Training Quantum Neural Networks with Graph-Structured Quantum Data

Masterthesis
by

Christian Struckmann

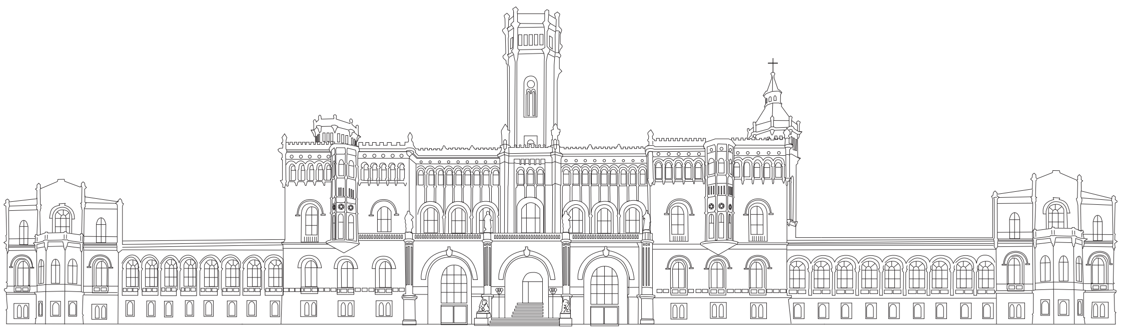
written at
Institut für Theoretische Physik

supervised by
Prof. Dr. Tobias J. Osborne

September 15, 2021

Training Quantum Neural Networks with Graph-Structured Quantum Data

Masterarbeit



vorgelegt der Fakultät für Mathematik und Physik
der Leibniz Universität Hannover

Referent: Prof. Dr. Tobias J. Osborne
Koreferent: Prof. Dr. Reinhard F. Werner

15. September 2021

Abstract

Quantum computation provides a tool for efficiently simulating quantum systems and thus offers classically unattainable computational possibilities. Machine learning models and, especially artificial neural networks, can learn and generalise provided information in a way not possible with generic algorithms. Combining these two groundbreaking techniques leads to quantum machine learning, whose quantum neural networks (QNNs) executed on quantum computers allow the efficient training and generalisation of quantum data unachievable by classical algorithms. The capability to generalise the provided quantum data is of particular interest as quantum data, in general, is limited. Defining the quantum perceptron (the building block of quantum neural networks) as a completely positive map leads to a *dissipative* QNN ($\mathrm{DQNN_U}$). The advent of noisy intermediate-scale quantum (NISQ) devices offers crucial opportunities to develop quantum algorithms. This quantum perceptron is defined in terms of parameterised gates to match the constraints of NISQ devices, leading to the $\mathrm{DQNN_{CAN}}$. Another important quantum algorithm in the field of quantum machine learning is the *quantum approximate optimisation algorithm* (QAOA) which features a sequence of alternating parameterised unitary operators. Here, the two different QNN architectures of the $\mathrm{DQNN_{CAN}}$ and the QAOA are compared to learn an unknown unitary operator. The analysis mainly focuses on their generalisation capabilities, especially under current NISQ device noise levels. It turns out that both networks succeed in learning and generalising the unitary transformation despite the noise. However, the $\mathrm{DQNN_{CAN}}$ is found to be less susceptible to gate noise and is thus outperforming the QAOA in the region of high noise levels. In addition to learning input-output relations of the training states, the training using graph-structured quantum data is analysed. The training states considered here can be associated with the vertices of a graph where the edges represent correlations between the vertices. By exploiting this graph information, the generalisation of the respective network can be significantly improved. The $\mathrm{DQNN_U}$, the $\mathrm{DQNN_{CAN}}$, and the QAOA are trained with and without the graph information to certify this claim. In fact, the exploration of the graph information notably improves every network's generalisation for a specific configuration of supervised states. This analysis is repeated for arbitrary supervised states to prove the generality of this success, and, indeed, there is an improvement in most cases. With the help of these results, the graph learning method can be further improved, resulting in a remarkable enhancement of the network's overall generalisation.

Contents

Chapter 1

Introduction

Since the 1980s, it has been known that simulating quantum mechanics is one of the most challenging problems in computational physics [1, 2]. As the computational complexity scales exponentially with the system size, a quantum many-body problem cannot be simulated efficiently using a classical computer. It was Feynman who first understood that the efficient simulation of quantum mechanics is only possible by exploiting quantum systems themselves [2]. Only a few years afterwards, Deutsch proposed the first universal fully quantum model for computation [3]. He then proceeded to show that a quantum computer could efficiently solve computational problems which have no efficient solution on a classical computer. The Deutsch-Jozsa algorithm is the first known quantum algorithm designed to outperform any classical algorithm [4]. Since then, various other efficient quantum algorithms have been developed, e.g., Shor's algorithm for integer factorisation [5], or Grover's algorithm for unstructured search problems [6].

In parallel to the rapid development of quantum algorithms, quantum machine learning (QML) gained much interest. Classically, machine learning models feature artificial neural networks composed of layers of single parameterised artificial neurons, called perceptrons [7]. The perceptron's parameters commonly are trained by exploiting gradient-based optimisation methods such as gradient descent [8]. Their ability to learn and generalise information have been unattainable for generic algorithms. The field of QML explores quantum algorithms that could speed up certain machine learning problems [9]. A famous QML architecture is the quantum neural network [10–19] which can be defined analogously to its classical counterpart as the composition of quantum perceptrons [10, 13, 20–34] and trained using quantum backpropagation [10, 35–37]. QML machine learning models are especially useful when combined with quantum computation to compute the layer-to-layer transition maps efficiently. Such models are implemented as parameterised quantum circuits and trained using classical optimisation methods [38–41]. They are commonly referred to as hybrid quantum-classical algorithms or variational quantum algorithms [41–44]. This work focuses on the QNN model of [10], who define the quantum perceptron to be a completely positive map. This definition represents a dissipative QNN as the quantum perceptron acts on two different layers of qubits. Here, it is called $\mathrm{DQNN}_{\mathrm{U}}$.

The current state of research in the field of quantum computation is referred to as the noisy intermediate-scale quantum (NISQ) era [45]. The early stage only allows quantum circuits which consist of a few qubits and short depth. To train a variational quantum algorithm using quantum computers, the quantum circuit representation of the QNN has to have only a few gates and parameters. The dissipative QNN model of [10] can be reinterpreted using parameterised gates to fulfil these conditions. In the following, this representation is called $\mathrm{DQNN}_{\mathrm{CAN}}$. Another famous variational quantum algorithm that gained significant interest in the past years is the *quantum approximate optimisation algorithm* (QAOA), which features a sequence of alternating unitary operators applied to one fixed set of qubits [46–48]. It has been applied to learning unitaries [49] and is famous for solving combinatorial optimisation problems [50–58]. [48] introduced the term *quantum alternating operator ansatz* and described it as a standalone ansatz.

Successfully executing QNNs and the QAOA on today's NISQ devices remains extremely challenging as the high noise levels hinder the accurate computation of costs and gradients [45, 53, 59, 60]. It is crucial to evaluate and optimise quantum algorithms with respect to this noise.

This thesis explores and compares the generalisation capabilities of the $\mathrm{DQNN}_{\mathrm{CAN}}$ and the QAOA on current quantum hardware. For this, the $\mathrm{DQNN}_{\mathrm{CAN}}$ and the QAOA are trained to learn an unknown unitary operator from a set of training state pairs. Both networks are implemented as parameterised quantum circuits and optimised to mitigate the effect of noise. The quantum circuits are implemented using Qiskit [61] and executed on real and simulated IBM quantum computers [62].

Besides learning the relations from a set of state pairs $\{|\phi^{\mathrm{in},k}\rangle, |\phi^{\mathrm{out},k}\rangle\}$, it is of particular interest to extend this analysis to the interrelations of the states $\{|\phi^{\mathrm{in},k}\rangle\}$ or $\{|\phi^{\mathrm{out},k}\rangle\}$. These interrelations are associated by a graph $G = (V,E)$ with vertices V and edges E. Here the focus is on the work of [63], who could significantly improve the network's generalisation by exploiting the graph information. The challenge remains to train the QNN using graph-structured quantum data by utilising hybrid quantum-classical algorithms.

This thesis explores the training of the $\mathrm{DQNN}_{\mathrm{U}}$, the $\mathrm{DQNN}_{\mathrm{CAN}}$, and the QAOA using graph-structured quantum data. It is studied whether the success of [63] can be employed to improve the QNNs' generalisation capabilities. The networks are trained using two training data examples that feature particularly interesting graph structures.

This thesis is structured as follows. First, in chapter 2, the main principles of quantum computation are explained. It includes the description of the quantum bit (section 2.2), quantum gates, and quantum circuits (section 2.3). Additionally, the Deutsch-Jozsa algorithm is presented, which is the first known quantum algorithm that solves a specific computational problem more efficiently than the best classical algorithm (section 2.4). Chapter 3 features the basics of machine learning. Here, the artificial neuron (section 3.2.1) and the feed-forward neural network are defined (section 3.2). The optimisation of neural networks using backpropagation and gradient descent is described in section 3.3. Chapter 4 introduces quantum machine learning. It includes the presentation of the quantum perceptron from [10] and its composition to quantum neural networks (section 4.2). The quantum algorithm representation of the QNN and its training using classical optimisation methods are described in section 4.3. The generalisation analysis of the $\mathrm{DQNN}_{\mathrm{CAN}}$ and the QAOA under the influence of noise is presented in chapter 5. It features the description of the learning task, the main quantities, and the results of the analysis. Chapter 6 deals with the analysis of training a QNN using graph-structured quantum data. The learning task, the particular graph-structured training data, and the results of exploiting the graph information are included. Concluding remarks are found in chapter 7.

Chapter 2

Quantum Computation

2.1 Introduction

In 1936, Alan Turing theoretically described the realisation of a computing device [64] which was only years later physically realised. With the further development of the transistor [65], the potential of computers has continuously improved since then. This rise was quantified by Gordon Moore in 1965, who stated that the computational power would double for a constant cost every two years. In other words, he predicted that technology advances so fast that every year a chip can fit twice as many transistors as the last year. Moore's Law held true for decades afterwards but is expected to break as manufacturers reach fundamental size difficulties. As transistors shrink, they reach the realm of quantum mechanical effects which interfere with their functionality.

A possible solution is to switch to a computer whose processing units utilise these quantum mechanical effects. These so-called *quantum computers* offer an essential speed advantage over classical computers. This advantage is reinforced considering the simulation of many-body quantum systems. The classical simulation of such systems is fundamentally limited as the required computational resources grow exponentially with the system size. This growth is only linear with a quantum computer. Feynman first understood that quantum systems could only be efficiently simulated using a quantum device [2]. The first quantum algorithm, which solves a computational problem better than any classical algorithm could, was developed by David Deutsch in 1992 [4]. Various other quantum algorithms have been developed afterwards, claiming to accomplish this *quantum supremacy* [5, 6].

This chapter is structured as follows. In section 2.2, the quantum bit and its description are presented. Section 2.3 features the possible manipulations of quantum bits, the so-called quantum gates. Section 2.4 features the presentation of the Deutsch-Jozsa algorithm, which was one of the first quantum algorithms designed to beat any classical algorithm.

This chapter is based on the famous book *Quantum Computation and Quantum Information* by Michael A. Nielsen and Issac L. Cuang [66]. Interested readers are therefore referred to this book.

2.2 Quantum Bits

The binary information digit {0,1} (short: bit) is the fundamental information unit of classical computing [67]. From punched cards in the 18th century to transistors in modern computers, this concept of information storage has been further developed to today's supercomputers. However, when it comes to simulating quantum mechanics, the classical supercomputers quickly reach their limits as the number of required bits scales exponentially with the quantum system's size.

The quantum bit (qubit) is the fundamental unit of quantum information. It is a two-state quantum system $\{|0\rangle, |1\rangle\}$ whose states corresponds to the states 0 and 1 of the classical bit. A general qubit state can be written as the superposition of the computational basis states $|0\rangle = (1,0)^T$ and $|1\rangle = (0,1)^T$ in the Hilbert space $\mathcal{H} = \mathbb{C}^2$:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}, \quad \alpha, \beta \in \mathbb{C}. \tag{2.1}$$

In contrast to a classical computer, the state of a qubit cannot be read simply by observation. During a measurement in the computational basis, the state will collapse into either state $|0\rangle$ or $|1\rangle$ with probability:

$$p_0 = \| \langle 0|\Psi\rangle \|^2 = \|\alpha\|^2 \tag{2.2a}$$

$$p_1 = \| \langle 1|\Psi\rangle \|^2 = \|\beta\|^2 \tag{2.2b}$$

where $\|\alpha\|^2 + \|\beta\|^2 = 1$. Thus, a qubit's state can generally be represented by a unit vector in a two-dimensional complex vector space.

A useful geometric representation of the qubit state (2.1) is given by

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

where $\theta, \phi, \gamma \in \mathbb{R}$. The term $e^{i\gamma}$ can be neglected as it does not make an observable difference:

$$|\Psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle. \tag{2.4}$$

Note that this representation is only sufficient for pure quantum states. See section 2.2.2 for the mixed state representation.

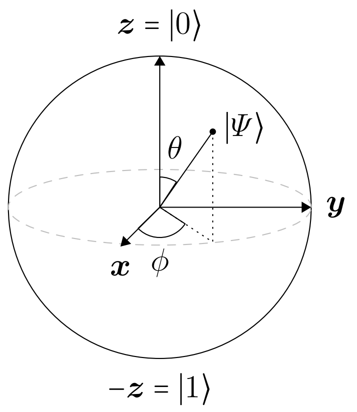


Figure 2.1: The Bloch sphere representation of a qubit.

2.2.1 Multiple qubits

Combining two classical bits results in four possible bit strings: 00, 01, 10, and 11. Analogously, two-qubit states have four computational basis states: $|00\rangle, |01\rangle, |10\rangle, |11\rangle \in \mathbb{C}^4$. Note that $|jk\rangle = |j\rangle \otimes |k\rangle$ for $j,k \in \{0,1\}$. Any two-qubit state can be written as the linear combination of these basis states:

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle, \quad \{\alpha_{jk}\}_{j,k\in\{0,1\}} \in \mathbb{C} \tag{2.5}$$

where $\|\alpha_{jk}\|^2$ is the probability of measuring the state $|jk\rangle$ for $j,k \in \{0,1\}$. Note that the amplitudes have to fulfil the normalisation condition $\sum_{j,k\in\{0,1\}} \|\alpha_{jk}\|^2 = 1$.

In quantum information theory, four very important two-qubit states are the so-called *Bell states* (or *EPR pairs*):

$$|\Phi^\pm\rangle = \frac{|00\rangle \pm |11\rangle}{\sqrt{2}}, \tag{2.6a}$$

$$|\Psi^\pm\rangle = \frac{|01\rangle \pm |10\rangle}{\sqrt{2}}. \tag{2.6b}$$

They are maximally entangled states in the superposition of $|0\rangle$ and $|1\rangle$. Measuring one qubit would also collapse the other qubit, meaning the measurement outcomes are correlated.

The representation of the two-qubit state can be easily generalised to n qubits. A bit string of length n has 2^n different configurations of bits. Similarly, the state of n quantum bits can be represented by a linear combination of the 2^n computational basis states in the Hilbert space $\mathcal{H} = \mathbb{C}^{2^n}$:

$$|\Psi\rangle = \sum_{j_1,j_2,\ldots,j_n\in\{0,1\}} \alpha_{j_1,j_2,\ldots,j_n} |j_1 j_2 \ldots j_n\rangle \in \mathbb{C}^{2^n}, \quad \{\alpha_{j_1,\ldots,j_n}\}_{j_1,\ldots,j_n\in\{0,1\}} \in \mathbb{C}. \tag{2.7}$$

As the Hilbert space grows exponentially with the number of qubits, it is very cost-efficient to simulate such systems classically. However, it also reveals the remarkable computational potential of computers based on such quantum systems.

2.2.2 Mixed qubit states

So far, the qubit's state was assumed to be pure. However, with interactions and decoherence, the qubit can be put into a mixed state. A mixed state is represented by a density operator ρ where $\operatorname{Tr}\rho = 1$ and $\rho \geq 0$ (all eigenvalues are positive).

Consider a system where $|\Psi_i\rangle$, $j = 1, \ldots, n$ have been prepared with probability p_i. Then, the system state can be described by the density operator

$$\rho = \sum_{j=1}^{n} p_j |\Psi_j\rangle \langle\Psi_j| . \tag{2.8}$$

The evolution of the density matrix with unitary operator U is described by

$$\rho = \sum_j p_j |\Psi_j\rangle \langle\Psi_j| \xrightarrow{U} \sum_j p_j U |\Psi_j\rangle \langle\Psi_j| U^\dagger = U\rho U^\dagger . \tag{2.9}$$

Consider the measurement of the qubit's state ρ. The probability of obtaining results 0 and 1, respectively, is

$$p_0 = \sum_{j=1}^{n} p_j \| \langle 0|\Psi_i\rangle \|^2 = \sum_{j=1}^{n} p_j \operatorname{Tr}\left(|0\rangle \langle 0|\Psi_j\rangle \langle\Psi_j|\right) = \operatorname{Tr}\left(|0\rangle \langle 0| \rho\right) = \langle 0| \rho |0\rangle , \tag{2.10a}$$

$$p_1 = \operatorname{Tr}\left(|1\rangle \langle 1| \rho\right) = \langle 1| \rho |1\rangle . \tag{2.10b}$$

Bloch sphere representation

Compared to a pure state, the Bloch sphere representation of a mixed state has an additional degree of freedom. A pure state lies on the surface of the Bloch sphere and can be fully represented by (θ, ϕ). A mixed state, in general, lies inside the Bloch sphere, and thus, its representation also requires the radius r. The Bloch sphere representation for an arbitrary mixed state is given by

$$\rho = \frac{\mathbb{1} + \boldsymbol{r} \cdot \boldsymbol{\sigma}}{2}, \quad \boldsymbol{r} \in \mathbb{R}^3, \quad \boldsymbol{\sigma} = (\sigma_x, \sigma_y, \sigma_z)^T, \tag{2.11}$$

where σ_x, σ_y, and σ_z are the Pauli matrices (see table 2.1) and $\boldsymbol{r}$ is called the Bloch vector with $\|\boldsymbol{r}\| \leq 1$. Note that $\|\boldsymbol{r}\| < 1$ for mixed states and $\|\boldsymbol{r}\| = 1$ for pure states.

Reduced density operator

Consider two systems A and B with Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$. Let $|\Psi\rangle \in \mathcal{H}_A \otimes \mathcal{H}_B$ be the state of the composite system. This can be used to define a density matrix on this system: $\rho = |\Psi\rangle\langle\Psi|$. To obtain the state of system A from the density operator ρ, one has to trace over the other subspace, namely system B:

$$\rho_A = \mathrm{Tr}_B\,\rho = \sum_{j=1}^{n_B} (\mathbb{1}_A \otimes \langle j|_B)\,|\Psi\rangle\langle\Psi|\,(\mathbb{1}_A \otimes |j\rangle_B) \tag{2.12}$$

where $\mathbb{1}_A$ is the identity operator in $\mathcal{H}_A$ and $\{|j\rangle_B\}_{j=1}^{n_B}$ are the basis states of $\mathcal{H}_B$. ρ_A is called the reduced density operator of ρ on subsystem A.

2.3 Quantum Computation

This section features the basic principles of quantum computation. First, in section 2.3.1, the most important quantum gates are presented. Section 2.3.2 shows the composition of quantum gates to create quantum circuits.

2.3.1 Quantum gates

In general, quantum gates are unitary operators acting on one or multiple qubit systems. This unitarity constraint is the only constraint on a quantum gate but a very important one as it conserves the state's normalisation:

$$\mathrm{Tr}\left(U\rho U^\dagger\right) = \mathrm{Tr}\left(U^\dagger U\rho\right) = \mathrm{Tr}\left(\rho\right) = 1 \tag{2.13}$$

where U is a unitary operator ($U^\dagger = U^{-1}$) and ρ is a density matrix in the Hilbert space where the unitary acts.

Single-qubit gates

Single-qubit gates are two-dimensional unitary operators acting on a single qubit state (see table 2.1).

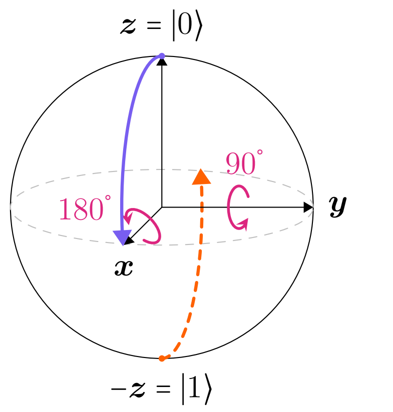
Some quantum gates have a classical counterpart. Consider, e.g., the NOT gate. Classically, it simply flips the bit from 0 to 1 or from 1 to 0. An analogous quantum NOT gate should map $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \xrightarrow{\mathrm{NOT}} |\Psi'\rangle = \mathrm{NOT}\,|\Psi\rangle = \beta|0\rangle + \alpha|1\rangle\,. \tag{2.14}$$

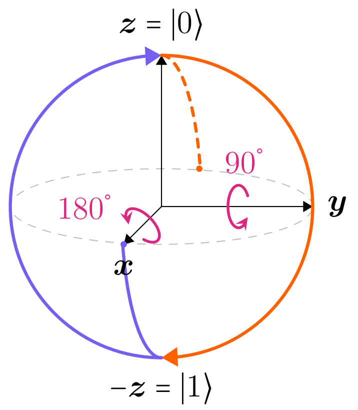
Thus, the NOT gate is defined as

$$\mathrm{NOT} = |0\rangle\langle 1| + |1\rangle\langle 0| = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \sigma_x. \tag{2.15}$$

As the NOT gate is equal to the Pauli-X gate, it can also be viewed as a rotation by π around the x-axis (see Fig. 2.1). The Pauli-Y and Pauli-Z gates rotate by π



(a) $H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ and $H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$.



(b) $H(H|0\rangle) = |0\rangle$ and $H(H|1\rangle) = |1\rangle$.

Figure 2.2: Visual representation of the Hadamard gate acting on (a) $|0\rangle$ and $|1\rangle$ and (b) $H|0\rangle$ and $H|1\rangle$. First, the vector is rotated by 90° around $\boldsymbol{y}$. Afterwards, it is rotated by 180° around $\boldsymbol{x}$. The arrows mark the path of the state as it is rotated.

around the y and z-axis. A continuous rotation by an angle θ around a certain axis $a \in \{x,y,z\}$ can be achieved by applying $e^{-i\theta\sigma_a}$. Such gates are called RX(θ), RY(θ), and RZ(θ).

Another crucial single-qubit gate is the Hadamard gate H which creates a superposition of $|0\rangle$ and $|1\rangle$:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle \tag{2.16a}$$

$$H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle \tag{2.16b}$$

where H itself is defined as

$$H = \frac{1}{\sqrt{2}}\left(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| - |1\rangle\langle 1|\right) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}. \tag{2.17}$$

Like every other quantum gate, the Hadamard gate can be represented by a sequence of rotations (see Fig. 2.2).

Another very important quantum gate is the u gate, as it is the most general of all single-qubit gates. It is parameterised by three angles $\theta, \phi, \lambda \in [0,2\pi)$:

$$u(\theta,\phi,\lambda) = \begin{pmatrix}\cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right)\\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right)\end{pmatrix}. \tag{2.18}$$

Every single-qubit gate can be represented by adjusting these angles.

Multi-qubit gates

Apart from quantum gates acting on single qubits, there exist multi-qubit gates that can create entangled qubit systems. Here, the focus is on two-qubit gates as every other multi-qubit gate can be constructed from single and two-qubit gates. In fact, the only two-qubit gate needed is the controlled-NOT (CNOT) gate. It is defined as

$$\begin{aligned}\mathrm{CNOT} &= |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes \mathrm{NOT} \\ &= |00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 11| + |11\rangle\langle 10| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \end{aligned} \tag{2.19}$$

Applying it to a two-qubit state (2.5) swaps the amplitudes α_{10} and α_{11}:

$$|\Psi\rangle = (2.5) \xrightarrow{\mathrm{CNOT}} |\Psi'\rangle = \mathrm{CNOT}\,|\Psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{11}\,|10\rangle + \alpha_{10}\,|11\rangle\,. \tag{2.20}$$

Suppose, the qubits' states $|\Psi_C\rangle$ and $|\Psi_T\rangle$ are initially either in the state $|0\rangle$ or $|1\rangle$. Swapping the amplitudes α_{10} and α_{11} of the two-qubit state $|\Psi_C\rangle \otimes |\Psi_T\rangle$ only changes the state if $|\Psi_C\rangle = |1\rangle$. Commonly, the two qubits on which the CNOT gate acts are called *control* and *target qubits* as the target qubit's state $|\Psi_T\rangle$ is flipped ($|0\rangle \mapsto |1\rangle$ or $|1\rangle \mapsto |0\rangle$) if and only if the control qubit $|\Psi_C\rangle$ is in the state $|1\rangle$.

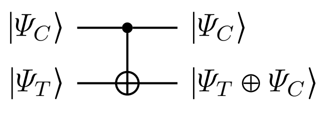


Figure 2.3: The CNOT gate applied to $|\Psi_C\rangle \otimes |\Psi_T\rangle$ where $|\Psi_{C,T}\rangle \in \{|0\rangle, |1\rangle\}$. Here, $\oplus$ denotes the addition modulo 2, which is analogous to applying the classical XOR gate.

The CNOT gate creates entanglement between both qubits. The target qubit's state is conditioned on the control qubit's state and thus, carries the information of both qubits. If the initial states are known, it suffices to measure one of the two qubits to get both of their output states.

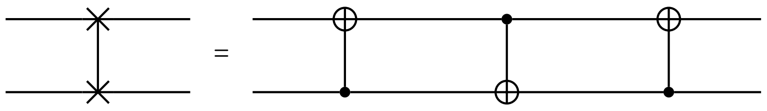
Another crucial two-qubit gate is the SWAP gate which swaps the states of two qubits. It is defined as

$$\mathrm{SWAP} = |00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 01| + |11\rangle\langle 11| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.21}$$

The SWAP gate can be expressed using three CNOT gates (see Fig. 2.4).

Universal set of gates

Classically, it can be shown that the AND, OR, and NOT gates are sufficient to compute any given function as this set of gates is universal for classical computation.



$$\begin{pmatrix} 1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1 \end{pmatrix} = \begin{pmatrix} 1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0 \end{pmatrix} \begin{pmatrix} 1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0 \end{pmatrix} \begin{pmatrix} 1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0 \end{pmatrix}$$

Figure 2.4: The SWAP gate written in terms of CNOT gates.

In quantum computation, there exist several combinations of such universal gates. As mentioned previously, any quantum gate can be composed of single-qubit and CNOT gates. Various possible single-qubit gates come into question here. The most famous is called the *standard set* of universal gates and consists of

$$\mathcal{G}_{\text{standard}} = \{\text{CNOT}, \text{H}, \text{S}, \text{T}\}. \tag{2.22}$$

Any given single-qubit gate can be approximated to an accuracy ϵ using $\mathcal{O}(\log^c(\frac{1}{\epsilon}))$ gates where $c \approx 2$ is a constant (Solovay-Kitaev theorem) [68].

The newer IBM quantum systems mostly use [62]

$$\mathcal{G}_{\text{IBM}} = \{\text{CNOT}, \text{ID}, \text{RZ}, \text{SX}, \text{X}\}. \tag{2.23}$$

Before execution, every gate of the quantum circuit is decomposed into these so-called *basis gates* as these are the actual gates being executed on the IBM quantum systems (see appendix B).

2.3.2 Quantum circuits

Quantum circuits can be illustrated similarly to classical circuit diagrams. A horizontal line represents a distinguishable qubit system that evolves from left to right. Gates onto these lines represent the evolution of the qubit system according to their unitary representation.

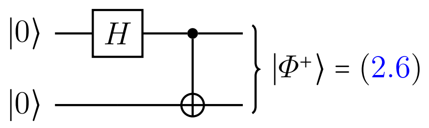


Figure 2.5: The quantum circuit to construct the bell state $|\Phi^+\rangle$.

A simple circuit is depicted in Fig. 2.5. The total quantum circuit consists of two qubits which are initially in the state $|0\rangle$. First, a Hadamard gate is applied to the first qubit, producing the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ (see (2.16a)). Afterwards, a CNOT gate is applied where the first qubit is the control qubit and the second qubit is the target qubit. The target qubit is flipped if and only if the control qubit is in the state $|1\rangle$. However, after the H gate application, the first qubit is in a superposition of states $|0\rangle$ and $|1\rangle$. The output of the circuit is the maximally entangled state:

$$\begin{aligned}\mathrm{CNOT}(\mathrm{H}\otimes\mathbb{1})|00\rangle &\overset{(2.16a)}{=} \mathrm{CNOT}\left(\tfrac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes|0\rangle\right)\\ &\overset{(2.19)}{=} \tfrac{1}{\sqrt{2}}\left(|0\rangle\langle 0|(|0\rangle+|1\rangle)\otimes|0\rangle+|1\rangle\langle 1|(|0\rangle+|1\rangle)\otimes\mathrm{NOT}|0\rangle\right)\\ &\overset{(2.15)}{=} \tfrac{1}{\sqrt{2}}\left(|0\rangle\otimes|0\rangle+|1\rangle\otimes|1\rangle\right)=\tfrac{1}{\sqrt{2}}\left(|00\rangle+|11\rangle\right)\overset{(2.6)}{=}|\Phi^{+}\rangle\,.\end{aligned} \tag{2.24}$$

If the qubits would be initialised in $|01\rangle$, applying the quantum circuit in Fig. 2.5 would result in $|\Psi^{+}\rangle$. Analogously, it maps $|10\rangle$ to $|\Phi^{-}\rangle$ and $|11\rangle$ to $|\Psi^{-}\rangle$.

