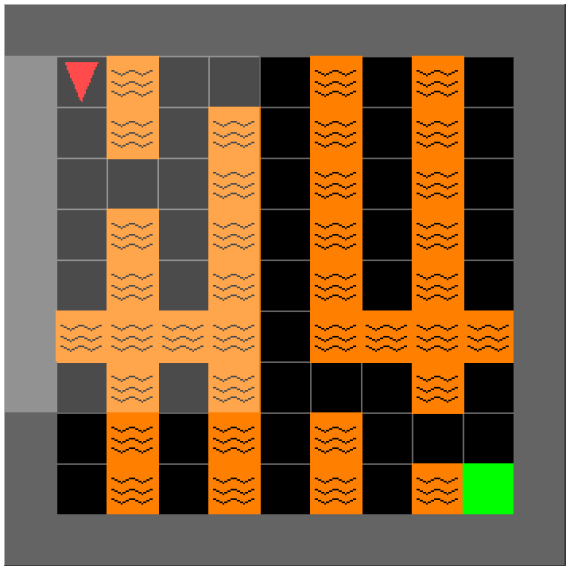
CHALMERS | GÖTEBORGS UNIVERSITET



Pathfinding med reinforcement learning i delvis observerbara miljöer

Anne Engström
Joel Lidin
Gustav Molander
Olle Månsson
Noa Onoszko
Hugo Ölund

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Pathfinding med reinforcement learning i delvis observerbara miljöer

Kandidat i fysik inom civilingenjörsprogrammet Teknisk fysik vid Chalmers

Joel Lidin
Gustav Molander
Olle Månsson

Kandidat i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers

Anne Engström
Noa Onoszko
Hugo Ölund

Handledare: Marina Axelson-Fisk
Examinator: Ulla Dinger, Maria Roginskaya

Institutionen för Matematiska vetenskaper
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Självlärande, självkörande – Pathfinding med artificiell intelligens

Utvecklingen av artificiell intelligens (AI) accelererar, och redan idag görs förändringar som påverkar hela samhället. Många uppgifter som idag utförs av människor kommer i framtiden utföras av maskiner, som arbetar outtröttligt och med högre effektivitet. När NASA skickar obemannade robotar till Mars är det svårt att styra dessa från jorden, eftersom det tar en signal flera minuter att färdas från jorden till Mars. Dessa robotar måste därför kunna navigera helt själva. Pathfinding och självnavigering av robotar på Mars är ett område där AI kan användas.

En stor del av forskningen inom AI fokuserar idag på konstruktionen av så kallade artificiella neurala nätverk. De neurala nätverken är uppbyggda för att efterlikna biologiska synapser i hjärnan. Likt hjärnan är nätverken uppbyggda av artificiella neuroner kallade noder, som är sammankopplade till varandra och uppstrukturerade i flera lager. Informationen processas genom att noder aktiveras i det första lagret, som sedan genom en kedjereaktion aktiverar noder i nästkommande lager. De noder som aktiveras i det sista lagret är den processade informationen, som till exempel kan vara ett beslut att svänga vänster.

Att lösa en labyrint med en dator kan göras på många sätt. En dator skulle kunna prova sig fram till en väg mellan start och mål. Men när hela labyrinten inte är synlig samtidigt, och en agent som försöker hitta till mål endast kan se delar av sin omgivning, blir problemet mer komplicerat. En sådan situation är dock mycket verklighetstrogen, där förutsättningar kan förändras allt eftersom och oförutsägbara hinder kan dyka upp. Det tillvägagångsätt som då lämpar sig bäst går ut på att datorn lär sig följa vissa principer, som till exempel att ta sig runt hörn och att utforska nya delar av miljön. Här är neurala nät mycket användbara.

För att träna upp ett neuralt nätverk att följa vissa principer kan belöningar användas för att motivera olika beteenden. Under träningen låter man en agent som styrs av nätverket ta beslut i en labyrint helt beroende av vad den ser. Genom att sedan belöna rätt beslut kan nätverket uppdateras så att belöningen till slut maximeras. Många träningsiterationer kan krävas för att helt träna upp ett nätverk, men när träningen är klar kommer nätverket med hög precision kunna ta de bästa besluten. Detta kallas för en *policy*.

Under träningen uppdateras hela tiden nätverket för att maximera belöningen. Noderna i nätverkets lager är sammankopplade med varandra via vikter, där storleken på vikten bestämmer hur mycket en nod bidrar till nästa nods aktivering. Vid varje tillstånd tas ett beslut som nätverket associerar med en belöning. Om nätverket tar ett icke-optimalt beslut kommer vikten mellan noderna att ändras vid nästa uppdatering, och på så sätt lär sig nätverket att fatta allt bättre beslut.

De neurala nätverken kan göras allt mer komplicerade för att fylla nya funktioner. Genom att lägga till fler lager av noder kan nätverket hantera olika abstraktionsnivåer. Exempelvis kan ett tidigt lager identifiera var det finns hörn i en labyrint och nästa lager sedan besluta att agenten ska gå tillbaka för att ta sig runt hörnet. Genom att lägga till ett minne av agentens tidigare steg kan beslutsfattandet planeras över längre sekvenser, vilket gör utforskandet mer effektivt. Träningen kan också stabiliseras genom att uppdatering inte sker efter varje steg utan först efter att ett antal steg utförts. Ändringar görs sedan så att besluten förbättras vid samtliga steg.

En algoritm med dessa tillägg kan lära sig att från grunden navigera en labyrint med väggar eller lava. Lava måste undvikas för att agenten inte ska dö. Efter några minuters träning på en vanlig persondator kan agenten lösa en slumpad labyrint den aldrig tidigare tränats i.

Abstract

Reinforcement learning algorithms have the ability to solve problems without explicit knowledge of their underlying model. Instead, they infer a strategy directly from observations and rewards acquired by interacting with their environment. This makes them suitable candidates for solving pathfinding problems in a partially observable setting, where the aim is to find a path in an environment with restricted vision.

This report aims to investigate how Markov decision processes and reinforcement learning can be used to model and solve partially observable pathfinding problems. Existing literature has been reviewed to give a theoretical background of the subject, before progressing to practical implementations. We have applied state-of-the-art algorithms taken from two subclasses of reinforcement learning methods: value based algorithms and policy based algorithms.

We find that partially observable Markov decision processes can be used to model pathfinding problems, but not all reinforcement learning algorithms are suitable for solving them. In theory, value based algorithms show potential but when implemented they did not yield positive results. Conversely, the policy based algorithm Proximal Policy Optimization is able to solve the problem convincingly. This algorithm also performs well in environments previously not trained in, thus displaying some ability to generalize its policy.

Sammanfattning

Reinforcement learning-algoritmer har förmågan att lösa problem utan explicit kunskap om deras underliggande modell. Istället utgår de från en strategi baserad på observationer och belöningar, vilka fås genom interaktion med deras omgivning. Detta gör att de lämpar sig väl för att lösa pathfinding-problem i delvis observerbara miljöer, där syftet är att hitta en rutt i en miljö med begränsad synfält.

Denna rapport syftar till att undersöka hur Markov decision processes och reinforcement learning kan användas för att modellera och lösa delvis observerbara pathfinding-problem. Befintlig litteratur har studerats för att ge en teoretisk bakgrund, varefter vi skiftar fokus till den praktiska implementeringen. Vi tillämpade state-of-the-art-algoritmer från två underkategorier till reinforcement learning-metoder: värdebaserade algoritmer och policybaserade algoritmer.

Vi finner att delvis observerbara Markov decision processes kan användas för att modellera pathfinding-problem, men att inte alla reinforcement learning-algoritmer lämpar sig väl för att lösa dem. I teorin visar värdebaserade algoritmer potential men då dessa implementerades påträffades inte goda resultat. I kontrast till dessa lyckades den policybaserade algoritmen Proximal Policy Optimization lösa problemet på ett övertygande sätt. Denna algoritm löser även problemet bra i miljöer den inte tränats i, vilket visar på en förmåga att generalisera sin policy.

Innehåll

Förord

Vi vill ge ett stort tack till vår handledare Marina Axelson-Fisk för bra handledning och värdefulla synpunkter under projektets gång. Vi vill även tacka Per Ljung för korrekturläsning.

Olle har utvecklat ramverket `purl` och implementerat de flesta algoritmer som Q-table, Q-network, DQN och PPO. I rapporten har han fokuserat främst på Artificella neurala nätverk i teorikapitlet, PPO i implementeringskapitlet och Delvis observerbara miljöer i disksussionskapitlet. Olle har även gjort många av figurerna i rapporten.

Gustav har främst arbetat med diskussionskapitlet och teorikapitlet i rapporten. Gustav har skrivit om policyteori och teori om delvis observerbara miljöer, även inledningen och huvuddelen av diskussionen har han ansvarat för. Författare av populärvetenskaplig rapport och utkast till etiska aspekter. Gustav har även uppvaktat gruppen med fika.

Noa har framförallt jobbat på teorin i rapporten. Fokus har legat på Reinforcement learning, Policy och optimalitetsmodeller samt inledning till Implementering. Han har också skrivit på Markov decision processes och Artificiella neurala nätverk, främst Bakåtpropagering. Noa har även skrivit på Slutsats och ett utkast till Abstract samt komponerat syftet.

Anne har fokuserat på att skriva rapporten. Framförallt har Anne jobbat med teoridelen, där huvudansvar har varit Värdefunktioner och Bellmans ekvation, Delvis observerbara miljöer och Q-learning. I implementeringskapitlet har Anne ansvarat för Q-table och Q-network. Övriga delar som Anne skrivit är Rapportöversikt, Avgränsningar, Etiska aspekter samt delar av Artificiella neurala nätverk, Syfte och Slutsats.

Joel och har varit med och implementerat algoritmer som DQN och DDQN. I rapporten har Joel fokuserat på implementationsdelen under DQN-avsnittet. Joel har även haft huvudansvar på Bakåtpropagering samt skrivit på RL som senare ändrats. Alla bilder i Implementering har Joel fixat till samt de tre algoritmerna i samma avsnitt. Joel har varit med och hjälpt på diskussionavsnitten.

Hugo har varit med och implementerat DQN och baserat på den algoritmen implementerat DRQN. I rapporten har Hugo skrivit teori om MDP och policygradientmetoder. I implementeringsdelen har Hugo skrivit resultat och diskussion på Q-table, Q-network och DQN. Hugo har även skrivit om implementeringen av DQN och DRQN. För bakgrunden, den populärvetenskapliga presentationen samt abstract var Hugo med och skrev utkast.

Under arbetets gång har dagbok och loggbok förts.

1 Inledning

Stephen Hawking sade en gång:

> Success in creating AI would be the biggest event in human history. Unfortunately, it might also be the last, unless we learn how to avoid the risks.

Kanske hade han rätt. AI och maskininlärning är dock alltför spännande och har alldeles för många användningsområden för att inte fördjupa sig i.

1.1 Bakgrund

Maskininlärning är ett område inom artificiell intelligens som berör algoritmer och statistiska modeller som lär sig utan att explicit programmeras. Typiska exempel är prediktion med regressionsmetoder, klassificering av kategorisk data och algoritmer som spelar spel som schack, med mera. Typiskt för många metoder är antagandet att det finns en bakomliggande funktion som kan approximeras. Denna funktion kan exempelvis beskriva hur bra en schackposition är för vit spelare och tar då in spelpjäsernas position som indata. Kan denna funktion approximeras finns möjlighet att avbilda indata på en användbar mängd utdata. Indatan behöver inte nödvändigtvis vara något som algoritmen eller funktionen sett förut, vilket öppnar möjligheter för metoder konstruerade med maskininlärning att agera på mycket mer generella problem än andra metoder.

I detta projekt behandlas ett fält inom maskininlärning som kallas reinforcement learning. Detta är, enligt ämnets pionjär Richard S. Sutton [1], en klass av algoritmer som lär sig hur de ska avbilda sitt och sin omgivnings tillstånd på olika handlingar med hjälp av ett belöningssystem. I miljön finns en agent vars handlingar styrs av algoritmen. Agentens handlingar och vilka nya tillstånd de leder till gör så att agenten får en belöning. Handlingarna utförs sekventiellt och ska på sikt leda till maximal belöning, eller minimal kostnad för agenten. I rapporten presenteras även Markov decision processes. Det är stokastiska processer som modellerar miljöer där varje tillstånd har tillhörande beslut som kan resultera i belöningar eller bestraffningar. Med schack som exempel kan agenten vara vit spelare som ska ta beslut om bästa möjliga drag. Efter varje drag får spelaren en bestraffning vars storlek beror på hur långt ifrån optimalt draget var.

