# Verslag labo 4: Chess Agent

Deze laatste opdracht omvat het maken van een Chess Agent. In den beginne, werd deze uitgewerkt met zoekalgoritmes in combinatie met een evaluatiefunctie. Zoekalgoritmes evalueren hun “game state” enkele stappen in de toekomst om zo een beslissing te maken in het heden. Het uitwerken hiervan heeft zijn vruchten afgeworpen, maar met extra tijd werd er besloten om ook aan andere methodes te werken. Machine learning met een neural network was één aanpak, en werken met een convolutional neural network was een andere invalshoek.

1. **Minimax Algorithm**

Minimax is verreweg het meest gekende zoekalgoritme in schaak. Voor een gegeven diepte zal het algoritme elke mogelijke game state (of node) evalueren met een getal. De static evaluation function speelt hierbij een belangrijke rol. Zijn opbouw in moeilijkheidsgraad is zeer breed. Bij de start van dit project is hij simpel. Naarmate de tijd verstrijkt, kan de implementatie zowel slimmer als complexer gemaakt worden. De evaluation function moet zodanig uitgevoerd zijn zodat:

* Negatieve evaluatie: zwart in voordeel.
* Positieve evaluatie: wit in voordeel.
* Nul als evaluatie: een gelijkspel

Wit neemt altijd het maxima van zijn onderste nodes. Zwart het minima. Het resultaat van dit algoritme is een positief of negatief getal. Dit duidt op het beste resultaat van het bord na het evalueren van alle states.

De code is voorzien van commentaar. Deze beschrijft, lijn per lijn, de exacte werking van ons programma. Dit verslag is geen replica hiervan, maar kaart eerder de problemen aan bij het komen tot een resultaat.

Het grootste probleem bij Minimax was het returnen van twee waarden: de evaluatie en de beste move van elke node. Bij implementaties op het internet wordt het vinden van de beste zet uitgevoerd in een aparte functie. Het bleek interessanter dit mee te verwerken in Minimax. Daardoor is er maar één lijn code nodig in “train\_stockfish.py”. De witte speler hoeft maar één regel uit toe voeren bij het achterhalen van zijn move.

Na een aantal dingen uit te proberen, werd er een correct resultaat verkregen. In een tuple - (“boardValue”, “foundMove”) – werden deze waarden bijgehouden. In de hoogste node wordt de grootste evaluatie gezocht en daarbij ook de beste zet.

1. **Evaluation Function**

De eerste verwezenlijking van deze functie is het meest gekend. Elk schaakstuk krijgt een vaste waarde. Deze zijn groter naarmate de stukken belangrijker worden. Zo heeft de koningin bijvoorbeeld een veel grotere waarde dan een loper. Bij de witte stukken zijn deze getallen positief, bij de zwarte negatief. De functie evalueert het bord door de waarden van alle stukken op te tellen. Hoe groter de evaluatie, hoe beter de witte speler er voor staat.

Deze implementatie heeft echter een paar problemen. Vaak krijgen nodes dezelfde evaluatie omdat er evenveel stukken op bord blijven. De functie hangt sterk af van het slaan van stukken. De speler zijn grootste prioriteit is het nemen van een stuk, wat niet altijd voordelig blijkt te zijn op lange termijn.

In het begin van het spel krijgt elke node de waarde 0 toegewezen. De reden hiervoor is simpel. Er zijn evenveel zwarte als witte stukken. Dit betekent dat de AI telkens dezelfde move zal kiezen (de laatste).

Wat is het besluit? De intelligentie van Minimax is zeer afhankelijk van de evaluation function. Hoe meer situaties we kunnen differentiëren, hoe beter Minimax is in het zoeken naar de beste move.

Een voordeel is dat deze simpele functie geïmplementeerd kon worden zonder for-loop door enkel de python-chess library te gebruiken. Zo werd de run time per node bijna gehalveerd ten opzichte van alle posities op het schaakbord te doorlopen. Dit werd mogelijk door de posities van bepaalde stukken op te vragen (een matrix met 1 waar de stukken van dit type en deze kleur staan en 0 voor alle andere velden). Door van deze matrix de som te nemen (ook een ingebouwde functie), werd het aantal stukken bekomen. Dit aantal werd telkens vermenigvuldigd met de waarde van het stuk. De witte stukken werden opgeteld bij de evaluatie en de zwarte stukken werden afgetrokken van de evaluatie.