Measurements in other bases

Measurements are crucial to receive information about the quantum circuit's result. The outcome of a measurement in the computational basis $\{|0\rangle,|1\rangle\}$ is described by (2.10). However, it is also possible to measure the qubit in a different basis even though the measurement operation itself is unchanged. Suppose, the new basis in which the measurement should be performed is $\{|+\rangle=\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)=\mathrm{H}|0\rangle,|-\rangle=\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)=\mathrm{H}|1\rangle\}$. Then, it follows that the probability of obtaining measurement outcome "+" and "−" after measuring an arbitrary state ρ with respect to $|+\rangle$ and $|-\rangle$ is given by

$$\mathrm{Tr}\left(\rho|+\rangle\langle+|\right)=\mathrm{Tr}\left(\rho\mathrm{H}|0\rangle\langle 0|\mathrm{H}^{\dagger}\right)=\mathrm{Tr}\left(\mathrm{H}^{\dagger}\rho\mathrm{H}|0\rangle\langle 0|\right)=\mathrm{Tr}\left(\mathrm{H}\rho\mathrm{H}^{\dagger}|0\rangle\langle 0|\right) \tag{2.25a}$$

$$\mathrm{Tr}\left(\rho|-\rangle\langle-|\right)=\mathrm{Tr}\left(\mathrm{H}\rho\mathrm{H}^{\dagger}|1\rangle\langle 1|\right) \tag{2.25b}$$

where it was used that $\mathrm{H}^{\dagger}=\mathrm{H}$. The measurement in the new basis $\{|+\rangle,|-\rangle\}$ is equivalent to the measurement in the computational basis $\{|0\rangle,|1\rangle\}$ after applying a Hadamard gate to the qubit.

Another useful basis is the Bell basis $\{|\Phi^{\pm}\rangle,|\Psi^{\pm}\rangle\}$ (see (2.6)). Equation (2.24) shows that the Bell states can be written in terms of a CNOT and an H gate. Thus, the measurement outcome of a measurement in the Bell basis is given by

$$\begin{aligned}\mathrm{Tr}\left(\rho|\Phi^{+}\rangle\langle\Phi^{+}|\right)&=\mathrm{Tr}\left(\rho\mathrm{CNOT}(\mathrm{H}\otimes\mathbb{1})|00\rangle\langle 00|(\mathrm{CNOT}(\mathrm{H}\otimes\mathbb{1}))^{\dagger}\right)\\ &=\mathrm{Tr}\left((\mathrm{H}\otimes\mathbb{1})\mathrm{CNOT}\rho\mathrm{CNOT}^{\dagger}(\mathrm{H}\otimes\mathbb{1})^{\dagger}|00\rangle\langle 00|\right)\end{aligned} \tag{2.26}$$

where, it was used that $\mathrm{CNOT}^{\dagger}=\mathrm{CNOT}$ and $\mathrm{H}^{\dagger}=\mathrm{H}$. Analogously for the other Bell states. A measurement in the Bell basis can be achieved by applying the circuit in Fig. 2.6. This measurement method will come in handy when computing the fidelity between two quantum states (see section 4.3.2).

In general, a measurement in the basis $\{\mathcal{U}|0\rangle,\mathcal{U}|1\rangle\}$ is obtained by simply applying $\mathcal{U}^{\dagger}$ before measuring in $\{|0\rangle,|1\rangle\}$.

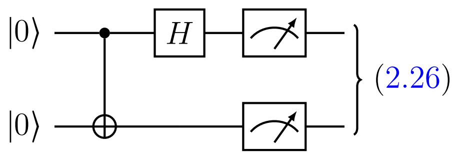


Figure 2.6: The quantum circuit for performing a measurement in the basis of the Bell states. Here, the qubits are initialised in $|00\rangle$ resulting in the measurement outcome (2.26).

2.3.3 Physical realisation of a quantum computer

The physical implementation of a quantum computer is hard as various requirements have to be taken into account. Here, the five main requirements are depicted. The interested reader is referred to the source [66, 69].

A scalable physical system with well-characterised qubits

As discussed previously, a quantum computer is a collection of qubits. A qubit is simply a quantum two-level system (see section 2.2). It can have many representations, e.g., the two spin states of a spin-1/2 particle, the ground and excited states of an atom, or the vertical and horizontal polarisation of a photon. To implement an ideal quantum computer, it is crucial to know the qubit's physical properties like its internal Hamiltonian (i.e., how it evolves over time) or the coupling to other qubits or external fields (to allow the definition of quantum gates). The qubits in today's quantum computers are mainly based on superconducting electrical circuits (superconducting quantum computers) [70, 71] or charged atoms in electromagnetic traps (ion trap quantum computers) [72–74].

Prepare the qubits in a fiducial state

This requirement is self-explanatory, as this is a fundamental requirement of computation itself. The initial state before the start of computation should be known; otherwise, the result of the computation cannot be interpreted. It suffices to produce one state with high fidelity as a sequence of gates can produce any desired input state.

Decoherence times much longer than the gate operation time

Decoherence times quantify the qubit's interactions with its environment. It is the time for a generic qubit state (2.1) to be transformed into the mixture $\rho = \|\alpha\|^2 |0\rangle\langle 0| + \|\beta\|^2 |1\rangle\langle 1|$. It is also referred to as *leakage* in quantum computing [75]. The decoherence time sets the main limitation to the quantum circuit's length, as each gate has its own operation time. The qubits in a quantum computer should be well isolated to have long decoherence times but also have to be accessible for measurements. A good balance has to be found.

A universal set of quantum gates

Of course, to have a universal quantum computer, i.e., a quantum computer that can express any quantum algorithm, the qubits should allow operations that together form a universal set, e.g., (2.22) or (2.23).

Measuring the qubit
To obtain information about the output of the quantum circuits, the qubits have to be measured (see (2.2)). The quality of the measurement can be quantified by the signal to noise ratio. It gives the signal strength with respect to the underlying noise. Commonly, the measurement of a qubit after applying the quantum circuit is repeated several times to average out most of the noise.

The current state of quantum computation is described as the NISQ (Noisy Intermediate-Scale Quantum) era [45]. "Intermediate-Scale" refers to the size of quantum computers or, more precisely, the number of qubits. The qubit number is currently limited to a few hundred, but upcoming quantum computers are already pushing the boundaries of classical simulations [76]. "Noisy" refers to the noise of these qubits. The imperfect conditions and limited control of qubits restrict the number of operations. A quantum circuit with more than a thousand gates is currently not practical as the noise will overwhelm the signal.

2.4 Quantum Algorithms

Since the development of quantum computing, various quantum algorithms have been proposed that, if realised, would outperform any classical algorithm. A very well known quantum algorithm that accomplishes this quantum supremacy is the *Deutsch-Jozsa* algorithm, first introduced in 1992 [4]. The fundamentals for it and the algorithm itself are described in sections 2.4.1 and 2.4.2.

2.4.1 Quantum parallelism

Suppose a function $f(x)$ should be evaluated for $x \in \{0, 1\}$ (see Fig. 2.7). Classi-

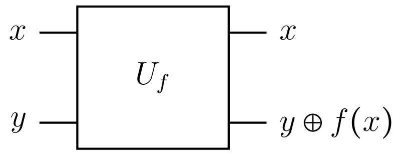
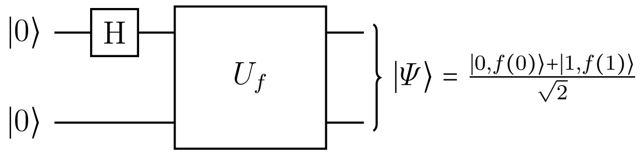


Figure 2.7: Exemplary circuit that computes $f(x)$ for $x \in \{0,1\}$. $\oplus$ denotes addition modulo two.

cally, the circuit of this function has to be evaluated twice, for each case $x = 0$ and $x = 1$. A quantum computer, however, is capable of evaluating both cases simultaneously. This is called *quantum parallelism*. By applying a Hadamard gate, a quantum bit can be put into a superposition of $|0\rangle$ and $|1\rangle$. The application of U_f

onto this superposition is evaluating $f(x)$ for $x = 0$ and $x = 1$ simultaneously (see Fig. 2.8).



$|0\rangle$
H
U_f
$|0\rangle$
$|\Psi\rangle = \frac{|0,f(0)\rangle+|1,f(1)\rangle}{\sqrt{2}}$

Figure 2.8: Exemplary quantum circuit that computes $f(x)$ for $x \in \{0,1\}$.

The parallel application of Hadamard gates to each qubit is called the *Hadamard transform.* The output of a Hadamard transformed two-qubit circuit is given by

$$\mathrm{H}^{\otimes 2}|00\rangle = \mathrm{H}|0\rangle \otimes \mathrm{H}|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{2}. \qquad (2.27)$$

This can be easily generalised to more qubits. The output of the Hadamard transform for n qubits is given by $\frac{1}{\sqrt{2^n}}\sum_x |x\rangle$ where the sum is over all 2^n possible values of $x = x_1 \ldots x_n$ with $x_1, \ldots, x_n \in \{0,1\}$. Thus, the Hadamard transform produces a superposition of all computational basis states.

The Hadamard transform makes it possible to evaluate the given function for all possible input values simultaneously. However, measuring the qubits will only give the result of precisely one x. This computational complexity is also achieved classically. The following section will feature a particular problem where quantum parallelism is exploited so that the quantum algorithm outperforms its classical counterpart.

2.4.2 The Deutsch-Jozsa algorithm

Consider the following problem. Alice produces a bit string $x \in \{0, \ldots, 2^n - 1\}$ and sends it to Bob. Bob chooses to either act a constant $(f(x) = y = \text{const}\,\forall x)$ or a balanced function ($f(x) = 0$ for one half of possible x values and $f(x) = 1$ for the other half) on the given bit string. The return value is passed to Alice, who now should decide whether Bob has chosen a constant or a balanced function. This is called *Deutsch's problem.*

Suppose the evaluation of the function is done classically. Each iteration, Alice sends one of the 2^n possible values of x to Bob. In the worst case, Alice would have to send $2^{n-1} + 1$ bit strings to Bob to make the decision (if she received 2^{n-1} times the 0, the $(2^{n-1} + 1)$th return value would classify the function: 0: constant, 1: balanced).

In the case of quantum bits, Alice only needs to query Bob a single time to decide whether he uses a constant or balanced function. The quantum algorithm making this possible is depicted in Fig. 2.9. It consists of $n + 1$ qubits where the first n qubits are initialised in $|0\rangle$ and the $(n + 1)$th in $|1\rangle$. First, the Hadamard

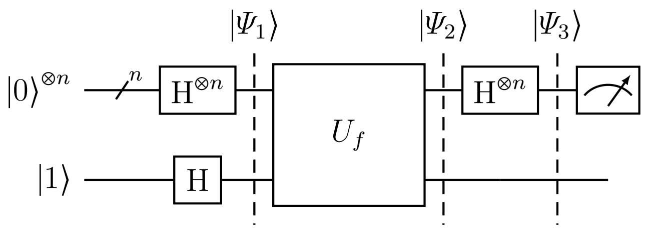


Figure 2.9: The quantum circuit implementing the Deutsch-Jozsa algorithm.

transform is applied, creating a superposition of all computational basis states

$$\begin{aligned}|\Psi_1\rangle = \mathrm{H}^{\otimes n}|0\rangle^{\otimes n} \otimes \mathrm{H}|1\rangle &= \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \otimes \cdots \otimes \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \\ &= \sum_{x\in\{0,1\}^n} \frac{|x\rangle}{\sqrt{2^n}} \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\end{aligned} \tag{2.28}$$

which is then send to Bob. He applies the unitary representation of the function $U_f : |x,y\rangle \mapsto |x, y \oplus f(x)\rangle$ of his choice to all the $n+1$ qubits, creating the state

$$|\Psi_2\rangle = U_f|\Psi_1\rangle = \sum_{x\in\{0,1\}^n} \frac{|x, 0\oplus f(x)\rangle - |x, 1\oplus f(x)\rangle}{\sqrt{2^{n+1}}}. \tag{2.29}$$

where $\oplus$ denotes addition modulo two. For a given x, $f(x)$ can take values 0 and 1. The expression (2.29) can be rewritten using

$$\begin{aligned}|x, 0\oplus f(x)\rangle - |x, 1\oplus f(x)\rangle &= \begin{cases} +|x\rangle \otimes (|0\rangle - |1\rangle), & \text{if } f(x) = 0 \\ -|x\rangle \otimes (|0\rangle - |1\rangle), & \text{if } f(x) = 1 \end{cases} \\ &= (-1)^{f(x)}|x\rangle \otimes (|0\rangle - |1\rangle).\end{aligned} \tag{2.30}$$

Thus, it follows that

$$|\Psi_2\rangle = \sum_{x\in\{0,1\}^n} \frac{(-1)^{f(x)}|x\rangle}{\sqrt{2^n}} \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right). \tag{2.31}$$

The amplitudes of this state carry the information of the function evaluation $f(x)$ and are returned to Alice. Similarly to (2.31), the action of the Hadamard gate on a single qubit can be written as

$$\mathrm{H}|x\rangle = \tfrac{1}{\sqrt{2}} \sum_{z\in\{0,1\}} (-1)^{z\cdot x}|z\rangle. \tag{2.32}$$

For n qubits this can be generalised to

$$\begin{aligned}\mathrm{H}^{\otimes n}|x\rangle = \mathrm{H}^{\otimes n}|x_1 \ldots x_n\rangle &= \tfrac{1}{\sqrt{2^n}} \sum_{z_1,\ldots,z_n\in\{0,1\}} (-1)^{z_1\cdot x_1 + \cdots + z_n\cdot x_n}|z_1 \ldots z_n\rangle \\ &= \tfrac{1}{\sqrt{2^n}} \sum_{z\in\{0,1\}^n} (-1)^{z\cdot x}|z\rangle\end{aligned} \tag{2.33}$$

where $x \cdot z$ is the bitwise product of $x = x_1 \dots x_n$ and $z = z_1 \dots z_n$, modulo two. Thus, the output state of the quantum circuit can be written as

$$\begin{aligned} |\Psi_3\rangle = \left(\mathrm{H}^{\otimes n} \otimes \mathbb{1}\right) |\Psi_2\rangle &= \sum_{x \in \{0,1\}^n} \sum_{z \in \{0,1\}^n} \frac{(-1)^{z \cdot x + f(x)} |z\rangle}{2^n} \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \\ &= |\Psi_3\rangle_{1,\dots,n} \otimes |\Psi_3\rangle_{n+1} . \end{aligned} \tag{2.34}$$

Measuring the first n qubits in the computational basis state $|0\rangle$ results in the probability

$$p_0 = \left\| \langle 0 \dots 0 | \Psi_3 \rangle_{1,\dots,n} \right\|^2 = \left\| \sum_{x \in \{0,1\}^n} \frac{(-1)^{f(x)}}{2^n} \right\|^2 . \tag{2.35}$$

Based on this measurement outcome, Alice can fully classify Bob's function. If Bob chose a constant function, i.e., either $f(x) = 0$ or $f(x) = 1$ for all values of x, the probability would be in both cases $p_0 = 1$ and thus, the measurement outcome is always 0 for all qubits. If Bob chose a balanced function, i.e., $f(x) = 0$ for one half of possible values x and $f(x) = 1$ for the other half, the probability would be $p_0 = 0$ as both halves would cancel each other out. Consequently, the measurement result of at least one qubit is not equal to 0. In summary, the probability of measuring $0 \dots 0$ is

$$p_0 = \begin{cases} 0, & \text{if } f(x) \text{ is constant} \\ 1, & \text{if } f(x) \text{ is balanced} \end{cases} . \tag{2.36}$$

If Alice's measurement results in only 0s for all qubits, Bob's function is constant; otherwise, it is balanced. Thus, Alice can classify Bob's function by only querying him once, whereas many more iterations are required classically.

It has to be noted that the Deutsch-Jozsa algorithm has no practical application but is only formulated to demonstrate the uniqueness of quantum algorithms. Besides the Deutsch-Jozsa algorithm, various other quantum algorithms claim to beat any classical algorithm. Shor's algorithm is one of the most famous quantum algorithms, which is based on the Quantum Fourier Transformation and can factor integers in polynomial time [5]. Grover's algorithm claims to speed up an unstructured search problem quadratically [6].

Table 2.1: Single-qubit gates and their matrix representations.

Operator	Gate	Matrix
Identity	$-\boxed{\mathbb{1}}-$	$\mathbb{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Pauli-X / NOT	$-\boxed{X}-$ $\quad -\oplus-$	$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y	$-\boxed{Y}-$	$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z	$-\boxed{Z}-$	$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Hadamard	$-\boxed{H}-$	$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Z rotation	$-\boxed{RZ(\theta)}-$	$e^{-i\frac{\theta}{2}\sigma_z} = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$
π/8	$-\boxed{T}-$	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$
Sqrt(X)	$-\boxed{SX}-$ $\quad -\boxed{\sqrt{X}}-$	$\sqrt{\sigma_x} = \frac{1}{2} \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$
Phase	$-\boxed{S}-$	$\sqrt{\sigma_z} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$

Chapter 3

Classical Machine Learning

3.1 Introduction

Biological brains process information entirely different from conventional computing systems. The ability to learn and generalise information have been unattainable for generic algorithms. In 1943, the first idea of an artificial brain was published [77, 78]. Since then, the idea has been further developed into an indispensable part of computer science named machine learning.

Artificial neural networks, as the name implies, are computing systems that mimic the processes of their biological counterpart. They are composed of artificial neurons which produce a real-valued activation from multiple binary inputs. The composition and linking of neurons make it possible to compute any given function [79]. This network of neurons can be trained to act as a specific function by utilising special optimisation techniques. Neural networks have been successfully applied to various tasks, e.g., image recognition or natural language processing [80].

There are various methods of composing these neurons in a way that suits the problem at hand. Commonly, one divides these structures into interconnected layers of neurons. The most straightforward way of applying these neurons is simply through the layers from first to last. Such a network is called *feed-forward neural network*. There exist many other neural network architectures that are fundamentally different from this approach. Exemplary, *recurrent neural networks* consist of layers of neurons whose output is inserted again in previous layers, resulting in a recursive operation. Such networks are commonly used for temporal problems such as speech recognition [81] or video analysis [82]. An overview of the most popular neural network architectures can be found in [83].

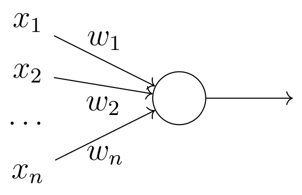
This chapter will focus on feed-forward neural networks as the goal is to explain the basic notions. First, in section 3.2, the neural network with its fundamental building block are presented. Section 3.3 features the optimisation of the neural network. It includes the description of general machine learning optimisation techniques such as gradient descent and backpropagation.

The content of this chapter is based on *Neural networks and deep learning* by Michael A. Nielsen [79]. It explains the main concepts of training artificial neural networks, including its extension to deep learning. The interested reader is referred to this book.

3.2 Neural Networks

3.2.1 Perceptrons

Just like the human brain, a neural network consists of the interconnection between multiple neurons. Every single neuron takes an input signal from adjacent neurons and produces an output signal used as an input for the next neuron. The artificial neuron is called the perceptron and was developed in 1961 by Frank Rosenblatt [7]. Mathematically speaking, a perceptron takes binary inputs $x_1, x_2, \dots, x_n \in \{0,1\}$ and produces a single binary output $a \in \{0,1\}$, called the neuron's *activation*.



$$a = \begin{cases} 0, & \text{if } \sum_{j=1}^{n} w_j x_j \leq \text{threshold} \\ 1, & \text{if } \sum_{j=1}^{n} w_j x_j > \text{threshold} \end{cases}$$

Figure 3.1: The Rosenblatt perceptron.

Rosenblatt defined the output a as the binary value which turns 1 if the weighted sum of all the inputs x_j with weights w_j is greater than a certain threshold and 0 otherwise (see Fig. 3.1). The weights and the threshold are parameters of the perceptron itself. By adjusting the weight w_j, the importance of the specific input x_j can be scaled. The threshold is scaling the overall acceptance rate of the perceptron (how likely it will accept its inputs).

Normally, the expression in Fig. 3.1 is rewritten such that

$$a_{\text{step}}(\boldsymbol{x}; \boldsymbol{w}, b) = \begin{cases} 0, & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \\ 1, & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \end{cases} \tag{3.1}$$

where $\boldsymbol{w} = (w_1, w_2, \dots, w_n)^T$ and $\boldsymbol{x} = (x_1, x_2, \dots, x_n)^T$. Note that the threshold has been exchanged with the *bias* b, which serves the same purpose but $b = -\text{threshold}$. This function $a(\boldsymbol{x}; \boldsymbol{w}, b)$ is called the *activation function* of the neuron.

It turns out that the perceptron is universal for computation as it represents a NAND gate by adjusting its weights and bias. Thus, neural networks can be as powerful as any other computing device. However, there is even more to it.

Improving the perceptron even further makes it possible to invent powerful learning algorithms that automatically update the neuron's parameters such that the network itself acts like the desired function. The most famous improved version of a perceptron is called the sigmoid neuron.

Sigmoid neuron

A crucial constraint on the activation function is the continuity in the weights and biases. Small changes in the weights and biases should only result in small changes in the output. Only then, the change of the network's output is linear in the change of the weights and biases. This linearity makes it easy to choose the small changes in the neuron's parameters that achieve the desired small change in the output (see section 3.3). The activation function of the perceptron (3.1) does not fulfil this requirement. Instead, this gives rise to the definition of the sigmoid neuron. Similar to the perceptron, it takes some inputs $\boldsymbol{x}$ and produces some output $a(\boldsymbol{x};\boldsymbol{w},b)$ according to its weights $\boldsymbol{w}$ and bias b. The key difference is that the sigmoid neuron's inputs and output can take any values between 0 and 1. The output is computed using the sigmoid function:

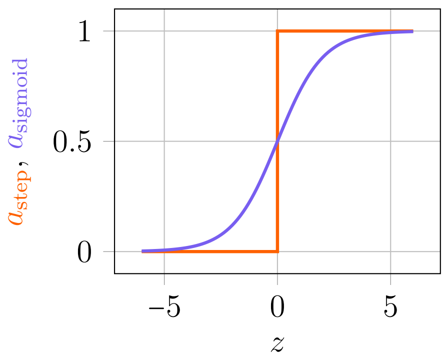


Figure 3.2: The step — and sigmoid — activation functions. $z = \boldsymbol{w}\cdot\boldsymbol{x} + b$.

$$\sigma(z) = \frac{1}{1+\exp(-z)}. \tag{3.2}$$

The activation of the sigmoid neuron is given by:

$$a_{\text{sigmoid}}(\boldsymbol{x};\boldsymbol{w},b) := \sigma(\boldsymbol{w}\cdot\boldsymbol{x}+b) = \frac{1}{1+\exp(-\boldsymbol{w}\cdot\boldsymbol{x}-b)}. \tag{3.3}$$

It is plotted together with the step function (3.1) in Fig. 3.2. The shape of the sigmoid function resembles a smooth step function.

3.2.2 The neural network architecture

The composition of neurons is called a neural network. The output of each neuron is linked to the input of other neurons. In general, these types of structures are divided into layers of neurons (see section 3.2.2). The first layer is called the input layer. The input information of the network is initialised in the neurons of this layer. The last layer is called the output layer, as these neurons store the output data of the network. All the layers in between are called the hidden layers.

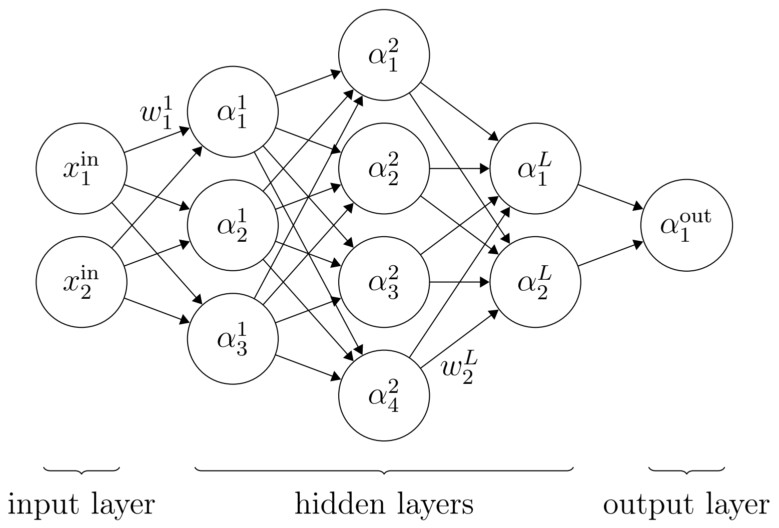


Figure 3.3: The feed-forward neural network architecture. It consists of one input layer, L hidden layers, and one output layer.

The size of the input and output layer generally is defined by the problem at hand (see section 3.3.1). The size of the hidden layer, however, should be set according to the problem complexity. Usually, more complex problems require a higher depth network as the network's computational power scales with the number of neurons.

The neural network depicted in section 3.2.2 is called *feed-forward neural network* as the information advances from left to right through the network. The activation of the jth neuron in layer l is given by:

$$\alpha_j^l = a(\boldsymbol{\alpha}^{l-1}; \boldsymbol{w}_j^l, b_j^l), \quad l = 1, \dots, L, \text{out}, \quad j = 1, \dots, n_l \tag{3.4}$$

where $\boldsymbol{\alpha}^{\text{in}} = \boldsymbol{x}^{\text{in}}$ and $\boldsymbol{\alpha}^l = (\alpha_1^l, \dots, \alpha_{n_l}^l)^T$. n_l denotes the number of neurons in layer l. $\boldsymbol{w}_j^l$ is the weight vector connecting all neurons in layer $l-1$ to the jth neuron of layer l. Thus, the network's output can be written as:

$$\boldsymbol{\alpha}^{\text{out}} = \boldsymbol{\alpha}^{\text{out}}(\boldsymbol{x}^{\text{in}}; \{\boldsymbol{w}\}, \{b\}) = \sum_{j=1}^{n_{\text{out}}} a(\boldsymbol{\alpha}^L; \boldsymbol{w}_j^{\text{out}}, b_j^{\text{out}})\, \boldsymbol{e}_j \tag{3.5}$$

where $\boldsymbol{e}_j$ has components $[\boldsymbol{e}_j]_k = \delta_{j,k}$. $\{\boldsymbol{w}\}$ and $\{b\}$ denote the weights and biases of every neuron.

3.3 Optimisation

3.3.1 The learning task

The goal of machine learning is to find a set of optimal parameters $\{\boldsymbol{w}, b\}$ for each neuron such that the network acts like a desired function. Suppose the learning task is the classification of handwritten digits taken from the MNIST[a] dataset, which consists of thousands of grayscale images (28×28 pixels) of handwritten digits [84]. Then, the network's input layer should contain $28 \times 28 = 784$ neurons as each neuron should inherit the grayscale value of one pixel. The output layer includes ten neurons where the value of neuron $d = 0, \ldots, 9$ represents the probability that the desired digit is d. The neurons' parameters are optimised such that the network correctly classifies the handwritten digit, i.e., the output neuron corresponding to digit d has the highest value when the input image shows a d.

The classification of handwritten digits belongs to the class of *supervised* learning tasks. The network is trained to produce a desired output from a given input (labeled training data). There exist, as well, *unsupervised* learning tasks where the entire training data is unlabeled. Such tasks mainly include clustering [85] or association [86]. However, in the following, the learning is assumed to be supervised.

The *raison d'être* of neural networks is their capability to generalise their knowledge gained by learning the features of the provided data to previously unknown data. In the task of image classification, the network is trained using example images (labeled data) such that the trained network can generalise this knowledge to new images (unlabeled data). For this, two sets of data pairs are constructed. Each pair consists of a handwritten image $\boldsymbol{x}^{\text{in}}$ (the grayscale value of each pixel) and the digit that is drawn $\boldsymbol{x}^{\text{out}}$ (the desired network output, e.g., $(1,0,\ldots,0)^T$ if the image shows a zero). The first set is used to train the network and thus is called the *training set*. After injecting the grayscale value of each pixel into the network and processing each neuron, the network's output is compared to the corresponding desired output. Then, the network's parameters are updated such that the network's classification of the training set is improved. The other set is used to measure the network's generalisation capabilities. It is called the *test set*, as it is used to test whether the network can generalise its knowledge to previously unknown images. Sometimes it is also called *validation set*.

The training $\mathcal{S}_{\text{train}}$ and the test $\mathcal{S}_{\text{test}}$ set can be written as:

$$\mathcal{S}_t = \{(\boldsymbol{x}^{\text{in},k}, \boldsymbol{x}^{\text{out},k})\}_{k=1}^{n_t}, \quad t \in \{\text{train,test}\} \tag{3.6}$$

where n_t specifies the number of data pairs in the respective set.

