UNIVERSITEIT ANTWERPEN

Academiejaar 2020-2021

Faculteit Toegepaste Ingenieurswetenschappen

Chess AI-Agent

5-Artificiële Intelligentie

Mathias Maes, Tijs Van Alphen en Willem Van der Elst

Bachelor of Science in de industriële wetenschappen: Elektronica-ICT

Universiteit Antwerpen

Inhoudsopgave

1. Keuze

Het type AI was een belangrijke keuze van dit project. Het moest haalbaar zijn om te implementeren binnen de beperkte tijdsperiode en we moesten onze eigen niet-supercomputers gebruiken om te treinen. Een lijst met de voor- en nadelen van de verschillende opties werd opgesteld.

Search:
Er zijn te veel mogelijke states ($\approx 10^{120}$) om de hele tree uit te werken.

Multi-Agent search (Minimax, Alpha-Beta pruning):
Deze methode wordt gebruikt door meerdere bronnen, maar we hebben er nog geen ervaring mee.

Reinforcement learning (Generalised Q learning):
We hebben dit al gebruikt en begrijpen de methode. Een snelle implementatie is dus mogelijk.
Eenmaal getraind, zal het optimaal functioneren. De training heeft echter veel tijd nodig en de keuze van de features is niet zo voor de hand liggend.

Neural network/ Deep learning: Onze ervaring met Deep learning is nogal beperkt. We weten niet goed wat we als in- en output moeten nemen.

Q-Learning leek ons het interessantst, maar Minimax is haalbaarder en zou sowieso een goed resultaat leveren. Uiteindelijk besloten we om beide methodes uit te werken. We hadden namelijk een thesis[**?**] gevonden die een minimax agent gebruikte als feature voor een generalized Q-Learner.

2. Alpha-Beta Pruning

2.1. Utility

Onze utility is grotendeels gebaseerd op de rewards van onze Q-Learner die later in dit verslag behandelt worden. Op dit moment volstaat het om een simpele versie van onze utility te aanschouwen:

$$u(S) = \begin{cases} 100 & \text{Schaakmat gewonnen} \\ -30 & \text{Gelijkspel of Stalemate} \\ -100 & \text{Schaakmat verloren} \\ c + C + \Delta M + \Delta m + s + f_q & \text{Bij overige situaties} \end{cases}$$

c = Of er gecastled is met de vorige move
C = Of het bord in schaak staat
ΔM = Het materiaal verschil op het bord
Δm = Het verschil in mobility
s = De positie score voor alle eigen stukken
f_q = Een gewogen(geleerd of niet-geleerd) aantal features

Deze kleine sub-functies worden later nog besproken in *3.2. Features*.

3. Q-Learning Agent

3.1. Generalization

De eerste agent die we maakten, was de Q-agent. Deze is generalized omdat, zoals besproken in *1. Keuze*, het praktisch onmogelijk is om elke staat te bezoeken. Daarom dat we niet 'gewoon' Q-learning kunnen gebruiken, maar wel generalized Q-learning.

3.2. Features

We hebben een hoop features gedefinieerd. Deze zijn voornamelijk gebaseerd op features uit de paper[?]. De bedoeling van deze features is om een state zo goed mogelijk te definiëren. Je kan hieronder een beschrijving vinden van elks van onze 55 features.

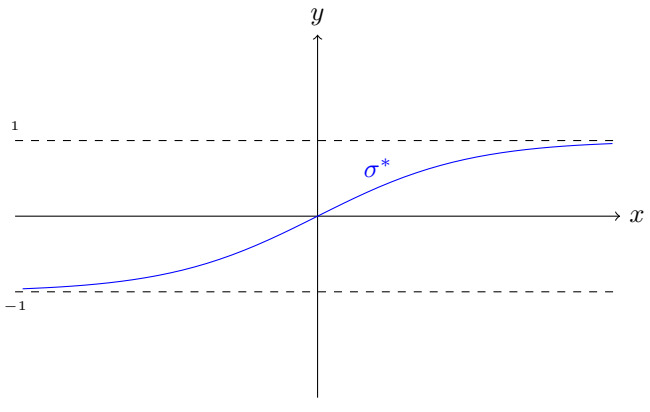
We berekenen onze features met deze formule:

$$Q(s,a) = \sum_i w_i \cdot \sigma\left(f_i(s,a)\right)$$

De $\sigma(x)$ die we in de berekening zien wordt als volgt gedefinieerd:

$$\sigma^*(x) = \frac{2}{1+e^{-x}} - 1$$

Dit is een aangepaste versie van de bekende sigmoid[?] functie. Onze versie zorgt in tegenstelling tot de originele er voor dat waardes in het interval $]-1,1[$ vallen.
Hier zie je de grafiek van de functie:



Figuur 3.1: $\sigma^*(x)$

3.2.1. Amount of Pieces

De meest voor de hand liggende feature die we gebruiken, is het aantal stukken per type op een bord. We berekenen dit aantal voor zowel de onze als die van de tegenspeler. Elk type krijgt dus 2 features.

Op deze manier zal onze Q-agent leren dat het voordelig is om het aantal stukken van zichzelf hoog te houden en die van de opponent laag te houden.

Het laatste wat we ook nakijken is de feature `AmountBalancePieces()` deze zal de material value van alle stukken van de speler berekenen en de material value van de tegenspeler er af trekken. Wanneer dit getal negatief is, betekent dat dus dat de opponent er op dit vlak beter voor staat.

Material value

Elk soort schaakstuk krijgt een cijfer dat zijn waarde bepaalt. Zo krijgt de koningin een hogere waarde dan een pion, omdat de koningin belangrijker is. In *Code 3.1* staat de implementatie van hoe de schaakstukken zich verhouden t.o.v. elkaar.

```
def getMaterialValue(piece_type):
    if piece_type is chess.PAWN:
        return 1
    elif piece_type is chess.KNIGHT or piece_type is chess.BISHOP:
        return 3
    elif piece_type is chess.ROOK:
        return 5
    elif piece_type is chess.QUEEN:
        return 9
    elif piece_type is chess.KING:
        return 10

    return 0
```

Code 3.1: Material Function

3.2.2. Checkmate

De feature `Checkmate()` gaat kijken of we bij de volgende actie de opponent schaakmat kunnen zetten.

3.2.3. Next Checkmate

In deze feature gaan we kijken of er een volgende zet van de tegenspeler bestaat die ons schaakmat zet.

3.2.4. Mobility

De `Mobility.py` file berekent voor elke type schaakstuk hoeveel vakjes het stuk kan bereiken.

Dit wordt gebruikt voor de mobility features. Hierbij gaan we voor elke type schaakstuk kijken op hoeveel vakjes die kan terechtkomen. Dit zullen we ook weer doen voor zowel de speler als de tegenspeler.

De feature `MobilityRookS()` zal voor elke Rook (toren) van de eigen speler (S in de functie staat voor self) berekenen op hoeveel vakjes die kan komen en bij elkaar optellen.

Voor het berekenen van de legal moves gebruikte we de functie `islegal(Move)` op het huidige bord. Hier hebben we wat problemen mee ondervonden voor de opponent omdat we enkel stukken kunnen verplaatsten van diegene die aan beurt is. Alle moves van de opponent zijn dus illegal. Onze oplossing hiervoor is om een copy van het bord maken gevolgd door een board turn, op deze manier speel je dan zogezegd even als de opponent op het gekopieerde bord.

3.2.5. Attackers

Deze attacked features laten weten hoeveel van de schaakstukken van een bepaald type aangevallen kunnen worden door de opponent met inferieure schaakstukken.

Zo zal bijvoorbeeld de `AttackedRooksS()` feature calculeren hoeveel van zijn torens kunnen worden aangevallen door een pion, een paard of een loper van het andere team.

