AI som spelar Fyra i rad

Jag började med Jupyter Notebook (fyra-i-rad-version-1.ipynb) och testade lite olika modeller för en AI som ska kunna spela Fyra-i-rad. Där kom jag fram till att den bästa var en modell som kombinerade MCTS och Q-learning. Jag tränade upp en Q-agent i Jupyter och sparade ner resultatet i en PKL-fil med Pickle. Sedan tog jag koden som jag använt för och skapade en Flask-app för att få ett grafiskt interface. Det är denna Flask-app som är uppladdad i mitt GitHub repository. Även Jupyter-filen finns med.

Hur kom jag fram till en gångbar modell?

Nedan följer en kortare förklaring över hur jag testade mig fram med olika modeller för att hitta en som kunde spela så pass bra att jag kunde förlora mot den.

Steg 1 – själva spelet i textformat

Det började med att jag skapade ett väldigt enkelt, textbaserat Fyra-i-rad-spel i Jupyter där det gick att spela två personer mot varandra genom att skriva in siffran 1-7 för vilken kolumn man ville lägga sin pjäs i. Kontroller för giltiga drag och om någon vunnit fanns men ingen AI.

Steg 2 – slumpmässig AI-spelare

Jag skapade sedan en ”AI” som bara gjorde slumpmässiga, tillåtna drag. Denna var givetvis väldigt enkel att slå. Jag förbättrade den genom att skapa en ny som var något bättre. Den fick tre regler.

1. Kontrollera om det finns ett direkt vinnande drag och lägg i så fall där.
2. Kontrollera om motståndaren har ett direkt vinnande drag och lägg i så fall där för att blockera och förhindra en direkt förlust i motståndarens nästa drag.
3. Gör ett slumpmässigt, tillåtet drag.

Även denna var väldigt enkel att besegra. Man kunde ”lura” den genom att skapa två möjliga vinnande positioner så den inte kunde blockera båda. Flesta gångerna vann jag dock genom att den slumpmässigt skapade ett vinnande läge för mig.

Steg 3 (denna kod är bortplockad då det mest var en träning på SVM-modeller)

Jag lät den förbättrade slump-spelaren spela mot sig själv tusentals gånger och sparade ner varje match i en CSV-fil. Jag sparade inte själva spelets gång utan endast brädets utseende vi matchens slut samt vem som vann.

Efter det tränade jag en SVM-modell på detta utan att ha någon förhoppning om att det skulle fungera. Det gick att skapa en spelar men den spelade inte bättre än slumpmässigt.

Steg 4 – CNN

Jag hittade ett dataset på Kaggle som innehöll 376 641 rader där varje rad representerade brädet efter ett färdigspelat parti samt vem som vunnit det. Alltså samma data som min CSV-fil jag skapade ovan. Det var dock inte helt slumpmässiga drag utan datasetet hade skapats under tiden ett neuralt nätverk hade tränats så jag antog att datan var lite bättre än min och dessutom större.

Jag använde denna fil för att i min tur träna ett CNN-nätverk. Resultatet blev inte särskilt bra. Jag gav det dock en chans till genom att använda den i en minimax-modell (se nedan).

Steg 5 – Minimax med CNN

Jag försökte här förbättra CNN-modellen genom att använda den för att utvärdera positionerna i en minmax-algoritm. Jag använde här alfa-beta pruning för att få ner antalet noder att söka igenom.

Fortfarande var resultatet rätt dåligt – jag hade inga problem att vinna.

En minimax-algoritm i denna typ av spel kan bli oslagbar om den får söka igenom alla möjliga noder men det blir för stor i fyra-i-rad (i tic-tac-toe skulle det gå). Det hade dock säkert gått att fortsätta utveckla koden här och få min variant bättre. Jag hade kunnat utöka sökdjupet t.ex. men då hade den blivit långsammare. Jag tänkte dessutom att det finns en brist i att CNN-modellen tränas på statisk data där det inte framgår i vilken ordning dragen gjorts. För att få en bra AI-spelare måste den förstå spelets gång och inte bara slutpunkt. Så möjligt att utvärderingen i minimax-modellen inte gjorde särskilt bra. Jag ville dessutom fortsätta att testa andra modeller.

Steg 6 – Monte Carlo Three Search

En MCTS-modell är bättre än en minimax-modell om man inte har en bra heuristisk utvärdering (eller om man inte kan brute force-söka hela spelträdet). Jag använde ju min CNN-modell för utvärdering och det var inte så lyckat därför ville jag testa en MCTS-modell.

Vid detta laget la jag även till en värdering av varje enskild position på brädet enligt följande matris:

```
POSITIONAL_VALUES_RAW = np.array([
    [3, 4, 5, 7, 5, 4, 3],
    [4, 6, 8, 10, 8, 6, 4],
    [5, 8, 11, 13, 11, 8, 5],
    [5, 8, 11, 13, 11, 8, 5],
    [4, 6, 8, 10, 8, 6, 4],
    [3, 4, 5, 7, 5, 4, 3]
])
```

Här indikerar varje siffra värdet på den positionen på brädet. Positionerna i mitten är alltså värdefullare. Själva siffrorna kom jag fram till genom att beräkna hur många möjliga vinstrader det finns som innehåller den angivna positionen. T.ex. 3:an nere i vänstra hörnet anger att det finns tre olika vinstrader där den ingår – en vertikal, en horisontell och en diagonal. 4:an ett steg till höger visar att det finns fyra vinstrader – en vertikal, en diagonal, två horisontella (en från ytterkanten, och en en speljäs in från kanten).

