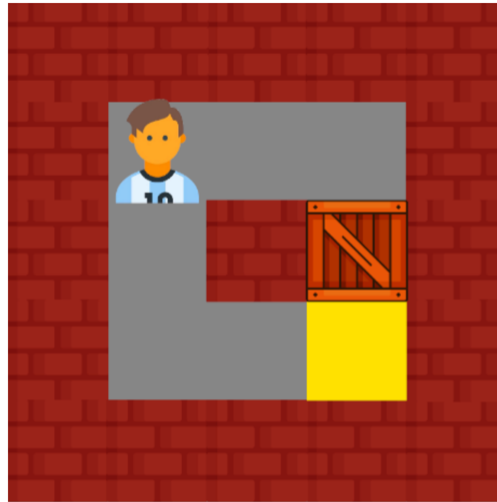


Utveckling och utvärdering av en AI-agent baserad på förstärkningsinlärning för problemlösning i spelet Sokoban

Examensarbete inom högskoleprogrammet Datateknik

Författare

Joacim Blom
Mohamad Alzein

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA

Göteborg 2026

www.chalmers.se

Examensarbete 2026

Utveckling och utvärdering av en AI-agent baserad på förstärkningsinlärning för problemlösning i spelet Sokoban

Joacim Blom
Mohamad Alzein

Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
Göteborg 2026

Utveckling och utvärdering av en AI-agent baserad på förstärkningsinlärning för problemlösning i spelet Sokoban

Joacim Blom
Mohamad Alzein

© Joacim Blom, Mohamad Alzein, 2026.

Handledare: Shirin Tavara, Data- och informationsteknik
Examinator: David Sands, Data- och informationsteknik

Examensarbete 2026
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslagsbild: Skärmbild av Sokoban-spelet implementerat i Unity.

Skriven i LaTeX
Göteborg 2026

Utveckling och utvärdering av en AI-agent baserad på förstärkningsinlärning för problemlösning i spelet Sokoban
Joacim Blom & Mohamad Alzein
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola

Sammanfattning

Detta arbete har genomförts som ett examensarbete vid Chalmers tekniska högskola och behandlar utveckling och utvärdering av en AI-agent baserad på förstärkningsinlärning för problemlösning i spelet Sokoban. Syftet med projektet var att undersöka huruvida olika RL-algoritmer (Reinforcement Learning) såsom PPO, REINFORCE, SARSA och Tabular Q-learning kan lösa problemet, samt att jämföra dessa med varandra och utvärdera hur de presterar.

Projektet har implementerats i Python och Unity. SARSA, Tabular Q-learning och REINFORCE tränades i en Python-implementation av Sokoban och kördes på Chalmers superdator Minerva, varefter de tränade modellerna utvärderades i Unity-miljön. PPO tränades och utvärderades direkt i Unity med hjälp av ML-Agents ramverket.

Resultatet från projektet visar att tabellbaserad algorithmer som SARSA och Tabular Q-learning inte är tillräckliga för att lösa Sokoban då algoritmerna memorerar lösningarna för tränade kartor och saknar förmågan att generalisera till nya, osedda kartor. REINFORCE, som bygger på ett neuralt nätverk, visade däremot en viss förmåga att lösa problemet givet tillräcklig träning och presterade bättre än SARSA och Q-learning på nya, osedda testkartor. Däremot överträffade den inte en slumpmässig agent på dessa kartor, vilket indikerar att modellen inte uppnådde tillräcklig generalisering.

Abstract

This work was done as a thesis project at Chalmers University of Technology and explores the development and evaluation of reinforcement learning agents for solving the puzzle game Sokoban. The aim of the project was to investigate whether RL algorithms such as PPO, REINFORCE, SARSA and Tabular Q-learning are capable of solving the problem, and to compare these against each other.

SARSA, Tabular Q-learning and REINFORCE were trained in a Python implementation of Sokoban using Chalmers supercomputer Minerva, after which the trained models were evaluated in a Unity environment. PPO was both trained and evaluated directly in Unity using the ML-Agents framework.

The results of this thesis show that tabular algorithms such as SARSA and tabular

Q-learning are not sufficient for solving Sokoban, as they tend to memorize solutions for the training maps and lack the ability to generalize to new, unseen maps. REINFORCE, which is based on a neural network, demonstrated some ability to solve the problem given sufficient training and performed better than SARSA and Q-learning on new, unseen test maps. However, it did not outperform a random agent on these maps, indicating that the model failed to achieve adequate generalization.

Nyckelord: Sokoban, AI-agent, Reinforcement learning, SARSA, REINFORCE, Q-learning, PPO

Förord

Detta projektet har genomförts som ett examensarbete inom högskoleprogrammet Datateknik vid Chalmers tekniska högskola under 2026.

Författarna vill rikta ett tack till handledare Shirin Tavara för värdefull vägledning och återkoppling under projektets gång.

Joacim Blom & Mohamad Alzein, Göteborg, Juni 2026

Akronymer

Nedanför är en lista över akronymer som används i denna rapport:

RL	Reinforcement Learning
SARSA	State, Action, Reward, State, Action
JSON	JavaScript Object Notation
PPO	Proximal Policy Optimization
API	Application Programming Interface

Innehåll

Beteckningar

Figurer

1 Inledning **1**
- 1.1 Bakgrund 1
- 1.2 Syfte 1
- 1.3 Mål 1
- 1.4 Avgränsningar 2

2 Teknisk Bakgrund **3**
- 2.1 Förstärkningsinlärning 3
- 2.2 Sokoban 3
- 2.3 Unity 4
- 2.4 ML-Agents 5
- 2.5 REINFORCE 5
- 2.6 SARSA 6
- 2.7 Q-learning 6
- 2.8 PPO 7
- 2.9 Python-bibliotek 7

3 Metod **9**
- 3.1 Litteraturstudie 9
- 3.2 Design och implementation av Sokoban-miljö 10
  - 3.2.1 Intern spelrepresentation 10
  - 3.2.2 Kartrepresentation 10
  - 3.2.3 Rendering och manuell testning 11
- 3.3 Integration med Unity ML-Agents 11
- 3.4 Validering av ML-Agents-miljön 11
- 3.5 Implementation av Sokoban-miljö i Python 12
- 3.6 Baseline 12
- 3.7 Implementation av RL-algoritmer 12
  - 3.7.1 REINFORCE 13
  - 3.7.2 SARSA 13
  - 3.7.3 Tabellbaserad Q-learning 14
- 3.8 Träning av RL-algoritmer 14
- 3.9 Utvärdering i Unity 16

3.10 Beräkning av framgångsgrad 17

4 Resultat **19**
4.1 Random Agent 19
4.2 REINFORCE 20
4.3 SARSA 21
4.4 Tabellbaserad Q-learning 22
4.5 Mastery Rate 23
4.6 Win Rate 24

5 Diskussion **25**
5.1 Analys av resultat 25
5.2 Val av metoder och tekniker 25
5.3 Val av kartor 26
5.4 Justering av hyperparametrar 27
5.5 Belöningssystem 27
5.6 Träningssystem 28
5.7 Sokoban som AI-problem 28
5.8 Förbättringsområden 29
5.9 Användning av AI-verktyg 30

6 Slutsats **31**

Litteraturförteckning **33**

Figurer

2.1 Exempel på en Sokoban-karta. 4
2.2 Unitys utvecklingsmiljö. 5

3.1 Flödesschema över träningsloopen. 16

4.1 Accuracy under träning för Random Agent. 19
4.2 Accuracy under testning för Random Agent. 19
4.3 Accuracy under träning för REINFORCE. 20
4.4 Accuracy under testning för REINFORCE. 20
4.5 Accuracy under träning för SARSA. 21
4.6 Accuracy under testning för SARSA. 21
4.7 Accuracy under träning för Q-learning. 22
4.8 Accuracy under testning för Q-learning. 22
4.9 Mastery rate per algoritm. 23
4.10 Win rate under testning per algoritm. 24

1
Inledning

Detta projekt åsyftar att utveckla och undersöka AI-agenter baserade på förstärkningsinlärning i spelet Sokoban.

1.1 Bakgrund

Artificiell Intelligens (AI) har under de senaste decennierna utvecklats till att bli en central del av moderna tekniska system. Spel och pusselproblem har länge använts som testmiljöer för AI-forskning, eftersom de erbjuder tydliga regler, väldefinierade tillstånd och mätbara mål. Ett välkänt exempel är AlphaGo, där förstärkningsinlärning användes för att träna en agent att spela brädspelet Go på en nivå som överstiger mänskliga experter [1]. I detta arbete undersöks förstärkningsinlärning i spelet Sokoban, där Unity tillsammans med ramverket ML-Agents [2] används för att möjliggöra kommunikation mellan spelmiljön och Python-baserade algoritmer.

1.2 Syfte

Syftet med examensarbetet är att undersöka om och hur RL kan användas för att lösa Sokoban i en Unity-baserad miljö, samt att analysera agentens inlärningsbeteende och prestanda. Projektet syftar även till att implementera och utvärdera AI-metoder såsom REINFORCE, SARSA och tabulär Q-learning i Python, samt att experimentera med den inbyggda PPO-metoden i Unity ML-Agents.

1.3 Mål

Målet med detta examensarbete är att utveckla en AI-agent som kan tränas för att lösa enkla Sokoban-kartor. Agenten ska därefter utvärderas på nya, tidigare osedda kartor för att undersöka huruvida den använda förstärkningsinlärningsalgoritmen uppnår tillräcklig generaliseringsförmåga.

För att möjliggöra detta utvecklas en fungerande Sokoban-miljö i både Python och Unity. Unity integreras med Python via ML-Agents för att möjliggöra träning och testning av olika förstärkningsinlärningsalgoritmer.

Agentens prestanda utvärderas utifrån flera kriterier, inklusive erhållna belöningar, antal steg som krävs för att lösa en karta, huruvida kartan löses, samt den tid som krävs för träning och testning av metoden och modellens generaliseringsförmåga.