3.3.2 The cost function

To evaluate the network's success in, e.g., classifying the input image, a quantity is needed, which compares the network's output $\boldsymbol{\alpha}^{\text{out}}$ for input $\boldsymbol{x}^{\text{in}}$ to the corresponding desired output $\boldsymbol{x}^{\text{out}}$. A common choice is the quadratic cost function, which is the sum of the squared differences of all network outputs and desired outputs:

$$C_t(\{\boldsymbol{w}\}, \{b\}; \mathcal{S}_t) = \frac{1}{2n_t} \sum_{k=1}^{n_t} \|\boldsymbol{\alpha}^{\text{out},k} - \boldsymbol{x}^{\text{out},k}\|^2, \quad t \in \{\text{train, test}\} \tag{3.7}$$

a MNIST stands for Modified data from National Institute of Standards and Technology.

where the sum is over all training/test inputs $\boldsymbol{x}^{\text{in},k}$ and $\boldsymbol{\alpha}^{\text{out},k} = \boldsymbol{\alpha}^{\text{out}}(\boldsymbol{x}^{\text{in},k}; \{\boldsymbol{w}\}, \{b\})$. Note that the cost function is non-negative. It becomes small if the network's output $\boldsymbol{\alpha}^{\text{out},k}$ is close to the desired output $\boldsymbol{x}^{\text{out},k}$ and larger the more they differ. The cost function evaluated on the training set is called the training cost. Analogously for the test cost.

To train the network, e.g., to classify handwritten digits, the parameters $\{\boldsymbol{w}, b\}$ need to be optimised such that the training cost (3.7) is reduced. In different words, the task is to find the global minimum of the cost landscape spanned by (3.7).

3.3.3 Gradient descent

The most famous optimisation algorithm to find the global minimum of a given cost function $C(\boldsymbol{w}, b)$ is called *gradient descent*. It is an iterative algorithm that optimises the parameters $\{\boldsymbol{w}, b\}$ according to the gradient of the cost function $\nabla_{\{\boldsymbol{w},b\}} C(\boldsymbol{w}, b)$.

In the following, the subscript train is dropped ($C = C_{\text{train}}$, $n = n_{\text{train}}$, $\mathcal{S} = \mathcal{S}_{\text{train}}$) as the test cost is not involved in the upcoming derivations.

The parameters $\{\boldsymbol{w}\}$ and $\{b\}$ are initialised randomly. Suppose the parameters are changed according to

$$[\boldsymbol{w}^l_j]_q \mapsto [\boldsymbol{w}^l_j]_q + [\mathrm{d}\boldsymbol{w}^l_j]_q, \tag{3.8a}$$

$$b^l_j \mapsto b^l_j + \mathrm{d}b^l_j. \tag{3.8b}$$

This induces the following change in the cost function:

$$\mathrm{d}C = \sum_{l=1}^{out} \sum_{j=1}^{n_l} \left(\frac{\partial C}{\partial b^l_j} \mathrm{d}b^l_j + \sum_{q=1}^{n_{l-1}} \frac{\partial C}{\partial [\boldsymbol{w}^l_j]_q} [\mathrm{d}\boldsymbol{w}^l_j]_q \right) \tag{3.9}$$

$\mathrm{d}C$ should be negative such that the cost for the new parameters is closer to the minimum. Thus, the parameter shifts should be

$$\mathrm{d}b^l_j = -\eta \frac{\partial C}{\partial b^l_j} \tag{3.10a}$$

$$[\mathrm{d}\boldsymbol{w}^l_j]_q = -\eta \frac{\partial C}{\partial [\boldsymbol{w}^l_j]_q} \tag{3.10b}$$

where $\eta > 0$ such that $\mathrm{d}C = -\eta \|\nabla C\|^2 < 0$. η is called the *learning rate* as it scales the size of the learning step. This defines the parameter update rule of gradient descent:

$$b \mapsto b - \eta \nabla_b C, \tag{3.11a}$$

$$\boldsymbol{w} \mapsto \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} C. \tag{3.11b}$$

The whole gradient descent algorithm is depicted in algorithm 1.

Algorithm 1: Gradient descent.

```
1 Initialise parameters $\boldsymbol{\theta} = \{\boldsymbol{w}, b\}$;
2 while $C(\boldsymbol{\theta})$ not converged do
3     $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} C$;
4 end
```

Stochastic gradient descent

The cost function C is defined as the average over costs L[a] of individual training data:

$$C(\mathcal{S}) = \frac{1}{n} \sum_{k=1}^{n} L([\mathcal{S}]_k), \tag{3.12a}$$

$$L(\boldsymbol{x}^{\text{in}}, \boldsymbol{x}^{\text{out}}) = \frac{1}{2} \|\boldsymbol{\alpha}^{\text{out}} - \boldsymbol{x}^{\text{out}}\|^2. \tag{3.12b}$$

Generally, the size of the training set is very large and, thus, makes the computation of the cost function and its gradient very slow. The training set is divided into smaller so-called *mini-batches* $\mathcal{S}^m \subset \mathcal{S}$ to speed up the training. $\mathcal{S}^m$ contains n_{batch} randomly picked training pairs of $\mathcal{S}$. The average of individual costs $L([\mathcal{S}^m]_k)$ gives an estimate of the total cost $C(\mathcal{S})$. Since the gradient is linear, these mini-batches can be used to approximate the gradient:

$$\nabla C(\mathcal{S}) = \frac{1}{n} \sum_{k=1}^{n} \nabla L([\mathcal{S}]_k) \approx \frac{1}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \nabla L([\mathcal{S}^m]_k) = \nabla C(\mathcal{S}^m). \tag{3.13}$$

With this, the parameter update rule yields

$$b \mapsto b - \frac{\eta}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \nabla_b C([S^m]_k), \tag{3.14a}$$

$$\boldsymbol{w} \mapsto \boldsymbol{w} - \frac{\eta}{n_{\text{batch}}} \sum_{k=1}^{n_{\text{batch}}} \nabla_{\boldsymbol{w}} C([S^m]_k). \tag{3.14b}$$

The mini-batches $\mathcal{S}^m$ are chosen at random during training. In one *training epoch*, each training pair is used once. Note that for each epoch $\mathcal{S} = \cup_{m=1}^{n/n_{\text{batch}}} \mathcal{S}^m$ must be fulfilled. Exemplary, if $n = 1000$ and $n_{\text{batch}} = 50$, then there are 20 iterations needed to complete one training epoch.

a L is usually called the *loss function*. However, cost and loss are sometimes used interchangeably.

3.3.4 The backpropagation algorithm

To be fully able to put these equations into code the computation of the cost function's gradient (3.14) has to be defined. *Backpropagation* is an algorithm that computes the gradient of the cost function with respect to the network's weights and biases for a single training pair. In order for it to work, there are two requirements the cost function needs to fulfil:

1. The cost function $C(\mathcal{S} = \{(\boldsymbol{x}^{\mathrm{in},j}, \boldsymbol{x}^{\mathrm{out},j})\}_{j=1}^{n})$ can be written as an average over cost functions for individual training data $(\boldsymbol{x}^{\mathrm{out},j}, \boldsymbol{x}^{\mathrm{out},j})$.

2. The cost function can be written as a function of the neural network's output: $C = C(\boldsymbol{\alpha}^{\mathrm{out}})$.

Both of these requirements are satisfied by the quadratic cost function (3.7).

For simplicity, the argument of the activation function is summarised as

$$z_j^l = \boldsymbol{\alpha}^{l-1} \cdot \boldsymbol{w}_j^l + b_j^l \tag{3.15}$$

such that $a(\boldsymbol{\alpha}^{l-1}; \boldsymbol{w}_j^l, b_j^l) = a(z_j^l)$. The partial derivatives of the cost function with respect to the weights and biases can be expressed by the partial derivative with respect to (3.15):

$$\frac{\partial C}{\partial b_j^l} = \sum_{k=1}^{n_l} \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_j^l} = \sum_{k=1}^{n_l} \frac{\partial C}{\partial z_k^l} \delta_k^j = \frac{\partial C}{\partial z_j^l} = c_j^l \tag{3.16a}$$

$$\frac{\partial C}{\partial \boldsymbol{w}_j^l} = \sum_{k=1}^{n_l} \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial \boldsymbol{w}_j^l} = \frac{\partial C}{\partial z_j^l} \boldsymbol{\alpha}^{l-1} = c_j^l \boldsymbol{\alpha}^{l-1} \tag{3.16b}$$

where $c_j^l = \partial C / \partial z_j^l$.

Only the computation of c_j^l is left. It can be rewritten as

$$c_j^l = \frac{\partial C}{\partial z_j^l} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_{k=1}^{n_{l+1}} c_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}. \tag{3.17}$$

With

$$\begin{aligned} \frac{\partial z_k^{l+1}}{\partial z_j^l} &= \frac{\partial}{\partial z_j^l} (\boldsymbol{w}_k^{l+1} \cdot \boldsymbol{\alpha}^l + b_k^{l+1}) = \frac{\partial}{\partial z_j^l} \Big(\sum_m [\boldsymbol{w}_k^{l+1}]_m a(z_m^l) + b_k^{l+1}\Big) \\ &= \sum_m [\boldsymbol{w}_k^{l+1}]_m \frac{\partial a(z_m^l)}{\partial z_j^l} = [\boldsymbol{w}_k^{l+1}]_j \, a'(z_j^l) \end{aligned} \tag{3.18}$$

it follows that

$$c_j^l = \sum_{k=1}^{n_{l+1}} c_k^{l+1} [\boldsymbol{w}_k^{l+1}]_j \, a'(z_j^l). \tag{3.19}$$

This gives rise to an iterative algorithm as c_j^l is determined by all $\{c_k^{l+1}\}_{k=1}^{n_k}$ of the following layer. The algorithm is initialised with

$$c_j^{\mathrm{out}} = \frac{\partial C}{\partial z_j^{\mathrm{out}}} = \frac{\partial C}{\partial \boldsymbol{\alpha}^{\mathrm{out}}} \cdot \frac{\partial \boldsymbol{\alpha}^{\mathrm{out}}}{\partial z_j^{\mathrm{out}}} = \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{\alpha}^{\mathrm{out}} - \boldsymbol{x}^{\mathrm{out}}) \cdot \boldsymbol{e}_j a'(z_j^{\mathrm{out}}) \tag{3.20}$$

which then can be used to calculate the values of previous layers. Hence the name backpropagation. The complete training algorithm using gradient descent and backpropagation is depicted in algorithm 2.

Algorithm 2: Training algorithm using backpropagation and gradient descent.

```
   /* Note: ⊙ indicates the elementwise (Hadamard) product:
      $\boldsymbol{a} \odot \boldsymbol{b} = \sum_k [\boldsymbol{a}]_k [\boldsymbol{b}]_k \boldsymbol{e}_k$                         */
 1 Initialise $\mathcal{W}^l$ (weight matrix with components $[\mathcal{W}^l]_{jk} = [\boldsymbol{w}_j^l]_k$) and $\boldsymbol{b}^l$ (bias
     vector with components $[\boldsymbol{b}^l]_j = b_j^l$) for every layer $l$ randomly;
 2 while $C(\boldsymbol{\theta})$ not converged do
       /* Iterate through $n_{\mathrm{batch}}$ mini-batches                */
 3     for $j = 1, \dots, n_{batch}$ do
           /* Iterate trough training data in mini-batch    */
 4         for $\{\boldsymbol{x}^{in}, \boldsymbol{x}^{out}\}$ in $\mathcal{S}^j$ do
               /* Feedforward                                 */
 5             $\alpha^{\mathrm{in}} = \boldsymbol{x}^{\mathrm{in}}$;
 6             for $l = 1, \dots out$ do
 7                 $\boldsymbol{z}^l = \mathcal{W}^l \cdot \boldsymbol{\alpha}^{l-1} + \boldsymbol{b}^l$;
 8                 $\boldsymbol{\alpha}^l = \boldsymbol{a}(\boldsymbol{z}^l)$;
 9             end
               /* Backpropagation                             */
10             $\boldsymbol{c}^{\mathrm{out}} = \nabla_{\boldsymbol{\alpha}^{\mathrm{out}}} C^j (\boldsymbol{\alpha}^{\mathrm{out}}; \boldsymbol{x}^{\mathrm{out}}) \odot \boldsymbol{a}'(\boldsymbol{z}^{\mathrm{out}})$;
11             for $l = L, \dots, 1$ do
12                 $c^l = (\mathcal{W}^{l+1})^T \cdot \boldsymbol{c}^{l+1} \odot \boldsymbol{a}'(\boldsymbol{z}^l)$;
13                 $\frac{\partial}{\partial \boldsymbol{b}^l} C(\boldsymbol{\alpha}^{\mathrm{out}}; \boldsymbol{x}^{\mathrm{out}}) = \boldsymbol{c}^l$;
14                 $\frac{\partial}{\partial \mathcal{W}^l} C(\boldsymbol{\alpha}^{\mathrm{out}}; \boldsymbol{x}^{\mathrm{out}}) = \boldsymbol{c}^l \cdot (\boldsymbol{\alpha}^{l-1})^T$;
15             end
16         end
           /* Gradient descent                                */
17         $\boldsymbol{b}^l = \boldsymbol{b}^l - \frac{\eta}{n_{\mathrm{batch}}} \sum_{\{\boldsymbol{x}^{\mathrm{in}}, \boldsymbol{x}^{\mathrm{out}}\} \in \mathcal{S}^j} \frac{\partial}{\partial \boldsymbol{b}^l} C(\boldsymbol{\alpha}^{\mathrm{out}}; \boldsymbol{x}^{\mathrm{out}})$;
18         $\mathcal{W}^l = \mathcal{W}^l - \frac{\eta}{n_{\mathrm{batch}}} \sum_{\{\boldsymbol{x}^{\mathrm{in}}, \boldsymbol{x}^{\mathrm{out}}\} \in \mathcal{S}^j} \frac{\partial}{\partial \mathcal{W}^l} C(\boldsymbol{\alpha}^{\mathrm{out}}; \boldsymbol{x}^{\mathrm{out}})$;
19     end
20 end
```

3.3.5 Improving the learning

The methods presented previously correspond to the basic techniques of machine learning and have been further improved ever since. Here, the focus is on advanced machine learning techniques, which turn out useful in later chapters. However, other techniques can be found in appendix A.

Adam

Besides optimisation via vanilla gradient descent, there are also different variations, which have shown a faster convergence of the cost function. One such variation is the Adaptive Momentum Estimation (Adam) [87]. Its principle is similar to gradient descent. The key concept is utilising the information of past gradients to get a better estimation of the present gradient. The algorithm is sketched in algorithm 3.

Algorithm 3: Adam [87].

```
1  Initialise parameters $\theta = \{\boldsymbol{w}, b\}$;
2  Initialise default values: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$;
3  $\boldsymbol{m}_0 = 0$ (First momentum vector);
4  $\boldsymbol{v}_0 = 0$ (Second momentum vector);
5  $t = 0$ (Timestep);
6  while $C(\theta)$ not converged do
7      $t = t + 1$;
8      $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} C$;
9      $\boldsymbol{m}_t = \boldsymbol{\beta}_1 \cdot \boldsymbol{m}_{t-1} + (1 - \boldsymbol{\beta}_1) \cdot \boldsymbol{g}_t$;
10     $\boldsymbol{v}_t = \boldsymbol{\beta}_2 \cdot \boldsymbol{v}_{t-1} + (1 - \boldsymbol{\beta}_2) \cdot \boldsymbol{g}_t^{\odot 2}$;
11     $\hat{\boldsymbol{m}}_t = \boldsymbol{m}_t / (1 - (\boldsymbol{\beta}_1)^t)$;
12     $\hat{\boldsymbol{v}}_t = \boldsymbol{v}_t / (1 - (\boldsymbol{\beta}_2)^t)$;
13     $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t + \epsilon})$;
14 end
```

Other notable gradient-based optimisation algorithms that are known to improve the convergence of the cost function are AdaGrad [88] and RmsProp [89]. A great overview can be found in [8].

Overfitting

When the training set is small, or the learning is performed too long, the network tends to concentrate too much on the training data, resulting in a poor generalisation. It optimises its parameters to classify the training data well but fails to generalise it to unseen data. The network learns specific features of the training data that the underlying model does not represent. This effect is called *overfitting*. The most apparent consequence of overfitting during training is a decreasing (or low) test cost while the training cost is increasing. A prevalent technique to reduce overfitting is to add a regularisation term to the cost function [90]. The new cost function then is given by:

$$C = C_0 + \frac{\lambda}{2n} \sum_{l,j} \|\boldsymbol{w}_j^l\|^2 \tag{3.21}$$

where C_0 is the unregularised cost function (e.g. (3.7)) and $\lambda > 0$ the regularisation parameter. This regularisation term is punishing neurons with large weights $\|\boldsymbol{w}_j^l\|$. With this, the neurons are prevented from concentrating too much on specific features/inputs.

Other improvements have been made using *early stopping* (stopping the training before the test cost drops) or *dropout* (randomly ignoring neurons in the network) [91].

Chapter 4

Quantum Machine Learning

4.1 Introduction

Machine learning models are used to learn patterns from given data to apply this knowledge to previously unknown data. Quantum machine learning (QML) models seek the same goal but with quantum data. This is of high interest as quantum data, in general, is limited. A famous QML model is the so-called quantum neural network (QNN) constructed in analogy to its classical counterpart [10]. It is composed of quantum perceptrons, which are defined as unitary operators that propagate the information from layer to layer through the QNN [10, 13, 20–34]. Such models are especially suited for the operation using quantum computers as quantum data cannot be simulated efficiently using classical computers [66] (see chapter 2). QNNs on quantum computers are primarily implemented as hybrid quantum-classical algorithms (also called variational quantum algorithms) [41–44]. These are parameterised quantum circuits, which are trained using classical optimisation methods [38–41]. The QNNs described in this chapter are implemented and trained using quantum computers and different learning tasks in chapters 5 and 6.

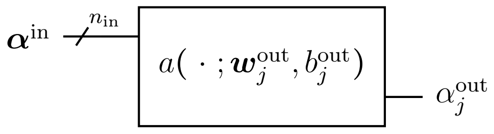
This chapter is structured as follows. Section 4.2 features the definition of quantum neural networks and their fundamental building block, the quantum perceptron. In section 4.3, the hybrid quantum-classical training of a QNN is described. Additionally, the quantum algorithms of important QNNs are presented.

4.2 Quantum Neural Networks

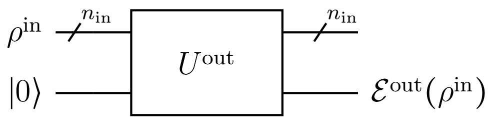
Quantum neural networks are the most famous quantum machine learning models. Here, the "quantum" denotes that the neural network acts on quantum states of some bounded Hilbert space $\mathcal{H} = \mathbb{C}^{2n_q}$ of an n_q-qubit system. Note that its classical counterpart acts on n_c floating values $\boldsymbol{x}^{\text{in}} \in \mathbb{R}^{n_c}$ (see section 3.2.2). These fundamentally different training data require a redefinition of the network itself. The following definition of a QNN is based on [10]. It is described as a sequence of completely positive layer transition maps, so-called *quantum perceptrons*.

4.2.1 The quantum perceptron

The quantum perceptron is the fundamental building block of QNNs. Classically, it is defined as a function acting on some input $\boldsymbol{x}$ that produces an output $a(\boldsymbol{x};\boldsymbol{w},b)$ dependent on its weights $\boldsymbol{w}$ and bias b (see section 3.2.1). Various proposals suggest a quantum version of this perceptron [10, 13, 20–34]. The quantum perceptron described in [10] is defined as a general unitary operator acting on all n_{in} input qubits ρ^{in} and one output qubit ρ^{out}. In general, there could be multiple output qubits. However, here it will be restricted to a single qubit. The difference between the classical and quantum perceptron is depicted in Fig. 4.1.


$\boldsymbol{\alpha}^{\text{in}}$
n_{in}
$a(\,\cdot\,;\boldsymbol{w}_j^{\text{out}}, b_j^{\text{out}})$
α_j^{out}

(a) A circuit representation of the classical perceptron. The neuron's activation is given by $\alpha_j^{\text{out}} = a(\boldsymbol{\alpha}^{\text{in}};\boldsymbol{w}_j^{\text{out}},b_j^{\text{out}})$ (see section 3.2.1).



(b) A quantum circuit representation of the quantum perceptron. U^{out} acts on the input state ρ^{in} and the output state initialised in $|0\rangle$. The first n_{in} qubits are traced out, leaving $\mathcal{E}^{\text{out}}(\rho^{\text{in}})$ = (4.1).

Figure 4.1: A comparison of the classical and quantum perceptron.

Mathematically, the output of the quantum perceptron can be written as

$$\rho^{\text{out}} = \mathcal{E}^{\text{out}}(\rho^{\text{in}}) = \text{Tr}_{\text{in}}\left(U^{\text{out}}\left(\rho^{\text{in}} \otimes |0\rangle\langle 0|\right)U^{\text{out}\dagger}\right) \tag{4.1}$$

where U^{out} is the unitary representation of the quantum perceptron acting on all the input qubits and the output qubit. Note that the trace is over the input space leaving only the output qubit after the application of U^{out}.

4.2.2 The quantum neural network architecture

Similar to its classical counterpart, a QNN is built by composing and linking quantum perceptrons. The structure and notation of the classical architecture can be recycled from section 3.2.2. The QNN works in a similar feed-forward fashion (see Fig. 4.2). The input state ρ^{in} is initialised in the input qubits. Then, the state's information is propagated through the network by applying each quantum perceptron. The state after applying the jth quantum perceptron of layer l is given by

$$\rho_j^l = \mathcal{E}_j^l(\rho^{l-1}) = \text{Tr}_{l-1}\left(U_j^l\left(\rho^{l-1} \otimes |0\rangle\langle 0|\right)U_j^{l\dagger}\right) \tag{4.2}$$

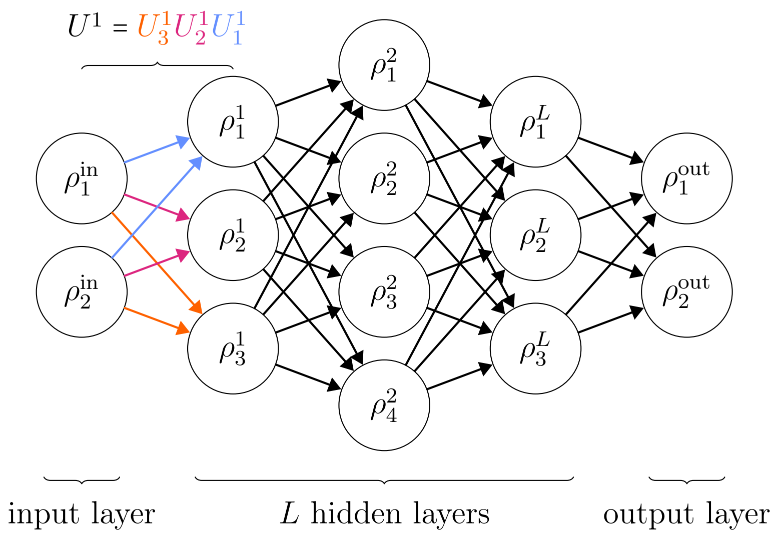


Figure 4.2: The feed-forward quantum neural network architecture. It consists of one input layer, L hidden layers, and one output layer. The perceptron unitaries are applied from top to bottom.

where $\rho^l = \bigotimes_{j=1}^{n_l} \rho_j^l$. U_j^l is the quantum perceptron acting on all n_{l-1} qubits of layer $l-1$ and the jth qubit of layer l. The full state of layer l can be written as

$$\rho^l = \mathcal{E}^l\left(\rho^{l-1}\right) = \text{Tr}_{l-1}\left(U^l\left(\rho^{l-1} \otimes |0\dots0\rangle_l \langle 0\dots0|\right) U^{l^\dagger}\right) \tag{4.3}$$

where $|0\dots0\rangle_l = |0\rangle^{\otimes n_l}$ and $U^l = \prod_{j=n_l}^{1} U_j^l$ is the layer unitary. The network's output is obtained by recursively applying (4.3) to the network's input state ρ^{in}:

$$\rho^{\text{out}} = \left(\mathcal{E}^{\text{out}} \circ \mathcal{E}^L \circ \dots \circ \mathcal{E}^1\right)\left(\rho^{\text{in}}\right) = \mathcal{E}\left(\rho^{\text{in}}\right). \tag{4.4}$$

This implements a quantum feed-forward neural network that can be optimised using a quantum analogous of the backpropagation algorithm (see appendix C.1).

4.3 Variational Quantum Algorithms

4.3.1 Introduction

The QNN described in section 4.2 using the quantum backpropagation algorithm (appendix C.1) can be fully simulated classically. However, as the Hilbert space dimension scales exponentially with the number of qubits, this simulation is restricted to a few qubits, even with today's supercomputers. Thus, training a QNN for a larger number of qubits can only be done using quantum computation (see chapter 2). The qubits of the quantum computer serve as the qubits for the QNN. The quantum perceptrons can be decomposed into basis gates, i.e., unitary transformations of the quantum computer's qubits. However, although a QNN can be implemented on a quantum computer, the quantum backpropagation algorithm cannot be applied as information about the qubits' states can only be obtained via measurements. The actual state of a qubit cannot be read, and thus, the parameter matrix (C.3) cannot be computed. The solution to this problem is replacing the quantum backpropagation algorithm with a classical optimisation method, e.g., gradient descent. Such *hybrid quantum-classical algorithms* or *variational quantum algorithms* feature the classical optimisation of a parameterised quantum circuit [92].

Thus, training a QNN on a quantum computer in a hybrid-classical manner requires a parameterised quantum circuit representation of the QNN and a quantum algorithm that computes the cost function These are defined in section 4.3.2. A straightforward quantum circuit implementation of the QNN described in section 4.2 is described in section 4.3.3. The number of parameters and, thus, the computational cost can be reduced by composing the unitary operator in terms of parameterised quantum gates. This approach is presented in section 4.3.4. Both of these networks are composed of dissipative quantum perceptrons. Another important QNN architecture is represented by the quantum approximate optimisation algorithm (QAOA). It is defined in section 4.3.5.

4.3.2 The training algorithm

In general, the goal of training a QNN is to learn correlations between some input states $\{\rho^{\text{in},k}_{\text{train}}\}_{k=1}^{n_{\text{train}}}$ and output states $\{\rho^{\text{out},k}_{\text{train}}\}_{k=1}^{n_{\text{train}}}$ such that this information can be generalised to input states $\{\rho^{\text{in},k}_{\text{test}}\}_{k=1}^{n_{\text{test}}}$ with unknown[a] output states $\{\rho^{\text{out},k}_{\text{test}}\}_{k=1}^{n_{\text{test}}}$. To optimise a QNN such that the network acts in the desired way: $\mathcal{E}(\rho^{\text{in},k}) = \rho^{\text{out},k}$, a quantity is needed that measures the closeness between $\mathcal{E}(\rho^{\text{in},k})$ and the corresponding desired output state $\rho^{\text{out},k}$. This quantity is called the *cost function*. The closeness between two states ρ and σ is quantified by the fidelity

$$F(\rho,\sigma) = \text{Tr}\left(\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right)^2 \overset{\rho=|\Psi_\rho\rangle\langle\Psi_\rho|}{=} \langle\Psi_\rho|\,\sigma\,|\Psi_\rho\rangle \overset{\sigma=|\Psi_\sigma\rangle\langle\Psi_\sigma|}{=} \|\,\langle\Psi_\rho|\Psi_\sigma\rangle\,\|^2. \tag{4.5}$$

Thus, a possible cost function for the training states is given by:

$$C_{\text{train}} = \frac{1}{n_{\text{train}}}\sum_{k=1}^{n_{\text{train}}} F(\mathcal{E}(\rho^{\text{in},k}_{\text{train}}), \rho^{\text{out},k}_{\text{train}}) \tag{4.6}$$

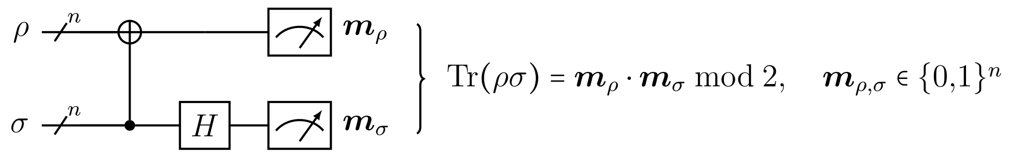
which ranges from 0 (worst) to 1 (best). Of course, this cost function can also be used to calculate the test cost C_{test}. In the rest of this chapter, the subscripts train and test are dropped as the computation of the training and test cost is analogous.

a The output states $\{\rho^{\text{out},k}_{\text{test}}\}_{k=1}^{n_{\text{test}}}$ are unknown to the QNN as they are not used to train it.

In general, the network's output $\mathcal{E}(\rho^{\text{in}})$ is a mixed state. Therefore, the fidelity (4.5) can only be efficiently calculated on a quantum computer if the desired output state ρ^{out} is pure.

Pure output states

First, suppose the desired output states are pure: $\rho^{\text{out}} = |\phi^{\text{out}}\rangle\langle\phi^{\text{out}}|$. Then, the fidelity is given by $F(\mathcal{E}(\rho^{\text{in}}), |\phi^{\text{out}}\rangle\langle\phi^{\text{out}}|)$ and can be computed using the *destructive swap test* (see Fig. 4.3) [93, 94]. It consists of Bell basis measurements of $|\phi^{\text{out}}\rangle\langle\phi^{\text{out}}| \otimes \mathcal{E}(\rho^{\text{in}})$ and classical post-processing of its result $\boldsymbol{m} = (m_1, \dots m_n)^T$ where the fidelity is obtained by $\sum_{j=1}^{n/2} m_j m_{j+n/2}$ modulo two.



