Deep Q-learning på Atari Breakout

Martin Johannes Nilsen

Denne rapporten inneholder skildringer, erfaringer og arbeid tilknyttet fagområdet Reinforcement Learning (RL). OpenAI, et av de største research-firmaene i verden når det kommer til AI [9], har utviklet et verktøy for å forenkle testing og trening av RL-algoritmer. Dette verktøysettet, kalt OpenAI Gym, er det som i denne rapporten tas i bruk for å teste algoritmene og modellene på Atari Breakout, et retro arkadespill originalt fra 1976 [11].

1 Introduksjon

Spillet Atari Breakout går ut på å styre en bevegende flate, og sørge for at ballen ikke går ut av skjermen. En sanker poeng ved å fjerne blokker radvis plassert på toppen av skjermen. Spilleren har fem liv, og etterhvert som du tar blokker lengre opp i radene, vil ballens hastighet øke. I kildekoden til dette prosjektet er det lagt ved en fil som tar i bruk OpenAIs eget bibliotek `Play`, der du selv kan teste ut spillet.

I dette prosjektet har det blitt tatt utgangspunkt i en publikasjon i fra Google DeepMind [1] utgitt i 2013. Reinforcement learning-systemet består av en agent som interagerer med et gitt miljø, der det er opp til agenten å klare få så god score som mulig ved å utføre de rette handlingene. Disse handlingene har den oversikt over i en tabell, som gitt en tilstand har en oversikt over hvilke handlinger agenten burde velge med utgangspunkt i en utregnet verdi. Disse verdiene, kalt rewards, er regnet ut basert på tidligere erfaring; hvis det går dårlig får den en lavere reward, mens en god handling belønnes ved en høyere verdi. Denne tabellen kan sees på som et slags guide, og blir oppdatert med bruk av et konvolusjonelt nettverk. Dette utgjør prinsippene bak en verdibasert DQN[1] algoritme.

Figur 1. Illustrasjon av spillet Atari Breakout i OpenAI Gym.

Etter å ha erfart at det er et drøss av konstanter og parametere som påvirker læringsprosessen med ulikt utslag, ble dette et fokusområde i denne rapporten. Prinsipper som erfaringsbuffer[2], bruk av tidligere erfaringer kontra utforskning av nye muligheter[3], samt effekten av ulike hyperparametere og optimaliseringsfunksjoner har blitt testet i praksis. Det blir videre diskutert hvilken effekt disse har på systemet, med et mål om å oppnå en høyest mulig score i spillet. Problemstillingen blir som følger; *hvilke parametere gir størst utslag på læringsprosessen til en DQN-basert RL-agent i et arkadespill?*

1. Deep Q-Network
2. Experience replay/replay buffer
3. Exploration vs. Exploitation

2 TIDLIGERE RELEVANT ARBEID

Deep learning på Atari Breakout ble først presentert i publikasjonen *Playing Atari with Deep Reinforcement Learning* [1] av DeepMind i 2013. Modellen de brukte var et konvolusjonelt nevralt nettverk, trent med en variant av Q-learning, med piksler som input og en verdifunksjon som estimerer fremtidige resultater som output. De testet ut metoden på syv Atari 2600 spill, og fant ut at den utkonkurrerte alle tidligere forsøk på seks av spillene, og en menneskelig ekspert på tre av dem. Poengsummen for DQN var følgende:

Tabell 1
Resultater for DQN på Atari Breakout [1].

Method	Avg. Score	Best Score
DQN	168	225

I etterkant av denne publikasjonen har DeepMind fortsatt arbeidet med å oppnå best mulig poengsum i en hel rekke Atari spill. I rapporten *Human-level control through deep reinforcement learning* (2015) [2] kom de ikke med noen nye teknologiske gjennombrudd, men de viser hva som skjer når teknikkene blir brukt i mye større skala. De klarte nå å oppnå 75% av poengsummen til en profesjonell tester i 29 av 49 arkadespill den prøvde ut, noe som gjenspeiles i navnet på utgivelsen. Demis Hassabis, en av DeepMinds grunnleggere, uttalte i etterkant av utgivelsen at de nå hadde brukt mye større nevrale nettverk, kommet opp med bedre treningsregimer og trent over lengre perioder [12]. I 2013 beskrev de resultatene som en tidlig smakebit på hva en kunne få til, men det er først nå de kunne se hva disse nettverkene var kapable til å oppnå i arkadespill.

I 2016 [3] introduserte de en alternativ måte å stabilisere det nevrale nettverket på. Istedenfor å ta i bruk et erfaringsbuffer, brukte de en serie av parallelle agenter som asynkront trente i miljøet, også kjent som Actor Critics. Dette innebar å kombinere en verdibasert og policybasert algoritme, der man kunne sette to agenter opp i mot hverandre; en *actor* som bestemmer hvilken handling den skal ta, i mens en *critic* forteller hvor god denne handlingen er og hva den kan gjøre annerledes. De konkurrerer mot og lærer av hverandre, og åpner opp for mer effektiv læring og utforsking av hva som gir bedre poengsum.

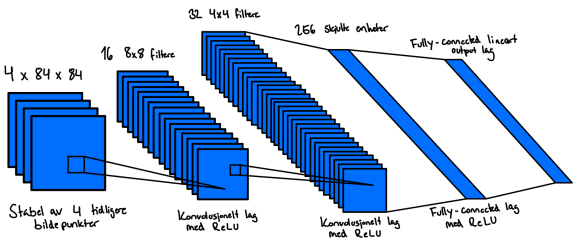
Det er ikke bare DeepMind som har utgitt relevant arbeid når det kommer til deep reinforcment learning. M. G. Bellemare introduserer i sin doktorgradsavhandling (2013) [6] at Atari videospill kunne bli brukt som benchmark-test [16] for RL-algoritmer. Han beskriver arkadespillene som utfordrende, men allikevel simple nok til at vi kan håpe på å oppnå målbar fremgang ettersom vi forsøker løse dem. D. Silver har laget en veiledning [8] der han går i dybden på viktigere konsepter innen RL, og det finnes et drøss av artikler som omhandler fagområdet. Dette prosjektet bygger i stor grad på tidligere relevant arbeid, en kan si at undertegnede *står på skuldrene til kjemper* (Isaac Newton, 1676) i arbeidet med å implementere disse konseptene i praksis.

