Hvilken effekt har Prioritized Experience Replay (PER)?

Undersøkt på tre ulike miljøer

William Dalheim

November 2020

Abstract

Reinforcement Learning has seen many improvements over the last years. One of them is replacing the uniform-selection-based Experience Replay (ER), with Prioritized Experience Replay (PER) to achieve more efficient training. With this approach, the agent will favour some transitions over others. I used three different environments to study the effect of PER, namely CartPole-v0, Gym-Snake and the self-made CliffWalking1D inspired by Blind Cliffwalk. This was done using PyTorch and performing runs with and without the usage of PER. As expected, CliffWalking1D showed substantial improvements using PER. CartPole-v0 showed no significant improvements. Surprisingly, PER made Snake perform worse. It does however seem to exist measures in order to improve the effect of PER, like increasing the number of episodes.

Innholdsfortegnelse

1 Forord

Hensikten med rapporten er å dokumentere resultatene og metodene i tilknytning prosjektet i *Anvendt Maskinlæring* (TDAT3025) ved Norges teknisk-naturvitenskapelige universitet. Jeg ønsker å takke veilederen min William Chakroun Jacobsen for god oppfølging gjennom prosjektet. I tillegg vil jeg takke undervisere Ole Christian Eidheim, Donn Alexander Morrison, og Jonathan Jørgensen for god innføring i faget. Maskinlæring har lenge vært en interesse for meg, men noe jeg ikke har hatt tid og ressurser til å lære om. Dette faget har vært til stor hjelp i det å forstørre og realisere denne interessen.

Ordliste

agent Programmet som utforsker et miljø, og lærer fra erfaringer den har med miljøet. 2, 5, 12, 14, 15

episode En runde i et miljø. 2, 5

miljø En mengde tilstander og handlinger. Hver handling fra en tilstand gir en bestemt belønning. 2–6, 12–15, 17, 25

overgang En tuppel som inneholder tilstand, handlingen tatt fra denne, belønningen handlingen førte til, den etterfølgende tilstanden, og om episoden terminerte. I PER inneholder den også en prioritetsverdi. 2, 8

prioritized experience replay prioritert opplevelsesbuffer. 2

Forkortelser

DDQN Double Deep Q-Network. 2, 7

DQN Deep Q-Network. 2, 15

IS Importance Sampling. 2, 9, 23

MDP Markov Decision Process. 2, 5, 13

PER Prioritized Experience Replay. 1–4, 8, 11, 13–15, 17–25

RL Reinforcement Learning. 2, 3, 5, 21

TD Temporal Difference. 2, 7, 8, 24

2 Innledning

Hensikten med prosjektet er å ta i bruk kunnskapene og erfaringene dannet fra øvinger og undervisninger i TDAT3025 for å løse en problemstilling. Jeg har valgt å gå dypere i emnet Reinforcement Learning (RL) og undersøke hvilke effekter bruken av et prioritert opplevelsesbuffer (Prioritized Experience Replay (PER)) har på tre ulike miljøer. Feltet RL er i stadig utvikling og opplever mye forbedring. Jeg ønsker i dette prosjektet å ta fatt i en slik forbedring, og undersøke hva den faktisk innebærer.

Rapporten begynner med å presentere tidligere arbeid som er gjort i tilknytning PER. Etter dette kommer en teoridel som skal gi leseren de nødvendige kunnskapene for å forstå Reinforcement Learning og PER. Videre presenteres resultatene og metodene jeg brukte for å produsere disse. Til slutt analyseres resultatene for å finne ut effekten av PER.

3 Tidligere arbeid

I artikkelen om Prioritized Experience Replay (PER) (Schaul, Quan, Antonoglou & Silver, 2016) omtales et miljø de kaller for *Blind Cliffwalk*, og bruker dette som et eksempel der agenten har full oversikt over hvilke overganger som må trenes på for å konvergere løsningen raskest mulig. Jeg ønsker derimot å undersøke effekten av PER i praksis på dette miljøet.

I en artikkel om bruk av ulike varianter av opplevelsesbuffere på CartPole-v0 ble det konkludert at PER er ødeleggende (Wan & Xu, 2018, s. 7). I en annen artikkel hevdes det at PER gir forbedringer på det samme miljøet (Kumar, 2020, s. 7). Jeg ønsker å undersøke dette nærmere og finne en eventuell årsak til hvorfor forfatterne av artiklene fikk ulike resultater.

4 Teori

Teorien som presenteres i dette kapittelet har som formål å gi en forståelse av RL og andre konsepter som diskuteres videre i rapporten.

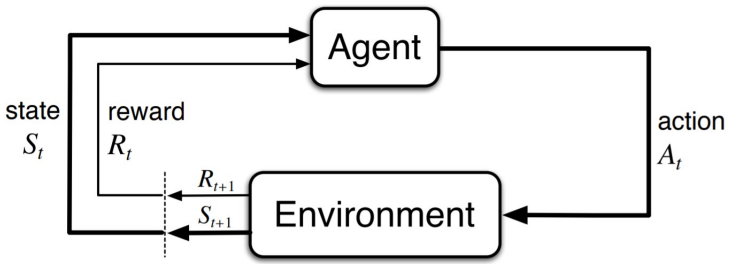
4.1 Reinforcement Learning

Den norske betegnelsen for reinforcement learning er forsterkende læring. Med forsterkende, menes det i denne sammenhengen å forbedre en oppførsel i flere iterasjoner med det målet om å nå en spesifikk ytelse. Vi studerer da hvordan en agent løser et problem mens den samhandler med et miljø. Hvilke handlinger agenten utfører i forhold til hvor den befinner seg i miljøet, bestemmer hvor stor belønning den mottar. Jo nærmere løsningen agenten befinner seg, jo mer belønning får den.

RL er en gren innenfor kunstig intelligens og ligger mellom veiledet læring og ikke-veiledet læring. Dette fordi man ikke spesifikt forteller hvilken oppførsel som er riktig, slik det er i veiledet læring. Derimot hinter man om en løsning gjennom et belønningssystem, som gjør at det ikke kan ligge under ikke-veiledet læring. Målet er å tilnærme en funksjon $f : \mathcal{S} \rightarrow \mathcal{A}$, der $\mathcal{S}$ er mengden som består av alle tillate tilstander, og $\mathcal{A}$ er en mengde som inneholder alle mulige handlinger som kan tas.

4.2 Markov Decision Process

En Markov Decision Process (MDP), er en matematisk modell som beskriver forholdet mellom vår agent og miljøet den befinner seg i. En slik prosess foregår i en syklus ved hvert tidssteg. Ved tidsteg t presenterer miljøet agenten for en tilstand S_t. Ut fra tilstanden bestemmer agenten seg for en handling A_t. Handlingen fører agenten inn i en ny tilstand S_{t+1} samtidig som den mottar en belønning R_{t+1}. Denne interaksjonen er illustrert i figuren under. (Sutton & Barto, 2018, s. 48).



Figur 1: Illustrasjon av hvordan en MDP foregår. Bildet er hentet fra Reinforcement Learning: An Introduction. (Sutton & Barto, 2018, s. 49)

Vi antar en runde å være ferdig når $t = T$. T er tidsteget når episoden terminerer. Tilstanden

S_T kan være løsningen, eller en tilstand der det ikke eksisterer noe vei til løsningen. Etter T iterasjoner av løkken i figur 1 kan episoden beskrives som en bane (trajectory):

$$S_0, A_0, R_1, S_1, A_1, \ldots, R_t, S_t, A_t, \ldots, R_{T-1}, S_{T-1}, A_{T-1}, R_T, S_T$$

S_0 er tilstanden som miljøet i starten av episoden presenterer agenten for. Å komme inn i denne tilstanden medfører ingen belønning. Tilstanden S_T er da episoden terminerer, og da vil ikke agenten kunne følge opp med en handling. (Sutton & Barto, 2018, s. 48-54)

Belønningen R er et reellt tall og gir en oppfatning av hvor god handlingen var i den gitte tilstanden. Hvis handlingen var god, vil R være positiv, og negativ hvis handlingen var dårlig. Er den negativ, er R en straff på agenten. Videre i rapporten brukes ordet belønning selv om den er positiv eller negativ. Verdien av R_t i forhold til den maksimale verdien den kan innta, forteller hvor mye vi verdsetter handlingen A_{t-1}. I RL ønsker vi å komme fram til en taktikk som maksimerer den forventede belønningen, som er summen av alle belønninger fram i tid. Fokuset setter vi derimot på de belønningene som befinner seg nærmest fram i tid. Dette modelleres med en diskonteringsfaktor $\gamma \in [0, 1]$ som multipliseres med belønningene. Hvis $\gamma = 1$ vil agenten ha like stort fokus på alle belønninger fram i tid. Hvis $\gamma = 0$, bryr den seg bare om R_{t+1}. Den forventede belønningen fra tidsteget t er da gitt ved:

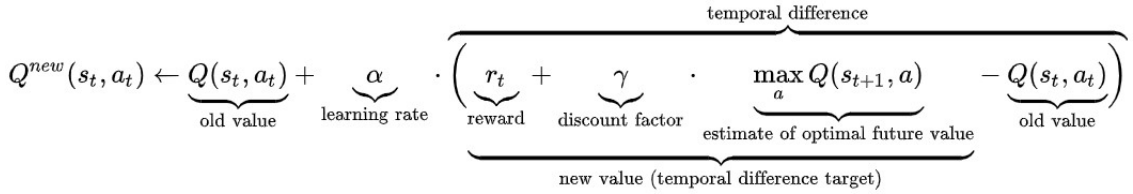
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T$$

(Sutton & Barto, 2018, s. 53-55)

Taktikken som brukes av agenten til å velge handlinger kalles for policy og noteres med symbolet π. Vi har verdifunksjonen $v_\pi(s)$, som angir den forventede belønningen ut fra tilstand s hvis agenten følger taktikken π. Fra denne funksjonen har vi kvalitetsfunksjonen $q_\pi(s, a)$ som angir verdien av å stå i en tilstand s og utføre handlingen a. (Fard & Pineau, 2011, s. 4-5) Den optimale taktikken kaller vi π_*, og tilknyttet denne, har vi den optimale kvalitetsfunksjonen $q_*(s, a) = \max_\pi q_\pi(s, a)$. (Sutton & Barto, 2018, s. 62-63)

4.3 Q-læring og DQN

I Q-læring jobber man med å tilnærme kvalitetsfunksjonen $q_\pi(s, a)$. Verdien av å ta en handling a_t i en tilstand s_t oppdateres ved hjelp av å anvende Bellman-likningen som vist i figur 2 under:



Figur 2: Bellman-likningen anvendt i Q-læring. Bildet er hentet fra artikkelen om Q-læring på Wikipedia. (Q-learning, 2020)

Med Q-læring i sin enkleste form, brukes en tabell for kvalitetsfunksjonen, med $|\mathcal{S}|$ rader og $|\mathcal{A}|$ kolonner. Men for miljø med store antall tilstander vil ikke en slik tabell være forsvarlig. Man kan da representere kvalitetsfunksjonen som et nevralt nettverk.

I figur 2 ser vi faktoren i det andre leddet:

$$\Big(r_t + \gamma \cdot \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t) \Big)$$

Dette kan skrives som til

$$(target - predicted)\,,$$

som er feilen på beregningen, og kalles for Temporal Difference (TD) (Q-learning, 2020). Denne feilen brukes til å trene det nevrale nettet.

Her vil man helst bruke et målnett Q_{target} for å regne ut $\max_a Q_{target}(s_{t+1}, a_t)$ og oppdatere dette med parameterne til det lokale nettet Q_{local} etter et bestemt antall tidsteg. Dette for å oppnå stabilitet siden man i DQN prøver å tilnærme seg en funksjon. Hvis ikke vil Q_{local} forsøke å tilnærme et mål den aldri vil nå.

4.4 Double DQN

I et Double Deep Q-Network (DDQN) lar man Q_{local} regne ut hvilken handling som passer til s_{t+1}. Man indekserer så denne handlingen inn i Q_{target} og bruker denne verdien i Bellman-likningen. Da blir feilen (TD) regnet ut slik:

$$r_t + \gamma \cdot Q_{target}(s_{t+1}, argmax(Q_{local}(s_{t+1}))) - Q_{local}(s_t, a_t) \tag{1}$$

(TheComputerScientist, 2019)

4.5 Opplevelsesbuffer

I delkapittelet om Q-læring så vi Bellman-likningen for å oppdatere verdien av et tilstand-handling-par. I alminnelig Q-læring utføres denne oppdateringen etter at agenten har havnet

i en ny tilstand. Istedenfor å forkaste informasjonen om hvordan agenten havnet i den nye tilstanden etter oppdatering av Q-funksjonen, kan man lagre det i et buffer til senere trening. Man kaller dette for et opplevelsesbuffer (Experience Replay). Dette bufferet vil ha en bestemt størrelse og fylles opp over tid. Når det er fullt, vil nye elementer erstatte de eldste.

Elementene som befinner seg i opplevelsesbufferet kaller vi for overganger. En slik består av en tilstand, handlingen tatt fra denne tilstanden, belønningen handlingen førte til, den etterfølgende tilstanden, og en verdi som forteller om episoden terminerte.

For hvert tidssteg henter man ut et utvalg med tilfeldige overganger fra opplevelsesbufferet, og trener på disse. Størrelsen på et slikt utvalg er en ny hyperparameter man må ta hensyn til.

4.6 Prioritert opplevelsesbuffer

Prioritized Experience Replay (PER) er en variant av opplevelsesbufferet omtalt i forrige delkapittel hvor vi introduserer en prioritetsverdi i overgangstuppelen. Prioriteten er høyere for overganger vi ønsker agenten skal trene mer på. Utplukkingen av overganger til utvalget foregår ikke lenger tilfeldig, men med hensyn på prioritetsverdiene. I dette prosjektet undersøkes den proporsjonale varianten som er en av to som omtales i artikkelen om PER. (Schaul et al., 2016).

Når vi legger til en overgang i bufferet setter vi prioriteten til den største prioriteten som allerede befinner seg der. Med dette sørger vi for at agenten trener på hver overgang minst én gang. Når denne overgangen en gang blir trent på, oppdaterer vi prioriteten med hensyn på den temporale differansen (TD). Nærmere bestemt setter vi prioriteten lik:

$$p_i = |TD| + \epsilon,$$

hvor epsilon er en liten verdi som sikrer at ingen null-verdier oppstår. Sannsynligheten for å trekke ut opplevelse i beregnes med formelen

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{2}$$

hvor k itererer over alle overganger i bufferet, og $\alpha \in [0, 1]$ er et hyperparameter som angir hvor sterk prioriteringen skal være. Merk, denne α er ikke læringsraten til det nevrale nettverket. Hvis $\alpha = 1$, vil $P(i)$ være $\frac{p_i}{\sum_k p_k}$, og vi får en sterk prioritering. Hvis $\alpha = 1$ vil $P(i)$ være $\frac{p_i^0}{\sum_k p_k^0} = \frac{1}{\sum_k 1} = \frac{1}{k}$. Altså tilsvarer $\alpha = 0$ det samme tilfellet som beskrevet i forrige delkapittel hvor det foregår et tilfeldig utvalg av overganger. (Schaul et al., 2016, s. 4)

Hvis vi bare trener på overganger som skaper store feil, vil det kunne føre til overskytning

i parameterne. Vi ønsker da å skalere ned gradientene med en vekt. Denne prosessen kalles Importance Sampling (IS). Vekten som skalerer gradienten for overgang i er gitt ved

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta}, \tag{3}$$

hvor $\beta \in [0, 1]$ er en hyperparameter som vanligvis starter med en liten verdi og går mot 1 gjennom treningen. N er antall elementer som befinner seg i bufferet ved tidsteget t. Når vektene er regnet ut for et utvalg, normaliserer vi disse slik at vi ikke forstørrer gradientene. (Schaul et al., 2016, s. 5)

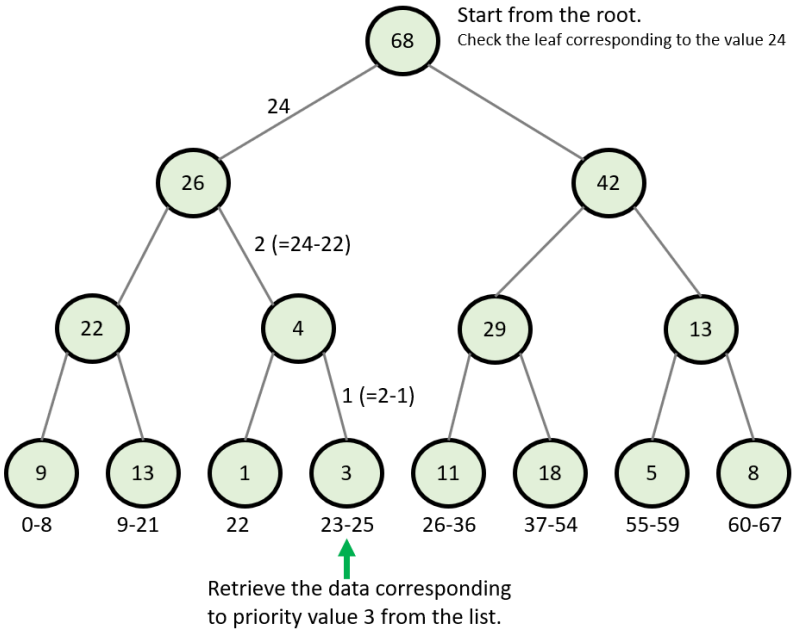
4.7 Sumtre

Et sumtre er et binært tre der alle noder er summen av sine barn. Av dette vil rotnoden være summen av alle ytternodene. I disse ytternodene lagrer vi prioritetene til elementene i opplevelsesbufferet. Da må treet bestå av $2 \cdot n - 1$ noder, hvor n er opplevelsesbufferets størrelse.