1.4 Avgränsningar

Arbetet begränsas till fyra RL-algoritmer: PPO, REINFORCE, SARSA och Tabular Q-learning. Dessa valdes då de bedömdes vara representativa för både värdebaserade och policybaserade metoder inom förstärkningsinlärning.

Endast Sokoban-kartor av storleken 5×5 studeras (inklusive väggar). Större kartor testades initialt men visade sig kräva alltför lång träningstid. När implementationen flyttades till Python behölls samma kartstorlek, och tidsramen medgav inte att utöka till större kartor.

Deadlock-hantering beaktas inte i detta arbete då det bedömdes ligga utanför arbetets tidsramar.

Endast ett begränsat antal kartor studeras, totalt 100 träningskartor och 10 testkartor.

2

Teknisk Bakgrund

I detta kapitel presenteras de tekniker som arbetet bygger på. Kapitlet ger en överblick av dessa tekniker och erbjuder insikt i hur de fungerar.

2.1 Förstärkningsinlärning

Förstärkningsinlärning är en maskininlärningsmetod där en agent lär sig att fatta beslut genom interaktion med en miljö [3, p. 1-4]. Agenten observerar ett tillstånd, utför en handling och får därefter en belöning beroende på resultatet av handlingen. Målet är att maximera den ackumulerade belöningen över tid, vilket kan uttryckas som:

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

där G_t är den totala framtida belöningen från tidpunkt t, R_{t+k+1} är den belöning agenten får vid ett framtida tidsteg, och γ är en diskonteringsfaktor som avgör hur stor betydelse framtida belöningar får i förhållande till omedelbara belöningar [3, p. 54-55]. På så sätt kan förstärkningsinlärning användas för problem där en agent måste fatta en sekvens av beslut, där varje handling kan påverka både den omedelbara belöningen och framtida tillstånd och beslut.

2.2 Sokoban

Sokoban är ett japanskt pusselspel där spelaren styr en arbetare i ett rutnät och ska flytta lådor till förutbestämda målpositioner. Spelet kännetecknas av ett stort tillståndsrum och förekomsten av återvändsgränder (deadlocks), där vissa drag leder till situationer som inte kan lösas. Detta gör Sokoban till en välanpassad testmiljö för RL-baserade AI-modeller [4, p. 221]. Ett exempel på en Sokoban-karta visas i figur 2.1.

Figur 2.1: Exempel på en Sokoban-karta.

Figuren visar en spelare som står till höger om en trälåda. Uppgiften är att flytta lådan till målpositionen, som markeras av en gul ruta. De röda tegelrutorna utgör väggar som avgränsar spelplanen.

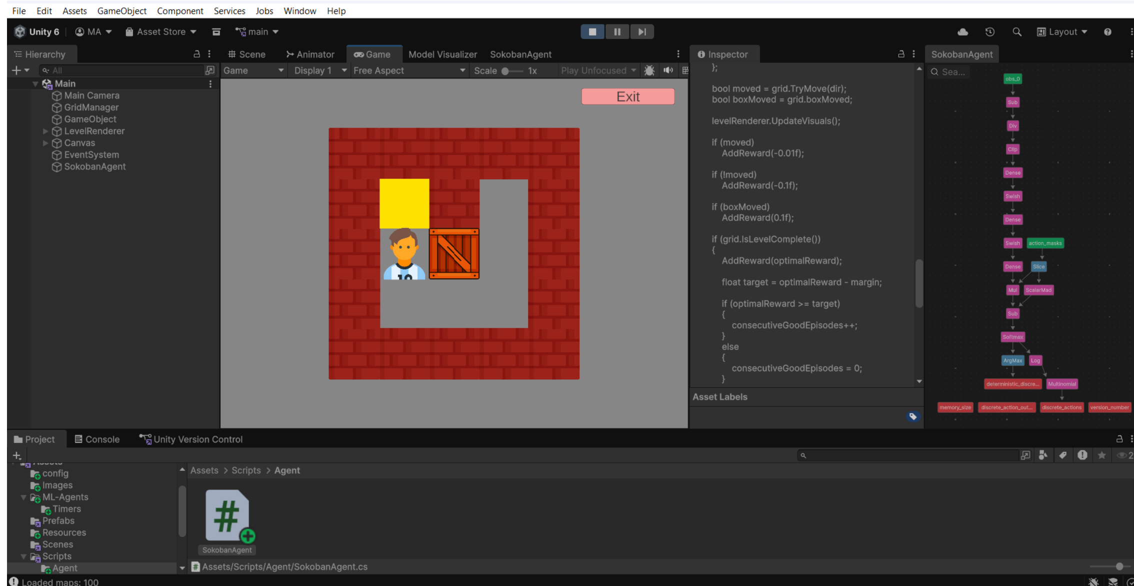
2.3 Unity

Unity är en spelmotor utvecklad av Unity Technologies och används för att utveckla spel och interaktiva applikationer för datorer, konsoler, mobila enheter och webbläsare [5]. Plattformen är utformad för att vara flexibel och skalbar, vilket gör den lämplig både för små projekt och större kommersiella spelutvecklingsprojekt. En av styrkorna med Unity är dess stora ekosystem av verktyg och dokumentation, vilket gör det relativt enkelt att snabbt prototypa och utveckla komplexa miljöer [6].

Unity bygger på en komponentbaserad arkitektur där varje objekt i en scen består av olika komponenter som definierar dess funktionalitet och beteende. Denna design gör det möjligt att återanvända och kombinera funktioner på ett modulärt sätt. Programmering sker huvudsakligen i C#, vilket ger stöd för objektorienterad utveckling och gör det möjligt att implementera avancerad logik, fysik och interaktion i spelmiljön.

I detta arbete har Unity använts för att implementera den grafiska och interaktiva miljön för Sokoban. Genom att separera spel-logik och rendering har miljön kunnat anpassas för både manuell testning och automatisk inlärning. I kombination med ML-Agents har Unity möjliggjort integration med Python-baserade förstärkningsinlärningsalgoritmer, vilket gjort det möjligt att träna och utvärdera olika AI-agenter

i en kontrollerad simuleringsmiljö. Unitys utvecklingsmiljö illustreras i figur 2.2.



Figur 2.2: Unitys utvecklingsmiljö.

Figuren visar Unitys editor. I mitten visas spelvyn med en Sokoban-spelplan. Till vänster syns hierarkipanelen med projektets objekt och till höger inspektörspanelen med inställningar för det markerade objektet. Längst ned visas projektets assets som vi har skrivit i C# och konsolfönster som används för debuging.

2.4 ML-Agents

Unity ML-Agents Toolkit är ett open-source projekt som tillåter spel och testmiljöer att användas för att träna AI-agenter. ML-Agents inkluderar PyTorch implementationer av avancerade moderna RL-metoder som PPO, SAC och MA-POCA.

ML-Agents fungerar även som ett API som möjliggör kommunikation mellan Unity-miljön och extern Pythonkod. Detta gör det möjligt att styra simuleringen från Python, samla in observationer och skicka tillbaka handlingar från en tränad modell. På så sätt kan förstärkningsinlärning implementeras där Unity fungerar som miljö och Python som träningsramverk [7].

2.5 REINFORCE

REINFORCE-algoritmen är en av de första policybaserade förstärkningsinlärningsmetoderna och introducerades av Ronald J. Williams 1992 [8]. Den bygger på Monte Carlo policy gradient, där en agent lär sig en stokastisk policy $\pi(a \mid s)$, som anger sannolikheten att välja handlingen a i ett givet tillstånd s [3, p. 326-329].

Policyn uppdateras baserat på den kumulativa diskonterade belöningen G_t, som beräknas bakåt i tid från tidpunkt t:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \tag{2.1}$$

där γ är diskonteringsfaktorn som avgör hur mycket framtida belöningar värderas i förhållande till omedelbara belöningar. Med hjälp av G_t kan policyn uppdateras enligt följande uppdateringsregel:

$$\theta \leftarrow \theta + \alpha G_t \nabla \ln \pi(A_t \mid S_t, \theta) \tag{2.2}$$

där α är inlärningshastigheten. Uppdateringsregeln ökar sannolikheten för handlingar som leder till hög kumulativ belöning.

2.6 SARSA

State-Action-Reward-State-Action (SARSA) är en av de tidigaste värdebaserade metoderna inom förstärkningsinlärning och introducerades av G. A. Rummery och M. Niranjan 1994 [9]. Metoden bygger på att agenten lär sig att approximera Q-värden $Q(s, a)$, vilka representerar den förväntade framtida belöningen för att utföra handlingen a i tillståndet s [3, p. 129-131].

SARSA uppdaterar sina värden baserat på den handling som faktiskt utförs i nästa tillstånd, vilket gör den till en on-policy-metod. Uppdateringen sker enligt följande uppdateringsregel:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \Big] \tag{2.3}$$

där α är inlärningshastigheten och γ är diskonteringsfaktorn. Uppdateringen baseras på den handling som faktiskt väljs i nästa tillstånd, vilket skiljer SARSA från off-policy-metoder såsom Q-learning.

2.7 Q-learning

Q-learning är en av de tidigaste och mest inflytelserika värdebaserade metoderna inom förstärkningsinlärning och introducerades av Christopher J. C. H. Watkins 1989 [10]. Precis som SARSA bygger metoden på att approximera Q-värden $Q(s, a)$. Till skillnad från SARSA är Q-learning en off-policy-metod, vilket innebär att uppdateringen baseras på den maximala möjliga framtida belöningen, oberoende av vilken handling som faktiskt väljs i nästa tillstånd [3, p. 131-132].

Q-learning uppdaterar sina värden baserat på den maximala uppskattade framtida belöningen från nästa tillstånd, oberoende av vilken handling som faktiskt väljs. Uppdateringen sker enligt följande uppdateringsregel:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big] \tag{2.4}$$

där α är inlärningshastigheten, γ är diskonteringsfaktorn och r_{t+1} är belöningen vid nästa tidssteg. Uppdateringen baseras på det maximala Q-värdet i nästa tillstånd, vilket gör metoden off-policy.