3 METODE

3.1 Tilnærminger til reinforcement learning

En tilnærming til reinforcement learning består av en eller flere av disse komponentene; en policy, verdifunksjon og/eller modell. En policy er agentens atferd, og en kan se på det som en kobling mellom tilstand og handling [8]. Dersom en har en policybasert tilnærming er det den optimale policyfunksjonen man prøver finne, den som gir høyest poengsum. En verdifunksjon er til forskjell en prediksjon av fremtidig poengsum. Der fokuserer den heller på hvordan den kan klare oppnå den høyeste verdien mulig. Med en verdibasert tilnærming prøver man estimere den optimale verdifunksjonen, den som gir høyest poengsum uansett policy. En modell er oppnådd ved erfaring. Dersom man har en modellbasert tilnærming prøver man bygge en modell av miljøet, og prøver løse det ved å ta i bruk denne modellen.

3.2 Deep Q-Network



Figur 2. Illustrasjon av implementert CNN.

Deep Q-Network (DQN) er en verdibasert tilnærming til reinforcement learning. Vi representerer en verdifunksjon med en Q-tabell, og oppdaterer denne med bruk av et konvolusjonelt nettverk. En stabel med 4 tidligere observasjoner, i form av et bilde med 84 x 84 bildepunkter, er stablet oppå hverandre og kjørt igjennom dette konvolusjonelle nettverket. Resultatet er en oppdatert Q-tabell for handlingene mulig å utføre gitt en tilstand.

Tabell 2
Lagdelingen i det konvolusjonelle nevrale nettverket. Siste output avhenger av antall handlinger i `action_space`.

Layer	Input	Activation	Output
Conv2d	4x84x84	ReLU	16x20x20
Conv2d	16x20x20	ReLU	32x9x9
Fully-connected	32x9x9	ReLU	256
Fully-connected	256	None	3 el. 4

3.3 Hyperparametere

En hyperparameter er en konfigurasjonskonstant som kan direkte påvirke hvor godt en maskinlæringsalgoritme trener og lærer. Disse kan ikke utregnes i koden slik som andre parametere, men må bli definert på forhånd. En vet ikke nødvendigvis hva som er den beste verdien for en hyperparameter for et gitt problem, men en kan bruke verdier brukt i andre problemer som utgangspunkt, og justere verdiene basert på prøving og feiling.

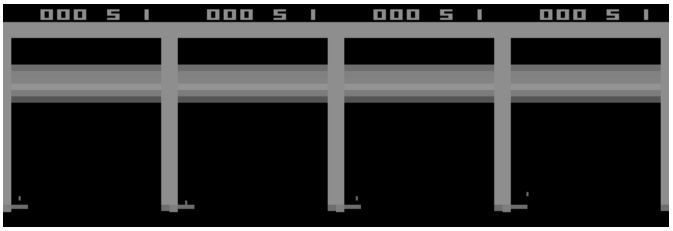
I dette prosjektet har det blitt observert viktigheten av de ulike hyperparameterne, og vi kommer tilbake til dette i senere kapitler. Hyperparametere som læringsrate, avtagningshastigheten på epsilon, minimum epsilon og størrelsen på erfaringsbufferet har vist seg å ha stor effekt. Avslutningsvis har det blitt tatt i bruk følgende hyperparametere, men det er ingen garanti for at det er de beste verdiene å ta i bruk:

Tabell 3
Hyperparametere tatt i bruk i dette prosjektet. `eps_min` avhenger av valgt utforskningsmetode.

Hyperparameter	Verdi
memory_size	1 000 000
min_rb_size	50 000
sample_size	32
lr	0.0001
eps_min	0.05 el. 0.1
eps_decay	0.999999
discount_factor	0.99
env_steps_before_train	16
epochs_before_tgt_model_update	5000
epochs_before_test	1500
episode_max_steps	10 000

3.4 Prosessering av obsevasjoner

Observasjonen vi får fra det originale miljøet er et bilde på formen 210 x 160 piksler, med 3 kanaler for farge (RGB). Problemet med dette er at vi for det første ikke har mulighet til å se ballbanen med kun ett stillbilde, i tillegg til at bildet inneholder en del data vi ikke trenger. Det viser seg at en kan skalere ned bildet til 84 x 84 piksler, og gjøre om bildet til gråtone, uten å miste nødvendig informasjon i bildene. I tillegg ønsker en å stable flere bilder oppå hverandre før man gir det videre som input til nettverket. Vi må altså endre på `observation_space` i miljøet til å være på formen 4x84x84, der 4 representerer antall bilder i en stabel, og ikke kanaler (gråtone har kun én kanal). Slik vil da hver observasjon se ut for agenten vår:



Figur 3. Illustrasjon over 4 påfølgende observasjoner i en stabel, med nyeste observasjon til venstre.

3.5 Experience replay

Samme som med oss mennesker, så trenger også maskinlæringsalgoritmen vår å huske på hva den har gjort tidligere. For hvert steg i miljøet vårt får vi tilbake en tuppel på formen:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

Tuppelen inneholder tilstanden[4] s_t, handlingen[5] a_t, belønningen[6] r_{t+1} gitt ved utført steg som et resultat av tidligere tilstand-handling-par, og den neste tilstanden[7] s_{t+1}. Denne tuppelen gir oss et sammendrag av agentens erfarig ved tid t.