En av de mest kända tillämpningarna av reinforcement learning presenteras i en artikel från 2015 [2], där författarna konstruerade en algoritm som lärde sig att spela klassiska Atari-spel på en högre nivå än de bästa mänskliga spelarna klarar av. I en artikel från 2018 [3] beskriver författarna en metod för att med hjälp av reinforcement learning hitta den optimala strategin för en glidfarkost att stiga från marken. Genom att anpassa farkostens orientering i förhållande till stigande varmluftsströmmar försöker de efterlikna fåglars förmåga att navigera i turbulenta flöden. Ett annat område där reinforcement learning kan användas är i så kallade pathfinding-problem, det vill säga problem där exempelvis en robot ska hitta kortaste vägen mellan två punkter. I vissa fall ser roboten inte hela omgivningen och då sägs den navigera en delvis observerbar miljö. Bredden av tillämpningar och lösningsmetoder gör reinforcement learning till ett intressant område att utforska.

1.2 Syfte

Detta arbete syftar till att utforska hur Markov decision processes och reinforcement learning kan användas för att modellera respektive lösa pathfinding-problem med en agent. Fokus ligger på att redogöra för den teori som ligger till grund för sådana problem samt att systematiskt utveckla en alltmer välfungerande algoritm. Målet är att implementera en algoritm som löser pathfinding-problem med reinforcement learning i en delvis observerbar miljö.

1.3 Rapportöversikt

Då syftet med projektet är att utveckla en pathfinding-algoritm vars beslutsprocess baseras på Markov decision processes behandlas inledningsvis teorin för detta ämne. Begreppet Markov decision process redogörs för och dess olika beståndsdelar definieras för att ge förståelse för hur en agent tar beslut sekventiellt. Två grundläggande modeller för lösningar till Markov decision processes behandlas, där lösningar avser optimala följder av beslut. Det redogörs också för hur olika

förlopp modelleras beroende på miljöns egenskaper. Detta följs av ett avsnitt om reinforcement learning, där skillnaden mellan värdebaserade och policybaserade metoder beskrivs. Efter detta beskrivs artificiella neurala nätverk. Det är en grupp funktionsapproximatorer inspirerade av hjärnans biologiska nätverk av neuroner som är effektiva i stora och komplexa miljöer. Detta följs av en beskrivning av hur dessa nätverk är uppbyggda samt en genomgång av hur indata tas in, bearbetas och ger ett resultat. Fokus ligger här på hur nätverk tränas för att lösa givna problem med hjälp av en kostnadsfunktion och bakåtpropagering. Därefter behandlas teorin för två grupper av metoder, Q-learning och policygradientmetoder, som ligger till grund för de algoritmer som behandlas i projektet.

Slutligen följer implementeringen som är uppdelad i fyra delar, en för varje typ av algoritm: Q-table, Q-network, Deep Q-network och Proximal Policy Optimization. Det slutliga målet med Deep Q-network och Proximal Policy Optimization är att lösa pathfinding-problem med en agent i delvis observerbara miljöer. Beskrivningar av algoritmernas uppbyggnad och särdrag ges och resultatet för respektive algoritm redovisas. Resultaten analyseras och en diskussion följer baserat på dessa resultat.

1.4 Avgränsningar

I detta arbete kommer endast algoritmer baserade på reinforcement learning behandlas. Algoritmer som lär sig själva då en perfekt modell av miljön finns tillgänglig, vilket kallas dynamisk programmering [4], kommer inte att användas.

Endast miljöer med en agent kommer användas i detta arbete. Anledningen är att komplexiteten bedöms vara för stor då fler agenter introduceras i miljöerna, både med avseende på implementering och beräkningar. Dessutom sammanfaller inte de metoder som används då flera agenter existerar i samma miljö med målet för detta arbete, som är att för en delvis observerbar miljö utveckla en pathfinding-algoritm.

Färdiga bibliotek har använts vid utvecklandet av algoritmerna. Det är tidskrävande att skriva vissa metoder från grunden och därmed har en avvägning gjorts där algoritmens funktionalitet anses vara viktigare än optimering av matrisoperationer.

2 Teori

I det här avsnittet redogör vi för den teori som är relevant för arbetet. Vi börjar med att behandla Markov decision processes, som kommer användas för att beskriva pathfinding som en stokastisk process. Därefter följer reinforcement learning, som är vårt verktyg för att lösa pathfinding-problem. Slutligen behandlas funktionsapproximatorerna artificiella neurala nätverk, innan slutligen Q-learning och policygradientmetoder förklaras.

2.1 Markov decision processes

För att modellera sekventiellt beslutsfattande där beslutens resulterande handlingar påverkar ett tillstånd används i stor utsträckning *Markov decision processes (MDP)*. De definierar ett tydligt matematiskt ramverk som, både konceptuellt och praktiskt, lämpar sig väl för reinforcement learning [4]. MDP är en stokastisk process som har *Markovegenskapen*, vilket innebär att sannolikheten att hamna i ett visst tillstånd endast beror på processens nuvarande tillstånd. Det vill säga

$$P(X_n = x_n | X_{n-1} = x_{n-1}, ..., X_0 = x_0) = P(X_n = x_n | X_{n-1} = x_{n-1}),$$

där $X_0, \ldots, X_n$ är slumpvariabler som antar värden i något diskret tillståndsrum.

MDP definieras av tupeln $\langle \mathbf{S}, \mathbf{A}, T, R \rangle$ med tillståndsrummet $\mathbf{S}$, handlingsrummet $\mathbf{A}$, övergångsfunktionen T och belöningsfunktionen R [4]. Processen hoppar mellan tillstånd $s \in \mathbf{S}$ och i varje tillstånd väljs en handling $a \in \mathbf{A}$. Efter varje hopp får agenten en belöning eller bestraffning $r \in \mathbb{R}$ baserat på handlingen samt på tillstånden före och efter handlingen. För att beskriva vad som händer i systemet efter att en handling väljs definierar vi övergångsfunktionen $T : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \longrightarrow [0, 1]$. T ger sannolikheten att hamna i tillstånd s' om handling a väljs i tillstånd s. Vi kräver att

$$\sum_{s' \in \mathbf{S}} T(s, a, s') = 1 \quad \forall s \in \mathbf{S}, a \in \mathbf{A}$$

så att T blir en sannolikhetsfördelning över möjliga övergångar från tillstånd s givet handling a. Eftersom vi vill behandla ordnade sekvenser av handlingar behöver vi definiera tiden $t \in \mathbb{N}$. Vi betecknar därför ett systems tillstånd vid tid t som s_t och tillståndet i nästa tidssteg som s_{t+1} och så vidare. Detsamma gäller för handlingar; vi låter a_t beteckna den handling som utförs vid tid t.

En MDP kan alltså ses som en Markovprocess utökad med handlingar [4]. Vi formulerar därmed också Markovegenskapen för en MDP. Med övergångsfunktionen T definierad som ovan har vi att

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1}).$$

I detta arbete antar vi att de stokastiska processer vi behandlar är stationära. Givet ett tillstånd s_t innebär detta exempelvis att sannolikhetsfördelningen över vilka andra tillstånd s_{t+1} som kan nås från s_t ej beror av tiden.

För att modellera systemets svar på de handlingar som utförs i motsvarande tillstånd definierar vi en belöningsfunktion $R : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \longrightarrow \mathbb{R}$, där $r_t = R(s_t, a_t, s_{t+1})$. Den avbildar övergångar mellan specifika tillstånd på ett reellt tal. Viktigt i sammanhanget är att belöningsfunktionen inte nödvändigtvis belönar. Då belöningsfunktionen antar ett negativt värde indikerar det ett straff för en viss övergång. Genom att anpassa belöningsfunktionen kan ett visst beteende styras i exempelvis autonoma agenter som tränas genom en reinforcement learning-metod.

2.1.1 Policy och optimalitetsmodeller

För att bilda en strategi för att välja handlingar används en matematisk funktion π som kallas *policy* [4]. Målet är att hitta en optimal policy, det vill säga en policy som ger maximal belöning. Policyn är antingen *deterministisk* eller *stokastisk*. En deterministisk policy $\pi : \mathbf{S} \to \mathbf{A}$ ger en handling a för varje tillstånd s medan en stokastisk policy $\pi : \mathbf{S} \times \mathbf{A} \to [0, 1]$ ger sannolikheten att välja handling a i tillstånd s. Det betyder att i en deterministisk policy kommer alltid samma handling väljas i ett givet tillstånd och om policyn är optimal kommer alltid en optimal handling väljas. I samma modell där istället en stokastisk policy följs finns det en möjlighet att någon annan handling väljs, trots att agenten är medveten om vilken den optimala handlingen är.

För att avgöra hur bra en policy är kan en optimalitetsmodell användas. En sådan modell utgår från hur mycket belöning agenten får, men kan även ta hänsyn till när belöningen genereras. Två enkla men användbara optimalitetsmodeller är *ändlig-horisontmodell* och *diskonterad oändlig-horisontmodell* [4]. Här innebär ändlig horisont att vi tar hänsyn till vad som sker i processen under ett begränsat tidsintervall medan oändlig horisont betyder att beslutshorisonten är oändlig eller okänd. En ändlig-horisontmodell definieras av att den optimala policyn π^* är den som maximerar väntevärdet av summan av de h kommande belöningarna, det vill säga

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \left\{ \mathbb{E}\left[\sum_{t=0}^{h-1} r_t \right] \right\}, \tag{1}$$

där väntevärdet är taget med avseende på π och r_t är belöningen i tidssteg t. I en diskonterad oändlig-horisontmodell går $h \to \infty$ och för att summan då ska konvergera behövs en *diskonteringsfaktor* $\gamma \in [0, 1)$ introduceras. Den optimala policyn definieras då som den policy som maximerar väntevärdet av summan av alla *diskonterade* belöningar, det vill säga

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \left\{ \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \right\}. \tag{2}$$

Diskonteringsfaktorn γ har fler funktioner än att göra oändliga beslutsprocesser hanterbara. Den används exempelvis i pathfinding-problem, där en agent ska nå ett mål så snabbt som möjligt. Diskonteringsfaktorn tjänar då till att värdera korta vägar högre än långa.

