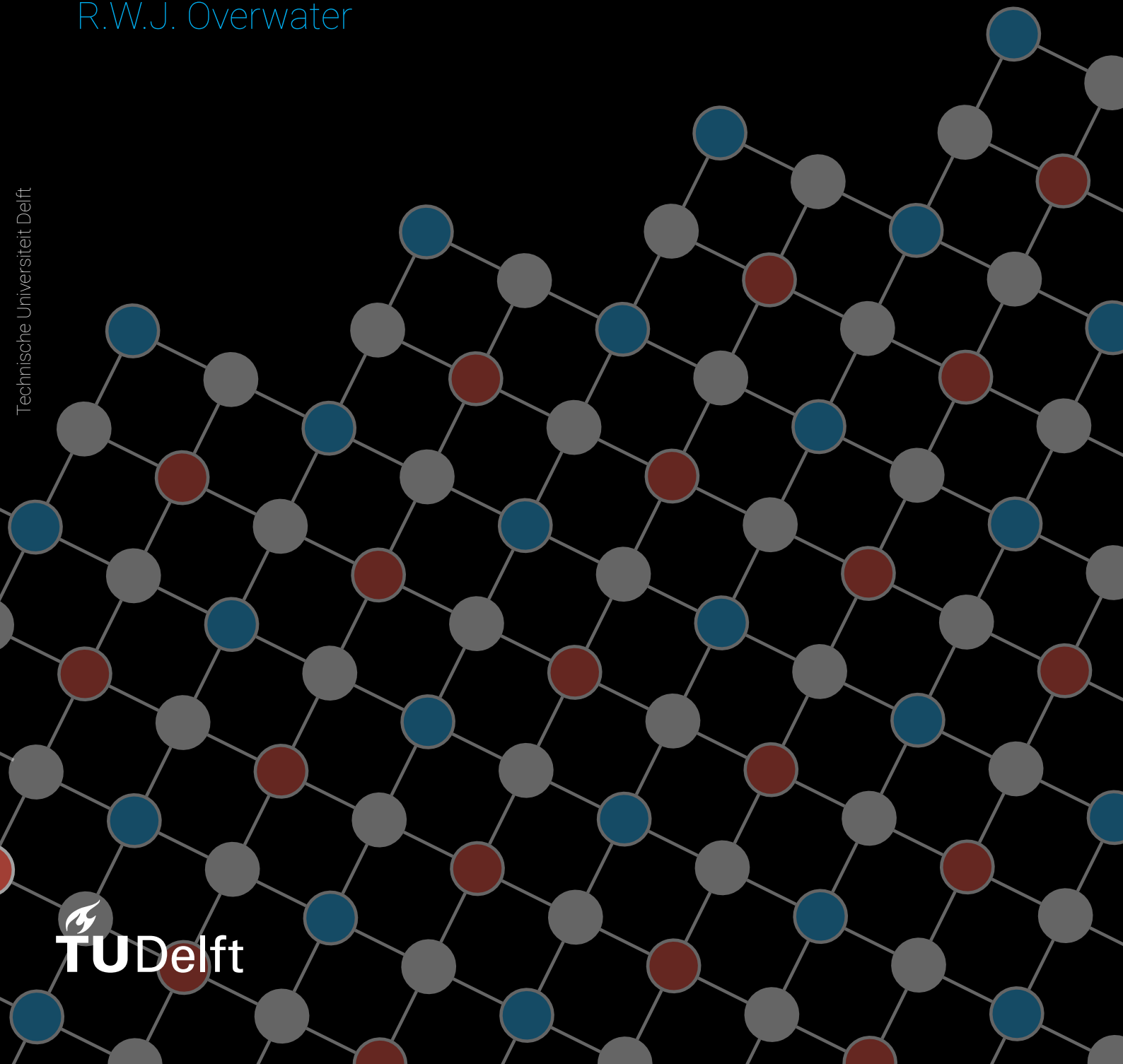
Cryogenic Hardware Considerations Of Neural Network Decoders For Quantum Error Correction Using Rotated Surface Codes

R.W.J. Overwater



Technische Universiteit Delft

TU Delft

Cryogenic Hardware Considerations Of Neural Network Decoders For Quantum Error Correction Using Rotated Surface Codes

by

R.W.J. Overwater

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 29th, 2019 at 10:00 AM.

Student number:	4305000	
Project duration:	September 1, 2018 – August 29, 2019	
Thesis committee:	Prof. dr. E. Charbon,	TU Delft
	Dr. F. Sebastiano,	TU Delft, supervisor
	Prof. dr. ir. K.L.M. Bertels	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.
Q&CE-AQUA-MS-2019-13

TU Delft

Abstract

The quantum bits (qubits) at the core of any quantum computers are so fragile that quantum error correction (QEC) schemes are needed to increase their robustness and enable fault-tolerant quantum algorithms. The surface code is one of the most popular QEC schemes, but it requires the availability of an efficient decoder. While neural networks have been shown to be well suited to this task, only software implementations have been studied in prior work. These have shown that neural network decoders can be on par or better than other decoding algorithms, but lack the required speed when running as software. The aim of this thesis is to investigate the hardware implementation of the neural networks for the decoders of surface codes to achieve the required speed. Most electronic hardware employed in quantum computers today operates at room temperature and is connected by bulky wires to the qubits, which are placed in a cryogenic chamber for proper operation. Since any useful quantum computer will comprise thousands or even millions of qubits, this work proposes to also move the QEC hardware to cryogenic temperatures (4 K). However, because at these temperatures the cooling power of cryogenic refrigerators is limited, the hardware needs to be low power, while ensuring enough speed to keep the pace of the QEC. The exploration of this work sweeps multiple parameters of a feed-forward neural network to find what the influence is on the decoder performance and the delay, the power, and the area. The parameters that are swept are the number of hidden layers, their sizes, the type of transfer functions and the number of bits used in the quantization of the weights and outputs. The results show that using the configurations found in this work it is possible to meet the performance and timing constraints using cryo-CMOS on both an FPGA and in a 40-nm technology. Although there are still some limitations in this work, such as the scaling in the power consumption and decoding performance for larger distances, there are multiple proposed improvements, making this is a stepping stone towards a future scalable implementation of QEC.

Acknowledgements

First of all I would like to thank my supervisor Dr. Fabio Sebastiano for giving me the opportunity to tackle this very interesting, challenging and exciting problem. After talking to him a year before this project started, I already got excited of some of the topics that his group (AQUA) and other groups in the Quantum and Computer Engineering (QCE) department worked on in this new and exciting field of Quantum Electronics. He took the time to talk with me and find a project that really excited me and would fit in my double degree MSc. curriculum. I would also like to thank him for having a lot of faith in me and supporting me in choosing a project in a topic that he himself also wanted to learn a lot in and for helping me getting supervision from other people who worked on the different aspects of my project. Finally I would like to thank him for the useful discussions we had figuring out in what direction we wanted this project to go with every big choice there was to make and teaching me how to prioritise in such a big project.

I would like to thank Dr. Savvas Varsamopoulos for helping me on my way when I started by project. He helped me by talking me about everything he found out when he did his PhD and was very enthusiastic in the discussions we had when I either had a question about something he had done, or when I suggested some different approach that I wanted to take.

I would also like to thank Edoardo Charbon and Koen Bertels, for being in the committee for my thesis defence. I would also like to thank Edoardo Charbon for giving me the opportunity to work in his group of Applied Quantum Electronics.
I also would like to thank all my colleagues in the Coolgroup and AQUA for all the technical discussions we had, but also all the other conversations and fun we had. You all made me feel at home right away.
I would especially like to thank Pinakin Padalia for all the conversations about our thoughts on life, work and food. For all the times we tried to explain our work to each other and learning something new about our own work with every discussion. But also for a new friendship.

I would like to thank the INSY group for letting me use the GPU's on their computing cluster to run the simulations.

I would like to thank Björn Minderman for sharing with me his experience on Pure Error Decoders.

I would also like to thank my roommates for putting my mind away from work when I was home.

I also would like to thank my parents and sisters for always making me feel at home when I went back and for supporting me in my life for the last couple of years.

Finally and most importantly I would like to thank my girlfriend Nina for supporting me during this least year. You always make sure I do not drown myself in my work at home and to take a break from time to time. I would also like to thank you for coping with me if I did suddenly realise something and dropped everything else to write it down.

R.W.J. Overwater
Delft, August 2019

Contents

1 Introduction

Quantum Computing

Quantum computers are being developed for their theoretical speed-up over classical computers. There are multiple known NP-hard problems, which would scale exponentially with classical hardware, that can scale polynomially in time with quantum computers [15][28]. This speed-up is possible because quantum computers use quantum bits (qubits) [30]. Contrary to classical bits, which can only be either in the zero or one state, quantum bits can be in any superposition state of the zero and one state and be entangled to each other [11]. However, due to the decoherence caused by interactions with the environment, some form of error correction is needed to increase the fidelity of the operations.

Surface Code

One popular form of quantum error correction is called the surface code [13], which is a planary form of the toric code [20]. This way to encode multiple physical qubits into one logical qubit is popular for several reasons. First, the surface code has a simple scalable 2D structure. Secondly, it only needs local interactions. And finally, the surface code has a very high error threshold. However, the measurement outputs of the surface code, called the error syndrome, need to be decoded, i.e. the output must be processed to determine whether any error on the physical qubits has happened.

Decoding

Many algorithms exist to perform decoding, but the approach used in this work is to use neural networks. This thesis continues on the work in [52][51][50]. The performance of these neural networks is as good, or better, compared to other commonly used algorithms, such as Blossom [9]. The neural networks have several advantages, such as the ability to learn to any data set, a constant execution time and highly parallelizable hardware solutions. However as stated in [52], the software implementation is too slow to cope with the short decoherence time of the qubits. Thus, in order to avoid creating a data backlog, a hardware solution is needed.

Hardware

The required hardware must be connected to the quantum processor. For an useful quantum algorithm, hundreds or thousands of logical qubits would be required at the least. Because each logical qubit requires at least 17 physical qubits, but probably also around 1000 qubits, more than a million interconnects are needed just for the decoding. In order to decrease the number of interconnects and increase the integration, the hardware can be put closer to the qubits. As the qubit technologies used in this work [22][40][54] operate near absolute zero (<< 1 K), the hardware needs to be cryogenic as well. Our group targets putting the hardware at 4 K [34]. This is a compromise, because it is a lot closer to the qubits than room-temperature (300 K) electronics, while it will still have a reasonable power budget of around 1 W, which is limited by the cooling power of existing refrigerators. The vision is that in the future the operating temperature of qubits and the available power will increase. This will allow both the electronics and the qubits to operate at the same temperature. This does give an additional constraint to the hardware, as it can only use some part of the available power of 1 W. This means that besides the need for a fast solution, a low power solution is needed as well.

Because of the need for a low-power, high-speed chip that still has a good decoding performance, an exploration of the design space of neural network hardware is needed. This study will explore the trade-offs between the decoding performance and the power, speed and area.

Challenges

The first challenge is the adjustment of the standard surface code cycle that uses Hadamard and CNOT gates to a surface code cycle that can be used in the qubit technologies used in this work. Using this technologies it is not possible to use these gates.

Next, is the choice of what neural network architecture will be used for a first estimate on the performance, delay, power and area trade-offs that this work tries to explore.

One of the biggest challenges of this work is the number of simulations that have to be performed in order to get a good overview of the design space. For this reason a complete simulation setup is written in C and CUDA to run the surface code simulations and the training of the neural networks on a NVIDIA GeForce GTX 1080 Ti GPU. With this simulator, the neural network will be trained with a lot of configurations. These will vary in the size of each layer, the transferfunctions and the number of bits used in the quantization of the weights, biases and layer outputs.

This requirement for a fast simulation speed also requires a pure error decoder that is more regular and easier to compute than the one used in previous work [52][51].

Finally, the power, speed and area are crudely estimated for both a Xilinx Artix 7, shown to work at 4 K [16] and the TSMC 40 nm technology. These estimates show that the hardware can be made fast enough to perform the surface code decoding, while still outperforming the commonly used Blossom decoder.

Thesis organization

This work starts with a literature study that explores the basics of quantum information in the second chapter. This chapter concludes with the need for quantum error correction. Next, the third chapter discusses the surface code cycle and what is needed from the decoder. This chapter also presents a surface code cycle that can be used with the transmons and single electron silicon spin qubits. The fourth chapter gives the background of neural networks and shows how they can be optimally used to decode the surface code. This is done by using a pure error decoder in parallel. The chapter concludes with a summary on which parameters will be varied in the design exploration of chapter 5. Chapter 5 shows the pseudo-threshold performance of the decoder while sweeping for different layer sizes, transfer functions and number of quantization bits. It finishes with a neural network decoder for the first 4 distances (3, 5, 7 and 9), that has a good trade-off between the decoding and hardware performance. These configurations are simulated in hardware in chapter 6, where it is shown that the decoders are fast enough, but can be optimized further. The report will be concluded with a conclusion and a discussion on future work in the final chapter.

2

Quantum Computing

Quantum computing promises a big speed-up over classical computing. Some NP-hard problems are shown to provide polynomial scaling in time, where classical hardware would scale exponentially [15]. Two well known quantum algorithms that have shown this speed-up are Shor's algorithm [42] and Grover's algorithm [14]. However, many more exist [28]. Some areas of application are in simulating quantum systems, simulating molecules for chemistry or proteins for medicine. Other examples are the factoring of large numbers, searching in big datasets and providing more secure ways of encrypting data with quantum cryptography.

This chapter discusses the basics of quantum information theory. This includes what qubits are, how they work,their limitations and the use of error correction to mitigate such limitations to some degree. For more information and background the reader is referred to [30].

2.1. Quantum Information

2.1.1. Single Qubit States

In classical computers the information is stored in bits. One bit can either be in the 0 state or the 1 state. These states can be defined in multiple ways. However, to get the largest separation between the states, they are usually defined to be at the supply rails. The 0 state is at V_{ss} and the 1 state at V_{dd}. For quantum bits (qubits for short), there are also multiple ways to assign the states. However, on the contrary to classical bits, a qubit can be in either the $|0\rangle$, the $|1\rangle$ state or a superposition of the two.

Superposition is one of the properties that makes qubits different from classical bits. The superposition state $|\psi\rangle$ of the qubit is shown in equation 2.1. This equation shows that the qubit state $|\psi\rangle$ can be any linear combination of the computational states $|0\rangle$ and $|1\rangle$, where the coefficients α and β are complex numbers. These coefficients are called the probability amplitudes of the states. As the names suggest the probability amplitudes give some information about the probability of the state being in either the $|0\rangle$ or the $|1\rangle$ state. If one were to measure the state shown in the z-base ($|0\rangle$ and $|1\rangle$), than measuring along the corresponding z-axis, the probability of finding it in the $|0\rangle$ and $|1\rangle$ state is given by the square of their probability amplitudes. This means $|\alpha|^2$ and $|\beta|^2$ respectively. As is usual with probabilities, the total probability is set to 1. This means that measuring the state will always give either $|0\rangle$ or $|1\rangle$. This normalization constraint is shown in equation 2.2.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.1}$$

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

One possible representation of the state is called the Bloch sphere. This maps the state onto spherical coordinates θ and ϕ as shown in equation 2.3. A picture of the Bloch sphere is shown in figure 2.1.

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{2.3}$$

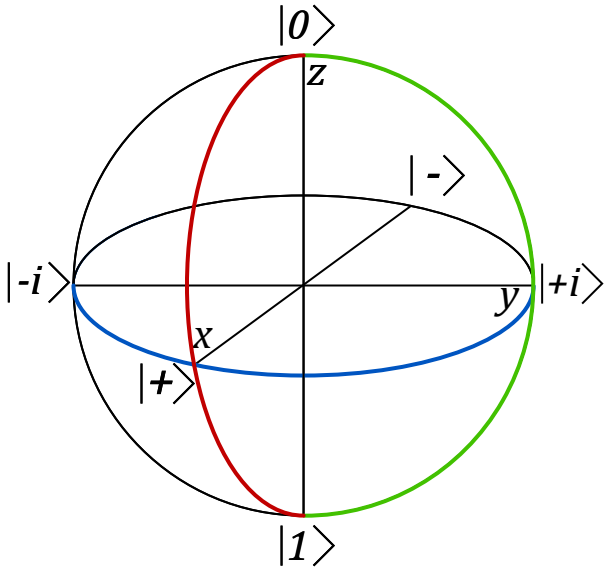


Figure 2.1: Image of the Bloch sphere indicating the $|0\rangle$ and $|1\rangle$ state along the z-axis, the $|+\rangle$ and $|-\rangle$ state along the x-axis and the $|+i\rangle$ and $|-i\rangle$ state along the y-axis. The X, Y and Z Pauli operators are shown in green, red and blue respectively.

This shows that the states along the x-axis are $|+\rangle$ and $|-\rangle$ and along the y-axis $|+i\rangle$ and $|-i\rangle$. This is shown below:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$
$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$
$$|+i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$$
$$|-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$$

One can check that for all of these states the normalization constraint in equation 2.2 holds.

2.1.2. Multi Qubit States

Just as in classical computing the state of two bits in the 0 state can be written as 00, the state of two qubits in the $|0\rangle$ can be written as $|00\rangle$. However, because both qubits can be in a superposition state, the total of the two qubits should be a combination of these two superposition states. This combination is calculated by taking the tensor product between the two individual qubit states as shown in equation 2.4.

$$|\psi_1\rangle \otimes |\psi_2\rangle = \alpha_1\alpha_2|00\rangle + \alpha_1\beta_2|01\rangle + \beta_1\alpha_2|10\rangle + \beta_1\beta_2|11\rangle \tag{2.4}$$

2.1.3. Entanglement

Sometimes the combined state of two qubits cannot be expressed as the tensor product of two individual states. This is because the two states are correlated which each other. In quantum mechanics this is called entanglement. Entanglement is the second property that makes quantum computers more powerful than classical computers.

2.1.4. Destructive Measurements and the No-Cloning Theorem

One big difficulty with the read-out of qubits is destructive measurements. If a qubit is measured, the superposition of the quantum state collapses. If for example a qubit was measured along the z-axis, the state would randomly collapse to either the $|0\rangle$ or the $|1\rangle$ state. As stated before, the chances of collapsing to either the $|0\rangle$ or the $|1\rangle$ state depends on the square of the coefficients, $|\alpha|^2$ and $|\beta|^2$.

Another difficulty with using qubits is that the (quantum) information cannot be copied or destroyed [33][55]. These are called the no-cloning and the no-deleting theorems respectively. This will become a problem later on as will be discussed when looking at quantum error correction.

2.1.5. Single qubit gates

The gates of the quantum bits are described by unitary operators. Unitary matrices U are matrices where the complex conjugate $U^\dagger$ is its own inverse: $U^{-1} = U^\dagger$. These single-qubit unitary operators describe rotations around the Bloch sphere and thus are reversible and order dependent. This also holds for multi-qubit operators, but they describe rotations in a multi-dimensional Hilbert space. For ease of calculation, it is nice to express the quantum states as vectors and the operators as unitary matrices. Equation 2.5, shows the state of a single qubit represented as a vector on the left and the tensor-product of the states as a vector on the right.

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \qquad |\psi_1\psi_2\rangle = \begin{bmatrix} \alpha_1\alpha_2 \\ \alpha_1\beta_2 \\ \beta_1\alpha_2 \\ \beta_1\beta_2 \end{bmatrix} \tag{2.5}$$

The four Pauli matrices are shown in equation 2.6. These also act as single qubit Pauli gates and rotate the state 180° around the corresponding axis. For example, the X-gate rotates the state 180° around the x-axes. The X, Y and Z-gates are shown in the Bloch sphere picture 2.1.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & \text{-}1 \end{bmatrix} \tag{2.6}$$

One can check that for any Pauli gate G that applying the gate twice results in the identity matrix, i.e. $G^2 = I$. The Hadamard gate, defined in equation 2.7, is used to bring a state from the z-axis ($|0\rangle$ and $|1\rangle$) into the xy-plane and vice versa. Applying the Hadamard gate twice, also results in the identity operation, i.e. $H^2 = I$.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & \text{-}1 \end{bmatrix} \tag{2.7}$$

$$R_v(\theta) = \cos\left(\frac{\theta}{2}\right) \cdot I - i \cdot \sin\left(\frac{\theta}{2}\right)\left(v_x X + v_z Z + v_y Y\right) \tag{2.8}$$

In principle any qubit can be rotated around any axis (vector) v by any arbitrary angle θ around the axis defined by the vector $v = [v_x v_y v_z]$ as described by equation 2.8.

2.1.6. Multi qubit gates

There are also multi qubit gates, such as the Controlled Not (CNOT) and Controlled Phase (CPHASE) as shown in equation 2.9. Both gates use one qubit as the control qubit and the other one as the target. In case of the CNOT gate a bit flip (a Pauli X operation) will be performed on the target depending on the state of the control qubit. In case of the CPHASE this is a phase flip (Z). Since the Hadamard changes the state between the Z and the X axis, this can also be used to change a CNOT into a CPHASE and vice versa by surrounding the target qubit by two Hadamard gates. This is shown visually in figure 2.2.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad CPHASE = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \text{-}1 \end{bmatrix} \tag{2.9}$$

2.1.7. Universal Gate Sets

Any set of gates that can create any arbitrary quantum operation is called a universal quantum gate set. This is analogous to classical logic, in which we know that with a classical NAND gate all possible logic operations can be synthesized. One such set is the set consisting of the Hadamard gate H, the CNOT and the T gate, which is a quarter Z rotation. However, not every physical implementation of qubits can perform these gates, thus other alternatives must be used. Furthermore, this is the minimum required set, but it can be

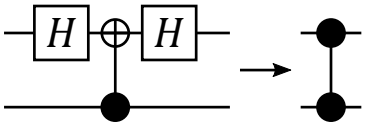


Figure 2.2: Picture showing the way to create a CPHASE gate by surrounding the target of the CNOT gate by two Hadamard gates.

beneficial to use more gates if they are available, so as to reduce computation time, complexity and errors.

An important subset of gates is the Clifford group. These are the gates that when used to surround a Pauli gate P will result in another Pauli gate P' as shown in equation 2.10.

$$P' = CPC^{\dagger} \tag{2.10}$$

Another interpretation of the last equation is that applying any gate from the Clifford group to a qubit in one of the states $|0\rangle$, $|1\rangle$, $|+\rangle$, $|-\rangle$, $|+i\rangle$ and $|-i\rangle$ will return result in another of these states. This can be visualised when looking at the Bloch sphere in figure 2.1. Any 90° rotation around any of the x, y, z-axis will be part of this set. Since the T gate is a non-Clifford gate, the minimal universal Clifford set is H, $CNOT$ and $S = T^2$.

2.1.8. Quantum circuits

Just as with classical digital circuits, it is often much easier to display a quantum algorithm graphically, rather than through equations, especially when multiple qubits are used. Since in quantum mechanics no data can be created or destroyed, all qubits available at the beginning of the algorithm will be available until the end, and hence they are represented by horizontal lines. This is different from classical circuits where most gates have more inputs than outputs, e.g. the OR, NOR, AND, NAND and XOR gates all having two inputs and one output.
The horizontal lines illustrate time moving forward to the right. Gates and operators are then drawn on the lines in the correct order, connecting multiple lines for multi qubit gates. This is illustrated in figure 2.3. Some common gates and the ones used in this thesis are shown in figure 2.4.

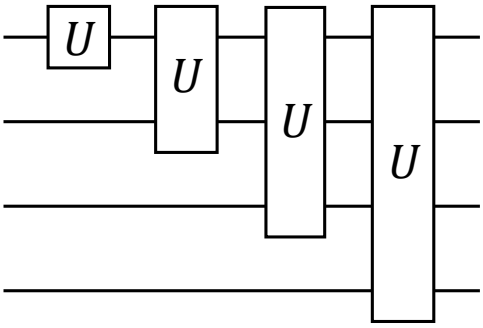


Figure 2.3: Picture of a single qubit and 2, 3 and 4 multibit unitary operators. The four horizontal lines describe the time paths of four different qubits where the forward movement of time is from left to right.

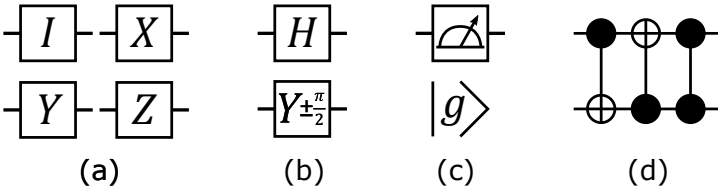


Figure 2.4: Picture with common operators in quantum circuits. (a) Four Pauli Gates. (b) Top: Hadamard gate, bottom: $\pm\pi/2$ rotation around the Y axis. (c) Top: Measurement of qubit (default along the z-axis), bottom: initialization to the ground state. (d) Left to right: $CNOT$ with target on the bottom, $CNOT$ with target on the top and finally the $CPHASE$.

2.2. Qubit technologies

In 2000, David P. DiVincenzo [8] proposed a list of five criteria needed to make a functional quantum computer. The DiVincenzo's criteria are:

1. Having a scalable system of well-characterized qubits.
2. Being able to initialize the qubit to a known state.
3. Having coherence times long enough to perform computation.
4. Being able to manipulate the state of the qubits with a universal set of gates.
5. Having a way to measure the state of the qubit.

There is a long list of technologies that are promising for the realization of a quantum computer. Some popular ones are ion traps [7][27], superconducting qubits, such as transmons [22][40], spin qubits, such as single electrons in silicon quantum dots [54] and nitrogen vacancies in diamonds [6]. This work will focus on two specific technologies: transmons and single-electron spin qubits in silicon. Both those technologies are based on solid-state fabrication and require only purely electrical control, thus being considered among the most promising for delivering a scalable qubit platform (DiVincenzo's criterium # 1). Furthermore, those two technologies also show a good promise for scaling to a 2D structure. This 2D structure is needed for the surface code, which is the quantum error correction scheme chosen in this work (discussed in the next chapter). Looking at the other DiVincenzo's criteria, both transmons and spin qubits have a limited coherence time due to interaction with the environment. This causes problems like decoherence, dephasing and depolarization. Additionally, the signals applied to the qubits to perform the gates are not perfect and some errors are introduced when applying gates [49], thus resulting in non-ideal gate fidelity.

Single electron silicon spin qubits have a decoherence time T_2^* of 120 μs on their own, which can be upped to 1.2 ms and 28 ms with special pulses [54]. These qubits also have a single gate time of 1 μs and a two qubit gate time of 0.1 μs [48]. Assuming an equal occurrence of both single and two qubit gates, only $\approx$ 51000 gates can be used. For transmons with a T_2^* of 90 μs [38] and gate times of 20 and 40 ns for single and two qubit gates respectively [48], only 3000 gates can be used. However, because the T_2^* means the time to complete decoherence, most gates need to be performed in a fraction of this time, resulting in even less possible gates. For running an algorithm like Shor's algorithm [42], which uses relatively few gates per qubit compared to other algorithms, on a state-of-the-art number of qubits is already hard. The records is the factorization of 21 [26] into its prime factors. So the technology is not good enough yet to run such an algorithm for any practical purposes.

2.3. Quantum Error Correction

As seen in the previous section, the coherence time of qubits is not long enough to perform the amount of gates needed in most algorithms [28]. Due to decoherence, a lot of errors accumulate in time while we still need to perform operations on the qubits. This scenario is similar to errors (noise, interference, ...) in signals that cause the bit error rate (BER) in classical electronics. Current digital chips, however, do not suffer significantly from errors, but digital communication systems do. In that case, some error correction schemes are used to encode the information in a single bit to be transmitted into multiple bits. A decoder at the receiver end retrieves the information correctly, even in the presence of errors, thanks to the redundancy introduced by the error correction scheme. This can be done, for example, by duplicating the bits and doing a majority vote at the receiver.

The problem with extending this scheme to qubits are the destructive measurements and the no-cloning theorem. The no-cloning theorem states that we can not clone the state of a qubit to other qubits, thus making it impossible to simply copy qubits to exploit redundancy for error robustness, while the destructive measurements make it impossible to measure the qubits without destroying their quantum state. Thus a classical scheme would not be able to copy the data, which would be allowed to be measured destructively, and we cannot simply perform destructive measurements on the data as this would destroy the quantum state of the logical bit. Fortunately, there are alternatives which mimic this simple idea of duplication and correction for quantum computing. Extra ancillary qubits are inserted and entangled to the qubits containing the relevant

data. If done correctly, when measuring these ancillas, the continuous errors on the data qubits collapse to discrete errors without destroying the logical quantum state. These discrete errors can then be corrected on the logical state. This is done for example in bit-flip codes, sign/phase-flip codes and the combination of them in Shor-codes [43]. There is a whole range of different quantum error correction (QEC) schemes developed in the last few decades. All use some encoding scheme to encode the state of some physical qubits into one logical qubit state and some decoding algorithm to measure the physical qubits to correct the logical state.

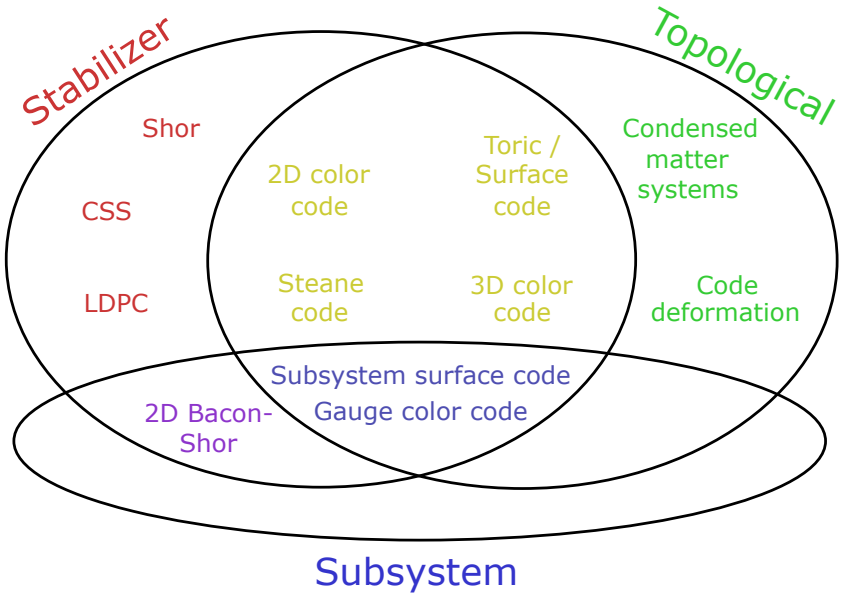


Figure 2.5: Image of common QEC codes in three categories.[50]

In [50, p.16], those schemes are classified into stabilizer codes, topological codes and subsystem codes. Figure 2.5 shows this classification for the more common QEC schemes.

Stabilizer codes are codes that use ancillary qubits (ancillas) to extract information from the data qubits by using parity checks. Then the ancillas can be measured to give discretized error information about the data qubits without the need to measure the data qubits.
Topological codes use non-trivial topologies to use the location of the qubits and often only require local interactions between qubits.
Finally, the subsystem codes use additional logical qubit called gauge qubits which do not encode information, but make the measurements on the other qubits easier.

The error correction scheme chosen in this work, called the Surface Code [13], based on the Toric Code [20][21], is both a stabilizer and topological code, thus inheriting the advantages from both families. The next chapter will discuss why this code was chosen and how it works.

3

Surface Code

This chapter will explain the choice behind using the surface code for this work. It will also give an overview on how the surface code works and what it looks like. Afterwards, the theoretical measurement cycle of the surface code is described, followed by the changes required to allow its use with transmons and single electron silicon spin qubits. Next, the concepts of error chains, logical operators and stabilizers are introduced. It is followed by the discussion about the need for correction or detection. And finally, the error models and decoders are discussed.

3.1. Structure

As stated at the end of the last chapter, the surface code is both a topological and a stabilizer code. This gives the surface code some advantages, such as a simple scalable 2D structure, the need for only local interactions and a very high error threshold. The error threshold is the amount of errors allowed on the physical qubits to still have a reliable correction. Figure 3.1 shows the 4 smallest rotated surface codes. A rotated surface code is rotated by 45° compared to the one described in [13]. For the interested reader, the computation using multiple logical qubits, each being their own surface code, can be found in that work. Here is also discussed how the surface code can be initialized.
Figure 3.1 shows a d by d grid of white dots representing the data qubits. In between these data qubits are two types of ancilla qubits. The blue dots represent the X-ancillas. These are the ancillas that are measured along the x-axis. The red dots represent the Z-ancillas, measured along the z-axis. As will be explained in the next section, the X-ancillas detect Z-errors on the data qubits and the Z-ancillas detect X-errors.
d is called the distance of the surface code and determines its size. The distance is also the minimum amount of errors that need to occur to change the logical state. However, as discussed later, the maximum amount of errors allowed by the optimal decoder to still be able to correct the logical state is $(d-1)/2$.

For a given distance, the surface code has d^2 data qubits and a total of d^2-1 ancillas. $\frac{d^2-1}{2}$ of each type. Table 3.1 shows the amount of ancillas and data qubits for the distances shown in figure 3.1.

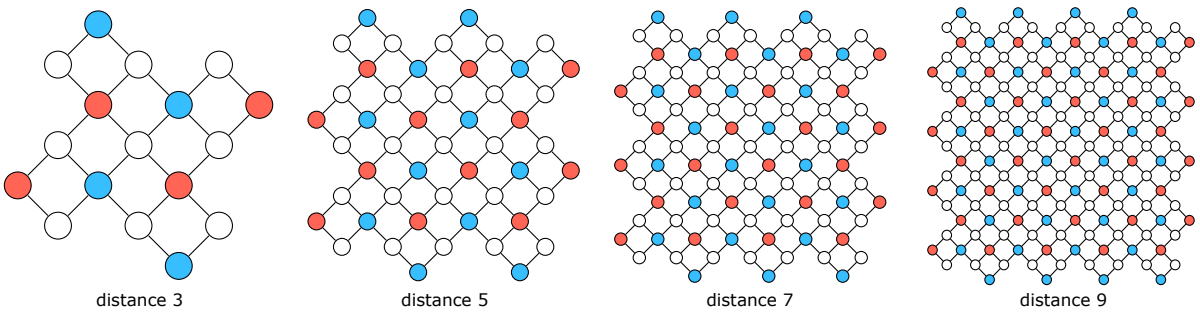


Figure 3.1: Image of the first four distances of the rotated surface code. Blue dots are X-ancillas, the red dots Z-ancillas and the white dots are the data qubits

Surface Code Distance	Number of Data qubits	Number of Ancillas	Total Number of qubits
3	9	8	17
5	25	24	49
7	49	48	97
9	81	80	161

Table 3.1: Number of data and ancilla qubits for the first 4 distances of the surface code.

3.2. Measurement Cycle

One error correction cycle (also called a measurement cycle or a surface code cycle) on the surface code is done in a specific order. This order is as follows:

1. Initialize the ancilla qubits to the ground state.
2. Move the X-ancillas to the x-axis of the Bloch sphere by performing Hadamard gates.
3. Perform 4 CNOT operations between the ancilla qubits and their adjacent data qubits, the so-called CNOT dance (see next section).
4. Move the X-ancillas back to the z-axis.
5. Measure the ancillas in the z-axis.
6. Correct the logical state depending on the decoded measurement outcomes.
7. Repeat step 1 to 6.

This sequence is illustrated as a quantum circuit on the left hand side of figure 3.4. One can check that doing this needs qubits that satisfy all of the DiVincenco Criteria. This is assuming that a large surface code is used with well behaved qubits that have a coherence time long enough to perform multiple measurement cycles.

After each measurement cycle, the X-ancilla qubits will have a measurement output which represents the parity of the amount of Z-errors on the four adjacent data qubits. In case of an even amount of errors, the output will be $|0\rangle$, in case of an odd amount it will be $|1\rangle$. This holds in an analog way for the Z-ancilla qubits, for which the measurement outcome depends on the parity of the X-errors of the adjacent qubits.

At the boundaries, some ancillas have only two adjacent qubits. The cycle can then be made complete by assuming two non-erroneous data qubits at the missing places.

3.2.1. Decoding

Step six says some decoding is required on the measurement outcomes. This decoding will be explained in more depth in section 3.5. In short it means finding the data qubit errors from the parity measurement outcomes.

3.2.2. CNOT Dance

To make the measurement cycle work, none of the data qubits can be used in the CNOT operation by two different ancillas at the same time. To make sure this never happens, the CNOTs are performed in a specific order called the CNOT dance. The CNOT dance is shown in figure 3.2. This figure shows the order in which the four adjacent data qubits are addressed by both the X and the Z-ancillas. The orientation is similar to the surface codes shown before (figure 3.1). This is the order of the CNOT dance that is used in this work. Another order could also work, as long as it is the same for all ancilla qubits.

3.2.3. Available Gates

A big prerequisite for performing the measurement cycle using Hadamards and CNOTs is that these gates need to be available in the adopted qubit technology. Since this work focuses on transmons and silicon spin qubits, this is not the case, since both the Hadamard and the CNOT gates are not directly available. One can however create a CNOT from a CPHASE and two Hadamards as discussed in the previous chapter and shown

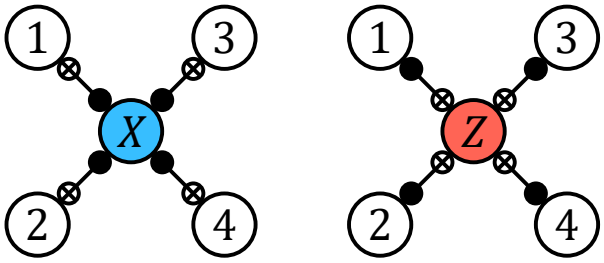


Figure 3.2: Illustration of the CNOT dance. The number on each qubit indicates the order in which the four adjacent data qubits are addressed by both the X-ancilla (blue on the left) and the Z-ancilla (red on the right). This picture could be overlayed on each ancilla qubit of the rotated surface code as shown in figure 3.1.

in figure 2.2. However, this introduces even more Hadamard gates. One gate which is similar to the Hadamard gate is the $R_y\left(\frac{\pi}{2}\right)$ gate. This quarter rotation around the y-axis will also bring states on the z-axis to the x-axis and vice versa. The $R_y\left(\frac{\pi}{2}\right)$ gate is related to the Hadamard gate as shown in equation 3.1 and figure 3.3. For simplicity, the $R_y\left(\frac{\pi}{2}\right)$ is drawn as [+] as done in figure 3.3 and the inverse rotation $R_y\left(-\frac{\pi}{2}\right)$ as [−].

$$H = X \cdot R_y\left(\frac{\pi}{2}\right) \tag{3.1}$$

$$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \tag{3.2}$$

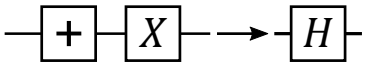


Figure 3.3: Circuit illustration of how the $R_y\left(\frac{\pi}{2}\right)$ is related to the Hadamard gate.

Using these found relations between the gates, the surface code can be modified as shown in figure 3.4 using $R_y\left(\pm\frac{\pi}{2}\right)$ and CPHASE gates. This figure extends on the one shown in [31, fig. 6(b)]. However, contrary to the original figure which only shows the case for Z-ancillas. This figure also shows the surface code adjust to be correct for both the Z and the X-ancillas. This includes having non-overlapping operations on the data qubits as discussed in the section about the CNOT dance.

3.3. Operators and Errors

Once the first surface code cycle is performed, some ancillas will give a measurement result of $|0\rangle$, some will give $|1\rangle$. This is the initial encoded logical state of the surface code. The measurement results do not necessarily signal that an error has occurred. We can say that if this is the initial logical state, then any change in the measurement results means an error has occurred. And thus, the error is the XOR between the initial state and the last cycle state. The pattern of ancillas that is triggered after one measurement cycle is called the syndrome.

A state which is obtained after performing all of the stabilizer measurements is called a quiescent state. It is important to notice that this state is not necessarily the ground state for the data qubits.

3.3.1. Error Chains

If a quiescent state is left without error correction, over time the physical qubits will start interacting with the environment. This will result in decoherence, dephasing and depolarization. If this is followed by another stabilizer measurement round, the stabilizer measurements will discretize the errors on the data qubits. This will result in each data qubit obtaining an additional X, Z or Y error. The probability of these errors occurring depends on the deviation from the state of the last measurement cycle.

As stated before, if a Z-error occurs on a data qubit, then the adjacent X-ancilla qubits will switch the parity of their measurement outcome. This holds similarly for Z-ancillas and X-errors. Figure 3.5(a) shows this.

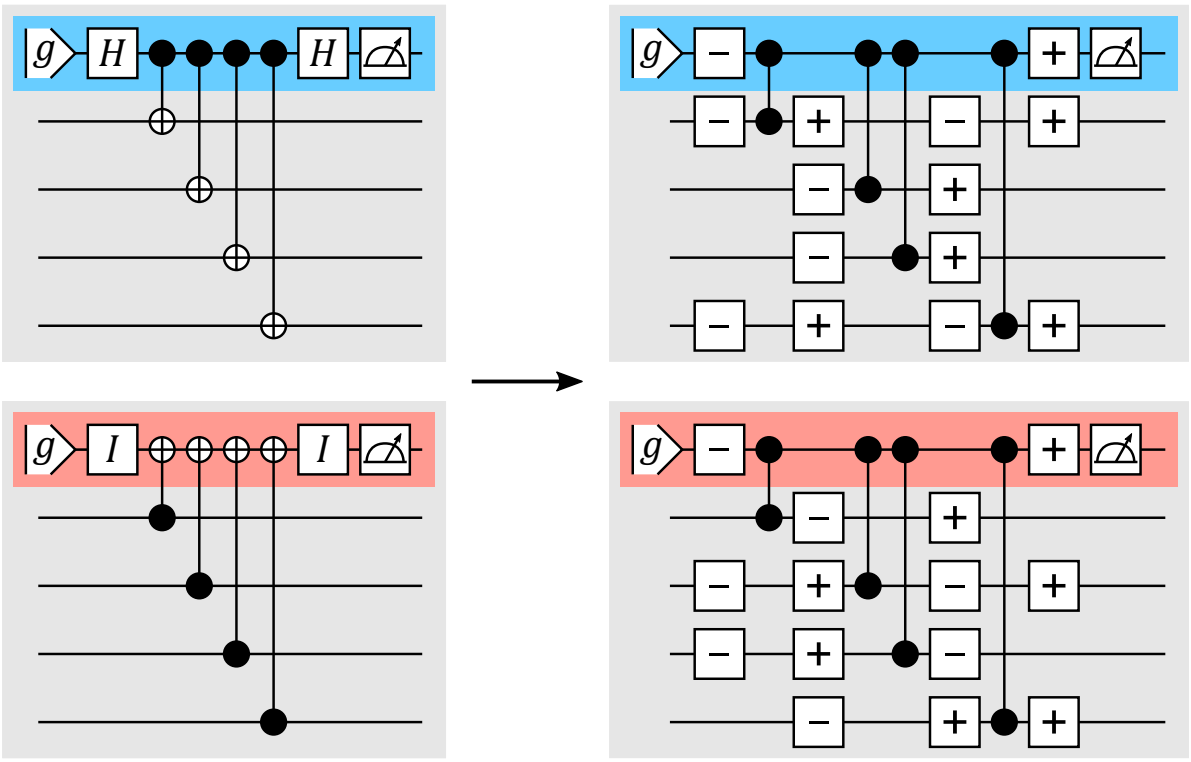


Figure 3.4: (left) Circuits for the commonly used surface code cycle with Hadamard and CNOT gates. (right) Adjusted circuit with CPHASE and $R_y\left(\pm\frac{\pi}{2}\right)$ on the right. In both figures the top circuit shows the circuits for the X-ancillas and the bottom circuit for the Z-ancillas. The colored qubits are the ancillas. The other gray qubits are the data qubits surrounding this ancilla. Top to bottom this is the same order as the 1 to 4 shown in figure 3.2 in the CNOT dance.

If data qubit 17 has a Z-error, then the adjacent X-ancillas 4 and 8 will give an error after the measurement cycle.

An error can be corrected by applying the gate of the same type as the error that occurred. For example if an X-error occurred on one of the data qubits, this could be corrected by applying an X-gate. This means that correcting the state of the logical qubit encoded in the surface code means finding the errors on the data qubits and applying the respective gates. In this way, errors can be thought of as gates that happened by accident. This is only possible because the single qubit gates on data qubits can be performed without influencing other data qubits. In other words, two single qubit operators on two different data qubits commute which each other.

Since errors can be seen as gates and gates can be represented as matrices operating on a Hilbert space with as much dimensions as there are qubits. We can describe multiple errors or gates as multiplying operators together. Thus an error can be given by a product of commuting operators and the correction is also a product of commuting operators. Applying the correction is than the same as multiplying these products together.

If more errors are adjacent, so called error chains are formed. See figure 3.5(b) for an example with two errors. Due to the fact that here X-ancilla 4 has two neighbouring errors, it will not trigger and only the endpoints of the chain (2 and 8) will trigger upon measurement. This can be thought of as the definition of an error chain. A set of errors of the same type on the data qubits, forming a path that only triggers ancillas at both ends.

Following this definition, figure 3.5(c) does not form one error chain (data 12, 16 17), since ancilla 4 and 8 also trigger. This means it is comprised of two chains. One of 16 and 17 and one of 12 as shown in figure 3.5(d). This is the way error chains are used in some decoding algorithms, for example in Edmonds' Minimum Weight Perfect Matching (MWPM) algorithm [9] discussed at the end of this chapter. This is a popular decoding algorithm, which finds the data errors by finding the shortest error chains between pairs of ancilla errors.

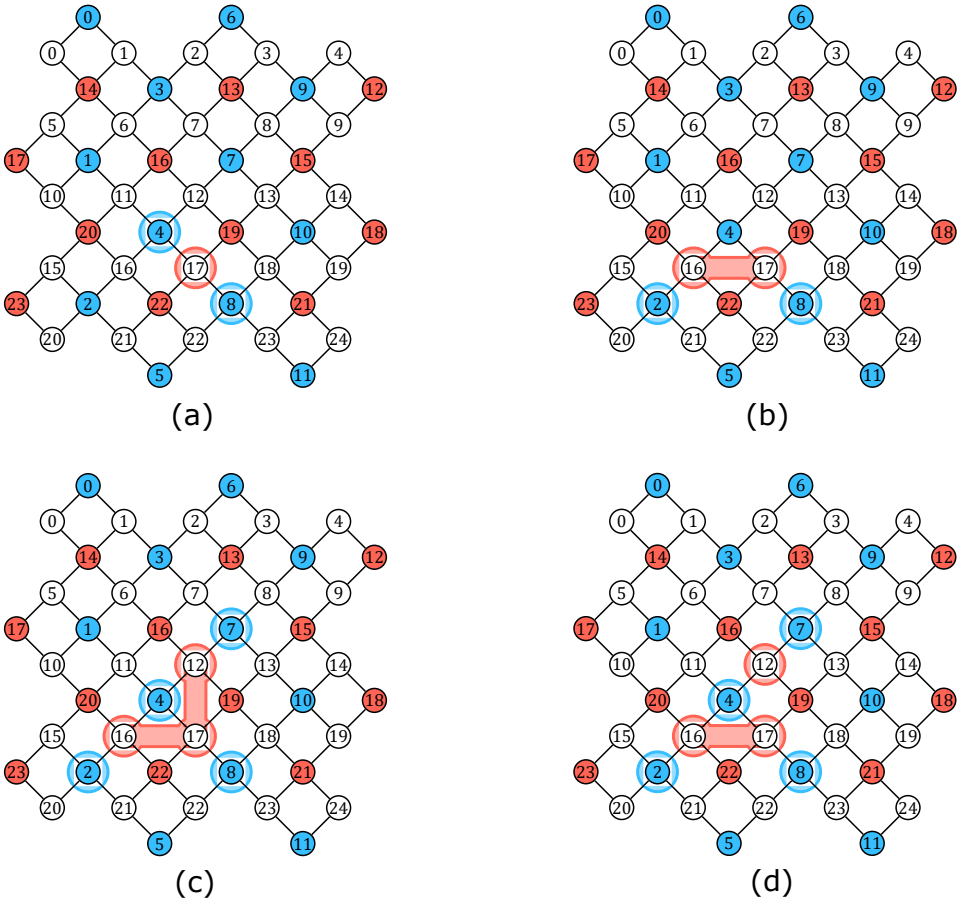


Figure 3.5: (a) A single Z-error on data qubit 17, triggering ancillas 4 and 8. (b) An error chain of two Z-errors on data qubits 16 and 17, only triggering ancillas 2 and 8 at the ends of the chain. (c) Error chain that would trigger 4 ancillas. (d) The error chain of (c) split into two chains, triggering two ancillas each.

3.3.2. Stabilizers

If all four data qubits around an ancilla qubit have an error and this error is of the same type as the ancilla, then it is called a stabilizer. Adding or removing a stabilizer will never change the syndrome. This is illustrated in figure 3.6(a). On top there is an X-stabilizer around ancilla 3 and a Z-stabilizer around ancilla 15. There are also two X-stabilizers around 4 and 2. Because both stabilizers require an error on data qubit 16, this double X-error means there is an identity error (or no error) on this data qubit.

There are of course also some boundary ancillas that only have two adjacent data qubits. This means that in this case the stabilizer is not a ring of errors on four data qubits, but just errors on the two adjacent data qubits.

An important consequence of stabilizers is that there are multiple data qubit configurations that give the same error syndrome on the ancillas as long as one can be obtained from the other by multiplying it with a product of stabilizers. Also, as discussed in the next section, qubit configurations that differ by a product of logical operators have the same error syndrome as well. This makes it harder to decode, because the decoder tries to find a data qubit configuration given the ancilla error syndrome. If multiple data configurations are possible, then the decoder has to guess the most likely configuration.

3.3.3. Logical Operators

One can imagine that, if data qubit 15 would give a Z-error in figure 3.5(b), only X-ancilla 8 would give an error. However, if data qubit 21 would have an error, both X-ancilla 5 and 8 would trigger. This gives a clue that for Z-error chains the left and the right boundary are different than the upper and the lower boundary.