Disse erfaringene blir lagret i et erfaringsbuffer med en gitt størrelse, i kildekoden definert som hyperparameteren `memory_size`. Man tar altså vare på et gitt heltall erfaringer, men hvordan skal man bruke dem? Jo, en henter tilfeldig ut en samling med en gitt mengde erfaringer, og trener nettverket med disse. Det å ta vare på erfaringer i et minne og utnytte disse erfaringene under trening er det vi kaller *experience replay*.

Hvorfor vil vi trene nettverket på tilfeldige partier fra erfaringsminnet, istedenfor å bare gi all erfaringen til nettverket med en gang? Jo, dette er for å ødelegge sammenhengene som kan føre til at nettverket overtrener på disse. Dersom nettverket kun lærte fra påfølgende erfaringer ettersom de kom sekvensielt i fra miljøet, ville prøvene vært sterkt avhengige av hverandre og derfor ført til ineffektiv læring. Det å ta tilfeldige samlinger fra et erfaringsminne bryter ned denne sammenhengen. Antall erfaringer som tas ut tilfeldig ved hvert steg er definert ved hyperparameteren *batch_size*.

Stegene for experience replay kan i pseudokode bli satt opp på følgende vis:

4. state
5. action
6. reward
7. next state

Algoritme 1 Experience replay

1. Fyll replay_buffer med 50 000 erfaringer
2. Initialiser nettverket med tilfeldige vekter
3. For hver episode:
    1. Initialiser første tilstand
    2. For hvert steg:
        1. Velg en handling
            - Exploration vs. exploitation
        2. Utfør handlingen i et miljø
        3. Observer poengsum og neste tilstand
        4. Lagre erfaring i erfaringsbufferet
        5. Hent ut en tilfeldig samling fra erfaringsbufferet
        6. Preprosesser tilstandene
        7. Send disse til nettverket i train_step()
        8. Beregn loss mellom ny Q-tabell og target Q-tabell
        9. Optimaliseringsfunksjonen oppdaterer vektene i nettverket for å minimere loss

3.6 Exploration vs. Exploitation

En viktig del av å lære er å kunne utforske og erfare hva som skjer når ting gjøres. Reinforcement Learning er ingen unntak, og det handler om å finne en balansegang mellom å utforske noe nytt og erfare hva dette fører til, og å basere handlingen sin på noe som har gitt gode resultater tidligere, og bygge videre på dette. En snakker da om *Exploration vs. Exploitation*

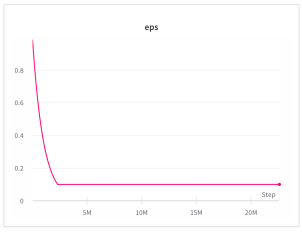
Exploration, eller utforskning på norsk, handler om å utforske nye muligheter istedenfor å gjøre det en har gjort tidligere. Her ønsker man prøve ut en handling en kanskje ellers ikke ville eller har prøvd, og se om den gir et positivt utslag på resultatet eller ikke. En ønsker ha en viss grad av tilfeldighet, og hvordan dette blir implementert er noe som skiller de ulike utforskningsmetodene fra hverandre. Utforskning er en viktig del av all læring, og er noe man ikke må glemme når maskiner skal prøve lære slik vi mennesker gjør.

Exploitation handler om at vi ønsker utnytte tidligere erfaringer til å løse et gitt problem. Her kommer erfaringsbufferet vi tidligere har beskrevet inn i bildet. Det viser seg at det å utnytte tidligere erfaringer er helt essensielt for læring, og er den tilnærmingen vi må ha størsteparten av tiden. Samtidig kan man ikke kun gjøre det som en vet fungerer best, for kanskje er det slik at en enda bedre løsning finnes men ikke er prøvd ut. Mer om dette i resultatkapittelet.

3.6.1 Epsilon greedy

Epsilon greedy er en enkel implementasjon for å løse balansegangen mellom utforskning og utnytting av det den vet fungerer. Epsilon representerer sannsynligheten for å prøve ut noe nytt, og dersom den er *grådig* vil dette innebære at den velger den løsningen som den vet fungerer istedenfor å utforske. For de gangene der den velger utforske utfører den en helt tilfeldig handling, og observerer hva som skjer ved utførelsen av denne.

I dette prosjektet har det blitt inkludert to konstanter `eps_decay` og `eps_min` som definerer hvor sannsynlig det er at en av tilnærmingene blir utført. Innledningsvis starter vi med $epsilon = 1$, altså 100% sannsynlighet for å utforske. Ved å opphøye `eps_decay`, en verdi på 0.999999, med `step_number`, vil den gradvis synke frem til den når `eps_min`. På denne måten sørger vi for at den utforsker en del i starten, før den omsider begynner fokusere mer på erfaringene den har oppgjort underveis. En ønsker ikke å nå $eps\_min = 0$, ettersom det da er usannsynlig at den kommer til å klare lære noe nytt som kan utvide dens lokale maksima.



Figur 4. Epsilonverdiens forløp.

Algoritme 2 Epsilon Greedy

p = random() {Velg et tall i intervallet [0, 1]}

if p $<$ eps
  Gjør en tilfeldig handling
else
  Gjør den beste handlingen basert på tidligere erfaringer

3.6.2 Boltzmann

Boltzmanntilnærmingen baserer seg på en distribusjon av sannsynligheter for elementene i en mengde. Denne Boltzmanndistribusjonen brukes for å finne sannsynligheten for at systemet ville valgt hver av handlingene, basert på forventet belønning, og velger en handling noe tilfeldig basert på disse sannsynlighetene. Dette er også kjent som softmax eller softargmax, der en omgjør en liste med handlinger til sine respektive sannsynligheter, f.eks på formen

$$softmax([0, 1, 2, 3]) = [0.24, 0.09, 0.46, 0.21]$$

Det dette fører med seg, er at en ikke gjør noe helt tilfeldig slik som med Epsilon Greedy. En har da noe tilfeldighet innblandet, som gjør at man kan utforske nye muligheter og potensielt finne en bedre løsning, samtidig som at man ikke tar handlinger som på ingen måte gir mening å ta.