2.1.2 Värdefunktioner och Bellmans ekvation

En *värdefunktion* är en funktion som uppskattar hur bra det är för en agent att befinna sig i ett specifikt tillstånd $s \in \mathbf{S}$, med utgångspunkt i att den vill maximera sin totala belöning [1]. Värdefunktioner betecknas $V^\pi(s)$ och representerar väntevärdet av den framtida belöningen då en

agent startar i tillstånd s och tar beslut enligt policyn π. Olika värdefunktioner fås beroende på vilken optimalitetsmodell som väljs [4]. Då vi har en ändlig-horisontmodell fås värdefunktionen

$$V^{\pi}(s_t) = \mathbb{E}\Big\{ \sum_{k=0}^{h-1} r_{t+k} | s_t \Big\} \tag{3}$$

och då vi istället har en diskonterad oändlig-horisontmodell blir värdefunktionen

$$V^{\pi}(s_t) = \mathbb{E}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t \Big\}, \tag{4}$$

där väntevärdet tagits utifrån policyn π. Som kan ses i (3) är värdefunktionen för en ändlig-horisontmodell en summa över ett ändligt antal tidssteg h, givet att agenten befinner sig i tillstånd s_t. På samma sätt beräknas värdefunktionen för en reducerad oändlig-horisontmodell enligt (4), med tillägget att antalet tidssteg som tas går mot ∞ samt att belöningen minskar med en diskonteringsfaktor γ för varje steg som tas i modellen. Om vi utvecklar värdefunktionen för den diskonterade oändlig-horisontmodellen får vi, givet en deterministisk policy π och starttillstånd s_t, *Bellmans ekvation*

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}\Big\{ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t \Big\} \\ &= \mathbb{E}\Big\{ r_t + \gamma V^{\pi}(s_{t+1}) | s_t \Big\} \\ &= \sum_{s_{t+1}} T\Big(s_t, a_t, s_{t+1}\Big)\Big(R(s_t, \pi(s_t), s_{t+1}) + \gamma V^{\pi}(s_{t+1}) \Big). \end{aligned}$$

Här vill vi påminna om att $r_t = R(s_t, a_t, s_{t+1})$ och att $\pi(s_t) = a_t$ eftersom policyn är deterministisk. Det blir här tydligt att värdefunktionen är definierad som väntevärdet av den direkta belöningen adderat med möjliga framtida värdefunktioner viktat med övergångssannolikheten. Eftersom en diskonterad modell används här multipliceras även värdefunktionerna med diskonteringsfaktorn γ för varje tillstånd. Då en optimal policy π^* följs fås $V^{\pi^*}(s)$ sådan att $V^{\pi^*}(s) \geq V^{\pi}(s)$ för alla $s \in \mathbf{S}$ och alla policyer π, där $V^{\pi^*}(s)$ kallas den *optimala värdefunktionen.*

Handlingsvärdefunktionen (action-value function) Q^{π} liknar värdefunktionen V^{π} men beror inte bara på vilket tillstånd agenten befinner sig i, utan även på vilken handling a den ska utföra i detta tillstånd [4] och definieras som

$$Q^{\pi}(s_t, a_t) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t, a_t \right\}. \tag{5}$$

Liksom det finns en optimal värdefunktion V^{π} finns det även en optimal handlingsvärdefunktion Q^{π^*} [4]. Dessa förhåller sig till varandra enligt

$$Q^{\pi^*}(s_t, a_t) = \sum_{s_{t+1}} T(s_t, a_t, s_{t+1})\Big(R(s_t, a_t, s_{t+1}) + \gamma V^{\pi^*}(s_{t+1}) \Big), \tag{6}$$

$$V^{\pi^*}(s_t) = \max_{a_t} \{ Q^{\pi^*}(s_t, a_t) \}. \tag{7}$$

Då reinforcement learning används för att lära en agent hitta ett mål är Q^{π^*} mer användbar än V^{π^*} eftersom ingen kunskap om miljöns dynamik behövs. Genom att låta en agent utforska miljön kan den bestämma Q^{π} utan att ha explicit kännedom om övergångsfunktionen eller belöningsfunktionen [4]. Utan att veta vilka tillstånd agenten senare kan hamna i, samt värdena i dessa tillstånd, kan optimala handlingar väljas [1].

2.1.3 Delvis observerbara miljöer

Det finns situationer då en agent inte har tillgång till all information om miljön som den befinner sig i. Det kan exempelvis bero på att miljön är för stor för att det ska vara möjligt att lagra

och göra beräkningar på informationen som den innehåller, eller att framtida tillstånd inte är kända. Det begränsade synfältet medför en osäkerhet då agenten tar beslut. Ett samlingsnamn för modeller där sådana situationer uppstår kallas *delvis observerbara Markov decision processes (partially observable Markov decision processes, POMDP)*.

POMDP definieras av tupeln $\langle \mathbf{S}, \mathbf{A}, T, R, \mathbf{\Omega}, O \rangle$. Här har vi utökat MDP:s tupel med observationsrummet $\mathbf{\Omega} = \{o_1, o_2, ..., o_k\}$, där $\mathbf{\Omega}$ är alla observationer som agenten totalt kan göra, och observationsfunktionen $O : \mathbf{S} \times \mathbf{A} \times \mathbf{\Omega} \to [0, 1]$ där

$$\sum_{o \in \mathbf{\Omega}} O(s', a, o) = 1, \quad \forall s', a$$

enligt [4]. Som för en MDP kommer en handling $a \in \mathbf{A}$ väljas för varje tillstånd $s \in \mathbf{S}$ som agenten befinner sig i. Denna handling får agenten att flytta sig till ett nytt tillstånd s' med sannolikheten $T(s, a, s')$, men till skillnad från i en MDP kommer inte tillståndet s' observeras i en POMDP. Istället görs observationer $o \in \mathbf{\Omega}$ som beror av s' och a med sannolikheten $O(s', a, o)$ [5]. Med observationerna kan agenten skapa sig en uppfattning om vart i miljön den befinner sig. Målet är sedan att maximera den totala belöningen, vilket görs utifrån den lokala informationen om miljön som agenten har tagit reda på. Ett sätt att hantera delvis observerbara miljöer är med neurala nätverk, som kommer att behandlas senare.

2.2 Reinforcement learning

Reinforcement learning (RL) är ett område inom maskininlärning där en eller flera agenter fattar beslut i en miljö och förbättras med hjälp av belöningar från systemet. Målet med RL är att optimera agentens val så att den kumulativa belöningen blir så stor som möjligt. Detta görs genom att iterativt förbättra agentens beteende. Reinforcement learning kan delas upp i två kategorier av metoder: värdebaserade och policybaserade metoder. Dessa skiljer sig främst i hur policyn tas fram.

2.2.1 Värdebaserade metoder

I *värdebaserade metoder* används en värdefunktion, antingen V^π eller Q^π, för att välja en handling [6]. Detta kan exempelvis göras genom att utvärdera $a = \text{argmax}_{\tilde{a}} Q(s, \tilde{a})$ i Q-learning som beskrivs i avsnitt 2.4. Här är målet att bestämma en värdefunktion och denna används sedan direkt för att välja handlingar.

En svaghet hos värdebaserade metoder är svårigheten att hantera stora eller kontinuerliga handlingsrum. Vid varje tidssteg behöver nämligen värdefunktionen maximeras med avseende på alla möjliga handlingar, vilket också kräver att värdefunktionen beräknas för varje tidssteg. En annan svaghet är att slumpen måste vävas in med en algoritm för att få agenten att utforska miljön och inte nöja sig med en suboptimal väg. Denna algoritm behöver optimeras för att få önskat beteende under träningsfasen.

2.2.2 Policybaserade metoder

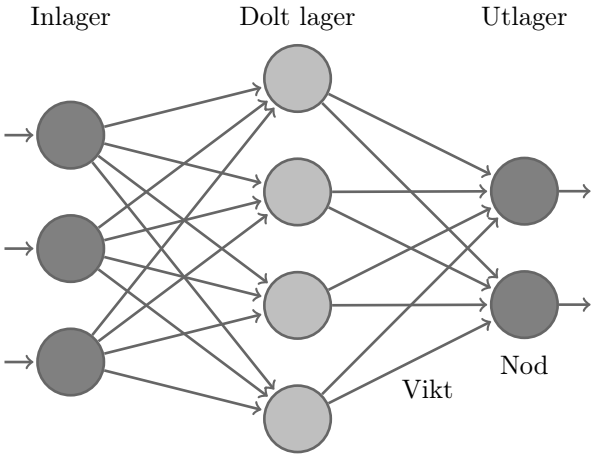
I *policybaserade metoder* är målet att bestämma en optimal policyn direkt, utan att maximera en värdefunktion. Här används en vektor av parametrar θ som förbättras iterativt genom att maximera en målfunktion, som är ett mått på hur väl policyn π presterar [7].

En viktig fördel med policybaserade metoder är att dessa algoritmer har bättre konvergensegenskaper än värdebaserade metoder. Medan värdebaserade metoders val av handling är känsligt för förändringar i värdefunktionen, är policybaserade metoder stabilare eftersom parametrarna endast uppdateras längs gradienten av målfunktionen. Detta sätt att förbättra modellen gör också att policybaserade metoder har potential att hantera stora och rentav kontinuerliga handlingsrum [1]. Med policybaserade metoder är det även möjligt att beräkna en stokastisk policy, som ger en sannolikhetsfördelning över de möjliga handlingarna som utdata. Då behöver inte förhållandet mellan att utforska miljön och att utföra de bästa handlingarna bestämmas manuellt. Detta sätt att involvera slump är bra eftersom slumpen är inbyggd i policyn som tränas och blir bättre med varje tidssteg. Nackdelar med policybaserade metoder är att de ofta konvergerar mot ett *lokalt* optimum för parametrarna samt att de konvergerar långsammare än värdebaserade metoder.

2.3 Artificiella neurala nätverk

Artificiella neurala nätverk (ANN) är ett samlingsbegrepp för en typ av funktionsapproximatorer som är inspirerade av biologiska neurala nätverk, som exempelvis den mänskliga hjärnan. Det som skiljer ANN från andra funktionsapproximatorer är deras förmåga att bli tränade till att själva generalisera och associera data [8]. Liksom biologiska neurala nätverk kan ANN tränas att lösa komplexa problem som ofta är mycket svåra att lösa med konventionella metoder, som exempelvis att klassificera handskrivna siffror [9].

ANN är uppbyggda av noder placerade i olika lager, där noderna är sammankopplade med varandra mellan angränsande lager, se figur 1. Neurala nätverk består alltid av ett indatalager och ett utdatalager, samt ett antal dolda lager av noder. Noderna och kopplingarna i nätet kan jämföras med neuronerna respektive synapserna i ett biologiskt neuralt nätverk och brukar därför ofta benämnas som artificiella neuroner. Varje nod har en eller flera ingående kopplingar och varje koppling har i sin tur en vikt. Då nätet tränas uppdateras dessa vikter och det är med dessa uppdateringar som nätet lär sig lösa ett problem [10].



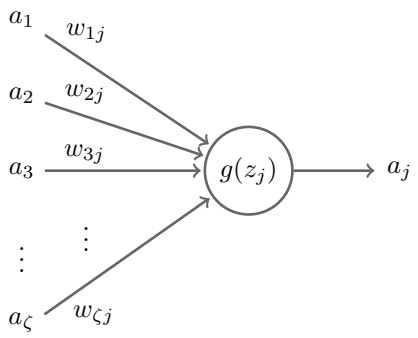
Figur 1: Ett artificiellt neuralt nätverk bestående av ett indatalager, ett utdatalager samt ett dolt lager av noder. Ett lager representeras av en kolumn av noder.

2.3.1 Aktivering och indata

Varje nod har ett tillstånd, också kallat en *aktivering*, som är ett reellt tal. Denna aktivering kan beskrivas som nodens interna representation av datan som skickats in i nätverket. En nods aktivering $a_j^{[l]}$, där j representerar vilken plats i lager $[l]$ noden befinner sig på, bestäms enligt

$$a_j^{[l]} = g\Big(\sum_{i=1}^{\zeta} a_i^{[l-1]} w_{ij}^{[l-1]} + b_j\Big), \tag{8}$$

där $a_i^{[l-1]}$ är aktiveringen från nod i i föregående lager $[l-1]$, som innehåller ζ noder. En godtycklig nod j i lager $[l]$ illustreras i figur 2. I (8) ser vi att ett lagers aktiveringar påverkas av tidigare lagers aktiveringar. Här har vi också introducerat *vikter* w_{ij} och *biaser* $b_j$som tillsammans utgör nätets parametrar. Bias används för att skifta aktiveringsfunktionen för att bättre modellera det problem som nätet försöker lösa. Vikterna kan sägas sitta på kopplingarna mellan noderna. Med hjälp av aktiveringsfunktionen $g(x)$ bestäms sedan aktiveringen för nod j i lager $[l]$. Det är vanligast att aktiveringsfunktionen är en ickelinjär funktion, där populära sådana är Fermifunktionen $g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ och ReLU $g(x) = \max(0, x)$ [8, 11].



Figur 2: Aktivering av en nod i lager $[l]$. Från lager $[l-1]$ går aktiveringarna a_i med vikter w_{ij} in i aktiveringsfunktionen g, vilket ger aktivering a_j i lager $[l]$. Här är $z_j = \sum_{i=1}^{\zeta} a_i w_{ij} + b_j$.

