	if(myDevHistory == NULL)
	{
		myDevHistory = new DevHistoryElement;
		myDevHistory->creation = acCreation;
		myDevHistory->next     = NULL;
	}
	else //at least one element !!
	{
		DevHistoryElement *temp = myDevHistory;
		while(temp->next != NULL)
		{
			temp = temp->next;
		}
		//now temp == last
		temp->next           = new DevHistoryElement;
		temp->next->creation = acCreation;
		temp->next->next     = NULL;
	}
}



void Car::revokeLastInDevHistory()
{
	if(myDevHistory == NULL) return;
	if(myDevHistory->next == NULL)
	{
		delete myDevHistory;
		myDevHistory = NULL;
		return;
	}
	DevHistoryElement *temp = myDevHistory;
	while(temp->next->next != NULL)
	{
		temp = temp->next;
	}
	delete temp->next;
	temp->next = NULL;
}

void Car::inheritDevHistory(Car* apoParentCar)
{
	DevHistoryElement *temp = apoParentCar->myDevHistory;
	while(temp != NULL)
	{
		addToDevHistory(temp->creation);
		temp = temp->next;
	}
}

void Car::printDevHistory()
{
	DevHistoryElement *temp = myDevHistory;
	printf("{");
	while(temp != NULL)
	{
```

```
                        printf("%c", temp->creation);
                        temp = temp->next;
                }
                printf("}");
}
```

Track.cpp

```
#include "Track.h"
#include "Car.h"
#include "Constants.h"
#include "Helpers.h"
#include <GL/glut.h>
#include <stdio.h>
//#include <iostream.h>
#include <fstream.h>

#include <direct.h>   //mkdir
#include <windows.h>


//#include <sys/types.h>
//#include <sys/stat.h>
//#include <fcntl.h>
//#include <unistd.h>

const int defaultTrackLength = 19;
TrackLine defaultTrackLines[defaultTrackLength] = {
                {{ -6.0,-3.0 },{ -5.0,-2.5 }},
                {{ -6.0,-2.0 },{ -5.0,-2.0 }},
                {{ -6.0, 0.0 },{ -5.0, 0.0 }},
                {{ -4.5, 1.0 },{ -3.0, 1.0 }},
                {{ -5.0, 2.0 },{ -4.0, 2.0 }},
                {{ -5.0, 4.5 },{ -4.0, 3.0 }},
                {{ -3.0, 4.5 },{ -3.0, 3.0 }},
                {{ -1.0, 5.5 },{ -1.0, 4.0 }},
                {{  2.0, 5.5 },{  2.0, 4.0 }},
                {{  4.0, 4.5 },{  4.0, 3.0 }},
                {{  7.0, 1.0 },{  5.0, 2.0 }},
                {{  6.0,-1.0 },{  5.0, 1.0 }},
                {{  3.0,-2.0 },{  4.0, 0.0 }},
                {{  1.0, 0.0 },{  3.0, 1.0 }},
                {{  0.0, 2.0 },{  1.0, 3.0 }},
                {{  0.0, 2.0 },{  1.0, 3.0 }},
                {{ -0.5, 2.0 },{ -2.0, 3.0 }},
                {{ -0.5,-3.0 },{ -2.0,-3.0 }},
                {{ -2.0,-5.0 },{ -3.0,-3.0 }}  };

TrackLine defaultTrackLines_OLD[defaultTrackLength] = {
                {{  0.0, 0.0 },{  1.0, 0.5 }},
                {{  0.0, 1.0 },{  1.0, 1.0 }},
                {{  0.0, 3.0 },{  1.0, 3.0 }},
                {{  1.5, 4.0 },{  3.0, 4.0 }},
                {{  1.0, 5.0 },{  2.0, 5.0 }},
                {{  1.0, 7.5 },{  2.0, 6.0 }},
                {{  3.0, 7.5 },{  3.0, 6.0 }},
                {{  5.0, 8.5 },{  5.0, 7.0 }},
                {{  8.0, 8.5 },{  8.0, 7.0 }},
                {{ 10.0, 7.5 },{ 10.0, 6.0 }},
                {{ 13.0, 4.0 },{ 11.0, 5.0 }},
                {{ 12.0, 2.0 },{ 11.0, 4.0 }},
                {{  9.0, 1.0 },{ 10.0, 3.0 }},
                {{  7.0, 3.0 },{  9.0, 4.0 }},
                {{  6.0, 5.0 },{  7.0, 6.0 }},
                {{  6.0, 5.0 },{  7.0, 6.0 }},
                {{  5.5, 5.0 },{  4.0, 6.0 }},
                {{  5.5, 0.0 },{  4.0, 0.0 }},
                {{  4.0,-2.0 },{  3.0, 0.0 }}  };

Track::Track()
{
                this->length = defaultTrackLength;
                trackLines  = defaultTrackLines;

                highlightedArray = new bool[this->length];

                trackIsSaved = true;
}

Track::Track(int numOfFields)
{
                this->length = numOfFields;
                trackLines  = new TrackLine[this->length];

                highlightedArray = new bool[this->length];

                trackIsSaved = false;
}

Track::~Track()
{
                if(trackLines != defaultTrackLines)
                {
                        delete [] trackLines;
                }
                delete [] highlightedArray;  //BUG - memory-mess, fucks up sometimes. Don't know why !!!
}

void Track::drawTrack()
{
                if(this->length == 0) return;

                int i;

                glLoadIdentity();

                glColor3fv(COLOR_BLUE);
                glPolygonMode(GL_FRONT, GL_FILL);
                glBegin(GL_QUAD_STRIP);
                        for(i=0; i<this->length; i++)
```

```
                        {
                                    glVertex2fv(trackLines[i].left);
                                    glVertex2fv(trackLines[i].right);
                        }
                        glVertex2fv(trackLines[0].left);
                        glVertex2fv(trackLines[0].right);
            glEnd();

            //highlight currentTrackField
            glColor3fv(COLOR_BLUE_BRIGHT);
            for(i=0; i<this->length; i++)
            {
                        if(highlightedArray[i])
                        {
                        glBegin(GL_QUAD_STRIP);
                                    glVertex2fv(trackLines[i].left);
                                    glVertex2fv(trackLines[i].right);
                                    if(i < this->length-1)
                                    {
                                                glVertex2fv(trackLines[i+1].left);
                                                glVertex2fv(trackLines[i+1].right);
                                    }
                                    else
                                    {
                                                glVertex2fv(trackLines[0].left);
                                                glVertex2fv(trackLines[0].right);
                                    }
                        glEnd();
                        }
            }

            glPolygonMode(GL_FRONT, GL_LINE);
            glColor3fv(COLOR_BLUE_DARK);
//          glColor3fv(COLOR_YELLOW);
            glBegin(GL_QUAD_STRIP);
                        for(i=0; i<this->length; i++)
                        {
                                    glVertex2fv(trackLines[i].left);
                                    glVertex2fv(trackLines[i].right);
                        }
                        glVertex2fv(trackLines[0].left);
                        glVertex2fv(trackLines[0].right);
            glEnd();

            glEnable(GL_LINE_STIPPLE);
            glLineWidth(3.0);
            glLineStipple(2, 0xcccc);
            glColor3fv(COLOR_YELLOW);
            glBegin(GL_LINES);
                        glVertex2fv(trackLines[0].left);
                        glVertex2fv(trackLines[0].right);
            glEnd();
            glDisable(GL_LINE_STIPPLE);
            glLineWidth(1.0);

            for(i=0; i<this->length; i++)
            {
                        highlightedArray[i] = false;
            }
}

void Track::setHighlightTrackElement(int pos)
{
            highlightedArray[pos] = true;
}



void Track::saveToFile()
{
            printf("Saving to File...");

            mkdir("tracks"); //just ignored, if allready exists !!!

            ofstream fout("tracks/NewTrack.track");
            for(int i=0; i<this->length; i++)
            {
                        fout << this->trackLines[i].left[0] << " ";
                        fout << this->trackLines[i].left[1] << " ";
                        fout << this->trackLines[i].right[0] << " ";
                        fout << this->trackLines[i].right[1] << "\n";
            }
            fout.close();


            printf("DONE!\n");
}

void Track::loadFromFile(int fileNum)
{
            printf("Load Track %i...", fileNum);

            extern Track *theTrack;
            extern Car *theCars[maxNumOfCars];

            WIN32_FIND_DATA data;
            HANDLE h = NULL;
            h = FindFirstFile("tracks\\*.track", &data);
            int i = 0;
            while(h != NULL && h != INVALID_HANDLE_VALUE)
            {
                        i++;
                        if(i == fileNum)
                        {
                                    printf("%s\n", data.cFileName);

                                    char path[256] = "tracks\\";
                                    ifstream fin(strncat(path, data.cFileName, 200));
                                    float tempFloat;
                                    if(trackLines != defaultTrackLines) delete [] trackLines;
                                    trackLines = new TrackLine[1000];
                                    length = 0;
                                    while(!fin.eof())
                                    {
```

```
                                        fin >> tempFloat;
                                        if(fin.eof()) break;
                                        trackLines[length].left[0] = tempFloat;
                                        fin >> trackLines[length].left[1];
                                        fin >> trackLines[length].right[0];
                                        fin >> trackLines[length].right[1];
                                        length++;
                                }
                                fin.close();
                                break;
                        }
                        if(!FindNextFile(h, &data)) h = NULL;
                }
                FindClose(h);

                for(i=0; i<maxNumOfCars; i++)
                {
                        if(theCars[i] != NULL) //found an existing Car
                        {
                                theCars[i]->restartOnTrack(i);
                        }
                }
}
```

NeuralDriver.cpp

```
#include "NeuralDriver.h"
#include "Constants.h"
#include "Helpers.h"
#include "Net.h"
#include <math.h>
#include <stdio.h>
//#include <GL/glut.h>  //just to get exit(0); !!!

