

# Parameterized Quantum Circuits for Machine Learning

by  
Kristian Wold

**Thesis**  
for the degree of  
**Master of Science**



Department of Physics  
Faculty of Mathematics and Natural Sciences  
University of Oslo  
September 2021

This master's thesis is submitted under the master's program *Computational Science*, with program option *Physics*, at the Department of Physics, University of Oslo. The scope of the thesis is 60 credits.

© Kristian Wold, 2021

[www.duo.uio.no](http://www.duo.uio.no)

Print production: Reprosentralen, University of Oslo

# Abstract

Can quantum computers be used for implementing machine learning models that are better than traditional methods, and are such methods suitable for today's noisy quantum hardware? In this thesis we made a Python framework for implementing machine learning models based on parameterized quantum circuits that are evaluated on quantum hardware. The framework is capable of implementing quantum neural networks (QNNs) and quantum circuit networks (QCNs), and train them using gradient-based method. To calculate the gradient of quantum circuit networks, we developed a backpropagation algorithm based on the parameters shift rule that leverage both classical and quantum hardware. We performed a numerical study where we sought to characterize how dense neural networks (DNNs), QNNs and QCNs behave as a function of model architecture. We focus on investigating the vanishing gradient phenomenon, and quantifying the models trainability and expressivity using the empirical fisher information matrix (EFIM) and trajectory length, respectively. We also test the performance of the models by training them on artificial data, as well as on real-world data sets.

Due to the multi-circuit nature of QCNs, large models can be constructed by using multiple layers of small circuits. For shallow circuits with few qubits, the local gradients of the individual circuits can be easily estimated on noisy quantum hardware. This is contrary to single-circuit QNNs that are deep and consist of many qubits, whose gradient is difficult to estimate on quantum hardware due to the vanishing gradient phenomenon. However, when the gradients of QCNs are calculated with backpropagation on classical hardware using the local gradients, the gradient tends to vanish exponentially fast as the number of layers increase. We showed that the vanishing gradient of QCNs manifests itself as a loss landscape that is very flat in most directions, with strong distortions in a single direction. This characteristic loss landscape is typical for DNNs, and is known to cause slow optimization. However, for a conservative number of qubits and layers, QCNs had significantly less distorted loss landscape than similar DNNs. We also showed that during training of QCNs and DNNs, the former models required two orders of magnitude fewer epochs in order to become exponentially expressive. Finally, we showed that QCNs of few qubits and layers trained faster than both DNNs and QNNs on the artificial data. QCNs also trained and generalize better on some real-world data sets, using both ideal and noisy simulation. This shows that QCNs may have merit for some data sets, even on noisy hardware, but not all.

# Acknowledgements

I would first like to thank my supervisor, Morten Hjorth-Jensen, for providing me outstanding support, guidance and inspiration. You are always available for a talk about anything, anytime. For that, I am truly grateful.

I would like to thank my co-supervisor, Stian Bilek, for being a mentor and friend. Thank you for all the lengthy discussions we have and the invaluable insight you have help me obtain.

From the deepest of my heart, I would like to thank my family and friends, for their tireless support during hard times. Without you, this thesis would not be possible. I would also like to extend a special thanks to my brother, Anders Wold-Dobbe, for giving excellent insight into scientific writing and help with correction.

I would like to thank Overwatch Squad, our own discord server of friends, for helping me through these times of isolation due to the pandemic. Thank you for all the late nights and long talks. A special thanks goes to my friend, Nicolai Haug, for always being available when times are difficult (for example when my LaTeX document crashes.)

Lastly, I would like to thank the computational physics group, for creating such a friendly environment and pleasant place to be. You have been very important to me, both academically and socially.

**✶ Kristian Wold**  
Oslo, September 2021

# Abbreviations

NISQ	Noisy Intermediate-Scale Quantum (Technology)
PQC	Parameterized Quantum Circuit
QNN	Quantum Neural Network
QCN	Quantum Circuit Network
DNN	Dense Neural Network
EFIM	Empirical Fisher Information Matrix
PCA	Principal Component Analysis

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>ii</b>
<b>Abbreviations</b>	<b>iii</b>
<b>1 Introduction and Objective of the Study</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.1.1 Machine Learning . . . . .	1
1.1.2 Quantum Computing . . . . .	2
1.1.3 Quantum Machine Learning . . . . .	3
1.2 Objectives . . . . .	5
1.3 The Organization of the Thesis . . . . .	6
<b>I Theoretical Background</b>	<b>7</b>
<b>2 Supervised Learning</b>	<b>8</b>
2.1 Parametric Models . . . . .	9
2.1.1 Regression . . . . .	9
2.1.2 Classification . . . . .	10
2.2 Optimization . . . . .	11
2.2.1 Batch Gradient Descent . . . . .	11
2.2.2 Adam Optimizer . . . . .	13
2.3 Dense Neural Network . . . . .	14
2.3.1 Feedforward . . . . .	14
2.3.2 Backpropagation . . . . .	15
2.3.3 Activation Functions . . . . .	16
2.3.4 Saturated Activations and Vanishing Gradient . . . . .	17
2.4 Generalizability . . . . .	18
2.5 Pre-processing Data . . . . .	19
2.5.1 Scaling Features . . . . .	19
2.5.2 Principal Component Analysis . . . . .	20
<b>3 Quantum Computing</b>	<b>22</b>

3.1	States in Quantum Mechanics . . . . .	22
3.1.1	The Qubit . . . . .	23
3.1.2	Multiple Qubits . . . . .	23
3.1.3	Measuring Qubits . . . . .	24
3.2	Quantum Circuits . . . . .	24
3.2.1	Single Qubit Operations . . . . .	25
3.2.2	Multi-Qubit Operators . . . . .	28
3.2.3	Observables . . . . .	30
3.2.4	Expectation Values . . . . .	31
3.2.5	Estimating Expectation Values . . . . .	31
3.3	Noisy Intermediate-Scale Quantum Computing . . . . .	33
3.3.1	Gate Fidelity . . . . .	33
3.3.2	Quantum Decoherence . . . . .	33
3.3.3	Coupling of Qubits . . . . .	34
3.3.4	Basis Gates . . . . .	34
<b>4</b>	<b>Quantum Machine Learning</b>	<b>35</b>
4.1	Quantum Neural Networks . . . . .	36
4.2	Feature Encoding . . . . .	37
4.2.1	Qubit Encoding . . . . .	37
4.2.2	RZZ Encoding . . . . .	38
4.2.3	Latent Qubits . . . . .	39
4.3	Ansatz . . . . .	39
4.4	Model Output . . . . .	40
4.5	Optimization of PQC . . . . .	41
4.5.1	Analytical Gradient-Based Optimization . . . . .	41
4.5.2	Barren Plateaus in QNN Loss Landscape . . . . .	43
4.6	Quantum Circuit Network . . . . .	44
4.6.1	Feed-Forward . . . . .	44
4.6.2	Backward Propagation . . . . .	45
<b>5</b>	<b>Tools for Analysis</b>	<b>48</b>
5.1	Trainability . . . . .	48
5.1.1	Hessian Matrix . . . . .	48
5.1.2	Empirical Fisher Information Matrix . . . . .	49
5.2	Expressivity . . . . .	50
5.2.1	Trajectory Length . . . . .	50
<b>II</b>	<b>Implementation</b>	<b>53</b>
<b>6</b>	<b>Implementation</b>	<b>54</b>
6.1	Qiskit . . . . .	54
6.1.1	Registers and Circuits . . . . .	55
6.1.2	Applying Gates . . . . .	55

6.1.3	Measurement	56
6.1.4	Exact Expectation Value	56
6.1.5	Simulating Real Devices	57
6.2	QNN Example	58
6.2.1	Encoding	58
6.2.2	Ansatz	59
6.2.3	Model Output	59
6.2.4	Gradient	60
6.2.5	Training	61
6.3	Quantum Circuit Network	62
6.3.1	Encoders, Ansatzes and Samplers	62
6.3.2	QLayer	63
6.3.3	Constructing QCNs from QLayers	64
6.3.4	Backpropagation	65
6.3.5	Training	65
6.3.6	Single-Circuit Models	66
6.3.7	Dense Neural Networks	66
6.3.8	Hybrid Models	67
6.4	Tools for Analysis	68
6.4.1	Magnitude of Gradients	68
6.4.2	Empirical Fisher Information	69
6.4.3	Trajectory Length	70
6.5	Numerical Experiments	70
6.5.1	Initialization	70
6.5.2	Pre-processing Data	71
6.5.3	Optimization	72
6.5.4	Configuring QCNs and DNNs	72

### III Results & Discussion

73

#### 7 Results and Discussion

74

7.1	Vanishing Gradient Phenomenon	74
7.1.1	Vanishing Gradient in QNNs	74
7.1.2	Vanishing Local Gradient in QCNs	76
7.1.3	Vanishing Total Gradient in QCNs	77
7.1.4	Discussion	78
7.2	Investigating the Loss Landscape	79
7.2.1	Discussion	82
7.3	Expressivity	83
7.3.1	Untrained Models	83
7.3.2	Trained Models	86
7.3.3	Single Node Expressivity	88
7.3.4	Discussion	89
7.4	Training Models on Mixed Gaussian Data	89



7.4.1	Ideal Simulation . . . . .	90
7.4.2	Noisy Simulation . . . . .	92
7.4.3	Discussion . . . . .	93
7.5	Real-World Data . . . . .	95
7.6	Discussion . . . . .	96
<b>IV</b>	<b>Conclusion &amp; Future Research</b>	<b>97</b>
<b>8</b>	<b>Summary &amp; Conclusions</b>	<b>98</b>
8.1	Summary & Conclusions . . . . .	98
8.2	Future Research . . . . .	101
	<b>Appendices</b>	<b>103</b>
<b>A</b>	<b>Data Sets</b>	<b>103</b>
A.1	Mixed Gaussian Data . . . . .	103
A.2	Real Data . . . . .	105
A.2.1	Boston Housing Data . . . . .	106
A.2.2	Breast Cancer Wisconsin Data . . . . .	106
A.2.3	Feature Reduction with PCA . . . . .	107
	<b>References</b>	<b>108</b>

# 1

## Introduction and Objective of the Study

### 1.1 Introduction

#### 1.1.1 Machine Learning

*Machine learning* is a highly successful field of study involving algorithms that allow computers to solve problems using data, relieving the need for tailoring problem-specific solutions [1]. This practice has transformed nearly every aspect of our modern society, from medicine [2] to finance [3]. One branch of machine learning is *supervised learning*, which is the practice of training a model to learn a relation between input and output data [4]. Typically, one starts by acquiring a *training set of labelled data*, which is a collection of pairs of inputs and outputs. As an example, the inputs and outputs can be *age* and *salary* of people, respectively. By training a machine learning model on the training set, the aim is that it "learns" the general relation between *age* and *salary*. If this is the case, the model can be used to predict the salary of people based on their age, even for values of *age* not present in the training set. If the latter is the case, the model is said to generalize to unseen data, which is required for prediction. How are machine learning models trained? One often starts by defining a *loss function* (also commonly known as *risk* or *cost*), which is a scalar function that measures how accurately a model predicts the outputs from the corresponding inputs [4]. The lower the value of the loss function, the better the model reproduces the outputs. Therefore, training a model involves minimizing the loss function with respect to the training set.

A particular powerful family of machine learning models are *neural networks* (NN). Neural networks are models consisting of layers of artificial neurons, originally inspired by the neural structures in the brain [5], that sequentially transform the

inputs it is fed. They are *parametric* models, meaning that the input-output relation that an NN computes are determined by a set of real-valued parameters. When setting up a NN, the goal of the training is to find the correct parameters such that the given loss function is minimized. This is done by first calculating the derivative of the loss function with respect to the parameters of the NN. This derivative is called the *gradient*, which quantifies how the loss function changes when the parameters are adjusted. Using gradient-based methods, such as gradient descent, the gradient can be utilized to adjust the parameters such that the loss decreases [5]. The *backpropagation algorithm*, tailored for accommodating their layered structure, is commonly used for calculating the gradient of NNs [5].

What kind of functions can a NN compute, and how is this affected by the number of neurons and layers present in the model? Quantifying the flexibility of NNs and relating this to how complex data the models can learn is considered a difficult problem [6]. In an effort to address these questions, Raghu et al. [6] introduced a heuristic called *trajectory length* for quantifying the flexibility, or *expressivity*, of NNs. To calculate the heuristic, a one dimensional trajectory of input data is fed to a neural network. As the trajectory is transformed by each layer, its length is calculated at each step. The authors found that the trajectory length increase exponentially with each transformation, suggesting that NNs with many layers can compute (and learn) functions that are exponentially complex.

Even though additional layers increase the expressivity of NNs, this comes with a drawback: with increasing number of layers, the *vanishing gradient phenomenon* emerges, meaning the magnitude of the gradient decreases exponentially [7]. This phenomenon manifests itself as a loss function that is insensitive to adjustments of the parameters, known as a flat *loss landscape*, making the training of many-layered NNs difficult [8]. To uncover the geometry of the loss landscape, and determine its flatness, it is common practice to assess the spectrum of the *empirical fisher information matrix* (EFIM) [8].

### 1.1.2 Quantum Computing

*Quantum computing* is the processing of information using systems that obey the laws of quantum mechanics [9]. In 1982, Richard Feynman pointed out that quantum mechanical systems are notoriously difficult to simulate on classical computers. He suggested that this complexity can be exploited by build a computer based on the principles of quantum mechanics [9]. Only three years later, David Deutsch formalized a theory describing such a device, a *universal quantum computer* [10]. Even though such a device was not yet realized physically, people started developing algorithms for quantum computers that was theorized to be more efficient than their classical counterparts. In 1996, physicist Seth Lloyd showed that quantum mechanical systems could be efficiently simulated on quantum computers [11]. This is perhaps a not too surprising result, since quantum mechanics is the "native language" of quantum computers. A more surprising discovery happened two years prior, when Peter Shor developed Shor's algorithm for prime factorization of integers

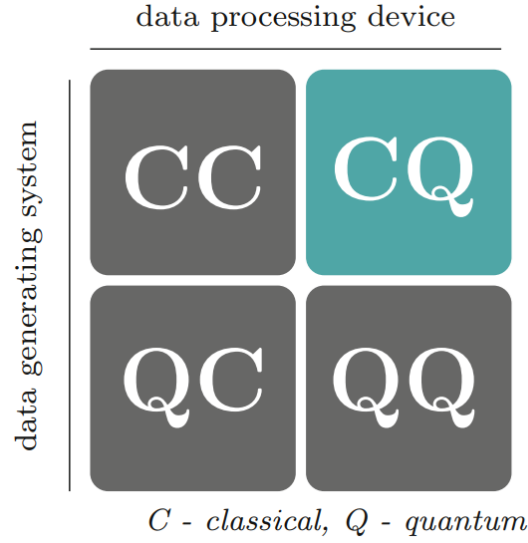
in polynomial time, potentially breaking the secure encryption protocols of today [12]. Interestingly, this is a type of problem which does not spring naturally from the realm of quantum mechanics. Prime factorization is believed to be exponentially hard on classical computers, and the effectiveness of Shor's algorithm shows that the capabilities of quantum computers go beyond the simulation of quantum mechanical systems [12]. This has sparked a huge interest in mapping out what type of problems quantum computing can excel at.

Today, there is a lot of focus on making quantum computers, with big companies such as Google and IBM at the forefront. Despite the effort, today's quantum computers are not able to implement useful quantum algorithms that would change the world right away. This is because today's quantum computers are small, typically supporting only tens of *qubits* [13], the quantum mechanical analog of the classical bit. In addition, the computations tends to be very noisy [14]. As it performs an algorithm, also called a *circuit*, it manipulates a very delicate quantum state. During the span of the computation, the state is susceptible to interference from the surrounding environment, causing the information of the system to degrade. This phenomenon is called *decoherence*, and puts a strict limit on the *circuit depth* [15], which can be thought of as the time it takes to execute the circuit. The limited size of hardware and presence of noise makes many highly anticipated algorithms, such as Shor's algorithm, unfeasible to implement today [13].

### 1.1.3 Quantum Machine Learning

Could quantum computing be useful for developing algorithms for machine learning, and could they be useful on the noisy quantum hardware of the near future? By combining machine learning and quantum computing, you get the emerging interdisciplinary field *quantum machine learning*. Just from the name, it is not immediately obvious what it might entail, and as matter of fact, it depends on the context. From Figure 1.1, we see the different ways machine learning and quantum computing can be combining. The CC case refers to classical data that is processed on classical devices, which is of course the traditional form of machine learning, e.g. NNs. The other case, CQ, investigates how classical data can be processed with help of quantum computers. These are the two cases we will focus on in this thesis.

Lately, there has been many proposed methods for implementing machine learning using quantum computers. One of the promising candidates is parameterized quantum circuits (PQC) used for machine learning. Parameterized quantum circuits are a family of quantum algorithms that construct a quantum state based on input data and parameters [13]. Due to the algorithm's parametric nature, it is often called a *quantum neural network* (QNN). Unlike algorithms that are tailored for solving specific problems, like Shor's algorithm, QNNs use data to learn a specific set of parameters that produce a solution to a problem. During training of such algorithms, quantum computers are used to evaluate the circuit, while classical computers are utilized to update the parameters. By leveraging both



**Figure 1.1:** Four approaches that combine machine learning and quantum computing. The figure is retrieved from Schuld and Petruccione [1].

classical and quantum computation, the quantum algorithms involved can be kept relatively small. Because of this, it is believed that QNNs are perfect candidates for implementation on noisy, near-term quantum hardware [16]. Quantum neural networks have already been used to solve several problems in supervised learning [13, 17, 18]. Abbas et al. [17] showed that a QNN could be trained to distinguish between different plants in the Iris data set. They also showed that their model trained faster and was more flexible than NNs with the same number of parameters, even when trained on today's noisy quantum computers.

QNNs and other methods for quantum machine learning have been shown to outperform traditional methods for prediction on some data sets [17, 19]. Still, McClean et al. [20] point out that many of these studies rely on heuristics, and that there are few rigorous proofs for their performance when used on larger learning problems. Further, they showed that a large family of QNNs suffer from similar problems as NNs: When the model size of QNNs are increased, e.g. by increasing the number of parameters and inputs, their gradients tend to vanish exponentially fast. This is the same behaviour as when the number of layers of NNs are increased, and cause QNNs to become intractable to train when scaled up to handle larger problems. A vanishing gradient is an even bigger problem when the QNN is trained on a noisy quantum computer, as this results in a bad signal-to-noise ratio for the gradient. Consequently, the optimization of the QNN fails to converge [21].

Is it possible to implement a quantum machine learning model that is scalable, fast to train, and performs well on today's noisy quantum computers? Bilek [22] introduced a multi-circuit model for machine learning that utilizes several QNNs to sequentially transform data. This model was later dubbed a *quantum circuit*

*network* (QCN), and corresponds closely to the layered structure of NNs where each node is a parametric circuit. While he showed that the QCN was able to train on and reproduce nonlinear 1D data, very little is known about its properties. Is it possible to derive an algorithm akin to backpropagation to calculate its gradient? How does its expressivity change as the model is scaled up? Does it suffer from a vanishing gradient? How does it perform on noisy quantum hardware? These are some of the questions we wish to explore in this thesis. As of now, we are preparing an article we hope to soon publish that includes many of the results and findings in this thesis [23].

## 1.2 Objectives

The overall objective of this study is to implement and investigate quantum circuit networks (QCNs), and characterize their behavior as a function of their architecture. This will be done by using various numerical methods. This investigation is then repeated on traditional methods, such as dense neural networks (DNNs) and quantum neural networks (QNNs). Finally, the mentioned models are benchmarked and compared against each other on artificial and real-world data, using both ideal and noisy simulation of quantum hardware. The main goals can be divided into the following seven points:

- Implement a Python framework for constructing quantum neural networks (QNNs), dense neural networks (DNNs) and QCNs.
- Develop a backpropagation algorithm based on the parameter shift rule for calculating the gradient of QCNs, allowing for gradient-based optimization with respect to a loss function.
- Investigate the vanishing gradient phenomenon for QCNs by calculating the magnitude of their gradients for different numbers of qubits and layers. Repeat this investigation for QNNs and DNNs and compare.
- Study the loss landscape for QNNs, DNNs and QCNs by calculating the spectrum of their empirical fisher information matrix (EFIM), and assess how trainable the different models are for different architectures.
- Assess the expressivity of QCNs and DNNs using the trajectory length metric, for both untrained and trained models, and compare.
- Train QNNs, DNNs and QCNs on mixed Gaussian data and assess whether the EFIM spectrum and trajectory length are good predictors for rate of optimization and expressivity of the models, respectively. Also train QCNs and QNNs using simulation of noisy hardware to study how they perform on real hardware.

- Train QCNs and DNNs on real-world data and compare how well they generalize to unseen data to see if they have any merit for practical problems, both on ideal and noisy quantum hardware.

### 1.3 The Organization of the Thesis

In part I of this thesis, we will introduce the theoretical background. We go through the theory of supervised learning and optimization of parametric models in [Chapter 2](#). In [Chapter 3](#), we present the fundamentals of quantum computing. Quantum machine learning, the intersection of the two former disciplines, is presented in [Chapter 4](#). We present the various numerical methods for analyzing the models implemented in this thesis in [Chapter 5](#).

In part II, we go through aspects surrounding the implementation of various models and methods used in this thesis. These are implemented using Python together with Qiskit [\[24\]](#), an IBM made open-source Python framework for quantum computing. We present details of various experiments and subsequent results and discussion in part III. We finally conclude our analysis and discuss future studies in part IV. In [Appendix A](#), we detail the various data sets used in this thesis.

# Part I

## Theoretical Background



# 2

## Supervised Learning

This chapter introduces the fundamentals of supervised learning, optimization and neural networks (NNs). The content of this chapter is mainly based on the material in Schuld and Petruccione [1], Hastie et al. [4] and Nielsen [25].

The goal of *supervised learning*, one of the big branches of machine learning, is to obtain a model for predicting an output  $y$  from an input  $\mathbf{x}$ . This is done by learning from input-output pairs  $\mathcal{T} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})\}$ , known as the training set. Here, we assume that  $\mathbf{x}^{(i)}$  is a vector and  $y$  is a scalar. The domain of the input and output depends on the specific learning problem. The output  $y$ , also called the target or the response, is often either of a quantitative or qualitative character. These two cases constitute two big paradigms in supervised learning: *regression* and *classification*, respectively. For regression, the goal of the learning task is to predict a real-valued target  $y$  from the input  $\mathbf{x}$ . Typical examples of targets to regress on are *temperature*, *weight* and *length*, which have in common a natural notion of distance measure in the sense that instances close in numerical value are also similar in nature. E.g., two fish weighing 12.1 kg and 12.2 kg are similar, while a third fish weighing 24.0 kg is notably different.

For classification, the goal is to predict one or more *classes* from an input  $\mathbf{x}$ . In this setting, the target  $y$  is discrete and categorical, such as *color*, *dead/alive* and *animal species*. In contrast to quantitative targets, qualitative targets lack a natural distance measure, in the sense that it is not meaningful to compare the distance between *dog* and *cat*, and *dog* and *seagull*. They are simply mutually exclusive classes.

The input  $\mathbf{x}$  is a vector consisting of elements  $(x_1, \dots, x_p)$  often called features or predictors. Each feature  $x_i$  can either be quantitative or qualitative in the same

manner as with the target previously discussed. In this thesis, we will investigate quantitative features  $\mathbf{x} \in \mathbb{R}^p$ .

## 2.1 Parametric Models

The approach of supervised learning often starts by acquiring a training set  $\mathcal{T} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})\}$ , where  $N$  is the number of samples in the training set. This is called labeled data, since the samples of features  $\mathbf{x}^{(i)}$  are accompanied by the ground truth target  $y^{(i)}$  that we want to predict. One often hypothesizes that the acquired training data was produced by some mechanism or process that we can mathematically express as

$$y = f(\mathbf{x}) + \epsilon, \quad (2.1)$$

where  $\epsilon$  is often included to account for randomness, noise or errors in the data, in contrast to the deterministic part  $f(\mathbf{x})$ . Depending on the context, the  $\epsilon$  may be neglected or assumed to be normally distributed such as  $\epsilon \sim \mathcal{N}(0, \sigma^2)$ , where  $\sigma^2$  is the variance.

The goal is to approximate the underlying mechanism  $f(\mathbf{x})$ . To do this, one often proposes a parametric model

$$\hat{y} = \hat{f}(\mathbf{x}; \boldsymbol{\theta}),$$

where  $\hat{y}$  is the predicted value,  $\hat{f}(\cdot; \cdot)$  defines a family of models, and  $\boldsymbol{\theta}$  is a vector of parameters that defines a specific model from that family. Training (or fitting) the model involves finding the parameters  $\boldsymbol{\theta}$  such that the model best reproduces the targets from the features found in the training data set. To quantify what is meant by "best" in this context, it is common to introduce a *loss function* that measures the quality of the model with respect to the training data set:

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N L(\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}), \quad (2.2)$$

The loss function returns a scalar value that indicates how good your model fits the training data for a particular set of parameters. In general, a lower value indicates a better model. This formulates the task of training the model as an optimization problem. In the next section, we will discuss different ways of training parameterized models, in particular with the use of gradient-based methods.

### 2.1.1 Regression

The choice of loss function is highly problem dependent, and there is a vast collection of different choices in the machine learning literature [4]. A common loss function

used for training supervised models on regression problems is the Mean Squared Error (MSE). This loss function is suitable since it implements a natural distance measure between prediction and target. It is formulated as

$$MSE = \frac{1}{N} \sum_{i=1}^N (\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) - y^{(i)})^2. \quad (2.3)$$

From this formulation, we see that the closer the predicted targets  $\hat{y}^{(i)} = \hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$  are to the real targets  $y^{(i)}$ , the smaller the MSE will be. In other words, a model with lower MSE than some other model fits the data better. Fitting the model using MSE as loss function is often referred to as the *least squares approach*.

### 2.1.2 Classification

In this thesis, we will be concerned with *binary classification*, where the targets of the data are one of two classes that we want to predict. Typically, the different classes are represented by discrete values, such as  $y \in \{0, 1\}$ . Here 0 and 1 corresponds to the first and second class, respectively. When parametric models are trained on discrete targets, by for example by minimizing the MSE loss, they tend to produce output values in the range  $\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) \in [0, 1]$ . The continuous value of  $\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$  are often interpreted as the probability that sample  $\mathbf{x}^{(i)}$  belongs to the second class. As an example, say  $\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) = 0.8$  for a particular sample  $\mathbf{x}^{(i)}$ . This tells us that it belongs to the second class with 80% probability, and to class 0 with 20% probability. A class can then be predicted from the sample by implementing a *threshold value*  $c$ , such as

$$\hat{y}^{(i)} = I(\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) > c), \quad (2.4)$$

where  $I()$  returns one if  $\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}) > c$  is true, and otherwise zero. Typically, a threshold value  $c = 0.5$  is used, as this causes the most probable class to be picked

Whereas MSE can be used to assess how closely regression models fits the data, a more suitable metric for assessing classification models is *accuracy*. The accuracy can be expressed as

$$\text{accuracy} = \frac{1}{N} \sum_{i=1}^N I(\hat{y}^{(i)} = y^{(i)}). \quad (2.5)$$

From [Equation \(2.5\)](#), we see that the accuracy of a model is the average number of targets it classifies correctly.

## 2.2 Optimization

Finding the optimal parameters  $\hat{\boldsymbol{\theta}}$  with respect to a chosen loss function  $L$  can be formulated as

$$\hat{\boldsymbol{\theta}} = \arg \min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^N L(\hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (2.6)$$

This optimization problem is generally not trivial, and depends highly on the choice of loss function and parametric model. Aside from a few exceptions, like the case of linear regression, Equation (2.6) does not generally have an analytical solution. Moreover, many popular parametric models result in non-convex optimization problems, meaning that the loss function has several local minima. In practice, such optimization problems can't be solved efficiently [26]. However, it is important to realize that an exact, or close to exact, minimization of the loss function is seldom needed or even favorable. What is ultimately interesting is whether the trained model has sufficient ability to predict. Over the years, several cheap and approximate methods for optimization have been invented to train machine learning models. We will discuss two such methods that implement gradient-based optimization.

### 2.2.1 Batch Gradient Descent

In the absence of an analytical expression that minimizes the loss function, *gradient descent* is an easy-to-implement method that iteratively decreases the loss. This is done by repeatedly adjusting the model parameters using information of the *gradient* of the loss function. The derivative of the loss function Equation (2.2) with respect to the model parameters can be calculated as

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_k} = \frac{1}{N} \sum_{i=1}^N \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial \hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial \theta_k}, \quad (2.7)$$

where  $\theta_k$  is the  $k$ 'th model parameter, and  $\hat{y}^{(i)} = \hat{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta})$ . To arrive at this expression, the chain rule was used under the assumption that the loss function  $L(\hat{y}^{(i)}, y^{(i)})$  and model output  $\hat{y}^{(i)}$  are differentiable with respect to  $\hat{y}^{(i)}$  and  $\theta_k$ , respectively. For MSE loss Equation (2.3), the derivative has the form

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_k} = \frac{2}{N} \sum_{i=1}^N (\hat{y}^{(i)} - y^{(i)}) \frac{\partial \hat{y}^{(i)}}{\partial \theta_k}. \quad (2.8)$$

Note that the derivative is calculated with respect to the entire training set, i.e. the whole *batch*, hence the name. The gradient is then constructed simply as a vector quantity containing the derivatives with respect to each model parameter:

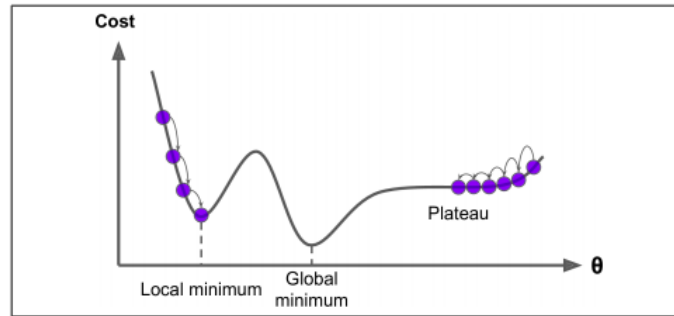
$$\nabla_{\theta} L(\theta) = \left( \frac{\partial}{\partial \theta_1} L(\theta), \dots, \frac{\partial}{\partial \theta_{n_{\theta}}} L(\theta) \right), \quad (2.9)$$

where  $n_{\theta}$  is the number of parameters. The gradient Equation (2.9) can be geometrically interpreted as the direction at point  $\theta$  in parameter space for which the value of the loss function increases most rapidly. In light of this, one can attempt to move all the parameters some small amount in the opposite direction, *the direction of steepest descent*, in order to decrease the loss. This can be done iteratively, and can be formulated as

$$\theta_t = \theta_{t-1} - \mu \nabla_{\theta} L(\theta_{t-1}) \quad (2.10)$$

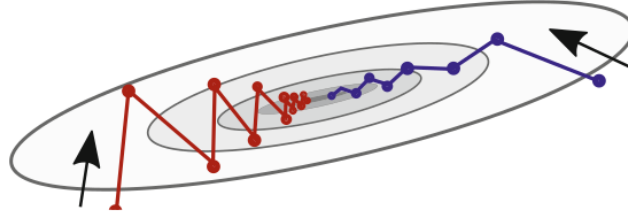
for  $t = 1, \dots, T$ . Here,  $T$  is the total number of iterations, or *epochs*, and  $\mu$  is some small positive value called the *learning rate*. Usually, some initial choice of parameters  $\theta_0$  is chosen at random. Analogous to walking down a mountain, the recalculation of the gradient and repeated adjustment of the parameters results in a gradual descent in the loss landscape. This is the heart of gradient descent.