För att bestämma aktiveringarna för samtliga noder i lager $[l]$ kan ett system sättas upp. Om antalet noder i lager $[l-1]$ betecknas med ζ och antalet i lager $[l]$ betecknas med η, fås följande system:

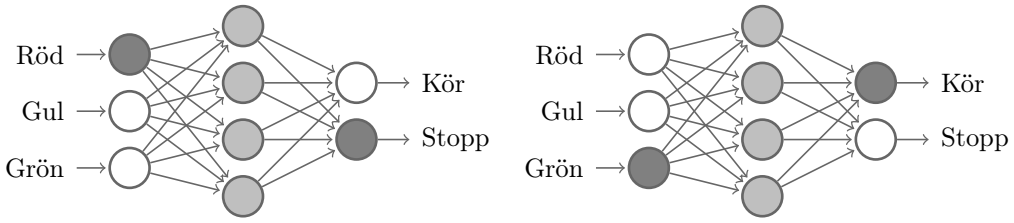
$$\begin{bmatrix} a_1^{[l]} \\ a_2^{[1]} \\ \vdots \\ a_\eta^{[1]} \end{bmatrix} = g\left(\begin{bmatrix} w_{11}^{[l-1]} & w_{12}^{[l-1]} & \dots & w_{1\eta}^{[l-1]} \\ w_{21}^{[l-1]} & w_{22}^{[l-1]} & \dots & w_{2\eta}^{[l-1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\zeta 1}^{[l-1]} & w_{\zeta 2}^{[l-1]} & \dots & w_{\zeta\eta}^{[l-1]} \end{bmatrix}^T \begin{bmatrix} a_1^{[l-1]} \\ a_2^{[l-1]} \\ \vdots \\ a_\zeta^{[l-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[l-1]} \\ b_2^{[l-1]} \\ \vdots \\ b_\eta^{[l-1]} \end{bmatrix} \right), \tag{9}$$

där aktiveringarna, vikterna samt bias för noderna i lager $[l-1]$ skapar aktiveringarna för lager $[l]$ med hjälp av aktiveringsfunktionen g.

2.3.2 Inlärning

Då vi har ett nätverk likt det i figur 1 låter vi den data som skickats in i nätverket propagera framåt genom nätverket. Lager för lager aktiveras noderna tills de i utdatalagret nås. Även dessa noder får en aktivering, vilket ger resultatet. Om nätets uppgift exempelvis är att klassificera bilder så representerar utdatalagrets noder olika typer av objekt som kan skickas in i nätverket som indata. Aktiveringarna i utlagret är då ett mått på hur troligt det är att bilden föreställer ett specifikt objekt.

När neurala nätverk skapas är de sällan välfungerande, vilket beror på att vikterna och biaserna inte har rätt värden. För att ett nätverk ska klara av att lösa ett specifikt problem, exempelvis det som nämndes ovan, måste värdena på vikterna och biaserna uppdateras. Nätverket *tränas* genom att vikterna optimeras så att nätverket löser en viss uppgift optimalt. Vad som menas med att lösa en uppgift optimalt skiljer sig från fall till fall. När ett nätverk exempelvis klassificerar siffror löser nätverket uppgiften optimalt då det klarar av att urskilja vilken siffra som skickas in i nätverket. I andra fall kan det handla om att avgöra om en bil ska stoppa eller köra vid ett trafikljus, se figur 3, eller något så abstrakt som att försöka skapa vacker musik [12]. Gemensamt för alla nätverk är dock att då utdatan jämförs med optimala utdatan ska denna skillnad vara så liten som möjligt. För att minimera denna skillnad används ofta *bakåtpropagering* vilket redogörs för i nästa avsnitt.



(a) Aktivering när ett trafikljus visar rött. (b) Aktivering när ett trafikljus visar grönt.

Figur 3: Ett nät som tränats att ta rätt beslut vid ett övergångsställe.

2.3.3 Bakåtpropagering

Vikterna och biaserna optimeras vanligtvis med metoden *bakåtpropagering*. Denna metod behöver ett mått på hur bra nätverket är, det vill säga hur väl nätverkets utdata förhåller sig till träningsmängdens verkliga utdata. För att mäta detta används en *förlustfunktion (loss function)* $L(\hat{y}, y)$ av nätverkets utdata y och träningsmängdens utdata $\hat{y}$. Det är vanligt att definiera $L(\hat{y}, y)$ som medelvärdet av felet i kvadrat för alla n aktiveringar i utdatalagret, det vill säga

$$L(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}^{(i)} - y^{(i)} \right)^2. \tag{10}$$

Med kostnadsfunktionen som mått på nätets prestanda, fås optimala parametrar genom att minimera L med avseende på dessa. Optimeringen görs med iterativa metoder, exempelvis *stokastisk gradient descent (SGD)* [13], där gradienten används för att minimera förlustfunktionen. Gradienten består av partialderivator av förlustfunktionen med avseende på biaserna och vikterna i det neurala nätverket. Vi låter $z_j^{[l]} = \sum_{i=1}^{n} a_i^{[l-1]} w_{ij}^{[l-1]} + b_j^{[l-1]}$ och beräknar partialderivatorna med kedjeregeln med avseende på vikterna i lager $[l]$ enligt

$$\frac{\partial L}{\partial w_{ij}^{[l]}} = \frac{\partial z_j^{[l]}}{\partial w_{ij}^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial L}{\partial a_j^{[l]}}. \tag{11}$$

Vidare beräknas partialderivatorna med avseende på aktiveringsfunktionen, där lager $[l]$ har η noder, som

$$\frac{\partial L}{\partial a_i^{[l-1]}} = \sum_{j=1}^{\eta} \frac{\partial z_j^{[l]}}{\partial a_i^{[l-1]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \frac{\partial L}{\partial a_j^{[l]}}. \tag{12}$$

Detta är en iterativ process som börjar i utdatalagret och sedan propagerar bakåt i nätverket. Den totala gradienten, för ett nät med N lager, kan skrivas som

$$\nabla L = \left[\frac{\partial L}{\partial W^{[1]}}, \frac{\partial L}{\partial b^{[1]}}, \frac{\partial L}{\partial W^{[2]}}, \frac{\partial L}{\partial b^{[2]}}, \ldots, \frac{\partial L}{\partial W^{[N]}}, \frac{\partial L}{\partial b^{[N]}} \right]^T, \tag{13}$$

där $W^{[l]}$ är viktmatrisen med element $w_{ij}^{[l]}$ och $b^{[l]}$ är biasvektorn för lager $[l]$. I SGD bestäms storleken på uppdateringen av den så kallade *hyperparametern* α, som är inlärningshastigheten. Till skillnad från vikterna och biaserna som uppdateras under körning är hyperparametrar värden som bestäms innan träningen börjar.

2.4 Q-learning

Q-learning är en värdebaserad metod som används i RL för att lära en agent ta de beslut som i slutändan maximerar belöningen. I pathfinding-problemet betyder det att agenten tar sig från en startpunkt till en slutdestination så snabbt som möjligt, genom att lära sig den optimala handlingsvärdefunktionen given i (6).

I sin mest grundläggande form använder Q-learning någon form av uppslagstabell för att spara de Q-värden som handlingsvärdefunktionen beräknat för miljöns olika tillstånd och handlingar.

För att använda en tabell på detta sätt behöver alla tillstånd i miljön vara diskreta samt kända för agenten. Detta begränsar användningsområdet till små tillståndsrum, eftersom metoden annars blir för beräkningstung. I tabellen finns ett värde $Q(s,a)$ för varje tillståndhandlingspar (s,a), och dessa värden uppdateras stegvis baserat på de belöningar som agenten får av miljön. Då Q_k uppdateras till Q_{k+1} görs detta med

$$Q_{k+1}(s_t,a_t) = Q_k(s_t,a_t) + \alpha\Big(r_t + \gamma \max_{\tilde{a}_t}\{Q_k(s_{t+1},\tilde{a}_t)\} - Q_k(s_t,a_t)\Big), \tag{14}$$

där agenten vid tiden t förflyttar sig från tillstånd s_t till s_{t+1} då handlingen a_t utförs. Hur stor vikt som ska läggas vid nya upptäckter och hur mycket det ursprungliga Q-värdet ska tas i beaktning när det nya värdet bestäms avgörs av inlärningshastigheten α. Inlärningshastigheten ligger i intervallet $0 \leq \alpha < 1$, där ett värde nära 1 beräknar det nya Q-värdet med stor hänsyn till det sista steget som tagits. Då $\alpha = 0$ förblir Q-värdet detsamma.

Avvägningen mellan *utforskning (exploration)* och *utnyttjande (exploitation)* är alltid närvarande i en självinlärningsalgoritm. Med utforskning menas att algoritmen med fördel tar reda på mer information om den miljö som den befinner sig i genom att utforska många olika handlingar och tillstånd. I motsats till utforskning kommer en metod baserad på utnyttjande optimera lösningen utifrån de tillstånd som är kända. För att en algoritm ska hitta den optimala lösningen krävs en viss utforskning, eftersom den optimala lösningen troligtvis inte upptäcks annars. Ju längre in i träningsfasen algoritmen kommer måste dock utnyttjande bli allt viktigare, eftersom det blir allt mer sannolikt att den optimala lösningen då redan funnits. Q-learning sägs vara *utforskningsokänslig (exploration insensitive)* [14]. Det betyder att oavsett vilken utforskningspolicy som följs kommer den konvergera till den optimala policyn π^* så länge varje tillståndhandlingspar i träningsfasen besöks oändligt många gånger. Detta följer av konvergenssatsen, se Sats 1 [15].

Sats 1 *Givet att Q-värdena uppdateras enligt* (14) *och att en begränsad belöning* $|r_n| \leq K$, $K \in \mathbb{R}$, *inlärningshastigheten* $0 \leq \alpha_n < 1$ *och*

$$\sum_{i=1}^{\infty} \alpha_{n^i(s,a)} = \infty, \qquad \sum_{i=1}^{\infty} \alpha^2_{n^i(s,a)} < \infty, \quad \forall s \in \mathbf{S}, \quad \forall a \in \mathbf{A},$$

uppfylls, där $n^i(s,a)$ *är indexet för i:te gången handling* a *utförs i tillstånd* s, *då gäller att* $Q_n(s,a) \to Q^*(s,a)$ *då* $n \to \infty$ $\forall s \in \mathbf{S}, \forall a \in \mathbf{A}$, *där* $Q^*(s,a)$ *är den optimala policyn.*

2.5 Policygradientmetoder

Ett alternativt tillvägagångssätt för att hitta en optimal policy är att låta en stokastisk policy representeras av någon funktionsapproximator. Vi ställer sedan upp ett optimeringsproblem där målfunktionen är något mått på hur policyn presterar, exempelvis V^π. Approximatorn bestäms av en uppsättning parametrar θ, exempelvis vikterna i ett neuralt nätverk, och det är med avseende på dessa som vi optimerar policyn. Vi söker alltså metoder som ger oss möjlighet att gradvis förbättra policys för att de ska ge bättre avkastning [6].

Givet en stokastisk policy π har vi att

$$V^\pi(s_0) = \int_{\mathbf{S}} \rho^\pi(s) \int_{\mathbf{A}} \pi(s,a) \int_{s' \in \mathbf{S}} T(s,a,s') R(s,a,s') \, ds' da ds \tag{15}$$

där

$$\rho^\pi(s) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}\{s_t = s | s_0, \pi\}$$

definieras som den diskonterade tillståndsfördelningen [6]. Om vi låter π_θ beteckna en deriverbar policy representerad av en funktionsapproximator som parametriseras av θ kan vi skriva

$$\nabla_\theta V^{\pi_\theta}(s_0) = \int_{\mathbf{S}} \rho^{\pi_\theta}(s) \int_{\mathbf{A}} \nabla_\theta \pi_\theta(s,a) Q^{\pi_\theta}(s,a) da ds \tag{16}$$

där $\nabla_\theta V^{\pi_\theta}$ betecknar gradienten av värdefunktionen med avseende på parametrarna θ. Genom att skatta gradienten av målfunktionen kan vi beräkna uppdateringar av parametrarna θ då vi kan låta skillnaden i θ vara proportionell mot $\nabla_\theta V^{\pi_\theta}$, det vill säga $\Delta\theta \propto \nabla_\theta V^{\pi_\theta}$.

