DTE-2602 | H24

Karaktersatt oppgave 1

Q-Learning

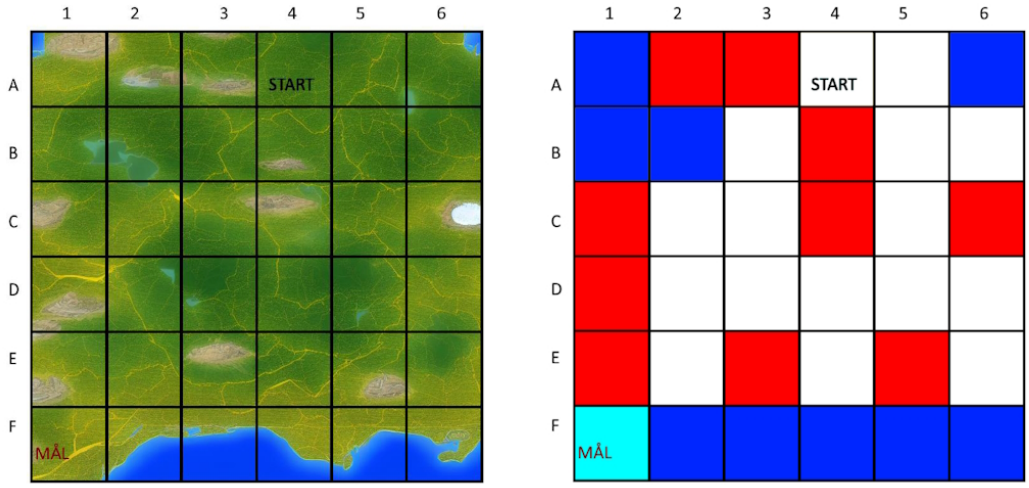
Bjarte Flø Lode

13. oktober 2024

1 Introduksjon

Et firma, *KI logistikk (KIL)*[1], skulle utvikle en robot for å transportere viktig last gjennom ukjent terreng. I utvikling av navigasjonsalgoritmene til roboten, brukte teamet deres *Supervised Learning* for å finne optimale løsninger i miljøer med kjente egenskaper. Dette fungerte godt i test-laben hos KIL, men når de beveget seg videre til å teste prototypen i et virkelig test-terreng, begynte roboten å oppføre seg merkelig. De klarte å justere modellen sin etter ny data fra test-terrenget, og roboten fant optimale veier. Men etter en litt skeptisk kollega tok en tur på testbanen i lunsjen, og endret litt på egenskapene i terrenget, begynte roboten å slite. Teamet ved KIL forstod at navigasjonsalgoritmene deres ikke egnet seg til å takle ukjente terreng, og det krevde mye datakraft å generere nye terreng-modeller hver gang miljøet endret seg.

Teamlederen hadde nylig lest en artikkel om «Reinforcement Learning (RL)», og foreslo at de skulle prøve ut denne læringsmodellen for roboten. Da ingen hos KIL hadde erfaring med RL, søkte de ekstern hjelp for å løse problemet. Som et utgangspunkt inkluderte de et av sine test-scenarier, som tar utgangspunkt i terrenget vist i figur 1a De inkluderte også en enkel klassifisering av terrenget, vist i figur 1b.



(a) «Flyfoto» (b) Forenklet klassifisering av miljøet.

Figur 1: Kart over terrenget roboten skal orientere seg i.

I figur 1b er terrenget delt inn i tre ulike klasser, i tillegg til den unike målcellen. Terrengklassene er bratt og krevende terreng (rød), vann (blå) og lett terreng (hvit). KIL fortalte at roboten klarer å ta seg frem i bratt terreng, men at det *krever mye energi* og er forbundet med *forhøyet risiko*. Å ta seg frem gjennom vann går greit, da roboten er vanntett, men dette krever *mer energi* og tar *svært lang tid* sammenlig-

[1] Firmaet og bakgrunnshistorien er oppdiktet.

net med å bevege seg gjennom de andre terrengene. I scenariet fra KIL er det også opplysninger om mulige sjømonstre i vannet, noe som ikke kan bekreftes, sammen med en risikovurdering av bevegelse i vann: *middels risiko*. De mulige bevegelsene roboten kan ta i rutenettet gitt i figur 1 er *opp*, *ned*, *høyre* og *venstre*.

Med utgangspunkt i denne problembeskrivelsen har vi fått i oppdrag å utvikle og dokumentere en læringsmodell basert på *RL*, der målet er å finne frem til en *policy* (strategi) roboten kan følge for å finne den **optimale** ruten til målet.

2 Teori

Robotens (heretter kalt agenten) reise frem til målet kan beskrives av en rekke tilstandsoverganger som skyldes agentens handlinger. Intuitivt forstår vi at agenten kan ta gode og dårlige valg, der gode valg maksimerer gitte målparametre, mens dårlige valg påvirker utfallet negativt.

Sammenhengen av valg, og hvordan valgene påvirker målparameterne, er imidlertid ikke nødvendigvis enkelt å se. Vi må altså ha en måte å skille gode og dårlige handlinger og tilstander fra hverandre, som vi kan gjøre ved å tilskrive en verdi til tilstander og handlinger, ut fra i hvor stor grad de leder til et godt resultat. I RL kaller vi denne verdien belønning, eller R for *reward*.

For at agenten skal kunne vite hva som er gode handlinger og tilstander, er vi nødt til å finne funksjoner som kan gi oss verdiene til handlinger og tilstander. Først kan vi definere et gitt miljø som et sett med tilstander $\mathcal{S} = s_0, s_1, \ldots, s_n$, og mulige handlinger $\mathcal{A} = a_0, a_1, \ldots, a_m$. Mengden $\mathcal{A}$ inneholder alle mulige handlinger for alle tilstander i S. Generelt kan handlingsrommet i en gitt tilstand s defineres som $A(s)$, der ($A(s) \subseteq \mathcal{A}$).

En handling a i en gitt tilstand s vil i deterministiske miljøer alltid ta oss til den samme neste tilstanden s'. I et stokastisk miljø, vil en handling a i en gitt tilstand s kunne ta oss til flere mulige neste tilstander. Vi kan uttrykke sannsynligheten for en overgang til mulige neste tilstander med sannsynlighetsfordelingen $\mathcal{P}$, som vil defineres nærmere senere.

Vi kaller verdifunksjonen, som gir oss verdien til en tilstand, for $\mathcal{V}$ og funksjonen som gir oss verdien av å ta en handling i en tilstand settes til $mathcalQ$. Belønningsfunksjonen er $\mathcal{R}$.

For at agenten skal kunne ta gode valg, trenger den en policy π som gir den beste handlingen for en gitt tilstand.