3.7 Bellmann equation

Gitt en tilstand, ønsker vi finne hvilken handling vi kan ta for å få den beste mulige belønningen fra systemet. Til dette trenger vi Bellmannlikningen, som for en verdifunksjon kan skrives på formen

$$v(s) = E[R_{t+1} + \gamma v(S_{t+1})|S_t = s]$$

der vi kan se at verdien v av en tilstand s består av belønningen den får pluss verdien av neste tilstand multiplisert med avslagsfaktoren[8] γ.

8. Discount factor

Vi omformer så denne videre for å uttrykke Q-funksjonen. Optimale Q-verdier bør følge Bellmannlikningen [8]. Denne funksjonen tar inn både tilstand og handling, og kan derav skrives på formen

$$Q^*(s,a) = E_{s'}[r + \gamma \max_{a'} Q(s',a')^*|s,a]$$

Av denne ser vi at en tilstand-handling verdi kan bli dekomponert til den umiddelbare belønningen vi får ved å utføre en handling i en gitt tilstand s og gå over til en annen tilstand s', pluss den nedjusterte verdien av tilstand-handling-verdien av en tilstand s' med hensyn til en handling a' som agenten vil ta fra nå av.

Det er denne likningen som har blitt kombinert med `SmoothL1Loss` fra `PyTorch`, en form for *Huber Loss*, på formen

$$loss(x,y) = \frac{1}{n}\sum_i z_i$$

der z_i er gitt av

$$z_i = \begin{cases} 0.5(x_i - y_i) & \text{hvis } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5 \end{cases}$$

for å fortelle hvor god en handling er gitt en tilstand.

3.8 Valg av miljø

OpenAI Gym tilbyr en hel rekke versjoner av spillet Breakout. For det første, har man versjon 0 og versjon 4, der den første har en sannsynlighet på 25% for å gjenta forrige handling istedenfor ny valgt handling. For det andre, har man en deterministisk versjon av disse to, der denne har en konstant frameskip[9] på 4, og den ikke-deterministiske velger en tilfeldig heltallsverdi i intervallet [2, 5]. Avslutningsvis har man en versjon av disse to uten frameskip, dvs. der denne er satt til 0. I Tabell 4 finner man en oversikt over de tilgjengelige miljøene, og senere vil det bli beskrevet hvilke som ble brukt i dette prosjektet.

9. Antall bilderuter den hopper over

Tabell 4
Versjoner av Breakout gitt i OpenAI Gym

Enviroment	Frameskip	Repeat action
Breakout-v0	[2, 5]	0.25
BreakoutDeterministic-v0	4	0.25
BreakoutNoFrameskip-v0	0	0.25
Breakout-v4	[2, 5]	0
BreakoutDeterministic-v4	4	0
BreakoutNoFrameskip-v4	0	0

3.9 Reduksjon av mulige handlinger

I de opprinnelige miljøene fra OpenAI Gym er det fire handlinger agenten kan velge å ta. Dette innebærer å stå stille (0), starte spillet (1), gå til høyre (2) og gå til venstre (3). Agenten må altså selv starte spillet, før den skal ta i bruk resterende tre handlinger for å fjerne blokker. Denne ekstra handlingen kan sees på som unødvendig, og noe som systemet kan gjøre automatisk, på tross av at en maskinlæingsalgoritme ikke skal ha for store utfordringer med å lære seg å starte spillet.

Det er derfor blitt implementert et eget miljø i dette prosjektet som prøver gjøre akkurat dette. Miljøet `NoFireInActionSpaceEnv` er å finne i `utils.py`, og løser det på følgende måte. For det første må en starte spillet hver gang en tilbakestiller miljøet, så handlingen er implementert i metoden `reset()` i miljløet. For det andre, må man nå holde oversikt over liv som går tapt, ettersom agenten selv ikke kan utføre handlingen. En har oversikt over liv gjennom attributtet `info`, og dersom en mister et liv vil den igjen starte spillet. I tillegg til dette har `action_space.n` blitt redusert til 3 ettersom det kun er tre mulige handlinger og `sample()` har blitt endret til å kun returnere et tall i intervallet [0, 2]. Ønskelig ville en unngått å tildele miljøet nye tallverdier for handlingene, men det ble støtt på problemer som gjorde det enklere å utføre dette istedenfor. Av den grunn er metoden `step()` i miljøet gjort om slik at de nye handlingene agenten og modellen tar i bruk blir: stå stille (0), gå til høyre (1) og gå til venstre (2).

3.10 Wandb

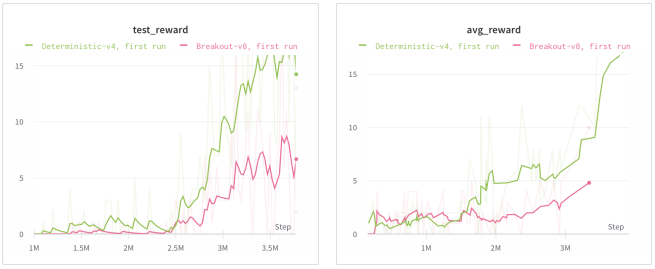
Weights & Biases Dashboard [15] er et verktøy for å overvåke og logge kjøringer, spesielt utformet for maskinlæring. Verktøyet gjør det enklere å kunne se utviklingen av læringsprosessen til algoritmen når den kjører, ettersom man kontinuerlig kan laste opp data til deres sky og få det oversiktlig presentert i et dashboard på nettsiden deres. En kan også velge kjøre maskinlæringen lokalt og synkronisere den med deres skytjeneste etter endt kjøring. Dette verktøyet har vært til stor hjelp når det kommer til å se tendenser i læringen, og få grafer og testvideoer presentert for ulike tidssteg.