ρ
σ
n
n
H
$\boldsymbol{m}_\rho$
$\boldsymbol{m}_\sigma$
$\text{Tr}(\rho\sigma) = \boldsymbol{m}_\rho \cdot \boldsymbol{m}_\sigma \bmod 2, \quad \boldsymbol{m}_{\rho,\sigma} \in \{0,1\}^n$

Figure 4.3: The quantum circuit implementing the destructive swap test. If either ρ, σ, or both are pure, this quantum algorithm can be used to calculate their fidelity (4.5).

The complete algorithm of computing $F(\mathcal{E}(\rho^{\text{in}}), |\phi^{\text{out}}\rangle\langle\phi^{\text{out}}|)$ is depicted in Fig. 4.4.

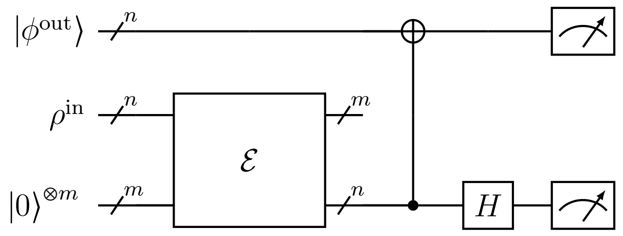


Figure 4.4: The quantum circuit implementing the QNN training. n denotes the number of qubits needed to initialise ρ^{in} and $|\phi^{\text{out}}\rangle$. m denotes the number of qubits the QNN additionally requires. $m = n$ for the dissipative QNNs (see sections 4.3.3 and 4.3.4). $m = 0$ for the QAOA (see section 4.3.5).

Mixed output states

There does not exist an efficient quantum algorithm for the computation of the fidelity (4.5) between two mixed states. In this case, an alternative measure has to be defined. Commonly, the Hilbert-Schmidt distance is chosen:

$$d_{\text{HS}}(\rho, \sigma) = \text{Tr}\left((\rho - \sigma)^2\right) = \text{Tr}(\rho^2) - 2\,\text{Tr}(\rho\sigma) + \text{Tr}(\sigma^2) \tag{4.7}$$

which is equivalent to the fidelity when ρ and σ are pure states. The computation of $d_{\text{HS}}(\mathcal{E}(\rho^{\text{in}}), \rho^{\text{out}})$ requires three evaluations of the destructive swap test shown in Fig. 4.3:

1. $\rho = \mathcal{E}(\rho^{\text{in}}), \sigma = \mathcal{E}(\rho^{\text{in}}) \;\rightarrow \text{Tr}(\mathcal{E}(\rho^{\text{in}})\mathcal{E}(\rho^{\text{in}}))$.

2. $\rho = \mathcal{E}(\rho^{\text{in}}), \sigma = \rho^{\text{out}} \rightarrow \text{Tr}(\mathcal{E}(\rho^{\text{in}})\rho^{\text{out}})$.

3. $\rho = \rho^{\text{out}}, \sigma = \rho^{\text{out}} \rightarrow \text{Tr}(\rho^{\text{out}}\rho^{\text{out}})$.

Of course, $\mathcal{E}(\rho^{\text{in}})$ is not computed classically but as shown in Fig. 4.4.

In a superconducting quantum computer [70, 71], the qubit states cannot be reset to the $|0\rangle$ state. Thus, each qubit of the QNN has to be represented by exactly one qubit of the quantum computer. As a result, the QNN requires additional $m = \sum_{l=1}^{\text{out}} n_l$ qubits. An ion trap quantum computer [72–74], however, would allow the reset of qubits and thus would only need $\max\{n_l + n_{l+1}\}_{l=\text{in}}^{L}$ qubits in total [95]. In the following, the quantum computer is assumed to consist of superconducting qubits as these are provided by IBM [96] and utilised in chapters 5 and 6.

Optimisation

The knowledge from optimising classical neural networks can be reinterpreted to train QNNs. Suppose the unitary representation of the quantum perceptron is parameterised by some n_p parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_{n_p})$. These parameters are equivalent to the weights and biases of classical perceptrons. By applying classical optimisers (see section 3.3.3) to the cost function (4.6), the parameters can be iteratively updated such that the QNN's output converges to the desired output states.

The algorithm for optimising a QNN is depicted in algorithm 4. It is very similar to the classical optimisation algorithm (see section 3.3.3). As the states of a certain neuron cannot be observed, the backpropagation algorithm is not applicable. The gradient of the cost function has to be computed by a numerical

Algorithm 4: Optimising QNNs using gradient descent.

```
1 Initialise the parameters $\boldsymbol{\theta}$ randomly.;
2 while $C(\boldsymbol{\theta}) = (4.11)$ not converged do
      /* Make trial parameters                                          */
3     $\theta^{\pm} = \{\boldsymbol{\theta} \pm \boldsymbol{e}_p\epsilon\}_{p=1}^{n_p}, \quad [e_p]_k = \delta_k^p$;
      /* Compute the cost gradient via (4.8) (or (4.10))                */
4     $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \sum_{\boldsymbol{\theta}_p^{\pm} \in \theta^{\pm}} \frac{C(\boldsymbol{\theta}_p^{+}) - C(\boldsymbol{\theta}_p^{-})}{2\epsilon} \boldsymbol{e}_p$;
      /* Update the parameters, here: via Gradient Descent (see
         algorithm 1)                                                   */
5     $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} C$;
6 end
```

approximation:

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \sum_{p=1}^{n_p} \frac{C(\boldsymbol{\theta} + \boldsymbol{e}_p \epsilon) - C(\boldsymbol{\theta} - \boldsymbol{e}_p \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2) \tag{4.8}$$

where $\boldsymbol{e}_p$ has components $[\boldsymbol{e}_p]_k = \delta_{p,k}$. A special case arises if the QNN's quantum circuit representation can be written as

$$U(\boldsymbol{\theta}) = V_m U_m(\theta_m) \dots V_2 U_2(\theta_2) V_1 U_1(\theta_1) \tag{4.9}$$

where V_j are constant arbitrary circuits and $U_j(\theta_j) = e^{-\frac{i}{2} H_j \theta_j}$ are parameterised gates generated from a hermitian operator H_j. Then the gradient can be computed using the so-called *parameter-shift rule* [39, 39, 41, 97, 98]:

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \sum_{p=1}^{n_p} \frac{C(\boldsymbol{\theta} + \boldsymbol{e}_p \epsilon) - C(\boldsymbol{\theta} - \boldsymbol{e}_p \epsilon)}{2\sin(\epsilon)}. \tag{4.10}$$

Note that this rule can be used to exactly compute the cost's gradient. For $\epsilon \to 0$, this expression is approximately equal to the finite-difference approximation (4.8).

The parameterised cost function can be written as

$$C(\boldsymbol{\theta}) = \frac{1}{n_{\text{train}}} \sum_{k=1}^{n_{\text{train}}} F(\mathcal{E}(\boldsymbol{\theta}; \rho^{\text{in},k}), \rho^{\text{out},k}). \tag{4.11}$$

Note that the gradient descent update rule (line 5 in algorithm 4) can be exchanged by any other optimiser, e.g. Adam.

Training a QNN in the NISQ era

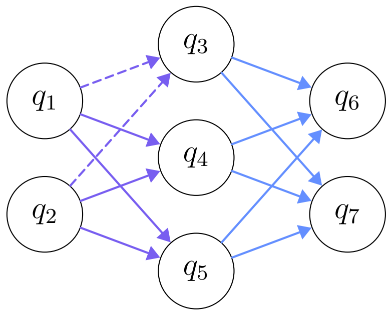
Today's NISQ devices are characterised by limited qubits and high noise levels restricting the quantum circuit size (see section 2.3.3). The noise is the central obstacle as it hinders the QNN's training by making the measurement results inaccurate and thus limiting the accuracy of the update steps. Thus, the cost function will be bounded to a specific value. This value can be determined by the so-called *identity cost* $C(\boldsymbol{\theta}_{\mathbb{1}})$ (4.11), where $\boldsymbol{\theta}_{\mathbb{1}}$ are network parameters such that $\mathcal{E}(\boldsymbol{\theta}_{\mathbb{1}}) = \mathbb{1}$ [99]. Each gate is still applied and adds noise to the quantum circuit. This feature is crucial to quantify the overall noise level.

An analysis of training QNNs on NISQ devices can be found in chapter 5.

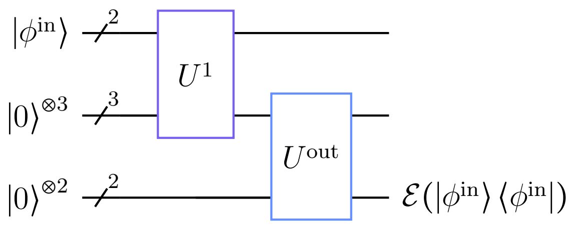
4.3.3 The general dissipative quantum neural network

The dissipative QNN presented here implements the QNN ansatz described in section 4.2. Its definition is taken from the supplementary information of [10]. In the following, it is referred to as DQNN_U.

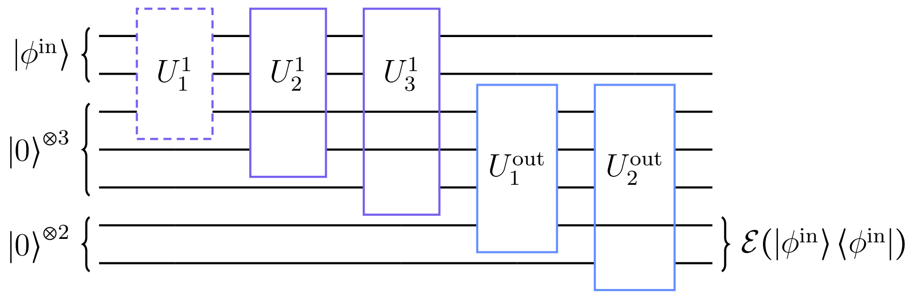
The QNN is composed of quantum perceptrons which are defined as general unitary operators U_j^l acting on all qubits of layer $l-1$ and the jth qubit of layer l. The network architecture and the corresponding quantum algorithm are depicted in Fig. 4.5. In order to train such a QNN on a quantum computer using a classical



(a) The QNN architecture of the DQNN$_\mathrm{U}$.


$|\phi^{\mathrm{in}}\rangle$
$|0\rangle^{\otimes 3}$
$|0\rangle^{\otimes 2}$
2
3
2
U^1
U^{out}
$\mathcal{E}(|\phi^{\mathrm{in}}\rangle\langle\phi^{\mathrm{in}}|)$

(b) The quantum circuit implementing the DQNN$_\mathrm{U}$. The gates are colored similarly to the quantum perceptron in (a).



(c) The decomposed quantum circuit of (b). The gates are styled according to (a).

Figure 4.5: The general dissipative quantum neural network.

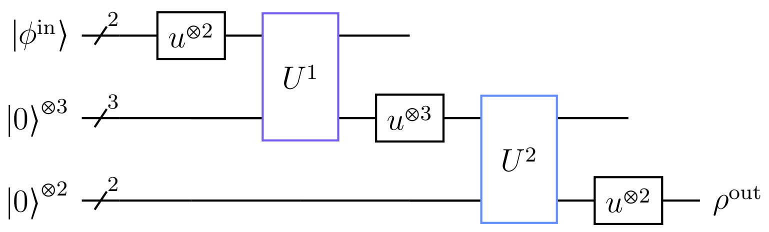
optimisation method, the quantum perceptron has to be parameterised. Here, the quantum perceptron U_j^l is defined as a $2^{n_{l-1}+1} \times 2^{n_{l-1}+1}$ parameterised unitary matrix defined by

$$U_j^l = e^{iK_j^l}, \quad K_j^l = \sum_{\alpha_1,\dots,\alpha_{n_{l-1}+1}} k^{l,j}_{\alpha_1,\dots,\alpha_{n_{l-1}+1}} \sigma^{\alpha_1} \otimes \dots \otimes \sigma^{\alpha_{n_{l-1}+1}}, \quad \{\alpha_j\}_{j=1}^{n_{l-1}+1} \in \{0,1,2,3\} \tag{4.12}$$

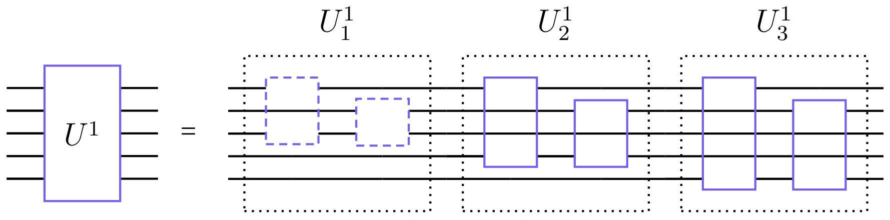
where $\boldsymbol{k}^{l,j} = (k^{l,j}_{0\dots0}, \dots, k^{l,j}_{3\dots3}) \in \mathbb{R}^{4^{n_{l-1}+1}}$ are the parameters and σ^α, $\alpha \in \{0,1,2,3\}$ are the Pauli matrices. Note that each unitary $U_j^l(\boldsymbol{k}^{l,j})$ has as many parameters as components, namely $4^{n_{l-1}+1}$. This enables the quantum perceptron to represent any unitary operator.

4.3.4 The CAN-based dissipative quantum neural network

Suppose the goal is to train a DQNN$_\mathrm{U}$, as shown in Fig. 4.5a. The whole network would have $n_p = \sum_{l=1}^{\mathrm{out}} n_l 4^{n_{l-1}+1} = 704$ parameters. Thus, in each epoch, the cost has to be computed $n_{\mathrm{train}} \times n_{\mathrm{test}} \times 2 \times n_p \sim 10^5$ times to get an approximation of its



(a) The quantum circuit implementing the DQNN$_{\text{CAN}}$ with the architecture from Fig. 4.5a. The gates colors are chosen according to Fig. 4.5a.



(b) The quantum circuit implementing the DQNN$_{\text{CAN}}$ quantum perceptron. The empty gates on the right hand site are canonical gates (4.14). The colors refer to (a) and Fig. 4.5a.

Figure 4.6: The CAN-based dissipative quantum neural network.

gradient. This is very costly, considering that one of these executions should be performed with about 2^{12} shots. Thus, the number of parameters of the quantum perceptron has to be reduced to efficiently train the QNN.

Here, the quantum perceptron U_j^l comprises general parameterised two-qubit gates $G_{k,j}^l(\boldsymbol{\theta})$

$$U_j^l = \prod_{k=n_{l-1}}^{1} G_{k,j}^l \tag{4.13}$$

where $G_{k,j}^l$ acts on the kth and jth qubit of layer l. $G_{j,k}^l$ is chosen to be the *canonical gate* [100] which is defined as

$$\begin{aligned}\text{CAN}(\theta_1,\theta_2,\theta_3) &= e^{-\frac{\pi}{2}\theta_1 \text{X}\otimes\text{X}} e^{-\frac{\pi}{2}\theta_2 \text{Y}\otimes\text{Y}} e^{-\frac{\pi}{2}\theta_3 \text{Z}\otimes\text{Z}} \\ &= \text{RXX}(\theta_1\pi)\text{RYY}(\theta_2\pi)\text{RZZ}(\theta_3\pi).\end{aligned} \tag{4.14}$$

The qubits which are not affected by the gate have been neglected in the definition (4.13). The precise definition of the quantum perceptron can be found in appendix C. In the following, this QNN is referred to as the DQNN$_{\text{CAN}}$.

Fig. 4.6 features the implementation of the DQNN$_{\text{CAN}}$'s quantum circuit with respect to the architecture depicted in Fig. 4.5a. Note that the quantum circuit features additional u gates. These are universal single-qubit gates needed to secure the generality of the quantum perceptron. Any two-qubit unitary transformation can be decomposed into $u^{\otimes 2}\text{CAN}u^{\otimes 2}$ [101–106].

The quantum circuit shown in Fig. 4.6 is constructed as follows. For each layer $l = 1,\dots,\text{out}$, first, single-qubit u gates are applied to every n_{l-1} input qubit. Afterwards, the layer unitary U^l is applied to all $n_{l-1} + n_l$ qubits. It is composed of quantum perceptrons U_j^l acting on all n_{l-1} input qubits and the jth output qubit: $U^l = \prod_{j=n_l}^{1} U_j^l$. Each U_j^l consists of canonical gates as defined in (4.13) and (4.14). The canonical gate is applied to each input qubit and the jth output qubit. In the end, single-qubit u gates are applied to every output qubit.

The u gate and the CAN gate are parameterised by three parameters. Thus, the total number of parameters of this network is given by $n_p = 3n_{\text{out}}+3\sum_{l=1}^{\text{out}} n_{l-1}(1+n_l)$. A DQNN$_{\text{CAN}}$ of the form shown in Fig. 4.5a only requires $n_p = 57$ parameters and thus, is perfectly suitable for the execution on NISQ devices.

4.3.5 The quantum approximate optimisation algorithm

An alternative parameterised quantum circuit class that has attracted considerable interest in the field of QML is the quantum approximate optimisation algorithm (QAOA) [46–48]. It features a sequence of alternating parameterised unitary operators $e^{-it_jA}, e^{-i\tau_jB} \in U(d)$, where $t_j, \tau_j \in \mathbb{R}$ and A and B are hermitian matrices randomly generated from the Gaussian unitary ensemble (GUE). The total action of the QAOA with $2N$ parameters can be written as

$$\mathcal{U} = e^{-i\tau_N B} e^{-it_N A} \cdots e^{-i\tau_1 B} e^{-it_1 A}. \tag{4.15}$$

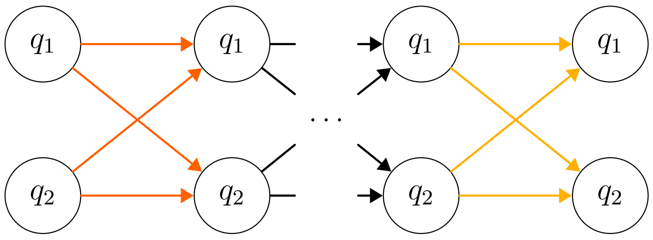
This sequence of parameterised unitary operators can be interpreted as an N-layer QNN where the quantum perceptron is defined layer-wise:

$$U^l = e^{-i\tau_l B} e^{-it_l A}. \tag{4.16}$$

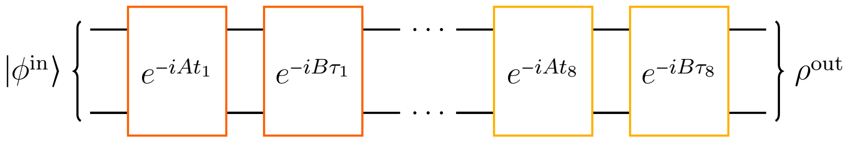
Note that every quantum perceptron of the QAOA acts on the same qubits. This reveals the main difference to the QNNs discussed before. The QNN has no dissipative nature but instead consists of layers of constant width. The QNN architecture and the quantum algorithm are depicted in Fig. 4.7.

The number of layers is determined by the dimension of the Hilbert space d. It has been shown that the number of parameters has to be at least d^2 such that the QNN converges to an optimal solution [49]. Optimal here means finding the global maximum of the cost function. Thus, a QNN of the form shown in Fig. 4.7a (two qubits) has to have $d^2 = 16$ parameters or eight layers.

So far, the architecture of the QNN has been assumed to be of constant width to match the quantum perceptron's definition. In fact, this is sufficient considering the task of learning a unitary transformation (see chapter 5). However, to suit more general learning tasks, the QNN representation has to be extended to QNNs with layers of variable width. The straightforward way of doing this is to apply $\mathcal{U} \in U(2^n)$ to $n = \max\{n_l\}_{l=\text{in}}^{\text{out}}$ qubits and to ignore the $n-n_l$ overlapping qubits. The total number of layers and parameters is determined by the number of components of the desired transformation. Here, the number of parameters is set equal to the number of components. This choice is validated in appendix C.4.



(a) The quantum neural network architecture representation of a two-qubit QAOA.



(b) The quantum circuit implementing a two-qubit QAOA.

Figure 4.7: The quantum approximate optimisation algorithm.

Chapter 5

Learning Unitaries on NISQ devices

5.1 Introduction

Quantum machine learning models executed on quantum computers are expected to outperform their classical counterparts in numerous tasks (see chapter 4) [107, 108]. However, today's NISQ devices (see section 4.3.2) limit the choice of QNNs as it sets constraints on the number of qubits and gates of the QNN's quantum circuit [9, 45]. The high noise levels pose a challenge for the training of QNNs as they make it difficult to accurately calculate costs and gradients [45, 53, 59, 60]. Any approach implementing QNNs on currently available quantum computers must be evaluated and optimised for their noise tolerance.

Sections 4.3.4 and 4.3.5 include the definitions of two promising QNN architectures suitable for the execution on NISQ devices as their quantum circuits feature few gates and few parameters. On the one hand, there is the CAN-based dissipative quantum neural network ($\mathrm{DQNN}_{\mathrm{CAN}}$), whose quantum perceptron is defined as a completely positive map. On the other hand, the quantum approximate optimisation algorithm (QAOA) features a sequence of alternating unitary operations. In this chapter, the subscript of $\mathrm{DQNN}_{\mathrm{CAN}}$ is dropped because the $\mathrm{DQNN}_{\mathrm{U}}$ will not be featured here ($\mathrm{DQNN}{=}\mathrm{DQNN}_{\mathrm{CAN}}$).

This chapter presents the comparison of training both networks under NISQ device noise circumstances. The task of both networks is the learning of an unknown unitary transformation and the application of this knowledge to unknown states. This learning task is described in section 5.2. The networks are implemented via the open-source SDK Qiskit [61] and executed on simulated and real quantum devices hosted by IBM [62]. The results of this analysis are presented in section 5.3.

The findings of this chapter have been published in [99]. It includes an analysis of the DQNN's and the QAOA's generalisation capabilities under the influence of different gate noise strengths. For the completeness of this thesis, these results can be found in appendix D.1.

5.2 The Learning Task

The application fields of quantum neural networks and classical neural networks overlap. However, QNNs are the most efficient when operated on quantum data as the encoding of classical information is very costly [107]. A very general learning task of the QNN is to learn a unitary transformation from a set of states $\{|\phi^{\text{in}}\rangle, |\phi^{\text{out}}\rangle = V|\phi^{\text{in}}\rangle\}$ where the unitary V itself is unknown to the network. The dimension of these states should be $2^{n_{\text{in}}}$ as they are initialised on qubits. The Hilbert space is given by $\mathcal{H} = \mathbb{C}^{2n_{\text{in}}}$.

Analogous to classical machine learning, a training and test set can be defined (see section 3.3.1):

$$\mathcal{S}_t = \{(|\phi^{\text{in},k}\rangle, |\phi^{\text{out},k}\rangle = V|\phi^{\text{in},k}\rangle)\}_{k=1}^{n_t}, \quad t \in \{\text{train,test}\}. \tag{5.1}$$

The states of the training set are used to train the QNN; the test set is used to quantify the QNN's generalisation capability. To guarantee the network's success the number of training pairs should be larger or equal the dimension of the states' Hilbert space $\dim(\mathcal{H}) = 2^{n_{\text{in}}}$ [109]. For an analysis of the network's generalisation capabilities, it is convenient to choose $n_{\text{train}} \leq \dim(\text{H})$ (see section 5.3.2).

5.2.1 The cost function

The cost function is a quantity that measures the network's success in producing states $\mathcal{E}(|\phi^{\text{in},k}\rangle\langle\phi^{\text{in},k}|)$ that are close to the desired output states $|\phi^{\text{out},k}\rangle$. The closeness of two states can be quantified using the fidelity (4.5). The normalised sum of all fidelities between the network's output state and the desired output state defines the cost function of the QNN:

$$C_t = \frac{1}{n_t}\sum_{k=1}^{n_t} \langle\phi^{\text{out},k}|\mathcal{E}(|\phi^{\text{in},k}\rangle\langle\phi^{\text{in},k}|)|\phi^{\text{out},k}\rangle, \quad t \in \{\text{train}, \text{test}\}. \tag{5.2}$$

It becomes 1 if the network's output precisely matches each of the desired output states and goes to 0 the more they differ. The quantum algorithm to compute the fidelity between two states is described in section 4.3.2.

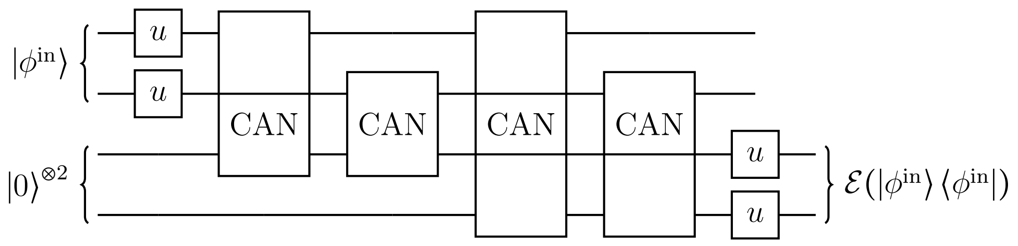

$|\phi^{\text{in}}\rangle$
u
u
CAN
CAN
CAN
CAN
$|0\rangle^{\otimes 2}$
u
u
$\mathcal{E}(|\phi^{\text{in}}\rangle\langle\phi^{\text{in}}|)$

Figure 5.1: The quantum circuit implementing a DQNN with input state $|\phi^{\text{in}}\rangle$.

5.3 Results

5.3.1 Setup

This analysis aims to compare two different QNN architectures, the DQNN (see section 4.3.4) and the QAOA (see section 4.3.5), under the influence of noise. The task of both networks is to learn a unitary transformation of dimension four: $V \in U(4)$ from the four-dimensional states in the training set (5.1). The training and the test set contain $n_{\text{train}} = n_{\text{test}} = 4$ states that are initialised using two qubits. Thus, the networks are constrained to $n_{\text{in}} = n_{\text{out}} = 2$ input and output qubits. A one-layer DQNN suffices as this shallow network already features the characteristics of the quantum perceptron and allows the generalisation to a more extensive network. Therefore, the DQNN considered here has the shape and requires $n_{\text{in}} + n_{\text{out}} = 4$ qubits (see Fig. 5.1). It incorporates $n_p = 3n_{\text{out}} + 3n_{\text{in}}(1 + n_{\text{out}}) = 24$ parameters which are initialised randomly in the range $(0,2\pi]$. The hyperparameters $\eta = 0.5$ and $\epsilon = 0.25$ turned out to be optimal. The QAOA acting on two qubits has to have $n_p = d^2 = 16$ parameters or eight layers and requires two qubits. Its parameters are initialised in the range $[-1,1]$. The hyperparameters $\eta = 0.075$ and $\epsilon = 0.05$ turned out to be optimal. Adding the number of qubits required for the fidelity computation results in an overall circuit that requires six qubits when using the DQNN and four qubits when using the QAOA.

The implementation and execution of the quantum algorithms are carried out using the open-source SDK Qiskit [61] and the quantum devices of IBM [62]. The quantum circuits of both networks are implemented as described in sections 4.3.4 and 4.3.5. These quantum circuits are then transpiled to quantum circuits consisting of basis gates (2.23) to allow their execution using IBM's quantum computers (see appendix B). Mapping the quantum circuit to the H-topology of *ibmq_ casablanca* results in 57 (97) single-qubit gates and 63 (57) CNOT gates for the DQNN (QAOA).

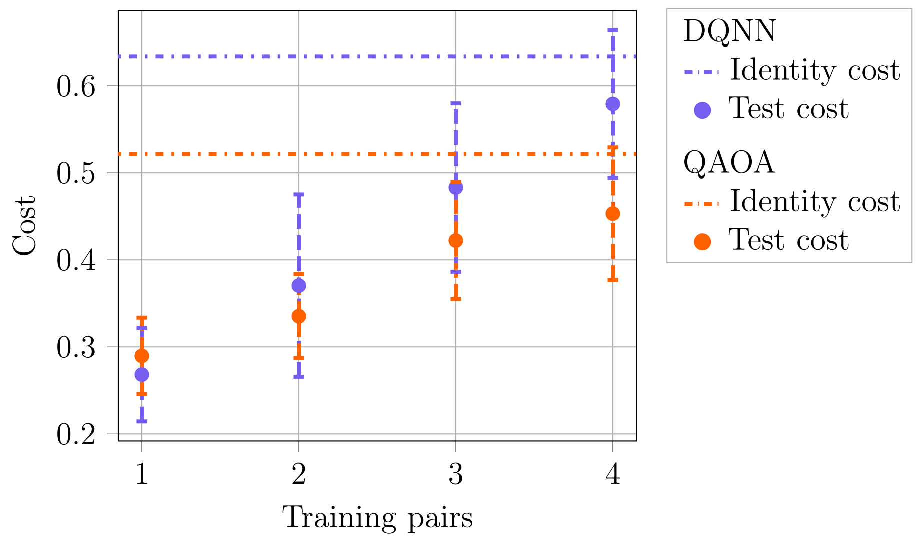


Figure 5.2: Generalisation analysis. The DQNN (two-qubit QAOA) using the simulated *ibmq_ casablanca* for $\epsilon = 0.5$, $\eta = 1.0$ ($\epsilon = 0.15$, $\eta = 0.1$).

