MCTS FOR Å SPILLE GO

AV ERIK BORGETEIEN HANSEN OG ODA ALIDA FØNSTELIEN HJELLJORD

Takk til Markus Johannes Pedersen for all hjelpen gjennom prosjektet. Takk til Ole Christian Eidheim, Donn Morrison, Jonathan Jørgensen for undervisning som vi brukte i dette prosjektet og for utsettelse som tillot oss å fullføre det.

ABSTRACT

I dette prosjektet ble det utviklet en implementasjon av Monte Carlo Tree Search (MCTS) som benytter et konvolusjonelt nevralt nettverk (CNN) til å gjøre avgjørelser under simulering av spill.

Kjøringer av MCTS ble brukt til å utforske spill og disse spillene ble brukt til å trene CNN modeller. For å teste treningen, ble modeller med forskjellige mengder trening spilt mot hverandre.

En feil i MCTS implementeringen førte til at CNN-modellene ble mer flinke som hvit-spiller enn svart-spiller, men modellene vinner over 80% av spillene mot tilfeldige trekk.

OPPGAVEBESKRIVELSE

Oppgaven var å implementere en kombinasjon av Monte Carlo Tree Search (MCTS) og et nevralt nettverk for å spille Go. Aigagrors GymGo [1] var oppgitt som miljø for Go. Oppgaven var relativt fri i forhold til hvordan det nevrale nettverket skulle implementeres.

MCTS kombinert med et nevralt nettverk er grunnlaget for AlphaGo. Oppgaven var i stor grad lagt opp til å lage en enkel, nedrustet versjon implementasjon av denne.

TEORI

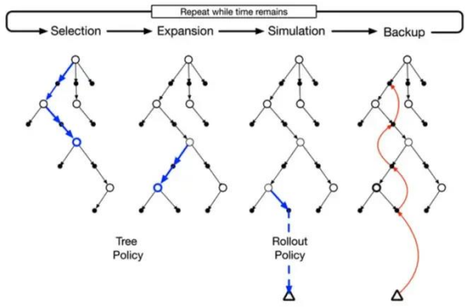
GO

Go er et asiatisk strategibrettspill for to spillere. Sammenlignet med sjakk har Go et større brett, lengre spill og, i gjennomsnitt, mange flere mulige trekk per tur. Antallet mulige posisjoner på et standard 19x19 brett har blitt kalkulert til omtrent 2.1×10^{170}. [4] Komi, en handikappregel i Go, er ikke relevant for denne oppgaven da brettet er for lite.

AlphaGo er kanskje verdens mest kjente spill AI og det første programmet til å slå en profesjonell Go-spiller. Den bruker både avanserte søketrær (MCTS) og dyp læring til å spille Go bedre enn noen tidligere algoritme. [2]

MONTE CARLO TREE SEARCH

MCTS bruker simulering til å gi en forventningsverdi til nodene i treet og fokuserer på mer lovende noder. På denne måten unngår man å teste alle trekkene som er mulig, noe som ville vært uholdbart tidkrevende for Go. Algoritmen består av fire hovedsteg; seleksjon, ekspansjon, simulering og tilbake-propagasjon som vist i illustrasjonen [5].



I seleksjons steget bruker vi en policy til å velge det mest lovende barnet til hver node. Dette er en løvnode som enten ikke er besøkt før eller den noden med høyest UCB.

Ekspansjon gjøres når en node har blitt besøkt før. Det gjøres ved å legge til alle lovlige trekk etter trekket til noden som barn til noden. Når dette gjøres velges et av barna for å simulere på. [6]

Under simulering ("rollout") spilles det ut et spill etter trekket i noden. Dette gjøres tilfeldig i vanlig MCTS, men i denne koden brukes et nevralt nettverk til å vekte hvilke trekk som velges. Når spillet er ferdig tilbake-propageres resultatet opp gjennom treet til alle foreldrene til noden. Resultatet legges til på verdien og antall besøk økes til hver node. [5]

METODE

For å bygge modellen vår har vi brukt PyTorch, hvor vi enkelt kan sette sammen en modell lagvis. Modellen bruker to konvolusjonslag, etterfulgt av todimensjonale "max pool" lag. Konvolusjonslagene går fra 6 kanaler (de 6 kanalene i en GymGo "state") til 64, til 128. "Max pool"-lagene benytter en kjernestørrelse på 2 og en skrittstørrelse på 2 (stride). Til slutt har vi et fullt sammenkoblet lag som gir oss en sannsynlighetsfordeling av alle mulige trekk, der høyest verdi er modellen sin prediksjon på beste trekk. Siste lag bruker "softmax" for å skalere prediksjonen.

For å trene modellene våre har vi iterert MCTS tusen ganger på tusen forskjellige noder. Under trening har vi gitt modellen et "bilde" av brettet og verdiene på alle trekkene derfra. Den første modellen fikk rene MCTS-trær å trene på, men de senere modellene fikk hjelp av tidligere modeller til å generere data i rollout-stadiet. I alt ble det trent fem modeller. For å teste modellene satte vi dem opp mot hverandre og simulerte opp mot 1000 spill.