Integralen i (16) är svårberäknad och behöver således skattas. En metod för att åstadkomma detta fås genom att

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a)\nabla_\theta\big(\ln\left(\pi_\theta(s,a)\right)\big)$$

och det gäller då att

$$\nabla_\theta V_Q^{\pi_\theta}(s_0) = \mathbb{E}\big[\nabla_\theta\big(\ln(\pi_\theta(s,a))\big)Q^{\pi_\theta}(s,a)\big], \tag{17}$$

där s och a kommer från fördelningarna ρ^{π_θ} repsektive π_θ. Detta ger oss en möjlighet att skatta gradienten $\nabla_\theta V^{\pi_\theta}$ så att parametrarna θ kan uppdateras i önskad riktning. En konceptuell tolkning är att vi uppdaterar våra parametrar så att policyn π_θ ger oss större sannolikhet att utföra handlingar med högre förväntad avkastning, då vi ju har valt Q^{π_θ} som ett mått på förväntad framtida avkastning.

En möjlig förbättring av gradientestimatorn fås genom att använda en *advantage*-funktion

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \tag{18}$$

istället för Q^{π_θ} i (17). Med detta uttryck definierar vi policygradienten som

$$\nabla_\theta V_A^{\pi_\theta}(s_0) = \mathbb{E}\big[\nabla_\theta\big(\ln(\pi_\theta(s,a))\big)A^{\pi_\theta}(s,a)\big]. \tag{19}$$

Funktionen A^{π_θ} blir ett mått på hur bra det är för agenten att utföra en specifik handling a i tillstånd s jämfört med det förväntade värdet på att utföra någon av alla andra möjliga handlingar. Typiskt för denna storhet är att dess magnitud typiskt är lägre än $Q^{\pi_\theta}(s,a)$, vilket har flera fördelar [6].

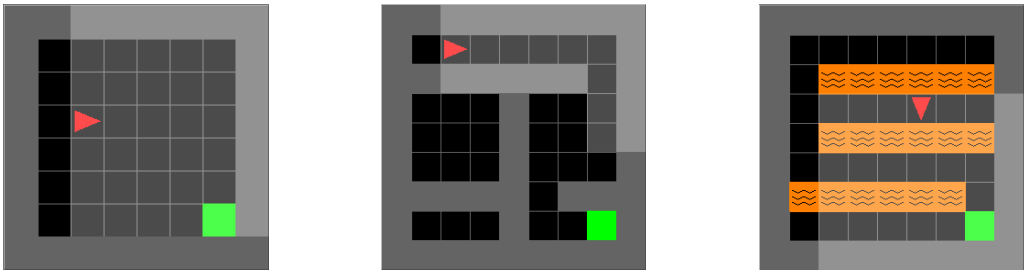
Eftersom en uppskattning av någon värdefunktion Q^{π_θ} eller A^{π_θ} krävs för att skatta gradienten ovan brukar en så kallad *actor-critic*-struktur användas i policygradientmetoder. Exempelvis definierar man två neurala nätverk, där det ena representerar policyn och utför handlingar (actor). Det andra skattar värdefunktionen och utvärderar den policy som följs (critic). Actor-nätverkets parametrar uppdateras då med hjälp av policygradienten enligt (19). Critic-nätverket uppdateras på samma sätt som de metoder som står att läsa om i avsnitten 3.2 respektive 3.3. Genom att critic-nätverket tränas på att uppskatta en värdefunktion får vi en möjlighet att beräkna A^{π_θ} i (19). Denna uppskattning kan sedan användas för att uppdatera actor-nätverkets vikter mot en optimal policy.

3 Implementering

I detta avsnitt behandlas algoritmer för att lösa pathfinding-problem. De presenteras i den ordning de implementerades under arbetets gång, vilket även till stor del sammanfaller med hur avancerade de är. Först presenteras algoritmen Q-table, som sparar Q-värden i en tabell. Detta följs av Q-network som också använder Q-värden men som får dessa från neurala nätverk utan dolda lager. Efter detta behandlas Deep Q-network som använder sig av neurala nätverk med flera dolda lager. Tre varianter av denna metod presenteras. Avslutningsvis behandlas Proximal Policy Optimization, som är en policybaserad metod.

Ett centralt begrepp är *episoder*, som är den tidsperiod som startar när agenten är vid startpunkten och slutar antingen när agenten nått målet eller när antalet steg nått en maxgräns. Vissa agenter har tränats i delvis observerbara miljöer. Detta innebär att agenten har ett begränsat, rektangulärt synfält, som förflyttar sig med agenten, se figur 4.

Den färdiga miljön MiniGrid [16] har använts för att träna, utvärdera och visualisera algoritmerna. MiniGrid är ett ramverk med ett antal olika rutnät innehållande olika komponenter som start, mål, väggar och lava, se figur 4. Agenten kan inte gå igenom väggarna och episoden avslutas omedelbart om den går in i lavan. Utöver denna miljö skapades ett ramverk med två syften. Dels fungerar ramverket som en bro mellan algoritmerna och miljön så att algoritmerna inte behöver anpassas till MiniGrid. Dessutom underlättar ramverket träning och utvärdering av algoritmerna samt visualisering av körningar.



(a) En miljö utan hinder. (b) En miljö med väggar. (c) En miljö med lava.

Figur 4: Tre exempel på miljöer i MiniGrid. Agenten representeras av en pil och slutmålet av en ruta i nedre högra hörnet. Som hinder finns väggar vilka representeras av mörkgråa rutor, se (b), och lava som representeras av ifyllda rutor med vågor, se (c). Det ljusare partiet är agentens synfält.

3.1 Q-table

Den första algoritmen, vilken också är mest rättfram av de som behandlas här, är *Q-table*. Denna algoritm är baserad på Q-learning, se avsnitt 2.4, där en matris $\mathbb{Q}$ håller informationen om hur fördelaktigt det är att gå från ett tillstånd i miljön till ett annat. Det är därför lätt att följa algoritmens utveckling eftersom alla värden kan hämtas i $\mathbb{Q}$. Matrisens storlek beror av hur stort tillståndsrummet $\mathbf{S}$ och handlingsrummet $\mathbf{A}$ är, där antalet rader och kolonner är lika många som $\mathbf{S}$ respektive $\mathbf{A}$ är stora. $\mathbb{Q}$ består här av tre kolonner eftersom $\mathbf{A}$ utgörs av handlingarna att vrida sig medurs, vrida sig moturs och att ta ett steg rakt fram. $\mathbf{S}$ kan beskrivas som (miljöns bredd) $\times$ (miljöns höjd) $\times$ 4, eftersom tillståndsrummet består av alla tillstånd i miljön som en agent kan befinna sig i, multiplicerat med alla riktningar agenten kan peka åt i dessa tillstånd, det vill säga nord, syd, väst och öst.

3.1.1 Träning av algoritmen

Alla värden i $\mathbb{Q}$ sätts ursprungligen till noll och när algoritmen sedan tränas kommer dessa värden uppdateras. Uppdatering av $\mathbb{Q}$ sker genom handlingsvärdefunktionen

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\Big(r + \gamma \max_{\tilde{a}}\{Q(s',\tilde{a})\}\Big), \tag{20}$$

där s' är det tillstånd som fås då handling a utförs i tillstånd s och $r = R(s,a,s')$ är belöningen. Ett lämpligt val av α krävs här för att algoritmen ska ta hänsyn till tidigare erfarenheter, samtidigt som nya upptäckter värdesätts. Beräkningen i (20) upprepas i varje nytt tillstånd agenten hamnar i, tills agenten antingen når målet eller antalet handlingar uppnår en satt maxgräns, varpå episoden tar slut. När träningen är klar är det därför möjligt för en agent att med hjälp av $\mathbb{Q}$ bestämma handling.

3.1.2 Tillägg för utforskning av hela miljön

Om agenten tar samma väg varje episod under träningsfasen kommer med största sannolikhet inte den optimala vägen hittas. De värden i $\mathbb{Q}$ som först blir positiva kommer besökas av agenten varje gång en ny episod startar, eftersom de tillstånd som dessa värden motsvarar är de enda som räknas som gynnsamma. För att få agenten att utforska nya vägar införs $\epsilon \in (0,1)$, vilket är sannolikheten för att en slumpmässig handling ska väljas istället för den optimala. Denna strategi kallas ϵ-*greedy*. Med detta tillägg tvingas agenten utforska även de delar av miljön som först inte verkar gynnsamma och risken att fastna i en slinga där endast samma del av miljön utforskas undviks. Vanligtvis minskar ϵ med ett konstant värde inför varje ny episod, vilket betyder att ju längre in i träningsfasen algoritmen kommer, desto större är sannolikheten att den mest fördelaktiga handlingen väljs. Algoritmen är utformad på detta sätt eftersom det längre in i träningsfasen är mer sannolikt att den optimala vägen redan upptäckts. Algoritmen för Q-table illustreras i Algoritm 1.

Algoritm 1 Q-table

Initialize:
$\mathbb{Q}(s, a)$
for each episode **do**
Choose an action a, from state s using policy derived from $\mathbb{Q}$ (e.g. ϵ-greedy)
for each step in episode **do**
Take action a, observe r and s'
Choose an action a', from state s' using policy derived from $\mathbb{Q}$ (e.g. ϵ-greedy)
$\mathbb{Q}(s, a) \leftarrow \mathbb{Q}(s, a) + \alpha \left[r + \gamma \mathbb{Q}(s', a') - \mathbb{Q}(s, a) \right]$
$s \leftarrow s'$
if s is terminal **then**
break
end if
end for
end for

3.1.3 Resultat och analys

Q-table presterade väl på relativt enkla problem, till exempel rutnät utan hinder. För ett sådant problem på ett förhållandevis litet tillståndsrum kunde metoden konvergera efter bara något hundratal episoder. Teoretiskt är det fastställt att algoritmen kommer konvergera mot en optimal policy i ändlig tid enligt 1, men det kunde bara verifieras för enkla problem i fullt observerbara miljöer.

På grund av att algoritmen kräver att handlingsvärdefunktionen kan approximeras av $\mathbb{Q}$ i samtliga av omgivningens tillstånd, har den mycket svårt att klara av problem som modelleras som POMDP. Detta leder som bekant till att stora problem kan bli mycket svårberäknade, då komplexiteten växer snabbt med tillståndsrummets storlek.

3.2 Q-network

Nästa algoritm som behandlas är *Q-network*. Istället för att spara de värden som representerar hur bra det är att gå från ett tillstånd till ett annat i en matris $\mathbb{Q}$, sätts denna information in i ett ANN. Att använda ett ANN istället för en matris är fördelaktigt när den miljö som en agent ska verka i är stor. Redan i relativt små tillståndsrum får Q-table problem att lösa pathfinding-problemet inom rimlig tid, då denna algoritm mycket snabbt blir beräkningstung. Q-network är även grundpelaren för senare algoritmer och det är därför viktigt att förstå denna algoritms uppbyggnad innan den utvecklas ytterligare.

3.2.1 Träning av algoritmen

Q-network består av ett nätverk uppbyggt av ett indatalager och ett utdatalager. För att få Q-värdena representeras endast tillståndsrummet $\mathbf{S}$ av en matris. Denna matris propagerar framåt i nätverket, vilket aktiverar utlagrets noder. Dessa noders aktiveringar motsvarar i sin tur Q-värden för olika handlingar a. Det är värt att nämna att Q-network kan hantera kontinuerliga tillståndsrum men den används i dessa pathfinding-problem endast för diskreta tillståndsrum. Här skiljer sig denna algoritm från Q-table, som endast kan hantera diskreta tillståndsrum.

Precis som i Q-table väljs under träningsfasen en handling med sannolikheten ϵ för att algoritmen ska utforska nya vägar och att förlustfunktionen inte ska fastna i lokala minima. Efter att en handling a har valts och nästa tillstånd samt dess belöning hämtats, är följande steg mycket likt det som sker i Q-table. Istället för att uppdatera värdena i matrisen $\mathbb{Q}$ med handlingsvärdefunktionen i (20), används istället en förlustfunktion. I förlustfunktionen sätts ett målvärde in tillsammans med det ursprungliga Q-värdet. Målvärdet fås med

$$Q(s, a) = r + \gamma \max_{\tilde{a}} \{Q(s', \tilde{a})\}, \tag{21}$$

där $Q(s, a)$ inte ska förväxlas med matrisen $\mathbb{Q}$. Här används förlustfunktionen (10) som är det genomsnittliga kvadratfelet. För att vikterna mellan noderna i nätverket ska uppdateras, se avsnitt 2.3, bakåtpropageras felet som förlustfunktionen beräknat. Genom att minimera felet med

en optimeringsfunktion, här SGD, fås en ändring av vikterna. Detta görs många gånger och när tillräckligt mycket data passerat genom nätverket så stabiliseras vikterna mellan noderna. Till slut fås en tränad algoritm som kan lösa pathfinding-problemet i en helt synlig miljö.