2.8 PPO

Proximal Policy Optimization (PPO) är en policybaserad metod inom förstärkningsinlärning som introducerades av Schulman et al. 2017 [11]. Metoden bygger på policy gradient och används för att uppdatera en stokastisk policy $\pi_\theta(a \mid s)$, där θ representerar modellens parametrar.

Till skillnad från enklare policy gradient-metoder använder PPO en modifierad målfunktion för att kontrollera hur mycket policyn förändras mellan uppdateringar. Detta görs genom att jämföra den nya policyn med en tidigare version av policyn. En central komponent är sannolikhetskvoten:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{2.5}$$

där θ_{old} representerar parametrarna innan uppdateringen.

PPO använder en klippt målfunktion för att begränsa stora förändringar:

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[\min\Big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big)\right] \tag{2.6}$$

där $\hat{A}_t$ är en uppskattning av fördelningsfunktionen (advantage) och ϵ är en hyperparameter. Klippningen innebär att värden på $r_t(\theta)$ begränsas till ett intervall kring 1, vilket påverkar hur uppdateringen beräknas.

Precis som andra policy gradient-metoder baseras uppdateringen på sampling från miljön. PPO samlar först in data genom interaktion med miljön och använder därefter denna data för att uppdatera policyn under flera steg med gradientbaserad optimering.

Metoden används ofta tillsammans med en värdefunktion $V(s)$ för att uppskatta framtida belöningar, men kärnan i PPO ligger i den klippta målfunktionen och hur policyn uppdateras mellan iterationer.

2.9 Python-bibliotek

PyTorch är ett Python-baserat ramverk för maskininlärning som möjliggör byggande och träning av neurala nätverk. I detta projekt används PyTorch för implementationen av REINFORCE (avsnitt 3.7.1), där det neurala nätverket `PolicyNet` definieras och tränas med hjälp av ramverket [12].

NumPy är ett Python-bibliotek för numeriska beräkningar med stöd för flerdimensionella arrayer. I projektet används NumPy bland annat för tillståndsrepresentation, slumpval av handlingar och beräkning av Q-tabellsvärden i SARSA och Tabular Q-learning [13].

Pickle är ett Python-bibliotek för serialisering och deserialisering av objekt till och från binärt format. I detta projekt används Pickle för att spara de tränade Q-tabellerna för SARSA och Q-learning till `.pkl`-filer, samt för att ladda dem inför utvärdering. Pickle valdes eftersom det är ett inbyggt Python-bibliotek som kan spara komplexa Python-objekt direkt utan extra konvertering. Detta är en fördel jämfört med exempelvis JSON, som är mer begränsat till enklare datatyper och ofta kräver manuell anpassning av datastrukturen. Eftersom Q-tabellerna används inom samma Python-baserade projektmiljö ger Pickle ett enkelt och praktiskt sätt att bevara tabellernas struktur mellan träning och utvärdering [14].

3 Metod

Arbetet genomfördes enligt en vattenfallsmetodik där projektet delades in i faserna litteraturstudie, design, implementation, testning och utvärdering. Detta möjliggjorde en systematisk progression från teoretisk grund till praktisk implementation och experimentell analys.

3.1 Litteraturstudie

Projektet inleddes med en fördjupad litteraturstudie inom förstärkningsinlärning. Den teoretiska grunden baserades främst på *Reinforcement Learning: An Introduction* av Sutton och Barto samt vetenskapliga artiklar om policybaserade och värdebaserade metoder.

Särskilt fokus lades på följande algoritmer:

- REINFORCE (policybaserad metod),
- Proximal Policy Optimization (PPO),
- SARSA,
- Tabellbaserad Q-learning.

Valet av algoritmer i detta arbete baseras på studier där liknande algoritmer har tillämpats i andra spel- och kontrollmiljöer.

Raut et al. [2] utforskade SARSA, Q-learning och PPO i ett racingspel utvecklat i Unity med ML-Agents, där PPO visade sig prestera effektivast. Eftersom PPO är inbyggt i ML-Agents och Unity användes i detta projekt valdes PPO.

Patel et al. [15] implementerade och jämförde DQN, REINFORCE och A2C på det klassiska Cart-pole-problemet. Studien visade att REINFORCE, trots sin höga varians och långsammare konvergens, är en effektiv policybaserad metod som är relativt enkel att implementera jämfört med DQN och A2C. Av dessa skäl valdes REINFORCE.

För de värdebaserade metoderna SARSA och Q-learning, valdes dessa av tre studier. Shoham och Elidan [16] visade att enkla värdebaserade metoder kan lösa Sokobankartor när belöningarna är tillräckligt frekventa, men att glesa belöningar utgör en stor utmaning. Mehta [20] demonstrerade vidare att Q-learning presterar väl i

pusselspel med väldefinierade tillstånd och frekventa belöningar. Zhong [21] visade att tabellbaserad SARSA och Q-learning konvergerar effektivt i det rutnätsbaserade cliff walking-problemet, en miljö med liknande struktur som Sokoban med avseende på diskreta tillstånd och handlingar. Av dessa anledningar valdes de tabellbaserade varianterna av Q-learning och SARSA då de är enkla att implementera och lämpar sig väl för diskreta tillståndsrum.

Dessa litteraturstudier gav en djupare förståelse för algoritmernas matematiska grund, konvergensbeteende, styrkor och begränsningar innan implementationen påbörjades. Detta låg till grund för designval gällande tillståndsrepresentation, belöningsstruktur och träningsupplägg.

3.2 Design och implementation av Sokoban-miljö

Som första praktiska steg utvecklades en fungerande Sokoban-miljö i Unity. Miljön designades med tydlig separation mellan spel-logik, rendering och agentstyrning för att möjliggöra flexibel testning och integration med RL-algoritmer.

3.2.1 Intern spelrepresentation

Den logiska kärnan implementerades i klassen `GridManager`. Spelplanen representeras med en tvådimensionell array.

Spelarens position lagras separat som en `Vector2Int`. Banor definieras som strängarrayer och parsas vid laddning genom metoden `ParseLevel()`, vilket initialiserar rutnätets dimensioner samt placerar ut väggar, mål, lådor och spelare.

Rörelselogiken implementerades i metoden `TryMove()`, vilket:

- kontrollerar kollisioner mot väggar,
- hanterar förflyttning av lådor enligt Sokobans regler,
- förhindrar att lådor trycks in i väggar eller andra lådor,
- uppdaterar spelarens position och räknar antal drag.

Vinstvillkoret implementerades i `IsLevelComplete()`, som kontrollerar att samtliga målpositioner innehåller en låda.

3.2.2 Kartrepresentation

Alla kartorna skapades manuellt och lagrades i textfiler. I textfilerna representeras spelplanen med följande tecken:

- `#` - vägg
- `@` - spelare
- `.` - mål
- `$` - låda

- - tomt utrymme

Totalt skapades 100 träningskartor och 10 testkartor, alla av storleken 5×5. Kartorna varierar i svårighetsgrad och verifierades manuellt för att säkerställa att de är lösbara. Testkartorna förekom aldrig under träningen för att möjliggöra utvärdering av algoritmernas generaliseringsförmåga.

3.2.3 Rendering och manuell testning

Rendering separerades från spel-logiken och implementerades i klassen `LevelRenderer`. Denna klass instansierar prefabs för väggar, mål, lådor och spelare baserat på den interna grid-representationen. Efter varje drag synkroniseras den visuella representationen med den logiska spelstaten.

För manuell testning implementerades `GridInputController`, som översätter tangenttryckningar till riktningsvektorer och anropar rörelselogiken i `GridManager`. Detta möjliggjorde verifiering av korrekt spelbeteende innan RL-integration.

3.3 Integration med Unity ML-Agents

För att möjliggöra förstärkningsinlärning integrerades miljön med Unity ML-Agents Toolkit. En agentklass, `SokobanAgent`, implementerades genom att ärva från `Agent`.

Vid episodstart återställs miljön och aktuell bana laddas. Tillståndsrepresentationen definierades i `CollectObservations()` och består av:

- spelarens position,
- en fullständig binär kodning av hela rutnätet med avseende på väggar, mål och lådor.

Handlingar definierades som diskreta med fyra möjliga riktningar (upp, ned, vänster, höger). Den valda handlingen översätts till en riktningsvektor och skickas till miljön via `TryMove()`.

Belöningsfunktionen utformades enligt följande:

- (-0.01) Liten negativ belöning per giltig drag (stegkostnad),
- (-0.1) Större negativ belöning för ogiltiga drag(agenten flyttas inte),
- (+0.1) Positiv belöning när en låda flyttas,
- (+10) Stor positiv belöning när banan löses,
- Episodavslut vid lösning eller efter ett maximalt antal steg (200 steg).

3.4 Validering av ML-Agents-miljön

Efter att spelmiljön implementerats och tillståndsrepresentation samt belöningsfunktion definierats, genomfördes en initial verifiering av integrationen med Unity

ML-Agents. Syftet var att säkerställa att kommunikationen mellan miljön och träningsramverket fungerade korrekt innan egna algoritmer implementerades.

För detta ändamål användes den inbyggda PPO-algoritmen i ML-Agents. En förenklad testkarta med en låda och ett mål skapades. Standardkonfigurationen för PPO användes.

Träningen initierades via kommandot:

```
mlagents-learn Assets/config/sokoban_ppo.yaml --run-id=ppo
```

Under träningen observerades att agenten initialt utförde slumpmässiga handlingar, men successivt lärde sig att lösa den enkla kartan. Efter ungefär 250 000 steg och ungefär 45 minuters träning uppnåddes en lösningsfrekvens på cirka 90%.

Denna verifiering bekräftade att systemet fungerade korrekt och utgjorde en stabil grund för vidare arbete med egna implementerade algoritmer.