#include <windows.h>


NeuralDriver::NeuralDriver(Net *apNetToUse)
{
                iLookAheadLength = (apNetToUse->GetInputLenght() - 1) / 4;
                pLookAheadArray = new LookAhead[iLookAheadLength];

                pMyNet = apNetToUse;

                pNetInputs  = new float[pMyNet->GetInputLenght()];
                pNetOutputs = new float[pMyNet->GetOutputLenght()];

                if(pMyNet->GetInputLenght() < 1 + 4*iLookAheadLength)
                {
                        printf("WARNING: Insufficient inputs to NN...!");
                }
}


NeuralDriver::NeuralDriver(int aiBrainFileNumber)
{
                printf("Load Brain: %i: ", aiBrainFileNumber);

                char fileNameWithPath[300] = "brains\\    ";

                WIN32_FIND_DATA data;
                HANDLE h = NULL;
                h = FindFirstFile("brains\\*.brain", &data);
                int i = 0;
                while(h != NULL && h != INVALID_HANDLE_VALUE && i<aiBrainFileNumber)
                {
                        i++;
                        if(!FindNextFile(h, &data)) h = NULL;
                } //loop stops, when correct file is found.

                for(i=0; i<256 && data.cFileName[i] != '\0'; i++) // adding the 'brains\' part to filename
                {
                        fileNameWithPath[i+7] = data.cFileName[i];
                }
                fileNameWithPath[i+8] = '\0';

                FindClose(h);

                printf("'%s'", fileNameWithPath);
                pMyNet = new Net(fileNameWithPath);

                iLookAheadLength = (pMyNet->GetInputLenght() - 1) / 4; //one for speed and 4 for each next line
                pLookAheadArray = new LookAhead[iLookAheadLength];

                printf(" ...LookAhead[%i]\n", iLookAheadLength);

                pNetInputs  = new float[pMyNet->GetInputLenght()];
                pNetOutputs = new float[pMyNet->GetOutputLenght()];
}


NeuralDriver::~NeuralDriver()
{
                delete [] pLookAheadArray;
                delete [] pNetInputs;
                delete [] pNetOutputs;
}


void NeuralDriver::decideSteering(float *speeder, float *steering, Track *theTrack, int currentTrackField, const float *carPosition, float
carDirection, float carSpeed)
{
                int nextLine = currentTrackField;
                int i;

                for(i=0; i<iLookAheadLength; i++)
```

```
            {
                        nextLine = (nextLine+1) % theTrack->length;

                        pLookAheadArray[i].left.angle  = angleBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].left)  -
carDirection;
                        pLookAheadArray[i].right.angle = angleBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].right) -
carDirection;

                        pLookAheadArray[i].left.distance  = distanceBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].left);
                        pLookAheadArray[i].right.distance = distanceBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].right);

                        if(pLookAheadArray[i].left.angle >  180) pLookAheadArray[i].left.angle -= 360;
                        if(pLookAheadArray[i].left.angle < -180) pLookAheadArray[i].left.angle += 360;

                        if(pLookAheadArray[i].right.angle >  180) pLookAheadArray[i].right.angle -= 360;
                        if(pLookAheadArray[i].right.angle < -180) pLookAheadArray[i].right.angle += 360;

/*                      printf("LookAhead[%i] = { a:%f   d:%f } , { a:%f   d:%f }\n", i,
                                    pLookAheadArray[i].left.angle,
                                    pLookAheadArray[i].left.distance,
                                    pLookAheadArray[i].right.angle,
                                    pLookAheadArray[i].right.distance ); */
            }

            pNetInputs[0] = carSpeed;
            for(i=0; i<iLookAheadLength; i++)
            {
                        pNetInputs[4*i +1] = pLookAheadArray[i].left.angle    / 180;
                        pNetInputs[4*i +2] = pLookAheadArray[i].left.distance  /  5;

                        pNetInputs[4*i +3] = pLookAheadArray[i].right.angle   / 180;
                        pNetInputs[4*i +4] = pLookAheadArray[i].right.distance /  5;
            }

            for(i=0; i<pMyNet->GetInputLenght(); i++)  // normalize (aka cut high values)
            {
                        if(pNetInputs[i] >  1.0f) pNetInputs[i] =  1.0f;
                        if(pNetInputs[i] < -1.0f) pNetInputs[i] = -1.0f;
            }


            pMyNet->SetInput(pNetInputs);
            pMyNet->PropagateNet();
            pMyNet->GetOutput(pNetOutputs);

            *steering = pNetOutputs[0];
            *speeder  = pNetOutputs[1];
//          *speeder  = 2*pNetOutputs[1];  //small cheat
//          *speeder  = 1.0f;              //cheating...
}



Net* NeuralDriver::getNet()
{
            return pMyNet;
}

Driver* NeuralDriver::clone()
{
            return new NeuralDriver(pMyNet->clone());
}

void NeuralDriver::saveBrain()
{
            pMyNet->SaveNet("brains\\NewBrain.brain");
            printf("Brain saved\n");
}
Adaline.cpp

#include "EvolutionLearner.h"
#include "Track.h"
#include "Car.h"
#include "Constants.h"
#include "NeuralDriver.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "Adaline.h"
#include "Net.h"
#include <iostream.h>

Adaline::Adaline(int aiHowManyLayers, int *apiHowManyNeuronsInLayers)
{
            cout<<"new Adaline Network"<<endl;
            iLayersInNN        = aiHowManyLayers;

            piNNStructure = new int[aiHowManyLayers];
            for(int i=0; i<aiHowManyLayers; i++)
            {
                        piNNStructure[i] = apiHowManyNeuronsInLayers[i];
            }
}


Net* Adaline::CreateAdalineNet()
{
            Net* net;
            net = new Net(iLayersInNN, piNNStructure);

            net->TrainAdaNet(iLayersInNN, piNNStructure, 1);

            return net;
}
```

Bilag 2, Net Source-code

Net.h

```
#ifndef NET
#define NET

class Net
{
	public:
		Net(char *apcFileName);
		Net(int aiHowManyLayers, int *apiHowManyNeuronsInLayers);
		~Net();
		void SetInput(float *apfInput);
		void GetOutput(float *apfOutput);
		void SaveWeights();
		void RestoreWeights();
		void PropagateNet();
		void ComputeOutputError(float *afTarget);
		bool SaveNet(char *apcFileName);
		void BackPropagateNet();

		int GetInputLenght();  //added by Martin
		int GetOutputLenght(); //added by Martin

		static Net* CreateNetFromParents(Net* apoMotherNet, Net* apoFatherNet); //added by Martin
		void mutate(float mutationProbability);                                 //added by Martin
		Net* clone();                                                           //added by Martin
		bool equals(Net* compareNet);                                           //added by Martin


	private:
		class Layer
		{
			public:
				class Neuron
				{
					public:

						Neuron();

						void Setup(int aiNeuronNumber, int aiHowManyPrevNeurons, Neuron
*apPrevNeurons, int aiHowManyNextNeurons, Neuron *apNextNeurons);
						void SetInput(float afInput);
						float GetOutput();
						void SaveWeights();
						void RestoreWeights();
						void PropagateNeuron();

						int iNeuronNumber;

						int iHowManyPrevNeurons;
						Neuron *pPrevNeurons;

						int iHowManyNextNeurons;
						Neuron *pNextNeurons;

						float *pfWeightToPrevLayerOfNeurons;
						float *pfWeightToPrevLayerOfNeuronsSaved;
						float *pfDWeightToPrevLayerOfNeurons;
						float fOutput;
						float fError;

				};

				Neuron *pTheNeurons;
				int iHowManyNeurons;

				Layer();
				void Setup(int aiHowManyNeurons);
				void SetNeighbours(int aiHowManyPrevNeurons, Neuron *PrevNeurons, int aiHowManyNextNeurons,
Neuron *NextNeurons);
				void SetInputOnNeurons(float *apfInput);
				void GetOutput(float *apfOutput);
				void SaveWeights();
				void RestoreWeights();
				void PropagateLayer();
				void BackPropagateLayer();

		};

		Layer *pTheLayers;
		int iHowManyLayers;
		Layer *pInputLayer;
		Layer *pOutputLayer;
		float fAlpha;
		float fEta;
		float fGain;
		float fError;
};

#endif //ndef NET
```

Net.cpp

```
#include "Net.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <windows.h>
#include <winbase.h>

#define NULL 0

#define BIAS 1

//------------------------------------------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------------------------------------------

Net::Net(char *apcFileName)
{
            FILE *pDumpFile = fopen("dumps\\LoadNetDump.txt", "w"); //DEBUG:

            HANDLE hFile = CreateFile(apcFileName, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_EXISTING, NULL, NULL);
            if(hFile==INVALID_HANDLE_VALUE)
                        return;