3.2.2 Resultat och analys

Liksom Q-table gav denna algoritm goda resultat för enklare problem i fullt observerbara miljöer. Intressant att notera är att de möjliga tillstånd agenten hittade och hur den där skulle agera optimalt inte representerades av en matris, utan av noderna och deras vikter i det neurala nätverket. Ett linjärt lager i det neurala nätverket räckte för att uppnå lika goda resultat som Q-table i en liten miljö utan hinder.

Algoritmen kräver med vår uppsättning av parametrar likt Q-table ett hundratal episoder för att hitta en optimal policy. Vinsten med att använda denna metod är dock att hela tillståndet inte behöver representeras i en matris. Detta betyder att stora effektiviseringar kan göras sett till komplexitet för större problem. Med vetskapen att neurala nät effektivt kan representera agentens kunskap om sin omgivning i sina interna tillstånd och således avbilda par av tillstånd och handlingar på förväntad avkastning, bör en algoritm kunna konstrueras för delvis observerbara miljöer.

3.3 Deep Q-network

I sin artikel från 2015 introducerade Mnih m. fl. begreppet *deep Q-network (DQN)* [2]. Här representeras agenten av ett djupt neuralt nätverk som syftar till att avbilda agentens tillstånd och dess möjliga handlingar på ett Q-värde, liknande de ovan nämnda metoderna. Artikeln beskriver en agent som lär sig spela klassiska spel till systemet Atari 2600, vilket den i många fall gör bättre än tidigare algoritmer och professionella mänskliga speltestare.

DQN implementerades på delvis observerbara miljöer. Istället för att agenten känner hela systemets tillstånd i alla tidssteg får den observationer av systemet som svarar mot dess synfält. Metoden bygger på att neurala nätverk kan approximera den optimala handlingsvärdefunktionen för ett tillståndhandlingspar (s, a). Precis som Q-network är DQN baserad på ANN men har till skillnad från Q-network tre dolda lager. Det används även två nätverk, ett *policynätverk (policy net)*, Q^{policy}, som används för att hämta ut Q-värdena samt ett *målnätverk (target net)*, Q^{target}, som används för att träna nätverket och hämta Q-målvärden som i (21). Efter en period synkroniseras nätverken med varandra. Detta görs inte kontinuerligt eftersom det är fördelaktigt att ha ett fixt Q-målvärde. På så sätt undviks att policynätverket ivrigt försöker nå Q-värden som hela tiden ändras, vilket skulle vara fallet om man uppdaterar nätverket vid varje träningsiteration. Det introduceras även ett *replay memory* $\mathcal{D}$, där det samlas *övergångar*. Övergången, $\langle s_t, a_t, s_{t+1}, r_t \rangle$, beskriver vilket tillstånd s_{t+1} agenten hamnar i samt vilken belöning r_t agenten får för att utföra handlingen a_t i tillstånd s_t. Vid träning hämtas ett antal slumpvist utvalda övergångar från $\mathcal{D}$ i form av en batch vilket har visats förbättra träningsprocessen och stabiliteten för DQN markant [17]. Algoritmen för DQN visas i algoritm 2.

3.3.1 Double DQN

DQN använder samma nätverk för att välja en handling som för att bestämma dess Q-värde. Detta gör att Q-värdena lätt kan bli överskattade [17], vilket resulterar i brusiga och mer slumpartade resultat. Ett sätt att minska bruset under träningen är att använda policynätverket för att hitta vilken handling som ger högst Q-värde och använda den handlingen i målnätverket för att till slut hitta Q-värdet. Detta gör att (21) istället ser ut som

$$Q(s, a) = r + \gamma Q^{\text{target}}\Big(s', \underset{\tilde{a}}{\text{argmax}}\{Q^{\text{policy}}(s', \tilde{a})\}\Big). \tag{22}$$

Här används policynätverket för att ta fram Q^{policy} respektive målnätverket för att få fram Q^{target}. De övriga parametrarna är som i (21). Denna ändringen i algoritmen kallas för DDQN (Double Deep Q-network).

Algoritm 2 DQN

Initialize:
replay memory $\mathcal{D}$
Initialize:
action-value function Q^{policy} with random parameters
Initialize:
target action-value function $Q^{\text{target}} = Q^{\text{policy}}$
for episode $= 1, 2, \ldots, M$ **do**
for $t = 1, 2, \ldots, \tau$ **do**
With probability ϵ select a random action a_t
otherwise select $a_t = \text{argmax}_a \, Q^{\text{policy}}(s_t, a)$
Execute action a_t in state s_t and observe reward r_t and next state s_{t+1}
Store transition $\langle s_t, a_t, r_t, s_{t+1} \rangle$ in $\mathcal{D}$
Sample random mini-batch of transitions $\langle s_j, a_j, r_j, s_{j+1} \rangle$ from $\mathcal{D}$
Set $y_j = \begin{cases} r_j, & \text{if episode terminates at j+1} \\ r_j + \gamma \max_{\tilde{a}} \left\{ Q^{\text{target}}\left(s_{j+1}, \tilde{a}\right) \right\}, & \text{otherwise} \end{cases}$
Optimize parameters in Q^{policy} on the loss function $L(y_j)$
Every 1000 step set $Q^{\text{target}} = Q^{\text{policy}}$
end for
end for

3.3.2 Deep recurrent Q-network

För att åtgärda problemen som DQN stöter på i samband med övergången till POMDP, utökas det bakomliggande neurala nätverket med en *återkopplande (recurrent)* komponent. Genom att låta ett återkopplande nätverks interna tillstånd propagera tillsammans med indatan flera gånger genom nätverket, kan det dra lärdomar ur sekventiell data. Tanken är här att en *long short-term memory cell (LSTM-cell)* kan hjälpa nätverket minnas sekvenser ur dess samlade erfarenhet under träningen [18]. Metoden bygger på en artikel där författarna introducerade just en LSTM-cell i ett nätverk för DQN [19]. De insåg att även om DQN presterar mycket bra i många spel, brister metoden där en långsiktig strategi belönas. De påstår att i dessa spel kan det fullständiga tillståndet ej erhållas genom att bara se en batch av bildrutor. Därför introduceras ett minne i metoden för att göra den mer robust mot de egenheter som kan uppstå när ett delvis observerbart problem behandlas.

Den huvudsakliga skillnaden mot implementationen av DQN, förutom det neurala nätverkets struktur, blir att replay memory modifieras. Istället för att spara undan enstaka övergångar sparas hela episoder, sedan dras slumpmässigt *sekvenser (traces)* ur dessa som nätverket tränar på. För en episod E av längd n ges alltså att

$$E = \bigcup_{i=1}^{n} \langle s_i, a_i, s'_i, r_i \rangle.$$

Ur episoder kan sedan sekvenser C med längd $k - m$ dras slumpmässigt som

$$C = \bigcup_{j=m}^{k} \langle s_j, a_j, s'_j, r_j \rangle$$

för något slumpmässigt valt heltal $m \in [1, n - k]$. Batches av sekvenser används sedan för att träna nätverket. Tanken är att återkopplingen i det neurala nätverket ska lära sig sekvenser av övergångar som leder till god avkastning.

3.3.3 Resultat och analys

Under samlingsbegreppet DQN finns ett antal metoder, däribland de som nämns ovan. I detta projekt har övergången till djupa neurala nätverksmodeller också inneburit en övergång till delvis observerbara problem. De policys som tagits fram med DQN har inte visat på ett övertygande

sätt att de kan lösa pathfinding-problem i delvis observbara miljöer, då de presterat lika bra eller sämre än en helt slumpmässig policy. Det finns flera sannolika förklaringar till att algoritmerna inte har lyckats lära sig optimala policys. Typiskt kan metoder som involverar neurala nätverk vara känsliga för vilka hyperparametrar som väljs, samt för antalet lager och noder som finns i nätverket. Det finns heller inga väl underbyggda metoder för att optimera dessa; heuristik är det vanligaste sättet att hitta dem. Implementeringen av DRQN i detta projekt har inte lett till resultat som pekat mot att problemen med DQN kan lösas med återkopplande neurala nätverk för problemen som behandlats. Här spelar återigen parameterval och nätverksstruktur stor roll, vilket gör att det i mån av tid mycket väl kan finnas möjlighet att hitta lösningar. Slutligen kan DQN och dess varianter endast ta fram deterministiska policys, vilket skulle kunna vara en begränsande faktor.

3.4 Proximal Policy Optimization

Utöver de värdebaserade metoderna som tidigare diskuterats finns det en familj av RL-algoritmer som är policybaserade, som till exempel policygradientmetoder som behandlades i avsnitt 2.5. En av de nyare metoderna i denna familj är *Proximal Policy Optimization (PPO)*, som introducerades i artikeln Proximal Policy Optimization Algorithms [20]. Författarna menar att PPO förbättrar effektiviteten och robustheten i jämförelse med andra policygradientmetoder, samtidigt som algoritmen ska vara relativt lätt att implementera. Den har också hög prestanda både när det kommer till resultat och träningshastighet.

Som tidigare nämnt i avsnitt 2.5 försöker samtliga policygradientmetoder uppskatta policyns gradient, se (19), och använda denna för att uppdatera policyn med en viss strategi. Gradienten kan också uppskattas genom att derivera någon förlustfunktion, till exempel

$$L^{PG}(\theta) = \mathbb{E}\Big[\, \ln \pi_\theta\,(a \,|\, s)\, A^{\pi_\theta} \,\Big] \tag{23}$$

där A^{π_θ} definieras som i (18). Det har dock visat sig vara problematiskt att få policygradientmetoder som använder sig av L^{PG} som förlustfunktion att konvergera till den optimala policyn. Antingen sätts inlärningshastigheten till ett för lågt värde, vilket gör att konvergens aldrig inträffar, eller så sätts den till ett för högt värde så att metoden gör för stora och destruktiva uppdateringar. I ett försök att få bukt med dessa problem togs förlustfunktionen

$$L^{CPI}(\theta) = \mathbb{E}\left[\, \frac{\pi_\theta\,(a \,|\, s)}{\pi_{\theta_{\text{old}}}\,(a \,|\, s)}\, A^{\pi_\theta} \,\right] \tag{24}$$

fram, där $\pi_{\theta_{\text{old}}}$ är föregående policy. L^{CPI} kombinerat med en viss uppdateringsstrategi kallas *Conservative Policy Iteration (CPI)* [21]. Med definitionen $\phi(\theta) = \frac{\pi_\theta\,(a \,|\, s)}{\pi_{\theta_{\text{old}}}\,(a \,|\, s)}$ kan L^{CPI} skrivas på den förenklade formen

$$L^{CPI}(\theta) = \mathbb{E}\Big[\, \phi(\theta)\, A^{\pi_\theta} \,\Big]. \tag{25}$$

Utan någon begränsning på uppdateringens storlek lider CPI av samma problem som många andra policygradientmetoder med långsam eller instabil träning. PPO använder sig av en liknande gradientuppskattning som CPI, men med ett inbyggt skydd mot för stora destruktiva uppdateringar av policyn genom sin förlustfunktion. Funktionen introducerades i artikeln som *Clipped Surrogate Objective*, vilket hädanefter kommer refereras till som PPO:s målfunktion. [20]

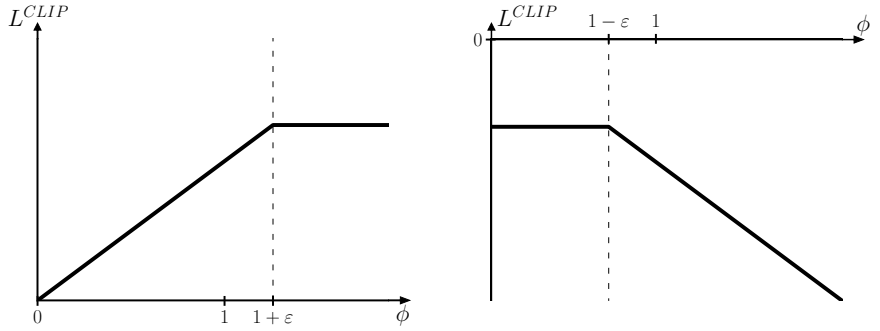
3.4.1 Målfunktion

Andra policygradientmetoder har med varierande framgång försökt förhindra destruktiva uppdateringar av policyn, genom att maximera målfunktionen under vissa bivillkor [22]. PPO har istället valt en annan väg där begränsningen av storleken på uppdateringen görs direkt i dess målfunktion. Denna kallas L^{CLIP} efter engelskans *clipping* och har formen

$$L^{CLIP}(\theta) = \mathbb{E}\Big[\min\big(\phi(\theta)\, A^{\pi_\theta},\ \text{clip}(\phi(\theta),\, 1-\varepsilon,\, 1+\varepsilon)\, A^{\pi_\theta}\big) \,\Big]. \tag{26}$$

Den första termen i L^{CLIP} kommer från L^{CPI}, i (25). Den andra termen begränsar $\phi(\theta)$ så att $\phi(\theta) \in [1-\varepsilon,\, 1+\varepsilon]$. Genom att ta ett minimum av dessa termer fås en förlustfunktion där

uppdateringen av policyn begränsas för vissa fall, genom att gradienten blir noll där målfunktionen är konstant. Hyperparametern ε sätts ofta till 0.2 men utrymme för optimering mot specifika problem finns [20]. Motivationen till L^{CLIP} förstås lättast genom att behandla olika fall: då $A^{\pi_\theta} > 0$



(a) $A^{\pi_\theta} > 0$. (b) $A^{\pi_\theta} < 0$.