Voor de opponent versie van deze features te berekenen hebben we het probleem op dezelfde manier opgelost als bij mobility.

3.2.6. Forks

De volgende twee features zijn zowel voor de speler als de tegenspeler geïmplementeerd.

Pawn Fork

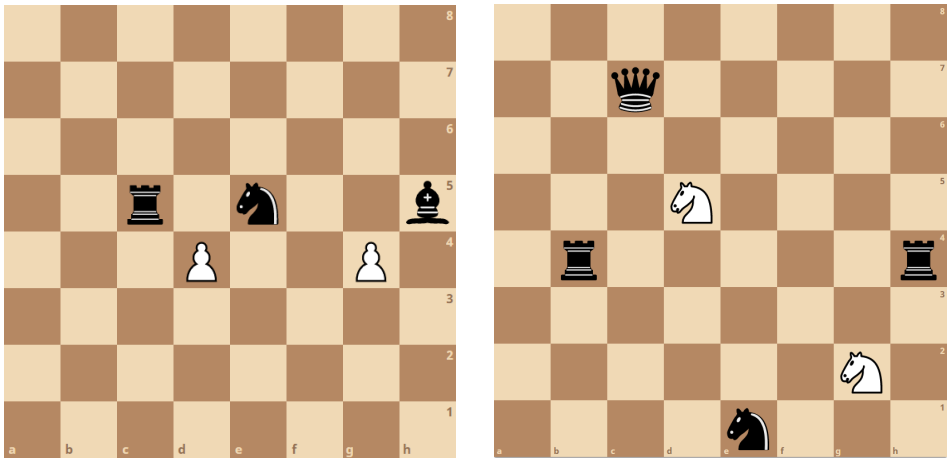
Pawn fork gaat kijken hoeveel pionnen 2 superieure stukken van de tegenstander aanvalt.

In *Figuur 3.2a* zal pion op d4 meetellen bij de pawn fork som, omdat deze 2 schaakstukken met een hogere material value kan nemen. Terwijl pion op g4 niet zal worden meegeteld.

Knight Fork

Gelijkend aan de Pawn fork zal de Knight fork ook kijken hoeveel paarden 2 superieure schaakstukken van de tegenspeler kan nemen.

In *Figuur 3.2b* zal het paard op d5 worden meegeteld, maar het paard op g2 niet. Het paard op g2 wordt niet meegeteld omdat het paard dat deze kan nemen niet superieur is t.o.v. zichzelf.

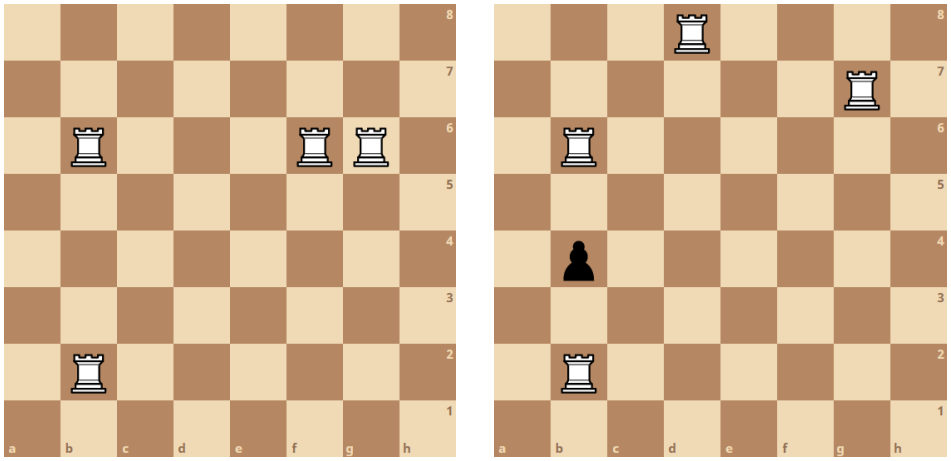


(a) Example of pawn fork (b) Example of knight fork

Figuur 3.2: Examples of forks

3.2.7. Connected Rooks

Er zijn 2 verschillende features, de `ConnectedRooksVertical()` en `ConnectedRooksHorizontal()` die op zo goed als hetzelfde werken. We gaan optellen hoeveel paren van torens er zijn die op ofwel dezelfde rang (horizontally connected) ofwel op dezelfde kolom staan (vertically connected) en waar geen andere stukken tussen staan.



