CHALMERS

Maskininlärningsmetoder tillämpade på StarCraft 2

En undersökning av reinforcement och imitation learning

JONATHAN BERGQVIST
CARL CLAESSON
PONTUS ELIASSON
ADAM GRANDÉN
EDVIN LAM
ARVID LUNDBERG

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sweden 2019

Maskininlärningsmetoder tillämpade på StarCraft 2
En undersökning av reinforcement- och imitation learning

Supervisor: K V S Prasad, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
SE-412 96 Göteborg
Telephone +46 31 772 1000

Typeset in LaTeX
Göteborg, Sweden 2019

Maskininlärningsmetoder tillämpade på StarCraft 2
En undersökning av reinforcement och imitation learning
JONATHAN BERGQVIST
CARL CLAESSON
PONTUS ELIASSON
ADAM GRANDÉN
EDVIN LAM
ARVID LUNDBERG
Institutionen för data- och informationsteknik
Chalmers tekniska högskola

Abstract

In the continuously developing field of artificial intelligence, machine learning has currently taken a central position. Previously rules were sufficient to solve mundane tasks, but today's challenges require more advanced methods. This thesis investigates different techniques for developing advanced artificial intelligence for the game StarCraft 2, and compares the different methods. In total three agents were developed. The first two were based on a reinforcement learning method called Advantage Actor Critic (A2C) where the second one also included imitation of an interactive expert policy. The third one was an behavioral cloning (BC) agent, which is a kind of imitation learning, trained on a dataset pre-generated by an expert policy. A2C with imitation resulted in the highest win rate of 75 % against the built-in hard AI while the BC agent only achieved a 17 % win rate. The basic A2C agent could only win against the very-easy AI. In conclusion our agents that used imitation learning performed better than the one only using reinforcement learning, possibly due to an insufficient implementation of A2C.

Sammandrag

Inom artificiell intelligens, som kontinuerligt utvecklas, har maskininlärning tagit en central roll. Medan regelbaserad AI varit tillräcklig för att lösa grundläggande uppgifter behöver dagens utmaningar mer avancerade metoder. Arbetet undersöker olika tekniker för att utveckla avancerad artificiell intelligens till spelet StarCraft 2, och jämför dem mot varandra. Totalt utvecklades tre agenter. De första två baserades på en reinforcement learning-metod kallad Advantage Actor Critic (A2C) där den andra även inkluderade imitation av en interaktiv expertpolicy. Den tredje var en behavioral cloning-agent (BC), som är en sorts imitation learning, tränad på en datamängd förgenererad av en expertpolicy. A2C med imitation resulterade i den högsta vinstandelen på 73,4 % mot den svåra inbyggda AI:n medan BC-agenten enbart uppnådde en vinstandel på 17 %. Den grundläggande A2C-agenten kunde enbart vinna mot den mycket lätta AI:n. Slutsatsen är att de agenter som använde sig av imitation learning presterade bättre än den som enbart använde reinforcement learning, möjligtvis på grund av en otillräcklig implementering av A2C.

Innehåll

1

Inledning

Beräkningshastigheten hos datorer har markant ökat årligen i flera decennier, precis som den amerikanske datavetaren Gordon Moore förutspådde redan 1965 (Moore 1965). Under de senaste 60 åren har beräkningshastigheten ökat en biljon gånger och ökningen verkar inte sluta (Expert Exchange 2019). Ett viktigt teknikområde som har stor nytta av dessa snabba beräkningar är forskningen kring artificiell intelligens.

1.1 Artificiell intelligens

Artificiell intelligens, eller AI, är ett vetenskapsområde med mål att skapa intelligenta maskiner som inom många områden kan automatisera komplext arbete där enbart beräkningskraft inte räcker till (McCarthy 2007). Det kan exempelvis vara bilkörning eller diagnostisering av sjukdomar. Verkligheten är ofta väldigt komplicerad och därför kan spel används som substitut. Spel har ofta strikta regler och är därför mer förutsägbara. Genom att utveckla AI som kan spela spel kan olika algoritmer utvärderas innan de appliceras på verkligheten.

Det går att dela upp AI i två kategorier: symbolisk AI och icke-symbolisk AI. Symbolisk AI bygger på en underliggande kunskapsbas som redan innehåller information ur vilken AI:n, med vissa regler, kan dra slutsatser eller utföra handlingar (D'souza 2018). Beslutsfattandet följer på så sätt ett resonemang som även människor kan följa (Bhatia 2017). Symbolisk AI kan även kallas regelbaserad AI. Ett stort framsteg med symbolisk AI gjordes år 1997 då IBM:s schackspelande dator *Deep Blue* vann över världsmästaren i schack, Garry Kasparov (Anderson 2017). Den kunde utforska upp till 200 miljoner möjliga schackpositioner per sekund och kunde med hjälp av en algoritm bestämma det optimala draget (IBM 2011).

Ett brädspel som en ren symbolisk AI:n ännu inte har lyckats besegra professionella spelare på är go. Det har en mycket större tillståndsrum att utforska jämfört med schack (Silver och Hassabis 2016). Med hjälp av ett artificiellt neuronnät lyckades dock företaget DeepMind skapa den icke-symboliska AI:n *AlphaGo* som snabbt kunde evaluera värdefulla drag enbart baserat på stenarnas nuvarande positioner. Det hade den lärt sig genom maskininlärning. Detta minskade utfallsrummet avsevärt och i kombination med klassiska sökalgoritmer kunde AlphaGo besegra världsmästaren Lee Sedol (Silver, Huang

m. fl. 2016; Hassabis 2016).

1.2 Maskininlärning och artificiella neuronnät

Maskininlärning är ett delområde inom AI som handlar om att få datorer att lära sig lösa problem genom analys av stora datamängder, utan att explicit programmera hur de ska lösa uppgiften (Chollet 2018). På senare år har stora framsteg gjorts inom området. Framförallt beror det på utveckling av bättre hårdvara och den stora mängd data som tillgängliggjordes genom internets framväxt under början på 2000-talet (Chollet 2018). Bild- och taligenkänning, självkörande bilar och chatbottar är alla exempel på ny teknik som möjliggjorts av maskininlärning och som ansågs vara mycket svåra problem att lösa bara för några år sedan. Många av de senaste framstegen inom maskininlärning bygger på en kategori beräkningsmodeller kallade artificiella neuronnät, ANN. De inspireras av sättet neuroner utför beräkningar i biologiska hjärnor. Utvecklingen av dessa började redan under 1940-talet och framsteg fortsatte att göras under hela 1900-talet. Otillräckliga mängder beräkningskraft och data gjorde dock att tillämpningarna länge förblev begränsade (Wikipedia 2019[a]).

Med utvecklingen som skett av beräkningshastigheter i samband med nya möjligheter att samla in data, samt lagring av denna data, har ett stort intresse uppstått under senare år (Chollet 2018). Idag är maskininlärning med neuronnät ett attraktivt forskningsområde med många stora företag och forskningsgrupper såsom Google och OpenAI inblandade(OpenAI 2018; Silver, Huang m. fl. 2016).

1.3 StarCraft 2

Ett spel som idag utforskas i maskininlärningssyfte är StarCraft 2. Det är ett realtidsstrategispel utvecklat av företaget Blizzard Entertainment. Spelarens främsta objektiv är att besegra sina motståndare genom att förstöra deras enheter och byggnader. För att åstadkomma detta behöver spelaren samla in resurser. Dessa resurser används för att bygga enheter som i sin tur används för att attackera motståndarna. Figur 1.1 visar en bild från spelet. Där syns spelet ur den mänskliga spelarens perspektiv, med karta, enheter, byggnad och källor för olika resurser.

Figur 1.1: Figuren visar StarCraft 2 från en spelares perspektiv. Här syns även gränssnittet som spelaren använder för interaktion med spelet. På den övre delen syns också enheter kallade SCV, som bygger byggnader och samlar resurser. I mitten är basbyggnaden och runt omkring blåa kristaller för att samla mineraler, och gröna källor för att samla gas.

StarCraft 2 innehåller intressanta utmaningar för maskininlärning. Detta är samma anledningar till att det anses vara ett av de svåraste spelen i världen att bemästra (Sun m. fl. 2018). Några av dessa är:

- *Realtid:* StarCraft 2 är ett realtidsstrategispel. Det innebär att alla spelare utför handlingar samtidigt. Detta skiljer sig från spel såsom Schack där spelare väntar på varandra. Det gör att det är viktigt att utföra handlingar vid rätt tidpunkt.
- *Handlingsrum:* Mängden handlingar som kan utföras vid varje tidpunkt kallas för handlingsrummet. Denna mängd uppskattas ha en storlek på ca 10^{26} i StarCraft 2. För liknelse har handlingsrummet i brädspelet go en storlek på 361 (DeepMind 2019).

1.4 Syftet med arbetet

Målet var att skapa en konkret applikation av maskininlärning i form av en AI till StarCraft 2. AI:n skulle kunna besegra de inbyggda regelbaserade AI-motståndarna upp till åtminstone svår svårighetsgrad. Fokus lades främst på området reinforcement learning då detta uppfattades ha större potential än de andra undersökta metoderna. Imitation learning

undersöktes också för att kunna jämföra och analysera mellan metoderna.

Syftet med detta är att undersöka hur maskininlärning kan användas för att låta en dator lösa komplexa problem som vanligtvis är lätta för människor att se svaret på. Spel i form av schack och go har tidigare använts som milstolpar för utvecklingen, och StarCraft 2 är nästa steg i den riktningen. Eftersom spelet kräver komplexa strategier och långtidsplanering skulle AI som med hjälp av maskininlärning lärt sig spela spelet utgöra ett tydligt framsteg i modellering av mänskligt beslutsfattande.

1.5 Etiska problem med AI

I dagens samhälle finns det vissa etiska svårigheter med AI. Om AI utvecklas oförsiktigt till att bli för avancerad fruktar många att det kan leda till stora samhälleliga problem. Ett problem handlar om att avancerad AI skulle kunna ersätta människor inom många yrken. Detta kan leda till utbredd arbetslöshet och växande samhällsklyftor. En annan aspekt handlar om när AI ska fatta moraliska beslut, exempelvis med självkörande bilar. Om en situation uppstår där antingen föraren eller en fotgängare skadas på grund av beslut fattande av AI, vems fel var olyckan (Nyholm och Smids 2016)? Vidare finns även diskussioner om de generella applikationsmöjligheterna som dagens AI-forskning siktar på. AI är sällan bunden till en viss applikation utan kan i många fall, med små modifikationer, direkt föras in i andra omgivningar. Detta skapar problem då de kan användas för skapa lösningar på olämpliga problem eller användas i militära applikationer, vilket många forskare är emot (Bossman 2016).

Vi anser dock att de etiska konsekvenser vårt projekt kan ha är mycket små då vår problemställning handlar om att skapa en AI för att spela ett spel. Eftersom vi applicerar principer snarare än utvecklar dem finns ingen risk att vi inför nya koncept som kan missbrukas.

2

Teori

Maskininlärning är ett begrepp inom datavetenskap som handlar om att få datorer att lära sig från olika typer av data. Maskininlärning är bra för att hitta lösningar till problem genom att identifiera mönster i datan. Det finns många olika möjliga tillvägagångssätt för att implementera maskininlärning. Nedan presenteras därför de begrepp och maskininlärningsmetoder som är relevanta för arbetet.

2.1 Grundläggande uttryck

Oavsett området inom artificiell intelligens används ibland olika uttryck för att beskriva de grundläggande koncepten. Här förtydligas hur de används i denna rapport.

2.1.1 Tillstånd

Tillståndet, även kallat *state*, är den nuvarande situationen i omgivningen som ska modelleras. Omgivningen kan vara ett spel eller någon annan typ av miljö. Exempelvis startar ett schackparti alltid i samma tillstånd, med alla pjäser på brädet på varsina sidor.

Beslut kommer ofta fattas baserat på nuvarande tillstånd. Tillståndet kan också innefatta saker som inte är direkt synliga, exempelvis hur många resurser som finns tillgängliga eller hur mycket tid som har gått. (Matiisen 2019)

2.1.2 Handlingsrum

Ett handlingsrum, eller *action space*, är de handlingar som är möjliga utifrån det nuvarande tillståndet. Exempelvis i spelet tre i rad, som spelas på en 3×3 ruta stor spelplan, har spelaren ett lika stort handlingsrum som tomma spelrutor. I StarCraft 2 är handlingsrummet otroligt stort, och definieras i sin renaste form som alla musklick spelaren kan göra.

2.1.3 Belöning

En belöning, *reward* på engelska, är sättet för en agent som tränas med maskininlärning att lära sig vilka handlingar som är bra och dåliga givet specifika tillstånd. När den utför handlingar som leder mot slutmålet är belöningarna positiva, vilket förstärker beteendet. Är handlingarna kontraproduktiva för målet kommer belöningen istället vara negativ och avråda för fortsatt beteende av den typen.

2.2 Reinforcement learning

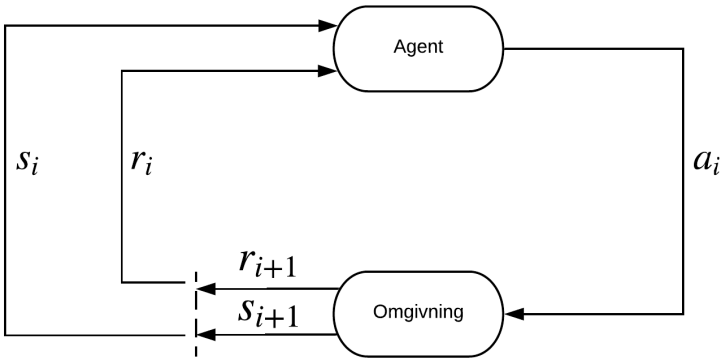
Reinforcement learning är ett område inom maskininlärning där agenter tränas till att lära sig utföra handlingar som maximerar en numerisk belöning i en omgivning. Belöningen är ett mått på hur bra agentens handlingar är på att uppnå det mål som träningen ska leda till. Exempelvis kan det handla om att vinna så många matcher i StarCraft 2 som möjligt. Agenten väljer till en början handlingar slumpmässigt och observerar vilken belöning de leder till. Den lär sig sedan över tid vilken handling som leder till den högsta förväntade belöningen i ett givet tillstånd.

2.2.1 Markov-beslutsprocesser

Det matematiska ramverket bakom reinforcement learning kan formuleras som en *Markov-beslutsprocess* (Sutton och Barto 2018), på engelska ofta förkortat *MDP*, vilket är ett stokastiskt sätt att modellera beslutsfattande. I en MDP befinner sig agentens omgivning vid varje tidpunkt t i ett tillstånd s_t. Agenten observerar vid varje tidpunkt omgivningens tillstånd och väljer utifrån det att utföra en handling a_t, som leder till att omgivningen hamnar i ett nytt tillstånd s_{t+1} och att en skalär belöning r_{t+1} ges till agenten. Processen fortsätter därefter att upprepas, antingen för evigt eller tills ett sluttillstånd s_T nås, där T är det totala antalet tidssteg (Sutton och Barto 2018). Denna iterativa process leder till en följd av tillstånd och handlingar

$$s_0, a_0, s_1, a_1, s_2, a_2, \ldots$$

som kallas för en *bana*. En visualisering av denna process kan ses i figur 2.1



Figur 2.1: Figuren visar interaktionen mellan en agent och dess omgivning. Agenten utför en handling som gör att omgivningen hamnar i ett nytt tillstånd och ger en belöning till agenten. Agenten utför sedan en ny handling beroende på det nya tillståndet och processen upprepas (Sutton och Barto 2018).

2.2.2 Begränsade Markov-beslutsprocesser

I en *begränsad* Markov-beslutsprocess är antalet möjliga tillstånd $\mathcal{S}$, handlingar $\mathcal{A}$ och belöningar $\mathcal{R}$ begränsade mängder (Sutton och Barto 2018). Varje möjlig kombination av nästa tillstånd $s_{t+1} = s'$ och belöning $r_{t+1} = r$, efter en vald handling $a_t = a$ i ett tillstånd $s_t = s$, har då en väldefinierad diskret sannolikhet

$$p(s', r|s, a)\,.$$

Både belöningen och nästa tillstånd är alltså slumpmässiga variabler. Sannolikheten för omgivningens nästa tillstånd $s_{t+1} = s'$ ges alltså av

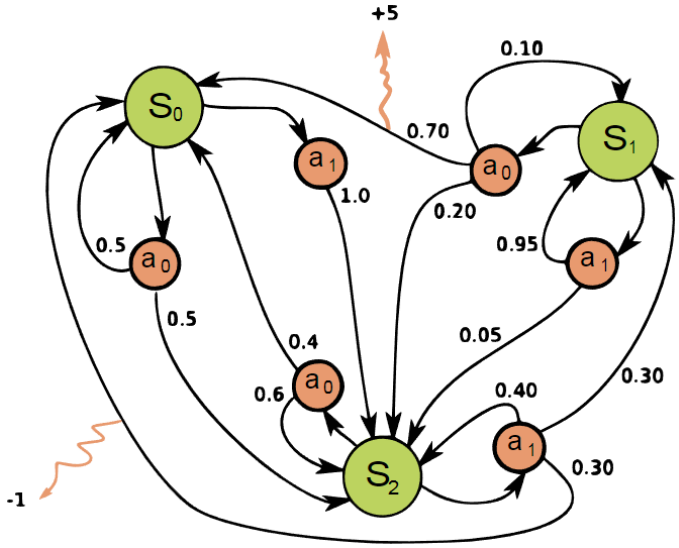
$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

vilket kallas för *tillståndsövergångssannolikheten* (Sutton och Barto 2018). Notera att detta inte är en begränsning till enbart stokastiska omgivningar, eftersom sannolikheten för nästa tillstånd i en deterministisk omgivning helt enkelt ges av

$$p(s'|s, a) = \begin{cases} 1 & \text{för ett enda tillstånd } s' \\ 0 & \text{för alla andra tillstånd} \end{cases}$$

Det tidigare nämnda exemplet schack är en omgivning med en sådan tillståndsövergångs-sannolikhet, eftersom samma drag vid en viss konfiguration av spelplanen alltid resulterar i samma nya konfiguration.