5.3.2 Generalisation analysis

The most important capability of a QNN is its ability to generalise the training data to the test data as quantum data, in general, is limited. Generalisation describes the network's ability to learn the correlation between the input and output states of the training set and transfer this knowledge to input states with unknown output. The test cost C_{test} (5.2) quantifies this ability. Each QNN is trained using $n_{\text{SV}} = 1, \ldots, 4$ states of the training set $\mathcal{S}_{\text{train}}$ and at each epoch tested using the $n_{\text{train}} = 4$ states in $\mathcal{S}_{\text{test}}$. Note that the states in $\mathcal{S}_{\text{test}}$ are not used to update the network's parameters. Both QNNs are executed using the simulated *ibmq_ casablanca*, i.e., a simulator incorporating the real-time noise of *ibmq_ casablanca*. Each training is repeated several times to average over different initial network parameters, input states, target unitaries, and noise snapshots.

The results of the analysis are shown in Fig. 5.2. The test and identity costs (see section 4.3.2) are plotted versus the number of training pairs n_{SV}. It can be seen that both networks are capable of generalising the information from the training pairs to the test states. The identity cost is an approximation for the cost of an ideally trained network. The results show that the DQNN's identity cost is higher than the QAOA's. This gives rise to the assumption that the DQNN is less susceptible to gate noise (see appendix D.1). The high identity cost of the DQNN also explains its higher test cost. In general, the test cost of the QNN with a higher identity cost is also higher if their test costs are similar in the noiseless case. The DQNN features only a few low-valued outliers, which make the main contribution to the standard deviation. The QAOA's test cost, however, is uniformly distributed around the mean value. Thus, it can be said that the DQNN generalises the training data with higher reliability.

5.3.3 NISQ device execution

So far, all the executions of the QNNs have been done using simulators. However, to validate our previous results and show the current state of research, this section features the execution of both QNNs using a real quantum computer. Both networks are trained and tested using $n_{\text{train}} = n_{\text{test}} = 4$ states on the seven-qubit device *ibmq_casablanca* (see Fig. 5.3) for 100 epochs.

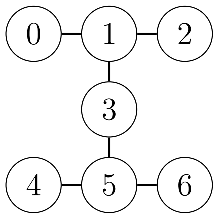


Figure 5.3: The qubit coupling map of *ibmq_casablanca*. It consists of seven qubits and has a quantum volume of 32 [62, 110].

The training, test, and identity costs while training the DQNN and the QAOA using a real quantum computer are shown in Fig. 5.4. In the noiseless case, the training and test cost would increase monotonously. Here, however, both costs seem to follow the identity cost during learning. The path of the identity cost indicates not only statistical fluctuations due to the gate noise but also a noise drift of the quantum device itself. For every epoch, the quantum algorithm has to join the queue of the device. Training one of the QNNs for a hundred epochs took about a month, where one execution of the quantum algorithm took only seconds. Thus, the noise drift of the device can be explained by recalibrations of the device[a]. After midway through the training, both QNNs' training costs exceed their identity costs. This shows their great capability to factor in the quantum computer's noise. Remarkably, both networks can learn the unitary transformation from the training states and generalise the information to the test states despite the high noise levels. This conclusion arises from the high test to identity cost ratio.

5.3.4 Conclusion

This analysis featured the comparison of two very different QNNs, the DQNN and the QAOA. Both QNNs have been implemented as quantum circuits and executed using IBM's simulated and real quantum devices.

Comparing the results of both QNNs reveals that the DQNN can generalise the unitary transformation from the training data better than the QAOA under the influence of noise. Increasing the noise levels also increases this difference because the DQNN's quantum algorithm design is less susceptible to gate noise (see appendix D.1). Both networks have been successfully trained on a real quantum

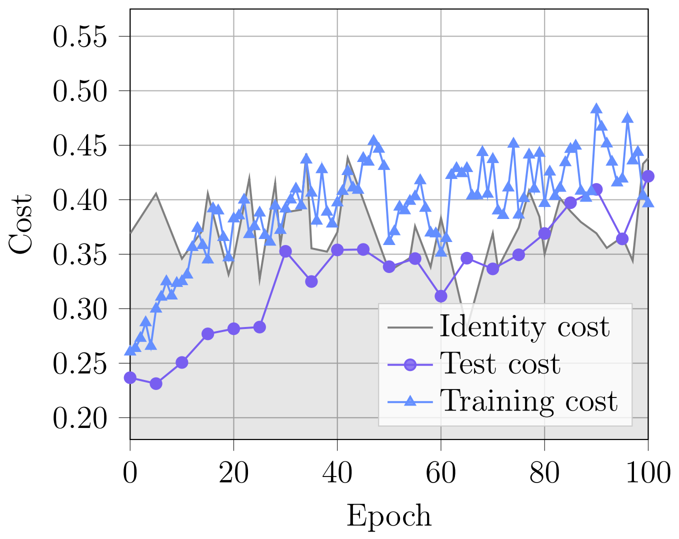
a The *ibmq_casablanca* has been recalibrated at least every week [62].

computer despite its high noise level. However, noise is still the main limiting factor that hinders the QNN from reaching high fidelities.

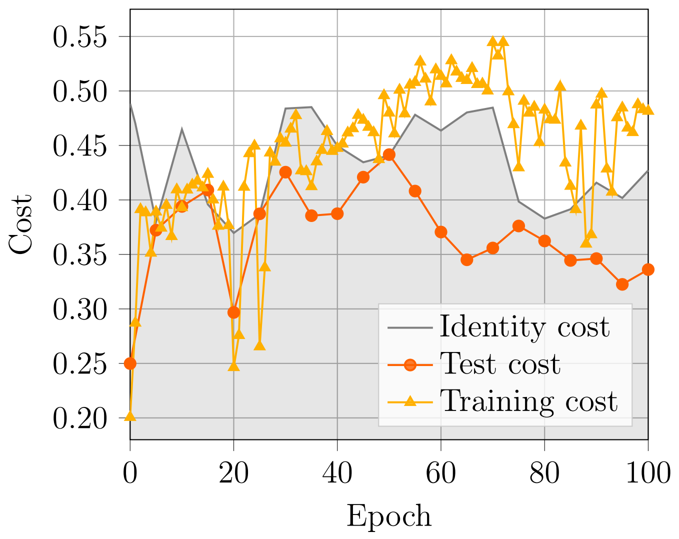
Besides the reduction of noise, the further development of quantum computers holds potential in providing resettable qubits [111]. This would reduce the number of qubits the dissipative QNN requires and thus allow the exploration of deeper networks.

This analysis can be further extended to higher-dimensional unitaries and non-unitary transformations. In chapter 6 the QNNs are trained to learn the graph structure of quantum data where the state pairs themselves are not correlated.

The code is available at `https://github.com/qigitphannover/DeepQuantumNeuralNetworks`.



(a) The DQNN on *ibmq_ casablanca* for $\epsilon = 0.5$ and $\eta = 1.0$.



(b) The two-qubit QAOA on *ibmq_ casablanca* for $\epsilon = 0.15$ and $\eta = 0.1$.

Figure 5.4: Training the DQNN (a) and the QAOA (b) on a real quantum computer. Both networks have been trained for 100 epochs. Each epoch, the training cost is calculated using four training pairs. Every fifth epoch, the test cost is measured using four test pairs, as well as the identity cost

Chapter 6

Training QNNs with Graph-Structured Quantum Data

6.1 Introduction

In the previous analysis (chapter 5), the focus was on teaching two different QNNs the unitary transformation V encoded in the input-output relation of the training states $\{|\phi^{\mathrm{in},k}\rangle, |\phi^{\mathrm{out},k}\rangle = V|\phi^{\mathrm{in}}\rangle\}$. In this analysis, however, the task is to teach the QNNs not only the input-output relation of training states $\{\rho^{\mathrm{in},k}, \rho^{\mathrm{out},k}\}$ but also the correlation between the states $\rho^{\mathrm{out},k}$, which is encoded into a graph.

Naturally, quantum data will always have structure because of the generally structured quantum device. Consider a distributed set of quantum information processors p, which produce an output ρ_p^{out} from input ρ_p^{in}. The output of these processors can be associated with a graph $G(V,E)$ where the vertices are labeled p, and the edges denote the correlations between the processors (e.g., caused by spacial vicinity). These correlations can be measured by their information-theoretical closeness (see section 6.2.1). The QNN's goal is to optimally learn the input-output relations for this distributed set of processors by exploiting the graph structure of the output states.

The learning task, including the quantities that measure the QNNs success, are described in section 6.2. The results of training QNNs using graph-structured quantum data are presented in section 6.3.

The results of this chapter are based on the work of [63].

6.2 The Learning Task

For simplicity reasons, it is assumed that the training and test sets consist of only pure states. The complete set of states is given by

$$\mathcal{S} = \left\{ \left(|\phi^{\text{in},k}\rangle , |\phi^{\text{out},k}\rangle \right) \right\}_{k=1}^{n} . \tag{6.1}$$

The training set consists of n_{SV} supervised (labeled) vertices and $n_{\text{USV}} = n - n_{\text{SV}}$ unsupervised (unlabeled) vertices:

$$\mathcal{S}_{\text{train}} = \left\{ \left(|\phi^{\text{in},1}\rangle , |\phi^{\text{out},1}\rangle \right), \dots, \left(|\phi^{\text{in},n_{\text{SV}}}\rangle , |\phi^{\text{out},n_{\text{SV}}}\rangle \right), |\phi^{\text{in},n_{\text{SV}}+1}\rangle , \dots, |\phi^{\text{in},n}\rangle \right\} . \tag{6.2}$$

The n_{SV} supervised states are used to teach the QNN their input-output relations, while the additional n_{USV} unsupervised states are used to exploit the graph structure (see section 6.2.1). The n_{USV} unsupervised (unlabeled) input states and their corresponding output states are used to measure the network's generalisation capability and form the test set:

$$\mathcal{S}_{\text{test}} = \left\{ \left(|\phi^{\text{in},n_{\text{SV}}+1}\rangle , |\phi^{\text{out},n_{\text{SV}}+1}\rangle \right), \dots, \left(|\phi^{\text{in},n}\rangle , |\phi^{\text{out},n}\rangle \right) \right\} . \tag{6.3}$$

Note that there is a significant difference in the definition of the training (6.2) and test sets (6.3) compared to the previous analysis (see section 5.2). Here, the training and test set emerges from a single set (6.1), whereas previously, they were independent (see section 5.2).

6.2.1 Cost functions

The supervised cost function

The supervised cost function measures the information distance between the network's output $\mathcal{E}(|\phi^{\text{in},k}\rangle \langle\phi^{\text{in},k}|)$ and the corresponding desired output $|\phi^{\text{out},k}\rangle$. The information distance is measured by the fidelity (4.5). The supervised cost function is the average fidelity of all network outputs and supervised training states:

$$C_{\text{SV}} = \frac{1}{n_{\text{SV}}} \sum_{k=1}^{n_{\text{SV}}} \langle\phi^{\text{out},k}| \mathcal{E}(|\phi^{\text{in},k}\rangle \langle\phi^{\text{in},k}|) |\phi^{\text{out},k}\rangle . \tag{6.4}$$

The graph-based cost function

The graph structure $G = (V,E)$ with vertices V and edges E of the output states is encoded in the adjacency matrix A, with components

$$[A]_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases} . \tag{6.5}$$

A pair of adjacent vertices $(i,j) \in E$ should be mapped close to each other. Two pairs of states $|\phi^{\text{out},i}\rangle$ and $|\phi^{\text{out},j}\rangle$ are defined to have adjacent vertices (i,j) if their fidelity $\| \langle \phi^{\text{out},i}|\phi^{\text{out},j}\rangle \|^2$ is above a certain threshold. The fidelity (4.5) could also be used to measure the information distance between two network outputs $\mathcal{E}(|\phi^{\text{in},k}\rangle\langle\phi^{\text{in},k}|)$ and $\mathcal{E}(|\phi^{\text{in},l}\rangle\langle\phi^{\text{in},l}|)$. However, the network's output states are not generally pure. Calculating the fidelity of two mixed states is too computationally complex, especially in terms of quantum algorithms. Thus, the Hilbert-Schmidt distance (4.7) is exploited (see section 4.3.2 for the quantum algorithm). The graph-based cost function measures the network's success in reproducing the graph structure with adjacency matrix A. It is defined as the sum of Hilbert-Schmidt distances of all adjacent vertices:

$$C_{\text{G}} = \sum_{k,l\in V,k\neq l} [A]_{kl} d_{\text{HS}} \left(\mathcal{E}(|\phi^{\text{in},k}\rangle\langle\phi^{\text{in},k}|), \mathcal{E}(|\phi^{\text{in},l}\rangle\langle\phi^{\text{in},l}|)\right) \tag{6.6}$$

where $V = \{1,\dots,n\}$. This function is minimised when the network maps states of adjacent vertices to information-theoretically close density matrices.

The training cost

The training cost, which is exploited to train the QNN, is given by the combination of the supervised and graph-based cost functions:

$$C_{\text{SV+G}} = C_{\text{SV}} + \gamma C_{\text{G}} \tag{6.7}$$

where the graph-based cost is scaled using a Lagrange multiplier $\gamma \leq 0$. Thus, the training cost is maximised when the network not only maps the supervised input states $\{|\phi^{\text{in},k}\rangle\}_{k=1}^{n_{\text{SV}}}$ to the desired output states $\{|\phi^{\text{out},k}\rangle\}_{k=1}^{n_{\text{SV}}}$, but also captures the graph structure of the vertices $\{|\phi^{\text{out},k}\rangle\}_{k=1}^{n}$. In the case of $1 \leq n_{\text{SV}} < n$ (semi-supervised learning), the graph-based cost function serves as an interpolation between supervised vertices.

The test cost

The test cost is defined similarly to the supervised cost function:

$$C_{\text{USV}} = \frac{1}{n_{\text{USV}}} \sum_{k=n_{\text{SV}}+1}^{n} \langle\phi^{\text{out},k}|\mathcal{E}(|\phi^{\text{in},k}\rangle\langle\phi^{\text{in},k}|)|\phi^{\text{out},k}\rangle . \tag{6.8}$$

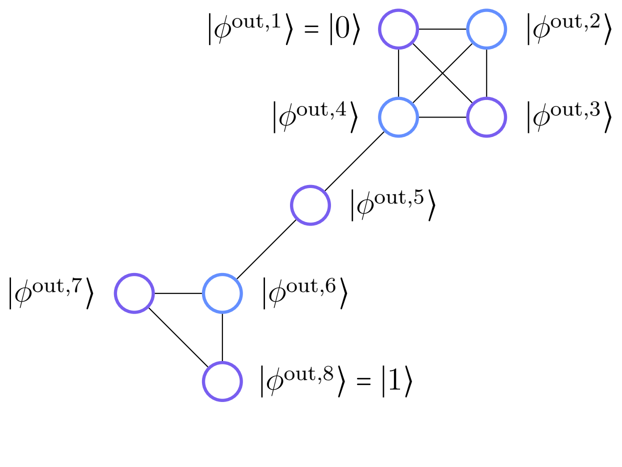
It averages the fidelities of the network's outputs $\mathcal{E}(|\phi^{\text{in},k}\rangle\langle\phi^{\text{in},k}|)$ and the desired unsupervised outputs $|\phi^{\text{out},k}\rangle$. The testing cost provides a measure of the network's generalisation capabilities as the states of the test set are unknown to the network.

6.2.2 Example A: connected clusters

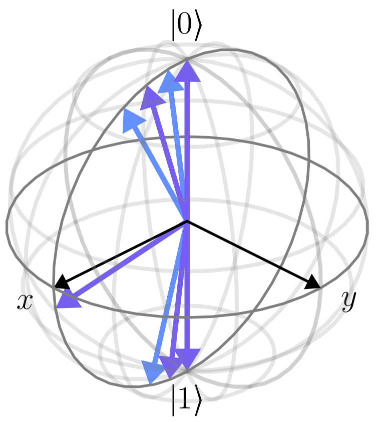
One important property of the training cost function is that adjacent states are mapped close together. This makes it especially applicable to clustered training data. Thus, this example features two clusters that are connected by a single vertex. The set of states is given by

$$\mathcal{S}^{\mathrm{CC}} = \left\{ \left(|\phi^{\mathrm{in},k}\rangle, |\phi^{\mathrm{out},k}\rangle = \frac{([\boldsymbol{m}]_k - 1)|1\rangle + (2n - 1 - [\boldsymbol{m}]_k)|0\rangle}{\|([\boldsymbol{m}]_k - 1)|1\rangle + (2n - 1 - [\boldsymbol{m}]_k)|0\rangle\|} \right) \right\}_{k=1}^{n} \tag{6.9}$$

where $[\boldsymbol{m}]_k$ denotes the kth component of $\boldsymbol{m} = (1, \ldots, \frac{n}{2}, n+1, \frac{3n}{2}+1, \ldots, 2n-1)$ and $|\phi^{\mathrm{in},k}\rangle$ are randomly generated $\log_2(n)$-qubit states. Note that this definition is only valid if $n \geq 4$ and n is even. In the following analysis, the case of $n = 8$ is considered. The input states are initialised on three qubits. The output states are assumed to be of dimension two and thus are initialised on one qubit. The graph of the output states is constructed from the adjacency matrix (6.5), where the threshold is chosen to be 0.65. The graph and the Bloch sphere representation of the output states are shown in Fig. 6.1.


$|\phi^{\mathrm{out},1}\rangle = |0\rangle$
$|\phi^{\mathrm{out},2}\rangle$
$|\phi^{\mathrm{out},4}\rangle$
$|\phi^{\mathrm{out},3}\rangle$
$|\phi^{\mathrm{out},5}\rangle$
$|\phi^{\mathrm{out},7}\rangle$
$|\phi^{\mathrm{out},6}\rangle$
$|\phi^{\mathrm{out},8}\rangle = |1\rangle$

(a) The graph for the adjacency matrix with threshold 0.65.



(b) The Bloch sphere representation of $\{|\phi^{\mathrm{out},k}\rangle\}_{k=1}^{8}$.

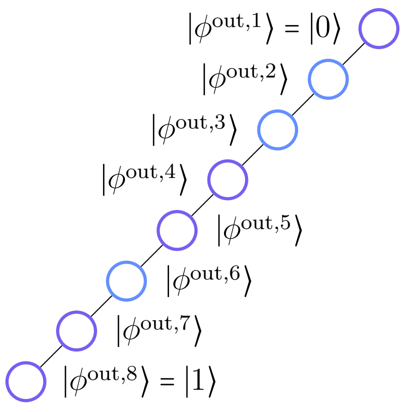
Figure 6.1: The connected clusters output states (6.9) for $n = 8$ and $n_{\mathrm{SV}} = 3$. Supervised states are colored in blue. Unsupervised states are purple.

6.2.3 Example B: connected line

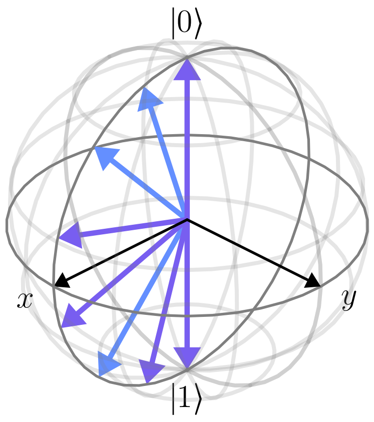
Another interesting example is the connected line training set

$$\mathcal{S}^{\mathrm{CL}} = \left\{ \left(|\phi^{\mathrm{in},k}\rangle, |\phi^{\mathrm{out},k}\rangle = \frac{(k-1)|1\rangle + (n-k)|0\rangle}{\|(k-1)|1\rangle + (n-k)|0\rangle\|} \right) \right\}_{k=1}^{n} \tag{6.10}$$

where $|\phi^{\text{in},k}\rangle$ are randomly generated $\log_2(n)$-qubit states, $n \geq 2$, and $|\phi^{\text{out},k}\rangle$ are one-qubit states. The graph-based cost function maps adjacent states close together. Suppose the training set contains multiple supervised output states. Then, the graph-based cost function is able to interpolate between the supervised output states as it favors unsupervised states which are "in-between" their adjacent neighbors. As in section 6.2.2, the case $n = 8$ is considered. The input states are randomly generated three-qubit states. The adjacency matrix is computed using a threshold of 0.9. The graph and the Bloch sphere representation of the output states are shown in Fig. 6.2.


$|\phi^{\text{out},1}\rangle = |0\rangle$
$|\phi^{\text{out},2}\rangle$
$|\phi^{\text{out},3}\rangle$
$|\phi^{\text{out},4}\rangle$
$|\phi^{\text{out},5}\rangle$
$|\phi^{\text{out},6}\rangle$
$|\phi^{\text{out},7}\rangle$
$|\phi^{\text{out},8}\rangle = |1\rangle$

(a) The graph for the adjacency matrix with threshold 0.9.



(b) The Bloch sphere representation of $\{|\phi^{\text{out},k}\rangle\}_{k=1}^{8}$.

Figure 6.2: The connected line output states (6.10) for $n = 8$ and $n_{\text{SV}} = 3$. Supervised states are colored in blue. Unsupervised states are purple.

6.3 Results

6.3.1 Setup

This analysis features the training of QNNs using graph-structured quantum data. The focus here is mainly on the unique learning task and, thus, will only feature noiseless simulations[a]. However, this enables the consideration of the DQNN_U[b].

a The simulations are performed using the *statevector simulator*. This is a unique simulator which is directly computing the fidelity of the output states using (4.5). In the previous chapter, this fidelity computation has been approximated using the destructive swap test (see section 4.3.2).

b Chapter 5 focuses on the comparison of two networks under the influence of noise. Here the DQNN_U was not considered because its decomposition into basis gates results in a large circuit depth. Additionally, the large number of parameters is making the training of the DQNN_U very slow. The efficient execution of noise simulations and real quantum computers require a network with low circuit depth and few parameters (see section 4.3.2).

The training data from sections 6.2.2 and 6.2.3 are exploited. These sets of $n = 8$ states contain pairs of three-qubit input states and one-qubit output states. Here, one-layer QNNs are chosen with $n_{\text{in}} = 3$ input qubits and $n_{\text{out}} = 1$ output qubit. The DQNN_{U} has the shape and requires $n_{\text{in}} + n_{\text{out}} = 4$ qubits. It has $n_p = n_{\text{out}} 4^{n_{\text{in}}+1} = 265$ parameters initialised randomly in $(0,2\pi]$. The same applies to the DQNN_{CAN}, except that it has only $n_p = 3n_{\text{out}} + 3n_{\text{in}}(1 + n_{\text{out}}) = 21$ parameters. The QAOA has eight layers and 16 parameters randomly initialised in $[-1,1]$. It only acts on $\max\{n_l\}_{l=\text{in}}^{\text{out}} = 3$ qubits (see section 4.3.5). The calculation of the Hilbert-Schmidt distance (4.7) requires the parallel execution of the same QNN for two different input states. Therefore the total number of qubits required for the training with the graph-based cost function (6.6) is twice the number of qubits required for one QNN evaluation. Training the DQNN_{U} and the DQNN_{CAN} requires eight qubits. Training the QAOA requires six qubits. The training without the graph-based cost function can be done using only five qubits for the DQNN_{U} and the DQNN_{CAN} and four qubits for the QAOA. The QNNs are trained using the Adam optimiser (see section 3.3.5) with $\eta = 0.05$ and $\epsilon = 0.05$. It has shown improvements in training classical and quantum machine learning models [38, 87].

6.3.2 Learning graph-structured quantum data

This analysis features the training of the DQNN_{U}, DQNN_{CAN}, and the QAOA with the connected cluster (section 6.2.2) and connected line (section 6.2.3) training data to show the impact of the graph-based cost function and to validate its definition.

Example A: connected clusters

The connected clusters training data is defined in section 6.2.2. Here, the number of supervised training pairs is fixed to $n_{\text{SV}} = 3$. The one-qubit output training states are shown in Fig. 6.1. The three-qubit input training states are randomly generated via a Gaussian distribution. The DQNN_{U}, DQNN_{CAN}, and QAOA are trained using $\gamma = 0$ (training using only the supervised cost function) and $\gamma = -0.5$ (training using the supervised and graph-based cost function).

The training and test cost while training the DQNN_{U} using $\gamma = 0$ and $\gamma = -0.5$ are shown in Fig. 6.3. At epoch zero, the training cost for $\gamma = -0.5$ is very small because the graph-based cost function is very large due to the randomly distributed network output states (the input states and the network parameters are initialised randomly). Thus, during the first few epochs, the graph-based cost function dominates the network's training. In this phase, the test cost of $\gamma = -0.5$ grows higher than the test cost of $\gamma = 0$. This shows the remarkable property of the graph-based cost function to utilise the graph information for a better generalisation. The generalisation of the QNN, which is trained with only the supervised cost function, is improving only slightly during the training. In fact, after some epochs, the test cost shows symptoms of overfitting because the test cost starts shrinking while the training cost is growing (see section 3.3.5). The test cost of $\gamma = -0.5$, however, shows no such signs.

The test costs while training the $\mathrm{DQNN_U}$, $\mathrm{DQNN_{CAN}}$, and QAOA with ($\gamma = -0.5$) and without ($\gamma = 0$) graph-based cost function is plotted in Fig. 6.4. The QNNs trained using $\gamma = -0.5$ reach higher test costs than those trained with $\gamma = 0$. The graph-based cost function improves the generalisation of all QNNs. There are, however, differences in the test costs of the different QNNs. The $\mathrm{DQNN_U}$ outperforms both QNNs in terms of generalisation. The $\mathrm{DQNN_{CAN}}$'s test cost is lower than the $\mathrm{DQNN_U}$'s but higher than the QAOA's. In the case of $\gamma = 0$, the test costs of all QNNs converge to about 0.5.

Example B: connected line

The results of training the QNNs using the connected line training data (section 6.2.3) is similar to the previous one. Again, $n_{\mathrm{SV}} = 3$ supervised training pairs are chosen. The exact supervised and unsupervised states are shown in Fig. 6.2. The input states are randomly generated three-qubits states. The one-qubit output states are computed using (6.10). The connected line training data is used to test the QNNs' ability to interpolate between supervised states. Again, all three QNNs (the $\mathrm{DQNN_U}$, $\mathrm{DQNN_{CAN}}$, and QAOA) are trained using only the supervised cost function ($\gamma = 0$) and using the entire training cost function ($\gamma = -1$).

The training and test cost while training the $\mathrm{DQNN_U}$ are shown in Fig. 6.5. The results of this analysis are very similar to the training using the connected cluster training data. Training the $\mathrm{DQNN_U}$ $\gamma = 0$ results in a perfect training cost but shows symptoms of overfitting as the test cost decreases during the last epochs. As expected, the training using $\gamma = -1$ results in a better test cost and thus also proves the ability of the graph-based cost function to interpolate between the supervised output states. The increase of the test cost is the fastest at the beginning of the training, where the parameter update steps are mainly in directions that reduce the graph-based cost function. The training using $\gamma = -1$ shows no signs of overfitting.

The test costs of all three different networks while training them using $\gamma = 0$ and $\gamma = -1$ for 500 epochs are plotted in Fig. 6.6. Again, the training using the graph-based cost function improves the generalisation of all QNNs. The $\mathrm{DQNN_U}$ and the $\mathrm{DQNN_{CAN}}$ perform significantly better than the QAOA.

The comparison of training the three different QNNs using the graph-based cost function shows a very big difference in the resulting test cost (see Figs. 6.4 and 6.6). This difference can be explained by the networks' implementation. A $\mathrm{DQNN_U}$ consists simply of one universal unitary operator (see section 4.3.3). This universality helps the QNN to generalise as it can represent every possible

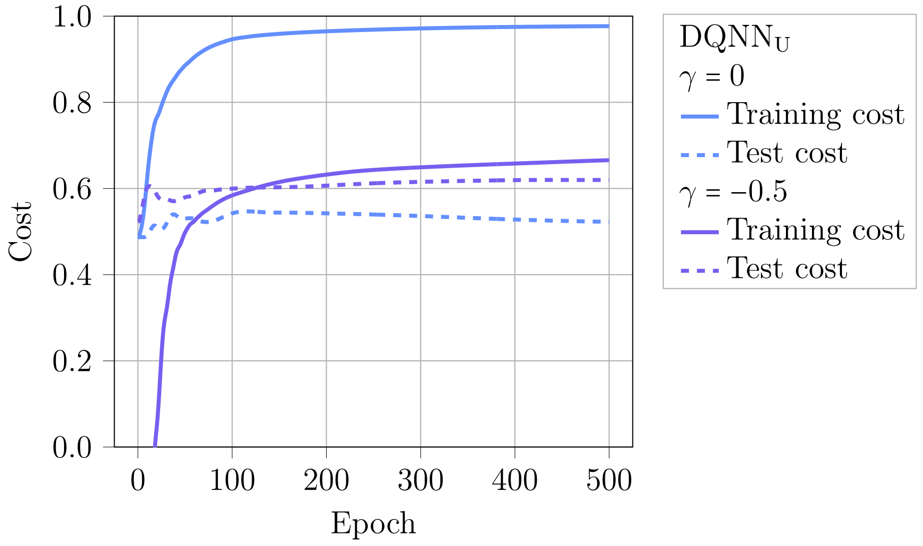


Figure 6.3: The training and test cost while training the $\mathrm{DQNN_U}$ for 500 epochs using the connected clusters training data shown in Fig. 6.1. It consists of $n_{\mathrm{SV}} = 3$ supervised states and randomly generated input states. The training and test costs for $\gamma = 0$ and $\gamma = -0.5$ are averaged for five different runs.