            BY_HANDLE_FILE_INFORMATION hFileInfo;
            //get the file infomation
            if(!GetFileInformationByHandle(hFile, &hFileInfo))                                                                  //
                        return;
                                                                  //

            unsigned long iReadOfFile=0;
            int *piTheFile = new int[hFileInfo.nFileSizeLow];   //entire file in one int-array!
            if(!ReadFile(hFile, piTheFile, hFileInfo.nFileSizeLow, &iReadOfFile, NULL))
                        return;

            iHowManyLayers = piTheFile[0];
            fprintf(pDumpFile, "%i\n", iHowManyLayers);  //DEBUG:
            printf("{%i: ", iHowManyLayers);
            int *piHowManyNeuronsInLayers = new int[iHowManyLayers];
            for(int i=0; i<iHowManyLayers; i++)
            {
                        piHowManyNeuronsInLayers[i] = piTheFile[i+1];
                        fprintf(pDumpFile, "%i\n", piHowManyNeuronsInLayers[i]);  //DEBUG:
                        if(i>0) printf(",");
                        printf("%i", piHowManyNeuronsInLayers[i]);
            }
            printf("}");

            pTheLayers = new Layer[iHowManyLayers];

            for(i=0; i<iHowManyLayers; i++)
                        pTheLayers[i].Setup(piHowManyNeuronsInLayers[i]);

            for(i=0; i<iHowManyLayers; i++)
            {
                        if(i==0)
                                    pTheLayers[i].SetNeighbours(0, NULL, pTheLayers[i+1].iHowManyNeurons, pTheLayers[i+1].pTheNeurons);
                        else if(i==iHowManyLayers-1)
                                    pTheLayers[i].SetNeighbours(pTheLayers[i-1].iHowManyNeurons, pTheLayers[i-1].pTheNeurons, 0, NULL);
                        else
                                    pTheLayers[i].SetNeighbours(pTheLayers[i-1].iHowManyNeurons, pTheLayers[i-1].pTheNeurons,
pTheLayers[i+1].iHowManyNeurons, pTheLayers[i+1].pTheNeurons);
            }

            pInputLayer  = &pTheLayers[0];
            pOutputLayer = &pTheLayers[iHowManyLayers-1];

            fAlpha = 0.5f;
            fEta = 0.05f;
            fGain = 1.0f;

//          printf("a");
            int offset=0; int p;
            for(i=1; i<iHowManyLayers; i++)
            {
                        for(p=0; p<pTheLayers[i].iHowManyNeurons; p++)
            //piHowManyNeuronsInLayers[i]
                        {
                                    pTheLayers[i].pTheNeurons[p].pfWeightToPrevLayerOfNeurons = (float*)&piTheFile[1 + iHowManyLayers +
p*pTheLayers[i-1].iHowManyNeurons+offset];
                                    for(int w=0; w<pTheLayers[i].pTheNeurons[p].iHowManyPrevNeurons; w++)                       //debug
                                    {                                                                                           //debug
                                                fprintf(pDumpFile, "%f ", pTheLayers[i].pTheNeurons[p].pfWeightToPrevLayerOfNeurons[w]);
//debug
                                    }                                                                                           //debug
                                    fprintf(pDumpFile, "\n");                                                                   //debug
//                                  printf("i");
                        }
                        offset+= p*pTheLayers[i-1].iHowManyNeurons;
            }
//          printf("b");

            delete piHowManyNeuronsInLayers;
            CloseHandle(hFile);

            fclose(pDumpFile);

}


Net::Net(int aiHowManyLayers, int *apiHowManyNeuronsInLayers)
{
            iHowManyLayers = aiHowManyLayers;

            pTheLayers = new Layer[iHowManyLayers];

            for(int i=0; i<iHowManyLayers; i++)
                        pTheLayers[i].Setup(apiHowManyNeuronsInLayers[i]);

            for(i=0; i<iHowManyLayers; i++)
            {
```

```
                        if(i==0)                        //first layer
                                    pTheLayers[i].SetNeighbours(0, NULL, pTheLayers[i+1].iHowManyNeurons, pTheLayers[i+1].pTheNeurons);
                        else if(i==iHowManyLayers-1)  //last layer
                                    pTheLayers[i].SetNeighbours(pTheLayers[i-1].iHowManyNeurons, pTheLayers[i-1].pTheNeurons, 0, NULL);
                        else                                    //middle layer
                                    pTheLayers[i].SetNeighbours(pTheLayers[i-1].iHowManyNeurons, pTheLayers[i-1].pTheNeurons,
pTheLayers[i+1].iHowManyNeurons, pTheLayers[i+1].pTheNeurons);
            }

            pInputLayer = &pTheLayers[0];
            pOutputLayer = &pTheLayers[iHowManyLayers-1];

            fAlpha = 0.5;
            fEta = 0.05f;
            fGain = 1;

//          printf("Nyt Net...\n");
}

Net::~Net()
{

}


bool Net::SaveNet(char *apcFileName)
{
            FILE *pDumpFile = fopen("dumps\\SaveNetDump.txt", "w"); //DEBUG:


            HANDLE hFile = CreateFile(apcFileName, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS, NULL, NULL);
            if(hFile==INVALID_HANDLE_VALUE)
                        return false;

            unsigned long iWrittenBytes=0;
            if(!WriteFile(hFile, &iHowManyLayers, sizeof(int), &iWrittenBytes, NULL))
                        return false;
            fprintf(pDumpFile, "%i\n", iHowManyLayers); //debug


            for(int q=0; q<iHowManyLayers; q++)
            {
                        if(!WriteFile(hFile, &pTheLayers[q].iHowManyNeurons, sizeof(int), &iWrittenBytes, NULL))
                                    return false;
                        fprintf(pDumpFile, "%i\n", pTheLayers[q].iHowManyNeurons); //debug
            }


            for(int i=1; i<iHowManyLayers; i++)
            {
                        for(int p=0; p<pTheLayers[i].iHowManyNeurons; p++)
                        {
                                    for(int w=0; w<pTheLayers[i].pTheNeurons[p].iHowManyPrevNeurons; w++)
                                    {
                                                if(!WriteFile(hFile, &pTheLayers[i].pTheNeurons[p].pfWeightToPrevLayerOfNeurons[w],
sizeof(float), &iWrittenBytes, NULL))           //
                                                            return false;
                                                fprintf(pDumpFile, "%f ", pTheLayers[i].pTheNeurons[p].pfWeightToPrevLayerOfNeurons[w]);
//debug
                                    }
                                    fprintf(pDumpFile, "\n"); //debug
                        }
            }

            CloseHandle(hFile);

            fclose(pDumpFile);

            return true;
}

/*
void Net::SaveNet()
{
            FILE *fileOutput = NULL;

            fileOutput = fopen("netWork.txt", "w");
            if(fileOutput == NULL)
                        return;

            fprintf(fileOutput, "%d ", iHowManyLayers);
            for(int q=0; q<iHowManyLayers; q++)
                        fprintf(fileOutput, "%d ", pTheLayers[q].iHowManyNeurons);

            fprintf(fileOutput, "\n");

            for(int i=1; i<iHowManyLayers; i++)
            {
                        for(int p=0; p<pTheLayers[i].iHowManyNeurons; p++)
                        {
                                    for(int w=0; w<pTheLayers[i].pTheNeurons[p].iHowManyPrevNeurons; w++)
                                    {
                                                fprintf(fileOutput, "%f", pTheLayers[i].pTheNeurons[p].pfWeightToPrevLayerOfNeurons[w]);
                                                fprintf(fileOutput, ";");
                                    }
                        }
                        fprintf(fileOutput, "\n");
            }

            fclose(fileOutput);
}
*/
/*
void Net::LoadNet(char *apcFileName)
{
            FILE *fileInput = NULL;

            fileInput = fopen(apcFileName, "r");
            if(fileInput == NULL)
                        return;

            char pcTemp[50];
            char cTemp;
```

```
            cTemp = (char) fgetc(fileInput);
            int i;
            for(i=0; cTemp!='\n'; i++)
            {
                        pcTemp[i] = cTemp;
                        cTemp = fgetc(fileInput);
            }


            fclose(fileInput);
}
*/

void Net::SetInput(float *apfInput)
{
            pInputLayer->SetInputOnNeurons(apfInput);
}

void Net::GetOutput(float *apfOutput)
{
            pOutputLayer->GetOutput(apfOutput);
}

void Net::SaveWeights()
{
            for(int i=0; i<iHowManyLayers; i++)
                        pTheLayers->SaveWeights();
}

void Net::RestoreWeights()
{
            for(int i=0; i<iHowManyLayers; i++)
                        pTheLayers->RestoreWeights();
}

void Net::PropagateNet()
{
            for(int i=1; i<iHowManyLayers; i++)
                        pTheLayers[i].PropagateLayer();
}

void Net::ComputeOutputError(float *afTarget)
{
            float fOut, fErr;

            fError = 0;

            for(int i=0; i<pOutputLayer->iHowManyNeurons; i++)
            {
                        fOut = pOutputLayer->pTheNeurons[i].GetOutput();
                        fErr = afTarget[i]-fOut;
                        pOutputLayer->pTheNeurons[i].fError = fGain * fOut * (1-fOut) * fErr;
                        fError += 0.5f * fErr*fErr;
            }
}

void Net::BackPropagateNet()
{

}



//--------------------------------------------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------------------------------------------



//--------------------------------------------------------------------------------------------------------------------------
//--------------------------------------------------------------------------------------------------------------------------
Net::Layer::Layer()
{

}

void Net::Layer::Setup(int aiHowManyNeurons)
{
            iHowManyNeurons = aiHowManyNeurons;

            pTheNeurons = new Neuron[iHowManyNeurons];
}


void Net::Layer::SetNeighbours(int aiHowManyPrevNeurons, Neuron *PrevNeurons, int aiHowManyNextNeurons, Neuron *NextNeurons)
{
            for(int i=0; i<iHowManyNeurons; i++)
                        pTheNeurons[i].Setup(i, aiHowManyPrevNeurons, PrevNeurons, aiHowManyNextNeurons, NextNeurons);
}

void Net::Layer::SetInputOnNeurons(float *apfInput)
{
            for(int i=0; i<iHowManyNeurons; i++)
                        pTheNeurons[i].SetInput(apfInput[i]);
}

void Net::Layer::GetOutput(float *apfOutput)
{
//          if(apfOutput!=NULL)
//                      delete apfOutput;

//          apfOutput = new float[iHowManyNeurons];

            for(int i=0; i<iHowManyNeurons; i++)
                        apfOutput[i] = pTheNeurons[i].GetOutput();
}

void Net::Layer::SaveWeights()
{
            for(int i=0; i<iHowManyNeurons; i++)
                        pTheNeurons[i].SaveWeights();
}
```