Ett exempel på en begränsad Markov-beslutsprocess visas i figur 2.2.



Figur 2.2: Figuren visar en grafisk representation av en begränsad Markov-beslutsprocess. Tillstånden representeras här av noderna s_0, s_1 och s_2 och de möjliga handlingarna av noderna a_0 och a_1. Pilar utgående från ett tillstånd visar handlingar som är möjliga från detta, och pilar utgående från en handling visar möjliga nästa tillstånd. Talen bredvid de sistnämnda pilarna är tillståndsövergångssannolikheterna $p(s'|s,a)$ från ekvation (2.2.2) och talen (-1, +5) vid ändarna av de pilar som pekar utåt är belöningar. (waldoalvarez [CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0]) (Wikipedia 2019[b])

2.2.3 Policy

En agent väljer handlingar enligt en så kallad *policy* $\pi(a,s)$ (Sutton och Barto 2018), som ger en betingad sannolikhet att välja handling a i tillstånd s, det vill säga

$$\pi(a,s) = p(a|s) \quad \text{eller ibland skrivet} \quad \pi(a|s)\,.$$

Om omgivningen utvecklas med upprepade tillämpningar av π fås en bana, och en sådan bana som börjar från initialtillståndet s_0 kallas för en *utrullning* av policyn π.

Vanligtvis är agentens mål att lära sig en policy som maximerar den totala framtida belöningen G_t från den nuvarande tidpunkten t

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots$$

För en icke-ändlig bana blir G_t dock oändligt stor, vilket är problematiskt. För att bland annat lösa detta problem införs ofta en *avdragsfaktor* $\gamma \in [0,1]$ i G_t (Sutton och Barto 2018) enligt

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k}$$

som garanterar att G_t konvergerar till ett ändligt värde. Avdragsfaktorn avgör hur högt agenten ska värdera framtida belöningar. Det kan ses som att agenten får en tidshorisont som den begränsas till att försöka maximera belöningen inom.

Inom imitation learning används också en så kallad *expertpolicy*, betecknat π^*, som ska vara en policy som maximerar eller är nära på att maximera G_t. Sådana policyer skulle kunna vara människostyrda såväl som optimala algoritmer.

För att återigen anknyta till schack-exemplet skulle en enkel policy kunna vara att alltid följa en förutbestämd ordning av drag, oberoende av hur spelplanen ser ut. Trots att en sådan policy kan leda till en och annan vinst är den långt ifrån optimal. En expertpolicy skulle då kunna vara en schackvärldsmästare, och förutsatt att belöningar ges utifrån vinster och förluster leder dennes drag till det högsta väntevärdet på G_t. För att avgöra vilket drag detta är kan *värdefunktioner* användas, vilka beskrivs i nästa delkapitel.

2.2.4 Värdefunktioner

Ett centralt begrepp inom reinforcement learning är *värdefunktionen* V_π av ett tillstånd s, vilket definieras som väntevärdet av G_t om agenten börjar i ett tillstånd och följer en specifik policy π (Sutton och Barto 2018)

$$V_\pi(s) = \mathbb{E}_\pi\big[\, G_t \,|\, s \,\big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s \right]. \tag{2.1}$$

Ett annat närbesläktat begrepp är *handlings-tillståndsvärdet*, oftare kallat *Q-värdet*, av en handling a vid tillstånd s. Det definieras som väntevärdet $\mathbb{E}_\pi$ av G_t om agenten befinner sig i tillstånd s, väljer handling a och sedan följer en specifik policy π (Sutton och Barto 2018)

$$Q_\pi(s,a) = \mathbb{E}_\pi\big[\, G_t \,|\, s,a \,\big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s,a \right] \tag{2.2}$$

I princip kan både tillståndsvärdena $V_\pi(s)$ och Q-värdena $Q_\pi(s,a)$ under en viss policy π uppskattas genom att låta agenten följa policyn i omgivningen ett stort antal gånger och medelvärdesbilda den totala belöningen G_t som erhölls efter varje tillstånd och handling som förekom. Upprepas detta tillräckligt många gånger kommer medelvärdena $\bar{V_\pi}(s)$ och $\bar{Q_\pi}(s,a)$ att konvergera till de korrekta värdena $V_\pi(s)$ och $Q_\pi(s,a)$. Denna metod kallas för *Monte Carlo-sampling* och är en av flera metoder för att uppskatta de båda värdefunktionerna (Sutton och Barto 2018).

I många fall är dock antalet möjliga tillstånd $\mathcal{S}$ och handlingar $\mathcal{A}$ väldigt stort, vilket dessvärre gör Monte Carlo-metoder ineffektiva (Sutton och Barto 2018). Detta eftersom agenten måste utforska omgivningen orimligt många gånger innan bra uppskattningar fås av de båda värdefunktionerna $V_\pi(s)$ och $Q_\pi(s,a)$. Då används ofta parametriserade uppskattningar $V_{\pi,\theta}(s)$ och $Q_{\pi,\theta}(s,a)$ av dessa istället, vars värden beror på en uppsättning anpassningsbara parametrar $\theta = (\theta_1, \theta_2, ...)$. Exempelvis kan artificiella neuronnät användas för sådana parametriserade uppskattningar, vilket beskrivs senare i kapitel 2.4.

2.2.5 Temporal Difference learning

Gemensamt för både tillståndsvärdena och Q-värdena är att de uppfyller *Bellmanekvationen*, vilket innebär att värdet för ett visst tillstånd kan uttryckas i värdet för nästkommande tillstånd (Sutton och Barto 2018). För tillståndsvärdena blir detta

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\big[\, r_{t+1} + \gamma r_{t+2} + \ldots \,|\, s \,\big] = \mathbb{E}_\pi\big[\, r_{t+1} + \gamma G_{t+1} \,|\, s \,\big] \\ &= \mathbb{E}_\pi\big[\, r_{t+1} + \gamma V_\pi(s') \,|\, s \,\big] \end{aligned} \tag{2.3}$$

där $V_\pi(s')$ är tillståndsvärdet för nästa tillstånd s'. Figur 2.3 visar en grafisk illustration av detta. Denna egenskap gör det möjligt att uppskatta $V_\pi(s)$ utan att behöva samla in fullständiga banor τ (Sutton och Barto 2018); tillståndsvärdet $V_\pi(s)$ kan enligt den sista likheten i (2.3) istället uppskattas av

$$\hat{V}_\pi(s) = r_{t+1} + \gamma \hat{V}_\pi(s') \tag{2.4}$$

där $\hat{V}_\pi(s')$ i sin tur är en uppskattning av nästa tillståndsvärde gjord på samma sätt.

Genom att likt Monte Carlo-sampling låta agenten följa policyn i omgivningen ett stort antal gånger, men istället uppdatera tillståndsvärdena enligt ekvation (2.4) efter varje steg, kommer dessa till slut konvergera till de korrekta tillståndsvärdena (Sutton och Barto 2018). Denna metod där ett tillståndsvärde uppskattas utifrån det följande tillståndsvärdet kallas för *one-step Temporal Difference learning* eller *TD(0)*. Det gör det alltså möjligt att förbättra uppskattningarna av tillståndsvärdena under träningsprocessen istället för att behöva vänta tills N stycken banor har samlats in som vid Monte Carlo-sampling.

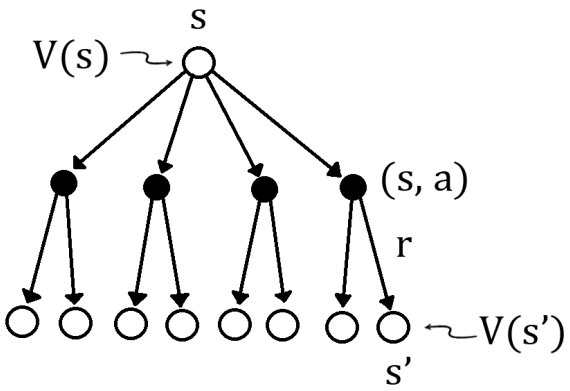
En mer generell variant av metoden är *n-step Temporal Difference learning* eller *TD(n)*. På ett liknande sätt som tidigare kan tillståndsvärdet skrivas som

$$V_\pi(s) = \mathbb{E}_\pi\big[\, r_{t+1} + \gamma r_{t+2} + \ldots \,|\, s \,\big] = \mathbb{E}_\pi\big[\, \sum_{k=0}^{n} \gamma^k r_{t+1+k} + \gamma^{n+1} V_\pi(S_{t+1+n} = s') \,|\, s \,\big]$$

vilket betyder att $V_\pi(s)$ kan uppskattas utav

$$\hat{V}_\pi(s) = \sum_{k=0}^{n} \gamma^k r_{t+1+k} + \gamma^{n+1} \hat{V}_\pi(S_{t+1+n} = s')$$

där $\hat{V}_\pi(S_{t+1+n} = s')$ alltså är värdet för det resulterande tillståndet efter n steg. Ovanstående ekvationer gäller även för Q-värdena som därför också kan uppskattas genom TD learning. (Sutton och Barto 2018)



Figur 2.3: Figuren visar en grafisk illustration av sambandet mellan värdet för ett tillstånd s och värdet för ett följande tillstånd s'. Enligt Bellmanekvationen 2.3 är värdet $V(s)$ väntevärdet av alla möjliga nästa tillstånds värden och belöningen som dessa ger. I figuren visas ett av flera sådana möjliga nästa tillstånd med värdet $V(s')$ och tillhörande belöning r.

2.2.6 Experience replay

För att mer effektivt utnyttja uppsättningar av tillstånd, handlingar och tillhörande belöningar, kallade *erfarenheter*, som ses av agenten kan en metod kallad *experience replay, DRLseries* användas (Matiisen 2019). Erfarenheterna sparas då i en så kallad *replay buffer*, vilken kan ses som ett minne för agenten, som de kan hämtas ifrån och tränas på flera gånger. Vanligtvis sker detta parallellt med att agenten utforskar omgivningen. Detta exempelvis genom att välja ut och träna agenten på ett antal slumpmässiga erfarenheter för varje val av handling den gör. Eftersom agenten då tränas på en blandning av erfarenheter från väldigt olika delar av omgivningen och som därmed har en låg korrelation leder det också till en policy som är stabilare i en större mängd tillstånd.

2.2.7 Q-learning

Q-learning är en reinforcement learning-metod som går ut på att använda Bellmanekvationen för att uppskatta den optimala Q-värdefunktionen (Russel och Norvig 2009). Q-värdena sparas i en så kallad Q-tabell, där varje handlings Q-värde givet ett visst tillstånd kan slås upp. Enligt Bellmanekvationen ges Q-värdefunktionen av (Sutton och Barto 2018)

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[r_{t+1} + \gamma V_\pi(s') \,|\, s,a\right]$$

Eftersom detta är ett väntevärde kan det uppskattas genom medelvärdesbildning. Då den optimala Q-värdefunktionen fås av en policyn som alltid väljer handlingen a med störst Q-värde blir uppskattningen enligt Bellmanekvationen för Q-värdena

$$\hat{Q}_\pi(s,a) = r_{t+1} + \gamma \max_a \hat{Q}(s',a) \tag{2.5}$$

där s' är nästa tillstånd. Medelvärdesbildning av tillräckligt många uppskattningar enligt (2.5) konvergerar alltså till de optimala Q-värdena. Då medelvärdesbildningen brukar sker kontinuerligt under träningsförloppet så görs ett viktat medelvärde mellan det gamla medelvärdet $\overline{Q}_\pi(s,a)_{\text{gammalt}}$ och den nya Q-värdesuppskattningen $\hat{Q}_\pi(s,a)$ efter varje val av handling enligt

$$\overline{Q}_\pi(s,a) = (1-\alpha)\overline{Q}_\pi(s,a)_{\text{gammalt}} + \alpha\hat{Q}_\pi(s,a)$$

som alltså efter tillräckligt många utrullningar konvergerar till den optimala Q-värdesfunktionen. Vikten α brukar kallas för inlärningshastighet och avgör hur högt nya Q-värdesuppskattningar $\hat{Q}_\pi(s,a)$ ska viktas i medelvärdesbildningen. Hur denna bäst väljs är något som behöver undersökas för varje omgivning, men små värden som exempelvis 10^{-5} är vanliga.

2.2.8 Policy gradient

En annan typ av metod inom reinforcement learning är så kallad *policy gradient*, vilken skiljer sig från värdebaserade metoder som exempelvis Q-learning. Policy gradient hittar varken optimala värdefunktioner eller väljer handling utefter dessa i varje tillstånd. Istället syftar den till att parametrisera agentens policy och justera denna så att väntevärdet av den totala framtida belöningen maximeras (Sutton och Barto 2018; Hui 2019a). Policyn är då beroende av, och differentierbar med avseende på, en uppsättning parametervärden $\theta \in \mathbb{R}^d$ och tecknas ofta som

$$\pi_\theta(a,s).$$

Likt värdefunktionen i ekvation (2.1), som är den förväntade totala framtida belöningen givet ett tillstånd, kan en målfunktion J införas. Målfunktionen beskriver den förväntade totala framtida belöningen från initialtillståndet givet en policy π_θ som ges av (Sutton och Barto 2018)

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)\right] = \sum_\tau P_\theta(\tau)R(\tau), \tag{2.6}$$

där τ är en bana.

$P_\theta(\tau)$ är sannolikheten att en bana τ följs givet en policy π_θ och den totala belöningen $R(\tau)$ som banan ger. En bra policy leder till en större belöning vilket ökar J. Inlärningen blir därför ekvivalent med att maximera J, alltså finna parametrar θ^* som maximerar målfunktionen:

$$\theta^* = \arg\max_\theta J(\theta) = \arg\max_\theta \sum_\tau P_\theta(\tau)R(\tau).$$

θ^* kan hittas iterativt genom *gradient ascent*. Metoden går ut på att finna målfunktionens gradient med avseende på θ och sedan uppdatera θ med gradienten och inlärningshastigheten α enligt

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta). \tag{2.7}$$

Valet av α blir en balansgång, då små α innebär små uppdateringar till θ och därmed långsammare träning, medan stora α kan leda till att maximumet i J passeras. Med gradient

ascent blir maximumet inte heller garanterat globalt då metoden inte skiljer på lokala och globala maximum. Detta innebär att policyn lär sig en strategi som är bättre än liknande strategier men är potentiellt långt från den bästa.

Det går att visa att $\nabla_\theta J(\theta)$ kan skattas som

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{t=0}^{T-1} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t}))R(\tau_i) \tag{2.8}$$

där N är det totala antalet episoder som ska användas i skattningen och i är den i:te episoden, samt att antalet tidssteg begränsats till T. För en härledning av detta uttryck hänvisas läsaren till appendix B. I praktiken blir flödet som i algoritm 1, som sedan körs många gånger för att förbättra policyn.

input : Policyns parametrar θ, antal episoder N, träningsparameter α
output: Uppdaterade parametrar θ

$gradient \leftarrow$ en array av N nollor;
for $i \leftarrow 0$ **to** N **do**
    Generera $\tau = (s_0, a_0, s_1, a_1, ..., s_{T-1}, a_{T-1})$ för en hel episod utefter π_θ;
    Beräkna belöningen R givet banan τ;
    $gradient[i] \leftarrow \frac{1}{N} \sum_{t=0}^{T-1} \nabla_\theta \log(\pi_\theta(a_t|s_t))R$;
end
return $\theta + \alpha \sum_{i=0}^{N-1} gradient[i]$

Algoritm 1: Algoritm som beskriver policy gradient. Här syns är processen för att gå igenom banor och uppdatera gradienten efter vilken belöning som erhölls.

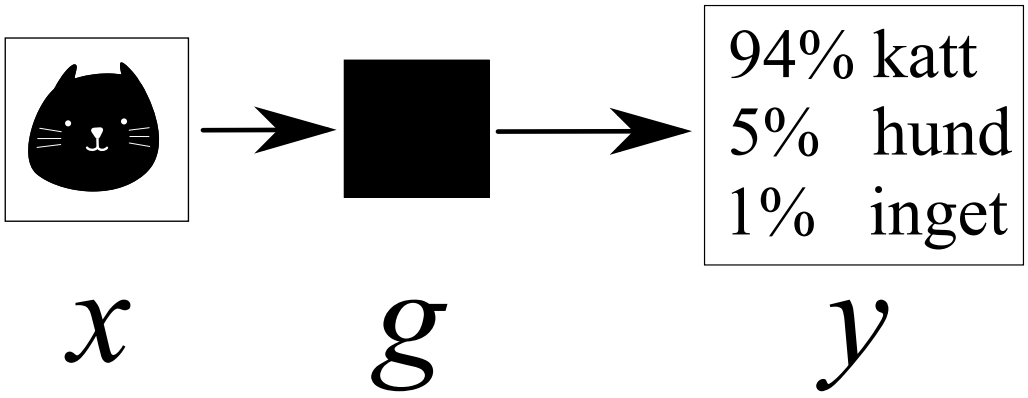
2.3 Supervised learning

Supervised learning är den gren av maskininlärning som går ut på att lära en AI ge rätt utdata givet en viss indata (Russel och Norvig 2009; deeplizard 2019) genom att träna den med ett stort antal korrekta in-utdata-par. Den gör detta genom att optimera en rad vikter θ för att minimera den så kallade *förlustfunktionen*. Det är en typ av målfunktion som ska minimeras istället för att maximeras.