Om dit te realiseren, werd er een extra check toegevoegd aan het programma. De evaluatie hangt nu niet enkel af van de schaakstukken hun waarde, maar ook van hun positie. Een loper staat best aan de zijkant van het schaakbord want daar bestrijkt deze het meeste velden zonder zichzelf nodeloos in gevaar te brengen. De evaluatie wordt groter of kleiner afhankelijk van zijn plaats op het bord.

Verder werd er nog een laatste toevoeging gedaan aan deze manuele evaluatie functie. Er werd gekeken naar het aantal velden dat ieder stuk kon bestrijken. Ook dit kon met een ingebouwde functie van python-chess. Dit gaf een matrix zoals eerder besproken: met een 1 op ieder veld dat een geldige zet was en een 0 voor alle andere velden. Ook hier werd de som van genomen en deze werd verrekend bij de evaluatie (optellen voor wit, aftellen voor zwart). Zo zouden de stukken betere posities in moeten nemen waar ze meer stukken konden bestrijken.

Een evaluation function manueel ontwerpen brengt veel moeilijkheden met zich mee. Het is zeer lastig om een goede evaluatie te maken die daarnaast ook nog eens zo performant mogelijk is. Minder lang nadenken per node resulteert in het kunnen onderzoeken van meer nodes. Er is een lange zoektocht nodig om een algoritme te vinden die het best mogelijke resultaat teruggeeft. Dit heeft deels te maken met de complexiteit van het spel en de intuïtie waar grootmeesters mee spelen. Zij kennen veel openingen vanbuiten en weten welke stelling ze krijgen door welke opening te spelen. Hetzelfde geldt bij het eindspel, hier kunnen grootmeesters het verst vooruit kijken. Een goede chess AI zou hier dus ook rekening mee moeten houden. Bij schaak is er niet altijd een beste zet mogelijk want vaak zijn er verschillende mogelijkheden die niet per se beter of slechter zijn dan elkaar maar gewoon een ander tactisch voordeel bieden (bv. Ruimtelijk voordeel, materieel voordeel, …).

1. **Alpha-beta Pruning en Quiescence Searching**

Alpha-beta pruning is een manier om de uitvoeringstijd van het Minimax algoritme aanzienlijk te verlagen. We slagen nodes over die geen effect hebben op het resultaat. Er is al een betere waarde beschikbaar voor de witte of zwarte speler. Het verwerken van verdere nodes zou zinloos zijn. Pruning geeft geen absolute zekerheid op tijdswinst. Het zal aankomen op de volgorde van moves of sommige nodes overgeslagen kunnen worden. Het implementeren hiervan vroeg slechts het toevoegen van enkele lijnen code.

Quiescence Searching was een poging om de keuze slimmer te maken. Ook hier was veel over te vinden op het internet. Het zou namelijk het Horizon Effect moeten oplossen. Strategieën vallen vaak onder de radar bij het ontwikkelen van een Chess Agent. We geven hiervan een voorbeeld. Minimax neemt met zijn koningin het zwarte paard. Hij weet echter niet dat zijn koningin in de volgende move gepakt kan worden door een zwarte pion. Quiescence Searching is een extra zoekalgoritme dat start bij de leaf nodes. Hij zal deze enkel uitvoeren bij de eerder uitlegde strategieën, die de evaluation function onmogelijk kan detecteren.

We hebben er uiteindelijk voor gekozen om dit algoritme te laten vallen. Het kiezen van een zet werd in de opening al met enkele seconden vertraagd en deze rekentijd zou exponentieel toenemen in het middenspel. Zijn complexiteit en inefficiëntie zijn de grootste redenen bij het weglaten van dit stuk code.

1. **Machine learning**

Noot: We zijn nog niet tevreden met de huidige stand van zaken. Vooral omdat het netwerk enkel gebruikt is om een test te doen met amper 100 evaluaties als training data. Natuurlijk is dat niet voldoende maar hier werd slechts voor geopteerd om de werking te verifiëren.

Het zoekalgoritme wordt nog steeds gebruikt maar in plaats van ValueFinder (een manuele evaluatie van het bord) te gebruiken, wordt er een neural network gebruikt worden die de evaluatie op zich zal nemen.

neural\_network.py Bevat het netwerk en neural\_network\_util.py bevat de utility (hulpfunctie) om een bord om te zetten in data. Eerst moet een neuraal netwerk getraind worden, dus wordt er een hoop data (zie later) met juiste (volgens stockfish) antwoorden in gestopt om aan supervised learning te doen. Zo leert het netwerk om waarden te mappen aan bepaalde bord states. Na het trainen kan hij dus een voorspelling proberen te doen.

