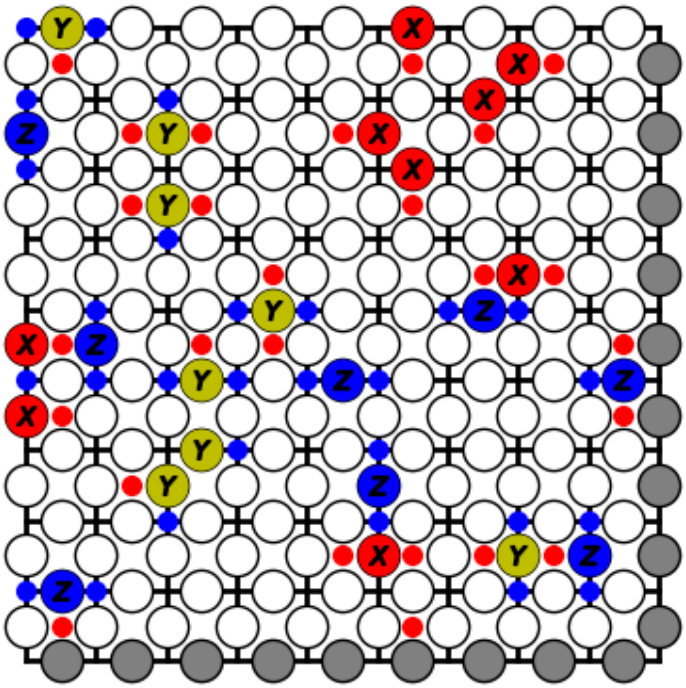
CHALMERS



Felkorrigering av kvantbitar på torisk kod för kvantdatorer

En applicering av djup förstärkningsinlärning och Monte Carlo-trädsökning

Kandidatarbete i fysik

Marcus Remgård, Christian Nilsson, Mikkel Opperud, Simon Sundelin, Joel Erikanders, Joel Harf Abili

Institutionen för fysik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2020

Felkorrigering av kvantbitar på torisk kod för kvantdatorer - En applicering av djup förstärkningsinlärning och Monte Carlo-trädsökning
Marcus Remgård[c], Christian Nilsson[c], Mikkel Opperud[c], Simon Sundelin[c], Joel Erikanders[c], Joel Harf Abili[c]

E-post:
[c]marrem@student.chalmers.se
[c]christni@student.chalmers.se
[c]mikkelo@student.chalmers.se
[c]simsunde@student.chalmers.se
[c]erijoel@student.chalmers.se
[c]abili@student.chalmers.se

Handledare: Mats Granath
Examinator: Lena Falk

Kandidatarbete 2020: TIFX04-20-73
Institutionen för fysik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Framsida: Visualisering av en torisk kod gjord i Python som illustrerar möjliga bitflipps X,Y och Z samt deras resulterande syndrom som röda och blå punkter.

Typsatt i LaTeX
Göteborg, Sverige 2020

Sammandrag

Potentialen för kvantdatorer är i dagsläget stor eftersom de kan komma att användas från allt till kryptering till att simulera komplexa naturvetenskapliga system. Detta är möjligt tack vare kvantdatorns kvantmekaniska egenskaper som möjliggör att kvantbitarna kan befinna sig i en superposition av två tillstånd. En av utmaningarna gällande att arbeta med kvantdatorer är att de ingående kvantbitarna är känsliga för störningar. Det kan uppstå bit- och fasflipp-fel som kan göra praktiska beräkningar omöjliga. Ett steg mot att lösa detta problem är att placera kvantbitarna på en torisk kod som skyddar mot fel och möjligt använda en avkodare för felkorrigering. I denna rapport kommer två typer av avkodare presenteras. Den första avkodaren använder ett sedan tidigare tränat artificiellt neuralt nätverk för att guida en Monte Carlo-trädsökning (MCTS). Den andra metoden bygger på att med djup förstärkningsinlärning träna ett nätverk med hjälp av MCTS. Beroende på initialtillstånd applicerar avkodaren Paulioperatorer X, Y och Z på lämpliga platser på den toriska koden fram tills det att felen är korrigerade. Det visade sig att den nätverksguidade MCTS-avkodaren hade större sannolikhet att lyckas med felkorrigeringen än tidigare algoritmer som baseras på djup förstärkningsinlärning och minimum-weight-perfect-matching (MWPM). Detta gällde för systemstorlekar $d \leq 9$ fast på en liten bekostnad av tid. Nätverken som tränades med MCTS lyckades inte alltid korrigera felen men framgångsrika korrigeringar gick snabbare än MCTS-avkodaren. Träningen av nätverken konvergerade dessutom snabbt mot lösningen för $d \leq 11$ och potential finns för förbättring. Den fullständiga koden tillsammans med färdigtränade nätverk finns tillgänglig på projektets GitHub.

Abstract

The future prospect for quantum computers is great because of their potential use in quantum cryptography and in simulating complex systems. This is possible thanks to the quantum mechanical properties of the quantum computer where the qubits can be in a superposition of two states. However, one of the challenges regarding working with quantum computers is that the quantum bits are unstable and sensitive to interference. This can cause bit flip and phase flip errors which could disrupt the calculations. One step towards solving this problem is to arrange the quantum bits on a toric code and use a decoder for error correction. In this report, two types of decoders will be presented. The first decoder uses a previously trained artificial neural network to guide a Monte Carlo tree search (MCTS). The second method is based on deep reinforcement learning to train a network with the help of MCTS. Depending on the initial state, the decoder applies Pauli operations X, Y and Z on the suitable place on the toric code until the errors are corrected. It turned out that the network-guided MCTS decoder had higher success than previous algorithms based on deep reinforcement learning and minimum-weight-perfect-matching (MWPM) for system sizes $d \leq 9$, at a small cost of time. The networks trained with the help of MCTS were not always able to correct the errors but successful correction went faster than the MCTS decoder. In addition, the training of the networks quickly converged towards the solution for $d \leq 11$ and there is potential for improvement. The full code along with pre-trained networks are available on the GitHub for this project.

Förord

Vi vill rikta ett stort tack till vår handledare Mats Granath som guidat oss genom detta väldigt intressanta arbete. Dessutom vill vi även tacka David Fitzek som i början presenterade den kod som låg till grund för vårt arbete.

Innehåll

1 Inledning

Vanliga datorer har revolutionerat vårt samhälle och många forskningsområden är helt beroende av dem för att utföra krävande beräkningar och simuleringar. I början av 1980-talet föreslogs en idé av fysikern Paul Benioff som bygger på en kvantmekanisk modell av den klassiska Turingmaskinen [1]. Denna idé står till grund för s.k. kvantberäkning och man insåg fort att kvantdatorer hade potential att utföra vissa beräkningar och simuleringar mycket snabbare än en klassisk dator [2]. Kvantberäkning grundar sig i användandet av s.k. kvantbitar som är den minsta enheten för kvantinformation. De påminner om vanliga bitar inom klassisk beräkning men till skillnad från dessa, som antingen är 0 eller 1, kan kvantbitar befinna sig i en kvantmekanisk superposition av 0 och 1 [3].

Ett problem med kvantdatorn är att dess kvantbitar är fundamentalt instabila, vilket innebär att kvantfel är oundvikliga [4]. Kvantfelen består då av bit- eller fasflipps [5] vilka kan liknas med en bitflipp i en vanlig dator där exempelvis $1 \rightarrow 0$. Detta skulle resultera i att praktiska beräkningar med en kvantdator blir mycket svårt. För att effektivt kunna använda en kvantdator ställs därmed höga krav på felkorrigering. För att korrigera de kvantfel som kan uppstå bör någon form av avkodare användas. Ett möjligt tillvägagångssätt för att ta fram en sådan avkodare är att träna en agent med hjälp av djup förstärkningsinlärning (DRL), se arbete av Fitzek et. al samt Andreasson et. al [6]. Principen är sådan att man bygger upp en s.k. logisk kvantbit som är en sammansättning av ett antal fysiska kvantbitar och representerar denna i en torisk kod likt Figur 1. Istället för att mäta kvantfelen direkt tittar man på syndrom som är paritetsmätningar runt varje fysisk kvantbit. Det neurala nätverkets uppgift blir att ta det aktiva tillståndet och generera Q-värden för de möjliga handlingarna. För varje position på den toriska koden kan tre olika handlingar utföras bestående av Paulioperatorern X, Y och Z. Varje Q-värde för en handling X, Y eller Z vid en viss position är då ett mått på hur bra handlingen är i förhållande till syftet att eliminera samtliga fel. I arbetet av Fitzek et. al tränades framgångsrikt nätverk som avkodare som lyckades bättre med felkorrigering än standardalgoritmen.

I arbetet AlphaGo Zero presenterat av Silver et. al [7], där målet var att träna en agent att spela Go, används också förstärkningsinlärning tillsammans med ett djupt neuralt nätverk. I detta fall matas nätverket med en representation av det aktuella spelbrädet och returnerar sannolikheterna att göra en viss handling samt ett värde som uppskattar sannolikheten att vinna spelet utifrån det aktuella spelbrädet. Utöver detta utförs i varje spelbräde en s.k. Monte Carlo-trädsökning (MCTS) som i princip givet ett tillstånd bygger upp ett träd med möjliga sätt att ta sig vidare i spelet från just det tillståndet [8, p. 113]. I slutet av trädsökningen returneras en sannolikhetsfördelning över de olika handlingarna för ursprungstillståndet (det aktuella spelbrädet). Skillnaden mellan trädsökningens och det neurala nätverkets sannolikhetsfördelningar är att trädsökningen oftast ger en bättre fördelning över handlingarna, s.a. bra handlingar får en högre sannolikhet medan dåliga handlingar får en lägre sannolikhet. Idén var att man med hjälp av trädsökningen kunde träna

det neurala nätverket genom att uppdatera dess parametrar s.a. nätverkets sannolikhetsfördelning liknar den fördelning som trädsökningen ger.

Syftet med detta arbete var att utveckla två typer av avkodare baserat på arbetet av Fitzek et. al och Silver et. al. I den första avkodaren som utvecklades användes de sedan tidigare tränade artificiella neurala nätverken för att guida en MCTS till att korrigera felen. Trädsökningen utforskade då givet ett syndrom flera möjliga sätt att korrigera felen på. Detta kan liknas med hur AlphaGo Zero utforskade möjliga handlingar givet ett tillstånd på brädet. För den andra typen av avkodare tränades nya nätverk från grunden till att korrigera felen. Detta gjordes genom att låta nätverket som tränades guida MCTS som då generera Q-värden som i sin tur användes för träning. Hypotesen var att båda avkodarna skulle resultera i en ökad sannolikhet i lyckad korrigering jämfört med tidigare algoritmer. Antagandet görs på grund av MCTS bidrag till att utforska flera olika sätt att korrigera samma syndrom.

2 Bakomliggande teori

2.1 Felkorrigering

Felkorrigering är en teknik som behövs för att information inte ska gå förlorad när den utsätts för slumpmässiga fel. I det klassiska fallet där information beskrivs av en serie av 1'or och 0'or kan dessa bitarna utsättas för s.k. bitflipps där $0 \rightarrow 1$ och $1 \rightarrow 0$. För att förhindra att informationen förstörs eller för att detektera fel använder man sig av flera bitar för att representera varje bit. Om få av bitarna förändras så kan detta därmed upptäckas och korrigeras. Ett exempel på detta är repeteringskoden. Här representeras 1'or och 0'or genom: $0 \rightarrow 000$, och $1 \rightarrow 111$. Om ett fel uppstår till exempel $000 \rightarrow 010$, kan detta korrigeras genom att göra om det till tillståndet som korresponderar med de bitarna som det är flest av, alltså: $001, 010,$ eller $100 \rightarrow 000$.

I det kvantmekaniska fallet är allmänt varje kvantbit i en superposition av tillstånden 1 och 0 [9, p. 13-14]. Här uppstår det såväl som bitflipp-fel två typer av fasflipps-fel. Dessa fel kan beskrivas med de tre Paulioperatorerna X, Y, och Z. En mätning av respektive operator ger resultatet 1 eller 0. Sedan gäller det att Y är en kombination av X och Z, alltså $Y \sim XZ$.

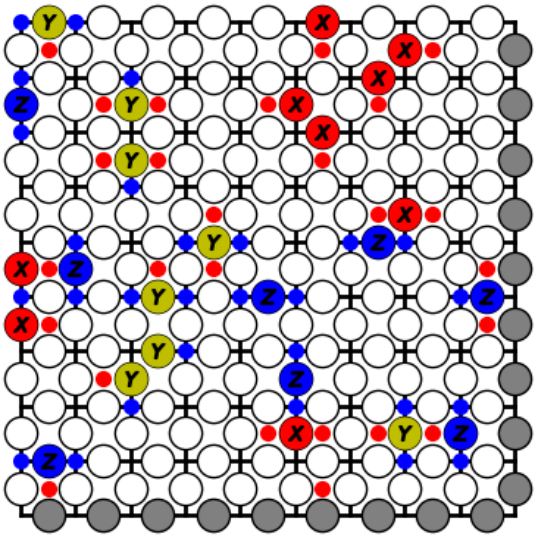
För att korrigera sådana fel används (som i det klassiska fallet) en kodning där så kallade fysiska kvantbitar används för att representera en logisk kvantbit. Däremot är det inte möjligt som i det klassiska fallet att direkt mäta de olika felen, då man i sådana fall kollapsar kvantillståndet och därmed förstör kvantberäkningen.