Videre skal vi se nærmere på hvordan vi kan bruke denne matematiske modellen av beslutningsproblemet til å finne *optimale policier* for agenten.

$$\mathcal{P}_a = \mathcal{P}(s'|s,a)|s'S \tag{1}$$

2.1 Modeller og RL

Noen prosesser kan fremstilles presist, slik at utfall er gitt fra egenskapene i den gjeldende tilstanden. En slik prosess er deterministisk, og så lenge man har tilgang til nødvendig data kan man foreta nøyaktige projeksjoner av fremtiden. Et slikt miljø kan representeres av en deterministisk modell. Et enkelt eksempel er at vi kan bruke enkle fysiske lover for å finne ut banen til et objekt om vi kjenner egenskaper som hastighet og akselerasjon. I et miljø uten friksjon og forstyrrelser er dette enkelt, men så lenge vi også kjenner til de andre variablene som påvirker objektets tilstand, vil vi fortsatt kunne operere i en deterministisk modell.

Andre prosesser domineres av tilfeldigheter. Det kan være noe så enkelt som å trille en terning. Ingen formel vil kunne gi et nøyaktig resultat på hva neste verdi på terningen vil bli —Dette er en *stokastisk* prosess. Selv om vi ikke kan gi en én-til-én-sammenheng mellom handlinger og tilstander, kan sannsynlighetsteori gi oss mye informasjon om slike prosesser. For en enkel prosess, kaste en terning, har vi tilstandene $\mathcal{S} = s_1, s_2, s_3, s_4, s_5, s_6$. Belønningsfunksjonen er gitt av $\mathcal{R}(s_t) = t \forall s_t \in \mathcal{S}$ Vi har bare én handling, kaste terning, og overgangene mellom alle tilstandene er like. Da kan vi beskrive alle overgangene i denne prosessen til neste tilstand s' med fordelingen $\mathcal{P}(s') = \frac{1}{6} \forall s' \in \mathcal{S}$. Ganske enkelt er det $frac16$ sannsynlig å få en av de 6 verdiene fra terningen når vi kaster den. Dette er kanskje ikke så spennende, men *Store talls lov* gir oss at gjennomsnittet av observasjoner nærmer seg sin *forventningsverdi* når antall observasjoner øker [1]. Ett enkelt tilfelle forblir et mysterium, men over tid vil utfallene konvergere til en fordeling. Forventningsverdien til terningkastet, altså snittet av verdien du vil få om du kaster terningen nok ganger, skriver vi som $\mathbb{E}[\mathcal{S}]$. Løsningen her er triviell, men formelen er interessant:

$$\begin{aligned} \mathbb{E}[\mathcal{S}] &= \sum_{t=1}^{6} t \cdot \mathcal{P}(\mathcal{S} = \mathcal{S}_t) \\ &= \sum_{t=1}^{6} t \cdot \frac{1}{6} = \frac{21}{6} = 3.5 \end{aligned} \tag{2}$$

Vi ser altså at selv i stokastiske miljøer kan vi bruke tradisjonell matematikk for å gjøre nødvendige beregninger. Så lenge vi kjenner $\mathcal{P}$ og $\mathcal{R}$, kan vi bruke denne modellen i *Model-based RL* for å finne optimal policy π for agenten. Når ligningene blir komplekse, kan også *dynamisk programmering* brukes til å bryte problemet ned i mindre deler og finne eksakte løsninger. Dette kan imidlertid være ressurskrevende, og ikke minst krever det at vi kjenner $\mathcal{P}$ og $\mathcal{R}$! Dette er ikke alltid tilfelle, særlig i tilfeller der en agent må forholde seg til naturlige omgivelser i stadig endring.

RL har en løsning for dette, og i *Model-free RL* kan vi bruke metoder for å finne optimale policier selv om vi ikke kjenner eller er i stand til å klart definere $\mathcal{P}$ og $\mathcal{R}$. Dette er nyttig for alle agenter som skal operere i «den virkelige verden», og dermed et viktig punkt for å løse problemstillingen i denne oppgaven.

2.2 Markovkjeder

Vi har sett at stokastiske prosesser er kjeder av tilstandsoverganger, der overgangen mellom en tilstand til en annen ikke er deterministisk, men styres etter sannsynlighetslover [2]. Rent intuitivt forstår vi at mange slike prosesser er avhengige av tidligere tilstander.

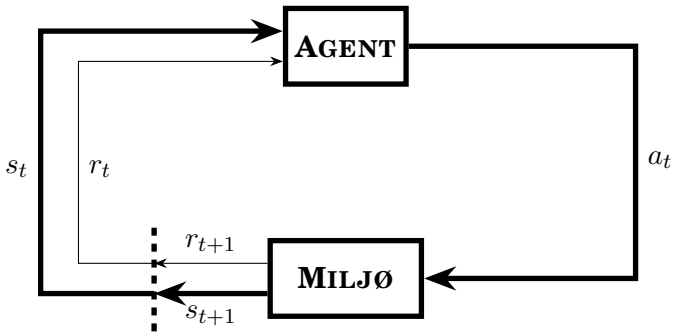
På den andre siden har vi enkle prosesser med uavhengige tilstandsoverganger, som terningkastet tidligere, hvor vi enkelt kan vise at fordelingen konvergerer over tid. Vi har også stokastiske prosesser der sannsynligheten fra å gå over fra en tilstand til en annen er avhengig av nåværende tilstand. Andrej Markov viste at også slike systemer konvergerer [3]. Her gjøres det en antakelse om at overgangen til neste tilstand bare er avhengig av nåværende tilstand, og ikke tidligere tilstander.

I figur 3 er et eksempel på en enkel markovkjede, som beskriver en kunde i en nettbutikk. Alle tilstandsovergangene, og sannsynligheten for å gå fra en tilstand til en annen, er fremstilt i en slik figur.

Markov-antagelsen, om at fremtidige valg bare er avhengig av nåværende tilstand, stemmer åpenbart ikke for alle sekvensielle beslutningsproblemer, men den holder for mange praktiske formål hvor en får gode tilnærminger til optimale policier [4].

2.3 Markov decision process (MDP)

MDP beskriver sekvensielle beslutningsprosesser, gitt at Markov-antagelsen kan legges til grunn. Markovrekker beskriver sannsynlighetsfordelingene for overganger fra en tilstand til en annen. MDP bygger på dette, men er interessert i å svare på hva som er gode handlinger for en agent å ta i en gitt beslutningsprosess i et stokastisk miljø. Siden Markov-antakelsen ligger til grunn, trenger vi ikke informasjon om tidligere handlinger og tilstander for å finne den beste handlingen i en gitt tilstand. Beslutningsprosessen MDP er fremstilt skjematisk i figur 2



Figur 2: Markov Decision Process

Som figuren viser, utfører agenten en handling a i miljøet, som resulterer i en belønning r og en ny tilstand s'. Denne prosessen fortsetter frem til man har nådd

en terminaltilstand. I *RL* søker vi å finne en måte å samle disse erfaringene på, for å komme frem til en optimal policy for beslutninger i det gitte miljøet. Kort sagt er målet å maksimere belønningen til agenten over tid. Vi er altså ikke like opptatt av umiddelbare belønninger, men søker å finne den policyen som gir størst belønning over tid.