2.3.1 Klassifikation och regression

I ett klassifikationsproblem har användaren någon data som ska klassificeras i någon av flera klasser (Russel och Norvig 2009). Exempelvis kan problemet vara att bestämma om en bild innehåller en katt, en hund eller inget alls. Indatan kan då vara en bild vars pixlar ses som matriser. När en sådan bild x matas in i klassificeringsfunktionen g fås ett utfall

y som i detta fallet är en vektor med tre element. Det första elementet är sannolikheten att det är en katt, det andra elementet är sannolikheten att det är en hund och det tredje elementet är sannolikheten att det inte är något av dem. Detta illustreras i figur 2.4.



Figur 2.4: Ett exempel på hur indata i form av en bild x klassificeras av g som en sannolikhet av tre möjliga utfall i en vektor $g(x) = y$. I exemplet är det enligt g väldigt troligt att x föreställer en katt.

2.3.2 Träning och validering

Vid supervised learning krävs insamlad data som modellen kan arbeta utifrån. Dessa delas in i träningsdata och valideringsdata. Det är viktigt för utvecklingen av en modell att få många olika tillstånd och handlingar att tränas på, för att den lättare ska kunna hitta mönster. Därför används träningsdata separat från valideringsdata. På så sätt kan valideringsdatan, som då är unik, användas för att se hur effektiv modellen har tränats. Så länge modellen inte tränas på valideringsdatan kan ytterligare validering utföras med hjälp av den efter fortsatt träning. Resultaten kan då jämföras mellan olika generationer av modellen för att se när den når sin topp. Risken med för mycket lik träningsdata är så kallad *overfitting* (Russel och Norvig 2009), när den tror att den hittar mönster som faktiskt inte finns då för mycket data är för lik.

För att se hur väl en modell lärt sig används valideringsdatan tillsammans med en förlustfunktion L (deeplizard 2019; Russel och Norvig 2009). En vanlig sådan är medelkvadratfelet, som ges av

$$L = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2$$

där y är en vektor som representerar utdatan som modellen ska lära sig, $\hat{y}$ är modellens uppskattning av y och n utdatans dimension. Minimeras förlustunktionen så innebär det alltså att modellens utdata $\hat{y}$ är bra uppskattningar av den korrekta utdatan y.

2.3.3 Imitation learning

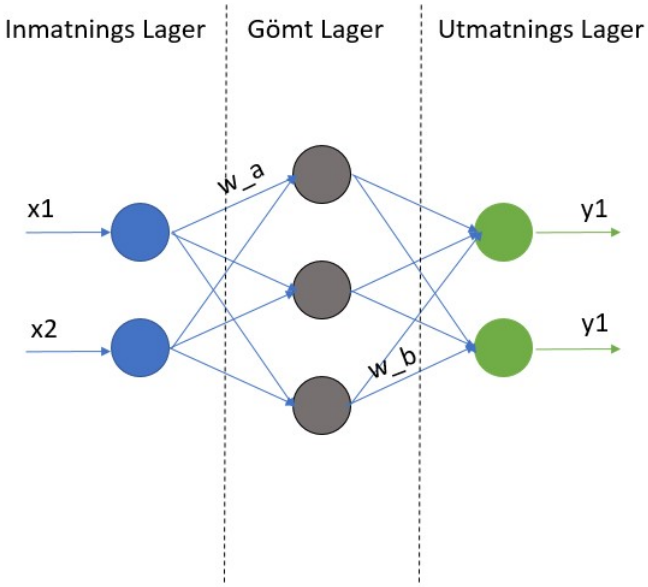
Imitation learning är en underkategori av supervised learning. Istället för att lära en AI att klassificera olika indata, lär AI:n sig en policy π som ska imitera en så kallad expertpolicy π^* (Yue och Le 2018). Det finns olika sätt att utföra imitation learning. Ett enkelt sätt är *behavioral cloning* där problemet behandlas som ett vanligt klassifikationsproblem med skillnaden att träningsmängden kommer från många utrullningar av π^*. AI:ns egna policy π kommer då tränas till att försöka utföra samma handlingar som expertpolicyn. En nackdel med den här metoden är att expertpolicyns utrullningar inte nödvändigtvis innehåller alla möjliga tillstånd; eftersom den endast utför bra handlingar kommer den exempelvis inte hamna i dåliga tillstånd. Det innebär att π inte kommer lära sig att hantera dessa tillstånd även om π^* vet vilka handlingar som är optimala i dessa fall (Yue och Le 2018). Sådana problem dyker fort upp då π sannolikt inte kommer härma π^* exakt.

En annan möjlighet är imitation learning med en interaktiv expertpolicy, där utrullningar av π^* inte är nödvändiga. Istället utförs träningen genom att i utrullningar av π låta π^* observera banan och skapa en imitationsförlustterm som beror på skillnaden i $\pi(a_t|s_t)$ och $\pi^*(a_t|s_t)$, exempelvis i form av medelkvadratfelet i ekvation (2.3.2). Detta låter agenten lära sig rätta till misstag, något som behavioral cloning inte kunde (Yue och Le 2018; Hui 2019b).

2.4 Artificiella neuronnät

Teorin bakom artificiella neuronnät började först utvecklas under 1940 talet där Warren McCulloch och Walter Pitts med hjälp av matematik och algoritmer baserade en metod som heter threshold logic (McCulloch 1990). Detta skapade vägen för början av neuronnät (Kleene 1956).

Ett artificiellt neuronnät är en beräkningsmodell som inspirerats av sättet biologiska hjärnor utför beräkningar och används vanligtvis för funktionsuppskattning (Wikipedia 2019[a]). De består av ett flertal sammankopplade lager av *noder*. Ett exempel på detta illustreras i figur 2.5 med ett neuronnät bestående av tre fullt sammankopplade lager med olika antal noder. Noderna i varje lager är kopplade till noderna i föregående lager såväl som de nästföljande. Om varje nod i ett lager är kopplade till alla noder i dess föregående lager kallas lagret för ett *dense layer*(deeplizard 2019), eller ett full sammankopplat lager. Det första lagret i ett nätverk kallas för inmatningslagret medan det sista lagret kallas för utmatningslagret. Data som ges till inmatningslagret kommer att propagera genom neuronnätet tills en utmatning fås från utmatningslagret. Vilken inmatning som ger en viss utmatning beror därmed på hur noderna i neuronnätet är sammankopplade och styrkan på dessa kopplingar. Nästa delkapitel beskriver vad detta innebär.



Figur 2.5: Ett neuronnät bestående av tre fullt sammankopplade lager: ett inmatningslager, ett gömt lager och ett utmatningslager. $x1, x2$ är indata och $y1, y2$ är utdatan från nätverket. w_a och w_b är exempel på vikter som finns på varje koppling.

2.4.1 Matematisk modell

Ett neuronnät fungerar genom att varje nod i ett lager ger ett utvärde som är en funktion av en viktad summa av utvärdena hos det föregående lagrets noder adderat med en bias b för varje nod (Skalski 2019) (Rashid 2016). För den i:te noden i det nuvarande lagret kan denna summa z_i tecknas

$$\begin{aligned} z_i &= w_{i,1}x_1 + w_{i,2}x_2 + ... + w_{i,n}x_n + b_i \\ &= \vec{w}_i\vec{x}^T + b_i, \end{aligned}$$

där

$$x = [x_1, x_2, ..., x_n]^T$$

är det föregående lagrets utvärden och

$$\vec{w}_i = [w_{i,1}, w_{i,2}, ..., w_{i,n}]^T$$

är deras respektive vikter tillhörande nod i. Denna beräkning utförs för varje nod i det nuvarande lagret. Om lagret innehåller m noder kan detta uttryckas med matrisekvationen

$$\vec{z} = W\vec{x}^T + \vec{b}.$$

Eller uttryckt fullständigt:

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

Nodens utvärde $\hat{y}_i$ bestäms med en funktion g, även kallad *aktiveringsfunktion*, så att

$$\hat{y}_i = g(z_i).$$

Detta beräknas för alla noder och skickas sedan vidare till nästa lager som en vektor, där processen börjar om på nytt (Skalski 2019; Sanjeevi 2019; Rashid 2016). Undantagen är då dessa är inmatnings- eller utmatningslager. Om neuronnätet används exempelvis som en policy i en reinforcement learning-algoritm kommer inmatningslagret få sina invärden från tillståndet s_t och utmatningslagret ge utvärdet a_t (eller $p(a_t|s_t)$).

2.4.2 Aktiveringsfunktioner

Aktiveringsfunktionen bestämmer karakteristiken av en nods utvärde (Babs 2019). Exempelvis finns den linjära aktiveringsfunktionen

$$g(z_i) = z_i$$

som inte ändrar på summan alls. Ett sådant utvärde kommer inte vara begränsat i storlek vilket inte alltid är önskvärt. Exempelvis om neuronnätet ska modellera sannolikheten att en bild föreställer en katt ska utvärdet varken kunna vara negativt eller större än 1.

En annan aktiveringsfunktion är *ReLU*, kort för *rectified linear unit*. Denna är samma som den linjära aktiveringsfunktionen förutom att alla negativa värden blir 0. ReLU är en mycket populär aktiveringsfunktion då den är både snabb och hjälper till att motverka problem med att gradienter blir för små (Skalski 2019).

2.5 Djupinlärning

Ett artificiellt neuronnät med fler än ett gömt lager kallas för *djupt*. En av de mest häpnadsväckande egenskaperna för djupa artificiella neuronnät är att de med tillräckligt många lager och noder samt rätt val av vikter kan visas kunna uppskatta *alla* möjliga kontinuerliga funktioner av dess inmatning godtyckligt bra (Sutton och Barto 2018). Notera dock att detta teoretiska resultat kan kräva enormt stora lager för att få exakta uppskattningar, men generellt är relativt små lager tillräckliga för att ge uppskattningar med god överensstämmelse för de flesta funktioner. Justering av ett neuronnäts vikter för att uppnå en viss typ av funktionsuppskattning brukar kallas för *inlärning*, och när detta görs för djupa neuronnät kallas det för *djupinlärning* eller på engelska *deep learning*.

Applikation av djupa neuronnät på reinforcement learning och supervised learning ger *deep reinforcement learning* och *deep supervised learning*. I deep supervised learning används neuronnät för att exempelvis ge en klassifikation av vad en bild visar. Nätverkets vikter uppdateras utifrån den eftersökta utsignalen. Vid deep reinforcement learning används djupa neuronnät exempelvis för att uppskatta värdefunktioner eller parametrisera agentens policy.

2.5.1 Deep Q-learning (DQL)

När ett neuronnät används för uppskattning av Q-värdena vid reinforcement learning kallas metoden för *Deep Q-learning*. Anledningen till att neuronnät används istället för en Q-tabell som vid den tidigare beskrivna metoden Q-learning är för att stora handlings- och tillståndsrum kräver en extremt stor mängd körningar för det ska gå att bestämma Q-värderna explicit. Dessutom krävs en stor mängd minne för att lagra dessa i en Q-tabell (Hui 2019a). Ett neuronnät däremot har en begränsad mängd justerbara vikter som enkelt kan lagras i datorns minne, och om tillräckligt många lager och noder används kan neuronnätets Q-värdesuppskattning ha god överensstämmelse med de korrekta Q-värdena.

2.5.2 Advantage Actor Critic (A2C)

Actor Critic är en metod som kombinerar de tidigare beskrivna metoderna Policy gradient och Deep Q-learning för effektivare inlärning (Nicholls 2019; Karagiannakos 2019). På samma sätt som i Policy gradient parametriseras agentens policy i A2C av en uppsättning justerbara parametrar θ i form av ett artificiellt neuronnät. I delkapitel 2.2.8 beskrevs målfunktionen $J(\theta)$ som ska maximeras av agenten vid Policy gradient, vilken ges av

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right] = \sum_{\tau} P_\theta(\tau) R(\tau).$$

Maximering av $J(\theta)$ kan göras genom så kallad gradient ascent, där parametrarna θ uppdateras enligt ekvation (2.7) med $\nabla_\theta J(\theta)$ uppskattad enligt

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{t=0}^{T-1} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) R(\tau_i). \tag{2.9}$$

För varje parameteruppdatering krävs då insamling och medelvärdesbildning av N stycken banor $\tau_1, \tau_2, \ldots, \tau_N$ enligt agentens nuvarande policy π_θ.

Ett sätt att tolka ekvation (2.9) är att policyns parametrar θ ändras så att den framtida totala belöningen utgående från initialtillståndet s_0 maximeras. Ett alternativt till detta är att istället ändra policyn till att maximera den totala framtida belöningen utgående från ett *slumpmässigt* tillstånd, vilket görs genom att ersätta $R(\tau) = V_\pi(s_0)$ med Q-värdet för den valda handlingen i ett sådant slumpmässigt tillstånd. Det nya uttrycket för parameteruppdateringen ges då av

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{t=0}^{T-1} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t})) Q_\pi(s_{i,t}, a_{i,t}) \tag{2.10}$$

Ytterligare ett neuronnät används vanligtvis för att göra en parametriserad uppskattning av Q-värdena $Q_{\pi,\phi}$ och tränas på samma sätt som i DQL.

Algoritmen använder alltså två stycken neuronnät. Det första parametriserar agentens policy och brukar kallas för en *actor*, och det andra gör en parametriserad uppskattning av Q-värdena och brukar kallas för en *critic*.

Temporal Difference learning, som beskrevs i delkapitel 2.2.5, kan även implementeras för att slippa samla in N stycken hela banor τ_i för varje uppdatering av neuronnätens parametrar. Uppdatering av neuronnätsparametrarna kan då ske efter n stycken par av tillstånd och handlingar har samlats in, vilket ger en avsevärt snabbare inlärning (Sutton och Barto 2018). För $n = 1$ sker uppdatering av neuronnätens parametrar efter varje val av handling. Criticens parametrar uppdateras som vid DQL och actorns parametrar θ uppdateras enligt

$$\widehat{\nabla_\theta J(\theta)} = \nabla_\theta \log(\pi_\theta(a_t|s_t)) Q_{\pi,\phi}(s_t, a_t). \tag{2.11}$$

Intuitivt kan detta tolkas som Q-värdena viktar uppdateringar av parametrarna θ mot policyer som leder till höga Q-värden.

Även om större parameteruppdateringar sker mot policyer som leder till höga Q-värden snarare än låga kan uppdateringar fortfarande ske mot de sistnämna med ekvation (2.11) så länge Q-värdena är positiva. En bättre strategi hade varit att uppdatera parametrarna bort från policyer med låga Q-värden. Av denna anledning brukar Q-värdena $Q_{\pi,\phi}(s_{i,t}, a_{i,t})$ i ekvation (2.11) ersättas med en så kallad *övertagsfunktion*, eller på engelska *advantage function*, $A_{\pi,\phi}(s_{i,t}, a_{i,t})$ som ges av

$$A_{\pi,\phi}(s_t, a_t) = Q_{\pi,\phi}(s_t, a_t) - V_{\pi,\phi}(s_t)$$

Eftersom värdefunktionen $V_{\pi,\phi}$ är lika med väntevärdet av ett tillstånds Q-värden

$$V_\pi(s) = \mathbb{E}_\pi\big[Q_\pi(s,a)\big] = \sum_a \pi(a|s) Q_\pi(s,a)$$

blir $A_\pi(s,a)$ ett mått på hur mycket bättre den valda handlingens Q-värde är än tillståndets medel-Q-värde. Uppdateringar kommer då ske bort från policyer som väljer handlingar med lägre Q-värden än tillståndens medel-Q-värde. Algoritmen brukar då kallas för *Advantage Actor Critic* eller *A2C*.

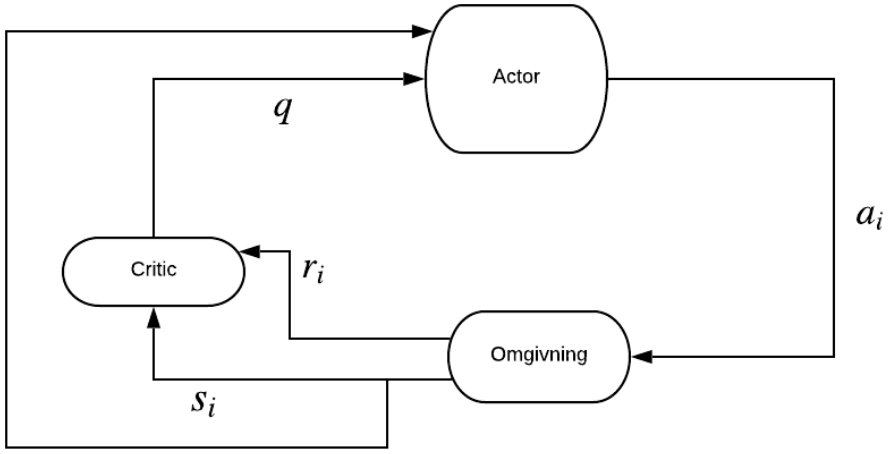
Entropin $H(\pi)$ av agentens policy kan även användas för öka agentens tendens att utforska omgivningen. Entropi är ett mått på hur utspridd en sannolikhetsfördelning är över de möjliga utfallen och definieras som

$$H(\pi) = -\sum_a \pi(a|s) \log \pi(a|s)$$

Hög entropi innebär en utspridd sannolikhetsfördelning där sannolikheten är jämförbart stor för varje möjligt utfall, medan låg entropi innebär en spetsig sannolikhetsfördelning där sannolikheten för ett fåtal utfall är hög och låg för övriga. Genom att lägga till en viktad entropiterm i ekvationen för parameteruppdateringar uppmuntras alltså agenten till att välja handlingar mer slumpmässigt och därmed utforska omgivningen.