4 Resultat

Underveis i prosjektet har det blitt implementert og justert en del på kildekoden for å erfare og observere, med den hensikt å prøve oppnpå en så høy poengsum i spillet som mulig. Videre følger det beskrivelser av resultater, med underkapitler som følger prosjektets progresjon.

4.1 Valg av miljø

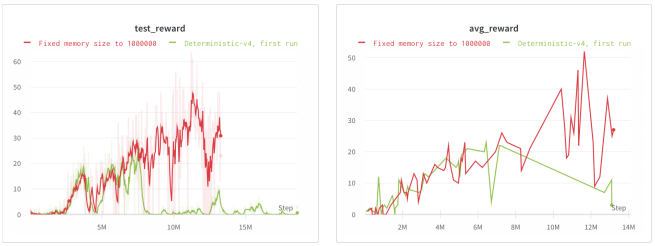
De aller første resultatene ble observert etter å ha kjørt miljøet `Breakout-v0`. Etter å ha lest litt mer om miljøet, ble det oppdaget at det var en hel rekke versjoner med samme form for output, men som hadde litt ulike egenskaper. Et annet miljø som virket spennende å teste var `BreakoutDeterministic-v4`, ettersom denne har en mer deterministisk tilnærming til både repetisjon av handlinger og konstant frameskip. Figur 5 viser forskjellen på agentens prestasjon i disse to miljøene. Den var ikke veldig stor, men en ønsker ha litt mer kontroll over tilfeldighetene som er tilstede. På grunnlag av disse to kjøringene ble den deterministiske versjonen valgt til fordel for den første i senere kjøringer.



Figur 5. Testscore og gjennomsnittsscore på *Breakout-v0* (rød) vs. *BreakoutDeterministic-v4* (grønn).

4.2 Rettelse av erfaringsbufferstørrelse

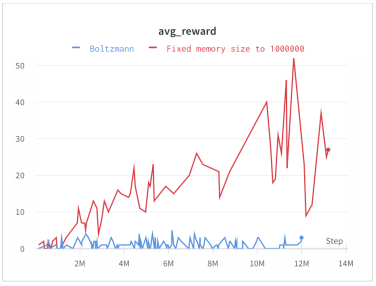
Etter å ha ekstrahert alle hyperparameterne og samlet disse på et sted, ble det oppdaget at størrelsen på bufferet feilaktig var satt til 100 000, en verdi som skulle vært satt til 1 000 000. Dette førte med seg at en ny kjøring måtte utføres med den riktige minnestørrelsen. Resultatene på denne nye kjøringen kan sees i Figur 6.



Figur 6. Testscore og gjennomsnittsscore med minnestørrelse på 100 000 (grønn) vs. 1 000 000 (rød).

4.3 Boltzmann

Videre ble det utført forsøk med en implementasjon av Boltzmannutforskning, der den alltid tok utgangspunkt i tilhørende sannsynlighetsdistribusjon. Dette viste seg å begrense effekten av erfaringsbufferet, og føre med seg en så godt som stagnert læringskurve. Som en kan se i Figur 7, holder gjennomsnitsscoren seg relativt lav hele veien. Selv om de ulike testresultatene kan variere, ser det ikke ut til at denne implementasjonen hadde lært spesielt mye dersom den fortsatte å prøve, og den ble derfor stoppet etter 15t med kjøring.

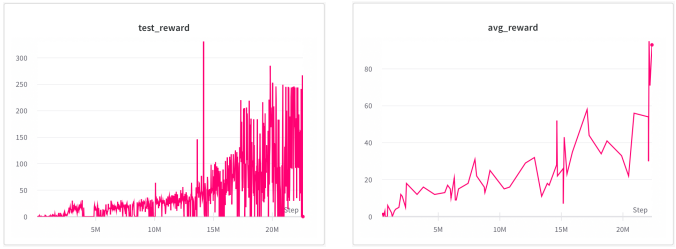


Figur 7. Gjennomsnittsscore for første implementasjon av Boltzmann (blå) vs. Epsilon Greedy (rød).

4.4 Boltzmann med synkende epsilon

Etter resultatene fra forrige kjøring, måtte noe gjøres for å få effekten av erfaringsbufferet tilbake. Noe som viste seg å gi gode resultater, var å innføre `eps_decay` fra Epsilon Greedy. Her starter en med en høy epsilon-verdi, som vil si at den utforsker mye, før den omsider når en verdi på 0.1 og forblir der for resten av kjøringen.

Denne endringen viste seg å gi et godt utslag, noe en kan se på Figur 8. Etter rett i underkant av 15m steg klarte den å oppnå en poengsum på 331 på en tilfeldig test. Samtidig begynte gjennomsnittet å legge seg på rundt 90 etter over 20m steg.

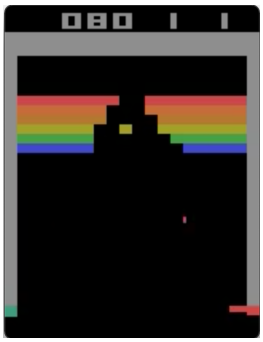


Figur 8. Testscore og gjennomsnittsscore på Boltzmann med synkende epsilon.

4.5 Redusering av mulige handlinger

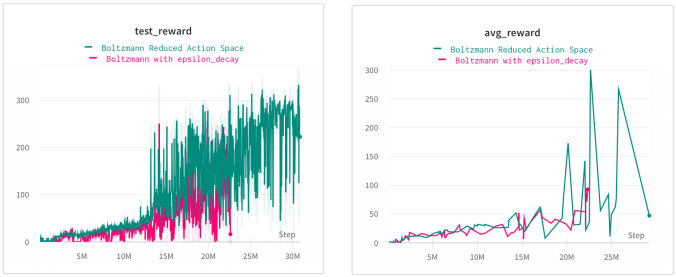
En ny ide som kom opp underveis, som beskrevet i kapittel 3.9, var å redusere antall handlinger agenten skal ha kontroll på. Opprinnelig har den fire handlinger, og en av disse var å starte spillet. Dette ble sett på som en oppgave som miljøet kan ta på seg, selv om det ikke skal være en stor utfordring for en maskinlæringsalgoritme og lære seg begynne et spill. En kan observere i forrige kjøring en situasjon der agenten ble stående fast og vibrere hyppig fra høyre til venstre, noe en vil prøve forhindre ettersom dette ikke fører noen vei.