Even though batch gradient descent is intuitively simple and sometimes sufficiently effective for training some models, it has several flaws that should be addressed when suggesting better methods of optimization. A common problem with batch gradient descent is that optimization has a tendency of getting stuck in local minima, as only local information in the loss landscape is used when updating the parameters. In addition, the presence of *plateaus*, areas of particular flatness in the loss landscape, tend to induce slow convergence. The two aforementioned phenomena are illustrated in Figure 2.1.



**Figure 2.1:** One-dimensional representation of the loss landscape for a parameterized model, showcasing the phenomenon of getting stuck in local minima, and slow convergence induced by plateaus. The figure is retrieved from Géron [5].

Furthermore, the presence of high degree of distortion in certain directions in parameter space, so called *thin valleys*, can lead to oscillations and inefficient optimization. This is exemplified in Figure 2.2. We will discuss how the popular *Adam optimizer* [27], which was used in this thesis, addresses these problems.



**Figure 2.2:** Two-dimensional representation of the loss landscape for a parameterized model, showcasing the phenomenon of "thin valleys", known to induce slow convergence due to oscillations. The blue optimizations steps incorporate momentum, which dampens the oscillations and leads to faster convergence. The figure is retrieved from Schuld and Petruccione [1].

### 2.2.2 Adam Optimizer

Introduced by Kingma and Ba [27], the Adam algorithm implements a moving average of the gradient, called *momentum*, together with a rescaling. Replacing Equation (2.9) and Equation (2.10), Adam implements the following algorithm:

---

**Algorithm 1:** *Adam*, [27]. The authors suggest default hyperparameters  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . The algorithm is applied parameter-wise.

---

$m_0 \leftarrow 0$ ;

$v_0 \leftarrow 0$ ;

$t \leftarrow 0$ ;

**while**  $\theta_t$  not converged **do**

$t \leftarrow t+1$

$g_t \leftarrow \nabla_{\theta} L(\theta_{t-1})$  (Get gradients w.r.t. loss at timestep  $t$ )

$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$  (Update biased first moment estimate)

$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$  (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \mu \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

**end**

**return**  $\theta_t$

---

Algorithm 1 updates moving averages of the gradient  $m_t$  and its square  $v_t$ , picking up information about the gradient from earlier update events. In particular, if the gradient tends to flip sign in certain directions, the averaging over previous iterations tends to dampen these oscillations. Likewise, directions of persistent sign tend to accumulate magnitude, making the optimisation gain "momentum" in these directions. This property helps the model overcoming thin valleys and plateaus. Also, the effect of momentum may also avoid getting stuck in local minima by gracing over them. Further, the moving average of the gradient and its square is

rendered unbiased by calculating  $\hat{m}_t = m_t / (1 - \beta_1)$  and  $\hat{v}_t = v_t / (1 - \beta_2)$ . Since the averages are initialized as zero, they are initially biased downward. Finally, the parameters are updated using  $\theta_t \leftarrow \theta_{t-1} - \mu \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ . Here, the rescaling term  $\sqrt{\hat{v}_t}$  serves to decrease the step size in directions where the gradient has a large magnitude and increase it where it is small. This effectively implements a variable learning rate for each direction, depending on whether big or small steps are needed.

Adam is a highly successful algorithm for optimizing machine learning models. In popular machine learning frames such as scikit-learn [28] and pyTorch [29], it is a default optimizer. Lately, Adam has also been used for optimizing quantum machine learning models [17, 21]. As pointed out by its authors, Adam requires very little tuning of hyperparameters to be efficient, making it attractive and easy to use. Adam is also suited for noisy gradients, which will be relevant for the work in this thesis.

## 2.3 Dense Neural Network

Originally inspired by the network structure of the brain [5], artificial neural networks are powerful parameterized machine learning models that have proven extremely useful for a vast number of applications. Over the years, a comprehensive collection of different network architectures has been developed to target specific problems, such as *Recurrent Neural Networks* for predicting time series data and *Convolutional Neural Networks* for image classification. In this thesis, we focus on *Dense Neural Networks* (DNNs), which is a type of simple *feedforward network*, meaning the information is processed in a forward fashion without any loops that direct information backwards.

### 2.3.1 Feedforward

Dense Neural Networks work by sequentially transforming input data by passing them through one or more *layers*, which each applies a parameterized and often nonlinear transformation. The result of the first layer of the neural network can be formulated as

$$\mathbf{a}^1 = f^1(\mathbf{z}^1) = f^1(W^1 \mathbf{x} + \mathbf{b}^1), \quad (2.11)$$

Here,  $\mathbf{x} \in \mathbb{R}^p$  is a single sample of  $p$  features.  $W^1 \in \mathbb{R}^{m \times p}$  and  $\mathbf{b}^1 \in \mathbb{R}^m$  are a matrix and a vector of parameters called the *weights* and *biases*, respectively. The operations  $W^1 \mathbf{x} + \mathbf{b}^1$  apply an affine transformation of the features resulting in  $m$  new derived features, each identified as a *node* in the specific layer. Further,  $f^1(\cdot)$  is a layer-specific function, often monotonous and nonlinear, applied element-wise on the derived features. This finally results in the output of the layer,  $\mathbf{a}^1$ , called the *activation*.

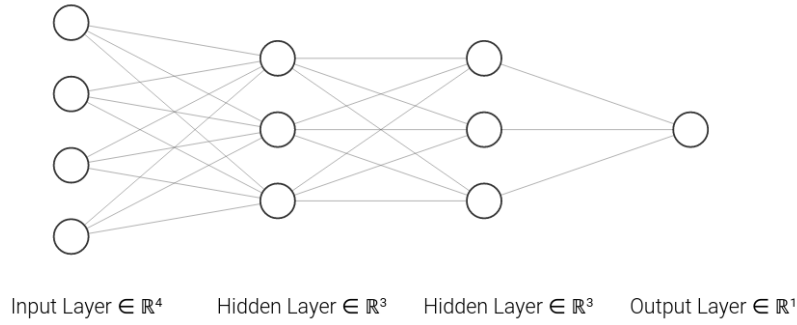
We will now generalize Equation (2.11) to an arbitrary layer. For a neural network with  $L$  layers, the feedforward procedure for layer  $l$  can be formulated as

$$\mathbf{a}^l = f^l(\mathbf{z}^l) = f^l(W^l \mathbf{a}^{l-1} + \mathbf{b}^l), \quad (2.12)$$

where  $\mathbf{a}^{l-1}$  is the activation of the previous layer with the exception  $\mathbf{a}^0 = \mathbf{x}$ . The output of the network is then the activation of the last layer, namely

$$\hat{y} = f_{DNN}(x; \boldsymbol{\theta}) = \mathbf{a}^L, \quad (2.13)$$

where  $\boldsymbol{\theta} = [W^1, \mathbf{b}^1, \dots, W^L, \mathbf{b}^L]$ . Using the recursive relation Equation (2.12), one is free to choose an arbitrary number of layers, and nodes for each layer, so long that the dimension of the input of one layer matches the shape of the output of the preceding layer. Typically, the initial input  $\mathbf{a}^0 = \mathbf{x}$  is called the input layer, while the last layer  $\mathbf{a}^L$  is called the output layer. All intermediate layers are called *hidden layers*, since we usually don't observe the internal transformations a neural network does. Figure 2.3 illustrates the connectivity of a typical DNN. Here, the DNN has four inputs, two hidden layer with three nodes each, and a single output.



**Figure 2.3:** Illustration of the connectivity of a typical DNN. Here, the DNN has four inputs, two hidden layer with three nodes each, and a single output. The connecting lines are identified as the weights  $W^l$ . This diagram was made using the NN-SVG tool [30].

Equation (2.12) and Equation (2.13) define the whole forward procedure of a neural network, and also highlight the role of the functions  $f^l(\cdot)$ , called the *activation function*. If set to identity,  $f^l(x) = x$ , the recursive application of Equation (2.12) would simply apply repeated linear operations. In other words, increasing the number of layers would not increase the expressiveness of the network, as all the layers would collapse into a single layer. Therefore, introducing nonlinear transformations is necessary to increase the flexibility of the neural network.

### 2.3.2 Backpropagation

Assume that  $\hat{y} = f(\mathbf{x}^{(i)}; \boldsymbol{\theta})$  is a dense neural network as defined by Equation (2.12) and Equation (2.13). In order to use gradient-based methods, one needs to calculate



the derivative of the loss function Equation (2.7) for an arbitrary parameter  $\theta_k$ , which could be any of the weights  $W^l$  or biases  $\mathbf{b}^l$  in the various layers. This is not trivial given the sequential structure of the neural network. Often attributed to Rumelhart et al. [31], the *backpropagation algorithm* calculates the gradient in a sequential manner, starting with the last layers first. Calculating for a single sample, the algorithm starts by calculating the *error* of the last layer

$$\delta_k^L = \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^L}, \quad (2.14)$$

where  $k$  indicates the node. This error can be defined for any layer recursively by repeated application of the chain-rule:

$$\delta_j^l = \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_j^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^{l+1}} \frac{\partial \mathbf{a}_k^{l+1}}{\partial \mathbf{a}_j^l} = \sum_k \delta_k^{l+1} \frac{\partial \mathbf{a}_k^{l+1}}{\partial \mathbf{a}_j^l}. \quad (2.15)$$

This relation is the origin of the name *backpropagation*, as the error terms  $\delta^l$  "propagate" backwards through the neural network as they are calculated.

Using that  $\frac{\partial \mathbf{a}_k^l}{\partial W_{ij}^l} = f'(\mathbf{z}_k^l) \mathbf{a}_j^{l-1} I_{ik}$  and  $\frac{\partial \mathbf{a}_k^l}{\partial \mathbf{b}_i^l} = f'(\mathbf{z}_k^l) I_{ik}$ , the derivative with respect to the weights and biases can then be calculated as

$$\frac{\partial L(\hat{y}, y)}{\partial W_{ij}^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^l} \frac{\partial \mathbf{a}_k^l}{\partial W_{ij}^l} = \sum_k \delta_k^l f'(\mathbf{z}_k^l) \mathbf{a}_j^{l-1} I_{ik} = \delta_i^l f'(\mathbf{z}_i^l) \mathbf{a}_j^{l-1}, \quad (2.16)$$

and

$$\frac{\partial L(\hat{y}, y)}{\partial \mathbf{b}_i^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^l} \frac{\partial \mathbf{a}_k^l}{\partial \mathbf{b}_i^l} = \sum_k \delta_k^l f'(\mathbf{z}_k^l) \mathbf{a}_j^{l-1} I_{ik} = \delta_i^l f'(\mathbf{z}_i^l). \quad (2.17)$$



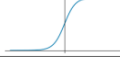



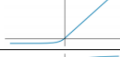




The final gradient, over all samples, is then the average of all the single-sample gradients

$$\nabla_{\theta} L(\theta) = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} L(\hat{y}^{(i)}, y^{(i)}), \quad (2.18)$$

which can be used to optimize the neural network with a gradient-based method as discussed earlier.

### 2.3.3 Activation Functions

As mentioned earlier, activation functions are important for introducing nonlinearities to the layers of DNNs. A comprehensive list can be found in Figure 2.4.

Name	Given as a function of $x \in \mathbb{R}$ by	Plot
linear	$x$	
Heaviside / step function	$\mathbb{1}_{(0,\infty)}(x)$	
logistic / sigmoid	$\frac{1}{1+e^{-x}}$	
rectified linear unit (ReLU)	$\max\{0, x\}$	
power rectified linear unit	$\max\{0, x\}^k$ for $k \in \mathbb{N}$	
parametric ReLU (PReLU)	$\max\{ax, x\}$ for $a \geq 0, a \neq 1$	
exponential linear unit (ELU)	$x \cdot \mathbb{1}_{[0,\infty)}(x) + (e^x - 1) \cdot \mathbb{1}_{(-\infty,0)}(x)$	
softsign	$\frac{x}{1+ x }$	
inverse square root linear unit	$x \cdot \mathbb{1}_{[0,\infty)}(x) + \frac{x}{\sqrt{1+ax^2}} \cdot \mathbb{1}_{(-\infty,0)}(x)$ for $a > 0$	
inverse square root unit	$\frac{x}{\sqrt{1+ax^2}}$ for $a > 0$	
tanh	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	

**Figure 2.4:** List of popularly used activation functions for DNNs and other NNs. The figure is retrieved from Berner et al. [32].

In this thesis, we will be mainly concerned with the activation functions *linear*, *sigmoid* and *tanh*. The linear activation function, or identity function, applies no transformation to the layer output  $\mathbf{z} = W^l \mathbf{a}^{l-1} + \mathbf{b}^l$ . This activation function is usually applied to the output layer only, and is useful when we don't want to transform the final output of the DNN into any particular shape. This is usually the case for regression, where the targets we want to predict can be any real value. The sigmoid activation function constraints the output to be in the interval  $a^L \in [0, 1]$ . This activation is often used on the output layer when doing classification, as the target labels are (in the binary case)  $y \in \{0, 1\}$ . For all hidden layers, we will use tanh activation. This choice is discussed in [Section 6.5.4](#).

### 2.3.4 Saturated Activations and Vanishing Gradient

A common problem with many widely used activation functions is *saturation*. An activation function  $f(x)$  is said to be saturated if the input  $x$  causes it to become locally very flat. From [Figure 2.4](#), one can see that this happens for the tanh activation for very high or low values of  $x$ . This causes the derivative of the activation to be approximately zero, i.e.  $f'(x) \approx 0$ . If sufficiently many activations are saturated in the network, backpropagation using [Equation \(2.15\)](#) tends to

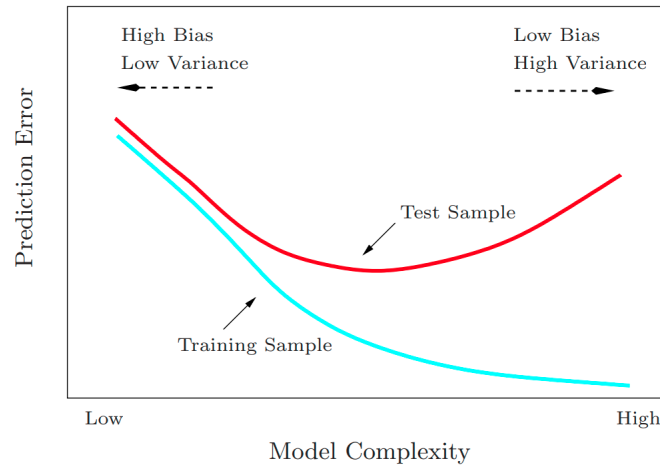
produce gradients close to zero. This phenomenon is known as a *vanishing gradient* [33], and is known to slow down training of the neural network. Typically, the effect is worse for NNs with many layers.

## 2.4 Generalizability

If we obtain a low loss on the training data after fitting a model, can we assume that the resulting model is good and useful? It depends on what we want to use the model for, but if we want to use the model for predicting on new data (data that was not seen during training), we often don't want to train the model too much. As explained earlier, the main goal of a model is to approximate the underlying mechanism  $f(\mathbf{x})$  of Equation (2.1) producing the data. If the model fits the data too well, it might also pick up details of the noise  $\epsilon$  in the training data, which is called *overfitting*. The problem with this is that if we gather a new data set, a *test set*  $y' = f(\mathbf{x}) + \epsilon'$ , the noise  $\epsilon'$  will be different from the noise in the training data because of its random nature. Since the overfitted model is very affected by the noise in the training data, it will likely perform poorly on new data where the noise is different, even though it performs well on the training data. Typically, the more complex and flexible a model is, the more likely it is to overfit the training data. This is because it has a greater capacity to fit the noise present in the training data. By restricting the complexity of the model, one often ends up with a model that resembles  $f(\mathbf{x})$  more closely. In turn, this results in a model that *generalizes* better, which means that it makes accurate predictions on values of  $\mathbf{x}$  not present in the training set. On the other hand, if the model is not complex enough, it might not be sufficiently flexible to recreate  $f(\mathbf{x})$ , causing *underfitting*.

To uncover overfitting, the standard procedure is to prepare independent training and test sets  $\mathcal{T}_{Train}$  and  $\mathcal{T}_{Test}$ , and train the model on the former set and test its performance on the latter. The behaviour of the prediction error, e.g. MSE, on the training and test set is expected to behave as in Figure 2.5. From this figure, we see that the prediction error on the training set is strictly decreasing, since a more complex model is able to fit the training data more closely. However, the prediction error of the test set obtains a minimum prediction error for some model complexity, providing the best generalization. Beyond this point, the model will start to overfit and produce worse predictions.

The number of epochs NNs are trained for can be thought of as a type of model complexity. For increasing number of epochs, the network will fit the training data better and better. Continuing this, the network will eventually start overfitting the data and produce worse prediction error for the test set. One technique for avoiding overfitting is to add a small amount of noise to the input data of the NN. By adding noise to the input, one forces the NN to learn the same output for slight variations of the same input. Ultimately, this leads to a more robust model that avoids overfitting [34].



**Figure 2.5:** Training and test error as a function of model complexity. For models trained iteratively, like neural networks, "Model Complexity" can be associated with the number of optimization steps. The figure is retrieved from Hastie et al. [4].

## 2.5 Pre-processing Data

In this section, we will discuss common techniques for preparing and processing features and data before training models, generally known as *pre-processing*. We start by presenting two methods of scaling features, *standardization* and *min-max scaling*, meant for improving the performance of models. Then, we will present *principal component analysis* (PCA), a technique for reducing the number of features and speeding up training of models.

### 2.5.1 Scaling Features

For data sets gathered for real world applications, it is often the case that the different features have very different units and numerical scales. E.g., a data set detailing health habits may include features such as *age* in the range 0 – 80, and *caloric intake* of order 2000. Many models, such as neural networks, are sensitive to the scales of the features and may perform poorly if they are very different [5]. Therefore, it is typical to scale the features in a way to avoid such outlier values.

#### Standardization

For neural networks of the type presented in Section 2.3, features are often scaled using standardization to improve performance [7]. Mathematically, this involves subtracting the mean and divide by the standard deviation over the data set, for each feature:

$$x_j^{(i)} \rightarrow \frac{x_j^{(i)} - \bar{x}_j}{\sigma(x_j)}, \quad (2.19)$$

where  $\bar{x}_j$  and  $\sigma(x_j)$  is the mean and standard deviation of the feature  $x_j$ , respectively. This ensures that each feature has zero mean and unit standard deviation.

### Min-Max Scaling

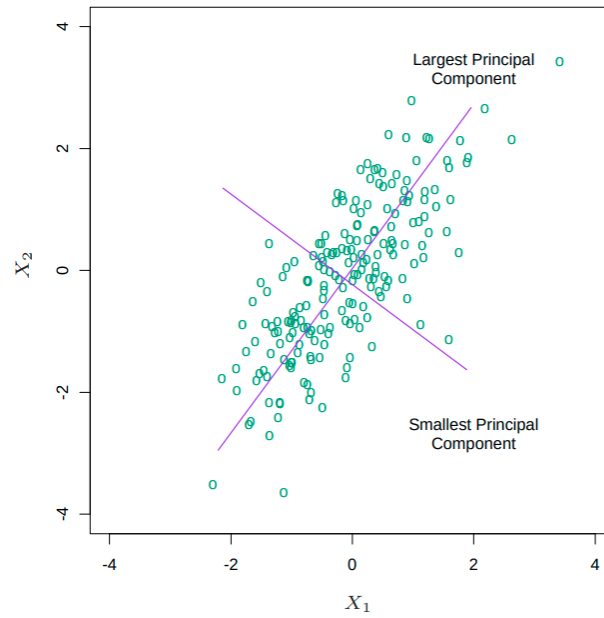
An alternative to standardization is min-max scaling, useful for when we want the features to lie in a certain interval. To scale the feature  $x_j$  to the interval  $[a, b]$ , we can apply the transformation

$$x_j^{(i)} \rightarrow (b - a) \frac{x_j^{(i)} - \min(x_j)}{\max(x_j) - \min(x_j)} + a \quad (2.20)$$

where  $\min(x_j)$  and  $\max(x_j)$  return the minimum and maximum value of  $x_j$  over the data set, respectively.

## 2.5.2 Principal Component Analysis

For data sets with many features, training models may become computationally expensive. Because of this, we often want to reduce the number of features without losing too much of the information of data that may be important for prediction. One way of accomplishing this is with the use of principal component analysis, which applies a linear transformation on the features  $x_j$  to derive new features  $z_j$  called principal components. The property of the principal components is that they determine the directions in feature space that capture the largest amount of variance in the data, and hence information. The first component  $z_1$  is the direction of largest variance,  $z_2$  the second largest, etc. In [Figure 2.6](#), the resulting two principal components are visualized for a data set with two features. For a data set where the features are highly correlated, which is typical for real data sets, performing PCA and keeping the first few components can greatly reduce the number of features without losing too much of the information of the data.



**Figure 2.6:** The two principal components resulting from PCA applied to a data set with two features  $x_1$  and  $x_2$ . The components give the directions in feature space with most and second most variance in the data. The figure is retrieved from Hastie et al. [4].

# 3

## Quantum Computing

This chapter introduces the fundamentals of quantum computing. The content of this chapter is mainly based on material in Nielsen and Chuang [9].

### 3.1 States in Quantum Mechanics

In quantum mechanics, isolated physical systems are described completely by its *state vector*, which lives in a complex vector space. In this thesis, we will focus on finite vector spaces  $\mathbb{C}^n$ , where states are n-tuples of complex numbers  $(z_1, \dots, z_n)$  called *amplitudes*. Adopting Dirac notation, a state is denoted as

$$|\psi\rangle \sim \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}, \quad (3.1)$$

where  $\psi$  is the label of the state, and  $|\cdot\rangle$  indicates that it is a vector. More specifically, in quantum mechanics, the states live in *Hilbert space*, which is a vector space that has a well-defined inner product. The inner product of two states  $|\psi\rangle, |\psi'\rangle \in \mathbb{C}^n$  is denoted

$$\langle\psi'|\psi\rangle \equiv [z_1'^*, \dots, z_n'^*] \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = \sum_{i=1}^n z_i'^* z_i, \quad (3.2)$$

where  $z^*$  indicates the complex conjugate. That is, if  $z = a + ib$ , the complex conjugate results in  $z^* = a - ib$ . As a constraint on the amplitudes, state vectors that describe physical systems have unit norm, meaning

$$\langle\psi|\psi\rangle \equiv [z_1^*, \dots, z_n^*] \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = \sum_{i=1}^n |z_i|^2 = 1. \quad (3.3)$$

### 3.1.1 The Qubit

As is common in quantum computing, we will focus on perhaps the simplest possible quantum system, the *qubit*, which is a two-level system defined on  $\mathbb{C}^2$ . There are multiple ways of implementing qubits in hardware, some of which will be discussed later, although the specific physical realization is not necessary to account for when discussing quantum computing. In abstract terms, the state of a qubit can be formulated as

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle, \quad (3.4)$$

where  $\alpha$  and  $\beta$  are complex numbers, and  $|0\rangle$  and  $|1\rangle$  are orthonormal states known as the *computational basis states* and are defined by the implementation of the hardware. This linear combination of states is an important principle of quantum mechanics and is called *superposition*; the system is in neither state  $|0\rangle$  nor  $|1\rangle$ , but both at the same time (unless either  $\alpha$  or  $\beta$  is zero). In general, if states  $|\psi\rangle$  and  $|\phi\rangle$  are allowed, then so is the linear combination  $\alpha|\psi\rangle + \beta|\phi\rangle$ , where  $|\alpha|^2 + |\beta|^2 = 1$ .

Being the "atom" of quantum computing, the qubit is reminiscent of the classical bit, which is always definitely "0" or "1". However, as we have seen, the qubit also may assume any normalized linear combination of the two states.

### 3.1.2 Multiple Qubits

As a central property of quantum mechanics, it is possible to create composite systems by combining several smaller quantum systems. This can be used to construct systems of multiple qubits, whose collective state can be expressed, if the qubits are independent, as

$$|\psi_1\psi_2\cdots\psi_n\rangle \equiv |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle. \quad (3.5)$$

Here, the tensor product " $\otimes$ " was used to indicate that each state  $|\psi_i\rangle$  lives in its own  $\mathbb{C}^2$  space. Using the principle of superposition, one may make a linear combination of several multi-qubit states, where each  $|\psi_i\rangle$  is either  $|0\rangle$  or  $|1\rangle$ . In general, this can be written as

$$|\psi\rangle = \sum_{\mathbf{v}} c_{\mathbf{v}} |\mathbf{v}_1\rangle \otimes |\mathbf{v}_2\rangle \otimes \cdots \otimes |\mathbf{v}_n\rangle, \quad (3.6)$$



where  $\mathbf{v} \in \{0,1\}^n$  sums over all possible binary strings of length  $n$ . As there are  $2^n$  unique strings, we arrive at the remarkable result that one also needs  $2^n$  amplitudes  $c_{\mathbf{v}}$  to describe the state of  $n$  qubits in general. In other words, the information stored in the quantum state of  $n$  qubits is exponential in  $n$ , as opposed to the linear information of an equivalent classical system of classical bits. In a sense, the quantum information is "larger" than the classical information. This is a fascinating property of the capabilities of quantum computing, which we will return to when discussing the usefulness of quantum computing in relation to machine learning.

### 3.1.3 Measuring Qubits

It appears the information encoded in quantum systems is much greater than the information in a corresponding classical system, at least in the case of qubits versus bits. How can one interact with this information? Unlike classical bits, whose state can always be measured exactly, the state of one or multiple qubits cannot be measured and determined. Returning to the single qubit example, one can choose to perform a measurement in the computational basis on a qubit in the state  $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ . The measurement will result in *either*  $|0\rangle$  *or*  $|1\rangle$ , with probabilities  $|\alpha|^2$  and  $|\beta|^2$ , respectively. For multiple qubits in a general state Equation (3.6), a measurement on all qubits will grant a state in the computational basis, i.e.  $|\mathbf{v}_1\rangle \otimes |\mathbf{v}_2\rangle \otimes \cdots |\mathbf{v}_n\rangle$  for some binary string  $\mathbf{v} \in \{0,1\}^n$ , with probability  $|c_{\mathbf{v}}|^2$ . This motivates why states in quantum mechanics need to have unity norm, i.e

$$\sum_{\mathbf{v}} |c_{\mathbf{v}}|^2 = 1, \quad (3.7)$$

as the probabilities of any outcome must sum to 1.

## 3.2 Quantum Circuits

We have discussed how quantum states can encode information, and how to interact with information through measuring the state. How then can quantum mechanics be used for computation? In order to perform computations, it is necessary to introduce some dynamical transformation of the quantum state. In quantum mechanics, transformations can be formulated as

$$|\phi\rangle = U |\psi\rangle, \quad (3.8)$$

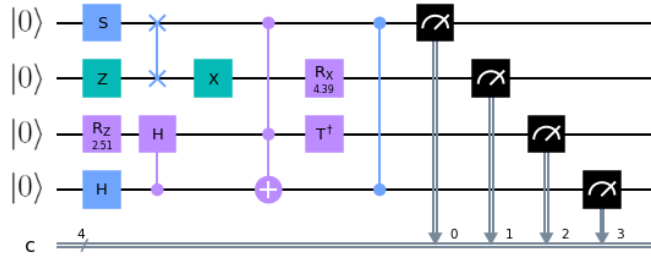
where  $U$  is a *unitary* operator that acts on the vector space where  $|\psi\rangle$  and  $|\phi\rangle$  live. "Unitary" means that the operator  $U$  is linear with the property that  $U^\dagger = U^{-1}$ , that is, the Hermitian conjugate is equal to its inverse. This is a necessary property of linear operators in quantum mechanics as to ensure that the state stays normalized to 1:

$$\langle \phi | \phi \rangle = \langle \psi | \underbrace{U^\dagger U}_I | \psi \rangle = \langle \psi | \psi \rangle = 1. \quad (3.9)$$

Assuming  $|\psi\rangle$  is initially normalized, so is  $|\phi\rangle$  after a unitary transformation.

By construction, quantum computers allow for the application of carefully selected sequences of operators that transform the state in a desired way, often called a *quantum circuit*. Typical operators used in quantum computing, often called quantum gates, act on one or multiple qubits, and are analogous to logical operations in the classical context.