```
void Net::Layer::RestoreWeights()
{
            for(int i=0; i<iHowManyNeurons; i++)
                        pTheNeurons[i].RestoreWeights();
}

void Net::Layer::PropagateLayer()
{
            for(int i=0; i<iHowManyNeurons; i++)
                        pTheNeurons[i].PropagateNeuron();
}

void Net::Layer::BackPropagateLayer()
{

}

//---------------------------------------------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------------------------------------------


//---------------------------------------------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------------------------------------------
Net::Layer::Neuron::Neuron()
{
            fOutput = BIAS;
}

void Net::Layer::Neuron::Setup(int aiNeuronNumber, int aiHowManyPrevNeurons, Neuron *apPrevNeurons, int aiHowManyNextNeurons, Neuron
*apNextNeurons)
{
            iNeuronNumber       = aiNeuronNumber;
            iHowManyPrevNeurons = aiHowManyPrevNeurons;
            pPrevNeurons        = apPrevNeurons;
            iHowManyNextNeurons = aiHowManyNextNeurons;
            pNextNeurons        = apNextNeurons;

            pfWeightToPrevLayerOfNeurons = new float[iHowManyPrevNeurons];

            for(int i=0; i<iHowManyPrevNeurons; i++)
            {
//                      pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))-0.5f);         // between -0.5 and +0.5
//                      pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))*4.0f - 2.0f);  // between -2.0 and +2.0
//                      pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))*8.0f - 4.0f);  // between -4.0 and +4.0
                        pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))*32.0f - 16.0f); // between -16 and +16
            }
            fOutput = 0.0;
}

void Net::Layer::Neuron::SetInput(float afInput)
{
            fOutput = afInput;
}

float Net::Layer::Neuron::GetOutput()
{
            return fOutput;
}

void Net::Layer::Neuron::SaveWeights()
{
            for(int i=0; i<iHowManyNextNeurons; i++)
                        pfWeightToPrevLayerOfNeuronsSaved[i] = pfWeightToPrevLayerOfNeurons[i];
}

void Net::Layer::Neuron::RestoreWeights()
{
            for(int i=0; i<iHowManyNextNeurons; i++)
                        pfWeightToPrevLayerOfNeurons[i] = pfWeightToPrevLayerOfNeuronsSaved[i];
}

void Net::Layer::Neuron::PropagateNeuron()
{
            fOutput = 0.0;

            for(int i=0; i<iHowManyPrevNeurons; i++)
                        fOutput += pfWeightToPrevLayerOfNeurons[i] * pPrevNeurons[i].GetOutput();

//          //overføringsfunktion: linear mellem -1 og +1
            fOutput /= 2.0f;
            if(fOutput >  1.0f) fOutput =  1.0f;
            if(fOutput < -1.0f) fOutput = -1.0f;

//          //overføringsfunktion: sigmoid mellem -1 og +1
//          fOutput = (2.0f / (1.0f + (float)exp(-fOutput))) - 1.0f;
}
//---------------------------------------------------------------------------------------------------------------------------
//---------------------------------------------------------------------------------------------------------------------------
```

Bilag 3, Adalain Source-code

Net_AdaXtra.cpp

```
#include "Net.h"
#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
#include <iomanip.h>

void Net::TrainAdaNet(int aiHowManyLayers, int *apiHowManyNeuronsInLayers, int iLookAheadLength )
{
            float errorMax= 0.3f;
            float netError= 0.4f;
            int iInputLength = 4*iLookAheadLength + 1;
            float* data = new float[iInputLength];
            float pNetOutputs[2];
            float learnRate= .00255f;
            float speeder, steering;

//          while (errorMax < netError)
            {
                        cout<<"Training the adaline Car."<<endl;
                        ifstream infile;
                        infile.open("carDump.txt");
                        while (infile)
                        {
                                    for(int i=0; i<iInputLength; i++)
                                    {
                                                infile>> data[i];
                                    }
                                    infile >> speeder>> steering;

                                    SetInput(data);
                                    PropagateNet();
                                    GetOutput(pNetOutputs);
                                    float error1 = pNetOutputs[0] - steering;
                                    float error2 = pNetOutputs[1] - speeder;

                                    if (error1>netError)
                                                netError = error1;
                                    else if (error2>netError)
                                                netError = error2;
                                    //cout<<netError;
                                    float tempWeight;
                                    for(int layerIndex=0; layerIndex < aiHowManyLayers; layerIndex++)
                                    {
                                                for(int neuronIndex=0; neuronIndex < pTheLayers[layerIndex].iHowManyNeurons; neuronIndex++)
                                                {
                                                            for(int weightIndex=0; weightIndex <
pTheLayers[layerIndex].pTheNeurons[neuronIndex].iHowManyPrevNeurons; weightIndex++)
                                                            {
                                                                        tempWeight =
pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex];
                                                                        tempWeight += (error2*learnRate*pNetOutputs[1]);
                                                                        tempWeight += (error1*learnRate*pNetOutputs[0]);


            pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex] = tempWeight;
                                                            }
                                                }
                                    }

                        }
                        infile.close();
            }

}
```

Car_AdaXtra.cpp

```
#include "Car.h"
#include "CarTrail.h"
#include "Driver.h"
#include "Track.h"
#include "Constants.h"
#include "Helpers.h"
#include "DecisionLines.h"
//#include "InputHandlers.h"
#include <iostream.h>
#include <GL/glut.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <strstrea.h>
#include <iomanip.h>

#include <iostream.h>
#include <fstream.h>
#include <assert.h>
#include <string>
#include <iomanip.h>
#include "Constants.h"
#include <fstream.h>
#include <math.h>

void Car::recordCar(Track *theTrack, int nextLine)
{
```

```
            nextLine = (nextLine+1) % theTrack->length;

            int iLookAheadLength = 1;
            LookAhead* pLookAheadArray = new LookAhead[iLookAheadLength];

            int iInputLenght = iLookAheadLength*4;
            float*      pNetInputs  = new float[iInputLenght];


            for(int i=0; i<iLookAheadLength; i++)
            {
                        nextLine = (nextLine+1) % theTrack->length;

                        pLookAheadArray[i].left.angle  = angleBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].left)  -
carDirection;
                        pLookAheadArray[i].right.angle = angleBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].right) -
carDirection;

                        pLookAheadArray[i].left.distance  = distanceBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].left);
                        pLookAheadArray[i].right.distance = distanceBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].right);

                        if(pLookAheadArray[i].left.angle >  180) pLookAheadArray[i].left.angle -= 360;
                        if(pLookAheadArray[i].left.angle < -180) pLookAheadArray[i].left.angle += 360;

                        if(pLookAheadArray[i].right.angle >  180) pLookAheadArray[i].right.angle -= 360;
                        if(pLookAheadArray[i].right.angle < -180) pLookAheadArray[i].right.angle += 360;
            }


            for(i=0; i<iLookAheadLength; i++)
            {
                        pNetInputs[4*i] = pLookAheadArray[i].left.angle    / 180;
                        pNetInputs[4*i +1] = pLookAheadArray[i].left.distance  /  5;
                        pNetInputs[4*i +2] = pLookAheadArray[i].right.angle   / 180;
                        pNetInputs[4*i +3] = pLookAheadArray[i].right.distance /  5;
            }

            for(i=0; i<iInputLenght; i++)  // normalize (aka cut high values)
            {
                        //printf("Inputs: %f\n", pNetInputs[i] );
                        if(pNetInputs[i] >  1.0f) pNetInputs[i] =  1.0f;
                        if(pNetInputs[i] < -1.0f) pNetInputs[i] = -1.0f;
            }

            outfile << carSpeed <<" ";

            for(i=0; i<iInputLenght; i++)
            {
                        outfile << pNetInputs[i]<<" ";
            }
            outfile << speeder<<" "
            //Correct speeder
                                    << steering<<" "
            //Correct Steering
                                    <<endl;


}


void Car::recordDecisions(char *fileName, int numberOfGameTicksToRecord)
{
            cout<<"Recording in progress"<<endl;
            outfile.open(fileName);
            outCountDown=numberOfGameTicksToRecord;
}
```

Bilag 4, Backpropagation Source-code

Net.cpp

```
void Net::ATComputeOutputError(float *afTarget)
{
	float fOut, fErr;

	fError = 0;

	for(int i=0; i<pOutputLayer->iHowManyNeurons; i++)
	{
		fOut = pOutputLayer->pTheNeurons[i].GetOutput();
		fErr = afTarget[i]-fOut;
		pOutputLayer->pTheNeurons[i].fError = fGain * fOut * (1-fOut) * fErr;
		fError += 0.5f * fErr*fErr;
	}
}

void Net::ATPropagateNet()
{
	for(int i=iHowManyLayers-1; i>1; i--)
		pTheLayers[i].ATPropagateLayer(fGain);
}

void Net::ATAdjustWieghts()
{
	float fErr, fOut, fDWeight;

	for(int l=1; l<iHowManyLayers; l++)
	{
		for(int i=0; i<pTheLayers[l].iHowManyNeurons; i++)
		{
			for(int j=0; j<pTheLayers[l-1].iHowManyNeurons; j++)
			{
				fOut = pTheLayers[l-1].pTheNeurons[j].fOutput;
				fErr = pTheLayers[l].pTheNeurons[i].fError;
				fDWeight = pTheLayers[l].pTheNeurons[i].pfDWeightToPrevLayerOfNeurons[j];

				pTheLayers[l].pTheNeurons[i].pfWeightToPrevLayerOfNeurons[j] += fEta * fErr * fOut + fAlpha *
fDWeight;
				pTheLayers[l].pTheNeurons[i].pfDWeightToPrevLayerOfNeurons[j] = fEta * fErr * fOut;