Etter å ha implementert handlingen med å starte spillet inn i miljøet, ble den samme agenten kjørt på ny. Som en kan se av Figur 9, fokuserer den også her på å lage en tunell, som viser seg være en god fremgangsmåte.



Figur 9. En taktikk som går igjen er å lage en tunell gjennom blokkene for å fange ballen på oversiden.

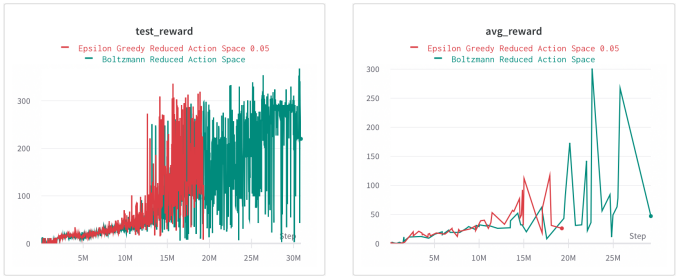
Etter å ha kjørt i 43t hadde den oppnådd enda litt bedre resultater enn tidligere, som en kan se i Figur 10. Den er fortsatt noe ustabil, da et treff kan ha veldig mye å si om den klarer få ballen over blokkene eller ikke, men en kan for eksempel legge merke til at den på et tidspunkt hadde en gjennomsnittsscore på like over 300. Flere stadier under treningen kjører de så og si likt, men der den spiller godt spiller den enda bedre, og en toppscore på 364 ble oppnådd under denne kjøringen.



Figur 10. Testscore og gjennomsnittsscore på Boltzmann i miljø med kun 3 handlinger (grønn) vs. originalt miljø (rosa).

4.6 Epsilon Greedy med nedjustert epsilon

Epsilon Greedy ble igjen tatt opp for å sammenlikne med Boltzmann, men nå med to endringer. For det første ble den nå kjørt i det nye miljøet med et redusert antall handlinger, som potensielt har enda mer å si ettersom utforskningshandlingene for Epsilon Greedy er helt tilfeldige. For det andre ble minimum utforskningsrate gitt en lavere verdi, i lys av tidligere kjøringer. Ettersom Boltzmann regner ut sannsynligheten for hva den skal ta med bakgrunn av modellen, og velger med dette i tankene er tilfeldigheten mindre enn `eps_min`. Det var da logisk å tenke at en kunne oppnå en liknende effekt ved å utforske tilfeldig litt sjeldnere også her.

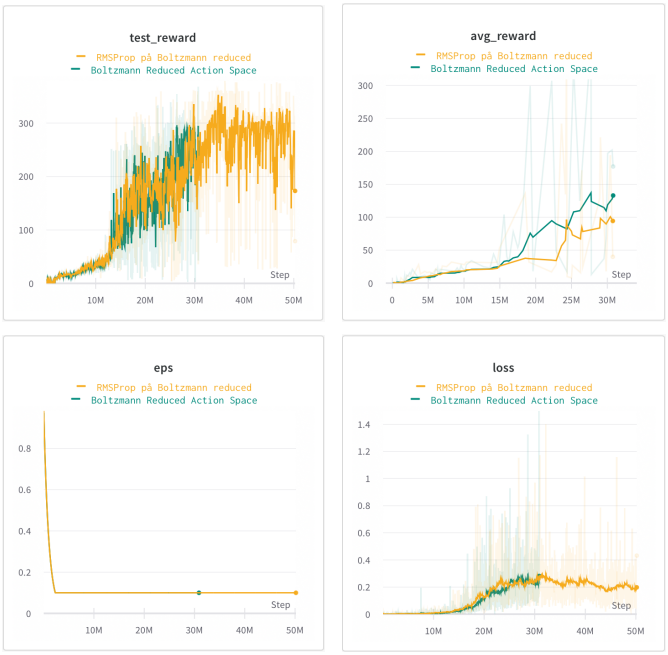


Figur 11. Testscore og gjennomsnittsscore for Epsilon Greedy i miljøet med redusert handlingsrom og nedjustert epsilon (rød) vs. Boltzmann fra forrige kjøring (grønn).

Som en kan observere i Figur 11 har ikke denne kjøringen vært gjennom på langt nær like mange steg som den tidligere kjøringen. Dette på tross av at denne kjøringen stoppet etter 36t og Boltzmann ble stoppet etter 43t. En kan allikvel få en god indikasjon på at det ikke er mye som skiller de nå, men en kan observere at Epsilon Greedy er mer stabil etterhvert som man tar i bruk erfaringsbufferet enda mer.

4.7 Adam vs. RMSProp

Den siste kjøringen i dette prosjektet bestod i å teste om optimaliseringsfunksjonen har stor påvirkning på resultatet eller ikke. Når det kom til hvilken utforskningsmetode som var ønskelig å teste denne endringen på, falt valget på Boltzmann basert på to faktorer. For det første, ble Epsilon Greedykjøringen terminert etter 36t på grunn av en semaforfeil, noe som nok ikke var på grunn av koden men f.eks at maskinen var i bruk i mens den kjørte, men ikke ønskelig at skulle skje igjen. For det andre, er Boltzmann enda mer avhengig av modellen ettersom distribusjonen bruker modellen som logits, og vil derfor kanskje kunne påvirkes enda mer av denne endringen. Valget falt på RMSProp for dette forsøket, til forskjell fra Adam som har blitt brukt frem til nå. Resultatene etter 75t kjøretid kan sees i Figur 12.