Siden vi følger Markov-antakelsen, kan vi kun se på hva som skjer fra et gitt tidspunkt t og videre fremover til terminaltiden T, altså er vi interessert i alle r fra r_t til r_T. Men en fremtidig belønning er forbundet med usikkerhet i et stokastisk miljø, og om vi vekter fremtidig belønning likt som belønning nå kan vi ende opp med belønningsverdier som går mot ∞ og problemer med å få konvergens. Vi kan motvirke dette ved å tilskrive fremtidige belønninger en devalueringsfaktor (*discount factor*), $\gamma \in [0, 1]$. Da kan vi uttrykke den fremtidige belønningen fra t som $\mathcal{G}_t$ slik:

$$\begin{aligned} \mathcal{G}_t &\triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= \sum_{i=0}^{\infty} \gamma^k \mathcal{R}_{t+1+i} \end{aligned} \tag{3}$$

Vi kan utvide verdifunksjonene vi nevnte innledningsvis, slik at verdien til en tilstand s når policy π følges, og verdien av å ta en handling a i tilstand s under samme policy. Verdifunksjonen $\mathcal{V}_\pi(s)$ og handling-tistands-funksjonen $\mathcal{Q}_\pi(s, a)$ defineres som [7, s. 282]:

$$\begin{aligned} \mathcal{V}_\pi(s) &\triangleq \mathbb{E}\left[\mathcal{G}_t \mid \mathcal{S}_t = s\right] \\ &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s \right] \\ \mathcal{Q}_\pi(s, a) &\triangleq \mathbb{E}\left[\mathcal{G}_t \mid \mathcal{S}_t = s, \mathcal{A}_t = a\right] \\ &= \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s, a_0 = a \right] \end{aligned} \tag{4}$$

Her er $\mathbb{E}_\pi$ forventingsverdien gitt at policy π følges. $s_0 = s$ gir at dette gjelder når vi starter i tilstand s. Målet i MDP er som nevnt å finne en optimal policy, π^*, slik at vi kan maksimere $\mathcal{G}_t$:

$$\pi^* = \arg\max_\pi \mathcal{V}_\pi(s) = \arg\max_\pi \mathcal{Q}_\pi(s, a) \tag{5}$$

For å finne denne policien, kan vi bruke Bellman-likninger.

2.4 Bellman equation

Videre tar vi bare for oss tilstand-handling-funksjonen $\mathcal{Q}$. Bellman ligningen bryter ned verdifunksjonene vi så på over til to deler, den umiddelbare belønningen og devaluert fremtidig belønning:

$$\mathcal{Q}_\pi(s,a) = \mathbb{E}_\pi \left[r_{t+1} + \gamma \mathcal{Q}_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a \right] \tag{6}$$

Forventningsligningen uttrykkes her rekursivt, som åpner muligheter for numeriske løsninger og bruk av dynamisk programmering. Når vi fokuserer på den optimale policyen π^*, forenkles det ekspanderte uttrykket:

$$\mathcal{Q}^*(s,a) = \sum_{s',r} P(s', r \mid s, a) \left[r + \gamma \max_{a'} \mathcal{Q}^*(s', a') \right] \tag{7}$$

Ligningen over gir oss Q-verdien for å ta handling a i tilstand s. $[r + \gamma \max_{a'} Q^*(s', a')]$ er summen av den umiddelbare belønningen r den maksimale fremtidige belønningen multiplisert med devalueringsfaktoren γ. Siden vi opererer i et stokastisk miljø (MDP) er vi ikke garantert hva den neste tilstanden vil bli. Dermed må vi se på alle mulige fremtidige tilstander s' gitt handling a, gitt overgangssannsynlighetene. Fremtidige mulige Q-verdier summeres og vektes mot overgangssannsynlighetene med uttrykket $\sum_{s',r} P(s', r \mid s, a)$.

Dette kan løses rekursivt med dynamisk programmering, men det krever mye datakraft! Derfor bruker vi gjerne metoder for å approksimere verdifunksjonene, som Monte Carlo metoder og Q-learning

2.5 Monte Carlo

Monte Carlo metoder bygger på store talls lov *LNN*, som vi nevnte tidligere, og er inspirert av tilfeldigheter i spill, hvorav navnet Monte Carlo som stammer fra et kjent casino. Vi kan bruke Monte Carlo for å estimere Q, ved å sveipe over tilstand-handlings-rommet mange ganger ved å ta tilfeldige valg. Tanken er at med mange nok tilfeldige samplinger så vil Q være omtrent lik:

$$Q(s,a) \approx frac1N(s,a) \sum_{i=1}^{N(s,a)} G_i \tag{8}$$

Altså, tar vi snittet av alle G_t vi oppdager for paret (s,a), så vil vi nærme oss verdien til $Q(s,a)$, som av LNN vil konvergere til Q^*.

Siden vi trenger verdier fra en hel episode for å beregne G_t, kan vi ikke oppdatere verdifunksjonen vår før en episode er over. For å løse for Q bruker vi inkrementell beregning av gjennomsnittet, ved følgende algoritme etter hver episode:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)}(G - Q(s,a)) \tag{9}$$

Når vi velger handlinger helt tilfeldig, er vi utsatt for støy. Et enkelt eksempel er at samme tilstand kan besøkes mange ganger i løpet av en episode, uten at det ville vært nødvendig. Da kan verdien bli forskjøvet. Det finnes mange måter å løse dette på, men en enkel metode er at man bare inkluderer den første verdien for et par

(s, a) når man itererer over leddene i G_t etter en episode. Dette kalles *Monte Carlo First-Visit*.

2.6 TD og Q-learning

er fortsatt ikke helt banalt å beregne. men her bruker vi kraften av dynamisk programmering for å løse oppgaven. Gjennom erfaring gjør agenten inkrementelle oppdateringer av estimatet av Q-verdien som lagres i en tabell, med følgende formel:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{10}$$

Q-verdien for tilstand-handling-paret (s, a) kan estimeres med:

$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a') \tag{11}$$

Vi oppdaterer så gjeldende verdi for $Q(s, a)$ med differansen til det nye estimatet for Q, multiplisert med læringsraten α. Vi «antar» her at Q-verdien til neste tilstand er riktig. Det stemmer nødvendigvis ikke, og den vil være langt unna riktig verdi under utforskingsfasen. Derfor må vi være litt «skeptisk» til det nye estimatet for Q, og redusere oppdateringen med læringsraten. Over tid, når Q-tabellen oppdateres under agentens utforsking, er målet at Q-verdiene i tabellen skal konvergere til løsningen på Bellman-ligningen.

3 Metode

Med mål om å finne en optimal policy for agenten, må vi først lage oss en modell av miljøet. Fra figur 1, med $6 * 6 = 36$ posisjoner, kan vi definere miljøets tilstandsrom som $S = \{S_1, S_2, S_3, \ldots, S_{36}\}$., vist i tabell 1.

	1	2	3	4	5	6
A	S_1	S_2	S_3	S_4	S_5	S_6
B	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
C	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}
D	S_{19}	S_{20}	S_{21}	S_{22}	S_{23}	S_{24}
E	S_{25}	S_{26}	S_{27}	S_{28}	S_{29}	S_{30}
F	S_{31}	S_{32}	S_{33}	S_{34}	S_{35}	S_{36}

Tabell 1: Miljøets tilstandsrom

Vi følger videre Markov-antakelsen, at den beste handlingen for en tilstand s ikke er avhengig av tidligere tilstander. Da kan vi håndtere problemstillingen som

en MDP. Vi har ikke tilgang til miljøet eller numerisk data vi kan omsette til belønningsverdier for de ulike tilstandene. Vi må derfor bestemme fornuftige belønninger for ulike tilstander.