(a) Example of connected rooks (b) Example where rooks are not connected

Figuur 3.3: Connected Rookd

In *Figuur 3.3a* zien we op kolom b een vertically connected rooks en op rang 6 een horizontally connected rooks. Op *Figuur 3.3b* zijn er geen connected rooks omdat ze ofwel niet tegenover elkaar staan of er staat een stuk in de weg.

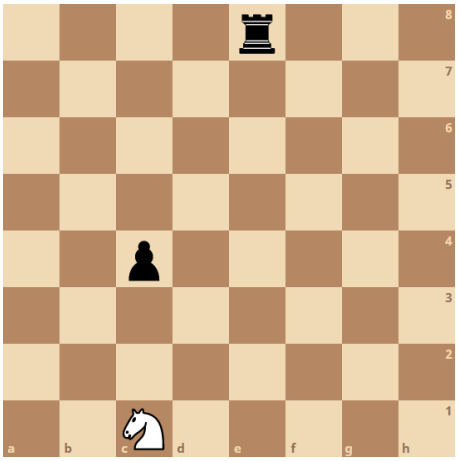
3.2.8. Control

Center Control

Center control is een feature die ons verteld hoeveel pawns je hebt staan op de vakjes D4, E4, D5, E5. Deze 4 vakjes zijn het belangrijkste van het bord, omdat het grootste trafiek hierover gaat. Deze feature wordt ook weer voor zowel de eigen speler als de tegenspeler gecreëerd.

Board Control

Het aantal lege vakjes dat een speler met zijn stukken aanvalt, bepaald zijn controle over het bord. Deze feature bepaald dit aantal voor beide teams en trekt ze dan van elkaar af. Als meerdere stukken hetzelfde lege vakje kunnen aanvallen, wordt er rekening gehouden met de waarde van de stukken. Het kleinste stuk heeft dan het voordeel. Zo kan een toren kan dus nooit controle hebben over een vakje dat een pion van het andere ook kan aanvallen.



Figuur 3.4: Example of board control

In het voorbeeld op *Figuur 3.4* zien we dat de zwarte pion controle heeft over C3 en C4. Het paard kan deze vakjes ook aanvallen, maar zou dan gepakt worden door een inferieur schaakstuk. Anderzijds heeft het paard wel controle over A2 en E2.

3.2.9. Connectivity

De connectivity beschrijft de mate waarin een speler zijn stukken kan beschermen. Met andere woorden: hoeveel ze geconnecteerd zijn met elkaar. Het kijkt dus niet enkel naar *3.2.7. Connected Rooks*, maar naar alle stukken die stukken van zijn eigen team kunnen 'aanvallen'.

3.2.10. Provokers

Deze feature lijkt sterk op *3.2.9. Connectivity*. Het verschil zit erin dat we hier gaan kijken naar welke stukken we van de tegenspeler kunnen aanvallen. Ze dagen de stukken dus uit.