En måte vi har forsøkt å redusere tiden det tar å trene er ved parallellisering. Dette har vi brukt både under testing av modellen våre ved å spille mange spill samtidig, og under trening for å generere mange trær samtidig.

RESULTAT

I testing av MCTS mot en spiller som velger tilfeldige trekk (tilfeldig-spiller) ble det oppdaget en feil i implementasjonen som førte til et bias for hvit spiller for MCTS. Sannsynligheten for MCTS til å vinne mot en tilfeldig-spiller ble beregnet til 0% dersom MCTS spilte som svart og 90% dersom MCTS spilte som hvit.

I modelltestingen ble biasen til MCTS tydelig. Hvit vinner mer enn svart uansett hvilken modell som spilte. Det ble også klart at flere treningsepoker hadde lite signifikans for forskjell mellom den mest trente og minst trente modellen. Det er tydelig fra tabellen under:

Svart vs hvit	*Svart %*	*Hvit %*	*Uavgjort %*
4 vs 0	40,4	57,7	1,9
4 vs 1	41,1	57,4	1,5
4 vs 2	44,2	53,3	2,5
4 vs 3	39,5	58	2,5
4 vs 4	36,8	60,9	2,3

DISKUSJON

Det ble tidlig i prosjektet klart at manglende dokumentasjon av Go miljøet [1] var en hindring. Mye tid ble brukt på å lage metoder som allerede var innebygd i miljøet eller å lese kildekoden til miljøet for å finne disse.

Med mer tid til å feilsøke hadde det vært mulig å finne årsaken til og korrigere feilen som gjør MCTS til å spille bedre som den ene fargen enn den andre. I implementasjonen som er lagt ved er det gjort et forsøk på å gjøre begge spillere bra, men hvit spiller bra og svart spiller dårligere enn tilfeldig, men ikke så dårlig som mulig.

Siden arbeidet startet med å utvikle MCTS ble denne implementasjonen gradvis utvidet etterhvert som nye behov ble oppdaget. Dette førte til at koden til MCTS ble mer uorganisert enn nødvendig. I videre utvikling kunne det vært ønskelig å lage en ny implementasjon eller refaktorere den eksiterende med den kunnskapen vi nå har.

Siden det ble laget en klasse for treet og en for nodene ble det gjort valg om hvilke metoder som skulle plasseres hvor. Dette var ikke absolutt nødvendig siden det er mulig å lage en implementasjon som bare har en nodeklasse, men det var ønskelig å ha en treklasse for to hovedgrunner: tilgang til CNN-modellen og enkel tilgang til totalt antall simuleringer. Siden treet skulle ha en CNN-modell som den kunne hente trekkvekter fra sparte det lagringsplass å ikke ha denne lagret i hver node. I forsøk på implementasjon av MCTS med bare nodeklasse ble det nødvendig å søke opp i treet etter totalt antall simuleringer. Selv om å traversere oppover i treet ikke er veldig tidkrevende, brukes ekstra tid hver seleksjon.

Det var usikkerhet rundt hvordan MCTS skulle generere treningsdata til CNN. Dette gikk hovedsakelig på hvor lenge tresøket skulle kjøre. Det vanlige for MCTS i henhold til litteraturen [5][6] er å kjøre søket i en gitt mengde iterasjoner, men det ble oppdaget at etter en stor mengde iterasjoner kjørte søket bare gjennom den samme grenen gjentatte ganger. På grunn av dette genereres heller flere trær med færre iterasjoner.

Det mest tidkrevende aspektet ved trening av modeller i denne oppgaven var generering av data fra MCTS. Dette tok såpass lang tid at vi satte koden til å kjøre over natten for å lage noen modeller. Dette resulterte også i at vi ikke fikk trent modellene våre så godt som vi ville, og heller ikke ofte nok. På grunn av dette rakk vi heller ikke å hyperparameteroptimalisere særlig, dog i etterkant ser vi måter vi kunne forsøkt på dette.

Når modellene våre spilte mot hverandre så vi fort at de gjentok det samme spillet om og om igjen. For å hindre dette har vi satt de to første trekkene til å være helt tilfeldige før modellen tar over, som hjalp betraktelig. Dette er nok på grunn av at vi trener alle modellene på lik data i starten. For å fikse dette kunne vi kanskje generert helt ny data til hver nye modell, i stedet for å gi den gammel data og ny data sammen. En annen ting som kunne vært behjelpelig er å optimalisere hyperparametere som læringsrate og momentum.

KONKLUSJON

MCTS ble implementert feil, noe som fører til at modellen blir trent til å være feil. De går mot å spille likt som MCTS, men siden MCTS bare spiller bra som hvit er dette ikke det ønskede resultatet.