Figure 3.6(b) shows an X-error chain from the top to the bottom. It is clear that this triggers no ancilla error

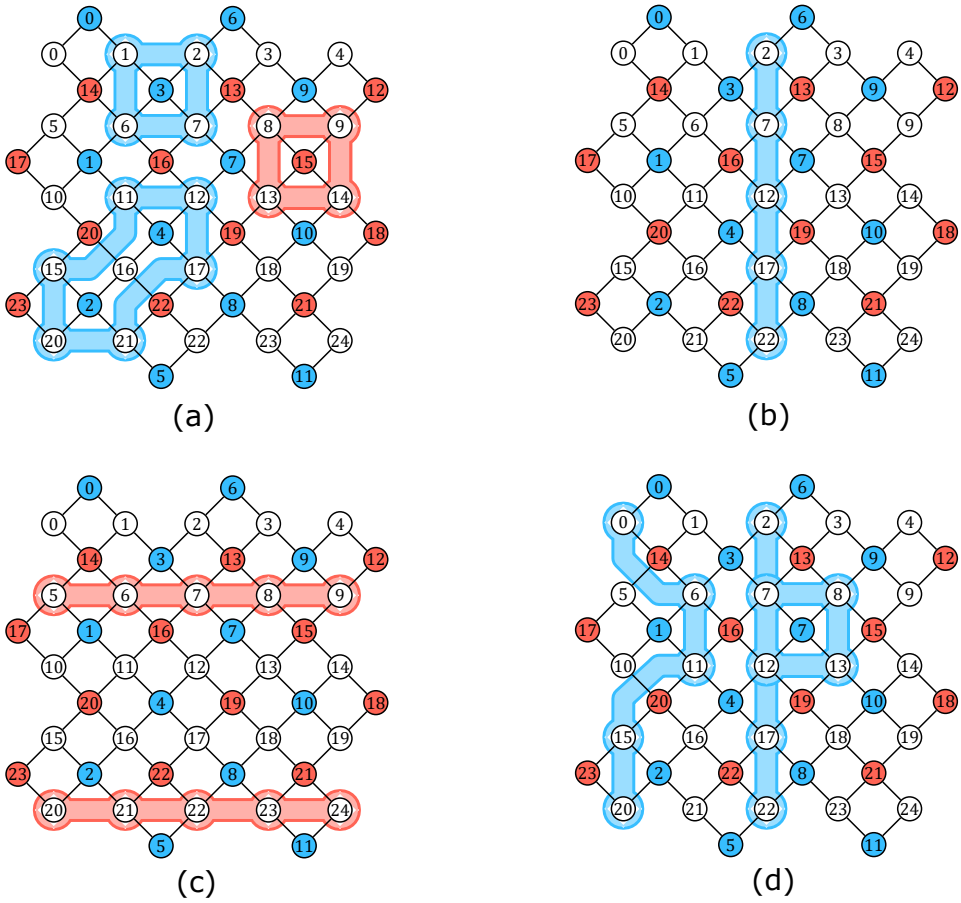


Figure 3.6: (a) Four stabilizers. One Z-stabilizers around Z-ancilla 15, one X-stabilizer around X-ancilla 2 and two overlapping stabilizers on X-ancillas 2 and 4. Due to the overlap the double X-errors on data qubit 16 cancel. Note that no ancillas are triggered. (b) A logical X-operator between the top and the bottom. This cannot be expressed as a product of stabilizers. (c) Two logical Z-operators that cancel to form a logical I-operator, thus possible to express as a product of stabilizers. (d) The logical operator of (b) multiplied by a stabilizer around X-ancilla 7, giving the bend line on the left. This is still a valid logical operator.

syndrome. This could mean that nothing changed the state and that it might be possible to obtain this chain from a product of stabilizers. However, one can check that this chain can not be produced by multiplying stabilizers together. A chain like this is called the X-logical operator for this surface code. This means that making a chain from the top to the bottom will change the logical state of the surface code.

It is not necessary that the chain is in the middle or in a straight line. Figure 3.6(d) shows the chain in (b) multiplied by one stabilizer around X-ancilla 7. Due to the double X-errors on data qubits 7 and 12, these errors will cancel to identity. This will result in a chain with a similar shape as the one drawn between 0 and 20 on the left. This chain will still not trigger ancillas and will still be a logical operator.

In figure 3.6(c), two Z-error chains are drawn between the left and the right boundary. As the vertical position of the logical Z-operator on the surface code does not matter as long as it connects the left and right boundary, this configuration would mean two logical Z-operators are performed on the surface code. Similar to physical operators, the two logical Z-operators will multiply together to a logical identity operator. This means that no ancillas will trigger and the logical state of the qubit does not change. This means that these two logical operators can also be created by multiplying stabilizers together. One can check that it is indeed the case that this can be represented as a product of Z-stabilizers.

3.4. Error Models

Simulating the whole state of the surface code would be impossible as it would require to keep track of the superposition and the entanglement of a lot of qubits. For a distance 3 surface code this would already be a state vector $|\psi\rangle$ with $2^{17} \approx 131$k entries. For a distance 5 (only the second size) this is already ≈ 0.5P ($0.5 \cdot 10^{15}$) entries. Simulating the density matrices $|\psi\rangle\langle\psi|$ to account for decoherence would require even more, namely the square of these numbers.

Luckily there are ways around this issue.A simplified error model can be assumed that simplifies the representation of the surface code and enable a simulation in such simplified cases.
If all the errors are I, X, Z and Y type errors on individual qubits, then only two bits are needed per qubit to represent the qubit state by keeping only track of the errors. These errors can be shifted easily through the gates, without the need for the actual state of all qubits. Because of the simplicity of this representation, it is very advantageous to consider only those types of errors.

[13] lists five ways errors can occur if these errors are assumed to occur with a certain probability. These are listed below:

1. Trying to perform an I (do nothing), but obtain an X, Z or Y error with equal probability $p/3$.
2. Wanting to initialize the state to $|0\rangle$, but preparing in $|1\rangle$. In other words starting the qubit with an initial X error with probability p.
3. Measuring the qubit in the z-axis, but collapsing to the wrong state and returning this value. Thus $|0\rangle \rightarrow |1\rangle$ or $|1\rangle \rightarrow |0\rangle$ with probability p.
4. Trying to perform a single qubit gate, but follow it by an X, Z or Y error with probabilities $p/3$.
5. Trying to perform a two qubit gate, but this is followed by the tensor product of two errors. $I \otimes X, I \otimes Z, I \otimes Y, X \otimes I, X \otimes X, X \otimes Z, X \otimes Y, Z \otimes I, Z \otimes X, Z \otimes Z, X \otimes Y, Y \otimes I, Y \otimes X, Y \otimes Z, Y \otimes Y$. All with probability $p/15$.

If all of these errors are taken into account, this is called the circuit noise model. However, there are also error models that are even simpler. The simplest model is the depolarization model without measurement errors. This model assumes initialization and measurement without errors and also perfect fidelity (no error) gates. The errors are introduced just after initialization, where each data qubit can obtain an X, Z or Y error with probability $p/3$. After this the surface code cycle is performed.
The depolarization model can be enhanced by including measurement errors during the measurements of the ancillas after each surface code cycle.

3.5. Decoding

As stated before, the decoding process of the surface code is the process of using the measurement outcomes of the ancilla qubits after the surface code cycle to guess the errors on the data qubits. Decoding can be quite hard, since as said before, only the parity information is available. Furthermore, the errors not only arise from data qubit errors, but also from gate fidelity errors, initialization errors and measurement errors. Especially measurement errors are difficult to detect as they can only be detected by taking multiple cycles into account. [13]. Most importantly, due to the nature of stabilizers and logical operators, multiple data qubit configurations are possible for a given ancilla configuration. And these data qubit configurations can even have different logical qubit states.

3.5.1. Decoders

There are a lot of decoders, but this work will only discuss the two mentioned. The first is the maximum likelihood as it is the best decoder possible, and second is MWPM as it is the most often used.

Maximum Likelihood

The best decoder possible is called the maximum likelihood decoder. This decoder calculates the most probable data qubit configuration possible taking multiple cycles into account. However, this requires a lot of calculations and time, making it less useful if we want to decode fast. There has been a paper describing an approximation to speed up the algorithm [4]. However it still scales super-linear with increasing distances.

Blossom
Blossom or Edmonds' minimum weight perfect matching (MWPM)[9] algorithm is a very popular decoder due to its trade-off between performance and speed. It works by finding the shortest error chains that generate the error syndrome. It can also decode measurement errors. It can be described as minimizing the total amount of errors. An important point with this algorithm is that it treats X and Z-errors as uncorrelated. Thus, Y-errors which are both an X and Z-error, are intrinsically assumed to be less likely.

3.5.2. Decoder threshold and pseudo threshold

To be able to compare the decoding performance of different decoders, some metrics need to be defined. First, lets assume the logical qubit to be actually just an ordinary physical qubit. In this case, no encoding is happening and the logical error rate would be the same as the physical error rate. This is shown in figure 3.7 by the black $y = x$ line.

Now we can ask the question: What does it mean for an encoded logical qubit to perform better than a single un-encoded physical qubit? It would mean that for a given physical error rate, the logical error rate would be lower. In other words, there would occur less errors on the logical qubit than occur on the physical qubits forming the logical qubit.

In reality, the encoded qubits behave as shown in figure 3.7. Let us take for example the upper-left red line. This line is below the $y = x$ line for small physical error rates and above the line for higher physical error rates. The point where it crosses the $y = x$ line, denoted by a red circle is called the pseudo-threshold. The pseudo-threshold is defined as the physical error rate at which the logical error rate equals the physical error rate. For an encoded logical qubit to be useful, the physical error rate must thus be below the pseudo-threshold.

It would only make sense to go to surface codes with larger distances if we would get a lower logical error rate for a given physical error rate. This could happen in two ways: First, the slope should be steeper or, second, the pseudo-threshold should be higher. Which one is more important depends on the physical error rate of our qubits. If the physical error rate is high (same order as the pseudo-threshold), we would want a higher pseudo-threshold, as this determines if our encoded qubits outperform un-encoded qubits. However, if our physical error rate is low enough, the slope is much more important, as this mainly determines the logical error rate. The slope is determined by the distance. Thus, the lower the error rate, the more is gained by going to a larger distance.

In practice, we will see that each decoder will have a similar slope for a given surface code distance and mainly the pseudo-thresholds will be different. Thus if we would like to compare decoders, an important thing to compare is the difference in pseudo-thresholds for a given distance.

However, if we would like to compare the decoders with one single number, one option that is given in literature is the so called decoder-threshold. This is the point at which the lines for different distances meet for a given decoder. This is shown in figure 3.7 by the gray circle. However, in practice the lines are not straight and do not go exactly through the same point. This makes the decoder-threshold less accurate than comparing individual distances using the pseudo-threshold. It also means that a better decoder-threshold does not always mean a better decoder as the curvature of the lines above the $y = x$ line can shift the decoder-threshold.

3.6. Requirements

As discussed in [52], the decoder must be able to decode n rounds of measurements in the time it takes n rounds to be measured to avoid creating a data backlog. If only Clifford gates are used, the state will remain in the x, y, z bases. This means that all the errors can be tracked in software with a so called Pauli frame [50][37]. This means that no correction needs to happen after each measurement cycle and some delay is allowed in the decoder as long as the throughput is high enough. However, if non-Clifford gates are used, some correction (feedback) must be done on the surface code and we preferably want every round to be decoded without any delay.

As shown in table 3.2, the required throughput of transmons is as high as ≈ 2.5 MHz. An algorithm running in software on a single core 2.5 GHz device could thus only use 1000 arithmetic operations. For a distances 3, 5,

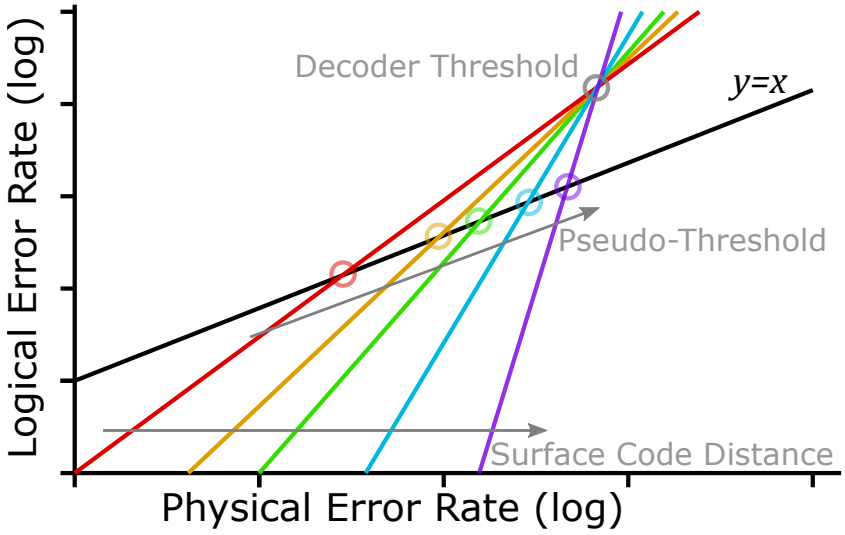


Figure 3.7: Sketch of the logical versus physical error rate of an un-encoded qubit (black) given by the $y = x$ line and five different distances of the surface code. The pseudo-threshold shown by colored circles and the decoder threshold is shown by the gray circle. Clear is that, for larger distances the slope increases and the pseudo-threshold becomes larger.

Operation	Transmons	Single-Electron Spin Qubit (Silicon)
Single Qubit Gate	20 ns [48]	1 μs [48]
Two Qubit Gate	40 ns [48]	0.1 μs [48]
Measure	200 ns [18]	1 μs [2]
Total surface code cycle	440 ns	5.4 μs
T_2^*	30-90 μs [38]	120 μs 1.2-28 ms [54]
Cycles per T_2^*	68-204	22-5185

Table 3.2: The shortest times found in literature for the two used technologies. The shortest time is used, since this is the most critical time for the decoder. The total is the sum of the operations using the adjusted surface code cycle of figure 3.4

7 and 9 this would require the processing of 8, 24, 48 and 80 ancilla errors respectively. This is excluding the time needed to move memory around, which is often the most time and power consuming part. This raises the question if there is an algorithm that can be easily sped up in hardware. If this hardware would be at room temperature, around 200 Mbit/s would need to be transferred from the qubits at cryogenic temperatures, and this only for a single logic qubit encoded with a relatively simple coding. The required data rate will easily go beyond a Tbit/s for a quantum processor with 1000 logical qubit encoded with a distance larger than 20. For example, Fowler [13] assumes that a practial quantum processor will have more than a thousand logical qubits each consisting of around 10^3 to 10^4 physical qubits. This would mean a data rate of 2.5 to 25 Tbit/s. A better option proposed by our group is to operate this hardware at 4 K close to the qubits. As a hardware solution, neural networks are proposed. The benefits of neural networks here are:

1. Highly parallelizable hardware.
2. Ability to be pipelined.
3. Constant execution time after being trained.
4. Ability to train on any data set.

This means that they have a good chance to be sped up to the required speeds and not consume to much power and area after optimization. The question that remains is whether they are able to outperform other algorithms in terms of decoding accuracy under practical hardware constraints.

3.7. Conclusion

This chapter discussed the use of the surface code as a quantum error correction scheme to improve the performance of a quantum computer. A modification to the standard measurement cycle was made so as to only use the gates that are currently available in the targeted qubit technologies, i.e. transmons and single-electron silicon spin qubits.

Using this cycle and the data from these technologies, it was found that using an algorithm running in software is too slow to process the data from the surface code running at a speed of 2.5 MHz. For this reason the next chapter will discuss the use of neural networks as decoders as they have the potential to be sped up in hardware.

4

Decoding With Neural Networks

The previous chapter discussed the decoding of the surface code. Decoding involves some data processing to guess the errors on the data qubits from the error syndrome. Because neural networks are good at finding patterns and giving out probabilities for classification problems, they appear to be a good candidate for decoding.

This chapter will discuss the basics of neural networks and how they can be used to decode the surface code. First, the basics of neural networks are discussed. This includes the architectures, learning methods and problems. The discussion on moving from software to hardware implementations follows, with highlights on relevant points for optimization. Finally, the usage of neural networks to the decoding of surface codes is treated by introducing the pure error decoder, which reduces the classification problem to a simpler one and thus eases the training process and reduces the hardware needed for the neural network implementation.

4.1. Neural Networks

Basic artificial neural networks are a regular multilayered structure made of computing nodes. For a complete background on neural networks, see [39]. Neural networks are a combination of artificial intelligence and machine learning with inspiration from biology. The main idea is to mimic the human brain or a collection of neurons (figure 4.1) to learn from a data set.

4.1.1. Nodes

The computing nodes of the neural network perform three functions that the biological neurons also have. This is shown in figure 4.2(a). First, they perform a dot product between the outputs of the previous layer

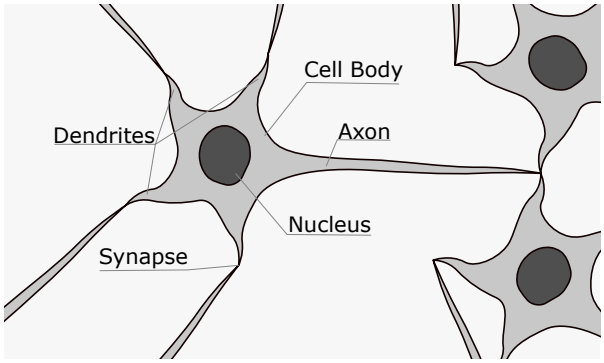


Figure 4.1: Simplified drawing of a collection of neurons. The input signals of the neuron arrive from the axons of previous neurons through the synapses in the dendrites. This transfer will determine the signal strength and whether the signal is activating or inhibiting. The collection of signals from the dendrites is then accumulated in the cell and if the sum crosses the threshold, a signal will be sent through the axon to following neurons.

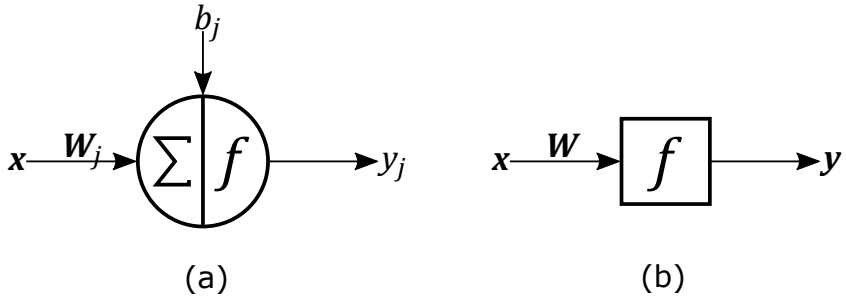


Figure 4.2: (a) A schematic representation of a node in a neural network. The input vector x of the node is multiplied by the weights vector W_j and summed together with the bias b_j into the sum s_j. The sum is then passed through the transfer function f to the output y_j. (b) A schematic representation of a collection of nodes in a layer. The input vector x is multiplied using a dot product by the layer weight matrix W. Then (for simplicity not drawn) the bias vector b is added. Then the function is applied element-wise to this sum vector s, resulting in the output vector y.

$\mathbf{y}^{(l-1)}$ and the scaling weights $\mathbf{W}_j^{(l-1)}$ of this node j . The result is the input sum $s_j^{(l)}$ of this node j and resembles the accumulation of all the signals from the dendrites of the neuron scaled by the width of the dendrite. Just as a neuron has inputs that can be activating or inhibiting, the weights of the computation node can be both positive or negative to mimic this behaviour.

Next is the threshold of the neuron. In order to output a signal (similar to triggering the action potential in a biological neuron), the accumulated input signal should be larger than the threshold. In the computation nodes, this threshold is implemented by the bias $b_j^{(l-1)}$ of node j. This bias is added to the sum as shown in equation 4.1. Another way of interpreting this is by describing the bias as another weight. However, instead of being multiplied by the output of a previous node, it is multiplied by a constant term of 1.

$$s_j^{(l)} = \sum_{i=0}^{N^{(l-1)}-1} \left(W_{j,i}^{(l-1)} y_i^{(l-1)} \right) + b_j^{(l-1)} \tag{4.1}$$

The final part of the node is the transfer function. This is a non-linear function that maps the sum to an output value $y_j^{(l)}$. It is equivalent to the action potential sent through the axon to the next neurons. The function must be non-linear to avoid the whole neural network from becoming a linear system. This non-linearity is what gives the neural network its computational power. As biological neurons have a non-negative output (there is a signal of a certain strength or there is no signal), the transfer functions could also map the input to a non-negative range $[0, y_{max}]$. However, there is no such restriction. The only restrictions are non-linearity and differentiability. Why the function must be differential is described in the section on back propagation. The total input-output relation of a node is shown in equation 4.2. Here again the vector $y^{(l)}$ is the vector containing all outputs of nodes j in layer l. This also means that the input of this layer (being the output of the previous layer) is $y^{(l-1)}$ containing all input nodes i. The bias vector $b^{(l-1)}$ contains all thresholds of layer l and weight matrix $W^{(l-1)}$ all the weights between input nodes i and current nodes j.

$$y_j^{(l)} = f\left(\sum_{i=0}^{N^{(l-1)}-1} \left(W_{j,i}^{(l-1)} y_i^{(l-1)} \right) + b_j^{(l-1)} \right) \tag{4.2}$$

$$y^{(l)} = f\left(W^{(l-1)} \cdot y^{(l-1)} + b^{(l-1)} \right) \tag{4.3}$$

The ensemble of multiple parallel nodes in one layer can also be written in terms of matrix vector notations. This is shown in equation 4.3 and graphically in figure 4.2(b). A drawing of a neural network consisting of multiple layers is shown in figure 4.7. However, before discussing different architectures, a closer look will be given to different transfer functions.

4.1.2. Transfer functions

There are a lot of transfer functions used in literature and, in principle, any function following the non-linearity and the differentiability rules is allowed. This section will describe some common functions and the functions used in this work.

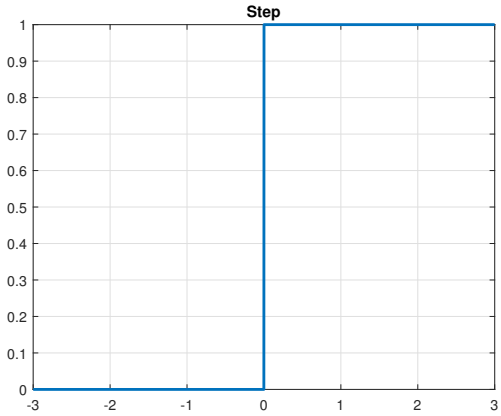


Figure 4.3: Plot of the stepfunction.

Step Function
The simplest function that will give a threshold behaviour is the step function, shown in figure 3.7 and equation 4.4. This function is one of the first functions used in neural networks. However, one might note that this function is not differentiable at the threshold. One solution to this problem could be to assume some monotonic function as the derivative, such as a simple linear line.

$$\text{step}(x) = \begin{cases} x < 0, & -1 \\ x \geq 0, & 1 \end{cases} \tag{4.4}$$

A node using the step function is known as a perceptron. Perceptrons were used in early neural network research, because the step functions were easy to compute and implement. Also, they do have a nice binary classification property, meaning they are intrinsically good at dividing the space into two separate regions.

Logistic Function
There are two common logistic functions: the sigmoid function and the hyperbolic tangent (tanh) function (Fig. 4.4 and equations 4.5 and 4.6). The sigmoid is often denoted as $\sigma(x)$. These functions are hyperbolic for large inputs. The sigmoid function will go to zero for negative x and to one for positive x. The hyperbolic tangent is a scaled version of the sigmoid function that maps the input to an output between -1 and $+1$ for large negative and large positive inputs, respectively. The hyperbolic tangent goes through the origin and the sigmoid goes through $y = 0.5$ for $x = 0$. An additional benefit of the use of the logistic functions is that their derivatives can be easily calculated.

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{4.5}$$

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \qquad \tanh'(x) = 1-\tanh^2(x) \tag{4.6}$$

The downside of these functions is the hardware implementation, as they require exponential functions and divisions. Both of those are expensive in time, power and area.

Rectified Linear Unit Function
Another popular function, foremost in convolutional neural networks, is the Rectified Linear Units (ReLUs) function. The ReLU, defined in equation 4.7, is similar to the step function in that it is off for values lower than the threshold (default of zero). However, when the ReLU is on, it does give information about the input sum, whereas the threshold does not. This also improves the differentiability of the ReLU. One problem that remains is the zero derivative for values smaller than the threshold. When training the network, this will set the derivative of the whole chainrule product to zero, as it will be discussed later when explaining back

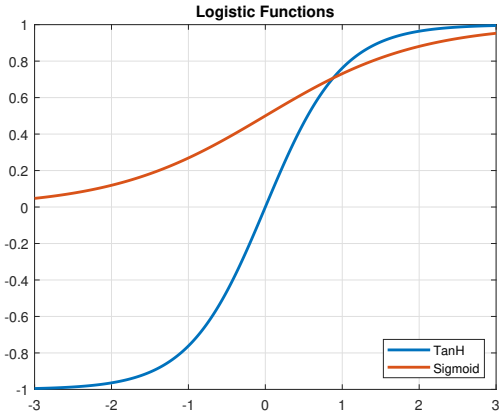


Figure 4.4: Plot of the two logistic functions, the sigmoid and the hyperbolic tangent.

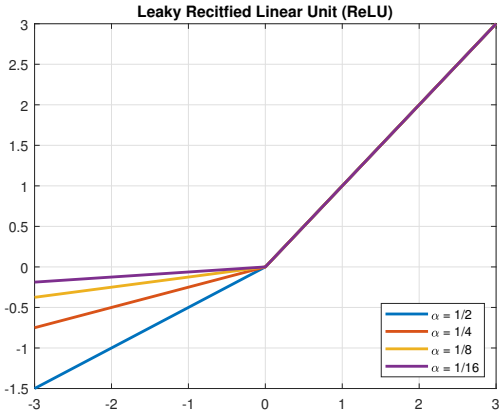


Figure 4.5: Plot of the leaky ReLU for different leakage factors α.

propagation. For this reason, a leaky ReLU can be used, as depicted in figure 4.5 and equation 4.8. This transfer function has a smaller slope for negative values. Thus, it maintains the non-linearity, but provides a non-zero derivative everywhere. Another benefit is the ease of computation, especially if α is a power of two, reducing the multiplication to a bit shift.

$$\mathrm{ReLU}(x) = \begin{cases} x < 0, & 0 \\ x \geq 0, & x \end{cases} \qquad \mathrm{ReLU}'(x) = \begin{cases} x < 0, & 0 \\ x \geq 0, & 1 \end{cases} \tag{4.7}$$

$$\mathrm{LeakyReLU}(x) = \begin{cases} x < 0, & \alpha x \\ x \geq 0, & x \end{cases} \qquad \mathrm{LeakyReLU}'(x) = \begin{cases} x < 0, & \alpha \\ x \geq 0, & 1 \end{cases} \tag{4.8}$$

Squared Non-Linearity Function

When looking for a function more closely resembling the logistic functions, a lot of options can be found, such as the softsign [10] and the inverse square root [3]. However, one function that is quite easy to compute in hardware is the SQuare Non-Linearity (SQNL) function. This function does not require divisions and complex functions, such as square roots, exponential functions, trigonometric functions and logarithms. As presented in equation 4.9, the SQNL function consists of four parts. The outer parts ($|x| > 1$) are constant, whereas the inner part consists of two parabolas. Similar to the ReLU, the constants have a zero derivative, thus some

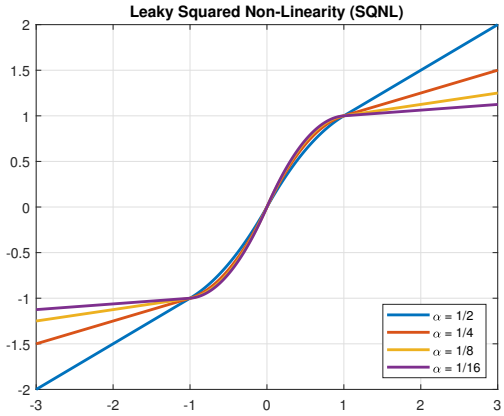


Figure 4.6: Plot of the leaky SQNL, for different values of the leakage factor α.

leaky SQNL can be used as shown in figure 4.6 and equations 4.10.

$$\text{SQNL}(x) = \begin{cases} x < -1, & -1 \\ -1 \le x < 0, & 2x + x^2 \\ 0 \le x \le 1, & 2x - x^2 \\ x > 1, & 1 \end{cases} \qquad \text{SQNL}'(x) = \begin{cases} x < -1, & 0 \\ -1 \le x < 0, & 2 + 2x \\ 0 \le x \le 1, & 2 - 2x \\ x > 1, & 0 \end{cases} \tag{4.9}$$

$$\text{LeakySQNL}(x) = \begin{cases} x < -1, & \alpha x + (\alpha - 1) \\ -1 \le x < 0, & (2-\alpha)x + (1-\alpha)x^2 \\ 0 \le x \le 1, & (2-\alpha)x + (\alpha - 1)x^2 \\ x > 1, & \alpha x + (1 - \alpha) \end{cases} \quad \text{LeakySQNL}'(x) = \begin{cases} x < -1, & \alpha \\ -1 \le x < 0, & (2-\alpha) + 2(1-\alpha)x \\ 0 \le x \le 1, & (2-\alpha) + 2(\alpha - 1)x \\ x > 1, & \alpha \end{cases} \tag{4.10}$$

4.1.3. Feed-Forward Neural Network

The simplest neural network, called a feed-forward neural network is shown in figure 4.7. A feed-forward neural network consists of an input layer and minimally one layer of computing nodes. The last computing layer is called the output layer. A network with only the input and the output layer is often called a two-layer neural network. However, this is not a correct name, because the input layer does not contain any computing nodes. This makes it is different from other layers in the network. If multiple layers are present in between the input and the output layer, these are called the hidden layers. Neural networks with many layers are called deep neural networks and the process of training them is called deep learning. However, a fully connected feed-forward neural network with one hidden layer can already map any non-recurrent function given enough nodes in the hidden layer [17]. The nodes of the input and output layer are not a design parameter but are fixed by the application and determined by the number of inputs and outputs of the function to be implemented, respectively.

4.1.4. Convolutional Neural Network

One can have a layer in a neural network which is not fully connected, but has multiple kernels of weights that are moved around the input space. Such a layer is called a convolutional layer [25]. A convolutional neural network usually consists of more layers than a feed-forward neural network and is often ended with some fully connected layers to perform classification. An image of a convolutional neural network is shown in figure 4.8.

Convolutional neural networks are mainly used in tasks with some spatial invariance in the input data, such as images. The benefit is that less weights are used, as not everything is fully connected. Furthermore, because the kernel is the same over the whole image, there is a lot of reuse of weights. And, finally, having both the sparsity and reuse of weights, the fully connected layers in the end can use the position of the kernels on

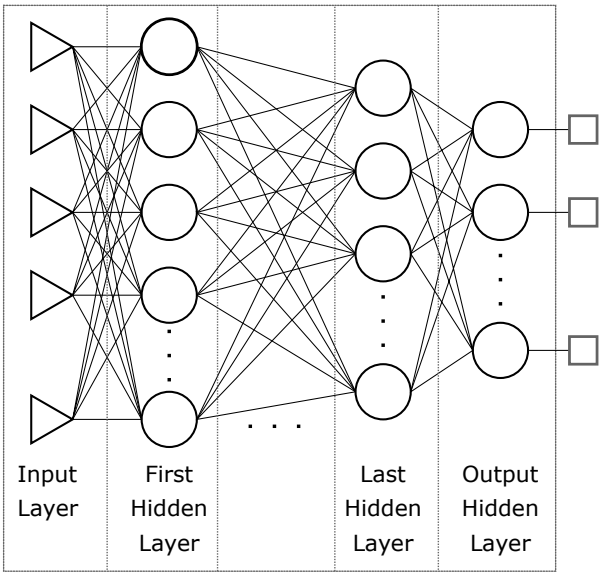


Figure 4.7: Schematic of a fully connected feed-forward neural network. Each node is indicated by a circle. As can be seen the input layer is denoted by triangles, as this layer does not contain computing nodes.

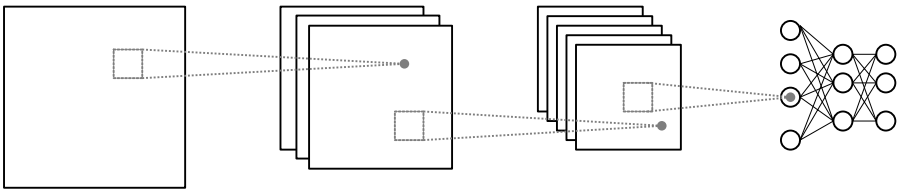


Figure 4.8: Image of a convolutional network with three convolutional layers and three fully connected feed-forward layers. The kernels in the convolutional layer are indicated by the dashed squares. The output of the convolution becomes one pixel in the input of the next layer. Due to the nature of convolutions, the image size is reduces at each layer. The number of images in each convolutional layer represents the amount of different kernels that was used in the previous layer. Since these layers are three dimensional, the kernels are as well.

the image as part of the data. An example of using all of these points is when used in a self-driving vehicle. The neural network can search, for example, with a group of kernels that represent cars. As the cars will probably not take up the entire image, the kernels can be smaller and does not need to be fully connected layers. Next, because the car can be anywhere in the picture, but will look the same everywhere, the kernel can be reused when shifted over the image. And finally, because the kernel that triggers has information about the position, just because this is the kernel which looks at a particular part of the picture, the fully connected layers can use this information to locate the car.

4.1.5. Recurrent Neural Network

Another modification to the feed-forward network is to use recurrent layers as shown in figure 4.9. If the current output depends on the current input data, but also on the output data at the previous time step, then it is called a recurrent function. Because the feed-forward networks do not have any memory, they cannot perform these recurrent functions. Thus, feed-forward networks will also be unable to use the temporal coherence in the data if this is present.

As can be seen in figure 4.9(b), the recurrent weights are denoted by R and operate on the output of the layer y. Applying this to equation 4.3 of the feed-forward network, gives us the matrix vector notation of a recurrent layer is as shown in equation 4.11.

$$y^{(l)}(t) = f\left(W^{(l-1)} \cdot y^{(l-1)}(t) + R^{(l-1)} \cdot y^{(l)}(t-1) + b^{(l-1)}\right) \tag{4.11}$$

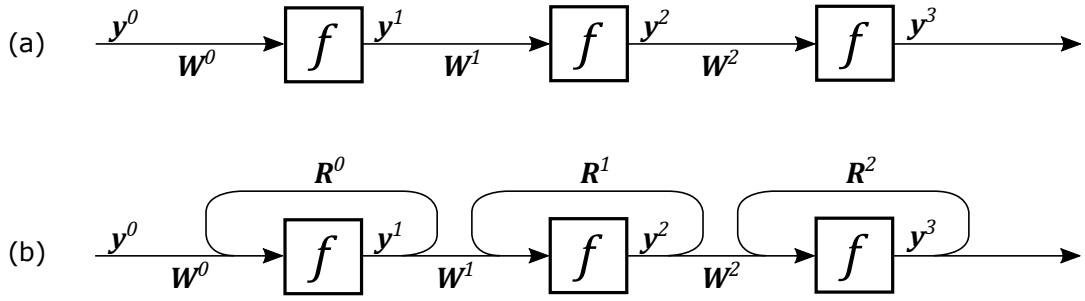


Figure 4.9: (a) A layer-wise representation of a feed-forward neural network. (b) A layer-wise representation of a recurrent neural network. Here the forward weights are denoted with W and the recurrent weights by R.

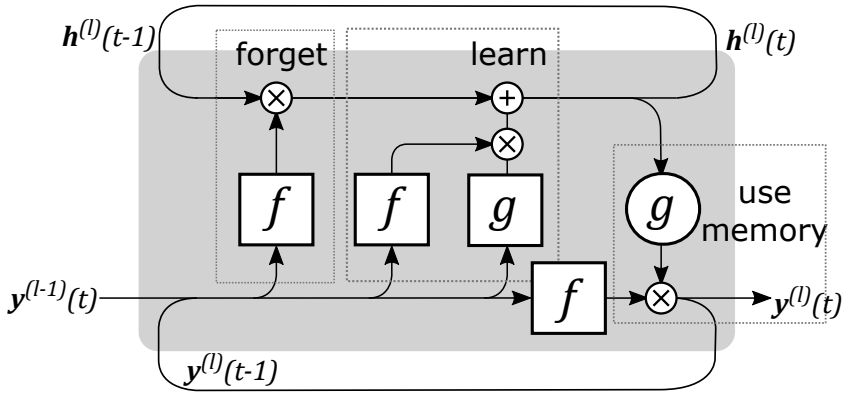


Figure 4.10: Long Short-Term Memory Layer. The f function represents a function with an output range of $[0,1]$. This allows it to scale the signals it is multiplied to. The square blocks are feed-forward layers, the circles represent element-wise operations. The g function can be any function. The bottom loop represents the short-term (recurrent) memory. The upper loop represents the hidden state or the long-term memory.

There is one variant of the recurrent neural networks called Long Short Term Memory (LSTM) networks. These networks have, as the name suggests, two memory paths, the short-term memory being the same memory as in the standard recurrent neural network and the long-term memory that has an additional path that does not contribute directly to the output but functions as a hidden state. The input (containing both the output of the previous layer at this time step and the current layer at a previous time step) will update this hidden state. The update is done by forgetting some of the current state, followed by learning from the new data. Then the hidden state is used to update the current output. An illustration is shown in figure 4.10.

4.2. Training

The main property of biological brains that neural networks try to mimic is the ability to learn. The learning is done by adjusting the weights depending on how the neural network reacts to the applied inputs. There are many ways to train a neural network. Some examples are:

- Supervised Learning, where both the input and the target output is given to the neural network.
- Unsupervised Learning, where only the input is supplied to the neural network and patterns have to be identified without a target output.
- Reinforcement Learning, where the inputs are applied without a target, but the trainer does give some performance feedback to the neural network.
- Genetic Learning, where the neural networks are encoded by data. This is similar to evolution. Depending on the performance of the neural networks they will reproduce. The best performing neural networks will interchange some part of the weights and/or structure and some of this data will be changed due to probabilistic mutation.

Since in this work both the input and the output data of the neural network are known, the most suited technique is supervised learning. Here, the input to the neural network will be the error syndrome from the ancilla measurements of the surface code and the target outputs are the effective errors on the data qubits that the neural network try to predict.

Training the neural networks in this way requires the minimization of a cost function. The cost function describes how far the outputs y of the neural network are from the target outputs t. For example the Mean-Squared-Error (MSE) function in equation 4.12 uses the L_2-norm as the cost function. The L_2-norm describes the squared distance between the output and the target in the output space.

$$C = \frac{1}{N}\sum_{n=0}^{N-1}\left(y_n - t_n\right)^2 \tag{4.12}$$

4.2.1. Back-propagation

After applying the inputs to the neural network and computing the cost function by comparing the output to the target, the neural network is ran in reverse in a process called back-propagation. Back-propagation calculates the influence of each weight on the error, by taking the partial derivative. This derivative is the reason that a differentiable transfer function is needed. It can be calculated by applying the derivative chain rule many times. This explains why a zero derivative in the transfer function is not desirable, as it will set this whole derivative to zero. In practice, ΔW for each weight is calculated as

$$\Delta W_{j,i}^{(l)} = \frac{\partial C}{\partial W_{j,i}^{(l)}} \tag{4.13}$$

which is subtracted from the current weight scaled by the learning rate α. This process of reducing the cost function in iterative steps is called gradient descent. The problem is that this process can get stuck in local minima of the cost function. These are points where the gradient is zero, but it is not the lowest obtainable error. One can tune the training rate to try to avoid getting stuck in local minima by trying to take large enough steps to avoid them. The problem with too big steps is that the exact minimum can be missed.

Another technique used to avoid a local minimum is using momentum. This can be thought of as the error rolling down the hill like a ball, developing momentum. This momentum μ can then be used to roll out of the minimum. A mathematical formulation of this is shown in equation 4.14. The new weight W will be the old weight minus the scaled sum of the current derivative plus a scaled version of the previous derivative.

$$W(t) = W(t-1) - \alpha\left(\Delta W(t) + \mu\Delta W(t-1)\right) \tag{4.14}$$

The training algorithm used in this work called ADAM [19], uses both a varying learning rate as well as a varying momentum.

4.2.2. Generalization

One important result of training should be that the neural network can identify what it has seen in the training data in the new data that it has not seen before. This is known as generalization. One problem that is associated with generalization is the problem of overfitting. In this case, the neural network is really well trained on the training data, called the training set, but does not perform well on new data.

One technique used to tackle overfitting is splitting the available data in two sets. The first is the training data set as mentioned before. The second set is called the test set. This set will not be used to train the neural network, and can be used to test how the neural network performs when subjected to new data.

As the problem with overfitting occurs when the data set is too small compared to the total data space, it is not a problem in our case. This is because using the error models of the surface code described in the previous section, we can keep generating data on the fly until the performance is as desired. If the desired performance is never reached, it is not due to the lack of generalization or to overfitting, as all test data is new. Simply, the neural network used is not capable of identifying the patterns in the data.

4.2.3. Training Techniques

There are more training techniques to improve the generalization and reduce overfitting. Although these problems might not be as problematic for this data set, the techniques can be used to improve the performance and to have shorter training times.

Batches

Instead of updating the weights after each new input data, one can choose to only update the weights after a number of input data sets. This will give the advantage that, even though some weights might need to be updated after certain inputs, a more clear direction of gradient descent is obtained after averaging over some input data sets. This way of training is called training in batches.

Regularization

Regularization is the technique where more terms are added to the cost functions that help steer the training in a certain direction.

Early Stopping

In early stopping, the training is stopped if the performance of the testing data set starts going down. This is motivated by the fact that reducing testing performance might indicate overfitting to the training data set.

Dropout

Dropout is a technique where some of the weights are set to zero during training, forcing the neural network to only use the remaining nodes. This will make sure that the neural network does not depend on only a few nodes, but tries to make use of all the nodes. Another way of looking at this is as training multiple networks in parallel during dropout and combined them if all nodes are included.

Weight Sharing

Weight sharing is a technique that is typically adopted in convolutional neural networks. If there is some symmetry in the structure of the neural network or in the input data set, this might benefit from weight sharing. During training, the constraint that some weights must be equal (hard weight sharing) or similar (soft weight sharing) will be enforced. This will not decrease the computational power of the neural network, but it will limit the amount of free parameters.

Weight Minimization

Weight minimization adds the L_1, L_2 or L_n norm on the weights to the cost function. Adding the L_1-norm, means that the total sum of weights is minimized and the L_2-norm will minimize the sum of squared weights. In other words, this will make the weight matrices sparser (set less important weights to zero) or decrease the size of the average weight.

Pruning

Pruning is a technique that is often used in larger networks. This technique will set some of the weights that are close to zero and thus less important, equal to zero during the course of the training and then continue training while still keeping them at zero. This will reduce the number of weights in the network. Having kernels like in a convolutional neural network can be seen as some sort of pre-pruning, as this will set all the weights except those in the kernel to zero.

4.3. Choosing a Neural Network Architecture

A lot of neural network architectures exist (see section 4.1.5) and some of them have also been used in quantum error correction decoding [52][51][12][1][47][23][5]. Due to the large amount of available architectures, we need to look for the one that is most suited to the task. As Yann LeCun said in his 1989 paper [25]:

> *"... good generalization performance can be obtained if some a priori knowledge about the task is built into the network. Although in the general case specifying such knowledge may be difficult, it appears feasible on some highly regular tasks such as image and speech recognition. Tailoring the network architecture to the task can be thought of as a way of reducing the size of the space of possible functions that the network can generate, without overly reducing its computational power."*

On the contrary Rich Sutton said in 2019 [45]:

> *"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. (...) Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation."*

The last quote would imply that the best performance is not obtained by steering the neural network in some direction, but giving it all the freedom it can get and use the exponential growth in sheer computational power as predicted by Moore's Law. However, when looking more closely to the type of a priori knowledge that Le Cun is talking about, one could argue that exploiting the concept of convolutions and other general invariances to the neural network can boost its performance. This can for example be seen in the vast acceleration in the performance of neural networks after the introduction of convolutions in deep learning.

Applying this discussion to the surface code, one could argue that some form of a convolutional neural network has potential as a decoder [53]. This is because the surface code has some spatial invariance, especially in case of the toric code or as the distance increases. This is because a toric code has no boundaries, meaning infinite spatial invariance. However, a surface code with a large distance will have a lot more surface than boundaries. For this reason the larger the distance the more it can be approximated by the toroidal case.

Another invariance (or actually more of a temporal coherence) arises when the measurement errors on the ancilla qubits are taken into account. Detecting these measurement errors and distinguishing them from errors due to decoherence, dephasing and non-ideal gate pulses requires the data from multiple cycles [13]. One possible implementation is to extend the amount of inputs of the neural network to include ancilla measurements from multiple cycles [52] . In the most generic case, this would mean that the neural network has to learn by itself which input correspond to a certain cycle and to a certain ancilla qubit. Another way would be to have each ancilla measurement in one cycle as the pixel in an image and each cycle as a separate channel, like red, green and blue would be the channels in a normal picture, and create a convolutional neural network. But the most intuitive approach would be to use the temporal data as actual temporal data and use some form of a recurrent neural network, such as an LSTM neural network. One could even use an LSTM with convolutional layers to benefit from both temporal and space invariance.

If an error model such as a simple depolarising error model without measurement errors is used, then no memory is needed in the neural network. However, even for the more extensive error models, researching feed-forward neural networks is useful. For an error model that includes measurement errors a recurrent neural network is needed. As discussed in section 4.1.5, an RNN or LSTM layer can be described as one or more feed-forward layers in a loop. Thus the results from a feed-forward neural network can be used here. Next, convolutional layers are a form of regularization where some weight sharing and pre-pruning is performed and this can always be applied to any neural network whether it is feed-forward or recurrent. Thus for larger distances, convolutional layers can be added to the neural network for any error model. Especially if the kernels used are similar to small distance fully connected layers as in [53]. Concluding that the most prominent architecture to explore first is the simple feed-forward neural network, with the possibility to extend the results to more complicated cases.

4.4. Pure Error Decoder

Since the decoding problem in a surface code (see section 3.5) is a function mapping between inputs and outputs, feed-forward neural networks can be a promising candidate for a decoder implementation as a feed-forward network with at least one hidden layer can map any arbitrary non-recursive function given enough nodes. It is then good to check whether the problem is suited for neural networks and if it is not, whether it can be changed to better suit the neural network. As is well known, neural networks are good at classification problems. The problem as it is now can be described as the classification of the error on each data qubit in four categories (I, X, Z, Y) given the measurements of the ancilla qubits as input and where the errors on the data qubits are correlated. This is depicted in 4.11(a). There might be a better alternative though as proposed in [52]. One could use the neural network in parallel with a pure error decoder, see figure 4.11(b). This would reduce the problem for the neural network to a classification problem of only the logical qubit state, instead of all the physical data qubits.

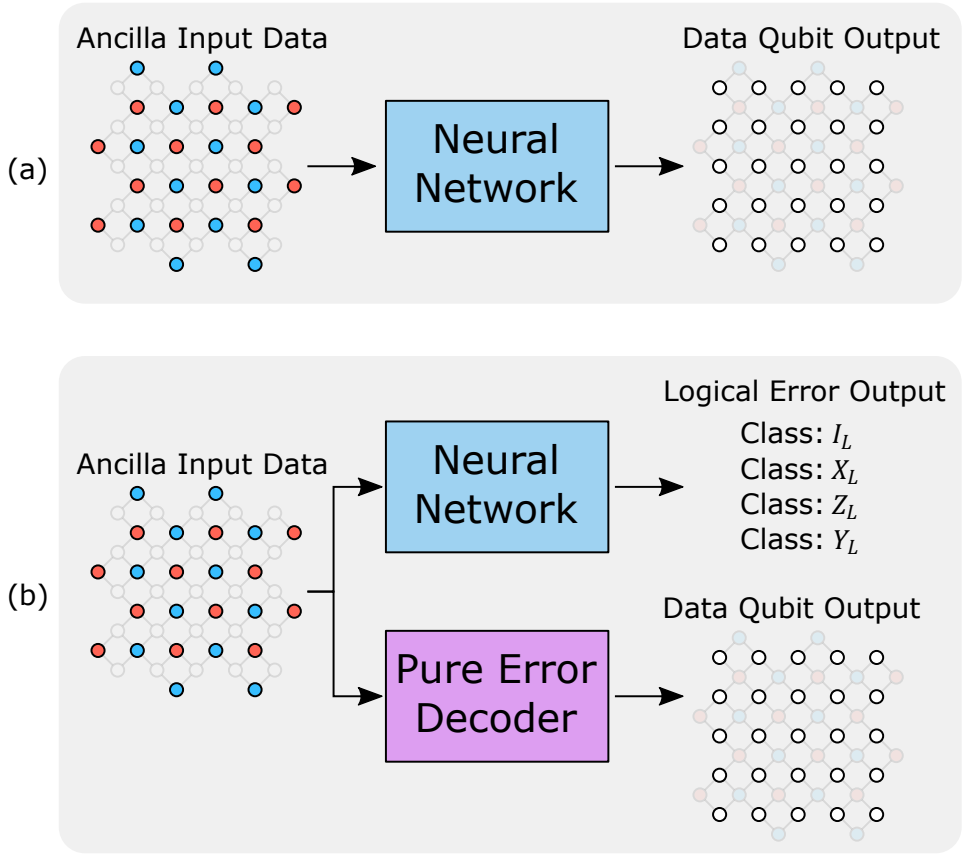


Figure 4.11: Figure showing the introduction of a pure error decoder in parallel with the neural network decoder. (a) The ancilla syndrome is inputted to the neural network and the neural network outputs the estimate of the data qubit errors that produced this ancilla syndrome. (b) The pure error decoder outputs the estimated data qubit errors from the ancilla syndrome. The neural network than outputs the estimated logical error between the pure error decoder output and the actual errors that occurred on the data qubits.