Fremgangsmåten for å hente ut noder fra treet er presentert nedenfor. $node_p$ er foreldrenoden, $node_r$ er høyre barnenode til $node_p$, og $node_l$ er venstre barnenode.

1. Velg en verdi v og begynn med $node_p$ lik rotnoden.
2. Undersøk $node_p$, hvis $node_p$ er ytternode, gå til 3
   - 2.1. Hvis $v \leq node_l$, sett $node_p = node_l$ og begynn 2 på nytt.
   - 2.2. Hvis $v > node_l$, sett $node_p = node_r$ og sett $v = v - node_l$ og begynn 2 på nytt.
3. Returner $node_p$

Et eksempel på en slik uthenting er illustrert i figur 3 under.



Figur 3: Eksempel med henting av node. Bildet er hentet fra (Ramesh, 2019)

I figuren over ser man at hver node danner et intervall med startverdi lik summen av alle noder i rekken før den og har lengde lik nodens verdi. Verdien man fører inn ligger innenfor dette intervallet. Hvis man velger et tilfeldig tall mellom 0 og 68, er det størst sannsynlighet for at man havner på noden med verdi 18 i figuren, da denne noden dekker $\frac{18}{68}$ av hele treet.

5 Fremgangsmåte

5.1 Loggføring av data

Tidlig i prosjektet brukte jeg matplotlib (Hunter, 2007) for å loggføre belønning og tidsbruk under trening. Etter kjøringer, lagret jeg figurene som bilder for å studere. Senere tok jeg i bruk Tensorboard, som er bedre tilpasset maskinlæring.

5.2 Kode

I dette delkapittelet presenteres klassene som er produsert i tilknytning prosjektet. Kildekoden til hver klasse, med unntak av Net, er lagt som vedlegg nederst i dokumentet. Kildekoden er godt kommentert slik at den skal kunne forstås.

5.2.1 Replay Buffer/Experience Replay

Dette er den varianten av opplevelsesbuffer som ikke bruker prioritert utvalg. Klassen `ReplayBuffer` ligger i `utils.py`.

5.2.2 Sumtre

Koden for sumtreet er inspirert av leksjonen om Prioritized Experience Replay (PER) på CartPole-v0 (Balsys, 2019). Klassen `SumTree` ligger i `utils.py`. Grunnen til at jeg bruker et sumtre i implementasjonen av PER er for å oppnå et balansert utvalg og unngå at kompleksiteten av å hente ut overganger avhenger av antall elementer som er lagret. (Schaul et al., 2016, s. 4)

5.2.3 Prioritized Experience Replay

Denne varianten av opplevelsesbuffer bruker prioriterte elementer. Klassen `PrioritizedReplayBuffer` befinner seg i `utils.py`. Denne klassen er lik `ReplayBuffer`, med unntak at den må forholde seg til sumtreet og bestemme prioritetsverdier for overgangene. I metoden som legger til overganger, gjorde jeg slik at prioriteten settes lik summen av den maksimale prioriteten i treet og belønningen overgangen medførte. Det første leddet sørger for at overgangen vil bli trent på minst én gang. Ved å bruke belønningen som del av prioriteten, sørger vi for et større fokus på overganger som gir belønning.

I metoden som regner ut viktighetsvektoren fra likning 3, tok jeg inn β som et parameter. Dette gjør at jeg kan endre denne verdien gjennom læringen slik som forseslås i artikkelen om PER. (Schaul et al., 2016, s. 5).

5.2.4 Net

Hvert miljø brukte ulike arkitekturer på det nevrale nettverket. Klassene har derimot samme struktur. Disse klassene ligger i mappene for hvert miljø med filnavnet `model.py`.

5.2.5 Agent

Klassen `Agent` befinner seg i `utils.py`. Denne er programmert slik at den skal kunne brukes til alle miljø med diskrete handlingsrom. I konstruktøren spesifiseres hyperparameterne og det nevrale nettet agenten skal bruke. Grunnen til dette er fordi de nevrale nettene må tilpasses formatet på tilstandene til det enkelte miljøet.

Denne klassen inneholder også metoden for å regne ut ϵ, utforskingssannsynligheten. Formelen som er brukt i denne metoden kom jeg fram til ved å utforske ulike funksjoner i GeoGebra (Hohenwarter et al., 2020). Jeg studerte spesielt når minimumsverdien ble nådd etter et bestemt antall episoder i forhold til antall episoder som skulle kjøres. Med formelen vil agenten nå minimumsverdien etter at 75% av episodene er ferdige, noe jeg tenkte var passende.

5.2.6 Mainmetoder

Siden jeg anvendte klassene over på flere ulike miljøer, presenterer jeg nå hovedtrekkene alle `main.py`-filene hadde, med noen eksempelverdier. Dette viser også hvordan jeg brukte Tensorboard for loggføring. Koden presenteres i deler.

```
# Imports #
USE_TB = True
SAVE_MODEL = False

# Hyperparameters
GAMMA = 0.98
ALPHA = 0.000075
MAX_MEMORY_SIZE = 50000
NUM_EPISODES = 1000
BATCH_SIZE = 64
REPLACE_AFTER = 100
USE_PER = (sys.argv[1] == "True")

RUN_ID =
↪  ("GAMMA(%.2f)-ALPHA(%f)-MEMSIZE(%d)-EPISODES(%d)-BATCHSIZE(%d)-REPLACE_AFTER(%d)-PER(%r)" %
↪  (GAMMA, ALPHA, MAX_MEMORY_SIZE, NUM_EPISODES, BATCH_SIZE, REPLACE_AFTER, USE_PER))
time_started = datetime.now().strftime("%d.%m.%Y-%H:%M:%S")
if USE_TB: writer = SummaryWriter("logdir/env-%s-%s" % (RUN_ID, time_started))

env = gym.make("environment-name")
agent = # Create Agent
```

Først spesifiseres hyperparameterne. En beskrivelse av disse er gitt under. Videre lages en unik identifikasjon for denne spesifikke kjøringen av programmet. Denne identifikasjonen brukte jeg til å gjenkjenne ulike kjøringer senere når resultatene skulle undersøkes.

GAMMA	Diskonteringsfaktoren γ
ALPHA	Læringsraten α
MAX_MEMORY_SIZE	Størrelsen på opplevelsesbufferet
EPISODES	Antall episoder som skal kjøres
BATCH_SIZE	Antall overganger som skal hentes ut fra bufferet når agenten skal lære
REPLACE_AFTER	Antall læringsteg mellom hver erstatning av målnettet
USE_PER	Spesifiserer om det skal brukes PER

```
21 print("Filling replay memory")
22 state = env.reset()
23 for t in range(2000):
24     action = env.action_space.sample() # Explore
25     next_state, reward, done, info = env.step(action)
26     agent.store_transition(state, action, reward, next_state, done)
27     state = next_state
28     if done:
29         state = env.reset()
30 print("Finished filling memory")
31 print("Started learning")
32 for i_episode in range(NUM_EPISODES):
33     state = env.reset()
34     done = False
35     while not done:
36         action = agent.choose_action(state, i_episode, train=True) # Explore or exploit
37         next_state, reward, done, info = env.step(action) # Perform action
38         agent.store_transition(state, action, reward, next_state, done) # Store transition in
           ↪  replay buffer
39         state = next_state
40         agent.learn(BATCH_SIZE, i_episode) # Perform learning step
41     if USE_TB:
42         # Log metrics
43
44 if SAVE_MODEL: torch.save(agent.Q_loc.state_dict(), "Path-to-save/trainedmodel-%s" % (RUN_ID))
45 if USE_TB: writer.close()
```