För att undvika detta används extra kvantbitar, som kallas för ancillabitar för att mäta felen indirekt. Detta görs genom att mäta stabiliseringsoperatorerna till koden [10]. Dessa är flerbitsoperatorer som kommuterar med operatorerna som används för att mäta tillståndet hos den kodade logiska kvantbiten (alltså dess X, Y och Z ope-

ratorer). Därmed fås ingen information om den kodade logiska kvantbitens tillstånd och således förhindras kollaps. Dessa stabiliseringsoperatorerna brukar oftast mäta pariteten hos flera kvantbitar. Det vill säga om antalet mätningar som ger +1 för en operator (X, eller Z) på n-st olika bitar är jämn eller udda. I detta fall kodas informationen på ett sådant sätt att mätningar av stabiliseringsoperatorerna på ett rent kvanttillstånd alltid ger jämn paritet. De mätningar som ger upphov till en udda paritet kallas för uppmätta defekter, och alla dessa defekter ger syndromet för felet som är det som används för att sedan korrigera felet med hjälp av en avkodare.

2.2 Torisk kod

Den toriska koden är ett exempel på en stabiliseringskod som visualiserar, rent topologiskt, problemet angående att rätta fel som uppstår i kvantberäkningar s.k. kvantfelskorrigering. Den toriska koden är definierad i planet genom ett tvådimensionellt rutnät med periodiska randvillkor och representeras i Figur 1. [11]



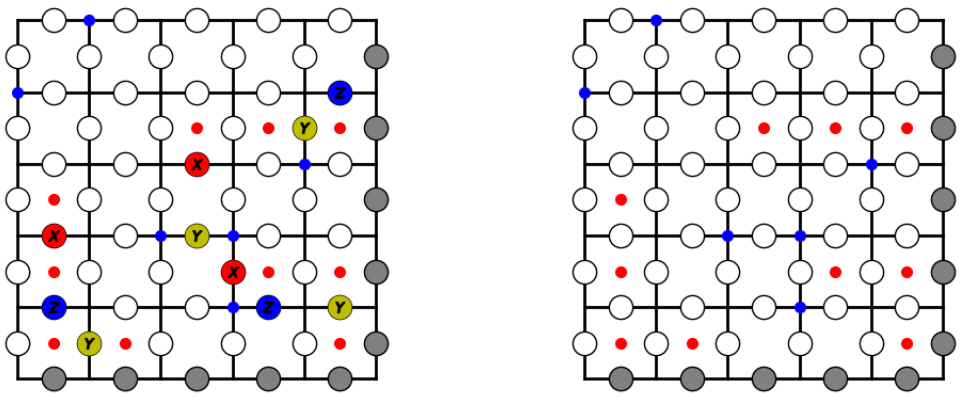
Figur 1: Torisk kod av storlek $d = 9$ med fel X,Y,Z och tillhörande defekter (röda samt blåa prickar). De röda X-operatorerna motsvarar bitflipps, de blå Z-operatorerna motsvarar fasflipps och de gula Y-operatorerna är en kombination av dessa. Cirklarna representerar de fysiska kvantbitarna och de gråa cirklarna symboliserar de periodiska randvillkoren.

Den toriska koden ovan är av storlek $d \times d$ med $d = 9$. Denna innehåller $2d^2$ noder (de stora cirklarna) vilka motsvarar kvantbitar. Det kan ses som att rutnätet är uppdelat i två lager, ett med plakettdefekter (de röda prickarna) och ett med vertexdefekter (de blå prickarna). En fördel med den toriska koden, för RL, är de periodiska randvillkoren som representeras av de gråmarkerade noderna i Figur 1. Dessa randvillkor skapar en translationsinvarians. Ett tillstånd under felkorrigeringen med ett antal defekter på specifika platser kan tack vare translationsinvariansen och de periodiska randvillkoren ses ur olika perspektiv. Detta innebär att man kan

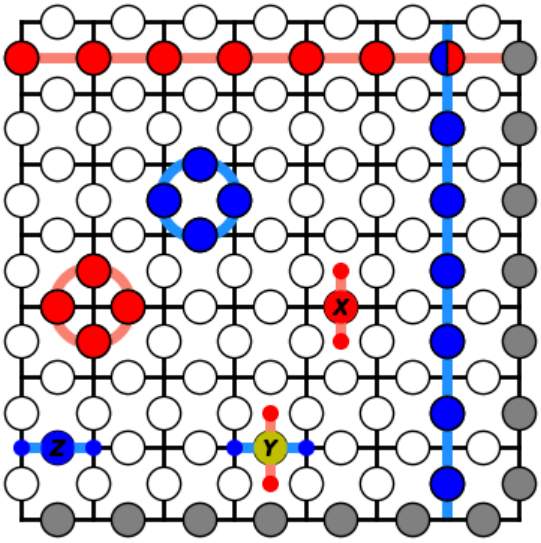
ha en så kallad händelse-plats i rutnätet dit man flyttar den kvantbiten som man vill operera på. Detta förenklar operationerna på den toriska koden då man kan flytta den kvantbiten man vill operera på till händelseplatsen. Detta möjliggör att avkodningsalgoritmen kan arbeta snabbare.

Huvudmålet med felkorrigeringen är att eliminera alla defekter. Detta görs som tidigare nämnt med hjälp av Paulioperatorerna X,Y och Z. En ensam bitflipp X eller fasflipp Z applicerad på ett tillstånd i kvantbitssektorn kommer att skapa ett specifikt nytt par av defekter (blå eller röda små prickar likt de i Figur 1) på intilliggande plaketter eller vertexer. Däremot kommer Pauli $Y \sim XZ$ att ge båda paren av defekter. Applicerar man till exempel en bitflipp på en bitflipp så kommer dessa att ta ut varandra och intilliggande plakettdefekter kommer försvinna.

När en avkodare ska utföra operationer på en torisk kod för att avlägsna alla defekter är det enbart syndromet som är synligt, se högra delen av Figur 2. Anledningen är som tidigare beskrivits att en mätning direkt på kvantbitarna skulle kollapsa vågfunktionen. Baserat på defekternas positioner ska avkodaren utföra en serie operationer som avlägsnar defekterna på det sätt som har lägst sannolikhet att resultera i en logisk bit- eller fasflipp. En sådan logisk flipp motsvaras av att det skapas en icke-trivial loop som spänner över hela den toriska koden vilket därmed resulterar i en misslyckad felkorrigering. Exempel på den enklaste formen av icke-triviala loopar illustreras i Figur 3 och motsvarar då slutna loopar över torusen som topologiskt inte går att dra ihop till en punkt. För en lyckad felkorrigering är det däremot tillåtet om det kvarstår triviala loopar som topolgiskt sätt går att dra ihop till en punkt. Enkla exempel på dessa triviala loopar illustreras i Figur 3. Som referens för hur en felkorrigering utförs presenteras i Appendix C en misslyckad felkorrigering där det uppstår en icke-trivial loop samt en lyckad korrigering, där det endast uppstår triviala loopar.



Figur 2: Till vänster ser man det faktiska felet och till höger ser man de uppmätta defekterna som är synligt för avkodaren och utgör själva syndromet.



Figur 3: Visualisering av två triviala loopar och motsvaras av de två slutna cirklarna. Vidare illustreras även icke-triviala loopar som spänner över hela torusen vilket skulle resultera i en misslyckad felkorrigering. Dessutom visas hur de olika operatorerna X, Y och Z verkar.

2.3 Felkorrigering med MWPM

Den felkorrigeringsalgoritm som används idag som baslinje för felkorrigeringsproblemet för den toriska koden är *Minimum Weight Perfect Matching* (MWPM)-algoritmen [12, 13, 14]. MWPM är en grafalgoritm som hittar den kortaste vägen att parvis matcha noder i en graf. I den toriska koden kan denna således användas för att hitta det kortaste sättet att para i hop defekter med minst möjliga steg. I rapporten används senare denna algoritm som en baslinje för avgöra hur bra vår felkorrigerningsmetod är.

2.4 Monte Carlo-trädsökning

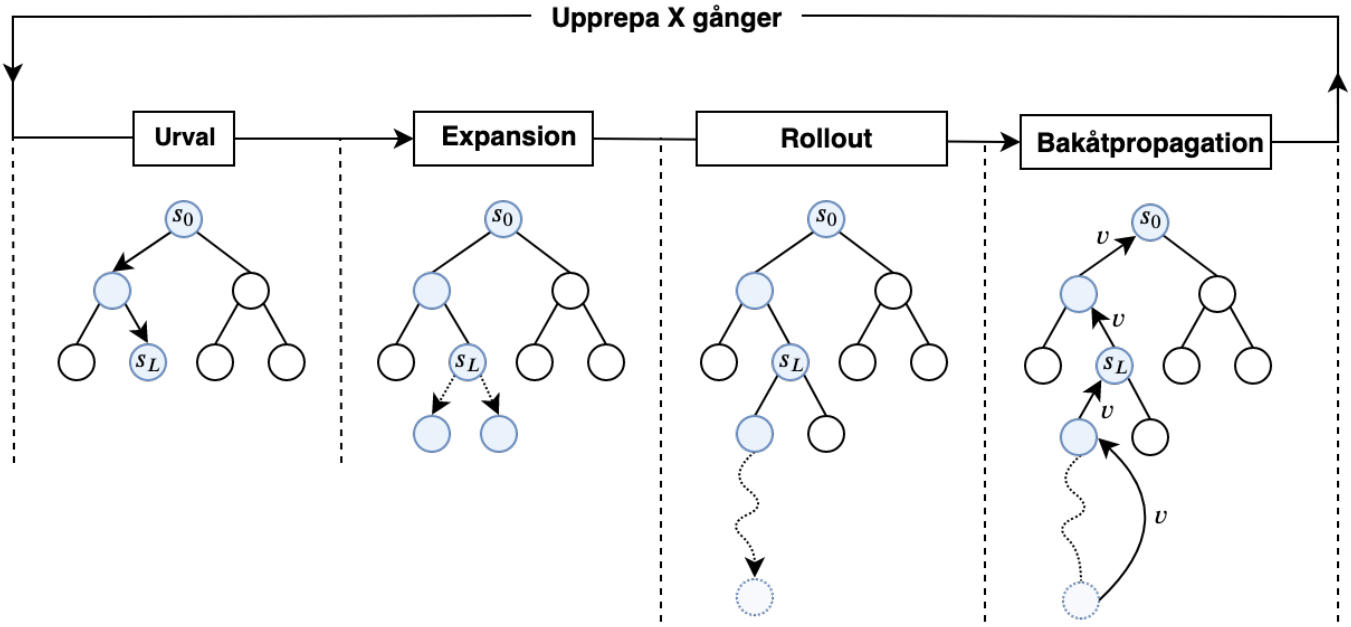
Monte Carlo-trädsökning (MCTS) är en form av sökalgoritm och kan appliceras i beslutsprocesser genom att simulera och utvärdera möjliga drag [8, p. 185]. Algoritmen ser i sin grundform ut som den i Figur 4 och består då av fyra faser som upprepas ett visst antal gånger. För kopplingen till felkorrigering av kvantbitar motsvaras hädanefter ett tillstånd av ett syndrom och en handling motsvaras av att en X,Y eller Z opperator appliceras på en av kvantbitarna i kvantbitssektorn.

Varje trädsökning består av en serie simulerade handlingar som påbörjas vid en rotnod s_0 och går ned till en lövnod s_L (nod utan några underliggande noder) vid tidssteget $t = L$ [8, p. 186-188]. I urvalsfasen görs ett urval bland de möjliga handlingarna som finns tillgängliga för att ta sig till nästa tillstånd. Urvalet görs med

hjälp av de set av parametrar som ligger sparade i grenarna mellan noderna och utgörs av,

$$\{N(s,a),\ W(s,a),\ Q(s,a)\}.$$

Här motsvarar $N(s,a)$ antalet gånger som MCTS gjort valet att utföra handlingen a från tillståndet s. Parametern $W(s,a)$ är det totala värdet för handlingen som erhålls i rollout fasen och Q-värdet $Q(s,a)$ är dess medelvärde och är ett mått på hur bra handlingen är. Stegvalet när $t < L$ görs då enligt en urvalsstrategi $\pi_t(a|s_t)$ som utformas för att balansera utforskning av trädet med utnyttjandet av vägar med höga $Q(s,a)$.



Figur 4: Algoritmen för MCTS i sin grundform. Steg tas från rotnoden s_0 fram till lövnoden s_L sker enligt urvalsstrategin π_t. Lövnoden expanderas och rolloutfasen förser trädet med värdet v för lövnoden. Detta värde propageras slutligen tillbaka upp i trädet.

Expansionsfasen inleds när MCTS hamnat vid en lövnod s_L [8, p. 186-188]. De underordnade noderna till s_L läggs till i trädet och grenparametrarna mellan lövnoden och varje ny expanderad nod sätts till $N(s_L,a) = W(s_L,a) = Q(s_L,a) = 0$. I nästa fas sker rollout där MCTS väljer en av de expanderade noderna och simulerar därifrån slumpmässigt fortsättningen tills det att ett fixt antal handlingar utförts eller ett slutmålet nåtts. Beroende på vilken situation som simuleras sparas någon form av belöning v. Slutligen sker bakåtpropagation när värdet v skickas uppför trädet enligt Figur 4. För varje steg $t < L$ mellan lövnoden och rotnoden uppdateras besöksantalet $N(s_t,a_t) = N(s_t,a_t) + 1$, totala värdet $W(s_t,a_t) = W(s_t,a_t) + v$ samt medelvärdet $Q(s_t,a_t) = \frac{W(s_t,a_t)}{N(s_t,a_t)}$.

Dessa fyra faser upprepas X antal gånger vilket ger upphov till ett träd där värdena $Q(s_0,a)$ i de olika grenarna representerar hur bra en viss handling är [8, p. 185]. Den

slutgiltiga handlingen kan därmed väljas genom att välja den handling från s_0 som motsvaras av högst Q-värde.

2.5 Q-inlärning

2.5.1 Klassisk Q-inlärning

Förstärkningsinlärning är en algoritm bestående av en agent som interagerar med en omgivning [8, p. 1-7]. Agenten utför handlingar a_t vid olika tidssteg t, vilket förflyttar omgivningen till nya tillstånd $s_t \rightarrow s_{t+1}$. Beroende på handling och omgivningens tillstånd erhåller agenten en viss belöning, r_{t+1}, baserad på förprogrammerade regler. Målet med algoritmen är att maximera den sammanlagda belöningen.

En agents sätt att agera beskrivs av en strategi, $\pi(s_t, a_t)$, vilket är en sannolikhetsfördelning som ger sannolikheten att agenten utför handlingen a_t i tillståndet s_t [8, p. 58]. För att hitta en optimal strategi behövs en uppskattning av hur bra olika tillstånd är att vara i eller hur bra olika handlingar är i de tillstånden. En sådan uppskattning är värdefunktionen, $v_\pi(s_t)$, som består av väntevärdet av den sammanlagda belöningen enligt

$$v_\pi(s_t) = E\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t, a_t\right], \tag{1}$$

där diskonteringsfaktorn $0 < \gamma < 1$ alltså avgör hur stor vikt som skall läggas på förväntade framtida belöningar.

Ett vanligt sätt för en agent att lära sig en värdefunktion är med hjälp av Q-inlärning [8, p. 131]. I praktiken kan en matris med dimensionerna $S \times A$ användas för att representera en Q-funktion, där $\max_a Q(s_t, a_t) \approx v_\pi(s_t)$. Inlärningen går då ut på att fylla hela Q-matrisen med korrekta förväntade belöningar associerade med olika (s_t, s_{t+1}, a_t). Träningen startar med att Q-matrisen fylls godtyckligt, exempelvis med nollor. Vid varje tidssteg uppdateras sedan Q-matrisen med hjälp av en förlustfunktion, $L(y, x)$, enligt

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha L(r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}), Q(s_t, a_t)), \tag{2}$$

där $0 < \alpha < 1$ är inlärningshastigheten. Vid klassisk Q-inlärning används $L(y, x) = y - x$, vilket gör att ekvation 2 resulterar i något som kallas Bellmanekvationen (ekvation 3). Rent intuitivt noteras att om $\alpha = 1$, reduceras Bellmanekvationen till $Q'(s_t, a_t) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1})$, vilket per definition ger den optimala $Q'(s_t, a_t)$ om $Q(s_{t+1}, a_{t+1})$ är optimal och omgivningen är deterministisk (det vill säga om (a_t, s_t) alltid resulterar i samma s_{t+1} och således samma r_{t+1}).

Ett generellt problem inom förstärkningsinlärning är att balansera utforskning mot exploatering. För att garantera en optimal strategi måste hela Q-matrisen fyllas och för det krävs att nya handlingar och tillstånd utforskas. För att agenten någon gång ska prestera väl krävs likväl att handlingar som ger hög belöning väljs. Ett tillvägagångssätt för att balansera utforskning mot exploatering är ϵ-greedy-strategin. I

ϵ-greedy-strategin väljs ett tal $0 < \epsilon < 1$ som ger sannolikheten (1 - ϵ) att handlingen med högst Q-värde väljs, annars väljs en slumpmässig handling.

2.5.2 Djup Q-inlärning

Problemet med förstärkningsinlärning är att det i praktiken bara är realiserbart på problem där tillståndsrummet är tämligen litet, dvs för miljöer som är små och diskreta. Många applikationer har emellertid oerhört stora tillståndsrum, vilket gör att det inte längre är möjligt att tabellera upp Q-värden för varje enskilt fall som i klassisk Q-inlärning. För dessa mer komplexa problem används istället djup Q-inlärning, där den huvudsakliga skillnaden är att man approximerar Q-värdet för ett givet tillstånd och en given handling $Q(s_t, a_t)$ med hjälp av ett neuralt nätverk. Detta gör att agenten kan tränas i att känna igen liknande situationer den tidigare befunnit sig i och därmed approximera Q-värden, istället för att behöva undersöka alla möjliga tillstånd för hela miljön.

För klassisk Q-inlärning kan Q-värden exempelvis uppdateras med Bellmanekvationen

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)], \tag{3}$$

där r_{t+1} är den faktiska belöningen för tillstånd s_t givet handling a_t, och där s_{t+1} samt a_{t+1} är efterföljande tillstånd respektive handling [8, p. 76]. Q-värden kan uppdateras enligt samma princip som ekvation 3 för djup Q-inlärning, med skillnaden att $Q(s_t, a_t)$ och $Q(s_{t+1}, a_{t+1})$ istället ges av nätverket. När nätverket sedan tränas uppdateras dess vikter genom att minimera skillnaden mellan $Q(s_t, a_t)$ och $r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1})$.

Att uppdatera $Q(s_t, a_t)$ och $Q(s_{t+1}, a_{t+1})$ under samma steg kan leda till divergenser i nätverket då dessa ofta är väldigt lika. För att göra nätverket mer stabilt krävs en uppdelning i två separata men identiska nätverk [5, p. 5]. Värdena på $Q(s_{t+1}, a_{t+1})$, även kallat målnätverket, kommer därmed frysas under ett antal iterationer innan dessa uppdateras.

Istället för förlustfunktionen $L(y, x) = y - x$ som används i Bellmanekvationen, kan exempelvis genomsnittet av felet i kvadrat (MSE) användas för djup Q-inlärning. Ett neuralt nätverks vikter uppdateras ofta med hjälp av flera data åt gången, då detta går snabbare att genomföra. MSE är bättre anpassad till detta då metoden använder sig av genomsnittet, $L(y, x) = \frac{1}{N} \sum_i^N (y_i - x_i)^2$, där N är antalet data per viktuppdatering.

En vanlig algoritm för att uppdatera ett neuralt nätverks vikter är gradientnedstigning [15, p. 172-173]. Gradientnedstigning går ut på att minimera förlusten med hjälp av gradienten av förlustfunktionen med avseende på nätverkets vikter. Algoritmen utnyttjar att gradienten pekar i den riktning funktionen ökar mest i, så för att minimera förlusten ändras vikterna så att ett steg tas i den negativa gradientens riktning. Stegstorleken bestäms av inlärningshastigheten $0 < \alpha < 1$.

Sammantaget kan viktuppdatering vid djup Q-inlärning med gradientnedstigning och förlustfunktionen MSE, beskrivas av ekvationen

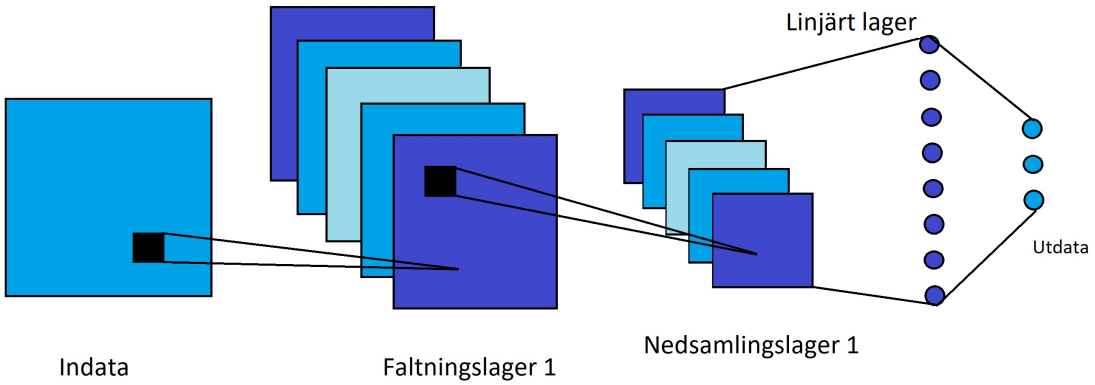
$$w' = w - \alpha \nabla_w \frac{1}{N} \sum_{i}^{N} (r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))^2, \tag{4}$$

där w är nätverkets vikter.

2.5.3 Faltningsnätverk

En väsentlig del av den djupa Q-inlärningen utgörs emellertid av själva nätverket, vilket är ett så kallat faltningsnätverk. Detta nätverk skiljer sig i många avseenden från ett vanligt linjärt nätverk, där varje neuron i föregående lager är kopplat till samtliga neuroner i nästkommande lager [16, p. 201]. Vid komplexa och stora typer av indata, såsom bilder, blir dessa linjära nätverk ofta överanpassade och har visat sig tämligen ineffektiva. Faltningsnätverk utnyttjar dock det faktum att avancerade strukturer ofta byggs upp av enklare underliggande mönster, och använder istället så kallade filter för att detektera olika strukturer. På så sätt blir nu varje neuron endast kopplad till en mindre del av det föregående lagret, istället för hela, vilket minskar komplexiteten avsevärt.

Själva kärnan i ett faltningsnätverk är således faltningslagrerna, i vilka själva filtersvepningarna äger rum [16, p. 202]. Andra viktiga delar av faltningsnätverket är så kallade nedsamplingslager, där man helt enkelt komprimerar informationen från indatan utan att de huvudsakliga strukturerna försvinner. Dessutom brukar ett linjärt lager läggas till i slutet av nätverket när den definitiva identifieringen skall göras, exempelvis för att bedöma huruvida en bild faktiskt är en hund eller katt. Strukturen för ett faltningsnätverk kan sammanfattas i figur 5.



Figur 5: Övergripande struktur över ett faltningsnätverk. Ofta förekommer flera faltningslager samt nedsamplingslager efter varandra innan datan slutligen processas av det linjära lagret.

2.5.4 Erfarenhetsrepris

Att få neurala nätverk att generalisera är ett centralt problem inom djupinlärning [15, p. 224]. Motsatsen till generalisering, när nätverket endast lär sig hur det ska agera mot redan använd data, kallas överanpassning. Överanpassning kan uppstå av flera anledningar. Ett exempel på anledning är när det finns en bias i datan, alltså när datans fördelning stämmer dåligt överens med populationens fördelning.

Ett relaterat fenomen kan uppstå till följd av temporal bias, dvs. när data nära varandra i tiden är kopplade på något sätt. Nätverket anpassar då sina vikter till datan det just har tränat med, men presterar inte nödvändigtvis bra med data från helt andra tidssteg. Detta kan leda till stora svängningar i inlärningen, då nätverket hela tiden ändrar sina vikter utefter datan det just har sett, utan att lyckas generalisera för populationen i sin helhet.

För att motverka temporal bias kan något som kallas erfarenhetsrepris användas. Erfarenhetsrepris går ut på att data samlas upp i en minnesbuffert för att sedan plockas ut i slumpmässig tidsordning vid inlärning.

Nätverket lär sig inte nödvändigtvis att hantera alla situationer lika snabbt, utan en del situationer är svårare. Prioriterad erfarenhetsrepris kan tillämpas för att hantera detta [17]. Från början väljs då all data med lika stor sannolikhet, men efter varje viktuppdatering ändras sannolikheten att den valda datan ska väljas igen, beroende på hur stort fel $|\delta_t| = |r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)|$ den ger upphov till. Sannolikheten att data från tidssteget t väljs, ges av

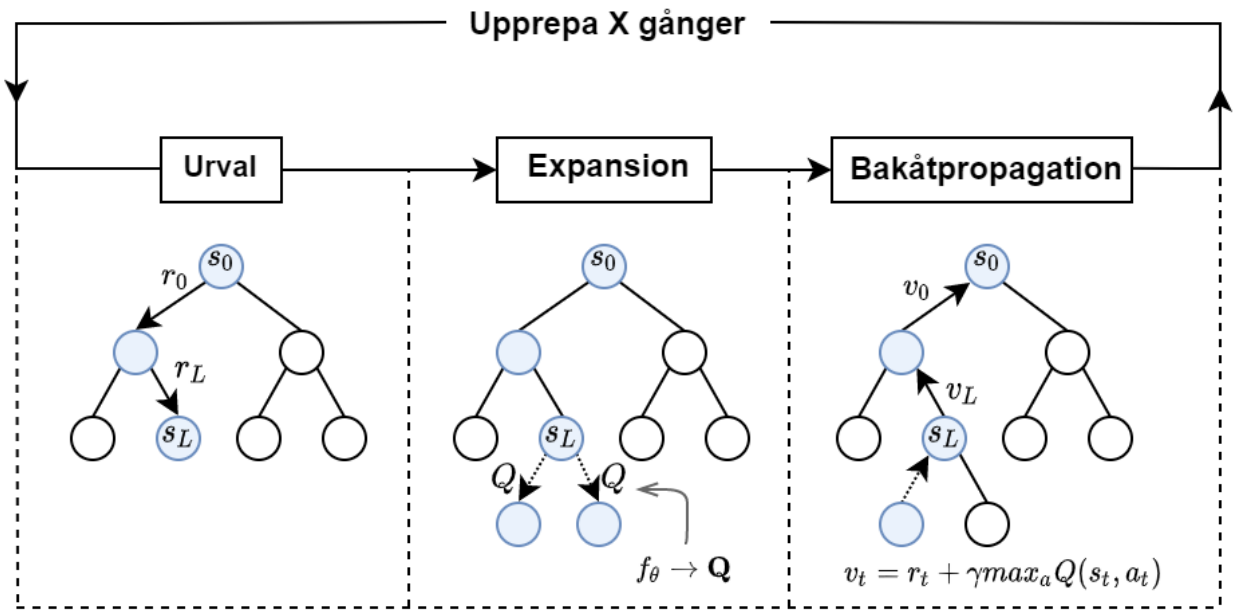
$$P_t = \frac{|\delta_t|^c}{\sum_i^M |\delta_i|^c}, \tag{5}$$

där M är storleken av minnesbufferten och $0 < c < 1$ är en konstant som avgör hur mycket prioritering som ska göras. Om till exempel $c = 0$ väljs datan likformigt. Detta ger dock upphov till att fördelningen av använd data förändras, vilket som sagt resulterar i försämrad generalisering. För att kompensera för detta används vikter, θ, som gör att data med hög prioritet istället leder till mindre uppdateringar av nätverkets vikter, w. Detta genom att förlusten ändras till $L' = \theta \cdot L$ där $\theta = (M \cdot P_t)^{-b}$ och $0 < b < 1$ är en konstant som avgör hur mycket kompensering som görs.

3 Funktionsbeskrivning

3.1 MCTS-algoritmen

För implementeringen av en MCTS-baserad avkodare modifierades MCTS från sin grundform som presenterades i teoriavsnittet till en förenklad variant av vad som användes i AlphaGo Zero [7]. Algoritmens struktur illustreras i Figur 6.



Figur 6: Algoritmen för MCTS. Steg från rotnoden s_0 fram till lövnoden s_L sker enligt en $\epsilon-$greedy-strategi och belöningar r_t samlas upp för varje tidssteg $t = 0, 1, ..., L$. Lövnoden expanderas med hjälp av ett tränat nätverk f_θ och tillhandahåller $max_a Q(s_t, a_t)$ som används för beräkning av värdet v_t. Detta värde propageras därefter tillbaka upp i trädet och uppdaterar grenparametrarna mellan s_L och s_0.

Rotnoden s_0 representerar det syndrom som matas in i MCTS och kommer från en torisk kod där X, Y och Z genererats slumpvis med lika stor sannolikhet P_e. Under urvalsfasen tillämpades en $\epsilon-$greedy-strategi för att välja en handling a. Antingen valdes en slumpmässig handling eller den med högst Q-värde, med 50% sannolikhet. Denna handling motsvarades av en X, Y eller Z opperator tillämpad på någon position på den toriska koden. Belöningen för en specifik handling som applicerades erhölls enligt

$$r_{t+1} = \begin{cases} 100 & \text{vid framgångsrik felkorrigering} \\ E_t - E_{t+1} & \text{annars} \end{cases} \tag{6}$$

och sparades som en grenparameter under urvalsfasen fram tills dess att en lövnod nåddes. Här motsvarade E_t och E_{t+1} antalet defekter på syndromet före respektive efter det att en handling utförts. Urvalsfasen fortsatte fram tills att en lövnod s_L nåddes.

Vid expansionsfasen approximerades initialt Q-värdena hos tillståndet, s_L, med hjälp av ett sedan tidigare tränat neuralt nätverk som självt hade kunnat agera som avkodare. Det högsta värdet i Q-matrisen, $max_a Q(s_t, a_t)$, som erhölls från nätverket motsvarade då den enligt nätverket, bästa handlingen från s_L. För systemstorlekar d = 5, 7 och 9 användes nätverken från arbetet av Fitzek et. al.

I nästa fas skedde bakåtpropagation uppför trädet där värdet $v_t = r_t + \gamma max_a Q(s_t, a_t)$

valdes. Här är r_t de tillhörande belöningarna som beräknades under urvalsfasen och γ en diskonteneringsfaktor. Varje besökt nod under simuleringen associerades med ett tidssteg $t = 0, 1, ..., L$. Med hjälp av v_L uppdaterades Q-värdena hos noden besökt vid tidssteget $t = L$. Därefter beräknades ett nytt värde v_{L-1} som användes för att uppdatera Q-värdena hos noden vid tidssteget $t = L - 1$. Detta upprepades enligt $Q'(s_{t-1}, a_{t-1}) \leftarrow r_t + \gamma max_a Q(s_t, a_t)$, för $t = L, L - 1, ..., 0$.

De tre faserna motsvarar tillsammans en simulering och för varje trädsökning som syftade till att välja bästa möjliga handling utfördes X simuleringar likt Figur 6. För varje simulering uppdateras något av Q-värdena ut från rotnoden enligt ekvation 1. Därmed resulterar flera simuleringar i en bättre approximation av rotnodens Q-värden. I algoritm 1 beskrivs en simulering

Algorithm 1: MCTS guidat av neuralt nätverk

```
function simulering(s):
    if s ej besökt tidigare then
        Approximera Q(s_t, a_t) med neuralt nätverk
        while t > t_start do
            Q'(s_{t-1}, a_{t-1}) ← r_t + γ max_a Q(s_t, a_t)
            t = t − 1
        end
        return
    end
    else
        Välj a_t med ϵ-greedy
        Utför a_t på den toriska koden och erhåll (s_{t+1}, r_{t+1})
        Simulering(s_{t+1})
    end
```

När X simuleringar hade slutförts och en trädsökning därmed avslutats kunde bästa möjliga handling väljas genom att välja den gren (handling) ut från rotnoden som motsvarade högst Q-värde. Den nod som man hamnade på efter denna handling sattes till den nya rotnoden i trädet. Eftersom grenparametrarna från den tidigare trädsökningen finns kvar utförs istället $\frac{X}{10}$ simuleringar från denna nya rotnod.

Denna procedur återupprepas fram tills dess att rotnoden motsvarar ett tomt syndrom alltså ett tillstånd utan några kvarstående defekter eller tills dess att 50 trädsökningar utförts. Om samtliga defekter eliminerats testades det om det hade uppstått icke-trivial loopar genom att kolla det fanns några kvarvarande X, Y eller Z opperatorer på den toriska koden. Om det kvarstod några och antalet X, Y eller Z var udda hade det uppstått en icke-trivial loop och felkorrigeringen hade därmed misslyckats. Det var dessutom av denna anledningen, att testa för icke-triviala loopar som endast udda storlekar på den toriska koden användes. Om däremot samtliga X, Y och Z försvunnit eller om bara triviala loopar kvarstod motsvarade detta en lyckad felkorrigering.

Algoritmen kördes på en Nvidia Tesla V100 SMX2 GPU på Chalmers C3SE kluster Vera. Värden för parametrar och andra relaterade värden som användes för MCTS avkodaren presenteras i Tabell 1.

Tabell 1: Olika värden som användes för MCTS avkodaren.

Antal simuleringar för första trädsökningen, d=5,7,9.	30,30,30
Antal simuleringar för resterande trädsökningar, d=5,7,9	3,3,3
Diskonteneringsfaktor γ	0.95

3.2 Nätverksalgoritmen

I de nätverksbaserade avkodarna användes MCTS för felkorrektion under träningen av nätverken och utformades precis som i avsnitt 3.1 med samma parametrar som i Tabell 1. För det guidande nätverket i MCTS användes däremot det nätverk som tränades. Integrationen av MCTS gjordes alltså i träningsdelen av den djupa Q-inlärningsalgoritm som användes av Fitzek et. al.

För ett träningssteg i träningsalgoritmen genererades inledningsvis en torisk kod där X, Y och Z fel genererats slumpvis med lika stor sannolikhet P_e. Därefter inleddes en loop där det genererade syndromet skickades in i MCTS. Från MCTS returnerades (Q, a, s) vilket motsvarar rotnodens Q-värden och deras korresponderande handlingar samt perspektiv. För de tillstånd under rotnoden som besökts mer än en gång under trädsökningen sparades (Q, a, s) i en minnesbuffert. Den handling som motsvarade högst Q-värde tillämpades och det nya syndromet skickades då återigen in i MCTS. Denna loop kördes fram tills dess att inga defekter längre kvarstod i syndromet (slutförd felkorrigering) eller tills det att 50 handlingar utförts. Efter det att loopen genomförts skedde erfarnhetsrepris om minst 400 datainlägg (Q, s, a) samlats upp i minnsebufferten. Vikterna hos nätverket uppdaterades då lika många gånger som antalet datainlägg i minnet. När en sådan träningsrunda var klar tömdes minnet. Pseudokoden för algoritmen för ett givet antal träningssteg återfinns i Algoritm 2.

Nätverksarkitekturen som användes för nätverken baserades på faltningsnätverk och designades av Fitzek et. al [5]. Denna arkitekturen anges i Tabell 4 i Appendix A. Vid själva träningen valdes initialt låga värden på P_e för att därmed kunna erhålla en snabbare konvergering av nätverket. Detta innebar att sannolikheten för att fel skulle uppstå på den toriska koden inledningsvis var tämligen låg men ökades efterhand. En modell som visade sig vara bra var att träna agenten i epoker om 1000 steg följt av 1000 test-korrigeringar. Efter varje epok ökades P_e från 0.01 i steg på 0.01 om minst 95% av test-korrigeringarna var lyckade. Denna process utfördes för samtliga storlekarna d = 5, 7, 9 och 11 på den toriska koden. Träningen skedde på en Nvidia Tesla V100 SMX2 GPU på Chalmers C3SE kluster Vera och de hyperparametrar som användes vid träningen presenteras i Tabell 2.

Algorithm 2: MCTS integrerat i djup Q-inlärning

```
for varje epok do
    while iterationer < antal träningssteg do
        T = Torisk kod
        Nod = None
        while Defekter kvarstår och antal tagna steg ≤ 50 do
            Nod(Q, a, s) = MCTS(T, Rotnod)
            Välj a_t som ger max_a(Nod.Q)
            Utför a_t på den toriska koden och erhåll s_{t+1}
            Spara alla besökta (Q, a, s) i minnet
            Nod = Nod[s_{t+1}]
        end
        if antal (Q, a, s) i minnet > 400 then
            for varje (Q, a, s) do
                Uppdatera vikterna hos nätverket mha en batch av (Q, a, s)
            end
            Minnet töms
        end
    end
end
```

Tabell 2: Träningsparametrar för träning av nätverken.

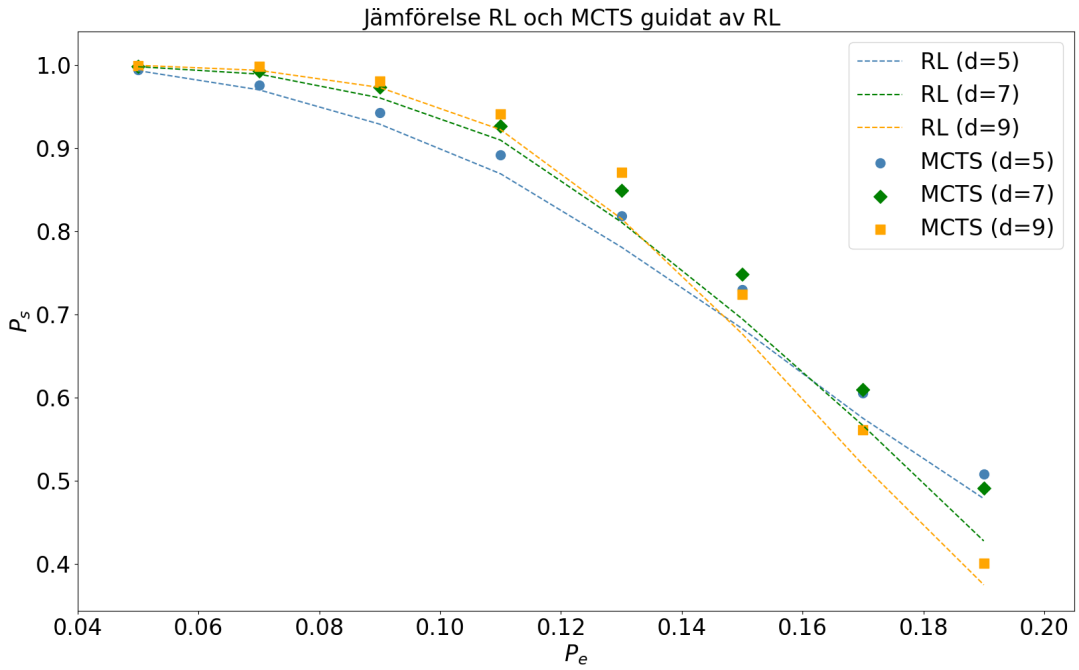
Hyperparameter	Värde	Beskrivning
Träningssteg	1000	Antal träningssteg per epok
Diskonteringsfaktor	0.95	γ som används i Q-inlärningen
Optimerare	Adam	Optimeringsalgoritm för uppdatering av nätverkets vikter
Inlärningshastighet	0.00025	

4 Resultat

Nedan följer resultat för hur de tränade nätverken samt MCTS avkodaren presterat för olika storlekar d på den toriska koden. I Appendix C presenteras resultatet av en misslyckad felkorrigering där det uppstår en icke-trivial loop samt en lyckad korrigering där det endast uppstår triviala loopar.

4.1 Avkodare - MCTS

I Figur 7 presenteras andelen lyckade felkorrigeringar P_s mot sannolikheten för fel P_e över intervallet $P_e \in [0.05, 0.19]$ i steg om 0.02 för storlekarna d=5, 7 och 9. Dessutom visas prestationen av nätverken som tränats med RL-algoritmen framtagen av Fitzek et. al. Varje datapunkt för MCTS motsvarar medelvärdet av 5000 felkorrigeringar och standardavvikelsen för var punkt blev $\sigma < 0.013$.



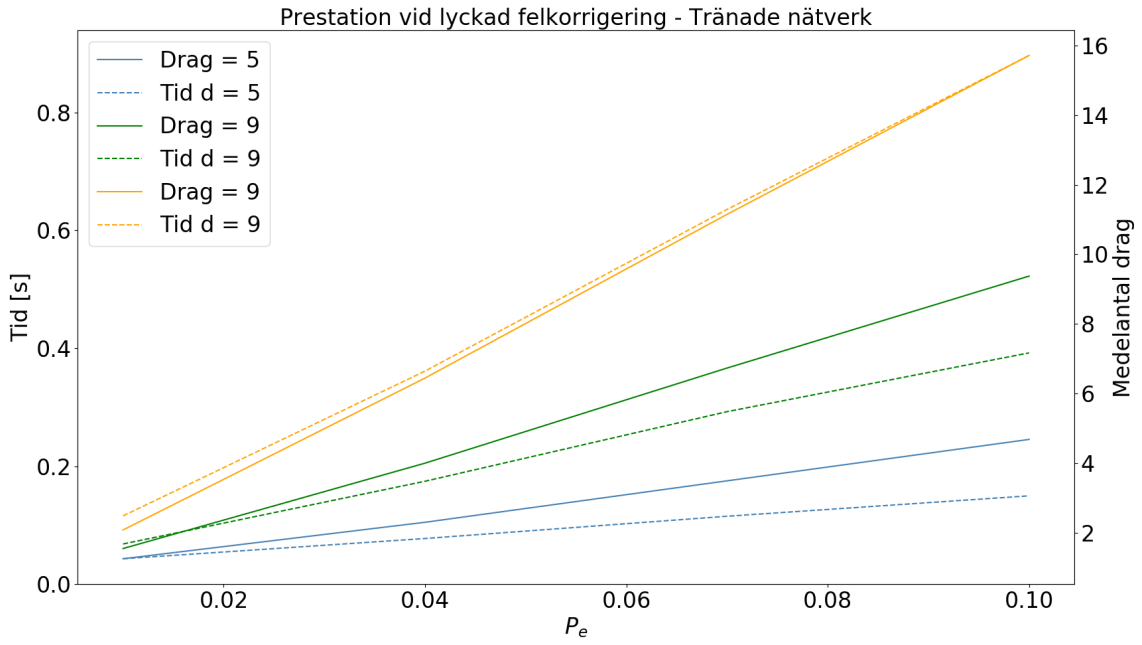
Figur 7: Jämförelse av prestanda, med arbetet av Fitzek et. al för storlekarna d=5 till 9.

Andelen lyckade felkorrigeringar P_s som berodde på att avkodaren lyckats eliminera alla fel X, Y och Z på den toriska koden presenteras i Tabell 3 som P_{s1}. Datan erhölls med $P_e = 0.1$ både för MCTS och RL med antalet felkorrigeringar likt ovan.

Tabell 3: Andelen lyckade felkorrigeringar som berodde på att avkodaren lyckats eliminera alla fel P_{s1} för $P_e = 0.1$.

	MCTS $d = 5$	**MCTS** $d = 7$	**MCTS** $d = 9$	**RL** $d = 5$	**RL** $d = 7$	**RL** $d = 9$
P_s	0.917	0.958	0.968	0.904	0.935	0.954
P_{s1}	0.748	0.6	0.448	0.734	0.59	0.42

I Figur 8 illustreras hur lång tid samt medelantalet drag som krävdes för en lyckad felkorrigering vid olika P_e. Medelvärdet togs över 3000 lyckade felkorrigeringar där standardavvikelsen för antalet drag var $\sigma < 0.3$. Antalet drag för RL avkodaren från Fitzek et. al visade sig behöva motsvarande antal som MCTS behövde (i Figur 8).



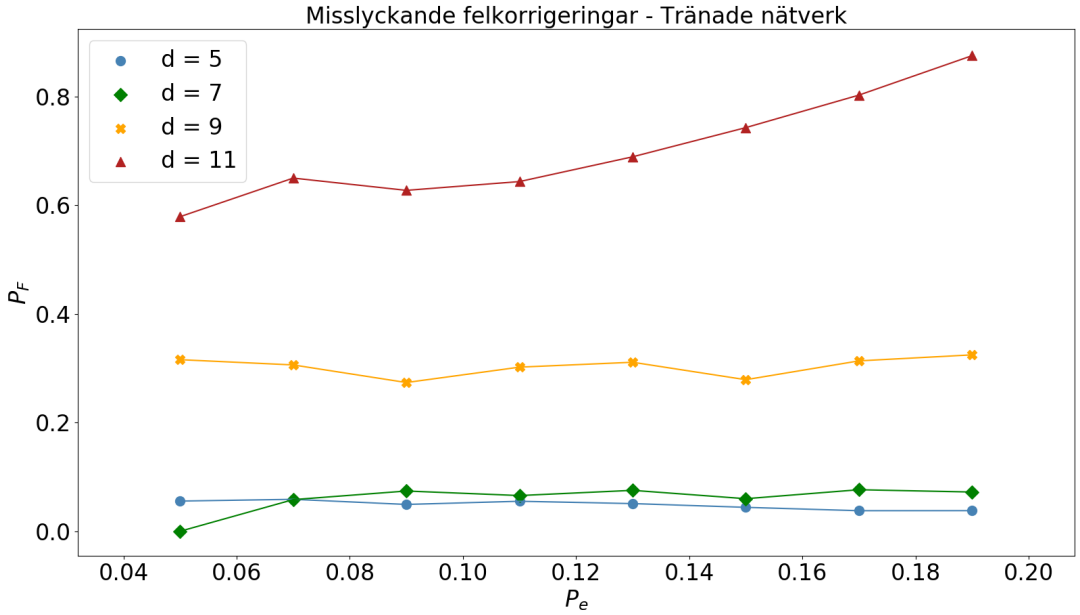
Figur 8: Tid och medelantal drag (handlingar) som krävdes för lyckad felkorrigering beroende på P_e för d=5,7 och 9.

4.2 Avkodare - Nätverk

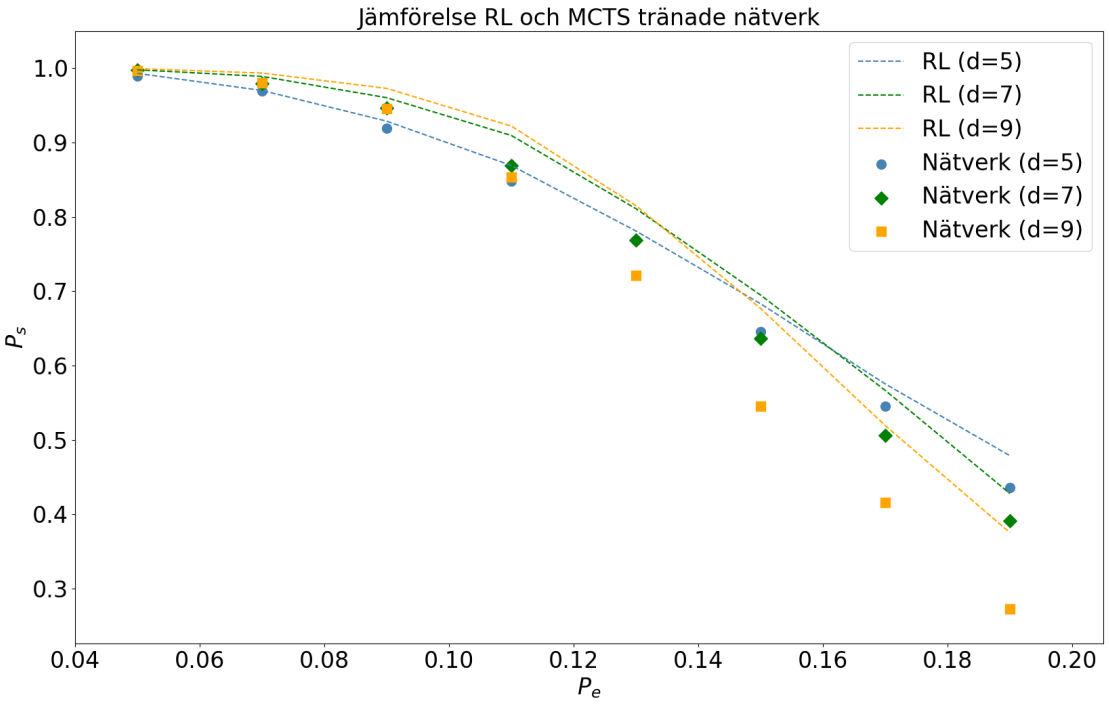
För nätverken som tränades med hjälp av MCTS lyckades inte alltid nätverken eliminera alla defekter eftersom de kunde fastna i loopar där de applicerade samma handling hela tiden. Andelen misslyckade korrigeringar P_F av samtliga misslyckade korrigeringar som berodde på att nätverken inte lyckades ta bort alla defekter presenteras i Figur 9. Varje datapunkt för MCTS motsvarar medelvärdet av 5000 felkorrigeringar.

I Figur 10 presenteras däremot andelen lyckade felkorrigeringar över intervallet $P_e \in [0.05, 0.19]$ i steg om 0.02 för storlekarna d=5, 7 och 9. I Figur 11 illustreras

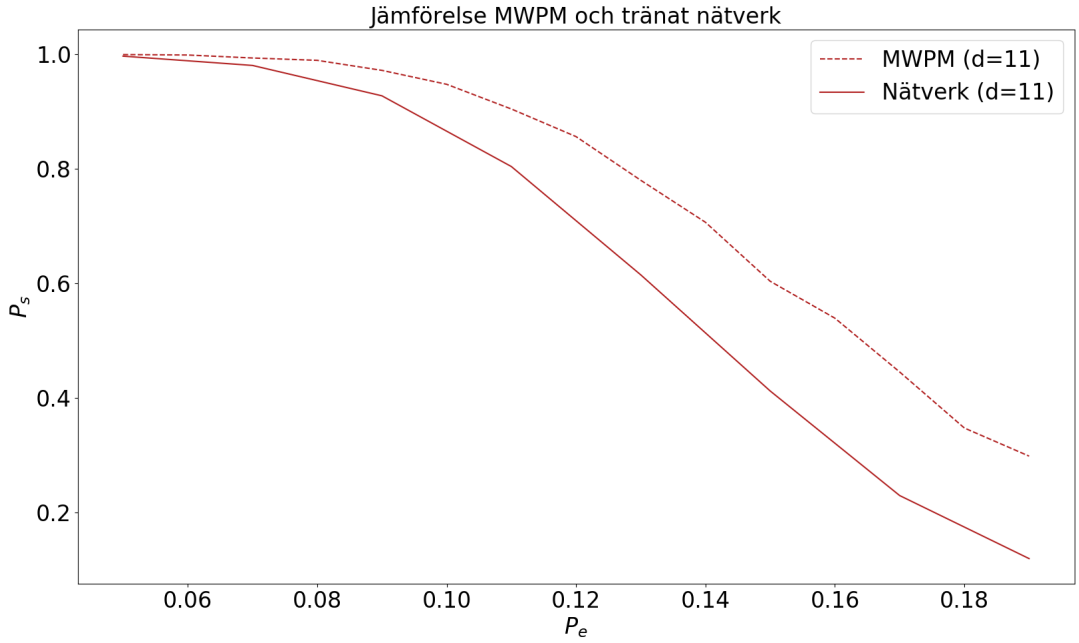
resultatet för agenten som tränades för d=11 samt prestationen för samma storlek för MWPM-algoritmen. Varje datapunkt för MCTS motsvarar medelvärdet av 5000 felkorrigeringar och standardavvikelsen för dessa punkt blev $\sigma < 0.0044$. Datan för MWPM erhölls med ett skript för den toriska koden gjord av David Fizek [5] med en MWPM implementering av Vladimir Kolmogorov [18].



Figur 9: Andelen av alla misslyckade korrigeringar P_F av samtliga misslyckade korrigeringar som berodde på att alla defekter ej lyckats elimineras.

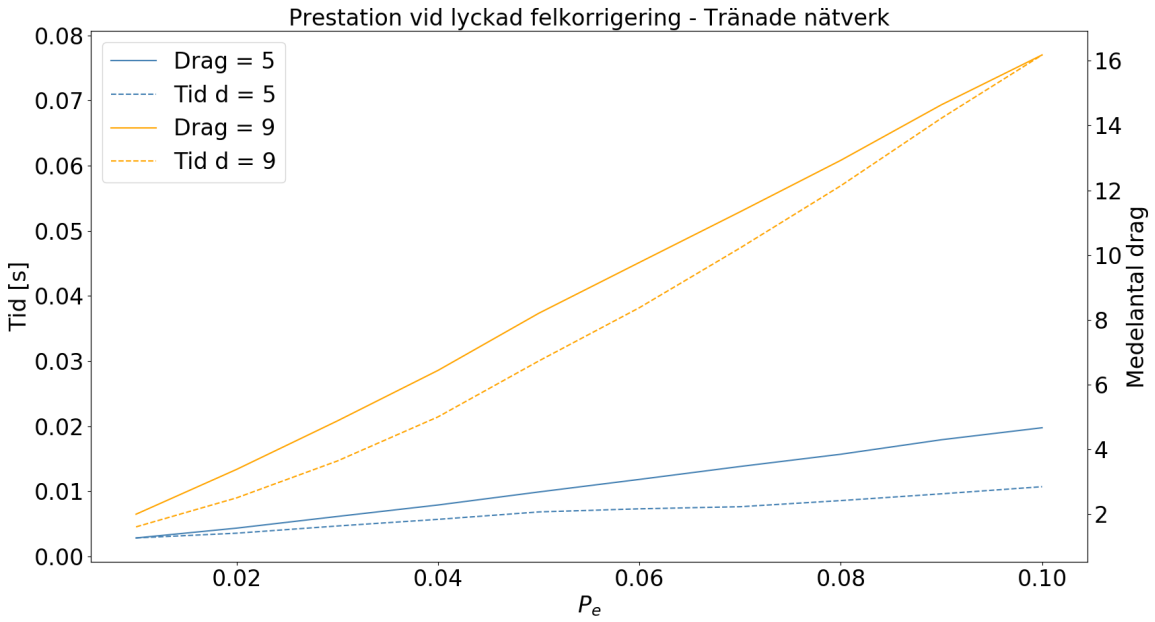


Figur 10: Jämförelse av prestanda, med arbetet av Fitzek et. al för storlekarna d=5 till 9.



Figur 11: Jämförelse av prestanda, med MWPM-algoritmen för storleken d=11. Datan för MWPM erhölls med ett skript för den toriska koden gjord av David Fizek [5] med en MWPM implementering av Vladimir Kolmogorov [18].

I Figur 12 illustreras hur lång tid samt medelantalet drag som krävdes för en lyckad felkorrigering. Medelvärdet togs över 3000 lyckade felkorrigeringar där standardavvikelsen för antalet drag var $\sigma < 0.3$.



Figur 12: Tid och medelantal drag (handlingar) som krävdes för lyckad felkorrigering beroende på P_e för d=5 och 9.

I Appendix B presenteras träningen av agenter för felkorrigeringar på storlekar d=5 till 11.

5 Diskussion

Fördelen med MCTS i kombination med artificiella neurala nätverk demonstrerades väl av Silver et. al när deras program AlphaGo Zero besegrade världsmästaren i spelet Go genom att simulera möjliga spelbräden. Istället för ett spelbräde kan man istället simulera framtida möjliga tillstånd för den logiska kvantbiten med flippade kvantbitar. Utifrån denna information kan därefter felen korrigeras. Syftet bakom detta projekt var att kombinera denna strategi med arbetet från Fitzek et. al och utveckla två typer av avkodare. Den första avkodaren utförde drag baserat på information som en nätverksguidad MCTS gav och den andra varianten baserades på träning av nätverk med hjälp av MCTS.

5.1 Analys av MCTS-baserade avkodaren

Från Figur 7 kan man konstatera att när ett redan färdigtränat nätverk kombineras med en MCTS sker en tydlig förbättring. En MCTS tillsammans med ett nätverk

som i sig redan är välfungerande avkodare resulterar alltså i en bättre fungerande avkodare. Eftersom RL presterade bättre än MWPM-algoritmen för systemstorlekarna i figuren kan slutsatsen dras att MCTS även presterade bättre än MWPM för motsvarande systemstorlekar. Vid en närmare undersökning gällande hur MCTS korrigerade felen visade det sig att denna algoritm oftare lyckades eliminera alla fel X, Y och Z på den toriska koden än RL (se Tabell 3). Detta kan ha varit en bidragande orsak till att MCTS presterade bättre eftersom sannolikheten för en icke-trivial loop minskar om avkodaren oftare lyckas eliminera samtliga fel.

Vidare illustrerar Figur 8 medelantalet drag och den tid som krävdes för lyckad felkorrigering vid en viss sannolikhet för fel. En torisk kod med storlek d bör i snitt besitta $P_e \cdot 2 \cdot d^2$ antal fel, vilket också borde korrelera direkt till medelantalet handlingar som krävs vid felkorrigeringen. Exempelvis bör vid storleken d = 9 i snitt kräva $0,10 \cdot 2 \cdot 9^2 = 16,2$ drag. Detta stämmer väl överens med vad som kan ses grafen i Figur 8. Liknande beräkningar kan göras för övriga storlekar, och resultaten följer graferna i Figur 8 väl. Även de tränade nätverken utförde i snitt de optimala antalen drag för ett visst P_e vilket illustreras i Figur 12. Däremot är tiden det tar för en lyckad felkorrigering cirka 10 gånger längre för MCTS avkodaren jämfört med tiden det tar för ett nätverk vid motsvarande systemstorlek. Tiden det tar för MCTS avkodaren och dess prestation är däremot direkt kopplad till antalet simuleringar som i trädsökningen. Om endast en simulering hade genomförts skulle tiden motsvara den för det rena nätverket men prestationen skulle ligga på samma nivå. Avvägningen mellan tid och prestation måste därför vägas in vid val av antal simuleringar.

5.2 Analys av nätverkens prestanda

För nätverken som tränades med hjälp av MCTS blev resultaten inte lika bra som de för MCTS avkodaren. Prestationen var sämre än nätverken från arbetet av Fitzek et al för d = 5, 7 och 9 (se Figur 10) och sämre än MWPM-algoritmen för d = 11 (se Figur 12). Anledningen till detta var huvudsakligen nätverkens oförmåga att eliminera samtliga defekter då de kunde fastna i loopar, där samma handling återupprepades hela tiden. Figur 9 illustrerar detta tydligt, där andelen av de misslyckade korrigeringarna när avkodaren inte lyckats eliminera alla defekter är mycket hög för $d = 9$ och $d = 11$. Från samma figur kan man även se att om looparna inte hade uppstått och samtliga fel alltid eliminerats, hade nätverken presterat bättre än jämförelsealgoritmerna för samtliga systemstorlekar. De tränade nätverken kan därmot för tillfället inte klassificeras som fungerande avkodare då de till skillnad från MCTS, RL och MWPM-algoritmerna inte alltid klarar av att eliminera felen.

Fördelen med en större systemstorlek är att sannolikheten för lyckad felkorrigering P_e där det inte uppstår en icke-trivial loop blir högre för större d fram till en viss punkt. För en avkodare likt MCTS är storleken d = 7 exempelvis bättre än d = 5 fram till då $P_e \approx 0.17$. Anledningen är att vid denna punkt blir sannolikheten för icke-triviala loopar större i d = 7 än för d = 5. Däremot ligger denna gräns för de icke optimala nätverken mycket tidigare, ungefär kring $P_e \approx 0.13$ för d = 7

och 5. Reflektionen kan nu också göras att MCTS lyckas förskjuta denna punkt när linjerna korsas till högre P_e jämfört med RL.

Nätverken som tränades av MCTS har däremot potential, speciellt om man ser till den kortare tid det tog för nätverket att konvergera till en lösning. Det tog ungefär 96 h för RL-algoritmen att konvergera för d = 7 [5]. För motsvarande systemstorlek kan man se i Figur 14 i Appendix B att detta skedde redan efter cirka 1.6 h för nätverken tränade med hjälp av MCTS. Däremot konvergerade de till lokala maxpunkter som inte lyckades korrigera felen varje gång. Därmed krävs vidare studier för att lösa detta problem.

5.3 Vidare studier

Nätverken tränade med hjälp av MCTS konvergerade relativt snabbt till icke optimala lokala maxpunkter, vilket kan ses Figur 16 - 15 i Appendix B. Detta kan ha flera orsaker. Exempelvis skulle noggrannare undersökning av olika inlärningshastigheter kunna hjälpa. Att minnesbufferten återställdes för ofta kan också ligga till grund, då detta kan ha lett till att nätverken ej hann fokusera tillräckligt mycket på de svåraste situationerna. Ett större intervall av återställning av minnet resulterade dock i att gamla träningsexempel blev kvar för länge och förhindrade inlärningen av de nya. Vidare genomfördes inga större försök att få fram neurala nätverksarkitekturer bättre anpassade till problemet, utan de som testades var direkt tagna från Fitzek et. al, varav den till synes bäst fungerande arkitekturen beskrivs i Tabell 4.

I utvecklingen av MCTS gjordes ingen avvägning mellan tid och prestation och antalet simuleringar för olika systemstorlekar valdes enligt Tabell 1 nästintill utan någon optimering. Därmed hade antalet simuleringar behövts optimeras för olika storlekar d för att få bästa möjliga resultat med avseende på tid och prestation. Det hade dessutom varit möjligt att testa hurvida MWPM-algoritmen skulle kunna implementeras i guidning av MCTS för storlekar d > 9.

En annan potentiell förbättring är att förändra hur uppsamlingen av data från MCTS sker. För tillfället när en trädsökning görs propageras Q-värden upp till den nuvarande noden som sedan sparas i minnesbufferten. Därefter tas ett steg och man förflyttar sig till en ny nod. Varefter de föregående stegen repeteras. Istället för detta kan man efter varje trädsökning propagera ända upp till den ursprungliga rotnoden. Efter att episoden är klar traverserar man trädet och sparar Q-värdes vektorerna för de tillstånd handlingsparen som har besökts nog gånger. Således får man bättre Q-värden för de tillstånd med fler defekter, som bör ge en bättre träning för större storlekar på den toriska koden.

Det direkta belöningssystem som användes likt ekvation 6 reflekterar inte sannolikheten att ett drag skulle resultera i en icke-trivial loop och därmed leda till en misslyckad felkorrigering. En slutlig punkt gällande MCTS avkodaren är därmed att belöningsfunktionen hade kunnat förbättras. Ett alternativ är att basera belöningsfunktion på en sannolikhetsdistribution genererad med exempelvis en Markovkedje-

Monte Carlo (MCMC) metod. Belöningen som erhålls under trädsökningen skulle då bättre representera hur bra draget var i förhållande till dess sannolikhet att generera icke-triviala loopar. Detta skulle troligen leda till en klar förbättring i prestanda.

Kvantdatorer har potentialen att knäcka dagens krypteringsalgoritmer som baseras på primtalsfaktorisering men även kraftigt förhöja säkerheten i överföring av information. På grund av detta bör vissa etiska frågor rörande informationsöverföring och säkerhet höjas. Däremot väger de positiva möjligheterna för kvantdatorn i vårt samhälle tyngre än de negativa eftersom dess beräkningskapacitet även öppnar upp för ny forskning och utveckling. Detta projekt där vi har tagit fram två metoder för felkorrigering av kvantfel på en torisk kod syftade till att bidra till detta forskningsområde. Att kunna eliminera fel på den toriska koden är en mycket liten del av kvantdatorn och har därmed inga direkta etiska aspekter att ta hänsyn till. Däremot bör helhetsbilden och konsekvenser av eventuella appliceringsområden av den slutliga kvantdatorn hållas i åtanke.

6 Slutsats

Syftet bakom detta projekt var att kombinera strategin bakom AlphaGo Zero-algoritmen med arbetet av Fitzek et. al och utveckla två typer av avkodare. Den första typen var neurala nätverk som tränades med hjälp av MCTS men kunde i slutändan inte klassificeras som en fungerande avkodare. Anledningen var att dessa inte alltid lyckades eliminera samtliga defekter från syndromen. Däremot är potentialen stor eftersom träningen av nätverken konvergerade snabbt. Om problemet med att nätverken fastnar i loopar löses skulle dessutom dessa prestera bättre än både RL och MWPM. Detta skulle innebära att nätverk som fungerar som avkodare även för $d > 9$ skulle gå snabbt och smidigt att träna. Avkodaren som baserades på den nätverksguidade MCTS-algoritmen fungerade däremot mycket bra och presterade bättre än både RL och MWPM-algoritmen. Tiden det tog för en lyckad felkorrigering var däremot lite längre än för nätverken men potential för förbättring finns i parameteroptimering, alternativt belöningssystem och guidning med hjälp av MWPM-algoritmen för $d > 9$.

7 Referenser

[1] P. Benioff, "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines," *Journal of Statistical Physics*, vol. 22, pp. 563–591, 05 1980.

[2] R. Phillips Feynman, "Simulating Physics with Computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 06 1982.

[3] E. Grumbling and M. Horowitz, *Quantum Computing: Progress and Prospects*. Washtington, DC: The National Academies Press, 2018, ch. 2, pp. 35–47. [Online]. Available: http://cs.brown.edu/courses/csci1800/sources/2018_NAE_QuantumComputing_ProgressAndProspects.pdf

[4] S. J. Devitt, W. J. Munro, and K. Nemoto, "Quantum error correction for beginners," *Reports on Progress in Physics*, vol. 76, p. 2, 06 2013. [Online]. Available: https://iopscience.iop.org/article/10.1088/0034-4885/76/7/076001

[5] D. Fitzek, M. Eliasson, A. Frisk Kockum, and M. Granath, "Deep Q-learning decoder for depolarizing noise on the toric code," 12 2019. [Online]. Available: https://arxiv.org/pdf/1912.12919.pdf

[6] P. Andreasson, J. Johansson, S. Liljestrand, and M. Granath, "Quantum error correction for the toric code using deep reinforcement learning," *Quantum*, vol. 3, p. 183, 09 2019. [Online]. Available: https://doi.org/10.22331/q-2019-09-02-183

[7] D. Silver, J. Schrittwieser, and K. Simonyan, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 10 2017. [Online]. Available: https://doi.org/10.1038/nature24270

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., ser. Adaptive computation and machine learning series | Includes bibliographical references and index. Mountain View, CA 94042, USA: Westchester Publishing Services, 2018, ch. 1, 3, 6, pp. 1–4, 58–67, 131–134.

[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010, vol. 2. [Online]. Available: https://www.cambridge.org/core/books/quantum-computation-and-quantum-information/01E10196D0A682A6AEFFEA52D53BE9AE

[10] D. Browne, "Lectures on topological codes and quantum computation," May 2014. [Online]. Available: https://sites.google.com/site/danbrowneucl/teaching/lectures-on-topological-codes-and-quantum-computation

[11] A. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, no. 1, p. 2–30, Jan 2003. [Online]. Available: http://dx.doi.org/10.1016/S0003-4916(02)00018-0

[12] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, p. 449–467, 1965.

[13] A. G. Fowler, "Minimum weight perfect matching of fault-tolerant topological quantum error correction in average o(1) parallel time," *Quantum Info. Comput.*, vol. 15, no. 1–2, p. 145–158, Jan. 2015.

[14] S. Bravyi, M. Suchara, and A. Vargo, "Efficient algorithms for maximum likelihood decoding in the surface code," *Phys. Rev. A*, vol. 90, Sep 2014. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.90.032326

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[16] S. Haykin, *Neural Networks and Learning Machines*, 3rd ed. Upper Saddle River, New Jersey 07458: Pearson Education, Inc, 2009, ch. 4, pp. 201–202.

[17] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016. [Online]. Available: https://arxiv.org/abs/1511.05952

[18] V. Kolmogorov, "Blossom v: a new implementation of a minimum cost perfect matching algorithm," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 43–67, Jul 2009. [Online]. Available: https://doi.org/10.1007/s12532-009-0002-8

A Nätverksarkitektur

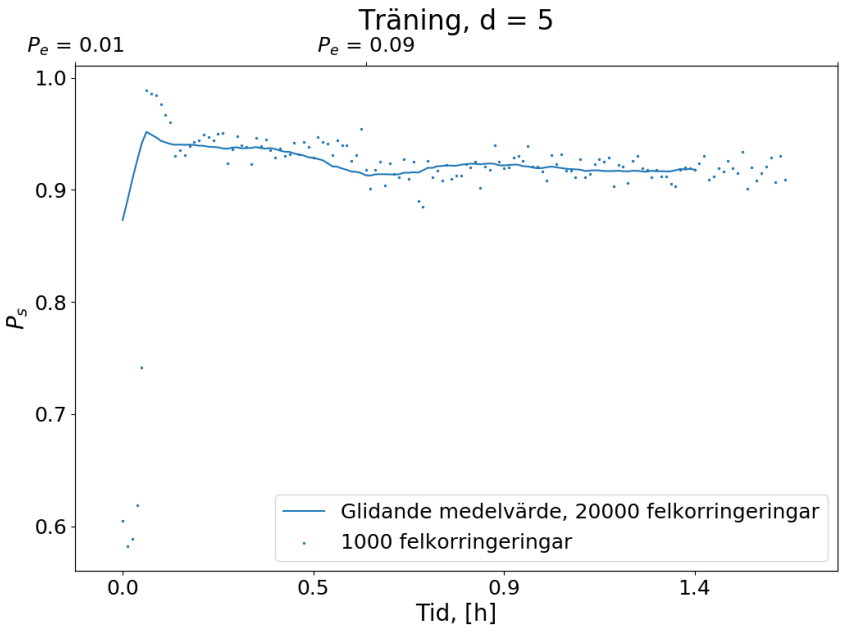
I Tabell 4 presenteras den neurala nätverks-arkitekturen som användes i kombination med MCTS, skapad av Fitzek et. al [5]. Varje faltningslager har filterstorleken 3 och sveper med stegstorleken 1. Periodisk utfyllnad tillämpas på det första lagret och utfyllning med nollor tillämpas på de resterande.

Tabell 4: Nätverksariktekturen som användes i kombination med MCTS för att träna agenter av samtliga storlekar, d = 5, 7, 9 och 11. Typ storlek och antalet justerbara parametrar för lager #.

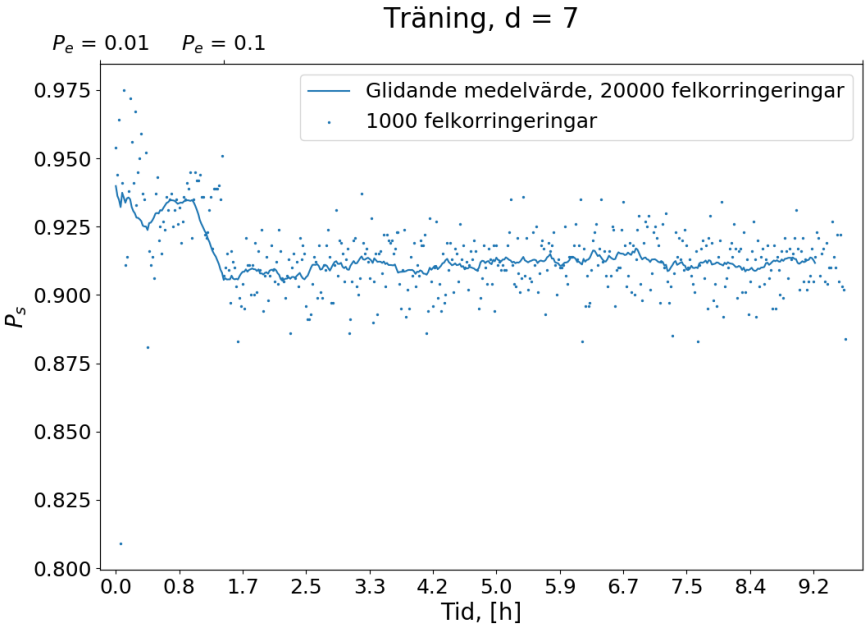
#	Typ	Storlek	Parametrar
1	Conv2d	128	2,432
2	Conv2d	128	147,584
3	Conv2d	120	138,360
4	Conv2d	111	119,991
5	Conv2d	104	104,000
6	Conv2d	103	96,511
7	Conv2d	90	83,520
8	Conv2d	80	64,880
9	Conv2d	73	52,633
10	Conv2d	71	46,718
11	Conv2d	64	40,960
12	Linear	3	1,731
			899,320

B Träning av nätverken

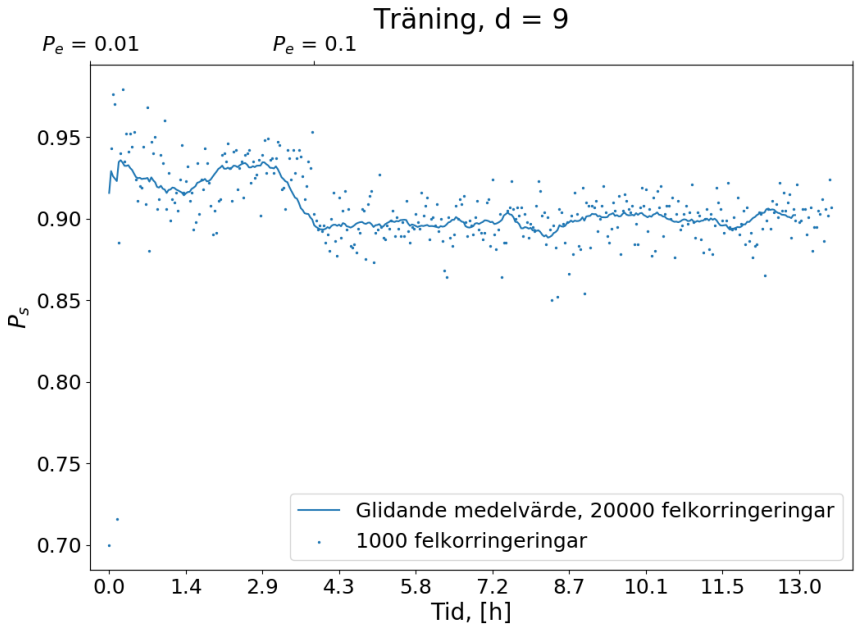
Figur 16 till 15 visar träningen av agenter för felkorrigering på storleken d = 11 på den toriska koden. En datapunkt visar resultatet av de 1000 felkorringeringar som gjordes efter varje träningsepok. Sannolikheten för fel, P_e, ökades stegvis som beskrivits i metodavsnittet och på den övre x-axeln visas det initiala samt slutliga värdet hos P_e.



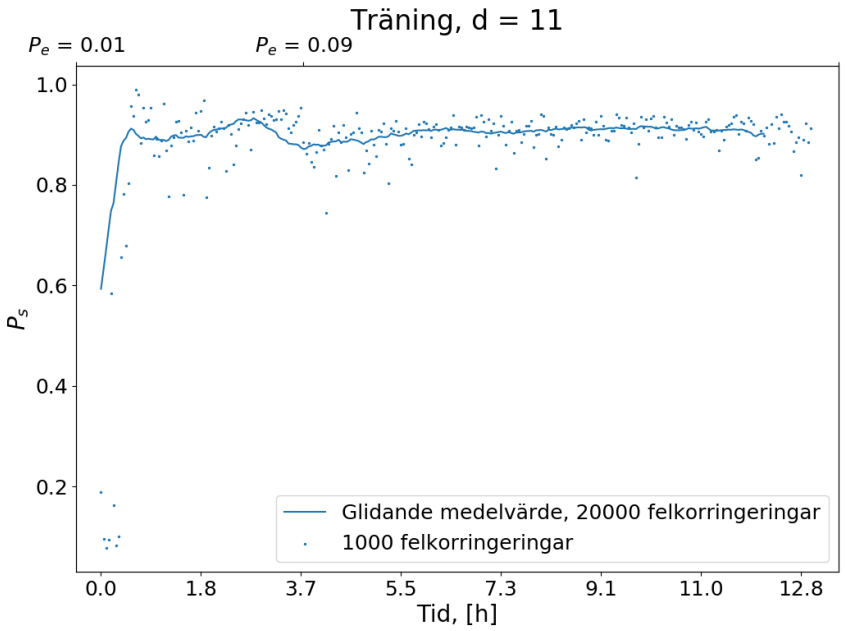
Figur 13: Träning av nätverk för storleken d=5. Datapunkter representerar 1000 felkorrigeringar som gjordes vid varje träningsepok. Linjen är ett glidande medelvärde över 20 datapunkter..



Figur 14: Träning av nätverk för storleken d=7.



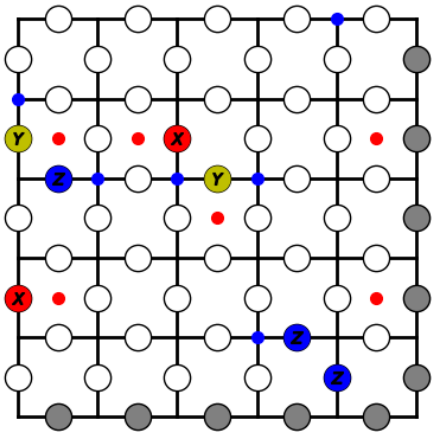
Figur 15: Träning av nätverk för storleken d=9.