Siden agenten skal bevege seg i et stokastisk miljø, vil overgangene mellom tilstander ikke være deterministisk bestemt. Vi kan ikke garantere at en handling som har intensjon om å flytte agenten fra B2 til B3 ikke ender opp med at agenten beveges til A3 i stedet. Vi har valgt å se bort fra dette i denne runden av miljøsimulasjonen. Som vi så i teoridelen vil vi ved bruk av en *model free RL* modell, kunne gjennomføre læring av miljøet uten å kjenne $\mathcal{P}$. Det vil si at den samme læringsmodellen vil fungere både for miljøer med deterministiske og stokastiske tilstandsoverganger. Ved endringer i miljøets egenskaper, vil selvfølgelig agenten måtte justere sin policy om den skal forbli optimal. Dette er en del av styrken med RL, at agenten kan fortsette å lære og justere modellen til endringer i miljøet underveis. Dermed kan vi trygt demonstrere læringsmodellen uten å implementere stokastiske tilstandsoverganger.

Vi må likevel fastsette en $\mathcal{R}$ for miljøet som kan dele ut belønninger til agenten. Som vi husker fra figur 1, har vi tre ulike terrengtyper med forskjellige egenskaper. Belønningsfunksjonen vår må ta utgangspunkt i beskrivelsene av de ulike tilstandsklassene. Informasjonen gitt om miljøet har ble kodet til belønningsverdiene i tabell 2:

Tilstandsklasse	Energi og tid	Risiko	Belønningsverdi
Lett terreng	(0) Lav	(0) Lav	0
Krevende terreng	(-5) Høy	(-30) Høy	-35
Vann	(-10) Svært høy	(-10) Moderat	-20

Tabell 2: Belønningsverdier

Energi og tidsbruk har blitt gruppert sammen, og risiko har blitt vektlagt med en relativt høyere straff enn bruk av ekstra ressurser. Tanken bak dette er at det er viktigere at agenten kommer trygt frem enn at det brukes minst mulig tid. Belønningsverdien for måltistanden settes til 100. For å hindre roboten i å bevege seg utenfor tilstandsrommet, har vi valgt en stor straff på -100 for å prøve å bevege seg utenfor kartet, og agenten vil forbli i samme tilstand.

Vi har implementert agent og miljø som separate moduler. Miljøet er definert av tilstandene gitt av kartet, som vist i tabell 1. Miljøet genererer så en rewardmatrise, der tilstandsklassene tildeles belønninger etter 2. Tillatte handlinger er *nord, sør, øst, vest*. Miljøets API gir agenten informasjon om tilstandsrommets størrelse og mulige handlinger. Videre tildeler miljøet ny posisjon og belønning når agenten utfører en handling, og agenten blir da varslet om måltilstanden er nådd.

Agenten utfører handlinger basert på feedback fra miljøet og læringsmodellens anbefalinger. Erfaringen fra gjennomførte handlinger sendes til læringsmodellen, som oppdaterer sin Q-matrise.

Som læringsmodell skal agenten bruke både *Monte Carlo First Visit* og *Q-learning*. Q-learning modellen bruker ϵ-greedy som policy for å balansere «exploration» og «exploitation». Det vil si at vi velger en tilfeldig handling i $\epsilon \cdot 100\%$ av tilfellene. Begge læringsmodellene søker å finne tilnærminger for $Q*$ for alle tilstandene i miljøet. Verdiene lagres i en Q-tabell, med estimater for $Q*$ for alle parene (s, a) i tilstand-handlingsrommet.

En oversikt over de viktigste klassene og kommunikasjonen mellom dem er skissert i figur 4. Koden inneholder ellers hjelpemoduler for plotting av grafer og simulering av episodene med animasjoner.

I listing 5 er pseudokode av implementeringen av den Monte Carlo baserte læringsmodellen. Fra formlene i teori-delen så vi at vi kan bruke gjennomsnittet av G_t til å estimere $Q^*(s, a)$. Siden G_t er avhengig av terminaltistanden, trenger vi her komplette episoder for å beregne G_t og gjøre estimater på $Q(s, a)$. Snittet av G_t for (s, a) kan regnes ut rekursivt:

$$\overline{G_t} = \overline{G}_{t-1} + \frac{G_t - \overline{G}_{t-1}}{N_t} \tag{12}$$

Vi ser da at vi kan oppdatere estimatet av Q ved å iterere over erfaringsdata etter hver episode, der vi ser på total belønning G_t fra terminatiden T, bakover til starttiden t_0. Om vi holder styr på antall besøk i til hvert par (s, a) i en egen tabell $N(s, a)$, kan vi da enkelt finne frem til G_t og oppdatere Q for parene (s, a) i episoden. For å unngå støy og at enkelte par av (s, a) får for mye «trafikk» fra en episode, oppdaterer vi Q bare med ny data fra episoden for paret (s, a) som ligger nærmest terminaltilstanden.

For Q-Learning modellen vår bruker vi *Temporal Difference* for å oppdatere Q-estimatet. Dette tillater oss å oppdatere estimatet for Q ved hvert steg. Implementeringen i 6 følger formler fra teoridelen direkte.

For Q-Learning har vi også valgt å implementere en konvergenssjekk, som kjører etter hver endte episode. Den sammenligner hele Q-tabellen på slutten av episoden med et snapshot som ble tatt av Q-tabellen før episoden startet. Om absolutt og relativ toleranse er lavere en gitt grenseverdi, vil utforskingen stoppe automatisk. For å sikre god balanse mellom utforsking og utnytting av kjent kunnskap, har vi også implementert en dynamisk ϵ for ϵ-greedy policy. Den gjør at vi kan starte med en høy epsilon (nærme 1) for å sveipe over tilstandsrommet ganske tilfeldig de første episodene. Så reduseres epsilon eksponensielt, slik at den synker fort i starten for så å avta gradvis saktere. Slik sikrer vi både god utforsking i starten, for å få god dekning over alle mulige tilstander, samtidig som vi sikrer rask konvergens ved å redusere støy fra tilfeldige valg etter hvert.

4 Resultat

Først skal vi se på resultatene fra Monte Carlo simuleringen. Fra plots i figur 10 ser vi som forventet en ganske tilfeldig fordeling over tid. Vår implementering følger

samme policy under hele utforskingen, og endrer ikke strategi underveis. Frekvensplot for hvor ofte agenten har vært i de forskjellige tilstandene og (s, a) parene, viser også tydelig at agenten har fulgt en tilfeldig policy (figur 7). Agenten er upåvirket av belønning og straff, men har oppholdt seg mer lengre borte fra målet, som gir mening når vi vet at episoden slutter så snart målet nås. Q-tabellen ser noe lovende ut i figur 13, men er litt vanskelig å tolke.