			}
		}
	}
}

bool Net::SimulateNet(float *apfInput, float *apfOutput, float *apfTarget, bool bTraning)
{
	SetInput(apfInput);
	PropagateNet();
	GetOutput(apfOutput);

	ATComputeOutputError(apfTarget);

	if(bTraning)
	{
		ATPropagateNet();
		ATAdjustWieghts();
	}
	return false;
}

float Net::ATGetNetError()
{
	return fError;
}


void Net::Layer::Neuron::Setup(int aiNeuronNumber, int aiHowManyPrevNeurons, Neuron *apPrevNeurons, int aiHowManyNextNeurons, Neuron
*apNextNeurons)
{
	iNeuronNumber       = aiNeuronNumber;
	iHowManyPrevNeurons = aiHowManyPrevNeurons;
	pPrevNeurons        = apPrevNeurons;
	iHowManyNextNeurons = aiHowManyNextNeurons;
	pNextNeurons        = apNextNeurons;

	pfWeightToPrevLayerOfNeurons = new float[iHowManyPrevNeurons];
	pfDWeightToPrevLayerOfNeurons = new float[iHowManyPrevNeurons];
	pfWeightToPrevLayerOfNeuronsSaved = new float[iHowManyPrevNeurons];

	for(int i=0; i<iHowManyPrevNeurons; i++)
	{
//		pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))-0.5f);        // between -0.5 and +0.5
//		pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))*8.0f - 4.0f); // between -4.0 and +4.0
		pfWeightToPrevLayerOfNeurons[i] = ((rand()/(RAND_MAX*1.0f))*64.0f - 32.0f); // between -4.0 and +4.0
		pfWeightToPrevLayerOfNeuronsSaved [i] = pfWeightToPrevLayerOfNeurons[i];
		pfDWeightToPrevLayerOfNeurons[i] = pfWeightToPrevLayerOfNeurons[i];
	}
	fOutput = 0.0;
}
```

SimpleDriver.cpp

```
int iWhatToRun2 = 0;
NeuralDriver *ATDriver2 = 0;
Net *ATNet2 = 0;

bool bDoSkip = false;

int iHowMuchTestDataUsed2 = 0;
float *pfTestData2 = 0;

#define MAX_TESTDATA 7000

void SimpleDriver::decideSteering(float *speeder, float *steering, Track *theTrack, int currentTrackField, const float *carPosition, float
carDirection, float carSpeed)
{
            if(pfTestData2 == 0)
            {
                        pfTestData2 = new float[MAX_TESTDATA];                                  //added by Adrian
                        iHowMuchTestDataUsed2 = 0;                                                                      //added by Adrian
            }

            if(iWhatToRun2 == 0)
            {

                        float leftAngleMax =   180.0f;
                        float rightAngleMax = -180.0f;

                        float leftAngleTemp =  leftAngleMax;
                        float rightAngleTemp = rightAngleMax;
                        float leftDistance;
                        float rightDistance;

                        float leftangle[4];
                        float rightangle[4];
                        float leftdistance[4];
                        float rightdistance[4];

                        float newDirection;

                        int nextLine = currentTrackField;
                        int nextLine2 = currentTrackField;

                        bool keepLooking = true;
                        int lookAheadLength = 0;

            nextLine = (currentTrackField+1) % theTrack->length;

            float nextMid[2];
            nextMid[0] = (theTrack->trackLines[nextLine].left[0] + theTrack->trackLines[nextLine].right[0]) / 2;
            nextMid[1] = (theTrack->trackLines[nextLine].left[1] + theTrack->trackLines[nextLine].right[1]) / 2;

            float angleToNextGoal = angleBetweenTwoPoints(carPosition, nextMid);
            float angleDiff = angleToNextGoal - carDirection;

            if(angleDiff >  180) angleDiff -= 360;
            if(angleDiff < -180) angleDiff += 360;

            *steering = angleDiff / 10;  //+  (((float)rand()) / RAND_MAX) - 0.5f;
            *speeder  = 1.0f - (float)abs((int)angleDiff) / 45;



            for(int i=0; i<4; i++)
            {
                        nextLine2 = (nextLine2+1) % theTrack->length;

                        leftangle[i]  = angleBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].left)  - carDirection;
                        rightangle[i] = angleBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].right) - carDirection;

                        leftdistance[i]  = distanceBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].left);
                        rightdistance[i] = distanceBetweenTwoPoints(carPosition, theTrack->trackLines[nextLine].right);

                        if(leftangle[i] >  180) leftangle[i] -= 360;
                        if(leftangle[i] < -180) leftangle[i] += 360;

                        if(rightangle[i] >  180) rightangle[i] -= 360;
                        if(rightangle[i] < -180) rightangle[i] += 360;


                        leftdistance[i]/=5;
                        rightdistance[i]/=5;
                        if(leftdistance[i]  >  1.0f)        leftdistance[i] = 1.0f;
                        if(rightdistance[i] >  1.0f)        rightdistance[i] = 1.0f;
                        if(leftdistance[i]  < -1.0f)        leftdistance[i] = -1.0f;
                        if(rightdistance[i] < -1.0f)        rightdistance[i] = -1.0f;
            }

            if(bDoSkip)
            {
                        if(iHowMuchTestDataUsed2 < 5700)                //MAX_TESTDATA
                        {
                                    pfTestData2[iHowMuchTestDataUsed2 + 0] = carSpeed/30.0f;
                                    pfTestData2[iHowMuchTestDataUsed2 + 1] = leftangle[0]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 + 2] = leftdistance[0];
                                    pfTestData2[iHowMuchTestDataUsed2 + 3] = rightangle[0]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 + 4] = rightdistance[0];

                                    pfTestData2[iHowMuchTestDataUsed2 + 5] = leftangle[1]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 + 6] = leftdistance[1];
                                    pfTestData2[iHowMuchTestDataUsed2 + 7] = rightangle[1]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 + 8] = rightdistance[1];

                                    pfTestData2[iHowMuchTestDataUsed2 + 9] = leftangle[2]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 +10] = leftdistance[2];
                                    pfTestData2[iHowMuchTestDataUsed2 +11] = rightangle[2]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 +12] = rightdistance[2];

                                    pfTestData2[iHowMuchTestDataUsed2 +13] = leftangle[2]/180;
```

```
                                    pfTestData2[iHowMuchTestDataUsed2 +14] = leftdistance[2];
                                    pfTestData2[iHowMuchTestDataUsed2 +15] = rightangle[2]/180;
                                    pfTestData2[iHowMuchTestDataUsed2 +16] = rightdistance[2];

                                    pfTestData2[iHowMuchTestDataUsed2 +17] = *speeder;
                                    pfTestData2[iHowMuchTestDataUsed2 +18] = *steering;

                                    iHowMuchTestDataUsed2+=19;
                          }
                          else
                                    iWhatToRun2 = 1;

                          bDoSkip = false;

                }
                else
                          bDoSkip = true;
//                        fprintf(FileSave2, "%f;%f;%f;%f;%f;%f;%f\n", carSpeed/30.0f, leftAngleTemp/180, leftDistance, rightAngleTemp/180,
rightDistance, *speeder, *steering);
                }
                else if(iWhatToRun2 == 1)
                {
                          int piLayers[] = {17, 80, 2};
                          ATNet2 = new Net(3, piLayers);
                          ATDriver2 = new NeuralDriver(ATNet2, 3);

                          bool bStop = false;
                          while(!bStop)//for(int q=0;q<1; q++)
                                    bStop = ATDriver2->TrainDriver(5700, pfTestData2);          //MAX_TESTDATA

//                        printf("ATNet error: %f\n", ATNet->ATGetNetError());
                          iWhatToRun2 = 2;
                }
                else if(iWhatToRun2 == 2)
                {
                          ATDriver2->decideSteering(speeder, steering, theTrack, currentTrackField, carPosition, carDirection, carSpeed);
                }
}
```

NeuralDriver.cpp

```
bool NeuralDriver::TrainDriver(int aiHowLongTestData, float *apfTestData)
{
                float fErrorSteering = 0.0f;
                float fErrorSpeed = 0.0f;
                float *pfTempInput = 0;
                float *pfOutput = new float[2];     //pMyNet->GetOutLenght()
                float *pfTarget = new float[2];     //pMyNet->GetOutLenght()

                bool bStop = false;
                for(int i=0; i<aiHowLongTestData-19; i+=(pMyNet->GetInputLenght()+2))
                {
                          pfTempInput = apfTestData + i;
                          pfTarget[0] = *(pfTempInput+17);
                          pfTarget[1] = *(pfTempInput+18);
                          bStop = pMyNet->SimulateNet(pfTempInput, pfOutput, pfTarget, true);

/*
                          if(((apfOutput[0] + fMargin) > apfTarget[0] && (apfOutput[0] - fMargin) < apfTarget[0]) && ((apfOutput[1] + fMargin) >
apfTarget[1] && (apfOutput[1] - fMargin) < apfTarget[1]))
                                    return true;*/
                          float temp1 = pfOutput[0];
                          float temp2 = pfOutput[1];


                          if((pfOutput[0] - pfTarget[0]) < 0)
                                    fErrorSteering += (pfOutput[0] - pfTarget[0])*-1.0f;
                          else
                                    fErrorSteering += pfOutput[0] - pfTarget[0];

                          if((pfOutput[1] - pfTarget[1]) < 0)
                                    fErrorSpeed += (pfOutput[1] - pfTarget[1])*-1.0f;
                          else
                                    fErrorSpeed += pfOutput[1] - pfTarget[1];

                          if(i==2090)
                                    int q = 100;