Det slutgiltliga uttrycket för parameteruppdateringar med Q-värdena utbytta till en övertagsfunktionen och med en viktad entropiterm blir

$$\widehat{\nabla_\theta J(\theta)} = \nabla_\theta \Big(\log(\pi_\theta(a_t|s_t)) A_{\pi,\phi}(s_t, a_t) - \beta_H \sum_a \pi(a|s) \log \pi(a|s) \Big)$$



Figur 2.6: Figuren visar sig hur actor, critic och omgivningen arbetar med varandra. Här ser man hur handlingen a_i påverkar omgivningen som i sin tur avger ett tillstånd s_i och en belöning r_i. Criticen tar emot tillståndet och belöningen och skattar Q-värdet q som används för att uppdatera actorns policy.

3

Avgränsningar

På grund av den begränsade hårdvaran och tid projektet utförts under har en rad förenklingar av spelet gjorts.

3.1 Handlingsrum

För att reducera spelets stora handlingsrum har ett antal förbestämda handlingar programmerats in. Exempel på sådana handlingar är att bygga en enhet/byggnad eller att gå till attack. Dessa handlingar byggs upp utav ett antal mindre handlingar såsom musklick och kamerahantering. Genom denna abstrahering behöver AI:n inte hantera dessa alternativ och handlingsrymden reduceras till de handlingar vi definierat. Ett större handlingsrum hade teoretiskt sett varit möjligt, men hade endast utökat träningstiden väldigt mycket.

3.2 Tillstånd

Tillstånden som agenten baserar sina handlingar på kommer att begränsas till en vektor bestående av väsentlig information. Dessa vektorparametrar är en förenkling av den tillgängliga informationen i spelet.

3.3 Raser

Det finns tre raser i StarCraft 2 som alla har olika enheter och spelsätt. Vi har därför beslutat att både vår AI och motståndaren enbart spelar som rasen *Terran*. Främst innebär det att alla grundläggande handlingar blir enklare att programmera, och lika många undantagsfall behöver inte göras. Det sparar mycket arbete i förberedelser. En annan fördel med detta är att AI:n kan tränas mot sig själv. Att AI:n endast behöver lära sig att spela med en specifik matchuppställning istället för alla nio möjliga påskyndar också inlärningsprocessen.

3.4 Karta och koordinater

Då det finns en stor mängd olika kartor som alla påverkar spelet, och därmed inlärningsprocessen, används enbart kartan *Abyssal Reef*. En agent som kan spela på flera kartor skulle behöva inbyggda funktioner för att läsa av områden på ett annat sätt än vad som kommer behövas nu. I samband med val av karta kan faktumet att koordinater blir konstanta utnyttjas.

3.5 Spelläge

Vår AI kommer bara spela i ett 1-mot-1 läge. Att öka antalet spelare introducerar mer komplexitet då tillstånd och framgång beror ännu mindre på agentens handlingar. Agenten hade teoretiskt sett kunnat spela en perfekt omgång i spelet, men de andra spelarna gjorde ändå så den förlorar och ändå får en negativ belöning. Även motsatsen kan ske. Denna avgränsning gjordes därför för att undvika för många utomstående osäkerheter.

3.6 Fog of War

För underlätta träning av agenter är spelmekaniken *Fog of War* avstängd. Fog of War döljer vanligtvis de delar av kartan som spelaren inte har enheter vid. Med Fog of War på skulle agenten behöva lära sig skicka iväg enskilda enheter för att samla information om motståndaren. Agenten skulle även behöva minnas vad den observerat. Dock hade agenten ändå inte haft perfekt information om hela speltillståndet, vilket skulle kunna leda till dåliga beslut. Att implementera ett minne eller informationsinsamling skulle kräva en mer komplex lösning som vi inte utför.

Genom att stänga av *Fog of War* har AI:n perfekt information. Detta innebär att vår AI har tillgång till hela spelets tillstånd vid varje tidpunkt.

4

Utförande

Innan arbetet med att utveckla en AI med hjälp av maskininlärning påbörjades behövde en struktur för programmet skapas. När det var gjort inleds arbetet med maskininlärning.

4.1 Ramverk för interaktion med StarCraft 2

För att en AI ska kunna interagera med spelet behövdes först ett ramverk skapas. Ramverket låg som ett skelett till Markov-beslutsprocessen för spelet. I StarCraft 2 involverar det nuvarande tillståndet många aspekter. Det kan vara hur mycket resurser spelaren har eller antalet och placeringen på varje fiendeenhet. Därför finns det extremt många olika diskreta tillstånd.

Handlingarna som spelaren kan utföra är också extremt många. Varje klick på skärmen har en potential att förändra tillståndet markant. Dessutom kan tillståndet förändras utan att en viktig handling utförs. Därför är de nya tillstånden oförutsägbara, baserat på vad motståndaren gör.

Till slut definieras belöningarna i StarCraft 2 ofta baserat på hur mycket närmare vinst varje handling tog spelaren. Andra definitioner så som att belöna specifika handlingar kan skapa problem, eftersom agenten då istället kan lägga fokus på andra saker än vinst. På grund av detta valdes slutligen en belöning utefter vinst eller förlust.

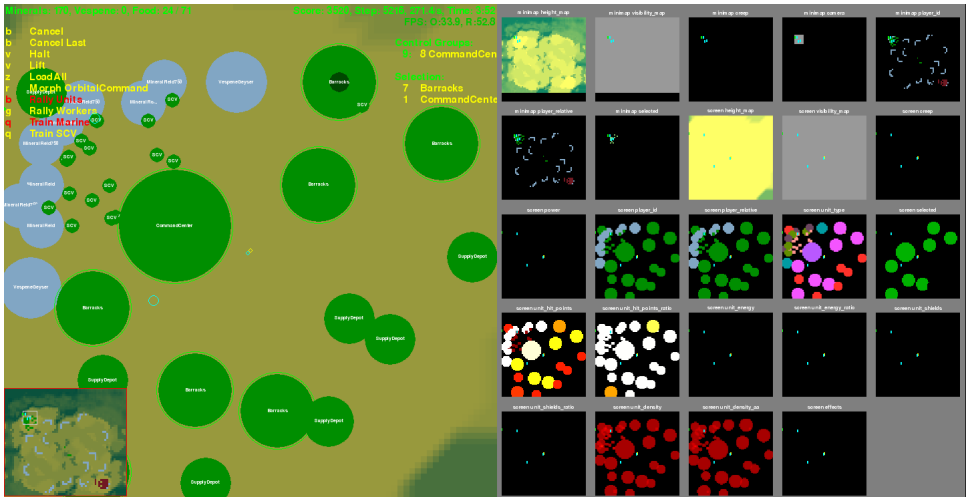
Den kod som framställts under projektet finns tillgänglig i Appendix B. Där beskrivs kodbasen både i skrift och grafiskt.

4.1.1 PySC2

För att skapa ramverket använder vi oss av Pythonbiblioteket PySC2. Detta bibliotek är utvecklat av DeepMind tillsammans med spelskaparna Blizzard. Biblioteket ger oss tillgång till en mängd funktioner för att bland annat hämta information och utföra handlingar i spelet.

Biblioteket, som är utvecklat i maskininlärningssyfte, ger oss även tillgång till en mängd

lågupplösta bilder av vad AI:n kan se, så kallade *feature layers*. Dessa ger information om exempelvis vilken spelare enheter tillhör, kartans utseende eller vad som sker där kameran är. Visualieringen utav dessa lager kan ses i figur 4.1.



Figur 4.1: Visualiseringen utav den information AI:n har tillgång till. På vänster sida är den nuvarande skärmpositionen och vad som genom den observeras. På höger sida ser man de olika ”feature layers” som finns tillgängliga, exempelvis ”minimap_player_relative” där man ser vilken spelare enheter tillhör eller ”screen_unit_hit_points” som ger information om enheters tillstånd i form av hälsa.

Interaktionen med spelet bygger på PySC2’s stegfunktion som med en parameter kan justeras hur ofta den anropas. Frekvensen av anrop begränsar hur ofta AI:n utför handlingar.

Varje anrop till stegfunktionen får tillbaka en handling att utföra. Det kan till exempel vara att flytta kameran, bygga eller markera en enhet. Denna stegfunktion används för att uppdatera tillståndet och utföra handlingar enligt Algoritm 2.

```
UpdatedState = False;
while True do
    if UpdatedState then
        välj handling och utför den (4 Steg, agent anrop);
        UpdatedState = False;
    else
        Uppdatera State (4 Steg);
        UpdatedState = True;
    end
end
```

Algoritm 2: En översikt över programloopen. Varje handling och tillståndsuppdatering tar 4 steg då det ramverk vi konstruerat kräver ett flertal steg per handling. Detta förklaras i kommande avsnitt.

4.1.2 Definiera tillstånd

Tillstånden som skulle matas in till agenten bestämdes till att vara data om antalet enheter bägge spelare har, mängden resurser i form av mineraler, gas och mat agenten har, tiden som har passerat sedan matchens början samt tidpunkten då agenten senast anföll tillsammans med dess dåvarande arméstorlek. Figur 4.1 innehåller en fullständig tabell över vilka maskininlärnings metoder som använt vilka tillståndsvariabler. Alternativet var att använda bilddata som PySC2 bidrar med, som också innehåller spatiell information, men på grund av begränsad tid och brist på datorresurser beslutades att inte ha med detta. Det hade potentiellt givit bättre resultat, men den utökade informationen hade antagligen lett till längre träningstider. Utöver detta användes två olika sätt att hämta tillstånd på, där den andra iterationen var mer effektiv men krävde modifikation av PySC2-biblioteket.

Den första iterationen krävde att AI:n gick igenom alla byggnader genom att flytta runt kameran till deras positioner och lade till dem i en kontrollgrupp. Det är en funktion i StarCraft som låter spelare återfå markering av en grupp enheter eller byggnader utan att manuellt behöva markera dem igen. Denna kontrollgrupp användes sedan för att bestämma vilka enheter och byggnader som tillhörde agenten i spelet. Denna implementation nämns då en del tester som kördes vid ett tidigt stadie använde sig utav denna.

En funktion för att räkna motståndarens enheter genom att använda kartan togs också fram. Den räknade ut hur den kunde använda så få skärmförflyttningar som möjligt för att se alla motståndarens enheter, då det krävdes. Den tog dock viktig tid från agentens handlingar, och kunde i vissa fall räkna fel. Därför ersattes det i den andra iterationen.

Den andra iterationen utvecklades då insikten av att den första implementationen skapar inkonsistenser i hur lång tid det tar att uppdatera tillståndet. Exempelvis varierade antalet platser kameran behövde flyttas till. För att agenten lättare ska kunna hitta mönster gjordes detta om genom att modifiera PySC2-biblioteket så att antalet enheter och byggnader kunde extraheras utan någon explicit handling. Byggnader läggs fortfarande till i kontrollgruppen då den uppfyller andra funktioner. Denna ändring gör att uppdateringen utav tillståndet tar konstant tid.

Tabell 4.1: Tillståndsparametrar och i vilken utsträckning de används under träning av olika maskininlärningsmetoder. IL står för Imitation Learning och BC för Behavioral Cloning.

Parameter	Q-learning	DQL	A2C	A2C-IL	BC
minerals	x	x	x	x	x
vespene gas	x		x	x	x
food_used		x	x	x	x
food_cap		x	x	x	x
idle_workers		x	x	x	x
steps		x	x	x	x
command_centers	x	x	x	x	x
supply_depots	x	x	x	x	x
refineries	x		x	x	x
barracks	x	x	x	x	x
factories			x	x	x
starports			x	x	x
scvs	x	x	x	x	x
marines		x	x	x	x
hellions			x	x	x
medivacs			x	x	x
reapers			x	x	x
vikings			x	x	x
enemy_command_centers					x
enemy_supply_depots					x
enemy_refineries					x
enemy_barracks					x
enemy_factories					x
enemy_starports					x
enemy_scvs					x
enemy_marines					x
enemy_hellions					x
enemy_medivacs					x
enemy_reapers					x
enemy_vikings					x
number_of_enemy_units			x	x	
number_of_enemy_buildings			x	x	
last_attacked			x	x	
attacking_units			x	x	

4.1.3 Skapa ett handlingsrum

Handlingsrummet då alla möjliga kombinationer av kamerarörelser eller enskilda markeringar är med är alldeles för stort. Därför skapas ett handligsrum som bygger på en

uppsättning av dessa mindre handlingar. Detta handlingsrum redovisas i tabell 4.2 tillsammans med vilka delar av det våra testade maskininlärningsmetoder använde. Om exempelvis en byggnad ska byggas blir hela sekvensen med att markera en arbetare, flytta kameran till basen, välja vilken byggnad som ska placeras och sedan var den ska placeras en enda handling. Handlingen *expand* bygger dock en ny bas på förutbestämda koordinater. Handlingen *return_scv* tar en ledig arbetare och beordrar denne att hämta resurser igen (något som behövs efter att de byggt en byggnad) och *distribute_scv* distribuerar arbetarna till alla lediga resursnoder. *attack* markerar alla egna trupper och beordrar dem att anfalla den närmaste fiendeenheten på kartan medan *retreat* samlar alla egna trupper vid en hårdprogrammerad koordinat i mitten av kartan. Slutligen gör *wait* ingenting, något som behövs ibland. Alla dessa handlingar var programmerade att ta lika lång tid att utföra, inklusive *wait*, vilket också innebar att vissa handlingar även innehöll kortare *wait*-handlingar.

Byggnadsplaceringen slumpades eftersom agenten inte får någon spatiell information i tillståndet. Detta innebär bland annat att vissa enheter inte kunde byggas eftersom de krävde byggnader med specifik placering. Vissa andra enheter krävde spatiell information för att kunna användas överhuvudtaget (exempelvis välja ett litet område att belägra) och implementerades således inte i handlingsrummet.

Tabell 4.2: En översikt på de handlingar som definierats och i vilken utsträckning de använts under träning eller datainsamling. IL står för Imitation Learning och BC för Behavioral Cloning.

Handling	Q-learning	DQL	A2C	A2C-IL	BC
wait	x	x	x	x	x
build_scv	x	x	x	x	x
build_supply_depot	x	x	x	x	x
build_barracks	x	x	x	x	x
build_marine	x	x	x	x	x
build_refinery	x				x
expand					x
build_factory					x
build_starport					x
build_reaper					x
build_hellion					x
build_medivac					x
build_viking					x
return_scv	x	x	x	x	x
attack		x	x	x	x
retreat					x

4.2 Test av olika maskininlärningsmetoder

Flera olika maskininlärningsmetoder testades parallellt. Dels testades Q-learning, behavioral cloning och olika deep reinforcement learning-metoder. I detta avsnitt diskuteras de metoder som provats men som övergivits på grund av olika anledningar. För de mest aktuella och framgångsrika metoderna hänvisas läsaren till avsnitt 4.2.3 och 4.2.4

4.2.1 Q-learning

Den första maskininlärningsmodellen som testades var Q-learning. För att snabbt försöka få insikt av potentialen av Q-learning kördes tester på de första minuterna där agenten försökte uppnå ett mål av att bygga en viss mängd enheter och samla en viss mängd resurser.

Trots ett mycket förenklat tillstånd av bara ett par variabler samt användning av enbart de simplaste handlingarna blev Q-tabellen för stor. Storleken av tabellen innebar att för att kunna bestämma om det ger resultat skulle träningen ta för lång tid. Tillståndets variabler kan hittas i tabell 4.1 och möjliga handlingar i tabell 4.2.

Då problem med antalet unika tillstånd identifierades gick arbetet vidare till Deep-Q-Learning. Istället för att ha vissa värden för varje tillstånd approximerar det vad som borde vara bäst vid ett givet tillstånd.

4.2.2 DQL

Då Q-learning inte gav några resultat då tabellen blev för stor provades DQL eftersom den, istället för att slå upp vilken handling som är bäst i en tabell, approximerar vilken handling som är bäst givet ett tillstånd. De möjliga handlingarna hittas i tabell 4.2.

Då antalet unika tillstånd inte längre är ett problem kördes agenten över hela matcher och tillståndet utökades till fler variabler. De kan hittas i tabell 4.1. Det verkade som att den lärde sig något men det slutade med att den lärde sig att enbart utföra en handling. Då det verkade som att träningen var instabil gick arbetet vidare till A2C där träningen stabiliseras.

4.2.3 Advantage Actor Critic (A2C)

Som sista reinforcement learning-metod prövades algoritmen *Advantage Actor Critic* eller *A2C*. Tillstånden som användes under träningen hittas i tabell 4.1 och de möjliga handlingarna i tabell 4.2.

Genom att låta critic-nätverket uppskatta tillståndsvärdefunktionen $V_\theta(s)$ kunde denna

användas denna till att i sin tur uppskatta $A_\theta(s,a)$ enligt

$$A_\theta(s,a) = r(s,a) + \gamma V_\theta(s') - V_\theta(s)$$

Agenten spelade mot den inbyggda AI:n och tränades med en så kallad experience replay, där tillstånd, handlingar, belöningar och nästa tillstånd sparades i en replay-buffer.