3.5 Implementation av Sokoban-miljö i Python

Utöver Unity-miljön implementerades Sokoban även i Python för att möjliggöra snabbare träning av SARSA, Tabular Q-learning och RERINFORCE. Python implementationen består av klasserna `Sokoban` och `SokobanEnv`, där `Sokoban` hanterar spellogik och `SokobanEnv` fungerar som ett träningsgränssnitt för RL-algoritmerna.

Python imlementationen av Sokoban är identisk med Unity-miljön, där varje cell kodas med fyra binära features med avseende på vägg, mål, låda och spelare, vilket resulterar i en obeservationsvektor med storlek $4 \times 5 \times 5 = 100$. Belöningsfunktionen är även identisk med Unity-implementationen.

Denna Python-miljön möjliggjorde träning av algoritmerna på Chalmers superdator Minerva [22]. Eftersom Python-miljön är fristående och inte kräver Unity kunde träningen köras effektivt i klustermiljön, vilket gav betydligt kortare träningstider jämfört med Unity-baserad träning.

3.6 Baseline

Innan RL-algoritmerna tränades implementerades en slumpmässig agent som baseline. Agenten är implementerad i funktionen `random_policy` och väljer vid varje tidssteg en slumpmässig handling från de fyra möjliga riktningarna. Agenten har ingen inlärning och tar inte hänsyn till det aktuella tillståndet. Syftet är att etablera en referensnivå för framgångsgrad, det vill säga vad som kan uppnås helt utan strategi.

3.7 Implementation av RL-algoritmer

Efter att baselinen verifierats implementerades och testades följande algoritmer.

3.7.1 REINFORCE

REINFORCE implementerades från grunden i Python med PyTorch. En något modernare variant av algoritmen används då den approximerar handlingssannolikheter via ett neuralt nätverk, `PolicyNet`, istället för en tabell. Nätverket består av tre fully connected lager med storlekarna 128, 64 och 4, där 4 motsvarar antalet möjliga handlingar. ReLU används som aktiveringsfunktion i de dolda lagren, och utgångslagret producerar logits som omvandlas till en sannolikhetsfördelning via en Categorical-fördelning. Handlingar samplas från denna fördelning, vilket innebär att algoritmen följer en stokastisk policy. En inlärningshastighet på 10^{-3} används tillsammans med AdamW-optimeraren.

Klassen `REINFORCE` består av följande instansvariabler:

- `policy_net` — det neurala nätverket som approximerar policyn,
- `optimizer` — AdamW-optimeraren som uppdaterar nätverkets vikter,
- `log_probs` — loggade sannolikheter för valda handlingar,
- `entropies` — entropi för varje tidssteg,
- `rewards` — belöningar samlade under ett avsnitt.

Klassen består även av två metoder, `calculate_returns` och `learn`. `calculate_returns` beräknar den kumulativa diskonterade belöningen G_t med diskonteringsfaktorn $\gamma = 0.99$. Returerna normaliseras inte. `learn` beräknar policyförlusten som används för att uppdatera nätverkets vikter. Algoritmen inkluderar även en entropiförlust med vikten 0.001 som uppmuntrar utforskning genom att motverka att policyn konvergerar för tidigt mot ett fåtal handlingar [15, p. 5].

Under träning tar algoritmen in en observation, predicerar en handling via `PolicyNet` genom sampling från Categorical-fördelningen, tar emot en belöning från miljön, och uppdaterar nätverkets vikter i slutet av varje avsnitt.

3.7.2 SARSA

SARSA implementerades från grunden i Python med NumPy. Den ursprungliga SARSA-algoritmen används, där Q-värden approximeras med hjälp av SARSA-uppdateringsregeln och lagras i en Q-tabell. Q-tabellen är implementerad som en Python-dictionary med nyckel-värdepar på formen `{(tillstånd, handling): q-värde}`. En inlärningshastighet på $\alpha = 0.1$, en diskonteringsfaktor på $\gamma = 0.99$ och en konstant utforskningsgrad på $\epsilon = 0.1$ används under träningen.

Klassen `SARSA` består av följande instansvariabler:

- `n_actions` — antalet möjliga handlingar,
- `alpha` — inlärningshastigheten,
- `gamma` — diskonteringsfaktorn,
- `epsilon` — utforskningsgraden i epsilon-girig strategi,

- `Q_table` — Q-tabellen som lagrar inlärda Q-värden,
- `episode_rewards` — belöningar samlade under ett avsnitt.

Klassen består av tre metoder, `get_Q`, `epsilon_greedy` och `update`. `get_Q` hämtar Q-värdet för ett givet tillstånd och handling, och returnerar 0.0 om paret inte finns i tabellen. `epsilon_greedy` väljer en slumpmässig handling med sannolikhet ϵ, annars väljs den handling med högst Q-värde. `update` uppdaterar Q-värdet för det aktuella tillstånds- och handlingsparet enligt SARSA-uppdateringsregeln, där den faktiskt valda nästa handlingen a' används i beräkningen.

Under träning väljer algoritmen en handling a via `epsilon_greedy`, tar emot en belöning r och observerar nästa tillstånd s'. Därefter väljs nästa handling a' via `epsilon_greedy`, varefter `update` anropas med (s, a, r, s', a') för att uppdatera Q-värdet.

3.7.3 Tabellbaserad Q-learning

Tabellbaserad Q-learning implementerades från grunden i Python med NumPy. Den ursprungliga Q-learning-algoritmen används, vilket innebär att den delar samma struktur som SARSA med avseende på Q-tabell, instansvariabler och metoder. Den enda skillnaden ligger i `update`-metoden, där Q-learning använder det maximala Q-värdet för nästa tillstånd istället för Q-värdet för den faktiskt valda nästa handlingen.

3.8 Träning av RL-algoritmer

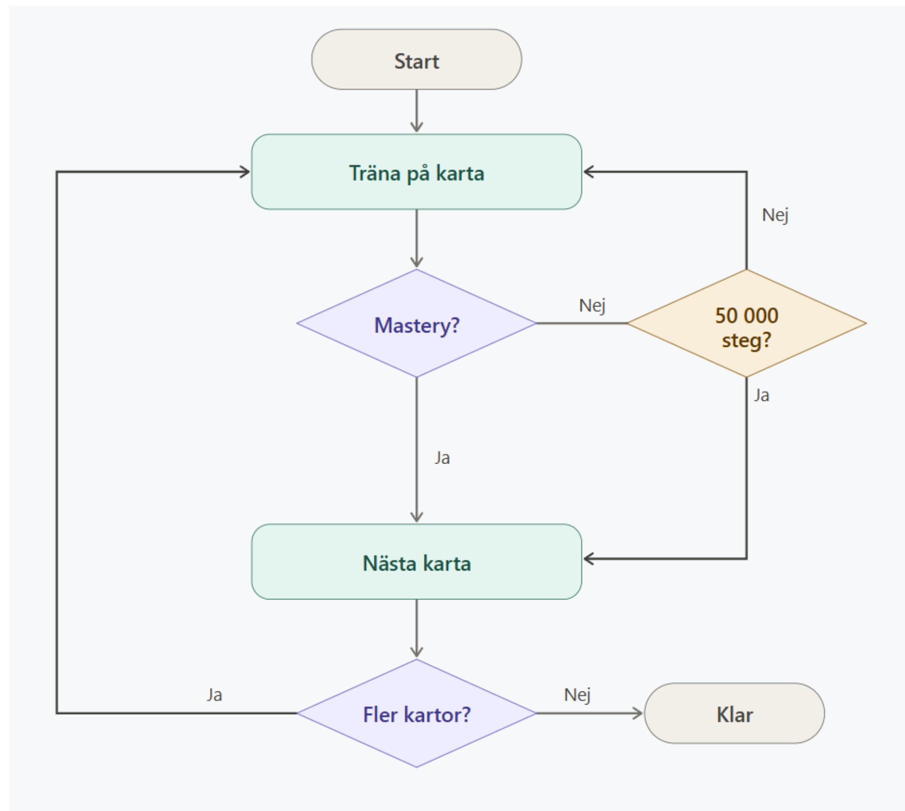
Träningen av RL-algoritmerna sker i Python-programmet `Train_Agent`. Programmet består av en `main`-metod med parametern `model`, en sträng som bestämmer vilken algoritm som ska tränas. Metoden innehåller fyra if-satser som hanterar `Random Agent`, `REINFORCE`, `SARSA` och `Q-learning`.

När en algoritm har valts initieras Python-miljön `SokobanEnv` samt agenten med dess hyperparametrar, såsom inlärningshastighet, diskonteringsfaktor och utforskningsgrad.

Under träningen loopar programmet över kartor och episoder. Vid varje tidssteg hämtas en observation från miljön som en NumPy-array. För REINFORCE konverteras observationen till en PyTorch-tensor och skickas genom det neurala nätverket `PolicyNet` för att sampla en handling. För SARSA och Q-learning används istället `epsilon_greedy` för att välja en handling baserat på Q-tabellen. Handlingen skickas sedan till miljön via `env.step()`, som returnerar nästa observation, belöning och en flagga som indikerar om episoden är avslutad.

Om agenten tar 200 steg utan att lösa kartan avslutas episoden och börjar om på samma karta. Om agenten uppnår en belöning nära den optimala belöningen för kartan ökar variabeln `streak` med ett. När `streak` når 10 anses kartan bemästrad och träningen fortsätter på nästa karta. Om kartan inte bemästras inom 50 000 steg sker bytet automatiskt. Om det inte finns några fler kartor så är träningen över.

Det finns totalt 100 träningskartor av storleken 5×5. Träningsloopen illustreras i figur 3.1.



Figur 3.1: Flödesschema över träningsloopen.

Efter varje avslutad episod loggas information om episodnummer, total belöning, antal steg, aktuell karta och streak i en textfil. REINFORCE loggar även förlustvärdet `loss` från `learn`-metoden.

Efter avslutad träning sparas modellerna. REINFORCE sparas som en `.pt`-fil via PyTorch, medan SARSA och Q-learning sparar sina Q-tabeller som `.pkl`-filer via Python-biblioteket `pickle`.