Figure 3.1 illustrates an example of a quantum circuit. Going from left to right indicates the chronological order of application of the different quantum gates. Note that the exact passing of time is not shown in this schematic, and is highly dependent on the implementation of physical hardware. The horizontal lines, called wires, each symbolize a qubit. The qubits are initialized in the zero state, as shown by the notation on the left-hand side. Then, various gates are applied to the qubits, acting on one, two or three qubits. Lastly, illustrated by the gauge symbol, each qubit is measured in the computational basis, yielding either 0 or 1. This information is then stored in the classical register  $c$ , indicated by the double line.



**Figure 3.1:** Example circuit consisting of 4 qubits initialized to  $|0\rangle$ . A random selection of quantum gates acting on one, two and three qubits are then applied. Finally, all qubits are measured in the computational basis and stored in a classical register.

### 3.2.1 Single Qubit Operations

Returning again to the single qubit, the state of a qubit can be represented as a vector

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \equiv \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \quad (3.10)$$

Likewise, linear operators acting on a single qubit can in general be represented by a  $2 \times 2$  matrix

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}, \quad (3.11)$$

which is unitary. A particularly interesting single qubit quantum gate is the Hadamard gate, which is formulated as

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = -\boxed{H}. \quad (3.12)$$

Acting on the computational basis, the Hadamard gate can be seen to produce superpositions

$$\begin{aligned} H|0\rangle &= \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \\ H|1\rangle &= \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle, \end{aligned}$$

which gives a 50% chance to yield either 0 or 1 upon measuring. In a sense, this is the quantum mechanical equivalent to a coin toss. Contrary to a coin toss, if applied a second time, we return to the original state, "unscrambling" the coin, or qubit:

$$HH|0\rangle = \frac{1}{\sqrt{2}}H|0\rangle + \frac{1}{\sqrt{2}}H|1\rangle = \frac{1}{2}(|0\rangle + |0\rangle) + \frac{1}{2}(|1\rangle - |1\rangle) = |0\rangle. \quad (3.13)$$

As pointed out in Schuld and Petruccione [1], this phenomenon has no classical equivalent. If one has a classical procedure of scrambling a coin, e.g. shaking it in your hands, a second shaking will not leave it unscrambled, but scrambled still. Quantum computation is able to reverse this because quantum mechanics is fundamentally not a theory of probabilities; Probabilities can be derived from the theory, but the underlying description revolves around amplitudes, as explained earlier. Whereas probabilities must be positive or zero, amplitudes can be positive or negative (and complex in general), which allows destructive interference. This can be seen in Equation (3.13), where the last term  $\frac{1}{2}(|1\rangle - |1\rangle)$  is cancelled out. In addition to the exponentially large size of Hilbert space, this is also an interesting property of quantum computing when discussing its capabilities over classical computing.

Further, a much-used set of single qubit gates are the Pauli operators:

$$\begin{aligned} X = \sigma_x &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = -\boxed{X} \\ Y = \sigma_y &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -\boxed{Y} \\ Z = \sigma_z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = -\boxed{Z} \end{aligned} \quad (3.14)$$

To visualize what these operators do, it is useful to introduce a geometrical representation called the *Bloch sphere*, illustrated in Figure 3.2. Rewriting the state of a qubit to

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = e^{i\gamma} \left( \cos \frac{\theta}{2} |0\rangle + e^{i\phi} \sin \frac{\theta}{2} |1\rangle \right) \sim \cos \frac{\theta}{2} |0\rangle + e^{i\phi} \sin \frac{\theta}{2} |1\rangle, \quad (3.15)$$

the new parameters  $\theta$  and  $\phi$  can be identified as the azimuthal and polar angles, respectively. Here, the factor  $e^{i\gamma}$  is known as a global phase, which is not physically important to include. Using this, any single qubit state can then be identified as a point on the Bloch sphere.

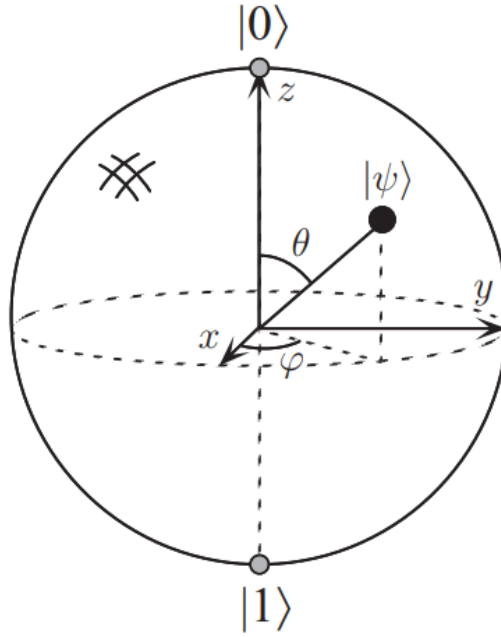


Figure 1.3. Bloch sphere representation of a qubit.

**Figure 3.2:** Geometrical representation of the state of a qubit, called the Bloch sphere. The figure is retrieved from Nielsen and Chuang [9].

The Pauli-gates represent a  $180^\circ$  rotation of the state around the corresponding axis. Particularly, the  $X$  gate, often called the *flip gate*, acts on the basis states as

$$\begin{aligned} X |0\rangle &= |1\rangle \\ X |1\rangle &= |0\rangle \\ X(\alpha |0\rangle + \beta |1\rangle) &= \beta |0\rangle + \alpha |1\rangle. \end{aligned} \quad (3.16)$$

Much like how the classical NOT-gate flips the bit, the  $X$  gate flips the qubit.

Another essential set of gates used for quantum machine learning, which can be derived from the aforementioned Pauli gates, are the *Pauli rotations*. They are formulated as exponentiated Pauli gates in the following way:

$$\begin{aligned}
 R_x(\theta) &= e^{-i\theta\sigma_x/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} \sigma_x = \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \\
 R_y(\theta) &= e^{-i\theta\sigma_y/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} \sigma_y = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \\
 R_z(\theta) &= e^{-i\theta\sigma_z/2} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} \sigma_z = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}
 \end{aligned} \tag{3.17}$$

The action of a gate  $R_j(\theta)$  on a state, for  $j \in \{x, y, z\}$ , is to rotate the state around the  $j$ -axis on the Bloch sphere, for an amount of  $\theta$  radians. As  $\theta$  can be any real number, these gates can be viewed as being quantum gates parameterized by  $\theta$ , which will be an essential component when we will construct *parameterized quantum circuits* later.

### 3.2.2 Multi-Qubit Operators

With only single qubit gates, the number of states we can access is greatly reduced as all the qubits stay independent, i.e. the state can be written as Equation (3.5). By introducing gates that operate on several qubits, one can for example transform the state of one qubit conditioned on the state of an other qubit. This makes the two qubits correlated, known as *entanglement* in quantum mechanics.

#### CNOT

One such conditional gate is the controlled NOT gate (CNOT gate). It is formulated as

$$CNOT = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \text{---} \bullet \text{---} \bigoplus \text{---}, \tag{3.18}$$

where we have also included its formulation in Dirac notation. Looking at the circuit representation, the black dot indicates that the gate is conditioned on the top qubit(control qubit). If it is in state  $|1\rangle$ , an  $X$  gate is applied to the bottom qubit(target qubit). Otherwise, it is left unchanged. By convention, the  $X$  gate here is denoted by  $\bigoplus$ . This operation has an interesting effect if the control qubit is in a superposition. Assume we begin in the state

$$|\psi\rangle = H|0\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle, \tag{3.19}$$

the application of the CNOT gate will yield the following:

$$CNOT |\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes I |0\rangle + |1\rangle \otimes X |0\rangle) = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle). \quad (3.20)$$

This is a well-known state called a *Bell state*. It has the interesting property that the qubits are correlated: When the first qubit is measured to be in either state 0 or 1, the second qubit will be found in the same state, and vice versa. This is known as an entangled state, which cannot be expressed as a product of independent single qubit states, such as Equation (3.5). By introducing controlled gates, we have increased the space of accessible states.

### Multi-Controlled Gate

The CNOT gate is just one example of a controlled quantum gate. In general, we can have a controlled gate on the form

$$|0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes U = \text{---} \begin{array}{c} \bullet \\ | \\ \boxed{U} \end{array} \text{---}, \quad (3.21)$$

where  $U$  is any single qubit gate. Moreover, there also exists *multi-controlled gates* on the form

$$\begin{array}{c} \bullet \\ \bullet \\ \bullet \\ | \\ \boxed{U} \end{array} . \quad (3.22)$$

This gate applies  $U$  to the target qubit if, and only if, all three control qubits (in general  $n$  control qubits) are in state  $|1\rangle$ . The application of the gate may be dependent on the control qubits being in  $|0\rangle$  rather than  $|1\rangle$ . This can be done by applying an  $X$  gate before and after the controlled operation to the qubit one wishes to invert, such as

$$\begin{array}{c} \text{---} \boxed{X} \text{---} \bullet \text{---} \boxed{X} \text{---} \\ | \\ \bullet \\ | \\ \bullet \\ | \\ \bullet \\ | \\ \boxed{U} \end{array} = \begin{array}{c} \text{---} \bullet \text{---} \\ | \\ \circ \\ | \\ \bullet \\ | \\ \bullet \\ | \\ \boxed{U} \end{array} . \quad (3.23)$$

Here, the conditioning on state  $|0\rangle$  is indicated by a white dot.

### SWAP gate

A well-known two-qubit gate is the SWAP gate. As the name indicates, the SWAP gate swaps the information of qubits. It is defined as

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{array}{c} \text{---}\times\text{---} \\ | \\ \text{---}\times\text{---} \end{array}, \quad (3.24)$$

For a two-qubit system, where the first qubit is in the state  $|\psi\rangle$  and the second is in  $|\phi\rangle$ , the swap gate has the following function:

$$\begin{array}{c} |\psi\rangle \text{---}\times\text{---} |\phi\rangle \\ |\phi\rangle \text{---}\times\text{---} |\psi\rangle \end{array} \quad (3.25)$$

### 3.2.3 Observables

In [Section 3.1.3](#), we introduced the process of measurement. We will now generalize this by introducing *quantum observables*. In quantum mechanics, an observable is an operator that acts on the state space of the system being measured. It can be expressed as

$$\hat{O} = \sum_m m P_m, \quad (3.26)$$

where  $m$  are real numbers called *eigenvalues* and  $P_m$  are projection operators (satisfying  $P_m^2 = P_m$ ) with the condition  $\sum_m P_m = I$ . Under these conditions, the operator  $\hat{O}$  is said to be *Hermitian*, which is a property required of all quantum observables. Upon measuring the observable [Equation \(3.26\)](#) on a state  $|\psi\rangle$ , the measured value will be  $m$  with probability

$$p(m) = \langle \psi | P_m | \psi \rangle, \quad (3.27)$$

and original state will be projected onto  $P_m$  yielding

$$|\psi\rangle \rightarrow \frac{P_m |\psi\rangle}{\sqrt{\langle \psi | P_m | \psi \rangle}}, \quad (3.28)$$

where the scaling factor ensures that the new state is still normalized. Using this formalism, one can identify the Pauli gate  $\sigma_z$  as a suitable observable for measuring the computational basis:

$$\sigma_z = |0\rangle \langle 0| - |1\rangle \langle 1|. \quad (3.29)$$

For the general single qubit state  $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ , we see that probabilities of measuring  $m = 1$  and  $m = -1$  are respectively

$$\begin{aligned} p(0) &= \langle \psi | 0 \rangle \langle 0 | \psi \rangle = (\alpha^* \langle 0 | + \beta^* \langle 1 |) | 0 \rangle \langle 0 | (\alpha | 0 \rangle + \beta | 1 \rangle) = |\alpha|^2, \\ p(1) &= \langle \psi | 1 \rangle \langle 1 | \psi \rangle = (\alpha^* \langle 0 | + \beta^* \langle 1 |) | 1 \rangle \langle 1 | (\alpha | 0 \rangle + \beta | 1 \rangle) = |\beta|^2, \end{aligned} \quad (3.30)$$

and the states after the corresponding measurement is

$$\begin{aligned}\frac{|0\rangle\langle 0|(\alpha|0\rangle + \beta|1\rangle)}{\sqrt{|\alpha|^2}} &= \frac{\alpha}{|\alpha|}|0\rangle \sim |0\rangle \\ \frac{|1\rangle\langle 1|(\alpha|0\rangle + \beta|1\rangle)}{\sqrt{|\beta|^2}} &= \frac{\beta}{|\beta|}|1\rangle \sim |1\rangle,\end{aligned}\tag{3.31}$$

where it was used that  $\frac{\alpha}{|\alpha|}$  and  $\frac{\beta}{|\beta|}$  are just global phases, and hence not important. Thus we see that the measurement leaves the state in the computational basis corresponding with the measured value, with the correct probability as described in [Section 3.1.3](#).

### 3.2.4 Expectation Values

What is the average value of an observable for a given state  $|\psi\rangle$ ? Using statistical formalism, we can formulate this as the *expectation values* of the observable

$$\mathbb{E}(m) = \sum_m mp(m) = \sum_m m \langle \psi | P_m | \psi \rangle = \langle \psi | \sum_m m P_m | \psi \rangle = \langle \psi | \hat{O} | \psi \rangle, \tag{3.32}$$

where  $\langle \psi | \hat{O} | \psi \rangle$  is a recurring expression in quantum mechanics, often denoted simply as  $\langle \hat{O} \rangle$ . The expectation value is crucial for quantum computing as it serves as a method for extracting a deterministic value from a quantum state. Whereas the total quantum information of a state is inaccessible to us, estimating the expectation value for some desired observable is possible for retrieving an output of a quantum algorithm. Desirably, this output serves as a solution to the problem one wishes to solve. Looking at the expected value [Equation \(3.32\)](#), it is easy to see why global phases of the state are physically insignificant. How does the expected value of an arbitrary observable change when we add a global phase  $|\psi\rangle \rightarrow e^{i\gamma}|\psi\rangle$ ? We get

$$\langle \psi | e^{-i\gamma} \hat{O} e^{i\gamma} | \psi \rangle = \langle \psi | e^{i(\gamma-\gamma)} \hat{O} | \psi \rangle = \langle \psi | \hat{O} | \psi \rangle. \tag{3.33}$$

The above results show that whatever measurement we do on the state, we cannot determine if the global phase is present or not. Therefore, we can assume two states that differ by a global phase are physically identical, as was assumed in [Equation \(3.15\)](#) and [Equation \(3.31\)](#).

### 3.2.5 Estimating Expectation Values

How do we practically calculate or estimate expectation values? For all observables to this thesis, we are able to express them as a spectral decomposition of the computational basis, meaning we can write them in the form



$$\hat{O} = \sum_{i=1} \lambda_i |i\rangle \langle i|, \quad (3.34)$$

where  $i$  sums over all computational basis vectors  $|i\rangle$ , and  $\lambda_i$  are real values. We calculate the expectation value by inserting a linear expansion of  $|\psi\rangle$  in terms of the computational basis:

$$\langle \psi | \hat{O} | \psi \rangle = \sum_i \alpha_i^* \langle i | \left( \sum_j \lambda_j |j\rangle \langle j| \right) \sum_k \alpha_k |k\rangle = \sum_i |\alpha_i|^2 \lambda_i. \quad (3.35)$$

Given that we know the eigenvalues  $\lambda_i$ , all we need to do is estimate  $|\alpha_i|^2$ . Even though we don't have direct access to the amplitudes of a state,  $|\alpha_i|^2$  coincide with the probability of measuring the corresponding basis state. We can introduce a Bernoulli random variable  $y_{ij}$  such that  $P(y_{ij} = 0) = 1 - |\alpha_i|^2$  and  $P(y_{ij} = 1) = |\alpha_i|^2$ . By repeatedly preparing the state  $|\psi\rangle$  and measuring it in the computational basis, called performing several *shots*, one can gather  $S$  such samples  $\{y_{i1}, \dots, y_{iS}\}$ . As pointed out in Schuld and Petruccione [1],  $|\alpha_i|^2$  can be estimated with a *frequentist estimator*  $\hat{p}_i$  given by

$$|\alpha_i|^2 \approx \hat{p}_i = \frac{1}{S} \sum_{j=1}^S y_{ij}. \quad (3.36)$$

The standard deviation of the estimator  $\hat{p}_i$  can be shown to be

$$\sigma(\hat{p}) = \sqrt{\frac{\hat{p}_i(1 - \hat{p}_i)}{S}}. \quad (3.37)$$

If  $S$  is reasonably large,  $\hat{p}$  is approximately normally distributed by the law of large numbers. Consequently, any one estimation of  $\hat{p}$  falls within the interval of one standard deviation around the mean with a probability of 68%. This means that in order to reduce the error of the estimation, i.e. the standard deviation, one needs to increase the number of shots  $S$ . Looking at the above expression, the error of  $\hat{p}_i$  goes as  $O(1/\sqrt{S})$ .

The expectation values can be estimated by inserting the estimates  $\hat{p}_i$  into [Equation \(3.35\)](#), giving

$$\langle \psi | \hat{O} | \psi \rangle \approx \sum_i \hat{p}_i \lambda_i. \quad (3.38)$$

From this expression, it can be seen that also the error of the expectation value goes as  $O(1/\sqrt{S})$ . This is a computationally expensive aspect of quantum computing, since a reduction of error by a factor 10 requires a factor 100 more shots. In practice, the output of quantum circuits tends to be noisy because of the use of a finite number of shots. When one tries to estimate vanishingly small quantities,

the number of shots required to overcome bad signal-to-noise ratio can become prohibitively high.

### 3.3 Noisy Intermediate-Scale Quantum Computing

So far, we have introduced abstract and rather idealized aspects of quantum mechanics in the context of quantum computing. We have not yet discussed how quantum algorithms are implemented on quantum hardware in practice, and what drawbacks such implementation might bring. Even in the ideal case, we saw in [Section 3.2.5](#) that outputs of quantum circuits are noisy as a result of finite number of shots. Quantum computing on near-term quantum hardware, so-called *noisy intermediate-scale quantum computing* (NISQ) [\[14\]](#), is characterized by few available qubits, low-fidelity computations and other restrictions. These aspects tend to make performing quantum computing even more challenging, and will be frequently discussed when we later motivate different ways of implementing quantum machine learning. The content of this section is mainly based on Schuld and Petruccione [\[1\]](#) and Preskill [\[14\]](#).

#### 3.3.1 Gate Fidelity

In physical quantum computers, it is important to implement ways to precisely control qubits and interactions between them in order to execute various quantum gates. One of the more promising implementations of qubits, *super-conducting qubits*, uses pulses of microwaves to control the qubits. Using this technique, Barends et al. [\[35\]](#) was able to implement quantum gates with as low as 1% measurement error probability, although oftentimes higher. In addition, it is unclear whether such low error can be maintained when the quantum computer is scaled up. In practice, the application of multiple noisy gates results in the accumulation of error rendering the outcome useless [\[14\]](#). Consequentially, the number of gates should be kept low in order to minimize error of the quantum algorithm.

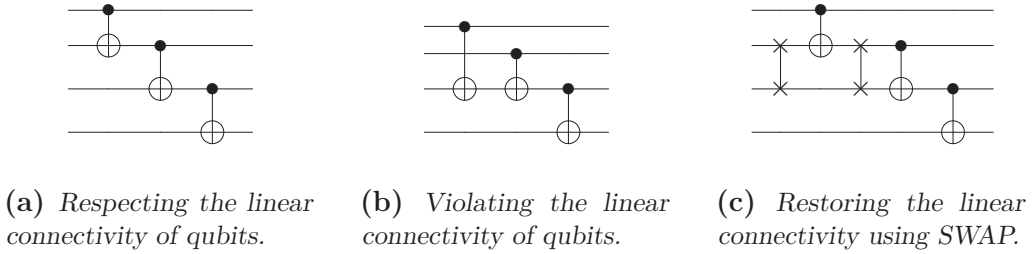
#### 3.3.2 Quantum Decoherence

In addition to the error introduced by the imprecision of the gates, the qubits themselves are susceptible to outside disturbance, causing *decoherence* of the state. In [Section 3.1](#) and [Section 3.2](#), we talked about states of isolated systems and how they transform under unitary operators. In this context, "isolated" means that the system of qubits is not affected by any external sources, with the exception of the mechanisms that implement quantum gates. In practice, quantum computers are only approximately isolated, as vibrations and external fields tends to leak into the system, degrading the information stored in the state. This effect tends to strengthen the longer the computation takes, and places another restriction on how

many gates one can implement. Specifically, decoherence limits the *circuit depth*, which refers to the number of gates applied in sequence.

### 3.3.3 Coupling of Qubits

Depending on the specific implementation of the hardware, it is not given that a two-qubit gate can be applied on any two qubits. Typical for near-term quantum computers, the qubits are arranged in a *linear array* [36]. This means that two-qubit gates may only be applied on neighboring qubits, i.e. they are linearly connected. Figure 3.3 gives examples on quantum circuits that either respect or violate the linear connectivity of qubits.



**Figure 3.3:** Different quantum circuit that either respect or violate the linear connectivity of qubits.

Figure 3.3a applies CNOT gate only on neighboring qubits, which is allowed on a linear architecture. In contrast, Figure 3.3b shows a violation of this. However, the circuit in Figure 3.3c has the equivalent functionality as the aforementioned circuit, while still respecting linear connection. This was achieved by using SWAP gates to essentially "move qubits around", but at the cost of a greater circuit depth. In order to limit the circuit depth as much as possible, we will often discuss quantum circuits respecting the linear connectivity going forward.

### 3.3.4 Basis Gates

Real quantum computers can rarely implement all of the quantum gates discussed earlier in this thesis directly. They often implement a small set of gates called the *basis gates*. This set of basis gates varies from computer to computer, but it is common that these sets are universal. This means that for any gate we would like to implement, there exists a finite number of basis gates that is able to reproduce our target gate to arbitrary precision. In this way, all quantum computers with universal basis gates are able to implement any conceivable algorithm, as the name suggests. This is of course disregarding any potential noise of the hardware, as discussed earlier.

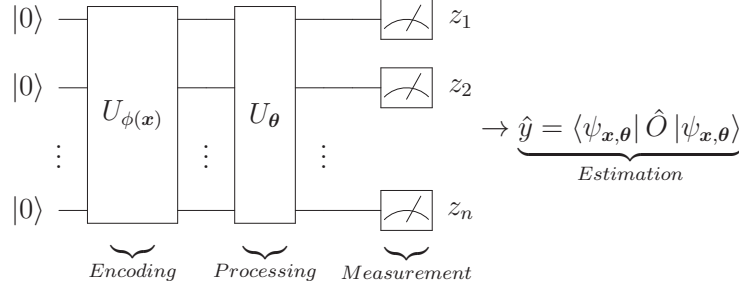
# 4

## Quantum Machine Learning

Over the years, many different quantum algorithms have been proposed that is anticipated to greatly outshine classical methods. Perhaps most famously is the Shor's algorithm [12], which promises to factor integers in polynomial time. This is believed to be an exponentially hard problem for classical computers. However, such useful quantum algorithms often need a large number of error-corrected qubits to be efficient, meaning noise introduced by the environment is corrected for. Also, their implementation requires a large number of quantum gates, requiring the quantum computer to handle deep circuits. As explained in [Section 3.3](#), near-term quantum computers are not able to accommodate these criteria, and are thus unsuitable. What would then be interesting candidate algorithms for useful near-term applications? A promising family of algorithms is *parameterized quantum circuits* (PQC), which are quantum circuits comprised of fixed gates, such as CNOT gates, and adjustable gates, such as Pauli rotations [13]. Unlike algorithms that are tailored to solve specific problems, such as Shor's algorithm for factoring integers, PQCs are general algorithms with free parameters that need to be adjusted in order to solve a given problem. In practice, quantum computers are used to evaluate the circuits, while classical hardware is used to post-process the results and optimized the parameters. In this sense, both quantum and classical hardware are leveraged to solve the problem in a variational manner. This hybrid approach is thought to be much less demanding on the number qubits and the depth of the circuit, as much of the computation is outsource to classical computers [16]. Thus, they are much more suitable for near-term applications. Moreover, since PQC are not problem specific, one is freed from the need to tailor algorithms for solving specific problems, which is otherwise difficult in practice because of how non-intuitive quantum computing can be.

## 4.1 Quantum Neural Networks

The use of PQC as machine learning models, often called *quantum neural network* (QNN), has been subject to extensive research [17, 13, 19]. In general, the typical structure of QNNs can be broken up into three stages: *feature encoding*, *processing* and *measurement*. The result of the measurements is then used to estimate a model output  $\hat{y}$ . This general procedure is summarized in Figure 4.1.



**Figure 4.1:** The general structure of a quantum neural network (QNN). The procedure consists of three steps: First, a routine  $|\psi_x\rangle = U_{\phi(x)} |0\rangle$  for encoding a feature vector onto an  $n$ -qubit Hilbert space is applied. Next,  $|\psi_{x,\theta}\rangle = U_{\theta} |\psi_x\rangle$  applies a circuit parameterized by  $\theta$ , transforming the state in Hilbert space. Lastly, the expectation value of some appropriate observable  $\hat{O}$  is estimated from measurements on the resulting state, yielding a model output  $\hat{y} = \langle \psi_{x,\theta} | \hat{O} | \psi_{x,\theta} \rangle$ .

The first stage of the QNN is to encode a feature vector of  $\mathbf{x}$  into the qubits of a quantum computer by applying a data dependent circuit  $U_{\phi(x)}$ , often called a quantum feature map:

$$|\psi_x\rangle = U_{\phi(x)} |0\rangle, \quad (4.1)$$

This procedure prepares a state  $|\psi_x\rangle$  which encodes the information of the sample. Typically,  $\phi(\cdot)$  is used as a function to pre-process the  $\mathbf{x}$ , like the methods described in Section 2.5. Next, a circuit  $U_{\theta}$  parameterized by  $\theta = (\theta_1, \dots, \theta_{n_{\theta}})$  is applied to  $|\psi_x\rangle$ , resulting in the transformed state

$$|\psi_{x,\theta}\rangle = U_{\theta} |\psi_x\rangle. \quad (4.2)$$

In this context, the circuit  $U_{\theta}$  is often called an *ansatz* and serves as a parameter-dependent way of transform the state  $|\psi_x\rangle$  encoding the data. Since the information of state  $|\psi_{x,\theta}\rangle$  is not directly accessible, as discussed in Section 3.1.3, we need perform a measurement and estimate some quantity to derive a model output. Typically, this is done by estimating an expectation value of some appropriate observable  $\hat{O}$ , resulting in a model output

$$\hat{y} = f_{QNN}(\mathbf{x}; \theta) = \langle \psi_{x,\theta} | \hat{O} | \psi_{x,\theta} \rangle. \quad (4.3)$$

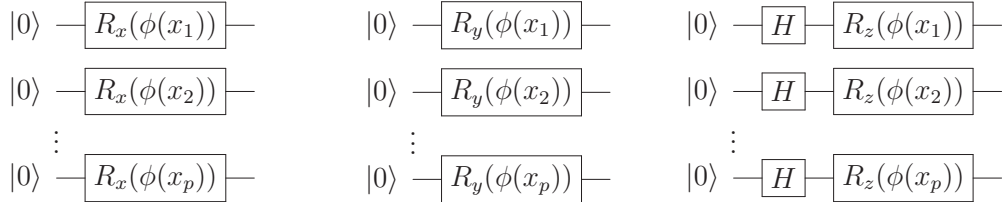
In the standard approach of supervised learning, the parameters  $\theta$  must be trained to minimize some loss function  $L(\theta) = \frac{1}{N} \sum_{i=1}^N L(\hat{y}^{(i)}, y^{(i)})$ , as explained in [Section 2.1](#). This process adapts the ansatz  $U_\theta$  such that the QNN is able to predict the labels  $y$  from the features  $\mathbf{x}$ . In the sections to come, we will discuss specific choices of feature encoding, ansatzes, and expectation values.

## 4.2 Feature Encoding

In this section, we will introduce two ways of doing feature encoding, namely *qubit encoding*, and *RZZ encoding*. We will also introduce *latent qubits*, used for increasing the circuit size and thus also the size of the resulting Hilbert space.

### 4.2.1 Qubit Encoding

A popular approach for feature encoding is the method often-called *qubit encoding* [\[13\]](#). This encoding requires  $p$  qubits, where  $p$  is the number of features, and it can be applied at a constant circuit depth. Before the encoding, the data is optionally pre-processed by some function  $\phi(x_i)$ , for example scaling of the data. Then, features are encoded by performing a Pauli-rotation [Equation \(3.17\)](#) on each qubit with a rotational angle equal to the corresponding feature. [Figure 4.2](#) shows how qubit encoding is implemented using  $R_x$ ,  $R_y$  and  $R_z$  rotation.



**Figure 4.2:** Qubit encoding using  $R_x$ ,  $R_y$  and  $R_z$  rotations to encode  $p$  features, left to right.  $\phi(\cdot)$  applies some kind of pre-processing to the samples, e.g. scaling.