4.4.1. Pure Error

Using the background in section 3.3, the theory of [35] states that any total error E can be expressed as shown in equation 4.15 as the product between stabilizers S, a pure error P and a logical error L.

$$E = S \cdot P \cdot L \tag{4.15}$$

This means that any error can be expressed by the product of loops (stabilizers), chains that start at an error in the middle and end at an error at the boundary (pure errors) and chains that end in an error at the boundary at two ends (logical errors). An illustration of this idea can be found in figure 4.12. Since both stabilizers and logical operators will never trigger a parity error in any ancilla measurement, this means that the pure error is the only error that will generate ancilla errors. A precise definition of the pure error is given below:

> *A product of data errors that will generate the same ancilla errors as the actual errors on the data qubits, but can differ by a product of stabilizers and logical operators.*

Thus, a pure error decoder, i.e. a decoder that finds a pure error from a given set of ancilla errors, is very easy to build, since it is in principle the simplest decoder to recreate the same ancilla errors and does not have to be necessarily close to the actual data qubit errors. Since stabilizers do not change the logical state or the triggered ancillas, the only thing that the neural network has to do now is to identify the logical error that the pure error decoder makes with respect to the actual errors on the data qubits. This greatly reduces the outputs of the neural network from the error on each data qubit, to only the errors on the logical state. Next, it also reduces the complexity of the problem that the neural network needs to learn. And finally, it makes sure that the pure error decoder will always remove all the triggered ancilla and the neural network can only change the logical state. This is a problem for the neural network decoders, as they estimate the error on each

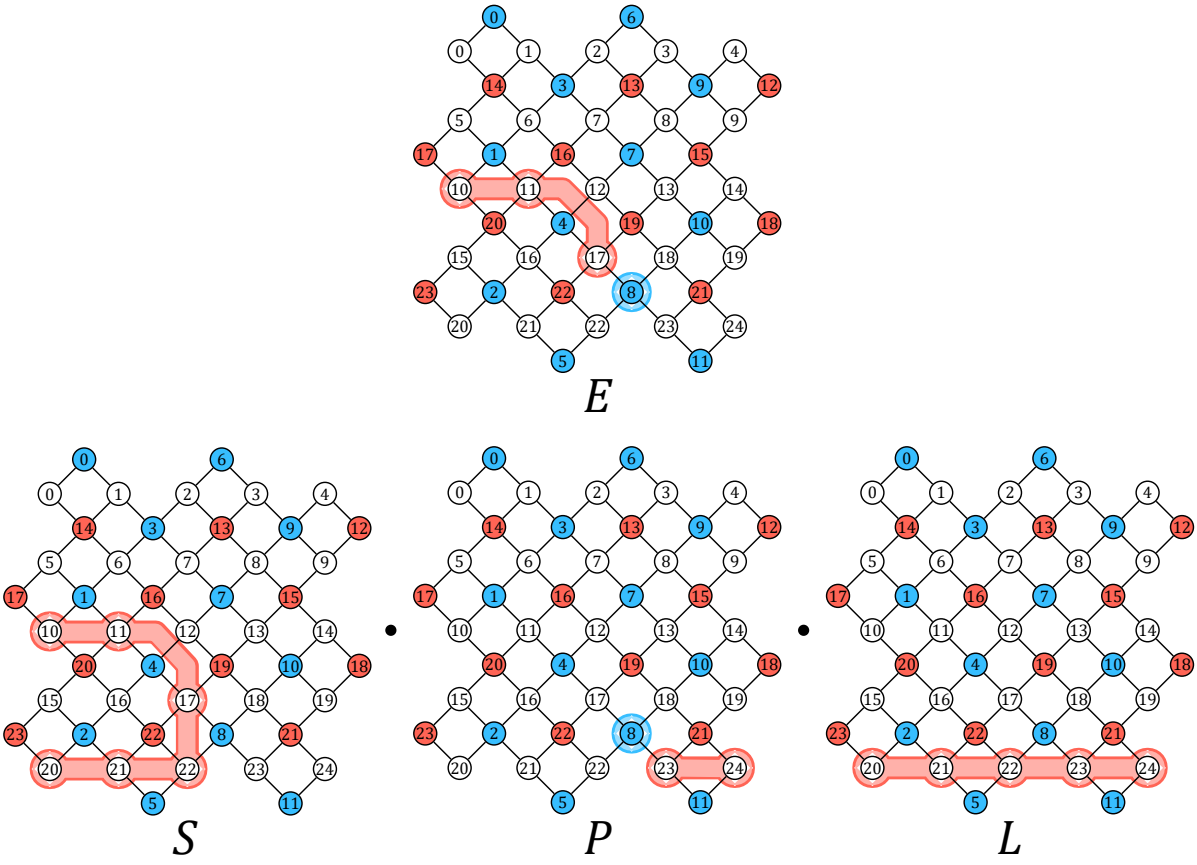


Figure 4.12: Example of the decomposition of the error E into the product of three parts for the distance 5 surface code. The first part is a product of stabilizers S. In this case it is the stabilizers around ancillas 20, 22 and 23. Data qubits 15 and 16 both have two errors and thus they cancel each other. Next is the pure error P, connecting an error in the surface to the nearest boundary. It is clear that the pure error is the only error that triggers an ancilla. Finally is the logical error L connecting two opposing boundaries.

data qubit separately and it thus can happen that applying the correction of such a neural network decoder will not remove all ancilla errors. This is discussed in more detail in [51]. However, it means that using only a neural network decoder without a pure error decoder will need multiple correction cycles, until an output is found for which all the ancilla errors disappear. This is because, if any ancilla errors remain, it is already clear that the found solution is not correct, as errors are still present. For an illustration of the total system containing both the neural network and the pure error decoder, see figure 5.1 in the next chapter.

4.4.2. Finding the pure errors

A very simple way to generate a pure error decoder with hardware is by using XOR-gate chains as used in [52][51][50]. Looking at the pure error P in figure 4.12, it shows the blue X-ancilla number 8 to be triggered. We need to find which error in the data qubits would generate the triggering of such ancilla error. The XOR-gate chain principle now works as follows. Since it is an X-error, the path must end at one of the Z-borders (left and right). In this figure, the path is chosen to the right border. This path goes through data qubits 23 and 24 and indeed results in the triggering of ancilla 8.

Starting from ancilla 8, the first data qubit seen is number 23. Since at the moment there is an error, data qubit 23 is labeled with the error that would cause this ancilla to trigger, so a Z-error. Following the chain to the next X-ancilla gives ancilla 11 and the next data qubit is 24. The error on 24 is now calculated as the XOR between the error on data qubit 23 and the error on ancilla qubit 11, which is in this case is $1 \oplus 0 = 1$, thus an

error is also applied on 24, which is the last qubit on the chain. Giving the equations:

$$d_{23,Z} = a_{8,X}$$
$$d_{24,Z} = a_{11,X} \oplus d_{23,Z}$$

If the chain were longer, this recursive function would also be longer, consequently increasing the amount of XOR-gate layers in the hardware.

4.4.3. Designing the pure error decoder

Making a pure error decoder with this concept would require all of the ancillas to form chains to the boundaries. Theoretically, it does not matter how this is done, because it does not need to be correct and the neural network will learn during training how the pure error decoder works. However, if any arbitrary pure error decoder can be used, then having one with a high computation speed when generating the data set is preferred. Also, keeping the pure error decoder as regular as possible will increase how transitional invariant the pure error decoder is, which might help the training as discussed in section 4.3. When looking at the surface code as depicted in figure 4.13(a), another invariance is highlighted by the grey cross, namely a 90 degree rotational invariance as the code is point symmetric around data qubit 12. It might seem as if this is not a 90 degree rotation, as it does interchange the X and Z-ancillas. However, as the boundaries also rotate, the hardware will be identical. This figure also shows all the data qubits at the edge, outlined in red and blue rectangles. For one boundary of a surface code with distance d, there are d data qubits (in this case 5) at that edge. In the corresponding half (up to the parallel grey line) there are $(d^2-1)/4$ (6) ancilla qubits of the corresponding color, which need to linked up to this boundary with the XOR-chains. However as shown in figure 4.13(b), the ancilla row nearest to the boundary has only $(d+1)/2$ (in this case 3) qubits and thus can only connect to $(d+1)/2$ data qubits, meaning there can only be $(d+1)/2$ chains per boundary and each must contain $(d-1)/2$ (2) qubits. This is because $(d-1)/2 \cdot (d+1)/2 = (d^2-1)/4$. An orderly way of doing this is shown in figure 4.13(c) for the top and bottom chains and in 4.13(d) for all chains. The arrows show the direction of the XOR-gate chains. It can be checked that this is still 90 degrees point symmetric.

The result of this discussion is a pure error decoder structure with qubit numbering, that exploits a lot of repetitions and both translational and rotational invariances. For example, for both the X and Z-ancillas, the difference between two adjacent ancillas in a chain is always $(d+1)/2$ (3), see figure 4.13(d). Next, the difference between two adjacent data qubits is always 1 and d for X and Z-chains respectively. Finally, rotating the pure error decoder is as simple as adding $(d^2-1)/2$ (12) to an X-ancilla qubit to obtain the Z-ancilla and for a Z-ancillas to subtract its index from d^2-2 to obtain the corresponding rotated X-ancilla. All these results give an algorithm without any exceptions (see equation 4.16), making it easy to implement in code (see get_pure_error function in appendix C.3). Next, the regularity makes it easy to implement the decoder in hardware. And finally, the invariances make it easier for a convolutional neural network to learn the function of the pure error decoder, following the reasoning in the previous section 4.3 about choosing a neural network architecture.

Closed Form Pure Error Decoder Algorithm

The adopted algorithm can be described by the following recursive formula:

$$E(q[c,i]) = E(q[c,i+1]) \oplus E(a[c,i]) \tag{4.16}$$

where $E(q[c,i]$ is the error on qubit q in chain c at index i and $E(a[c,i])$ is ancilla a in chain c at index i. The numbers of the qubits at these indices are shown as below:

$$q[c,i] = \left\lfloor \frac{d-1}{2} + p\cdot(i+1) + 1 \right\rfloor \cdot \left\lfloor t\cdot d + (1-t) \right\rfloor - 1 + 2c \cdot \left\lfloor d\cdot(1-t) - t \right\rfloor$$
$$a[c,i] = \left\lfloor \frac{d^2-1}{4} \right\rfloor \left\lfloor 1+2t \right\rfloor + \left\lfloor \frac{p-1}{2} + p\cdot i \right\rfloor \cdot \left\lfloor \frac{d+1}{2} \right\rfloor + c$$

where t is either 0 or 1 for an X or Z-chain respectively. p is the parity of the algorithm, being $+1$ or -1 for left or right for X-chains and up or down for Z-chains. As can be seen, one can easily write a code to iterate through this set of closed form equations to obtain all pure errors on the data qubits, as shown in appendix C.3.

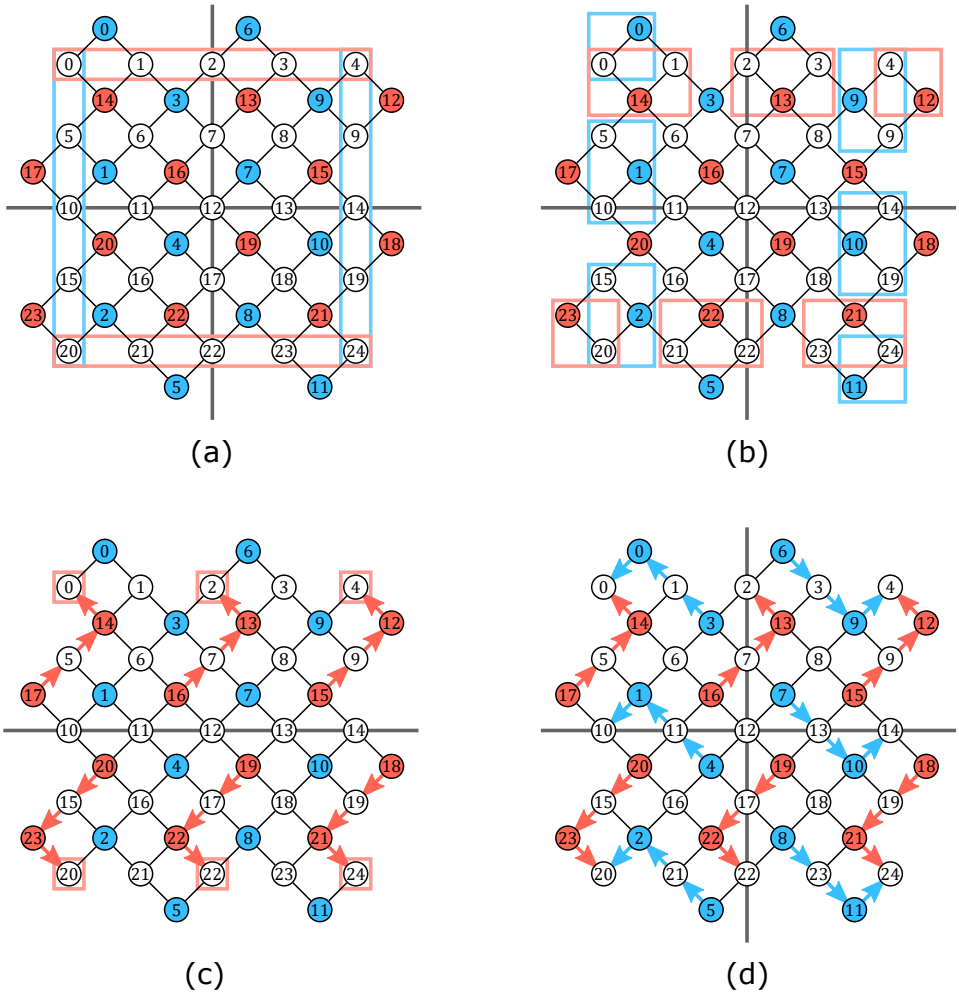


Figure 4.13: Example of the pure error decoder for the distance 5 surface code. (a) Outlining the 90 degrees point symmetry around data qubit 12 and showing the data qubits at the boundaries. (b) Showing that not all data qubits at the edge are needed and what ancillas can connect to which boundary data qubits. (c) A regular pattern for the top and bottom half. (d) The full 90 degrees point symmetric pure error decoder where the arrows denote the direction of the XOR-gate chains as in equation 4.16.

4.5. Conclusion

This chapter found that, for a first exploration of hardware decoders, the best option is to use a feed-forward neural network. This is the most basic neural network. If measurement errors are included in the error model, a recurrent neural network like an LSTM can be used. If the distance is larger, then the resulting spatial invariance can be exploited using convolutional neural networks.

The neural network decoder should be used together with a pure error decoder. This will take the ancilla syndrome and output the pure error. The neural network then only has to output the logical error of the pure error decoder. This logical error is the logical difference between the actual errors on the data qubits and the errors on the data qubits estimated by the pure error decoder.

This work improves upon the pure error decoder used in [52][51][50]. The pure error decoder introduced in this work exploits more invariances and is more regular. This means it is easier to implement in software and hardware. It will also aid a convolutional neural network due to the regularity.

The main aim of the following chapters is exploring the trade-off between the decoding performance of the neural network, measured by the pseudo-threshold, and the power, speed and area. The speed must be fast enough to keep up with the surface code cycle, which generates outputs at a 2.5 MHz rate. The power and

area can be optimized using the following parameters, also are shown in figure 4.14.

- The number of layers of the neural network. More layers means more delay, more power and more area. However, more layers might be needed to be able to perform the decoding.
- The sizes of the hidden layers of the neural network. A smaller layer size will reduce the decoding performance, but require less operations, thus saving time, area and power.
- The transfer functions. Transfer functions that are easier to compute will be faster in execution and consume less power and area. It could even be the case that a very simple transfer functions, such as the ReLU or the SQNL, can outperform the commonly used logistic functions.
- The number of bits used to represent the data. As less bits means less power, area and delay, this could be one of the most important parameters to explore.

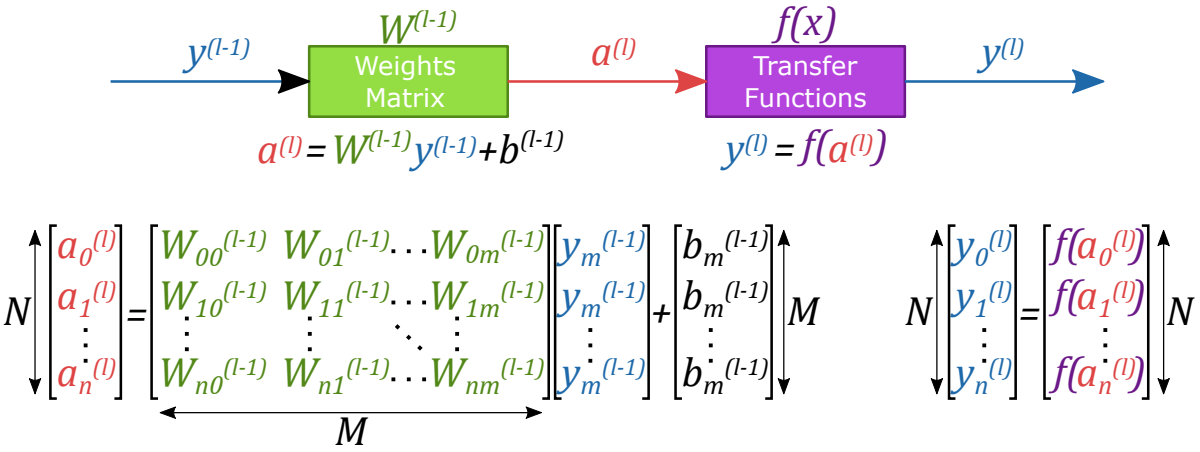


Figure 4.14: Figure showing a single feed-forward layer illustrating which parameters can be optimized. The first option is the amount of layers. The second option is the layer sizes, denoted by N and M in the figure. The third option is the type of transfer function, denoted by $f(x)$. The final option is changing the number of bits that represents each value shown in all vectors and matrices.

5

Design Space Exploration

This chapter will explore the design space of the parameters of the neural network for surface-code decoding and compare the decoder performance with MWPM and the work of [52][51][53]. As concluded in the previous chapter, the parameters that will be varied are: the number of layers, the amount of nodes in the hidden layers, the transfer functions and the number of bits used to represent the different values in the nodes (weights, inputs, outputs, biases). These parameters are expected to have the largest impact on the performance of the decoder and the power of the hardware. However, first the simulation setup is presented.

5.1. Simulation Setup

Due to the vast number of simulations that need to be performed to explore the design space, one important aspect is the simulation speed. As each simulation needs to include both the generation of the data set, the training of the neural network and the testing of the neural network, this is quite time consuming. This section will discuss the choices made to reduces the computation time and will show the final simulation setup.

5.1.1. Previous Setup

The work of [52][51] introduces the use of the pure error decoder alongside the neural network decoder. This will allow the neural network to only determine the logical difference between the actual logical error that has happened and the error that the pure error decoder identifies. The previous work calls this a high level decoder. This is contrary to a low level decoder, where the error on each physical qubit needs to be identified by the neural network. The mentioned work uses LSTM layers, even for the depolarizing model. This is in theory unnecessary, because no temporal data is present.

Outputs

The adopted approach reduces the output of the neural network to four classifications I, X, Y and Z. These can be outputted in two different ways. The first way is to have four nodes in the output layer, each corresponding to a different classification. This would be a one-hot representation. The target output is the correct output high and the others low. This is the typical classification used in neural networks and also used in [52][51].

Another way is to have two output nodes. The first node would signal an X-error, the second node a Z-error. If both outputs are low, no error is present (I). If both outputs are high a Y-error is present. This solution requires the least amount of output nodes, and thus weights.

The work of [51] also introduces a second neural network. However, this one is used for the circuit noise model to distinguish measurement errors from data qubit errors. Since this work assumes a depolarizing error model without measurement errors, this second neural network will not be discussed.

Inputs

The number of inputs to the neural network is also fixed. It is equal to the amount of ancilla measurements in the error syndrome for a given distance d. This is equal to $d^2 - 1$. As all the ancillas are measured along

the z-axis, the output of each ancilla will only be one bit. The bit that is sent to the neural network indicates whether or not an error has occurred, i.e. if the current ancilla output is different from the initial quiescent state output.

Data generation

In contrary to the work in this thesis, [52] first starts with a data generation step. The full data set is generated before training. The first step is to introduce errors on the data qubits with a certain probability. Next, the surface code cycle is run. The measurement outputs are then given to the pure error decoder. This will output a guess on the data error configuration. Next, the data error configuration will be compared with the actual generated data errors. From this the difference between the actual logical error and the one predicted by the pure error decoder is calculated. This is the target output of the neural network given the same ancilla error syndrome. Because multiple data qubit error configurations can exist for each ancilla error syndrome, the number of times that a logical difference occurs for each syndrome is saved. If enough error syndromes are sampled, the probability distribution is calculated for each syndrome. This means that for a given encountered ancilla syndrome, each encountered logical difference from the set I, X, Z, Y is divided by the total amount of times this ancilla syndrome was generated. These probabilities will be the target output for this syndrome. This syndrome will be the input applied to the neural network and the neural network will be trained to find the logical difference with the highest probability. Thus, the final data set will be a list of inputs (the syndromes) with four corresponding target outputs (the probabilities of the logical differences).

Training

The training was done after data generation. The input data was applied in batches of 1000 to 10000 syndromes, depending on the amount of data used for the particular distance the neural network is trained on. For larger distances, there are more possible syndromes, each with more possible data qubit configurations. This required larger batches to improve averaging per weight update. After one batch, the weights were updated until the the outputs of the neural network met the target outputs, up to the third decimal, within 95% for all samples. Training on a CPU with 28 cores at 2 GHz took between half a day to three days per network excluding data generation using a data set of at most 10^6 samples [52]. [51] observes that the decoder performance is best at the physical error rate used when sampling. For this reason, multiple data sets are used to train multiple neural networks.

5.1.2. Main Changes

The general structure is the same as the one as described above. The neural networks works in parallel to the pure error decoder and has to find the difference between the actual logical error and the one guessed by the pure error decoder.

Data Generation

A problem with the data set generation of the previous work is the chance of overfitting. Due to the use of a fixed data set, the neural network could overfit to this set and not generalize. Especially, since a probability distribution is sampled and used for training, if this distribution is undersampled, which can happen for less probable syndromes, the neural network will learn false data. Because the data can be generated infinitely, the data will be generated on the fly and applied in batches. This will always give true data and the neural network will learn the averages by itself, both due to the use of batches and by training long enough.

Another problem with the data set generation is that only for the surface code with distance 3 and 5 some syndromes are seen multiple times. For the higher distances, due to the vast number of configurations, even when sampling millions of syndromes, none were observed more than once, thus making calculating probability distributions useless. It will only limit the total available data and thus result in more overfitting.

One final downside is that the data set is quite large and will consume lots of memory. This means that addressing it will take more time, due to the scheduling of reading a lot of memory addresses. Having a large data set also is hard to generate as summing the data, and sorting it is hard to parallelize.

Training

The training in this work was done in a custom simulator running on an NVIDIA GeForce GTX 1080 Ti. This GPU has 3584 cores, and thus 3584 simulations were done per batch. Each simulation trains the same neural

network, but applies a different syndrome generated just before training. Because the decoder performs best at the physical error rate at which the data is sampled, the simulator searches for the pseudo-threshold when training, as this is the point we use to compare decoders and thus want to optimize. This is done by starting the physical error rate at the pseudo-threshold of MWPM and then updating the physical error rate every 10000 batches. The update is done by moving the physical error rate to a value which is guessed to be the pseudo-threshold of the decoder based on the logical error rate from the previous physical error rate. This is done until the logical error rate is at the physical error rate. This means that the decoder is operating at the pseudo-threshold. If this is not reached within 100 updates, the training is also terminated. This process of doing 100 updates of 10000 batches containing 3584 samples takes 10 minutes for distance 3 and just under a day for distance 9. It is hard to quantify how much faster this code is than the one used in the previous work, because this code was run on multiple GPUs in parallel on a cluster, performing multiple simulations simultaneously. Another problem is that the neural networks used in this work are smaller than those of the previous work, reducing the training time. However, if one single GPU is assumed on a similar sized network, this speed-up would be more than 10 times. Also, the neural network sees $\approx 3.5^9$ data points, reducing the risk of overfitting. Finally, the stopping condition is now based on the pseudo-threshold of the previous batches. This is because doing it this way, is the same as testing the pseudo-threshold on new data (not seen before by the neural network). Since this is the performance measure used, it is more appropriate for the problem, than compared to stopping based on the match to a data set in the previous work.

5.2. Final Setup

In order to run the final simulation setup used in this work, as shown in figure 5.1 on a GPU, the simulator had to be written. This simulator includes:

- Simulation of the depolarizing error model, introducing the errors with the chosen physical error rate on the data qubits.
- Simulation of the surface code, propagating the data errors through the surface code to find the ancilla syndrome. This follows the surface code cycles introduced in section 3.2.
- Simulation of the syndrome measurement on the ancillas.
- Simulation of the pure error decoder described in section 4.4.
- Calculation of the logical difference between the data qubit errors and the output of the pure error decoder.
- Simulation of the fully customizable and quantizable neural network, able to change all the parameters discussed in this work.
- Training of the neural network using backpropagation and the ADAM optimizer, discussed in section 4.2.1.

The full code is shown in appendix C and was written in C and CUDA. The main challenge was the large amounts of data flowing through the memory.

5.3. Layer Sizes

The first parameters that are varied are the layer sizes of the neural network. Because the input and output layer sizes are determined by the number of ancilla qubits and the encoding of the output to either 2 or 4 outputs, only the hidden layer sizes can be varied. However, before the sizes of the layers can be varied, first the number of layers must be determined.

As stated in section 4.1.3, a single hidden layer already enables the neural network to map any arbitrary non-recurrent function, provided that this hidden layer has enough nodes. However, having more layers might give better performance for a lower total number of weights and nodes, thus consuming less power and area.

If a large hidden layer is used, this also means that the summation of the dot product between the weights and the outputs of the previous layer is larger. Since summing cannot be fully parallelized and the best parallelization of a sum with N entries has a depth of $\log_2(N)$ [32], this can be a speed bottleneck. Finally, having

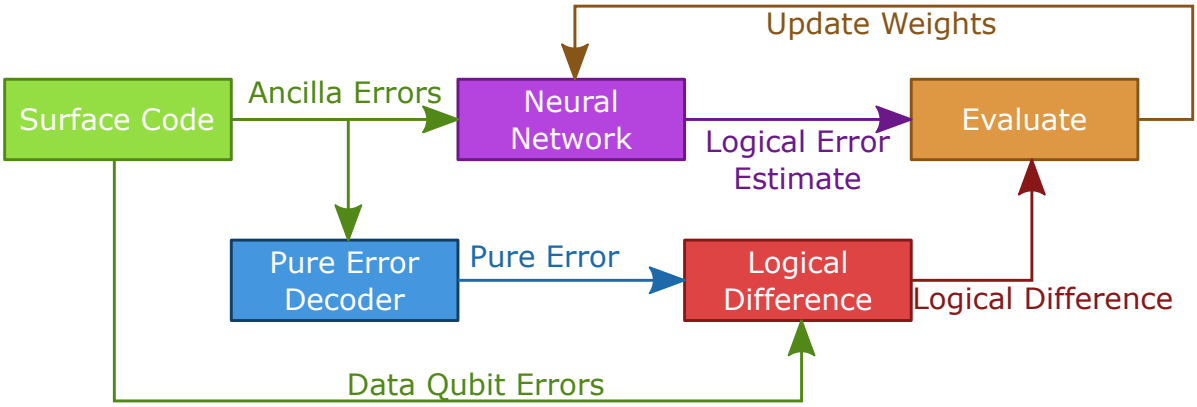


Figure 5.1: Figure showing the final simulation setup for training the neural network based on data generated by the surface code simulator. All the code was written in C and CUDA and runs on an NVIDIA GeForce GTX 1080 Ti. The performance is calculated by de evaluation block, which keeps track of the logical error rate and compared it to the physical error rate.

larger layer sizes will mean that the total maximum sum can be larger, thus requiring more bits to represent the sum. Again this leads to more area, power and delays.

Previous work (figure 9 in [51],) shows that two hidden layers gives better performance than having only one hidden layer. This holds even if the total amount of nodes is less than used in one layer. However, having more than two hidden layers does not give that much more performance and might also make it harder to train. For these reasons, this work will also cover neural networks with two hidden layers.

Because more recent work [53] shows that for larger distances it might be beneficial to go to a more convolutional style neural network, this work will cover simulations for surface codes up to distance 9. This also makes it easier to compare to the earlier work [52] and [51].

First, we varied the layer sizes while using floating point arithmetic and the logistic sigmoid function, as these were also used in earlier work. This will result in a better comparison and will also give a good starting point for continuing to the other parameters.

5.3.1. Simulation Results

As discussed the first simulations were done on the smaller distances of in this case 3,5 and 7. The distance 9 simulations were not done for this first sweep, because they still take a long time even with the sped up simulations. The choice to skip the distance 9 was made as the trends are already clear from the first 3 distances. This distance will be swept for the next sweeps. In these sweeps the layer sizes are also swept.

The used neural networks have two hidden layers, whose sizes are varied independently of each other. For each sweep, the pseudo-threshold the neural network decoder obtains after training is compared with the pseudo-threshold of the MWPM (Blossom) decoder for the same distance. The results of the sweeps are shown in figures 5.2, 5.3 and 5.4 for distances 3, 5 and 7, respectively. In these figures the pseudo-threshold is plotted on the y-axis compared to Blossom. The data is plotted twice in two different ways. The top plots show the first hidden layer size varied with a constant second hidden layer size. The bottom plots show the second hidden layer size varied with a constant first hidden layer size.

Distance 3

Figure 5.2 shows the simulation results of the hidden layer size sweeps of a neural network with 8 inputs, 4 outputs, 2 hidden layers and sigmoid functions as transfer functions. The top plot shows the pseudo-threshold (y-axis) of the decoder when varying the size of the first hidden layer (shown on the x-axis), while keeping the second hidden layer size constant. The different lines represent different constant second hidden layer sizes. One thing that is clear is that the size of the first hidden layer is more important than that of the second hidden layer, in determining the pseudo-threshold. The top figure shows lines that follow the same

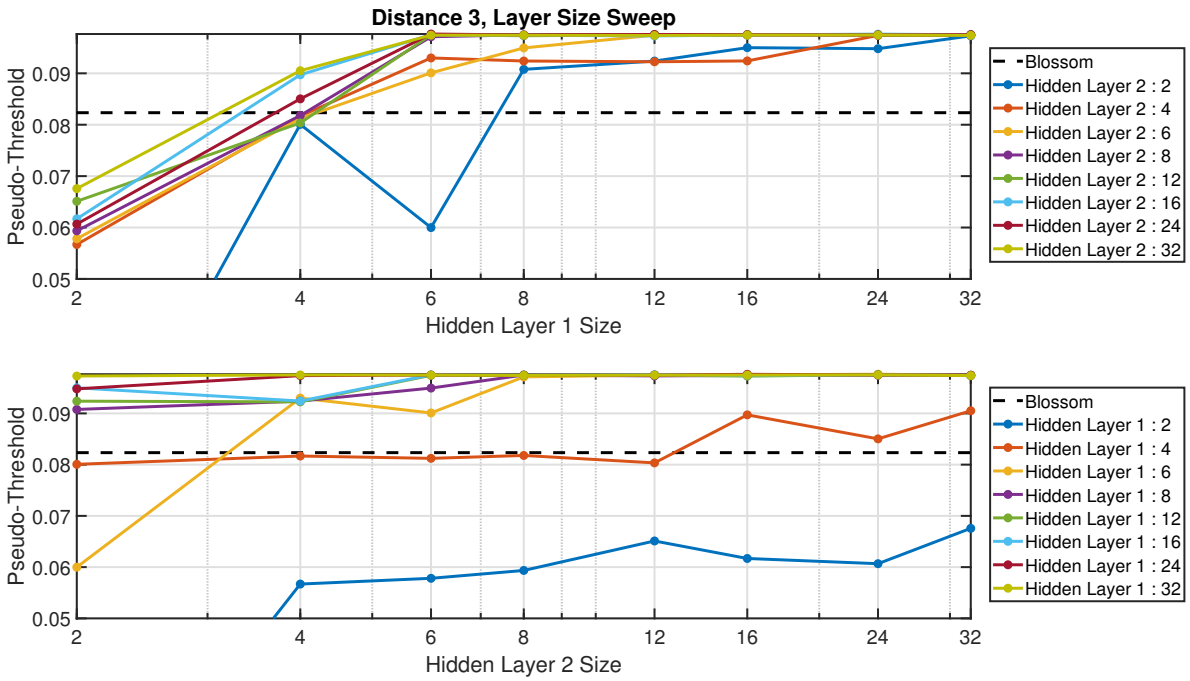


Figure 5.2: Simulation results for the hidden layer size sweeps for a surface code of distance 3. The top plot shows the sweep of the first hidden layer with constant second hidden layer sizes. The bottom plot shows the same data, but here the second hidden layer size is varied for constant first hidden layer sizes. In both figures the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

trend, where a smaller first layer size will reduce the pseudo-threshold and saturates around 8 nodes, for all second hidden layer sizes.

This can also be seen in the bottom plot, where all lines are more or less constant with respect to the second hidden layer size. Here it is also observed that the neural networks with first hidden layer sizes larger than 8 are not affected as much by the second hidden layer size. However, a size of 4 already performs as well, or better, than Blossom.

If these small neural networks with layer sizes in the range of 4-8 are compared to those in [52], with approximately 128 nodes, this is a significant reduction in complexity.

Distance 5

An even more pronounced dependence from the size of the first layer is observed for distance 5 with respect to distance 3, as shown in figure 5.3. The minimum size to outperform Blossom is here a size of the first layer of 24 for any size of the second hidden layer.

Distance 7

The same trends that were seen in distance 3 and 5, are also seen with the distance 7, as shown in figure 5.4. Here the minimum layer size needed is between 96 nodes.

5.3.2. Conclusion

It is clear is that two hidden layer is enough to get performance better than Blossom. For distances 3, 5 and 7, the pseudo-threshold is better than Blossom, for sizes that are smaller than found by previous research [52].

The distance 9 sweeps were skipped for this sweeps, as they take a long time and were not deemed necessary to improve the conclusion drawn from the sweeps presented for the smaller distances.

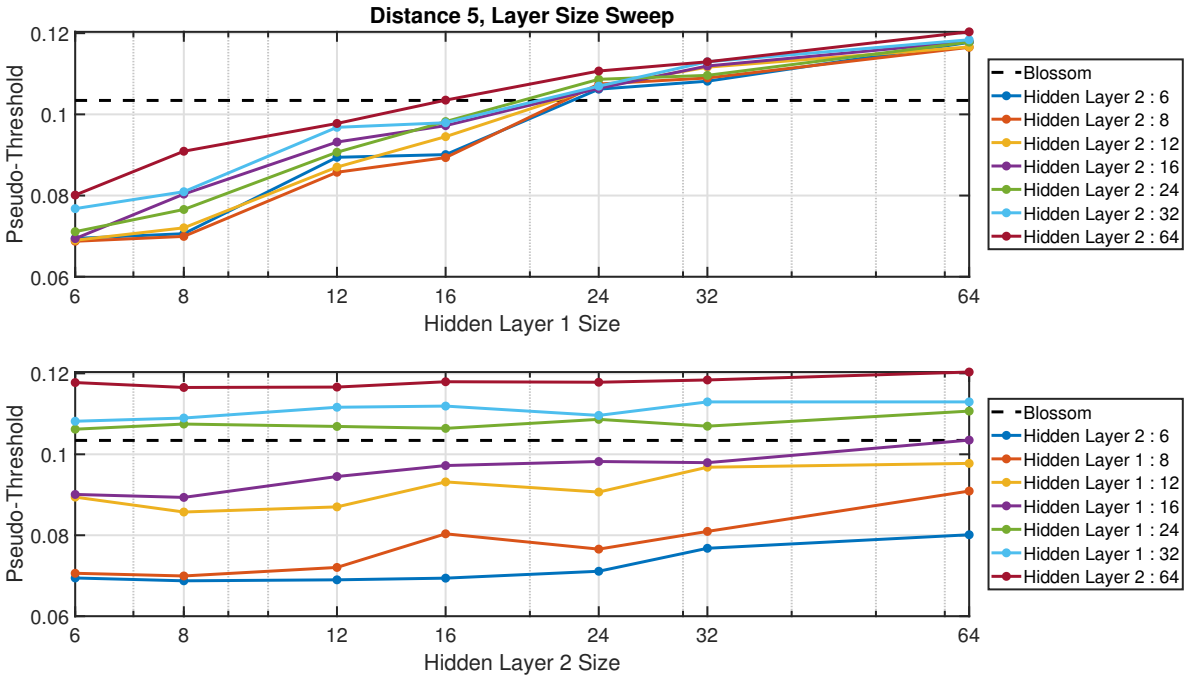


Figure 5.3: Simulation results for the hidden layer size sweeps for a surface code of distance 5. The top plot shows the sweep of the first hidden layer with constant second hidden layer sizes. The bottom plot shows the same data, but here the second hidden layer size is varied for constant first hidden layer sizes. In both figures the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

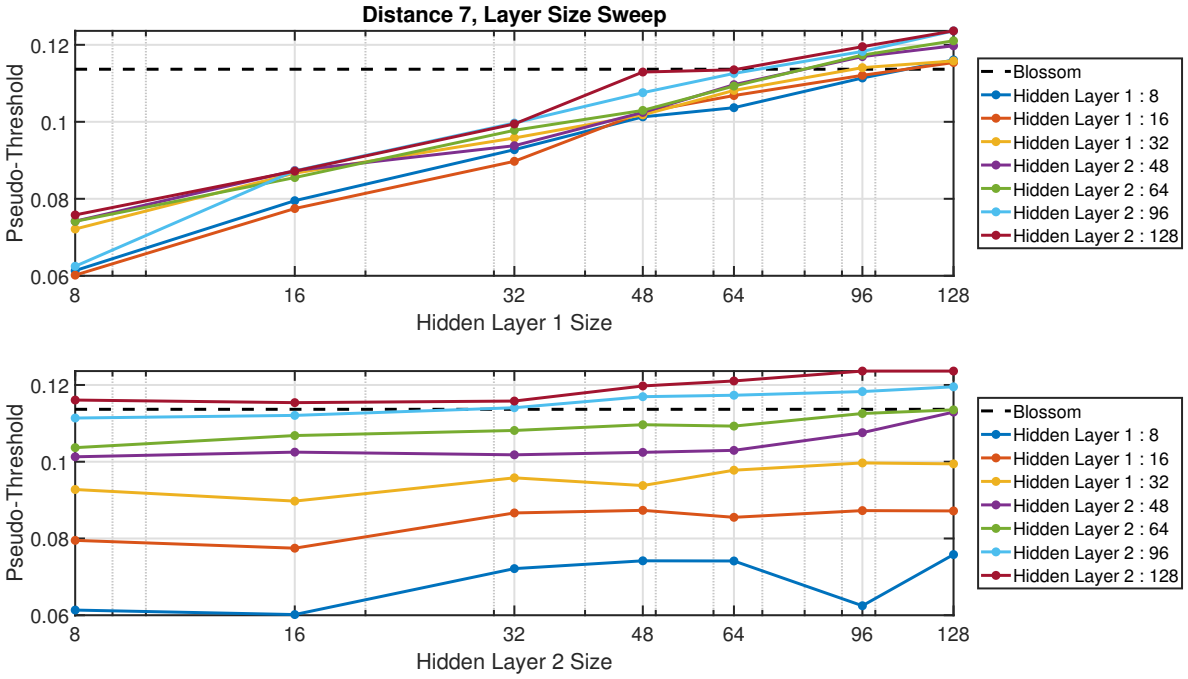


Figure 5.4: Simulation results for the hidden layer size sweeps for a surface code of distance 7. The top plot shows the sweep of the first hidden layer with constant second hidden layer sizes. The bottom plot shows the same data, but here the second hidden layer size is varied for constant first hidden layer sizes. In both figures the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

Among the hidden layers, the first one is a lot more important in determining the performance than the second layer. It also does not make sense to make the second layer larger than 16, since, even for smaller sizes, the performance is already good enough. However, when sweeping other parameters, the extra nodes might be needed to maintain this performance.

One final note is that, looking at the figures for distances 5 and 7, the pseudo-threshold did not settle to a constant value for a larger size of the layer. Thus, it is a good idea to simulate larger layer sizes when sweeping the next parameters.

5.4. Transfer functions

As this work focuses on the power and speed of the decoder, the most important requirement on the transfer functions is their complexity in implementation and computing in hardware. This makes the ReLU a very attractive solution, as it does not require any complex computation, even in the case of the leaky ReLUs. This is because the only hardware needed is a multiplexer, that chooses either the original input or a scaled down version of the input depending in the sign of the input. If the scaled down version is a negative power of 2, this can simply be done by appropriately routing the bit in the digital word.

The SQNL is also promising, as it more closely resembles the logistic functions. It is also relatively cheap to compute, only involving some multiplications, additions and comparisons. Compared to other functions containing exponential, logarithms, square roots and divisions, this saves a lot in terms of power, delay and area. It might be better for the SQNL to encode the output in two bits as described in the beginning of this chapter. This is better suited to the SQNL and tanh functions, as it is on at both negative and positive inputs, instead of being on for positive input or off for negative input like the sigmoid and ReLU. This means that an output of -1 can correspond to one situation, while an output of +1 corresponds to the other. In this case -1 means no error (I) and +1 means an error (X or Z) occurred.

For both the ReLU and the SQNL, the leakage parameter α can also be swept. Having an α more close to one will decrease the non-linearity of the function and might decrease the performance. However, having a value of α closer to 0 will decrease the derivatives and the number bits available to the next layer. This is because a lower $alpha$ will scale down the negative outputs. Given a certain amount of fractional bits, this will thus limit the precision for negative outputs.

5.4.1. Results

The results of all the sweeps are shown in appendix A. However, the most important results are covered in this section. As found when sweeping the layer sizes, the second hidden layer size was less important. For the distance 3 code a second hidden layer size of 8 is shown, because a layer size of 16 was too large and all networks performed more or less equally. This made it hard to draw any conclusions from this data. The data can be seen in the appendix A. For the other distances the results shown here have a second hidden layer size of 16. The other values are shown in the appendix.

The results are shown for the sigmoid and ReLU functions for neural networks with four outputs and for the tanh and SQNL with two outputs as discussed in the previous section. The pseudo-thresholds are compared to the performance of Blossom.
Figure 5.5 shows the results for distance 3. All of the simulations outperform Blossom and the maximum is already achieved with a hidden layer size of 1. It appears as if the networks with two outputs outperform the networks with four outputs. This would be a positive results, as this means that the total amount of weights between the second and output layer are halved.
When looking at the other distances (5, 7 and 9 in figures 5.6, 5.7 and 5.8 respectively), this trend holds. Furthermore, in all cases the SQNL performs equal or better than the tanh, which is an advantage for the implementation.

We can also see that the performance starts to saturate for larger first hidden layer sizes as we expected. However, the performance of the distance 9 surface code does not reach the performance of Blossom for the simulated hidden layer sizes. This could have three consequences:

1. The neural network layers should be larger. This is not wanted, as this increases the dot product hard-

ware in terms of area, power and speed due to the larger sum.

2. The number of layers should be higher. This could be beneficial if this also means that the sizes of the layers will decrease. However, this will mean that the total path between input and output will be longer and more transfer functions are needed compared to dot products. This might increase the power, as the transfer functions (besides the ReLU) cost more than simple multiplications and summations. The longer delay might also be too long to complete the decoding in the period of a surface code cycle. This might then require pipelining, meaning that when using non-clifford gates the surface code cannot be corrected at each cycle, thus introducing more errors.

3. Change the architecture to a convolutional neural network, with kernels shaped like distance 3 kernels, as done in [53]. In this work, the neural network is not completely convolutional, as pre-trained distance 3 neural networks are used as the first layer. This could be further optimized by also using the 90 degree rotational invariance created by the pure error decoder used in this work. This means that not only the kernels are shifted over the surface code, but also rotated four times by 90 degrees, enabling even more weight reusing.

As the third option is the most promising, especially if even larger networks are considered, this work will not delve into the other two options and, in the following, we will keep using a two hidden layer neural network with a maximum layer size of 256 for the distance 9 surface code.

Another option that one might consider is using a SQNL layer for the output layer, but ReLU for the two hidden layers, as this dramatically reduces the number of the more expensive SQNL functions to only two. However, when this was tried, it did not increase the performance compared to the ReLU alone for larger layer sizes. It only improved the performance for very small second layer sizes of 2 or 4.

5.4.2. Conclusion

From this results it can be concluded that the ReLU does not perform as well as the other functions, and that the SQNL does perform equal or better than the tanh and the sigmoid function. It is nice that for the SQNL and tanh simulations only two output nodes where used This cuts the number of weights between the second hidden layer and the output layer in half as compared to the results in the layer size simulations.

Another point is that besides $\alpha = 2$, which does perform worst, there is no clear optimum value of α that works best in all situations. Thus, for the next section a value of $\alpha = 16$ is used to limit the number of bit needed to present the output.

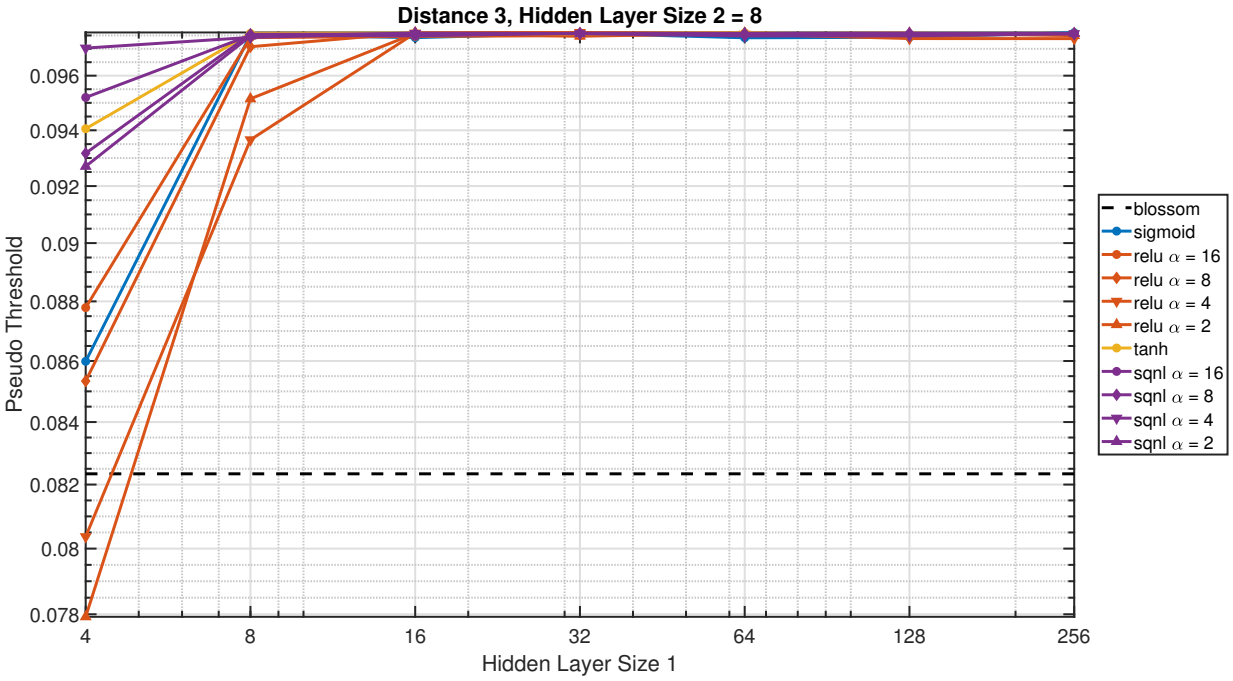


Figure 5.5: Simulation results for the transfer function sweep for a surface code of distance 3. For this plot the second hidden layer size is 8. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom ($p_{th} = 0.08234$) shown in the black dashed line.

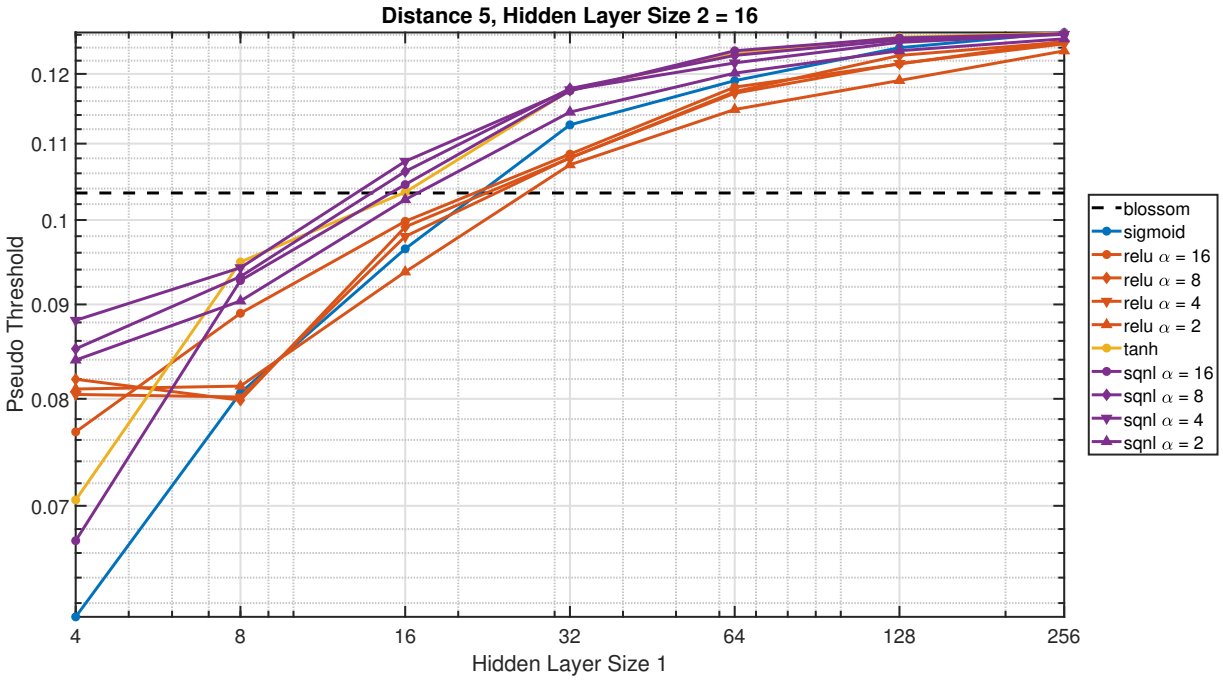


Figure 5.6: Simulation results for the transfer function sweep for a surface code of distance 5. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom ($p_{th} = 0.10.343$) shown in the black dashed line.