Om vi kjører en grådig simulering av den Monte Carlo approksimerte Q-funksjonen, etter vi har utforsket 500 episoder, får vi et lovende utfall, se figur 9. Vi ser helt tydelig at agenten har lært en optimal vei gjennom terrenget.

For Q-læringen ser vi tydeligere trender i frekvenskartet etter gjennomført læringsfase. Dette er å forvente, da policy til denne læringsmodellen oppdateres underveis. Vi ser i figur 8 at agenten vår har funnet seg ruter den liker å ta, som ligner veldig på den grådige policyen vi fant med MC i figur 7. Plots av belønninger og antall steg per episode, er litt mer interessant for Q-læringen figur 11. Vi ser tydelig en relativt rask konvergering til en stabil total belønning per episode. Vi ser fortsatt varians mot slutten, som skyldes både tilfeldige startposisjoner og ϵ-greedy policy som også tar tilfeldige valg. I plottet er det valgt 0.1 og $\gamma = 0.8$. ϵ starter på 0.9 og ved konvergens er den rundt 0.1. Her har konvergensparametrene væt satt ganske strengt, med absolutt toleranse på 1×10^{-8} og relativ toleranse på 1×10^{-6}. Med noe lavere toleranser finner vi konvergens på rundt 500 episoder. Q-tabellen etter utforskingen er i figur 12.

5 Diskusjon og konklusjon

I denne oppgaven har vi sett på grunnleggende implementeringer av RL algoritmer for å estimere Q^* og den optimale policy π^* for å oppnå størst mulig belønning over tid, $\mathcal{G}_t$. Både Q-læringsalgoritmen og Monte Carlo fant optimaliserte policyer. Q-Learning algoritmen har fortrinn med at den som en TD algoritme oppdateres og bruker ny kunnskap med en gang. MC er avhengig av å fullføre en hel episode før lærinngen kan skje. Dette gjør at den henger litt igjen. På den andre siden sørger Monte Carlo for høy grad av utforsking, som sikrer at agenten ikke kjører seg fast i et sub-optimalt spor, og aldri finner den optimale policyen.

Ved å bruke en *decaying* $epsilon$, sikrer vi å få noen episoder med høy grad av utforsking, før policyen bruker mulighten til å utnytte lært kunnskap om miljøet i større grad. Da får vi også kjørt flere episoder med mindre varians fra tilfeldige beslutninger, slik at det er lettere å nå konvergens. Høyere grad av *exploitation* vil også forsterke de valgene agenten allerede har funnet ut er gode. Konsekvensen av dette ser vi noe av i sammenligningen av q-tabellene til MC og Q, i figur 13 og 12. Selv om QL her hadde kjørt flere episoder enn MC, ser vi at MC har mye lavere negativ skåre for bevegelsene som tar agenten utenfor brettet. I QL lærer agenten fort at dette ikke er lurt, og prøver dette i mindre grad igjen. MC, som har en helt tilfeldig policy, vil prøve seg på å ta disse valgene oftere. Men er ikke dette bra da? Både ja og nei, QL unngår uønskede tilstander i større grad, men si at det

plutselig ble «lov» å bevege seg til rutene øst for grensen i brettet, og det oppstod en ny terminaltilstand med belønning 200 her! Da ville MC funnet denne mye raskere enn Q, spesielt om Q hadde en lav ϵ.

Selv om QL og andre RL metoder kan lære underveis, og kan tilpasse seg nye miljøer, er måten de lærer på og tilpasser seg avhengig av hvordan de tunes. *Model free RL* er et kraftig verktøy når det er vanskelig å gi en komplett modell av miljøet vi skal jobbe i, men krever også at vi følger med på læringen og gjør nødvendige justeringer slik at agenten kan fungere effektivt både under læring, under drift og ved nye endringer i miljøet.

Referanser

[1] K. F. Frøslie, *Store talls lov*, i *Store norske leksikon*, 2022. [Online]. Hentet fra: https://snl.no/store_talls_lov Lastet ned: 08.10.2024.

[2] K. F. Frøslie, *stokastisk*, i 2023. [Online]. Hentet fra: https://snl.no/stokastisk Lastet ned: 15.10.2024.

[3] B. Hayes, «First Links in the Markov Chain,» *American Scientist*, mar. 2013. DOI: 10.1511/2013.101.92.

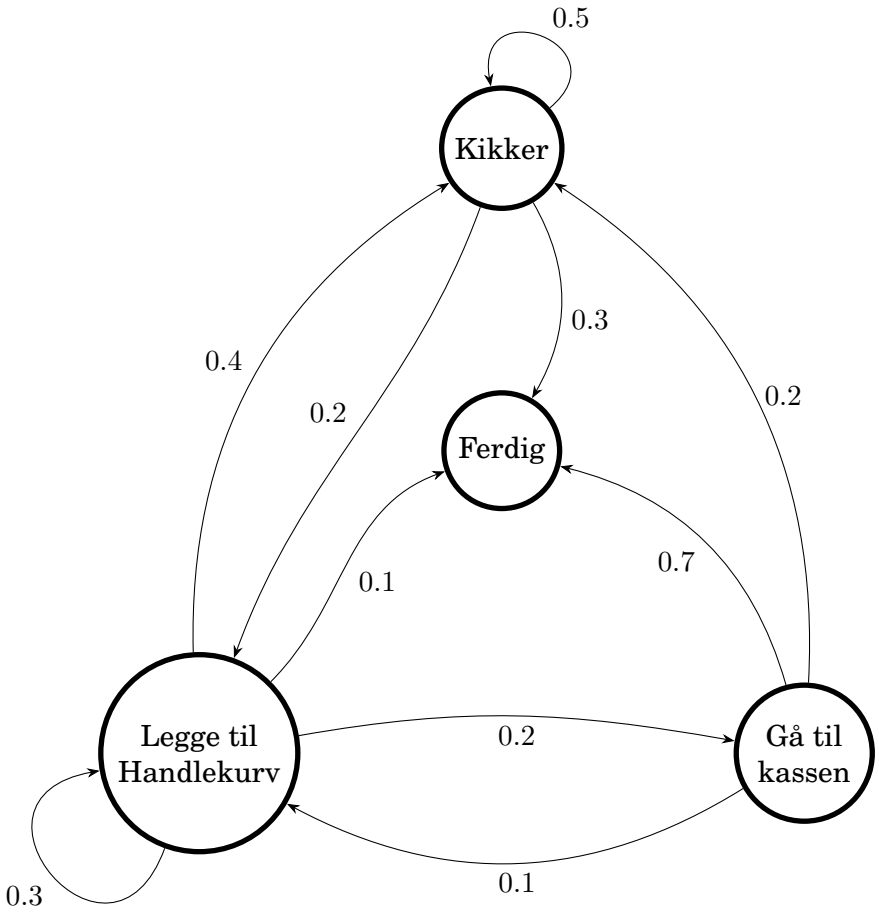
[4] C. Shi, R. Wan, R. Song, W. Lu og L. Leng, «Does the Markov Decision Process Fit the Data: Testing for the Markov Property in Sequential Decision Making,» i *Proceedings of the 37th International Conference on Machine Learning*, H. D. III og A. Singh, red., ser. Proceedings of Machine Learning Research, bd. 119, PMLR, 2020, s. 8807–8817. [Online]. Hentet fra: https://proceedings.mlr.press/v119/shi20c.html.