When using  $R_z$  rotations, a Hadamard gate [Equation \(3.12\)](#) is used on each qubit to create a super-position, as these rotations would otherwise leave  $|0\rangle$  unchanged. As an example, two features  $\mathbf{x} = (x_1, x_2)$  can be qubit encoded onto two qubits in the following way using  $R_y$  rotations:

$$R_y(x_1) \otimes R_y(x_2) (|0\rangle \otimes |0\rangle) = \left( \cos\left(\frac{x_1}{2}\right) |0\rangle + \sin\left(\frac{x_1}{2}\right) |1\rangle \right) \otimes \left( \cos\left(\frac{x_2}{2}\right) |0\rangle + \sin\left(\frac{x_2}{2}\right) |1\rangle \right) \quad (4.4)$$

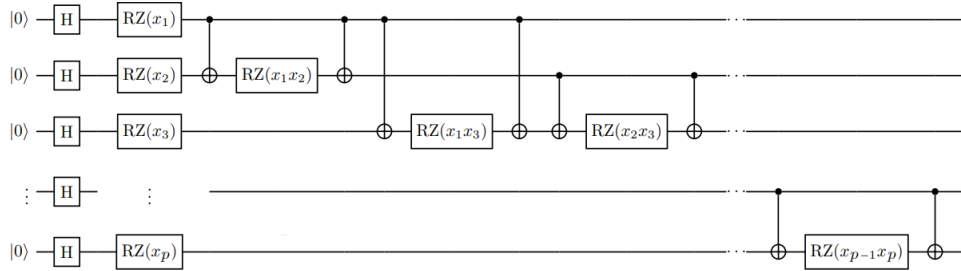
By writing out the tensor product as an explicit state vector, we get

$$\begin{bmatrix} \cos\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right) \\ \cos\left(\frac{x_1}{2}\right) \sin\left(\frac{x_2}{2}\right) \\ \sin\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right) \\ \sin\left(\frac{x_1}{2}\right) \sin\left(\frac{x_2}{2}\right) \end{bmatrix}. \quad (4.5)$$

We see from this that qubit encoding provides a state whose amplitudes encode interactions between the features. In this sense, it computes an exponential number of interactions between features in constant time, potentially creating a powerful representation of the data useful for solving a give learning problem. As the circuit depth is quite low, qubit encoding is an effective method for embedding  $p$  features into a  $2^p$  dimensional Hilbert space. However, the state resulting from qubit encoding is mathematically simple, which may limit the overall expressive power of the final model. In the next section, we will present a more complex way of encoding features which has been shown to improve the overall flexibility of the machine learning model in some contexts.

### 4.2.2 RZZ Encoding

Abbas et al. [17] implemented a quantum feature map for encoding quantum features up to second order, meaning the encoding is dependent on terms such as  $x_i x_j$ , for  $i \neq j$ . The method can be seen as an extension of qubit encoding using  $R_z$  gates, with additional extra RZZ gates that act on qubit  $i$  and  $j$  with rotational angle  $\phi(x_i, x_j) = (\pi - x_i)(\pi - x_j)$ . This is done for  $i \in [1, \dots, p-1]$  and  $j \in [i+1, \dots, p]$ . As this way of encoding was never given a name, we will call it *RZZ encoding* in this thesis. The circuit implementing RZZ encoding can be seen in Figure 4.3



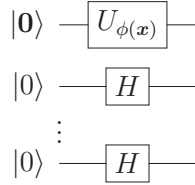
**Figure 4.3:** Circuit visualizing implementation of RZZ encoding of  $p$  features. The functions  $\phi(\cdot, \cdot)$  in the RZZ gates are suppressed for clarity. The figure is retrieved from [17] and adapted to fit our notation.

This procedure produces a much more complex feature map, and can be made more complex still by repeating the whole encoding process several times in a row. The number of such repetitions is often called the *depth* of the feature map. Abbas et al. [17] conjectured that this feature map is difficult to simulate classically for

depth  $\geq 2$ . It was shown by the same authors that RZZ encoding produces much more flexible models than qubit encoding. However, it is also more computationally demanding, as it requires a circuit depth  $\mathcal{O}(p^2)$  and full connectivity between all the qubits.

### 4.2.3 Latent Qubits

For both qubit encoding and RZZ encoding, the number of qubits in the circuit is restricted to be equal to the number of features  $p$  that is being encoded. This constrained can be relaxed by introducing *latent qubits*, which are additional qubits added to the circuit that no features are encoded to. This technique was originally proposed by Lloyd et al. [18], and was implemented to increase the flexibility of the model by making the Hilbert space larger. The authors applied a Hadamard gate to each latent qubit so that they don't start out in the  $|0\rangle$  state. Subsequent ansatzes used for processing the data are then applied to all qubits, including the latent qubits. Figure 4.4 illustrate how latent qubits are added to a circuit. Here,  $U_{\phi(\mathbf{x})}$  is any encoder, e.g. qubit encoding or RZZ encoding, that maps features to some number of qubits denoted  $|0\rangle$ . Then, an arbitrary amount of qubits may be added to the circuit to increase the Hilbert space.



**Figure 4.4:** Circuit for encoding features expanded by adding an arbitrary number of latent qubits. Here,  $U_{\phi(\mathbf{x})}$  is any encoder, e.g. qubit encoding or RZZ encoding, that maps features to some number of qubits denoted  $|0\rangle$ . Then, an arbitrary amount of qubits may be added to the circuit to increase the size of Hilbert space.

## 4.3 Ansatz

What kinds of unitary transformation are interesting as ansatzes used for processing information encoded in quantum states? In principle, we are able to explore every conceivable unitary transformation as a parameterized circuit, since there are circuit designs that are known to be *universal* [37]. In this context, universality means that for any unitary transformation, there exists a sufficiently deep ansatz that approximates the operator to an arbitrary accuracy. However, such approximations are often exponentially deep [9], meaning the vast majority of unitary transformations are inaccessible on ideal quantum computers, let alone near-term quantum computers. Still, it is believed that there exist reasonably shallow ansatzes that are



useful for constructing powerful machine learning models. Many of these ansatzes are also believed to be classically hard to simulate, eluding to a possible quantum advantage for quantum machine learning [18].

In this thesis, we will investigate an ansatz that respect limitations of near-term quantum computers, constrained to circuit depths that scale linearly with the number of qubits and with linear connectivity between qubits. We will refer to this ansatz as the *simple ansatz*. It can be visualised as

$$U_{SA}(\boldsymbol{\theta}) = \begin{array}{c} \text{---} \bullet \text{---} R_y(\theta_1) \\ | \\ \text{---} \oplus \text{---} \bullet \text{---} R_y(\theta_2) \\ | \\ \text{---} \oplus \text{---} \bullet \text{---} R_y(\theta_3) \\ | \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ \text{---} \oplus \text{---} R_y(\theta_{n_\theta}) \end{array} \quad (4.6)$$

The simple ansatz applies CNOT gates on neighboring qubits in sequence until the final qubit is reached. This creates entanglement between the qubits and enables access to a larger space of transformations, as explained in Section 3.2.2. Then, an  $R_y$  rotation is applied to each qubit, each parameterized with its own parameter. Both the number of parameters and the circuit depth of the simple ansatz scales linearly with the number of qubits, making it hardware-efficient and hopefully suitable for near-term applications.

In order to produce a more expressive ansatz that can produce more complicated transformations, we may repeat the simple ansatz  $r$  number of times with independent parameter as such:

$$|\psi_{\mathbf{x},\boldsymbol{\theta}}\rangle = U_{SA}(\boldsymbol{\theta}^r) \cdots U_{SA}(\boldsymbol{\theta}^2) U_{SA}(\boldsymbol{\theta}^1) |\psi_{\mathbf{x}}\rangle, \quad (4.7)$$

where  $\boldsymbol{\theta}^i$  are independent vectors of parameters. We call  $r$  the repetitions of the ansatz.

## 4.4 Model Output

To derive a model output  $\hat{y}$ , we must estimate the expectation value of some observable with respect to the state prepared by the encoder and ansatz, i.e.  $\hat{y} = \langle \psi_{\mathbf{x},\boldsymbol{\theta}} | \hat{O} | \psi_{\mathbf{x},\boldsymbol{\theta}} \rangle$ . In the same manner as Abbas et al. [17], we use the *parity* of the state to derive a model output for inference. The parity of an  $n$  qubit state can be formulated the operator

$$P = \frac{1}{2} (I^{\otimes n} + \bigotimes_{i=1}^n \sigma_z). \quad (4.8)$$

Applying this operator on a computational basis state  $|v_1 \cdots v_n\rangle$  computes

$$P |v_1 \cdots v_n\rangle = \frac{1}{2}(|v_1 \cdots v_n\rangle - (-1)^{\sum_{i=1}^n v_i} |v_1 \cdots v_n\rangle) = \bigoplus_{i=1}^n v_i |v_1 \cdots v_n\rangle, \quad (4.9)$$

where  $\bigoplus_{i=1}^n v_i$  is the mod 2 sum of the terms  $v_i$ , also known as the parity of the bitstring  $v_1 \cdots v_n$ . In short, the parity of a state  $|v_1 \cdots v_n\rangle$  is zero if the number of qubits in state  $|1\rangle$  is even, and is otherwise one. This is called even and odd parity, respectively. To estimate the expected parity  $\langle \psi_{x,\theta} | P | \psi_{x,\theta} \rangle$ , we use the technique described in [Section 3.2.5](#) by preparing the state repeatedly and measure it in the computational basis. The expected parity can then be estimated as

$$\langle \psi_{x,\theta} | P | \psi_{x,\theta} \rangle \approx \frac{1}{S} \sum_{j=1}^S p_j, \quad (4.10)$$

where  $S$  is the number of shots used (repeated measurements), and  $p_j$  is the parity resulting from measurement  $j$ .

## 4.5 Optimization of PQC

A key component of hybrid methods such as PQCs is the optimization of the parameters  $\theta$  entering the ansatz. These parameters are usually optimized with respect to some objective function in order to solve a given problem [\[13\]](#). In the context of machine learning, one seeks to minimize the loss function [Equation \(2.2\)](#) to fit labeled data. There are multiple popular methods for optimization in the context of PQC. One such method is numerical differentiation of the loss function:

$$\frac{\partial}{\partial \theta_i} L(\theta) \approx \frac{L(\theta_1, \dots, \theta_i + \epsilon, \dots, \theta_{n_\theta}) - L(\theta_1, \dots, \theta_i, \dots, \theta_{n_\theta})}{\epsilon}, \quad (4.11)$$

for a sufficiently small  $\epsilon > 0$ . Having an approximation of the gradient, one can optimize the parameters using gradient descent or similar techniques. However, because of the high amount of noise of near-term quantum computers, finite difference approximations of derivatives can be unfavorable in practice. Recently, analytical techniques have been proved to be very efficient for calculating the gradient on quantum computers [\[17, 13\]](#). In the next section, we will detail how this gradient can be calculated.

### 4.5.1 Analytical Gradient-Based Optimization

Based on the derivation presented by Schuld et al. [\[38\]](#), we will now present how the gradient of a large class of PQC's can be calculated on quantum computers using the *parameter shift rule*. Assume we have some circuit parameterized by  $\theta$  that prepares a state  $|\psi_\theta\rangle = U_\theta |0\rangle$ . The expectation value of some observable  $\hat{O}$  can be formulated as

$$a = \langle \psi_\theta | \hat{O} | \psi_\theta \rangle = \langle 0 | U_\theta^\dagger \hat{O} U_\theta | 0 \rangle. \quad (4.12)$$

Assume for that any parameter  $\theta_i$  affects only a single gate. Then, since any circuit can be decomposed into a sequence of gates, we can decompose the circuit as  $U_\theta = AG(\theta_i)B$ , where  $G(\theta_i)$  is the only gate dependent on  $\theta_i$ , and  $A$  and  $B$  are the rest of the circuit. This allows us to rewrite the expectation value as

$$a = \langle \psi' | G(\theta_i)^\dagger \hat{O}' G(\theta_i) | \psi' \rangle, \quad (4.13)$$

where  $|\psi'\rangle = B|0\rangle$  and  $\hat{O}' = A^\dagger \hat{O} A$ . Starting from this expression, it is easy to mathematically compute the derivative of the expectation value. For easier notation, we will use  $\partial_x y$  for the partial derivative  $\frac{\partial y}{\partial x}$ :

$$\partial_{\theta_i} a = \langle \psi' | G(\theta_i)^\dagger \hat{O}' (\partial_{\theta_i} G(\theta_i)) | \psi' \rangle + h.c., \quad (4.14)$$

where h.c. refers to the Hermitian conjugate of the whole term to the left of it. In its current form, the terms of the above expression cannot be computed on a quantum computer since they don't have the form of expectation values. However, it is possible to rewrite it as a linear combination of two expectation values

$$\begin{aligned} \partial_{\theta_i} a = \frac{1}{4} & (\langle \psi' | [G(\theta_i) + 2\partial_{\theta_i} G(\theta_i)]^\dagger \hat{O}' [G(\theta_i) + 2\partial_{\theta_i} G(\theta_i)] | \psi' \rangle - \\ & \langle \psi' | [G(\theta_i) - 2\partial_{\theta_i} G(\theta_i)]^\dagger \hat{O}' [G(\theta_i) - 2\partial_{\theta_i} G(\theta_i)] | \psi' \rangle). \end{aligned} \quad (4.15)$$

Are  $[G(\theta_i) + 2\partial_{\theta_i} G(\theta_i)]$  and  $[G(\theta_i) - 2\partial_{\theta_i} G(\theta_i)]$  unitary operators? In the case that they are not, it is not possible to implement them as circuits. However, for gates such as Pauli rotations, they turn out to be unitary up to a constant factor and actually quite easy to implement. Given that  $G(\theta_i) = R_j(\theta_i) = e^{-i\theta_i \sigma_j/2}$ , where  $j \in [x, y, z]$ , we have that

$$G(\theta_i) \pm 2\partial_{\theta_i} G(\theta_i) = \underbrace{(I \mp i\sigma_j)}_{\sqrt{2}G(\pm\frac{\pi}{2})} G(\theta_i) = \sqrt{2}G(\theta_i \pm \frac{\pi}{2}), \quad (4.16)$$

where the relation  $R_j(m)R_j(n) = R_j(m+n)$  was used in the last step. Inserting this result back into [Equation \(4.15\)](#), we get the final expression

$$\begin{aligned} \partial_{\theta_i} a = \frac{1}{2} & (\langle \psi' | G(\theta_i + \frac{\pi}{2})^\dagger \hat{O}' G(\theta_i + \frac{\pi}{2}) | \psi' \rangle - \\ & \langle \psi' | G(\theta_i - \frac{\pi}{2})^\dagger \hat{O}' G(\theta_i - \frac{\pi}{2}) | \psi' \rangle). \end{aligned} \quad (4.17)$$

The form of the expression above reveals the origin of the name "parameter shift rule". To calculate the derivative of the expectation value of a circuit, one simply has to estimate this expectation value twice: Once with the corresponding parameter shifted by  $\frac{\pi}{2}$ , and once shifted by  $-\frac{\pi}{2}$ . The derivative is finally found by combining

the two results in a linear combination. This is an efficient approach for computing the gradient, since the number of expectation values that needs to be estimated is proportional to the number of parameters.

For QNNs, the features  $\mathbf{x}$  enter the state  $|\psi_{\mathbf{x},\boldsymbol{\theta}}\rangle$  in the same way as the parameters  $\boldsymbol{\theta}$  if qubit encoding is used (see [Section 4.2.1](#)), i.e. with Pauli rotations. In this case, the parameter shift rule can also be applied to calculate the derivative of the output with respect to the input features, i.e.  $\partial_{x_i} a$ . This will be relevant later when we introduce models consisting of multiple circuits.

### 4.5.2 Barren Plateaus in QNN Loss Landscape

While recent studies have shown several promising characteristics of QNNs, such as faster training and greater flexibility [\[17\]](#), these studies have largely been focused on smaller systems and heuristic measures. As such, few scaling relations of QNNs have been rigorously proven. Recently, McClean et al. [\[20\]](#) discovered an important result connecting the magnitude of the gradient to the number of qubits for a large class of PQCs. First, they point out that expectation values measured on a state  $|\psi_n\rangle$  sampled randomly from an  $n$ -qubit Hilbert space tend to concentrate around their mean over Hilbert space as  $n$  increases. Mathematically, we can express this as

$$\lim_{n \rightarrow \infty} V_{|\psi_n\rangle}(\langle\psi_n|\hat{O}|\psi_n\rangle) = 0, \quad (4.18)$$

where  $V_{|\psi_n\rangle}(\cdot)$  calculates the variance over the whole Hilbert space, and  $\hat{O}$  is some observable. This concentration also occurs exponentially fast in  $n$ . Are the states  $|\psi_{\boldsymbol{\theta}}\rangle$  prepared by PQC susceptible to this concentration of expectation values? As stated in [Section 4.3](#), we know that  $U_{\boldsymbol{\theta}}$  can only prepare any random state  $|\psi_{\boldsymbol{\theta}}\rangle = U_{\boldsymbol{\theta}}|\mathbf{0}\rangle$  if it is exponentially deep. However, McClean et al. [\[20\]](#) showed that shallow PQC with polynomial circuit depth were still susceptible for the same exponential concentration of expectation values. Further, they showed that the gradient of PQC has a mean around zero, i.e.

$$E(\partial\theta_i \langle\psi_{\boldsymbol{\theta}}|\hat{O}|\psi_{\boldsymbol{\theta}}\rangle) = 0. \quad (4.19)$$

These two results show that the gradients of PQC with many qubits tend to concentrating around zero. In other words, as the circuit depth and number of qubits grow, PQC essentially approach a random circuit whose gradients vanish exponentially fast.

The vanishing of PQC gradients manifests itself as loss landscapes that are extremely flat in most of parameter space, reminiscent of how the gradients vanish for classical neural networks as the number of layers increase (see [Section 2.3.4](#)). Since the gradients of QNNs are estimated on quantum hardware, an exponentially vanishing gradient requires exponentially many shots in order to estimate it accurately, causing

a very large overhead on the quantum computer. As discussed in [Section 3.2.5](#), using an insufficient number of shots will generally lead to a very bad signal-to-noise ratio when estimating small quantities. If this is the case when estimating the gradient, a bad signal-to-noise ratio can cause gradient-based methods to essentially perform a random walk in parameters space which fails to converge [\[21\]](#). This suggests that QNNs of many qubits and large circuit depths are intractable to train in practice.

## 4.6 Quantum Circuit Network

We extend the QNN framework discussed so far by implementing multi-circuit models consisting of several such QNNs. These models exhibit a network-like structure, consisting of layers of parameterized circuits. These layers of circuits transform input features in a sequential manner until a model prediction is obtained. To avoid confusion with the similarly themed "quantum neural networks", we have opted to call the multi-circuit model a *quantum circuit network* (QCN). This type of architecture was explored by Bilek [\[22\]](#) and was found able to sufficiently fit a nonlinear function in one dimension when optimized with Nelder-Meads algorithm, a gradient-free optimization algorithm.

### 4.6.1 Feed-Forward

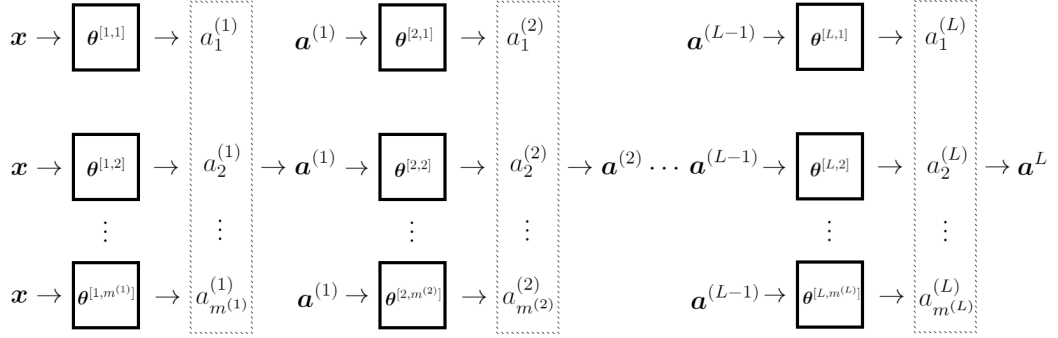
[Figure 4.5](#) illustrates the general structure of a QCN, which exhibits a neural network-like architecture. Here, each node in the network is a QNN model  $f_{QNN}^{(l)}$  (see [Equation \(4.3\)](#)), with some layer specific choice of encoder, ansatz and observable. Each QNN is parameterized by  $\theta^{[l,j]}$ , where  $l$  is the layer and  $j$  is the node in the current layer. The feed-forward procedure is described as follows: For layer  $l$ , each node receives the feature vector  $\mathbf{a}^{(l-1)}$  resulting from the previous layer (with the special case that  $\mathbf{a}^{(0)} = \mathbf{x}$ ). For each node  $j$  in layer  $l$ , an output  $a_k^{(l)}$  is produced, i.e.

$$a_j^{(l)} = f_{QNN}^{(l)}(\mathbf{a}^{(l-1)}; \theta^{[l,j]}) \quad (4.20)$$

These outputs are then concatenated to make a new feature vector  $\mathbf{a}^{(l)} = (a_1^l, \dots, a_{m^{(l)}}^l)$ , where  $m^{(l)}$  is the number of nodes in layer  $l$ . This is then repeated for each layer 1 through  $L$ . Finally, the output of the last layer is identified as the model output

$$\hat{y} = f_{QCN}(\mathbf{x}; \theta) = \mathbf{a}^{(L)}, \quad (4.21)$$

where  $\theta$  is the collection of all  $\theta^{[l,j]}$ .



**Figure 4.5:** General structure of a quantum circuit network. Each node, indicated by a box, is a QNN model parameterized by  $\theta^{[l,j]}$ , where  $l$  is the layer and  $j$  is the node in the that layer.  $m^{(l)}$  is the number of nodes in layer  $l$ . For layer  $l$ , each node receives the feature vector  $\mathbf{a}^{(l-1)}$  resulting from the previous layer (with the special case that  $\mathbf{a}^{(0)} = \mathbf{x}$ ). For each node  $j$ , an output  $a_j^{(l)}$  is produced, which is then concatenated to make a new feature vector  $\mathbf{a}^{(l)}$ . This is then repeated for each layer 1 through  $L$ . Finally, the output of the last layer is identified as the model output  $\hat{y} = \mathbf{a}^{(L)}$ .

Because of the sequential nature of QCNs, the models can be evaluated one circuit at a time during feedforward and the resulting values  $a_j^{(l)}$  can be stored on a classical computer. At the expense of more circuit evaluation, this enables the possibility of constructing large QCNs using several shallow circuit. Quantum neural networks, on the other hand, must execute the whole procedure coherently on a quantum computer to derive a model output. For QNNs with high circuit depth, this may be unsuitable on noisy quantum hardware due to decoherence, as explained in [Section 3.3](#).

Another interesting aspect of QCNs is that their sequential nature actually incorporate nonlinear transformations of the input for every layer, much like dense neural networks [Equation \(2.13\)](#). When estimating the outputs of any layer using [Equation \(4.20\)](#), we know from [Section 3.2.5](#) that the estimated expectation value relates to the amplitudes  $\alpha_i$  of the state via the squared value  $|\alpha_i|^2$ . This causes a nonlinear relationship between the inputs and outputs. This type of nonlinearity is introduced for every layer for QCNs, but only once for QNNs. This could potentially enable QCNs to learn more complicated functions compared to QNNs.

#### 4.6.2 Backward Propagation

When comparing the formulation of QCNs [Equation \(4.21\)](#) and DNNs [Equation \(2.13\)](#), it becomes apparent that they are structurally identical, excluding the mathematical operations happening inside each node. For the QCN, each node implements a QNN model, while the DNNs implements a weighted sum of the inputs, followed by a nonlinear activation. Using this observation, it is possible

to implement a slightly modified backpropagation algorithm for calculating the gradient of QCNs analytically. This assumes that we are able to calculate the derivative of the outputs of each QCN node. As described in [Section 4.5.1](#), this is indeed possible using the parameter shift rule.

For  $\hat{y} = f_{QCN}(\mathbf{x}; \boldsymbol{\theta})$  and a loss function  $L(\hat{y}, y)$ , the error of the last layer can be computed as

$$\delta_k^L = \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^L}, \quad (4.22)$$

where  $k$  indicates the node. This error can be defined for any layer recursively by repeated application of the chain-rule:

$$\delta_j^l = \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_j^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^{l+1}} \frac{\partial \mathbf{a}_k^{l+1}}{\partial \mathbf{a}_j^l} = \sum_k \delta_k^{l+1} \frac{\partial \mathbf{a}_k^{l+1}}{\partial \mathbf{a}_j^l}. \quad (4.23)$$

The derivative of the loss function with respect to any parameter in any node can then be calculated as

$$\frac{\partial L(\hat{y}, y)}{\partial \theta_n^{[l,j]}} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \mathbf{a}_k^l} \frac{\partial \mathbf{a}_k^l}{\partial \theta_n^{[l,j]}} = \delta_j^l \frac{\partial \mathbf{a}_j^l}{\partial \theta_n^{[l,j]}}, \quad (4.24)$$

where it was used that  $\frac{\partial \mathbf{a}_k^l}{\partial \theta_n^{[l,j]}} = 0$  for  $k \neq j$ , since the output of node  $k$  is independent of the parameters in node  $j$ . The terms

$$\frac{\partial \mathbf{a}_j^l}{\partial \theta_n^{[l,j]}} \quad (4.25)$$

$$\frac{\partial \mathbf{a}_k^{l+1}}{\partial \mathbf{a}_j^l} \quad (4.26)$$

are the derivatives of the node outputs with respect to the parameters and inputs, respectively. Since they are calculated locally for each node, we call them the *local gradients* of the QCN model. As explained in [Section 4.5.1](#), the local gradients can be calculated analytically using the parameter shift rule with respect to the parameters  $\theta_n^{[l,j]}$  and the inputs  $\mathbf{a}_j^l$ . By first performing a forward pass to calculate  $\mathbf{a}^l$  for all the layers  $l$ , the local gradients can then be estimated and stored one at a time. Finally, [Equation \(4.24\)](#) can be used to classically compute the *total gradient*  $\nabla_{\boldsymbol{\theta}} L(\hat{y}, y)$  based on the stored values for the single sample  $\mathbf{x}$ . By repeating this for all samples  $\mathbf{x}^{(i)}$ , we can calculate the average gradient

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \nabla_{\boldsymbol{\theta}} L(\hat{y}^{(i)}, y^{(i)}). \quad (4.27)$$

The calculation of the gradient allows us to leverage more information of the loss function and enables for gradient-based optimization. Potentially, this could

mean faster optimization relative to derivative-free optimization, such as scikit-learn's numerical optimizer algorithm [28] which was originally used to train QCNs [22]. However, as explained in [Section 2.2.1](#), gradient-based methods such as gradient descent are prone to getting stuck in local minima, potentially causing slow optimization.



# 5

## Tools for Analysis

In this chapter, we introduce the various numerical methods used for investigating the models used in this thesis.

### 5.1 Trainability

In machine learning, *trainability* refers to how easily a particular model can be trained under different conditions [17]. A common way to assess the trainability is by investigating the geometry of the loss landscape. For example, the loss function of dense neural networks exhibits local flatness for most directions in parameter space, and strong distortion in others [8]. In a loss landscape that is mostly flat, the gradient of the model tends to diminish, known as *vanishing gradient*, making it difficult to train the model using gradient-based methods. This problem is known to worsen with the number of layers, making the training of deep models prohibitive.

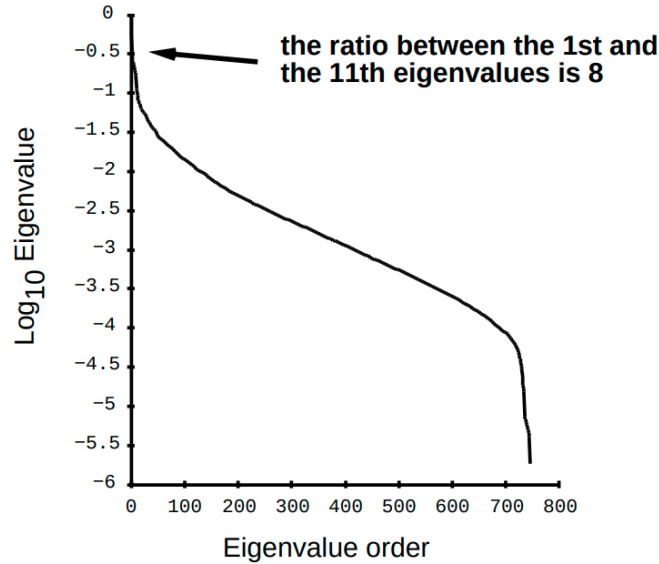
To investigate the flatness and distortions of the loss landscape, a common metric to use is the *Hessian* of the loss, which we will introduce in the next section.

#### 5.1.1 Hessian Matrix

Let  $f(\mathbf{x}^{(k)}; \boldsymbol{\theta})$  be a parameterized and differentiable model, where  $\mathbf{x}^{(k)} \in \mathbb{R}^p$  are  $p$  features, and  $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$  are  $n_\theta$  model parameters. For a general loss function on the form Equation (2.2),  $L(\boldsymbol{\theta}) = \sum_{k=1}^N L(f(\mathbf{x}^{(k)}; \boldsymbol{\theta}), y^{(k)})$ , where  $N$  is the number of samples in the data set, the Hessian matrix of the loss function is given by

$$H_{ij} = \frac{\partial^2 L(\boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}. \quad (5.1)$$

The Hessian matrix is an  $n_\theta \times n_\theta$  matrix that quantifies the curvature of the loss function locally in the parameter space at the point of  $\boldsymbol{\theta}$ . This is an extensively studied quantity in the machine learning community, and it has been used to study the loss landscape both for classical and quantum machine learning models [7, 39]. In particular, its eigenvalue spectrum quantifies the amount of curvature in various directions. Typically for classical neural networks, the spectrum is characterized by the presence of many eigenvalues near zero, with the exception of a few large ones (so-called "big killers") [7]. This indicates that the loss landscape is mostly flat, with large distortions in a few directions, which in turn causes slow optimization as discussed earlier. Figure 5.1 shows an example of this.



**Figure 5.1:** Figure showing an example of a strongly skewed spectrum of the Hessian for a classical neural network. Most eigenvalues are close to zero, with the presence a few large ones. This figure is retrieved from LeCun et al. [7].

### 5.1.2 Empirical Fisher Information Matrix

An apparent shortcoming of the Hessian matrix Equation (5.1) is the large computational cost of computing it, requiring the evaluation of  $\mathcal{O}(n_\theta^2)$  double derivatives. This is particularly expensive for models of many parameters, which e.g. neural networks tend to be. An alternative and related quantity, called the *Empirical Fisher Information Matrix* (EFIM) [8], can be calculated using  $\mathcal{O}(n_\theta)$  first order derivatives, which is much better suited for big models. We will now derive the EFIM and relate it to the Hessian matrix.