3.9 Utvärdering i Unity

För utvärdering i Unity används ML-Agents som API för kommunikation mellan Python och Unity. Detta sker i Python-programmet `Eval_Agent`, där ML-Agents importeras via `UnityEnvironment`. Unity-miljön initieras genom att starta en förbyggd Unity-applikation, vilket görs med hjälp av Unitys inbyggda byggfunktionalitet i förväg innan koden körs.

`Eval_Agent` har en strängvariabel `model` som bestämmer vilken modell som ska utvärderas, på samma sätt som i `Train_Agent`. När modellen har valts laddas den förtränade hjärnan in, REINFORCE laddas från en `.pt`-fil via PyTorch, medan SARSA och Q-learning laddas från `.pkl`-filer via `pickle`.

Under utvärderingen skickar Unity observationer till Python via `env.get_steps()`, som returnerar en NumPy-array representerande det aktuella tillståndet i miljön. Python-agenten väljer en handling baserat på den inladdade modellen och skickar

tillbaka handlingen till Unity via `env.set_actions()`. Unity utför handlingen i spelmiljön och returnerar nästa observation samt belöning. Varje modell spenderar 1000 steg per karta, varefter bytet sker automatiskt till nästa karta. Totalt finns 10 testkartor som inte förekom under träningen.

Resultaten loggas efter varje avslutad episod i textfiler med information om episodnummer, total belöning och antal steg, på samma sätt som under träningen.

3.10 Beräkning av framgångsgrad

Efter att alla RL-metoder hade tränats och utvärderats beräknades framgångsgraden för varje metod. Detta gjordes i Python-programmet `Calc_success_rate`, som består av de sex funktionerna `get_reward`, `get_mastery`, `plot_train`, `plot_test`, `plot_mastery_rate` och `plot_win_rate`.

`get_reward` hämtar alla belöningar från de loggade textfilerna för varje episod och returnerar dessa som en lista. Funktionen används som hjälpfunktion av övriga funktioner.

`get_mastery` hämtar alla streaks från textfilerna och beräknar hur många gånger en streak på 10 förekommer, vilket motsvarar hur många gånger en karta har bemästrats.

`plot_train` beräknar och plottar prestandan under träningen. Accuracy beräknas genom att dividera belöningen för varje episod med den optimala belöningen för den aktuella kartan:

$$\text{accuracy} = \max\left(0, \min\left(1, \frac{r}{\bar{r}_{\text{optimal}}}\right)\right) \tag{3.1}$$

De optimala belöningarna är förberäknade och lagrade i en Python-dictionary, samma mappning som användes under träningen för beräkning och loggning av mastery.

`plot_test` gör samma sak som `plot_train` men använder istället resultaten från de osedda testkartorna, med en separat dictionary för de optimala belöningarna för testkartorna.

Eftersom accuracy ibland kan vara missvisande introducerades även två egenutvecklade mått, `plot_mastery_rate` och `plot_win_rate`, för att ge en mer rättvis bild av algoritmernas prestation.

`plot_mastery_rate` beräknar hur många av de 100 träningskartorna varje algoritm har bemästrat. En karta anses bemästrad när agenten löser den 10 gånger i rad med en belöning inom marginalen 0.5 från den optimala belöningen, det vill säga:

$$\text{threshold} \geq r_{\text{optimal}} - 0.5$$

Mastery beräknas under träningen och kan därför inte användas för testresultaten.

För testresultaten introducerades därför `plot_win_rate`, som beräknar andelen episoder där agenten uppnår en belöning över ett tröskelvärde. Tröskeln beräknas per testkarta som:

$$\text{threshold} = r_{\text{optimal}} - 0.5$$

där r_{optimal} är den förberäknade optimala belöningen för den aktuella testkartan.

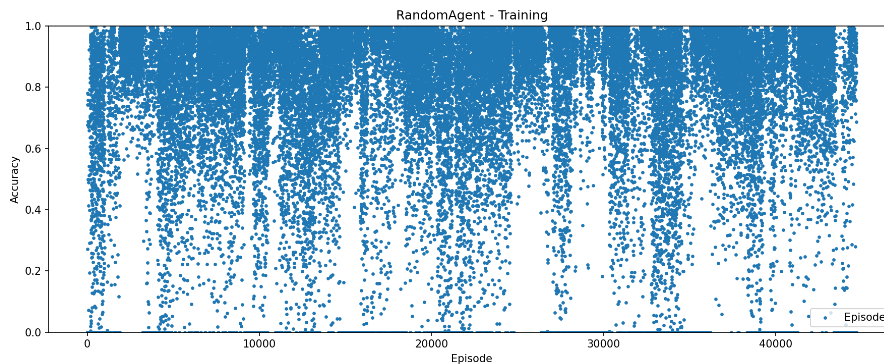
4

Resultat

Alla mått som används i detta kapitel beskrivs i avsnitt 3.10, Beräkning av framgångsgrad.

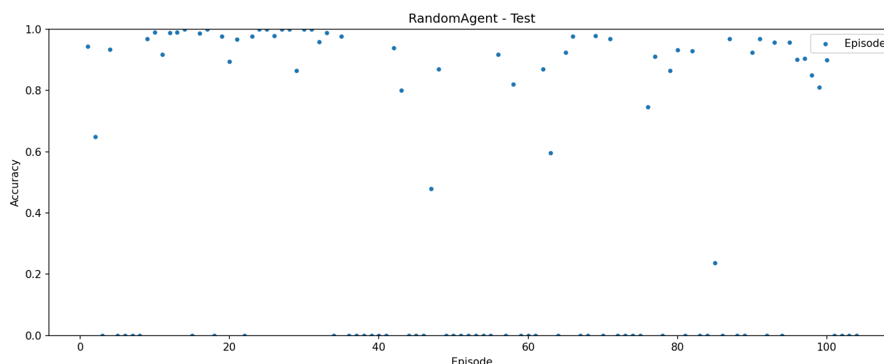
4.1 Random Agent

Accuracy under träning och testning för Random Agent visas i figur 4.1 och figur 4.2.



Figur 4.1: Accuracy under träning för Random Agent.

Det stora antalet punkter beror på att agenten sällan bemästrar en karta och därmed spenderar alla 50 000 steg per karta, vilket resulterar i betydligt fler episoder totalt. Agenten uppvisar hög varians utan någon tydlig inlärning över tid.

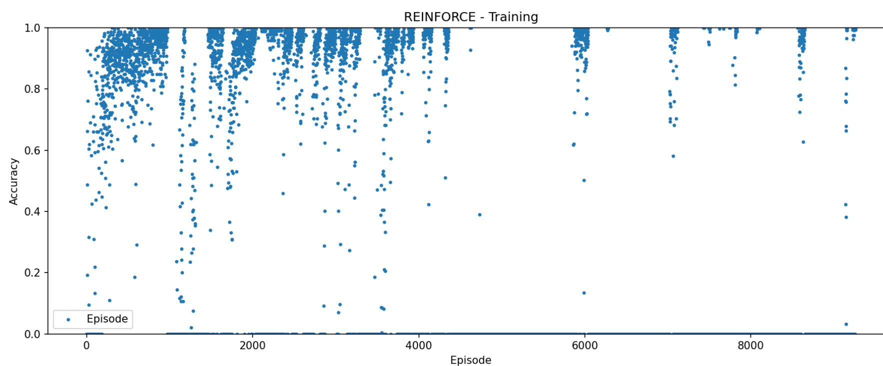


Figur 4.2: Accuracy under testning för Random Agent.

Grafen visar hög varians med många episoder vid både 0 och nära 1 i accuracy, vilket återspeglar agentens slumpmässiga beteende på de osedda testkartorna.

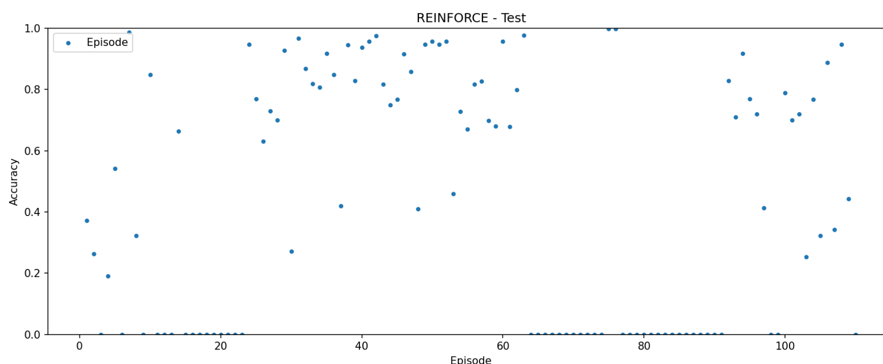
4.2 REINFORCE

Accuracy under träning och testning för REINFORCE visas i figur 4.3 och figur 4.4.



Figur 4.3: Accuracy under träning för REINFORCE.

Jämfört med Random Agent genomför REINFORCE betydligt färre episoder totalt, då den bemästrar kartor och avancerar vidare. Grafen uppvisar hög varians vilket är vanligt för REINFORCE, men visar också att algoritmen tidvis uppnår hög accuracy under träningen.

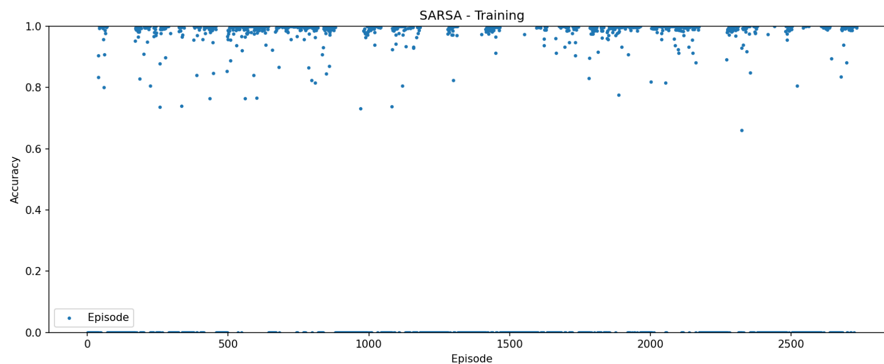


Figur 4.4: Accuracy under testning för REINFORCE.