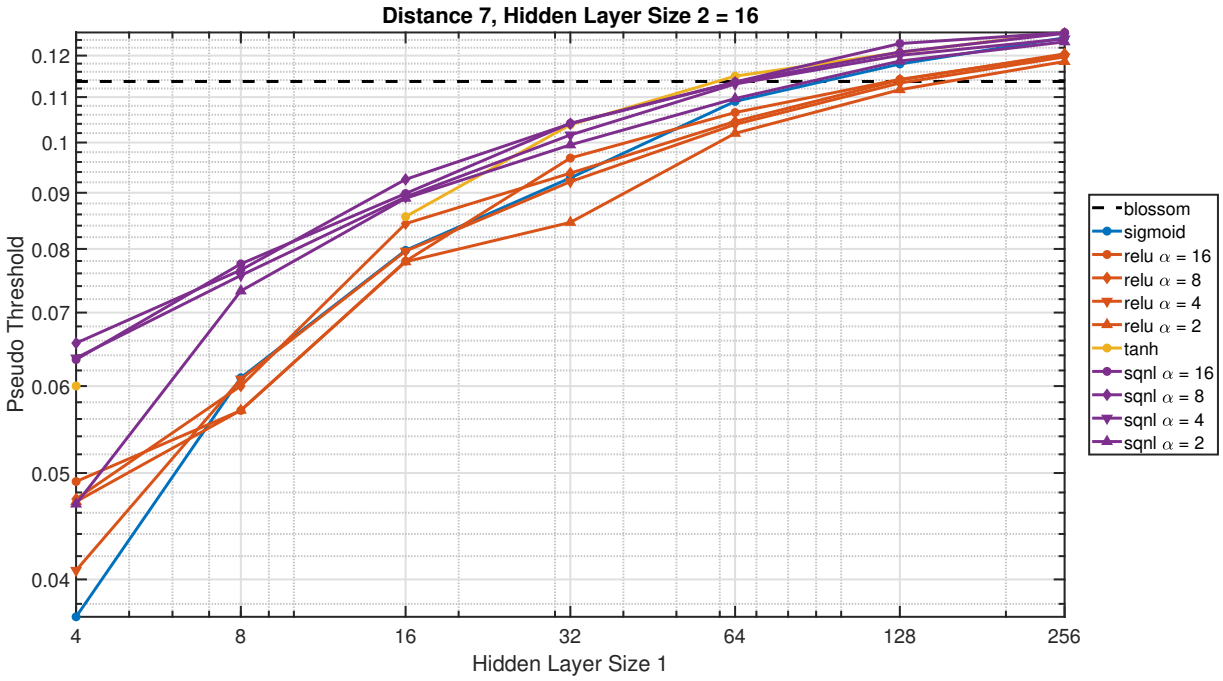


Figure 5.7: Simulation results for the transfer function sweep for a surface code of distance 7. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom ($p_{th} = 0.11.366$) shown in the black dashed line.

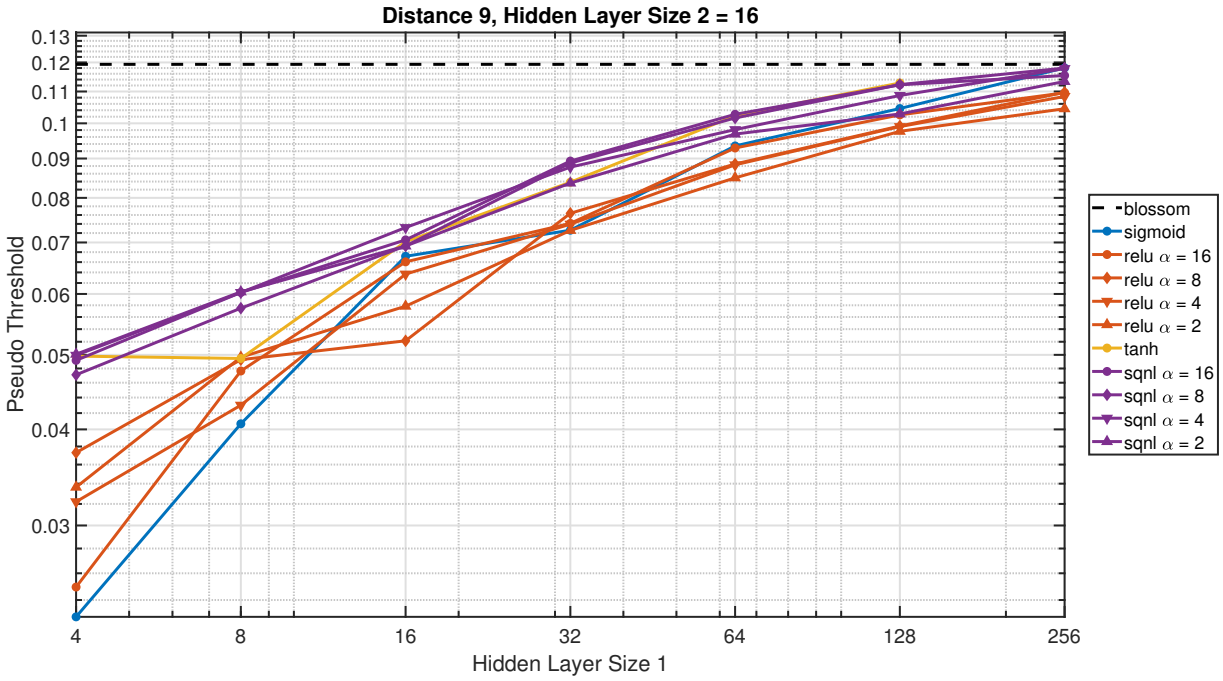


Figure 5.8: Simulation results for the transfer function sweep for a surface code of distance 9. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom ($p_{th} = 0.11932$) shown in the black dashed line.

5.5. Quantization

Next is moving from floating point arithmetic to include fixed point arithmetic and change the bit vector sizes. In other words, how many bits do we need to represent each data value when quantizing? The most interesting points to look at when comparing these networks to Blossom, are the points where the neural networks just outperform Blossom. This will give a good visual reference to how much the performance decreases when going to smaller bit vectors. However, as the distance 3 always outperforms Blossom for the tested sizes, and the distance 9 always performs worst, for these the first hidden layer sizes are chosen to be 8 and 256 respectively. For the distance 5 and 7, first hidden layer sizes of 32 and 128 were chosen. In all simulations the second hidden layer sizes is set constant. Because the SQNL function is used the output layer is set to a size of 2.

As a first exploration, this work will focus on the situation where all the data is represented by the same number of fractional bits. Thus the weights, the bias and the outputs of the transfer function are all quantized the same way. It might be that the weights are less important than the outputs or vice versa. This is however outside the scope for this discussion and thus a topic for future research.

Recent research has been more and more focused on moving to less bits for neural networks. However, most research has been mainly done for 8 and 16 bits, because this is supported by most libraries and binary nets [36][56], most using the large existing convolutional neural networks, such as AlexNet [24] and VGG [44]. There is also the research from [29], which focuses on low bit fixed point arithmetic. This work [29] found, that the optimum trade-off point for power and performance was around 4 bits. This work also highlights the fact that one should train the neural network with the same amount of bits as it should eventually run at, and also to keep in mind how many transfer functions are used. In other words, keep in mind the hardware, when doing the exploration and training.

Since this work searches for ultra low power performance, it is preferred that the networks work at less than 8 bits. The binary nets would be great, as this simplifies the hardware significantly. However, when it was tried using the results found in the previous sections, the neural networks were too small to get enough generalization and the transfer functions discussed before do not apply to binary nets. Thus, although it might be a viable option, it does not fit the scope of this project and could be done in future research.

The first step to move from floating point arithmetic to fixed point arithmetic was to train the neural network in floating points, and round to a certain number of bits when testing. This did however not degrade the performance even if no fractional bits was used, which pointed out that something was going on. As shown in figure 5.9(a) the initialization of the weights before training was in such a way that all the weights were smaller than 1. However, due to training all the weights were increasing as shown in 5.9(b). This means that after quantization as shown in figure 5.9(c), the assumption that all the weights were still less than one was false. In reality, they were all bigger than one. This meant that they were represented by more bits than was anticipated. In principle it does not matter what the maximum size of the weights is, however one needs to determine the amount of integer and fractional bits used to know the amount of bits needed. For ease of calculations this work tries to keep all the weights in magnitude smaller or equal to 1. Thus requiring only the fractional part and a sign bit.

One straightforward solution is to clip the weights to have a magnitude less or equal to 1. The outputs of the transfer functions can also be clipped to unity. However, another way to do this is to set the leakage of the SQNL function to zero ($\alpha = 0$). This was done for this research.

Clipping does give a problem if it is done after training, as this will set all the weights with an absolute value larger than one to ± 1. This is visually show in figure 5.9(d). These weights were trained to give the neural network the correct function, however this clipping will alter the performance. It was found after trying that simply clipping during training also did not have the desired effect. This is because the weights were still growing to ± 1 and thus did not optimally and evenly use all the quantization levels between 0 and ± 1.

One technique that can be used is to add a regularization term to the cost function that pushes the weights to zero. This technique called weight minimization was discussed in section 4.2.3 and tries to minimize the L_n norm of the weights. In this work, an L_2 norm was chosen. This means the sum of the squares of the absolute

values of the weights are minimized, $\min\left(\sum|w|^2\right)$. This will push the weights back to zero as shown in figure 5.9(e).

Using all of the techniques mentioned (weight minimization regularization during training and clipping and quantizing after training) did perform the best so far. However, it did not perform as well as Blossom for bit vectors lower than 6 and not even close to performing as well as the floating points. Our hypothesis was that, due to the small layer sizes, having a large deviation between the floating point value and the quantized value gave to much errors and altered the function of the neural network too much. This is relevant especially since the literature stated that it is optimal to train with the same quantization as was done when running. However, simply using quantization during training, the neural network did not converge, which is probably due to the small layer sizes. The function of the neural network changed too much if one of the weights was rounded to a different quantization level to converge.

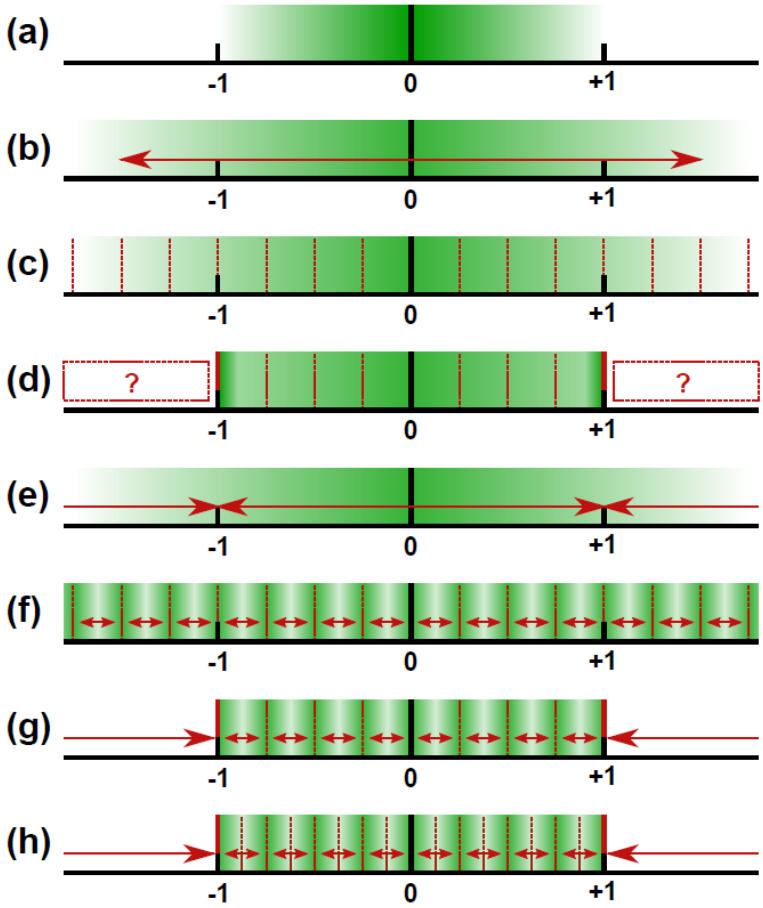


Figure 5.9: (a) Initialization of the weights to values smaller than 1. (b) Spreading and growing of the weights during training. (c) Quantization of the weights without a maximum value. (d) Clipping at the end of training, setting a lot of value larger than 1 to 1, changing the function of the neural network. (e) Using weight minimization. (f) Using soft quantization regularization. (g) Combination of both regularization terms during training, and quantization and clipping after training. (h) Quantizing at more one extra bit compared to the regularization term.

The found solution to this problem was to take inspiration from the weight minimization regularization technique. However, instead of just minimizing the total weights, the distance between the weights and the nearest quantization level was also minimized: $\min\left(\sum|w|^2+\sum|w-w_q|^2\right)$. This allowed us to keep the values continuous in floating point when training, but converge to the quantization levels smaller than one. The relative strength of both techniques can also be varied, by multiplying one of the regularization terms by a factor r. The idea is shown in figure 5.9(f) and the combination with the other techniques in 5.9(g). It did however occur that some of the weights did not properly reach the quantization level. This can be fixed by in-

creasing both the amount of quantization levels and the quantization levels used by the regularization. It was however found that having more quantization levels than quantization levels used in the regularization, gave similar performance. This meant that attracting the weights to sparser quantization levels than they would eventually be quantized on was also feasible. This allowed us to train to only a few bits and than quantize at a lot of bits and finding the optimum.

5.5.1. Results

All the simulation results are shown in appendix B. However, the most important results are highlighted in this section. One important note with these results is that the number of bits shown is not the actual number of bits needed to represent the weights. This is because both one and negative one are included. In a 2's complement representation [32] this is not possible. However because the neural network will learn the actual 2's complement just as easily, for simulation simplicity and as a proof of concept having both $+1$ and -1 was used. The results can still be used as general guidelines for future research.

Figures 5.10, 5.11, 5.12 and 5.13 show the results for the distances 3, 5, 7 and 9, respectively. The sizes are those discussed at the end of the previous section about transfer functions, i.e. for distance 3 a first hidden layer size of 8 was chosen, 32 for distance 5, 128 for distance 7 and 256 for distance 9. These numbers were chosen, because they were just above the pseudo-threshold of the Blossom decoder for the same distance. All figures show the pseudo-threshold on the y-axis compared to the Blossom decoder and the best performing floating point neural network (often using larger layer sizes). In most plots this line is the top of the y-axis shown by the thick line. The x-axis shows the amount of fractional bits that were used to quantize all the values. Here the weights are also clipped to be less than ± 1 and due to the fact that there is no leakage used on the SQNL, this is also limited to ± 1.

The different lines show different quantization settings on the regularization terms. Again, this regularization assumes a two's complement where both values $+1$ and -1 are present. As it can be seen from the figures, the quantization regularization term does make a difference, but is not as monotonic as might be expected. It mainly depends on how well the initialization can converge to these quantization levels. This is especially true because the actual quantization is done after training. It can be seen that this quantization does behave the same for all distances and regularization terms. For every distance, there is a certain number of bits below which performance starts dropping. This value is approximately the $\log_2$ of the layer size. This can be even better observed in the data from the appendices. This is probably due to the fact that the individual weights are smaller if more inputs are summed together. This is due to the fixed range of the SQNL function. Since its parabolic input range is ± 1, it can be expected that the sum of weights is also around ± 1 and, consequently, the expected value of each weight is around $1/(\text{LayerSize})$. Such a number needs to be represented by approximately $\log_2(\text{LayerSize})$ fractional bits. In practice the values will be slightly larger than $1/(\text{LayerSize})$ as some values are positive and others negative, meaning that they can be larger and still sum to 1. Thus it might be beneficial to either scale the SQNL function and/or the maximum weight value according to the number of inputs for each layer, or use small bit floating point arithmetic. A final solution could be to use a convolutional style neural network, where each kernel has less inputs, and thus can have larger weights.

5.6. Conclusion

This chapter presented the simulation setup used in this work and explained which GPU code was written to run these simulations. A summary of the results is outlined in this section.

As can be seen in the last section, although the performance drops slightly, the layer sizes and transfer function determined in the first two sections did hold for the quantized neural networks. The training with small layer sizes proved difficult when using quantization and clipping during training. However, when using a soft quantization, and a weight minimization term in the regularization, this problem was mitigated. The final results shown are with an SQNL function with no leakage to clip the outputs. The 2's complement representation of the bit vectors was used. However, it was not completely accurate as both $+1$ and -1 were present, while in a correct 2's complement this does not hold.

In general, slightly less than $\log_2(\text{LayerSize})$ bits were needed in the final quantization before the performance started to drop. The soft quantization bits (being the number of bits used in the regularization term) did not

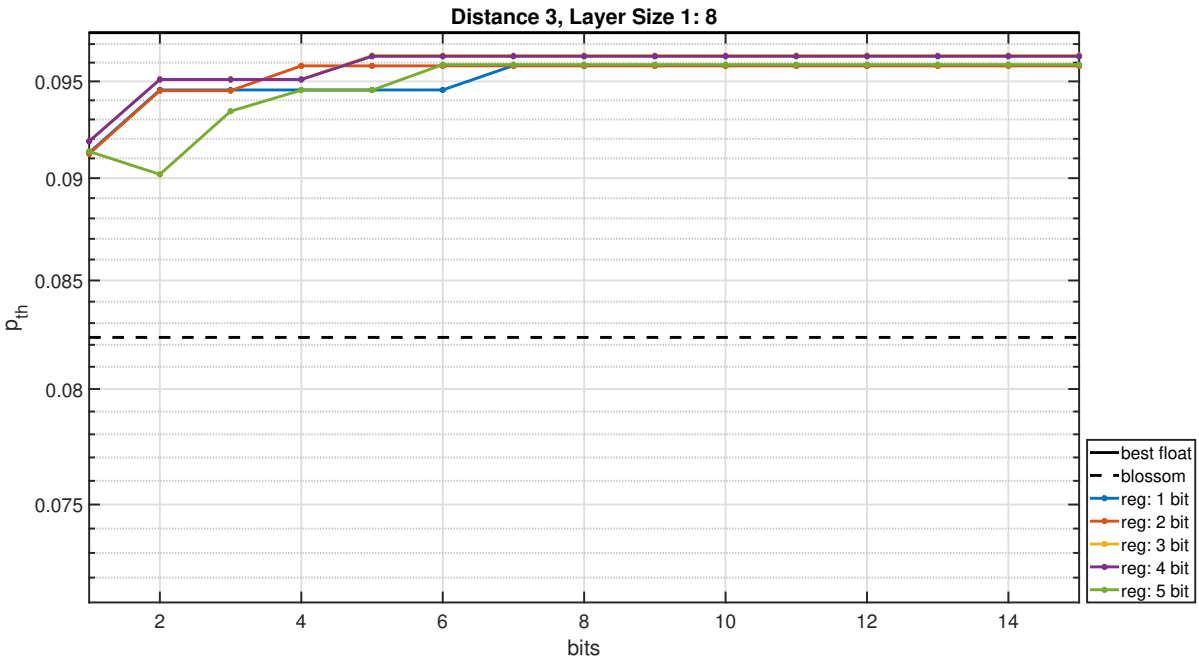


Figure 5.10: Simulation results for the quantization sweeps for a surface code of distance 3. This is done on a neural network with a first hidden layer size of 8, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

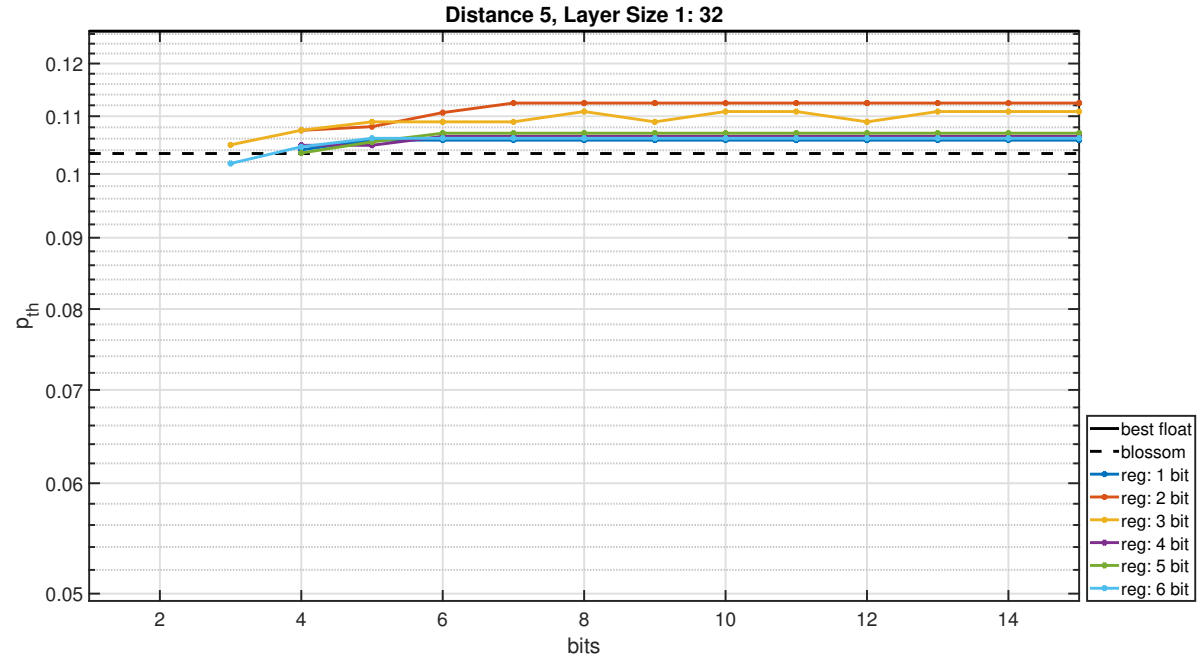


Figure 5.11: Simulation results for the quantization sweeps for a surface code of distance 5. This is done on a neural network with a first hidden layer size of 32, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

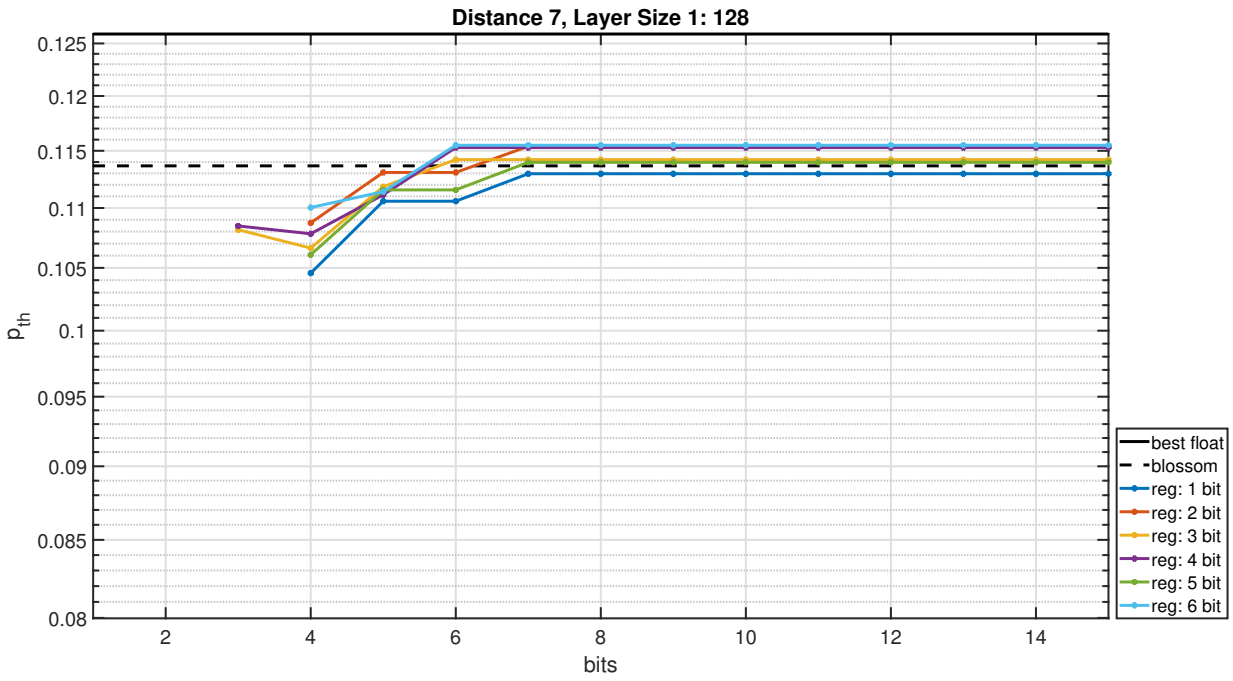


Figure 5.12: Simulation results for the quantization sweeps for a surface code of distance 7. This is done on a neural network with a first hidden layer size of 128, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

show a monotonic dependence on the performance. This is probably because the final performance mainly depends on possibility to completely convergence to the nearest quantization levels from the initialization point. If less bits need to be used than $\log_2(\text{LayerSize})$ several approaches are possible:

- Scale the input range of the SQNL function to allow a larger input sum.
- Change the maximum value of the weights, i.e. clip to a smaller value. This will mean that the most significant bits will be removed.
- Use floating point arithmetic with a small amount of bits.
- Use a different neural network architecture, such as the convolutional neural network. This will gives less inputs for each kernel, allowing the use of less bits per weight.

From these solutions, the convolutional neural network seems the most promising one. This is because it already has been discussed that this is a good solution also for different reasons, such as the exploitation of translational invariances in the weights and the reduction of free parameters. If the distance 3 kernel is used, as in [53], and the same number of bits can be used as the quantized distance 3 results suggest, this would mean that 3 bits are already enough.

A summary of the final results of this chapter is shown in table 5.1. This table shows the performance of Blossom, [51] against the best floating point performance shown in this work and the quantized neural network with the lowest number of bits that outperforms Blossom. This configuration is called Optimum Fixed.

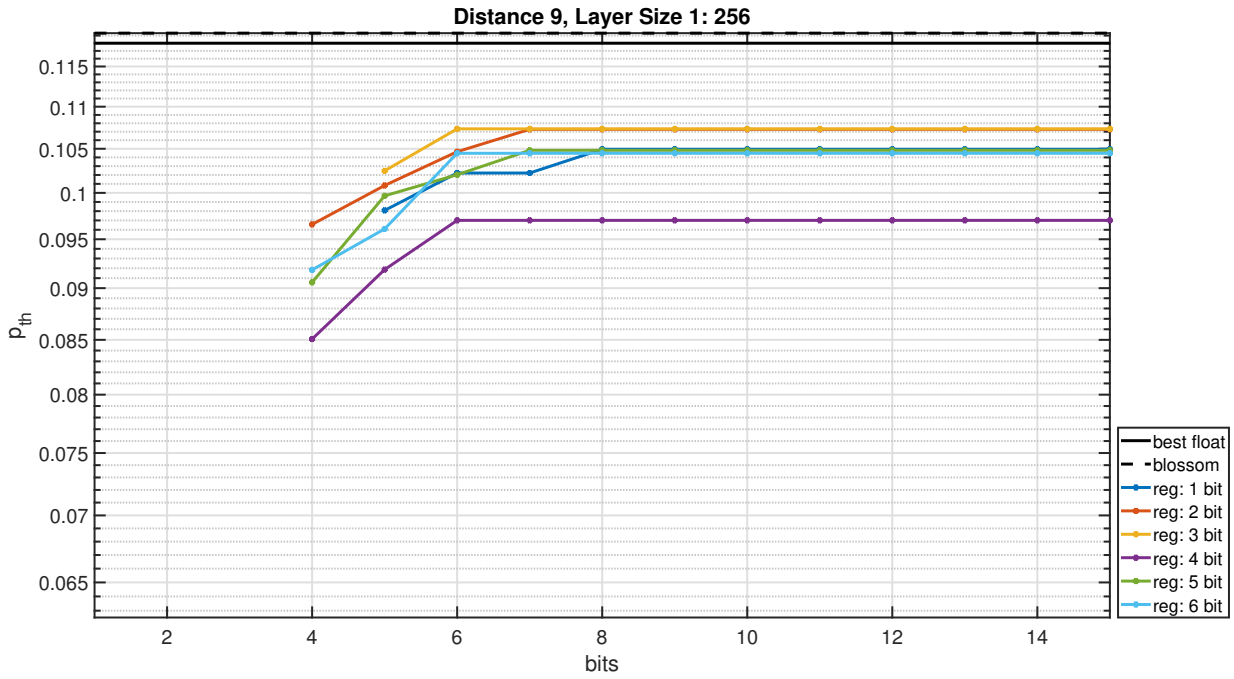


Figure 5.13: Simulation results for the quantization sweeps for a surface code of distance 9. This is done on a neural network with a first hidden layer size of 256, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

Decoder	$d = 3$	$d = 5$	$d = 7$	$d = 9$
Blossom	8.234 %	10.343 %	11.366 %	11.932 %
[51]	9.815 %	12.191 %	12.721 %	12.447 %
Best Float	9.762 %	12.659 %	12.592 %	11.800 %
Optimum Fixed	9.510 %	10.898 %	11.549 %	10.734 %

Table 5.1: Performance (pseudo-thresholds) of this work compared to the Blossom decoder and the work of [51]. Two pseudo-thresholds are denoted for this work. One being the best floating point performance using the SQNL function. For these networks the best results were always obtained by taking the largest network tested. This network has two hidden layers, the first having 256 nodes the second 16 nodes. The output layer has two outputs, one for logical X and one for logical Z-errors. The second performance reported, denoted by optimum fixed, is the one performing just better than Blossom (or for distance 9, the best found), after being quantized. The chosen networks configurations are all with a second hidden layer size of 16. This was for simplifying the search, because as discussed, varying this did not influence the performance a lot. The first hidden layer sizes are in order of distance: 8, 32, 128 and 256. The number of bits chosen is in the same order: 2, 5, 6, 6. The reported performance is that of the best performing regularization term at that number of bits.

6

Hardware Estimates

All the work up to this point, has been done with the target to minimize the power of the neural network decoder hardware, while keeping the decoding performance on par with the best known software implementation. However, one important requirement is that the decoder can keep up with the estimated 2.5 MHz rate of the surface code. Estimation of the neural network delay and power can only be obtained by synthesizing the decoder in hardware. The pure error decoder is not included in the analysis of this chapter, because the couple of xor-gates that this decoder needs will not be the dominant factor in the delay, power and area.

Some power, area and delay estimates on this will be presented and discussed in this chapter. Next, a discussion on the cryogenic hardware is also presented.

6.1. Test Setup

Before the performance of the hardware can be tested, a hardware description is needed. Because for this work the delay is the most important requirement, a fully parallel implementation is written. The tested hardware is as depicted in figure 6.1 and appendix D.

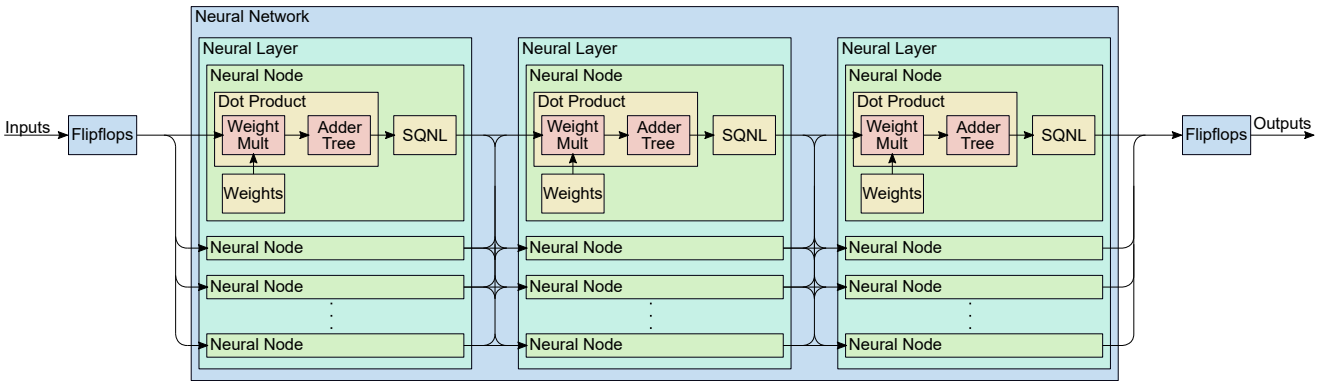


Figure 6.1: Hardware block design of the VHDL code in appendix D. Configurable VHDL code to change the layer sizes and number of bits. Weights are randomly generated and hard coded. Best performance, better suited for FPGA than ASIC.

The fully parallel neural network consists of three layers. The number of inputs depends on the distance, the first hidden layer size and fixed point bit vector size are the same as the optimum of the last chapter, the second layer size is 16 and the number of outputs is 2. Each layer is made up of nodes, consisting of an element-wise multiplication followed by an adder tree. The output of the adder three is fed through the SQNL function. And finally, this output is fed to all the inputs of the next layer.

To get the critical path delay, the inputs and outputs of the neural network are buffered with flip-flops and no other flip-flops are present in the design. This means that the whole network only consists of combinational logic.

To get a power estimate that does not depend on the way that the weights are saved, each weight is a constant random number, where each bit is routed to V_{ss} and V_{dd}. Thus the reported power estimate excludes any additional memory to save the weights. Such an hardware implementation is more suited to an FPGA, because here the hardware is re-configurable and the thus the weights can be adjusted. If it were to be implemented on an ASIC, some re-writable memory is needed.

All design were verified by visual inspection, because the larger synthesized designs were too large to simulate in Modelsim. Luckily, due to the simplicity and regularity of the designs this was feasible. The visual inspection was done by checking if the adder trees, multiplications and SQNL functions were synthesized for random nodes in each layer. It was then assumed that if these were correctly synthesized for these randomly selected nodes, they were also correct for the others because they were generated by the same VHDL code.

6.1.1. FPGA

The FPGA that is used in this work is the Artix 7 (xc7a100tcsg324-1), which has been shown to work at cryogenic temperatures [16]. The software used to synthesize and place and route the hardware and used to get the delay and power estimate is Vivado by Xilinx. The estimate shown in this work is post-implementation. For bigger neural networks, the design did not fit on the FPGA, thus no results are available.

6.1.2. ASIC

The technology used for the ASIC estimates is the TSMC 40 nm technology. This has been tested at cryogenic temperatures [41][46]. The synthesis is done in Genus by Cadence. The area, power and delay estimates shown in this work are the post-synthesis results from Genus. The delay estimate in this work is a worst-case estimate of the hardware at room temperature. This is because the estimate is done before place and route, meaning the propagation delays can not be correctly determined. However, because a worst-case estimate will already be able to tell us if the delay is good enough, this suffices for now. The power and area estimates are also different from the post layout results. This is mainly due to the fact that the synthesis results do not include any wiring. This means that both power and delay are in practice worse than that of the post-synthesis results. To improve on the results a post layout simulation can be done, with in the best case cryogenic models. However, since these cryogenic models are not available yet, the best solution would be to tape-out the chip and measure it at 4 K. These improvements are all left for future work, due to the limited time of this project.

6.2. Results

The FPGA post-layout and ASIC post-synthesis results are presented in table 6.1. As discussed, these values are not accurate, but will give us some insight in the scaling of power, area and delay for different neural network hardware configurations.

6.2.1. Delay

Looking at the delay estimates in table 6.1, all configurations show to be a lot faster than the required 440 ns needed for the surface code cycle. This means, that the decoder hardware can be delayed by a lot to still meet the requirements to avoid a data backlog.

One thing that can be seen from the data, is that the main influence on the delay is the number of bits used for each value. This can be explained by the fact that the delay is the longest path between input and output. Because the layer sizes mainly increase the number of parallel paths, they do not increase the length of the critical path. The only influence that they have is some additional delay in the adder tree. However, the number of bits will increase the delay in the multipliers, each addition on the adder tree and the SQNL function. This is because all of these blocks include full adders, where the delay is determined by the number of bits used. This fact is important as it shows that further research in minimization of the number of required bits is needed to further improve the speed of the decoder. Another possibility is to use a different adder and multiplier architecture to improve the performance with larger bit sizes.

FPGA, Post-Implementation, Artix 7

Distance	**Layer Size 1**	**Bits**	**Delay** (ns)	**Power** (mW)
3	8	2	24.176	< 1
3	8	4	63.381	3
5	32	2	35.132	2
5	32	4	84.649	10
5	32	5	91.259	18
7	128	2	50.444	13

Genus, Post-Synthesis, TSMC 40 nm

Distance	**Layer Size 1**	**Bits**	**Delay** (ns)	**Power** (μW)	**Cell Area** (μm^2)
3	8	2	6.798	13.291	2,498
5	32	2	9.825	89.705	17,990
5	32	5	20.588	432.630	81,664
7	128	2	11.113	541.891	119,871
7	128	6	24.157	3,973.881	704,018
9	256	2	12.329	1,600.817	366,196
9	256	6	25.836	11,717.942	2,051,126

Table 6.1: Delay, power and area estimates for the neural network decoder hardware. The post-implementation FPGA estimates were done for the Xilinx Artix 7 in Xilinx Vivado. The post-synthesis ASIC estimates were done in the TSMC 40 nm technolgy using Cadence Genus. The block diagram of the hardware is shown in figure 6.1 and the VHDL code in appendix D. For all neural networks, three layers are used, where the second hidden layer is always of size 16 and the output layer always of size 2. The number of inputs is related to the surface code distance: $d^2 - 1$ and the first hidden layer size is as used in the optimum fixed configuration of the previous chapter. Finally the number of bits is either 2 or as in the optimum fixed configuration of the previous chapter.

6.2.2. Power

The power estimates of the ASIC show that both the number of bits and the layer sizes increase the power. This is expected as, contrary to the delay, the power does depend on the number of parallel paths, and thus the layer sizes. The power estimates of the FPGA were not possible for larger distances, so no conclusions could be drawn from them. However, the results that are shown show a similar trend as the ASIC designs.

It also shows that the power required for the larger distances is in the order of milliwats. As only 1 W is available in total at 4 K, this power is too much to have this decoder for multiple logical bits. However, because the delay can be a lot longer, some sequential hardware or other solutions to optimize the power of multiplication and addition hardware [32] can be explored to find a better trade-off between power and speed.

6.2.3. Area

Table 6.1 shows a clear correlation between the power and area of the ASIC hardware. For this reason the same analysis that is done on the power can be used to decrease the area of the hardware.

6.3. Cryogenic Electronics

6.3.1. FPGA

The FPGA used in the previous analysis is shown to work at 4 K [16]. The measurements at 4 K show that the logic will be slightly faster, however the auxilary hardware will consume a lot more power. As the estimated delay is already short enough, a more sequential solution could be investigated in the future to save area and power.

6.3.2. ASIC

[41] discusses the different types of digital logic styles that could be used to operate at 4 K. In general, due to the increase in the electron and hole mobility, the transistors operate faster. There is also less leakage power at 4 K, suggested to be due to a steeper sub-threshold slope [46]. This sub-threshold slope could also, together with the lowered thermal noise, allow for a smaller supply voltage. This will further lower the total power. The area however will increase. This is due to the need of larger cells to avoid mismatch [46] and an increase in

the number of well-taps due to an increase bulk resistance. This bulk resistance will increase the amount of latch-up in the substrate.

6.4. Conclusion

As shown in this chapter, a fully parallel hardware solution for a neural network decoder of the surface code is fast enough for the surface code cycle of 440 ns. However, because the delay for the ASIC is always lower than 26 ns, a slower but more efficient architecture could be used. Two solutions could be to either make parts of the hardware sequential to reuse parts of the hardware, or to implement the multiplications and adder tree to a different structure than the full adders as used in this work. Both will reduce the power and area and increase the delay.

Another improvement to the power, area and delay would be to further reduce the number of bits needed to represent the signals in the neural network. However, this means additional research needs to be done to improve the performance when moving to lower bits. This could include a different trade-off between the layer sizes and the number of bits. It could also be that a convolutional neural network with small kernels could help.

Moving to cryogenic temperatures will only further reduce the delay of the logic, thus leading to a different hardware solution.

7

Conclusion, Discussion and Future Work

7.1. Conclusion

This work extends on the previous work on neural network decoders for the surface code [52][51]. In this work [52][51], it was shown that a software solution was not fast enough to perform the decoding, but can outperform the common used Blossom decoder. The recommendation of [52][51] was to develop a hardware solution, that would be fast enough to keep up with the 440 ns surface code cycle for a transmon based quantum computer. However, another requirement introduced in this work is the need for a cryogenic hardware solution. The need for a cryogenic stack (4 K) is proposed by our group [34] as a step in the integration of the qubits operating just above absolute zero (<< 1 K) and the control hardware, currently operating at room temperature (300 K). One problem with cryogenic temperatures is the limited cooling power available. At 4 K the total power available is ≈ 1 W. For this reason, this work focuses on exploring the design space for feed-forward neural networks to minimize the power.

This thesis shows it is possible to create a neural network based decoder in hardware, that outperforms the Blossom algorithm in terms of the pseudo-threshold, and meets the time requirements on the surface code cycle using transmons or single electron silicon spin qubits. The pseudo-threshold is presented for multiple configurations for the smallest four distances (3, 5, 7 and 9) of the surface code. These configurations are varied for layers, layer sizes, transferfunctions and the number of bits used in quantization. For some of the configurations, a crude estimate for the delay, power and area is presented as well.

The proposed optimum configurations are those that perform just better than Blossom and have the lowest layer sizes and number of bits for which that holds true. This is in all case a neural network with two hidden layers, where the second one is 16 nodes and the output layer is two nodes. All nodes use the SQNL transfer function. The distance 3 has a first hidden layer of size 8 and used 5 bits. The distance 5 has a first hidden layer size of 32 and uses 5 bits. The distance 7 has a first hidden layer size of 128 and uses 6 bits and finally the distance 9 has a first hidden layer size of 256 and uses 6 bits.

For the hardware it is found that the delays are a lot better than the required 440 ns of the surface code cycle. For the distance 9 configuration synthesized in the TSMC 40 nm technology, the delay is less than 26 ns. However, a lot of optimization can still be done, because these designs are fully parallel and the power and area are too much. Some suggested improvements are different adders and multipliers, a convolutional neural network architecture and making the neural network more sequential.

7.2. Contributions

All the original contributions and findings of this work are listed below:

- Adjusted surface code cycle that replaces CNOTs with CPHASEs and Hadamards with $R_y(\pi/2)$ rotations as these are not available for transmons and single electron silicon spin qubits.
- New pure error decoder with numbering that is more regular and spatially and rotationally invariant than the one used in previous work. This allows for an easier hardware and software implementation

and might also improve the performance of a convolutional neural network as it uses spatial invariances.

- Choice of using a feed-forward neural network as a first target, to be able to extend the findings to a recurrent and/or convolutional neural network if measurement errors are included in the error model and/or larger distances are used.
- GPU code written in C and CUDA that increases the simulation speed of the surface code, error model, pure error decoder and the training of neural networks.
- Simulation results for a large design space exploration, sweeping the layer sizes, transferfunctions and quantization bits for feed-forward neural network decoders.
- Analysis of the results, comparing the performance to previous work [51] and the Blossom decoder.
- Crude estimate of the power, area and delay that show the feasibility of the neural network decoder hardware. The estimates that were done are a post-implementation estimate on the Artix 7 Xilinx FPGA and a post-synthesis estimate for the TSMC 40 nm technology.

7.3. Discussion and Future Work

This work was meant as a first exploration of the low-power hardware solutions of neural network decoders for the surface code. This means a lot of future research is possible to improve on and extend the results shown in this thesis. Below a selection of important improvements is listed.

To start, the delay and power estimates in this work were performed after synthesis. This means they are not accurate. For a more accurate estimate a post-layout estimate has to be performed. However, even these results are not accurate at 4 K. To get an actual number at 4 K, a tape-out with power measurements needs to be performed. However, as the neural network decoders in this work are only useful for the simple depolarizing error model and thus not apply to actual qubits, it might not be as useful as a more advanced decoder, such as a convolutional neural network and a recurrent neural network.

Extending on the power of this decoder, as presented in the last chapter, the delay of a fully parallel neural network is short enough to fit into the 440 ns surface code cycle. Thus, some parts can be serialized to reduce the power and area of the hardware. Another possibility is to use more advanced dot product hardware, reducing the power and area, despite maybe increasing the delay slightly.

Simulating the quantization with a more accurate 2's complement model. This work uses a quantization model using a 2's complement fixed point number with 1 sign bit and a fractional part, where both $+1$ and -1 are used. In hardware, only the -1 is available. This means the quantization results of this work might be slightly different to a correct model. However, because the neural network can adjust the weights of the neural network, this is probably not very significant.

Furthermore, it is relevant to try to reduce the number of bits needed to quantize the values in the neural networks for larger distance surface codes by either:

- Scale the input range of the SQNL function to allow a larger input sum.
- Change the maximum value of the weights, i.e. clip to a smaller value. This will mean that the most significant bits will be removed.
- Use floating point arithmetic with a small amount of bits.
- Use a different neural network architecture, such as the convolutional neural network. This will gives less inputs for each kernel, allowing the use of less bits per weight.

This reduction will have a big effect on the delay and power of the decoder hardware.

Another reason to move to convolutional neural networks is to improve the decoding performance and neural network training generalization for larger distances [53]. Next to this a recurrent neural network can be used, such as the LSTM, to be able to learn the temporal coherence in the data from an error model that includes measurement errors, such as the circuit noise model.

Bibliography

[1] P. Baireuther, T. E. O'Brien, B. Tarasinski, and C. W. J. Beenakker. Machine-learning-assisted correction of correlated qubit errors in a topological code. *arXiv e-prints*, art. arXiv:1705.07855, May 2017.

[2] C. Barthel, M. Kjærgaard, J. Medford, M. Stopa, C. M. Marcus, M. P. Hanson, and A. C. Gossard. Fast sensing of double-dot charge arrangement and spin state with a radio-frequency sensor quantum dot. *Phys. Rev. B*, 81:161308, Apr 2010. doi: 10.1103/PhysRevB.81.161308. URL https://link.aps.org/doi/10.1103/PhysRevB.81.161308.

[3] J. Bergstra, G. Desjardins, P. Lamblin, and Y. Bengio. *Quadratic polynomials learn better image features*. Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 2009.

[4] Sergey Bravyi, Martin Suchara, and Alexander Vargo. Efficient algorithms for maximum likelihood decoding in the surface code. *Physical Review A*, 90(3):032326, Sep 2014. doi: 10.1103/PhysRevA.90.032326.

[5] Christopher Chamberland and Pooya Ronagh. Deep neural decoders for near term fault-tolerant experiments. *Quantum Science and Technology*, 3(4):044002, Oct 2018. doi: 10.1088/2058-9565/aad1f7.

[6] L. Childress, M. V. Gurudev Dutt, J. M. Taylor, A. S. Zibrov, F. Jelezko, J. Wrachtrup, P. R. Hemmer, and M. D. Lukin. Coherent dynamics of coupled electron and nuclear spin qubits in diamond. *Science*, 314(5797): 281–285, 2006. ISSN 0036-8075. doi: 10.1126/science.1131871. URL https://science.sciencemag.org/content/314/5797/281.

[7] J. I. Cirac and P Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74 (20), 1995.

[8] D. P. DiVincenzo. The physical implementation of quantum computing,. *arXiv:quant-ph/0002077*, 2000.

[9] J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449, 1965.

[10] D.L. Elliott and David L. Elliott. *A better Activation Function for Artificial Neural Networks*. University of Maryland, College Park, 1993.

[11] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467, 1982.

[12] Thomas Fösel, Petru Tighineanu, Talitha Weiss, and Florian Marquardt. Reinforcement Learning with Neural Networks for Quantum Feedback. *Physical Review X*, 8(3):031084, Jul 2018. doi: 10.1103/PhysRevX.8.031084.

[13] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. Surface codes: Towards practical large-scale quantum computation. *PHYSICAL REVIEW A*, 86, 032324, 2012. doi: 10.1103/PhysRevA.86.032324.

[14] L. K. Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219, 1996.

[15] Aram W. Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203–209, Sep 2017. doi: 10.1038/nature23458.

[16] Harald Homulle and Edoardo Charbon. Performance characterization of altera and xilinx 28 nm fpgas at cryogenic temperatures. In *ICFPT*, pages 25–31, 12 2017. doi: 10.1109/FPT.2017.8280117.

[17] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2 (5):359–366, 1989.

[18] Evan Jeffrey, Daniel Sank, J. Y. Mutus, T. C. White, J. Kelly, R. Barends, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. Megrant, P. J. J. O'Malley, C. Neill, P. Roushan, A. Vainsencher, J. Wenner, A. N. Cleland, and John M. Martinis. Fast accurate state measurement with superconducting qubits. *Phys. Rev. Lett.*, 112:190504, May 2014. doi: 10.1103/PhysRevLett.112.190504. URL https://link.aps.org/doi/10.1103/PhysRevLett.112.190504.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, art. arXiv:1412.6980, Dec 2014.