[5] R. Hurbans, *Grokking artificial intelligence algorithms*. Shutter Island, NY, USA: Manning, 2021.

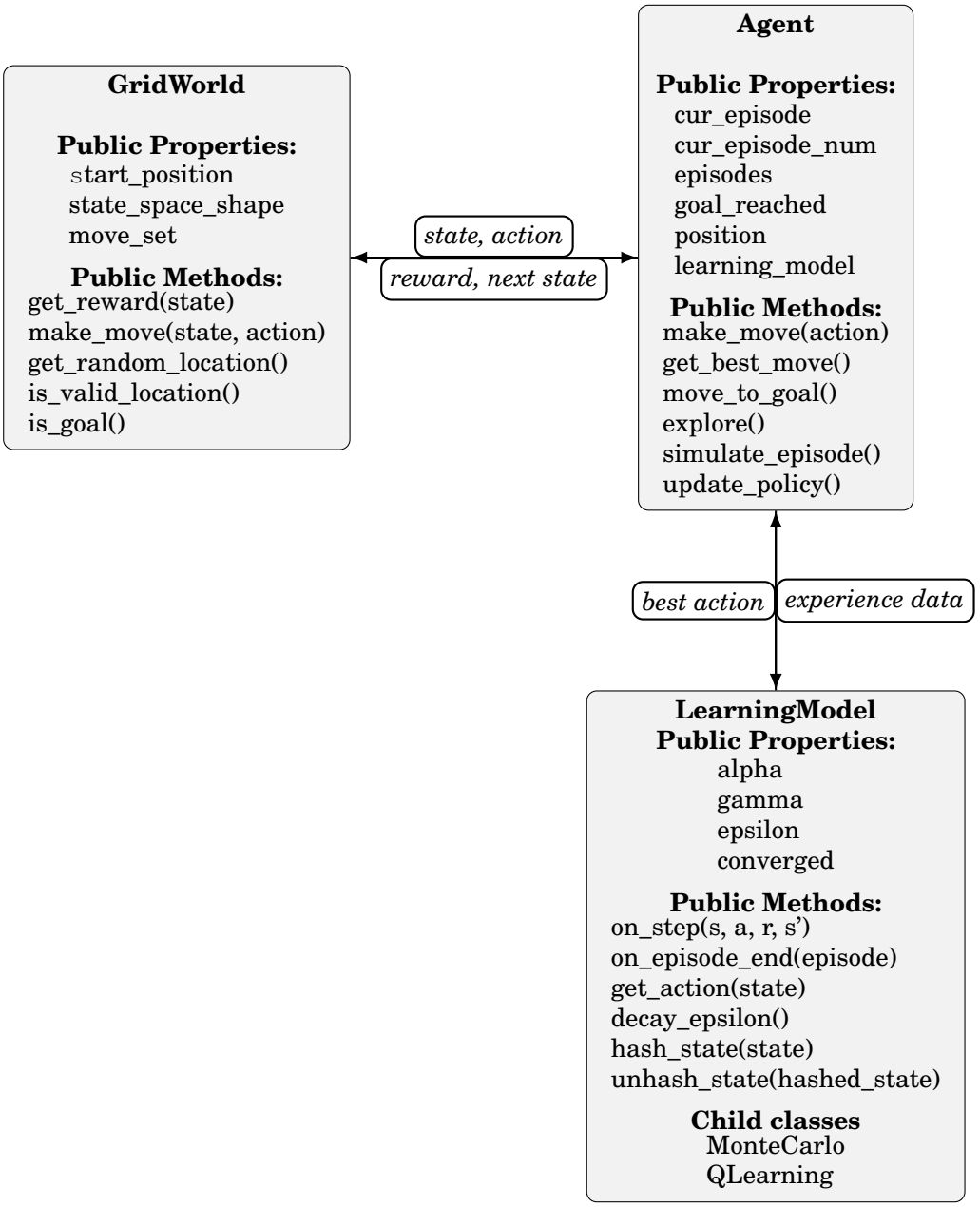
[6] B. Jang, M. Kim, G. Harerimana og J. W. Kim, «Q-Learning Algorithms: A Comprehensive Classification and Applications,» *IEEE Access*, årg. 7, s. 133 653–133 667, 2019. DOI: 10.1109/ACCESS.2019.2941229.

[7] J. Clifton og E. Laber, «Q-learning: Theory and applications,» *Annual Review of Statistics and Its Application*, årg. 7, nr. 1, s. 279–301, 2020. DOI: 10.1146/annurev-statistics-031219-041220.

6 Vedlegg: Figure mc



Figur 3: Markovkjede



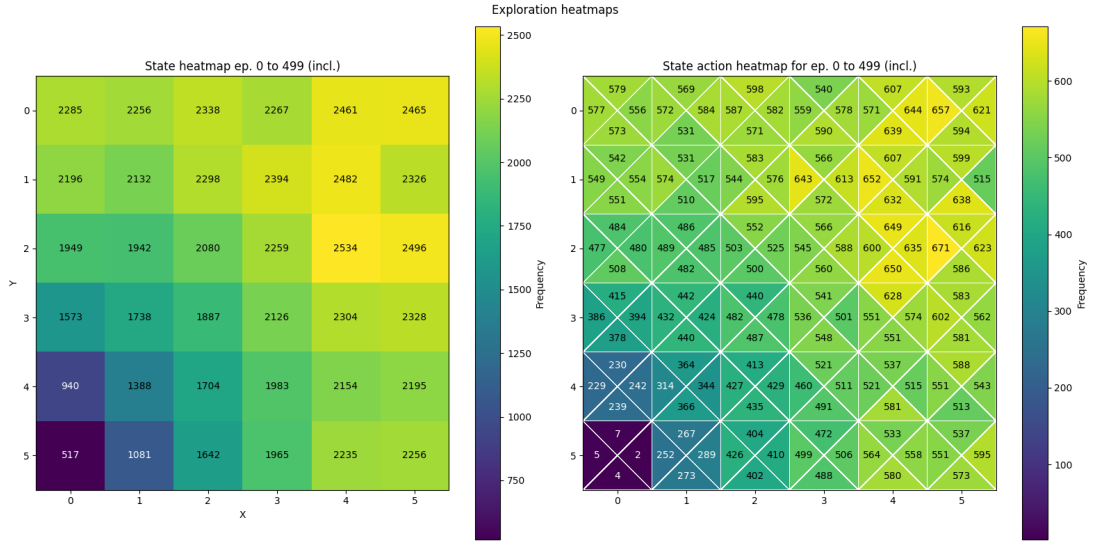
Figur 4: Main classes

```
Initialize Q(s, a)  # elements set to 0
Initialize N(s, a)  # elements set to 0
Set discount factor gamma
Set number of episodes to simulate

for n episodes:
    Run episode to terminal state taking random actions
    G = 0  # Total accumulative reward
    for each step in episode, starting from last step:
        if state not already visited this episode:
            G = gamma * G + r  # Calculates accumulative reward
            N(s, a) = N(s, a) + 1
            Q(s, a) = Q(s, a) + (G - Q(s, a)) / N(s, a)
```