Figur 5: Målfunktionen L^{CLIP} mot ϕ, för olika tecken på A^{π_θ}.

och då $A^{\pi_\theta} < 0$. Om en handling är bra, det vill säga $A^{\pi_\theta} > 0$, och handlingen blir mindre sannolik, $\phi(\theta) < 1$, tillåts policyn att korrigeras fritt genom en uppdatering. Då istället en handling blir mer sannolik, $\phi(\theta) > 1$, begränsas vissa uppdateringar av policyn så att inte ett för stort steg tas, mer specifikt de uppdateringar där $\phi(\theta) > 1 + \varepsilon$. Begränsningarna syftar till att förhindra destruktiva uppdateringar som skulle försätta policyn i ett icke önskvärt tillstånd.

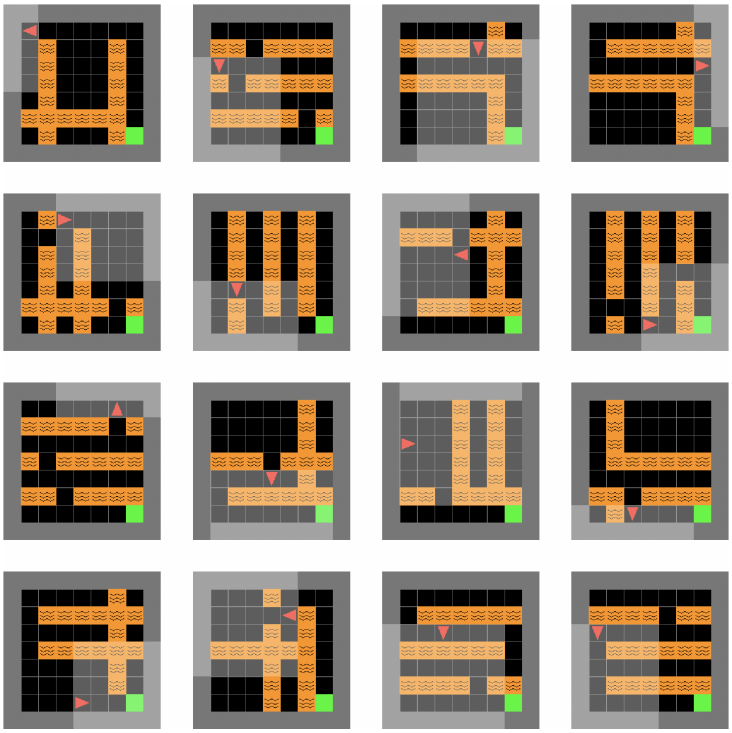
Om en handlingen skulle visa sig vara dålig, det vill säga då $A^{\pi_\theta} < 0$, fås två andra fall. Då handlingen blir mer sannolik kan tidigare misstag korrigeras fritt, här utan någon begränsning alls. Om handlingen däremot blir mindre sannolik kan instabilitet återigen motverkas i träningen genom att ignorera vissa uppdateringar.

3.4.2 Ökad effektivitet genom upprepad träning

Tack vare målfunktionen kan ytterligare förbättringar göras jämfört med vanliga policygradient-metoder. För att öka en metods effektivitet, det vill säga hur många unika observationer som krävs för att metoden ska lära sig problemet, är det önskvärt att uppdatera policyn flera gånger utifrån varje observation. Det är för de flesta policygradientmetoder inte möjligt i praktiken då det skulle leda till instabil träning. PPO kommer runt problemet genom att ha ett inbyggt skydd i form av målfunktionen. Träningen kan då ske i flera epoker i taget på varje batch av observationer, där en epok syftar på ett enskilt optimeringssteg. I artikeln föreslogs en algoritm där N agenter kör policyn $\pi_{\theta_{\text{old}}}$ parallellt och samlar in observationer i τ tidssteg, se figur 6. Parallellisering öppnar upp för snabbare träning på processorer som kan exekvera flera trådar samtidigt. Därefter beräknas $A_1^{\pi_\theta}, \ldots, A_\tau^{\pi_\theta}$. Slutligen kan parametrarna θ uppdateras och optimeras genom att träna i K epoker. Optimering sker genom att gradienten uppskattas med hjälp av målfunktionen och sedan används för att uppdatera policyn med till exempel SGD. Algoritmen finns i stora drag återgiven nedan i algoritm 3.

Algoritm 3 PPO

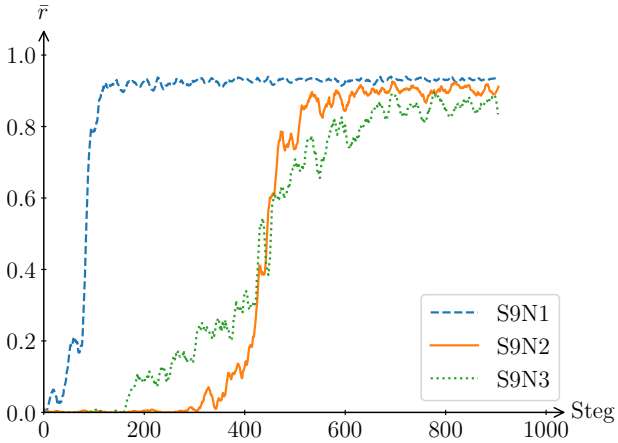
for iteration$= 1, 2, \ldots$ **do**
  for actor$= 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for τ timesteps
    Compute advantage estimates $A_1^{\pi_\theta}, \ldots, A_\tau^{\pi_\theta}$
  end for
  Optimize surrogate L^{CLIP} wrt θ, with K epochs and minibatch size $M \leq N\tau$
  $\theta_{\text{old}} \leftarrow \theta$
end for



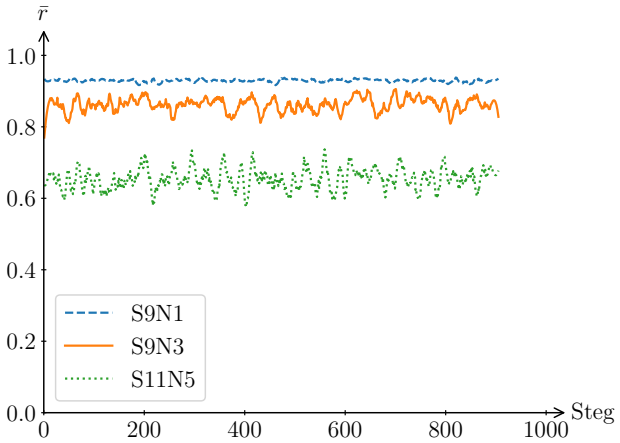
Figur 6: Visualisering av PPO med 16 agenter. Varje agent följer policyn $\pi_{\theta_{\text{old}}}$ och samlar in observationer.

3.4.3 Resultat och analys

PPO har presterat utmärkt på samtliga typer av problem som testats, både för miljöer med och utan olika typer av hinder som väggar och lava. PPO har även presterat väl i delvis observerbara miljöer, se figur 7. Dessutom har vissa modeller som tränats i en typ av miljö kunnat generalisera sin policy så att den kunde prestera bra i andra, liknande miljöer som den aldrig hade sett förut, se figur 8. Slutligen skedde träningen relativt snabbt i jämförelse med andra metoder.



Figur 7: Genomsnittlig belöning för 16 agenter mot antal optimiseringssteg. S9N{1,2,3} är en indikation på hur svår miljön är, där S9N1 är lättast och S9N3 är svårast.



Figur 8: Genomsnittlig belöning för 16 agenter mot antal steg, utan optimering. Modellen är färdigtränad och har endast tidigare tränats i miljön S9N2 men presterar ändå bra i de övriga miljöerna.

4 Diskussion

Projektet syftar till att besvara hur väl Markov decision processes och reinforcement learning kan användas till att lösa pathfinding-problem. Olika lösningar presenteras med varierande grad av komplexitet och prestanda. Till en början användes Q-table, som på grund av sin översiktlighet och enkelhet lämpade sig väl för att ge förståelse inom ämnet och lyckades producera lovande resultat i små miljöer. Sedermera studerades olika sorters neurala nätverk: ett Q-network utan dolda lager, ett djupt Q-network och ett djupt återkopplande Q-network. Policygradientmetoder med neurala nätverk såsom PPO visade goda resultat. I detta kapitel diskuteras samtliga metoder utifrån användningsområden, svagheter och styrkor.

4.1 Fullt observerbara miljöer

En stor anledning till att vi använde Q-table i första delen av projektet var för att på ett naturligt sätt introducera reinforcement learning och MDP. Vi tyckte att Q-table fyllde en viktig funktion för att själva lära oss om ämnet samtidigt som det bidrog till stringens i rapporten. Vi hade inga förhoppningar om att Q-table skulle prestera väl, men förutsåg att små pathfinding-problem ändå skulle kunna lösas och av vi tydligt skulle kunna följa hur policyn växte fram. Med tanke på att Q-table garanterar konvergens är det inte förvånande att algoritmen lyckades i små miljöer, se Sats 1. Det faktum att Q-table använder en matris som representerar tillståndsrummet och att samtliga element måste uppdateras minst en gång för att garantera konvergens gör att det finns en tydlig gräns för vad algoritmen klarar av på rimlig tid. Stora tillståndsrum kräver mycket minne för att spara matrisen och gör att policyn konvergerar långsamt. Det finns ytterligare en nackdel med Q-table som är värd att uppmärksamma; Q-table känner inte igen nya tillstånd som liknar tidigare besökta tillstånd, vilket medför att alla tillstånd måste besökas för att policyn ska konvergera.

Det första steget in på neurala nätverk var ett enkelt Q-network med två lager; ett indata-lager och ett utdata-lager. Nätverkets policy konvergerade på samma sätt som Q-table till den optimala lösningen, givet att den fick tillräckligt med tid. Styrkan med att använda Q-network för att lösa pathfinding-problem ligger inte i konvergenshastigheten utan i metodens generalitet. Q-network tar in ett tillstånd i indatalagret som sedan ger upphov till aktivering i utdatalagret. Den anpassning som behöver göras av nätverket för olika miljöer är därför mindre än för metoder som exempelvis Q-table. Nätverkets struktur bygger på att ett tillstånd propagerar genom nätverket som gör en förutsägelse av Q-värden vilka sedan används i träning till att uppdatera vikterna för att nå ett målvärde. Uppbyggnaden gör att nätet alltid har en uppskattning av Q-värdet även om nätet inte har sett tillståndet innan.