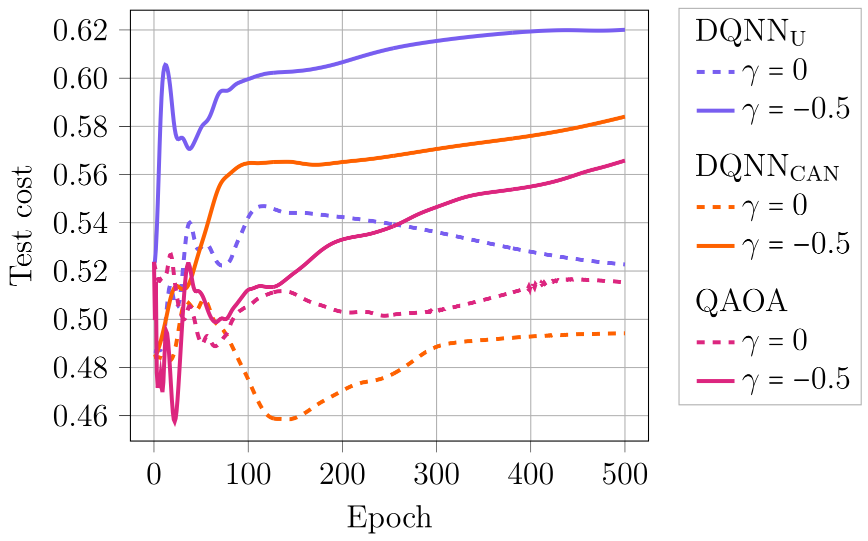


Figure 6.4: The test cost while training the $\mathrm{DQNN_U}$, the $\mathrm{DQNN_{CAN}}$, and the QAOA for 500 epochs using the connected clusters training data shown in Fig. 6.1. It consists of $n_{\mathrm{SV}} = 3$ supervised states and randomly generated input states. The training and test costs for $\gamma = 0$ and $\gamma = -0.5$ are averaged for five different runs for the $\mathrm{DQNN_U}$ and for ten different runs for the $\mathrm{DQNN_{CAN}}$ and the QAOA.

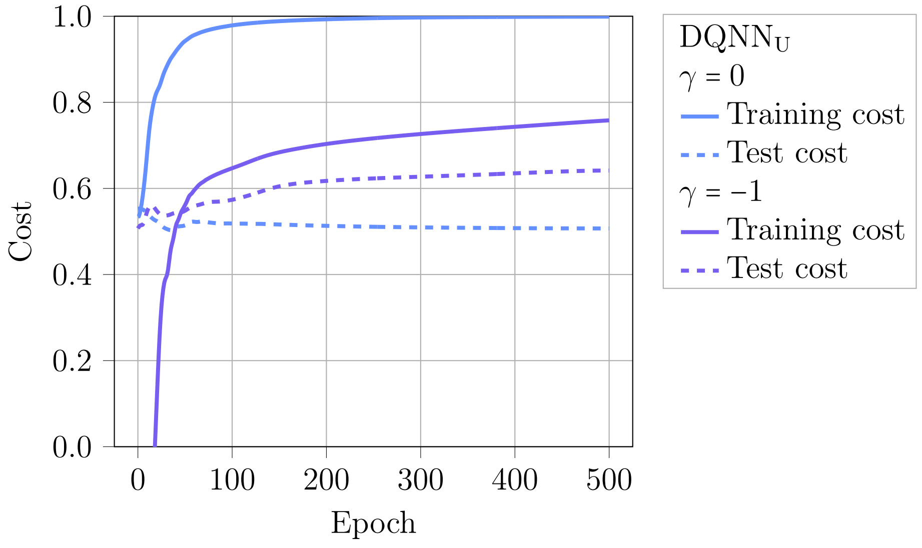


Figure 6.5: The training and test cost while training the DQNN$_U$ for 500 epochs using the connected line training data shown in Fig. 6.2. It consists of $n_{SV} = 3$ supervised states and randomly generated input states. The training and test costs for $\gamma = 0$ and $\gamma = -1$ are averaged for five different runs.

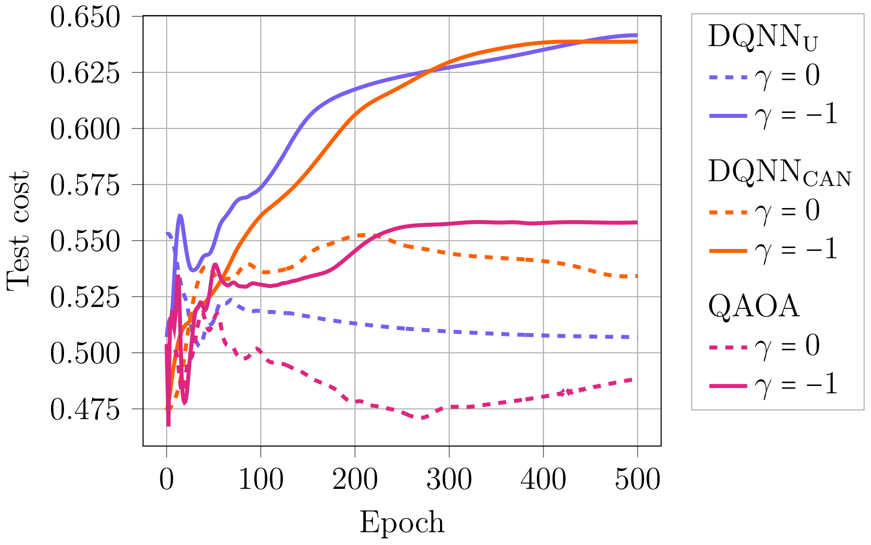


Figure 6.6: The test cost while training the DQNN$_U$, the DQNN$_{CAN}$, and the QAOA for 500 epochs using the connected line training data shown in Fig. 6.2. It consists of $n_{SV} = 3$ supervised states and randomly generated input states. The training and test costs for $\gamma = 0$ and $\gamma = -1$ are averaged for five different runs for the DQNN$_U$ and for ten different runs for the DQNN$_{CAN}$ and the QAOA.

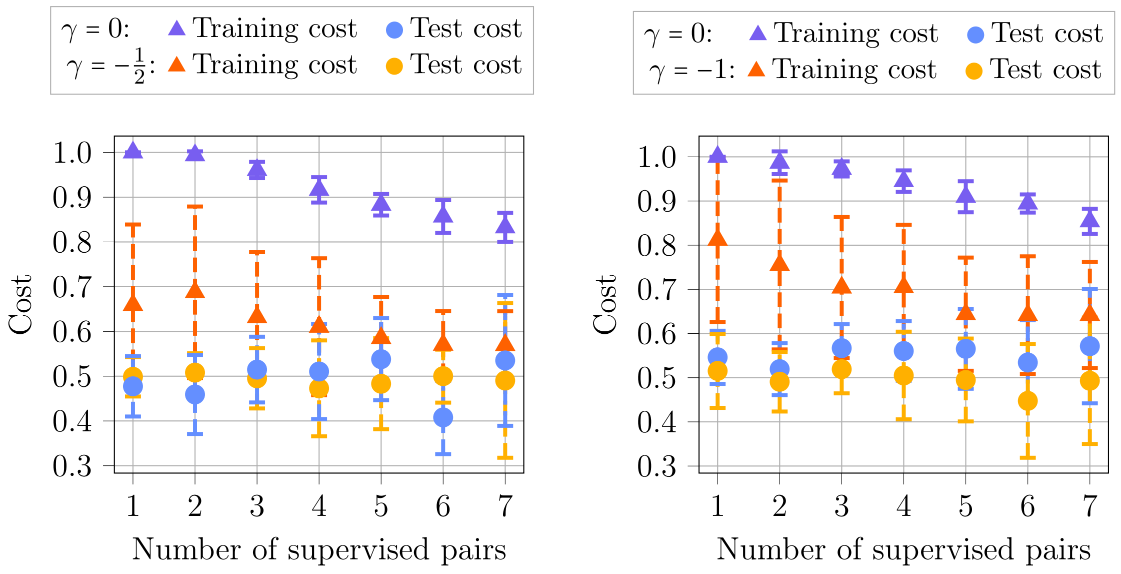
unitary transformation. The $\mathrm{DQNN}_{\mathrm{CAN}}$ is composed of single-qubit u gates and two-qubit CAN gates (see section 4.3.4). The parameterised gates allow the reduction of the network parameters and, thus, the operation on a real quantum computer (see chapter 5). However, the composition of these gates is, in general, not universal. Therefore, the $\mathrm{DQNN}_{\mathrm{CAN}}$'s generalisation is slightly worse than the $\mathrm{DQNN}_{\mathrm{U}}$'s. The architecture of the QAOA is not well defined for learning non-unitary transformations. The number of layers is chosen such that the total number of parameters matches the number of components of the desired transformation (see appendix C.4). Here, the $2^3 \times 2^1$ matrix transformation leads to 16 parameters. A universality analysis has shown that the eight-layer QAOA is slightly worse in learning random state pairs than the ✂ $\mathrm{DQNN}_{\mathrm{CAN}}$. Thus, the QAOA is slightly worse in generalising the supervised states.

6.3.3 Generalisation analysis

Besides training the QNNs using specific pairs of supervised and unsupervised training states, it is of particular interest to analyse the QNNs' generalisation capabilities using arbitrarily supervised and unsupervised training states. With this, the universality of the improvements catalysed by the graph-based cost function can be tested. This analysis features the training of the $\mathrm{DQNN}_{\mathrm{CAN}}$ using $n_{\mathrm{SV}} = 1,\dots,7$ supervised training states randomly selected from the training set. Note that the $\mathrm{DQNN}_{\mathrm{U}}$ and the QAOA are not considered here[a]. This iteration is repeated ten times to average over different supervised output states, input states, and initial parameters.

The results of training the $\mathrm{DQNN}_{\mathrm{CAN}}$ for $n_{\mathrm{SV}} = 1,\dots,7$ are shown in Fig. 6.7 for the connected clusters and connected line training data, respectively. In the case of the connected line training data, the graph-based cost function improves the network's generalisation for every number of supervised states. This finding shows the graph-based cost function's remarkable capability to improve the generalisation no matter which states are supervised or unsupervised. In the case of the connected clusters training data, the graph-based cost function does not continuously improve the network's generalisation for all numbers of supervised states. However, the individual results reveal that the test costs of $\gamma = -0.5$ have a few significant outliers for $n_{\mathrm{SV}} = 3,4,5,7$. These outliers can partly be explained by looking at the particular supervised and unsupervised states. The poor test costs of training the QNN using the connected cluster training data and $\gamma = -0.5$ occurs mainly when only one cluster contains supervised states. Then, the graph-based cost function hinders the generalisation as the network is trained to map all the states to this one cluster.

[a] The $\mathrm{DQNN}_{\mathrm{U}}$ does not allow such an iterative analysis as its execution is too costly. The results of the QAOA are not shown due to its small test cost (see Figs. 6.4 and 6.6).



(a) The connected clusters training data (see section 6.2.2).

(b) The connected line training data (see section 6.2.3).

Figure 6.7: The training and test cost after training the DQNN_{CAN} for 500 epochs, $n_{\text{SV}} = 1,\ldots,7$ randomly chosen supervised states, and randomly generated input states. Each data point is averaged over ten runs.

There are additional reasons why the difference between the test costs for $\gamma = 0$ and $\gamma < 0$ are so small. These will be discussed in the following section.

6.3.4 Improving the generalisation

Comparing the results of the previous sections 6.3.2 and 6.3.3 with the results of [63] (training the QNN using quantum backpropagation (see appendix C.1.1)), reveals that the QNNs trained on a quantum computer do not reach as high test costs as when trained via quantum backpropagation. This raises the question of whether the QNNs themselves are not universal enough or whether the classical optimiser is not suitable for training graph-structured quantum data. This question can be answered by looking at the QNN's output states after and, more importantly, during the training via the classical optimiser and quantum backpropagation.

The network's output states after training a DQNN_{CAN} for 500 epochs using the connected clusters training data from Fig. 6.1 are shown in Fig. 6.8. The individual states can hardly be distinguished as they are all mapped closely together on the Bloch sphere. Analysing the evolution of the network's output states while training a DQNN_{CAN} using the connected clusters training data and $\gamma = -0.5$ reveals that the states, which are initially widely spread apart on the Bloch sphere, are mapped closely together just after a few training epochs. During these first training epochs, the graph-based cost function dominates the cost function's gradient as its value outweighs the supervised cost function. Therefore,

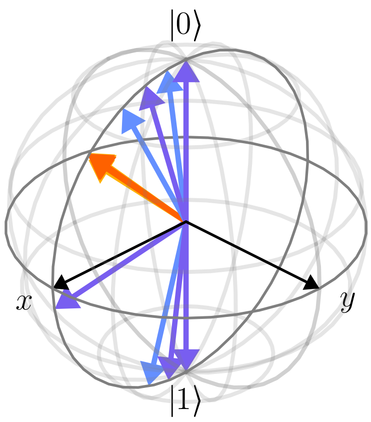


Figure 6.8: The DQNN$_{\text{CAN}}$'s output states (supervised states, unsupervised states) after training for 500 epochs using the connected clusters training data (supervised states, unsupervised states) from Fig. 6.1 and randomly generated three-qubit input states.

the parameters are updated such that mainly the graph-based cost function is decreased. The graph-based function is small if the information-theoretical distance between adjacent output states is small. Thus, in the first training epochs, the network maps stats of adjacent vertices onto a single point on the Bloch sphere. The point is chosen to result in an optimal supervised training cost (to increase the overall training cost) and thus, is located in-between the supervised output states. In summary, the graph-based cost function outweighs the supervised cost function as it is not normalised. Thus, the network parameters run into a local minimum right at the beginning of training. Normalising the graph-based cost function is not trivial as the individual Hilbert-Schmidt distances are not normalised in contrast to the fidelity.

Two possible workarounds help to avoid these local minima at the beginning of training. The method which came out best is presented here. Another method can be found in appendix E.1.

The delayed normalised graph-based cost function

One way to improve the networks' generalisation is by introducing a *delayed normalised graph-based cost function*. For the first few epochs, the network is only trained using the supervised cost function. During this phase, the network learns to correctly map the supervised input states to the corresponding output states. Note that before, the graph-based cost function was dominant during the first epochs. Swapping this importance helps to avoid local minima at the start of training. Afterwards, the graph-based cost function is added to the supervised cost function where the γ is chosen such that $\gamma C_{\text{G}} \in [0,\frac{1}{2}]$. This is achieved by adjusting the γ's definition:

$$\gamma \mapsto \tilde{\gamma}(x_G) = -\frac{C_G(\text{epoch} = x_G)}{2} \tag{6.11}$$

where x_G is the epoch where the graph-based cost function is activated. γ is set constant for all epochs greater than x_G. This way, the normalisation of the graph-based cost function is achieved. Of course, this is linked to the assumption that the graph-based cost function stays below its initial value $C_G(\text{epoch} = x_G)$ with increasing training epochs.

The results of training the DQNN$_U$, the DQNN$_{CAN}$, and the QAOA for 500 epochs with $\gamma = 0$ ($C_{\text{train}} = C_{SV}$), $\gamma = -0.5$ and $\gamma = -1$ ($C_{\text{train}} = C_{SV} + \gamma C_G$), and $\gamma = \tilde{\gamma}(250)$ ($C_{\text{train}} = C_{SV} + \tilde{\gamma}(250) C_G$) are shown in Fig. 6.9. Note that except for the latter, the results are taken from the previous analysis (see Figs. 6.4 and 6.6) to compare the constant γ with $\tilde{\gamma}(250)$. After training the QNNs using only the supervised cost for 250 epochs, the graph-based cost function is added with $\gamma = \tilde{\gamma}(250)$. The transition from $\gamma = 0$ to $\tilde{\gamma(250)}$ at the 250th training epoch is visible in the plot of the test cost as it significantly grows after the 250th training epoch. This again shows the remarkable capability of the graph-based cost function to improve the QNNs' generalisation. Both plots, Figs. 6.9a and 6.9b, show that the delayed normalised graph-based cost function improves the QNNs' generalisation even further than for constant γ. For the connected clusters training data, the DQNN$_{CAN}$ and the QAOA trained using $\tilde{\gamma}(250)$ reach higher test costs than the DQNN$_U$ trained with a constant γ. The DQNN$_U$'s test cost for $\tilde{\gamma}(250)$ even surpasses the test cost of the QNN trained with quantum backpropagation. The training using the connected line training data shows similar results. The test cost of the DQNN$_U$ and the QAOA is significantly improved for $\tilde{\gamma}(250)$.

To test the universality of the previous results, the DQNN$_{CAN}$ is trained for $n_{SV} = 1, \ldots, 7$ randomly selected supervised states in the same manner as for constant γ (see Fig. 6.7). For each n_{SV}, the DQNN$_{CAN}$ is trained for 500 epochs using $\gamma = \tilde{\gamma}(250)$ and the connected clusters and connected line training data. This is repeated ten times to average over different supervised output states, input states, and initial parameters.

The resulting test costs after training the DQNN$_{CAN}$ with $n_{SV} = 1, \ldots, 7$ using $\tilde{\gamma}(250)$ are plotted alongside with $\gamma = 0$ and $\gamma = -0.5$ ($\gamma = -1$) for the connected clusters (connected line) training data in Fig. 6.10. The test costs for the constant γs are taken from the previous analysis to compare them to $\tilde{\gamma}(250)$. Both plots, Figs. 6.10a and 6.10b, show a significant improvement of the test cost resulting from training the DQNN$_{CAN}$ using $\tilde{\gamma}(250)$ instead of $\gamma \leq 0$ for almost all numbers of supervised states. For the connected clusters training data, the graph-based cost function improved the generalisation of the QNN only for a few numbers of supervised states (see Fig. 6.7a). The $\tilde{\gamma}(250)$, however, shows an improvement for almost every n_{SV} (see Fig. 6.10a). Training the DQNN$_{\mathrm{CAN}}$ using the connected line training data and $\gamma = -1$ improved the generalisation for arbitrary supervised states (see Fig. 6.7b). The training using the $\tilde{\gamma}(250)$ further enhances this improvement resulting in an even better generalisation (see Fig. 6.10b).

6.3.5 Conclusion

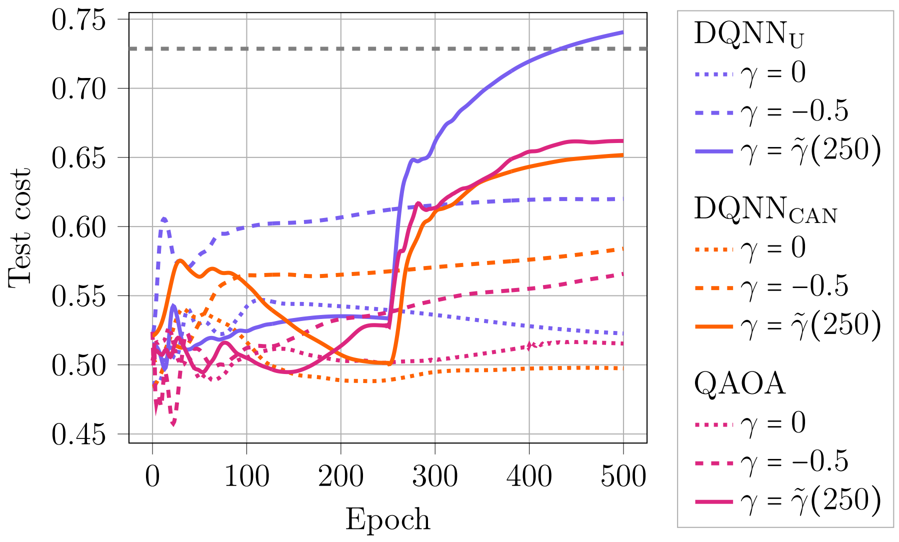
This analysis featured the training of the DQNN$_{\mathrm{U}}$, the DQNN$_{\mathrm{CAN}}$, and the QAOA using graph-structured quantum data. All QNNs have been trained using the connected cluster and connected line training data and utilising the supervised cost function ($\gamma = 0$) and the training cost function including the graph-based cost function ($\gamma < 0$). Additionally, the DQNN$_{\mathrm{CAN}}$ has been trained for arbitrary supervised states.

The comparison of training the QNNs with $\gamma = 0$ and $\gamma < 0$ has shown that the graph-based cost function generally improves the generalisation of the QNN for the specific training data shown in Figs. 6.1 and 6.2. Iterating through randomly chosen supervised states revealed that the graph-based cost function is also practical for arbitrary supervised states. This shows its remarkable capability of utilising the training state's graph structure to enhance the QNN's generalisation capability. This improvement is further increased by adjusting the scaling factor of the graph-based cost function to avoid local minima and equalising the influence of the competing cost functions.

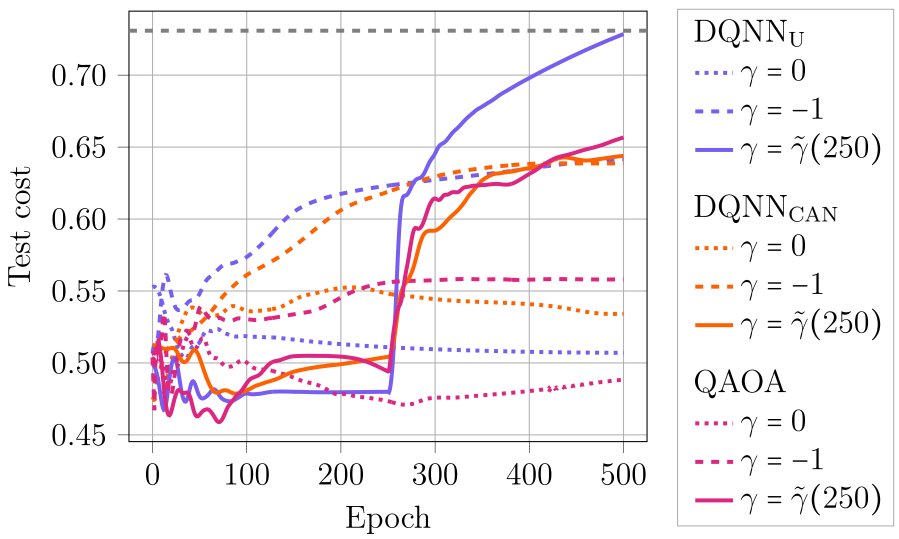
Comparing the results of this analysis with the results of training a QNN using backpropagation [63] shows that classically training the QNN's quantum algorithm using graph-structured quantum data can compete with the initially defined QNN. Training the variational algorithms for specifically chosen supervised states resulted in a comparable test cost. However, it has to be noted that the analysis for arbitrary supervised states did not result in similar test costs [63].

Further analyses could feature different optimisation methods, cost functions, or quantum algorithms to close the cap to the findings of [63]. Additionally, this analysis can be extended to different graph-structured quantum data.

The code is available at `https://github.com/qigitphannover/DeepQuantumNeuralNetworks`.

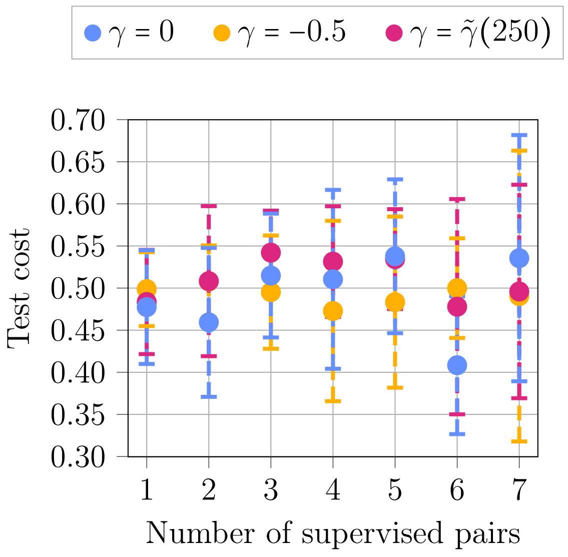


(a) The connected clusters training data (see section 6.2.2).

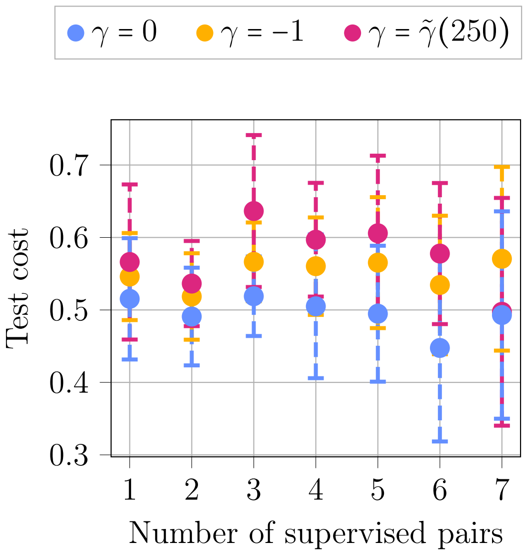


(b) The connected line training data (see section 6.2.3).

Figure 6.9: The test cost while training the DQNN$_U$, the DQNN$_{CAN}$, and the QAOA for 500 epochs using the supervised cost function ($\gamma = 0$), the untouched training cost ($\gamma = -0.5$ and $\gamma = -1$), and the training cost including the delayed normalised graph-based cost function ($\gamma = \tilde{\gamma}(250)$). Additionally the test cost after training the QNN using the quantum backpropagation algorithm (see [63]) is marked as - - -.



(a) The connected clusters training data (see section 6.2.2).



(b) The connected line training data (see section 6.2.3).

Figure 6.10: Training the DQNN_{CAN} using the delayed normalised graph-based cost function for different number of supervised states.

Chapter 7

Conclusion

This thesis features the analysis of training different QNN architectures on quantum computers and using graph-structured quantum data.

In order to exploit the advantages of quantum computation, the quantum circuit representations of the quantum neural network from [10] have been defined. The $\mathrm{DQNN_U}$ defined in [10] has been optimised to the $\mathrm{DQNN_{CAN}}$, which features fewer gates and fewer parameters to satisfy the conditions of today's NISQ devices. Additionally, the QNN representation of the QAOA has been defined to compare the noise robustness of the $\mathrm{DQNN_{CAN}}$'s and the QAOA's different architectures.

The $\mathrm{DQNN_{CAN}}$ and the QAOA have been successfully implemented and executed using simulated and real quantum computers and trained to learn an unknown unitary operator using classical optimisation methods to analyse their generalisation capability under the influence of noise. The generalisation analysis was performed by training the QNNs for different numbers of training states while the number of test states remained constant. The noise of *ibmq_ casablanca* has been added to simulate the training on a real quantum computer. This analysis has shown that both networks are capable of generalising the provided training states to the test states despite the high noise levels. Comparing both QNNs reveals that the $\mathrm{DQNN_{CAN}}$ performs slightly better and more reliable than the QAOA. This can be attributed to the fewer gates of the $\mathrm{DQNN_{CAN}}$'s transpiled quantum circuit. Furthermore, both QNNs have shown the ability to withstand the noise of today's NISQ devices.

Besides learning an unknown unitary operator, the training using graph-structured quantum data was studied based on the work of [63]. Here, the focus is on the special task to learn the interrelations of the training states to improve the QNNs' generalisations. After defining the framework for training a QNN the graph structure of the training states, the $\mathrm{DQNN_U}$, the $\mathrm{DQNN_{CAN}}$, and the QAOA have been trained using only a supervised cost function and using the supervised and a graph-based cost function. Two very interesting training examples have been constructed. One example features output states arranged in two connected clusters. In the other example, the output states are positioned in a connected line [63]. With this, the influence of the graph-based cost function can be studied. The analysis using three supervised states has shown that the graph-based cost function significantly improves the generalisation of the QNNs. In order to prove the universality of this result, the $\mathrm{DQNN}_{\mathrm{CAN}}$ has been trained for arbitrary supervised states. Here, the graph-based cost function shows only a slight improvement. One reason for this is unfavorable supervised states, e.g., where only one cluster was supervised. Then the interpolation of the graph-based cost function hinders the learning. Additionally, it was discovered that local minima occur right after the start of the training due to a comparatively high graph-based cost function. The delayed normalised graph-based cost function poses a possible solution. The local minima could be avoided by training the QNN using only the supervised cost function for the first half of training epochs and by normalising the graph-based cost function with its initial value. It has been shown that the delayed normalised graph-based cost function significantly improves the QNNs' generalisation even further. Additionally, the $\mathrm{DQNN}_{\mathrm{CAN}}$'s generalisation could be improved for arbitrary supervised states.

The analyses of this thesis have shown the excellent noise robustness and variability of the dissipative QNN architectures. The work can be easily extended by studying higher-dimensional or non-unitary operators and different graph-structured quantum data. Of course, the QNNs presented here can be applied to different learning tasks and trained using other optimisation methods.

The codes for both analyses can be found here: `https://github.com/qigitphannover/DeepQuantumNeuralNetworks`.

Bibliography

[1] Yuri Manin.
Computable and uncomputable.
Sovetskoye Radio, Moscow, 128, 1980.
1

[2] Richard P Feynman.
Simulating physics with computers.
Int. J. Theor. Phys, 21(6/7), 1982.
1, 5

[3] David Deutsch and Roger Penrose.
Quantum theory, the Church–Turing principle and the universal quantum computer.
Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences, 400(1818):97–117, 1985.
doi: 10.1098/rspa.1985.0070.
1

[4] David Deutsch and Richard Jozsa.
Rapid solution of problems by quantum computation.
Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences, 439(1907):553–558, 1992.
doi: 10.1098/rspa.1992.0167.
1, 5, 15

[5] Peter W Shor.
Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer.
SIAM review, 41(2):303–332, 1999.
doi: 10.1137/S0036144598347011.
1, 5, 18

[6] Lov K Grover.
Quantum mechanics helps in searching for a needle in a haystack.
Physical review letters, 79(2):325, 1997.
doi: 10.1103/PhysRevLett.79.325.
1, 5, 18

[7] Frank Rosenblatt.
Principles of neurodynamics. perceptrons and the theory of brain mechanisms.
Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.
1, 22

[8] Sebastian Ruder.
An overview of gradient descent optimization algorithms.
arXiv:1609.04747, 2017.
1, 30

[9] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd.
Quantum machine learning.
Nature, 549(7671):195–202, 2017.
doi: 10.1038/nature23474.
1, 45

[10] Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf.
Training deep quantum neural networks.
Nature communications, 11(1):1–6, 2020.
doi: 10.1038/s41467-020-14454-2.
1, 2, 3, 33, 34, 39, 69, vii

[11] Alexandr A. Ezhov and Dan Ventura.
Quantum Neural Networks.
Physica-Verlag HD, Heidelberg, 2000.