3.2.11. Position score

We starten met het definiëren van een aantal matrices waarbij de matrix het bord voorstelt en elk vakje een bepaald score krijgt. Voor elk type schaakstuk hebben we een ingevulde matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 50 & 50 & 50 & 50 & 50 & 50 & 50 & 50 \\ 10 & 10 & 20 & 30 & 30 & 20 & 10 & 10 \\ 5 & 5 & 10 & 25 & 25 & 10 & 5 & 5 \\ 0 & 0 & 0 & 20 & 20 & 0 & 0 & 0 \\ 5 & -5 & -10 & 0 & 0 & -10 & -5 & 5 \\ 5 & 10 & 10 & -20 & -20 & 10 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) Pawn score matrix

$$\begin{bmatrix} -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \\ -40 & -20 & 0 & 0 & 0 & 0 & -20 & -40 \\ -30 & 0 & 10 & 15 & 15 & 10 & 0 & -30 \\ -30 & 5 & 15 & 20 & 20 & 15 & 5 & -30 \\ -30 & 0 & 15 & 20 & 20 & 15 & 0 & -30 \\ -30 & 5 & 10 & 15 & 15 & 10 & 5 & -30 \\ -40 & -20 & 0 & 5 & 5 & 0 & -20 & -40 \\ -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \end{bmatrix}$$

(b) Knight score Matrix

$$\begin{bmatrix} -20 & -10 & -10 & -10 & -10 & -10 & -10 & -20 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & 0 & 5 & 10 & 10 & 5 & 0 & -10 \\ -10 & 5 & 5 & 10 & 10 & 5 & 5 & -10 \\ -10 & 0 & 10 & 10 & 10 & 10 & 0 & -10 \\ -10 & 10 & 10 & 10 & 10 & 10 & 10 & -10 \\ -10 & 5 & 0 & 0 & 0 & 0 & 5 & -10 \\ -20 & -10 & -10 & -10 & -10 & -10 & -10 & -20 \end{bmatrix}$$

(c) Bischop score matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 10 & 10 & 10 & 10 & 10 & 10 & 5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ 0 & 0 & 0 & 5 & 5 & 0 & 0 & 0 \end{bmatrix}$$

(d) Rook score matrix

$$\begin{bmatrix} -20 & -10 & -10 & -5 & -5 & -10 & -10 & -20 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & 0 & 5 & 5 & 5 & 5 & 0 & -10 \\ -5 & 0 & 5 & 5 & 5 & 5 & 0 & -5 \\ 0 & 0 & 5 & 5 & 5 & 5 & 0 & -5 \\ -10 & 5 & 5 & 5 & 5 & 5 & 0 & -10 \\ -10 & 0 & 5 & 0 & 0 & 0 & 0 & -10 \\ -20 & -10 & -10 & -5 & -5 & -10 & -10 & -20 \end{bmatrix}$$

(e) Queen score matrix

$$\begin{bmatrix} -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -20 & -30 & -30 & -40 & -40 & -30 & -30 & -20 \\ -10 & -20 & -20 & -20 & -20 & -20 & -20 & -10 \\ 20 & 20 & 0 & 0 & 0 & 0 & 20 & 20 \\ 20 & 30 & 10 & 0 & 0 & 10 & 30 & 20 \end{bmatrix}$$

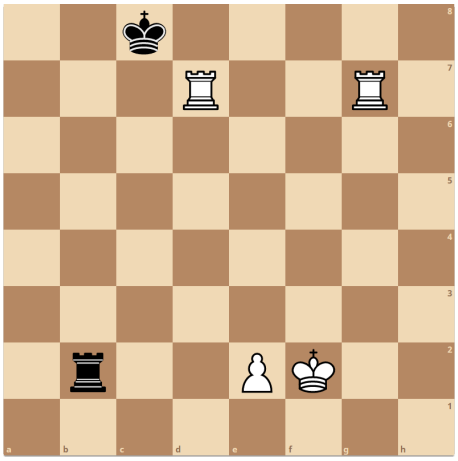
(f) King score matrix

Figuur 3.5: Score matrices

We gebruiken dit in de feature `PositionScoreBalance()` . Hierin gaan we al onze schaakstukken af en sommeren we de punten van elk stuk met zijn overeenkomende plaats op de bijhorende matrix. Dit doen we voor zowel de speler als de tegenspeler.