Assume a square loss  $\frac{1}{2N} \sum_{k=1}^N (f(\mathbf{x}^{(k)}; \boldsymbol{\theta}) - y^{(k)})^2$ . Computing Equation (5.1) with this loss results in

$$H_{ij} = F_{ij} - \frac{1}{N} \sum_{k=1}^N (y^{(k)} - f(\mathbf{x}^{(k)}; \boldsymbol{\theta})) \frac{\partial^2 f(\mathbf{x}^{(k)}; \boldsymbol{\theta})}{\partial \theta_i \partial \theta_j}, \quad (5.2)$$

where F is identified as the EFIM, given by

$$F_{ij} = \frac{1}{N} \sum_{k=1}^N \frac{\partial f(\mathbf{x}_k; \boldsymbol{\theta})}{\partial \theta_i} \frac{\partial f(\mathbf{x}_k; \boldsymbol{\theta})}{\partial \theta_j}. \quad (5.3)$$

From Equation (5.2), the EFIM can be seen to coincide with the Hessian matrix if  $f(\mathbf{x}^{(k)}; \boldsymbol{\theta}) = y^{(k)}$ , since the terms in the last sum vanishes. This is the case if the model manages to perfectly replicate the targets from the features, which is approximately true for well-trained models that fit the data sufficiently. However, even for untrained models, the EFIM is sometimes used as a cheaper alternative to the Hessian matrix, particularly for investigating the geometry of the loss landscape via its eigenvalue spectrum. This has been done both for classical and quantum mechanical machine learning models [8, 17]. It is worth pointing out that these investigations, as well as this thesis, are mainly concerned with untrained models. Consequently, the EFIM does not coincide with the Hessian matrix and does not give a mathematically accurate description of the curvature of the loss landscape. However, the EFIM still serves as a heuristic for addressing the flatness and distortions of the loss landscape.

## 5.2 Expressivity

*Expressivity* in machine learning, especially in the context of neural networks, is a way of characterizing how architectural properties of a model affect the space of functions it can compute Raghu et al. [6]. More simply put, expressivity measures how flexible and complex the model is. The first attempts to measure expressivity of neural networks took a highly theoretical approach, for example when Bartlett et al. [40] calculated the VC dimension of shallow neural networks. The VC dimension, or *Vapnik–Chervonenkis* dimension [4], is a well-established measure of complexity. However, it is known to be hard to compute in practice for a variety of models [17].

### 5.2.1 Trajectory Length

In order to assess the expressivity of deep neural networks, Raghu et al. [6] introduced a more practical alternative to VC dimension called *trajectory length*. This is an easy-to-compute heuristic that measures how small perturbations in the input of neural networks grow as it is passed through the various layers of the model.

Given a trajectory  $\mathbf{x}(t)$  in a  $p$ -dimensional space, its arc length  $l(\mathbf{x}(t))$  is given by

$$l(\mathbf{x}(t)) = \int_t \left\| \frac{d\mathbf{x}(t)}{dt} \right\| dt \quad (5.4)$$

where  $\|\cdot\|$  indicates the Euclidean norm. Conceptually, the arc length of the trajectory  $\mathbf{x}(t)$  is the sum of the norm of its infinitesimal segments. By approximating the trajectory with a finite number of points  $\mathbf{x}(t_i)$ , its arc length can be estimated as

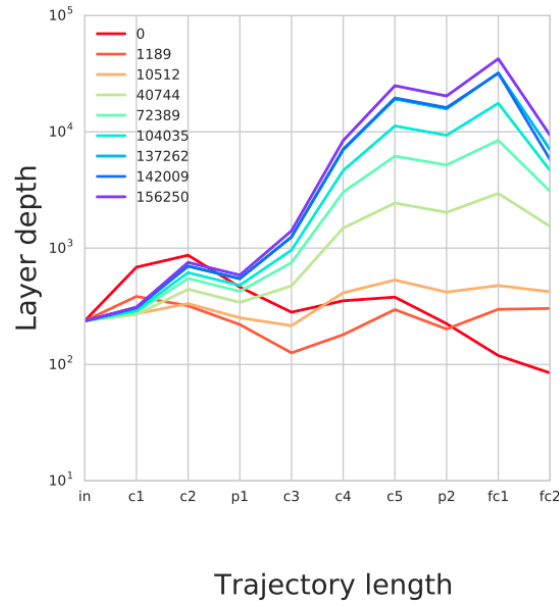
$$l(\mathbf{x}(t)) \approx \sum_{i=1}^{N-1} \|\mathbf{x}(t_{i+1}) - \mathbf{x}(t_i)\|. \quad (5.5)$$

By making an appropriate trajectory  $\mathbf{x}(t_i)$  in some input space, it is possible to investigate how its length changes as it is passed through each layer of a neural network. To be concrete, the quantity of interest is  $l(\mathbf{a}^l(t_i))$ , where  $\mathbf{a}^l(t_i)$  are the outputs of layer  $l$  resulting from the input  $\mathbf{x}(t_i)$  for some neural network. As an example, one can make a trajectory  $\mathbf{x}(t_i) \in \mathbb{R}^2$  in the shape of a circle. By projecting  $\mathbf{a}^l(t_i)$  down to two dimensions, it is possible to visualize how each layer of the neural network distorts the trajectory. This has been exemplified in Figure 5.2.



**Figure 5.2:** Figure showing a trajectory increasing with the depth of a network. Starting with a circular trajectory (left most pane), it is fed through a fully connected tanh network with width 100. Pane second from left shows the image of the circular trajectory (projected down to two dimensions) after being transformed by the first hidden layer. Subsequent panes show the trajectory after being transformed by multiple layers. This figure is retrieved from Raghu et al. [6].

Figure 5.2 shows that the inputs gets transformed in a highly nonlinear way as they are being transformed by each layer. Especially, neighboring points in the input trajectory get mapped further and further apart for each transformation, indicating that small perturbations in the input grow for each layer. Raghu et al. [6] showed that the trajectory length of neural networks increase exponentially with depth. The authors also showed that untrained neural networks that initially lacked this feature could be brought into the exponential regime by training them on data. This is exemplified in Figure 5.3, showing that neural networks can be trained to compute exponentially complex functions.



**Figure 5.3:** Figure showing the trajectory length of a neural network as it is trained on data. The legend shows the number of epochs. After about  $7 \times 10^4$  epochs, the neural network enter the exponential regime, where each additional layer (more or less) increases the trajectory length exponentially. The figure is retrieved from Raghu et al. [6].

## Part II

# Implementation

# 6

## Implementation

In this chapter, we will present details surrounding implementation of algorithms and methods presented in [Part I](#). For this thesis, we have developed an Python framework for doing machine learning, capable of implementing dense neural networks (DNN) [Equation \(2.13\)](#), quantum neural networks (QNNs) [Equation \(4.3\)](#) and quantum circuit networks (QCN) [Equation \(4.21\)](#). In addition, various numerical tools for analyzing the models are available. The code base is object-orientated using Python, focusing on flexibility. This grants a high degree freedom when specifying model architecture, such as setting the number of layers, number of nodes, type of activation functions, loss function and optimizer. The frame work is also capable of implementing hybrid models mixing both DNN and QCN layers. To allow implementation of quantum machine learning, the frame work is built around Qiskit [\[24\]](#), an IBM-made python-package used for emulating quantum circuits.

All source code developed for this thesis can be found on our GitHub page <https://github.com/KristianWold/Master-Thesis>, together with notebooks containing training of models, generation of data, analysis and plotting. For easier reading, all python types referred to in this thesis will be highlighted in **bold**. All simulations in this thesis were performed using an Ubuntu desktop computer equipped with a Ryzen 3900x CPU and 32 GB RAM.

### 6.1 Qiskit

Qiskit [\[24\]](#) is an open source Python-package used for practically emulating quantum circuits and quantum algorithms. It can be installed using pip with the following command:

```
$ pip install qiskit
```

To import the package, include the following in any python scrip:

```
import qiskit as qk
```

### 6.1.1 Registers and Circuits

To create a quantum circuit in Qiskit, one can first create one or more *quantum registers*, which are list structures containing qubits. The registers can be put together into a circuit in the following way:

```
1 q_reg_1 = qk.QuantumRegister(2)
2 q_reg_2 = qk.QuantumRegister(2)
3 c_reg = qk.ClassicalRegister(2)
4 circuit = qk.QuantumCircuit(q_reg_1, q_reg_2, c_reg)
```

Here, each of the registers **q\_reg\_1** and **q\_reg\_2** contain two qubits. By default, they are each initialized in the state  $|0\rangle$ . Thus, the total circuit can be written as

$$|00\rangle |00\rangle, \quad (6.1)$$

where each *ket* refers to one register. Further, **c\_reg** is a *classical register* of classical bits, meant for storing classical information when the circuit is later measured. In general, a circuit can contain any number of quantum registers with any number of qubits. However, if one wishes to include a classical register, it must be included as the last argument in **qk.QuantumCircuit()**.

### 6.1.2 Applying Gates

Continuing after creating our circuit, we can apply a variety of quantum gates by calling different methods for the **circuit** object. Hadamard gates can be applied to the two qubits of register **q\_reg\_1** in the following way:

```
1 circuit.h(q_reg_1[0])
2 circuit.h(q_reg_1[1])
```

This prepares the state

$$|00\rangle |00\rangle \rightarrow \left( \frac{1}{2} |00\rangle + \frac{1}{2} |01\rangle + \frac{1}{2} |10\rangle + \frac{1}{2} |11\rangle \right) |00\rangle.$$

A possible way of creating entanglement between the registers is to use CNOT gates on **q\_reg\_2** conditioned on the qubits in **q\_reg\_1**:

```
1 circuit.cx(q_reg_1[0], q_reg_2[0])
2 circuit.cx(q_reg_1[1], q_reg_2[1])
```



resulting in the state

$$\left(\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle\right)|00\rangle \rightarrow \frac{1}{2}|00\rangle|00\rangle + \frac{1}{2}|01\rangle|01\rangle + \frac{1}{2}|10\rangle|10\rangle + \frac{1}{2}|11\rangle|11\rangle.$$

For a complete documentation of the gates available in Qiskit, see <https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html>.

### 6.1.3 Measurement

To measure the state of `q_reg_2` in the computational basis and store it to `c_reg`, we make use of the method `measure()`:

```
1 circuit.measure(q_reg_2, c_reg)
```

Finally, to repeatedly execute the circuit and sample the results, we make use of `qk.execute` together with the backend `qasm_simulator`:

```
1 job = qk.execute(circuit,
2                 backend =
3                 qk.Aer.get_backend("qasm_simulator"),
4                 shots = 1000)
5 result = job.result().get_counts(circuit)
6 print(result)
```

$\rightarrow \{'00': 261, '01': 242, '10': 253, '11': 244\}$

This results in a python dictionary whose keys are strings indicating the different states that were measured. The value corresponding to each key is the number of times that state was measured. To set the number of times to execute and measure the circuit, the argument `shots` may be used. In this case, it was set to 1000. Using `qk.Aer.get_backend("qasm_simulator")` as the backend, the circuit is simulated locally on the classical machine in an ideal fashion, meaning the imperfections of real quantum computers as described in Section 3.3 are disregarded. Still, since a finite number of shots was used, the normalized results  $\frac{261}{1000}$ ,  $\frac{242}{1000}$ ,  $\frac{253}{1000}$  and  $\frac{244}{1000}$  only approximate the exact value  $\frac{1}{4}$ .

### 6.1.4 Exact Expectation Value

When simulating circuits on classical hardware, e.g. with Qiskit, we have the luxury of accessing the state resulting from some computation directly and calculate exact expectation values. As explained in Section 3.1.3, this is not possible when using real quantum computers. Having access to exact expectation values is useful when the noise of finite sampling or realistic hardware is uninteresting for the analysis.

To access the exact state, the classical register `c_reg` must be omitted from the circuit, as we never plan to measure it in the ordinary way. Then, the circuit is simulated using the *statevector\_simulator* backend:

```

1 job = qk.execute(circuit,
2                   backend =
3                   qk.Aer.get_backend("statevector_simulator"))
4 result = job.result().get_counts(circuit)
5 print(result)

```

$\rightarrow \{ '00' : 0.25, '01' : 0.25, '10' : 0.25, '11' : 0.25 \}$

Here, the "measurements" are exact and normalized, as if infinitely many shots were used for sampling.

### 6.1.5 Simulating Real Devices

Even though a given quantum algorithm may produce a satisfying result when executed on idealized hardware, like simulated with Qiskit, this may not apply when run on real hardware. This is due to the introduced noise and inaccuracies as explained in [Section 3.3](#). To produce a more realistic result, Qiskit allows for realistic simulation of many noisy IBM quantum computers by choosing an appropriate backend.

#### Noisy Backend

There are three main components for simulating noisy devices. The first is the *noise model*, which is responsible for implementing noise and inaccuracies present in the real hardware. In Qiskit, noise models have 4 components that determine the total model:

- **Decoherence** Error resulting from interactions from the environment on the qubits, such as thermal noise.
- **Gate Error:** Error resulting from application of gates.
- **Gate Length:** Error resulting from the length of the gate.
- **Measurement Error:** Error resulting from the measurement of the qubits.

The next component is a *coupling map* that defines how the different qubits couple, i.e. what pairs of qubits that can interact via two-qubits gates (see [Section 3.3](#)). The last component defines the basis gates of the quantum computer.

In this thesis, we will be using the IBM quantum computer Santiago [\[41\]](#), which has 5 qubits with only linear connectivity between the qubits. This choice is deliberate, as qubit encoding and the simple ansatz are specially made to work efficiently

on such architectures. To access and store the backend simulating this quantum computer, we can use the following code:

```
1 import pickle
2 from qiskit.providers.aer import QasmSimulator
3 from qiskit import IBMQ
4
5 IBMQ.enable_account(token)
6 backend = provider.get_backend('ibmq_santiago')
7 backend = QasmSimulator.from_backend(backend)
8 pickle.dump(backend, open("backend_santiago", "wb"))
```

Here, **token** is a string identifying an IBM account necessary for accessing the correct backend. **backend = QasmSimulator.from\_backend(backend)** pulls the necessary components describes earlier and implements them into the **Qasm-Simulator** backend, ready to be used for noisy simulation.

### Transpiling

When using a noisy backend, it is important to *transpile* the circuit prior to simulation such that gates of the circuit are decomposed into the correct basis gates. It is also important that the transpiled circuit respects the coupling map of the target quantum computer. Transpiling a circuit with respect to a given backend is done automatically when executing the circuit:

```
1 job = qk.execute(circuit, noisy_backend, seed_transpiler=42,
    seed_simulator=42)
```

Here, the **seed\_transpiler** and **seed\_simulator** are seeds that make the work of the transpiler and the noise model reproducible, which is otherwise random in nature. Transpiling a circuit will also apply a light optimization by attempting to reduce the number of basis gates used. This is particularly important, since we want the depth to be as low as possible. Heavier optimization routines may be used to reduce the circuit depth even more, with the price of more time spent transpiling.

## 6.2 QNN Example

We will now present a possible way of implementing a QNN [Equation \(4.3\)](#) for fitting data. The model will be optimized using the parameter shift rule and gradient descent.

### 6.2.1 Encoding

To encode input features, we will make a callable object on the form **circuit = encoder(circuit, data\_reg, data)**, where **data\_reg** is a quantum register of **circuit**, and **data** is a numpy array containing the features of one sample. Qubit

encoding using  $R_x$  rotations, as presented in [Section 4.2.1](#), can be implemented as the following function:

```

1 def qubit_encoder(circuit, data_reg, data):
2     for i, x in enumerate(data):
3         circuit.rx(x, data_register[i])
4
5     return circuit

```

This function requires that the number of features does not exceed the number of qubits.

### 6.2.2 Ansatz

To implement the ansatz, we want a callable object on the form **circuit = ansatz(circuit, data\_reg, theta)**, where **theta** is a numpy array containing the parameters. The "simple ansatz" detailed in [Section 4.3](#) can be implemented as a python function:

```

1 def simple_ansatz(circuit, data_reg, theta):
2     n_qubits = data_reg.size()
3
4     for i in range(n_qubits - 1):
5         circuit.cx(data_reg[i], data_reg[i+1])
6
7     for i, w in enumerate(theta):
8         circuit.ry(w, data_register[i])
9
10    return circuit

```

Here, **data\_reg.size()** was used to retrieve the number of qubits present in the register. The first for-loop entangles the qubits in sequence using CNOT gates. The last for-loop applies an  $R_y$  rotation to each qubit corresponding to the parameters.

### 6.2.3 Model Output

To derive a model output, we must estimate an expectation value as explained in [Section 4.4](#). We can do this by making a callable object **y\_pred = sampler(counts)**, where **counts** is a python dictionary containing the measuring results, as explained in [Section 6.1.3](#) and [Section 6.1.4](#). We can implement a function estimating the parity (see [Section 4.4](#)) in the following way:

```

1 def parity(counts):
2     shots = sum(counts.values())
3
4     output = 0
5     for bitstring, samples in counts.items():

```

```

6         if parity_of_bitstring(bitstring) == 1:
7             output += samples
8
9         output = output / shots
10
11     return output

```

where **parity\_of\_bitstring(bitstring)** is a function that calculates the parity of a bitstring, which can be implemented as

```

1 def parity_of_bitstring(bitstring):
2     binary = [int(i) for i in bitstring]
3     parity = sum(binary) % 2
4
5     return parity

```

The function **parity()** iterates over the different measured states and makes a weighted average of their parities, resulting in the estimation of the average parity of the state.

To make a prediction, we can implement a function **qnn(x, theta)**, where **x** is a numpy array containing features of a single sample, and **theta** is a numpy array containing parameters. The function can be implemented as

```

1 def qnn(x, theta):
2     n_qubits = len(x)
3     data_reg = qk.QuantumRegister(n_qubits)
4     clas_reg = qk.ClassicalRegister(n_qubits)
5     circuit = qk.QuantumCircuit(data_reg, clas_reg)
6
7     circuit = qubit_encoder(circuit, data_reg, x)
8     circuit = simple_ansatz(circuit, theta)
9
10    job = qk.execute(circuit,
11                     backend =
12                     qk.Aer.get_backend("qasm_simulator"),
13                     shots = 1000)
14    counts = job.result().get_counts(circuit)
15    y_pred = parity(counts)
16
17    return y_pred

```

### 6.2.4 Gradient

Having a function that implements a QNN that produces a model output, we can apply the parameter shift rule described in [Section 4.5.1](#) to calculate the gradient of the output with respect to the parameters:

```

1 def gradient(x, theta):
2     deriv_plus = np.zeros(len(theta))
3     deriv_minus = np.zeros(len(theta))
4
5     for i in range(len(theta)):
6         theta[i] += np.pi/2 #parameter shifted forward
7         deriv_plus[i] = qnn(x, theta)
8
9         theta[i] -= np.pi #parameter shifted backwards
10        deriv_minus[i] = qnn(x, theta)
11
12        theta[i] += np.pi/2 #parameter reset
13
14    return 0.5*(deriv_plus - deriv_minus) #linear combination

```

This function returns a numpy array with the same length as **theta**, containing the derivatives  $\frac{\partial y}{\partial \theta_i}$ .

### 6.2.5 Training

Finally, we can implement a function **train(x\_list, y\_list, theta, lr, epochs)** that calculates the average gradient [Equation \(2.18\)](#) resulting from the samples and targets **x\_list** and **y\_list**, using MSE loss. The function then updates the parameters **theta** iteratively using gradient descent with learning rate **lr**:

```

1 def train(x_list, y_list, theta, lr, epochs):
2     loss = []
3     for i in range(epochs):
4         grad = np.zeros(len(theta))
5         loss.append(0)
6
7         for x, y in zip(x_list, y_list):
8             y_pred = qnn(x, theta) #prediction
9             loss[-1] += (y_pred - y)**2 #accumulate loss
10            grad = grad + (y_pred - y)*gradient(x, theta)
11
12            loss[-1] = loss[-1]/len(y) #normalize
13            grad = grad/len(y)
14            theta += -lr*grad #update parameters
15
16    return theta, loss

```

At line 10, **grad = grad + (y\_pred - y)\*gradient(x, theta)** accumulates the gradients of the MSE loss function with respect to the parameters, i.e. [Equation \(2.7\)](#). The parameters are then updated at line 14 using gradient descent.

## 6.3 Quantum Circuit Network

In this section, we will introduce how our framework can be used to implement various QCN architectures, and how these can be used to fit data. We will also show how the framework can be used to implements single-circuit QNNs, DNNs, and hybrid models combining QCNs and DNNs.

### 6.3.1 Encoders, Ansatzes and Samplers

We implement encoders, ansatzes and samplers as callable python classes with much the same functionality as described in [Section 6.2](#). We will now go through the use of the most important classes.

#### Encoder

The encoder class **Encoder** can be instantiated for implementing qubit encoding ([Figure 4.2](#)) as

```
1 from encoders import QubitEncoder
2
3 encoder = QubitEncoder(mode)
```

Here, **mode** is a string that specifies the rotations used for encoding, either "x", "y" or "z". See [Section 4.2.1](#) for details.

#### Ansatz

The ansatz class **Ansatz** can be instantiated as

```
1 from ansatzes import Ansatz
2
3 ansatz = Ansatz(block, reps)
```

Here, **block** is a python list containing strings that specify gates applied to the circuit. For example, **block** = ["entangle", "ry"] will first apply CNOT gates to all neighboring qubits in sequence.  $R_y$  rotations are then applied to every qubit. This particular argument recreates the simple ansatz described in [Section 4.3](#). **reps** specifies the number of times the ansatz is then repeated.

#### Parity

The sampler class **Parity** can be instantiated as

```
1 from samplers import Parity
2
3 sampler = Parity()
```

This class implements the same functionality as the **parity** function described in [Section 6.2](#).

### 6.3.2 QLayer

Our framework for constructing QCN models implements layers consisting parameterized circuits, as explained in [Section 4.6](#). A QCN layer can be created as a python object of the type **QLayer** as follows:

```

1 from layers import QLayer
2
3 qlayer = QLayer(n_qubits,    #number of qubits in each QNN
4                 n_features,  #number of input features
5                 n_targets,   #number of outputs, i.e. nodes
6                 scale,       #scaling of output
7                 encoder,
8                 ansatz,
9                 sampler,
10                backend,
11                shots)

```

The arguments **encoder**, **ansatz** and **sampler** define the architecture of the circuits in the layer. Examples of possible choices are described in [Section 6.3.1](#). As an example, a **QLayer** can be instantiated and used on a data in the following way:

```

1 import numpy as np
2 from layers import QLayer
3 backend = qk.Aer.get_backend("qasm_simulator")
4
5 x = np.random.normal((4,3))
6
7 np.random.seed(42)
8 qlayer = QLayer(n_qubits = 3,
9                 n_features = 3,
10                 n_targets = 2,
11                 scale = 2*np.pi,
12                 encoder = Encoder(mode = "x"),
13                 ansatz = Ansatz(blocks=["entangle", "ry"],
14                                     reps = 2),
15                 sampler = Parity(),
16                 backend = backend,
17                 shots=1000)
18
19 y_pred = qlayer(x)
20 print(y_pred)

```

```

→ [[5.19619425 4.29769875]
    [3.84530941 2.14256619]
    [3.08504399 2.19283167]
    [4.09663682 2.58867235]]

```



Here, **x** is a dataset containing four samples of three features each. By specifying **n\_targets=2**, the resulting layer will consist of two nodes and thus produces two output targets. The layer is callable, and performs prediction on the input **x** sample-wise. If the number of shots are set to zero, i.e. **shots=0**, the outputs are exactly calculated with the *statevector\_simulator* backend, as explained in [Section 6.1.4](#).

### 6.3.3 Constructing QCNs from QLayers

In general, a QCN can be constructed with any number of layers, with any number of inputs and outputs. The only constraint is that the outputs of one layer and the inputs of a subsequent layer must match in shape. A two-layer QNC can be constructed in the following way using the **NeuralNetwork** class:

```

1 from neuralnetwork import NeuralNetwork
2
3 x = np.random.normal(0, 1, (4,3))
4
5 #unspecified arguments assumes default values
6 layer1 = QLayer(n_qubits=3,
7                 n_features=3,
8                 n_targets=4)
9
10 layer2 = QLayer(n_qubits=4,
11                 n_features=4,
12                 n_targets=1)
13
14 network = NeuralNetwork(layers = [layer1, layer2],
15                           cost = MSE(),
16                           optimizer = Adam(lr=0.1))
17 y_pred = network.predict(x)
18 print(y_pred)

```

In the above code, the **NeuralNetwork** class stores the layer objects in a python list **self.layers**. When doing prediction, the class implements feed forward, as described in [Section 4.6.1](#), using a **\_\_call\_\_** method:

```

1 def __call__(self, x):
2     self.a = []
3     self.a.append(x)
4     for layer in self.layers:
5         x = layer(x)
6         self.a.append(x)

```

The outputs of all layers are stored, as they are needed during backpropagation. **network.predict(x)** returns only the output of the last layer, i.e. the model output.

### 6.3.4 Backpropagation

The class **NeuralNetwork** performs back propagation, as described in [Section 4.6.2](#), using a class method **network.backward(x,y)**. In simplified terms, the method is implemented as:

```

1 def backward(self, x, y):
2     self(x)                #feed forward
3     y_pred = self.a[-1]    #inference
4     delta = self.cost.derivative(y_pred, y)
5
6     #work thru layers in reverse
7     for i, layer in reversed(list(enumerate(self.layers))):
8         weight_gradient, delta = layer.grad(self.a[i],
9                                             delta)
10        self.weight_gradient_list.append(weight_gradient)
11
12    self.weight_gradient_list.reverse()
```

**network.backward(x,y)** starts by performing feed forward and predictions. In line 4, the error of the last layer [Equation \(4.22\)](#) is initiated as **delta** based on the specific loss function **self.cost**. For each layer, starting with the last, the gradient is calculated and the error **delta** is updated. This is done using [Equation \(4.24\)](#) and [Equation \(4.23\)](#), respectively, which is implemented in the layer method **layer.grad(self.a[i], delta)**.

### 6.3.5 Training

To train the QCN, the class method **network.train(self, x, y, epochs)** can be used. In simplified terms, it is implemented as

```

1 def train(self, x, y):
2     self.loss = []
3     for i in range(epochs):
4         self.backward(x, y)
5         self.step()
6
7         y_pred = self.a[-1]
8         self.loss.append(self.cost(y_pred, y))
9
10    y_pred = self.predict(x)
11    self.loss.append(self.cost(y_pred, y))
```

The method for training starts by calling **self.backward** in order to calculate and store the gradient in **self.weight\_gradient\_list**. Then, the method **self.step()** is used to update the parameters of the layers. This is done using [Equation \(2.10\)](#), but with the gradient modified by the specified optimizer. This is then repeated a number of times specified by **epochs**.

### 6.3.6 Single-Circuit Models

Using the **NeuralNetwork**, it is possible to construct single-circuit QNNs in addition to the usual multi-circuit QCNs. This can be done using a single **QLayer** with a single node, as this will constitute only one QNN:

```

1 layer = QLayer(n_qubits = 4,
2                 n_features = 4,
3                 n_targets = 1,
4                 encoder = RZZEncoder(),
5                 ansatz = Ansatz(blocks=["entangle", "ry"],
6                                   reps=2),
7                 sampler = Parity(),
8                 backend = backend,
9                 shots = 1000)
10
11
12 qnn_model = NeuralNetwork(layers = [layer1])

```

As there are no intermediate layers, we do not need to compute the derivative of the node outputs with respect to their inputs. This opens up for ways of encoding where the parameter shift rule, as implemented in [Section 4.5.1](#) and [Section 6.2.4](#), fails. A possible choice is RZZ encoding, as used in the above code.

### 6.3.7 Dense Neural Networks

In addition to **QLayer** layers, the neural network framework also implements **Dense** layers, which are densely connected layers as defined in [Section 2.3](#). It can be instantiated in the following way, and put together to form a DNN:

```

1 from layers import Dense, Sigmoid
2
3 dense1 = Dense(n_features = 4,
4               n_targets = 3,
5               scale = 1,
6               activation = Sigmoid(),
7               bias = True)

```

Here, **activation** specifies the activation function of the layer, in this case the sigmoid function. Setting **bias** to true enables the use of bias parameters in the layer. Like **QLayer**, **Dense** also implements methods like **\_\_call\_\_** for feed forward, and **grad()** for back-propagation. A DNN can be set up the following way:

```

1 from layers import Dense, Sigmoid
2
3 dense1 = Dense(n_features = 4,
4               n_targets = 3,
5               scale = 1,
6               activation = Sigmoid(),

```

```

7         bias = True)
8
9 dense2 = Dense(n_features = 3,
10               n_targets = 2,
11               scale = 1,
12               activation = Sigmoid(),
13               bias = True)
14
15 layers=[dense1, dense2]
16 network = Neuralnetwork(layers)

```

This opens up for construction of neural networks that consists of an arbitrary combination of **QLayer** and **Dense** layers, which can be simultaneously optimized using the methods described in [Section 6.3.5](#).

### 6.3.8 Hybrid Models

Due to the flexibility of the framework and compatibility of the different layers, it is possible to make hybrid models by combining **Dense** and **QLayer** layers:

```

1 dense = Dense(n_features = 64,
2               n_targets = 4,
3               scale = np.pi,
4               activation = Tanh(),
5               bias = True)
6
7 qlayer = QLayer(n_qubits = 4,
8                 n_features = 4,
9                 n_targets = 1,
10                encoder = Encoder(),
11                ansatz = Ansatz(blocks=["entangle", "ry"],
12                                reps=2),
13                sampler = Parity(),
14                backend = backend,
15                shots = 1000)
16 layers=[dense, qlayer]
17 network = Neuralnetwork(layers)

```

In this code, the initial **Dense** layer is used to produce 4 outputs from 64 features, which are subsequently fed to the following **QLayer** layer. It may be useful to construct such hybrid models for training on data sets with many features, since feeding 64 features directly to a QCN would require 64 qubits when using qubit encoding. This high amount of qubits is generally not feasible for near-term hardware. Note that the use of tanh activation and a scaling of  $\pi$  ensure that the output of the **Dense** layer is in  $[-\pi, \pi]$ , which we argue in [Section 6.5.4](#) is an appropriate scaling for QCNs.