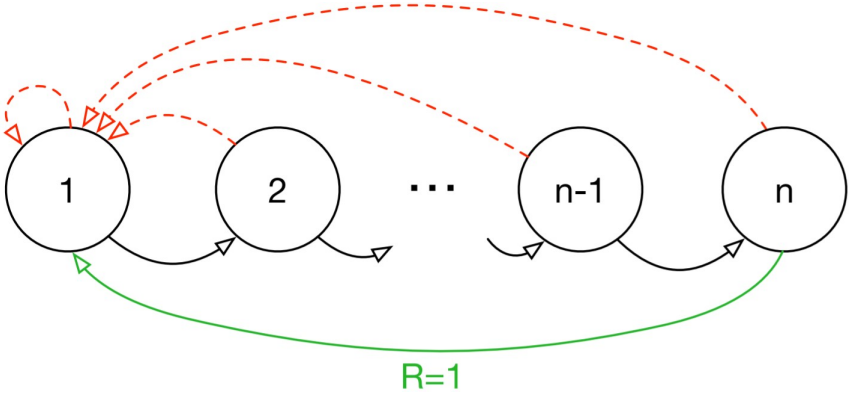
Den første for-løkken gjør at agenten vår kan utforske litt av miljøet før den begynner å lære. I den neste for-løkken begynner vi med å hente ut starttilstanden s_0 og setter i gang en while-løkke som kjører fram til agenten når en terminerende tilstand s_T. Hvis agenten ikke er i tilstanden s_T, avgjører den en handling a_0. Ved å utføre denne handlingen, havner den i tilstand s_1 og mottar belønningen r_1. Denne overgangen lagres i opplevelsesbufferet. Før neste iterasjon av while-løkken ser den på tilstanden s_1 som forrige tilstand. Vi ser at denne løkken er en MDP

som presentert i teorikapittelet.

Antall iterasjoner av den første løkken kan anses som et hyperparameter da den kan påvirke i hvor stor grad agenten får nytte av det prioriterte opplevelsesbufferet. Som tidligere nevnt, sørger jeg for at alle overganger som legges i bufferet skal bli trent på minst én gang. Jeg opplevde da noen problemer hvis jeg brukte denne løkken til å fylle opp bufferet. I noen tilfeller var bufferet for stort til at agenten ville klare å overskrive det før kjøringen ble ferdig. Da ville man ende opp med at agenten hadde brukt en stor del av episodene på å trene seg gjennom bufferet, og hatt lite trening på overgangene den hadde funnet selv. Derfor valgte jeg alltid å holde dette tallet lavt, og la den utforske mest under læringsprosessen.

5.3 CliffWalking1D

I artikkelen om PER (Schaul et al., 2016), beskriver de et ideellt miljø for en agent som bruker et prioritert opplevelsesbuffer. Miljøet består av N tilstander som ligger i rekke. Hver tilstand har 2 handlinger. Den ene er å gå et skritt framover, mens den andre er å falle ned en klippe. Fullfører man siste tilstand mottar man belønning, mens alle andre tilstander gir ingen belønning.



Figur 4: Illustrasjon av Blind Cliffwalk. (Schaul et al., 2016, s. 3)

Et slikt miljø tenkte jeg ville være bra for å teste ut implementasjonen med PER. Jeg fant ikke dette miljøet på nettet, og bestemte meg derfor for å lage det selv. Klassen CliffWalking1D befinner seg i filen `cliffwalking1d.py`. Denne bruker ikke OpenAIs offisielle struktur for å sette opp miljø, men inneholder de nødvendige metodene.

I dette miljøet er starttilstanden $s_0 = 0$. Hvis agenten velger riktig handling vil den havne i tilstand $s_1 = 1$. Tilstandsrommet er altså et intervall fra 0 til n. Hvis agenten fullfører tilstand s_n, får den 1 i belønning, og havner i en terminerende tilstand. Ellers får den 0 i belønning. I hver ikke-terminerende tilstand har agenten valget mellom to handlinger, 0 og 1.

5.4 Trening med CliffWalking1D

Å finne en balanse mellom n, og antall episoder var spesielt krevende. Tidlig forsøkte jeg med for eksempel 13 noder og 20 000 episoder. I dette tilfellet vil en agent i gjennomsnitt nå målet 2.44 ganger, hvis den kunne utforsket hele tiden. Da opplevde jeg at noen agenter ikke nådde målet i det hele tatt. I tillegg måtte resten av hyperparameterne endres ved prøving og feiling. Til dette miljøet brukte jeg et nevralt nettverk bestående av tre lineære lag, med ReLU mellom disse.

5.5 Trening med CartPole

Som tidligere nevnt, kom jeg over to artikler, der den ene hevdet at PER ga forbedringer (Kumar, 2020, s. 7), mens den andre sa at PER kunne være ødeleggende (Wan & Xu, 2018, s. 7). I begge disse artiklene ble det brukt DQN, i motsetning til DDQN som jeg har brukt på Snake og CliffWalking1D. Jeg valgte derfor å ta i bruk DQN for dette miljøet.

Forfatteren som påstår at PER gir forbedringer, brukte et nevralt nettverk bestående av tre lag med 24 nevroner, til 24 nevroner, til 2 nevroner. Jeg forsøkte å kopiere denne strukturen for å gjenskape resultatene.

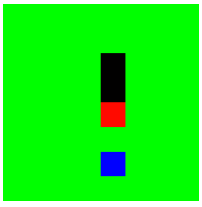
5.6 Trening med Snake

For trening i et snake-miljø brukte jeg Gym-Snake (Grant & Rishaug, 2018). Slangen får -1 i belønning hvis hodet havner utenfor rutenettet, 1 hvis den spiser matbiten, og 0 ellers. Det nevrale nettverket jeg satte opp består av tre konvolusjonelle lag og to lineære lag. Mellom disse brukes ReLU.

5.6.1 Tilstandsrommet

Med Gym-Snake hadde jeg frihet til å velge størrelse på rutenettet. Jeg endret på følgende egenskaper:

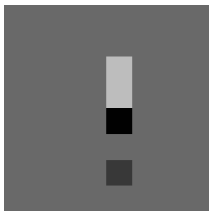
```
env = gym.make("snake-v0")
env.unit_gap = 0
env.unit_size = 1
env.grid_size = [8, 8]
```



Figur 5: Tilstand fra snake med koden til venstre

Jeg ønsket å fjerne fargene som er til stede i figur 5, og brukte da metoden under. Matbiten og bakgrunnen har samme lystyrke, derfor endret jeg fargen på matbiten før jeg konverterte til svart-hvitt. På denne måten hadde jeg et rutenett med 64 piksler, uten farge.

```
1 def downsample(state):
2         state = np.array(state, copy=True)
3         state[state == np.array([0, 0, 255])] = 200 # Set food
          ↪  color
4         downsampled = np.mean(state, axis=2, dtype=np.int16) #
          ↪  To grayscale
5         downsampled[downsampled == 88] = 255 # Set
          ↪  snake-head-color
6         return downsampled
```



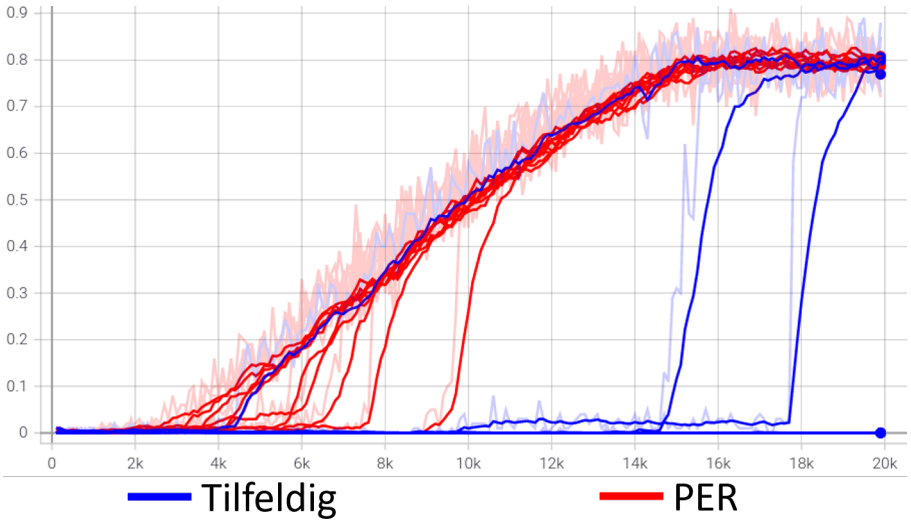
Figur 6: Tilstand i svart-hvitt

6 Resultater

6.1 CliffWalking1D

I et av eksperimentene jeg gjorde, kjørte jeg 20 runder med CliffWalking1D, halvparten med PER. I dette tilfellet satte jeg også prioriteringskonstanten α fra likning 2 lik 1 i motsetning til de andre miljøene der jeg brukte verdien 0.6 som anbefalt i artikkelen om PER (Schaul et al., 2016, s. 6). Dette gjorde jeg for å oppnå mest mulig prioritering. Agentene hadde $n = 9$ noder å komme seg gjennom. Dette ga en sannsynlighet på $\frac{1}{2^9} = \frac{1}{512}$ for suksess med tilfeldige handlinger. Eksperimentet tok omtrent 1.5 timer. Hyperparameterne er listet opp til høyre. Jeg valgte $\gamma = 0.8888$ siden i artikkelen om PER (Schaul et al., 2016, s. 13) foreslo de $\gamma = 1 - \frac{1}{n} = 1 - \frac{1}{9} = 0.\overline{8}$. Resten av hyperparameterne kom jeg fram til med prøving og feiling.