                          if(fErrorSteering > 1000)
                                    int p = 100;
                }

                float fMargin = 2.1;
                fErrorSteering/= (aiHowLongTestData/(pMyNet->GetInputLenght()+2));
                fErrorSpeed/= (aiHowLongTestData/(pMyNet->GetInputLenght()+2));

                printf("fErrorSteering: %f\t", fErrorSteering);
                printf("fErrorSpeed: %f\n", fErrorSpeed);

                pMyNet->SaveNetHuman("humanread.txt");

                if(fErrorSteering < fMargin)
                          return true;

                return false;
}
```

Bilag 5, Evolution Source-code

EvolutionLearner.cpp

```
#include "EvolutionLearner.h"
#include "Track.h"
#include "Car.h"
#include "Constants.h"
#include "NeuralDriver.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>


EvolutionLearner::EvolutionLearner(int aiHowManyLayers, int *apiHowManyNeuronsInLayers)
{
            printf("new EvolutionLearner...\n");
            iPopulationSize    = 0;
            iGenerationsNumber = 0;
            iBestPoints        = 0;
            iAveragePoints     = 0;
            iWorstPoints       = 0;
            iLayersInNN        = aiHowManyLayers;
            iLookAheadLength   = (apiHowManyNeuronsInLayers[0] - 1) / 4;

            pDumpFile1 = fopen("dumps\\EvoDump.csv", "w");
            pDumpFile2 = fopen("dumps\\EvoDump_resume.csv", "w");
            fprintf(pDumpFile2, "gen max avg min\n");

            piNNStructure = new int[aiHowManyLayers];
            for(int i=0; i<aiHowManyLayers; i++)
            {
                        piNNStructure[i] = apiHowManyNeuronsInLayers[i];
            }
}


void EvolutionLearner::CreateRandomizedPopulation(int aiSize, int aiSRandValue)
{
            extern Track *theTrack;

            printf("CreateRandomizedPopulation...\n");

            iGenerationsNumber = 0;
            ppCarsInPopulation = new Car*[aiSize];
            iPopulationSize    = aiSize;

            if(aiSRandValue > 0) srand(aiSRandValue);

            for(int i=0; i<aiSize; i++)
            {
                        ppCarsInPopulation[i] = new Car(i);
                        ppCarsInPopulation[i]->setDriver(new NeuralDriver(new Net(iLayersInNN, piNNStructure)));
                        ppCarsInPopulation[i]->addToDevHistory('r');

            }
            iGenerationsNumber = 1;
}



void EvolutionLearner::EvaluateGeneration(int aiTimeTicksToLive)
{
            extern Track *theTrack;

            printf("EvaluateGeneration   %i:", iGenerationsNumber);
            iBestPoints  = 0;
            iWorstPoints = 1000000000; //very high number
            int sumOfPoints  = 0;
            for(int i=0; i<iPopulationSize; i++)
            {
                        if(i%1000 == 0 && i>0) printf(" %i\n", i);
                        else if(i%20 == 0) printf(".");

                        ppCarsInPopulation[i]->restartOnTrack(i);     // <-- place on track
                        for(int j=0; j<aiTimeTicksToLive; j++)
                        {
                                    ppCarsInPopulation[i]->move();            // <-- the testing
                        }

                        if(iWorstPoints > ppCarsInPopulation[i]->gatheredPoints)
                        {
                                    iWorstPoints = ppCarsInPopulation[i]->gatheredPoints;
                        }
                        if(iBestPoints < ppCarsInPopulation[i]->gatheredPoints)
                        {
                                    iBestPoints = ppCarsInPopulation[i]->gatheredPoints;
                        }
                        sumOfPoints += ppCarsInPopulation[i]->gatheredPoints;
//                      printf("%i\n", ppCarsInPopulation[i]->gatheredPoints);
                        fprintf(pDumpFile1, "%i ", ppCarsInPopulation[i]->gatheredPoints);
            }
            iAveragePoints = sumOfPoints/iPopulationSize;

            printf("(p: %i, %i, %i)\n", iBestPoints, iAveragePoints, iWorstPoints);
            fprintf(pDumpFile1, "\n");
            fprintf(pDumpFile2, "%i %i %i %i\n", iGenerationsNumber, iBestPoints, iAveragePoints, iWorstPoints);
}


int findIndexOfValue(long alValue, long* aplArray, int aiSize)
```

```
{
	int i;
	for(i=0; i<aiSize; i++)
	{
		if(aplArray[i] >= alValue) return i;
	}
	printf("[FAULT in findIndexOfValue]");
	return 0; //should never happen...
}

int inline max(int a, int b)  //only used to give points to the roulette
{
	if(a>b) return a;
	return b;
}

void EvolutionLearner::CreateNextGeneration(int aiNewPopulationSize, int aiSurvivors, int aiMutants, float afMutationProbability)
{
	extern Car *theCars[maxNumOfCars];
	extern Track *theTrack;

	iGenerationsNumber++;
	printf("CreateNextGeneration %i:", iGenerationsNumber);

	Car** newPopulation  = new Car*[aiNewPopulationSize];
	long* rouletteArray  = new long[iPopulationSize];
	bool* oldCarSurvives = new bool[iPopulationSize];
	long sum=0, r;
	int i, foundIndex;
	NeuralDriver* motherDriver;
	NeuralDriver* fatherDriver;

	if(aiSurvivors > aiNewPopulationSize) aiSurvivors = aiNewPopulationSize;
	if((aiSurvivors + aiMutants) > aiNewPopulationSize) aiMutants = aiNewPopulationSize - aiSurvivors;

	for(i=0; i<iPopulationSize; i++) //fill accumulated points array for roulette
	{
//		sum += ppCarsInPopulation[i]->gatheredPoints;
//		sum += ppCarsInPopulation[i]->gatheredPoints * ppCarsInPopulation[i]->gatheredPoints / iBestPoints;
//		sum += max(ppCarsInPopulation[i]->gatheredPoints - iWorstPoints,   0);
		sum += max(ppCarsInPopulation[i]->gatheredPoints - iAveragePoints, 0);

		rouletteArray[i]  = sum;
		oldCarSurvives[i] = false;
//		printf(" -acc[%i]:  %i\n", i, sum);
	}
//	printf("(sum:%i,RMAX:%i)", sum, RAND_MAX);
	for(i=0; i<(aiSurvivors + aiMutants); i++) //select survivors (roulette)
	{
		if(i%20 == 0)
		{
			if(i<aiSurvivors) printf(".");
			else              printf(",");
		}

		r = (long)((double)rand() * sum) / RAND_MAX;
		foundIndex = findIndexOfValue(r, rouletteArray, iPopulationSize);

//		printf(" - - picked(%i):  %i\n", r, foundIndex);
		if(oldCarSurvives[foundIndex])
		{
			newPopulation[i] = new Car(ppCarsInPopulation[foundIndex]); //make copy
			newPopulation[i]->revokeLastInDevHistory();
		}
		else
		{
			newPopulation[i] = ppCarsInPopulation[foundIndex]; //re-use this instance
			oldCarSurvives[foundIndex] = true;
		}
		if(i<aiSurvivors) newPopulation[i]->addToDevHistory('s');
	}
	for(i=aiSurvivors; i<(aiSurvivors + aiMutants); i++) // mutate the new mutants
	{
		((NeuralDriver*)newPopulation[i]->getDriver())->getNet()->mutate(afMutationProbability);
		newPopulation[i]->addToDevHistory('m');
	}
	for(i=(aiSurvivors + aiMutants); i<aiNewPopulationSize; i++) // fill new population with children
	{
		if(i%20 == 0) printf(":");

		newPopulation[i] = new Car(i);

		r = (long)((double)rand() * sum) / RAND_MAX;
		foundIndex = findIndexOfValue(r, rouletteArray, iPopulationSize);
		newPopulation[i]->inheritDevHistory(ppCarsInPopulation[foundIndex]);
		if(oldCarSurvives[foundIndex]) newPopulation[i]->revokeLastInDevHistory();
		motherDriver = (NeuralDriver*)ppCarsInPopulation[foundIndex]->getDriver();

		r = (long)((double)rand() * sum) / RAND_MAX;
		foundIndex = findIndexOfValue(r, rouletteArray, iPopulationSize);
		newPopulation[i]->inheritDevHistory(ppCarsInPopulation[foundIndex]);
		if(oldCarSurvives[foundIndex]) newPopulation[i]->revokeLastInDevHistory();
		fatherDriver = (NeuralDriver*)ppCarsInPopulation[foundIndex]->getDriver();

		newPopulation[i]->setDriver(new NeuralDriver(Net::CreateNetFromParents(motherDriver->getNet(), fatherDriver-
>getNet())));
		newPopulation[i]->addToDevHistory('c');
	}
	for(i=0; i<iPopulationSize; i++) // clean up those that didn't survive
	{
		if(!oldCarSurvives[i])
		{
			//delete ppCarsInPopulation[i];
		}
	}

	iPopulationSize = aiNewPopulationSize;
	ppCarsInPopulation = newPopulation;
	printf("\n");
}


void EvolutionLearner::loopInEvolution(int aiGenerationsToGoThroug, int aiEvaluationTime, int aiNewPopulationSize, int aiSurvivors, int
aiMutants, float afMutationProbability)
```

```
{
	for(int i=0; i<aiGenerationsToGoThroug; i++)
	{
		EvaluateGeneration(aiEvaluationTime);
		CreateNextGeneration(aiNewPopulationSize, aiSurvivors, aiMutants, afMutationProbability);
	}
}


void EvolutionLearner::UseBestCarsInGame(int aiNumberOfCarsToUse)
{
	extern Car *theCars[maxNumOfCars];
	extern Track *theTrack;

	printf("UseBestCarsInGame...\n");