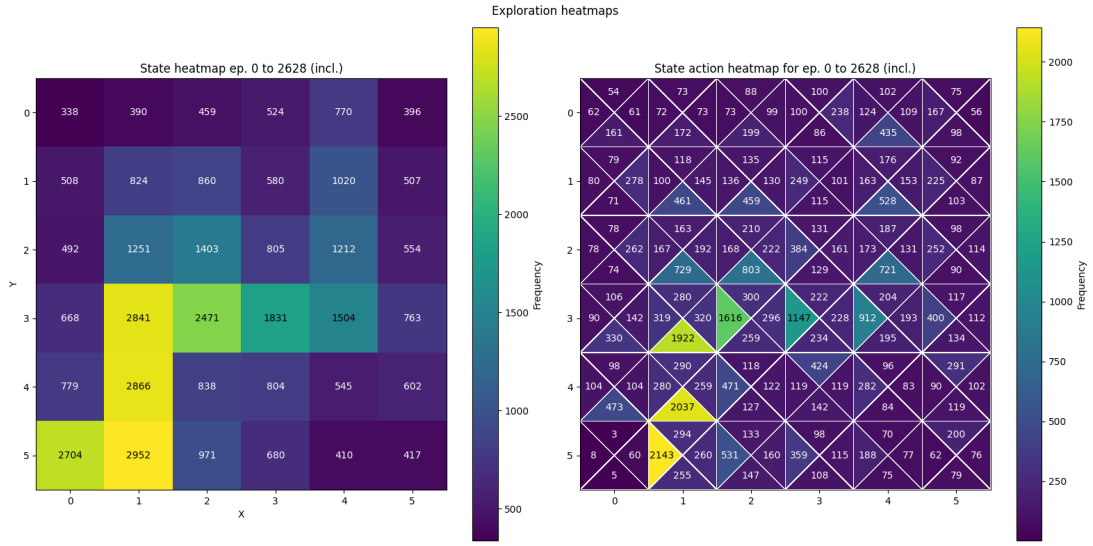
Figur 5: Implementering av Monte Carlo First Visit algoritme

```
Initialize Q(s, a)  # elements set to 0
Set hyperparameters alpha, gamma, epsilon

while not converged:
    Get random start state
    while state s is not goal:
        Get action with epsilon-greedy policy
        Take action a
        Receive reward r and next state s' from environment
        Update Q-value:
            Q(s, a) = Q(s, a) + alpha
                        * [r + gamma * max(Q(s', a')) - Q(s, a)]
        Set s = s'
```

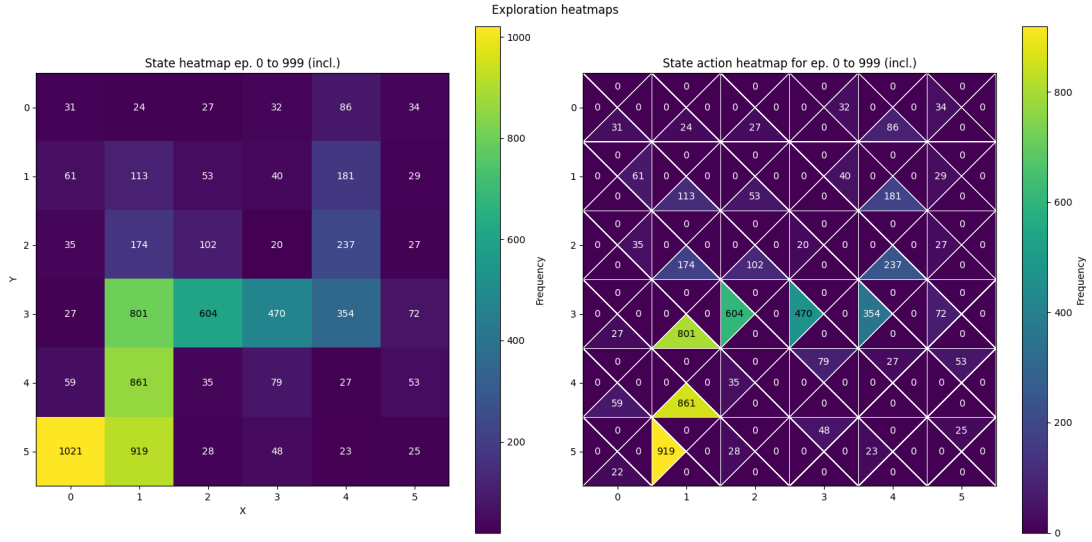
Figur 6: Implementering av Q-Learning algoritme