Figur 12. Testscore, gjennomsnittsscore, epsilon og loss med RMSProp (gul) vs. Adam (grønn).

5 Diskusjon

I dette prosjektet har det blitt kjørt en rekke forsøk for å finne ut hva som skal til for å få en best mulig poengsum i retrospillet Atari Breakout ved hjelp av DQN. Deep Q-learning fungerer på mange måter på samme vis som vi mennesker lærer. Med prinsippet om Q-learning følger det at en utforsker og får en belønning eller straff, avhengig av om det en gjør er bra eller dårlig, i form av en god eller dårlig poengsum. Slik forstår vi om noe burde gjøres eller ikke, og kombinerer man dette med en hjerne som prøver se for seg hva en burde gjøre, samt et erfaringsminne som en

tilfører erfaringer etterhvert som man prøver seg frem, har man kommet langt på vei med å etterlikne vår læring. I lys av observasjoner gjort underveis skal disse resultatene nå settes opp mot hverandre for å tolke hva som påvirker denne læringsprosessen mest.

En kan starte med å se på valg av miljø. Som nevnt i kapittel 4.1 ble innledningsvis versjonen *Breakout-v0* valgt, men etter å ha blitt bevisst på elementene av tilfeldighet som finnes i denne versjonen, ble denne stilt opp mot *BreakoutDeterministic-v4* for sammenlikning. Et element av tilfeldighet en har i v0 er en sannsynlighet på 25% for at systemet bruker forrige handling igjen istedenfor ny handling. Dette gjør at på tross av at Q-tabellen sier vi burde ta en bestemt handling, kan systemet velge en annen, og utregningen av en ny reward blir unøyaktig ettersom anbefalt handling faktisk ikke ble utført. I tillegg, ettersom vi samler de 4 siste observasjonene til én observasjon vi ser på, vil ulik frameskip kunne gjøre det vanskeligere å finne konsekvente tilstander. Denne forskjellen i frameskip gir også en unøyaktighet i sammenslåingen av bilder, som kan forhindre systemet i å se ballens bane slik den faktisk er.

Noe en kan observere fra de første to kjøringene (Figur 5), var at selv om de ikke ble kjørt veldig lenge, er det en merkbar forskjell i hvordan agenten klarer seg i miljøene, og det var logisk å ta i bruk *BreakoutDeterministic-v4* videre. En ønsker å ha kontroll over de handlingene som faktisk blir utført, og en kan tenke seg til at det er bedre for modellen å alltid ha en konsekvent frameskip for tolkningen av bla. ballbanen i de 4 siste bildene. Derfor gikk valget på dette miljøet i de resterende kjøringene.

Videre ble det oppdaget at erfarings-bufferstørrelsen var definert feil i kildekoden. Det var tenkt at $1\,000\,000$ var en god verdi etter å ha lest andres erfaringer med sine implementasjoner, men den var feilaktig satt til $100\,000$. I Figur 6 kan en se hva dette utgjorde i forskjell. Frem til omtrent 7 millioner steg har det ikke allverdens å si, men så kan man observere at prestasjonen til agenten med mindre erfaringsminne stagnerer. Dette kommer mest sannsynligvis av at den har byttet ut gode erfaringer med mindre gode, og at den ikke hadde de riktige erfaringene til å prestere på samme nivå eller bedre enn tidligere.

Som en kan se av testresultatene, som er kjørt på samme tidspunkt for begge kjøringene, fortsetter agenten med større minne å prestere bedre og bedre. Dette kan ikke sies om den andre agenten. Figuren viser at minnestørrelsen var en hindring fra å prestere bedre, og når denne størrelsen ble justert klarte modellen fortsette forbedre seg og nå en poengsum over dobbelt så god som den tidligere beste. Denne avhengigheten av erfaringsminnet skyldes også balanseringen av *exploration vs. exploitation*, som forklart i kapittel 3.6, og blir naturlig nok større jo større andel av tiden vi utnytter erfaringsminnet til fordel for å utforske nye muligheter.

Dette bringer oss over til valg av utforskningsmetode. I dette prosjektet ble det tatt i bruk to former for utforskning, Epsilon Greedy og Boltzmann. Epsilon Greedy var brukt i de implementasjonene det ble tatt utgangspunkt i fra tidligere relevant arbeid, og var derfor et naturlig valg å starte med. Med en riktig balansegang i form av en relativt lav `eps_min`, kan denne prestere svært godt - og det er ikke største forskjellen i prestasjon mellom beste kjøring på denne og Boltzmann. Dette kommer av hvor stor betydning *experience replay* har. Resultatene med Boltzmann, med prinsippet om en stadig mindre utforskningsrate, var noe bedre enn med Epsilon Greedy, og ble derfor valgt for videre kjøringer. Det er også denne metoden som oppnådde den høyeste poengsummen på 381.

Under original kjøring fikk den en toppscore på 364, men her var det ikke tap av liv som var begrensende faktor. Fra det originale miljøet var det satt et begrenset antall operasjoner

for å forhindre den i å kjøre uendelig dersom den for eksempel skulle sette seg fast i en loop der den ikke starter spillet. Dette skjedde under en kjøring på det originale miljøet, og det ble derfor implementert en maksgrense. I det nye reduserte miljøet var denne verdien dog ikke nødvendig. Ved å øke maksverdien fra 1000 til 10 000 og kjøre Boltzmann på nytt i det reduserte miljøet med utgangspunkt i den beste modellen fra forrige kjøring, klarte den oppnå 381 som beste resultat. En ting som kunne vært spennende eksperimentere med videre, er de to metodene med en større `eps_decay`, slik at den hadde utforsket enda mer i starten før den begynner sterkt avhenge av erfaringsminnet.