	Car** bestCars = new Car*[aiNumberOfCarsToUse + 1]; //plus on to ease the push down the list...
	bool allreadyInBestCars = false;
	int i, j, tempScore;

	for(i=0; i<aiNumberOfCarsToUse; i++)
	{
		bestCars[i] = NULL;
	}
	//Place best Cars i top-ranking array
	for(j=0; j<iPopulationSize; j++) //each Car
	{
		tempScore = ppCarsInPopulation[j]->gatheredPoints;
		if(bestCars[aiNumberOfCarsToUse-1] == NULL || tempScore > bestCars[aiNumberOfCarsToUse-1]->gatheredPoints) //if better
than worst in top
		{
			allreadyInBestCars = false;
			for(i=0; i<aiNumberOfCarsToUse; i++) //check not allready selected
			{
				if(bestCars[i] != NULL)
				{
					if(((NeuralDriver*)ppCarsInPopulation[j]->getDriver())->getNet()->equals(
((NeuralDriver*)bestCars[i]->getDriver())->getNet() ))
					{
						allreadyInBestCars = true;
						break;
					}
				}
			}
			if(!allreadyInBestCars)
			{
				for(i=aiNumberOfCarsToUse-1; i>=0; i--) //each top-ranking place - backwards
				{
					if(bestCars[i] == NULL || tempScore > bestCars[i]->gatheredPoints)
					{
						bestCars[i+1] = bestCars[i];
						bestCars[i]   = ppCarsInPopulation[j];
					}
					else
					{
						break; //no worse Cars obove in rank.
					}
				}
			}
		}
	}

	//Place cars in visual game
	for(i=0; i<aiNumberOfCarsToUse && i<maxNumOfCars; i++)
	{
		theCars[i] = bestCars[i];
		if(bestCars[i] != NULL)
		{
			printf("\nChosen score: %i   ", bestCars[i]->gatheredPoints);
			theCars[i]->restartOnTrack(i);
			theCars[i]->printDevHistory();
		}
		else
		{
			printf("\nChosen score: NULL   ");
		}
	}
	printf("\n");
}
```

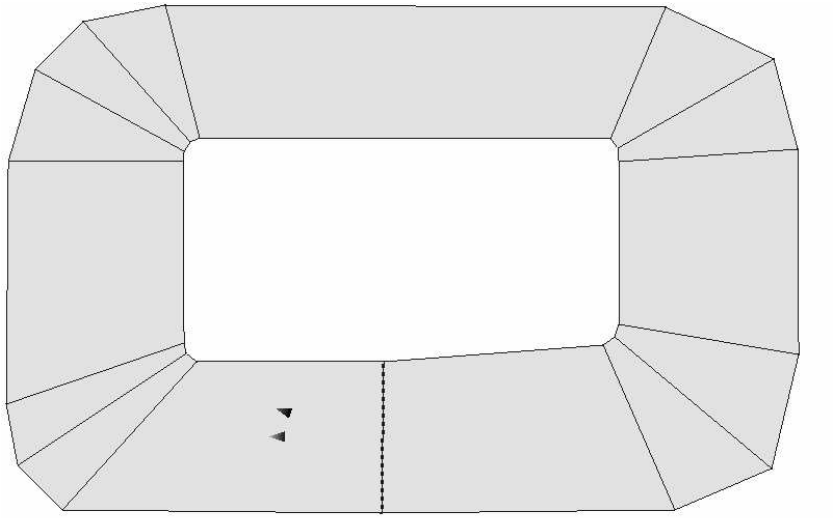
Net_EvoExtra.cpp

```
#include "Net.h"
#include <stdlib.h>
#include <stdio.h>



int Net::GetInputLenght()  { return pInputLayer->iHowManyNeurons;  }
int Net::GetOutputLenght() { return pOutputLayer->iHowManyNeurons; }



Net* Net::CreateNetFromParents(Net* apoMotherNet, Net* apoFatherNet)
{
	int howManyLayers = apoMotherNet->iHowManyLayers;
	int* layerStructure = new int[howManyLayers];
	float motherWeight, fatherWeight, newWeight, r;
	for(int i=0; i<howManyLayers; i++)
	{
		layerStructure[i] = apoMotherNet->pTheLayers[i].iHowManyNeurons;
	}

	Net* newNet = new Net(howManyLayers, layerStructure);

	for(int layerIndex=0; layerIndex < howManyLayers; layerIndex++)
	{
```

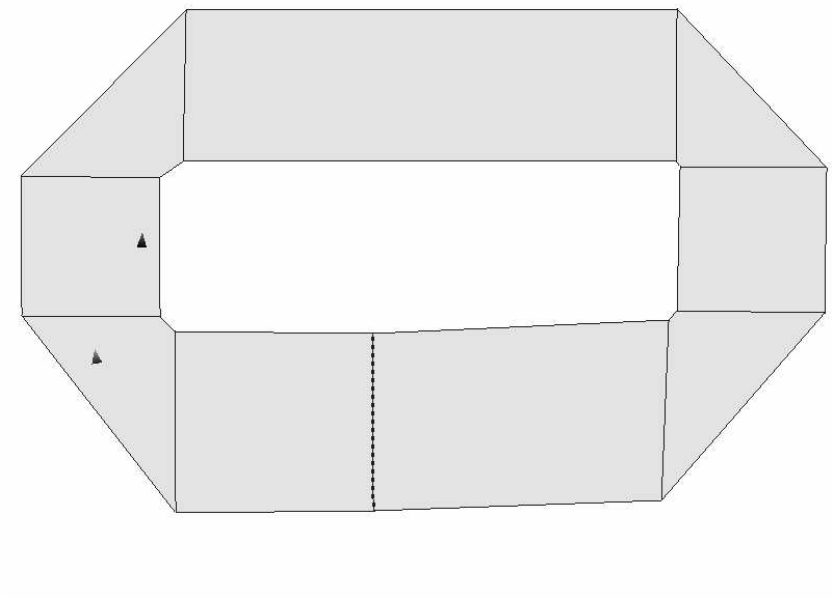
```
                        for(int neuronIndex=0; neuronIndex < newNet->pTheLayers[layerIndex].iHowManyNeurons; neuronIndex++)
                        {
                                    for(int weightIndex=0; weightIndex < newNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].iHowManyPrevNeurons; weightIndex++)
                                    {
/*
                                                if(rand() % 2 == 0) //inherit weight from mother
                                                {
                                                            newNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex] = apoMotherNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex];
                                                }
                                                else //inherit weight from father
                                                {
                                                            newNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex] = apoFatherNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex];
                                                }
*/
                                                motherWeight = apoMotherNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex];
                                                fatherWeight = apoFatherNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex];

                                                r = (float)rand() / (float)RAND_MAX;
                                                newWeight = motherWeight * r + fatherWeight * (1.0f-r);

                                                newNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex] = newWeight;
                                    }
                        }
            }

            return newNet;
}


void Net::mutate(float mutationProbability)
{
            float* pfTempWeightPointer;

            for(int layerIndex=0; layerIndex < this->iHowManyLayers; layerIndex++)
            {
                        for(int neuronIndex=0; neuronIndex < this->pTheLayers[layerIndex].iHowManyNeurons; neuronIndex++)
                        {
                                    for(int weightIndex=0; weightIndex < this-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].iHowManyPrevNeurons; weightIndex++)
                                    {
                                                if(((float)rand() / (float)RAND_MAX) < mutationProbability) // then mutate this weight.
                                                {
                                                            pfTempWeightPointer = &this-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex];
//                                                          *pfTempWeightPointer = (float)(rand() - (RAND_MAX / 2)) / (float)RAND_MAX; //
between -0.5 and +0.5
                                                            *pfTempWeightPointer = (((float)rand()/(float)RAND_MAX) - 0.5f) *  8.0f;   //
between -4.0 and +4.0
//                                                          *pfTempWeightPointer = (((float)rand()/(float)RAND_MAX) - 0.5f) * 32.0f;   //
between -16 and +16
//                                                          *pfTempWeightPointer = *pfTempWeightPointer * 10.0f;                       //
boosting existing weight
                                                }
                                    }
                        }
            }
}



Net* Net::clone()
{
            return CreateNetFromParents(this, this); //lazy solution...
}