[20] A. Y. Kitaev. Quantum computations: algorithms and error correction. *Proceedings of the Third International Conference on Quantum Communication, Computing and Measurement*, 1997.

[21] A. Y. Kitaev. Fault-tolerant quantum computation by anyons. *Annals of Physics*, 303 (1):2–30, 1 2003. doi: 10.1016/S0003-4916(02)00018-0.

[22] Jens Koch, Terri M. Yu, Jay Gambetta, A. A. Houck, D. I. Schuster, J. Majer, Alexandre Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Introducing the Transmon: a new superconducting qubit from optimizing the Cooper Pair Box. *arXiv e-prints*, art. cond-mat/0703002, Feb 2007.

[23] Stefan Krastanov and Liang Jiang. Deep Neural Network Probabilistic Decoder for Stabilizer Codes. *Scientific Reports*, 7:11003, Sep 2017. doi: 10.1038/s41598-017-11266-1.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, page 1106–1114, 2012.

[25] Y. le Cun. Generalization and network design strategies. Technical report, Technical Report CRG-TR-89-4, University of Toronto, Department of Computer Science, 1989.

[26] Enrique Martín-López, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L. O'Brien. Experimental realization of Shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773–776, Nov 2012. doi: 10.1038/nphoton.2012.259.

[27] Antonio Mezzacapo, Jorge Casanova, Lucas Lamata, and E Solano. Topological qubits with majorana fermions in trapped ions. *New Journal of Physics*, 15, 11 2011. doi: 10.1088/1367-2630/15/3/033005.

[28] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, Jan 2016. doi: 10.1038/npjqi.2015.23.

[29] Bert Moons, Koen Goetschalckx, Nick Van Berckelaer, and Marian Verhelst. Minimum energy quantized neural networks. *Circuits, Systems and Computers, 2017. Conference Record. 2017 51th Asilomar Conference on*, 11 2017.

[30] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press., 2000. ISBN 978-0-521-63503-5.

[31] T. E. O'Brien, B. Tarasinski, and L. DiCarlo. Density-matrix simulation of small surface codes under current and projected experimental noise. *Nature Partner Journals Quantum Information*, September 2017.

[32] B. Parhami. *Algorithms and design methods for digital computer arithmetic*. Oxford Universiy Press, Inc., 2012. ISBN 978-0-19-976693-2.

[33] J. Park. The concept of transition in quantum mechanics. *Foundations of Physics*, 1:23–33, 1970. doi: 10.1007/BF00708652.

[34] B. Patra, R. M. Incandela, J. P. G. van Dijk, H. A. R. Homulle, L. Song, M. Shahmohammadi, R. B. Staszewski, A. Vladimirescu, M. Babaie, F. Sebastiano, and E. Charbon. Cryo-cmos circuits and systems for quantum computing applications. *IEEE Journal of Solid-State Circuits*, 53(1):309–321, Jan 2018. ISSN 0018-9200.

[35] D. Poulin. Optimal and efficient decoding of concatenated quantum block codes. *Physical Review A*, November 2006. doi: 10.1103/PhysRevA.740523333.

[36] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv e-prints*, art. arXiv:1603.05279, Mar 2016.

[37] L. Riesebos, X. Fu, C. G. Almudever, and K. Bertels. Pauli frames for quantum computer architectures. *Proceedings of the 54th Annual Design Automation Conference*, page 76, 2017.

[38] Chad Rigetti, Jay M. Gambetta, Stefano Poletto, B. L. T. Plourde, Jerry M. Chow, A. D. Córcoles, John A. Smolin, Seth T. Merkel, J. R. Rozen, George A. Keefe, Mary B. Rothwell, Mark B. Ketchen, and M. Steffen. Superconducting qubit in a waveguide cavity with a coherence time approaching 0.1 ms. *Physical Review B*, 86(10):100506, Sep 2012. doi: 10.1103/PhysRevB.86.100506.

[39] R. Rojas. *Neural Networks, A Systematic Introduction*. Springer, 1996.

[40] J. A. Schreier, A. A. Houck, Jens Koch, D. I. Schuster, B. R. Johnson, J. M. Chow, J. M. Gambetta, J. Majer, L. Frunzio, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Suppressing charge noise decoherence in superconducting charge qubits. *Physical Review B*, 77(18):180502, May 2008. doi: 10.1103/PhysRevB.77.180502.

[41] E. Schriek. A low-power standard cell library for cryogenic operation. Master's thesis, Delft University of Technology, 2018.

[42] P. W. Shor. Polynomial-time algorithms for prime factorizaiton and discrte logarithms on a quantum computer. *SIAM Journal on Computing*, 26, 1484, 1997. doi: 10.1137/S0097539795293172.

[43] W. Shor. Scheme for reducing decoherence in quantum computer memory. *AT&T Bell Laboratories*, 1995.

[44] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, art. arXiv:1409.1556, Sep 2014.

[45] R. Sutton. The bitter lesson, March 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

[46] P. A. 't Hart, J. P. G. van Dijk, M. Babaie, E. Charbon, A. Vladimircscu, and F. Sebastiano. Characterization and model validation of mismatch in nanometer cmos at cryogenic temperatures. In *2018 48th European Solid-State Device Research Conference (ESSDERC)*, pages 246–249, Sep. 2018. doi: 10.1109/ESSDERC.2018.8486859.

[47] Giacomo Torlai and Roger G. Melko. Neural Decoder for Topological Codes. *Physical Review Letters*, 119 (3):030501, Jul 2017. doi: 10.1103/PhysRevLett.119.030501.

[48] Jeroen P. G. van Dijk, Edoardo Charbon, and Fabio Sebastiano. The electronic interface for quantum processors. *arXiv e-prints*, art. arXiv:1811.01693, Nov 2018.

[49] Jeroen P. G. van Dijk, Erika Kawakami, Raymond N. Schouten, Menno Veldhorst, Lieven M. K. Vand ersypen, Masoud Babaie, Edoardo Charbon, and Fabio Sebastiano. The impact of classical control electronics on qubit fidelity. *arXiv e-prints*, art. arXiv:1803.06176, Mar 2018.

[50] S. Varsamapoulos. *Neural Network Based Decoders For The Surface Code*. PhD thesis, Delft University of Technology, 2019.

[51] S. Varsamopoulos, K. Bertels, and C. G. Almudever. Designing neural network based decoders for surface codes. *arXiv e-prints*, November 2018.

[52] S. Varsamopoulos, B. Criger, and K. Bertels. Decoding small surface codes with feedforward neural networks. *Quantum Science and Technology*, 3(1):015004, January 2018. doi: 10.1088/2058-9565/aa955a.

[53] Savvas Varsamopoulos, Koen Bertels, and Carmen G. Almudever. Decoding surface code with a distributed neural network based decoder. *arXiv e-prints*, art. arXiv:1901.10847, Jan 2019.

[54] M. Veldhorst, J. C. C. Hwang, C. H. Yang, A. W. Leenstra, B. de Ronde, J. P. Dehollain, J. T. Muhonen, F. E. Hudson, K. M. Itoh, A. Morello, and A. S. Dzurak. An addressable quantum dot qubit with fault-tolerant control-fidelity. *Nature Nanotechnology*, 9(12):981–985, Dec 2014. doi: 10.1038/nnano.2014.216.

[55] W. Wootters and W. Zurek. A single quantum cannot be cloned. *Nature*, 299, 5886:802–803, 1982. doi: 10.1038/299802a0.

[56] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv e-prints*, art. arXiv:1606.06160, Jun 2016.

A

Neural Network Transferfunction Pseudo-Threshold Plots

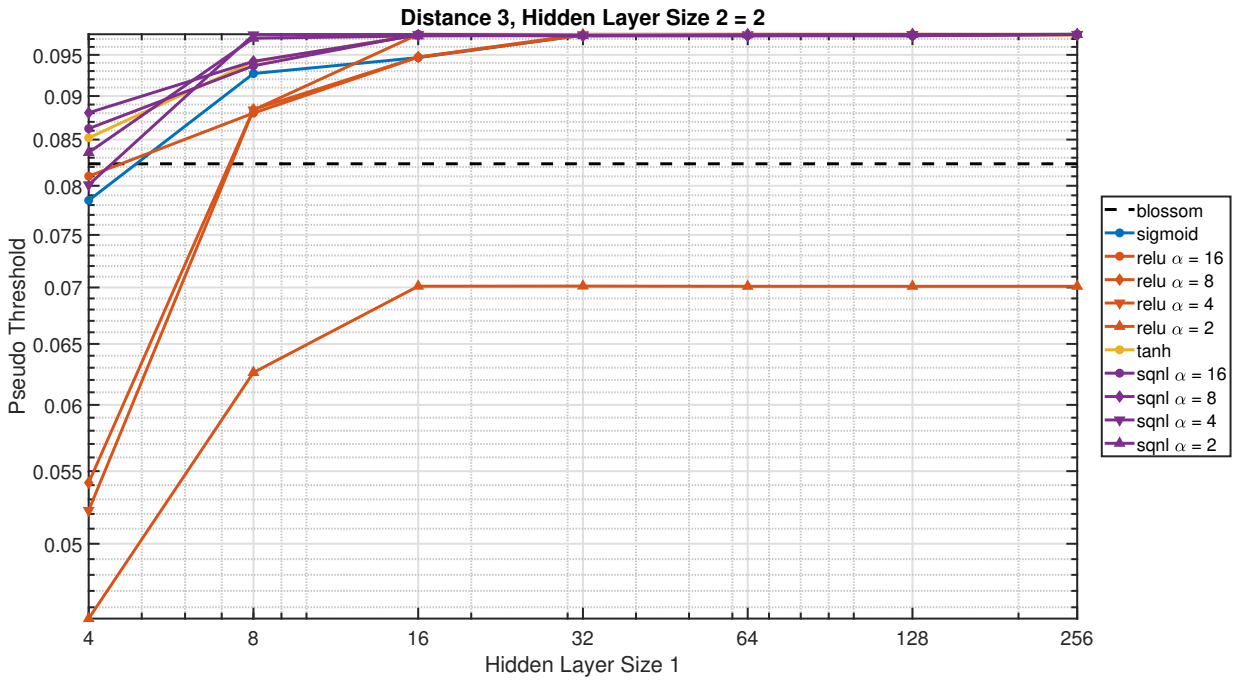


Figure A.1: Simulation results for the transfer function sweep for a surface code of distance 3. For this plot the second hidden layer size is 2. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

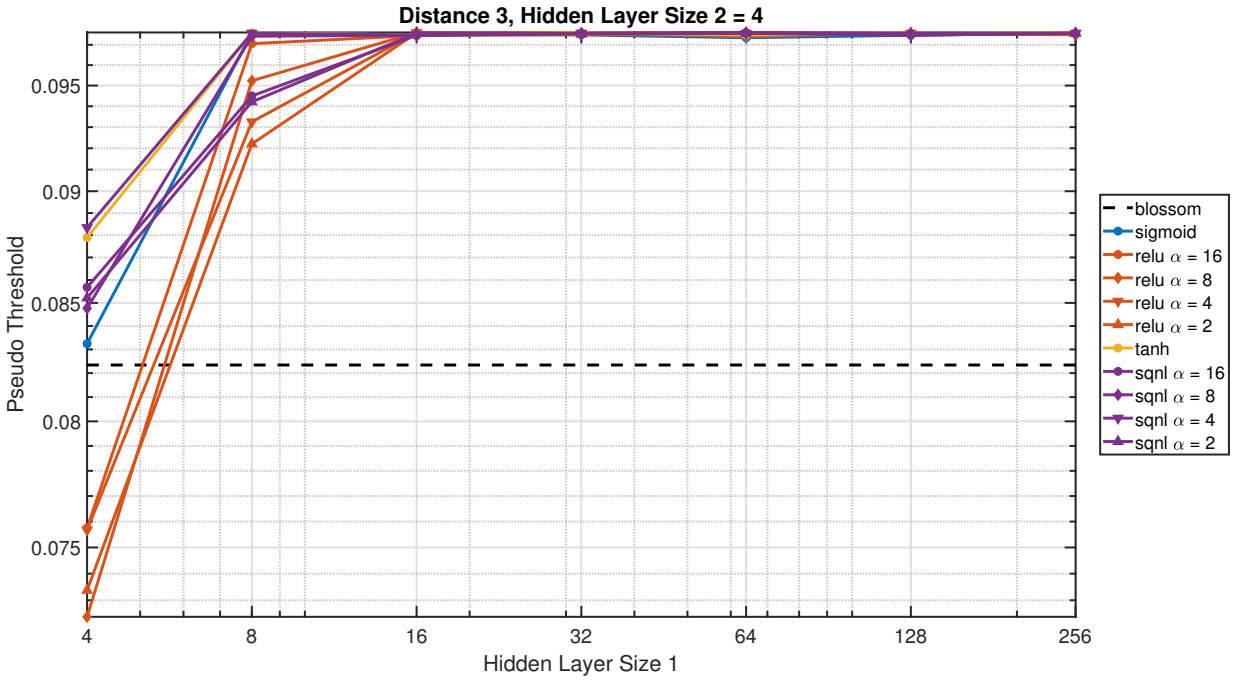


Figure A.2: Simulation results for the transfer function sweep for a surface code of distance 3. For this plot the second hidden layer size is 4. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

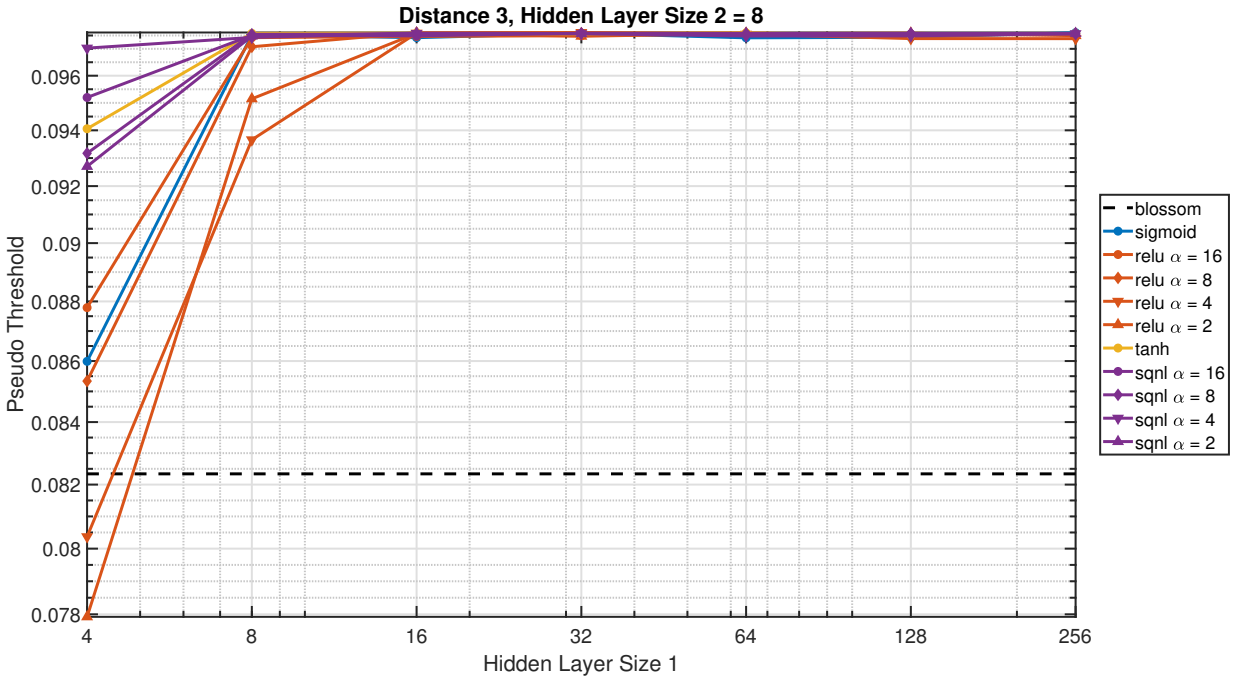


Figure A.3: Simulation results for the transfer function sweep for a surface code of distance 3. For this plot the second hidden layer size is 8. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

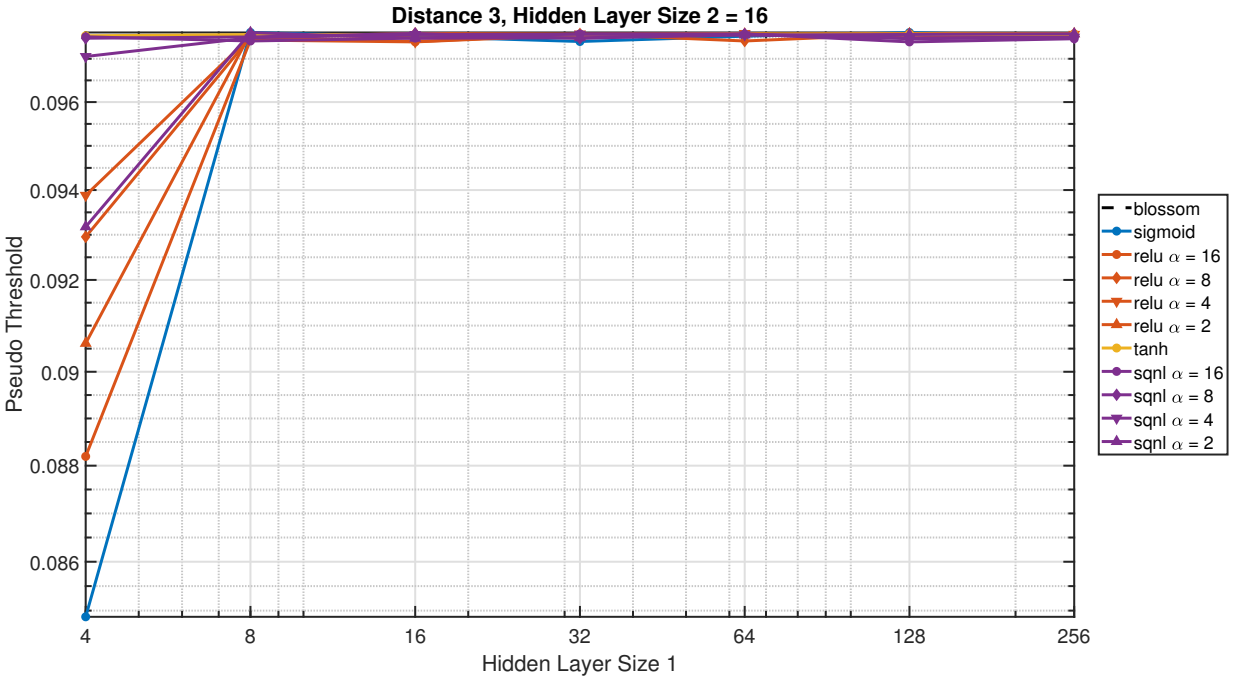


Figure A.4: Simulation results for the transfer function sweep for a surface code of distance 3. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

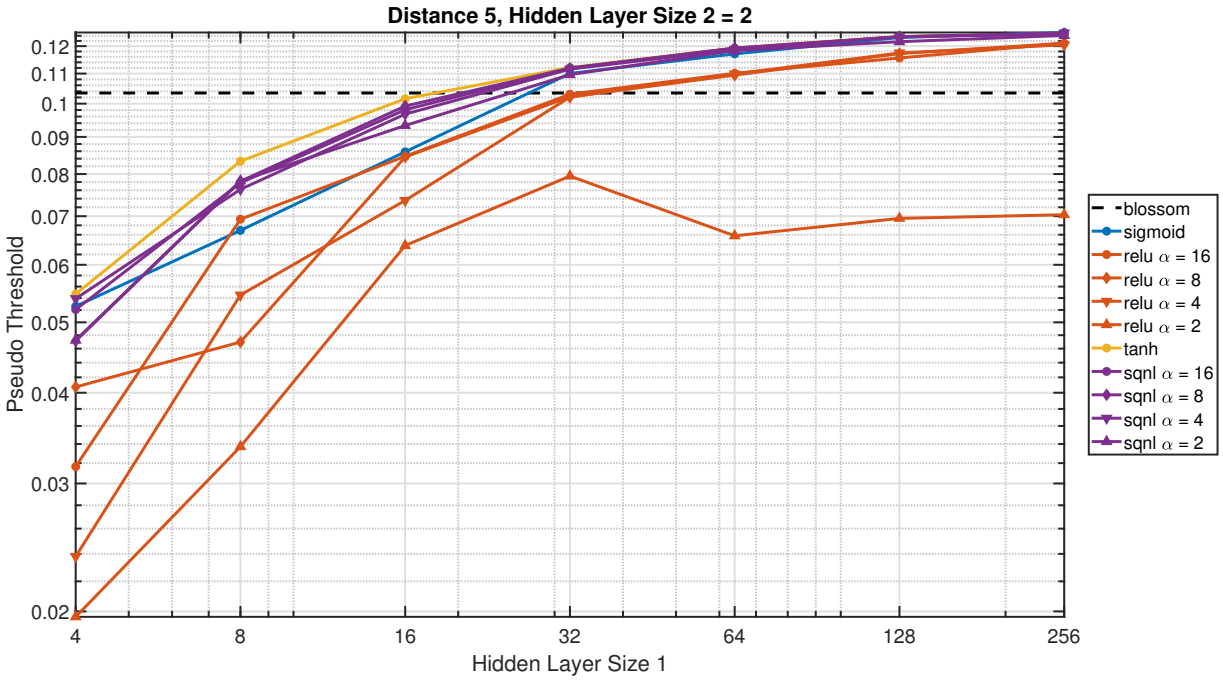


Figure A.5: Simulation results for the transfer function sweep for a surface code of distance 5. For this plot the second hidden layer size is 2. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

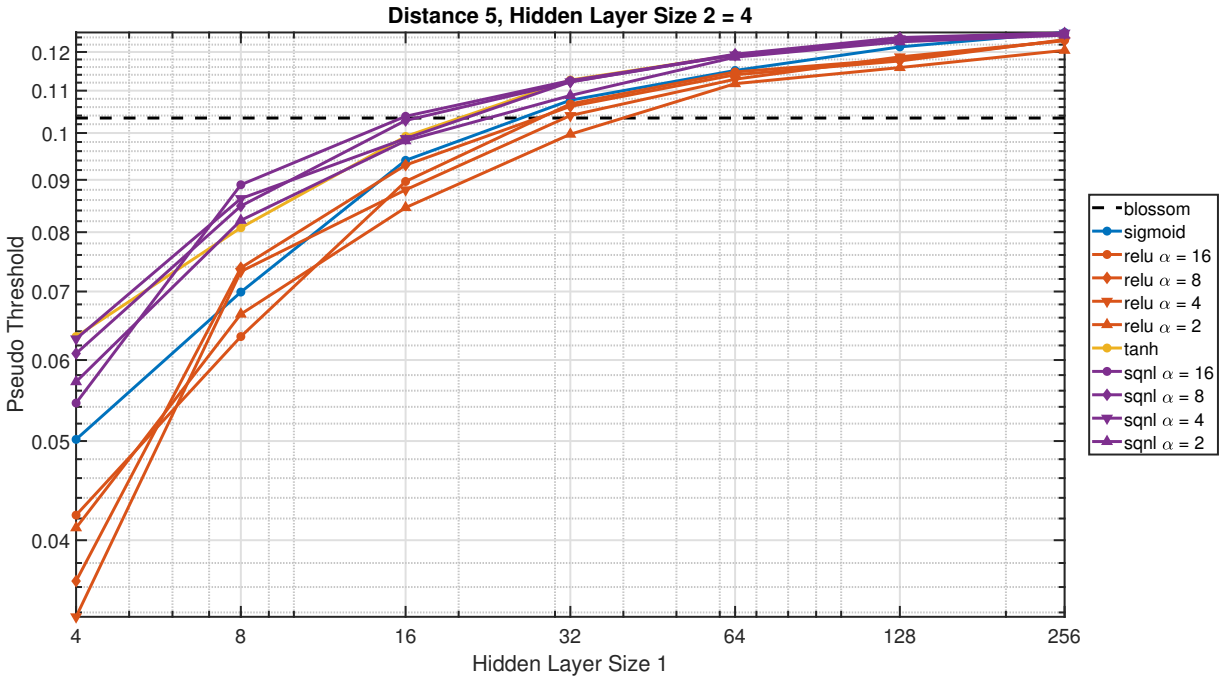


Figure A.6: Simulation results for the transfer function sweep for a surface code of distance 5. For this plot the second hidden layer size is 4. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

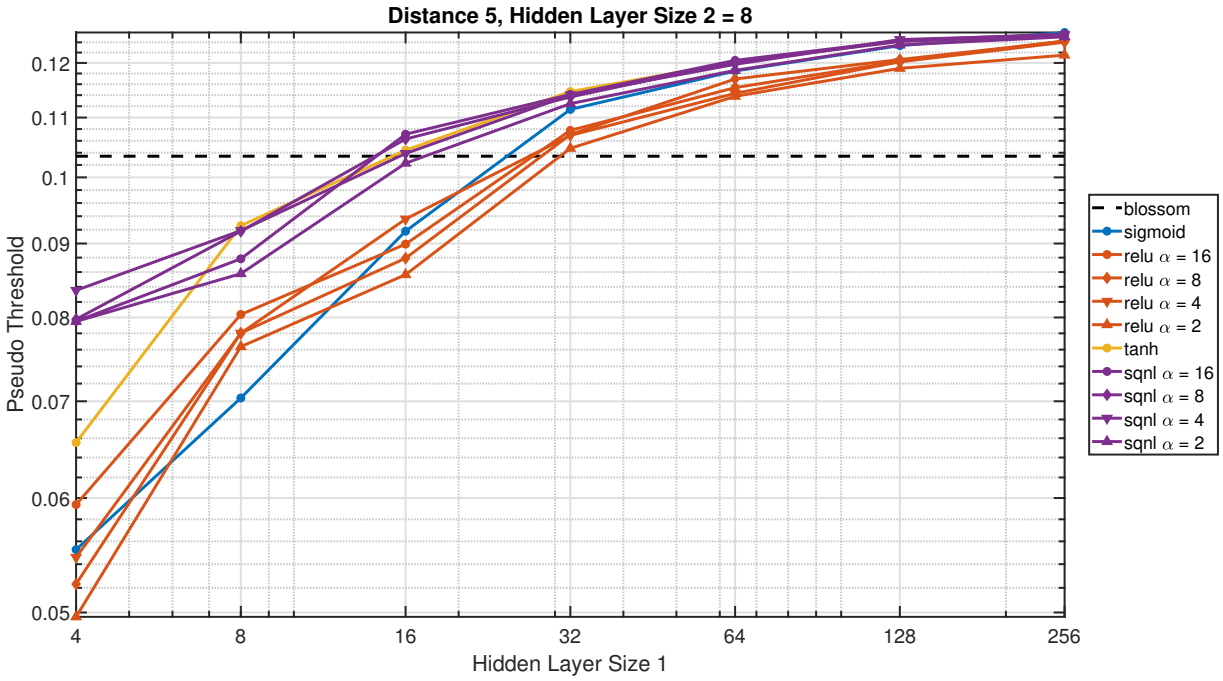


Figure A.7: Simulation results for the transfer function sweep for a surface code of distance 5. For this plot the second hidden layer size is 8. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

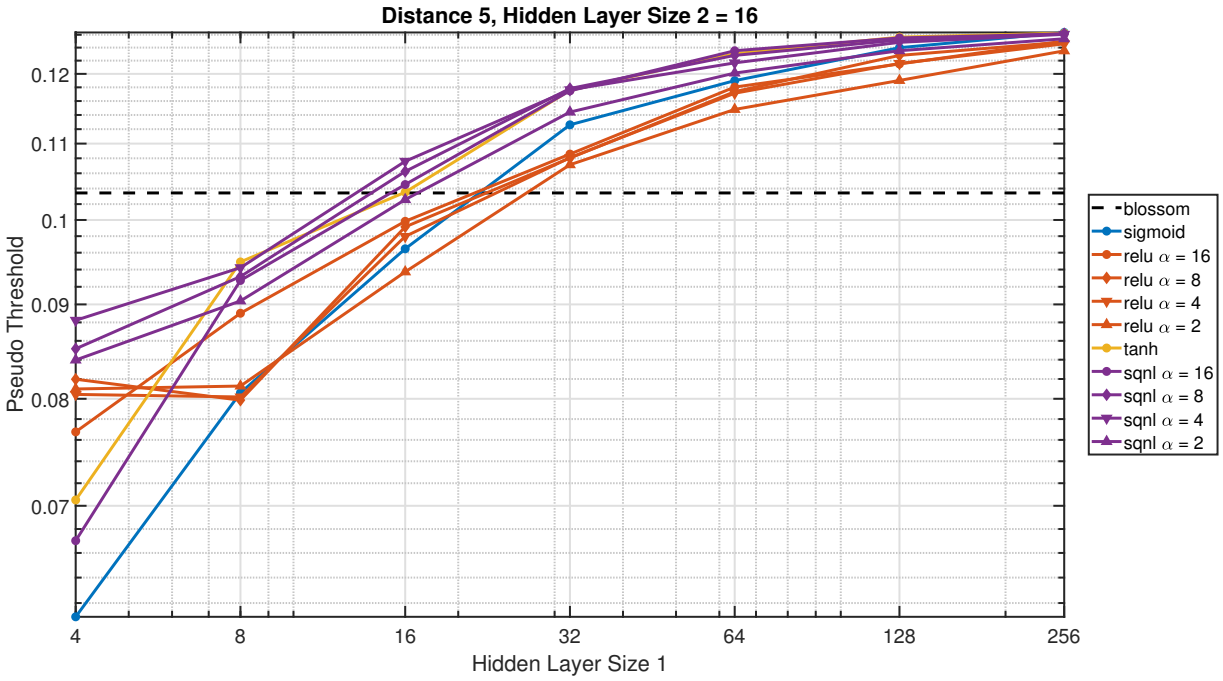


Figure A.8: Simulation results for the transfer function sweep for a surface code of distance 5. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

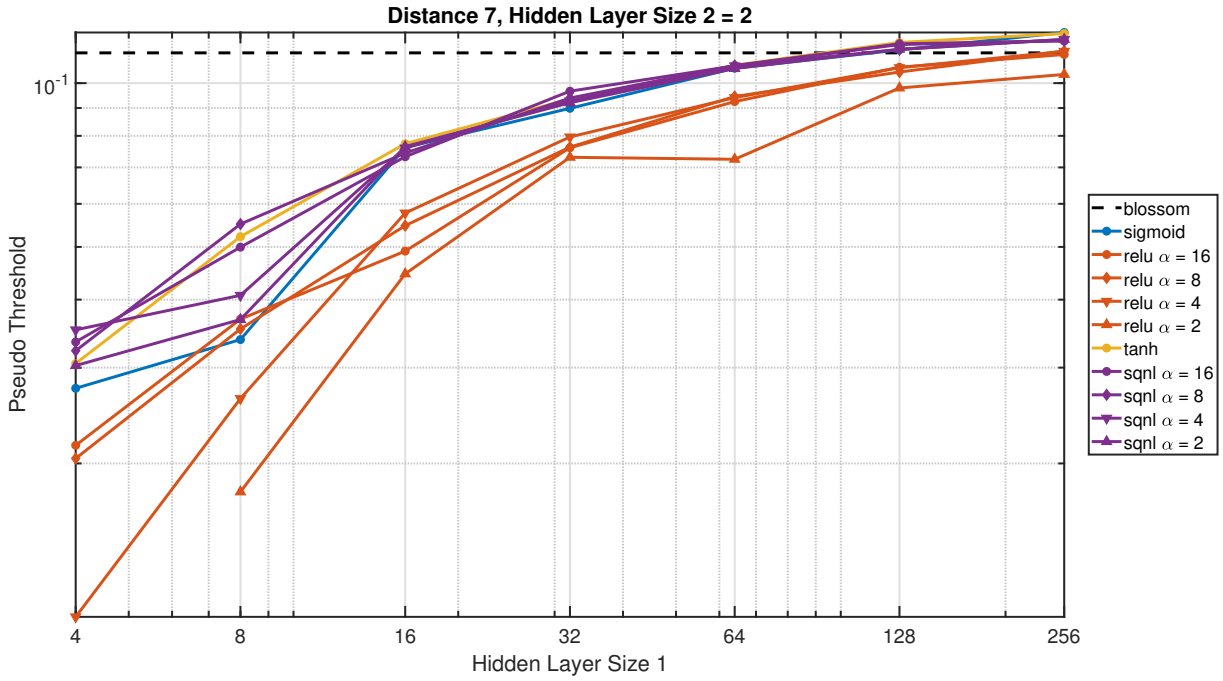


Figure A.9: Simulation results for the transfer function sweep for a surface code of distance 7. For this plot the second hidden layer size is 2. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

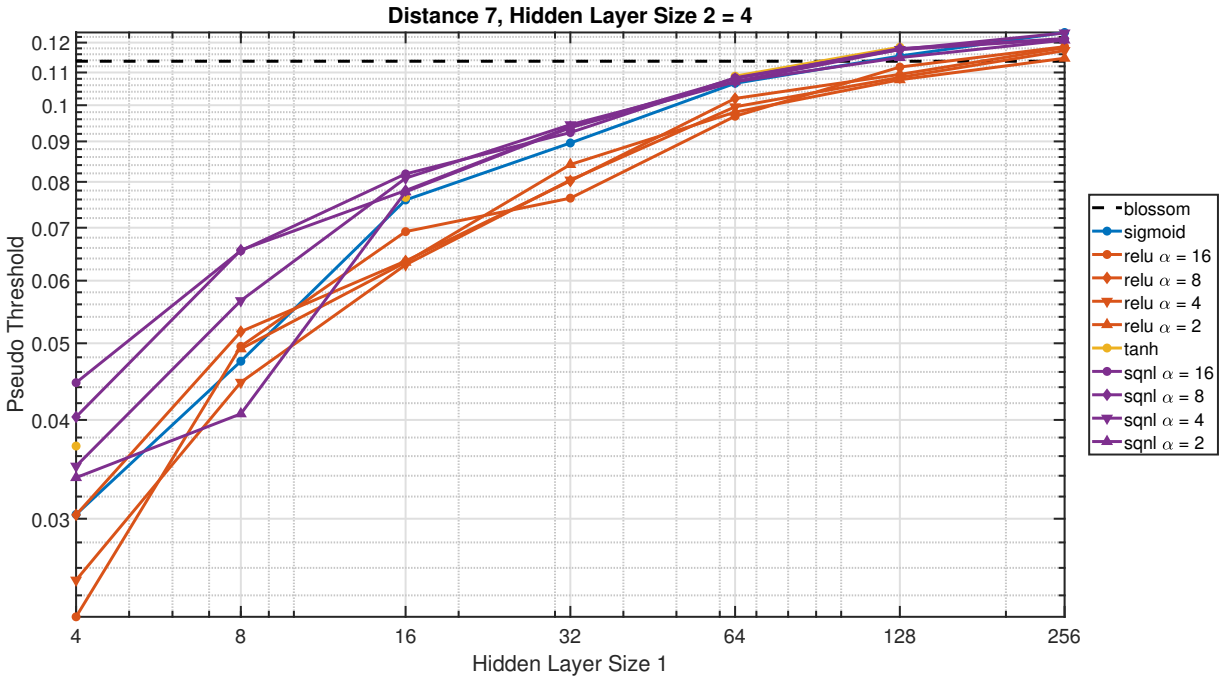


Figure A.10: Simulation results for the transfer function sweep for a surface code of distance 7. For this plot the second hidden layer size is 4. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

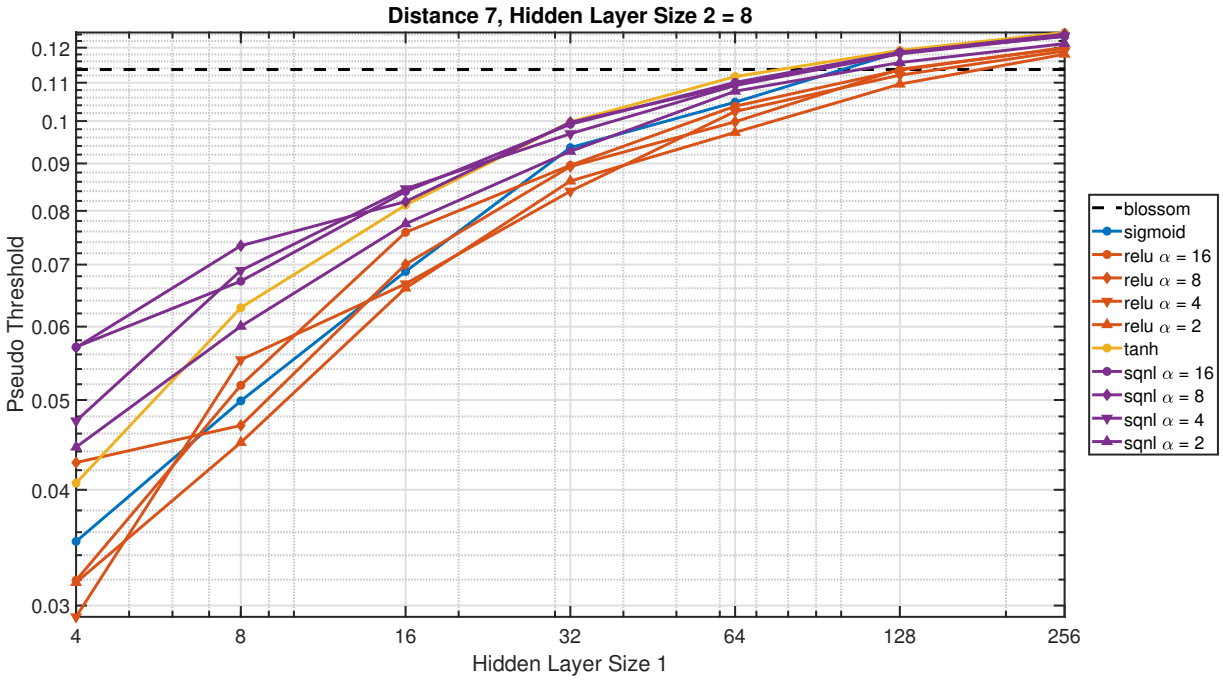


Figure A.11: Simulation results for the transfer function sweep for a surface code of distance 7. For this plot the second hidden layer size is 8. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

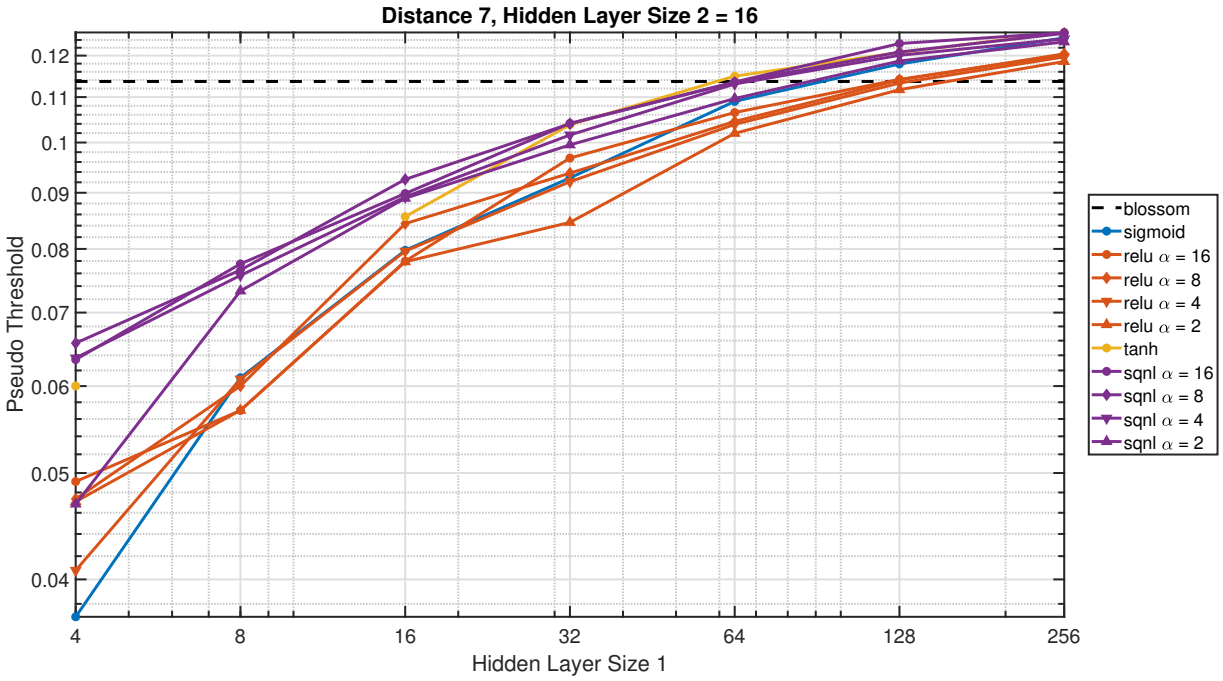


Figure A.12: Simulation results for the transfer function sweep for a surface code of distance 7. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

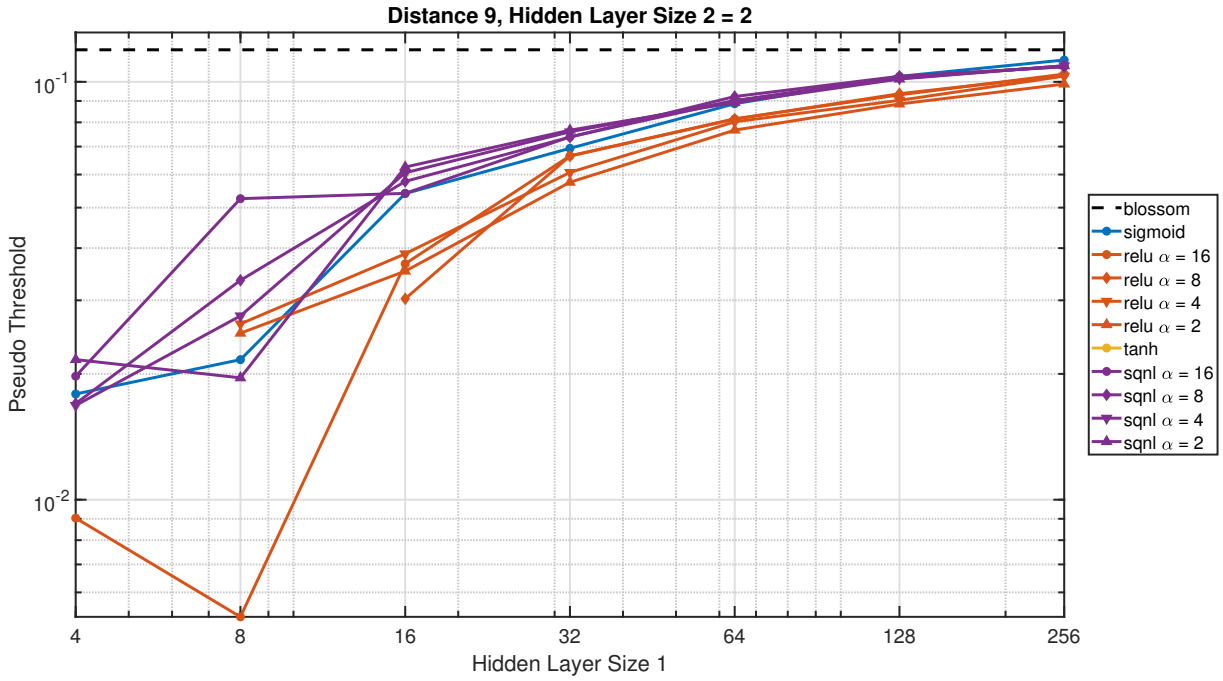


Figure A.13: Simulation results for the transfer function sweep for a surface code of distance 9. For this plot the second hidden layer size is 2. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

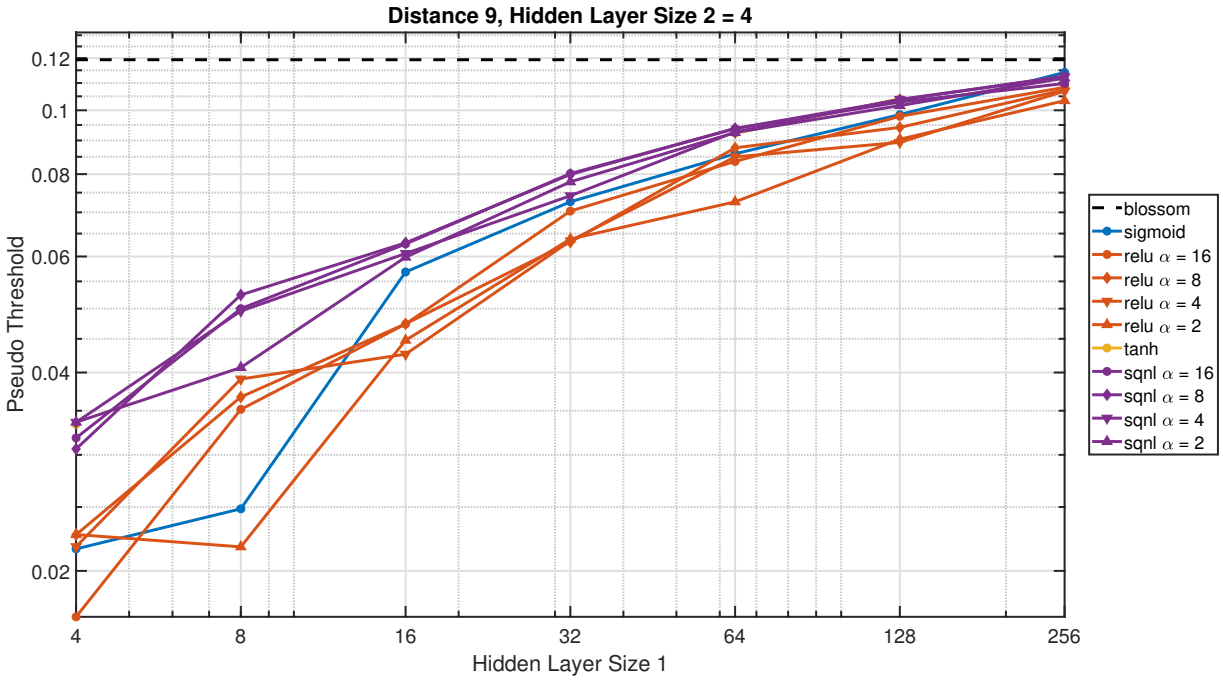


Figure A.14: Simulation results for the transfer function sweep for a surface code of distance 9. For this plot the second hidden layer size is 4. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

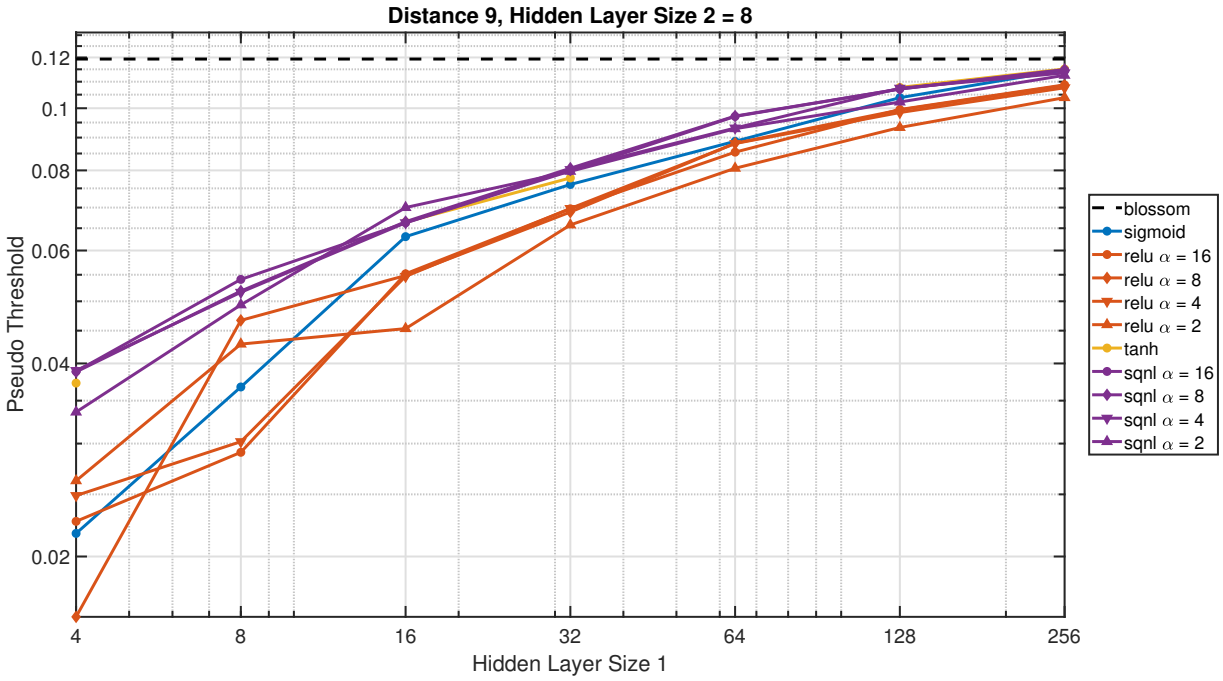


Figure A.15: Simulation results for the transfer function sweep for a surface code of distance 9. For this plot the second hidden layer size is 8. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

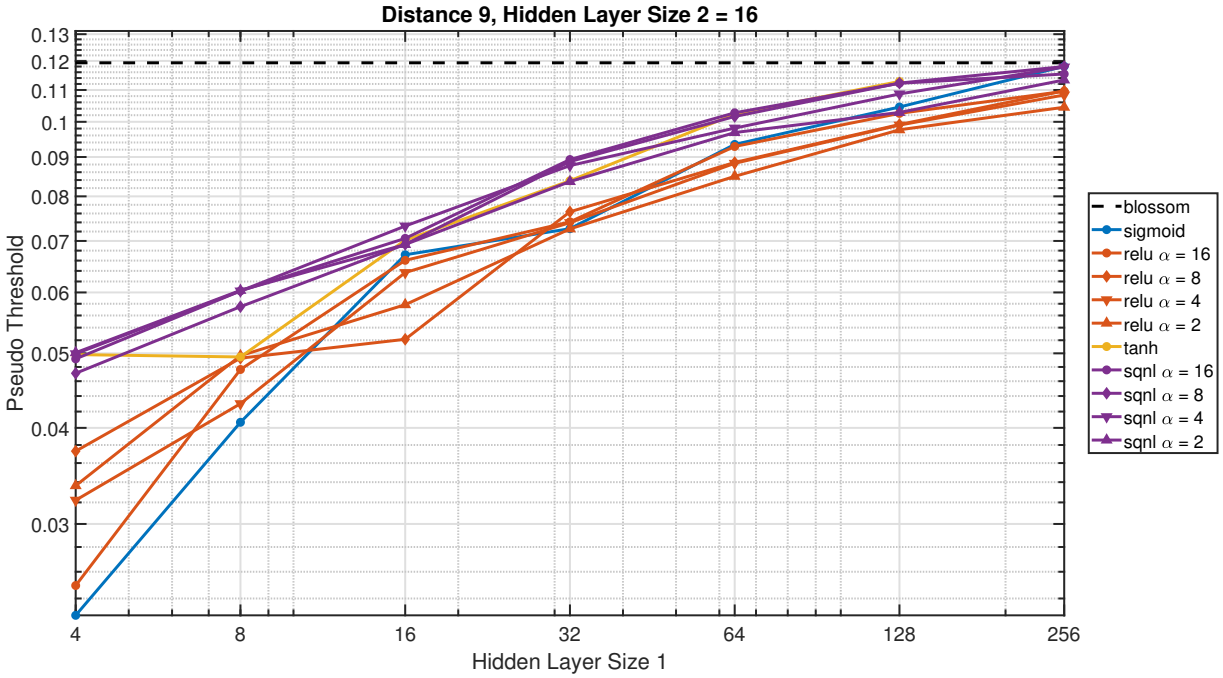


Figure A.16: Simulation results for the transfer function sweep for a surface code of distance 9. For this plot the second hidden layer size is 16. On the x-axis is the first hidden layer size. On the y-axis, the pseudo-threshold is compared with that of Blossom shown in the black dashed line.