Avslutningsvis i dette prosjektet ble de to optimaliseringsmetodene Adam og RMSProp satt opp i mot hverandre. Adam ble valgt med utgangspunkt i tidligere relevant arbeid, samtidig som at det er en populær optimaliseringsfunksjon for maskinlæringsmodeller i nyere tid. RMSProp har som fremgangsmåte å prøve stabilisere oscilleringer, og dette er noe en kan se i praksis om en studerer resultatene vist i Figur 12. Denne har mer stabile resultater enn Adam i testene, men til gjengjeld klatrer agenten med Adam raskere. Dette støttes av teorien, ettersom Adam kombinerer AdaGrad og RMSProp [7]. AdaGrad har en læringsrate per vekt som øker ytelsen på problemer med mangelfulle gradienter (som f.eks i naturlig språk og maskinsyn). RMSProp har også disse læringsratene, men disse baseres på gjennomsnittet av omfanget av gradienten til en vekt, altså hvor mye den varierer. Dette gjør at den fungerer bedre på problemer med støy. Adam henter fordeler fra begge to, og som vi har sett i praksis, konvergerte den kjappere mot en beste score på 381, og dette var en score RMSprop ikke helt nådde selv om den ble kjørt et døgn lengre.

6 KONKLUSJON

I lys av de forsøkene som har blitt gjort, er det et par erfaringer undertegnede har oppgjort seg underveis. For det første, tar disse kjøringene lang tid. Det er viktig å gi agenten tid til å utforske, og gi den muligheten til å erfare og lære av dette. Kjøretiden har variert fra 17-75t, og en kan observere fra resultatene som har blitt presentert at den må kjøre relativt lenge før en ser resultater. Det finnes andre tilnærminger som skal prestere bedre enn det som har blitt gjort i dette prosjektet, for eksempel ved å ta i bruk en actor-critic-tilnærming som beskrevet i senere publikasjoner av DeepMind [3]. Når det er sagt, var ønsket for dette prosjektet å implementere en løsning som bygger på det tidligste gjennombruddet til DeepMind, og som replikerer slik vi mennesker lærer på egenhånd. De siste kjøringene har agenter som presterer bedre enn flertallet av mennesker som spiller spillet, og den første implementasjonen til DeepMind sin prestasjon med en enkel DQN er blitt forbigått. Den siste lagrede modellen oppnår en score på 360, og dersom du ønsker kjøre denne skal den oppnå det samme ettersom systemet er deterministisk.

En ide for fremtidig arbeid er å implementere en actor-critic-løsning i form av for eksempel enten A2C eller A3C, og sammenlikne med implementasjonen av DQN i dette prosjektet. En annen ide kunne vært å forsøkt erstattet framestackingen og det konvolusjonelle nevrale nettverket med et recurrent neural network som tar inn forrige observasjon som input istedenfor, og sett hvordan denne presterte i forhold. Hadde det vært mer tid igjen i dette prosjektet hadde nok en av disse to tingene blitt forsøkt implementert. Allikevel kan en si seg fornøyd med hva som har blitt gjort av arbeid, og undertegnede avslutter dette prosjektet med en god del mer kunnskap om emnet enn ved inngangen til prosjektet.

7 REFERANSER

[1] Google DeepMind. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Weristra, M. Riedmiller. (2013). *Playing Atari with Deep reinforcement Learning*. https://arxiv.org/pdf/1312.5602v1.pdf

[2] Google DeepMind. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis. (2015). *Human-level control through deep reinforcement learning*. https://www.nature.com/articles/nature14236

[3] Google DeepMind. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, A. P. Badia, M. Mirza, T. Harley, T. P. Lillicrap. (2016). *Asynchronous Methods for Deep Reinforcement Learning*. https://arxiv.org/pdf/1602.01783.pdf

[4] E. Bonilla, J. Zeng, J. Zheng. (2016). *Asynchronous Deep Q-Learning for Breakout with RAM inputs*. http://cs229.stanford.edu/proj2016/report/BonillaZengZheng-AsynchronousDeepQLearningforBreakout-Report.pdf

[5] N. Cesa-Bianchi, G. Lugosi, C. Gentile, G. Neu. (2017). *Boltzmann Exploration Done Right*. https://papers.nips.cc/paper/7208-boltzmann-exploration-done-right.pdf

[6] M. G. Bellemare. (2013). *Fast, Scalable Algorithms for Reinforcemnt Learning in High Dimensional Domains*. Department of Computing Science. Edmonton, Alberta. https://deepsense.ai/wp-content/uploads/2018/07/bellemare13fast.pdf

[7] D. P. Kingma, J. Ba. (2017). *Adam: A Method for Stochastic Optimization*. https://arxiv.org/pdf/1412.6980.pdf

[8] Google DeepMind. D. Silver. (2016). *Tutorial: Deep Reinforcement Learning*. https://icml.cc/2016/tutorials/deep_rl_tutorial.pdf

[9] OpenAIs nettside, hentet ut den 01.11.20 fra: https://openai.com/about

[10] Nettsiden til OpenAI Gym, med oversikt over miljøer. Hentet ut den 18.11.20 fra: https://gym.openai.com/envs/#atari

[11] Informasjon om det originale Atari Breakout, hentet ut den 01.11.20 fra: https://en.wikipedia.org/wiki/Breakout_(video_game)

[12] Nyhetsartikkel om fremskrittene gjort av DeepMind i etterkant av deres nye publikasjon [2]. Hentet ut den 12.11.20 fra: https://www.wired.com/2015/02/google-ai-plays-atari-like-pros/

[13] J. Brownlee. (2017). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. Hentet ut den 14.11.20 fra: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[14] A. Singh. (2019). *Reinforcement Learning: Bellmann Equation and Optimality (Part 2)*. Hentet ut den 17.11.20 fra: https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3

[15] Nettsiden til Weights and Biases Dashboard, hentet ut den 15.11.20 fra: https://www.wandb.com/experiment-tracking

[16] Store norske leksikon. *Benchmark-test*. Hentet ut den 21.11.20 fra: https://snl.no/benchmark-test