Het genereren van data

Via de Stockfish engine kan er een evaluatie van een bord verkregen worden en een tijd om te zoeken wordt als parameter gegeven: hoe meer tijd, hoe beter de evaluatie maar hoe langer het duurt om veel data te verkrijgen en omgekeerd dus ook. Er worden per simulatie verschillende games gespeeld tussen telkens andere spelers (zoals: random with 1st level search, stockfish met verschillende zoektijden, de eigen evaluatiefunctie). Na iedere zet wordt het bord beoordeeld door stockfish en met behulp van neural\_network\_util.py omgezet naar data in een vorm die het netwerk kan lezen. Vervolgens worden zowel het bord (in data vorm) als de beoordeling opgeslagen in een array. Als het spel afgelopen is dan wordt alles weggeschreven (toegevoegd) aan twee (.txt) bestanden.

Het neural/convolutional network

Er staan momenteel twee versies op GitHub (twee branches). Eén branch werkt: het bord gaat als array vorm in het netwerk en geeft 1 waarde als output weer. Dit is de master branch.  
Er is ook een tweede branch, hier is de code iets anders en werd er geprobeerd om een convolutional neural network te maken. Dit soort netwerk is zeer goed in het herkennen van eigenschappen in foto’s. In zekere zin kan het schaakbord ook als afbeelding beschouwd worden maar of dit betere resultaten zou geven dan het neural network is nog onduidelijk. Dit netwerk krijt een matrix als input en zal het behandelen als een afbeelding (verschillende 'pixels' samen nemen en bekijken met behulp van een kernel). De bedoeling is hier ook een enkele output te krijgen maar deze geeft momenteel een array als output.

De werking van [neural\_network.py](https://l.facebook.com/l.php?u=http%3A%2F%2Fneural_network.py%2F%3Ffbclid%3DIwAR02WzKaY8OJPcN9lnmrB2_QXilBTmdlJNaK1vYF5EPB6u5pX5t-loBPo1g&h=AT3YEN1oP6q3Vp4LLsOlitMiQ4VFDlnpKR1tX_smG1JO1rUGlLUKTfzhSeyhjpnMRuuqNrWMt2ZPR24CKXgHegd_De9QtUhoPNcG5WX6t29utLqCRm-sX8VIcOGC-h39_pSiw7_Zm5ao1ZD78GbrNw)

prepare\_network() zal alle nodige stappen doen om het netwerk te trainen en zet alles klaar om een evaluatie te doen voor de n aantal games die gespeeld zullen worden. Als evaluatie wordt evaluate() nog steeds opgeroepen zoals eerder maar valuefinder.py werd gerefactored. In plaats van enkel evaluate() is er een opsplitsing gemaakt tussen MLevaluate() en manualEvaluate(). Evaluate() returnt nu gewoon één van de twee evaluatie functies. Zo is het makkelijk om van evaluatiefunctie te veranderen zonder code aan te moeten passen op verschillende locaties.

neural\_network.py

In prepare\_network() zal het netwerk getraind worden. De data wordt lijn per lijn verwerkt (data/boardData.txt en data/valueData.txt). Vervolgens wordt alle data van string naar array en int (values) of float (board) omgezet. De endlines '\n' worden verwijderd na iedere lijn.   
train\_test\_split zal de data opsplitsen in train data en test data.

keras.sequential is de vorm van het netwerk: soort van lagen en aantal nodes per laag etc. Er wordt geëindigd met 1 node (linear) omdat er maar één getal als output gewenst is en dit getal mag eender welk getal zijn (linear geeft deze mogelijkheid).

Compile bepaalt welke fout er geminimaliseerd moet worden: hier is dat de afstand ten opzichte van de gegeven waarde van de data. Zo zal het netwerk proberen om zijn gok zo dicht mogelijk bij de evaluatie van stockfish te brengen.

model.fit gaat het trainen zelf allemaal regelen met de gegeven data.

model.evaluate zal testen hoe correct zijn voorspellingen zijn aan de hand van de testdata.

Predict(Board) zal een board omzetten in data, deze data door het netwerk sturen, de output omzetten naar een echte waarde en deze returnen. Dit wordt opgeroepen in MLevaluate in valueFinder.py. Deze evaluatie is performanter dan de manuele evaluatiefunctie.