Det ska tilläggas att jag säkerligen hade kunnat använda denna positionsvärdering i minimax-modellen för att få den bättre men detta var något jag kom på under tiden jag höll på med min MCTS-modell.

Denna positioneringsvärdering kommer in i tredje fasen i MCTS-algoritmen, simuleringsfasen. Här används inte ett helt slumpmässigt spel vilket är ganska vanligt vid MCTS-modellering utan här används en enkel heuristik istället. Först kollar den efter direkt vinnande drag, andra prioritet är att

blocka eventuellt vinnande läge för motståndaren och i tredje läge använder den positionsvärderingen. Det finns även ett slumpmässigt inslag på slutet.

När det gäller denna modell drog jag på en del när det gäller antalet iterationer för att den skulle spela bättre, men då blev den samtidigt långsammare. Den blev så pass långsam att de första dragen under en spelomgång kunde ta c:a 45 sekunder.

Modellen presterade bättre än de tidigare modellerna men jag kunde fortfarande vinna över den enkelt. Dessutom var den nu långsammast.

Steg 7 – Q-learning

Nu har vi kommit fram till det jag var mest intresserad av att testa. En reinforcement learning-algoritm. Något som bör vara bra på denna typ av problem som detta spel utgör.

Här började jag med att låta en Q-agent spela som första spelare mot den förbättrade slumpmässiga spelaren. Jag testade att låta den spela 100 000 matcher åt gången ett par gånger. Hela tiden sparade jag ner resultaten i en fil. Tyvärr är min dator inte så kraftfull så det tog lång tid att spela igenom dessa spel och ett par gånger på vägen kraschade den mitt i och då skadades filen så jag fick börja om med en helt ny fil. Jag testade att köra 500 000 matcher två gånger. Själva pkl-filen blev då väldigt stor (över 1 GB) men tyvärr kraschade den och blev oanvändbar. Jag fick nöja mig med färre matcher för att kunna hålla en fil intakt.

Prestationen från Q-agenten var inte så bra. Jag vann fortfarande enkelt. Jag testade dock att göra en förbättring av min modell genom att byta ut den slumpmässiga spelaren mot min Q-agent version 1 och låta träna upp en ny Q-agent version 2. Här spelade jag bara ett par 100 000 matcher. Detta sparade jag ner i en ny pkl-fil (connect4_q_agent_selfplay_gen2.pkl).

Jag testade att låta denna version spela mot min MCTS-modell och det visade sig att MCTS-modellen vann även när den fick spela som spelare 2 (spelar 1, som börjar, har en stor fördel i fyra-i-rad). Q-agenten var dock mycket snabbare. Min gissning hade från början varit att denna skulle vara den bästa modellen. Jag hade alltså fortfarande ingen modell som slog mig (det ska tilläggas att jag inte på något sätt är en särskilt bra spelare).

Steg 8 – Kombination av Q-learning och MCTS

Nu hade jag en idé om att testa att kombinera MCTS-modellen med min version 2 av Q-agenten. Jag bytte ut heuristiken i simuleringsfasen i MCTS-algoritmen mot min Q-agent. Jag sänkte även antal iterationer i MCTS något för att få den snabbare.

Nu visade det sig att jag fått en betydligt kraftfullare spelare. Den vann nu flertalet matcher mot mig. Jag kunde inte lura den alls på samma sätt längre utan den fintade snarare bort mig många gånger. Även som spelare 2 vann den för det mesta.

När jag kommit hit bestämde jag mig för att göra ett bättre gränssnitt så jag använde Flask för att få till ett spel i webbläsaren.

Det ska tilläggas att denna modell har en liten bit slumpmässighet inbyggd i sig. Detta är önskvärt annars skulle den alltid spela likadant om jag spelar samma sekvens av drag från gång till gång. Det skulle betyda att om jag kommer på ett vinnande spel och memorerar det så kan jag allt vinna efter det. Men nu finns en slumpmässighet här:

- I MCTS nod-expansion när en lövnod har untried_moves används slumpen för att avgöra
- Om det blir lika i en simulering med Q-agent så väljs ett drag slumpmässigt.

Jag tycker fortfarande denna modell är för långsam. I början tar dragen upp till 10 sekunder. Det går dock fortare mot slutet när det finns färre kombinationer kvar och den hittar blockar och vinnande drag på en gång.

Förbättringsmöjligheter

Det vore intressant att fortsätta utveckla detta med exempelvis följande, både för att få upp hastigheten och för att få den att spela bättre:

- Ersätta Q-agenten med DQN där q-tabellen ersätts av ett deep neural network.
- Parallella MCTS som kör samtidigt på olika processorer men då behöver jag en annan dator.
- Implementera ett policy-nätverk i likhetmed AlphaZero
- Införa en tabell över öppningsdrag likt vad som finns i schack så att den i början kan slå upp vilka som är de bästa dragen.
- Förbättra MCTS genom att implementera RAVE (Rapid Action Value Estimation) och AMAF (All Moves As First). Något jag inte satt mig in I ytterligare.
- Enklaste skulle dock vara att träna Q-agenten bättre. Skulle kunna göra en version 3 där den t.ex. tränas på min kombinerade modell. Det behövs dock betydligt bättre datorkraft då.