B

Quantized Neural Network Pseudo-Threshold Plots

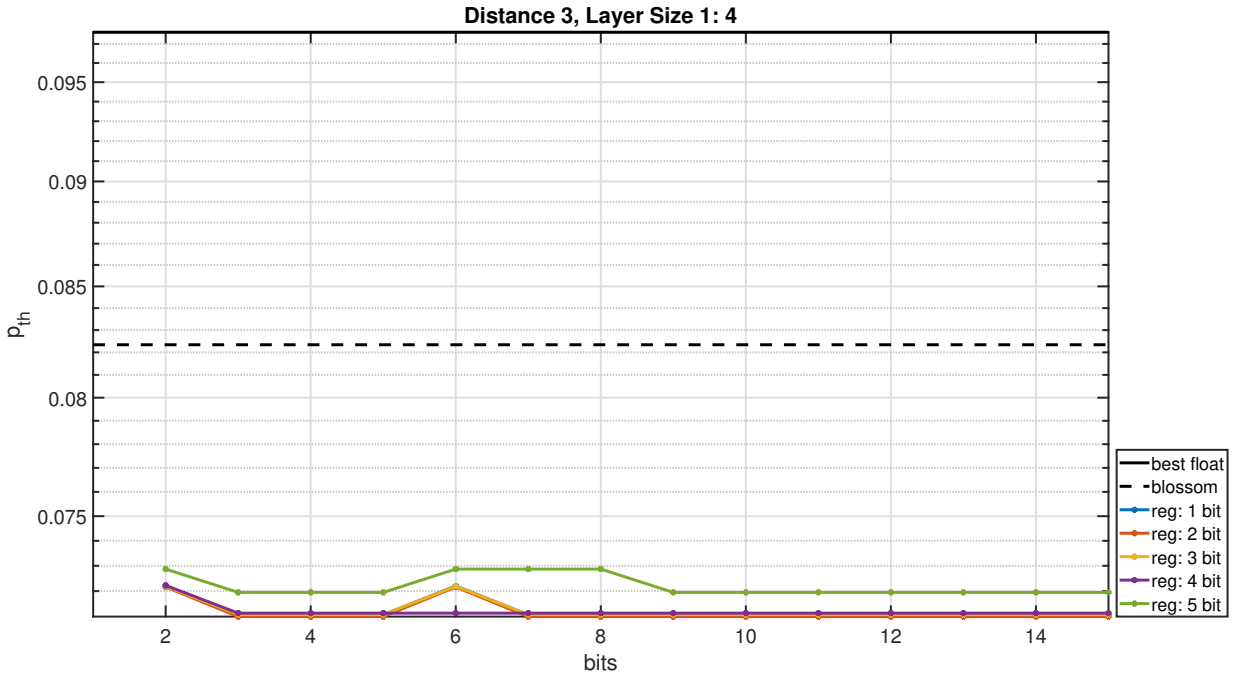


Figure B.1: Simulation results for the quantization sweeps for a surface code of distance 3. This is done on a neural network with a first hidden layer size of 4, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

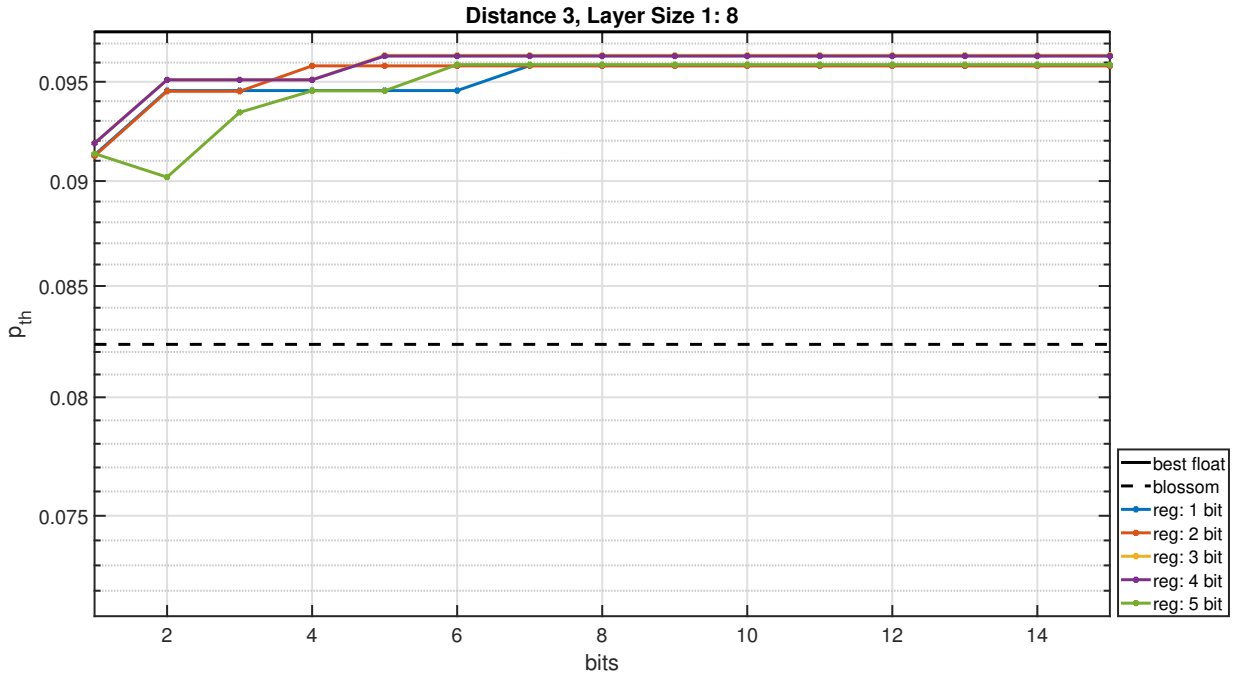


Figure B.2: Simulation results for the quantization sweeps for a surface code of distance 3. This is done on a neural network with a first hidden layer size of 8, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

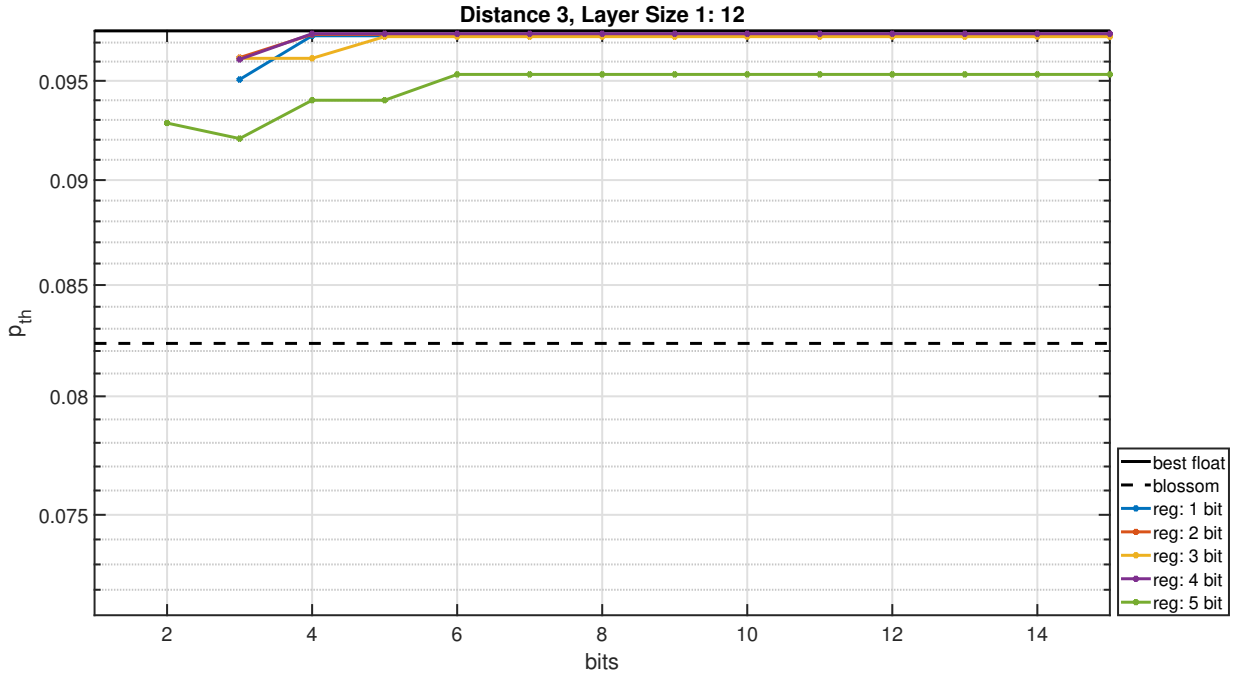


Figure B.3: Simulation results for the quantization sweeps for a surface code of distance 3. This is done on a neural network with a first hidden layer size of 12, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

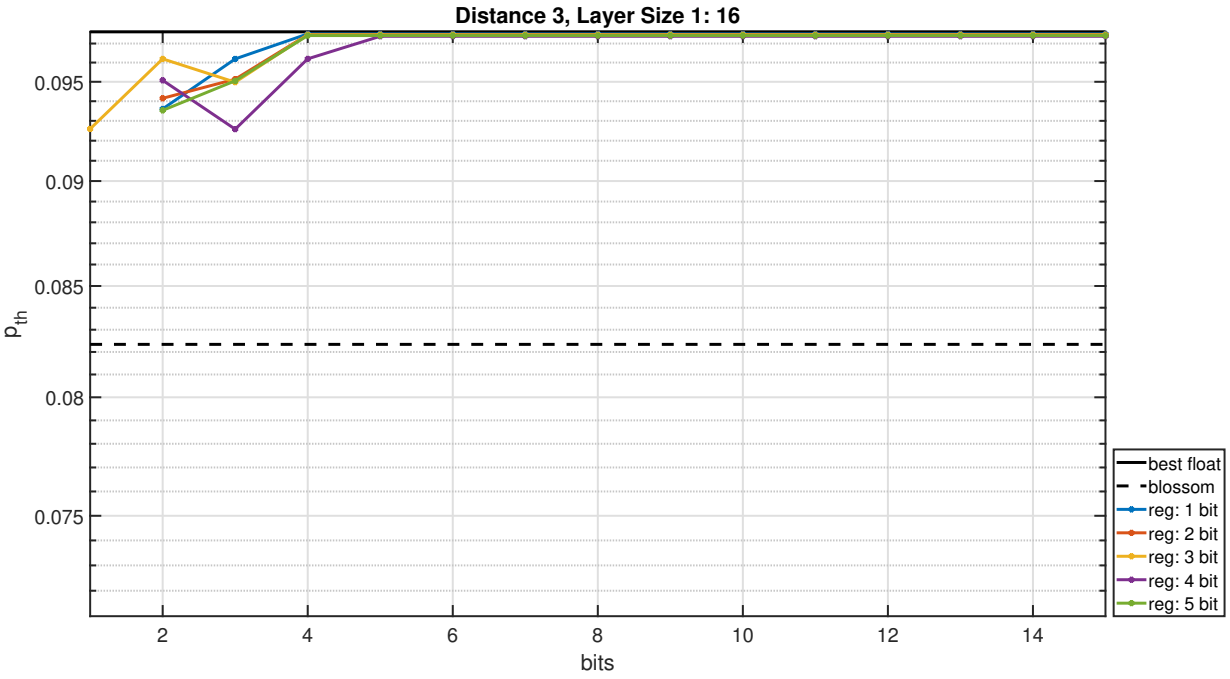


Figure B.4: Simulation results for the quantization sweeps for a surface code of distance 3. This is done on a neural network with a first hidden layer size of 16, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

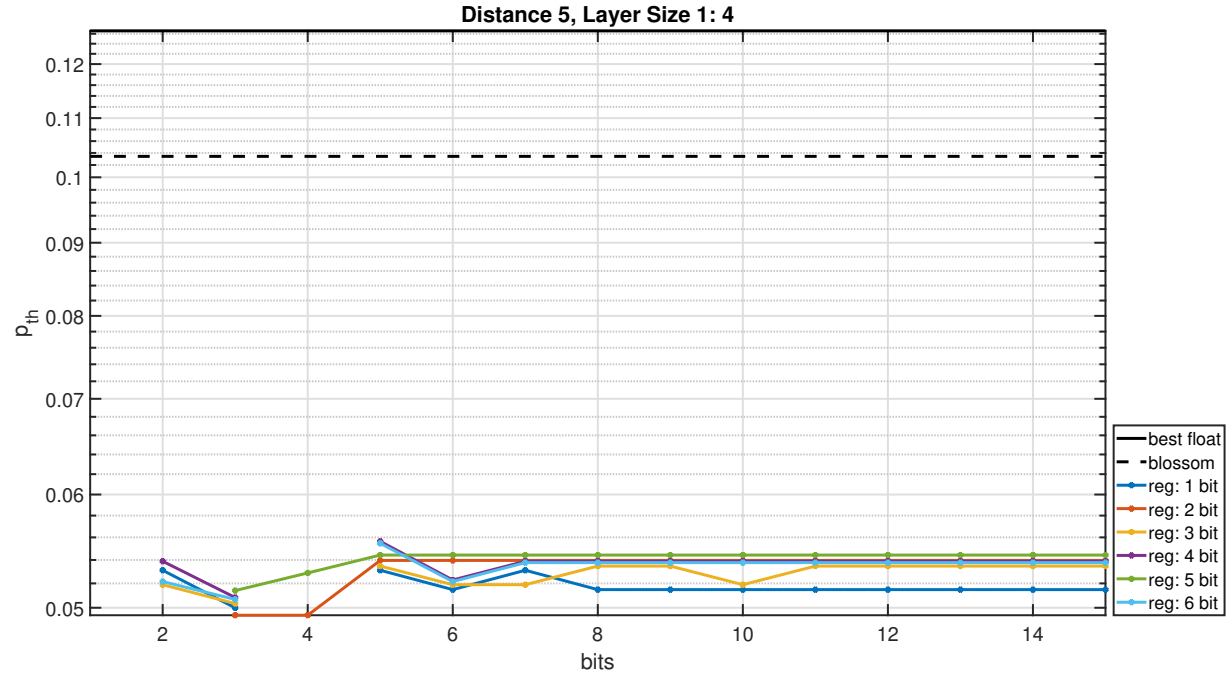


Figure B.5: Simulation results for the quantization sweeps for a surface code of distance 5. This is done on a neural network with a first hidden layer size of 4, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

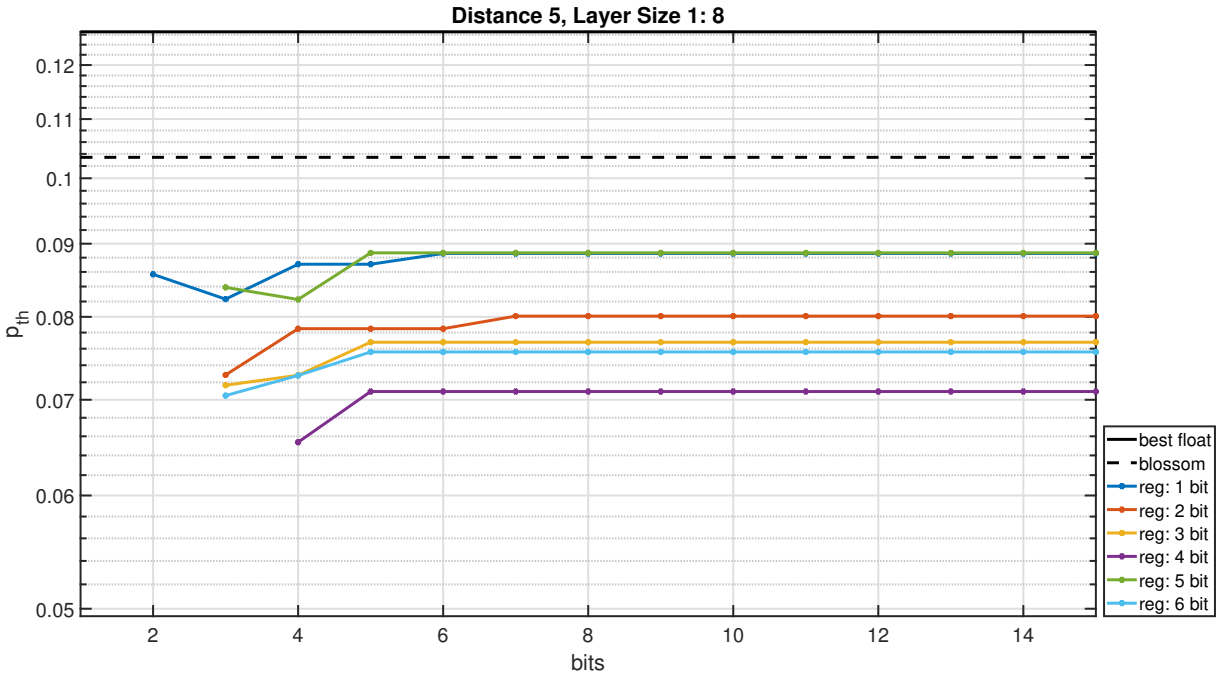


Figure B.6: Simulation results for the quantization sweeps for a surface code of distance 5. This is done on a neural network with a first hidden layer size of 8, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

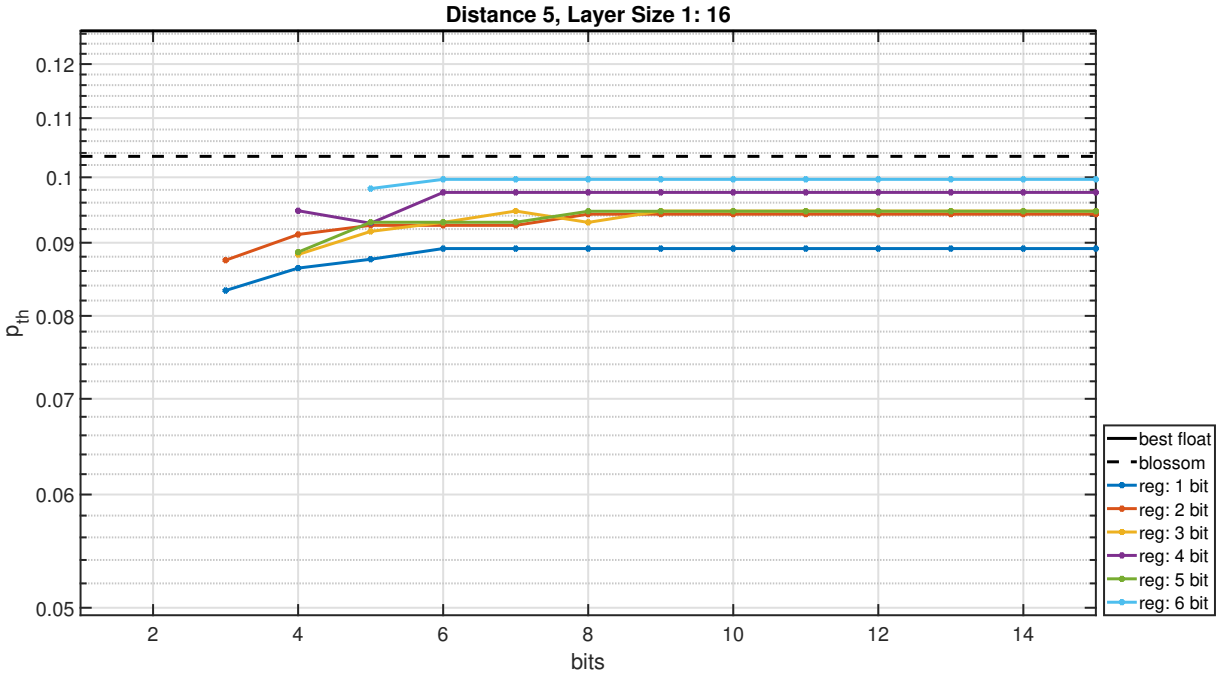


Figure B.7: Simulation results for the quantization sweeps for a surface code of distance 5. This is done on a neural network with a first hidden layer size of 16, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

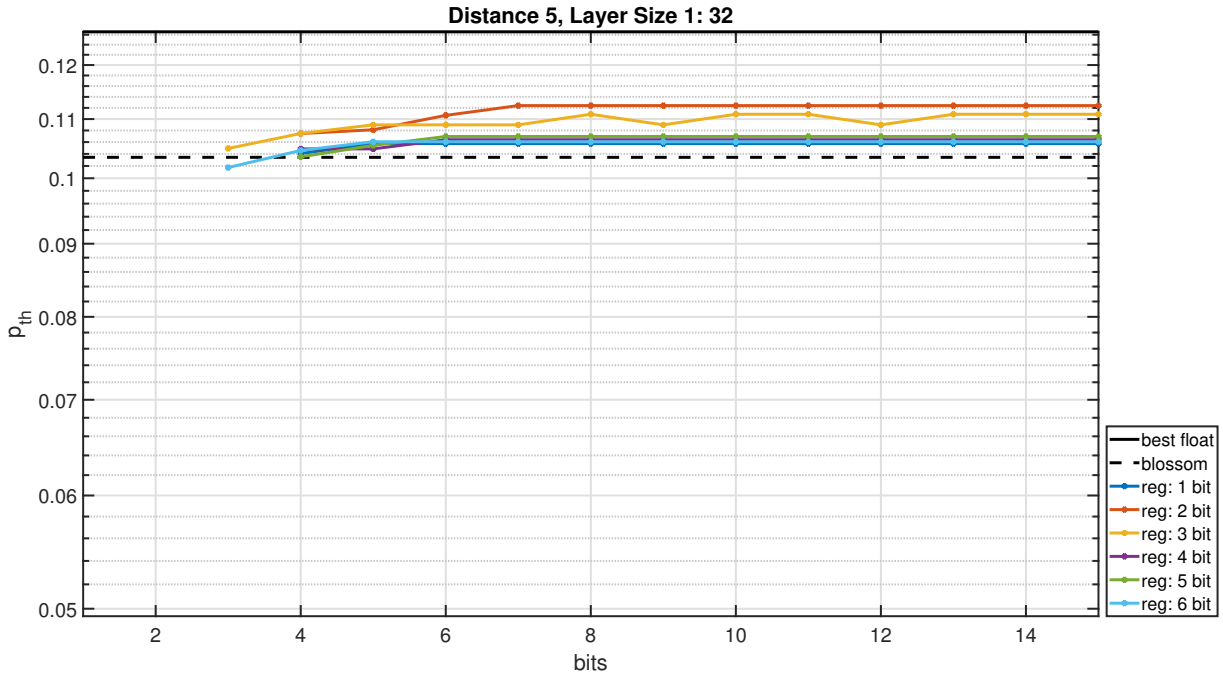


Figure B.8: Simulation results for the quantization sweeps for a surface code of distance 5. This is done on a neural network with a first hidden layer size of 32, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

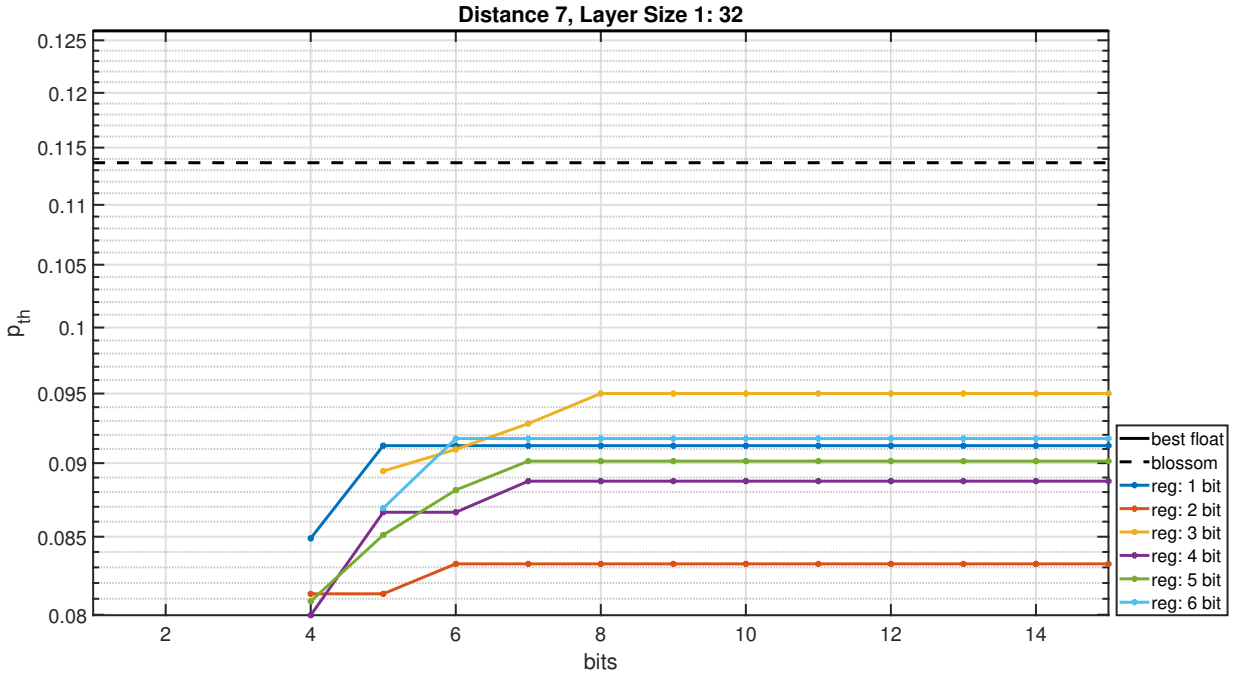


Figure B.9: Simulation results for the quantization sweeps for a surface code of distance 7. This is done on a neural network with a first hidden layer size of 32, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

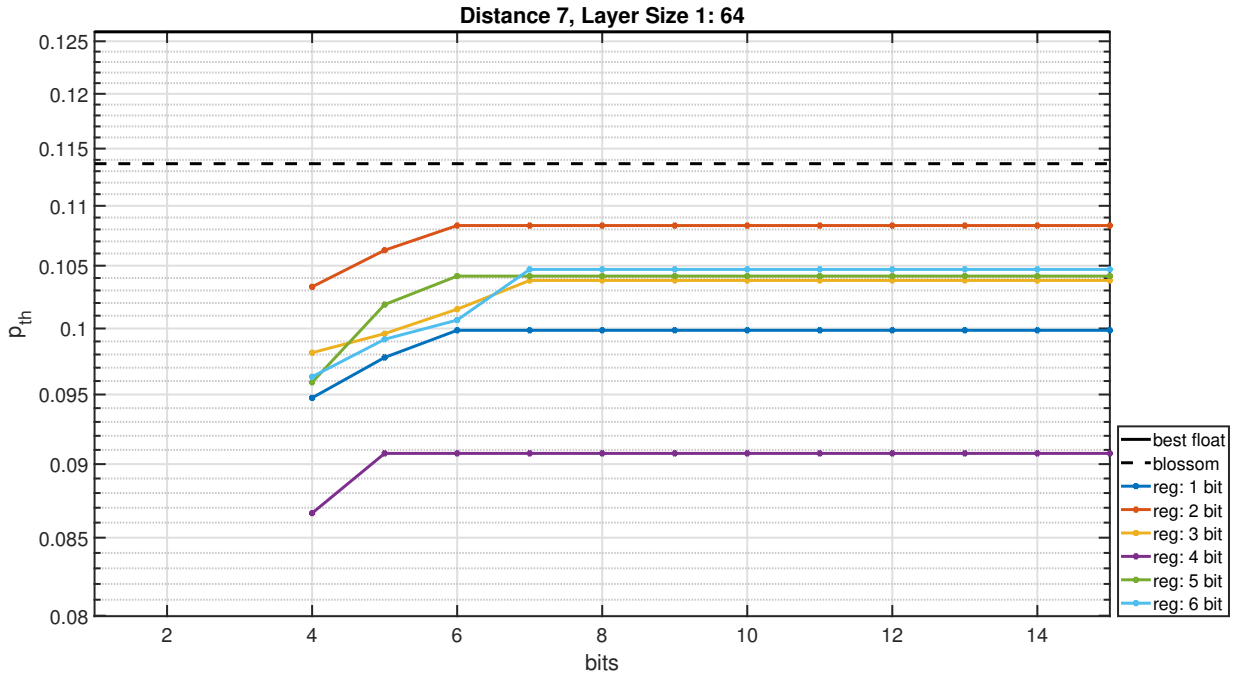


Figure B.10: Simulation results for the quantization sweeps for a surface code of distance 7. This is done on a neural network with a first hidden layer size of 64, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

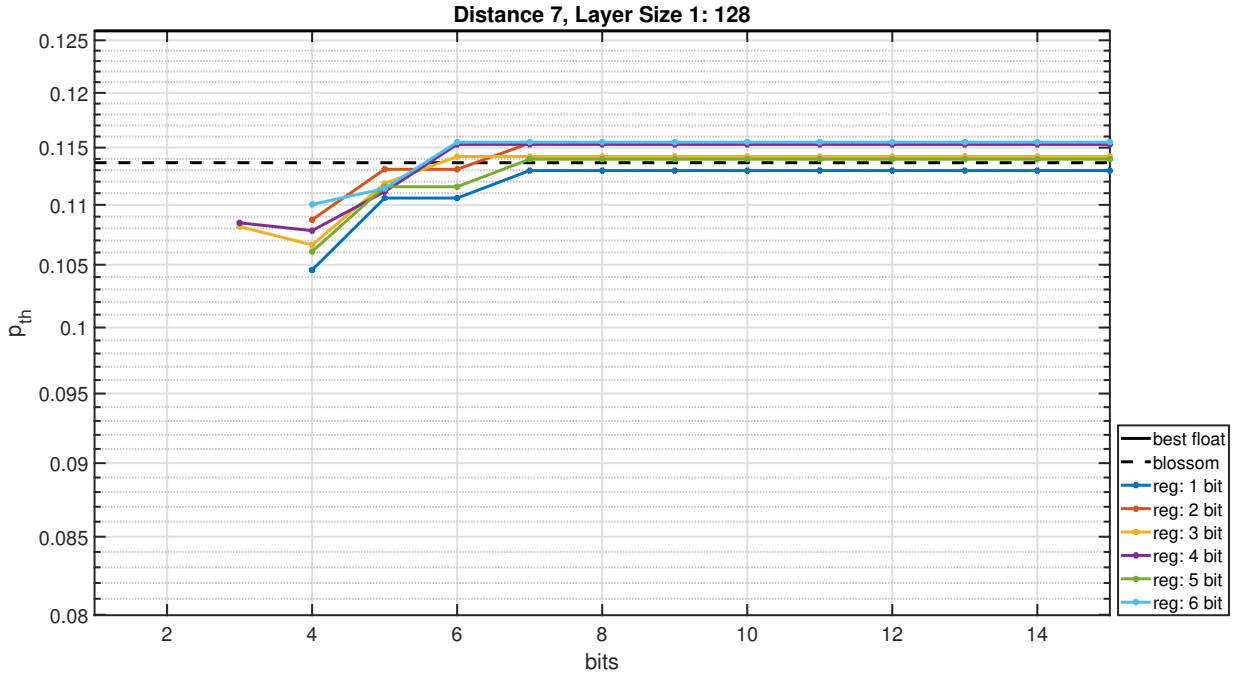


Figure B.11: Simulation results for the quantization sweeps for a surface code of distance 7. This is done on a neural network with a first hidden layer size of 128, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

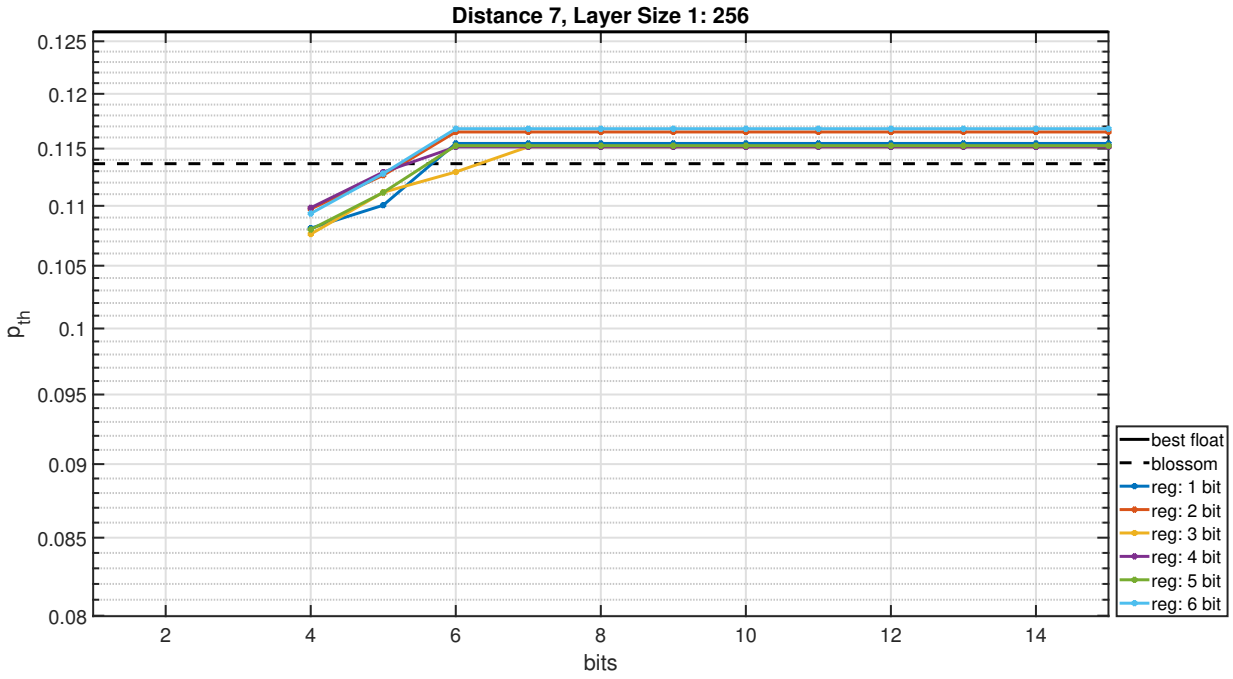


Figure B.12: Simulation results for the quantization sweeps for a surface code of distance 7. This is done on a neural network with a first hidden layer size of 256, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

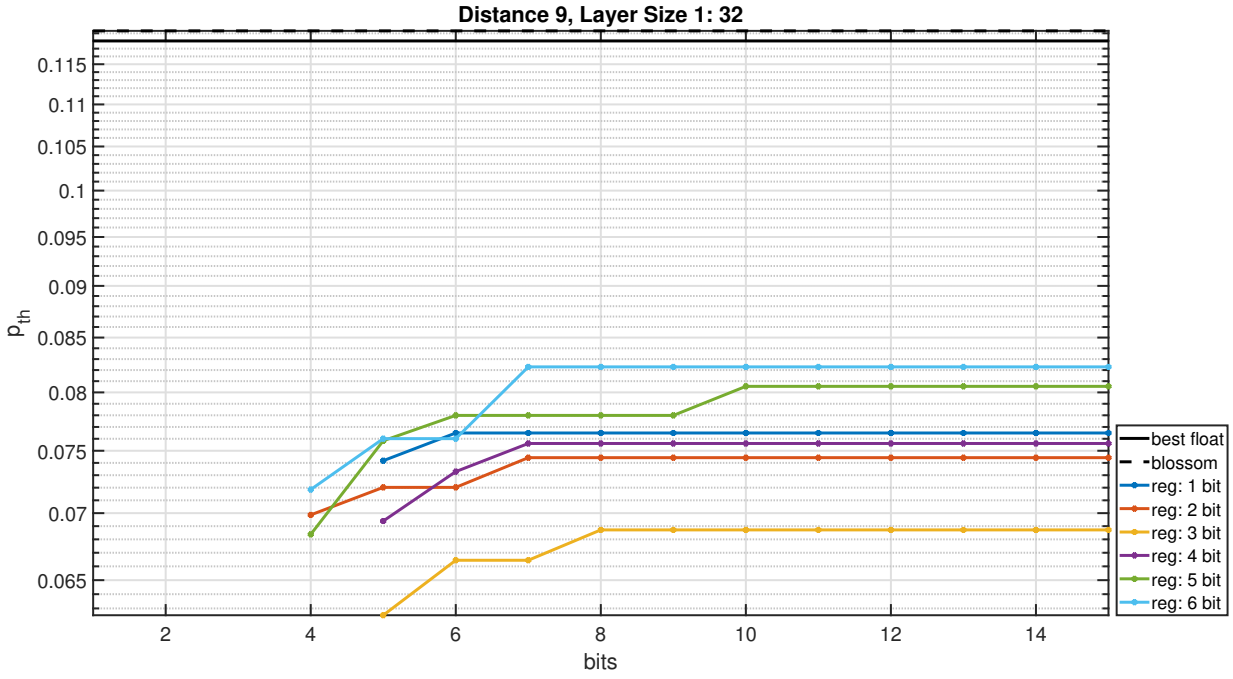


Figure B.13: Simulation results for the quantization sweeps for a surface code of distance 9. This is done on a neural network with a first hidden layer size of 32, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

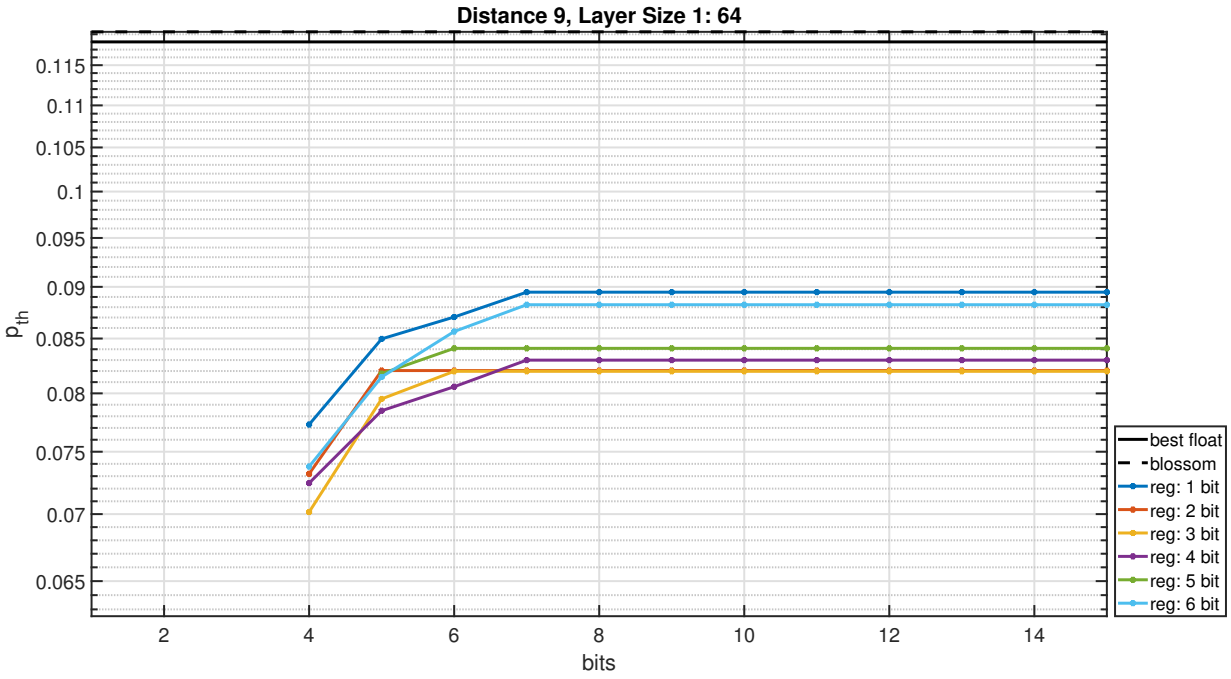


Figure B.14: Simulation results for the quantization sweeps for a surface code of distance 9. This is done on a neural network with a first hidden layer size of 64, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

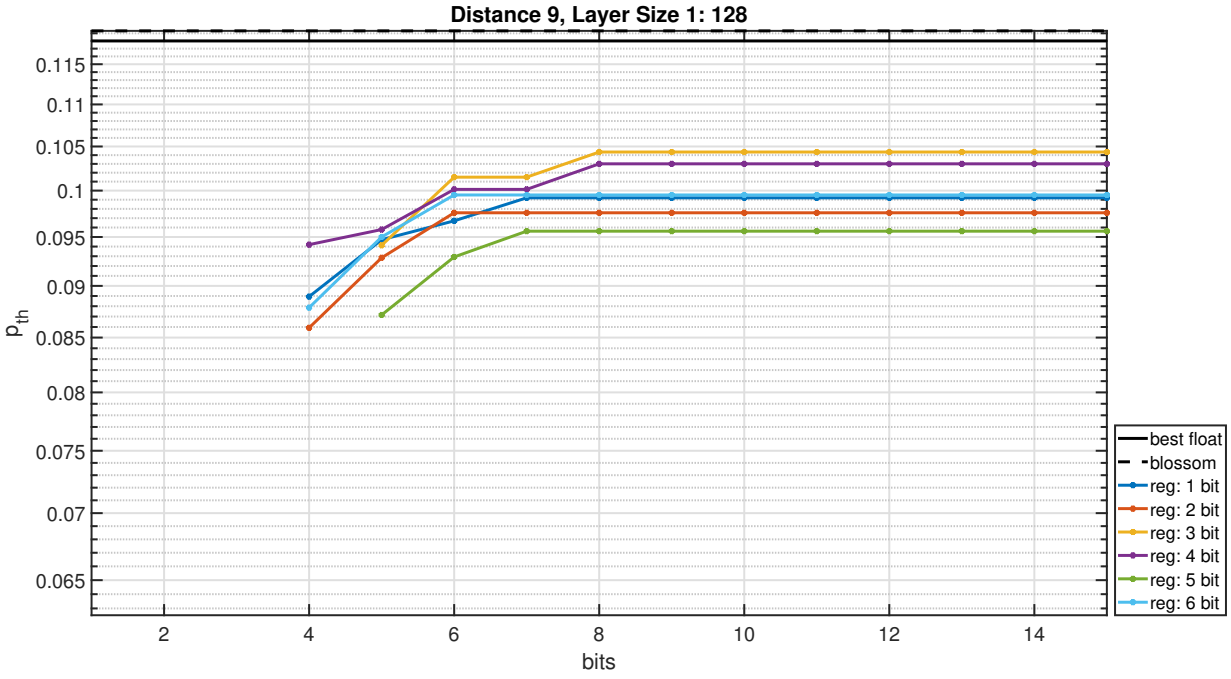


Figure B.15: Simulation results for the quantization sweeps for a surface code of distance 9. This is done on a neural network with a first hidden layer size of 128, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

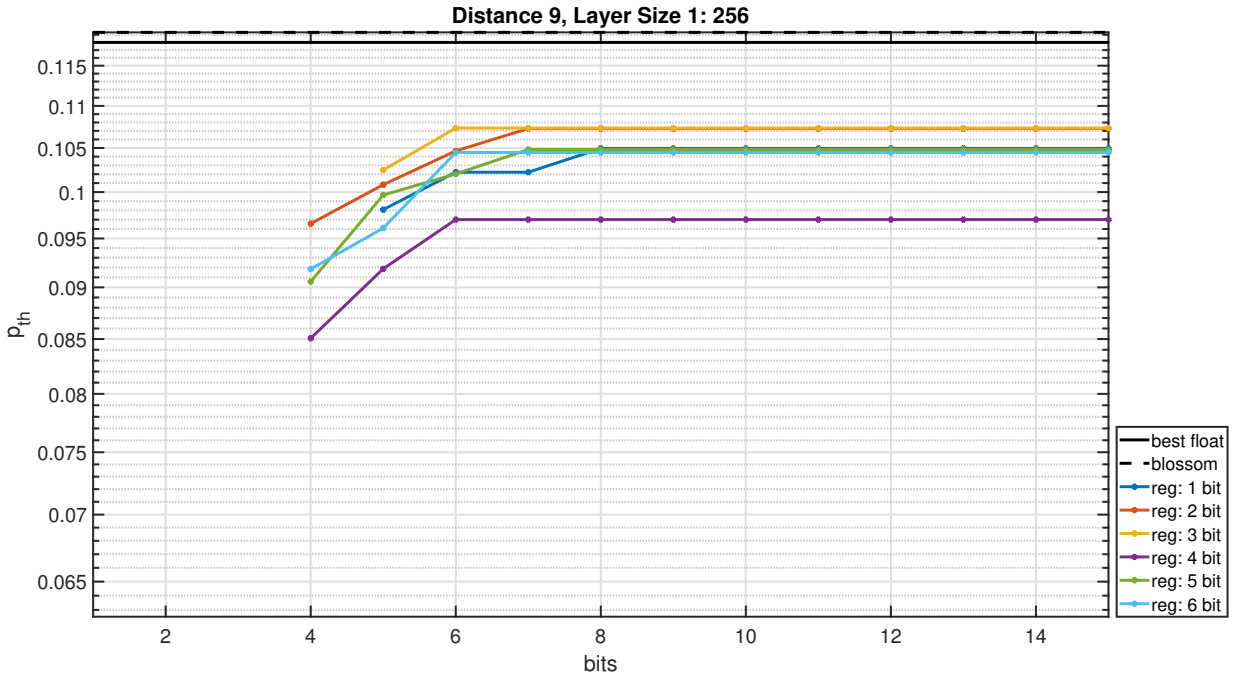


Figure B.16: Simulation results for the quantization sweeps for a surface code of distance 9. This is done on a neural network with a first hidden layer size of 256, second hidden layer size of 16 and 2 outputs using the SQNL function. On the y-axis the pseudo-threshold is to Blossom and the best floating point network. The different lines show at how many bits the regularization term is added. The x-axis shows on how many bits is rounded during quantization.