Optimering av nätverkets parametrar gjordes till en början med vanlig gradient ascent, men i ett försök att snabba upp träningstiden prövades även optimeringsalgoritmen *RMSprop*. För att undvika för stora uppdateringar av actor-nätverket implementerades *gradientklippning* vilket begränsade gradientens storlek till intervallet [-1,1].

För att förbättra agentens tendens till att utforska omgivningen introducerades även en viktad entropiterm i actorns målfunktion. Två sätt att vikta denna prövades, ett där den hölls på en konstant låg nivå under hela träningsförloppet och ett där den började högt och successivt minskades till en låg nivå.

Målfunktionen utökades även med en imitationsterm för att hjälpa agenten hitta en bra första policy, i vad vi kallar för en *A2C-IL-hybrid*. Denna implementation kombinerar A2C med en interaktiv expertpolicy π^* bestående av en uppsättning regler som valde nästa handling enligt en rad kriterier. Expertpolicy kunde vinna mot den svåra motståndar-AI:n. Imitationstermens vikt minskade ju fler träningsmatcher agenten spelar.

4.2.4 Behavioral cloning

Förutom olika reinforcement learning-metoder testades även ren imitation learning i form av behavioral cloning. För att kunna göra detta krävdes data i form av par av tillstånd och tillhörande korrekt handling. Eftersom tillstånds- och handlingsrummen var väldigt begränsade jämfört med de riktiga tillstånds- och handlingsrummen kunde inte data från riktiga spelare användas; det blev svårt att översätta riktiga spelares tillstånd och handlingar till data agenten kunde hantera. För att lösa detta problemet användes data från matcher där en slumppolicy mötte en inbyggd AI. Tillståndsvariablerna som sparades finns i tabell 4.1 och de olika handlingarna agenten kunde lära sig finns i tabell 4.2.

Då slumpmässigt val av handling inte var tillräckligt för att vinna mot den inbyggda lätta motståndaren, viktades sannolikheten för slumppolicyn att utföra en viss handling beroende på vilken minut i spelet som pågick. Med de viktade sannolikheterna för handlingarna spelades över 2000 matcher. Från de ca 1400 vinsterna sparades data i form av par av tillstånd och handling.

Vissa handlingar valdes, på grund av viktningen, oftare än andra handlingar. För att få balanserad indata slängdes således överflödiga tillstånds- och handlingspar slumpmässigt så att det fanns lika mycket data, 2500 tillstånd, för varje handling med undantag för *retreat*. Då implementationen av retreat-handlingen visade sig vara problematisk togs ingen data för retreat med i indatan till neuronnätet.

För att underlätta träningen av neuronnätet normerades tillstånden innan datan användes.

Normeringen gjordes genom att sätta ett maxvärde för varje parameter. Sedan dividerades parametervärdet med maxvärdet för att få ett tal mellan 0 och 1 för varje parameter. Handlingen för det givna tillståndet beskrevs av en vektor av längd 16 med nollor på varje index förutom en etta på det indexet som motsvarande den utförda handlingen. Detta kallas för *one-hot encoding*.

4.2.5 Byggnadsplacering

I StarCraft 2 spelar det roll hur spelaren har byggt sin infrastruktur. En dålig placerad byggnad kan exempelvis leda till att resursinsamligen blir långsammare. Dessutom kan en strategisk placering av byggnader användas för att försvara sig mot fiendens armé. Därför inleddes arbete med att försöka förbättra den befintliga, slumpmässiga utplaceringsmetoden. Detta görs genom att utveckla en reinforcement learning agent vars mål är att hitta bästa plats för utplacering av byggnader.

För att finna vilken plats agenten ska bygga på observeras skärmen och identifierar vart nuvarande byggnader finns. Dessa sparas som 1:or och 0:or där byggnader har värdet 0. Från detta kan agenten skapa en lista av listor där den kan bestämma den största ytan den kan placera nästa byggnad på. Detta är med hjälp av en algoritm som räknar ut den största arean av de andra areorna som finns i det nya uppgjorda miljön.

```
Result: Hämtar största värdet av listan
initialisering av Listan och andra värden;
max Area = 0, area = 0, temporär y =0, minimumsize = 9;
while Till slutet av listan do
    Beräkna ut det nya värdet för listan;
    Hämtar position för max värdet i listan;
    for x i Range(minimum_size, maxvrdet) : do
        Sätter Längden på nuvarande lista och längden -> 0;
        for y i Kolumn do
            if Kolla för det största värdet then
                nuvarande Längden =+ 1
            else
                nuvarande Längden = 0
            end
            if nuvarande Längden > Längden then
                längden = nuvarande längden;
                temporära y värdet = y
            end
            area = längden* x;
            if area >= max_area then
                Sätter värdena för listan
            end
            if Något av värdena inte lista är noll then
                Lista[i] = värdet av arean
            else
                return Lista
            end
        end
    end
    Sätt värdena där man kan bygga till noll;
    Kollar om all värdena är noll
end
return Lista
```

Algoritm 3: Hittar alla byggareor från skärmen.

Algoritmen 3 hämtar alla positioner agenten kan bygga på. Denna algoritm löser ut den största arean genom att bygga upp ett histogram av areor i miljön. Belöning för vald plats ges beroende på tillståndet av arenan. Exempelvis om det redan finns en byggnad eller enhet där så straffas den medans om platsen är fri får den en belöning. Med detta ska agenten lära sig vart det är bra eller dåligt att placera byggnader. Den försöker hitta alla positioner som är störst i listan med hjälp av att reducera ner den den redan har hittat med hjälp av 0, sedan så försöker man hitta näst största förbygg arean , tillslut så har man då hittat alla areor.

5

Resultat

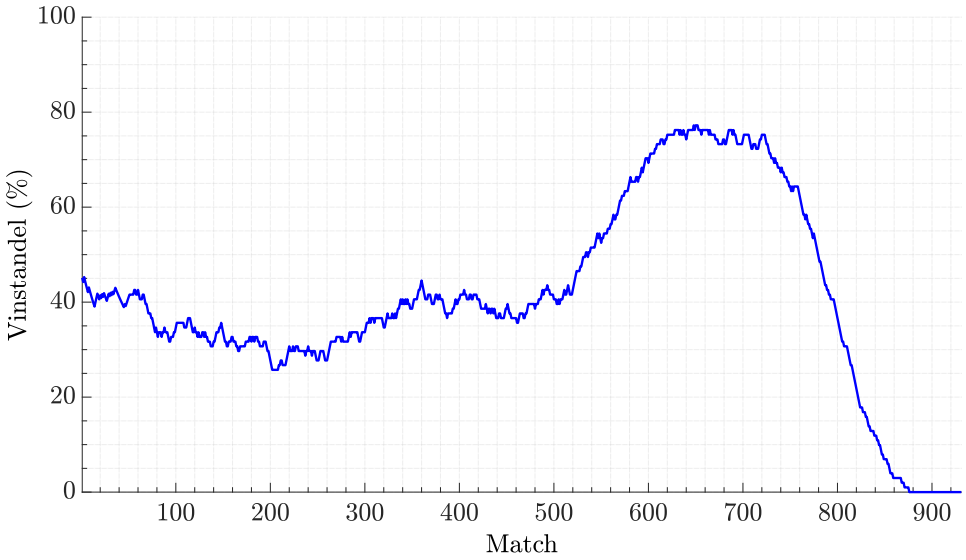
I denna del presenteras resultat för tre agenter som spelar spelet och byggnadsplaceringsagenten. De två första agenterna baserades på A2C där den andra var A2C-IL-hybriden. Den tredje var baserad på ren imitation learning. A2C-IL-hybriden resulterade i den högsta vinstandelen på 73,4 % mot den svåra inbyggda AI:n medan imitation learning-agenten enbart uppnådde en vinstandel på 17 %. Den grundläggande A2C-agenten kunde enbart vinna mot den mycket lätta AI:n. Byggnadsplaceringen lyckades delvis men hann inte färdigställas.

5.1 Advantage Actor Critic (A2C)

De två A2C-implementationerna som testades var en med imitation learning och en utan. Bägge använde tre gömda lager med 32, 32 respektive 16 noder.

5.1.1 Endast A2C

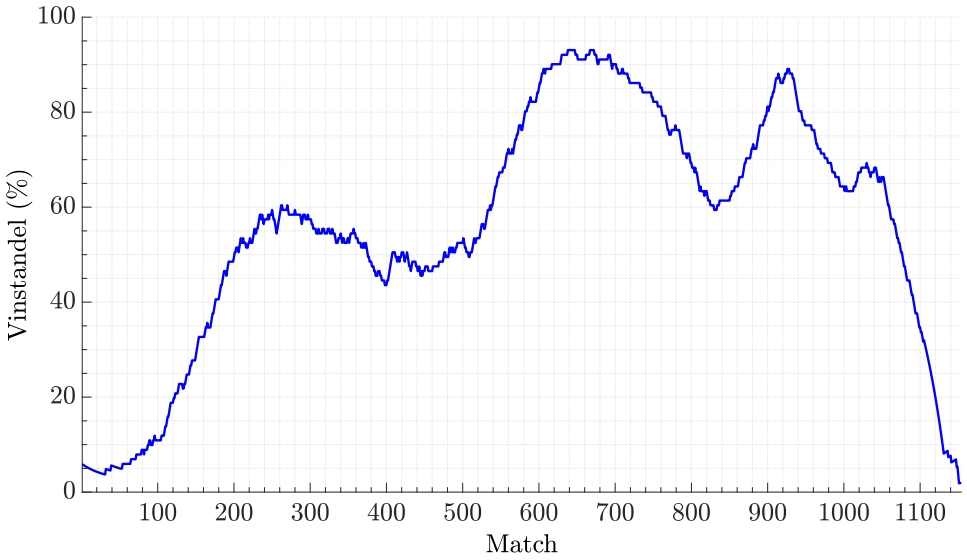
A2C-agenten tränades mot svårighetsgraden väldigt lätt och spelade totalt 931 matcher med en replay-buffer som kunde lagra 500 000 gamla erfarenheter. Exakt hur vinstandelen förändrades med antalet träningsmatcher visas i figur 5.1. Initialt kunde den vinna med ren slump innan den efter ungefär 650 matcher lärde sig att vinna nästan 80 % av matcherna, tills den i slutändan förlorade alla matcher.



Figur 5.1: En A2C-agents vinstandel visualiserat som det glidande medelvärdet av de 50 föregående och 50 följande matcherna. Den spelade mot den inbyggda ”väldigt lätt”-AI:n och använde sig av både policyförlust och entropiförlust för inlärning.

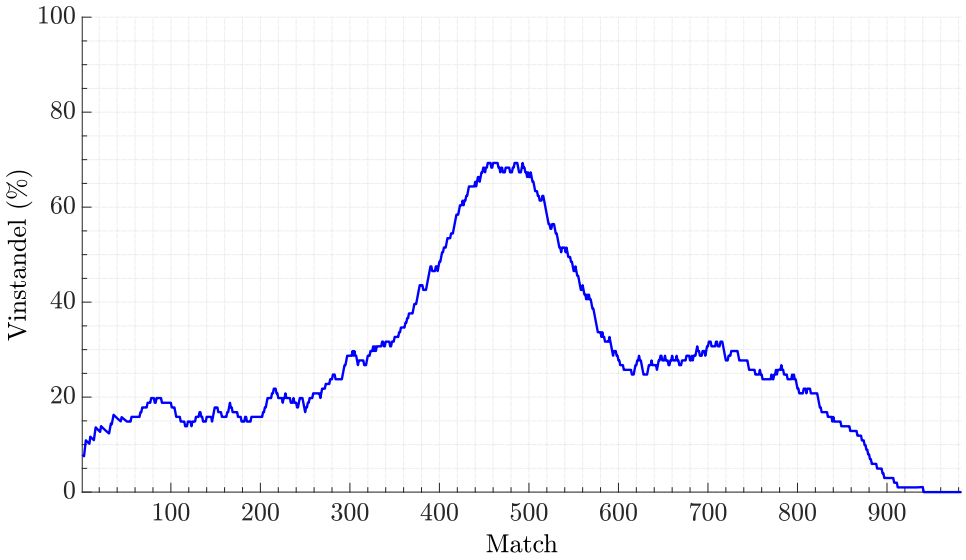
5.1.2 A2C-IL-hybrid

A2C-IL-hybriden, som kombinerade A2C med en interaktiv expertpolicy, gav blandade resultat. Generellt sett blev träningsprocessen alltid instabil i längden när agenten från början lärt sig vinna för att sedan endast kunna förlora. Ett exempel på detta visas i figur 5.2, där en agent som tränades mot den inbyggda medium-AI:n hade lärt sig att vinna väldigt ofta innan den efter 1100 träningsmatcher endast förlorade. Förhållandet mellan imitationsförlusten och policyförlusten minskades för varje träningsmatch tills imitationsförlusten efter 1000 matcher var obetydlig. Replay-buffern hade maxstorlek på 100 000 tillstånd vilket motsvarar ungefär 200 matcher.



Figur 5.2: En A2C-IL-hybridagents vinstandel visualiserat som det glidande medelvärdet av de 50 föregående och 50 följande matcherna. Den spelade mot den inbyggda medium-AI:n och använde sig av en A2C-algoritm tillsammans med imitation learning.

Neuronnätets vikter sparades när agenten var som bäst, vilket var när den hade tränats i 700 matcher, och användes som bas för att träna en ny agent mot svårighetsgraden medium-svår med förhållandet mellan imitations- och policyförlusten återställd. Med denna iterativa process tränades sedan en agent mot den svåra svårighetsgraden, vars vinstandel över matcher illustreras i figur 5.3. När den var som bäst kunde den vinna 369 av 501 matcher mot svår svårighetsgrad, vilket motsvarar en vinstandel på 73,4 %, men träningen visade sig i slutändan ändå vara instabil.



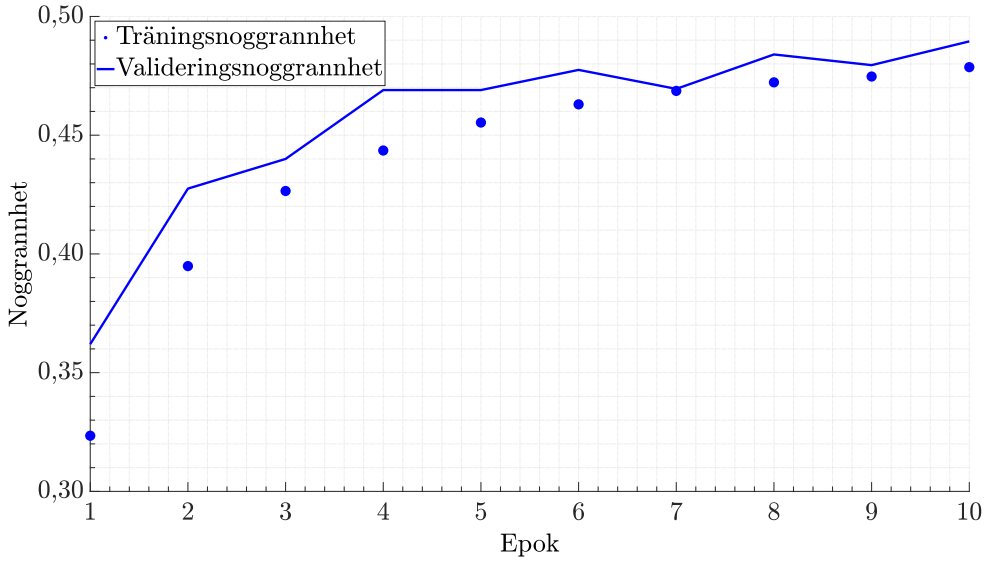
Figur 5.3: A2C-IL-hybridagents vinstandel visualiserat som det glidande medelvärdet av de 50 föregående och 50 följande matcherna mot en inbyggd AI med svår svårighetsgrad. Agenten baserades på andra agenter som var tränade mot lättare inbyggda AI:n som bas.

5.2 Behavioral cloning

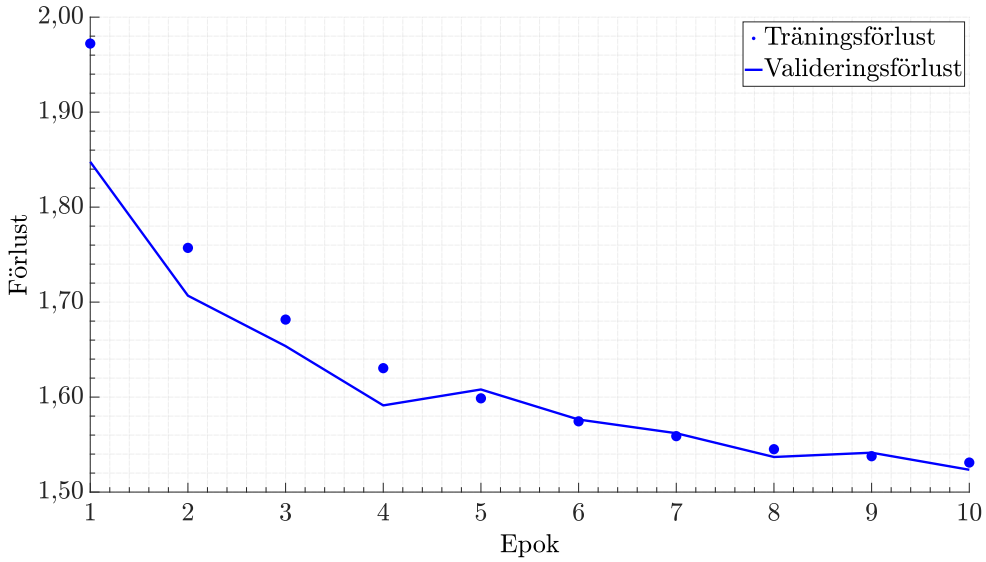
Det artificiella neuronnätet som tränades med behavioral cloning lyckades som bäst slå den inbyggda svåra motståndaren ca 17 % av matcherna den spelade. Mot den lätta motståndaren vann modellen ca 95 % av matcherna. Notera att nätverket tränades på data tagen från vinster mot den lätta motståndaren.