bool Net::equals(Net* compareNet)
{
            if(this->iHowManyLayers != compareNet->iHowManyLayers) {
                        return false;
            }
            for(int layerIndex=0; layerIndex < this->iHowManyLayers; layerIndex++)
            {
                        if(this->pTheLayers[layerIndex].iHowManyNeurons != compareNet->pTheLayers[layerIndex].iHowManyNeurons)
                        {
                                    return false;
                        }
                        for(int neuronIndex=0; neuronIndex < this->pTheLayers[layerIndex].iHowManyNeurons; neuronIndex++)
                        {
                                    if(this->pTheLayers[layerIndex].pTheNeurons[neuronIndex].iHowManyPrevNeurons !=
pTheLayers[layerIndex].pTheNeurons[neuronIndex].iHowManyPrevNeurons)
                                    {
                                                return false;
                                    }
                                    for(int weightIndex=0; weightIndex < this-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].iHowManyPrevNeurons; weightIndex++)
                                    {
                                                if(this-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex] != compareNet-
>pTheLayers[layerIndex].pTheNeurons[neuronIndex].pfWeightToPrevLayerOfNeurons[weightIndex])
                                                {
                                                            return false;
                                                }
                                    }
                        }
            }
            return true;
}
```

Bilag 6, Banerne

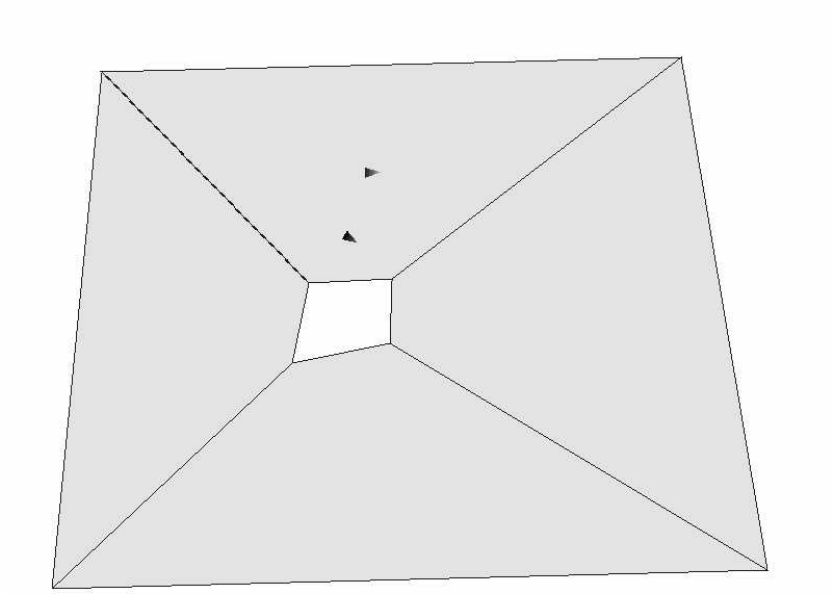
Indianapolis 500



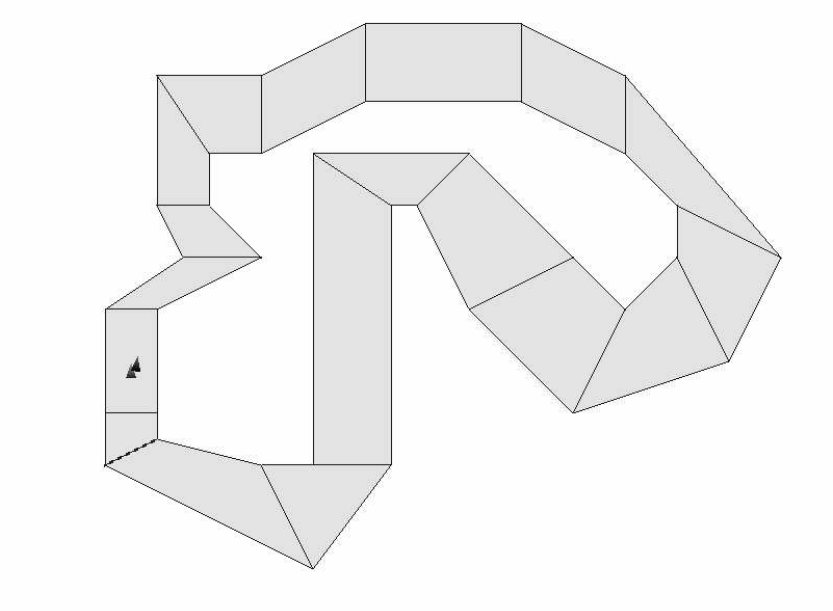
Indianapolis Ogly



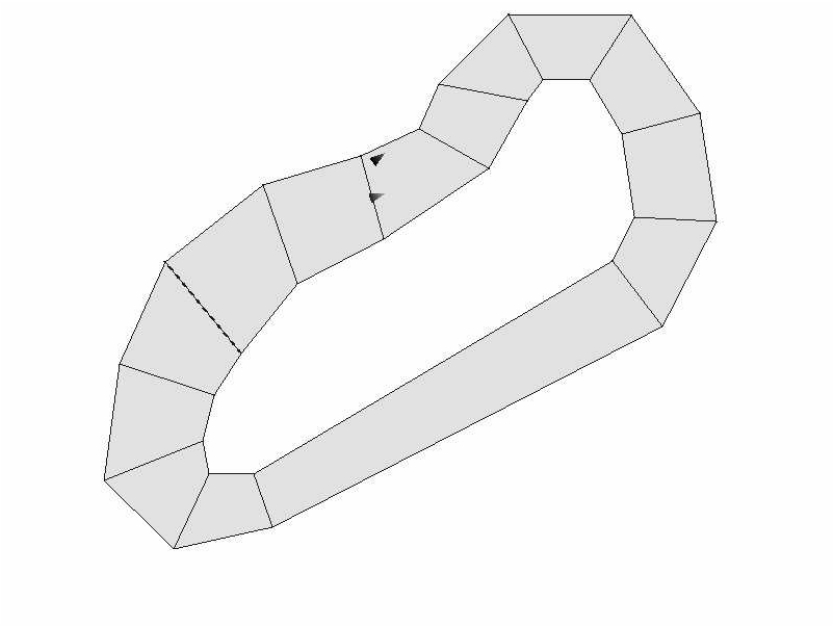
Ogly Square



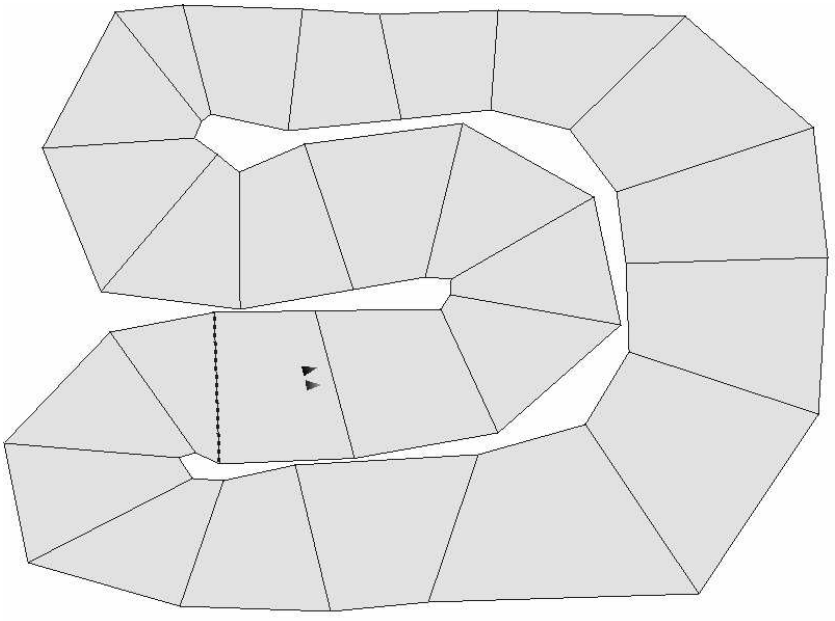
Original Frosch Track



Simple Oval

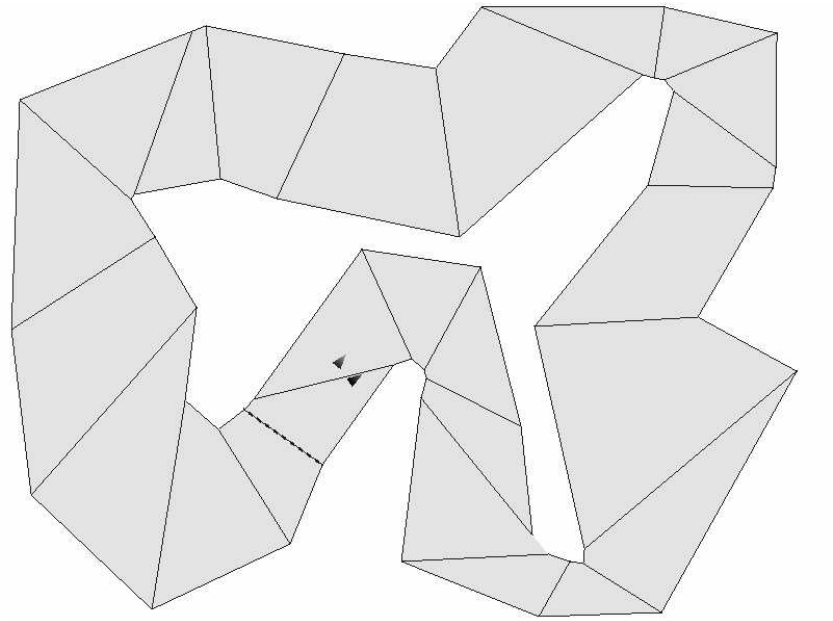


Teach Me To Drive

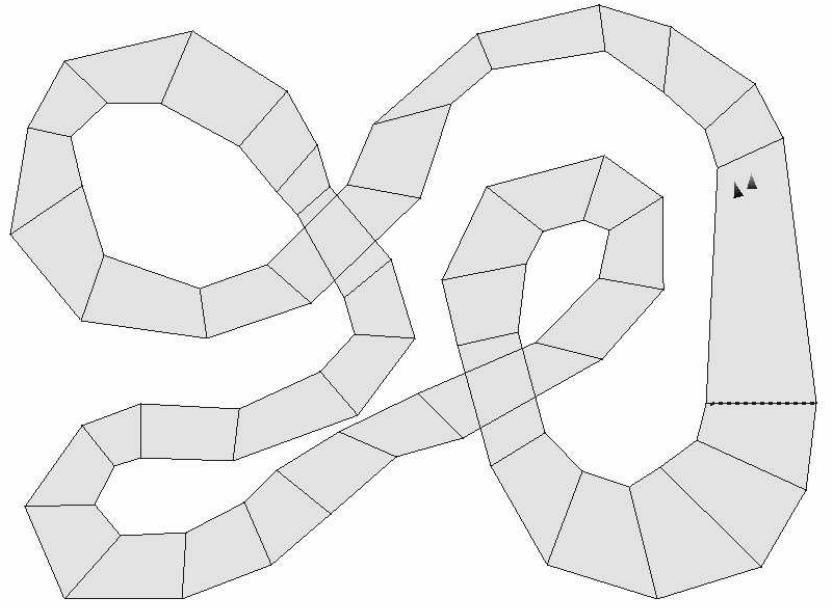


Teach Me To Drive 2

Teach Me To Drive 3



The Long and Snurkly



Trick The Simple Driver