Hyperparametre	
γ	0.8888
α	0.002
MAX_MEMORY_SIZE	200000
EPISODES	20000
BATCH_SIZE	8
REPLACE_AFTER	100

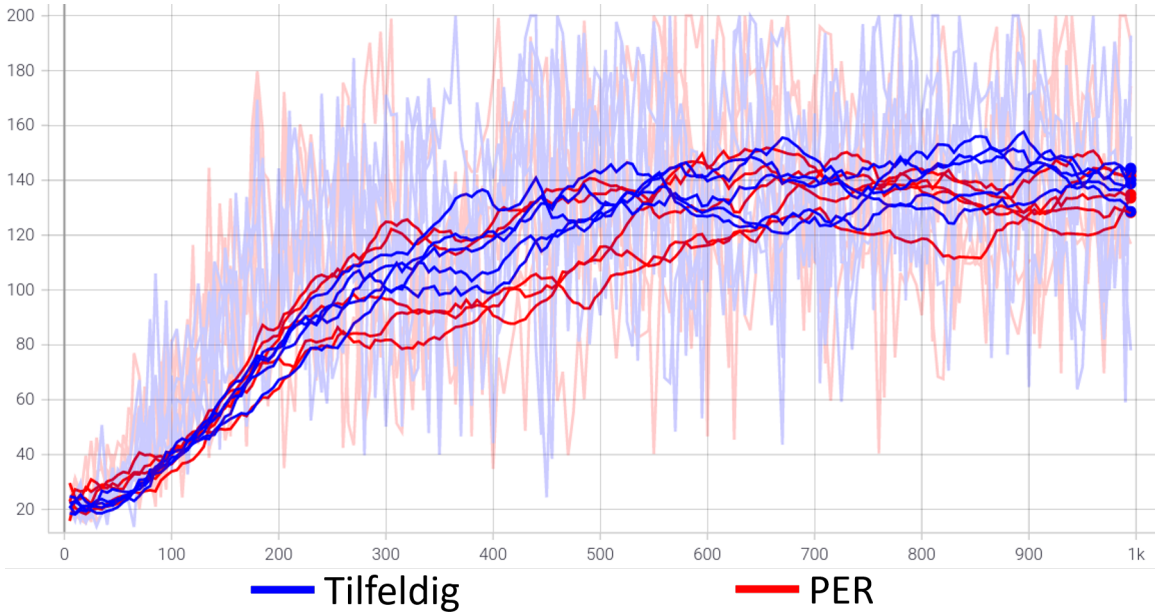


Figur 7: Gjennomsnittlig belønning fra de siste 100 episodene. 10 kjøringer med PER, 10 uten. Glatting=0.84

6.2 CartPole

Som tidligere nevnt, forsøkte jeg å gjenskape resultatene fra artikkelen som hevdet forbedringer (Kumar, 2020). Jeg forsøkte da å bruke de samme hyperparameterne. Disse er gitt i tabellen til høyre. Kumar brukte soft-update av målnettet, mens jeg brukte hard-update. Da måtte jeg velge en verdi for `REPLACE_AFTER`. Siden han hadde $\tau = 0.1$, tenkte jeg 10 ville gi meg mest mulig like resultater. Jeg merket senere at på noen av kjøringene hans brukte han hard-update. I disse tilfellene oppdaterte han målnettet etter hver episode. Uansett tenker jeg at resultatet jeg fikk er mulig å studere. Eksperimentet tok omtrent 1 time.

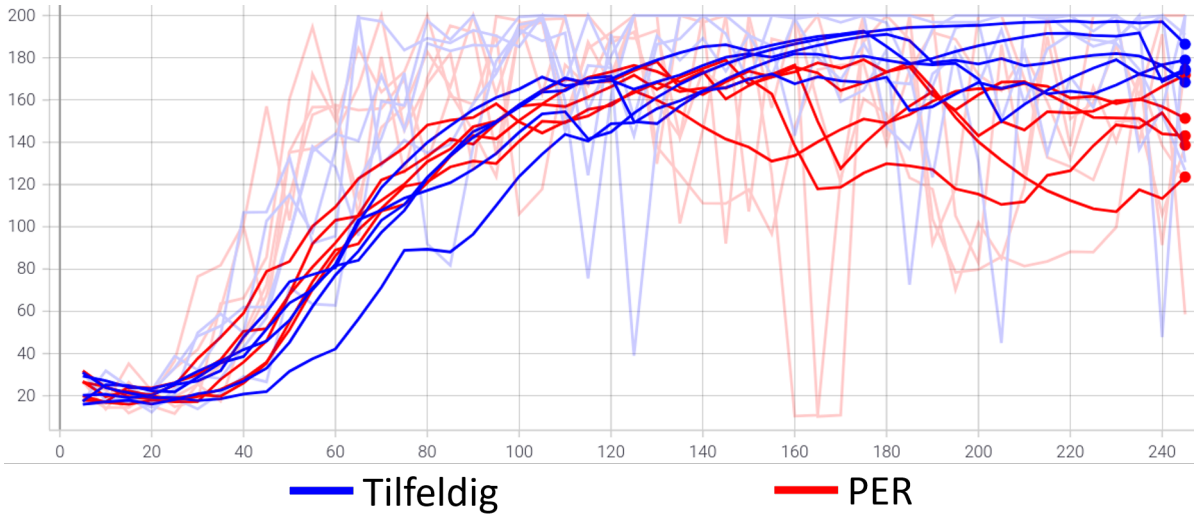
Hyperparametre	
γ	0.9
α	0.001
MAX_MEMORY_SIZE	2000
EPISODES	1000
BATCH_SIZE	24
REPLACE_AFTER	10



Figur 8: 5 kjøringer med PER, 5 uten. Gjennomsnittlig belønning fra de siste 5 episodene. Glatting=0.95

Jeg forsøkte også med eget valg av hyperparametere. Med disse ble den gjennomsnittlige belønningen økt, men kjøringene med PER ble ustabile. Eksperimentet tok omtrent en halvtime.

Hyperparametre	
γ	0.97
α	0.001
MAX_MEMORY_SIZE	1000
EPISODES	250
BATCH_SIZE	64
REPLACE_AFTER	50

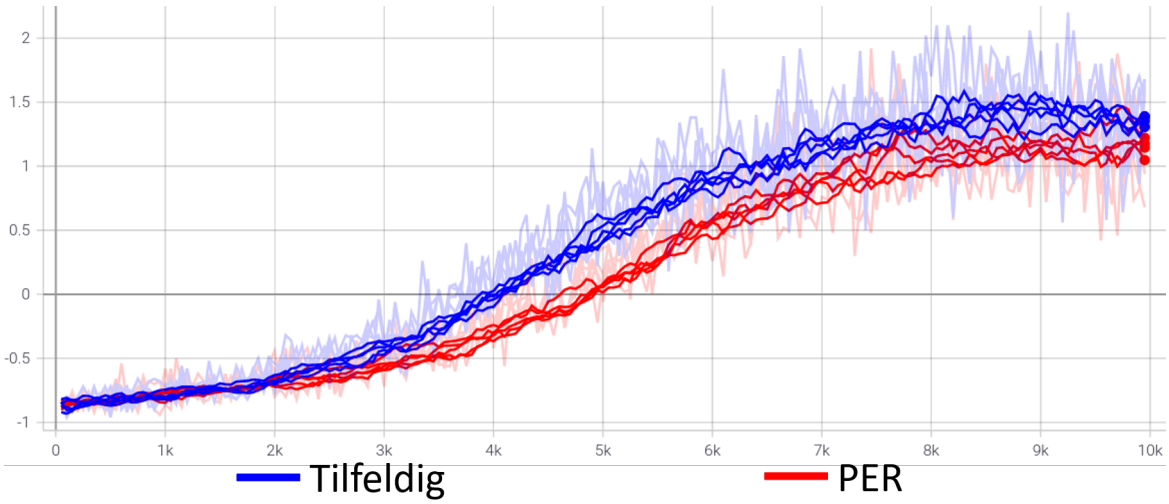


Figur 9: Eget valg av hyperparametere. 5 kjøringer med PER, 5 uten. Gjennomsnittlig belønning fra de siste 5 episodene. Glatting=0.84

6.3 Snake

I dette eksperimentet kjørte jeg 10 runder der halvparten brukte PER. Hyperparameterne er listet opp i tabellen til høyre. I figur 10 under er den gjennomsnittlige belønningen for 50 runder visualisert. På de røde linjene er det brukt PER. Til sammen tok dette omtrent 15 timer.