C

GPU Simulation Code

C.1. main.cu

Listing C.1: main.cu

```
#include "settings.h"
#include "functions.h"
#include "sc.h"
#include "nn.h"

int main(int argc, char** argv)
{
    SIGMOID_OUTPUT = 1;
    LAYERSIZES[0] = 8;
    for(unsigned l = 1; l < N_LAYERS+1; l++)
        LAYERSIZES[l] = N_OUTPUTS;

    int c;
    // read arguments
    while((c = getopt(argc, argv, "d:f:t:1:2:")) != -1)
    {
        switch(c)
        {
        case 'd':
            DISTANCE = atoi(optarg);
            LAYERSIZES[0] = (DISTANCE*DISTANCE-1);
            checkCudaError(cudaMemcpyToSymbol(d_DISTANCE,&DISTANCE, sizeof(unsigned
                int)));
            break;
        case 'f':
            TRANSFERSHAPE = atoi(optarg);
            checkCudaError(cudaMemcpyToSymbol(d_TRANSFERSHAPE,&TRANSFERSHAPE, sizeof(
                unsigned int)));
        case 't':
            TRANSFERFUNCTION = atoi(optarg);
            checkCudaError(cudaMemcpyToSymbol(d_TRANSFERFUNCTION,&TRANSFERFUNCTION,
                sizeof(unsigned int)));
            break;
        case '1':
            LAYERSIZES[1] = atoi(optarg);
            break;
```

```
34          case '2':
35              //LAYERSIZES[2] = atoi(optarg);
36
37              SIGMOID_OUTPUT = atoi(optarg);
38              break;
39          case 's':
40              SIGMOID_OUTPUT = 1;
41              break;
42          case '?':
43              printf("options require arguments\n");
44              exit(EXIT_FAILURE);
45              break;
46          default:
47              break;
48          }
49      }
50      LAYERSIZES[2] = 8;
51
52      POINTER_Y[0] = 0;
53      POINTER_W[0] = 0;
54      for(unsigned i = 0; i < N_LAYERS; i++)
55      {
56          POINTER_Y[i+1] = LAYERSIZES[i+1] + 1 + POINTER_Y[i];
57          POINTER_W[i+1] = (LAYERSIZES[i] + 1)*LAYERSIZES[i+1] + POINTER_W[i];
58      }
59      SIZE_W_ZP = POINTER_W[N_LAYERS] + 32 - (POINTER_W[N_LAYERS] % 32);
60      SIZE_DW_ZP = SIZE_W_ZP * BATCH_SIZE;
61
62      checkCudaError(cudaMemcpyToSymbol(d_LAYERSIZES,LAYERSIZES,sizeof(unsigned int)*(
            N_LAYERS+1)));
63      checkCudaError(cudaMemcpyToSymbol(d_POINTER_Y,POINTER_Y,sizeof(unsigned int)*(
            N_LAYERS+1)));
64      checkCudaError(cudaMemcpyToSymbol(d_POINTER_W,POINTER_W,sizeof(unsigned int)*(
            N_LAYERS+1)));
65      checkCudaError(cudaMemcpyToSymbol(d_SIZE_W_ZP,&SIZE_W_ZP,sizeof(unsigned int)));
66      checkCudaError(cudaMemcpyToSymbol(d_SIZE_DW_ZP,&SIZE_DW_ZP,sizeof(unsigned int))
            );
67      checkCudaError(cudaMemcpyToSymbol(d_SIGMOID_OUTPUT,&SIGMOID_OUTPUT,sizeof(
            unsigned char)));
68      checkCudaError(cudaMemcpyToSymbol(d_PER,&PER,sizeof(float)));
69
70      //printf("D %d TF %d,%d NN %d %d %d %d\n",DISTANCE,TRANSFERSHAPE,
            TRANSFERFUNCTION,LAYERSIZES[0],LAYERSIZES[1],LAYERSIZES[2],LAYERSIZES[3]);
71
72      NN<float> *nn = new NN<float>();
73      nn->train();
74      nn->run();
75      for(unsigned i = 1; i < 16; i++)
76      {
77          float fp = pow(2.0f,float(i));
78          printf("%d bits\n",i);
79          checkCudaError(cudaMemcpyToSymbol(d_FP,&fp,sizeof(float)));
80          nn->run_f();
81      }
82      delete nn;
83  }
```

C.2. settings.h

Listing C.2: settings.h

```
1  #pragma once
2
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <cuda.h>
6  #include <curand.h>
7  #include <curand_kernel.h>
8
9  // Having a * comment means it can be redefined in user_settings.h
10 // This is done by either
11 //
12 // Uncommenting the define line here and adding it in user_setings.h
13 //
14 // Not uncommenting the define line here but adding the #undef pragma before the new
       define in user_settings.h
15
16 // Define Working files
17 #define WEIGHT_FILE "./data/weights.file" // *
18 #define RAND_FILE "./data/rand.file" // *
19 #define WEIGHT_UPDATE_FILE "./data/update.file" // *
20
21 // Define the debugging / printing settings
22 #define DEBUG 0 // *
23
24 // Define the GPU configuration
25 // Change depending on your GPU
26 #define N_MP 28 // number of multiprocessors
27 #define N_BLOCK_PER_MP 8 // number of blocks per multiprocessor
28 #define BLOCK_SIZE 16 // number of threads per block (half a warp)
29
30 #define GRID_SIZE (N_MP * N_BLOCK_PER_MP * BLOCK_SIZE) // size of the grid
31
32 // BATCH SIZE
33 // preferable multiple of grid_size to make full use of GPU
34 #define BATCH_SIZE GRID_SIZE // *
35
36 // Neural Network Defines
37 #define N_LAYERS 3 // *
38 #define N_LAYERS_MAX 5
39 #define N_OUTPUTS 2
40
41 #define BETA1 0.9f
42 #define BETA2 0.999f
43 #define ALPHA 0.001f
44 // Transferfunction defines
45 #define TF_SIGMOID 0
46 #define TF_RELU 1
47
48 #define TF_MODE TF_SIGMOID // *
49
50 // Arithmetic defines
51 #define ARITH_FLOAT 0
52 #define ARITH_FIXED 1
```

```
53 #define ARITH_MODE ARITH_FLOAT // *
54
55 // TEST
56 #define P_TEST_MIN 0.03946f
57 #define P_TEST_MAX 0.3f
58 #define P_TEST_INC 1.08517f
59 #define P_TRAIN_MIN 0.005f
60 #define P_TRAIN_START 0.15f
61 #define RUN_MAX 10000
62 #define TRAIN_MAX 100
63
64 // RNG defines
65 #define SEED 0 // *
66
67 // CPHASE
68 #define CPBR 0
69 #define CPBL 1
70 #define CPUR 2
71 #define CPUL 3
72
73 // globals
74 unsigned int DISTANCE = 3;
75 unsigned int LAYERSIZES[N_LAYERS+1] = {0};
76 unsigned int POINTER_Y[N_LAYERS+1] = {0};
77 unsigned int POINTER_W[N_LAYERS+1] = {0};
78 unsigned int TRANSFERFUNCTION = 0;
79 unsigned int TRANSFERSHAPE = 0;
80 unsigned int SIZE_W_ZP = 0;
81 unsigned int SIZE_DW_ZP = 0;
82 unsigned int SIGMOID_OUTPUT = 0;
83 float PER = 0.1f;
84
85 __constant__ unsigned int d_DISTANCE;
86 __constant__ unsigned int d_LAYERSIZES[N_LAYERS+1];
87 __constant__ unsigned int d_POINTER_Y[N_LAYERS+1];
88 __constant__ unsigned int d_POINTER_W[N_LAYERS+1];
89 __constant__ unsigned int d_TRANSFERFUNCTION;
90 __constant__ unsigned int d_TRANSFERSHAPE;
91 __constant__ unsigned int d_SIZE_W_ZP;
92 __constant__ unsigned int d_SIZE_DW_ZP;
93 __constant__ unsigned char d_SIGMOID_OUTPUT;
94 __constant__ float d_PER;
95 __constant__ float d_FP;
96 __constant__ float d_MAX_W[N_LAYERS];
97
98 #define DISTANCE_MAX 9
99 #define SIZE_Y_MAX 300
```

C.3. functions.h

Listing C.3: functions.h

```
1 #pragma once
2
3 #include "settings.h"
4 #define checkMalloc(x) checkMallocLine(__FILE__,__LINE__,x)
```

```
5  #define checkCudaError(x) checkCudaErrorLine(__FILE__,__LINE__,x)
6
7  // quantize
8  inline __device__ float quantize(float f)
9  {
10     return round(f*d_FP)/d_FP;
11 }
12
13 // cuda error
14 inline cudaError_t checkCudaErrorLine(const char *file, unsigned line,cudaError_t
       cudaStatus)
15 {
16     #if defined(DEBUG)
17         if(cudaStatus != cudaSuccess)
18         {
19             printf("cuda failed! file: %s, line: %d, error code (%d): %s\n",file,
                   line,(unsigned)cudaStatus, cudaGetErrorString(cudaStatus));
20             exit(EXIT_FAILURE);
21         }
22     #endif
23     return cudaStatus;
24 }
25
26 // regular malloc check
27 inline void* checkMallocLine(const char *file, unsigned line, void *ptr)
28 {
29     #if defined(DEBUG)
30         if(ptr == NULL)
31         {
32             printf("malloc failed! file: %s, line: %d\n",file,line);
33             exit(EXIT_FAILURE);
34         }
35     #endif
36     return ptr;
37 }
38
39 // surface code
40 inline __device__ void qubit_ry(char *q)
41 {
42     *q = (((*q & 2) >> 1) + ((*q & 1) << 1));
43 }
44
45 //cphase
46 inline __device__ void qubit_cp(char *q1, char *q2)
47 {
48     *q1 = *q1 ^ ((*q2 & 1) << 1);
49     *q2 = *q2 ^ ((*q1 & 1) << 1);
50 }
51
52 inline __device__ void roty_data(char *data, unsigned idx)
53 {
54     for(unsigned d = idx * (d_DISTANCE*d_DISTANCE); d < (idx + 1) * (d_DISTANCE*
          d_DISTANCE); d++)
55     {
56         qubit_ry(&data[d]);
57     }
```

```
58  }
59
60  inline __device__ void roty_data_odd(char *data, unsigned idx)
61  {
62      for(unsigned d = 1 + idx * (d_DISTANCE*d_DISTANCE); d < (idx + 1) * (d_DISTANCE*
            d_DISTANCE); d+=2)
63      {
64          qubit_ry(&data[d]);
65      }
66  }
67
68  inline __device__ void roty_ancilla(char *ancilla, unsigned idx)
69  {
70      for(unsigned d = idx * (d_DISTANCE*d_DISTANCE-1); d < (idx + 1) * (d_DISTANCE*
            d_DISTANCE-1); d++)
71      {
72          qubit_ry(&ancilla[d]);
73      }
74  }
75
76  inline __device__ void cphase_dir_ext_norot(char *data, char *ancilla, char p,
        unsigned idx)
77  {
78      for(unsigned r = 0; r < d_DISTANCE; r++)
79      {
80          for(unsigned c = (p+1)/2; c < d_DISTANCE + (p-1)/2; c++)
81          {
82              // odd
83              unsigned d = r * d_DISTANCE + c;
84              if(d%2 == 1)
85              {
86                  unsigned a = (d_DISTANCE+1)/2 * (c - (p+1)/2) + (r - p * (r%2))/2;
87                  a += idx * (d_DISTANCE*d_DISTANCE-1);
88                  d += idx * (d_DISTANCE*d_DISTANCE);
89                  qubit_cp(&data[d],&ancilla[a]);
90              }//even
91              else
92              {
93                  d = c * d_DISTANCE + r;
94                  unsigned a = (d_DISTANCE*d_DISTANCE-1)/2 + (d_DISTANCE+1)/2 * (c - (
                        p+1)/2) + (d_DISTANCE - 1 - r + p * (r%2))/2;
95                  a += idx * (d_DISTANCE*d_DISTANCE-1);
96                  d += idx * (d_DISTANCE*d_DISTANCE);
97                  qubit_cp(&data[d],&ancilla[a]);
98              }
99          }
100     }
101 }
102 inline __device__ void cphase_dir_ext_rot(char *data, char *ancilla, char p,
        unsigned idx)
103 {
104     for(unsigned r = 0; r < d_DISTANCE; r++)
105     {
106         for(unsigned c = (p+1)/2; c < d_DISTANCE + (p-1)/2; c++)
107         {
108             // even
```

```
109             unsigned d = r * d_DISTANCE + c;
110             if(d%2 == 0)
111             {
112                 unsigned a = (d_DISTANCE+1)/2 * (c - (p+1)/2) + (r + p * (r%2))/2;
113                 a += idx * (d_DISTANCE*d_DISTANCE-1);
114                 d += idx * (d_DISTANCE*d_DISTANCE);
115                 qubit_cp(&data[d],&ancilla[a]);
116             }// odd
117             d = (c-p) * d_DISTANCE + r;
118             if(d%2 == 1)
119             {
120                 unsigned a = (d_DISTANCE*d_DISTANCE-1)/2 + (d_DISTANCE+1)/2 * ((c-p)
                        + (p-1)/2) + (d_DISTANCE - 1 - r + p * (r%2))/2;
121                 a += idx * (d_DISTANCE*d_DISTANCE-1);
122                 d += idx * (d_DISTANCE*d_DISTANCE);
123                 qubit_cp(&data[d],&ancilla[a]);
124             }
125         }
126     }
127 }
128 // cphase in direction
129 inline __device__ void cphase_dir(char *data, char *ancilla, char dir, unsigned idx)
130 {
131     char r = -1 + (dir & 2);
132     char c = -1 + 2*(dir & 1);
133
134     if(r == c)
135         cphase_dir_ext_norot(data, ancilla, r, idx);
136     else
137         cphase_dir_ext_rot(data,ancilla, r, idx);
138 }
139
140 inline __device__ void zero_data(char *data, unsigned idx)
141 {
142     for(unsigned d = idx*(d_DISTANCE*d_DISTANCE); d < (1 + idx) * (d_DISTANCE*
        d_DISTANCE); d++)
143         data[d] = 0;
144 }
145
146 inline __device__ void depolarize(char *data, float p, curandStateMRG32k3a *rand,
    unsigned idx)
147 {
148     for(unsigned d = idx * (d_DISTANCE*d_DISTANCE); d < (1 + idx) * (d_DISTANCE*
        d_DISTANCE); d++)
149     {
150         float r = curand_uniform(rand);
151         if(r < p)
152             data[d] ^= 1 + (curand(rand) % 3);
153     }
154 }
155
156 inline __device__ void zero_ancilla(char *ancilla, unsigned idx)
157 {
158     for(unsigned d = idx*(d_DISTANCE*d_DISTANCE-1); d < (1 + idx) * (d_DISTANCE*
        d_DISTANCE-1); d++)
159         ancilla[d] = 0;
```

```
160 }
161
162 inline __device__ void measure_ancilla(char *ancilla, unsigned idx)
163 {
164     for(unsigned a = idx*(d_DISTANCE*d_DISTANCE-1); a < (1 + idx)* (d_DISTANCE*
        d_DISTANCE-1); a++)
165         ancilla[a] &= 1;
166 }
167
168 inline __device__ void get_pure_error(char *pure, char *ancilla, unsigned idx)
169 {
170     for(char p = -1; p < 2; p+=2)
171     {
172         unsigned r0 = (d_DISTANCE - 2 + p)/2;
173         for(unsigned c0 = 0; c0 < (d_DISTANCE+1)/2; c0++)
174         {
175             unsigned tx = 0, tz = 0;
176             for(unsigned i = 0; i < (d_DISTANCE-1)/2; i++)
177             {
178                 unsigned r = r0 + p * i;
179                 unsigned ax = r * (d_DISTANCE + 1)/2 + c0;
180                 unsigned az = ax + (d_DISTANCE*d_DISTANCE-1)/2;
181                 unsigned dx = r + 2*d_DISTANCE*c0 + (1+p)/2;
182                 unsigned dz = (r + (3+p)/2)*d_DISTANCE -1 - 2 * c0;
183
184                 tx ^= ancilla[ax + idx * (d_DISTANCE*d_DISTANCE-1)] << 1;
185                 tz ^= ancilla[az + idx * (d_DISTANCE*d_DISTANCE-1)];
186
187                 pure[dx + idx * (d_DISTANCE*d_DISTANCE)] ^= tx;
188                 pure[dz + idx * (d_DISTANCE*d_DISTANCE)] ^= tz;
189             }
190         }
191     }
192 }
193
194 inline __device__ unsigned char get_logical_error(char *pure, char *data, unsigned
    idx)
195 {
196     unsigned char diff = 0;
197
198     for(unsigned i = idx*(d_DISTANCE*d_DISTANCE); i < d_DISTANCE + idx*(d_DISTANCE*
        d_DISTANCE); i++)
199     {
200         diff ^= (pure[i]^data[i])&1;
201     }
202     for(unsigned i = idx*(d_DISTANCE*d_DISTANCE); i < (1+idx) * (d_DISTANCE*
        d_DISTANCE); i+=d_DISTANCE)
203     {
204         diff ^= (pure[i]^data[i])&2;
205     }
206     return diff;
207 }
208
209 inline __device__ void save_generated_data(unsigned char *ancilla_data, unsigned
    char *target_data, char *ancilla, unsigned char target, unsigned idx, unsigned g)
210 {
```

```
211     ancilla_data[(d_DISTANCE*d_DISTANCE-1)*BATCH_SIZE+g] = 1;
212
213     //save
214     target_data[g] = target;
215     for(unsigned i = 0; i < (d_DISTANCE*d_DISTANCE-1); i++)
216     {
217         ancilla_data[i*BATCH_SIZE + g] = (unsigned char)ancilla[i + idx*(d_DISTANCE*
                d_DISTANCE-1)];
218     }
219 }
```

C.4. iofunctions.h

Listing C.4: iofunctions.h

```
1  #pragma once
2
3  #include "settings.h"
4  #include "functions.h"
5
6  void read_weights(float** h_weights)
7  {
8      *h_weights = (float*)checkMalloc(malloc(SIZE_W_ZP * sizeof(float)));
9      FILE *fp = fopen(WEIGHT_FILE,"r");
10     if(fp == NULL)
11     {
12         printf("WEIGHT FILE %s does not exist!\n",WEIGHT_FILE);
13         exit(EXIT_FAILURE);
14     }
15     fread(*h_weights,sizeof(float),SIZE_W_ZP,fp);
16 }
17
18 void write_weights(float* h_weights)
19 {
20     FILE *fp = fopen(WEIGHT_FILE,"w+");
21     fwrite(h_weights,sizeof(float),SIZE_W_ZP,fp);
22     fclose(fp);
23 }
24
25 void copy_weights_to_tex(float *h_weights, cudaArray* d_weights_array)
26 {
27     checkCudaError(cudaMemcpyToArray(d_weights_array, 0, 0, h_weights, SIZE_W_ZP *
            sizeof(float),cudaMemcpyHostToDevice));
28 }
29
30 void copy_weights_from_tex(float *h_weights, cudaArray* d_weights_array)
31 {
32     checkCudaError(cudaMemcpyFromArray(h_weights, d_weights_array, 0, 0, SIZE_W_ZP *
             sizeof(float), cudaMemcpyDeviceToHost));
33 }
34
35 void init_weights_to_tex(float *h_weights, cudaArray** d_weights_array,
        cudaTextureObject_t *d_weights_texture)
36 {
37     cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0 ,0 ,0,
            cudaChannelFormatKindFloat);
```

```
38      checkCudaError(cudaMallocArray(d_weights_array, &channelDesc, SIZE_W_ZP));
39
40      struct cudaResourceDesc resDesc;
41      memset(&resDesc, 0, sizeof(resDesc));
42      resDesc.resType = cudaResourceTypeArray;
43      resDesc.res.array.array = *d_weights_array;
44
45      struct cudaTextureDesc texDesc;
46      memset(&texDesc, 0, sizeof(texDesc));
47      texDesc.addressMode[0] = cudaAddressModeBorder;
48      texDesc.filterMode = cudaFilterModePoint;
49      texDesc.readMode = cudaReadModeElementType;
50      texDesc.normalizedCoords = 0;
51
52      cudaCreateTextureObject(d_weights_texture, &resDesc, &texDesc, NULL);
53
54      checkCudaError(cudaMemcpyToArray(*d_weights_array, 0, 0, h_weights, SIZE_W_ZP *
            sizeof(float),cudaMemcpyHostToDevice));
55  }
56
57  void read_rng(curandStateMRG32k3a *d_randstates)
58  {
59      curandStateMRG32k3a *h_randstates = (curandStateMRG32k3a*)checkMalloc(malloc(
            GRID_SIZE * sizeof(curandStateMRG32k3a)));
60
61      FILE *fp = fopen(RAND_FILE,"r");if(fp == NULL)
62          {
63              printf("RAND FILE %s does not exist!\n",RAND_FILE);
64              exit(EXIT_FAILURE);
65          }
66          // read rng
67      fread(h_randstates,sizeof(curandStateMRG32k3a),GRID_SIZE,fp);
68
69      // copy
70      checkCudaError(cudaMemcpy(d_randstates,h_randstates,GRID_SIZE*sizeof(
            curandStateMRG32k3a),cudaMemcpyHostToDevice));
71
72          // clean up
73          fclose(fp);
74          free(h_randstates);
75  }
76
77  void write_rng(curandStateMRG32k3a *d_randstates)
78  {
79      // allocate host side memory
80      curandStateMRG32k3a *h_randstates = (curandStateMRG32k3a*)checkMalloc(malloc(
            GRID_SIZE * sizeof(curandStateMRG32k3a)));
81
82      // open file
83      FILE *fp = fopen(RAND_FILE,"w+");
84      if(fp == NULL)
85      {
86          printf("RAND FILE %s does not exist!\n",RAND_FILE);
87          exit(EXIT_FAILURE);
88      }
89      // copy
```

```
90      checkCudaError(cudaMemcpy(h_randstates,d_randstates,GRID_SIZE*sizeof(
            curandStateMRG32k3a),cudaMemcpyDeviceToHost));
91
92      // save rng
93      fwrite(h_randstates,sizeof(curandStateMRG32k3a),GRID_SIZE,fp);
94
95      // clean up
96      fclose(fp);
97      free(h_randstates);
98  }
99
100 void read_update_weights(float **d_wm, float **d_wv)
101 {
102     // allocate
103     checkCudaError(cudaMalloc(d_wm, SIZE_W_ZP * sizeof(float)));
104     checkCudaError(cudaMalloc(d_wv, SIZE_W_ZP * sizeof(float)));
105
106     float *h_w = (float*)checkMalloc(malloc(2*SIZE_W_ZP * sizeof(float)));
107
108     // read
109     FILE *fp = fopen(WEIGHT_UPDATE_FILE,"r");
110     if(fp == NULL)
111     {
112         printf("WEIGHT UPDATE FILE %s does not exist!\n",WEIGHT_UPDATE_FILE);
113         exit(EXIT_FAILURE);
114     }
115     //read
116     fread(h_w,sizeof(float),2*SIZE_W_ZP,fp);
117
118     // copy
119     checkCudaError(cudaMemcpy(*d_wm,h_w,SIZE_W_ZP*sizeof(float),
            cudaMemcpyHostToDevice));
120     checkCudaError(cudaMemcpy(*d_wv,&h_w[SIZE_W_ZP],SIZE_W_ZP*sizeof(float),
            cudaMemcpyHostToDevice));
121
122     //clean
123     fclose(fp);
124     free(h_w);
125 }
126
127 void write_update_weights(float *d_wm, float *d_wv)
128 {
129     // allocate
130     float *h_w = (float*)checkMalloc(malloc(2*SIZE_W_ZP * sizeof(float)));
131
132     // open
133     FILE *fp = fopen(WEIGHT_UPDATE_FILE,"w+");
134
135     // copy
136     checkCudaError(cudaMemcpy(h_w,d_wm,SIZE_W_ZP*sizeof(float),
            cudaMemcpyDeviceToHost));
137     checkCudaError(cudaMemcpy(&h_w[SIZE_W_ZP],d_wv,SIZE_W_ZP*sizeof(float),
            cudaMemcpyDeviceToHost));
138
139     // write
140     fwrite(h_w,sizeof(float),2*SIZE_W_ZP,fp);
```

```
141
142     //clean
143     fclose(fp);
144     free(h_w);
145 }
```

C.5. kernels.h

Listing C.5: kernels.h

```
1  #pragma once
2
3  #include "settings.h"
4  #include "functions.h"
5
6  __global__
7  void generate_surface_code_data(unsigned char *ancilla_data, unsigned char *
       target_data, curandStateMRG32k3a *randstates)
8  {
9      // get indices
10     unsigned block_thread = threadIdx.x;
11     unsigned grid_thread = blockIdx.x*blockDim.x + threadIdx.x;
12
13     // rng
14     curandStateMRG32k3a *randstate = &randstates[grid_thread];
15
16     // shared variables
17     const unsigned size_all = DISTANCE_MAX * DISTANCE_MAX * BLOCK_SIZE;
18     __shared__ char ancilla[size_all], data[size_all], pure[size_all];
19
20     // grid stride loop
21     for(unsigned g = grid_thread; g < BATCH_SIZE; g += GRID_SIZE)
22     {
23         zero_ancilla(ancilla,block_thread);
24         zero_data(pure,block_thread);
25         zero_data(data,block_thread);
26
27         depolarize(data, d_PER, randstate, block_thread);
28
29         __syncthreads();
30
31         // run surface code cycle
32         roty_ancilla(ancilla,block_thread);
33         roty_data_odd(data,block_thread);
34         cphase_dir(data,ancilla,CPBR,block_thread);
35         roty_data(data,block_thread);
36         cphase_dir(data,ancilla,CPUR,block_thread);
37         cphase_dir(data,ancilla,CPBL,block_thread);
38         roty_data(data,block_thread);
39         cphase_dir(data,ancilla,CPUL,block_thread);
40         roty_data_odd(data,block_thread);
41         roty_ancilla(ancilla,block_thread);
42
43         measure_ancilla(ancilla,block_thread);
44
45         __syncthreads();
```

```
46
47          get_pure_error(pure,ancilla,block_thread);
48
49          unsigned char diff = get_logical_error(pure,data,block_thread);
50
51          save_generated_data(ancilla_data,target_data,ancilla,diff,block_thread,g);
52
53          //printf(" %d  \t%d%d%d\t%d%d%d\n %d%d%d\t%d%d%d\t%d%d%d\n%d%d%d \t%d%d%d\t%
                d%d%d\n  %d\n%d\n",ancilla[0+block_thread*(d_DISTANCE*d_DISTANCE-1)],data
                [0+block_thread*(d_DISTANCE*d_DISTANCE)],data[1+block_thread*(d_DISTANCE*
                d_DISTANCE)],data[2+block_thread*(d_DISTANCE*d_DISTANCE)],pure[0+
                block_thread*(d_DISTANCE*d_DISTANCE)],pure[1+block_thread*(d_DISTANCE*
                d_DISTANCE)],pure[2+block_thread*(d_DISTANCE*d_DISTANCE)],ancilla[5+
                block_thread*(d_DISTANCE*d_DISTANCE-1)],ancilla[2+block_thread*(
                d_DISTANCE*d_DISTANCE-1)],ancilla[4+block_thread*(d_DISTANCE*d_DISTANCE
                -1)],data[3+block_thread*(d_DISTANCE*d_DISTANCE)],data[4+block_thread*(
                d_DISTANCE*d_DISTANCE)],data[5+block_thread*(d_DISTANCE*d_DISTANCE)],pure
                [3+block_thread*(d_DISTANCE*d_DISTANCE)],pure[4+block_thread*(d_DISTANCE*
                d_DISTANCE)],pure[5+block_thread*(d_DISTANCE*d_DISTANCE)],ancilla[7+
                block_thread*(d_DISTANCE*d_DISTANCE-1)],ancilla[1+block_thread*(
                d_DISTANCE*d_DISTANCE-1)],ancilla[6+block_thread*(d_DISTANCE*d_DISTANCE
                -1)],data[6+block_thread*(d_DISTANCE*d_DISTANCE)],data[7+block_thread*(
                d_DISTANCE*d_DISTANCE)],data[8+block_thread*(d_DISTANCE*d_DISTANCE)],pure
                [6+block_thread*(d_DISTANCE*d_DISTANCE)],pure[7+block_thread*(d_DISTANCE*
                d_DISTANCE)],pure[8+block_thread*(d_DISTANCE*d_DISTANCE)],ancilla[3+
                block_thread*(d_DISTANCE*d_DISTANCE-1)],diff);
54
55          __syncthreads();
56      }
57  }
58
59  __global__
60  void run_fixed(unsigned char *input, unsigned char *output, cudaTextureObject_t
        weights)
61  {
62          // get batch number referred to both the total grid and the sub block
63          unsigned grid_thread = blockIdx.x*blockDim.x + threadIdx.x;
64          unsigned block_thread = threadIdx.x;
65
66          // allocate shared memory
67      // holds the output (y) and its derivative (d) of each layer for all threads in
            this block
68      // during backpropagation these are also used for calculating the deltas and
            derivatives
69          __shared__ float y[BLOCK_SIZE * SIZE_Y_MAX];
70
71      // run for all data in this batch using a grid stride loop
72          #pragma unroll
73          for(unsigned g = grid_thread; g < BATCH_SIZE; g+= GRID_SIZE)
74          {
75
76              // set biasses
77          // the bias is the last input of a layer, it should always be 1
78              #pragma unroll
79              for(unsigned l = 0; l < N_LAYERS; l++)
80              {
```

```
81              // for layer n, the index of the bias should be the one before the
                    pointer to the next layer
82              unsigned ii = d_POINTER_Y[l+1] - 1;
83              // select the y that corresponds with this thread (add the size of the y
                     vector, thread times)
84              ii += block_thread * d_POINTER_Y[N_LAYERS];
85              // set the bias to 1
86                      y[ii] = d_LAYERSIZES[l]/d_TRANSFERFUNCTION;
87              }
88
89          // run network given the input data and weights
90              // run through all layers one by one
91              #pragma unroll
92              for(unsigned l = 0; l < N_LAYERS; l++)
93              {
94                      // loop through the outputs of layer l
95                      #pragma unroll
96                      for(unsigned j = 0; j < d_LAYERSIZES[l+1]; j++)
97                      {
98                              // initialize the input (x) to the transferfunction
99                              float x = 0.0f;
100                             // loop through the inputs of the layer including the
                                    bias (+1)
101                             #pragma unroll
102                             for(unsigned i = 0; i < d_LAYERSIZES[l]+1; i++)
103                             {
104                                     // read weights from texture memory
105                     // first get the pointer to the weights of this layer
106                     unsigned iiw = d_POINTER_W[l];
107                     // get the weights at [j,i]
108                     iiw += j*(d_LAYERSIZES[l]+1) + i;
109                     // load weight
110                     float w = tex1D<float>(weights,iiw);
111                                     w = round(d_FP*w)/d_FP;
112                     __syncthreads();
113
114                     // get the input of this layer
115                     float input_data = 0.0f;
116                     // if it is the first layer, get the networks input, otherwise
                            the output of the previous layer
117                     if(l == 0)
118                     {
119                         // get the index of this node from the correct data
120                         unsigned ii = i * BATCH_SIZE + g;
121                         // get data from memory
122                         input_data = (float)input[ii];
123                     }
124                     else
125                     {
126                         // get the index of this node
127                         unsigned ii = d_POINTER_Y[l-1] + i;
128                         // offset with the data of this thread
129                         ii +=  block_thread * d_POINTER_Y[N_LAYERS];
130                         // get data
131                         input_data = y[ii];
132                     }
```

```
133
134                                 // add w*i to the input sum x
135                                 x += w * input_data;
136
137                             }// end of inputloop
138
139                 // get the index of the output node
140                 unsigned jj = d_POINTER_Y[l] + j;
141                 // offset with the correct thread
142                 jj += block_thread * d_POINTER_Y[N_LAYERS];
143
144                             // calculate the output
145                 /*if(d_TRANSFERFUNCTION == 0 || ((d_SIGMOID_OUTPUT == 1) && (l ==
                        N_LAYERS-1)))
146                     y[jj] = 1.0f/(1.0f + __expf(-x));
147                 else
148                                 y[jj] = (x > 0 ? x : x/(float)(d_TRANSFERFUNCTION));
149                 */
150                     float da = 0;//1/(float)(d_TRANSFERFUNCTION);
151
152                 x = 8*x/d_LAYERSIZES[l];
153                 switch(d_TRANSFERSHAPE)
154                 {
155                 case 0: //tanh
156                     y[jj] = round(d_FP*(__expf(x) - __expf(-x))/(__expf(x) + __expf
                            (-x)))/d_FP;
157                     break;
158                 case 1: //sqnl
159                     if(x >= 1)
160                     {
161                         y[jj] = round(d_FP*(x*da + (1-da)))/d_FP;
162                     }
163                     else if(x >= 0)
164                     {
165                         y[jj] = round(d_FP*(x*((da-1)*x+(2-da))))/d_FP;
166                     }
167                     else if(x > -1)
168                     {
169                         y[jj] = round(d_FP*(x*((1-da)*x+(2-da))))/d_FP;
170                     }
171                     else
172                     {
173                         y[jj] = round(d_FP*(da*x + da - 1))/d_FP;
174                     }
175                     break;
176                 case 2: //linear th
177                     if(l == N_LAYERS)
178                     {
179                     if(x >= 1)
180                     {
181                         y[jj] = x*da + (1-da);
182                     }
183                     else if(x >= 0)
184                     {
185                         y[jj] = x*((da-1)*x+(2-da));
186                     }
```

```
187                     else if(x > -1)
188                     {
189                         y[jj] = x*((1-da)*x+(2-da));
190                     }
191                     else
192                     {
193                         y[jj] = da*x + da - 1;
194                     }
195                     }
196                     else
197                     {
198                     if(x >= 0)
199                     {
200                         y[jj] = round(d_FP*x)/d_FP;
201                     }
202                     else
203                     {
204                         y[jj] = round(d_FP*x*da)/d_FP;
205                     }
206                     }
207                     break;
208                 case 3:
209 
210                     if(x >= 0)
211                     {
212                         y[jj] = x;
213                     }
214                     else
215                     {
216                         y[jj] = x*da;
217                     }
218                     break;
219                 default:
220                     y[jj] = 0.0f;
221                     break;
222                 }
223                 __syncthreads();
224             } //end of output loop
225 
226             }//end of layerloop
227 
228             __syncthreads();
229 
230         char maxindex = 0;
231             // calculate errors of the last layer
232             // loop through outputs of the last layer
233             /*#pragma unroll
234             for(unsigned j = 0; j < d_LAYERSIZES[N_LAYERS]; j++)
235             {
236             // index to this output node
237             unsigned jj = d_POINTER_Y[N_LAYERS-1];
238             // get from correct thread data
239             jj += block_thread * d_POINTER_Y[N_LAYERS];
240 
241             // error output = actual output
242             maxindex += (y[jj + maxindex] < y[jj+ j] ? j : maxindex);
```

```
243             }*/
244         unsigned jj = d_POINTER_Y[N_LAYERS-1] + block_thread * d_POINTER_Y[N_LAYERS
                ];
245         maxindex += (y[jj] >= 0 ? 1 : 0);
246         maxindex += (y[jj+1] >= 0 ? 2 : 0);
247         output[g] = maxindex;
248 
249             __syncthreads();
250     }
251 }
252 
253 __global__
254 void run_float(unsigned char *input, unsigned char *output, cudaTextureObject_t
        weights)
255 {
256         // get batch number referred to both the total grid and the sub block
257         unsigned grid_thread = blockIdx.x*blockDim.x + threadIdx.x;
258         unsigned block_thread = threadIdx.x;
259 
260         // allocate shared memory
261     // holds the output (y) and its derivative (d) of each layer for all threads in
            this block
262     // during backpropagation these are also used for calculating the deltas and
            derivatives
263         __shared__ float y[BLOCK_SIZE * SIZE_Y_MAX];
264 
265     // run for all data in this batch using a grid stride loop
266         #pragma unroll
267         for(unsigned g = grid_thread; g < BATCH_SIZE; g+= GRID_SIZE)
268         {
269 
270             // set biasses
271         // the bias is the last input of a layer, it should always be 1
272             #pragma unroll
273             for(unsigned l = 0; l < N_LAYERS; l++)
274             {
275             // for layer n, the index of the bias should be the one before the
                    pointer to the next layer
276             unsigned ii = d_POINTER_Y[l+1] - 1;
277             // select the y that corresponds with this thread (add the size of the y
                     vector, thread times)
278             ii += block_thread * d_POINTER_Y[N_LAYERS];
279             // set the bias to 1
280                     y[ii] = d_LAYERSIZES[l]/d_TRANSFERFUNCTION;
281             }
282 
283         // run network given the input data and weights
284             // run through all layers one by one
285             #pragma unroll
286             for(unsigned l = 0; l < N_LAYERS; l++)
287             {
288                     // loop through the outputs of layer l
289                     #pragma unroll
290                     for(unsigned j = 0; j < d_LAYERSIZES[l+1]; j++)
291                     {
292                             // initialize the input (x) to the transferfunction
```

```
293                         float x = 0.0f;
294                         // loop through the inputs of the layer including the
                                bias (+1)
295                         #pragma unroll
296                         for(unsigned i = 0; i < d_LAYERSIZES[l]+1; i++)
297                         {
298                                 // read weights from texture memory
299                     // first get the pointer to the weights of this layer
300                     unsigned iiw = d_POINTER_W[l];
301                     // get the weights at [j,i]
302                     iiw += j*(d_LAYERSIZES[l]+1) + i;
303                     // load weight
304                                 float w = tex1D<float>(weights,iiw);
305
306                     // get the input of this layer
307                     float input_data = 0.0f;
308                     // if it is the first layer, get the networks input, otherwise
                            the output of the previous layer
309                     if(l == 0)
310                     {
311                         // get the index of this node from the correct data
312                         unsigned ii = i * BATCH_SIZE + g;
313                         // get data from memory
314                         input_data = (float)input[ii];
315                     }
316                     else
317                     {
318                         // get the index of this node
319                         unsigned ii = d_POINTER_Y[l-1] + i;
320                         // offset with the data of this thread
321                         ii +=  block_thread * d_POINTER_Y[N_LAYERS];
322                         // get data
323                         input_data = y[ii];
324                     }
325
326                                 // add w*i to the input sum x
327                                 x += w * input_data;
328
329                         }// end of inputloop
330
331                 // get the index of the output node
332                 unsigned jj = d_POINTER_Y[l] + j;
333                 // offset with the correct thread
334                 jj += block_thread * d_POINTER_Y[N_LAYERS];
335
336                         // calculate the output
337                 /*if(d_TRANSFERFUNCTION == 0 || ((d_SIGMOID_OUTPUT == 1) && (l ==
                        N_LAYERS-1)))
338                     y[jj] = 1.0f/(1.0f + __expf(-x));
339                 else
340                                 y[jj] = (x > 0 ? x : x/(float)(d_TRANSFERFUNCTION));
341                 */
342                     float da = 0;//1/(float)(d_TRANSFERFUNCTION);
343
344                 x = 8*x/d_LAYERSIZES[l];
345                 switch(d_TRANSFERSHAPE)
```

```
346                 {
347                 case 0: //tanh
348                     y[jj] = (__expf(x) - __expf(-x))/(__expf(x) + __expf(-x));
349                     break;
350                 case 1: //sqnl
351                     if(x >= 1)
352                     {
353                         y[jj] = x*da + (1-da);
354                     }
355                     else if(x >= 0)
356                     {
357                         y[jj] = x*((da-1)*x+(2-da));
358                     }
359                     else if(x > -1)
360                     {
361                         y[jj] = x*((1-da)*x+(2-da));
362                     }
363                     else
364                     {
365                         y[jj] = da*x + da - 1;
366                     }
367                     break;
368                 case 2: //linear th
369                     if(l == N_LAYERS)
370                     {
371                     if(x >= 1)
372                     {
373                         y[jj] = x*da + (1-da);
374                     }
375                     else if(x >= 0)
376                     {
377                         y[jj] = x*((da-1)*x+(2-da));
378                     }
379                     else if(x > -1)
380                     {
381                         y[jj] = x*((1-da)*x+(2-da));
382                     }
383                     else
384                     {
385                         y[jj] = da*x + da - 1;
386                     }
387                     }
388                     else
389                     {
390                     if(x >= 0)
391                     {
392                         y[jj] = x;
393                     }
394                     else
395                     {
396                         y[jj] = x*da;
397                     }
398                     }
399                     break;
400                 case 3:
401
```

```
402                     if(x >= 0)
403                     {
404                         y[jj] = x;
405                     }
406                     else
407                     {
408                         y[jj] = x*da;
409                     }
410                     break;
411                 default:
412                     y[jj] = 0.0f;
413                     break;
414                 }
415                 __syncthreads();
416             } //end of output loop
417 
418             }//end of layerloop
419 
420             __syncthreads();
421 
422         char maxindex = 0;
423             // calculate errors of the last layer
424             // loop through outputs of the last layer
425             /*#pragma unroll
426             for(unsigned j = 0; j < d_LAYERSIZES[N_LAYERS]; j++)
427             {
428             // index to this output node
429             unsigned jj = d_POINTER_Y[N_LAYERS-1];
430             // get from correct thread data
431             jj += block_thread * d_POINTER_Y[N_LAYERS];
432 
433             // error output = actual output
434             maxindex += (y[jj + maxindex] < y[jj+ j] ? j : maxindex);
435             }*/
436         unsigned jj = d_POINTER_Y[N_LAYERS-1] + block_thread * d_POINTER_Y[N_LAYERS
                ];
437         maxindex += (y[jj] >= 0 ? 1 : 0);
438         maxindex += (y[jj+1] >= 0 ? 2 : 0);
439         output[g] = maxindex;
440 
441             __syncthreads();
442     }
443 }
444 
445 __global__
446 void train_float(unsigned char *input, unsigned char *target, unsigned char *output,
        float *dWeights, cudaTextureObject_t weights)
447 {
448         // get batch number referred to both the total grid and the sub block
449         unsigned grid_thread = blockIdx.x*blockDim.x + threadIdx.x;
450         unsigned block_thread = threadIdx.x;
451 
452         // allocate shared memory
453     // holds the output (y) and its derivative (d) of each layer for all threads in
            this block
```

```
454     // during backpropagation these are also used for calculating the deltas and
            derivatives
455         __shared__ float y[BLOCK_SIZE * SIZE_Y_MAX], d[BLOCK_SIZE * SIZE_Y_MAX];
456
457     // run for all data in this batch using a grid stride loop
458         #pragma unroll
459         for(unsigned g = grid_thread; g < BATCH_SIZE; g+= GRID_SIZE)
460         {
461
462             // set biasses
463         // the bias is the last input of a layer, it should always be 1
464             #pragma unroll
465             for(unsigned l = 0; l < N_LAYERS; l++)
466             {
467             // for layer n, the index of the bias should be the one before the
                    pointer to the next layer
468             unsigned ii = d_POINTER_Y[l+1] - 1;
469             // select the y that corresponds with this thread (add the size of the y
                     vector, thread times)
470             ii += block_thread * d_POINTER_Y[N_LAYERS];
471             // set the bias to 1
472                     y[ii] = d_LAYERSIZES[l]/d_TRANSFERFUNCTION;
473             }
474
475         // run network given the input data and weights
476             // run through all layers one by one
477             #pragma unroll
478             for(unsigned l = 0; l < N_LAYERS; l++)
479             {
480                     // loop through the outputs of layer l
481                     #pragma unroll
482                     for(unsigned j = 0; j < d_LAYERSIZES[l+1]; j++)
483                     {
484                             // initialize the input (x) to the transferfunction
485                             float x = 0.0f;
486                             // loop through the inputs of the layer including the
                                    bias (+1)
487                             #pragma unroll
488                             for(unsigned i = 0; i < d_LAYERSIZES[l]+1; i++)
489                             {
490                                     // read weights from texture memory
491                     // first get the pointer to the weights of this layer
492                     unsigned iiw = d_POINTER_W[l];
493                     // get the weights at [j,i]
494                     iiw += j*(d_LAYERSIZES[l]+1) + i;
495                     // load weight
496                                     float w = tex1D<float>(weights,iiw);
497
498                     // get the input of this layer
499                     float input_data = 0.0f;
500                     // if it is the first layer, get the networks input, otherwise
                            the output of the previous layer
501                     if(l == 0)
502                     {
503                         // get the index of this node from the correct data
504                         unsigned ii = i * BATCH_SIZE + g;
```

```
505                         // get data from memory
506                         input_data = (float)input[ii];
507                     }
508                     else
509                     {
510                         // get the index of this node
511                         unsigned ii = d_POINTER_Y[l-1] + i;
512                         // offset with the data of this thread
513                         ii +=  block_thread * d_POINTER_Y[N_LAYERS];
514                         // get data
515                         input_data = y[ii];
516                     }
517 
518                                 // add w*i to the input sum x
519                                 x += w * input_data;
520 //                  printf("g: %d\tx[%d,%d,%d] = %f * %f = %f\n",g,l,j,i,w,
    input_data,x);
521                             }// end of inputloop
522 
523                 // get the index of the output node
524                 unsigned jj = d_POINTER_Y[l] + j;
525                 // offset with the correct thread
526                 jj += block_thread * d_POINTER_Y[N_LAYERS];
527 
528                             // calculate the output
529                             /*if(d_TRANSFERFUNCTION == 0 || ((d_SIGMOID_OUTPUT == 1)
                                 && (l == N_LAYERS-1)))
530                 {
531                     y[jj] = 1.0f/(1.0f + __expf(-x));
532                     d[jj] = y[jj] * (1 - y[jj]);
533                 }
534                 else
535                 {
536                                 y[jj] = (x > 0 ? x  : x/(float)(d_TRANSFERFUNCTION))
                                     ;
537                                 d[jj] = (x > 0 ? 1 : 1.0f/(float)(d_TRANSFERFUNCTION
                                     ));
538                 }*/
539                     float da = 0;//1/(float)(d_TRANSFERFUNCTION);
540                 x = 8*x/d_LAYERSIZES[l];
541                 switch(d_TRANSFERSHAPE)
542                 {
543                 case 0: //tanh
544                     y[jj] = (__expf(x) - __expf(-x))/(__expf(x) + __expf(-x));
545                     d[jj] = 1-(y[jj] * y[jj]);
546                     break;
547                 case 1: //sqnl
548                     if(x >= 1)
549                     {
550                         y[jj] = x*da + (1-da);
551                         d[jj] = da;
552                     }
553                     else if(x >= 0)
554                     {
555                         y[jj] = x*((da-1)*x+(2-da));
556                         d[jj] = 2*(da-1)*x+(2-da);
```

```
557                     }
558                     else if(x > -1)
559                     {
560                         y[jj] = x*((1-da)*x+(2-da));
561                         d[jj] = 2*(1-da)*x+(2-da);
562                     }
563                     else
564                     {
565                         y[jj] = da*x + da - 1;
566                         d[jj] = da;
567                     }
568                     break;
569                 case 2: //linear th
570                     if(l == N_LAYERS)
571                     {
572                     if(x >= 1)
573                     {
574                         y[jj] = x*da + (1-da);
575                         d[jj] = da;
576                     }
577                     else if(x >= 0)
578                     {
579                         y[jj] = x*((da-1)*x+(2-da));
580                         d[jj] = 2*(da-1)*x+(2-da);
581                     }
582                     else if(x > -1)
583                     {
584                         y[jj] = x*((1-da)*x+(2-da));
585                         d[jj] = 2*(1-da)*x+(2-da);
586                     }
587                     else
588                     {
589                         y[jj] = da*x + da - 1;
590                         d[jj] = da;
591                     }
592                     }
593                     else
594                     {
595                     if(x >= 0)
596                     {
597                         y[jj] = x;
598                         d[jj] = 1;
599                     }
600                     else
601                     {
602                         y[jj] = x*da;
603                         d[jj] = da;
604                     }
605                     }
606                     break;
607                 case 3:
608                     if(x >= 0)
609                     {
610                         y[jj] = x;
611                         d[jj] = 1;
612                     }
```

```
613                     else
614                     {
615                         y[jj] = x*da;
616                         d[jj] = da;
617                     }
618                     break;
619                 default:
620                     y[jj] = 0.0f;
621                     d[jj] = 0.0f;
622                     break;
623                 }
624                 __syncthreads();
625 //              printf("g: %d l: %d: y[%d] = %f\n",g,l,j,y[jj]);
626                     } //end of output loop
627
628             }//end of layerloop
629
630             __syncthreads();
631
632         unsigned char maxindex = 0;
633         /*
634         for(unsigned j = 0; j < d_LAYERSIZES[N_LAYERS]; j++)
635         {
636             maxindex = (y[jj + maxindex] < y[jj + j] ? j : maxindex);
637 //          printf("g: %d, y[%d] %f\n",g,j,y[jj+j]);
638         }*/
639         unsigned jj = d_POINTER_Y[N_LAYERS-1] + block_thread * d_POINTER_Y[N_LAYERS
                ];
640         maxindex += (y[jj] >= 0 ? 1 : 0);
641         maxindex += (y[jj + 1] >= 0 ? 2 : 0);
642         output[g] = maxindex;
643
644             // calculate errors of the last layer
645             // loop through outputs of the last layer
646             #pragma unroll
647             for(unsigned j = 0; j < d_LAYERSIZES[N_LAYERS]; j++)
648             {
649             // index to this output node
650             unsigned jj = d_POINTER_Y[N_LAYERS-1] + j;
651             // get from correct thread data
652             jj += block_thread * d_POINTER_Y[N_LAYERS];
653
654             float tg = (float)((target[g] & (1<<j))>>j);
655             tg = (tg > 0.5f ? 1.0f : -1.0f);
656
657             y[jj] -= tg;
658             __syncthreads();
659
660                     // delta y = e * d
661                     y[jj] *= d[jj];
662 //          printf("g: %d, delta[%d] = %f\n",g,j,y[jj]);
663             }
664
665             __syncthreads();
666
667             // backpropagation calculate delta errors and delta weights
```

```
668         // loop through all layers back to front
669             #pragma unroll
670             for(unsigned l = N_LAYERS-1; l > 0; l--)
671             {
672                     // loop through all inputs of layer l (including bias)
673                     #pragma unroll
674                     for(unsigned i = 0; i < d_LAYERSIZES[l] + 1; i++)
675                     {
676                 // get index to the input node and offset it to the correct data
677                 unsigned ii = d_POINTER_Y[l-1] + i + block_thread * d_POINTER_Y[
                        N_LAYERS];
678 
679                             // set initial sum to zero
680                             float x = 0.0f;
681 
682                             // loop through all outputs of layer l (exluding bias,
                                since there is no dependency)
683                             for(unsigned j = 0; j < d_LAYERSIZES[l+1]; j++)
684                             {
685                     // get index to the output node and offset it to correct data
686                     unsigned jj = d_POINTER_Y[l] + j + block_thread * d_POINTER_Y[
                            N_LAYERS];
687                     // get index to weight
688                     unsigned ii_w = d_POINTER_W[l] + j * (d_LAYERSIZES[l] + 1) + i;
689                     // get index to delta weights
690                     unsigned ii_dw = ii_w * BATCH_SIZE + g;
691 
692                     // get the delta weight
693                     float dw = y[jj] * y[ii];
694                     // write to memory
695                     dWeights[ii_dw] = dw;
696 
697                                     // read weight
698                                     float w = tex1D<float>(weights,ii_w);
699                                     // add to the sum of the input node error of
                                        previous layer x += w * d * e = (w * d * y)
700                                     x += w * d[ii] * y[jj];
701 //                  printf("g: %d, x[%d,%d,%d] = %f * %f = %f\n",g,l,j,i,w,d[ii]*y[
        jj],x);
702 
703                             } // end of output for
704 
705                             // write error
706                             y[ii] = x;
707 
708                     } // end of input for
709             } // end of layer for
710 
711             __syncthreads();
712 
713             // backpropagation of input layer
714         // only need for delta weights, no further backpropagation
715         unsigned l = 0;
716         // loop through all inputs of layer l (including bias)
717                 #pragma unroll
718                 for(unsigned i = 0; i < d_LAYERSIZES[l] + 1; i++)
```

```
719                 {
720                         // get index to the input node and offset it to the correct
                                data
721                         unsigned ii = i * BATCH_SIZE + g;
722
723                         // loop through all outputs of layer l (exluding bias,
                                since there is no dependency)
724                         for(unsigned j = 0; j < d_LAYERSIZES[l+1]; j++)
725                         {
726                                 // get index to the output node and offset it to
                                        correct data
727                                 unsigned jj = d_POINTER_Y[l] + j + block_thread *
                                        d_POINTER_Y[N_LAYERS];
728                                 // get index to weight
729                                 unsigned ii_w = d_POINTER_W[l] + j * (d_LAYERSIZES[l
                                        ] + 1) + i;
730                                 // get index to delta weights
731                                 unsigned ii_dw = ii_w * BATCH_SIZE + g;
732
733                                 // get the delta weight
734                                 float dw = y[jj] * (float)input[ii];
735                                 // write to memory
736                                 dWeights[ii_dw] = dw;
737
738                         } // end of output for
739                 } // end of input for
740
741             __syncthreads();
742
743     }// end grid stride loop
744 }
745
746
747 __global__
748 void k_weights(float *dWeights, cudaTextureObject_t weights, float *wm, float *wv,
        const unsigned tau)
749 {
750         // get index
751         unsigned ii = blockIdx.x*blockDim.x + threadIdx.x;
752     float w = tex1D<float>(weights,ii);
753
754         float dW = (w + (d_SIGMOID_OUTPUT*w) - round(d_SIGMOID_OUTPUT*w))/4;
755
756         // loop through batches
757         for(unsigned b = 0; b < BATCH_SIZE; b++)
758         {
759                 dW += dWeights[ii + b * d_SIZE_W_ZP];
760         }
761         __syncthreads();
762
763         wm[ii] = BETA1*wm[ii] + (1-BETA1)*dW;
764         wv[ii] = BETA2*wv[ii] + (1-BETA2)*dW*dW;
765
766
767         float wmp = wm[ii]/(1-__powf(BETA1,tau+1));
768         float wvp = wv[ii]/(1-__powf(BETA2,tau+1));
```

```
769
770 //          printf("wm[%d] = %f\twv[%d] = %f\tdw[%d] = %f\twmp = %f\twvp = %f\n",ii,wm
        [ii],ii,wv[ii],ii,dW,wmp,wvp);
771     w = w - (ALPHA * wmp / (__fsqrt_rn(wvp) + 0.0000000000000001f));
772     dWeights[ii] = (w < -1.0f ? 1.0f : w > 1.0f ? 1.0f : w);
773
774         __syncthreads();
775 }
776
777
778 __global__
779 void k_transpose(float *input, float *output)
780 {
781         const unsigned tile_bw = 32; //tile size in batch and weight direction
782         const unsigned tile_c = 33; //tile size in weight direction
783
784         __shared__ float tile[tile_bw * tile_c]; //[B][W] avoid shared bank
                conflicts by using the 33 columns, the last column won't be filled
785
786         //unsigned x = blockIdx.x * tile_bw + threadIdx.x; //starting position
787
788         for(unsigned w = 0; w < tile_bw; w++)//loop through 32 weights
789         {
790                 tile[threadIdx.x * tile_c + w] = input[(blockIdx.y * tile_bw + w) *
                        BATCH_SIZE + blockIdx.x * tile_bw + threadIdx.x]; //get 32
                        consecutive batches for each weight
791         }
792
793         __syncthreads();
794
795         for(unsigned b = 0; b < tile_bw; b++)
796         {
797                 output[(blockIdx.x * tile_bw + b) * d_SIZE_W_ZP + blockIdx.y *
                        tile_bw + threadIdx.x] = tile[b * tile_c + threadIdx.x]; //save
                        32 consecutive weights for each batch
798         }
799
800         __syncthreads();
801 }
802
803 __global__
804 void init_rng(curandStateMRG32k3a *randstates)
805 {
806     unsigned thread = threadIdx.x + blockIdx.x * blockDim.x;
807     curand_init(SEED, thread, 0, &randstates[thread]);
808 }
```