## 6.4 Tools for Analysis

In this section, we present details surrounding the implementation of methods discussed in [Chapter 5](#).

### 6.4.1 Magnitude of Gradients

In order to investigate the vanishing gradient phenomenon discussed in [Section 4.5.2](#) for QNNs, QCNs and DNNs, we will calculate the average magnitude of the gradient for different models and architectures. Instead of calculating the gradient of some loss function, i.e. [Equation \(2.7\)](#), we will instead calculate the gradient of the model output itself, i.e.

$$\frac{\partial f(\mathbf{x}^{(i)}; \boldsymbol{\theta})}{\partial \theta_j}. \quad (6.2)$$

Using our framework for neural networks, [Equation \(6.2\)](#) is calculated for each data sample using the `NoCost()` loss function together with sample-wise backward propagation. Given a model `network`, this can be implemented in the following way:

```
1 network.cost = NoCost()
2 network.backward(samplewise=True)
```

If [Equation \(6.2\)](#) vanishes, so does [Equation \(2.7\)](#), for any loss function. The quantity [Equation \(6.2\)](#) also has the added benefit of being data agnostic, i.e., it is independent of labels  $y^{(i)}$ .

Given a QNN  $f_{QNN}$ , we wish to compute the magnitude of the gradient averaged over  $N$  samples  $\mathbf{x}^{(i)}$  and the parameters  $\theta_j$ . This allows us to capture the average behavior of the model's gradient. To produce a more significant result, this quantity will finally be averaged over  $T$  different realizations of the parameters  $\boldsymbol{\theta}^{(k)}$ . This quantity can be expressed as

$$\frac{1}{TNn_\theta} \sum_{k=1}^T \sum_{i=1}^N \sum_{j=1}^{n_\theta} \left| \frac{\partial f_{QNN}(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)})}{\partial \theta_j} \right|. \quad (6.3)$$

For QCNs and DNNs, we want to compute the same quantity as [Equation \(6.3\)](#), but averaged over the parameters within each layer, rather than all parameters of the model. This enables us to investigate how the average behavior of the gradient changes from layer to layer. This quantity can be formulated as

$$\frac{1}{TNn_\theta} \sum_{k=1}^T \sum_{i=1}^N \sum_{j=1}^{n_\theta^l} \left| \frac{\partial f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)})}{\partial \theta_j^{[l]}} \right|, \quad (6.4)$$

where  $f$  is a QCN or DNN model,  $n_\theta^l$  are the number of parameters in layer  $l$ , and  $\theta_j^{[l]}$  is the  $j$ 'th of these parameters.

We are also interested in the local gradients Equation (4.26) of QCNs, i.e.,  $\frac{\partial a_m^{(l)}}{\partial a_n^{(l-1)}}$ . We will average the magnitude of the local gradients within the same layer  $l$ , over  $N$  samples  $\mathbf{x}^{(i)}$ , and over  $T$  different realizations of the parameters. This quantity can be expressed as

$$\frac{1}{TNn_{\theta}m^{(l)}m^{(l-1)}} \sum_{k=1}^T \sum_{i=1}^N \sum_{m,n} \left| \frac{\partial a_m^{(l)}}{\partial a_n^{(l-1)}} \right|, \quad (6.5)$$

where  $m^{(l)}$  is the number of nodes in layer  $l$ , and the indices  $m$  and  $n$  iterates over nodes in layer  $l$  and  $l-1$ , respectively. Note that the dependence of  $\left| \frac{\partial a_m^{(l)}}{\partial a_n^{(l-1)}} \right|$  on the samples and parameter realizations has been suppressed for clarity.

### 6.4.2 Empirical Fisher Information

To investigate the loss landscape of the QCN model, we calculate the empirical Fisher information matrix (EFIM) Equation (5.3) and its eigenvalue spectrum using a class **FIM**. Given some **network** and data set **x**, it can be used in the following way:

```
1 from analysis import FIM
2
3 fim = FIM(network)
4 fim.fit(x)
```

Calling **fim.fit(x)** calculates the EFIM of **network** over the data **x**, and is implemented as

```
1 def fit(self, x):
2     n_samples = x.shape[0]
3
4     self.model.backward(x, samplewise=True)
5     gradient = self.model.weight_gradient_list
6
7     gradient_flattened = []
8     for grad in gradient:
9         gradient_flattened.append(grad.reshape(n_samples, -1))
10
11     gradient_flattened = np.concatenate(gradient_flattened,
12                                       axis=1)
13
14     self.fim = 1 / n_samples * gradient_flattened.T @
15     gradient_flattened
```

At line 4, the cost function of the network is set to **NoCost()**. This ensures that **backward** calculates the gradient of the model output and not any particular loss, which is required by the EFIM. In addition, specifying **samplewise=True** in the **backward()** method stops the gradient from being averaged over all the samples.

Rather, it is stored individually for each sample. The following for-loop unravels and concatenates all the gradients of the various layers into a single matrix with dimension  $(N, n_\theta)$ , which is the number of samples and parameters, respectively. The EFIM is calculated as a matrix product in line 13, normalized by the number of samples. This matrix is then stored in **self.fim**. After calculating the EFIM, its eigenvalue spectrum can be calculated as

```
1 eigenvalue_spectrum = fim.eigen(x)
```

This performs a simple eigenvalue decomposition of **self.fim** using numpy's **linalg.eigh** to extract the eigenvalues. Because the EFIM is often highly degenerate, some of the lower lying eigenvalues turn out negative, likely because of floating-point errors. To combat this, any eigenvalue lower than  $10^{-25}$  is set to this value.

### 6.4.3 Trajectory Length

Assume we have a **NeuralNetwork** object and a trajectory  $\mathbf{x}(t_i)$ , as described in [Section 5.2.1](#). Then, we can calculate the trajectory length [Equation \(5.5\)](#) of the resulting outputs of each layer using the class **TrajectoryLength**:

```
1 from analysis import TrajectoryLength
2
3 tl = TrajectoryLength(network)
4 traj_len, traj_proj = tl.fit(x)
```

This produces two python lists: **traj\_len** containing the trajectory length of the layer outputs, and **traj\_proj** containing the layer outputs themselves, projected down onto 2D for visualization. The projection is done using scikit-learn's PCA decomposition [\[28\]](#).

## 6.5 Numerical Experiments

For easy reading, we will present hyperparameters and configuration details in [Chapter 7](#) alongside the relevant results and discussion. However, for sake of tidiness, we will present and discuss configuration choices that appear frequently in the analysis in this section.

### 6.5.1 Initialization

For QNNs and QCNs, the parameters will be initialized uniformly as  $\theta_i \sim U(-\pi, \pi)$ . Note that Pauli rotations (used in the simple ansatz [Equation \(4.7\)](#)) have a periodicity of  $2\pi$ , i.e.  $R_j(x) = R_j(x + 2\pi)$ , so all possible rotation angles can be realized using this initialization. Thus, it is possible to reach the whole space of models defined by the architecture. Initializations with this characteristic is often used for machine learning models based on PQCs [\[17, 21\]](#).

For DNNs, we will use the popular Xavier initialization [42], which is the default initialization used by pyTorch [29] when using densely connected layers. The Xavier initialization is obtained by sampling the weights and biases  $\theta^l$  of layer  $l$  uniformly as

$$\theta^l \sim U\left(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right), \quad (6.6)$$

where  $n$  is the number nodes in the previous layer.

### 6.5.2 Pre-processing Data

For QNNs and QCNs, features will be min-max scaled such that  $x_i \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ . This is deliberately half the periode of Pauli rotations, which are used when performing qubit encoding (Section 4.2.1). If inputs were rather scaled to one period, i.e.  $x_i \in [-\pi, \pi]$ , extremal inputs would result in the same quantum state after encoding. Since mapping of different data point to the same quantum state can potentially inhibit learning, we opt for the original way of scaling. For DNNs, we apply standardization of the input, as explained in Section 2.5.1, by subtracting the mean and dividing by the standard deviation feature-wise.

To scale the data, we use the scalers included in scikit-learn [28]. They can be applied the following way:

```
1 from sklearn.preprocessing import StandardScaler, MinMaxScaler
2
3 standard = StandardScaler()
4 x_train_standard = standard.fit_transform(x_train)
5 x_test_standard = standard.transform(x_test)
6
7 minmax = MinMaxScaler(feature_range=(-np.pi/2, np.pi/2))
8 x_train_minmax = standard.fit_transform(x_train)
9 x_test_minmax = standard.transform(x_test)
```

Note that when using a training and test set **x\_train** and **x\_test**, like in the above code, the test set must be scaled after the training data. In practice, it should not be scaled together with the training data, as it is usually not available when training the model.

To perform feature reduction using PCA, as described in Section 2.5.2, we will again use scikit-learn. It can be implemented using:

```
1 from sklearn.decomposition import PCA
2
3 standard = StandardScaler()
4 x_train = standard.fit_transform(x_train)
5 x_test = standard.transform(x_test)
6
7 pca = PCA(n_components=4)
8 x_train = pca.fit_transform(x_train)
9 x_test = pca.fit_transform(x_test)
```



Note that we first perform a standardization of the data. This is to ensure that the amount of variance the features contribute is not affected their units and scale.

### 6.5.3 Optimization

We use Adam, as presented in [Section 2.2.2](#), to optimize all models in this thesis. We use default hyperparameters suggested by the authors, Kingma and Ba [\[27\]](#), together with a learning rate of  $lr = 0.1$ . This is the same learning rate as used by Abbas et al. [\[17\]](#) for training QNNs. Being a relatively high learning rate, it leads to quick optimization of the models. This is a useful property, as each training iteration can be quite time consuming due to the overhead associated with simulating quantum circuits on classical hardware.

### 6.5.4 Configuring QCNs and DNNs

Unless specified otherwise, the outputs of intermediate layers of QCNs will be scaled to the interval  $a_i^{(l)} \in [-\pi, \pi]$ . Using the same argument about the periodicity of Pauli rotations from [Section 6.5.1](#), this enables the outputs of one layer to potentially realize any rotation angle in the next layer. For the last layer, the output will be scaled to  $a_i^{(L)} \in [0, 1]$ . This is sufficient, given that the target  $y$  is scaled to within this interval.

For DNNs, we will be using tanh activations on all hidden layers. This is partly because it is a hugely popular choice, known to produce fast converging neural models [\[5\]](#). Further, tanh is also a bounded function, returning values on the interval  $[-1, 1]$ , unlike the also popular activation function, namely ReLU. This results in the outputs of intermediate layers to be bounded, making them easier to compare to QCNs.

## Part III

### Results & Discussion

# 7

## Results and Discussion

In this chapter, we will investigate and characterize the models discussed in the earlier chapters. We will also detail how the various models are configured, and how numerical methods are used for characterization. The results will be briefly commented on as they are presented, with a more in-depth discussion at the end of each section.

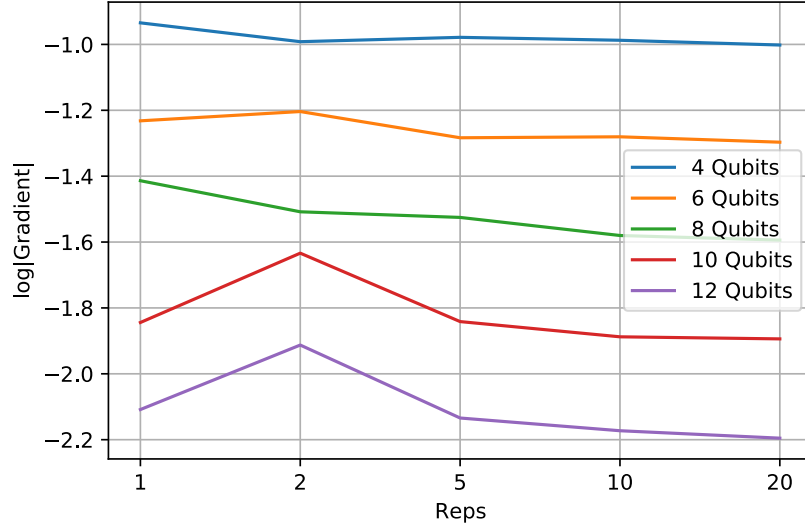
### 7.1 Vanishing Gradient Phenomenon

In this section, we investigate and compare the vanishing gradient phenomenon for quantum neural networks (QNNs) [Equation \(4.3\)](#), quantum circuit networks (QCNs) [Equation \(4.21\)](#) and dense neural networks (DNNs) [Equation \(2.13\)](#) by studying how the magnitudes of their gradients vary as a function of architecture. First, we study how the gradients of QNNs behave as the number of qubits and repetition of the ansatz increase. Then, the local gradients [Equation \(4.26\)](#) of QCNs are studied for different number of qubits, nodes and layers. Lastly, the vanishing of the gradient [Equation \(4.24\)](#) is investigated.

#### 7.1.1 Vanishing Gradient in QNNs

We start by investigating the magnitude of the gradient of QNNs for different number of qubits and repetitions of the ansatz. We use qubit encoding [Figure 4.2](#) with  $R_y$  rotations for feature encoding, and the simple ansatz [Equation \(4.7\)](#) for processing. This combination was chosen as it was found to yield the largest gradient. To derive an output, we calculate the expected parity [Equation \(4.8\)](#) using ideal simulation, as explained in [Section 6.1.4](#). We calculate the average magnitude of the

gradient using Equation (6.3), with  $T = 10$  different realizations of the parameters to increase the statistical significance of the result. To get a representative result for a large feature space, we sample features  $\mathcal{X} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}\}$  uniformly as  $\mathcal{X} \sim U(-\frac{\pi}{2}, \frac{\pi}{2})^{[N,p]}$ . We use  $N = 100$  samples, and the number of features  $p$  is set equal to the number of qubits for each QNN. The QNNs are also initialized in the standard way, i.e. sampling parameters as  $\theta_j \sim U(-\pi, \pi)$ . The resulting magnitude of the gradients for different number of qubits and ansatz repetitions can be seen in figure Figure 7.1.



**Figure 7.1:** Average magnitude of gradients for QNNs with different numbers of ansatz repetitions (Reps) and qubits. The QNNs utilize qubit encoding with  $R_y$  rotations, the simple ansatz for processing, and parity sampling to derive an output. The QNNs are fed  $N = 100$  points of uniformly sampled points, where the number of features is set equal to the number of qubits.

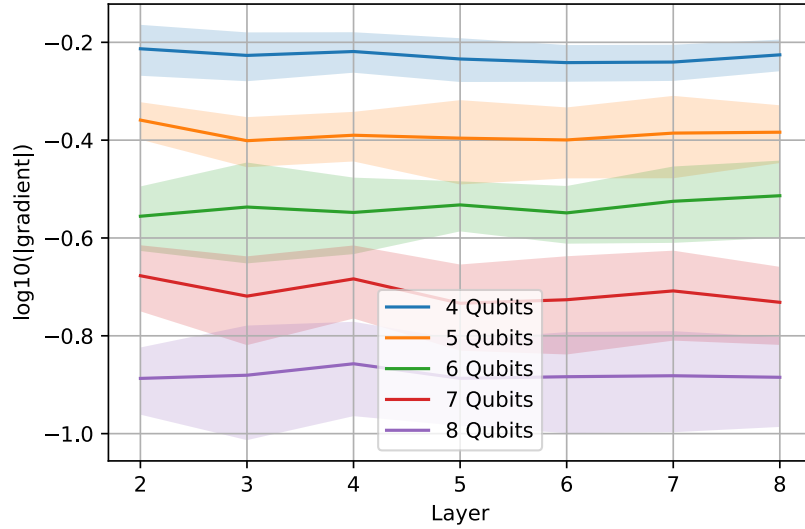
From the above figure, we see that our implementation of QNNs results in a gradient that vanishes in the exponential regime with respect to the number of qubits. Also, the vanishing is stronger for a higher number of repetitions of the ansatz. This behavior can be explained by the fact that QNNs are a special variant of parameterized quantum circuits (PQCs). By encoding random inputs and randomly initializing the parameters, the QNN essentially approaches a random circuit as they grow deeper. As shown by McClean et al. [20], and discussed in Section 4.5.2, randomly initialized PQCs tend to produce gradients closely centered around zero as the number of qubits are increased, which also applies for our implementation of QNNs.

### 7.1.2 Vanishing Local Gradient in QCNs

An interesting feature of QCNs is the ability to create larger models by introducing more circuits, instead of increasing the number of qubits and repetitions. As the gradients of QNNs tend to vanish with higher numbers of qubits, we want to investigate how the gradients of smaller parametric circuits behave when used as nodes a QCN architecture.

The QNNs used as nodes to construct QCNs here are set up and initialized in the manner explained in [Section 7.1.1](#), but always with two repetitions of the simple ansatz to keep each circuit shallow. In this section, all QCNs have eight layers with  $d$  number of nodes each and  $d$  number of qubits per node. Here,  $d$  range from four to eight. Further, the outputs of each hidden layer is scaled to the interval  $[-\pi, \pi]$  to make full use of the qubit encoding in the subsequent layer, as explained in [Section 6.5.4](#). We use uniformly distributed data with  $N = 100$ , as earlier.

In order to investigate the average behavior of the local gradients [Equation \(4.26\)](#) for each layer, we calculate their magnitude averaged over each layer, the samples and  $T = 10$  different realizations. This quantity is given by [Equation \(6.5\)](#), and is plotted in [Figure 7.2](#) for various layers of different QCNs. In addition, the standard deviation of this quantity is estimated over the different realizations.



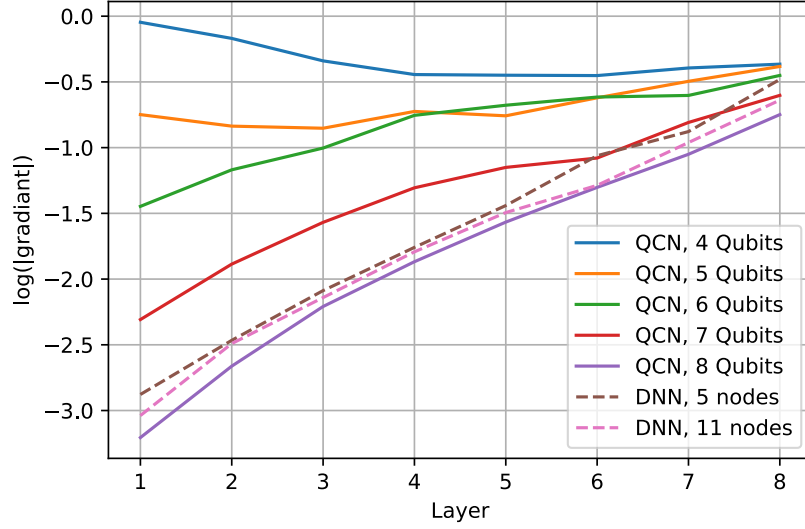
**Figure 7.2:** Average magnitude of local gradients [Equation \(4.26\)](#) calculated for each layer for various QCNs with eight layers. This quantity is calculated using [Equation \(6.5\)](#). The number of qubits per node is constant for each QCN, and the number of nodes per layer is set equal the number of qubits. The QCNs are fed  $N = 100$  points of uniformly sampled points, where the number of features is set equal the number of qubits. The standard deviation is calculated over ten different realizations of the model parameters.

In the above figure we see the average magnitude of the local gradients for different layers and number of qubits. The local gradients of the QCNs tend to vanish exponentially in the number of qubits, similar to the gradients of single-circuit QNNs seen in [Figure 7.1](#). However, the relative position along the depth of the QCN does not seem to affect the magnitude. Even though there is some variation of the magnitude from layer to layer, these variations are smaller than the standard deviation, and thus not statistically significant.

### 7.1.3 Vanishing Total Gradient in QCNs

In the previous section, we showed that the average magnitude of local gradients of any layer is independent of the relative position of that layer in the QCN model. However, the parameters are not updated using the local gradients directly. Rather, they are updated using the gradient [Equation \(4.24\)](#), which is calculated by combining the local gradients with the backpropagation [Equation \(4.23\)](#). We need to investigate how the magnitude of the total gradient [Equation \(4.24\)](#) behaves as a function of layers and qubits.

In this section, the gradient is calculated using the local gradients from the models in [Section 7.1.2](#). As previously mentioned, the magnitude of the total gradient is averaged over each layer, the samples and ten realizations of the parameters using [Equation \(6.4\)](#). This quantity is plotted in [Figure 7.3](#) for different layers and number of qubits. For comparison, it also shows the magnitude of the total gradient of two DNNs with the same number of layers and similar number of parameters as the four and eight-qubit QCN.



**Figure 7.3:** Average magnitude of the total gradient of the model output calculated for each layer for various 8 layer QCNs. This quantity is calculated using Equation (6.4). The number of qubits per node are constant for each QCN, and the number of nodes per layer is set equal to the number of qubits. For comparison, the same quantity is also calculated for two DNNs with 5 and 11 nodes, respectively. The DNNs have  $\tanh$  activations on all layers.

From Figure 7.3, we see that the gradient for a given layer in the DNN tends to vanish exponentially fast for the more initial layers. This is a manifestation of the vanishing gradient phenomenon for DNNs caused by saturation of the activation functions, as explained in Section 2.3.4.

As with the DNN, the QCNs also exhibit a vanishing gradient with increasing number of layers, with a strong dependence on the number of qubits in each node. As we see in Figure 7.2, the gradient vanishes faster for higher number of qubits in each node. This phenomenon can be related to the magnitude of the local gradients: As the error  $\delta_j^l$  of layer  $l$  is propagated backwards using Equation (4.23), it accumulates the local gradients  $\frac{\partial \mathbf{a}_k^{l+1}}{\partial \mathbf{a}_j^l}$  as factors. Accumulating small factors will cause  $\delta_j^l$  to decrease faster, exponentially so for each layer. This is the case when the number of qubits per circuit is high. However, if these factors are large, the  $\delta_j^l$  tend to decrease more slowly, and hence also the gradient. This is the case for architectures with few qubits per node, as discussed in Section 7.1.2.

#### 7.1.4 Discussion

The results of Section 7.1.1 show that increasing the model size of QNNs by adding more qubits and repetitions of the ansatz results in an exponential decay of their gradients. As explained in Section 4.5.2, this means that exponentially many shots

are required in order to obtain a good signal-to-noise when estimating the gradient. If the gradient is too noisy, optimization using gradient descent and similar methods may essentially result in a random walk in parameter space that fails to converge [21]. This becomes even more problematic in the presence of noise introduced by real hardware, as discussed in Section 7.4.2. Ultimately, this indicates that the training of QNNs can become intractable as they are increased in size to solve harder learning problems.

In Section 7.1.2, we see that the local gradients of QCNs also vanish exponentially fast in the number of qubits, but are independent of the overall number of layers. In other words, the local gradients of any layer are unaffected by outputs produced by the previous layer. This suggests that QCNs can be scaled up by making them deeper, without affecting the magnitude of the local gradients. Consequently, a constant number of shots can be used for each node during estimation to obtain a certain signal-to-noise ratio, making their estimations tractable on a quantum computer.

Even though the magnitude of local gradients of QCNs tends to stay constant in the number of layers, we see in Section 7.1.3 that backpropagation still induces an exponentially vanishing gradient for sufficiently many qubits. This is due to the accumulation of small factors when the local gradients are combined using backpropagation. This behavior is similar to that of DNNs, with the gradient vanishing faster for initial layers. However, the vanishing was not as severe for a conservative number of qubits. For 4 qubits, the gradient actually tended to increase. For 8 qubits, and presumably above, the gradient vanished faster for QCNs than for similarly sized DNNs.

An interesting observation is that the vanishment caused by back propagation happens in a purely classical part of the optimization, with the local gradients stored as floating-point numbers. This means that even though the total gradient tends to decrease exponentially with the number of layers, it does not introduce an exponential overhead on the quantum computer by requiring more shots. However, the overhead increases exponentially for single-QNN models as discussed in Section 4.5.2. In other words, QCNs' use of several smaller circuits, rather than one large, moves the estimation of vanishing quantities (e.g. the gradient) from quantum expectation values to classical computation.

## 7.2 Investigating the Loss Landscape

We explore the geometry of the loss landscape of randomly initialized QNNs, QCNs and DNNs, and quantify their degree of distortion and flatness by studying the eigenvalue spectrum of the empirical fisher information matrix (EFIM) Equation (5.3). Looking at Equation (5.3), we see that the EFIM, unlike the Hessian, is independent of the targets  $y^{(i)}$ , making the analysis independent of the specific problem.

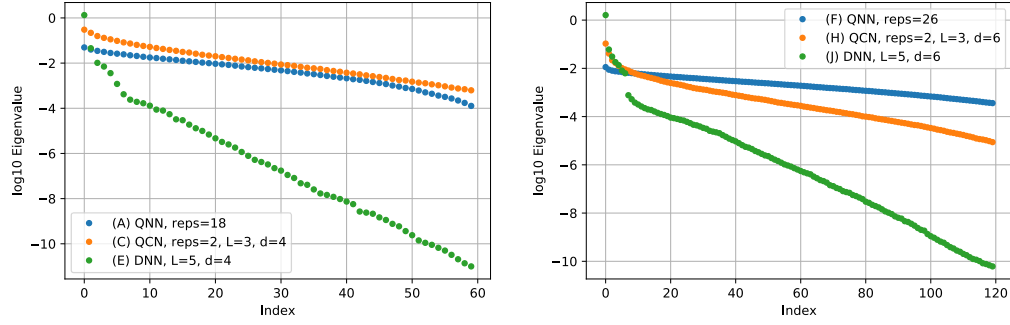


In this section, the QNNs utilize RZZ encoding (see Figure 4.3), while the QCNs utilize qubit encoding (see Figure 4.2) with  $R_y$  rotations. Both types of models use a variable number of repetitions of the simple ansatz Equation (4.7) for processing. The EFIM is calculated using features sampled uniformly as  $\mathcal{X} \sim U(-\frac{\pi}{2}, \frac{\pi}{2})^{[N,p]}$ , with  $N = 200$  samples and either  $p = 4$  or  $p = 6$  inputs, depending on the model. For each model, the EFIM is calculated ten times for different realizations of the model parameters. The resulting spectrum is then averaged over different realizations to produce a more statistically significant result. For a complete description of the models analyzed in this section, see Table 7.1.

**Table 7.1:** Description of the architecture of the models analyzed in this section. Reps is the number of ansatz repetitions, and  $n_\theta$  is the number of model parameters. NA denotes undefined quantities.

Model	Type	Qubits	Reps	Layers	Nodes	Encoder	$n_\theta$
A	QNN	4	18	NA	4	RZZ Enc	72
B	QCN	4	3	2	4	Qubit Enc	60
C	QCN	4	2	3	4	Qubit Enc	72
D	QCN	4	1	4	4	Qubit Enc	68
E	DNN	NA	NA	3	6	NA	79
F	QNN	6	26	NA	6	RZZ Enc	156
G	QCN	6	4	2	6	Qubit Enc	168
H	QCN	6	2	3	6	Qubit Enc	156
I	QCN	6	1	4	6	Qubit Enc	150
J	DNN	NA	NA	3	9	NA	163

Figure 7.4 compares the EFIM spectra of QNNs, QCNs and DNNs. Their architectures are chosen so that the models have approximately equal numbers of parameters. This is to ensure a fair comparison.

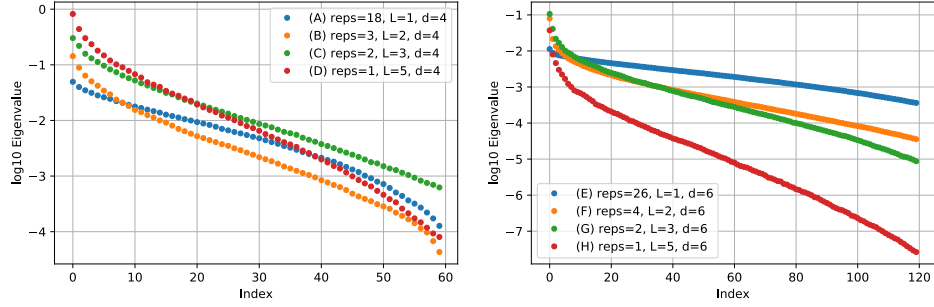


(a) QNN and QCN with four qubits in each circuit. (b) QNN and QCN with six qubits in each circuit.

**Figure 7.4:** Comparison of EFIM spectrum between QNNs, QCNs and DNNs, calculated for uniformly sampled data. The spectrum is truncated at 60 and 120 eigenvalues for easier comparison. For details about the architectures, see Table 7.1. reps is the number of ansatz repetitions, L is the number of layers, and d is the number of nodes.

Looking at the spectra of the DNNs in Figure 7.4, we see the characteristic result of a singular large eigenvalue, with the rest sitting close to zero. This is reminiscent of the typical Hessian spectrum of DNNs, shown in Figure 5.1. This indicates that DNN models exhibit a loss landscape that is very flat in all but one direction where it is extremely distorted. We also see that the spectra of our implementation of QNNs are much more uniformly distributed compared to the DNN models. This results in a loss landscape that is significantly distorted in most directions, rather than just one.

Moving over to the QCNs, we see from Figure 7.4 that the spectra of the three layer QCNs exhibit much the same uniformity as the QNNs for four qubits, but more skewed for six qubits. A more thorough comparison between different QCNs is shown Figure 7.5. In this figure, we vary the number of layers of the QCNs and the number of repetitions (i.e. the number of times the ansatz is repeated for each node), while keeping the total number of parameters roughly constant.



(a) Comparison of the EFIM spectra for different QNNs and QCNs with four qubits. (b) Comparison of the EFIM spectra for different QNNs and QCNs with six qubits.

**Figure 7.5:** Comparison of the EFIM eigenvalue spectrum between different QNNs and QCNs, calculated for uniformly sampled data. The spectrum is truncated at 60 and 120 eigenvalues for easier comparison. For details about the architectures, see Table 7.1. reps is the number of ansatz repetitions, L is the number of layers, and d is the number of nodes.