3.2.12. Rooks on 7th rank

Deze feature gaat kijken hoeveel torens er van de speler op de zevende rang staan. Hierbij wordt er geteld vanaf de kant van de speler zelf. De tegenspeler zal dus naar rang 2 kijken. Wanneer er torens op de 7^{de} rang staan, sta je sterk in het spel. Dit komt doordat de bewegingsruimte van de opponent zijn koning wordt beperkt en omdat de andere stukken op die rang bedreigd worden.



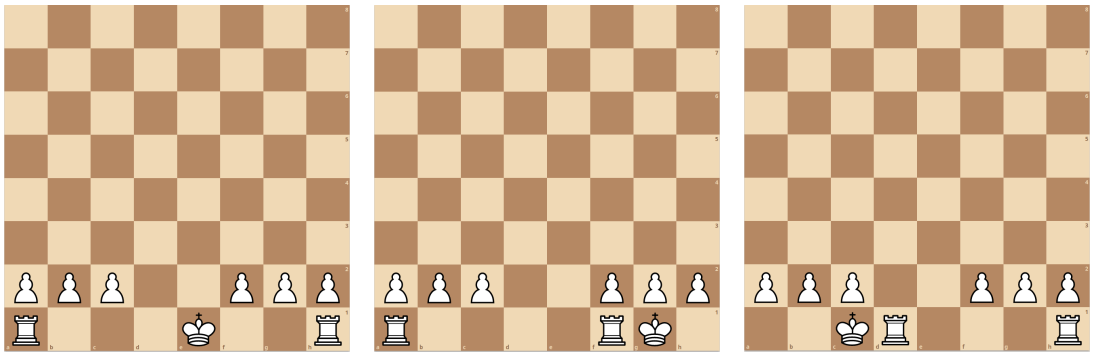
Figuur 3.6: Example of rooks on 7^{th} rank

3.2.13. King distance to center

Zoals de naam het verteld zal deze feature de afstand van de koning tot één van de middelste vakjes (E4, D4, E5 en D5) berekenen. We drukken de afstand uit in het aantal stappen dat de koning zou moeten nemen om in het midden terecht te komen.

3.2.14. Castling

Dit is een vrij eenvoudige feature die kijkt of de agent gecastled heeft. Castling is voordelig aangezien de koning dan in een veiligere positie terecht komt. Bovendien krijg je er een geactiveerde toren bij. In *Figuur 3.7b* en *Figuur 3.7c* staan 2 voorbeelden van castling.

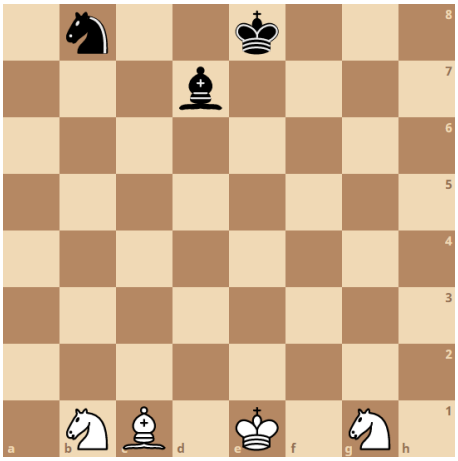


(a) Initial state (b) Short castling (c) Long castling

Figuur 3.7: Examples of castling

3.2.15. Light pieces on first rank

Hier gaan we kijken of hoeveel light pieces van een speler nog op de eerste rang staan. De eerste rang zal dus voor de speler rij 1 zijn en voor de tegenspeler rij 8. Met de light pieces verstaan we de paarden en lopers.



Figuur 3.8: Example of Light pieces on 1^{th} rank

In *Figuur 3.8* zal de witte speler een score van 3 krijgen voor zijn 2 paarden en loper. De zwarte speler heeft enkel een paard op zijn eerste rank staan en zal dus bij `LightFirstRank0()` een berekening van 1 krijgen.

3.2.16. Doubled pawns

Deze feature zal kijken op hoeveel kolommen er meer dan 1 pion staat. De kleur maakt hierbij niet uit.