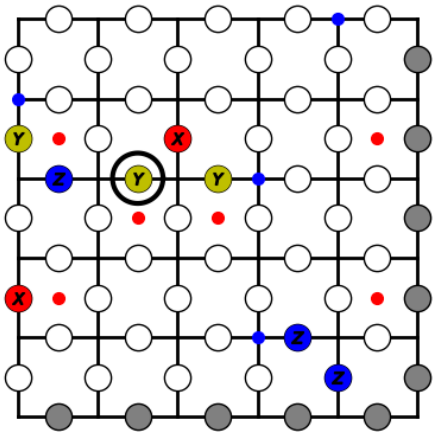


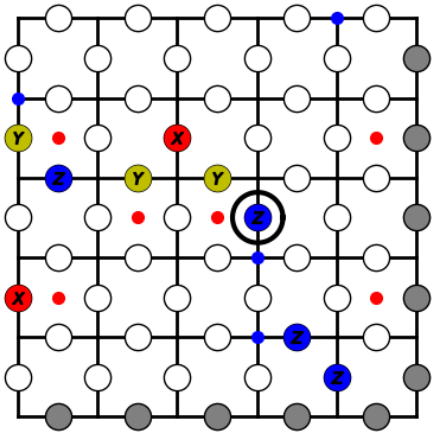
Figur 16: Träning av nätverk för storleken d=11.

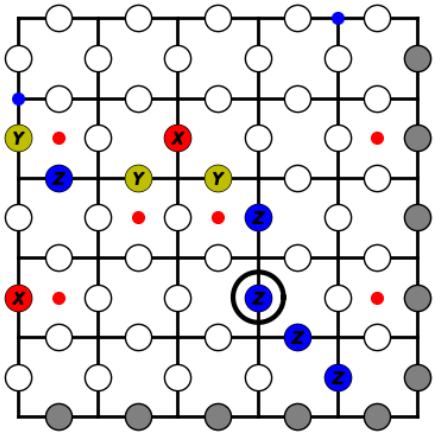
C Utvalda episoder

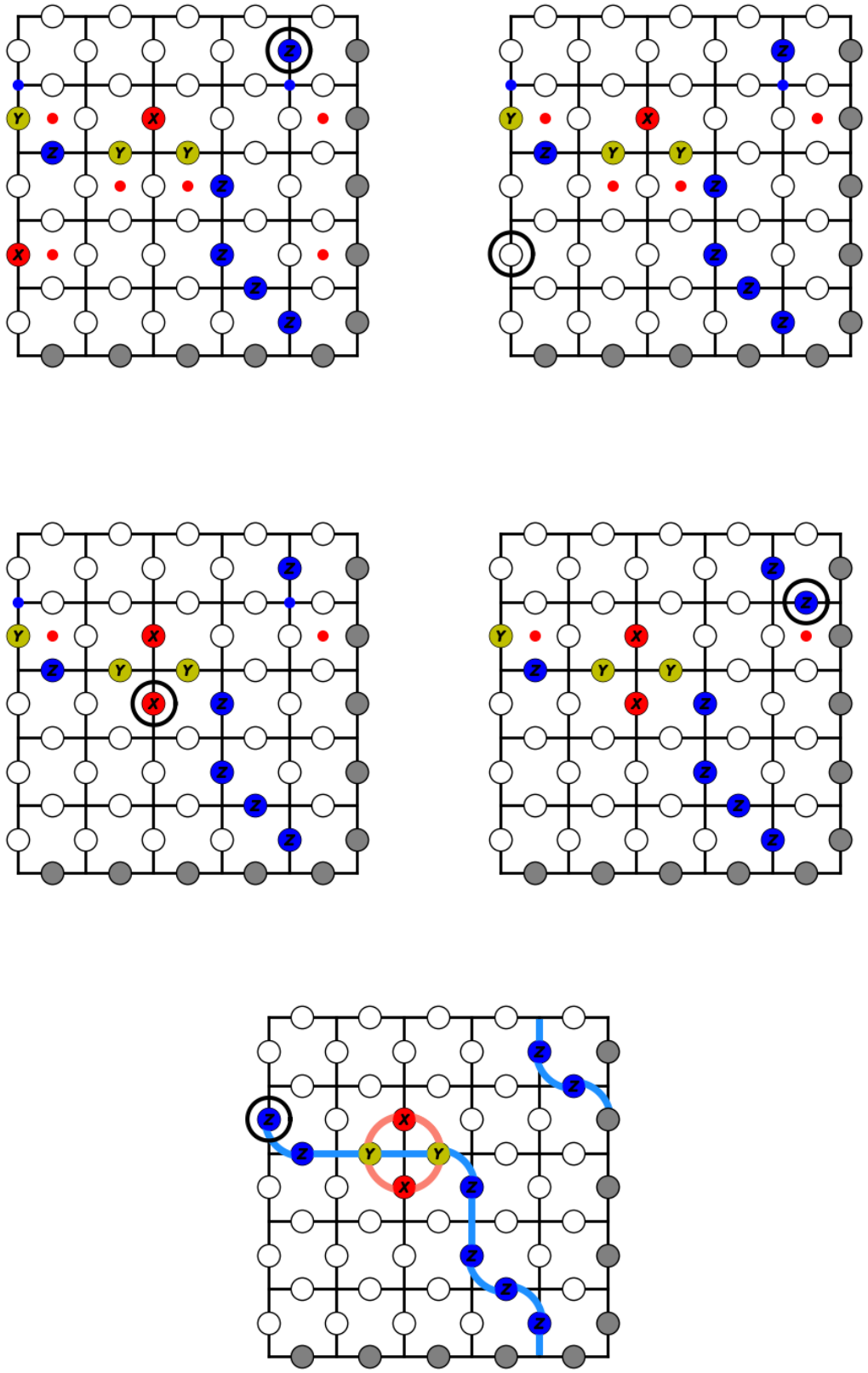
I detta avsnitt presenteras två utvalda episoder där en tränad agent för d = 5 används som avkodare. Först illustreras en misslyckad felkorrigering där det uppstår en icke-trivial loop. Därefter följer en lyckad felkorrigering där två triviala loopar uppstår.



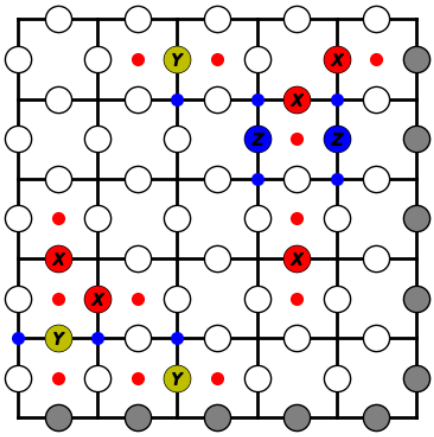


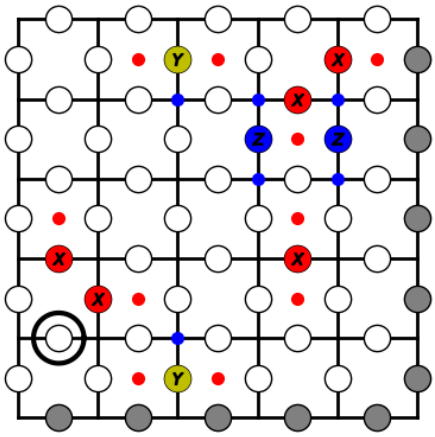


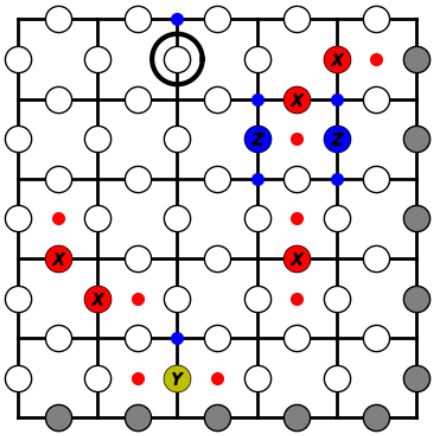


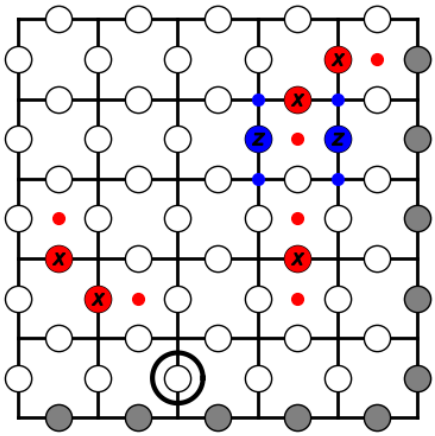


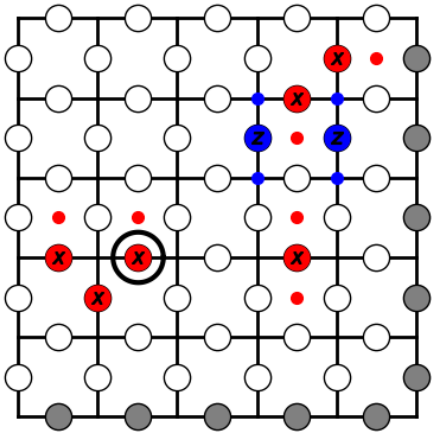
Figur 17: Misslyckad felkorrigering där en icke-trivial loop (blå linje) uppstår. I sista steget verkar en X-operator som ger XY∼Z.

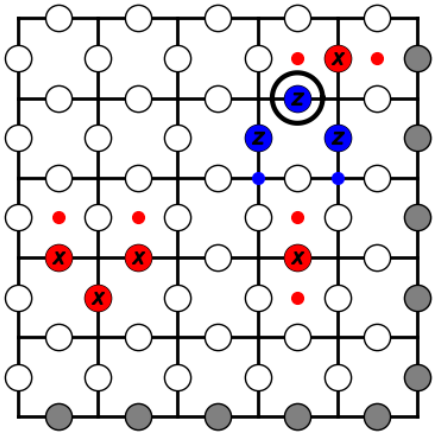


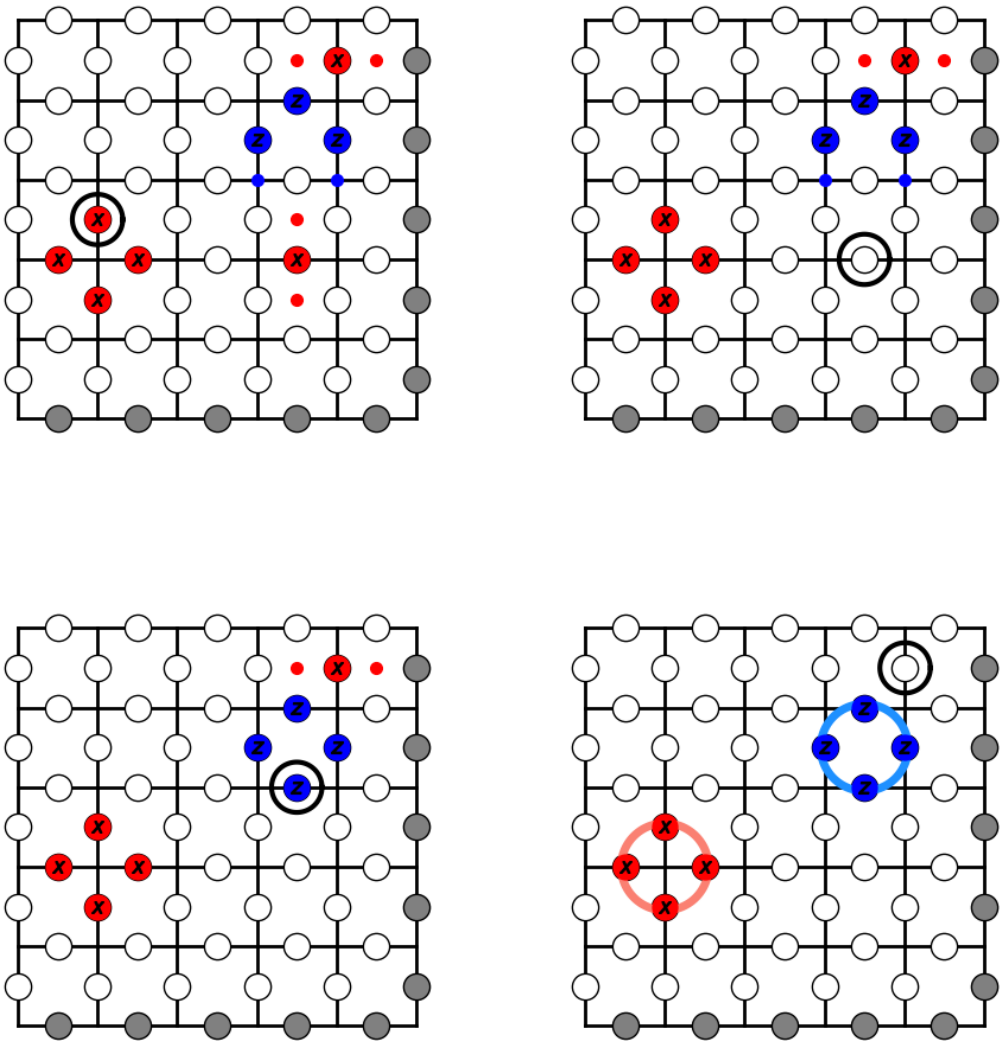












Figur 18: Lyckad felkorrigering där endast två triviala loopar kvarstår.