Hyperparametre	
γ	0.98
α	0.000075
MAX_MEMORY_SIZE	70000
EPISODES	10000
BATCH_SIZE	64
REPLACE_AFTER	100



Figur 10: Gjennomsnittlig belønning fra de siste 50 episodene. 5 kjøringer med PER, 5 uten.

7 Diskusjon

I Gym-Snake, forsøkte jeg ulike måter å ledsage slangen mot matbiten, som for eksempel med en funksjon som beregnet avstanden fra hodet til matbiten. På denne måten kunne jeg straffe slangen for å være langt unna, og belønne den for å være nær. I boken av Sutton & Barto står det derimot:

> In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what we want it to do. For example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent's pieces or aining control of the center of the board. (Sutton & Barto, 2018, s. 54)

Etter å ha lest dette la jeg fra meg tanken om å forsøke å endre belønningssystemet i Snake. Skal man oppnå den mest naturlige løsningen må agenten danne veien dit selv, uten for stor grad av ledsaging. Hvis belønningssystemet blir for komplisert, vil agenten kunne finne en måte å utnytte dette, eller misforstå hva vi ønsker den skal oppnå. Jeg opplevde at slangen utnyttet belønningssystemet basert på distanse ved å gå i sirkler nært matbiten. På denne måten unngikk den at matbiten ville dukke opp langt unna.

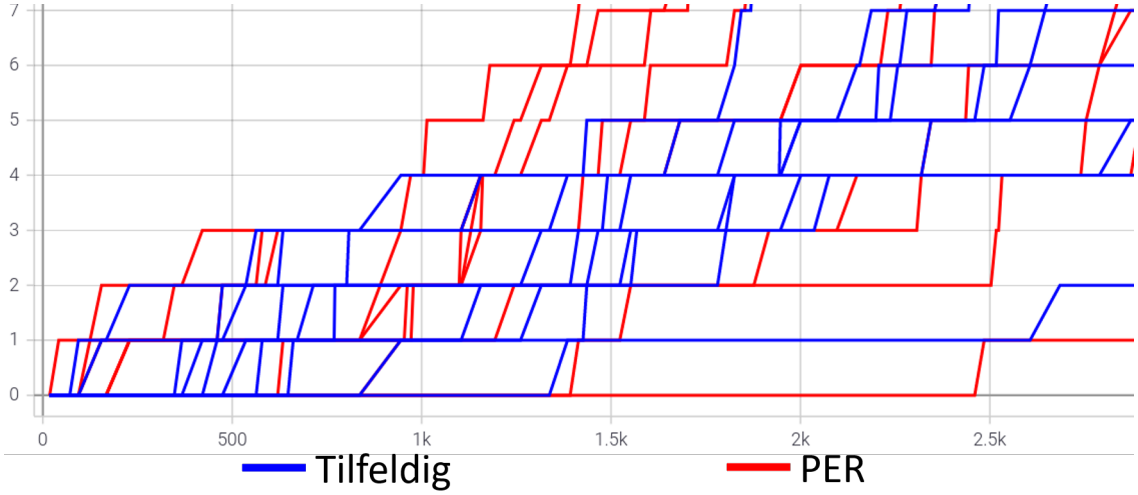
Ideen om PER er interessant da det som mange andre ideer innenfor RL er inspirert av egenskaper vi mennesker har. Vanlig opplevelsesbuffer er hentet fra det at mennesker bruker tidligere opplevelser for å lære hvilke avgjørelser man skal ta hvis en liknende situasjon dukker opp senere. Videre bygger PER på tanken om at mennesker husker noen opplevelser bedre i forhold til andre, gjerne fordi de ga et spesielt inntrykk.

Gjennom prosjektet har jeg fått en bedre oversikt over maskinlæring generelt, og dannet gode kunnskaper innenfor RL. Derfor har jeg noen tanker om hva jeg ville gjort annerledes og undersøkt nærmere. Det eksisterer uendelige kombinasjoner med hyperparametere, og jeg er sikker på at det finnes en kombinasjon som ville gi større forbedringer på PER enn det jeg har opplevd. Det samme gjelder strukturen på nevrale nettverk. Hvis nettverkene er for enkle eller for avanserte vil dette kunne skade ytelsen til agenten vår. (Heaton, 2017). Å undersøke disse faktorene er tidskrevende, men på grunn av den store rollen de spiller, er det et spennende tema. I tilleg til PER, eksisterer det andre ulike forbedringer på Experience Replay, som for eksempel Prioritized Sequence Experience Replay, og Hindsight Experience Replay. Det hadde vært interessant å undersøke disse sammen med PER.

7.1 Effekten av PER på CliffWalking1D

Fra figur 7 er det tydelig at PER gir forbedringer. Alle kjøringene med PER konvergerte innen 20k episoder. Det var derimot tre kjøringer uten PER som konvergerte. To av disse kommer senere fram til løsningen. Bare én av kjøringene uten PER konvergerer like raskt som de med. Kjøringene med PER begynner å konvergere før 10k episoder er gått, og de uten omtrent etter 14k episoder. Dette er omtrent når ϵ har nådd minimumsverdien. Det kan bety at noen av de blå linjene når løsningen sent på grunn av at de har vært uheldige med å utforsket for mye.

For å være sikker på at resultatene mine var troverdige, og ikke tilfeldige, loggførte jeg også en verdi som telte hvor mange runder agenten hadde fullført til sammen. Denne verdien er visualisert i figuren under med de første 2500 episodene.



Figur 11: Det totale antallet episoder med suksess vist i de første 2500 episodene.

I løpet av de første 1000 episodene ligger alle kjøringene omtrent likt med tanke på hvor mange suksesser de har hatt. Etter dette begynner vi å se effekten av PER. De røde linjene begynner å vokse raskere, mens de blå linjene henger litt igjen. Etter omtrent 2500 episoder ligger alle linjene over 0. Dette forteller oss at i løpet av de første episodene fikk alle muligheten til å lagre noen overganger der målet ble nådd.

Resultatene fra figur 7 var forventet. Det var likevel interessant å se hvor stor forskjell PER kan gi i et tilfelle hvor belønninger er veldig sjeldne. I samme figur er det kjøringer uten PER som likevel klarte å komme fram til løsningen. En mulighet er da at hvis det ble kjørt flere episoder, ville flere kunne løse miljøet. Som tidligere nevnt, ble prioriteringskonstanten satt til 1 for PER, for å oppnå sterk prioritering. Med andre ord, er dette eksperimentet satt opp til å være vennlig med PER. Uansett er dette et godt eksempel som demonstrerer hvor stor effekt PER kan ha.

7.2 Effekten av PER på CartPole

I forsøket å bruke Kumars hyperparametre (Kumar, 2020, s. 4, 6-7) oppnådde jeg ikke noen forbedringer ved bruk av PER, som illustrert i figur 8. Jeg fikk heller ikke muligheten til å bruke akkurat de samme hyperparameterne som Kumar brukte.

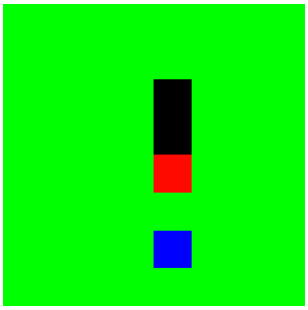
Fra eksperimentet med eget valg av hyperparametre (figur 9) ble den gjennomsnittlige belønningen økt, men samtidig oppsto noen endringer på kjøringene med PER som vi ikke så i figur 8. I de første 80 episodene leder PER marginalt. Halvveis gjennom episodene begynner den derimot å falle av.

I det første tilfellet ser det ikke ut til at PER har noen effekt. I det andre tilfellet gir det forbedringer i starten, men skaper problemer mot slutten. Den mest fornuftige forklaring på det første tilfellet, hvor jeg ønsket å oppnå Kumars resultater, er forskjellen i bruk av hyperparameterne. Likevel tenker jeg at effekten av PER burde vært synlig, enten om det var positivt eller negativt. Årsaken til resultatene kan være at CartPole-v0 er et for enkelt miljø, hvor agenten ikke ser særlig stor forskjell på tilstander, og da at bruken av Importance Sampling gir en negativ effekt.