3.2.17. Alpha-beta Agent prediction feature

Onze alpha-beta agent is verwerkt als feature in de Q-agent. Als deze keuze van Alpha-beta overeenkomt met de keuze van de Q-agent dan returned deze feature 1. Zo niet, dan krijgen we een return waarde van 0. Deze feature kent verschillende optimalisaties maar hierover komt meer in *4.2. Caching*.

3.3. Reward

De reward functie is als volgt gedefinieerd:

$$R(S,a,NS) = \begin{cases} 100 & \text{Schaakmat gewonnen} \\ -30 & \text{Gelijkspel of Stalemate} \\ -100 & \text{Schaakmat verloren} \\ C_O(S,a) - C_S(NS) + c(S) + \Delta M & \text{Bij overige situaties} \end{cases}$$

C_O = of de opponent in schaak staat
C_S = of de agent in schaak staat
c = of er gecastled is
ΔM = het materiaal verschil op het bord

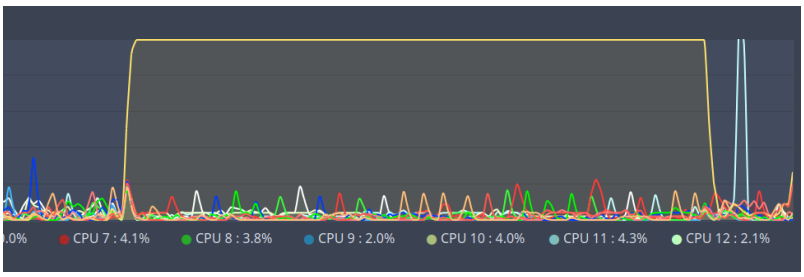
4. Optimalisaties

We hebben een hoop optimalisaties moeten toevoegen aan onze Q-Learner om hem zo rap mogelijk te laten denken. Dit is nodig voor de training sneller te laten verlopen, maar ook voor wedstrijden. De tijd is dan meestal gelimiteerd.

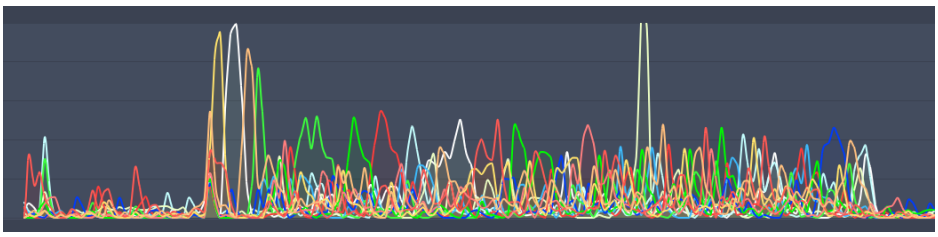
4.1. Multi-Threading

Multi-threading werd toegevoegd aan onze Q-learner. Zo berekent hij de Q-waarde van elke actie in een aparte thread. We hebben bewust niet gekozen om elke feature in een aparte thread te steken omdat dit onze timing issues verergerde in de plaats van verbeterde. We hebben door multi-threading onze iteratie tijd met 20% weten dalen.

In *Figuur 4.1* en *Figuur 4.2* is het verschil te zien tussen single-threaded (4.1) en multi-threaded (4.2) CPU usage.



Figuur 4.1: Single Threaded Q-Learner CPU usage



Figuur 4.2: Multi Threaded Q-Learner CPU usage

4.2. Caching

We gebruiken ook op veel plaatsen caching zodat we niet 2 keer hetzelfde moeten uitrekenen. De meest bijzondere cache is die voor onze alpha-beta feature. Deze moet per bord state maar 1 keer uitgevoerd worden waardoor we een hoop tijd besparen. De tijd die we hierdoor besparen per iteratie kan soms tot een minuut of 2 oplopen.

Dit was uiteraard wel een trade off met memory usage, echter werd de extra gebruikte memory overschaduwd door de tijdswinsten.