4.2 Delvis observerbara miljöer

För att praktiskt kunna hantera en delvis observerbar miljö där målet inte alltid är synligt för agenten, är det nödvändigt att använda flera dolda lager i det neurala nätverket. Anledningen till detta är komplicerad, men förenklat kan det beskrivas som att ett djupt nätverk lär sig hantera situationer med högre abstraktionsnivå, så som kanter och gångar. I och med de extra lager och noder som tillkommer i ett djupt nätverk blir också träningstiden längre och kravet på beräkningskapacitet högre.

Genom att lägga till fler funktioner och lager och därmed övergå till DQN trodde vi att pathfinding-problem skulle kunna lösas i en delvis observerbar miljö. Detta visade sig inte vara fallet. En möjlig anledning till att DQN inte fungerade som tänkt kan vara felinställda hyperparametrar. På grund av arbetets begränsade längd och begränsade resurser har ingen ordentlig prövning gjorts av hyperparametrarna och en av dessa skulle ensamt kunna vara anledningen till att algoritmen inte konvergerar. En annan möjlig anledning till att DQN inte lyckades lära sig att lösa problemen skulle kunna vara avsaknaden av minne i modellen. Försök med DRQN gjordes, men det visade sig vara svårt att verifiera korrektheten hos implementationen samtidigt som ännu fler hyperparametrar introducerades. I artikeln som introducerade DQN buntades en sekvens av observationer ihop till en enda observation genom så kallad *frame stacking* [2]. Något liknande hade kanske hjälpt DQN att uppnå bättre resultat, men det utforskades aldrig.

Det finns också en möjlighet att DQN inte lämpar sig för pathfinding-problem. Eftersom DQN väljer en handling genom att maximera handlingsvärdefunktionen är policyn deterministisk. När målet inte finns inom agentens synfält och handlingsvärdesfunktionen ger snarlika värden kommer DQN alltid att välja handlingen som ger det högsta Q-värdet. Utforskning bör i detta läge prioriteras över utnyttjande men kommer alltid att ske med sannolikhet ϵ. Agenten kan därför lätt hamna i en slinga där den upprepar en sekvens av handlingar tills episoden är slut. Ett minne skulle förmodligen hjälpa agenten att undvika sådant beteende.

PPO var den algoritmen som utan tvekan bäst klarade av pathfinding-problem med delvis observerbar miljö. Den tränade ungefär lika snabbt som Q-network gjorde för fullt observerbara miljöer och lyckades dessutom träna modellen så att den kunde uppnå bra resultat i miljöer den aldrig sett förut. När agentens beteende visualiserades i en lite större värld sågs den i början av episoden försöka utforska världen för att hitta målet. När ett mål identifierats försökte den sedan hitta vägen dit, oftast med gott resultat. Dessutom är modeller som tränats med PPO bra på att undvika lava, det vill säga hinder som avslutar episoden omedelbart om agenten går in i dem. Detta beteende har ingen annan algoritm uppvisat.

Tack vare sin målfunktion är PPO mer robust mot valet av hyperparametrar, vilket gör det lättare att hitta ett val av sådana som ger bra resultat. Handlingar väljs också på ett helt annat sätt än i DQN, vilket tillåter PPO att ta fram en stokastisk policy. Troligtvis är det det stokastiska utforskandet som gör att PPO får så bra resultat även utan någon form av minne.

4.3 Begränsningar

En av de största begränsningarna under projektet har varit bristen på datorkraft. Detta är väsentligt för att finjustera och testa hyperparametrar i ett neuralt nätverk. Det gör oss även ovetande om varför DQN och DRQN inte kan träna på något så enkelt som en tom 3×3-miljö. En mer organiserad och detaljerad hyperparameteroptimering behövs där endast enstaka parametrar ändras lite åt gången för att se hur bland annat förlustfunktionen ändrar sig. I de fåtal tester som kördes ändrades saker i algoritmen som förlustfunktionen, optimeringsmetoden och batchstorleken på samma gång vilket gjorde det svårt att urskilja vad ändringarna faktiskt gjorde med resultatet.

4.4 Förslag på vidareutveckling

Ett naturligt sätt att vidareutveckla PPO är att lägga till ett minne till algoritmen. Detta skulle lära agenten att inte gå samma väg flera gånger under en och samma episod. Man kan även göra problemet mer intressant genom att lägga till fler agenter i miljön vars uppgift blir att ta sig till enskilda eller samma mål utan att kollidera med varandra. Detta kan skala upp problemets komplexitet avsevärt då agenterna måste ha en uppfattning om vilken agent som ska prioriteras.

Agenterna måste även se till att inte blockera varandras mål eller vägar. Utöver detta kan agenternas uppgift göras mer komplicerad, till exempel att hämta upp ett objekt innan de tar sig vidare till respektive mål. Ett användningsområde för algoritmen skulle då kunna vara att få robotar att placera om lådor i en lagerlokal.

4.5 Etiska aspekter

Det finns några aspekter av AI som inte tidigare behandlats i detta arbete men som bör nämnas. Detta arbete handlar om hur ett antal algoritmer utvecklats för att få en agent att lösa olika pathfinding-problem. Då detta appliceras i verkligheten kan agenten exempelvis vara en robot placerad i en lagerlokal som flyttar pallar av varor. Det är idag inte ovanligt att robotar ersätter mänsklig personal i scenarion som detta. Den automatiseringsrevolution som idag pågår, och troligtvis kommer bli allt större, kommer ha stor betydelse för arbetsmarknaden och samhället i stort. Enligt en studie från 2014 [23] kan 46% av de jobb Sveriges befolkning idag livnär sig på automatiseras inom 20 år. Flertalet av dessa jobb kan använda sig av tekniker som bygger på algoritmer utvecklade för att lösa pathfinding-problem. Det förs en samhällsdiskussion om huruvida det är etiskt rätt att ersätta människor med robotar i industrin, med tanke på att det kan leda till ökad arbetslöshet. En studie från 2018 utförd av World Economic Forum hävdar dock att fler jobb kommer skapas än vad som försvinner på grund av ny teknik [24]. I denna studie uppskattas 75 miljoner jobb försvinna men 133 miljoner nya skapas, vilka kommer vara anpassade till samspel mellan människa, maskin och algoritm. Några yrken som kommer bli mer efterfrågade är dataanalytiker, mjukvaruutvecklare, men även yrken som kräver mänsklig kommunikation. Enligt studien kommer det bli allt viktigare med specialister inom bland annat AI och maskininlärning. Det är viktigt att denna förändring som håller på att ske på arbetsmarknaden kommuniceras så att människor i framtiden skaffar sig rätt kompetens.

5 Slutsats

MDP lämpar sig väl för att modellera pathfinding-problem i delvis observerbara miljöer med en agent. Det är ett användbart verktyg eftersom det beskriver den väsentliga informationen i pathfinding-problem som löses med RL: tillstånd, handlingar och belöningar.

De värdebaserade algoritmerna som implementerats lyckas inte lösa pathfinding-problemet i en delvis observerbar miljö. Möjliga anledningar till att DQN och dess varianter misslyckas är att hyperparametrarna inte optimerats, att algoritmen saknar ett välfungerande minne och att agenten har problem att utforska miljön längre in i träningsfasen. PPO-algoritmen lyckas lösa pathfinding-problemet i delvis observerbara miljöer med en agent och gör det snabbt. Denna algoritm lyckas lösa problemet även i miljöer den inte tränats i tack vare att den följer en stokastisk policy, vilket förenklar utforskningen. PPO är även robust mot valet av hyperparametrar tack vare dess målfunktion.

Det går att konstatera att RL fungerar väl för att lösa delvis observerbara pathfinding-problem eftersom PPO effektivt löser problemet. Dessutom har DQN potential att fungera om ovan nämnda problem åtgärdas. För att göra PPO mer effektiv kan denna algoritm vidareutvecklas genom att minne läggs till. En intressant utveckling av pathfinding-problemet vore att lägga till fler agenter.

Referenser

[1] Richard S. Sutton och Andrew G. Barto. *Reinforcement Learning: an introduction*. 2. utg. Cambridge,MA: The MIT Press, 2018, s. 28, 64–65. ISBN: 9780262039246.

[2] Volodymyr Mnih m. fl. "Human-level control through deep reinforcement learning". I: *Nature* 518 (2015). DOI: 10.1038/nature14236.

[3] Gautam Reddy m. fl. *Glider soaring via reinforcement learning in the field*. 2018. DOI: 10.1038/s41586-018-0533-0.

[4] Dr. Marco Wiering och Dr. ir. Martijn van Otterlo. *Reinforcement Learning State-of-the-Art*. Utg. av Meng-Hiot Lim och Yew-Soon Ong. Vol. 12. Springer, 2012. ISBN: 9783642015267. DOI: 10.1007/978-3-642-27645-3.

[5] Hossein Kamalzadeh och Michael Hahsler. *POMDP: Introduction to Partially Observable Markov Decision Processes*. Tekn. rapport. 2019.

[6] Vincent Francois-Lavet m. fl. "An Introduction to Deep Reinforcement Learning". I: *Foundations and Trends in Machine Learning* 11.3-4 (2018). DOI: 10.1561/2200000071.

[7] Philip S. Thomas och Emma Brunskill. "Policy Gradient Methods for Reinforcement Learning with Function Approximation and Action-Dependent Baselines". I: (juni 2017). URL: http://arxiv.org/abs/1706.06643.

[8] David Kriesel. *A Brief Introduction to Neural Networks*. Tekn. rapport. University of Bonn, 2005, s. 3–5, 37–38.

[9] Dan C Ciresan, Ueli Meier och Jürgen Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". I: (2012). URL: http://arxiv.org/abs/1202.2745.

[10] Josh Patterson och Adam Gibson. *Deep Learning A Practitioner's Approach*. Tekn. rapport. 2017. URL: http://oreilly.com/safari.

[11] Vinod Nair och Geoffrey E Hinton. *Rectified Linear Units Improve Restricted Boltzmann Machines*. Tekn. rapport.

[12] Li-Chia Yang, Szu-Yu Chou och Yi-Hsuan Yang. "MidiNet - A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation using 1D and 2D Conditions". I: (2017). URL: http://arxiv.org/abs/1703.10847.

[13] Herbert Robbins och Sutton Monro. "A Stochastic Approximation Method". I: *Ann. Math. Statist.* 22.3 (maj 1951), s. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://doi.org/10.1214/aoms/1177729586.

[14] Rajendra Akerkar och Priti Srinivas Sajja. *Intelligent Techniques for Data Science*. Cham: Springer International Publishing, okt. 2016. DOI: 10.1007/978-3-319-29206-9.

[15] Christopher J C H Watkins och Peter Dayan. *Technical Note Q-Learning*. Tekn. rapport. Boston, 1992.

[16] Maxime Chevalier-Boisvert, Lucas Willems och Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym*. 2018. URL: https://github.com/maximecb/gym-minigrid.

[17] Hado van Hasselt, Arthur Guez och David Silver. "Deep Reinforcement Learning with Double Q-learning". I: (2015). URL: http://arxiv.org/abs/1509.06461.

[18] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network RNN and Long Short-Term Memory LSTM Network". I: (2018). URL: http://arxiv.org/abs/1808.03314.

[19] Matthew Hausknecht och Peter Stone. "Deep Recurrent Q-Learning for Partially Observable MDPs". I: (juli 2015). URL: http://arxiv.org/abs/1507.06527.

[20] John Schulman m. fl. "Proximal Policy Optimization Algorithms". I: (juli 2017). URL: http://arxiv.org/abs/1707.06347.

[21] Sham Kakade och John Langford. "Approximately Optimal Approximate Reinforcement Learning". I: *19th ICML*. 2002, s. 267–274.

[22] John Schulman m. fl. "Trust Region Policy Optimization". I: (2015). URL: http://arxiv.org/abs/1502.05477.

[23] Stiftelsen för strategisk forskning. *Vartannat jobb automatiseras inom 20 år : utmaningar för Sverige*. Tekn. rapport. 2014.

[24] Till Alexander Leopold, Vesselina Stefanova Ratcheva och Saadia Zahidi. *Future of Jobs 2018 - Reports - World Economic Forum*. Tekn. rapport. 2018.

A Källkod

All källkod finns att hitta på `https://github.com/ollema/purl/`