[12] M Andrecut and MK Ali.
A quantum neural network model.
International Journal of Modern Physics C, 13(01):75–88, 2002.
doi: 10.1142/S0129183102002948.

[13] MV Altaisky.
Quantum neural network.
arXiv:quant-ph/0107012, 2001.
1, 33, 34

[14] Sanjay Gupta and R.K.P. Zia.
Quantum neural networks.
Journal of Computer and System Sciences, 63(3):355–383, 2001.
doi: 10.1006/jcss.2001.1769.

[15] EC Behrman, V Chandrashekar, Z Wang, CK Belur, JE Steck, and SR Skinner.
A quantum neural network computes entanglement.
arXiv:quant-ph/0202131, 2002.

[16] Li Fei and Zheng Baoyu.
A study of quantum neural networks.
In *International Conference on Neural Networks and Signal Processing, 2003. Proceedings of the 2003*, volume 1, pages 539–542 Vol.1, 2003.
doi: 10.1109/ICNNSP.2003.1279330.

[17] Rigui Zhou, Huian Wang, Qian Wu, and Yang Shi.
Quantum Associative Neural Network with Nonlinear Search Algorithm.
International Journal of Theoretical Physics, 51(3):705–723, 2012.
doi: 10.1007/s10773-011-0950-4.

[18] Wilson R. de Oliveira, Adenilton J. Silva, Teresa B. Ludermir, Amanda Leonel, Wilson R. Galindo, and Jefferson C.C. Pereira.
Quantum Logical Neural Networks.
In *2008 10th Brazilian Symposium on Neural Networks*, pages 147–152, 2008.
doi: 10.1109/SBRN.2008.9.

[19] Geza Toth, Craig S. Lent, P.Douglas Tougaw, Yuriy Brazhnik, Weiwen Weng, Wolfgang Porod, Ruey-Wen Liu, and Yih-Fang Huang.
Quantum cellular neural networks.
Superlattices and Microstructures, 20(4):473–478, Dec 1996.
doi: 10.1006/spmi.1996.0104.
1

[20] Maria Schuld, Alex Bocharov, Krysta M Svore, and Nathan Wiebe.
Circuit-centric quantum classifiers.
Phys. Rev. A, 101(3):032308, 2020.
doi: 10.1103/PhysRevA.101.032308.
1, 33, 34

[21] Kaining Zhang, Min-Hsiu Hsieh, Liu Liu, and Dacheng Tao.
Toward Trainability of Quantum Neural Networks.
arXiv:2011.06258, 2020.

[22] Francesco Tacchino, Panagiotis Barkoutsos, Chiara Macchiavello, Ivano Tavernelli, Dario Gerace, and Daniele Bajoni.
Quantum implementation of an artificial feed-forward neural network.
Quantum Sci. Technol., 5(4):044010, 2020.
doi: 10.1088/2058-9565/abb8e4.

[23] Kunal Sharma, M. Cerezo, Lukasz Cincio, and Patrick J. Coles.
Trainability of Dissipative Perceptron-Based Quantum Neural Networks.
arXiv:2005.12458, 2020.

[24] Andrea Skolik, Jarrod R. McClean, Masoud Mohseni, Patrick van der Smagt, and Martin Leib.
Layerwise learning for quantum neural networks.
arXiv:2006.14904, 2020.

[25] Erik Torrontegui and Juan José García-Ripoll.
Unitary quantum perceptron as efficient universal approximator.
EPL, 125(3):30004, 2019.
doi: 10.1209/0295-5075/125/30004.

[26] Nathan Killoran, Thomas R Bromley, Juan Miguel Arrazola, Maria Schuld, Nicolás Quesada, and Seth Lloyd.
Continuous-variable quantum neural networks.
Phys. Rev. Research, 1(3):033063, 2019.
doi: 10.1103/PhysRevResearch.1.033063.

[27] Gregory R Steinbrecher, Jonathan P Olson, Dirk Englund, and Jacques Carolan.
Quantum optical neural networks.

npj Quantum Inf, 5(1):1–9, 2019.
doi: 10.1038/s41534-019-0174-7.

[28] Edward Farhi and Hartmut Neven.
Classification with Quantum Neural Networks on Near Term Processors.
arXiv:1802.06002, 2018.

[29] Yudong Cao, Gian Giacomo Guerreschi, and Alán Aspuru-Guzik.
Quantum Neuron: an elementary building block for machine learning on quantum computers.
arXiv:1711.11240, 2017.

[30] Kwok Ho Wan, Oscar Dahlsten, Hlér Kristjánsson, Robert Gardner, and MS Kim.
Quantum generalisation of feedforward neural networks.
npj Quantum Inf, 3(1):1–8, 2017.
doi: 10.1038/s41534-017-0032-4.

[31] Adenilton José da Silva, Teresa Bernarda Ludermir, and Wilson Rosa de Oliveira.
Quantum perceptron over a field and neural network architecture selection in a quantum computer.
Neural Networks, 76:55–64, 2016.
doi: 10.1016/j.neunet.2016.01.002.

[32] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione.
Simulating a perceptron on a quantum computer.
Physics Letters A, 379(7):660–663, 2015.
doi: 10.1016/j.physleta.2014.11.061.

[33] Maciej Lewenstein.
Quantum perceptrons.
Journal of Modern Optics, 41(12):2491–2501, 1994.
doi: 10.1080/09500349414552331.

[34] Yao Zhang and Qiang Ni.
Design of Quantum Neuron Model for Quantum Neural Networks.
Quantum Engineering, page e75, 2021.
doi: 10.1002/que2.75.
1, 33, 34

[35] Carlos Pedro Gonçalves.
Quantum Neural Machine Learning - Backpropagation and Dynamics.
arXiv:1609.06935, 2016.
1

[36] Masaya Watabe, Kodai Shiba, Masaru Sogabe, Katsuyoshi Sakamoto, and Tomah Sogabe.
Quantum Circuit Parameters Learning with Gradient Descent Using Backpropagation.
arXiv:1910.14266, 2019.

[37] Guillaume Verdon, Jason Pye, and Michael Broughton.
A Universal Training Algorithm for Quantum Deep Learning.
arXiv:1806.09729, 2018.
1

[38] James Stokes, Josh Izaac, Nathan Killoran, and Giuseppe Carleo.
Quantum natural gradient.
Quantum, 4:269, 2020.
doi: 10.22331/q-2020-05-25-269.
1, 33, 58

[39] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran.
Evaluating analytic gradients on quantum hardware.
Phys. Rev. A, 99:032331, Mar 2019.
doi: 10.1103/PhysRevA.99.032331.
39

[40] Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti.
Structure optimization for parameterized quantum circuits.
Quantum, 5:391, Jan 2021.
doi: 10.22331/q-2021-01-28-391.

[41] K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii.
Quantum circuit learning.
Phys. Rev. A, 98:032309, Sep 2018.
doi: 10.1103/PhysRevA.98.032309.
1, 2, 33, 39

[42] Kaifeng Bu, Dax Enshan Koh, Lu Li, Qingxian Luo, and Yaobo Zhang.
On the statistical complexity of quantum circuits.
arXiv:2101.06154, 2021.

[43] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini.
Parameterized quantum circuits as machine learning models.
Quantum Sci. Technol., 4(4):043001, 2019.
doi: 10.1088/2058-9565/ab5944.

[44] Yuxuan Du, Min-Hsiu Hsieh, Tongliang Liu, and Dacheng Tao.
Expressive power of parametrized quantum circuits.
Physical Review Research, 2(3), Jul 2020.
doi: 10.1103/physrevresearch.2.033125.
2, 33

[45] John Preskill.
Quantum Computing in the NISQ era and beyond.
Quantum, 2:79, Aug 2018.
doi: 10.22331/q-2018-08-06-79.
2, 15, 45

[46] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann.
A Quantum Approximate Optimization Algorithm.
arXiv:1411.4028, 2014.
2, 42

[47] Edward Farhi and Aram W Harrow.
Quantum Supremacy through the Quantum Approximate Optimization Algorithm.
arXiv:1602.07674, 2019.

[48] Stuart Hadfield, Zhihui Wang, Bryan O'Gorman, Eleanor G Rieffel, Davide Venturelli, and Rupak Biswas.
From the quantum approximate optimization algorithm to a quantum alternating operator ansatz.
Algorithms, 12(2):34, 2019.
doi: 10.3390/a12020034.
2, 42

[49] Bobak Toussi Kiani, Seth Lloyd, and Reevu Maity.
Learning Unitaries by Gradient Descent.
arXiv:2001.11897, 2020.
2, 42, xi

[50] Michael Streif and Martin Leib.
Comparison of QAOA with Quantum and Simulated Annealing.
arXiv:1901.01903, 2019.
2

[51] Zhihui Wang, Stuart Hadfield, Zhang Jiang, and Eleanor G. Rieffel.
Quantum approximate optimization algorithm for MaxCut: A fermionic view.
Phys. Rev. A, 97:022304, Feb 2018.
doi: 10.1103/PhysRevA.97.022304.

[52] Dave Wecker, Matthew B. Hastings, and Matthias Troyer.
Training a quantum optimizer.
Phys. Rev. A, 94:022309, Aug 2016.
doi: 10.1103/PhysRevA.94.022309.

[53] Filip B Maciejewski, Flavio Baccari, Zoltán Zimborás, and Michał Oszmaniec.
Modeling and mitigation of cross-talk effects in readout noise with applications to the Quantum Approximate Optimization Algorithm.
Quantum, 5:464, Jun 2021.
doi: 10.22331/q-2021-06-01-464.
2, 45

[54] J. S. Otterbach, R. Manenti, N. Alidoust, A. Bestwick, M. Block, B. Bloom, S. Caldwell, N. Didier, E. Schuyler Fried, S. Hong, P. Karalekas, C. B. Osborn, A. Papageorge, E. C. Peterson, G. Prawiroatmodjo, N. Rubin, Colm A. Ryan, D. Scarabelli, M. Scheer, E. A. Sete, P. Sivarajah, Robert S. Smith, A. Staley, N. Tezak, W. J. Zeng, A. Hudson, Blake R. Johnson, M. Reagor, M. P. da Silva, and C. Rigetti.
Unsupervised Machine Learning on a Hybrid Quantum Computer.
arXiv:1712.05771, 2017.

[55] Wolfgang Lechner.
Quantum Approximate Optimization With Parallelizable Gates.

IEEE Transactions on Quantum Engineering, 1:1–6, 2020.
doi: 10.1109/TQE.2020.3034798.

[56] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann.
A Quantum Approximate Optimization Algorithm Applied to a Bounded Occurrence Constraint Problem.
arXiv:1412.6062, 2015.

[57] E. Farhi, J. Goldstone, S. Gutmann, and H. Neven.
Quantum Algorithms for Fixed Qubit Architectures.
arXiv:1703.06199, 2017.

[58] Cedric Yen-Yu Lin and Yechao Zhu.
Performance of QAOA on Typical Instances of Constraint Satisfaction Problems with Bounded Degree.
arXiv:1601.01744, 2016.
2

[59] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh.
Analysis of Quantum Approximate Optimization Algorithm under Realistic Noise in Superconducting Qubits.
arXiv:1907.09631, 2019.
2, 45

[60] Cheng Xue, Zhao-Yun Chen, Yu-Chun Wu, and Guo-Ping Guo.
Effects of Quantum Noise on Quantum Approximate Optimization Algorithm.
arXiv:1909.02196, 2019.
2, 45

[61] Héctor Abraham et al.
Qiskit: An Open-source Framework for Quantum Computing.
2019.
2, 45, 47, v

[62] IBM Quantum team.
IBM Quantum Experience.
`https://quantum-computing.ibm.com`.
2021.
2, 12, 45, 47, 49, v, xiii

[63] Kerstin Beer, Megha Khosla, Julius Köhler, and Tobias J. Osborne.
Quantum machine learning of graph-structured data.
arXiv:2103.10837, 2021.
2, 53, 63, 66, 67, 69, 70, viii, xvii

[64] Alan Mathison Turing.
On computable numbers, with an application to the Entscheidungsproblem.
Proceedings of the London mathematical society, 2(1):230–265, 1937.
doi: 10.1112/plms/s2-42.1.230.
5

[65] John Bardeen and Walter Hauser Brattain.
The transistor, a semi-conductor triode.
Physical Review, 74(2):230, 1948.
doi: 10.1103/PhysRev.74.230.
5

[66] Michael A Nielsen and Isaac Chuang.
Quantum computation and quantum information.
2002.
6, 14, 33, xiii

[67] Claude Elwood Shannon.
A mathematical theory of communication.
The Bell system technical journal, 27(3):379–423, 1948.
6

[68] Alexei Yu Kitaev, Alexander Shen, Mikhail N Vyalyi, and Mikhail N Vyalyi.
Classical and quantum computation.
Number 47. American Mathematical Society, 2002.
doi: 10.1090/gsm/047.
12

[69] David P. DiVincenzo.
The Physical Implementation of Quantum Computation.
Fortschritte der Physik, 48(9-11):771–783, 2000.
doi: 10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E.
14

[70] John Clarke and Frank K Wilhelm.
Superconducting quantum bits.
Nature, 453(7198):1031–1042, 2008.
doi: 10.1038/nature07128.
14, 38

[71] JQ You and Franco Nori.
Atomic physics and quantum optics using superconducting circuits.
Nature, 474(7353):589–597, 2011.
doi: 10.1038/nature10122.
14, 38

[72] J. I. Cirac and P. Zoller.
Quantum Computations with Cold Trapped Ions.
Phys. Rev. Lett., 74:4091–4094, May 1995.
doi: 10.1103/PhysRevLett.74.4091.
14, 38

[73] Dietrich Leibfried, Rainer Blatt, Christopher Monroe, and David Wineland.
Quantum dynamics of single trapped ions.
Reviews of Modern Physics, 75(1):281, 2003.
doi: 10.1103/RevModPhys.75.281.

[74] Rainer Blatt and Christian F Roos.
Quantum simulations with trapped ions.

Nature Physics, 8(4):277–284, 2012.
doi: 10.1038/nphys2252.
14, 38

[75] M. B. Plenio and P. L. Knight.
Decoherence limits to quantum computation using trapped ions.
Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences, 453(1965):2017–2041, 1997.
doi: 10.1098/rspa.1997.0109.
14

[76] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al.
Quantum supremacy using a programmable superconducting processor.
Nature, 574(7779):505–510, 2019.
doi: 10.1038/s41586-019-1666-5.
15

[77] Warren S McCulloch and Walter Pitts.
A logical calculus of the ideas immanent in nervous activity.
The bulletin of mathematical biophysics, 5(4):115–133, 1943.
doi: 10.1007/BF02478259.
21

[78] Stephen Cole Kleene.
Representation of events in nerve nets and finite automata.
Princeton University Press, 2016.
doi: 10.1515/9781400882618-002.
21

[79] Michael A Nielsen.
Neural networks and deep learning, volume 25.
Determination press San Francisco, CA, 2015.
21, 22

[80] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad.
State-of-the-art in artificial neural network applications: A survey.
Heliyon, 4(11):e00938, 2018.
doi: 10.1016/j.heliyon.2018.e00938.
21

[81] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton.
Speech recognition with deep recurrent neural networks.
In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.
doi: 10.1109/ICASSP.2013.6638947.
21

[82] Samira Ebrahimi Kahou, Vincent Michalski, Kishore Konda, Roland Memisevic, and Christopher Pal.
Recurrent neural networks for emotion recognition in video.

In *Proceedings of the 2015 ACM on international conference on multimodal interaction*, pages 467–474, 2015.
doi: 10.1145/2818346.2830596.
21

[83] Stefan Leijnen and Fjodor van Veen.
The neural network zoo.
In *Multidisciplinary Digital Publishing Institute Proceedings*, volume 47, page 9, 2020.
doi: 10.3390/proceedings2020047009.
21

[84] Yann LeCun.
The MNIST database of handwritten digits.
`http://yann.lecun.com/exdb/mnist/`.
1998.
25

[85] Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze.
Deep clustering for unsupervised learning of visual features.
In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 132–149, 2018.
25

[86] Trevor Hastie, Robert Tibshirani, and Jerome Friedman.
Unsupervised learning.
In *The elements of statistical learning*, pages 485–585. Springer, 2009.
25

[87] Diederik P. Kingma and Jimmy Ba.
Adam: A Method for Stochastic Optimization.
arXiv:1412.6980, 2017.
30, 58

[88] John Duchi, Elad Hazan, and Yoram Singer.
Adaptive subgradient methods for online learning and stochastic optimization.
Journal of machine learning research, 12(7), 2011.
30

[89] Tijmen Tieleman, Geoffrey Hinton, et al.
Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.
COURSERA: Neural networks for machine learning, 4(2):26–31, 2012.
30

[90] Steven J Nowlan and Geoffrey E Hinton.
Simplifying neural networks by soft weight-sharing.
Neural computation, 4(4):473–493, 1992.
doi: 10.1162/neco.1992.4.4.473.
31

[91] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov.
Dropout: a simple way to prevent neural networks from overfitting.

The journal of machine learning research, 15(1):1929–1958, 2014.
31

[92] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles.
Variational Quantum Algorithms.
arXiv:2012.09265, 2020.
36

[93] Lukasz Cincio, Yiğit Subaşi, Andrew T Sornborger, and Patrick J Coles.
Learning the quantum algorithm for state overlap.
New Journal of Physics, 20(11):113022, 2018.
doi: 10.1088/1367-2630/aae94a.
37

[94] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada.
Swap test and Hong-Ou-Mandel effect are equivalent.
Phys. Rev. A, 87:052330, May 2013.
doi: 10.1103/PhysRevA.87.052330.
37

[95] D. Mc Hugh and J. Twamley.
Quantum computer using a trapped-ion spin molecule and microwave radiation.
Phys. Rev. A, 71:012315, Jan 2005.
doi: 10.1103/PhysRevA.71.012315.
38

[96] M. Steffen, D. P. DiVincenzo, J. M. Chow, T. N. Theis, and M. B. Ketchen.
Quantum computing: An IBM perspective.
IBM Journal of Research and Development, 55(5):13:1–13:11, 2011.
doi: 10.1147/JRD.2011.2165678.
38

[97] Andrea Mari, Thomas R. Bromley, and Nathan Killoran.
Estimating the gradient and higher-order derivatives on quantum hardware.
Physical Review A, 103(1), Jan 2021.
doi: 10.1103/physreva.103.012405.
39

[98] Jun Li, Xiaodong Yang, Xinhua Peng, and Chang-Pu Sun.
Hybrid Quantum-Classical Approach to Quantum Optimal Control.
Phys. Rev. Lett., 118:150503, Apr 2017.
doi: 10.1103/PhysRevLett.118.150503.
39

[99] Kerstin Beer, Daniel List, Gabriel Müller, Tobias J. Osborne, and Christian Struckmann.
Training Quantum Neural Networks on NISQ Devices.
arXiv:2104.06081, 2021.
39, 46

[100] Gavin E. Crooks.
Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition.
arXiv:1905.13311, 2019.
41

[101] Jun Zhang, Jiri Vala, Shankar Sastry, and K. Birgitta Whaley.
Geometric theory of nonlocal two-qubit operations.
Physical Review A, 67(4), Apr 2003.
doi: 10.1103/physreva.67.042313.
41

[102] Jun Zhang, Jiri Vala, Shankar Sastry, and K. Birgitta Whaley.
Optimal quantum circuit synthesis from controlled-unitary gates.
Physical Review A, 69(4), Apr 2004.
doi: 10.1103/physreva.69.042309.

[103] M Blaauboer and RL De Visser.
An analytical decomposition protocol for optimal implementation of two-qubit entangling gates.
Journal of Physics A: Mathematical and Theoretical, 41(39):395307, sep 2008.
doi: 10.1088/1751-8113/41/39/395307.

[104] Byron Drury and Peter Love.
Constructive quantum Shannon decomposition from Cartan involutions.
Journal of Physics A: Mathematical and Theoretical, 41(39):395305, Sep 2008.
doi: 10.1088/1751-8113/41/39/395305.

[105] Paul Watts, Maurice O'Connor, and Jiří Vala.
Metric Structure of the Space of Two-Qubit Gates, Perfect Entanglers and Quantum Control.
Entropy, 15(6):1963–1984, 2013.
doi: 10.3390/e15061963.

[106] Eric C. Peterson, Gavin E. Crooks, and Robert S. Smith.
Two-Qubit Circuit Depth and the Monodromy Polytope.
Quantum, 4:247, Mar 2020.
doi: 10.22331/q-2020-03-26-247.
41

[107] Carlo Ciliberto, Mark Herbster, Alessandro Davide Ialongo, Massimiliano Pontil, Andrea Rocchetto, Simone Severini, and Leonard Wossnig.
Quantum machine learning: a classical perspective.
Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 474(2209):20170551, 2018.
doi: 10.1098/rspa.2017.0551.
45, 46

[108] Noriaki Kouda, Nobuyuki Matsui, Haruhiko Nishimura, and Ferdinand Peper.
Qubit neural network and its learning efficiency.

Neural Computing & Applications, 14(2):114–121, 2005.
doi: 10.1007/s00521-004-0446-8.
45

[109] Kyle Poland, Kerstin Beer, and Tobias J. Osborne.
No Free Lunch for Quantum Machine Learning.
arXiv:2003.14103, 2020.
46

[110] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta.
Validating quantum computers using randomized model circuits.
Phys. Rev. A, 100:032328, Sep 2019.
doi: 10.1103/PhysRevA.100.032328.
49

[111] Piotr Czarnik, Andrew Arrasmith, Lukasz Cincio, and Patrick J. Coles.
Qubit-efficient exponential suppression of errors.
arXiv:2102.06056, 2021.
50

[112] Sagar Sharma and Simone Sharma.
Activation functions in neural networks.
Towards Data Science, 6(12):310–316, 2017.
i

[113] Abraham Asfaw, Antonio Corcoles, Luciano Bello, Yael Ben-Haim, Mehdi Bozzo-Rey, Sergey Bravyi, Nicholas Bronn, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, Hwajung Kang, Amir h. Karamlou, Robert Loredo, David McKay, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicini, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, John Stenger, Kristan Temme, Madeleine Tod, Stephen Wood, and James Wootton.
Learn Quantum Computation Using Qiskit.
`http://community.qiskit.org/textbook`.
2020.
v

[114] Benjamin Nachman, Miroslav Urbanek, Wibe A de Jong, and Christian W Bauer.
Unfolding quantum computer readout noise.
npj Quantum Information, 6(1):1–7, 2020.
xiii

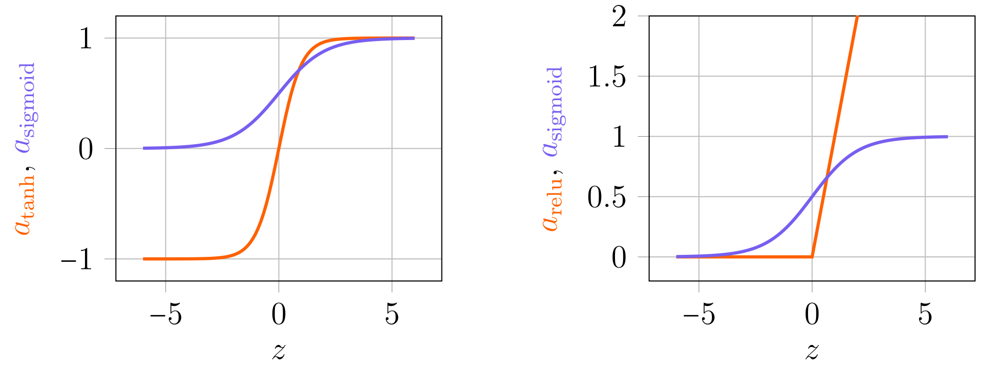
Appendix A

Supplementary Materials to Chapter 3

A.1 Improving the learning

A.1.1 Different activation functions

In chapter 3, the neural network was assumed to consist of only sigmoid neurons. An important property of this neuron is that it can represent any given function. There are, in fact, neurons based on different activation functions with this property, which sometimes outperform the sigmoid neuron [112].



(a) The tanh — and sigmoid — activation function.

(b) The ReLu — and sigmoid — activation function.

Figure A.1: Alternative activation functions. Note that $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$.

One popular alternative to the sigmoid neuron is the tanh neuron. Its activation function is given by the hyperbolic tangent:

$$a_{\text{tanh}}(\boldsymbol{x}; \boldsymbol{w}, b) = \tanh(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{A.1}$$

where $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. It is closely related to the sigmoid function:

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2}. \tag{A.2}$$

The tanh and sigmoid activation functions are plotted in Fig. A.1a. One key difference between both activation functions is the output range. The sigmoid neuron only allows positive activations. However, the activation of the tanh neuron ranges from −1 to +1 and thus, allows positive and negative activations.

Another important neuron is the rectified linear neuron/unit (ReLu). Its activation function is a linear function except that it maps negative values to 0:

$$a_{\text{relu}}(\boldsymbol{w} \cdot \boldsymbol{x} + b) := \max(0, \boldsymbol{w} \cdot \boldsymbol{x} + b). \tag{A.3}$$

It is plotted in Fig. A.1b alongside the sigmoid function. As it is zero for all negative values, the ReLu activation function allows the neuron to neglect some of its inputs.

A.1.2 The cross-entropy cost function

The most crucial step in training a neural network is finding the cost function as its form defines the landscape of the minimisation problem. For gradient-based optimisers, it is crucial to have a cost gradient that scales with the error/cost of the network. Like its biological counterpart, the neural network should learn the most after being the wrongest.

The gradient of the quadratic cost function is given by:

$$\frac{\partial C}{\partial b^l_j} = \frac{1}{n} \sum_{k=1}^{n} a'(z^l_j)(\boldsymbol{\alpha}^{\text{out},k} - \boldsymbol{x}^{\text{out},k}), \tag{A.4a}$$

$$\frac{\partial C}{\partial \boldsymbol{w}^l_j} = \frac{1}{n} \sum_{k=1}^{n} [\boldsymbol{\alpha}^{l-1}]_j a'(z^l_j)(\boldsymbol{\alpha}^{\text{out},k} - \boldsymbol{x}^{\text{out},k}). \tag{A.4b}$$

The step size scales linearly with the derivative of the activation function. Usually, the activation function's slope decreases with increasing distance from the origin (see Fig. A.1b). If the absolute of the neuron's output is large, it scales down the cost gradient (see (A.4)). Thus, the gradient of the quadratic cost can be small even though if the error is not.

This problem is solved by introducing the *cross-entropy cost function*:

$$C = -\frac{1}{n} \sum_{k=1}^{n} \sum_{j=1}^{n_{\text{out}}} \left([\boldsymbol{x}^{\text{out},k}]_j \ln\left([\boldsymbol{\alpha}^{\text{out},k}]_j\right) + \left(1 - [\boldsymbol{x}^{\text{out},k}]_j\right) \ln\left(1 - [\boldsymbol{\alpha}^{\text{out},k}]_j\right) \right). \tag{A.5}$$

This cost function is defined such that its gradient does not depend on the activation function's derivation:

$$\frac{\partial C}{\partial b_j^l} = \frac{1}{n}\sum_{k=1}^{n}\sum_{j=1}^{n_{\text{out}}}(\boldsymbol{\alpha}^{\text{out},k} - \boldsymbol{x}^{\text{out},k}), \tag{A.6a}$$

$$\frac{\partial C}{\partial \boldsymbol{w}_j^l} = \frac{1}{n}\sum_{k=1}^{n}\sum_{j=1}^{n_{\text{out}}}[\boldsymbol{\alpha}^{l-1}]_j(\boldsymbol{\alpha}^{\text{out},k} - \boldsymbol{x}^{\text{out},k}). \tag{A.6b}$$

The cross-entropy cost function's gradient only scales with the network's error $\boldsymbol{\alpha}^{\text{out},k} - \boldsymbol{x}^{\text{out},k}$.

Appendix B

The IBM Quantum Experience

The IBM Quantum Experience (IBMQE) is a website providing public and premium access to the IBM quantum computers [62]. It can be accessed via Qiskit, an open-source *quantum information software development kit* written in Python [61]. It allows the building, compiling, running, and analysing of quantum circuits. The interested reader is referred to [113].

Building a quantum circuit

Building a quantum circuit using Qiskit is straightforward. First, the quantum circuit has to be defined by specifying the number of quantum and classical bits. Then, predefined but also arbitrary unitary operators can be applied to the qubit register. Finally, a measurement provides information about the qubits' state.

Compiling a quantum circuit

The compilation of a quantum circuit can be divided into two main steps. First, the quantum circuit is transpiled into a different quantum circuit consisting of basis gates. These basis gates, of course, depend on the quantum device itself. This step is crucial for the operation using real quantum computers. The transpile function takes many arguments that can be used to optimise the quantum circuit with respect to the quantum device. Second, the quantum circuit is assembled to a so-called Qobj. This is an executable for the quantum device.