I Figur 5.4 och 5.5 visas noggrannhet respektive förlust för nätverket under träningen när nätverket bestod av tre sekvensiella fullt sammankopplade lager med 128, 32 respektive 16 noder var. Den optimeringsalgoritm som användes var Adam med en inlärningshastighet på 10^{-3} och avtagande på 10^{-6}. Nätverket tränades på 10 epoker med en *batch size* på ett. Epoker är antalet gånger nätverket tränas på datan och batch size är antalet tillstånd-handlingspar som matas in åt gången innan nätverket uppdateras.

z



Figur 5.4: Noggrannhet som funktion av epoker under träningen av nätverket med behavioral cloning. Noggrannhet är hur ofta modellen väljer rätt handling. Notera att ca 6 % noggrannhet är förväntat om nätverket valde handlingar helt slumpmässigt.



Figur 5.5: Förlust som funktion av epoker under träningen av nätverket med behavioral cloning. Förlust är en summa av logaritmerat fel (cross entropy) och beskriver hur väl modellen är anpassad på indatan. En förlust av 0 är optimalt. Det intressanta är inte det exakta värdet på förlusten utan skillnaden mellan träningsförlusten och valideringsförlusten. Är valideringsförlusten påtagligt större än träningsförlusten har antagligen så kallad overfittning skett. Overfittning innebär modellen har tränats för mycket på samma data så att den blir sämre på att känna igen generella mönster och bara väljer rätt handling då den känner igen indatan.

5.3 Byggnadsplacering

Det andra reinforcement learning-området som arbetades på är byggnadsplaceringen. Här har ytterligare en actor-critic-modell utnyttjats för att bestämma hur byggnader ska placeras. Belöningen som användes av agenten var baserad på hur stor yta som valts och möjligheten att bygga på den. Resultatet blev att den lyckades utnyttja byggytan bättre, exempelvis kan byggnader med säkerhet placeras nära varandra. Det blev även en förminskning av byggnader på oönskade platser, exempelvis bredvid mineralfält.

Tyvärr har byggnadsplacerings-AI:n inte undersökts tillsammans med de andra agenterna, vars mål är att vinna matcher, då tiden ej räckte till. Därför har inget resultat av denna AI:s inverkan på matchresultat kunnat tas fram. En anledning till tidsbristen var att framställningen av omgivningen agenten arbetar mot tog längre tid än förväntat. Detta då algoritmen som framställer omgivningen behövde göras om då en första implementation var för långsam och i samband med frekvensen av anrop till den saktades spelet ner till en oacceptabel nivå.

6

Diskussion

Diskussionen kommer inledas med separat analys av de olika metoderna. Därefter kommer en jämförelse mellan dem där för- och nackdelar diskuteras.

6.1 A2C-implementationer

De olika implementationerna av A2C som testades, en ren A2C och en A2C-IL-hybrid, gav väldigt olika resultat. Den väsentliga skillnaden var att renodlad A2C endast kunde vinna mot svårighetsgraden mycket lätt. Tvärtemot detta kunde A2C-IL-hybriden efter kort träning redan vinna mot medium-AI:n och med kontrollerade träningsregimer även vinna mot svåra AI-motståndare. I slutändan började även denna agenten förlora utan återhämtning inom de hundratals matcherna som spelades.

6.1.1 Orsaker till förlusterna

A2C-agenten lyckades aldrig bibehålla sin vinstandel ens mot mycket lätt medan A2C-IL-hybriden kunde upprätthålla en hög vinstandel mot svårare motståndare så länge imitationstermen var betydande. Detta tyder på att det var criticen som inte lyckades lära sig uppskatta värdefunktionen bra. I hybridagenten var imitationstermen högt viktad i början och avtog sakta när antalet träningsmatcher ökade. Att den till en början lyckades lära sig vinna mot svårare motståndare kan därför tillskrivas imitationsinlärningen, medan förlusterna som började uppstå senare i träningsprocessen orsakades av criticen.

Criticens funktion i algoritmen är att vara en skattning av värdefunktionen som i sin tur påverkar målfunktionen. En otränad eller felaktig critic leder därför till att actorns policy uppdateras på fel sätt när exempelvis gradient descent-optimering utförs. Det är denna felaktiga uppdatering som vi tror ledde till att A2C inte fungerade och att hybridagenten i slutändan slutade vinna.

6.1.2 Att förbättra agenterna

Problemen med de A2C-implementationerna som testades hade lösts om criticen lärde sig uppskatta värdefunktionen fortare. En uppenbar förbättring är att införa *n-step TD learning* som beskrevs i delkapitel 2.2.5. Det skulle sannolikt ge snabbare konvergering av criticens uppskattade värdefunktionen till den verkliga värdefunktionen och därmed leda till en policy som bättre maximerar väntevärdet av den framtida belöningen G_t. Antalet steg n som TD learning görs för blir då ytterligare en parameter som behöver undersökas för att hitta ett optimalt värde.

Vidare skulle A2C-algoritmen kunna utökas från att använda en enda agent till flera agenter som vardera utforskar olika oberoende omgivningar. De skulle alltså spela olika matcher av StarCraft 2. Det optimala antalet agenter är då sannolikt det antal processorkärnor som den dator träningen sker på har, eftersom varje agent då kan köras parallellt. Eftersom neuronnätets parametrar då uppdateras utefter flera olika oberoende omgivningar fyller detta delvis samma syfte som replay-buffern tidigare gjorde, och det borde förslagsvis därför undersökas om denna helt kan plockas bort utan påverkan på resultatet.

Under utvecklingen av A2C-agenterna hade också en rad olika parameterinställningar testats, vilket inkluderade bland annat olika inlärningshastigheter, storleken på neuronnätet, vilka handlingar och tillståndsvariabler som skulle användas, hur stor replay-buffern skulle vara samt hur de olika termerna i förlustfunktionen skulle viktas. Även olika belöningar testades. Justering av dessa skulle också kunna bidra till ökad prestanda för agenterna.

- **Inlärningshastigheten** påverkar hur mycket neuronnätet uppdaterar sina vikter varje träningsiteration. Actorns och criticens inlärningshastighet kan ändras oberoende av varandra, vilket kan vara till fördel om criticen verkar lära sig långsamt; om actorns inlärningshastighet minskas kan det ge mer tid åt criticen att lära sig skatta värdefunktionen. Alternativet vore att öka criticens inlärningshastighet, men om den är för stor finns det risk att criticen inte hittar ett minimum när den minskar sin egna förlustfunktion. Hur dessa två inlärningshastigheter ska förhålla till varandra kan därför experimenteras mycket med.

- **Neuronnätstorlek** i form av antal lager och antal noder undersöktes också, där antalet gömda lager var mellan två och tre och antalet noder per lager varierades mellan 200 och 16. Målet var att försöka ha ett neuronnät som har få parametrar att träna men ändå har potentialen att vara en bra policy. Mycket i denna iterativa process var bara spekulationer och i slutändan verkade det inte ha bidragit med någon märkbar förändring på inlärningen.

- **Handlings- och tillståndsrummen** begränsades i hopp om att träningen skulle gå fortare i utbyte mot att AI:ns potential begränsades, då vi antog att det krävdes mer komplexa tillstånd och handlingar för att kunna vinna mot de svåraste AI-motståndarna.

- **Belöningar** var till en början tänkt att vara baserat på det inbyggda poängsystemet. Detta system belönar och ger avdrag beroende på händelser under matchen. Pro-

blem som uppstod med detta var att agenten kunde nöja sig med att ackumulera poäng av saker som inte leder till vinst. Därför övergick vi till att enbart belöna och straffa vinster och förluster vilket garanterar att agenten strävar efter vinst. Ett problem med detta är att när belöningarna är så glesa kan det ta tid innan agenten uppnår en vinst för första gången och därmed inte vet vad som ska strävas efter.

- **Replay-buffern** innehåller gamla val av handlingar enligt dåvarande policy π'. Problematiken som kan uppstå är att criticen, som i idealfallet ska lära sig uppskatta den totala framtida belöningen under policyn π, istället lär sig uppskatta den totala framtida belöningen under π'. I extremfallet kanske π' är en väldigt gammal och suboptimal policy som gör att criticen tror att en viss handling a givet tillstånd s är dålig, när den nuvarande policyn π egentligen oftast får en vinnande bana om den väljer a givet s. Förutom minnesproblem kommer för stora replay-buffrar därför i extremfall kunna leda till en diskrepans mellan criticen och actorn. Å andra sidan kommer för små buffrar kunna leda till sämre inlärning om exempelvis vinster sker sällan och försvinner ur buffern innan agenten lär sig från dem.

- **Förlustfunktionen** bestod av tre viktade termer: policyförlusten (som beror på criticen), entropiförlusten samt imitationsförlusten, där den sistnämnda endast var aktuell för A2C-IL-hybriden. Kvalitativt bestämmer entropivikten hur mycket agenten kommer utforska i förhoppning att det leder till en bättre policy medan imitationsvikten bestämmer hur mycket policyn ska härma expertpolicyn. Vid inspektion av policyerna efter lång träningstid, vilket innebar att endast policyförlusten och entropiförlusten är kvar, verkade det som att de började utföra endast ett par olika handlingar oavsett tillstånd. Då vinster i spelet kräver mer komplexa sekvenser av handlingar ledde detta till att agenterna förlorade. Genom att öka entropivikten skulle dessa dåliga beteenden kunna motverkas till en viss grad; om criticen ändå vill att policyn fixerar sig på ett par handlingar kommer entropiförlusten endast bromsa ner konvergensen. Detta löser därför inte på det underliggande problemet med criticen.

6.1.3 Förbättringars inverkan på tidiga tester

Många förbättringar, så som inkludering av entropi-term med mera, implementerades under utvecklingen av A2C agenten. Denna utveckling inträffade därför efter att vi lämnat Q-learning och DQL. Det är därför fullt möjligt att bättre resultat från speciellt DQL skulle kunna uppnås genom att utnyttja dessa förbättringar även där.

Utöver dessa förbättringar använde sig både Q-learning och DQL av den första iterationen av tillståndet. Detta är något som potentiellt begränsat deras förmåga att identifiera mönster i datan då intervallen mellan handlingarna varierade.

6.2 Behavioral cloning

Det neuronnät som tränades med behavioral cloning nådde upp till målet att kunna vinna mot den inbyggda svåra motståndaren. Dock är vinst 17 % av matcherna naturligtvis inte optimalt. Eftersom nätverket tränades med data tagen från vinster, som uppnåtts med en sub-optimal expertpolicy i from av en viktad slumppolicy, mot den lätta motståndaren är det dock inte rimligt att den skulle vinna varje match. Om vi hade haft mer tid eller kraftfullare hårdvara hade vi kunnat fortsätta genom att generera data från vinster mot den svåra motståndaren och efter det vinster mot nätverket själv. Förmodligen hade det lett till en förbättring då nätverket har visats kunna lära sig att imitera den data den tränas på.

Det nämndes i kapitel 4.2.4 att retreat-handlingen var med i handlingsrummet när träningsdatan genererades men inte användes när vi tränade nätverket. Då retreat handlingen i sig inte ändrade på tillståndet, utifrån de parametrar vi valde att definera tillståndet med, uppstod det inget problem med att inte ta med den. Resultatet var bara att modellen aldrig valde retreat eftersom den aldrig tränades på någon indata där retreat var rätt val.

Som kunde förväntas fungerade behavioral cloning inte speciellt bra i situationer som den inte var tränad på. Då modellen endast var tränad på vinster av relativt lik karaktär så hamnade den ibland i tillstånd som den inte kände igen alls. Till exempel när modellen höll på att börja förlora slutade den att ta bra beslut och började välja en och samma handling. Detta kan jämföras med att gång på gång fatta beslutet att flytta en bonde i schack. Ibland och under begränsade tider kan det vara rätt beslut, men görs det om och om igen kommer det till slut inte gå att göra längre och matchen kommer förloras.

En annan nackdel med behavioral cloning, och imitation learning i allmänhet, är att nätverket kan aldrig bli bättre än den data som den tränas på. Detta problemet har man inte med reinforcement learning där det inte finns någon teoretisk övre gräns på hur bra nätverket kan bli.

6.3 Byggnadsplacering

Vi kan se att målet som den nuvarande byggnadsplacerings agenten har, har uppnåtts. Det skulle därför vara intressant att se dess inflytande på matchers utfall då vi skulle slippa vissa inkonsekvenser av exempelvis mineralackumulering eller trupprörelser. Detta då byggnader placerades på platser som förhindrade optimal insamling av mineraler eller blockerade avgörande korridorer som får trupper att gå en och en eller fastna.

Ytterligare förbättringar skulle kunna nås i vidareutveckling av denna agent. Exempelvis kan man utöka dess miljö och få den att identifiera andra potentiella blockeringar så som klippor. Man skulle även kunna ge den feedback på om byggnaden den placerat faktiskt påbörjats då oförutsägbara händelser kan ske från det att placeringen beordras till det att byggnaden påbörjas. Exempelvis kan det ske att någon enhet har rört sig till ytan blockerar

den efter det att byggnaden placerats.

Dessa förbättringar är saker som troligtvis skulle hjälpa de övriga agenterna identifiera mönster i sin data eftersom men då med säkerhet kan säga att en byggnad som beordrats faktiskt lyckades placeras och påbörjas. I dagsläget kan det exempelvis ske att en begäran att bygga en byggnad inte resulterar i att den påbörjas då de slumpade koordinater som tagits fram inte är en giltig plats för byggnaden. Detta skapar slump i vad en handling faktiskt åstadkommer vilket inte är önskvärt.

6.4 Jämförelse mellan reinforcement och imitation learning

Eftersom vi har undersökt olika maskininlärningskoncept är det intressant att se hur de skiljer sig åt i prestanda och träningstid. Det mått som är intressant att jämföra mellan metoderna när det gäller prestanda är huvudsakligen vinstandel. Andra mått så som noggrannhet i behavioral cloning har ingen direkt motsvarighet i våra implementationer av A2C och säger därför inget relevant.

6.4.1 Prestanda

Det mått som framförallt är intressant att jämföra mellan A2C-agenterna och behavioral cloning-agenten är andelen vinster som uppnås. AI:n som tränats via behavioral cloning uppnåde en vinstandel på över 90 % på mot den svårighetsgrad av motstånd den tränats mot, vilket var svårighetsgraden lätt. A2C-agenterna, framför allt IL-hybriden, lyckades periodvis uppnå liknande vinstandelar mot både lätt, medium, och svår. Mot svår, som är den högsta svårighetsgraden den tränades mot, lyckades den uppnå en vinstandel på cirka 73 %. Skiftet av svårighetsgrad påverkar inte reinforcement learning lika mycket som behavioral cloning; reinforcement learning lär sig kontinuerligt att överkomma sitt problem medan behavioral cloning-AI:n är baserad på att data framställts som den kan träna mot och kan därför inte anpassa sig under speltid. Den behavioral cloning-AI som framställts hade trots detta en 17 % vinstandel mot den svåra AI:n vilket var den AI som vi hade som mål att besegra.

6.4.2 Träningshastighet

Det största hindret för båda varianterna av inlärningen har varit insamlingen av data eller genereringen av spelet.

A2C och A2C-IL-hybriden har varit beroende av att vid varje träningsförsök generera nya matcher vilket gör att ändringar av parametrar eller förändringar av spelomgivningen är

en förödande tidskostnad. Detta eftersom datagenereringen måste startas om i samband med en ny träningssession.

Behavioral cloning har istället varit beroende av redan insamlad data vilket tar lika lång tid som för A2C-varianterna. Fördelen är dock att vi kan ändra sättet vi behandlar datan på eller ändra inlärningsparametrar utan att på nytt behöva generera data. Detta gör att förändringar av parametrar och jämförelse mellan dem är mycket lättare. Träningen av nätverket är inte det som är tidsödande, utan det är datagenereringen som för A2C-varianterna inte är går snabbt att genomföra.

Dock behöver behavioral cloning nya dataset vid förändring av exempelvis svårighetsgrad vilket, beroende på den önskade storleken av det, kan ta lång tid att generera. Under träningen av vår imitation AI dedikerades inte så mycket resurser åt datagenereringen och datamängden var på gräns till för liten vilket riskerar att resultera i overfitting.

6.5 Slutsats

De agenter som använde sig av imitation learning, både behavioral cloning och interaktiv expertpolicy, presterade bättre än den som enbart använde A2C. Detta är troligtvis på grund av en otillräcklig A2C-implementering som resulterade i en undermålig critic. Vi lyckades dock på två sätt att med imitation learning uppnå vinster mot den inbyggda svåra AI:n där den bästa versionen (A2C-IL-hybriden) lyckas vinna 369 av 501 matcher och därmed uppnås vårt mål.

7

Bibliografiska anteckningar

För att få historik kring datorer och processorkraft går det att läsa Moores ursprungliga artikel (Moore 1965) som legat till bas för mycket datorvetenskap. Mer information om modern processorkraft i datorer går också att finna på Exper Exchanges hemsida (Expert Exchange 2019).