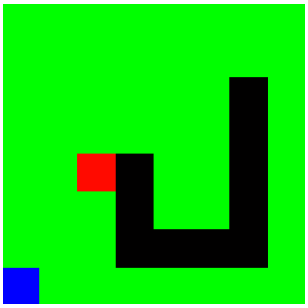
7.3 Effekten av PER på Snake

I figur 10 ser vi at de røde linjene faller under de blå ettersom treningen foregår. Ut fra mine eksperimenter gir altså ikke PER noen god effekt på Snake. Fram til omtrent 2000 episoder ligger begge variantene likt, men etter det deler de seg opp. En mulig årsak til splittelsen kan være verdien til β som brukes til å regne ut viktighetsvektoren (likning 3). β er konstant lik 0.4, fram til 23% av episodene er gått, da den begynner å stige mot 1. I figur 10 starter splittelsen ved det samme tidspunktet. Det hadde vært interessant å undersøke om dette faktisk er årsaken, ved for eksempel å endre hvordan β regnes ut og se om splittelsen endrer seg.

Hvorfor gir ikke PER forbedringer på Snake? I starten av prosjektet anså jeg Snake som et godt eksempel der PER kunne skinne. Snake inneholder en stor mengde tilstander, hvor posisjonen på matbiten, slangens tidligere bevegelse og dens lengde kan være forskjellige. Min første tanke var at agenten ville ha godt av å trene på tilstander langt ute i spillet.



Figur 12: Vanlig tilstand



Figur 13: Sjelden tilstand

Agenten møter ofte tilstander som vist i figur 12. Når den trener på overgangen tilknyttet en slik tilstand vil det gi liten feil (TD), som gir overgangen lav prioritet. Med andre ord, så har agenten blitt vant til slike tilstander. Møter den derimot en tilstand som i figur 13, vil feilen gi stort utslag da den ikke har møtt en slik tilstand tidligere. Da vil agenten sørge for at denne overgangen skal trenes på senere. Spørsmålet er da om agenten kommer til å møte på denne tilstanden på nytt iløpet av samme kjøring. Hvis ikke, har den da sløst vekk ressurser ved å trene på denne?

Det finnes tiltak for at PER skal gi forbedringer. Jo flere episoder vi kjører, jo høyere sannsynlighet er det for at hver mulige tilstand forekommer. Øker man antallet episoder vil agenten kunne havne i en sjelden tilstand den har vært i før. Når agenten forstår sjeldne tilstander vil den møte nye, sjeldnere tilstander, helt til den fullfører spillet. Jeg mener at gitt nok antall episoder, vil nytten av PER begynne å vise seg. En annen observasjon som peker på nødvendigheten med flere episoder er at agenten ofte havnet i lokale maksimum, der den gikk i sirkel for å unngå å treffe kantene, og ikke spiste matbiten. Dette observerte jeg da jeg testet noen av de opptrente nevrale nettverkene.

Et alternativ til PER er Dual Experience Replay, som foreslått i en annen artikkel hvor Snake brukes som miljø. (Wei et al., 2018). Ideen er å bruke to opplevelsesbuffere, et for overganger som gir høy belønning, og et for overganger som forekommer oftere. En interessant problemstilling kunne vært å studere forholdet mellom en slik fremgangsmåte og PER.

8 Konklusjon

Ut fra resultatene er det åpenbart at effekten av et prioritert opplevelsesbuffer kan være både positivt og negativt. Det er mange faktorer som kan spille inn i ytelsen av PER, som for eksempel hvordan miljøet er bygd opp, og hvor mange episoder man kjører og verdiene på hyperparameterne.

I CliffWalking1D ga PER betydelige forbedringer. Miljøet er satt opp for å demonstrere et tilfelle der det egner seg å prioritere enkelte overganger. I CartPole-v0 var det et tilfelle hvor PER ikke ga noen endring, og et annet hvor det i starten ga forbedringer, men etter hvert førte til mindre belønninger. Her er det mulig at ved riktig valg av hyperparametere, vil man kunne se forbedringer. I det siste miljøet, Gym-Snake, tydet resultatene på at PER hindrer agenten i å yte godt. Dette ser derimot ut til å komme av begrensninger ved valg av antall episoder. Øker man dette, vil man kunne se forbedringer siden agenten vil oppleve sjeldne tilstander oftere.

PER gir nok best effekt på miljøer der hver episode består av de samme tilstandene, som CliffWalker1D. Da er det garantert at agenten har nytte av å lære av tilstander den ikke har vært i før.

9 Referanser

Balsys, R. (2019). *Reinforcement learning tutorial.* Hentet fra `https://pylessons.com/CartPole-PER/`

Fard, M.M. & Pineau, J. (2011). Non-deterministic policies in markovian decision processes. Hentet fra `https://arxiv.org/ftp/arxiv/papers/1401/1401.3871.pdf`

Grant, S. & Rishaug, J. (2018). *Gym-snake.* GitHub. Hentet fra `https://github.com/grantsrb/Gym-Snake`

Heaton, J. (2017). *The number of hidden layers.* Hentet fra `https://www.heatonresearch.com/2017/06/01/hidden-layers.html`

Hohenwarter, M., Borcherds, M., Ancsin, G., Bencze, B., Blossier, M., Éliás, J., ... Tomaschko, M. (2020, november). *GeoGebra 5.0.613.0.* (`http://www.geogebra.org`)

Hunter, J.D. (2007). Matplotlib: A 2d graphics environment. *Computing in science & engineering*, *9*(3), 90–95.

Kumar, S. (2020). *Balancing a cartpole system with reinforcement learning – a tutorial.* Hentet fra `https://arxiv.org/abs/2006.04938v2`

Q-learning. (2020). *Q-learning — Wikipedia, the free encyclopedia.* Hentet fra `https://en.wikipedia.org/wiki/Q-learning` ([Online; accessed 17-November-2020])

Ramesh. (2019). *Introduction to sum tree.* Hentet fra `https://www.fcodelabs.com/2019/03/18/Sum-Tree-Introduction/` ([Online; accessed 19-November-2020])

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2016). Prioritized experience replay. Hentet fra `https://arxiv.org/pdf/1511.05952.pdf`

Sutton, R.S. & Barto, A.G. (2018). *Reinforcement learning: An introduction.* Cambridge, MA: The MIT Press. Hentet fra `http://incompleteideas.net/book/the-book.html`

TheComputerScientist. (2019, February). *Increasing training stability with double dqns.* Hentet fra `https://youtu.be/ILDLT97FsNM`

Wan, T. & Xu, N. (2018). Advances in experience replay. *CoRR*, *abs/1805.05536*. Hentet fra `http://arxiv.org/abs/1805.05536`

Wei, Z., Wang, D., Ming, Z., Tan, A.-H., Miao, C. & Zhou, Y. (2018, 07). Autonomous agents in snake game via deep reinforcement learning.. Hentet fra `https://www.researchgate.net/publication/327638529_Autonomous_Agents_in_Snake_Game_via_Deep_Reinforcement_Learning`

10 Vedlegg

10.1 ReplayBuffer

```python
# Non-prioritized experience replay
class ReplayBuffer:
    def __init__(self, cap):
        self.buffer = deque(maxlen=cap) # Using deque from collections library
        self.cap = cap

    # Append transition to buffer
    def append(self, transition):
        self.buffer.append(transition)

    # Fetches a batch of transitions from the buffer
    def get_samples(self, batch_size, device):
        if batch_size > len(self.buffer):
            batch_size = len(self.buffer)

        # Sample random batch
        samples = np.array(random.choices(self.buffer, k=batch_size), dtype=object)

        # Create tensors of transitions
        states = torch.tensor(samples[:, 0].tolist(), dtype=torch.float32).to(device)
        actions = torch.tensor(samples[:, 1].tolist(), dtype=torch.long).to(device)
        rewards = torch.tensor(samples[:, 2].tolist(), dtype=torch.float32).to(device)
        next_states = torch.tensor(samples[:, 3].tolist(), dtype=torch.float32).to(device)
        dones = torch.tensor(samples[:, 4].tolist(), dtype=torch.float32).to(device)
        return  states, actions, rewards, next_states, dones
```