Grafen uppvisar hög varians med punkter spridda mellan 0 och nära 1, vilket indikerar att algoritmen tidvis lyckas lösa de osedda testkartorna men inte gör det konsekvent.

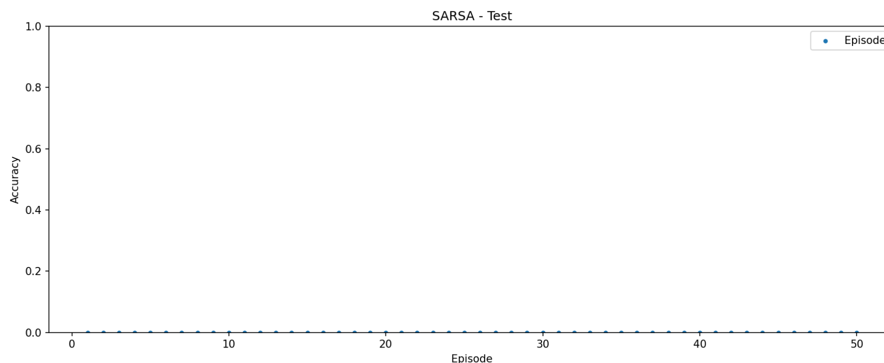
4.3 SARSA

Accuracy under träning och testning för SARSA visas i figur 4.5 och figur 4.6.



Figur 4.5: Accuracy under träning för SARSA.

Algoritmen konvergerar snabbt mot hög accuracy och genomför betydligt färre episoder än Random Agent, vilket indikerar att SARSA effektivt lär sig att lösa träningskartorna. De enstaka punkterna vid 0 motsvarar episoder i början av träningen på en ny karta, innan algoritmen har lärt sig tillräckligt.

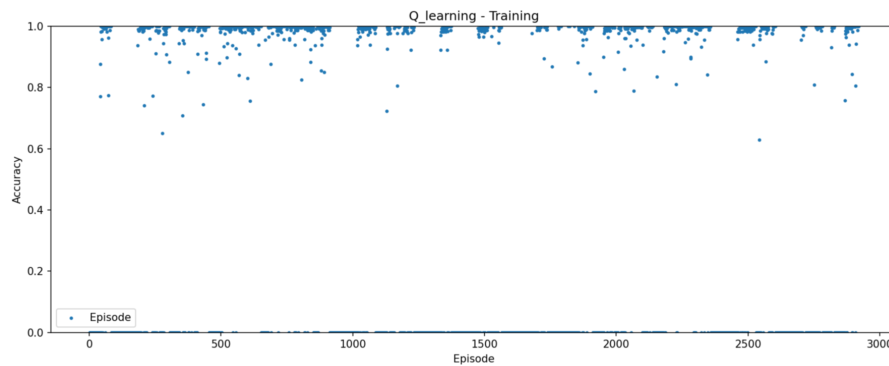


Figur 4.6: Accuracy under testning för SARSA.

Samtliga episoder uppnår 0 i accuracy, vilket innebär att algoritmen helt misslyckas med att lösa de osedda testkartorna. Detta beror på att de nya kartorna introducerar tillstånd som saknas i Q-tabellen.

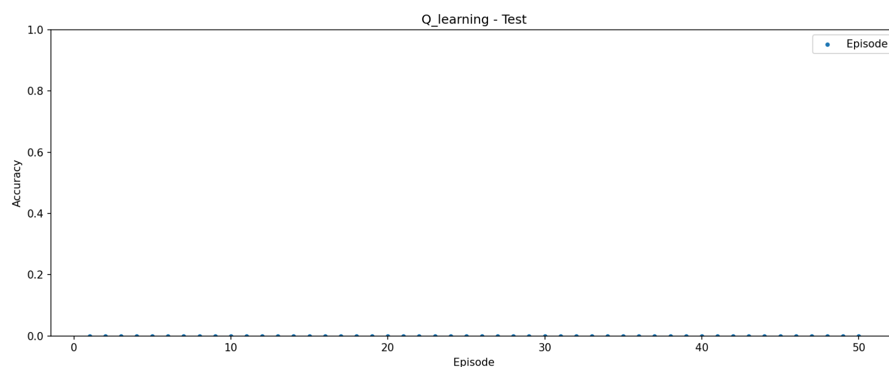
4.4 Tabellbaserad Q-learning

Accuracy under träning och testning för Q-learning visas i figur 4.7 och figur 4.8.



Figur 4.7: Accuracy under träning för Q-learning.

Liksom SARSA konvergerar Q-learning snabbt mot hög accuracy och genomför färre episoder totalt, vilket indikerar att algoritmen effektivt lär sig att lösa träningskartorna.

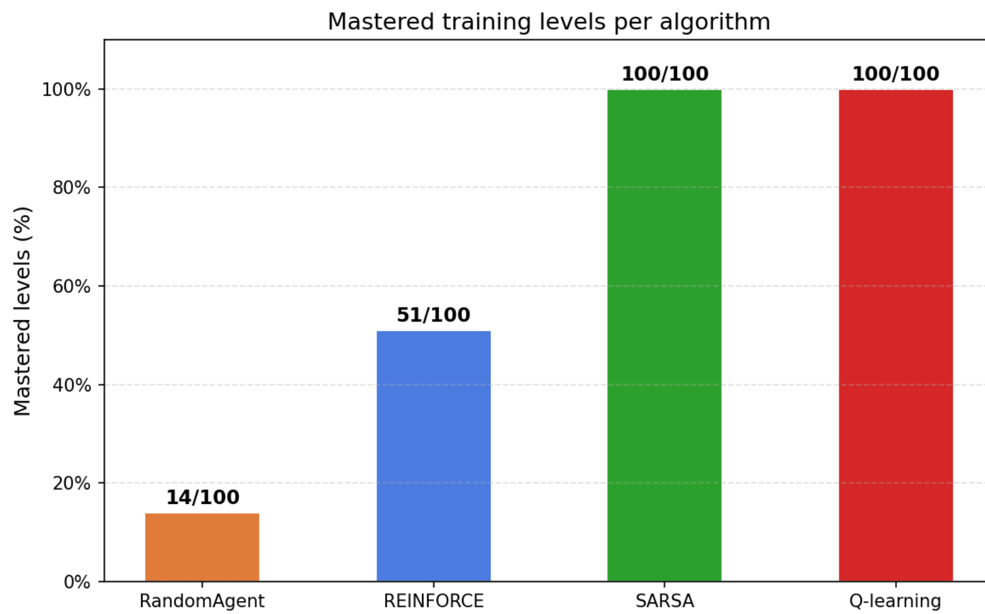


Figur 4.8: Accuracy under testning för Q-learning.

Precis som SARSA uppnår samtliga episoder 0 i accuracy, vilket bekräftar att tabellbaserade algoritmer inte kan generalisera till osedda kartor.

4.5 Mastery Rate

Antal bemästrade träningskartor per algoritm visas i figur 4.9.

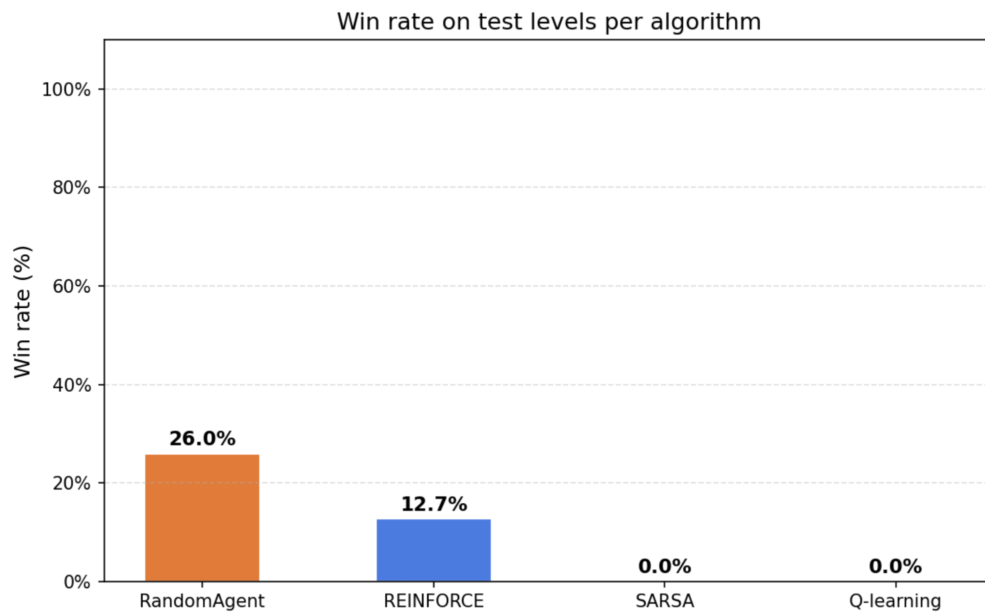


Figur 4.9: Mastery rate per algoritm.

SARSA och Q-learning bemästrar samtliga 100 kartor, medan REINFORCE bemästrar 51 och Random Agent endast 14.

4.6 Win Rate

Win rate under testning per algoritm visas i figur 4.10.



Figur 4.10: Win rate under testning per algoritm.

Random Agent uppnår högst win rate med 26%, följt av REINFORCE med 12.7%, medan SARSA och Q-learning uppnår 0%.

5
Diskussion

5.1 Analys av resultat

Resultaten visar att samtliga undersökta RL-metoder misslyckas med att generalisera tillräckligt väl för att lösa nya, osedda kartor. REINFORCE visar på viss förmåga att generalisera tack vare sitt neurala nätverk och klarar av att lösa vissa kartor, men lyckas inte prestera bättre än Random Agent.