D'souzas artikel (D'souza 2018) och Bhatias artikel (Bhatia 2017) jämför bägge symbolisk AI med icke-symbolisk AI; D'souza ger en lekmannabeskrivning av båda grenarna. Bhatia diskuterar dem mer djupgående och även med en kort historisk inblick. Relaterat till grundläggande teori om artificiell intelligens är Markovbeslutprocesser som kan läsas mer om på Wikipedia (Wikipedia 2019[b]). Stanford har också en hemsida med tydliga frågor och svar på ämnet (McCarthy 2007).

Både Andersons artikel (Anderson 2017) och IBM:s artikel (IBM 2011) ger en tillbakablick till den historiska schack-matchen mellan Kasparov och Deep Blue. Anderson ger ett mer historisk perspektiv kring schack-AI samt diskuterar hur tekniken har tillämpats i andra områden. IBM ger en kortare sammanfattning av IBM:s arbete med och påverkan på datorberäkning.

Gällande spelet go och artificiell intelligens framsteg beskrevs arbetet i en artikel från DeepMind (Silver, Huang m. fl. 2016) samt i en kortare text från densamma (Silver och Hassabis 2016). Efter att dessa var skrivna åkte de till Seoul för att besegra världsmästaren. Det kan läsas mer om i Hassabis blogginlägg (Hassabis 2016).

För att lära sig mer om imitation learning finns det en hel del teori från ett blogginlägg av Hui (Hui 2019b). Föredrar man istället en tutorial går det istället att titta på en som kommer ifrån Yue och Le (Yue och Le 2018). Den togs fram i förberedelse för en maskininlärningskonferens.

Några artiklar som hanterar reinforcement learning och erbjuder lättläsliga och simpla introduktioner till detta område är Jonathan Hui i första delen av sin serie om reinforcement learning (Hui 2019a) och Tambet Matiisen i sin artikel hos Intel (Matiisen 2019). Matiisen lägger även grunden för några mer avancerade koncept. Hui går i sin serie in i mer djup inom de olika koncepten vilket ger upphov till en bra basförståelse om området. En bok Hui haft som grund för sitt skrivande är boken (Sutton och Barto 2018) där Richard Sutton och Andrew Barto ger en mer ingående beskrivning. Några andra böcker som reflekterar innehållet i de tidigare källorna är boken (Chollet 2018) skriven av François Chollet samt

boken (Russel och Norvig 2009) skriven av Stuart Russel och Peter Norvig.

Chollets bok har även en explicit introduktion till neuronnät ett ämne som även Russel och Norvig tar upp. Men om man vill ha en mjukare introduktion till detta ämne finns ett par anonyma video serier (deeplizard 2019), (3Blue3Brown 2019) där dessa koncept presenteras på ett pedagogiskt sätt i videoformat. Vill man hellre ha något mer lättläsligt har Skymind en artikel (Skymind 2019) som behandlar detta. Det finns också en svensk sida från Göteborgs Universitet som (Malmgren 2003) tar upp artificiella neuronnät.

Rashids (Rashid 2016) bok om hur uppbyggnaden av neuronnät användes och hur matematiken bakom dem funkar ger en enkel men också pedagogisk förståelse. Vill man ha en mjukare förståelse över hur allt funkar ihop så finns det flera bra blogginlägg som pratar om detta (Babs 2019), (Sanjeevi 2019) och (Skalski 2019)

För att gå närmare in på den specifika tekniken inom reinforcement learning som använts finns en grundläggande beskrivning i ett blogginlägg (Karagiannakos 2019). Mer avancerade metoder, A3C, gås också igenom lite mer grundligt på en annan sida (Nicholls 2019). Ett blogginlägg (Juliani 2019) som går in specifikt hur entropi används inom reinforcement learning kan också vara intressant.

Flera andra har arbetat med att ta fram AI för StarCraft2, mycket av deras arbete ligger till grund för denna rapport. En artikel från DeepMind (Deepmind n.d.) beskriver deras initiala arbete med att ta fram en AI för StarCraft 2. De tog sedan fram en agent som kan läsas mer om i deras artikel (Team 2019) samt ses på Youtube (DeepMind 2019). En annan grupp som tagit fram en agent är tencent, som beskriver sitt arbete i en artikel (Sun m. fl. 2018). Ett liknande arbete som är utvecklat i spelet Dota 2 istället för Starcraft 2 kan läsas om i OpenAIs artikel (OpenAI 2018).

För mer läsning om etik inom artificiell intelligens finns en hemsida som beskriver de största generella problemen (Bossman 2016), samt en artikel som beskriver ett känt etiskt dilemma gällande självstyrande bilar i synnerhet (Nyholm och Smids 2016).

Gällande utvecklingen av neuronnärt kom till kan man see detta från McCulloch och Pitt som står en del i (McCulloch 1990). Den andra delen av detta är var den sedan blev applicerbart i (1950) Kleene bok (Kleene 1956). Annras om man vill lite mer grundlägnade approch till vad neuralnätverk så funkar wikipedia artikelern om det rätt bra (Wikipedia 2019[a])

Litteratur

Wikipedia (2019[a]). *Artificial neural network*. URL: https://en.wikipedia.org/wiki/Artificial_neural_network (hämtad 2019-05-14).

— (2019[b]). *Markov decision process*. URL: https://en.wikipedia.org/wiki/Markov_decision_process (hämtad 2019-04-14).

3Blue3Brown (2019). *Neural networks*. https://youtu.be/aircAruvnKk. (Hämtad 2019-04-28).

Anderson, Mark Robert (2017). "Twenty years on from Deep Blue vs Kasparov: how a chess match started the big data revolution". I: *The Conversation*. (Hämtad 2019-02-14).

Babs, Temi (2019). *The Mathematics of Neural Networks*. https://medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05. (Hämtad 2019-05-11).

Bhatia, Richa (2017). *Understanding the difference between Symbolic AI & Non Symbolic AI*. https://www.analyticsindiamag.com/understanding-difference-symbolic-ai-non-symbolic-ai/. (Hämtad 2019-02-13).

Bossman, Julia (2016). *Top 9 ethical issues in artificial intelligence*. URL: https://www.weforum.org/agenda/2016/10/top-10-ethical-issues-in-artificial-intelligence/.

Chollet, François (2018). *Deep Learning with Python*. Manning Publications.

deeplizard (2019). *Machine Learning & Deep Learning Fundamentals*. https://www.youtube.com/playlist?list=PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU. (Hämtad 2019-04-28).

DeepMind (2019). *DeepMind StarCraft II Demonstration*. https://youtu.be/cUTMhmVh1qs?t=251. (Hämtad 2019-02-15).

Deepmind, Blizzard (n.d.). "StarCraft II: A New Challenge for Reinforcement Learning". I:

D'souza, Rhett (2018). *Symbolic AI v/s Non-Symbolic AI, and everything in between?* https://medium.com/datadriveninvestor/symbolic-ai-v-s-non-symbolic-ai-and-everything-in-between-ffcc2b03bc2e. (Hämtad 2019-02-13).

Expert Exchange (2019). *Processing Power Compared*. https://pages.experts-exchange.com/processing-power-compared. (Hämtad 2019-02-14).

Hassabis, Demis (2016). "What we learned in Seoul with AlphaGo". I: (hämtad 2019-02-14).

Hui, Jonathan (2019a). *Deep Reinforcement Learning Series*. https://medium.com/@jonathan_hui/rl-deep-reinforcement-learning-series-833319a95530. (Hämtad 2019-04-28).

Hui, Jonathan (2019b). *RL—Imitation Learning*. https://medium.com/@jonathan_hui/rl-imitation-learning-ac28116c02fc. (Hämtad 2019-05-08).

IBM (2011). *Deep Blue*. https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/. (Hämtad 2019-02-13).

Juliani, Arthur (2019). *Maximum Entropy Policies in Reinforcement Learning & Everyday Life*. (Hämtad 2019-04-28).

Karagiannakos, Sergios (2019). *The idea behind Actor-Critics and how A2C and A3C improve them*. https://sergioskar.github.io/Actor_critics/. (Hämtad 2019-04-28).

Kleene, S. C. (1956). *Representation of Events in Nerve Nets and Finite Automata.* Princeton: Princeton. DOI: \url{https://doi.org/10.1515/9781400882618}.

Malmgren, Helge (2003). *Artificiella Neurala Nätverk en kort introduktion*. URL: http://www.phil.gu.se/ann/annintr.html.

Matiisen, Tambet (2019). *Demystifying Deep Reinforcement Learning*. (Hämtad 2019-04-28).

McCarthy, John (2007). *What is AI? / Basic Questions*. http://www-formal.stanford.edu/jmc/whatisai/node1.html. (Hämtad 2019-02-13).

McCulloch Warren, och S. Pitt (1990). *A logical calculus of the ideas immanent in nervous activity*. Kluwer Academic Publishers.

Moore, Gordon E. (1965). ”Cramming More Components onto Integrated Circuits”. I: (hämtad 2019-02-15).

Nicholls, Chris (2019). *Reinforcement learning with the A3C algorithm*. https://cgnicholls.github.io/reinforcement-learning/2017/03/27/a3c.html. (Hämtad 2019-04-28).

Nyholm, Sven och Jilles Smids (2016). ”The Ethics of Accident-Algorithms for Self-Driving Cars: an Applied Trolley Problem?” I: *Ethical Theory and Moral Practice* 19.5, s. 1275–1289. ISSN: 1572-8447. DOI: 10.1007/s10677-016-9745-2.

OpenAI (2018). *OpenAI Five*. https://blog.openai.com/openai-five/.

Rashid, Tariq (2016). *Make your own neural network : a gentle journey through the mathematics of neural networks, and making your own using the Python computer language*. CreateSpace Independent Publishing Platform.

Russel, Stuart och Peter Norvig (2009). *Artificial Intelligence A Modern Apporoach, Third Edition*. Pearson.

Sanjeevi, Madhu (2019). *Chapter 7 : Artificial neural networks with Math.* https://medium.com/deep-math-machine-learning-ai/chapter-7-artificial-neural-networks-with-math-bb711169481b. (Hämtad 2019-05-11).

Silver, David och Demis Hassabis (2016). ”AlphaGo: Mastering the ancient game of Go with Machine Learning”. I: (hämtad 2019-02-14).

Silver, David, Aja Huang m. fl. (2016). ”Mastering the game of Go with deep neural networks and tree search”. I: *Nature* 529, s. 484–503. DOI: 10.1038/nature16961. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Skalski, Piotr (2019). *Deep Dive into Math Behind Deep Networks*. https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba. (Hämtad 2019-05-11).

Skymind (2019). *A Beginner’s Guide to Neural Networks and Deep Learning*. https://skymind.ai/wiki/neural-network. (Hämtad 2019-04-28).

Sun, Peng m. fl. (2018). ”TStarBots: Defeating the Cheating Level Builtin AI in StarCraft II in the Full Game”. I:

Sutton, Richard S. och Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. MIT Press.

Team, The AlphaStar (2019). *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. (Hämtad 2019-02-06).

Yue, Yisong och Hoang M. Le (2018). *ICML2018: Imitation Learning Tutorial*. https://sites.google.com/view/icml2018-imitation-learning/. (Hämtad 2019-05-08).

A

Ordlista

A2C: Förkorting för Advantage Actor Critic, en reinforcement learning-algoritm baserat på Actor Critic-algoritmen.

Accuracy: På svenska noggranhet. Beskriver hur ofta en modell väljer rätt svar. Används framför allt vid supervised learning.

Actor Critic: En metod inom reinforcement learning som går ut på att en agent (actor) lär sig välja rätt handlingar, och en annan agent (critic) lär sig värdefunktionen för tillstånden för att kunna evaluera actorns beslut.

Adam: En specifik optimeringsalgoritm.

Agent: Ett annat namn för det program som utvecklas för att fatta rätt beslut.

AI: Artificiell intelligens.

AlphaStar: En agent utvecklad av DeepMind som var först i världen med att besegra professionella spelare.

ANN: Artificiellt neuronnät.

Avdragsfaktor: Tal mellan 0 och 1 som garanterar att den framtida belöningen konvergerar till ett ändligt värde.

Bana: En följd av tillstånd och handlingar.

Belöning: Mått på hur bra en handling var som används för en agent att lära sig vad den borde göra.

Blizzard: Blizzard Entertainment är företaget som utvecklade StarCraft 2.

Cross entropy: En typ av förlustfunktion.

Deep learning: En maskininlärningsmetod som på något sätt använder ett neuralt nät-

verk med åtminstone ett lager mellan input och output.

DeepMind: Dotterföretag till Google som tillsammans med Blizzad utvecklat både PySC2 och AlphaStar.

Dense layer: Om alla noder i ett lager är kopplade till alla noder i dess föregående lager kallas lagret för ett dense layer.

Feature layer: Bilder som visar information grafiskt istället för med siffror eller tecken. Oftast i form av kartor.

Fog of War: Fog of War är en term som beskriver att kartan i StarCraft 2 endast är belyst runt omkring sina egna enheter, vilket innebär att man inte har fullständig information om motståndaren.

Förlustfunktion: Är funktionen som används för att beräkna förlust.

Gradient ascent: En optimeringsmetod som finner lokala maximum genom att uppdatera parametrarna i gradienten av målfunktionens riktning.

Handling: Ett beslut som en agent tar som kan påverka tillståndet.

Handlingsrum: Mängden av möjliga handlingar givet ett tillstånd.

Imitation learning: Imitation learning är en underkategori av supervised learning.

Förlust: En summa av fel som vill minimeras vid träning av nätverk.

Maskininlärning: Maskininlärning är ett begrepp inom datavetenenskap som handlar om att få datorer att lära sig från olika datatyper.

Matchup: Matchup är vilken ras du och motståndaren spelar som.

MDP: Markovbeslutsprocess. Ett ramverk för att en dator ska kunna fatta beslut.

Monte Carlo sampling: Uppskattning som görs genom att samla in slumpmässig data.

MVC pattern: Står för Model-View-Controller pattern och är ett arkitekturmönster.

Omgivning: Omgivningen är miljön som används. Exempelvis ett spel som StarCraft 2.

Policy: I sammanhanget Markovbeslutsprocesser är policyn funktionen som tar besluten. Denna modell tillämpas bland annat i reinforcement learning där policyn är agentens beslutfattande. Den tar in ett tillstånd och ger ut en sannolikhetsfördelning över olika

handlingar som kan tas.

Policy gradient: En reinforcement learning-metod som syftar till att parametrisera agentens policy och justera denna så att väntevärdet av den totala framtida belöningen maximeras.

PySC2: Python bibliotek utvecklat i maskininlärningssyfte som underlättar interaktion med StarCraft 2.

Q-learning: En reinforcement learning-metod som beräknar Q-värdet för varje handling.

Q-värde: Värdefunktionen givet en specifik handling.

Reinforcement learning: Reinforcement learning är ett område inom maskininlärning där agenter tränas till att lära sig utföra handlingar som maximerar en numerisk belöning i en omgivning.

StarCraft 2: Realtidsstrategi-spel utvecklat av Blizzard.

Steg: Ett diskret mått på tid inom den miljö som agenter arbetar i.

Supervised learning: En maskininlärningsmetod där AI:n matas med indata och tillhörande korrekta utdata.

Temporal Difference learning: Inlärningsmetod där agenten förutser vilka resultat som kommer ske i framtiden och förändrar modellen efter detta.

Tillstånd: Ett tillstånd inom programmet är en beräknings metod för en viss postion där man kan räkna för att komma till då ett annat tillstånd.

Tillståndsövergångssannorlikhet: Sannolikheten för att nästa tillstånd kommer vara tillstånd s.

UML diagram: Förkorting för "unified modelling language", vilket är ett språk som kan visuliera koddelar med diagram.

Utrullning: En utrullning är en bana som börjar från initialtillståndet s_0 och följer en viss policy $\pi(a_t|s_t)$.

Värdefunktion: Väntevärdet av den totala framtida belöningen givet en policy.