Figure 7.5a shows that, for four qubits, the spectra of the different QCNs exhibit roughly the same uniformity as the QNN, with the eigenvalues staying within roughly an order of magnitude of each other. Going up to six qubits, Figure 7.5b shows that the spectrum tends to concentrate more around zero for increasing number of layers. This is likely related to the vanishing of the gradient induced by backpropagation. For four qubits, this is not as big of a problem since the local gradients are relatively big and hence also the gradient. However, for six qubits, the local gradients tend to vanish. This results in the gradient vanishing faster when increasing the number of layers, which in turn results in a flatter landscape.

### 7.2.1 Discussion

The highly uneven EFIM spectrum of DNNs indicates a loss landscape that is strongly distorted in one direction and mostly flat otherwise. This result is consistent with the findings of Karakida et al. [8] and Abbas et al. [17]. The former authors point out that strong distortions in some directions indicate that the model outputs are highly sensitive to changes in parameter space in exactly these directions, and likewise not sensitive to changes in the others. This tend to slow training when using gradient descent and similar methods, as too high learning rate leads to overstepping in the distorted directions, while a low learning rate changes the model insignificantly in the flat directions.

For the QNNs, we found the EFIM spectrum to be much more uniform than that of a comparable DNN. Abbas et al. [17] came to the same conclusion for their QNN models, and argued that this uniformity of the spectrum meant that the landscape was more well-condition for optimization, and thus should train faster.

They strengthened this hypothesis by showing experimentally that their QNNs reduced loss faster than DNNs for equal numbers of epochs.

We found that QCNs with four qubits exhibit similar uniformity of the EFIM spectrum as QNNs. For six qubits, the spectrum became increasingly more skewed, with a worsening effect for more layers. However, they still showed several order of magnitude larger eigenvalues than DNNs, suggesting that small-scale QCNs should train comparably faster than DNNs.

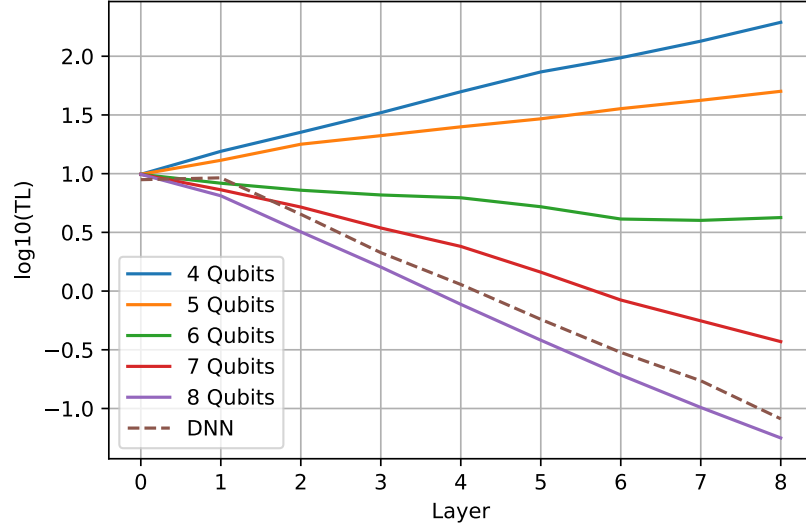
## 7.3 Expressivity

We investigate the expressivity of QCNs and DNNs using the trajectory length method of Raghu et al. [6], as described in [Section 5.2.1](#). The trajectory length will first be studied for randomly initialized QCNs with a varying number of qubits in each node. Then, for some selected QCNs, the trajectory length will be investigated as the models are gradually fitted on two dimensional mixed Gaussian data. The results in both cases will be compared to similar DNNs, with approximately the same number of parameters for fair comparison. We use an input trajectory  $\mathbf{x}(t_i) \in \mathbb{R}^2$  in the shape of a circle, with radius  $\frac{\pi}{2}$  and centered around 0, divided up into 1000 equally spaced points. The input trajectory and two dimensional mixed Gaussian data will be pre-processed appropriately, depending on the model, as described in [Section 6.5.2](#).

### 7.3.1 Untrained Models

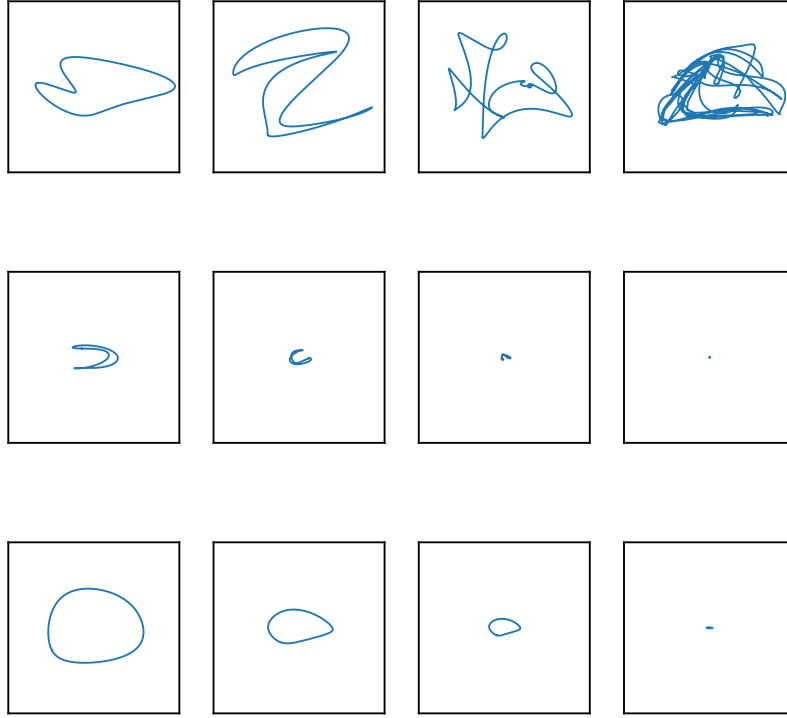
In this section, we investigate the same QCN and DNN architectures as in [Section 7.1.2](#). To summarize, all models have eight layers. The QCNs vary between four and eight nodes and qubits for each layer. They use qubit encoding for encoding features, and two repetitions of the simple ansatz. The number of nodes in the DNN is chosen so that it has approximately the same number of parameters as the biggest QCN. The models are initialized randomly as described in [Section 6.5.1](#), like earlier.

[Figure 7.6](#) shows how the trajectory length varies as a function of layer, when the different models are fed the circle trajectory  $\mathbf{x}(t_i)$  defined earlier. Since the trajectory only has two features, and the circuits of the QCNs utilize qubit encoding, we make use of latent qubits [Figure 4.4](#) to extend the circuits in the initial layer to the correct amount of qubits.



**Figure 7.6:** Logarithmic trajectory length (TL) as a circular trajectory  $\mathbf{x}(t_i)$  is propagated through QCNs with different number of qubits. The QCNs are defined as in Section 7.1.2. For comparison, the TL of a DNN with 11 nodes is also shown.

Figure 7.7, accompanying Figure 7.6, shows the trajectories of selected models and layers, projected onto 2D. The rows correspond to the QCN with four qubits, the QCN with eight qubits, and the DNN, from top to bottom. The columns correspond to the trajectory resulting from the first layer, second layer, third layer and last layer, left to right.

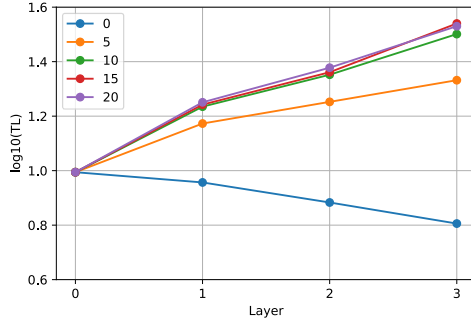


**Figure 7.7:** Trajectory length projected onto two dimensions for selected models from [Figure 7.6](#). The rows show the trajectory length of the four qubits QCN, eight qubit QCN and DNN, from top to bottom. The columns show the trajectory of the first layer, the second layer and last layer, from left to right.

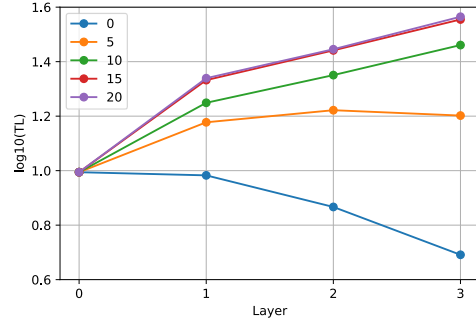
From [Figure 7.6](#), we see that the trajectory length of QCNs with four and five qubits tend to grow exponentially with the number of layers, meaning they are in the exponential growth regime. This growth is however diminishing as the number of qubits increase, and switches over to an exponential decay for 6 qubits and above. For 8 qubits, the decay is similar to that of the DNN with similarly many parameters. Comparing the results to [Figure 7.7](#), we see how the increasing and decreasing trajectory length manifests themselves. Seen from the top row, the trajectory produced by the four qubit QCN tends to become increasingly distorted and complex. This is similar to the behaviour of neural networks seen in [Figure 5.2](#), produced by Raghu et al. [6], and shows that also QCNs can compute functions exponentially complex in the number of layers. In contrast, the trajectory of the eight qubit QCN and DNN (seen in the next two rows) can be seen to gradually concentrate for each layer, resulting in a function that is very little sensitive to the input.

### 7.3.2 Trained Models

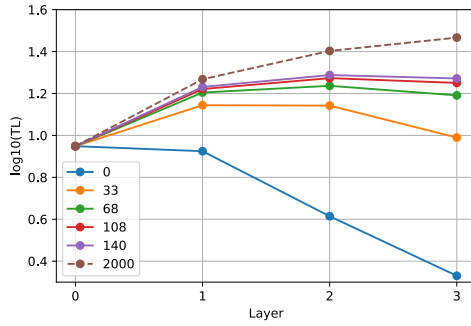
Raghu et al. [6] showed that networks that initially don't lay in the exponential growth regime can be pushed there via training. In this subsection, we will train different models by incrementally fitting them to the two dimensional mixed Gaussian data (see [Appendix A.1](#)). The trajectory length will be recalculated for each layer after each increment. The models being investigated here are QCNs with six and seven qubits, respectively. These will be compared to DNNs with approximately the same number of parameters. The models have three hidden layers, with a single node in the output layer. For more information about the models, see [Table 7.2](#). All the models are trained using Adam optimizer (see [Section 2.2.2](#)) with the standard hyperparameters and a learning rate of 0.1. The QCNs are trained for a total of 20 epochs in increments of 5. In order to produce a fair comparison, the DNNs will not be trained for the same increments of epochs. Rather, they will be trained until they achieve approximately the same MSE on the training set as the QCNs, for each increment. In this way, we get to compare the expressivity of QCNs and DNNs that fit the data to an equal degree. [Figure 7.8](#) shows how the trajectory length changes as the different models presented here are incrementally trained, for each layer.



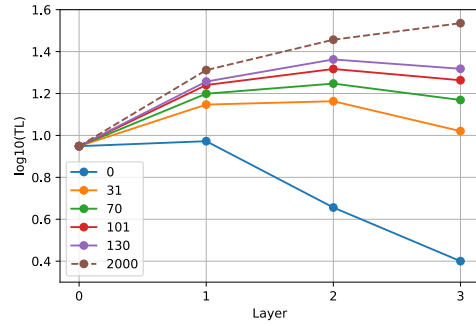
(a) QCN, 6 qubits. 228 parameters.



(b) QCN, 7 qubits. 308 parameters.



(c) DNN, 9 nodes. 217 parameters.



(d) DNN, 11 nodes. 309 parameters.

**Figure 7.8:** Logarithmic trajectory length (TL) as a circular trajectory  $\mathbf{x}(t_i)$  is propagated through QCNs and DNNs defined in Table 7.2. The models are gradually fitted for different number of epochs, shown in the legend, and the TL recalculated.

From Figure 7.8, we see that training the QCNs and DNNs progressively increases the trajectory length of the models. After only five epochs, the trajectory length of the six qubit QCN enters the exponential growth regime. This demonstrates that randomly initialized QCNs can be made to produce complex functions through training, even though their outputs initially tend to concentrate around a mean, as shown in Figure 7.6. The seven-qubit QCN is brought into the exponential growth regime after ten epochs, twice the amount required for six qubits.

Moving over to the corresponding DNNs, we see that they fail to enter the same exponential growth regime as the QCNs, even when trained until they fit the data to the same degree. This indicates that QCNs, in this context, may be more expressive than DNNs for the same number of parameters.

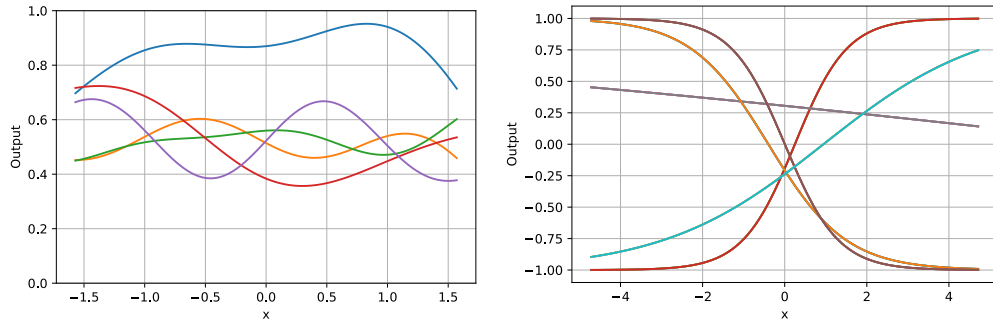


**Table 7.2:** Hyperparameters of the models trained on the 2D mixed Gaussian data in Figure 7.8.  $n_\theta$  is the number of model parameters. The DNNs have tanh activation on all layers, including the output layer.

Model	Type	Qubits	Hidden Layers	Nodes	$n_\theta$
A	QCN	6	3	6	228
B	QCN	7	3	7	308
C	DNN	NA	3	9	217
D	DNN	NA	3	11	309

### 7.3.3 Single Node Expressivity

To elucidate the discrepancy of expressivity between QCNs and DNNs, we investigate the functional form of node outputs for QCNs and DNNs. To do this, we use a four-qubit QCN node with qubit encoding and two repetitions of the simple ansatz. We prepare data  $\mathbf{x} = (x, x, x, x)$ , and encode it in the usual way. For comparison, we also feed the samples  $\mathbf{x}$  to a single DNN node with tanh activation. In Figure 7.9, we plot the output the different nodes for different parameter realizations and values of  $x$ .



(a) QCN node with four qubits. It utilizes qubit encoding Figure 4.2 with  $R_y$  rotations, and two repetitions of the simple ansatz Equation (4.7).

(b) DNN node with tanh activation.

**Figure 7.9:** Comparison between the functional form of the outputs of a QCN node and a DNN node. The different functions stem from different parameters realizations in the nodes.

Seen from Figure 7.9a, the QCN node produces a more flexible output, reflecting the polynomial representation of the data in the prepared state Equation (4.5). Figure 7.9b shows that the output of the DNN node is constrained to the functional form of the tanh activation function.

### 7.3.4 Discussion

For higher number of qubits, we saw from [Figure 7.7](#) that the trajectory of untrained QCNs tend to concentrate progressively for each layer they pass through. This is a manifestation of the phenomenon where the outputs of randomly initialized PQC tend to concentrate around their mean, as discussed in [Section 4.5.2](#). Since each node of a QCN is a PQC, they are also subject to this behaviour. In fact, as the inputs are sequentially transformed by multiple layers, this concentration of the outputs is applied multiple times. As seen in [Figure 7.6](#), this causes an exponential decrease of the trajectory length.

From [Figure 7.8](#) we see that training brought the QCNs into the exponential growth regime, increasing the sensitivity of the node outputs with respect to the inputs. In other words, optimizing the QCNs updates the parameters in a way that brings structure to each of the circuits, moving them away from being random circuits. This causes the outputs to no longer concentrate around the mean, which in turn lets the model compute more complex functions.

Comparing [Figure 7.8a](#) and [Figure 7.8b](#), we see that the seven-qubit QCN required twice the number of epochs to enter the exponential growth regime compared to the six-qubit QCN. As discussed in [Section 7.1](#), increasing the number of qubits makes the magnitude of the gradient exponentially smaller. Thus, a larger number of epochs may be required to significantly change the parameters for circuits with many qubits. This will likely result in a large overhead for QCNs with many qubits, requiring a significant amount of epochs in order to train them to be expressive.

Looking at [Figure 7.8c](#) and [Figure 7.8d](#), we see that the similarly sized DNNs fail to enter the exponential growth regime after an amount of training equivalent to the QCNs. The DNNs approached exponential growth first after training for more than two order of magnitude more epochs than the QCNs. This suggests that QCNs can be trained to be more expressive than DNNs of similar number of parameters. How can we explain this increased expressivity? As seen from [Figure 7.9a](#), QCN nodes are able to produce flexible functions, likely resulting from the feature interactions computed by qubit encoding [Equation \(4.5\)](#). Conversely, the functions produced by DNN nodes was very constricted to the form of the activation function, as seen from [Figure 7.9b](#). In this sense, it seems likely that QCNs can achieve greater expressivity by essentially learning unique activation functions for each node.

## 7.4 Training Models on Mixed Gaussian Data

In this section, we will study the ability of various models to fit one, two and three dimensional mixed Gaussian data. For more details on the data, see [Appendix A.1](#). We will train QNNs, QCN and DNNs with varying complexity and use MSE on the training data to evaluate how good the fit is. This will be done first in the ideal case, with exact calculation of outputs. Then, we will repeat the training

using the simulated noise model of the IBM Santiago quantum computer [41]. The hyperparameters, such as number of layers, nodes and qubits, are chosen by trial and error such that the resulting QCNs are relatively small in number of parameters, but still fit the data sufficiently. The QNNs and DNNs are then chosen such that they have similar number of parameters as the QCNs. For a detailed description of the models trained in this section, see Table 7.3.

**Table 7.3:** Hyperparameters of the different models fitted to the one, two, and three dimensional mixed Gaussian data (1D, 2D, 3D). Qubits refer to the number of qubits used in each circuit, Reps refer to the number of repetitions of the simple ansatz Equation (4.7), and  $n_\theta$  refers to the number of parameters in the model. NA denotes undefined quantities.

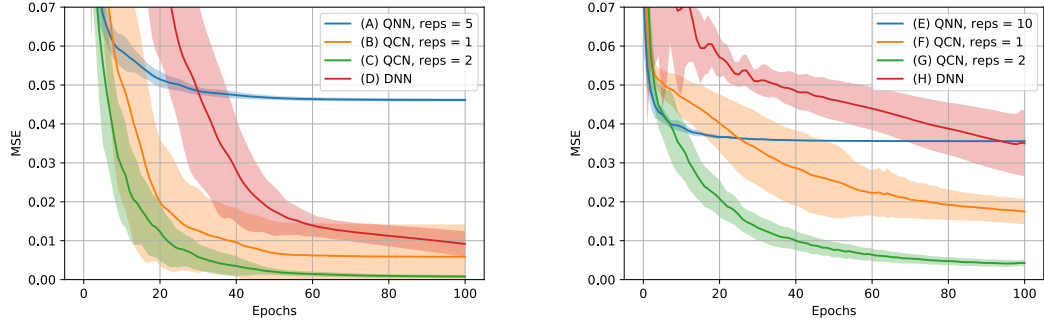
Model	Type	Data	Qubits	Reps	Layers	Nodes	$n_\theta$
A	QNN	1D	4	5	NA	NA	20
B	QCN	1D	4	1	2	4	20
C	QCN	1D	4	2	2	4	40
D	DNN	1D	NA	NA	2	13	40
E	QNN	2D	4	10	NA	NA	40
F	QCN	2D	4	1	3	4	40
G	QCN	2D	4	2	3	4	80
H	DNN	2D	NA	NA	3	7	85
I	QNN	3D	5	11	NA	NA	55
J	QCN	3D	5	1	3	5	55
K	QCN	3D	5	2	3	5	110
L	DNN	3D	NA	NA	3	8	113

For the QNNs trained in this section, we will utilize RZZ encoding (Figure 4.3) together with latent qubits (Figure 4.4) in an effort to increase the flexibility of the models. From Section 4.2.2, we know that the circuit depth of this encoding scales as  $\mathcal{O}(p^2)$ , where  $p$  is the number of features. To get a constant depth across the one, two and three dimensional Gaussian data, we will repeat features until all the data sets have the same number of features: For the one dimensional data, we repeat the one feature three times:  $(x_1) \rightarrow (x_1, x_1, x_1)$ . For the two dimensional data, we repeat the first feature once:  $(x_1, x_2) \rightarrow (x_1, x_2, x_1)$ . The three dimensional data is left unchanged, since it already has three features. In this way, RZZ encoding prepares similarly complicated encoding for all data sets.

#### 7.4.1 Ideal Simulation

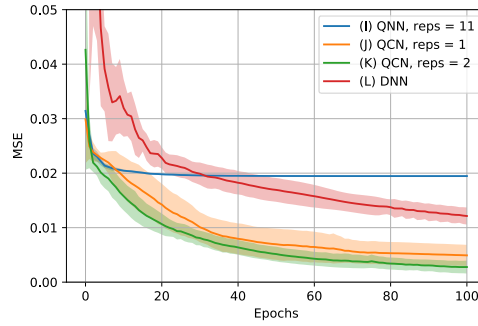
Figure 7.10 shows the MSE during training of the models defined in Table 7.3 on the one, two and three dimensional mixed Gaussian data. All models are trained for 100 epochs, using Adam optimizer and ideal simulation. In order to produce a more significant result, each model is randomly initialized ten times and trained

separately. The resulting MSE for each model is then averaged over the ten runs and plotted together with one standard deviation. In this way, we get to see the average model behaviour during training and how it varies between different runs. [Table 7.4](#) shows the final MSE on the training set after 100 epochs for the various models. In addition, the final MSE after  $10^4$  epochs is included for the DNNs.



(a) MSE of models trained on 1D mixed Gaussian data.

(b) MSE of models trained on 2D mixed Gaussian data.



(c) MSE of models trained on 3D mixed Gaussian data.

**Figure 7.10:** MSE during training of the models defined in [Table 7.3](#), trained on the mixed Gaussian data (see [Appendix A.1](#) for details on the data). The QNNs and QCNs are trained using ideal simulations.

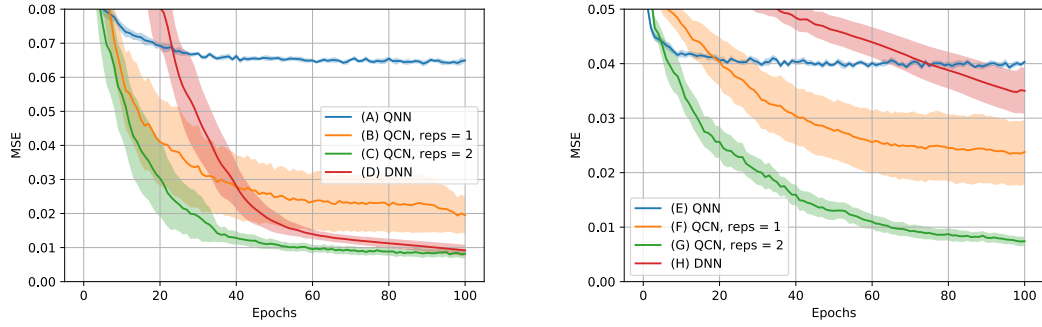
**Table 7.4:** Final MSE on the training set after training of the models detailed in Table 7.3 on the one, two and three dimensional mixed Gaussian data (1D, 2D, 3D). Ideal simulation was used. These results accompany the training shown in Figure 7.10. The lowest MSE after 100 epochs is highlighted. NA denote undefined quantities.

Model	Type	Data	MSE, $10^2$ Epochs	MSE, $10^4$ Epochs
A	QNN	1D	$4.61 \times 10^{-2}$	NA
B	QCN	1D	$5.88 \times 10^{-3}$	NA
C	QCN	1D	<b><math>7.99 \times 10^{-4}</math></b>	NA
D	DNN	1D	$9.17 \times 10^{-3}$	$2.23 \times 10^{-4}$
E	QNN	2D	$3.56 \times 10^{-2}$	NA
F	QCN	2D	$1.75 \times 10^{-2}$	NA
G	QCN	2D	<b><math>4.27 \times 10^{-3}</math></b>	NA
H	DNN	2D	$3.50 \times 10^{-2}$	$4.26 \times 10^{-3}$
I	QNN	3D	$1.95 \times 10^{-2}$	NA
J	QCN	3D	$4.91 \times 10^{-3}$	NA
K	QCN	3D	<b><math>2.77 \times 10^{-3}</math></b>	NA
L	DNN	3D	$1.21 \times 10^{-2}$	$1.79 \times 10^{-3}$

From Figure 7.10, we see that the QCNs minimize the MSE quicker than both the QNNs and DNNs on the mixed Gaussian data, for any number of dimensions. The QCNs with the two ansatz repetitions also trained faster than those with just one. Further, we see that QNNs perform overall worst among the models. After initially fast optimization, the models quickly flatten out at a relatively high MSE, struggling to obtain a good fit. From Table 7.3, we see that the DNNs obtains the lowest MSE among all models when trained until saturation, after  $10^4$  epochs.

### 7.4.2 Noisy Simulation

In this section, we investigate how QNNs and QCNs behave when trained on the Gaussian data using simulated quantum hardware. We do this by repeating the training of the models of last section using a simulation of the Santiago quantum computer [41]. We exclude, however, the training on the 3D mixed Gaussian due to the huge computational burden of simulating real hardware. The resulting MSE during training can be seen in Figure 7.11. Each model was trained for 100 epochs, using 1024 shots to estimate the output of each circuit. Table 7.5 shows the final MSE after 100 epochs for the QNNs and QCNs, trained on simulated hardware. As earlier, the resulting MSE after 100 and  $10^4$  epochs is also included for the DNNs.



(a) MSE of models trained on 1D mixed Gaussian data.

(b) MSE of models trained on 2D mixed Gaussian data.

**Figure 7.11:** MSE during training of the models defined in Table 7.3, trained on the mixed Gaussian data, excluding the 3D data (see Appendix A.1 for details on the data). The QNNs and QCNs are trained using noisy simulation of the Santiago quantum computer [41], using the methods of Section 7.4.2.

**Table 7.5:** Final MSE on training set after training the models detailed in Table 7.3 on the one and two dimensional mixed Gaussian data (1D, 2D). Noisy simulation of the Santiago quantum computer [41] was used. See Section 7.4.2 for details. These results accompany the training shown in Figure 7.11. NA denotes undefined quantities.

Model	Type	Data	MSE, $10^2$ Epochs	MSE, $10^4$ Epochs
A	QNN	1D	$6.49 \times 10^{-2}$	NA
B	QCN	1D	$1.95 \times 10^{-2}$	NA
C	QCN	1D	$8.18 \times 10^{-3}$	NA
D	DNN	1D	$9.17 \times 10^{-3}$	$2.23 \times 10^{-4}$
E	QNN	2D	$4.03 \times 10^{-2}$	NA
F	QCN	2D	$2.38 \times 10^{-2}$	NA
G	QCN	2D	$7.42 \times 10^{-3}$	NA
H	DNN	2D	$3.50 \times 10^{-2}$	$4.26 \times 10^{-3}$

Comparing Figure 7.11 and Figure 7.10, we see that the QNNs and QCNs trained on noisy hardware train slower and obtain a overall higher final MSE after 100 epochs, compared to the ideal simulation.

### 7.4.3 Discussion

In Section 7.1 and Section 7.2, it was suggested that QCNs of few qubits and layers would train faster than DNNs with similar number of parameters, due to their relatively larger gradient and more uniform EFIM spectrum. As we see from Figure 7.10, this is indeed the case when training on the mixed Gaussian data

using exact and noiseless simulation. Further, we see that the QCNs with two ansatz repetitions show faster training and less variance than those with only one repetition. This shows that QCNs can be made more flexible by adding additional complexity to each node in the form of additional repetitions.

In [Section 5.2](#), it was suggested that QCNs are more expressive and flexible than DNNs with similar number of parameters. Looking at [Table 7.4](#), we see that the final MSE obtained by the QCNs with two repetitions is within the same order of magnitude as the final MSE obtained by the DNNs. This is despite the fact that the QCNs were only trained for 100 epochs, while the DNNs were allowed to train for  $10^4$  epochs (basically until saturation, i.e. the lowest possible MSE for that particular model). In light of this, it is not unlikely that the QCNs would eventually reach a lower MSE than the DNNs if given the opportunity to train for more epochs. This suggests that it is possible for QCNs to fit complicated data more easily than DNNs, and hence are more flexible. However, this is speculative, as we are unable to train the QCNs much further due to limited computational resources.

We see in [Figure 7.10](#) that the QNNs struggle to obtain a good fit on the mixed Gaussian data, even though they have a similar number of parameters as the QCNs with one ansatz repetition. This shows that RZZ encoding combined with latent qubits and multiple repetitions of the simple ansatz produce models that are not able to learn the structure of the mixed Gaussian data. As explained in [Section 4.6.1](#), our QNNs perform a unitary (and thus linear) transformation of the input, except at the stage of encoding and measurement. This results in a perhaps too constrained model, which might explain the QNNs' inability to fit the data. QCNs, on the other hand, incorporate multiple nonlinearities for each layer as a result of measurements done to estimate the output of each node, as explained in [Section 4.6](#). As with DNNs, this is likely the key to their greater flexibility, as it enables them to compute a larger family of functions.

Moving over to the training using the simulated noisy hardware, we see from [Figure 7.11](#) that the QNN and QCN models performed overall worse than in the ideal case. This is not surprising, since the simulation of real hardware and the low number of 1024 shots cause a significant amount of noise to be added to the outputs of the QNNs and QCNs. The slowdown of the training is likely due to noise being added to the calculation of the gradient. This causes it to misalign with the direction of steepest descent, as discussed in [Section 4.5.2](#), and which results in the optimization slowing down. However, the QCNs with two ansatz repetitions still obtained a lower MSE than the DNNs after 100 epochs, even in the presence of noise. This shows that QCNs have the ability to outperform DNNs on some data sets, even on noisy quantum hardware with few shots.