Random Agents höga accuracy är missvisande då agenten genomför betydligt fler episoder än de övriga algoritmerna, eftersom den sällan bemästrar en karta. Mastery rate-diagrammet ger en tydligare bild av detta då Random Agent bemästrar inte fler än 14 kartor, medan REINFORCE bemästrar 51, och både SARSA och Q-learning bemästrar samtliga 100 kartor.

REINFORCE har ett neuralt nätverk och en väldefinierad policy, men problemet är att policyn uppdateras baserat på den ackumulerade belöningen i slutet av varje episod. Detta leder till hög varians och därmed varierande accuracy och inte lika många bemästrade kartor som SARSA och Q-learning.

SARSA och Q-learning uppnår 100% accuracy och bemästrar samtliga träningskartor. Detta är inte oväntat då båda algoritmerna uppdaterar Q-värden efter varje handling, vilket snabbt fyller Q-tabellen med korrekta Q-värden. Under testning löser de däremot inga kartor, då de nya kartorna introducerar tillstånd som saknas i Q-tabellen. Tabellbaserade algoritmer memorerar helt enkelt lösningarna för träningskartorna och lär sig inte att generalisera.

Win rate-diagrammet bekräftar detta, Random Agent och REINFORCE är de enda algoritmerna som lyckas lösa några testkartor, medan SARSA och Q-learning inte löser någon enda. Det faktum att REINFORCE inte lyckas prestera bättre än en slumpmässig agent tyder på att algoritmen inte generaliserar tillräckligt väl.

5.2 Val av metoder och tekniker

Unity valdes initialt eftersom spelmotorn erbjuder färdiga verktyg för utveckling av spelmiljöer samt direkt stöd för förstärkningsinlärning genom Unity ML-Agents. Detta gjorde det möjligt att snabbt skapa en fungerande Sokoban-miljö med visuell representation och att verifiera att observationer, handlingar och belöningar

fungerade korrekt genom den inbyggda PPO-algoritmen

I efterhand bedöms valet av Unity inte ha varit optimalt för just detta projekt. Sokoban är ett relativt enkelt rutnätsbaserat spel utan behov av avancerad grafik, fysik eller andra funktioner som normalt motiverar användningen av en spelmotor. En ren Python-implementation hade sannolikt varit tillräcklig för både utveckling, träning och utvärdering av algoritmerna. Detta hade förenklat projektets arkitektur och minskat beroendet av kommunikationen mellan Unity och Python.

PPO inkluderades initialt i arbetet eftersom algoritmen är inbyggd i Unity ML-Agents och lämpar sig väl för att verifiera att miljön, belöningsstrukturen och kommunikationen mellan Unity och Python fungerade korrekt. Den inledande testträningen visade att agenten gradvis kunde lära sig att lösa en mycket enkel karta, vilket indikerade att implementationen fungerade som avsett.

Samtidigt observerades att även denna förenklade träningsuppgift krävde relativt lång träningstid. Eftersom PPO-träningen var beroende av att Unity-miljön kördes aktivt via ML-Agents var det dessutom inte praktiskt möjligt att utnyttja Chalmers beräkningskluster Minerva på samma sätt som för de Python-baserade implementationerna.

Baserat på dessa begränsningar bedömdes PPO vara mindre lämplig för systematiska experiment på ett större antal kartor inom projektets givna tidsram. Fokus lades därför istället på REINFORCE, SARSA och tabellbaserad Q-learning, vilka kunde tränas betydligt mer effektivt i Python-miljön.

5.3 Val av kartor

Under projektet skapades 100 träningskartor och 10 testkartor. Detta antal valdes utifrån två faktorer. Dels bedömdes 100 träningskartor vara tillräckligt för att ge algoritmerna möjlighet att lära sig en variation av olika layouter, och dels begränsades antalet av träningstiden, eftersom ett större antal kartor hade medfört längre träningstid. De 10 testkartorna valdes då de ansågs vara tillräckligt för att ge en representativ bild av generaliseringsförmågan utan att förlänga på utvärderingstiden allför mycket. Det är dessutom svårt att skapa unika testkartor på 5×5 som inte liknar träningskartorna.

Eftersom kartorna är begränsade till 5×5 finns det ett begränsat antal möjliga layouter, vilket innebär att vissa kartor oundvikligen liknar varandra. Kartorna skapades därför manuellt för att minimera likheten mellan dem och säkerställa en rimlig variation. Samtliga kartor verifierades dessutom manuellt för att säkerställa att de är lösbara.

Utöver variation i layout eftersträvades även variation i svårighetsgrad. Majoriteten av kartorna utformades så att de kräver fler steg och mer strategi för att lösas, medan ett mindre antal enklare kartor inkluderades för att ge agenten möjlighet att lära sig grundläggande strategier.

De osedda testkartorna är av likvärdig svårighetsgrad som träningskartorna men

förekom aldrig under träningen. Detta innebär att en agent som har enbart memorerat lösningarna för träningskartorna kan inte lösa testkartorna utan att ha utvecklat en mer generell strategi. Syftet var därmed att utvärdera algoritmernas generaliseringsförmåga snarare än deras förmåga att memorera specifika lösningar.

5.4 Justering av hyperparametrar

Under projektet justerades flera hyperparametrar, detta inkluderar inlärningshastighet, antal steg per episod, antal steg per karta, kartstorlek och antal kartor. Inkluderingen av mastery visade sig göra störst skillnad under träning då det låter algoritmerna träna på kartor tills dem är lösta och de spenderar inte onödigt extra tid på en redan bemästrad karta.

För REINFORCE, som uppvisade hög varians, experimenterades det även med baseline [3, p. 329-331], vilket är tillägg till REINFORCE algoritmen som syftar till att reducera varians. Detta visade sig dock försämra inlärningen då belöningarna sällan översteg -1. Algoritmen fick lägre varians och blev mer konsekvent, men den övergripande prestandan blev mycket låg.

För SARSA och Q-learning presterade dessa metoder väldigt väl på alla kartor och det observerades inte göra någon signifikant skillnad att justera dessa parametrar. Vilket indikerar att dessa algoritmer presterar väldigt väl på just detta problem.

Trots dessa justeringar lyckades ingen av algoritmerna prestera väl under testning. REINFORCE var den enda algoritmen som visade förmåga att generalisera, men lyckades aldrig prestera bättre än Random Agent på de osedda testkartorn

5.5 Belöningssystem

I detta projekt implementerades ett belöningssystem som visade sig vara effektivt, då metoder som SARSA och Q-learning lyckades lösa samtliga träningskartor. Belöningarna justerades flera gånger under projektets gång för att hitta de värden som fungerade bäst. Det upptäcktes att relativt små bestraffningar för enkla fel kombinerat med större belöningar för bra beslut fungerade väl.

En stor belöning när en låda placeras på ett mål motiverar agenten att prioritera detta, medan en mindre belöning för att flytta en låda uppmuntrar agenten att interagera med lådan ofta. Bestraffningen per steg motiverar dessutom agenten att undvika att upprepa drag i onödan, då dessa ackumuleras till en stor negativ belöning över en hel episod.

En brist i det implementerade belöningssystemet är att en agent teoretiskt kan ackumulera en högre belöning genom att flytta en låda fram och tillbaka flera gånger innan den placeras på målet, jämfört med att placera lådan på målet direkt. Detta beror på att varje lådförflyttning ger en liten positiv belöning. I detta projekt förhindrades detta beteende i praktiken av kartornas begränsade storlek, då det inte finns tillräckligt med utrymme för att flytta en låda fram och tillbaka utan att ham-

na i en deadlock. På större kartor hade detta dock kunnat utgöra ett större problem, och en möjlig lösning hade varit att minska på belöningen för att flytta på lådan.

Det finns även andra alternativa belöningsstrukturer som inte undersöktes i detta projekt. Ett alternativ hade varit att tilldela en bestraffning varje gång en låda flyttas bort från ett målområde, vilket hade kunnat vara relevant för kartor med flera lådor. Ett annat alternativ hade varit att mäta avståndet mellan en låda och närmaste mål och tilldela belöning eller bestraffning baserat på om avståndet minskar eller ökar. Detta är dock ett tvivelaktigt tillvägagångssätt då ett minskat avstånd till målet inte nödvändigtvis innebär att kartan är närmare att lösas. En låda kan exempelvis fastna i en deadlock trots att den befinner sig nära målet. Av dessa anledningar undersöktes dessa alternativ inte i detta projekt.

5.6 Träningssystem

Träningssystemet utvecklades och förändrades flera gånger under projektets gång. Inledningsvis tränades alla algoritmer med ett fast antal steg per karta, vilket visade sig vara ineffektivt och gav dåliga resultat.

Problemet med ett fast antal steg är att en agent kan hitta den optimala lösningen på en karta långt innan det maximala antalet steg har uppnåtts. Agenten spenderar därmed onödig tid på en karta den redan behärskar. Detta är problematiskt av två anledningar. Träningstiden förlängs avsevärt samtidigt som risken för att agenten memorerar lösningar ökar. När agenten därefter byter till nästa karta tenderar den att upprepa tidigare inlärda beteenden, vilket försvårar inlärningen av nya strategier.

För att lösa dessa problem implementerades mastery-systemet, där agenten avancerar till nästa karta så snart den aktuella kartan anses bemästrad. Detta sparar träningstid och minskar risken för memorering, vilket resulterade i bättre och mer effektiv träning.

5.7 Sokoban som AI-problem

Sokoban är i grunden ett enkelt spel men är förvånansvärt svårt att lösa med datorer. En stor del av problematiken är att Sokoban är NP-hard, vilket innebär att antalet möjliga tillstånd växer exponentiellt med kartstorleken, med upp till 10^{98} på vissa kartor. För jämförelse är antalet möjliga kombinationer för ett Rubiks kub 10^{19}.

Det är fullt möjligt att lösa Sokoban med traditionella heuristiska sökningsmetoder som BFS eller A*, men endast för relativt små till medel stora kartor [4, p. 221-222]. Fördelen med att använda en AI modell är att den kan göra en prediktion av nästa handling baserat på ett fåtal observationer, utan att behöva söka igenom hela söktillståndsutrymmet.