Running a quantum circuit

Executing a quantum circuit can be done either using a simulator or a real quantum computer. The IBMQE and Qiskit provide various methods for each method. Additionally, Qiskit also allows the combination of both, the simulated quantum computer. It is possible to create a specific quantum device with customisable properties, including its noise. The IBMQE features quantum computers that are free for public use and quantum computers of more qubits for premium access only. To run the transpiled quantum circuit on the quantum device, it has to be submitted to the IBMQE, where it first joins a queue. The queue is processed one after the other. After running the quantum circuit, IBMQE returns a report including the measurement results.

Analysing a quantum circuit

The IBMQE and Qiskit provide various tools for analysing quantum circuits. The IBMQE website features a quantum circuit composer where the quantum circuit can be constructed by dragging and dropping quantum gates. The result is computed directly and visible in the console. Qiskit includes many predefined plotting routines to visualise the measurement outcomes or the quantum states (if executed using a simulator).

Appendix C

Supplementary Materials to Chapter 4

C.1 Quantum backpropagation

The presentation of the quantum backpropagation algorithm is based on [10]. The interested reader is referred to this work.

Classically, the neural network is trained by updating the weights and biases of each neuron in a way, such that the cost function is minimised. A QNN's neuron, has no weights and bias, but instead, a unitary operator. All the processing information of the weights and bias are stored as components of the unitary operator. The update rule of the unitary operator of the jth neuron in layer l is given by

$$U_j^l \mapsto e^{i\epsilon K_j^l} U_j^l \tag{C.1}$$

where ϵ is the step size and K_j^l is a hermitian operator defined to maximise the training cost function (4.6). The change in the cost function after updating the quantum perceptrons according to (C.1) is given by

$$\Delta C = \frac{\epsilon}{N} \sum_{k=1}^{n_{\text{train}}} \sum_{l=1}^{\text{out}} \operatorname{Tr}\left(\sigma^{l,k} \Delta\mathcal{E}(\rho^{l-1,k})\right). \tag{C.2}$$

This change is maximised by setting

$$K_j^l = \eta \frac{2^{n_{l-1}} i}{n_{\text{train}}} \sum_{k=1}^{n_{\text{train}}} \operatorname{Tr}_{\text{rest}}\left(M_j^{l,k}\right) \tag{C.3}$$

where η is the learning rate and the trace is over all qubits on which U_j^l does not act on. $M_j^{l,k}$ is defined as the commutator of a forward and backward application of quantum perceptrons:

$$M_j^{l,k} = \left[\prod_{\alpha=j}^{1} U_\alpha^l \left(\rho^{l-1,k} \otimes |0\dots0\rangle_l \langle 0\dots 0|\right) \prod_{\alpha=1}^{j} {U_\alpha^l}^\dagger, \prod_{\alpha=j+1}^{n_l} {U_\alpha^l}^\dagger \left(\mathbb{1}_{l-1} \otimes \sigma^{l,k}\right) \prod_{\alpha=n_l}^{j+1} U_\alpha^l\right] \tag{C.4}$$

where

$$\sigma^l = \left(\mathcal{F}^{l+1} \circ \dots \circ \mathcal{F}^{\text{out}}\right)\left(\rho^{\text{out}}\right). \tag{C.5}$$

$\mathcal{F}^l$ is the adjoint channel of $\mathcal{E}^l$ and is defined as

$$\mathcal{F}^l(\sigma^l) = \text{Tr}_l \left(\mathbb{1}_{l-1} \otimes |0\dots0\rangle_l \langle 0\dots0| {U^l}^\dagger \left(\mathbb{1}_{l-1} \otimes \sigma^l\right) U^l\right) = \sigma^{l-1}. \tag{C.6}$$

This channel incorporates the backpropagation of the classical optimisation algorithm as the state $\sigma^{\text{out}} = \rho^{\text{out}}$ is propagated backwards through the QNN by reversing the transformations/actions $(U^l)^\dagger = (U^l)^{-1}$. The parameter matrix K_j^l can be fully calculated by computing ρ^{l-1} (see (4.3)) and σ^l (see (C.5)). Thus, the parameter matrices K_j^l can be computed layerwise, meaning its computation does not require the evaluation of the whole QNN.

C.1.1 Graph-structured quantum data

Similar to training a unitary transformation, the quantum perceptrons are updated using a hermitian matrix K_j^l (see (C.1)) [63]. It is composed of the update matrix of the supervised states (C.3) and of the matrix for the graph structure

$$K_j^l = \eta \frac{2^{n_{l-1}} i}{n_{\text{SV}}} \sum_u \text{Tr}\left(M_j^{l,u}\right) + \eta 2^{n_{l-1}+1} i \sum_{v \sim w} [A]_{vw} \, \text{Tr}_{\text{rest}} \left(M_j^{l,\{v,w\}}\right) \tag{C.7}$$

where $v \sim w$ denotes all adjacent vertices $v,w \in V$ and

$$\begin{aligned} M_j^{l,\{v,w\}} = \Big[& U_j^l \cdots U_1^l \left(\left(|\phi^{\text{in},v}\rangle\langle\phi^{\text{in},v}| - |\phi^{\text{in},w}\rangle\langle\phi^{\text{in},w}|\right) \otimes |0\dots0\rangle_l \langle 0\dots0|\right) {U_1^l}^\dagger \cdots {U_j^l}^\dagger, \\ & {U_{j+1}^l}^\dagger \cdots {U_{n_{\text{out}}}^{\text{out}}}^\dagger \left(\mathbb{1}_{\text{in},\dots,L} \otimes \left(|\phi^{\text{out},v}\rangle\langle\phi^{\text{out},v}| - |\phi^{\text{out},w}\rangle\langle\phi^{\text{out},w}|\right)\right) U_{n_{\text{out}}}^{\text{out}} \cdots U_{j+1}^l \Big] \end{aligned} \tag{C.8}$$

Again, the update matrices K_j^l allow a layer-wise computation.

C.2 The CAN-based dissipative quantum neural network

The qubits which are not affected by the gate have been neglected in the definition (4.13). The precise definition of the quantum perceptron is given by

$$U_j^l = \text{SWAP}_j^l \left(\prod_{i=1}^{n_{l-1}} G^l(\theta_{3*(i-1)+1}, \theta_{3*(i-1)+2}, \theta_{3*(i-1)+3}) \, \text{SWAP}_j^{l-1}\right) \text{SWAP}_j^l \tag{C.9}$$

where

$$G^l(\theta_1,\theta_2,\theta_3) = \mathbb{1}_{1,\dots,n_{l-1}-1} \otimes G^l_{n_{l-1},n_{l-1}+1}(\theta_1,\theta_2,\theta_3) \otimes \mathbb{1}_{n_{l-1}+2,\dots,n_l+n_{l-1}} \tag{C.10a}$$

$$\mathbb{1}_{1,\dots,n} = \bigotimes_{k=1}^{n} \mathbb{1}, \quad \mathbb{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{C.10b}$$

$$\mathrm{SWAP}^{l-1} = \prod_{k=n_{l-1}-1}^{1} \mathrm{SWAP}_{k,k+1} \tag{C.10c}$$

$$\mathrm{SWAP}^l_j = \prod_{k=n_{l-1}+1}^{n_{l-1}+j-1} \mathrm{SWAP}_{k,k+1} \tag{C.10d}$$

and U^l_j acts on $n_{l-1} + n_l$ qubits. The swaps are needed such that $G^l_{k,j}$ acts on adjacent qubits: $j = k + 1$.

C.3 The quantum approximate optimisation algorithm

The quantum approximate optimisation algorithm (QAOA) was first introduced in 2014 by Edward Farhi and Jeffrey Goldstein to approximately solve combinatorial optimisation problems. Such problems are specified by n bits and m clauses, where each of these clauses is a constraint on the n bit strings. The challenge is to find the best solution out of the set of possible solutions. The quality of a solution is determined by the number of satisfied constraints and is given by the objective function:

$$C(z) = \sum_{\alpha=1}^{m} C_\alpha(z) \tag{C.11}$$

where $z = z_1 \dots z_n$ is the bit string and $C_\alpha(z)$ is the binary function that is 1 if z satisfies clause α and 0 otherwise. This objective function can be viewed as a diagonal operator in the computational basis:

$$H_C = \sum_z C(z) \left|z\right\rangle \left\langle z\right| \tag{C.12}$$

where $z \in \{0,1\}^n$ labels the computational basis states $|z\rangle \in \mathbb{C}^{2n}$. H_C is called the cost hamiltonian. From this the unitary time evolution can be defined:

$$U(C,\gamma) = e^{-i\gamma H_C} = \prod_{\alpha=1}^{m} e^{-i\gamma H_{C_\alpha}} \tag{C.13}$$

which is parameterised by an angle γ. Note that each element of the product acts according to a specific clause α and, in general, commutes with the others.

Consider the sum of single bit operators σ^x:

$$B = \sum_{j=1}^{n} \sigma_j^x \tag{C.14}$$

and the parameterised unitary

$$U(B,\beta) = e^{-i\beta B} = \prod_{j=1}^{n} e^{-i\beta\sigma_j^x} \tag{C.15}$$

where σ_j^x is the NOT-gate acting on qubit j. B is called the mixer hamiltonian.

The QAOA is defined as an alternating sequence of unitary operators acting on some initial state $|s\rangle$:

$$|\boldsymbol{\gamma},\boldsymbol{\beta}\rangle = U(B,\beta_p)U(C,\gamma_p)\dots U(B,\beta_1)U(C,\gamma_1)\,|s\rangle \tag{C.16}$$

where $|s\rangle$ is chosen to be the uniform superposition of the computational basis states. The optimal solution can be found by maximising the expectation value of the cost hamiltonian:

$$\langle\boldsymbol{\gamma},\boldsymbol{\beta}|\, H_C\,|\boldsymbol{\gamma},\boldsymbol{\beta}\rangle = \sum_z C(z)|\,\langle z|\boldsymbol{\gamma},\boldsymbol{\beta}\rangle\,|^2. \tag{C.17}$$

This expectation value can be easily evaluated using a quantum computer. After the state $|\boldsymbol{\gamma},\boldsymbol{\beta}\rangle$ is prepared it can be measured in the computational basis to get $|\,\langle z|\boldsymbol{\gamma},\boldsymbol{\beta}\rangle\,|^2$. Classical optimisation of $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ with respect to C.17 then gives an approximation for an optimal bit string z.

C.3.1 The quantum alternating operator ansatz

The quantum approximate optimisation algorithm described above features the time evolution under fixed local Hamiltonians. This framework has been extended to the quantum alternating operator ansatz, which is the alternating application of general parameterised unitaries:

$$U = e^{-i\alpha_n A}e^{-i\beta_n B}\cdots e^{-i\alpha_2 A}e^{-i\beta_2 B}\,e^{-i\alpha_1 A}e^{-i\beta_1 B} \tag{C.18}$$

where A and B are hermitian matrices. Both definitions of the acronym QAOA are used interchangeable for both algorithm.

C.4 Learning non-unitary transformations

Applying the QAOA to a problem that requires non-unitary transformations, e.g., training a ✄ QNN, is not trivial since the QAOA is defined as a sequence of unitary operators. The most natural way of redefining the QAOA for such problems is to apply the QAOA to all the qubits and trace out the superfluous qubits. For a QNN, this would mean applying the QAOA to the three input qubits and, in the end, ignore the first two qubits such that the third qubit serves as the output of the QNN. The only question that remains is concerning the number of layers. For a QNN of constant width, the number of layers is chosen according to the Hilbert space dimension [49]. If the number of layers equals $d^2/2$, the QNN trained using gradient descent always converges to the optimal solution. In other words, the number of parameters of the QAOA should match the number of components of the desired matrix transformation. For example, the matrix representation of a QNN has $2^2 \times 2^2 = 16$ components resulting in a QAOA of 16 parameters. Thus, the natural guess would be that the QAOA of a QNN should also have $2^3 \times 2^1 = 16$ parameters. In order to validate this guess, QAOAs of different lengths are trained to map eight random input states to the eight output states of the connected line training data (see section 6.2.3). This analysis quantifies the QAOA's capability to produce the desired output states from a given input. This ability sets the limit for the network's generalisation. Both networks are trained using the supervised cost function (6.4) for 500 epochs. The training is repeated four times to average over different start parameters, input states, and A and B matrices.

The resulting training costs after training the DQNN ($n_p = 21$) and the QAOA with $n_p = 14{,}16{,}20{,}24{,}26{,}32$ parameters are shown in Fig. C.1. As expected, the training cost of the QAOA increases with increasing numbers of parameters. The results validate the initial guess that the QAOA with $n_p = 2^3 \times 2^1 = 16$ parameters shows a similar training cost as the DQNN$_{\text{CAN}}$. Thus, the analysis in chapter 6 features a QAOA with $n_p = 16$ parameters or eight layers.

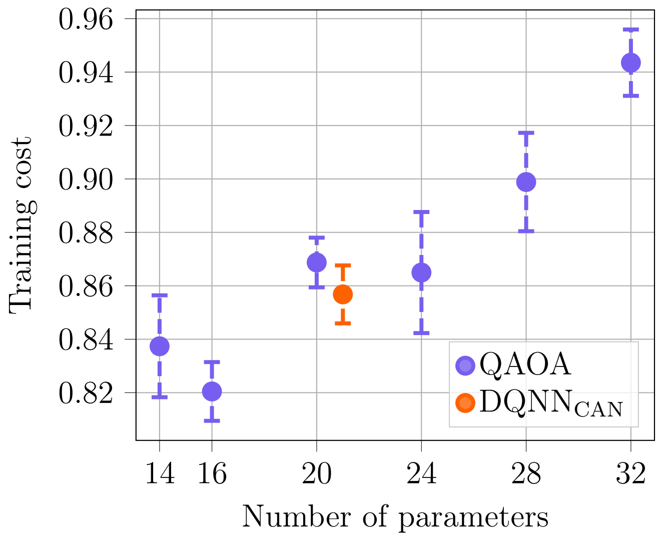


Figure C.1: Comparing the DQNN$_{\mathrm{CAN}}$ with QAOAs of different lengths. Both networks are trained for 500 epochs using the connected line training data (see section 6.2.3) and the supervised cost function (6.4). The training costs averaged over four trainings are plotted versus the number of parameters. The QAOA is defined to act on three qubits where the last qubit serves as the output. The number of layers equals half the number of parameters. The ✂ DQNN$_{\mathrm{CAN}}$ has 21 parameters.

Appendix D

Supplementary Materials to Chapter 5

D.1 Gate noise analysis

The success of training a QNN on a real quantum computer is highly dependent on the device's noise, as this is the main limiting factor of today's quantum algorithms. Readout noise and gate noise are the dominant noise sources [114]. This analysis focuses on the training of the DQNN and the QAOA under different noise levels. As both networks experience the influence of readout noise equally (they are measured and updated using the same methods), this noise source is neglected in this analysis. The influence of gate noise, however, differs as both networks feature different gates. To analyse this influence, the DQNN and the QAOA are trained for $n_{\text{train}} = n_{\text{test}} = 4$ states using different gate noise levels. Here, the gate noise is approximated by a depolarising error channel [66]. The state of the qubit after applying a noisy gate $\tilde{U}$ is given by

$$\tilde{U}\rho\tilde{U}^\dagger = p^U \frac{\mathbb{1}}{2} + (1 - p^U) U\rho U^\dagger \tag{D.1}$$

where U is the noiseless gate, ρ the qubit's initial state, and p the depolarisation probability. With probability p^U the qubits are replaced by a completely mixed state $\mathbb{1}/2$. With probability $(1 - p^U)$, the gate is applied to the qubit. The depolarisation probability of a gate $g \in \mathcal{G}_{\text{IBM}}$ (2.23) is parameterised by a scaling factor k: $p^g = kp_0^g$. p_0^g is the initial depolarisation probability chosen to match current NISQ device noise levels. Here, $p_0^{\text{CNOT}} = 3.14 \times 10^{-2}$, $p_0^{\text{SX}} = 1.18 \times 10^{-3}$, $p_0^{\text{RZ}} = 0$ are chosen, which is an approximation for *ibmq_ 16_ melbourne* [62]. Additionally, *ibmq_ 16_ melbourne's* coupling map is chosen for the transpilation.

The results of this analysis are shown in Fig. D.1. The training, test, and identity cost is plotted versus the error probability factor k. k ranges from 0 to 4.

$k = 0$ simply corresponds to the noiseless training. Both networks reach a training and test cost of nearly 1. The identity cost is 1 since there is no noise involved. The identity cost is decreasing with increasing noise scaling k. The case $k = 1$ corresponds to current NISQ device noise levels. It can be compared to the results of section 5.3.2 for $n_{\text{test}} = 4$. The range $k \in [0,1]$ greatly visualises the opportunities of future improved quantum computers. As k is growing bigger, so is the influence of the noise. Remarkably, both networks can achieve training and test costs close to the identity cost, no matter the noise level. The DQNN's costs are generally higher than the QAOA's. This proves the assumption of section 5.3.2 that the DQNN is less susceptible to gate noise. In general, this can be explained by the fact that the DQNN's transpiled quantum circuit has a smaller width than the QAOA's. The noise level is smaller due to the fewer gates that are involved in the QNN evaluation. The difference between both networks is growing bigger with increasing k. At some point, however, the difference converges to zero. At this point, the noise level is completely dominating the signal. Thus, the network's different parameters do not have a significant influence on the resulting fidelities.

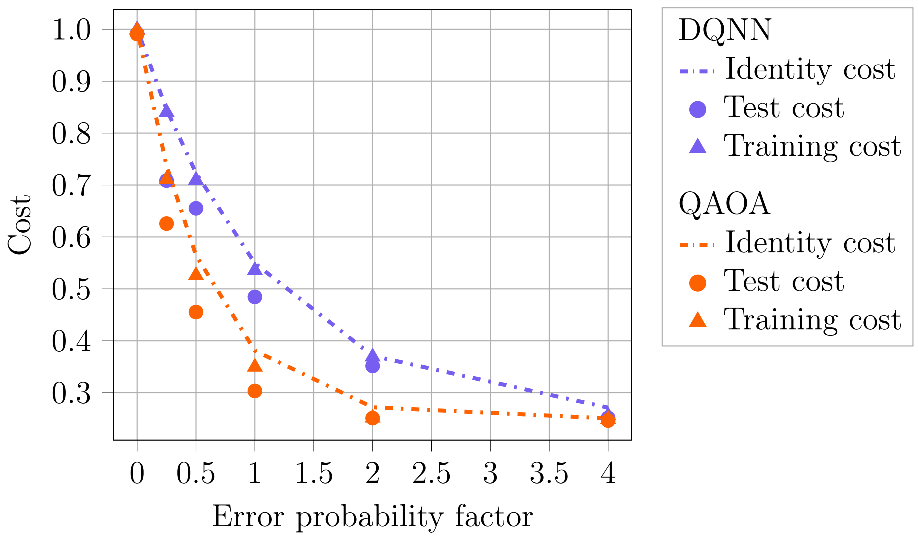


Figure D.1: Gate noise analysis. The 2-3-2 DQNN (two-qubit QAOA) using the qubit coupling map of *ibmq_16_melbourne* for $\epsilon = 0.25$, $\eta = 0.5$ ($\epsilon = 0.05$, $\eta = 0.05$).

Appendix E

Supplementary Materials to Chapter 6

E.1 Alternating cost functions

Another way of improving the generalisation can be defined very similar to the delayed normalised graph-based cost function. Here, the training with *alternating cost functions* is considered. Every epoch $x = 0, \dots, n_{\text{epoch}}$, the training cost function switches between the supervised cost function $C = C_{\text{SV}}$ and the cost function including the normalised graph-based cost function $C = C_{\text{SV}} + \tilde{\gamma}(0)C_{\text{G}}$ (see (6.11)). In summary, the new training cost function is controlled by the current training epoch:

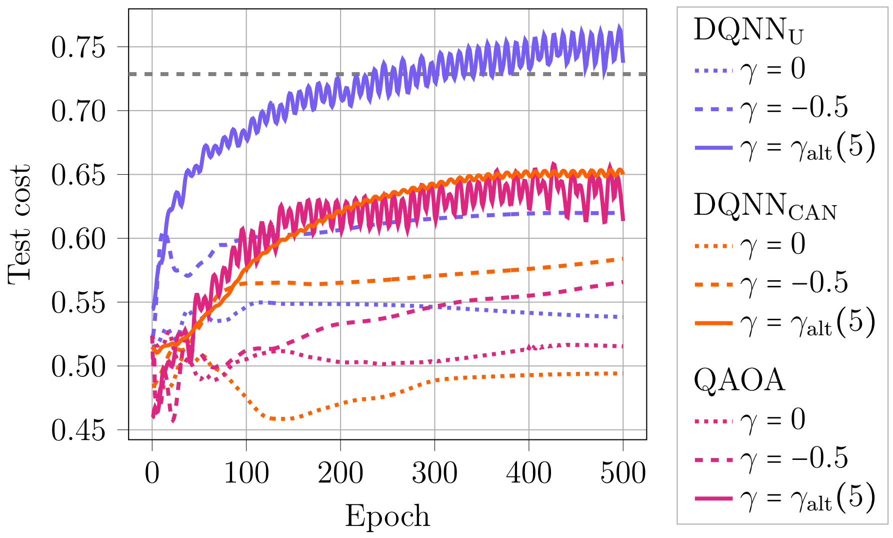
$$\begin{aligned} C(x; x_{\text{alt}}) &= \begin{cases} C_{\text{SV}}(x), & \text{if } x \bmod 2x_{\text{alt}} < x_{\text{alt}} \\ C_{\text{SV}}(x) + \tilde{\gamma}(0)C_{\text{G}}(x), & \text{if } x \bmod 2x_{\text{alt}} \geq x_{\text{alt}} \end{cases} \\ &= C_{\text{SV}}(x) + \gamma_{\text{alt}}(x_{\text{alt}})C_{\text{G}}(x), \quad \gamma_{\text{alt}}(x_{\text{alt}}) = \begin{cases} 0, & \text{if } x \bmod 2x_{\text{alt}} < x_{\text{alt}} \\ \tilde{\gamma}(0), & \text{if } x \bmod 2x_{\text{alt}} \geq x_{\text{alt}} \end{cases} \end{aligned} \tag{E.1}$$

where x_{alt} specifies the number of epochs after which the cost function is changed. In different words, if $x \bmod x_{\text{alt}} = 0$, then change γ to 0 if it previously was $\tilde{\gamma}(0)$, otherwise change it to $\tilde{\gamma}(0)$.

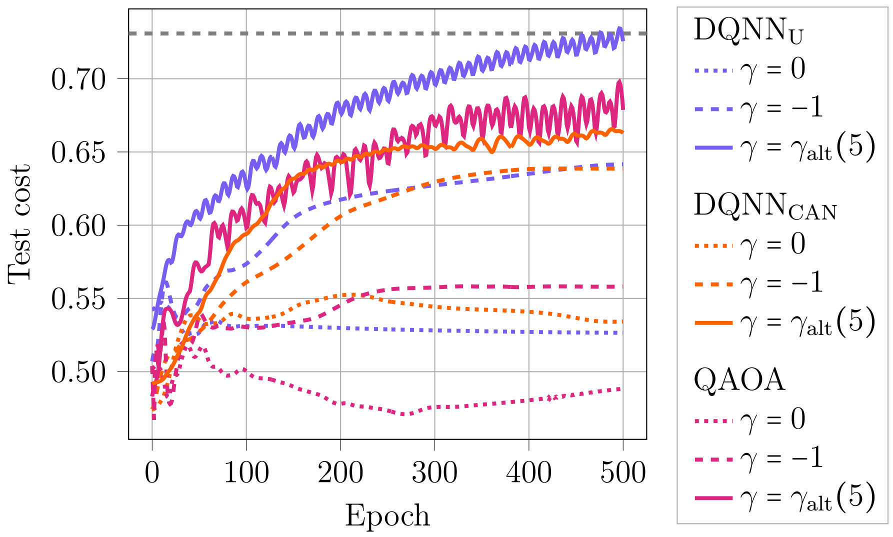
The test costs while training the DQNN$_{\text{U}}$, DQNN$_{\text{CAN}}$, and the QAOA for 500 epochs using the supervised cost function ($\gamma = 0$), the training cost ($\gamma = -0.5$ and $\gamma = -1$), and alternating cost functions ($\gamma = \gamma_{\text{alt}}(5)$) are shown in Fig. E.1. It features a plot for the connected clusters training data (Fig. E.1a) and the connected line training data (Fig. E.1b). The alternation of the cost functions is clearly visible in the test costs of $\gamma = \gamma_{\text{alt}}(5)$ in Figs. E.1a and E.1b as the supervised and graph-based cost functions have different objectives. For both examples, alternating the cost functions improves the generalisation of every QNN. The resulting test costs are comparable to the ones obtained via the training using the delayed normalised graph-based cost function.

Proving the universality of this improvement can be done analogously to section 6.3.4. The DQNN_{CAN} is trained for 500 epochs using alternating cost functions for $n_{\text{SV}} = 1, \ldots, 7$ randomly selected supervised states from the connected clusters and connected line training data. This is repeated ten times to average over different start parameters, input states, and supervised training states.

The test costs after training the DQNN_{CAN} using the supervised cost function ($\gamma = 0$), the cost function ($\gamma = -0.5$ and $\gamma = -1$), and alternating cost functions ($\gamma = \gamma_{\text{alt}}(5)$) for $n_{\text{SV}} = 1, \ldots, 7$ supervised states is shown in Fig. E.2. Surprisingly, the improvement of the DQNN_{CAN}'s generalisation for the selected supervised states (see Fig. E.1) cannot be generalised for arbitrary supervised states. The test costs after training the DQNN_{CAN} using alternating cost functions show no consistent increase compared to $\gamma < 0$. A significant improvement can only be found for a few numbers of supervised states. For comparison, the delayed normalised graph-based cost function shows a more reliable improvement (see Fig. 6.10).

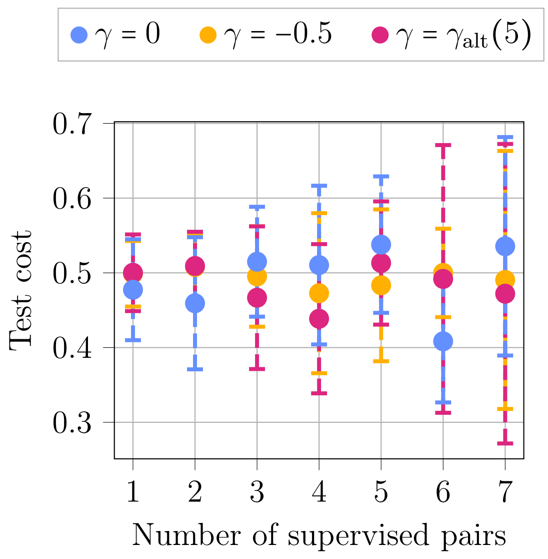


(a) The connected clusters training data (see section 6.2.2).

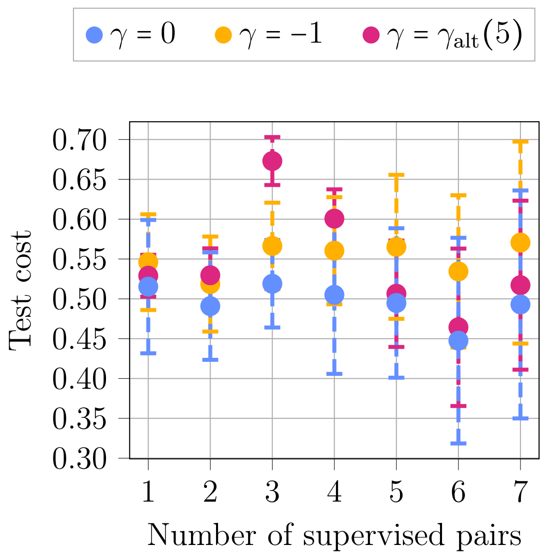


(b) The connected line training data (see section 6.2.3).

Figure E.1: The test cost while training the DQNN$_U$, the DQNN$_{CAN}$, and the QAOA for 500 epochs using the supervised cost function ($\gamma = 0$), the untouched training cost ($\gamma = -0.5$ and $\gamma = -1$), and alternating cost functions ($\gamma = \gamma_{alt}(5)$). Additionally the test cost after training the QNN described in [63] is marked as - - -.



(a) The connected clusters training data (see section 6.2.2).



(b) The connected line training data (see section 6.2.3).

Figure E.2: Training the DQNN_{CAN} using alternating cost functions for different number of supervised states.

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my supervisor Tobias Osborne for the countless instructive discussions, helpful teachings, and guidance throughout the entire project. He never failed to amaze me with his immense knowledge and intuition.

Also, I want to thank Kerstin Beer. Without her work, this thesis would not have been possible. She was a great help along the way, and I highly appreciate the time she invested in aiding me.

I do not want to miss thanking my co-supervisor Reinhard Werner and the group members of the QIG. I gratefully acknowledge the many seminars and meetings we had.

Of course, I have to thank Gabriel Müller for being my friend and colleague in these past years. I greatly enjoyed our countless Zoom sessions, endless discussions, and chess games. I also want to thank Simon Bittner for the fun distractions.

Lastly, I would like to thank my family and my girlfriend, who were always on my side and have supported me ever since.

I acknowledge the use of IBM Quantum services for this work. The views expressed are those of the author, and do not reflect the official policy or position of IBM or the IBM Quantum team.