C.6. nn.h

Listing C.6: nn.h

```
1 #pragma once
2
3 #include "sc.h"
4 #include "settings.h"
5 #include "functions.h"
```

```
6  #include "io_functions.h"
7
8  template <typename T> class NN
9  {
10     public:
11
12     // DATA
13     SC *sc;
14     unsigned char *h_outputs, *d_outputs;
15
16     T *h_weights;
17     cudaArray *d_weights_array;
18     cudaTextureObject_t d_weights_texture;
19
20     float *d_wm, *d_wv;
21
22     T *d_dweights, *d_dweights_t;
23
24     // Functions
25     NN()
26     {
27         h_weights = (T*)checkMalloc(malloc(SIZE_W_ZP * sizeof(T)));
28         checkCudaError(cudaMalloc(&d_wm, sizeof(float)*SIZE_W_ZP));
29         checkCudaError(cudaMalloc(&d_wv, sizeof(float)*SIZE_W_ZP));
30         checkCudaError(cudaMalloc(&d_dweights, sizeof(T)*SIZE_DW_ZP));
31         checkCudaError(cudaMalloc(&d_dweights_t, sizeof(T)*SIZE_DW_ZP));
32
33         sc = new SC();
34
35         h_outputs = (unsigned char*)checkMalloc(malloc(sizeof(unsigned char)*
               BATCH_SIZE));
36         checkCudaError(cudaMalloc(&d_outputs, sizeof(unsigned char)*BATCH_SIZE));
37
38         init_weights();
39     }
40
41     ~NN()
42     {
43         checkCudaError(cudaMemcpyFromArray(h_weights, d_weights_array, 0, 0, sizeof(
               float)*SIZE_W_ZP, cudaMemcpyDeviceToHost));
44         cudaDeviceSynchronize();
45         for(unsigned l = 0; l < N_LAYERS; l++)
46             {
47                 for(unsigned j = 0; j < LAYERSIZES[l+1]; j++)
48                 {
49                     for(unsigned i =0; i < LAYERSIZES[l] + 1; i++)
50                 {
51                 float w = h_weights[POINTER_W[l] + j * (LAYERSIZES[l]+1) + i];
52                     printf("w[%d %d %d] = %f, %f\n",l,j,i,w, w - round(
                           SIGMOID_OUTPUT*w)/SIGMOID_OUTPUT );
53                 }
54             }
55             }
56         free(h_outputs);
57         cudaFree(d_outputs);
58         free(h_weights);
```

```
59          cudaFree(d_weights_array);
60          cudaFree(d_wm);
61          cudaFree(d_wv);
62          cudaFree(d_dweights);
63          cudaFree(d_dweights_t);
64          delete sc;
65      }
66
67      void init_weights()
68      {
69          for(unsigned l = 0; l < N_LAYERS; l++)
70              {
71                   for(unsigned j = 0; j < LAYERSIZES[l+1]; j++)
72                   {
73                           for(unsigned i =0; i < LAYERSIZES[l] + 1; i++)
74                      {
75                               float p = 1.0f / ((float)(2.0f * LAYERSIZES[l]));
76                               float q = ((float)rand()) / ((float)LAYERSIZES[l] * (
                                     float)RAND_MAX);
77                               h_weights[POINTER_W[l] + j * (LAYERSIZES[l]+1) + i] =
                                     2*(q - p);
78                           }
79                  }
80              }
81          d_weights_texture = 0;
82          init_weights_to_tex(h_weights,&d_weights_array,&d_weights_texture);
83          checkCudaError(cudaMemset(d_wm,0,SIZE_W_ZP));
84          checkCudaError(cudaMemset(d_wv,0,SIZE_W_ZP));
85          checkCudaError(cudaMemset(d_dweights,0,SIZE_DW_ZP));
86          checkCudaError(cudaMemset(d_dweights_t,0,SIZE_DW_ZP));
87      }
88
89      void run()
90      {
91          float ler;
92          unsigned int ler_count;
93          for(PER = P_TEST_MIN; PER < P_TEST_MAX; PER *= P_TEST_INC)
94          {
95              // initialize
96              ler = 0.0f;
97              ler_count = 0;
98              cudaMemcpyToSymbol(d_PER,&PER,sizeof(float));
99
100             // gather data
101             for(unsigned i_run = 0; i_run < RUN_MAX; i_run++)
102             {
103                 // generate data
104                 sc->run_cycle();
105
106                 // run nn
107                 run_once();
108
109                 // logical error
110                 for(unsigned i_batch = 0; i_batch < BATCH_SIZE; i_batch++)
111                 {
```

```
112                     ler_count += (sc->h_targets[i_batch] == h_outputs[i_batch] ? 0 :
                            1);
113                 }
114             }
115             ler = (float)(ler_count)/(float)(BATCH_SIZE * RUN_MAX);
116             printf("%.12f\t%.12f\n",PER,ler);
117         }
118     }
119
120
121     void run_f()
122     {
123         float ler;
124         unsigned int ler_count;
125         for(PER = P_TEST_MIN; PER < P_TEST_MAX; PER *= P_TEST_INC)
126         {
127             // initialize
128             ler = 0.0f;
129             ler_count = 0;
130             cudaMemcpyToSymbol(d_PER,&PER,sizeof(float));
131
132             // gather data
133             for(unsigned i_run = 0; i_run < RUN_MAX; i_run++)
134             {
135                 // generate data
136                 sc->run_cycle();
137
138                 // run nn
139                 run_once_fixed();
140
141                 // logical error
142                 for(unsigned i_batch = 0; i_batch < BATCH_SIZE; i_batch++)
143                 {
144                     ler_count += (sc->h_targets[i_batch] == h_outputs[i_batch] ? 0 :
                            1);
145                 }
146             }
147             ler = (float)(ler_count)/(float)(BATCH_SIZE * RUN_MAX);
148             printf("%.12f\t%.12f\n",PER,ler);
149         }
150     }
151
152     void run_once()
153     {
154         run_float<<<N_BLOCK_PER_MP*N_MP,BLOCK_SIZE>>>(sc->d_ancillas, d_outputs,
                d_weights_texture);
155         cudaDeviceSynchronize();
156         checkCudaError(cudaGetLastError());
157         checkCudaError(cudaMemcpy(h_outputs,d_outputs,sizeof(unsigned char)*
                BATCH_SIZE,cudaMemcpyDeviceToHost));
158         cudaDeviceSynchronize();
159     }
160
161     void run_once_fixed()
162     {
```

```
163         run_fixed<<<N_BLOCK_PER_MP*N_MP,BLOCK_SIZE>>>(sc->d_ancillas, d_outputs,
                d_weights_texture);
164         cudaDeviceSynchronize();
165         checkCudaError(cudaGetLastError());
166         checkCudaError(cudaMemcpy(h_outputs,d_outputs,sizeof(unsigned char)*
                BATCH_SIZE,cudaMemcpyDeviceToHost));
167         cudaDeviceSynchronize();
168     }
169
170     void train_once(unsigned i_run, unsigned i_train)
171     {
172         train_float<<<N_BLOCK_PER_MP*N_MP,BLOCK_SIZE>>>(sc->d_ancillas, sc->
                d_targets, d_outputs, d_dweights, d_weights_texture);
173         cudaDeviceSynchronize();
174         checkCudaError(cudaGetLastError());
175         checkCudaError(cudaMemcpy(h_outputs,d_outputs,sizeof(unsigned char)*
                BATCH_SIZE,cudaMemcpyDeviceToHost));
176         cudaDeviceSynchronize();
177
178         //update
179         dim3 dimBlock(32,1);
180         dim3 dimGrid((BATCH_SIZE/32 < 1 ? 1 : BATCH_SIZE/32),SIZE_W_ZP/32);
181         k_transpose<<<dimGrid,dimBlock>>>(d_dweights,d_dweights_t);
182         cudaDeviceSynchronize();
183         checkCudaError(cudaGetLastError());
184         k_weights<<<SIZE_W_ZP/32,32>>>(d_dweights_t, d_weights_texture, d_wm, d_wv,
                i_train*RUN_MAX+i_run);
185         cudaDeviceSynchronize();
186         checkCudaError(cudaGetLastError());
187         checkCudaError(cudaMemcpy(h_weights,d_dweights_t, SIZE_W_ZP*sizeof(float),
                cudaMemcpyDeviceToHost));
188         cudaDeviceSynchronize();
189
190         unsigned lng = (DISTANCE*DISTANCE);
191
192         for(int l = 0; l < LAYERSIZES[1]; l+=4)
193         {
194             float sb = 0.0f;
195             for(int i = 0; i < (lng-1)/2; i++)
196             {
197                 float so = 0.0f;
198                 float si = 0.0f;
199                 unsigned ii = i;
200                 unsigned io = i+(lng-1)/2;
201                 for(int j = 0; j < 4; j++)
202                 {
203                     so += h_weights[(l+j)*lng+ii];
204                     si += h_weights[(l+j)*lng+io];
205                     ii = (j % 2 == 0 ? ii + (lng-1)/2 : lng - 2 - ii);
206                     io = (j % 2 == 1 ? io + (lng-1)/2 : lng - 2 - io);
207                 }
208                 ii = i;
209                 io = i+(lng-1)/2;
210                 for(int j = 0; j < 4; j++)
211                 {
212                     h_weights[(l+j)*lng+ii] = so/4;
```

```
213                     h_weights[(l+j)*lng+io] = si/4;
214                     ii = (j % 2 == 0 ? ii + (lng-1)/2 : lng - 2 - ii);
215                     io = (j % 2 == 1 ? io + (lng-1)/2 : lng - 2 - io);
216                 }
217             }
218             for(int k = 0; k < 4; k++)
219             {
220                 sb += h_weights[(l+k+1)*lng-1];
221             }
222             for(int k =0; k < 4; k++)
223             {
224                 h_weights[(l+k+1)*lng-1] = sb/4;
225             }
226         }
227         checkCudaError(cudaMemcpyToArray(d_weights_array, 0, 0, h_weights, SIZE_W_ZP
                *sizeof(float),cudaMemcpyHostToDevice));
228         cudaDeviceSynchronize();
229     }
230
231     void train()
232     {
233         float ler;
234         unsigned int ler_count;
235         PER = P_TRAIN_START;
236         for(unsigned i_train = 0; i_train < TRAIN_MAX;i_train++)
237         {
238             ler = 0.0f;
239             ler_count = 0;
240             checkCudaError(cudaMemcpyToSymbol(d_PER,&PER,sizeof(float)));
241
242             for(unsigned i_run = 0; i_run < RUN_MAX; i_run++)
243             {
244                 sc->run_cycle();
245                 train_once(i_run, i_train);
246
247                 for(unsigned i_batch = 0; i_batch < BATCH_SIZE; i_batch++)
248                     ler_count += (sc->h_targets[i_batch] == h_outputs[i_batch] ? 0.0
                            f : 1.0f);
249             }
250             ler = (float)(ler_count)/(float)(BATCH_SIZE *  RUN_MAX);
251             printf("%d: %.12f\t%.12f\n",i_train,PER,ler);
252
253             float c = (float)DISTANCE / (0.5f + pow(0.5f,DISTANCE));
254             PER = pow(ler,1/(1-c))/pow(PER,c/(1-c));
255             PER = (PER < P_TRAIN_MIN ? P_TRAIN_MIN : PER);
256         }
257     }
258 };
```

C.7. sc.h

Listing C.7: sc.h

```
1 #pragma once
2
3 #include "settings.h"
```

```
4  #include "functions.h"
5  #include "kernels.h"
6
7  class SC
8  {
9      public:
10
11     // DATA
12     unsigned char *h_ancillas, *d_ancillas;
13     unsigned char *h_targets, *d_targets;
14
15     curandStateMRG32k3a *d_randstates;
16
17     // Functions
18     SC()
19     {
20         h_ancillas = (unsigned char*)checkMalloc(malloc(sizeof(unsigned char)*(
               DISTANCE*DISTANCE)*BATCH_SIZE));
21         checkCudaError(cudaMalloc(&d_ancillas,sizeof(unsigned char)*(DISTANCE*
               DISTANCE)*BATCH_SIZE));
22         h_targets = (unsigned char*)checkMalloc(malloc(sizeof(unsigned char)*
               BATCH_SIZE));
23         checkCudaError(cudaMalloc(&d_targets,sizeof(unsigned char)*BATCH_SIZE));
24
25         checkCudaError(cudaMalloc(&d_randstates,sizeof(curandStateMRG32k3a)*
               GRID_SIZE));
26
27         init_randstates();
28
29     }
30
31     ~SC()
32     {
33         free(h_ancillas);
34         free(h_targets);
35
36         cudaFree(d_ancillas);
37         cudaFree(d_targets);
38
39         cudaFree(d_randstates);
40     }
41
42     void init_randstates()
43     {
44         init_rng<<<N_BLOCK_PER_MP*N_MP,BLOCK_SIZE>>>(d_randstates);
45         checkCudaError(cudaGetLastError());
46     }
47
48     void run_cycle()
49     {
50         generate_surface_code_data<<<N_BLOCK_PER_MP*N_MP,BLOCK_SIZE>>>(d_ancillas,
               d_targets,d_randstates);
51         cudaDeviceSynchronize();
52         checkCudaError(cudaGetLastError());
53         checkCudaError(cudaMemcpy(h_targets,d_targets,sizeof(unsigned char)*
               BATCH_SIZE,cudaMemcpyDeviceToHost));
```

```
54              cudaDeviceSynchronize();
55          }
56
57  };
```

D

VHDL Code

D.1. top.vhd

Listing D.1: top.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.package_nn.all;
4
5  entity top is
6      port(
7          clk         : in  std_logic;
8          reset       : in  std_logic;
9
10         id : in  std_logic_vector(n_bits * s_0 - 1 downto 0);
11         od : out std_logic_vector(n_bits * s_3 - 1 downto 0)
12     );
13 end top;
14
15 architecture structural of top is
16
17     component neural_network is
18         port(
19         clk         : in  std_logic;
20         reset       : in  std_logic;
21         input_data  : in  std_logic_vector(n_bits * s_0 - 1 downto 0);
22         output_data : out std_logic_vector(n_bits * s_3 - 1 downto 0)
23         );
24     end component neural_network;
25
26 begin
27
28     l_nn: neural_network port map(
29         clk => clk,
30         reset => reset,
31         input_data => id,
32         output_data => od
33         );
34
35 end structural;
```

D.2. neural_network.vhd

Listing D.2: neural_network.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.package_nn.all;
4
5  entity neural_network is
6  port(
7      clk         : in  std_logic;
8      reset       : in  std_logic;
9      input_data  : in  std_logic_vector(n_bits * s_0 - 1 downto 0);
10     output_data : out std_logic_vector(n_bits * s_3 - 1 downto 0)
11 );
12 end entity neural_network;
13
14 architecture structural of neural_network is
15
16     component neural_layer is
17         generic(
18             constant input_size : integer;
19         constant input_log : integer;
20             constant output_size : integer
21         );
22         port(
23             input_data  : in  std_logic_vector(n_bits * input_size - 1 downto 0);
24             output_data : out std_logic_vector(n_bits * output_size - 1 downto 0)
25         );
26     end component neural_layer;
27
28     signal data_in, new_data_in : std_logic_vector(n_bits * s_0 - 1 downto 0);
29     signal data_1 : std_logic_vector(n_bits * s_1 - 1 downto 0);
30     signal data_2 : std_logic_vector(n_bits * s_2 - 1 downto 0);
31     signal data_out, new_data_out : std_logic_vector(n_bits * s_3 - 1 downto 0);
32
33 begin
34
35     output_data <= data_out;
36     new_data_in <= input_data;
37
38     l_proc: process(clk) begin
39         if(clk'event and clk = '1') then
40             if(reset = '1') then
41                 data_in <= (others => '0');
42                 data_out <= (others => '0');
43             else
44                 data_in <= new_data_in;
45                 data_out <= new_data_out;
46             end if;
47         end if;
48     end process;
49
50     l_1: entity work.neural_layer(structural)
51         generic map(
52                 input_size => s_0,
53         input_log => sl_0,
```

```
54                  output_size => s_1
55          )
56          port map(
57              input_data => data_in,
58              output_data => data_1
59          );
60
61      l_2: entity work.neural_layer(structural)
62          generic map(
63                  input_size => s_1,
64          input_log => sl_1,
65                  output_size => s_2
66          )
67          port map(
68              input_data => data_1,
69              output_data => data_2
70          );
71
72      l_3: entity work.neural_layer(structural)
73          generic map(
74                  input_size => s_2,
75          input_log => sl_2,
76                  output_size => s_3
77          )
78          port map(
79              input_data => data_2,
80              output_data => new_data_out
81          );
82
83  end structural;
```

D.3. neural_layer.vhd

Listing D.3: neural_layer.vhd

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use work.package_nn.all;
4
5   entity neural_layer is
6       generic(
7           constant input_size : integer;
8       constant input_log : integer;
9           constant output_size : integer
10      );
11      port(
12          input_data  : in  std_logic_vector(n_bits * input_size - 1 downto 0);
13          output_data : out std_logic_vector(n_bits * output_size - 1 downto 0)
14      );
15  end neural_layer;
16
17  architecture structural of neural_layer is
18
19      component neural_node is
20          generic(
21              input_size : integer;
```

```
22          input_log : integer
23          );
24          port(
25              input_data  : in  std_logic_vector(n_bits * input_size - 1 downto 0);
26              output_data : out std_logic_vector(n_bits - 1 downto 0);
27              weights     : in  std_logic_vector(n_bits * input_size - 1 downto 0)
28          );
29      end component neural_node;
30
31      component weights_rand is
32          generic(
33              nweights : integer
34          );
35          port(
36              weights_out : out std_logic_vector(nweights-1 downto 0)
37          );
38      end component weights_rand;
39
40      signal weights : std_logic_vector(input_size * output_size * n_bits - 1 downto
            0);
41
42  begin
43
44    l_w:  entity work.weights_rand(Behavioral)
45            generic map(
46                nweights => input_size * output_size * n_bits
47            )
48            port map(
49                weights_out => weights
50            );
51
52
53    l_gen_for : for ii in 0 to output_size-1 generate
54        l_n: entity work.neural_node(structural)
55            generic map(
56                input_size => input_size,
57            input_log => input_log
58            )
59            port map(
60                input_data => input_data,
61                output_data => output_data((n_bits * (ii+1) - 1) downto (n_bits * ii))
                      ,
62                weights => weights(n_bits * (ii+1) * input_size -1 downto n_bits * ii
                      * input_size)
63            );
64    end generate l_gen_for;
65
66  end structural;
```

D.4. neural_node.vhd

Listing D.4: neural_node.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.package_nn.all;
```

```
4
5  entity neural_node is
6      generic(
7          input_size : integer;
8      input_log : integer
9      );
10     port(
11         input_data  : in  std_logic_vector(n_bits * input_size - 1 downto 0);
12         output_data : out std_logic_vector(n_bits - 1 downto 0);
13
14         weights     : in  std_logic_vector(n_bits * input_size -1 downto 0)
15     );
16 end entity neural_node;
17
18 architecture structural of neural_node is
19
20     component dot_product is
21         generic(
22             input_size : integer;
23         input_log : integer
24         );
25         port(
26             input_data  : in  std_logic_vector(n_bits * input_size - 1 downto 0);
27             output_data : out std_logic_vector(2*n_bits + input_log - 1 downto 0);
28             weights     : in  std_logic_vector(n_bits * input_size - 1 downto 0)
29         );
30     end component dot_product;
31
32     component sqnl is
33         generic(
34             input_size  : integer;
35         input_log : integer
36         );
37         port(
38             input_data  : in  std_logic_vector(2*n_bits + input_log - 1 downto 0);
39             output_data : out std_logic_vector(n_bits - 1 downto 0)
40         );
41     end component sqnl;
42
43     signal sum : std_logic_vector(2*n_bits + input_log - 1 downto 0);
44
45 begin
46
47     l_dot: entity work.dot_product(behavioural)
48         generic map(
49                 input_size => input_size,
50         input_log => input_log
51         )
52         port map(
53             input_data => input_data,
54             output_data => sum,
55             weights => weights
56         );
57
58     l_sqnl: entity work.sqnl(behavioural)
59         generic map(
```

```
60                  input_size => input_size,
61          input_log => input_log
62          )
63          port map(
64              input_data => sum,
65              output_data => output_data
66          );
67  end structural;
```

D.5. weights_rand.vhd

Listing D.5: weights_rand.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.package_nn.all;
4
5
6  entity weights_rand is
7      generic(
8          nweights   : integer
9      );
10     port(
11         weights_out : out std_logic_vector(nweights - 1 downto 0)
12     );
13 end weights_rand;
14
15 architecture Behavioral of weights_rand is
16
17     signal weights : std_logic_vector(127999 downto 0);
18
19 begin
20
21     weights(31999 downto 0) <= x"...";
22     weights(63999 downto 32000) <= x"...";
23     weights(95999 downto 64000) <= x"...";
24     weights(127999 downto 96000) <= x"...";
25
26     lblw:
27     weights_out <= weights(nweights-1 downto 0);
28
29 end Behavioral;
```

D.6. package_nn.vhd

Listing D.6: package_nn.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4
5  package package_nn is
6
7
8      constant n_bits : integer := 6;
```

```
9       constant distance : integer := 9;
10      constant n_layers : integer := 3;
11      constant s_0 : integer := 80; --distance*distance -1;
12      constant sl_0 : integer := 7;
13      constant s_1 : integer := 256;
14      constant sl_1 : integer := 8;
15      constant s_2 : integer := 16;
16      constant sl_2 : integer := 4;
17      constant s_3 : integer := 2;
18
19  end package package_nn;
```

D.7. dot_product.vhd

Listing D.7: dot_product.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4   use work.package_nn.all;
5
6   entity dot_product is
7       generic(
8           input_size : integer;
9           input_log : integer
10      );
11      port(
12          input_data  : in  std_logic_vector(n_bits * input_size - 1 downto 0);
13          output_data : out std_logic_vector(2*n_bits + input_log - 1 downto 0);
14          weights     : in  std_logic_vector(n_bits * input_size - 1 downto 0)
15      );
16  end entity dot_product;
17
18  architecture behavioural of dot_product is
19
20      component add_8 is
21          generic(
22              constant bitwidth : integer
23          );
24          port(
25              input_data  : in  std_logic_vector(bitwidth * 8 - 1 downto 0);
26              output_data : out std_logic_vector(bitwidth + 2 downto 0)
27          );
28      end component add_8;
29
30      signal products : std_logic_vector(n_bits * 2 * input_size - 1 downto 0);
31
32  begin
33
34      l_gen_for : for ii in 0 to input_size-1 generate
35          products((ii+1)*n_bits*2-1 downto ii*n_bits*2) <= std_logic_vector(signed(
                input_data((ii+1)*n_bits-1 downto ii*n_bits)) * signed(weights((ii+1)*
                n_bits-1 downto ii*n_bits)));
36      end generate l_gen_for;
37
38      g_add_2 : if input_size = 2 generate
```

```
39              l_add: entity work.add_2(behavioural)
40                      generic map(
41                              bitwidth => 2*n_bits
42                      )
43                      port map(
44                          input_data_1 => products(n_bits*4-1 downto n_bits * 2),
45                          input_data_2 => products(n_bits*2-1 downto 0),
46                          output_data => output_data
47                      );
48      end generate g_add_2;
49
50      g_add_4: if input_size = 4 generate
51              l_add: entity work.add_4(structural)
52                      generic map(
53                              bitwidth => 2*n_bits
54                      )
55                      port map(
56                          input_data => products,
57                          output_data => output_data
58                      );
59      end generate g_add_4;
60
61      g_add_8: if input_size = 8 generate
62              l_add: entity work.add_8(structural)
63                      generic map(
64                              bitwidth => 2*n_bits
65                      )
66                      port map(
67                          input_data => products,
68                          output_data => output_data
69                      );
70      end generate g_add_8;
71
72      g_add_16: if input_size = 16 generate
73              l_add: entity work.add_16(structural)
74                      generic map(
75                              bitwidth => 2*n_bits
76                      )
77                      port map(
78                          input_data => products,
79                          output_data => output_data
80                      );
81      end generate g_add_16;
82
83      g_add_24: if input_size = 24 generate
84          l_add: entity work.add_24(structural)
85                      generic map(
86                              bitwidth => 2*n_bits
87                      )
88                      port map(
89                          input_data => products,
90                          output_data => output_data
91                      );
92          end generate g_add_24;
93
94      g_add_32: if input_size = 32 generate
```

```
95              l_add: entity work.add_32(structural)
96                      generic map(
97                              bitwidth => 2*n_bits
98                      )
99                      port map(
100                         input_data => products,
101                         output_data => output_data
102                     );
103     end generate g_add_32;
104
105      g_add_48: if input_size = 48 generate
106         l_add: entity work.add_48(structural)
107                     generic map(
108                             bitwidth => 2*n_bits
109                     )
110                     port map(
111                         input_data => products,
112                         output_data => output_data
113                     );
114         end generate g_add_48;
115
116     g_add_64: if input_size = 64 generate
117             l_add: entity work.add_64(structural)
118                     generic map(
119                             bitwidth => 2*n_bits
120                     )
121                     port map(
122                         input_data => products,
123                         output_data => output_data
124                     );
125     end generate g_add_64;
126
127     g_add_80: if input_size = 80 generate
128             l_add: entity work.add_80(structural)
129                     generic map(
130                             bitwidth => 2*n_bits
131                     )
132                     port map(
133                         input_data => products,
134                         output_data => output_data
135                     );
136     end generate g_add_80;
137
138     g_add_128: if input_size = 128 generate
139             l_add: entity work.add_128(structural)
140                     generic map(
141                             bitwidth => 2*n_bits
142                     )
143                     port map(
144                         input_data => products,
145                         output_data => output_data
146                     );
147     end generate g_add_128;
148
149     g_add_256: if input_size = 256 generate
150             l_add: entity work.add_256(structural)
```

```
151                     generic map(
152                             bitwidth => 2*n_bits
153                     )
154                     port map(
155                         input_data => products,
156                         output_data => output_data
157                     );
158     end generate g_add_256;
159
160 end behavioural;
```

D.8. sqnl.vhd

Listing D.8: sqnl.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.package_nn.all;
5
6  entity sqnl is
7      generic(
8          input_size  : integer;
9      input_log : integer
10     );
11     port(
12         input_data  : in  std_logic_vector(2*n_bits + input_log - 1 downto 0);
13         output_data : out std_logic_vector(n_bits - 1 downto 0)
14     );
15 end entity sqnl;
16
17 architecture behavioural of sqnl is
18
19     signal sign : std_logic;
20
21     signal frac: std_logic_vector(2*(n_bits-1) downto 0);
22
23     signal square : signed(1 + 4 * (n_bits-1) downto 0);
24
25     signal product : signed(2*(n_bits - 1) + 1 downto 0);
26
27 begin
28
29     lbl1: process(input_data, sign, frac, square, product)
30     begin
31
32         sign <= input_data(2*n_bits + input_log - 1);
33         frac <= sign & input_data(2*(n_bits-1)-1 downto 0);
34         square <= signed(frac) * signed(frac);
35
36         if(sign = '0') then --positive
37           product <= -square(1+4*(n_bits-1) downto 1+2*(n_bits-1)) + signed(sign &
               frac);
38         else
39           product <= square(1+4*(n_bits-1) downto 1+2*(n_bits-1)) + signed(sign &
               frac);
```

```
40          end if;
41
42          if(signed(input_data) >= 2**(2*(n_bits-1)) or product >= 2**(2*(n_bits-1)-1)
                ) then
43              output_data(n_bits - 1) <= '0';
44              output_data(n_bits - 2 downto 0) <= (others => '1');
45          elsif(signed(input_data) <= -2**(2*(n_bits-1))) then
46              output_data(n_bits - 1) <= '1';
47              output_data(n_bits - 2 downto 0) <= (others => '0');
48          else
49              output_data <= std_logic_vector(product(2*n_bits -3 downto  n_bits-2));
50          end if;
51
52      end process;
53     --output_data <= input_data(n_bits-1 downto 0);
54
55  end behavioural;
```

D.9. add2.vhd

Listing D.9: add2.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity add_2 is
6       generic(
7           constant bitwidth : integer
8       );
9       port(
10          input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
11          input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
12          output_data     : out std_logic_vector(bitwidth downto 0)
13      );
14  end entity add_2;
15
16  architecture behavioural of add_2 is
17
18  begin
19
20      output_data <= std_logic_vector(signed(input_data_1(bitwidth - 1) & input_data_1
            ) + signed(input_data_2(bitwidth - 1) & input_data_2));
21
22  end behavioural;
```

D.10. add4.vhd

Listing D.10: add4.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity add_4 is
```

```
6       generic(
7           constant bitwidth : integer
8       );
9       port(
10          input_data  : in  std_logic_vector(bitwidth * 4 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 1 downto 0)
12      );
13  end entity add_4;
14
15  architecture structural of add_4 is
16
17      component add_2 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
23              input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
24              output_data     : out std_logic_vector(bitwidth downto 0)
25          );
26      end component add_2;
27
28      signal data_1, data_2 : std_logic_vector(bitwidth downto 0);
29
30
31  begin
32
33      l_1 : entity work.add_2(behavioural)
34          generic map(
35                  bitwidth => bitwidth
36          )
37          port map(
38              input_data_1 => input_data(bitwidth * 4 -1 downto bitwidth * 3),
39              input_data_2 => input_data(bitwidth * 3 -1 downto bitwidth * 2),
40              output_data => data_1
41          );
42
43      l_2 : entity work.add_2(behavioural)
44          generic map(
45                  bitwidth => bitwidth
46          )
47          port map(
48              input_data_1 => input_data(bitwidth * 2 -1 downto bitwidth * 1),
49              input_data_2 => input_data(bitwidth * 1 -1 downto bitwidth * 0),
50              output_data => data_2
51          );
52
53      l_out: entity work.add_2(behavioural)
54          generic map(
55                  bitwidth => bitwidth + 1
56          )
57          port map(
58              input_data_1 => data_1,
59              input_data_2 => data_2,
60              output_data => output_data
61          );
```

```
62
63 end structural;
```

D.11. add8.vhd

Listing D.11: add8.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity add_8 is
6      generic(
7          constant bitwidth : integer
8      );
9      port(
10         input_data  : in  std_logic_vector(bitwidth * 8 - 1 downto 0);
11         output_data : out std_logic_vector(bitwidth + 2 downto 0)
12     );
13 end entity add_8;
14
15 architecture structural of add_8 is
16
17     component add_4 is
18         generic(
19             constant bitwidth : integer
20         );
21         port(
22             input_data  : in  std_logic_vector(bitwidth * 4 - 1 downto 0);
23             output_data : out std_logic_vector(bitwidth + 1 downto 0)
24         );
25     end component add_4;
26
27     component add_2 is
28         generic(
29             constant bitwidth : integer
30         );
31         port(
32             input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
33             input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
34             output_data     : out std_logic_vector(bitwidth downto 0)
35         );
36     end component add_2;
37
38     signal data_1, data_2 : std_logic_vector(bitwidth + 1 downto 0);
39
40
41 begin
42
43     l_1 : entity work.add_4(structural)
44         generic map(
45                 bitwidth => bitwidth
46         )
47         port map(
48             input_data => input_data(bitwidth * 8 -1 downto bitwidth * 4),
49             output_data => data_1
```

```
50          );
51
52      l_2 : entity work.add_4(structural)
53          generic map(
54                  bitwidth => bitwidth
55          )
56          port map(
57              input_data => input_data(bitwidth * 4 -1 downto 0),
58              output_data => data_2
59          );
60
61      l_out: entity work.add_2(behavioural)
62          generic map(
63                  bitwidth => bitwidth + 2
64          )
65          port map(
66              input_data_1 => data_1,
67              input_data_2 => data_2,
68              output_data => output_data
69          );
70
71  end structural;
```

D.12. add16.vhd

Listing D.12: add16.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity add_16 is
6       generic(
7           constant bitwidth : integer
8       );
9       port(
10          input_data  : in  std_logic_vector(bitwidth * 16 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 3 downto 0)
12      );
13  end entity add_16;
14
15  architecture structural of add_16 is
16
17      component add_8 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 8 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 2 downto 0)
24          );
25      end component add_8;
26
27      component add_2 is
28          generic(
29              constant bitwidth : integer
```

```
30          );
31          port(
32              input_data_1     : in  std_logic_vector(bitwidth - 1 downto 0);
33              input_data_2     : in  std_logic_vector(bitwidth - 1 downto 0);
34              output_data      : out std_logic_vector(bitwidth downto 0)
35          );
36      end component add_2;
37
38      signal data_1, data_2 : std_logic_vector(bitwidth + 2 downto 0);
39
40
41  begin
42
43      l_1 : entity work.add_8(structural)
44          generic map(
45                  bitwidth => bitwidth
46          )
47          port map(
48              input_data => input_data(bitwidth * 16 -1 downto bitwidth * 8),
49              output_data => data_1
50          );
51
52      l_2 : entity work.add_8(structural)
53          generic map(
54                  bitwidth => bitwidth
55          )
56          port map(
57              input_data => input_data(bitwidth * 8 -1 downto 0),
58              output_data => data_2
59          );
60
61      l_out: entity work.add_2(behavioural)
62          generic map(
63                  bitwidth => bitwidth + 3
64          )
65          port map(
66              input_data_1 => data_1,
67              input_data_2 => data_2,
68              output_data => output_data
69          );
70
71  end structural;
```

D.13. add24.vhd

Listing D.13: add24.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity add_24 is
6      generic(
7          constant bitwidth : integer
8      );
9      port(
```

```
10          input_data  : in  std_logic_vector(bitwidth * 24 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 4 downto 0)
12      );
13  end entity add_24;
14
15  architecture structural of add_24 is
16
17      component add_8 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 8 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 3 downto 0)
24          );
25      end component add_8;
26
27      component add_2 is
28          generic(
29              constant bitwidth : integer
30          );
31          port(
32              input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
33              input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
34              output_data     : out std_logic_vector(bitwidth downto 0)
35          );
36      end component add_2;
37
38      signal data_1, data_2, data_3 : std_logic_vector(bitwidth + 2 downto 0);
39      signal data_o1, data_o2 : std_logic_vector(bitwidth + 3 downto 0);
40
41
42  begin
43
44      data_o2 <= '0' & data_3;
45
46      l_1 : entity work.add_8(structural)
47          generic map(
48                  bitwidth => bitwidth
49          )
50          port map(
51              input_data => input_data(bitwidth * 24 -1 downto bitwidth * 16),
52              output_data => data_1
53          );
54
55      l_2 : entity work.add_8(structural)
56          generic map(
57                  bitwidth => bitwidth
58          )
59          port map(
60              input_data => input_data(bitwidth * 16 -1 downto bitwidth * 8),
61              output_data => data_2
62          );
63
64      l_3 : entity work.add_8(structural)
65          generic map(
```

```
66                  bitwidth => bitwidth
67          )
68          port map(
69              input_data => input_data(bitwidth * 8 -1 downto 0),
70              output_data => data_3
71          );
72
73      l_out1: entity work.add_2(behavioural)
74          generic map(
75                  bitwidth => bitwidth + 3
76          )
77          port map(
78              input_data_1 => data_1,
79              input_data_2 => data_2,
80              output_data => data_o1
81          );
82
83      l_out: entity work.add_2(behavioural)
84          generic map(
85                  bitwidth => bitwidth + 4
86          )
87          port map(
88              input_data_1 => data_o1,
89              input_data_2 => data_o2,
90              output_data => output_data
91          );
92
93  end structural;
```

D.14. add32.vhd

Listing D.14: add32.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity add_32 is
6       generic(
7           constant bitwidth : integer
8       );
9       port(
10          input_data  : in  std_logic_vector(bitwidth * 32 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 4 downto 0)
12      );
13  end entity add_32;
14
15  architecture structural of add_32 is
16
17      component add_16 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 16 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 3 downto 0)
```

```
24          );
25      end component add_16;
26
27      component add_2 is
28          generic(
29              constant bitwidth : integer
30          );
31          port(
32              input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
33              input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
34              output_data     : out std_logic_vector(bitwidth downto 0)
35          );
36      end component add_2;
37
38      signal data_1, data_2 : std_logic_vector(bitwidth + 3 downto 0);
39
40
41  begin
42
43      l_1 : entity work.add_16(structural)
44          generic map(
45                  bitwidth => bitwidth
46          )
47          port map(
48              input_data => input_data(bitwidth * 32 -1 downto bitwidth * 16),
49              output_data => data_1
50          );
51
52      l_2 : entity work.add_16(structural)
53          generic map(
54                  bitwidth => bitwidth
55          )
56          port map(
57              input_data => input_data(bitwidth * 16 -1 downto 0),
58              output_data => data_2
59          );
60
61      l_out: entity work.add_2(behavioural)
62          generic map(
63                  bitwidth => bitwidth + 4
64          )
65          port map(
66              input_data_1 => data_1,
67              input_data_2 => data_2,
68              output_data => output_data
69          );
70
71  end structural;
```

D.15. add48.vhd

Listing D.15: add48.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
```

```
4
5  entity add_48 is
6      generic(
7          constant bitwidth : integer
8      );
9      port(
10         input_data  : in  std_logic_vector(bitwidth * 48 - 1 downto 0);
11         output_data : out std_logic_vector(bitwidth + 5 downto 0)
12     );
13 end entity add_48;
14
15 architecture structural of add_48 is
16
17     component add_32 is
18         generic(
19             constant bitwidth : integer
20         );
21         port(
22             input_data  : in  std_logic_vector(bitwidth * 32 - 1 downto 0);
23             output_data : out std_logic_vector(bitwidth + 4 downto 0)
24         );
25     end component add_32;
26
27     component add_16 is
28         generic(
29             constant bitwidth : integer
30         );
31         port(
32             input_data  : in  std_logic_vector(bitwidth * 16 - 1 downto 0);
33             output_data : out std_logic_vector(bitwidth + 3 downto 0)
34         );
35     end component add_16;
36
37     component add_2 is
38         generic(
39             constant bitwidth : integer
40         );
41         port(
42             input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
43             input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
44             output_data     : out std_logic_vector(bitwidth downto 0)
45         );
46     end component add_2;
47
48     signal data_1, data_2 : std_logic_vector(bitwidth + 4 downto 0);
49     signal data_16 : std_logic_vector(bitwidth + 3 downto 0);
50
51
52 begin
53
54     data_2 <= '0' & data_16;
55
56     l_1 : entity work.add_32(structural)
57         generic map(
58                 bitwidth => bitwidth
59         )
```

```
60          port map(
61              input_data => input_data(bitwidth * 48 -1 downto bitwidth * 16),
62              output_data => data_1
63          );
64
65      l_2 : entity work.add_16(structural)
66          generic map(
67                  bitwidth => bitwidth
68          )
69          port map(
70              input_data => input_data(bitwidth * 16 -1 downto 0),
71              output_data => data_16
72          );
73
74      l_out: entity work.add_2(behavioural)
75          generic map(
76                  bitwidth => bitwidth + 5
77          )
78          port map(
79              input_data_1 => data_1,
80              input_data_2 => data_2,
81              output_data => output_data
82          );
83
84  end structural;
```

D.16. add64.vhd

Listing D.16: add64.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity add_64 is
6       generic(
7           constant bitwidth : integer
8       );
9       port(
10          input_data  : in  std_logic_vector(bitwidth * 64 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 5 downto 0)
12      );
13  end entity add_64;
14
15  architecture structural of add_64 is
16
17      component add_32 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 32 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 4 downto 0)
24          );
25      end component add_32;
26
```

```
27      component add_2 is
28          generic(
29              constant bitwidth : integer
30          );
31          port(
32              input_data_1     : in  std_logic_vector(bitwidth - 1 downto 0);
33              input_data_2     : in  std_logic_vector(bitwidth - 1 downto 0);
34              output_data      : out std_logic_vector(bitwidth downto 0)
35          );
36      end component add_2;
37
38      signal data_1, data_2 : std_logic_vector(bitwidth + 4 downto 0);
39
40
41  begin
42
43      l_1 : entity work.add_32(structural)
44          generic map(
45                  bitwidth => bitwidth
46          )
47          port map(
48              input_data => input_data(bitwidth * 64 -1 downto bitwidth * 32),
49              output_data => data_1
50          );
51
52      l_2 : entity work.add_32(structural)
53          generic map(
54                  bitwidth => bitwidth
55          )
56          port map(
57              input_data => input_data(bitwidth * 32 -1 downto 0),
58              output_data => data_2
59          );
60
61      l_out: entity work.add_2(behavioural)
62          generic map(
63                  bitwidth => bitwidth + 5
64          )
65          port map(
66              input_data_1 => data_1,
67              input_data_2 => data_2,
68              output_data => output_data
69          );
70
71  end structural;
```

D.17. add80.vhd

Listing D.17: add80.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity add_80 is
6      generic(
```

```
7           constant bitwidth : integer
8       );
9       port(
10          input_data  : in  std_logic_vector(bitwidth * 80 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 6 downto 0)
12      );
13  end entity add_80;
14
15  architecture structural of add_80 is
16
17      component add_64 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 64 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 5 downto 0)
24          );
25      end component add_64;
26
27      component add_16 is
28          generic(
29              constant bitwidth : integer
30          );
31          port(
32              input_data  : in  std_logic_vector(bitwidth * 16 - 1 downto 0);
33              output_data : out std_logic_vector(bitwidth + 3 downto 0)
34          );
35      end component add_16;
36
37      component add_2 is
38          generic(
39              constant bitwidth : integer
40          );
41          port(
42              input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
43              input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
44              output_data     : out std_logic_vector(bitwidth downto 0)
45          );
46      end component add_2;
47
48      signal data_1, data_2 : std_logic_vector(bitwidth + 5 downto 0);
49      signal data_16 : std_logic_vector(bitwidth + 3 downto 0);
50
51
52  begin
53
54      data_2 <= "00" & data_16;
55
56      l_1 : entity work.add_64(structural)
57          generic map(
58                  bitwidth => bitwidth
59          )
60          port map(
61              input_data => input_data(bitwidth * 80 -1 downto bitwidth * 16),
62              output_data => data_1
```

```
63          );
64
65      l_2 : entity work.add_16(structural)
66          generic map(
67                  bitwidth => bitwidth
68          )
69          port map(
70              input_data => input_data(bitwidth * 16 -1 downto 0),
71              output_data => data_16
72          );
73
74      l_out: entity work.add_2(behavioural)
75          generic map(
76                  bitwidth => bitwidth + 6
77          )
78          port map(
79              input_data_1 => data_1,
80              input_data_2 => data_2,
81              output_data => output_data
82          );
83
84  end structural;
```

D.18. add128.vhd

Listing D.18: add128.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity add_128 is
6       generic(
7           constant bitwidth : integer
8       );
9       port(
10          input_data  : in  std_logic_vector(bitwidth * 128 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 6 downto 0)
12      );
13  end entity add_128;
14
15  architecture structural of add_128 is
16
17      component add_64 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 64 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 5 downto 0)
24          );
25      end component add_64;
26
27      component add_2 is
28          generic(
29              constant bitwidth : integer
```

```
30          );
31          port(
32              input_data_1     : in  std_logic_vector(bitwidth - 1 downto 0);
33              input_data_2     : in  std_logic_vector(bitwidth - 1 downto 0);
34              output_data      : out std_logic_vector(bitwidth downto 0)
35          );
36      end component add_2;
37
38      signal data_1, data_2 : std_logic_vector(bitwidth + 5 downto 0);
39
40
41  begin
42
43      l_1 : entity work.add_64(structural)
44          generic map(
45                  bitwidth => bitwidth
46          )
47          port map(
48              input_data => input_data(bitwidth * 128 -1 downto bitwidth * 64),
49              output_data => data_1
50          );
51
52      l_2 : entity work.add_64(structural)
53          generic map(
54                  bitwidth => bitwidth
55          )
56          port map(
57              input_data => input_data(bitwidth * 64 -1 downto 0),
58              output_data => data_2
59          );
60
61      l_out: entity work.add_2(behavioural)
62          generic map(
63                  bitwidth => bitwidth + 6
64          )
65          port map(
66              input_data_1 => data_1,
67              input_data_2 => data_2,
68              output_data => output_data
69          );
70
71  end structural;
```

D.19. add256.vhd

Listing D.19: add256.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity add_256 is
6      generic(
7          constant bitwidth : integer
8      );
9      port(
```

```
10          input_data  : in  std_logic_vector(bitwidth * 256 - 1 downto 0);
11          output_data : out std_logic_vector(bitwidth + 7 downto 0)
12      );
13  end entity add_256;
14
15  architecture structural of add_256 is
16
17      component add_128 is
18          generic(
19              constant bitwidth : integer
20          );
21          port(
22              input_data  : in  std_logic_vector(bitwidth * 128 - 1 downto 0);
23              output_data : out std_logic_vector(bitwidth + 6 downto 0)
24          );
25      end component add_128;
26
27      component add_2 is
28          generic(
29              constant bitwidth : integer
30          );
31          port(
32              input_data_1    : in  std_logic_vector(bitwidth - 1 downto 0);
33              input_data_2    : in  std_logic_vector(bitwidth - 1 downto 0);
34              output_data     : out std_logic_vector(bitwidth downto 0)
35          );
36      end component add_2;
37
38      signal data_1, data_2 : std_logic_vector(bitwidth + 6 downto 0);
39
40
41  begin
42
43      l_1 : entity work.add_128(structural)
44          generic map(
45                  bitwidth => bitwidth
46          )
47          port map(
48              input_data => input_data(bitwidth * 256 -1 downto bitwidth * 128),
49              output_data => data_1
50          );
51
52      l_2 : entity work.add_128(structural)
53          generic map(
54                  bitwidth => bitwidth
55          )
56          port map(
57              input_data => input_data(bitwidth * 128 -1 downto 0),
58              output_data => data_2
59          );
60
61      l_out: entity work.add_2(behavioural)
62          generic map(
63                  bitwidth => bitwidth + 7
64          )
65          port map(
```

```
66              input_data_1 => data_1,
67              input_data_2 => data_2,
68              output_data => output_data
69          );
70
71  end structural;
```