B

Matematiska härledningar

B.1 Skattning av $\nabla_\theta J(\theta)$

För att beräkna $\nabla_\theta J(\theta)$ kan sampling användas (). Det går att visa att gradienten av $P_\theta(\tau)$ med avseende på parametrarna kan uttryckas som

$$\nabla_\theta P_\theta(\tau) = P_\theta(\tau)\nabla_\theta P_\theta(\tau) \tag{B.1}$$

genom att derivera en logaritm av en funktion:

$$f(\mathbf{x})\nabla\log(f(\mathbf{x})) = f(\mathbf{x})\frac{\nabla f(\mathbf{x})}{f(\mathbf{x})} = \nabla f(\mathbf{x}).$$

Insättning av ekvation (B.1) i uttrycket för gradienten ger

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_\tau \nabla_\theta P_\theta(\tau) R(\tau) \\ &= \sum_\tau P_\theta(\tau)\nabla_\theta \log(P_\theta(\tau)) R(\tau) \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log(P_\theta(\tau)) R(\tau)\right].\end{aligned} \tag{B.2}$$

Gradienten är alltså ett väntevärde och kan därför samplas. Minns nu att $P_\theta(\tau)$ är sannolikheten för en bana givet en policy $\pi_\theta(a,s)$. Detta uttrycks som

$$P_\theta(\tau) = p(s_0)\prod_{t=0}^{\infty}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t),$$

där p är tillståndsövergångssannolikheten. Logaritmen av uttrycket blir en summa

$$\log(P_\theta(\tau)) = \log(p(s_0)) + \sum_{t=0}^{\infty}\log(\pi_\theta(a_t|s_t)) + \sum_{t=0}^{\infty}\log(p(s_{t+1}|s_t,a_t))$$

och det faller sedan att gradienten med avseende på θ endast kommer påverka mittentermen, då p är oberoende av parametriseringen. Det innebär att

$$\nabla_\theta \log(P_\theta(\tau)) = \sum_{t=0}^{\infty}\nabla_\theta \log(\pi_\theta(a_t|s_t)).$$

Med hjälp av detta går det att uttrycka gradienten i ekvation (B.2) i termer av π_θ, så att

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log(\pi_\theta(a_t|s_t))R(\tau)\right].$$

$\nabla_\theta J(\theta)$ kan nu samplas genom att köra flera episoder, alltså fullständiga banor, vilket ger skattningen

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N}\sum_{i=0}^{N-1}\sum_{t=0}^{T-1} \nabla_\theta \log(\pi_\theta(a_{i,t}|s_{i,t}))R(\tau_i) \tag{B.3}$$

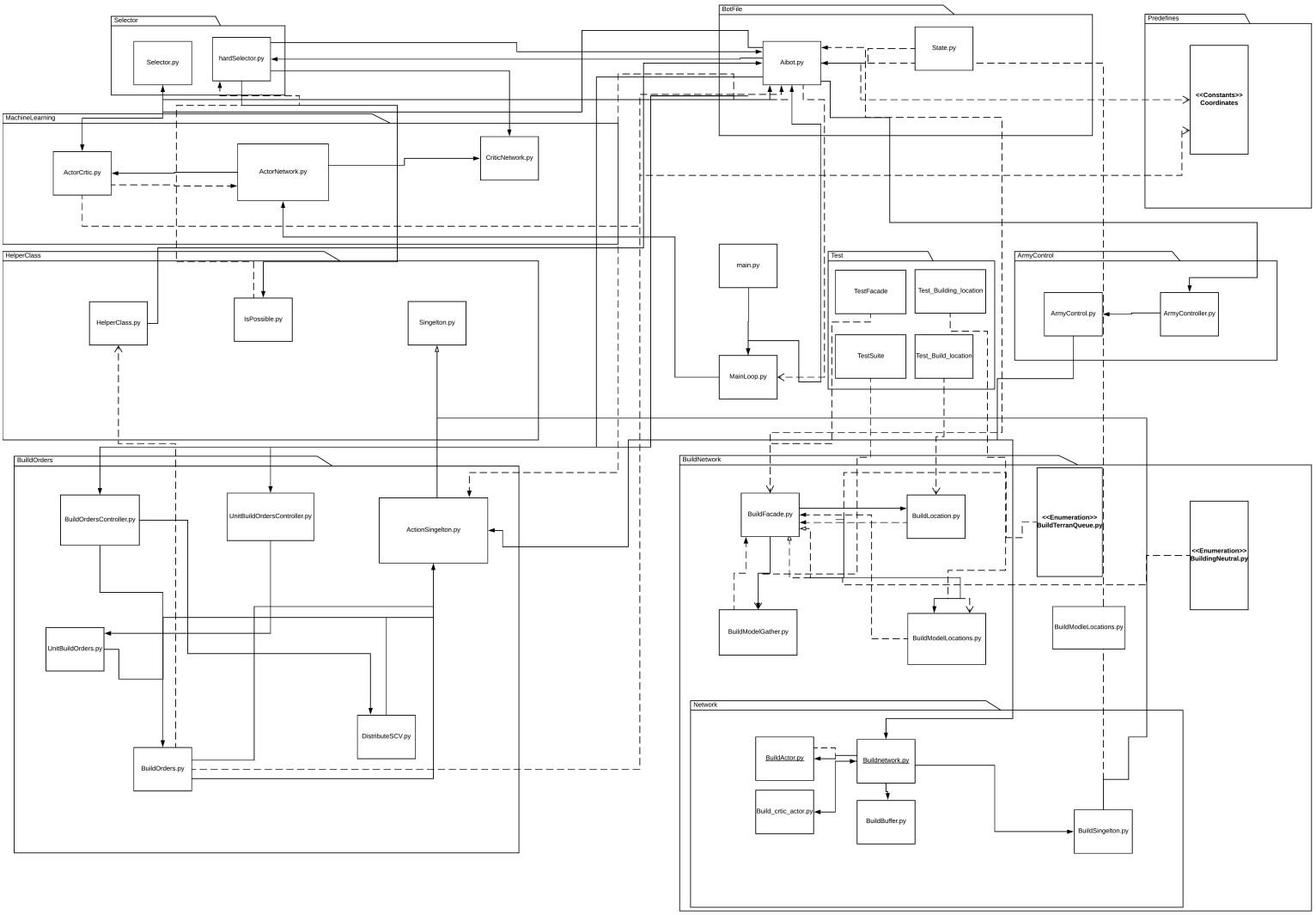
där N är det totala antalet episoder som ska användas i skattningen och i är den i:te episoden, samt att antalet tidssteg begränsats till T.

C

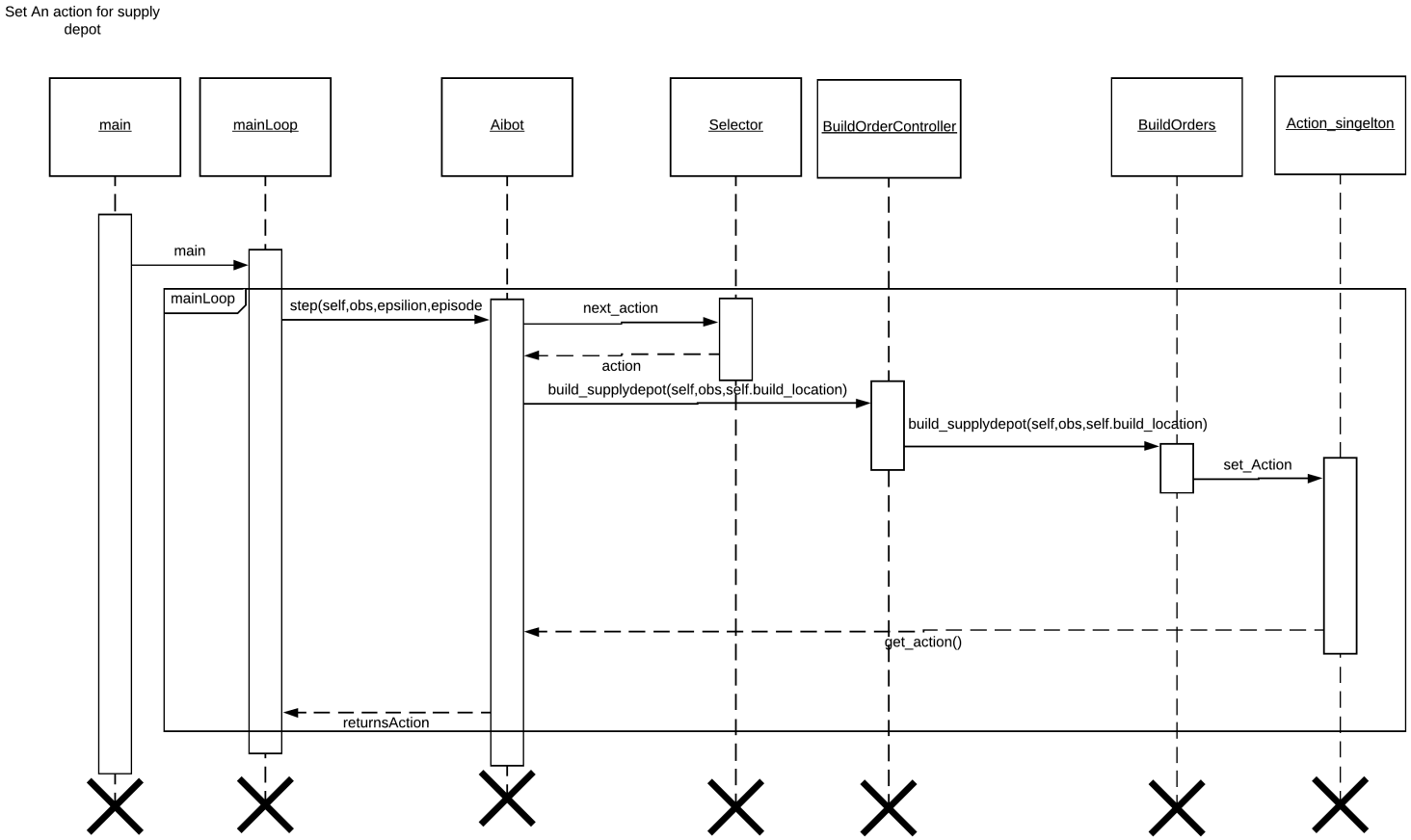
Kodbas

Vi kodbas finns att hitta via denna länk:
`https://github.com/DukeA/DAT02X-19-03-MachineLearning-Starcraft2`.
Koden är strukturerad så att huvudprogrammet *main* startar *mainLoop* vilket i varje tidssteg kör stegfunktionen implementerad i *aiBot*. *aiBot* i sin tur frågar *selector*-klassen om vilket handling som ska utföras. Selektorn ger tillbaka en handling som ska utföras som i sin tur hämtas i singleton-filerna för *buildUnit*/*buildOrder*/*armyControl*. Selektorn baserar sitt val beroende på den implementerade agenten. Exempelvis kan det vara A2C-agenten som då anropas och finns placerad under *MachineLearning*-mappen. Här konstrueras och definieras funktionerna agenten behöver för att kunna träna och göra beslut.

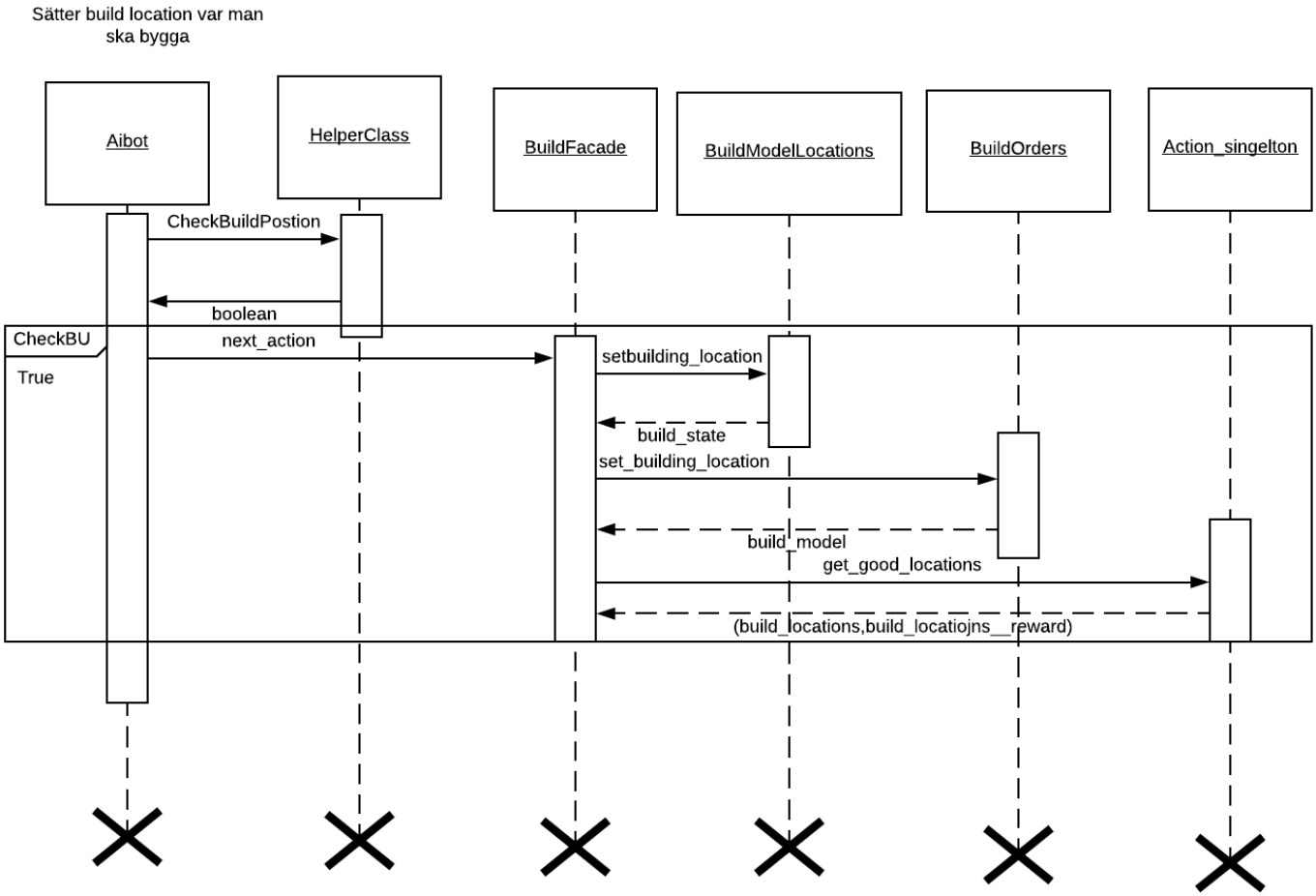
Vi har försökt efterfölja ett MVC-mönster i koden. Delar av koden var nerstukterade med att de skulle göra det minsta möjligt samt att också ta bort vissa cirkulära beroenden. Detta gjordes genom att dela ner dessa klasser till mindre klasser med endast en uppgift. Detta kan synas i figur C.1 som är ett UML-diagram över de klasser och filer som ingår i kodbasen. Vissa specifika funktioner och kan ses i sekvensdiagrammen i figur C.2, figur C.3 och figur C.4.



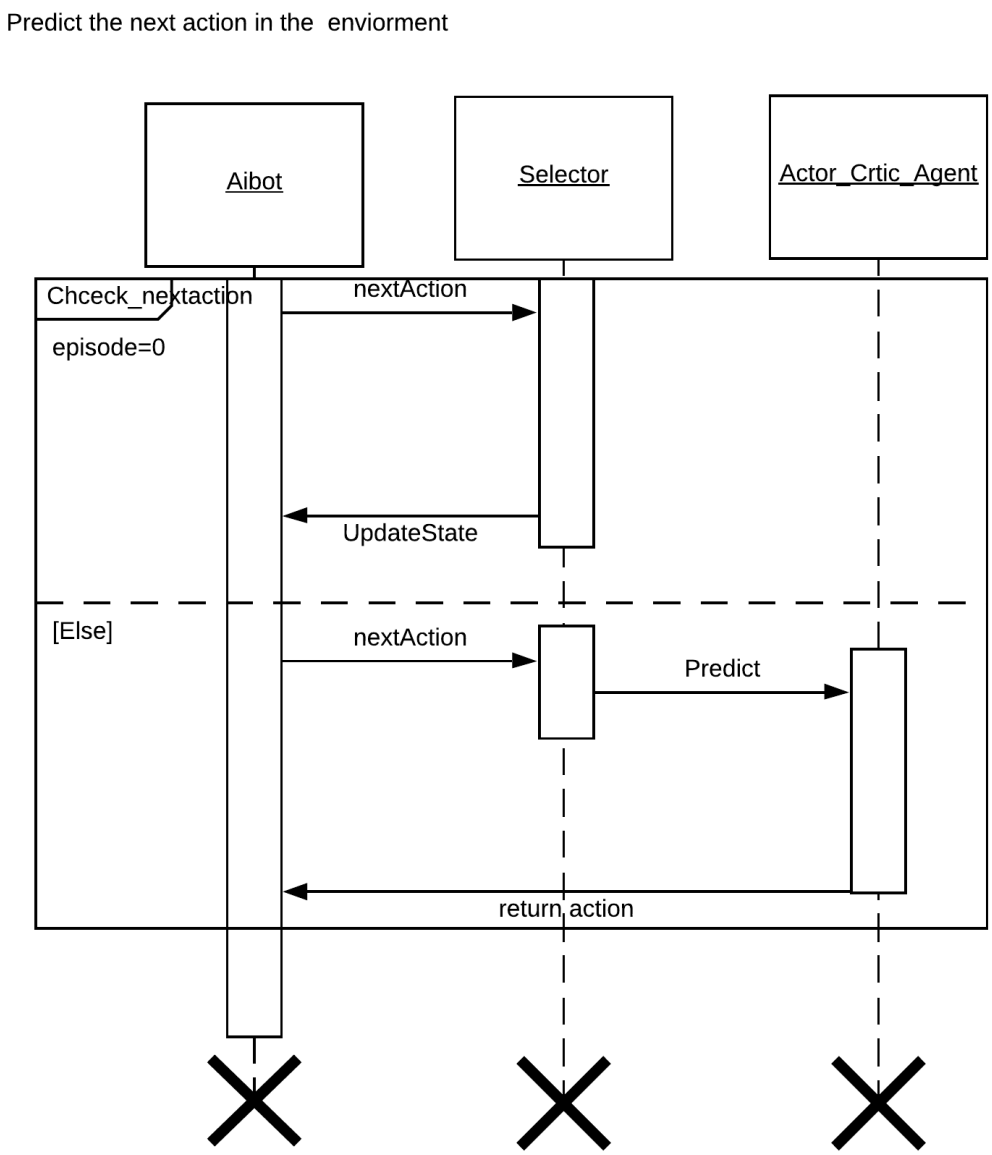
Figur C.1: Ett UML-diagram av klass- och filkopplingar i kodbasen.



Figur C.2: Sekvensdiagram över hur en supply depot byggs. De andra byggmetoderna är snarlika.



Figur C.3: Visar punkten där byggnaden som ska byggas bestäms.



Figur C.4: Visar hur nästa handling bestäms.