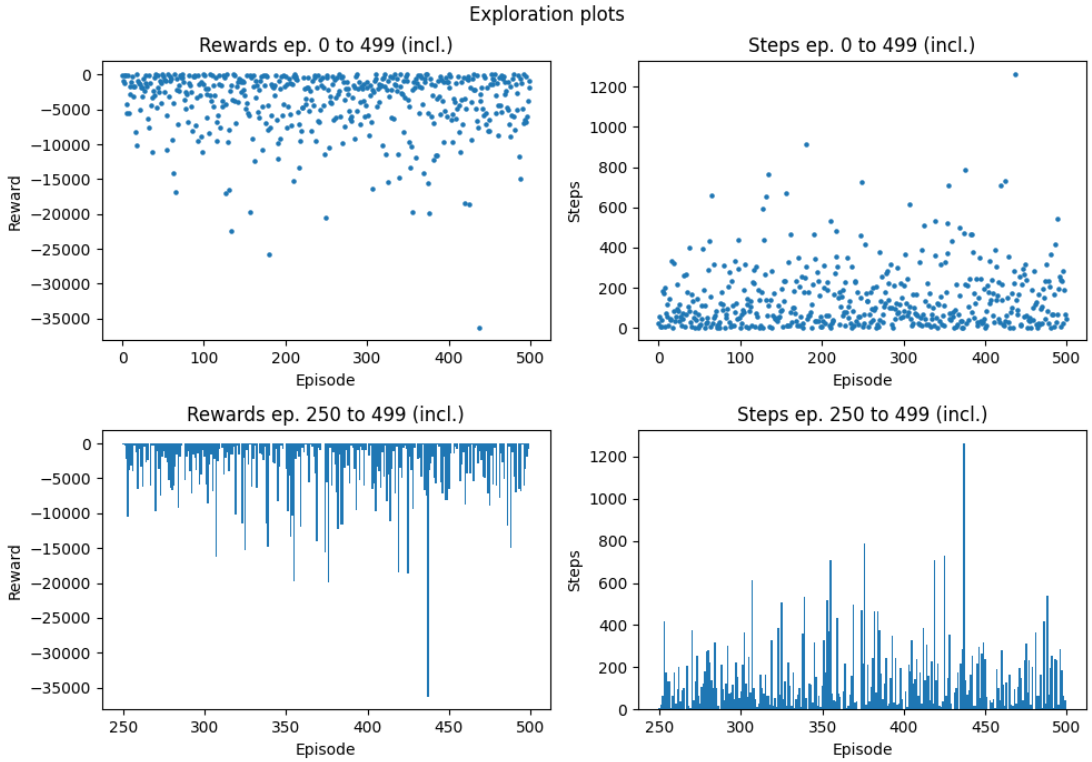
Figur 7: MC Heatmap



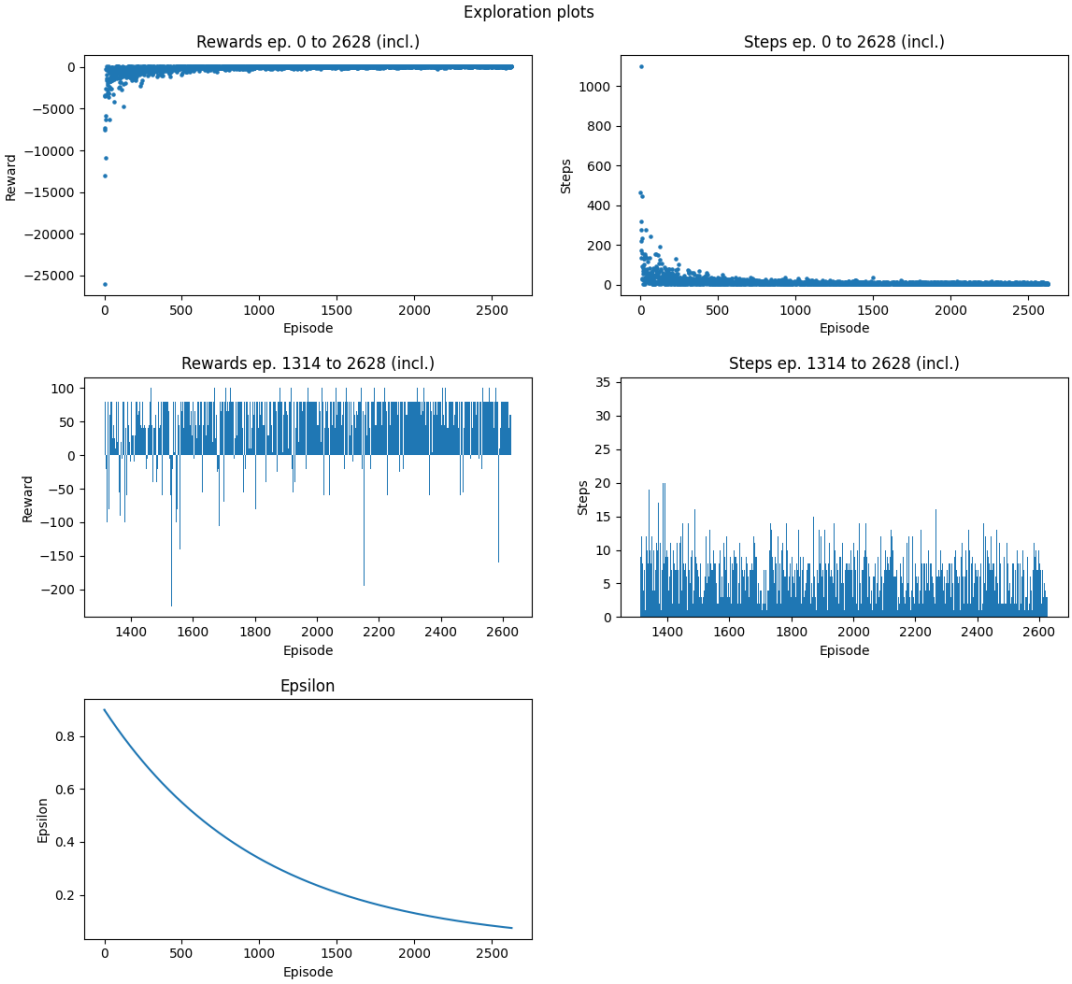
Figur 8: Q-Learning Heatmap



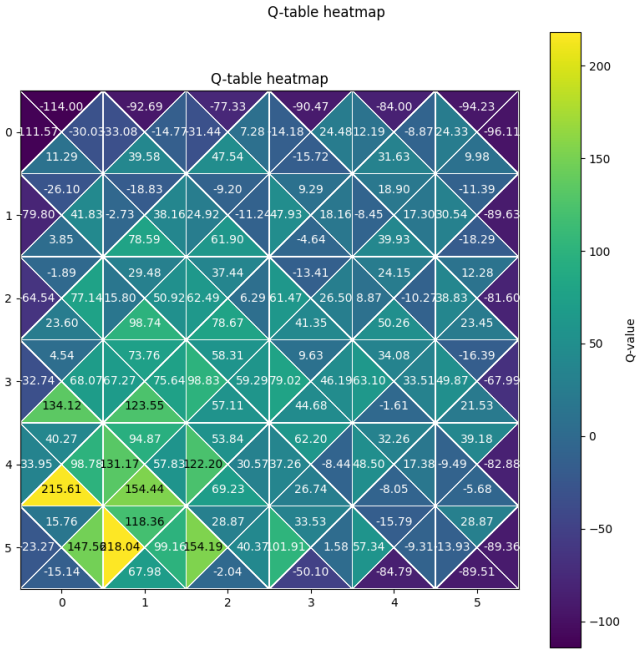
Figur 9: Grådig simulering etter 500 episoder med MC-læring.



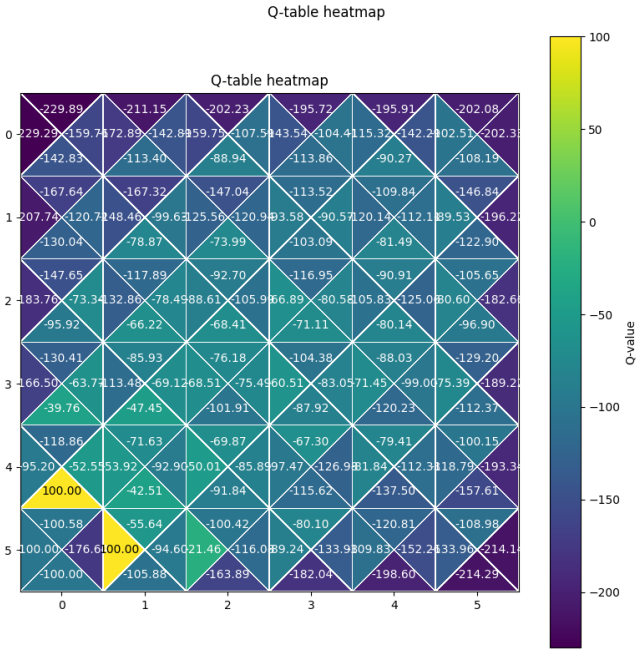
Figur 10: MC Plots



Figur 11: Q-Learning Plots



Figur 12: Q-Learning Q-table



Figur 13: MC Q-Table