10.2 SumTree

```python
class SumTree(object):
    pointer = 0 # Position of next element in row of leaf nodes
    def __init__(self, cap):
        self.cap = cap
        self.tree = np.zeros(2 * cap - 1) # In total 2 * cap - 1 nodes
        self.data = np.zeros(cap, dtype=object) # Separate array for storing transitions
        self.entries = 0 # Count for stored transitions

    # Add new transition and priority to tree
    def add(self, priority, data):
        index = self.pointer + self.cap - 1 # Position of child node
        self.data[self.pointer] = data # Store the transition
        self.update(index, priority) # Update sums

        # Reset pointer back to start to replace elements when buffer is full
        self.pointer += 1
        if self.pointer >= self.cap:
            self.pointer = 0

        if self.entries < self.cap:
            self.entries += 1

    # Update parent node's values
    def update(self, index, priority):
        difference = priority - self.tree[index]
        self.tree[index] = priority # Set the new priority

        # Add the difference to the parent nodes
        while index != 0:
            index = (index - 1) // 2
            self.tree[index] += difference

    # Retrieve the node that has the input value in its interval
    # This process is described in the theory section
    def get_leaf(self, val):
        parent_index = 0
        while True:
            left_index = 2 * parent_index +1
            right_index = left_index + 1
            if left_index >= len(self.tree):
                leaf_index = parent_index
                break
            else:
                if val <= self.tree[left_index]:
                    parent_index = left_index
                else:
                    val -= self.tree[left_index]
                    parent_index = right_index
        data_index = leaf_index - self.cap + 1
        return leaf_index, self.tree[leaf_index], self.data[data_index]
```

```
51
52      # Get the sum of all priorities
53      def get_total_priority(self):
54          return self.tree[0] # Value of root node
```

10.3 PrioritizedReplayBuffer

```python
# Prioritized experience replay
class PrioritizedReplayBuffer:
    def __init__(self, cap, a=0.6, epsilon = 0.1):
        self.cap = cap
        self.a = a # Hyperparameter for degree of prioritization
        self.epsilon = epsilon # Bias
        self.tree = SumTree(cap) # Sumtree for storing transitions and priorities

    # Add transition to buffer
    def append(self, transition):
        max_priority = np.max(self.tree.tree[-self.cap:]) + abs(transition[2]) # + abs(reward)
        if max_priority == 0: # To avoid transitions with zero priority
            max_priority = 1
        self.tree.add(max_priority, transition)

    # Update priority after training and calculating error of transition
    def update_priorities(self, indices, errors):
        for i, e in zip(indices, errors):
            priority = self.error_to_priority(e)
            self.tree.update(i, priority)

    # Convert error to priority
    def error_to_priority(self, error):
        return (np.abs(error) + self.epsilon) ** self.a

    # Calculate importance weights
    def get_importance(self, probabilites, b):
        importance = np.power(1/(self.tree.entries * probabilites), b)
        normalized = importance/max(importance) # Normalizing to not enlarge gradients
        return normalized

    # Fetches a batch of transitions from the buffer
    def get_samples(self, batch_size, b, device):
        if batch_size > self.tree.entries:
            batch_size = self.tree.entries

        indices = []
        priorities = []
        batch = []

        # Divide row of leaf nodes into segments
        segment_length = self.tree.get_total_priority() / batch_size

        # Fetch random transitions from each segment
        for i in range(batch_size):
            segment_start = segment_length * i
            segment_end = segment_length * (i + 1)

            # Preventing rare case of sampling empty node
            retries = 0
```

```
51              while True:
52                  s = random.uniform(segment_start, segment_end) # Get random number in segment
53                  index, priority, transition = self.tree.get_leaf(s)
54                  if priority != 0:
55                      break
56                  retries += 1
57                  # Discard if no non-empty leaf found after 5 attempts
58                  if retries == 5:
59                      print("Stopped at 5 retries")
60                      break
61              if retries != 5: # Store non-empty element in batch
62                  indices.append(index)
63                  priorities.append(priority)
64                  batch.append(transition)
65
66          probabilities = priorities / self.tree.get_total_priority() # Calculate probabilities
67          importance =  self.get_importance(probabilities, b) # Get importance weights
68
69          samples = np.array(batch, dtype=object)
70
71          # Create tensors of transitions
72          states = torch.tensor(samples[:, 0].tolist(), dtype=torch.float32).to(device)
73          actions = torch.tensor(samples[:, 1].tolist(), dtype=torch.long).to(device)
74          rewards = torch.tensor(samples[:, 2].tolist(), dtype=torch.float32).to(device)
75          next_states = torch.tensor(samples[:, 3].tolist(), dtype=torch.float32).to(device)
76          dones = torch.tensor(samples[:, 4].tolist(), dtype=torch.float32).to(device)
77          return  states, actions, rewards, next_states, dones, np.array(indices), importance
```

10.4 Agent

```python
class Agent(object):
    def __init__(self, Net, input_size, output_size, gamma, alpha, max_memory_size, num_episodes, replace_after, use_per, eps_min=0.05, per_a = 0.6):
        self.gamma = gamma
        self.eps_min = eps_min
        self.action_size = output_size
        self.action_space = range(output_size)
        self.learn_step_counter = 0
        self.replace_after = replace_after
        self.Q_loc = Net(input_size, output_size, alpha)
        self.Q_tar = Net(input_size, output_size, alpha)
        self.num_episodes = num_episodes
        self.use_per = use_per

        # Specify type of buffer
        if use_per:
            self.PER = PrioritizedReplayBuffer(max_memory_size, a=per_a)
        else:
            self.buffer = ReplayBuffer(max_memory_size)
        print("Using PER" if use_per else "Not using PER")

    # Return epsilon, used exploring and importance sampling (IS)
    def get_eps(self, episode):
        # Arbitrary function for calculating epsilon
        return max(self.eps_min, (1-4/(self.num_episodes))**episode)

    # Choose an action
    def choose_action(self, state, episode, train):
        action = None
        if train and np.random.random() < self.get_eps(episode): # train = exploring
            action = np.random.choice(self.action_space) # Choose random action for explring
        else:
            state = torch.tensor([state], dtype=torch.float32).to(self.Q_loc.device)
            actions = self.Q_loc.forward(state) # Run the state through the local Q-Network
            action = torch.argmax(actions).item() # Get the index corresponding to the largest Q-value
        return int(action)

    # Replace the target network's parameters with the one's from the local network
    def replace_network(self):
        self.Q_tar.load_state_dict(self.Q_loc.state_dict())

    # Stores a transition in the used buffer
    def store_transition(self, state, action, reward, next_state, done):
        # Specify type of buffer to store transition in
        if self.use_per:
            self.PER.append([state, action, reward, next_state, done])
        else:
            self.buffer.append([state, action, reward, next_state, done])

```

```
49      # Replay remembered transitions
50      def learn(self, batch_size, episode, use_DDQN=True):
51          self.Q_loc.optimizer.zero_grad()
52          if self.learn_step_counter % self.replace_after == 0: # Replace target network every
            ↪  fixed number of learn-iterations
53              self.replace_network()
54
55          # Sample from the used buffer
56          if self.use_per:
57              # Beta goes from 0.4 to 1 throughout training
58              states, actions, rewards, next_states, dones, indices, importance =
                ↪  self.PER.get_samples(batch_size, max(0.4, min(1, 1.05-self.get_eps(episode))),
                ↪  self.Q_loc.device)
59          else:
60              states, actions, rewards, next_states, dones = self.buffer.get_samples(batch_size,
                ↪  self.Q_loc.device)
61
62          actions = actions.unsqueeze(1) # to shape(batch_size, 1)
63          rewards = rewards.unsqueeze(1) # to shape(batch_size, 1)
64          dones = dones.unsqueeze(1) # to shape(batch_size, 1)
65
66          # Calculate predicted Q-values from states and gather the Q-value for each
            ↪  corresponding to the action taken
67          Qpred = self.Q_loc.forward(states).gather(1, actions.long()).to(self.Q_loc.device)
68
69          with torch.no_grad():
70              if use_DDQN:
71                  next_Qpred = self.Q_loc.forward(next_states).to(self.Q_loc.device) # Get
                    ↪  Q-values for next states
72                  next_actions = torch.argmax(next_Qpred, dim=1) # Get the actions with the
                    ↪  highest Q-values
73                  target = self.Q_tar.forward(next_states).to(self.Q_loc.device) # Calculate
                    ↪  target Q-values from next states
74                  target = target.gather(1, next_actions.unsqueeze(1)) # Get Q-values
75                  target = rewards + (1 - dones) * self.gamma * target # Bellman equation
76              else:
77                  target = self.Q_tar.forward(next_states) # Get Q-values for next states
78                  target = rewards + (1 - dones) * self.gamma * torch.max(target, dim=1,
                    ↪  keepdim=True)[0] # Bellman equation
79
80          loss = None
81          if self.use_per:
82              error = Qpred - target
83              errors_np = (torch.abs(error)).cpu().data.numpy()
84              self.PER.update_priorities(indices, errors_np) # Update priorities for transitions
                ↪  in batch
85              # Calculate loss with importance sampling (IS)
86              importance_tensor = torch.FloatTensor(importance).to(self.Q_loc.device)
87              loss = torch.mean((importance_tensor * torch.square(error)))
88          else:
89              loss = self.Q_loc.loss(Qpred, target) # Normal mean-squared-error
90
91          loss.backward() # Calculate gradients
```

```
92          self.Q_loc.optimizer.step() # Update network
93          self.learn_step_counter += 1
```