5. Resultaten

Hier vindt u al onze resultaten van het uitbundig trainen. We hebben in de loop van 5 dagen op een cloud server gespeeld tegen stockfish (4 dagen) en onze alpha-beta agent (1 dag).

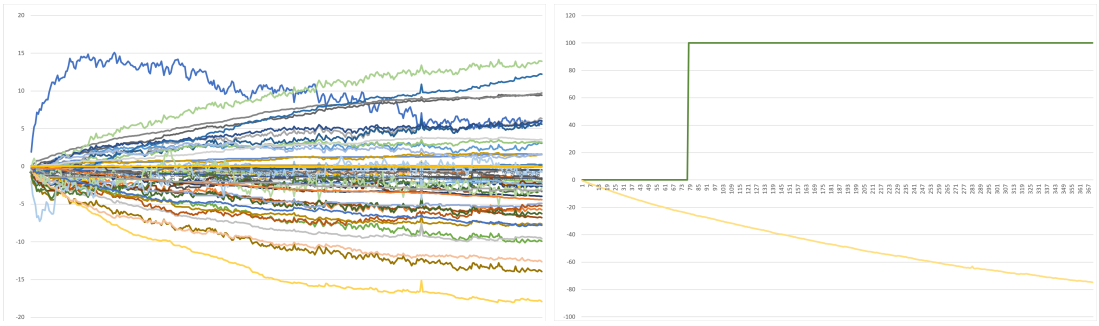
Je kan ook altijd onderaan de commando's vinden die we gebruikt hebben bij het trainen van onze agent.

5.1. GrandQ vs. Stockfish

In de eerste leer sessie hebben we onze Q-agent tegen stockfish laten trainen zoals verwacht heeft onze agent nooit kunnen winnen, maar heeft wel hieruit nuttige info kunnen leren.

Command:

```
python3 train -o stockfish --skill 0 --depth 3 -tt 15 -dt 0 -e 0.3 -d 0.3 -l 0.01 -q -i -1
```



(a) All features except checkmates

(b) Check mate & Next Checkmate

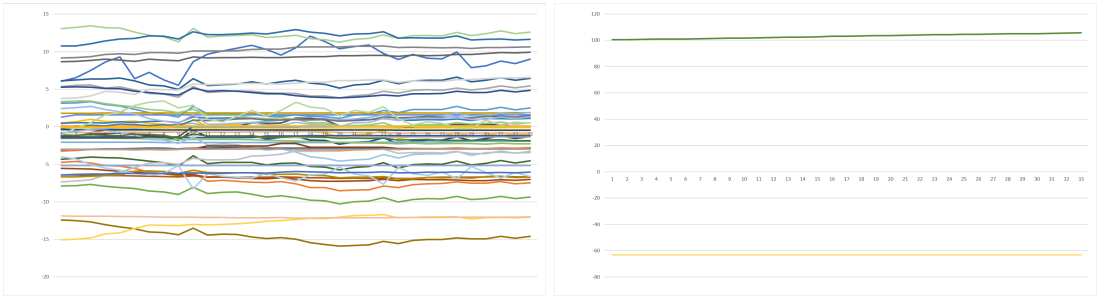
Figuur 5.1: Weight history vs. Stockfish

5.2. GrandQ vs. Alpha-Beta

Later hebben we nog een paar spelletjes gespeeld tegen onze alpha-beta agent. Deze waren bijna allemaal gewonnen.

Command:

```
python3 train -o ab --depth 3 -tt 15 -dt 0 -e 0.3 -d 0.3 -l 0.01 -q -i -1
```



(a) Weight history vs. Stockfish

(b) Check mate & Next Checkmate

Figuur 5.2: Weight History vs. Alpha-Beta

5.3. Git Repository:

Onze code en ons getrained bestand is ook terug te vinden op gitlab:

- Repository: https://gitlab.com/Artificiele_Intelligentie/chess
- Getraind bestand