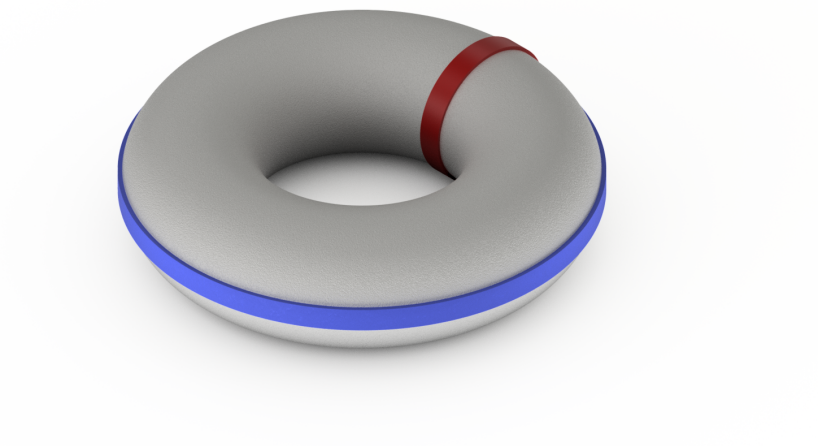
CHALMERS



# Neurala nätverk inom kvantmekanisk felkorrektion

En undersökning av neurala nätverks potential att med hjälp av beteendestyrt lärande genomföra kvantmekanisk felkorrektion på system beskrivna av toric code-modellen

Kandidatarbete vid institutionen för fysik - TIFX04-18-30


Philip Andreasson
phiandr@student.chalmers.se

Niklas Forsström
nikfor@student.chalmers.se

Elias Hölén Hannouch
hannouch@student.chalmers.se

Joel Johansson
gusjohjodt@student.gu.se

Gustav Karlsson
guskarls@student.chalmers.se

Simon Liljestrand
simlilj@student.chalmers.se

Handledare: Mats Granath
Institutionen för fysik
Chalmers tekniska högskola
Göteborg, Sverige 2018

## Sammandrag

Syftet med det här projektet är att undersöka hur väl artificiella neurala nätverk kan tillämpas inom kvantmekanisk felkorrektion i toric code-modellen, vilket är en kvantmekanisk felkorrigerande kod beskriven av ett gitter med spinn. Gittret har periodiska randvillkor, och vid minimerad energi finns det fyra möjliga grundtillstånd. Energin hos systemet kan öka om slumpmässiga spinn flippas i gittret och då uppstår så kallade fel i koden. När koden används inom kvantmekanisk informationslagring är det av intresse att korrigera dessa fel på ett sätt som låter systemet återgå till samma grundtillstånd som före feluppkomsten. Det är denna process som kallas kvantmekanisk felkorrektion. I detta projekt implementeras beteendestyrt lärande (eng. reinforcement learning) i algoritmer som använder artificiella neurala nätverk för att utföra felkorrektion. Konvolutionella samt fullt kopplade nätverk implementeras tillsammans med SARSA-algoritmen (State-Action-Reward-State-Action). Det konvolutionella nätverket uppvisar bättre prestanda och kortare inlärningstid än det fullt kopplade nätverket. Nätverket återför systemet till rätt grundtillstånd i 94 % av fallen för ett gitter innehållande $7 \times 7$ plaketter, där 5 % av gitterpunkterna är utsatta för spinnflipp.

**Nyckelord** neurala nätverk, maskininlärning, toric code, beteendestyrt lärande, kvantmekanisk felkorrektion

## Abstract

This project aims to explore the application of artificial neural networks in quantum error correction and more specifically in the toric code model, which is a quantum error correcting code represented by a spin lattice. The lattice has periodic boundary conditions and it has four possible ground states at minimum energy. The energy of the system may be increased by randomly occurring spin flips in the lattice which gives rise to so called errors in the code. When the code is used for quantum information storage, it is desirable to cancel these errors in such a way that the ground state of the system is preserved. This process is referred to as quantum error correction. In this project, reinforcement learning is implemented in algorithms that use artificial neural networks for the purpose of error correction. Convolutional and fully connected neural networks are implemented in combination with the SARSA (State-Action-Reward-State-Action) algorithm. The convolutional network performs better and is trained faster than the fully connected network. It returns the system to the correct ground state in 94 % of test cases for a lattice containing $7 \times 7$ plaquettes, where 5 % of spins are flipped.

**Keywords** neural networks, machine learning, toric code, reinforcement learning, quantum error correction

# Innehåll

# 1

# Inledning

Kvantmekanisk informationslagring är ett område som skiljer sig från klassisk informationslagring genom att kvantmekaniska effekter i form av linjärkombinerade tillstånd och sammanflätning utnyttjas [1]. En dator som bygger på kvantmekanisk informationslagring, en så kallad kvantdator, skulle kunna lösa vissa beräkningsintensiva problem effektivare än en klassisk dator [2]. För att kvantdatorer ska kunna konstrueras och användas krävs dock metoder för att skydda lagrad information genom att korrigera fel som slumpmässigt kan uppstå i ett kvantmekaniskt system. En vanlig metod är att lagra informationen i så kallade felkorrigerande koder, varav ett viktigt exempel är toric code [3].

Maskininlärning är ett område inom datavetenskap där smarta algoritmer används för att lösa komplexa problem i fall där analytiska lösningar är svårformulerade. Effektivisering av de bakomliggande algoritmerna och kraftfullare datorer har gjort maskininlärning till ett problemlösningsverktyg i domäner där det tidigare ansågs omöjligt på grund av de stora datamängder som behöver hanteras [4]. Framgångarna har lett till ett stort intresse för att utforska nya tillämpningsområden för maskininlärning, exempelvis inom fysik [5]. Ett verktyg som vanligen används inom maskininlärning idag är artificiella neurala nätverk, vilket är en typ av funktionsapproximator som fått sitt namn av att verktyget tagit inspiration från processer i den biologiska hjärnan [6]. Fördelen med artificiella neurala nätverk är att de kan approximera ett vitt spann av funktioner, även sådana som är icke-linjära och använder sig av brusig och varierad input [7].

En metod för maskininlärning som ofta används för beslutsprocesser är beteendestyrt lärande (eng. reinforcement learning) [8]. Fördelen med metoden är att man inte på förhand behöver ha någon information om vilka beslut som är optimala, vilket är fallet i många verkliga processer. Ett exempel på ett problem där beteendestyrt lärande har applicerats framgångsrikt är brädspelet go, som på grund av sin komplexitet gör det svårt att utvärdera alla tänkbara drag [9].

Det är inte uppenbart hur kvantmekanisk felkorrektion i toric code ska genomföras på optimalt sätt, vilket gör det intressant att försöka använda maskininlärningsmetoder. Tidigare har ett neuralt nätverk, tränat med oövervakat lärande (eng. unsupervised learning), lyckats uppnå resultat jämförbara med de klassiska algoritmer för felkorrektion

som finns [10]. Felkorrektionen kan även formuleras som en beslutsprocess och det skulle därför vara av intresse att applicera beteendestyrt lärande på problemet.

## 1.1 Syfte och avgränsningar

Detta projekt syftar till att undersöka hur väl neurala nätverk med hjälp av beteendestyrt lärande kan implementeras för att korrigera fel på ett sätt som bevarar grundtillståndet i toric code. Vidare syftar projektet till att undersöka hur väl olika nätverksstrukturer presterar samt hur nätverken på bästa sätt tränas för att uppnå maximal prestanda.

På grund av att de tillgängliga beräkningsresurserna är begränsade till persondatorer utan möjlighet att utnyttja grafikprocessorer måste en gräns sättas på hur stora modeller som kan behandlas. För att träningstiden inte ska överskrida en vecka begränsas storleken av modellerna till att innehålla $11 \times 11$ plaketter. Givet tillgång till kraftfullare beräkningsresurser kan simuleringarna i framtida studier med fördel skalas upp till att omfatta större matriser för att få en bättre förståelse för hur storleken på koden påverkar algoritmens prestanda.

Det finns flera olika typer av fel som kan uppkomma i en toric code, men dessa kan med fördel studeras oberoende av varandra [11]. Därför betraktas endast så kallade kvantmekaniska bitflippfel i detta projekt.

## 1.2 Etiska aspekter

Maskininlärning kan inom en snar framtid komma att bli en ovärderlig stöttepelare inom IT-världen. Precis som för andra tekniker finns även möjligheten att maskininlärning kan användas i applikationer som motstrider våra etiska ideal. Exempelvis är krigföring ett område där maskininlärning kan föra med sig allvarliga konsekvenser. För att förhindra att denna typ av utveckling blir verklighet kommer det att krävas globala insatser och samarbeten för att bygga upp ett regelverk kring användandet av maskininlärning, neurala nätverk och artificiell intelligens. Historien har många gånger visat vad som händer när användandet av nya tekniker inte regleras från första början. Ett talande exempel är kärnvapen, vars konsekvenser lever kvar i våra samhällen än idag. Om en artificiell intelligens ska ha en plats i detta samhälle är det därför viktigt att ha en tydlig plan för vilken roll tekniken ska tjäna, samt vilka åtgärder som kommer att behöva vidtas för att utveckla tekniken i den önskade riktningen. Vi är av åsikten att fördelarna som maskininlärningsmetoder kan bidra med till samhället övervinner de nackdelar som finns och gör det etiskt försvarbart att bedriva forskning inom området. Ansvaret för att denna forskning utförs på ett hållbart sätt är en politisk fråga och är därför inget som ska beslutas av enskilda aktörer.

Då toric code-modellen kan komma att visa sig ha en viktig roll i framtida kvantdatorer är även etiska aspekter gällande dessa viktiga att beakta. En framgångsrik realisering av kvantdatorer skulle kunna lösa många problem inom forskningsfält där beräkningsresurser utgör en begränsning i dagsläget. Till de etiska nackdelarna hör bland annat kvantdatorers effekter inom IT-säkerhet. Krypteringsmetoderna som i dagsläget används för att skydda information på nätet förlitar sig på att vanliga datorer inte kan primtalsfaktorisera stora tal [12]. Kvantdatorer skulle däremot inte ha några som helst problem att primtalsfaktorisera dessa tal. Effekten av kvantdatorernas introducering skulle därför bli att all kryptering för känsliga dokument skulle bli föråldrad. Det är därför av stor vikt att det sker utveckling av nya krypteringsmetoder parallellt med utvecklingen av kvantdatorer [13]. Precis som i fallet med maskininlärning så bör beslut för hur tekniken ska utvecklas fattas på en högre nivå.

# 2

# Teori

Detta kapitel presenterar den teori varpå projektet bygger. Först behandlas toric code-modellen och därefter hur uppkomna fel i en toric code kan korrigeras. Grunderna för artificiella neurala nätverks arkitektur presenteras, följt av ett avsnitt om träningsmetoder som används för att optimera maskininlärningsprogram för olika applikationer.

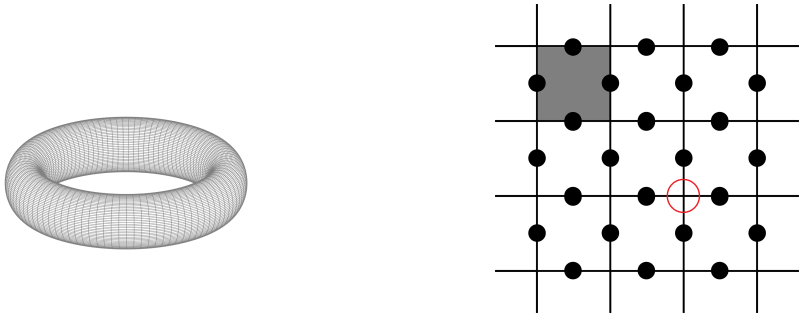
## 2.1 Toric code-modellen

Inom klassisk informationslagring används en bit som den minsta informationsenheten, som kan anta ett av två logiska värden [11]. Kvantmekanisk informationslagring bygger, till skillnad från klassisk informationslagring, på att information lagras som ett superponerat tillstånd hos ett kvantsystem [14]. Den minsta informationsenheten kallas i det här fallet en kvantbit. Vid både klassisk och kvantmekanisk informationslagring finns ett behov av att skydda lagrad information mot fysikaliska fel som uppkommer i praktiken [15].

Det är viktigt att skilja mellan logiska kvantbitar och fysikaliska kvantbitar, där de första utgör de minsta faktiska informationsenheterna, medan de senare är fysikaliskt implementerade kvantbitar, exempelvis en elektrons spinn [11]. Skydd vid klassisk informationslagring bygger ofta på att kopior av samma information lagras parallellt, för att möjliggöra identifiering och korrigering av fel. Denna metod kan dock inte användas vid kvantmekanisk informationslagring, då en identisk kopia inte kan skapas av ett superponerat kvanttillstånd utan att kollapsa det ursprungliga tillståndet [16]. En lösning som fungerar i det kvantmekaniska fallet är att koda informationen som lagras i en logisk kvantbit i termer av flera fysikaliska kvantbitar, vilket först gjordes av Peter Shor [17]. Dessa konstruktioner kallas för kvantmekaniskt felkorrigerande koder och ett exempel på en sådan kod är toric code [3], vilket utgör fokus för detta projekt.

En toric code defineras på ett tvådimensionellt gitter med periodiska randvillkor som gör att den geometriskt antar formen av en torus, se figur 2.1a [3]. Gitterpunkterna ligger ordnade så att varje gitterpunkt kan ses som kanten på en kvadratisk plakett, se figur 2.1b. Hörnet på en sådan plakett kommer vidare att benämnas som ett vertex. Varje gitterpunkt motsvarar en fysikalisk kvantbit vilket i detta projekt motsvaras av ett spinn. Spinnet

befinner sig i ett tvådimensionellt tillståndsrum med bastillstånd $|0\rangle$ som motsvarar spinn ner och $|1\rangle$ som motsvarar spinn upp.



(a) (b)

Figur 2.1: ***a)*** *På grund av de periodiska randvillkoren i toric code tar gittret formen av en torus.* ***b)*** *Tvådimensionell gitterrepresentation av en toric code. Gitterpunkterna är markerade med en cirkel och motsvarar en fysikalisk kvantbit med ett spinn, som syns ligger ordnade i ett kvadratiskt gitter. Det mörklagda området visar en plakett och det inringande området visar ett vertex.*

### 2.1.1 Operatorer för toric code

Det finns olika typer av fel som kan uppkomma på en enskild fysikalisk kvantbit i koden [11]. Det här projektet behandlar enbart den typ av fel som uppkommer under inverkan av en Pauli X-operator, så kallade bitflippfel. Pauli X-operatorn, som är en linjär operator, kan på matrisform skrivas som

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Denna verkar på ett bastillstånd genom att byta riktning på spinnet, enligt

$$X|0\rangle = |1\rangle, X|1\rangle = |0\rangle. \tag{2.1}$$

Toric code är ett exempel på en stabilisatorkod, vilket är en speciell typ av kvantmekaniskt felkorrigerande kod [18]. För en stabilisatorkod definieras stabilisatorgeneratorer, vilka är operatorer som kan verka på koden utan att kollapsa det kvantmekaniska tillståndet [11]. För toric code-modellen finns det två olika typer av stabilisatorgeneratorer: vertexoperatorer och plakettoperatorer. Varje vertexoperator har ett associerat vertex och verkar med Pauli X-operatorer på varje spinn som angränsar till detta. Plakettoperatorn verkar på en plakett och är relaterad till systemets energi, vilket diskuteras i avsnitt 2.1.2.

I ett kvadratiskt gitter med sidlängd $L$, det vill säga $L$ plaketter i horisontellt respektive vertikalt led, finns det $2L^2$ fysikaliska kvantbitar [11]. I samma gitter finns $L^2$ plaketter och $L^2$ vertex. Det är möjligt att införa $L^2 - 1$ oberoende plakettoperatorer och $L^2 - 1$

oberoende vertexoperatorer, vilket gör att antalet logiska kvantbitar $k$ som kan kodas i toric code-modellen är

$$k = 2L^2 - (L^2 - 1) - (L^2 - 1) = 2, \tag{2.2}$$

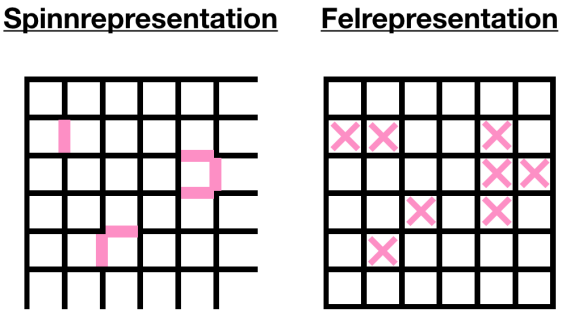
där varje logisk kvantbit kan anta ett av två värden [11]. De två logiska kvantbitarna kan därför tillsammans anta fyra olika kombinationer av värden. Varje kombination av värden för de logiska kvantbitarna motsvaras av ett topologiskt grundtillstånd och det finns således $2^k = 4$ topologiska grundtillstånd. På det här sättet kopplas alltså grundtillståndet hos ett system av fysikaliska kvantbitar till värden på logiska kvantbitar.

### 2.1.2 Energi och grundtillstånd i toric code

Ett spinn kan representeras av ett tal $\sigma$, som antar värde 1 för spinn upp, respektive -1 för spinn ner. Energin för en plakett i gittret är proportionell mot produkten av alla spinn $\sigma_i$ på plakettens rand [3],

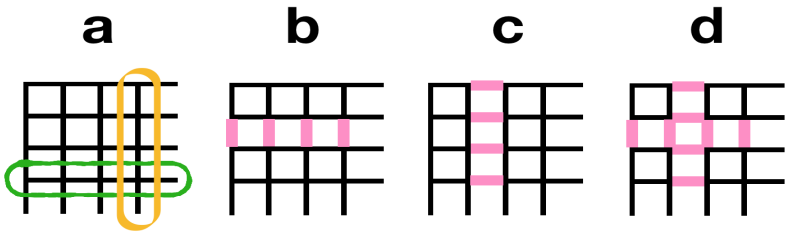
$$E \propto -\prod \sigma_i. \tag{2.3}$$

I ett grundtillstånd är energin alltid minimerad, vilket inträffar om antalet spinn med negativt tecken i varje plakett är jämnt. En plakett med udda antal negativa spinn sägs därför innehålla ett fel [11], se figur 2.2. Tecknet på $\sigma$ för ett individuellt spinn kan inte mätas direkt utan att kollapsa dess kvanttillstånd [20]. Däremot kan energin hos en plakett mätas genom att applicera plakettoperatorn, eftersom det är en stabiliseringsoperator. Felrepresentationen i figur 2.2 kan därmed alltid fås genom mätning, till skillnad från spinnrepresentationen. Notera att grundtillståndets energi alltid bevaras under inverkan av vertexoperatorn eftersom den flippar två spinn i de plaketter som den påverkar.




Figur 2.2: *Figuren till vänster visar ett system som har utsatts för bitflippfel på de markerade plakettkanterna. I figuren till höger är de plaketter som sägs innehålla fel markerade med ett kryss. Den vänstra figuren kommer i fortsättningen att kallas för spinnrepresentationen och den högra figuren kommer att kallas för felrepresentationen. Notera att randen på spinnrepresentationen saknas för att ta hänsyn till gittrets periodiska randvillkor.*

Produkten av alla $\sigma$ längs en rad av gitterpunkter som korresponderar mot horisontella plakettkanter, eller en kolumn av gitterpunkter som korresponderar mot vertikala plakettkanter är antingen 1 eller -1. Mellan olika grundtillstånd skiljer sig minst en av dessa produkter [20], vilket visas i figur 2.3. Om tecknet hos produkten av de $\sigma$ som ligger inom någon av markeringarna i bild 2.3 **a** skiljer sig mellan två tillstånd, befinner de sig i olika grundtillstånd. Varje grundtillstånd motsvaras alltså av en kombination av produkttecken. Relativt ett grundtillstånd nås de övriga tre genom att flippa de markerade spinnen i någon av nedanstående delfigurer. Flippning av spinnen i figur 2.3 **b** ändrar tecken hos den vertikala produkten i figur 2.3 **a**, definierad som produkten av de vertikala spinnen i det ljusgula området. På samma sätt ändrar flippning av spinnen i figur 2.3 **c** respektive figur 2.3 **d** tecknet hos den horisontella produkten respektive båda produkterna. Den horisontella produkten definieras som produkten av de horisontella spinnen i det mörkgröna området. Under verkan av vertexoperatorn bevaras produkterna och därmed grundtillståndet, vilket även gäller för plakettoperatorn.

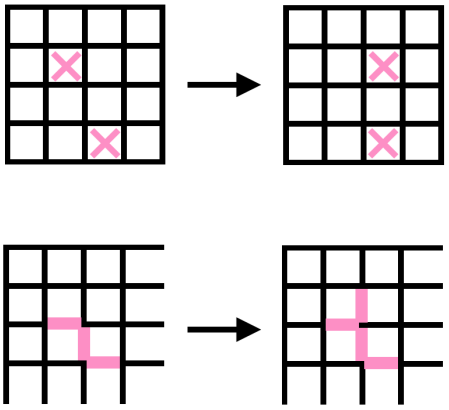



Figur 2.3: *De fyra toric code-grundtillstånden. Genom att beräkna produkterna av de spinn som ligger i de inringade sektionerna i figur* ***a*** *ges vilket grundtillstånd systemet befinner sig i. För figur* ***a*** *blir båda produkterna 1. I figur* ***b*** *blir den horisontella produkten 1 och den vertikala -1. I figur* ***c*** *blir tvärtom den horisontella produkten -1 och den vertikala 1. I figur* ***d*** *blir båda produkterna -1.*

Från figur 2.3 går det alltså att få en förståelse för vad som skiljer grundtillstånden åt, men i praktiken kan inte motsvarande information fås genom mätning utan att kollapsa det kvantmekaniska tillståndet [11].

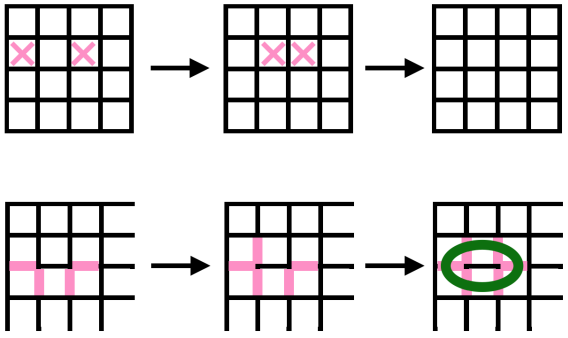
## 2.2 Kvantmekanisk felkorrektion

Tidigare motiverades behovet av felkorrektion i toric code och de plaketter som sägs innehålla fel definierades. I det här avsnittet behandlas hur felkorrektion genomförs. Figur 2.4 visar felrepresentationen och spinnrepresentationen för ett tillstånd. Det visas i figuren att om ett spinn på den högra sidan av en plakett med ett fel flippas, så kommer även felet i felrepresentationen att röra sig åt höger [11]. Felet kan genom spinnflipp på motsvarande sätt flyttas uppåt, till vänster eller nedåt.



Figur 2.4: *Figuren illusterar vad som händer om ett spinn som angränsar till en felplakett flippas. Notera att det övre felet rör sig över plakettgränsen för det spinn som flippas.*

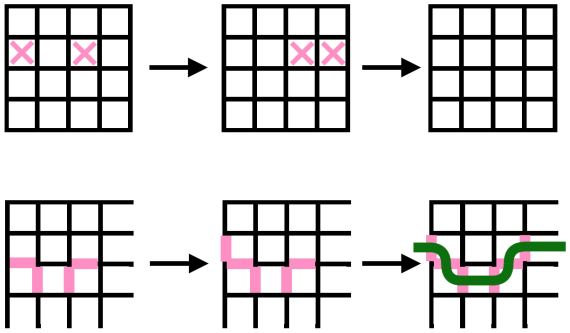
Energin för det fullständiga tillståndet minimeras om alla plaketter har minimal energi, vilket kan åstadkommas genom att para ihop fel. I figur 2.5 illustreras denna process. Notera att felen i felrepresentationen kancellerades genom att de flyttades till samma plakett, vilket är något som gäller i allmänhet. När felen har kancellerats fås i spinnrepresentationen en sluten kurva som går genom de spinn som är flippade. Kurvan utgör en gräns mellan två separata segment i torusen och kallas därmed för en trivial loop [20]. Alla triviala loopar kan skapas genom verkan på ett grundtillstånd med vertexoperatorer, och en toric code där spinn flippats så att en trivial loop bildats är således kvar i samma grundtillstånd.



Figur 2.5: *Figuren illustrerar metoden som används för att minimera tillståndets energi. Det är möjligt att kancellera felen genom att flytta dem till samma position. Notera att den kurva som ritas genom de flippade spinnen i spinnrepresentationen är en trivial loop.*

Figur 2.6 visar ett alternativt sätt att korrigera samma fel som i figur 2.5, genom att flippa andra spinn. Även i detta fall kan en sluten kurva som går genom de spinn som

flippats identifieras. Dock märks en viktig skillnad från den slutna kurvan i figur 2.5. På båda sidorna av kurvan i figur 2.6 återfinns exakt samma segment av torusen. En sådan kurva kallas för en icketrivial loop och karaktäriseras av att den går ett helt varv runt torusen [20].



Figur 2.6: *Figuren visar en kvantmekanisk felkorrektion av samma tillstånd som i figur 2.5, men som resulterar i en icketrivial loop.*

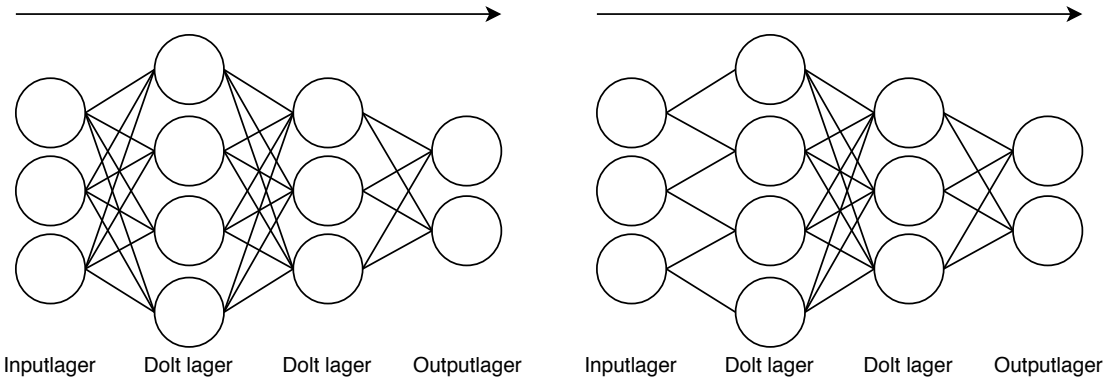
En icketrivial loop som denna kan inte skapas i ett tillstånd med hjälp av vertexoperatorer, och således vet vi att de två olika vägarna till felkancellering har slutat i olika grundtillstånd [20]. Målet med felkorrektion är att bevara informationen som lagrades i koden före det att felen uppkom. För att det ska vara möjligt måste korrektionen ske så att koden återgår till samma grundtillstånd som den hade före feluppkomsten.

## 2.3 Artificiella neurala nätverk

Artificiella neurala nätverk, vanligtvis benämnt neurala nätverk eller nätverk, är en typ av funktionsapproximator som är vanligt förekommande inom maskininlärning. Fördelen med ett neuralt nätverk är att det inte kräver djupgående explicit förståelse för den funktion det ska approximera, vilket också innebär att det kan appliceras på ett vitt spann av problem. I följande stycken beskrivs vad neurala nätverk är samt olika metoder för att optimera dessa för givna applikationer.

### 2.3.1 Uppbyggnad av ett artificiellt neuralt nätverk

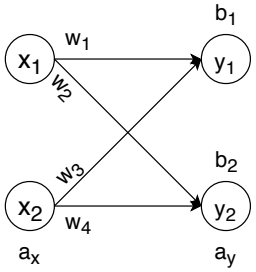
Ett artificiellt neuralt nätverk kan ses som en riktad graf av artificiella neuroner som ordnas i lager, enligt figur 2.7 [21]. Nätverkets första lager kallas för inputlager och nätverkets sista lager kallas för outputlager. De lager som ligger mellan dessa kallas för dolda lager [6]. Sammankopplade neuroner skickar signaler i grafens riktning. Varje koppling mellan neuroner har en associerad viktparameter $w_i$ som viktar signalen $x_i$. De $n$ viktade signa-




(a) *Ett fullt kopplat neuralt nätverk bestående av ett inputlager med tre neuroner, ett dolt lager med fyra neuroner, ett dolt lager med tre neuroner samt ett outputlager med två neuroner.*

(b) *Ett konvolutionellt neuralt nätverk bestående av ett inputlager med tre neuroner. Det första dolda lagret med fyra neuroner utgör det konvolutionella lagret. Resterande är som figur 2.7a.*

Figur 2.7: *Illustration av två olika typer av neurala nätverk.*




Figur 2.8: *Insignalen $y_k$ produceras genom att aktiveringsfunktionen $a_x$ verkar på den viktade insignalen $z_k$ som uttrycks av $z_k = \sum w_i \cdot x_i + b_k$, se ekvation 2.4.*

lerna till en neuron summeras ihop och till dem adderas en bias, $b$ för att ge neuronens viktade insignal $z$ enligt

$$z = \sum_{i=1}^{n} w_i x_i + b. \tag{2.4}$$

För att bestämma vilken signal neuronen skickar vidare används en aktiveringsfunktion, som tar $z$ som argument och matar ut neuronens signal [22]. För en illustration av ett lagers funktionalitet, se figur 2.8.

Vikter och biaser mellan neuronerna i det neurala nätverket initialiseras vanligtvis slumpmässigt med någon sannolikhetsfördelning [21]. För att nätverket ska bli användbart i en tänkt applikation måste sedan vikterna och biaserna justeras genom träning. Det finns olika metoder för att träna ett neuralt nätverk och de metoder som används i det här pro-

jektet beskrivs i avsnitt 2.4. Målet med träningen är att minimera någon kostnadsfunktion, vilket är en funktion som beskriver skillnaden mellan nätverkets faktiska och önskade output. Justering av vikter och biaser sker med gradientoptimering, vilket innebär att man justerar vikter och biaser i den riktning som snabbast minskar kostnadsfunktionen. Kostnadsfunktionen kan definieras på olika sätt som lämpar sig olika väl beroende på nätverkets uppgift. En något mer djupgående framställning av kostnadsfunktioner och gradientoptimering återfinns i appendix A.

Det kan finnas olika kopplingskonfigurationer mellan lagren i ett neuralt nätverk. Två vanliga konfigurationer är fullt kopplade lager, respektive konvolutionella lager [6], vilka illustreras i figur 2.7a respektive 2.7b. Ett fullt kopplat lager är ett lager där varje neuron tar emot samtliga signaler från lagret före. Signalerna till ett konvolutionellt lager skickas däremot bara till vissa av lagrets neuroner.

Konvolutionella lager är främst användbara om det redan finns viss kännedom om vilka insignaler som har väsentliga korrelationer med varandra [6]. I det typiska fallet kan insignalerna i detta fall ordnas i en matris, så att de känt korrelerade signalerna ligger nära varandra i matrisen och kan tolkas som någon geometrisk form. Dessa korrelerade signaler skickas till samma neuron i nästa lager. Vanligen kan samma typ av korrelation förekomma på flera ställen i matrisen, varför förskjutningar av samma geometriska mönster av signaler i matrisen skickas till andra neuroner i nästa lager, med samma vikter och biaser för varje förskjutning av mönstret. Ett sådant mönster, med givna vikter och biaser och förskjutningar, kallas ett filter, och ett konvolutionellt lager har ofta flera olika filter. Ett typiskt exempel där konvolutionella lager är att föredra är vid bildigenkänning. Om en matris innehåller intensiteten hos pixlar i en bild, så kommer ett filter att hjälpa nätverket att identifiera någon typ av detalj, som hörn eller kanter, samt var i bilden de förekommer. Denna information kan i senare lager användas till att identifiera vad bilden föreställer lättare än om all information i bilden skickades vidare till varje neuron i nästa lager som om den vore okorrelerad. Nätverk som inkorporerar åtminstone ett konvolutionellt lager brukar kallas konvolutionella neurala nätverk (CNN, från Convolutional Neural Network).

### 2.3.2 Över- och underanpassade neurala nätverk

Ett problem vid träning av neurala nätverk är över- och underanpassning av nätverkets parametrar. Att ett nätverk är överanpassat innebär att dess parametrar har bestämts på ett sätt som gör att nätverket endast klarar av att hantera träningsdatan [23]. Data som påminner om träningsdatan men som ännu inte påträffats kommer därmed inte att beskrivas väl av nätverket. För att upptäcka överanpassning används testdata separat från träningsdata till att kontrollera nätverkets prestanda. Överanpassning orsakas vanligen av att nätverket har för många parametrar i förhållande till mängden träningsdata. Ett underanpassat nätverk har istället för få parametrar i förhållande till mängden träningsdata, vilket får som konsekvens att nätverket varken kan lära sig träningsdatan eller testdatan

på ett bra sätt. För att motverka överanpassning används dropout, vilket innebär att signaler från neuroner med någon sannolikhet sätts till noll.

## 2.4 Maskininlärning

Metoder som används för att lära ett program att lösa uppgifter som det inte fått direkt programmering för kallas maskininlärning. Optimeringsmetoder som involverar neurala nätverk faller i regel inom detta område. Det finns olika tillvägagångssätt med olika styrkor och svagheter inom maskininlärning. En vanlig metod är övervakat lärande (eng. supervised learning), som innebär att varje träningsfall förses med ett önskat rätt svar, vilket är att föredra när rätt svar är känt [24]. Övervakat lärande kan ofta appliceras direkt på ett neuralt nätverk, genom gradientoptimering av en kostnadsfunktion. För vissa beslutsproblem inkorporeras det neurala nätverket i en större algoritm, där nätverket används för att approximera värdet hos olika beslut. För sådana problem finns det typiskt inte en känd önskad output för nätverket, utan denna får uppskattas med någon form av mål, men i övrigt tränas nätverket i sig på samma sätt som i övervakat lärande. Denna typ av algoritm kallas beteendestyrt lärande (eng. reinforcement learning), och är den metod som främst används i detta projekt.

### 2.4.1 Beteendestyrt lärande

Beteendestyrt lärande är en träningsmetod som kan vara användbar i beslutsprocesser där det går att definiera någon belöningsparameter som ska maximeras i processen [8]. I en beslutsprocess av det här slaget introduceras en beslutsfattande agent i ett system som befinner sig i något tillstånd [25]. Till agenten matas information om systemets tillstånd i form av en observation, och agenten ges handlingar att välja mellan. Efter handlingen tas ett tidssteg. Systemet befinner sig sedan i ett nytt tillstånd och agenten ges den belöning och den observation som hör till det nya tillståndet. Ett exempel på en sådan beslutsprocess är när en beslutsfattande agent ska navigera sig till ett mål i en labyrint på så få steg som möjligt. I ett sådant system kan agenten få koordinater som observation, möjliga förflyttningar som handlingar och negativ belöning varje gång agenten tar ett steg som inte slutar i målet, så att vägen med minst antal steg ger störst ackumulerad belöning.

Låt $R_t$ vara belöningen som agenten ges i tidssteg $t$ och $\gamma$ vara någon diskonteringsfaktor så att $0 \leq \gamma \leq 1$. Då definieras den diskonterade avkastningen $G_t$ för tillståndet som nåddes i tidssteg $t$ enligt [8]

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \tag{2.5}$$

Diskontering sker bland annat för att ta hänsyn till den större osäkerhet som i praktiken finns för prognoser om händelser som ligger fler tidssteg framåt [26]. Varje beslut

agenten fattar bestäms av dess policy, $\pi$, som är en sannolikhetsfördelning över tillgängliga handlingar $a$ för det givna tillståndet $s$ [8],

$$\pi(a|s) = \mathbb{P}(A_t = a|S_t = s), \tag{2.6}$$

där $A_t$ är handlingen vid tidpunkten $t$ och $S_t$ är tillståndet vid tidpunkten $t$. Den förväntade avkastningen givet att systemet befinner sig i tillstånd $s$ och givet att agenten följer policyn ges av den så kallade värdefunktionen $v_\pi(s)$ definierad enligt [8]

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s]. \tag{2.7}$$

Målet för beteendestyrt lärande blir således att finna den policy $\pi$ som maximerar $v_\pi(s)$ för alla $s$, alltså den policy som maximerar den förväntade avkastningen oavsett tillstånd [25]. Det visar sig ibland fördelaktigt att förbättra en policy i termer av dess enskilda handlingar, varför Q-värdefunktionen $q_\pi(s, a)$ definieras som [26],

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a]. \tag{2.8}$$

Q-värdesfunktionen beskriver den förväntade avkastningen om systemet befinner sig i tillstånd $s$ och agenten väljer handling $a$ för att sedan följa policy $\pi$. Målet med beteendestyrt lärande kan nu skrivas om som att finna den policy $q_*(s, a)$ som maximerar Q-värdefunktionen för varje tillstånd $s$ och varje handling $a$ [25],

$$q_*(s, a) = \max_\pi q_\pi(s, a). \tag{2.9}$$

I allmänhet är inte Q-värdesfunktionen för en given policy känd a priori. För att finna en policy som tillfredsställer ekvation (2.9), eller en som åtminstone kommer nära att göra det, måste i regel en iterativ metod användas som initieras med en policy $\pi$ och en uppskattning $Q(s, a)$ av dess sanna Q-värdesfunktion $q_\pi(s, a)$ [8]. Genom att sedan följa policyn fås information om $q_\pi(s, a)$, så att $Q(s, a)$ kan korrigeras, vilket brukar kallas policyutvärdering [26]. Efter korrigering av $Q(s, a)$ uppdateras policyn så att sannolikheten att välja den handling som maximerar funktionen blir större, vilket kallas policyuppdatering. På samma sätt som tidigare följs nu den nya policyn så att $Q(s, a)$ åter kan korrigeras från ny information och bättre approximera den nya Q-värdesfunktionen.

När och hur korrigering av $Q(s, a)$ sker i policyutvärderingen, samt hur lång utvärdering som sker innan policyn uppdateras, beror på vilken specifik metod för beteendestyrt lärande som används [8]. Det finns olika funktioner som kan användas för att uppskatta Q-värdesfunktionen. Ett sätt att uppskatta den är med hjälp av en Q-värdestabell, det vill säga en tabell med ett värde för varje handling i varje tillstånd som agenten stött på [25]. En sådan tabell blir dock snabbt väldigt stor för problem där många tillstånd är möjliga. Vanligt är därför att istället uppskatta Q-värdesfunktionen med ett neuralt nätverk [8].

En vanlig metod för policyutvärdering är *Temporal Difference* (TD). TD innebär att $Q(s, a)$ uppdateras efter varje handling. Antag att agenten vid tidssteg $t$ befinner sig i tillstånd $s_t$ och väljer handling $a$. Agenten befinner sig i nästa tidssteg i det nya tillståndet $s_{t+1}$ och får belöningen $R_{t+1}$. TD-målet $Q_{TD}$, som är en uppskattning av den sanna värdefunktionen, definieras enligt

$$Q_{TD}(s, a) = R_{t+1} + \gamma V(s_{t+1}), \tag{2.10}$$

där $\gamma$ är diskonteringsfaktorn och $V(s)$ ges av

$$V(s) = \max_a Q(s, a). \tag{2.11}$$

Korrigeringen sker nu så att skillnaden mellan $Q(s, a)$ och $Q_{TD}(s, a)$ minskas [8]. Om $Q(s, a)$ är ett neuralt nätverk sker detta lämpligtvis genom gradientoptimering (se appendix A.2) och lämpligt val av kostnadsfunktion. Om $Q(s, a)$ är en Q-värdestabell kan det ske enligt

$$Q(s, a) \leftarrow Q(s, a) + \eta(Q_{TD} - Q(s, a)), \tag{2.12}$$

där $\eta$ är inlärningshastigheten. Som ses hamnar $Q(s, a)$ närmre $Q_{TD}$ efter en sådan uppdatering. På det här sättet uppdateras $Q(s, a)$ efter varje steg. Den TD-metod där policyuppdatering också sker när $Q(s, a)$ uppdateras, så att en ny policy fås efter varje tidssteg, kallas SARSA-algoritmen (State, Action, Reward, State, Action) [8].

Andra metoder som finns för policyutvärdering är Monte Carlo och TD($\lambda$) [26]. Vid Monte Carlo-inlärning är målet som används för justering fullständig avkastning, något som kräver att beslutsprocessen kan delas in i episoder med väldefinierad början och slut. TD($\lambda$) kan kan sägas vara ett mellanting mellan Monte Carlo och TD, där målet som används är ett viktat medelvärde av nådd avkastning efter varje steg följandes tidssteget för vilket justering sker. Viktningen sker med hjälp av en parameter $\lambda$ på ett sätt som tar större hänsyn till senare händelser ju större $\lambda$ är, och så att $0 < \lambda < 1$. SARSA($\lambda$) är en policyuppdateringsmetod som hör ihop med TD($\lambda$). För det här projektets omfattning ges ingen djupare förklaring av dessa metoder.

Ovan har policyuppdatering nämnts som ett steg i beteendestyrt lärande, och nedan beskrivs en vanlig metod för detta. När utvärdering av en policy har skett, exempelvis efter ett utvärderingssteg i SARSA, väljs en policy definierad så att agenten i varje tillstånd $s$ väljer den handling $a$ som maximerar $Q(s, a)$ med sannolikhet $1 - \epsilon$, och annars väljer en handling slumpmässigt [8]. En sådan policy kallas en *$\epsilon$-greedy policy* [25]. Variabeln $\epsilon$, som kallas för explorationsfaktor, införs för att agenten inte för tidigt i inlärningen ska sluta besöka tillstånd som den har liten erfarenhet av. Vanligtvis minskas $\epsilon$ med antalet inlärningsiterationer, exempelvis enligt formeln

$$\epsilon = (k + 1)^{-\alpha} \tag{2.13}$$

där $\alpha$ är en positiv konstant och $k$ är antalet policyuppdateringar som gjorts. Detta för att agenten ska ta större hänsyn till sin erfarenhet ju mer erfarenhet den har.

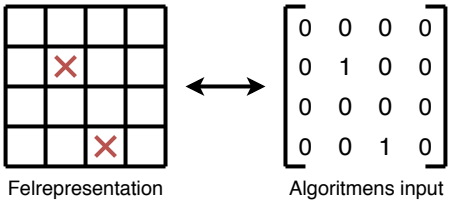
# 3
# Metod

För att utföra kvantmekanisk felkorrektion i toric code-modellen behövde en algoritm designas och implementeras. Algoritmens uppgift var att ta tillståndsrepresentationer som input och föreslå handlingar som korrigerar systemets fel. Inlärningsalgoritmen färdigställdes till en SARSA-metod med Q-värdesfunktion approximerad av neurala nätverk.

För att implementera de neurala nätverken utnyttjades biblioteket Tensorflow som erbjuder en mängd färdigställda funktioner på låg abstraktionsnivå. Därutöver användes biblioteket Keras, som erbjuder Tensorflows funktionalitet på en högre abstraktionsnivå. Algoritmen tränades och testades med ett fullt kopplat respektive ett konvolutionellt nätverk.

## 3.1 Beteendestyrt lärande med neurala nätverk

För att implementera beteendestyrt lärande på kvantmekanisk felkorrektion behövde en tillståndsrepresentation definieras. Som beskrivet i avsnitt 2.1 är det endast fysikaliskt möjligt att mäta produkten av alla spinn som omsluter en plakett, därför definierades tillståndsrepresentation till felrepresentationen presenterad i avsnitt 2.2. I figur 3.1 illustreras hur tillståndsrepresentationen presenterades för algoritmen.




Figur 3.1: *Relation mellan felrepresentationen och motsvarande input till algoritmen. En nolla motsvarar låg energi och en etta motsvarar hög energi.*

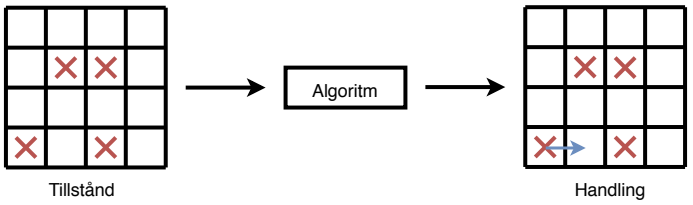
För att flytta felen använde algoritmen enbart felrepresentationen, men den bakomliggande spinnrepresentationen uppdaterades parallellt för att slutligen avgöra om grundtillståndet var bevarat.

Det neurala nätverket implementerades för att ge en approximation av systemets Q-värden för de fyra potentiella handlingarna $a \in A$ givet ett specifikt fel $e \in E$ i tillståndet, se figur 3.2. Handlingsrummet $A$ och mängden $E$ definierades som

$$A := \{\text{upp(U), ner(D), höger(R), vänster(L)}\}, \tag{3.1}$$

$$E := \{\text{mängden av fel i tillståndet}\}. \tag{3.2}$$

Eftersom det neurala nätverkets outputlager hade ett fixt antal neuroner behövde outputstorleken hållas fixerad. Q-värdet kunde därför inte utgöras av en matris beroende av antal fel $e \in E$ då mängden fel ändras under en sekvens. Nätverkets output definierades istället till att enbart utgöra Q-värden för handlingarna tillhörande ett specifikt fel $e$, vilket betecknades $Q_e(s, a)$.

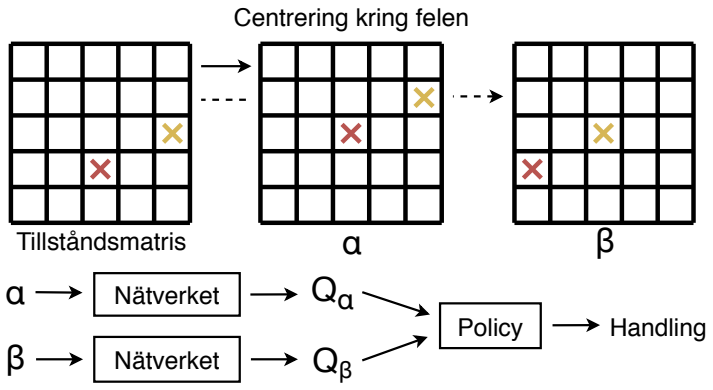



Figur 3.2: *Figuren illustrerar hur algoritmen, givet ett tillstånd, planerades att returnera ett förslag till en handling.*

Det krävdes en representation som det neurala nätverket kunde associera med ett specifikt fel, för att sedan returnera Q-värdena för samtliga av felets möjliga handlingar. För att lösa detta implementerades en metod som centrerade tillståndet kring fel i systemet, se figur 3.3. Centreringen begränsade gittrets, och därmed matrisernas, sidlängder till att vara udda. De centrerade matriserna matades sedan in till nätverket som returnerade

$$Q_e(s, :) = \begin{bmatrix} q_e(s, U) & q_e(s, D) & q_e(s, R) & q_e(s, L) \end{bmatrix}. \tag{3.3}$$

En fördel med att använda de centrerade representationerna av tillstånden var att systemets periodicitet inte behövde tas hänsyn till av nätverket. Det är translationssymmetrin som en följd av kodens periodicitet som tillåter denna centrering. Algoritmen betraktade en centrering av varje individuellt fel, varifrån policyn sedan bestämde vilken handling som skulle utföras, se figur 3.3.




Figur 3.3: *Exempel för algoritmen vid två fel. Först skapades två centrerade representationer $\alpha$ och $\beta$ kring de två felen $e_\alpha$, $e_\beta \in E$. Dessa representationer gavs som input till nätverket som returnerade Q-värdesvektorer för handlingarna tillhörande de två felen. Policyn valde ut den handling med störst Q-värde.*

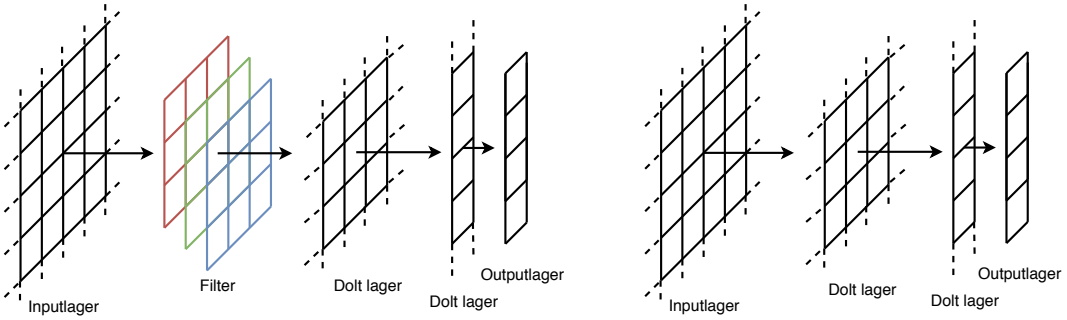
Inlärningsmetoden som användes var SARSA, förklarad i sektion 2.4.1. Belöningen definierades till

$$r = \begin{cases} 10, \text{ om två fel kancelleras,} \\ -1, \text{ efter varje handling,} \end{cases} \tag{3.4}$$

med diskonteringsfaktorn $\gamma = 0.9$. För att säkerställa balans mellan exploration och att använda policyn ansattes en explorationsfaktor $\epsilon$ som funktion av antalet inlärningssekvenser $x$. I takt med att antalet besökta tillstånd ökade var det önskvärt att minska explorationsgraden, därför gjordes $\epsilon$ avtagande enligt

$$\epsilon(x) = \frac{1}{\sqrt{x+1}}. \tag{3.5}$$

Två olika nätverksstrukturer testades separat för att approximera Q-värdesfunktionen, ett konvolutionellt samt ett fullt kopplat nätverk. Båda nätverken bestod av ett inputlager, två dolda lager och ett outputlager. Skillnaden var att det första dolda lagret var konvolutionellt i det ena och fullt kopplat i det andra, se figur 3.4. Notera att enbart det första lagret i det konvolutionella nätverket utgjorde ett konvolutionellt lager, resterande är fullt kopplade.




(a) *Det konvolutionella nätverket.* (b) *Det fullt kopplade nätverket.*

Figur 3.4: *I figuren illustreras de två nätverkstyperna. Inputlagren samt de dolda lagren dimensionerades av tillståndsmatrisens storlek. Outputlagren utgjorde de olika potentiella handlingarna givet ett fel. För specifik struktur, se appendix E.1.*

För att motverka att nätverket överanpassades på den givna datan introducerades dropout med en 20 % randomisering mellan varje dolt lager, se avsnitt 2.3.2.

## 3.2 Datagenerering för toric code-modellen

Två olika metoder användes för att generera träningsdata.

*Metod 1* flippade $n$ olika spinn i gittret, för att skapa högst $2n$ fel. Antalet fel kunde bli mindre om de slumpmässigt flippande spinnen istället flyttade på ett redan skapat fel. Typiskt blev felen utplacerade i par, på var sida om ett flippat spinn. För att generera data användes spinnrepresentationen som sedan översattes till en motsvarande felrepresentation.

*Metod 2* placerade istället ut ett jämnt antal fel på slumpmässiga positioner i gittret; dessa fel skapades inte nödvändigtvis intilliggande, utan låg oftast på längre avstånd från varandra.

En algoritm skapades i Python som med hjälp av metoderna beskrivna ovan kunde generera en mängd olika starttillstånd. I denna algoritm kunde storleken på gittret, antal skapade fel och antalet önskade starttillstånd varieras. För att skapa mer variation i datan och motverka brott av rotationssymmetrin roterades varje genererat tillstånd $90, 180$ och $270$ grader. De två metoderna var viktiga för att avgöra hur eventuella skillnader i felens placering påverkade prestandan.

## 3.3 Prestandatester

De neurala nätverken tränades för att undersöka dels antalet handlingar som krävdes för att korrigera alla fel, men framför allt andelen sekvenser som resulterade i bevarat grund-

tillstånd. En sekvens definierades som följden tillstånd från den initiala felrepresentationen till dess att inga fel kvarstår.

Först testades de två nätverksstrukturerna för att jämföra hur väl de kancellerade fel utan att byta grundtillstånd. De två datagenereringsmetoderna utnyttjades för att generera 560 000 felrepresentationer med $7 \times 7$-gitter och fyra fel enligt *metod 1*. Vid testning evaluerades 10.000 felrepresentationer och resultaten presenteras i avsnitt 4.1 samt 4.2.

För att undersöka hur träningsparametrar påverkade prestandan av nätverket tränades identiska konvolutionella nätverk på gitter av olika storlek och olika antal flippade spinn enligt tabell 3.1. Hur antalet flippade spinn som nätverket tränades på påverkade prestandan studerades sedan genom att testa nätverken för $7 \times 7$-gitter på 1, 2, 3, 4, 5, 6, 7, 8, 9 och 10 flippade spinn. Resultaten presenteras i avsnitt 4.3.

Tabell 3.1: *Prestandatest för det konvolutionella nätverket. Det genererades* 560 000 *felrepresentationer med metod ett.*

| Gitterstorlek | $5 \times 5$ | $7 \times 7$ | $9 \times 9$ | $11 \times 11$ |
|---|---|---|---|---|
| Antal flippade spinn | 4 | 2, 4, 6 | 4 | 4 |

Slutligen tränades två konvolutionella nätverk med data genererad av *metod 1* respektive *metod 2* för att undersöka hur skillnaderna i träningsdata påverkade prestandan. Testningen skedde med data genererad av *metod 1*. Gitterstorleken var $7 \times 7$ och antalet fel var 8. Resultaten presenteras i sektion 4.3.

# 4

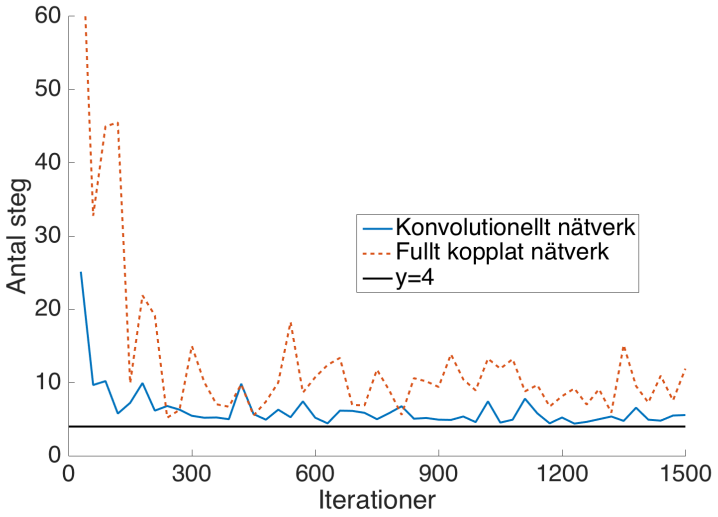
# Resultat

Två olika nätverksstrukturer, en fullt kopplad och en konvolutionell, tränades med beteendestyrt lärande. Inledningsvis presenteras hur snabbt de två nätverken lär sig att kancellera fel. Därefter följer en jämförelse av hur ofta de olika nätverksstrukturerna återför systemet till rätt grundtillstånd. Slutligen undersöks mer ingående hur olika val i träningsprocessen såsom storlek på gittret och antalet flippade spinn påverkar nätverkets förmåga att återföra systemet till rätt grundtillstånd. Samtliga resultat presenteras i form av figurer där ett konfidensintervall med konfidensgraden 95 % inkluderats för att illustrera hur noggranna resultaten är. Viktigt att påpeka är att detta konfidensintervall inte behandlar variansen mellan identiska nätverk utan fungerar som ett mått på variansen hos precisionen för just dessa nätverk.

## 4.1 Träning av de neurala nätverken

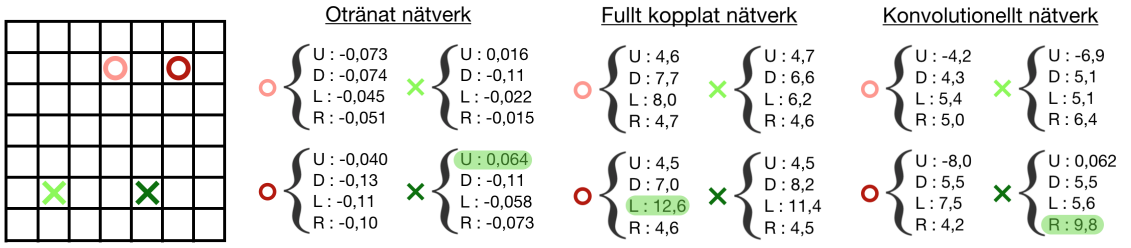
Nätverken konstruerades enligt metodavsnitt 3.1 och tränades sedan under 560 000 iterationer. I figur 4.1 visas hur många steg nätverket behöver ta för att eliminera alla fel i toric code-modellen som funktion av antalet träningsiterationer. Träningsdatan är genererad enligt *metod 1* i avsnitt 3.2, där gittret har sidlängd 7 och 4 slumpmässiga spinn har flippats. För varje datapunkt i figuren har en medelvärdesbildning över 30 mätvärden genomförts. Detta för att minska bruset och på så sätt göra figuren mer överskådlig. Värt att notera från figuren är att nätverken redan efter 500 iterationer är nära det minimala antalet steg som krävs för att kancellera felen. Vidare tycks det konvolutionella nätverket konvergera snabbare än det fullt kopplade och ligger närmare den undre gränsen på 4 steg[1].

[1]För varje spinn som flippas uppkommer oftast två fel och annars flyttas två redan existerande fel ifrån varandra. Det minsta antalet steg som krävs för att kancellera samtliga fel är därför samma som det antalet spinn som initialt flippats




Figur 4.1: *Antal steg som krävs för att kancellera samtliga fel hos ett gitter med sidlängden 7 och 4 slumpmässigt flippade spinn. Den horisontella linjen y=4 representerar den undre gränsen för antalet steg som krävs. Både det konvolutionella och det fullt kopplade nätverket tycks konvergera mot denna gräns även om det finns en tydlig varians. Notera att det konvolutionella nätverket i regel kräver mindre steg än det fullt kopplade.*

Ett annat sätt att illustrera hur nätverken förbättrats under beteendestyrt lärande är att konkret visa vilket drag ett otränat respektive tränat nätverk skulle göra på ett specifikt tillstånd. I figur 4.2 visas Q-värden för tre typer av nätverk - ett otränat och två tränade (varav ett fullt kopplat och ett konvolutionellt). Handlingen med det största Q-värdet, som är markerat för samtliga nätverkstyper, väljs som nästa handling med sannolikhet $1 - \epsilon$. Från figuren noteras att ett otränat nätverk hade flyttat upp det mörka krysset, vilket inte är optimalt om man vill minimera antalet steg som krävs för att eliminera samtliga fel. De två tränade nätverken tar däremot bättre beslut även om det konvolutionella nätverket hade kunnat spara ett steg genom att flytta det mörka krysset åt vänster istället för åt höger.




(a) *Tillståndet som användes för att generera Q-värden.*

(b) *Q-värden för ett otränat nätverk.*

(c) *Q-värden för det fullt kopplade nätverket*

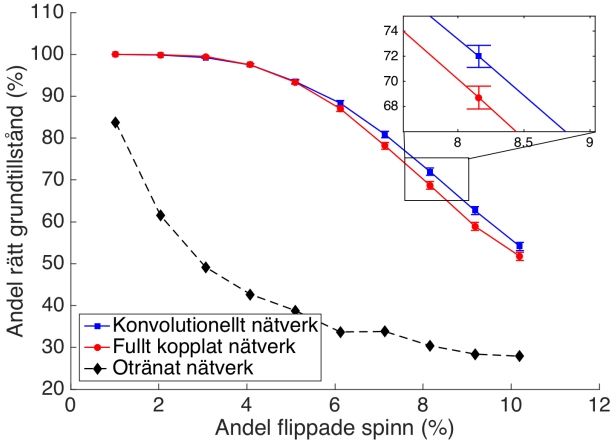
(d) *Q-värden för det konvolutionella nätverket*

Figur 4.2: *Q-värden för tre olika nätverk givet tillståndet i (a). De två kryssen och cirklarna i (a) representerar fel. U, D, L och R betecknar de olika riktningar som ett givet fel kan flyttas. Notera att det otränade nätverket tar ett ofördelaktigt beslut medan de två tränade nätverken tar beslut som möjliggör snabbare kancellering av felen.*

## 4.2 Olika nätverksstrukturers förmåga att välja rätt grundtillstånd

Tidigare presenterades hur träning av nätverken resulterade i en ökad förmåga att eliminera fel i toric code-modellen. För att undersöka hur denna förmåga överförs till att inte bara kancellera alla fel utan även återföra systemet till det ursprungliga grundtillståndet, testas två olika färdigtränade nätverksstrukturer enligt metodavsnitt 3.3. Båda nätverken har tränats på fel som orsakats av slumpmässigt flippade spinn och alltså angränsar i regel felen till varandra, se *metod 1* i avsnitt 3.2. Figur 4.3 visar hur stor andel av gångerna som algoritmen för tillbaka systemet till rätt grundtillstånd som funktion av andelen spinn som initialt blev flippade.

Värt att notera är att båda tränade nätverken ger mycket goda resultat då en låg andel spinn är flippade medan nätverkens prestanda försämras avsevärt då andelen fel ökar. Vidare visar figuren att det konvolutionella nätverket, representerat med kvadrater, presterar bättre än det fullt kopplade nätverket som är representerat med cirklar. Notera dock att båda nätverksstrukturerna presterar betydligt bättre än ett otränat nätverk, vilket representeras av streckade linjer i figuren.

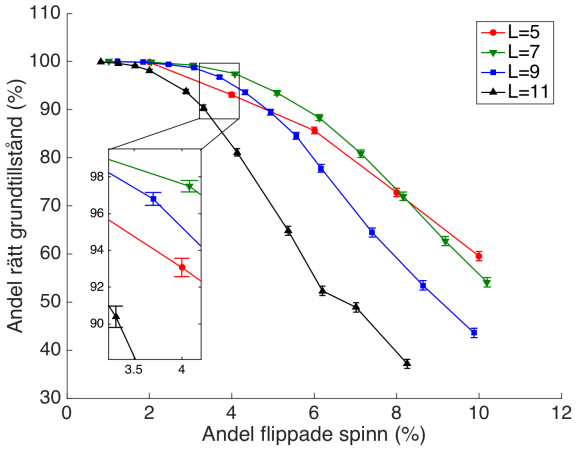



Figur 4.3: *Prestandan för ett konvolutionellt och fullt kopplat nätverk som tränats med storlek 7 och 4 slumpmässigt flippade spinn. De båda tränade nätverken presterar betydligt bättre än det otränade och för de två tränade nätverken visar det konvolutionella upp bäst precision. Notera att för denna och kommande figurer så representerar linjerna mellan punkterna inte någon data utan är där av illustrativa skäl. För varje datapunkt har ett konfidensintervall med konfidensgraden 95 % beräknats.*

## 4.3 Träningsmetodens påverkan på det konvolutionella nätverket

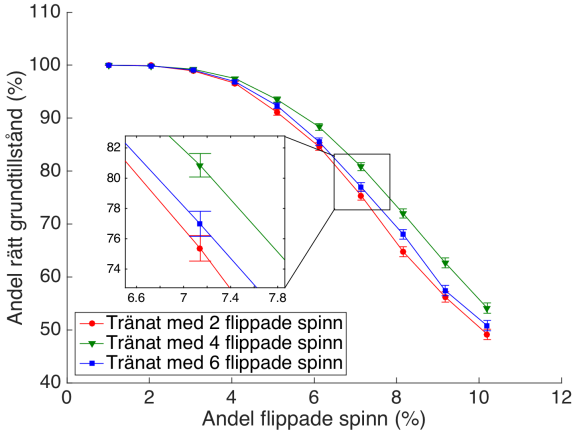
Då det konvolutionella nätverket presterade bättre än det fullt kopplade utgår vi hädanefter från det konvolutionella nätverket och undersöker hur olika val i träningsprocessen påverkar dess förmåga att föra systemet tillbaka till rätt grundtillstånd. Figur 4.4 visar hur denna förmåga beror på vilken gitterstorlek nätverket tränas på[2]. Precis som tidigare figurer visar denna hur ofta algoritmen återför systemet till rätt grundtillstånd beroende på andelen flippade spinn i gittret. Sämst presterar de nätverk som tränats på störst gitterstorlekar. En intressant iakttagelse som bryter denna tendens är att nätverket som tränats med sidlängden 7 har högre precision i de flesta fall jämfört med det nätverk som tränats på gittret med sidlängd 5.

[2]Nätverkets dimensioner skalas med gitterstorleken, se appendix E.



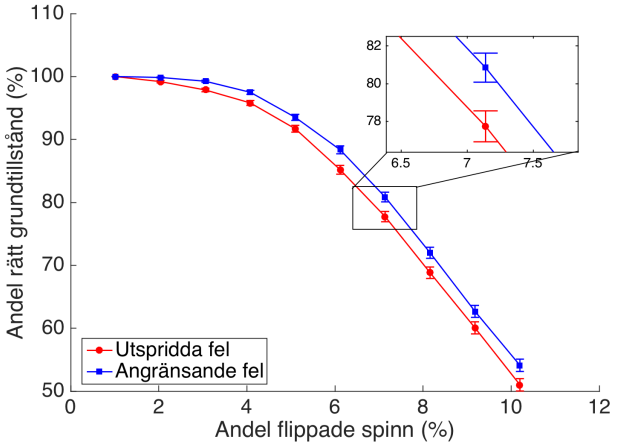

Figur 4.4: *Fyra konvolutionella nätverk som tränats på olika gitterstorlekar, $L$, men med samma antal flippade spinn. Fram till cirka 2 % flippade spinn uppvisar samtliga storlekar maximal prestanda. Därefter presterar det nätverk som tränats på sidlängden $L = 7$ bäst för de flesta felkoncentrationerna.*

I figur 4.5 har nätverken istället tränats på den konstanta sidlängden $L = 7$, men med varierande antal flippade spinn. Fram till ungefär 3 % flippade spinn visar samtliga nätverk mycket god prestanda med en precision på nästan 100 %. Då andelen flippade spinn i gittret ökar visar dock det nätverk som tränats på 4 flippade spinn upp bäst resultat, även om skillnaderna är relativt små. Sämst prestanda ger det nätverk som tränats på lägst andel flippade spinn medan det nätverk som tränats på högst antal fel hamnar emellan de andra två.




Figur 4.5: *Tre konvolutionella nätverk som tränats på gitterstorlek $7 \times 7$, men på olika antal flippade spinn. Bäst presterar det nätverk som tränats på 4 flippade spinn.*

I samtliga fall ovan är nätverken tränade på fel som uppkommit från slumpmässigt flippade spinn. Detta ger med hög sannolikhet upphov till angränsande fel, vilket är det sätt som felen uppkommer i praktiken. Vidare undersöktes om nätverkets precision ökar om det istället tränas på utspridda fel. I figur 4.6 illustreras hur just detta val påverkar nätverkets prestanda. Träningsdatan är generad med *metod 1* och *metod 2*, enligt avsnitt 3.2, för att sedan evalueras på testdata genererad enligt *metod 1*. Från figuren noteras att nätverket som tränats på angränsande fel har en högre precision än det nätverk som tränats på utspridda fel.




Figur 4.6: *Två konvolutionella nätverk har tränats på gitterstorlek* $7 \times 7$ *med 4 flippade spinn, men med olika typ av träningsdata. Datan med de angränsande felen är genererad enligt metod 1 medan datan med de utspridda felen är genererad med metod 2, se sektion 3.2. Det nätverk som tränats med metod 1 uppvisar högst precision.*

# 5

# Diskussion

Projektet syftar till att besvara hur väl artificiella neurala nätverk lämpar sig som verktyg för kvantmekanisk felkorrektion. Om det inte hade varit för de fyra grundtillstånden så hade problemet enkelt kunnat lösas med hjälp av traditionella algoritmer. Svårigheten att identifiera en optimal handling som tar hänsyn till grundtillståndet gör det intressant att undersöka om beteendestyrt lärande effektivt kan tillämpas. I detta kapitel diskuteras den valda metodens styrkor och svagheter. Den konstruerade algoritmens prestanda analyseras för att ge förslag på intressanta områden för fortsatta studier av beteendestyrt lärande som metod för kvantmekanisk felkorrektion.

## 5.1 Tillståndsrepersentation

Algoritmen beslutar vilken handling som ska utföras enbart med hjälp av tillståndsrepresentationen, vilket gör det viktigt att den innehåller så mycket information som möjligt om systemet. Tillståndsrepresentationen som används visas i figur 3.1. För att träna det neurala nätverket så effektivt som möjligt kan det i vissa fall vara motiverat att ge nätverket en representation av problemet som skiljer sig från tillståndsrepresentationen. I den konstruerade algoritmen centreras ett fel åt gången och genererar en uppsättning representationer som var och en utvärderas av nätverket innan en handling bestäms, se figur 3.3.

Vid första anblick kan den representation som nätverket ser framstå som problematisk. Exempelvis kan endast de spinn som ligger i direkt anslutning till ett fel flippas av algoritmen. Detta visar sig emellertid inte hindra nätverket från att agera optimalt. Genom att flippa ett annat spinn skulle nya fel genereras snarare än att befintliga fel flyttas eller kancelleras. Därmed försämras varken algoritmens förmåga att återgå till rätt grundtillstånd eller antalet steg algoritmen kräver för att göra det, samtidigt som problemet blir mer hanterligt. Centreringen begränsar även gittrets sidlängder till att vara udda, då det för jämna sidlängder inte finns en plakett i mitten. Det är däremot enkelt att konstruera en analog algoritm som istället hanterar gitter med jämna sidlängder. Detta görs genom att placera det aktuella felet på en av de fyra positionerna som då finns i centrum.

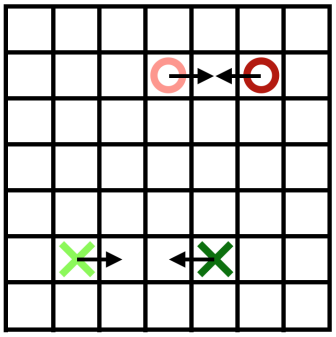
Gittret begränsas i denna studie till att ha samma sidlängd i horisontell- respektive vertikalled, men det är även möjligt att applicera algoritmen på gitter vars sidlängder skiljer sig i de båda leden. Den valda representationen inför med andra ord få begränsningar. Valet är dock inte entydigt och det är möjligt att det finns representationer som underlättar nätverkets arbete ytterligare. Ett alternativ som bygger på att modellera gittrets periodicitet genom att expandera tillståndsrepresentationen visas i Appendix D.

## 5.2 Granskning av resultat

Nätverken tränas för att kancellera alla fel på så litet antal steg som möjligt. Detta medför i regel också att grundtillståndet kommer att bevaras, vilket är det som önskas av algoritmen.

### 5.2.1 Q-värden

Figur 4.2 visar vilka Q-värden nätverket returnerar för de möjliga handlingarna i ett exempeltillstånd. Dessa kan jämföras med de optimala handlingarna givet samma tillstånd, se figur 5.1.



Figur 5.1: *Pilarna representerar de handlingar som förväntas felkorrigera tillståndet på minimalt antal steg.*

Precis som förväntat så ser det otränade nätverkets föreslagna handlingar slumpmässiga ut, vilket visas i figur 4.2b. Det fullt kopplade nätverket i figur 4.2c föreslår att den mörka cirkeln ska flyttas åt vänster, vilket överensstämmer med de optimala handlingarna i figur 5.1. Slutligen noteras att den föreslagna handlingen från det konvolutionella nätverket är att flytta det mörka krysset åt höger, se figur 4.2d. Denna taktik bör, om nätverket konsekvent föreslår denna handling, kancellera de två kryssen. Kancellationen kan dock göras mer effektiv genom att flytta det mörka krysset åt vänster, då detta ökar sannolikheten att systemet återförs till rätt grundtillstånd. Med andra ord agerar det konvolutionella nätverket inte optimalt för detta tillstånd.

Betrakta Q-värdena från det fullt kopplade nätverket och det konvolutionella nätverket i figur 4.2. Notera tendensen för nätverket att värdera samma riktning högst för de båda kryssen respektive cirklarna. Eftersom optimalt beteende är att para ihop kryssen för sig och cirklarna för sig hade det varit önskvärt att det ljusa krysset och den ljusa cirkeln hade värderat handlingen R framför handlingen L. Detta är varken fallet för det fullt kopplade- eller konvolutionella nätverket. Trots att detta indikerar att algoritmens värdering av handlingar kan förbättras lyckas den i regel ändå snabbt eliminera alla fel. För att få en djupare förståelse bakom vad som orsakar de icke-ideala handlingarna skulle ett första steg kunna vara att kartlägga huruvida effekten kommer som en konsekvens av nätverksstrukturen eller inlärningsmetoden.

### 5.2.2 Algoritmens förmåga att återföra systemet till rätt grundtillstånd

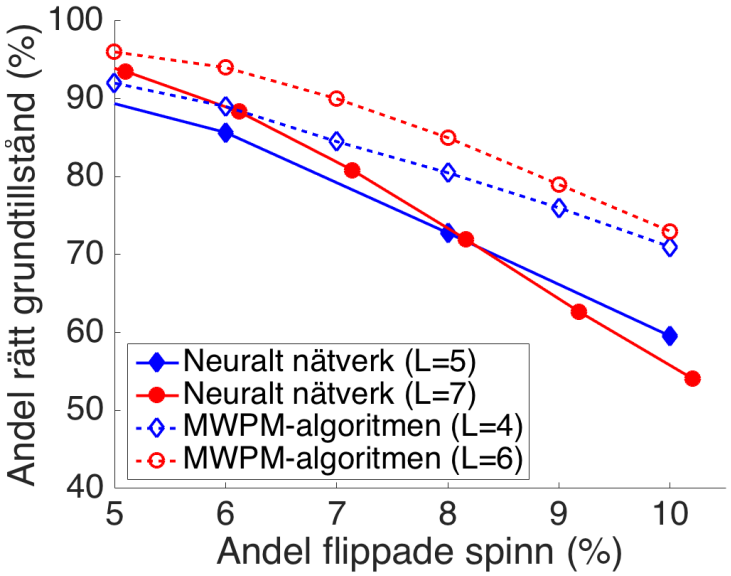
Figur 4.3 visar att det konvolutionella nätverket återför systemet till rätt grundtillstånd oftare än det fullt kopplade nätverket, men att skillnaden i prestanda mellan de två är relativt liten. Notera att konfidensintervallen för de två nätverken inte överlappar då andelen flippade spinn övergår 5 %. Med statistisk signifikans presterar det konvolutionella nätverket bättre än det fullt kopplade nätverket i detta område. Resultatet är lite överraskande i och med att konvolutionella nätverk huvudsakligen används för att identifiera lokala mönster i given input. I tillståndsrepresentationen som algoritmen betraktar är det dock positionerna av felen som är det relevanta och det är därför inte uppenbart att det konvolutionella nätverket har en fördel. Däremot kan det förväntas att strukturen som det konvolutionella nätverket använder resulterar i snabbare konvergens, något som figur 4.1 antyder. Den kortare konvergenstiden i kombination med det konvolutionella nätverkets bättre prestanda gör denna till den föredragna strukturen. Det bör dock nämnas att resultaten för två nätverk av samma typ, som tränats på samma data, kan komma att variera. En anledning till detta är att nätverkens vikter och biaser slumpmässigt initieras och att de på så sätt kan skilja sig lite då nätverket inte tränas i oändlig utsträckning. En annan anledning är explorationens slumpmässiga inverkan på träningen. Dessutom kan små varianser i nätverkens parametrar, såsom storleken av dess lager, komma att påverka de erhållna resultaten. För att kunna bestämma vilken av nätverksstrukturerna som är bäst lämpad för kvantmekanisk felkorrektion behöver därför en större uppsättning nätverk än de som visas i figur 4.3 jämföras.

Förutom jämförelsen mellan de olika nätverksstrukturerna undersöktes hur nätverkens träning påverkar resultaten. Exempelvis tränades ett konvolutionellt nätverk på gitter med olika antal fel. Ett intressant resultat, som illustreras av figur 4.5, är att nätverket som tränats på fyra flippade spinn presterar bättre på testdata med sex flippade spinn än vad nätverket som tränats med sex flippade spinn gör. Detta antyder att det möjligen finns en optimal felkoncentration att träna nätverket med struktur enligt tabell E.2 på.

Som tidigare diskuterats i avsnitt 3.2 används två uppsättningar träningsdata. Det är i praktiken möjligt att hamna i tillstånd där inga fel angränsar. Därför undersöktes det om nätverken kunde uppnå bättre prestanda genom att träna på utspridda fel. Tanken var att nätverken, genom denna träningsprocess, även skulle lära sig att att para ihop intilliggande fel. Vid testning av nätverken användes emellertid data med fel orsakade av slumpmässiga spinnflipps, då det är denna typ av fel som nätverken behöver hantera i realistiska situationer. I figur 4.6 noteras att nätverk som tränats på utspridda fel presterar sämre. Detta beror antagligen på att nätverket får kompromissa med hur bra det är på att para ihop närliggande fel. De flesta fel som uppkommer vid slumpmässigt flippade spinn angränsar till varandra, vilket ger nätverken som tränas på den situationen en fördel.

I figur 4.4 noteras att nätverket som tränats på ett gitter med sidlängd 7 presterar bättre än det nätverk som tränats på sidlängd 5, samtidigt som sidlängder större än 7 verkar leda till betydligt sämre resultat. Storleken av de neurala nätverkens lager är proportionella mot kvadraten av gittrens sidlängder och de större nätverken har därmed fler vikter och biaser att justera, se Appendix E.1. Eftersom nätverken är tränade med samma antal iterationer oavsett storlek kan anledningen till de större nätverkens sämre prestanda vara relaterad till att dessa nätverk inte konvergerar lika snabbt.

Torlai och Melko presenterar i *A Neural Decoder for Topological Codes* MWPM-algoritmen (Minimum Weight Perfect Matching) vilken är en klassisk algoritm som utför felkorrektion [10]. En jämförelse mellan MWPM-algoritmen och den algoritm som utvecklats visas i figur 5.2.




Figur 5.2: *Figuren visar prestandan för det konvolutionella neurala nätverket (heldragna linjer) samt MWPM-algoritmen (streckade linjer) som funktion av andel flippade spinn. MWPM-algoritmen har konsekvent högre precision än det neurala nätverket.*

Notera från figur 5.2 att MWPM-algoritmen konsekvent presterar bättre än det neurala nätverket. MWPM-algoritmen korrigerar fel genom att flippa så få spinn som möjligt [10]. Om nätverket hade följt den avsedda policyn hade prestandan för de två algoritmerna varit lika och det är därmed föga förvånande att det neurala nätverket inte överträffar MWPM-algoritmen. Från detta kan slutsatsen dras att nätverket inte agerar optimalt och att det således finns förbättringsmöjligheter.

## 5.3 Framtida studier

Från figur 4.3 noteras att det konvolutionella nätverket presterar mycket bättre än ett otränat nätverk. Däremot noterades det i föregående avsnitt att nätverket inte presterar lika bra som MWPM-algoritmen. En fortsatt utveckling av algoritmen skulle eventuellt kunna höja dess prestanda och göra den konkurrenskraftig gentemot MWPM-algoritmen. I detta kapitel presenteras konkreta förslag på åtgärder som kan komma att förbättra algoritmens prestanda.

### 5.3.1 Q-värden

Som diskuterat är Q-värdena i figur 4.2c och figur 4.2d inte optimala. Det hade därför varit intressant att fortsatt studera hur dessa Q-värden kan optimeras. För att kartlägga

om icke-idealiteter introducerats av träningsmetoden eller nätverksstrukturen vore det intressant att ersätta nätverken med en Q-värdestabell. Q-värdestabellen är till skillnad från nätverken inte en kontinuerlig funktionsapproximator. Därför blir Q-värdena för olika handlingar oberoende av varandra och på så sätt spelar enbart träningsmetoden roll för de slutliga Q-värdena. Problemet med Q-värdestabellen är att dess storlek växer snabbt med antalet fel samt gittrets storlek. Vid förbättring uppdateras dessutom ett värde åt gången i Q-värdestabellen medan nätverken utför gradientoptimering som påverkar nätverkets värdering av flera tillstånd samtidigt. Det är därför tidsödande att träna Q-värdestabellen, exempelvis för ett $7 \times 7$-gitter, på en vanlig persondator. Med tillgång till mer datakraft skulle dock denna typ av undersökning möjliggöras.

### 5.3.2 Nätverksstrukturer

I figur 4.3 noteras att det konvolutionella nätverket presterar bättre än det fullt kopplade. Det vore intressant att i framtida studier fortsatt betrakta hur skillnader i nätverksstruktur påverkar den slutliga algoritmens prestanda.

De använda nätverksstrukturerna är fullt kopplade och konvolutionella. Det finns emellertid andra typer av nätverk vars prestanda möjligtvis kan överträffa den hos det konvolutionella. För att undersöka om det finns ett nätverk som presterar bäst oberoende av gitterstorlek och antal fel skulle det vara intressant att studera en bredare uppsättning nätverksarkitekturer.

Det har på senare tid gjorts stora framsteg inom användandet av djupa neurala nätverk, vars stora antal lager har visat sig vara en framgångsfaktor för att lösa komplexa problem. Ett konkret exempel är Alphago, som är Deepminds go-spelande algoritm[1]. Ett av nätverken som Alphago bygger på har hela 79 lager, vilket kan jämföras med de 2 dolda lagren för nätverken i detta projekt [9]. Det bör dock nämnas att antalet vikter och biaser ökar med nätverksdjupet. Således ökar träningstiden för djupare nätverk, vilket är anledningen till att dessa ännu inte undersökts.

För att kunna jämföra olika nätverk på ett mer exakt sätt vore det även nödvändigt att veta hur stor variansen är mellan olika instanser av en och samma nätverksarkitektur, tränade på samma träningsdata. Det finns en mängd nätverksarkitekturer som vore av stort intresse att studera. Den begränsande faktorn för samtliga av dessa jämförelser är dock träningstid och en ökad datakraft skulle således öppna upp för en mer ingående undersökning.

### 5.3.3 Monte Carlo-inlärning

Inlärningsmetoden som används i projektet är SARSA, med anledning av att den är effektiv samt enkel att implementera. Nätverket tränas enbart på att kancellera fel med så

[1] Deepmind är ett artificiell intelligens-bolag som ägs av Google.

få handlingar som möjligt och det försöker därför inte aktivt återföra systemet till sitt ursprungliga grundtillstånd. Vid en låg felkoncentration återför algoritmen oftast systemet till rätt grundtillstånd ändå, men när felkoncentrationen ökar misslyckas algoritmen allt oftare med detta. Genom att träna nätverket till att även ta hänsyn till grundtillståndets bevarande kan möjligen en algoritm konstrueras vars precision överträffar den hos exempelvis MWPM-algoritmen.

Det gjordes försök att konstruera denna typ av algoritm med hjälp av Monte Carlo-träning. Inom tidsramarna för projektet var det inte möjligt att slutföra en fungerande implementation av denna inlärningsmetod, vilket dock hade varit intressant att fortsatt undersöka. Som beskrivet i avsnitt 2.4.1 bygger Monte Carlo-inlärning på att nätverket uppdateras utifrån den totala avkastningen i slutet av en sekvens. För att avgöra om algoritmen fört tillbaka systemet till rätt grundtillstånd kan en jämförelse mellan spinnrepresentationen av det felkorrigerade tillståndet och det ursprungliga grundtillståndet göras. Om tillståndet är rätt ges en positiv belöning, annars är belöningen negativ. Förhoppningen är således att nätverket lär sig vilka typer av sekvenser som har en hög sannolikhet att föra tillbaka systemet till rätt grundtillstånd.

För att träningsmetoden ska vara så effektiv som möjligt är det nödvändigt att först träna nätverket med SARSA-metoden eller liknande. Detta för att Monte Carlo-inlärning endast uppdaterar nätverkets vikter och biaser efter en hel sekvens, det vill säga när alla fel kancellerats. En implementation av enbart Monte Carlo-inlärning skulle därför ta mycket lång tid att träna. Även om ett nätverk sedan tidigare är tränat med SARSA kan det dock bli problem att övergå till Monte Carlo-inlärning. Anledningen är att hela sekvensen av handlingar får en negativ avkastning om fel grundtillstånd i slutet erhålls. Detta inkluderar även de sista handingarna som enbart kancellerar fel som redan angränsar till varandra. Det är därför sannolikt att nätverket skulle glömma bort hur det kancellerar fel efter ett visst antal träningsiterationer. En möjlig lösning vore därför att kombinera SARSA och Monte Carlo-inlärning samtidigt. Det är ofta de tidiga handlingarna i en sekvens som avgör om rätt grundtillstånd kommer att erhållas och man vill därför att Monte Carlo-inlärningen ska ha störst effekt på dessa handlingar. För att tillskriva de tidiga handlingarna störst vikt kan en idé vara att låta diskonteringen av belöningen ske i motsatt riktning.

En kombination av de två träningsmetoderna skulle kunna implementeras genom att nätverket uppdateras efter varje steg, enligt SARSA. Efter fullföljd sekvens skulle sedan ytterligare uppdatering ske ske av Q-värdet för varje handling med målet $R_T + R_1 + \max_a Q(s_2, a)$ för första handlingen, målet $\gamma R_T + R_2 + \max_a Q(s_3, a)$ för andra handlingen, och så vidare, där $R_t$ är den erhållna belöningen efter handlingen i tidssteg $t$, och $R_T$ är den sista erhållna belöningen i sekvensen.

# 6

# Slutsats

I detta projekt har olika typer av artificiella neurala nätverk tillämpats för att genomföra felkorrektion av uppkomna kvantmekaniska fel i toric code-modellen. Resultaten från projektet indikerar att det finns potential att använda neurala nätverk inom kvantmekanisk felkorrektion. De bästa resultaten erhålls för ett konvolutionellt nätverk som tränats på en toric code av storlek $7\times7$ med 4 slumpmässigt flippade spinn. Då 5 % av spinnen är flippade lyckas nätverket återföra systemet till rätt grundtillstånd i 94 % av fallen, vilket är marginellt sämre än MWPM-algoritmen som lyckas uppnå samma precision med 6 % av spinnen flippade. De Q-värden som approximeras av nätverket uppvisar vissa icke-idealiteter för vilka felförflyttningar som algoritmen anser vara bäst. Detta tyder på att träningen av algoritmen inte, på ett helt korrekt sätt, lyckas framföra dynamiken i problemet. Möjligen är detta anledningen till att nätverken presterar något sämre än MWPM-algoritmen. För att fortsatt utveckla algoritmen är det därför intressant att i framtiden undersöka hur olika nätverksarkitekturer samt andra typer av beteendestyrt lärande, exempelvis Monte Carlo-inlärning, kan påverka prestandan.

# Litteraturförteckning


[1] A. Zecca, "Superposition, Entanglement, and Product of States", *International Journal of Theoretical Physics*, vol. 42, no. 7, pp. 1621-1628, juli, 2003.

[2] M. Siomau och S. Fritzsche, "Quantum computing with mixed states", *The European Physical Journal D*, vol. 62, no. 3, pp. 449-456, maj, 2011.

[3] A. Kitaev, "Fault-tolerant quantum computation by anyons", *Annals of Physics*, vol. 303, no. 1, pp. 2-30, januari, 2003.

[4] M. Awad och R. Khanna, *Efficient Learning Machines*. Berkeley, CA: Apress, 2015.

[5] J. Carrasquilla och R. G. Melko, "Machine learning phases of matter", *Nature Physics*, vol. 13, no. 5, pp. 431-434, maj, 2017.

[6] I. Goodfellow, Y. Bengio och A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[7] M. Kordos et al., "Combining the Advantages of Neural Networks and Deci-sion Trees for Regression Problems in a Steel Temperature Prediction System", Hybrid Artificial Intelligent Systems: proceedings of 7th International Conference, 2012, pp 36-45.

[8] R. Sutton och A. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA: MIT Press, 1998.

[9] D. Silver et al., "Mastering the game of Go without human knowledge", *Nature*, vol. 550, no. 7676, pp. 355-359, oktober, 2017.

[10] G. Torlai och R. G. Melko, "A Neural Decoder for Topological Codes", *Physical Review Letters*, vol. 119, no. 3, juli, 2017.

[11] D. Browne, "Topological Codes and Computation", Universität Innsbruck, 2014.

[12] M. Fischlin och S. Katzenbeisser. *Number Theory and Cryptography Papers in Honor of Johannes Buchmann on the Occasion of His 60th Birthday*, Berlin: Springer, 2013.

[13] R. de Wolf, "The Potential Impact of Quantum Computers on Society", *Ethics and Information Technology*, vol. 19, no. 6, pp. 271-276, december, 2017.

[14] Z. Song och C. Sun, "Quantum information storage and state transfer based on spin-system", *Low Temperature Physics*, vol. 31, no. 8, pp. 686-697, augusti, 2005.

[15] J. Chiaverini et al., "Realization of quantum error correction", *Nature*, vol. 432, no. 7017, pp. 602-605, december, 2004.

[16] M. A. Nielsen och I. L. Chaung, *Quantum Computation and Quantum Information.* Cambridge: Cambridge University Press, 2010.

[17] S. J. Devitt, K. Nemoto och W. J. Munro, "Quantum Error Corrections for Beginners", *Reports on Progress in Physics*, vol. 76, no. 7, juli, 2013.

[18] J. T. Anderson, "Homological stabilizer codes", *Annals of Physics*, vol. 330, no. 1, pp. 1-22, mars, 2013.

[19] H. Weimer, "Quantum simulation of many-body spin interactions with ultracold polar molecules", *Molecular Physics*, vol. 111, no. 12, pp. 1753-1758, juli, 2013.

[20] E. Dennis et al., "Topological quantum memory", *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452-4505, september, 2002.

[21] M. A. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015. [E-bok] Tillgänglig: `http://neuralnetworksanddeeplearning.com`

[22] S. Haykin, *Neural Networks - A comprehensive foundation.* Upper Saddle River, NJ: Prentice Hall, 1999.

[23] T. Dietterich, "Overfitting and undercomputing in machine learning", *ACM Computing Surveys*, vol. 27, no. 3, pp. 326-327, september, 1995

[24] R. D. Reed och R. J. Marks, *Neural smithing : supervised learning in feed-forward artificial neural networks.* Cambridge, MA: MIT Press, 1999.

[25] S. Whiteson, *Adaptive Representations for Reinforcement Learning.* Berlin: Springer, 2010.

[26] C. Szepesvari, *Algorithms for Reinforcement Learning.* San Rafael, CA: Morgan and Claypool Publishers, 2010.

# A

# En mer detaljerad beskrivning av neurala nätverk

I följande avsnitt presenteras en mer rigorös beskrivning av kostnadsfunktioner, gradientoptimering och bakåtpropagering.

## A.1 Kostnadsfunktion

För att beskriva hur väl ett artificiellt neuralt nätverk uppfyller funktionen som det tränats för införs begreppet kostnadsfunktion. En kostnadsfunktion beskriver skillnaden mellan det önskade resultatet och det resultat som erhålls från det neurala nätverket [21]. Det finns många möjliga sätt att definiera kostnadsfunktionen, varav en av de mest grundläggande funktionerna är det genomsnittliga kvadratiska felet,

$$C(w,b) = \frac{1}{2n}\sum_{x} ||y(x) - y_{expected}(x)||^2, \tag{A.1}$$

där $n$ motsvarar antalet träningsdata, $y(x)$ är den output som nätverket ger för träningsdata $x$ och $y_{expected}(x)$ är nätverkets önskade output för träningsdata $x$ [21]. En fördel med denna definition är att en intuitiv bild av kostnadsfunktionens syfte framgår. En annan vanlig kostnadsfunktion är korsentropifunktionen [21] som definieras enligt

$$C(w,b) = -\frac{1}{n}\sum_{x}[y(x)\ln y_{expected}(x) + (1-y(x))\ln(1-y_{expected}(x))]. \tag{A.2}$$

Denna funktion har numeriska fördelar jämfört med det genomsnittliga kvadratiska felet och är därför mer användbar i vissa situationer, speciellt vid klassificeringsproblem [21].

## A.2 Gradientoptimering

Införandet av kostnadsfunktionen ger ett mått på hur bra det neurala nätverket presterar under träning. Om kostnadsfunktionen ger ut låga värden är prestandan hög, vilket gör det naturligt att låta målet med träningen vara att minimera kostnadsfunktionen $C$ [21]. En

effektiv metod för att minimera funktionen är gradientnedstigning (eng. gradient descent) [21]. Enligt denna metod uppdateras vikter och biaser enligt

$$w_{uppdaterad} = w_{gammal} - \eta \frac{\partial C}{\partial w_k} \tag{A.3}$$

respektive

$$b_{uppdaterad} = b_{gammal} - \eta \frac{\partial C}{\partial b_l}, \tag{A.4}$$

där $\eta$ är en parameter som brukar kallas för inlärningshastighet som gör det möjligt att kontrollera hastigheten för vilken vikter och biaser uppdateras. Indexen $k$ och $l$ innebär att derivatan ska beräknas med avseende på varje vikt och varje bias i nätverket. Enligt denna definition ändras vikter och biaser mest i de riktningar som förändrar kostnadsfunktionen mest [21]. Nya värden på vikterna och biaserna kan på detta sätt itereras fram tills nätverket presterar på eftersträvad nivå, det vill säga när kostnadsfunktionen når ett lokalt minimum. För att denna träning ska vara effektiv måste derivatorna i ekvation (A.3) och (A.4) beräknas på ett kostnadseffektivt sätt. Algoritmen som används för att beräkna derivatorna kallas för bakåtpropagering.

## A.3 Bakåtpropagering

Tekniken som används för att effektivt beräkna derivatorna i gradientoptimering kallas bakåtpropagering (eng. backpropagation) och ekvationerna varpå bakåtpropageringen bygger är [21]

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{A.5}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{A.6}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{A.7}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j. \tag{A.8}$$

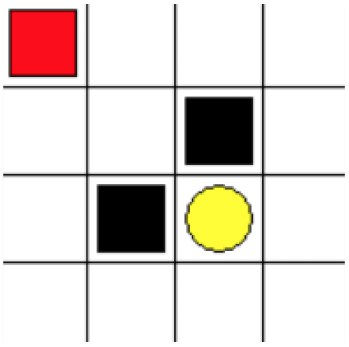
I ekvation (A.5) beräknas felet i utsignalslagret, $\delta^L$, som den elementvisa produkten av gradienten av kostnadsfunktionen med avseende på varje neuron i utsignalslagret, $\nabla_a C$, och derivatan av varje neurons aktiveringsfunktion i utsignalslagret, $\sigma'(z^L)$. Felet i utsignalslagret propageras sedan bakåt till lagret innan genom att multiplicera det med vikterna mellan de två lagren och sedan ta den elementvisa produkten med derivatan av aktiveringsfunktionen för det bakre lagret, enligt ekvation (A.6). På samma sätt propageras felet i varje lager bakåt till lagret innan tills inputlagret nås. När felet i varje lager är känt kan derivatan av kostnadsfunktionen med avseende på biaserna i lager $l$ fås direkt enligt ekvation (A.7). Derivatan av kostnadsfunktionen med avseende på vikterna fås

genom att multiplicera felet i lager $l$ med alla aktiveringar från det tidigare lagret, $a_k^{l-1}$ enligt ekvation (A.8).

# B

# Inspirationsprogram för beteendestyrt lärande

Algoritmen som implementerades för kvantmekanisk felkorrektion tar inspiration från ett existerande maskininlärningsprogram, vars grafiska gränssnitt visas i figur B.1. Programmets uppgift är att flytta den röda fyrkanten till den gula cirkeln, utan att gå över de svarta rutorna.



Figur B.1: *Det grafiska gränssnittet för programmet som modifierades. Den röda fyrkanten uppe till vänster har i uppgift att ta sig till den gula cirkeln, samtidigt som den ska undvika de svarta rutorna.*

Programmet initieras alltid på det sätt som visas i Figur B.1 och den röda fyrkanten (agenten) är det enda rörliga objektet. Agenten har fyra tillåtna operationer: upp, ner, vänster och höger och kan inte passera rutnätets rand. Vidare får agenten en belöning då den lyckas ta sig till den gula cirkeln och en bestraffning då den hamnar på någon av de svarta rutorna. Tillståndet är agentens koordinater, vilket är tillräckligt då den omgivande miljön aldrig förändras. Källkoden till programmet kan hittas på följande webbadress: `https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow`.

# C

# Beteendestyrt lärande med Q-värdestabell

Som utgångspunkt för implementeringen av beteendestyrt lärande användes en färdigställd, offentlig kod som inspiration. Programmet skapade en labyrint på ett rutnät med två utplacerade objekt. Det ena objektet kunde flyttas och hade i uppgift att lära sig hitta den kortaste vägen till det andra objektet. Objektet hade möjlighet att förflytta sig upp, ner, höger eller vänster beroende på vilken handling som ansågs bäst, se appendix B för en djupare förklaring.

Programmet använde sig av beteendestyrt lärande där Q-värden från tidigare påträffade tillstånd sparades i en tabell på följande form

$$Q_{\text{tabell}} = \begin{pmatrix} Q(\mathbf{s}_1, a_1) & Q(\mathbf{s}_1, a_2) & Q(\mathbf{s}_1, a_3) & Q(\mathbf{s}_1, a_4) \\ \vdots & \vdots & \vdots & \vdots \\ Q(\mathbf{s}_n, a_1) & Q(\mathbf{s}_n, a_2) & Q(\mathbf{s}_n, a_3) & Q(\mathbf{s}_n, a_4) \end{pmatrix}. \tag{C.1}$$

Det objekt som rörde sig i labyrinten kunde ses som ett fel i toric code-modellen. Ett första steg var därför att modifiera koden på ett sådant sätt att den tog bort labyrinten och initierade $n$ objekt (motsvarande fel i toric code-modellen) på slumpmässiga koordinater. Med denna modifikation kunde ett tillstånd definieras som

$$\mathbf{s} = \begin{bmatrix} x_1 & x_2 & \cdots & x_k & \cdots & x_{n-1} & x_n \\ y_1 & y_2 & \cdots & y_k & \cdots & y_{n-1} & y_n \end{bmatrix}, \tag{C.2}$$

där $(x_k, y_k)$ anger koordinaterna för fel $k$ i tillståndet. Felen skulle sedan paras ihop och kancellera varandra tills alla fel var borta, vartefter ett nytt starttillstånd med nya fel skapades. Programmet fortsatte sedan iterativt på samma sätt för att lära sig hur felen kunde paras ihop på minst antal steg. Givet ett tillstånd som input använde agenten en $\epsilon$-greedy policy för att bestämma den bästa handlingen. Uppdateringen av tabellen skedde enligt SARSA-metoden, se avsnitt 2.4.1. Belöningarna definierades som

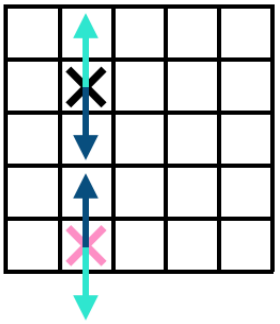
$$r = \begin{cases} 10, & \text{om två objekt förs samman,} \\ -1, & \text{då objektet rör sig.} \end{cases} \tag{C.3}$$

Användandet av Q-värdestabell kunde bli problematiskt när storleken på tillståndsrummet ökar. För att hantera ett större tillståndsrum och för att minska algoritmens tidskomplexitet fanns ett behov av att implementera ett neuralt nätverk istället för Q-värdestabellen.

# D

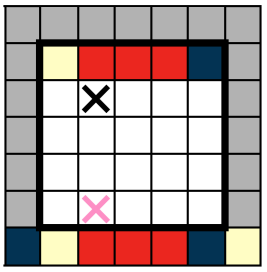
# Representationer av nätverkets input

Informationen som algoritmen baserar sina handlingar på kommer från tillståndsrepresentationen, som i detta projekt ser ut enligt figur 3.1. För att erhålla optimalt beteende av det neurala nätverket kan det dock vara motiverat att representera nätverkets input på ett annat sätt. För att illustrera varför betraktar vi figur D.1. Genom att utnyttja gittrets periodicitet kan felen elimineras med två operationer (ljus väg).



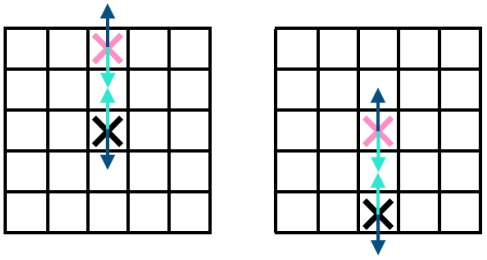
Figur D.1: *Ett tillstånd som felkorrigeras med minst antal steg genom att flytta ett fel längs den ljusa vägen.*

Det är dock problematiskt att applicera ett konvolutionellt nätverk vars input ser ut som i figur D.1. Det konvolutionella nätverket letar efter lokala korrelationer, men fel på varsin sida av matrisen kommer inte ses som lokalt närliggande, trots att de egentligen är nära varandra. Om nätverket skulle tränats med figur D.1 som input är det därför sannolikt att nätverket skulle para ihop felen på det icke-optimala sättet (mörk väg). En potentiell lösning på problemet skulle vara att klona några av de översta raderna och placera underst (samt göra ekvivalent för de andra kanterna). Figur D.2 illustrerar hur denna kloning skulle gå till.



Figur D.2: *En alternativ representation av tillståndet i figur D.1. Genom att klona rader från gittrets ena kant och placera dem på andra sidan av gittret kan periodiciteten modelleras. Den tjocka linjen representerar gittrets ursprungliga rand. Plaketterna utanför denna linje är de klonade plaketterna. Notera att plaketterna i hörnen av det ursprungliga gittret blir klonade till två platser. Figuren illustrerar endast hur kloningen för den översta raden går till.*

Representationen som visas i figur D.3 används som input till nätverket. Nätverkets input modifieras på så sätt att det fel som betraktas blir centrerat i matrisen (genom att utnyttja gittrets periodicitet). Nätverket utvärderar sedan förväntad belöning som funktion av vilken riktning det centrerade felet flyttas. Denna centrering utförs för alla fel och algoritmen väljer den av alla handlingar som ger högst Q-värde. Med andra ord delas tillståndsrepresentationen upp i lika många representationer som det finns fel. Detta innebär att nätverket kan användas oberoende av hur många fel som finns i matrisen. En ytterligare fördel representationen nedan för med sig är att tillståndsrummet minskas utan att nätverket tappar någon information. Detta är för att två förskjutna, men annars ekvivalenta tillstånd blir ekvivalenta under centreringsprocessen. Den förminskade storleken av tillståndsrummet borde minska inlärningstiden för nätverket. Nackdelen med den centrerade representationen är att den förutsätter att tillståndet har udda sidlängd.



Figur D.3: *Två alternativa representationer av tillståndet i figur D.1. Gittret har, med hjälp av sin periodicitet, förskjutits så att ett fel hamnar i mitten av gittret. Det bildas på så sätt lika många representationer som det finns fel i gittret. Med hjälp av denna representation kommer det kortaste avståndet från det centrerade felet till ett godtyckligt fel i gittret aldrig gå genom gittrets rand.*

# E

# Nätverksstrukturer

Detta avsnitt presenterar strukturerna hos de nätverk som användes i projektet. Förkortningen *FKL* betecknar ett fullt kopplat lager och *KL* betecknar ett konvolutionellt lager. Vid implementering av felkorrektionen så användes också dropout, *DO*. Nedan beskrivs nätverket för den kvantmekaniska felkorrektionen.

## E.1 Nätverk för kvantmekanisk felkorrektion

Storleken hos de nätverk som används för implementering av kvantmekanisk felkorrektion beror på inputstorleken $L \times L$. De fyra storlekarna som användes under simuleringen var 5, 7, 9 och 11. Den ena nätverkstypen är fullt kopplad och den andra är konvolutionell, se tabell E.1 respektive E.2. Randomisering vid *DO* innebär att en procentuell mängd av signalerna sätts till noll. Den kostnadsfunktion som används är genomsnittliga kvadratiska felet.

Tabell E.1: Det fullt kopplade nätverkets struktur för kvantmekanisk felkorrektion.

| | |
|---|---|
| FKL | Avbildningen är $L \times L \to L^2 \times L^2$ och aktiveringsfunktionen är *Relu*. |
| DO | Randomiseringen är 20 %. |
| FKL | Avbildningen är $L^2 \times L^2 \to L \times L$ och aktiveringsfunktionen är *Relu*. |
| DO | Randomiseringen är 20 %. |
| FKL | Avbildingen är $L \times L \to 4$, ingen aktiveringsfunktion används. |

Tabell E.2: Det konvolutionella nätverkets struktur för kvantmekanisk felkorrektion.

| | |
|---|---|
| KL | Avbildningen är $L \times L \to L - 2 \times L - 2 \times L^2$, med $L^2$ stycken $3 \times 3$-filter som stegar enligt $1 \times 1$ och aktiveringsfunktionen är *Relu*. |
| DO | Randomiseringen är 20 %. |
| FKL | Avbildningen är $L - 2 \times L - 2 \times L^2 \to L \times L \times L$ och aktiveringsfunktionen är *Relu*. |
| DO | Randomiseringen är 20 %. |
| FKL | Avbildingen är $L \times L \times L \to 4$, ingen aktiveringsfunktion används. |