## 7.5 Real-World Data

In this section, we compare the performance of QCNs and DNNs by performing regression on the Boston Housing data [43] and classification on the Breast Cancer Wisconsin data [44]. To reduce the computational burden of training the models, both data sets are feature reduced to four features (down from 13 and 30 features, respectively) using principal component analysis (PCA). In addition, the targets  $y$  of the Boston Housing data is scaled such that  $y \in [0, 1]$ . See Appendix A for more information about the data sets. We pick independent training and test sets with  $N = 100$  samples from each of the two data sets. To uncover how well the models generalize to unseen data, we calculate the MSE on the test set during training and find the minimum value. Doing this, we find the point during training where the models generalized the best, as discussed in Section 2.4. For the models trained on the Breast Cancer data, we also find the final test accuracy Equation (2.5) after training.

The QCNs models are chosen to have four qubits in each circuit in order to match the number of features used. Using trial and error, the number of hidden layers is set to one. As usual, qubit encoding (see Figure 4.2) with  $R_y$  and two repetitions of the simple ansatz (see Equation (4.7)) is used. The DNNs are chosen such that the number of parameters approximately match that of the QCNs. The hyperparameters of the various models are listed in Table 7.6.

**Table 7.6:** Hyperparameters of the various models fitted to the feature reduced Boston Housing data (BHD) and Breast Cancer data (BCD). Reps is the number of ansatz repetitions, and  $n_\theta$  is the number of model parameters. NA denotes undefined quantities.

Model	Type	Data	Qubits	Reps	Hidden Layers	Nodes	$n_\theta$
A	QCN	BHD	4	2	1	4	40
B	DNN	BHD	NA	NA	1	6	37
C	QCN	BCD	4	2	1	4	40
D	DNN	BCD	NA	NA	1	6	37

The models defined in Table 7.6 are trained for 100 epochs. As usual, this is repeated ten times for each model for different parameter realizations. The QCNs are trained using both ideal simulation and noisy simulation. For the latter, a simulation of the Santiago quantum computer [41] with 1024 shots is used. In Table 7.7, we see the average minimum training and test MSE for each model obtained during training. In addition, the final average training and test accuracy is included for the models trained on the Breast Cancer data.



**Table 7.7:** Minimum training and test MSE obtained during training for the models defined in Table 7.6. The models are trained on the feature reduced Boston Housing data (BHD) and Breast Cancer data (BCD) for 100 epochs. The QCNs are trained using both ideal and noisy simulation. The final training and test accuracy is also included for the models trained on BCD. The best test MSE and accuracy is highlighted. NA denotes undefined quantities.

Model	Type	Data	MSE (train/test)	Accuracy (train/test)
A (Ideal)	QCN	BHD	$4.29 \times 10^{-3} / 4.32 \times 10^{-2}$	NA
A (Noisy)	QCN	BHD	$5.53 \times 10^{-3} / \mathbf{4.26 \times 10^{-2}}$	NA
B	DNN	BHD	$5.59 \times 10^{-3} / 5.22 \times 10^{-2}$	NA
C (Ideal)	QCN	BCD	$2.78 \times 10^{-2} / 5.77 \times 10^{-2}$	0.989/0.947
C (Noisy)	QCN	BCD	$4.3 \times 10^{-2} / 6.5 \times 10^{-2}$	0.979/0.941
D	DNN	BCD	$1.11 \times 10^{-3} / \mathbf{3.19 \times 10^{-2}}$	1.000/ <b>0.965</b>

Comparing the models trained on the Boston Housing data, we see that the QCN has a lower test MSE than the DNN, meaning it generalizes better to unseen data. The same is however not true for the Breast Cancer data, where the QCN obtained almost twice the test MSE compared to the DNN.

## 7.6 Discussion

From Table 7.7, we see that the training MSE for the QCN trained on the Boston Housing data increased for the noisy simulation compared to the ideal. This is not surprising, since the added noise makes it harder to approximate the training data. However, the test error is actually slightly lower for the noisy simulation than the ideal. As discussed in Section 2.4, it has been shown that adding noise to the inputs of neural networks help them avoid overfitting [34], which in turn decreases the test error. This shows that noise introduced from noisy quantum hardware is not necessarily detrimental for the quality of the model, but could be actually helpful.

For the Breast Cancer data, the QCN performed overall worse than the DNN using both ideal and noisy simulation. While the QCN managed to obtain a test accuracy of 94.7% and 94.1% with ideal and noisy simulation, respectively, the DNN obtained a test accuracy of 96.5%. This is in contrast to the mixed Gaussian data and Boston Housing data, where QCNs were shown to outperform DNNs. This shows that QCNs may perform better than DNNs for some, but not all, problems.

## Part IV

### Conclusion & Future Research

# 8

## Summary & Conclusions

### 8.1 Summary & Conclusions

In this thesis we have developed a Python framework capable of implementing and training dense neural networks (DNNs), quantum neural networks (QNNs), and quantum circuit networks (QCNs) on various data sets. The models are optimized using gradient-based methods, such as Adam optimizer [27]. For the QCN, we developed a backpropagation algorithm based on the parameter shift rule for calculating its gradient analytically. As of now, we are preparing an article we hope to soon publish that includes many of the results and findings in this thesis [23].

#### Vanishing Gradient

Quantum neural networks are parameterized circuits used as machine learning models. The QNNs implemented in this thesis were inspired by the ones proposed by Abbas et al. [17]. We showed in Section 7.1.1 that increasing the number of qubits of QNNs caused their gradients to vanish exponentially, with a worsening effect for deeper circuits, due to the barren plateau phenomenon [20]. This causes an exponential overhead on the quantum hardware used for estimating the gradients, suggesting that training QNNs consisting of many qubits and a high circuit depth is intractable, especially on noisy quantum hardware.

Unlike QNNs, QCNs are constructed by combining several layers of parameterized circuits. In Section 7.1.2 we showed that the magnitude of the local gradients (the partial gradient of each circuit) of QCNs is unaffected when increasing the number of layers, but tends to vanish for increased number of qubits in each circuit. This

property enables the construction of large QCNs with several layers of small circuits such that their local gradients are easily estimated on quantum hardware.

In [Section 7.1.3](#), we showed that the gradients of QCNs still vanish exponentially fast in the number of layers when calculated with backpropagation using the local gradients. However, since backpropagation is a classical computation, this vanishment of the gradient does not cause an exponential overhead on the quantum hardware.

### Loss Landscape

In [Section 7.2](#), we investigated the loss landscapes of DNNs, QNNs and QCNs by computing their empirical fisher information matrix (EFIM) spectra [\[8\]](#). We showed that the spectra of QCNs with few qubits and layers are similarly uniform as those of QNNs, indicating that the loss landscape lacks particular strong distortions in any direction. This property is known to speed up gradient-based optimization [\[8\]](#). However, when increasing the number of qubits and layers of the QCNs, we observe that the EFIM spectra become increasingly skewed like for DNNs, resulting in loss landscapes that are flat in most directions, but highly distorted in one direction. This is a direct result of the vanishing gradient phenomenon, which causes the gradient of initial layers in DNNs and QCNs to vanish exponentially fast. Consequently, QCNs with many layers and qubits are likely to be slow to train, like DNNs.

### Expressivity

In [Section 7.3.1](#), we showed that untrained QCNs of sufficiently many qubits and similarly sized DNNs exhibit an exponentially decaying trajectory length. This indicates that these models compute approximately constant functions that are insensitive to the input features. For the QCNs, this behaviour results from the fact that untrained parameterized circuits approximate random circuits, which tend to produce outputs that concentrate closely around some mean value [\[20\]](#). As QCNs are constructed using layers of parameterized circuits, this concentration of the outputs is applied for each layer, leading to an exponentially decaying trajectory length.

We showed in [Section 7.3.2](#) that the expressivity of QCNs and DNNs can be moved into the exponential regime through training, producing exponentially more expressive models for each layer. In addition, the trajectory length increases faster for the QCNs compared to the DNNs, requiring two orders of magnitude fewer epochs to reach the exponential growth regime for the latter. This suggests that QCNs can be trained to be more expressive than similarly sized DNNs, and hence fit more complex data.

### Training on Mixed Gaussian Data

In [Section 7.4.1](#), we showed that QCNs using four to five qubits, and two to three layers, minimize their training MSE faster than similarly sized DNNs when fit to Gaussian data, as was suggested by the analysis of the loss landscape in [Section 7.2](#). When the DNNs are trained for 10000 epochs (until saturation), they obtain a lower MSE than the QCNs, but still within the same order of magnitude. As the QCNs were trained for only 100 epochs, this might suggest that they can outperform the DNNs given enough training. While speculative, this indicates that QCNs are more expressive than DNNs and can thus fit more complicated data, as was predicted by the analysis of its trajectory length in [subsection 5.2.1](#). On the other hand, the QNNs implemented in this thesis are unable to fit the mixed Gaussian data, suggesting that the RZZ encoding of the data or the repetitions of the simple anstaz, or both, are unfit for producing a QNN that can successfully train on mixed Gaussian data.

In [Section 7.4.2](#), we showed that QCNs still outperform DNNs when trained using a noisy simulation of the Santiago quantum computer [\[41\]](#). Due to the low circuit depth and number of qubits for each circuit, the local gradients have a relatively large magnitude, as was found in [Section 7.1](#). Thus, it is easy to obtain a good signal-to-noise ratio, making QCNs robust with respect to noise. Not surprisingly, the QNNs perform even worse when using noisy simulation due to their high circuit depths.

### Training and Generalization for Real-World Data

In [Section 7.5](#), we trained QCNs and DNNs on the real-world data sets Boston Housing data [\[43\]](#) and Breast Cancer Wisconsin data [\[44\]](#). We showed that QCNs generalize better on the former data set than DNNs. Surprisingly, the QCNs generalized slightly better when using noisy simulation compared to ideal. A possible explanation is that the noise added during training acts regularizing on the model, making it less susceptible to overfitting and more likely to generalize. This is known to happen for neural networks when adding noise to the input data [\[34\]](#), and shows that noise introduced by real quantum hardware is not necessarily detrimental for QCNs, but could actually be beneficial. This is in contrast to other quantum algorithms, like Shor's algorithm [\[12\]](#).

For the Breast Cancer data, the QCNs obtained a test accuracy of 95% and 94% using ideal and noisy simulation, respectively. The DNN outperformed them both, obtaining a 97% test accuracy. This shows that QCNs are not necessarily always superior to DNNs with similar number of parameters, but are better for specific training problems.

## 8.2 Future Research

For future research, we suggest experimenting with different ansatzes beyond the simple ansatz Equation (4.7) for constructing QCNs. Cerezo et al. [16] recently showed that there exists parameterized circuits with circuit depth logarithmic in the number of qubits that does not suffer from an exponential vanishing gradient. While such shallow circuits likely are unfit as QNNs on their own due to their low complexity, they could be useful as nodes in a QCN as one could utilize multiple circuits to build a more expressive model. This can potentially alleviate the vanishing gradient problem to some degree.

In this thesis, we found that QCNs outperform DNNs on mixed Gaussian data and the Boston Housing data, but not the Breast Cancer data. This begs the question: On what type of problems can quantum machine learning outperform classical methods? Recently, Liu et al. [45] provided a rigorous proof that *quantum kernels*, a type of quantum machine learning algorithm, have an exponential advantage over classical methods for supervised classification on certain data sets. These data sets are related to the *discrete logarithm problem*, which is widely believed to be hard to solve for classical computers. An interesting line of research would be to continue the search for problems for which quantum machine learning could outperform classical methods.

To establish a better comparison between QCNs and QNNs, we suggest exploring different QNN architectures that are able to fit nonlinear data. Recently, Schuld et al. [46] showed that QNNs can be universal function approximators if multiple repetitions of alternating feature encoding and ansatzes are applied. This means that a sufficiently deep QNN can approximate any function to an arbitrary accuracy. It would be interesting to see whether QCNs can outperform such QNNs when trained on noisy quantum hardware, since the QNN would likely have higher circuit depth and be more prone to noise.

Due to limited computational resources, the training of QNN and QCN models were limited to a small number of qubits and layers. This hindered the exploration of larger models to see how they perform and if they become intractable to train, as earlier analyses suggests. This can be explored by adapting the Python framework to run on supercomputers, or obtain access to real quantum computers with many qubits.

# Appendices



## Data Sets

In this chapter, we will present details surrounding the data sets used for training and testing models in this thesis.

### A.1 Mixed Gaussian Data

In order to obtain a complex, varying surface suited for regression, we choose to generate such data artificially by summing multiple Gaussian functions with different means and standard deviations. This creates what is known as mixed Gaussian data. Given a data point  $\mathbf{x}$  with  $p$  features, the output of a general multivariate Gaussian (without normalization and correlations) can be computed using

$$y = e^{(\mathbf{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})}, \quad (\text{A.1})$$

where  $\boldsymbol{\mu}$  is a  $p$ -dimensional vector that defines the position of the center of the Gaussian function, and  $\Sigma$  is a  $p \times p$  diagonal matrix defining the extension of the Gaussian in each direction. In this thesis, we will prepare samples  $\mathbf{x}^{(i)} \in [0, 1]^p$  as a meshgrid that uniformly fills the input space  $[0, 1]^p$ . This ensures a dense data set that captures the details of the mixed Gaussian function. We will generate data sets for  $p \in [1, 2, 3]$ . These differed data sets are described in Table A.1. For a visualization, see Figure A.1, Figure A.2 and Figure A.3. For a complete description on how the data is generated, see <https://github.com/KristianWold/Master-Thesis/blob/main/src/utils.py>.

To import the mixed Gaussian data, the following code can be used:

```
1 from utils import generate_1D_mixed_gaussian
2 from utils import generate_2D_mixed_gaussian
```



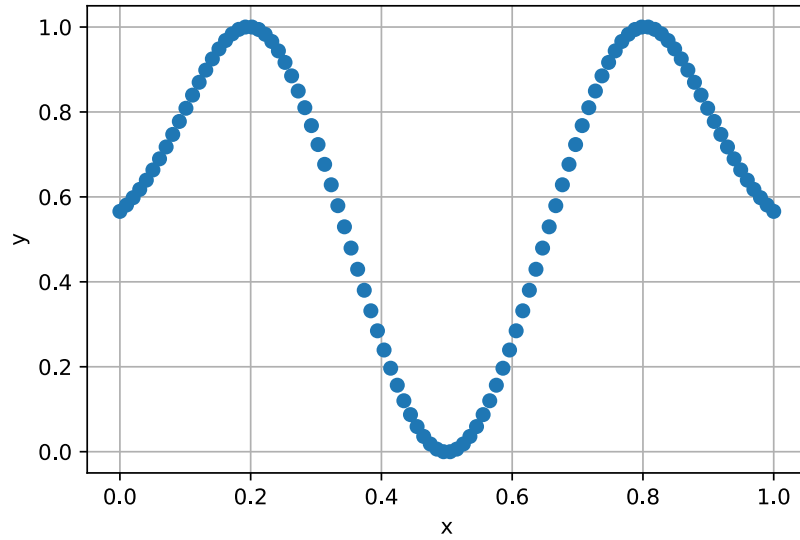
```

3 from utils import generate_3D_mixed_gaussian
4
5 x1, y1 = generate_1D_mixed_gaussian()
6 x2, y2 = generate_2D_mixed_gaussian()
7 x3, y3 = generate_1D_mixed_gaussian()

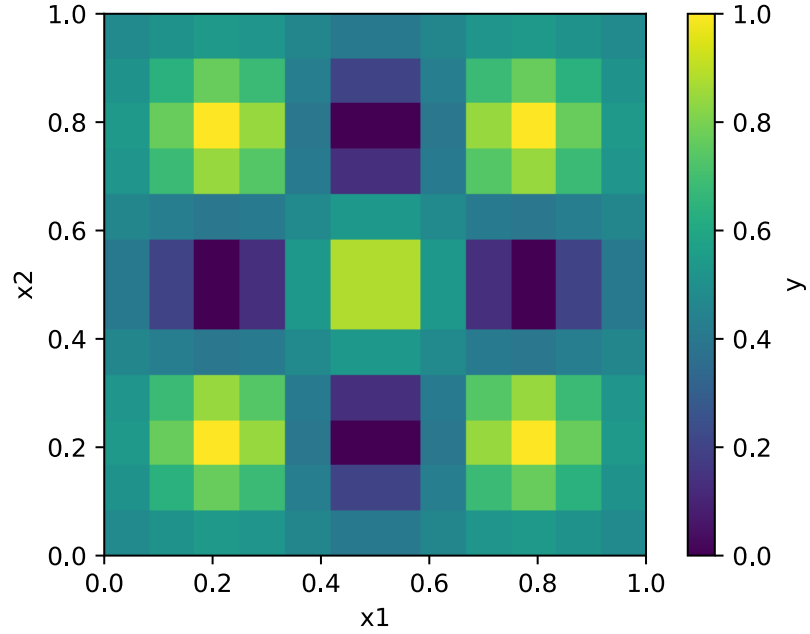
```

**Table A.1:** Details on the various mixed Gaussian data sets. For a complete description on how to produce it, see <https://github.com/KristianWold/Master-Thesis/blob/main/src/utils.py>

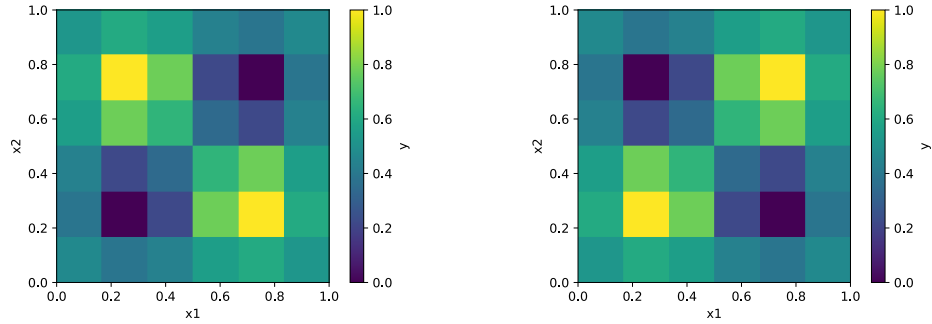
Name	#Samples	# Features	Feature Type	Target Type
1D mixed Gaussian	100	1	$x_i \in [0, 1]$	$y \in [0, 1]$
2D mixed Gaussian	144	2	$x_i \in [0, 1]$	$y \in [0, 1]$
3D mixed Gaussian	216	3	$x_i \in [0, 1]$	$y \in [0, 1]$



**Figure A.1:** Visualization of the 1D mixed Gaussian dataset. For more details, see [Table A.1](#).



**Figure A.2:** Visualization of the 2D mixed Gaussian dataset. For more details, see [Table A.1](#)



(a) Slice of the data set at  $x_3 = \frac{1}{6}$ .

(b) Slice of the data set at  $x_3 = \frac{5}{6}$ .

**Figure A.3:** Visualization of the 3D mixed Gaussian dataset. For more details, see [Table A.1](#)

## A.2 Real Data

For benchmarking the dense neural networks (DNNs) and quantum circuit networks (QCNs) implemented in this thesis against realistic data sets, and for investigating how they generalize to unseen data, we will be using the popular Boston Housing

data [43] and Breast Cancer Wisconsin data [44]. In this section, we will presents details about these data sets.

### A.2.1 Boston Housing Data

The Boston Housing data is a popular data set used for regression, readily available though the scikit-learn python package [28]. The data set can be loaded using the following code:

```
1 from sklearn.datasets import load_boston
2 data = load_boston()
3 x = data.data
4 y = data.target
```

The targets  $y$  of the Boston Housing data are the *median values of owner-occupied homes by town*, in \$1000, which can be predicted using methods for regression. Some of the features include quantities such as *the per capita crime rate by town*, *the average number of rooms per dwelling* and *the pupil-teacher ratio by town*. For a complete description of the features, see <https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html>. For some general details of the data set, see Table A.2.

**Table A.2:** Some details on the Boston Housing data set.

Name	#Samples	# Features	Feature Type	Target Type
Boston Housing Data	506	13	$x_i \in \mathbb{R}$	$y \in \mathbb{R}$

### A.2.2 Breast Cancer Wisconsin Data

The Breast Cancer Wisconsin data set is another popular data accessible through scikit-learn. The data set can be loaded using the following code:

```
1 from sklearn.datasets import load_breast_cancer
2 data = load_breast_cancer()
3 x = data.data
4 y = data.target
```

The targets  $y$  of the data set are binary values indicating whether breast tissue is malignant or benign, suitable for classification methods. Some of the features include quantities such as *mean radius*, *mean area* and *mean smoothness* describing the breast tissue. For a complete description of the features, see <https://www.kaggle.com/uciml/breast-cancer-wisconsin-datal>. For some general details of the data set, see Table A.3.

**Table A.3:** Some details on the Breast Cancer Wisconsin data set.

Name	#Samples	# Features	Feature Type	Target Type
Breast Cancer Wisconsin	569	30	$x_i \in \mathbb{R}$	$y \in \{0, 1\}$

### A.2.3 Feature Reduction with PCA

For both the Boston Housing data and the Breast Cancer Wisconsin data, we will perform a principal component analysis (PCA) to reduce the number of features from 13 and 30 to four, respectively. This makes the training of QCNs more feasible, as a high number of features also require a high number of qubits when using qubit encoding. For more details, see [Section 2.5.2](#).

## References

- [1] Maria Schuld and Francesco Petruccione. *Supervised Learning with Quantum Computers*. 1st. Springer Publishing Company, Incorporated, 2018 (cit. on pp. 1, 4, 8, 13, 26, 32, 33).
- [2] George D Magoulas and Andriana Prentza. *Machine learning in medical applications*. Berlin: Springer, 2001, pp. 300–307 (cit. on p. 1).
- [3] Ahmet Murat Ozbayoglu, Mehmet Ugur Gudelek, and Omer Berat Sezer. *Deep Learning for Financial Applications : A Survey*. 2020. arXiv: [2002.05786 \[q-fin.ST\]](#) (cit. on p. 1).
- [4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. 2nd. New York, NY, USA: Springer New York Inc., 2009 (cit. on pp. 1, 8, 9, 19, 21, 50).
- [5] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O’Reilly Media, 2017 (cit. on pp. 1, 2, 12, 14, 19, 72).
- [6] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. *On the Expressive Power of Deep Neural Networks*. 2017. arXiv: [1606.05336 \[stat.ML\]](#) (cit. on pp. 2, 50–52, 83, 85, 86).
- [7] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. “Efficient BackProp.” In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 32–34 (cit. on pp. 2, 19, 49).
- [8] Ryo Karakida, Shotaro Akaho, and Shun-ichi Amari. *Universal Statistics of Fisher Information in Deep Neural Networks: Mean Field Approach*. 2019. arXiv: [1806.01316 \[stat.ML\]](#) (cit. on pp. 2, 48–50, 82, 99).
- [9] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011 (cit. on pp. 2, 22, 27, 39).
- [10] D. Deutsch. “Quantum theory, the Church–Turing principle and the universal quantum computer.” In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400 (1985), pp. 117–97 (cit. on p. 2).
- [11] Seth Lloyd. “Universal Quantum Simulators.” In: *Science* 273.5278 (1996), pp. 1073–1078 (cit. on p. 2).

- [12] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.” In: *SIAM Journal on Computing* 26 (1997), pp. 1484–1509 (cit. on pp. 3, 35, 100).
- [13] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini. “Parameterized quantum circuits as machine learning models.” In: *Quantum Science and Technology* 4 (2019), p. 043001 (cit. on pp. 3, 4, 35–37, 41).
- [14] John Preskill. “Quantum Computing in the NISQ era and beyond.” In: *Quantum* 2 (2018), p. 79 (cit. on pp. 3, 33).
- [15] Abdullah Ash Saki, Mahabubul Alam, and Swaroop Ghosh. *Study of Decoherence in Quantum Computers: A Circuit-Design Perspective*. 2019. arXiv: [1904.04323 \[cs.ET\]](#) (cit. on p. 3).
- [16] M. Cerezo, Akira Sone, Tyler Volkoff, Lukasz Cincio, and Patrick J. Coles. “Cost function dependent barren plateaus in shallow parametrized quantum circuits.” In: *Nature Communications* 12.1 (2021) (cit. on pp. 4, 35, 101).
- [17] Amira Abbas, David Sutter, Christa Zoufal, Aurélien Lucchi, Alessio Figalli, and Stefan Woerner. *The power of quantum neural networks*. 2020. arXiv: [2011.00027 \[quant-ph\]](#) (cit. on pp. 4, 14, 36, 38, 40, 41, 43, 48, 50, 70, 72, 82, 98).
- [18] Seth Lloyd, Maria Schuld, Aroosa Ijaz, Josh Izaac, and Nathan Killoran. *Quantum embeddings for machine learning*. 2020. arXiv: [2001.03622 \[quant-ph\]](#) (cit. on pp. 4, 39, 40).
- [19] Maria Schuld, Alex Bocharov, Krysta M. Svore, and Nathan Wiebe. “Circuit-centric quantum classifiers.” In: *Physical Review A* 101 (2020) (cit. on pp. 4, 36).
- [20] Jarrod R. McClean, Sergio Boixo, Vadim N. Smelyanskiy, Ryan Babbush, and Hartmut Neven. “Barren plateaus in quantum neural network training landscapes.” In: *Nature Communications* 9 (2018) (cit. on pp. 4, 43, 75, 98, 99).
- [21] Andrea Skolik, Jarrod R. McClean, Masoud Mohseni, Patrick van der Smagt, and Martin Leib. *Layerwise learning for quantum neural networks*. 2020. arXiv: [2006.14904 \[quant-ph\]](#) (cit. on pp. 4, 14, 44, 70, 79).
- [22] Stian Bilek. *Quantum Computing: Many-Body Methods and Machine Learning*. unpublished (cit. on pp. 4, 44, 47).
- [23] Kristian Wold and Stian Bilek. *Quantum Circuit Networks*. In preparation. (cit. on pp. 5, 98).
- [24] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. DOI: [10.5281/zenodo.2562110](#) (cit. on pp. 6, 54).
- [25] Michael A. Nielsen. *Neural Networks and Deep Learning*. 1st ed. Determination Press, 2018. URL: <http://neuralnetworksanddeeplearning.com/> (cit. on p. 8).
- [26] S. A. Vavasis. *Nonlinear Optimization: Complexity Issues*. New York: Oxford University Press, 1991 (cit. on p. 11).
- [27] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: [1412.6980 \[cs.LG\]](#) (cit. on pp. 12, 13, 72, 98).
- [28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

- D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 14, 47, 70, 71, 106).
- [29] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” In: *Advances in Neural Information Processing Systems* 32. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf> (cit. on pp. 14, 71).
- [30] Alexander LeNail. “NN-SVG: Publication-Ready Neural Network Architecture Schematics.” In: *Journal of Open Source Software* 4.33 (2019), p. 747 (cit. on p. 15).
- [31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Internal Representations by Error Propagation.” In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362 (cit. on p. 16).
- [32] Julius Berner, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. “The Modern Mathematics of Deep Learning.” In: *CoRR* abs/2105.04026 (2021). arXiv: 2105.04026 (cit. on p. 17).
- [33] Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. *Failures of Gradient-Based Deep Learning*. 2017. arXiv: 1703.07950 [cs.LG] (cit. on p. 18).
- [34] Hyeonwoo Noh, Tackgeun You, Jonghwan Mun, and Bohyung Han. *Regularizing Deep Neural Networks by Noise: Its Interpretation and Optimization*. 2017. arXiv: 1710.05179 [cs.LG] (cit. on pp. 18, 96, 100).
- [35] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell, and et al. “Superconducting quantum circuits at the surface code threshold for fault tolerance.” In: *Nature* 508 (2014), pp. 500–503 (cit. on p. 33).
- [36] Adam Holmes, Sonika Johri, Gian Giacomo Guerreschi, James S Clarke, and A Y Matsuura. “Impact of qubit connectivity on quantum algorithm performance.” In: *Quantum Science and Technology* 5 (2020), p. 025009 (cit. on p. 34).
- [37] Seth Lloyd. *Quantum approximate optimization is computationally universal*. 2018. arXiv: 1812.11075 [quant-ph] (cit. on p. 39).
- [38] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. “Evaluating analytic gradients on quantum hardware.” In: *Physical Review A* 99 (2019) (cit. on p. 41).
- [39] Patrick Huembeli and Alexandre Dauphin. “Characterizing the loss landscape of variational quantum circuits.” In: *Quantum Science and Technology* 6.2 (2021), p. 025011 (cit. on p. 49).
- [40] Peter Bartlett, Vitaly Maiorov, and Ron Meir. “Almost Linear VC Dimension Bounds for Piecewise Polynomial Networks.” In: *Advances in Neural Information Processing Systems*. Ed. by M. Kearns, S.olla, and D. Cohn. Vol. 11. MIT Press, 1999 (cit. on p. 50).

- [41] IBM Q Team. *IBM Q Santiago 5 Qubit Quantum Computer v1.0.3* (cit. on pp. [57](#), [90](#), [92](#), [93](#), [95](#), [100](#)).
- [42] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks.” In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterton. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256 (cit. on p. [71](#)).
- [43] D. Harrison and D.L. Rubinfeld. “Hedonic Housing Prices and the Demand for Clean Air.” In: *Journal of Environmental Economics and Management* 5 (1978), pp. 81–102 (cit. on pp. [95](#), [100](#), [106](#)).
- [44] W Nick Street William H Wolberg and Olvi L Mangasarian. “Breast cancer Wisconsin (diagnostic) data set.” In: *Analytical and quantitative cytology and histology* 17 (1995), pp. 77–87 (cit. on pp. [95](#), [100](#), [106](#)).
- [45] Yunchao Liu, Srinivasan Arunachalam, and Kristan Temme. “A rigorous and robust quantum speed-up in supervised machine learning.” In: *Nature Physics* (2021) (cit. on p. [101](#)).
- [46] Maria Schuld, Ryan Sweke, and Johannes Jakob Meyer. “Effect of data encoding on the expressive power of variational quantum-machine-learning models.” In: *Physical Review A* 103 (2021) (cit. on p. [101](#)).