Utmaningarna med AI sammanfattas i valet av rätt modell och tillräcklig med träning. I detta arbete har vi visat att tabellbaserade metoder som Q-learning presterar väl på små kartor, och att ett naturligt nästa steg är att kombinera dessa med neu-

rala nätverk. Hur väl detta översätter till större och mer komplexa kartor är något för framtida studier att undersöka.

5.8 Förbättringsområden

Bland förbättringsområdena är större kartor ett potentiellt förbättringsområde. För framtida arbete rekommenderas det att undersöka kartstorlekar större än 5×5, exempelvis 7×7 eller 10×10. Det finns dock en viktig nyans, större kartor löser inte det underliggande problemet. Att de undersökta algoritmerna misslyckas med att generalisera på små kartor indikerar att de sannolikt även kommer misslyckas på större kartor. Däremot skulle större kartor innebära att den slumpmässiga agenten presterar sämre, vilket skulle ge en tydligare bild av skillnaderna mellan algoritmerna. Självfallet rekommenderas det även mer träning på fler kartor samt ytterligare finjustering av hyperparametrar.

Ett annat förbättringsområde skulle vara att undersöka träning på kartor av varierande storlek. För att möjliggöra detta behöver observationsvektorn hanteras dynamiskt, så att agentens nätverk är tillräckligt stort för att hantera större kartor. För mindre kartor kan man exempelvis använda padding och fylla de resterande inputvärdena med nollor.

Ett annat förbättringsområde vore att undersöka Deep Q-Networks (DQN), som kombinerar Q-learnings uppdateringsregel med ett neuralt nätverk för approximation av Q-värden. Resultaten i detta arbete visar att tabellbaserad Q-learning presterar mycket väl på träningskartorna och bemästrar samtliga 100 kartor, men misslyckas helt på de osedda testkartorna. Detta tyder på att begränsningen inte främst ligger i Q-learnings uppdateringsregel utan i den tabellbaserade representationen, som endast kan lagra tidigare observerade tillstånd. Genom att ersätta Q-tabellen med ett neuralt nätverk kan modellen potentiellt generalisera mellan liknande tillstånd och därmed prestera bättre på nya kartor. DQN framstår därför som ett naturligt nästa steg för framtida arbete. Valet stöds av Mnih et al. [17], som introducerade Deep Q-Networks och visade att ett neuralt nätverk kunde användas för att approximera Q-värden direkt från pixeldata och framgångsrikt lära sig flera Atari-spel. Även Weber et al. [18] presenterade Imagination-Augmented Agents (I2A), en arkitektur som kombinerar modellfria och modellbaserade metoder genom att simulera möjliga framtida tillstånd innan beslut fattas. Studien utvärderades bland annat på Sokoban och visade förbättrad prestanda jämfört med flera baslinjemetoder. Vidare demonstrerade Silver et al. [19] med AlphaZero hur djupa neurala nätverk i kombination med trädsökning kan uppnå mycket stark prestation i komplexa problemlösningsmiljöer. Dessa arbeten visar hur moderna deep reinforcement learning-metoder kan hantera stora tillståndsrum och uppvisa bättre generaliseringsförmåga än klassiska tabellbaserade metoder. Valet stöds även av Mehta [20], som visade att Deep Q-learning kan prestera väl i pusselspel med väldefinierade tillstånd.

Ett ytterligare förbättringsområde skulle vara att hantera deadlocks. Ett förslag skulle vara att implementera en deadlock-detection som avslutar episoden med en stor negativ belöning när ett sådant tillstånd uppstår.

5.9 Användning av AI-verktyg

Under arbetets gång användes AI-verktyg såsom ChatGPT och Claude i begränsad omfattning som stöd i vissa moment. Verktygen användes främst för enklare felsökning av kod samt för att kontrollera språk, grammatik och formuleringar i rapporttexten.

AI-verktygen användes även vid enstaka tillfällen som stöd för LaTeX-formatering, exempelvis vid justering av figurer, referenser och dokumentlayout.

All implementation, analys, metodutformning och tolkning av resultat genomfördes av författarna själva. AI-verktygen användes enbart som kompletterande hjälpmedel.

6

Slutsats

Syftet med detta examensarbete var att undersöka om förstärkningsinlärning kan användas för att lösa spelet Sokoban samt att jämföra olika RL-algoritmers prestanda i samma miljö. För detta ändamål utvecklades en Sokoban-miljö i både Python och Unity, där algoritmerna REINFORCE, SARSA, tabellbaserad Q-learning samt PPO studerades.

Resultaten visar att tabellbaserade metoder såsom SARSA och Q-learning kan lära sig att lösa de träningskartor de exponeras för, men att de saknar förmåga att generalisera till nya kartor. Trots att båda algoritmerna bemästrade samtliga 100 träningskartor misslyckades de helt på de osedda testkartorna, vilket indikerar att de huvudsakligen memorerar tidigare observerade tillstånd.

REINFORCE uppvisade däremot viss generaliseringsförmåga tack vare användningen av ett neuralt nätverk. Algoritmen lyckades lösa vissa osedda testkartor och presterade bättre än de tabellbaserade metoderna under testning. Generaliseringen var dock inte tillräcklig för att uppnå stark prestanda, då REINFORCE fortfarande presterade sämre än Random Agent enligt win rate-måttet.

Resultaten visar även att utvärderingsmåtten påverkas starkt av kartornas storlek och antalet försök som genomförs. Att Random Agent uppnådde högst win rate trots avsaknad av inlärning indikerar att små kartor kan ge missvisande resultat om måtten inte tolkas tillsammans med mastery rate och övriga analyser.

Sammanfattningsvis visar arbetet att tabellbaserade RL-metoder inte är tillräckliga för att lösa Sokoban på ett generaliserbart sätt, medan neurala nätverk erbjuder viss generaliseringsförmåga men kräver mer avancerade metoder för att uppnå robust prestanda på nya kartor.

Litteraturförteckning

[1] D. Silver et al., “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, Jan. 2016. doi: 10.1038/nature16961.

[2] U. Raut, P. Galchhaniya, A. Nehete, R. Shinde, and A. Bhoite, “Unity ML-Agents: Revolutionizing Gaming Through Reinforcement Learning,” in *Proc. 2024 2nd World Conf. Commun. Comput. (WCONF)*, pp. 1–7, 2024.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., Cambridge, MA, USA: MIT Press, 2018.

[4] A. Junghanns and J. Schaeffer, “Sokoban: Enhancing general single-agent search methods using domain knowledge,” Artificial Intelligence, vol. 129, no. 1, pp. 219–251, 2001.

[5] Unity Technologies, “Maximize Multiplatform Game Development,” Unity. Accessed: Jun. 4, 2026. [Online]. Available: `https://unity.com/features/multiplatform`

[6] Unity Technologies,“Unity User Manual,” Unity Documentation. Accessed: Apr. 16, 2026. [Online]. Available: `https://docs.unity.com/`

[7] Unity Technologies,“ML-Agents Toolkit Documentation,” Unity Documentation. Accessed: Apr. 16, 2026. [Online]. Available: `https://docs.unity3d.com/Packages/com.unity.ml-agents@4.0/manual/index.html`

[8] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” Machine Learning, vol. 8, pp. 229–256, 1992.

[9] G. A. Rummery and M. Niranjan, “On-line Q-learning using connectionist systems,” Tech. Rep. CUED/F-INFENG/TR 166, Univ. Cambridge, Cambridge, U.K., 1994.

[10] C. J. C. H. Watkins and P. Dayan,“Q-learning,” Machine Learning, vol. 8, pp. 279–292, 1992.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” 2017, arXiv:1707.06347.

[12] PyTorch Contributors, “PyTorch Documentation,” Version 2.11. Accessed: Apr. 16, 2026. [Online]. Available: `https://docs.pytorch.org/docs/2.11/index.html`

[13] NumPy Developers, "NumPy," Accessed: Apr. 16, 2026. [Online]. Available: https://numpy.org/

[14] Python Software Foundation, "pickle — Python object serialization," Python 3 Documentation. Accessed: Apr. 16, 2026. [Online]. Available: https://docs.python.org/3/library/pickle.html

[15] D. Patel, D. Patel, and R. Kant, "Implementation of Reinforcement Learning on Cart-Pole System Using Deep Q-Network, REINFORCE a Policy-Based, and Advanced Actor-Critic Methods," in *Proc. 2025 IEEE Int. Conf. Intell. Signal Process. Effective Commun. Technol. (INSPECT)*, 2025.

[16] Y. Shoham and G. Elidan, "Solving Sokoban with Forward-Backward Reinforcement Learning," in *Proc. 14th Int. Symp. Combinatorial Search (SoCS 2021)*, pp. 191–193, 2021.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," 2013, arXiv:1312.5602.

[18] T. Weber, S. Racanière, D. Reichert, L. Buesing, A. Guez, D. Jimenez Rezende, A. Puigdomènech Badia, O. Vinyals, N. Heess, Y. Li, et al., "Imagination-Augmented Agents for Deep Reinforcement Learning," 2017, arXiv:1707.06203.

[19] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., "A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play," Science, vol. 362, no. 6419, pp. 1140–1144, 2018.

[20] A. Mehta, "Reinforcement Learning For Constraint Satisfaction Game Agents (15-Puzzle, Minesweeper, 2048, and Sudoku)," 2021, arXiv:2102.06019.

[21] L. Zhong, "Comparison of Q-learning and SARSA Reinforcement Learning Models on Cliff Walking Problem," in *Proc. 2023 Int. Conf. Data Sci., Adv. Algorithm Intell. Comput. (DAI 2023)*, Advances Intell. Syst. Res., vol. 180, pp. 207–213, 2024.

[22] M. Karppa, "Minerva cluster," GitLab repository, Chalmers University of Technology, 2024. Accessed: Apr. 16, 2026. [Online]. Available: https://git.chalmers.se/karppa/minerva/-/blob/main/README.md?ref_type=heads

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS