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

August 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.

www.duo.uio.no

Print production: Reprosentralen, University of Oslo

Abstract

Stuff

Acknowledgements

The work in this thesis was conducted at the Centre for Integrative Neuroplasticity (CINPLA).

Kristian Wold
Oslo, August 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

Contents

1

Introduction and Objective of the Study

1.1 Introduction

1.2 Objective of the Study

1.3 How This Thesis is Organized

How This Book is Organized (ISL)

Part I

Theoretical Background

2

Supervised Learning

This chapter introduces the fundamentals of supervised learning, optimization and neural networks. The content of this chapter is mainly based on the material in [1], [2] and [3].

The goal of *supervised learning*, one of the big branches of machine learning, is to obtain a function for predicting an output y from an input $\boldsymbol{x}$. This is done by learning from input-output pairs $\mathcal{T} = \{(\boldsymbol{x}^{(1)}, y^{(1)}), \cdots, (\boldsymbol{x}^{(N)}, y^{(N)})\}$, known as the training set. 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 constitutes two big paradigms in supervised learning: *regression* and *classification*, respectively. In the case of regression, the goal of the learning task is to predict a real-valued target y from the input $\boldsymbol{x}$. Typical examples of targets to regress on are *temperature*, *weight* and *number of people*, which have in common a natural notion of distance measure in the sense that instances close in numerical value are also close in nature. E.g., two fish weighing 12.1 kg and 12.2 kg are similar, while a third fish weighing 24.0kg is notably different.

For classification, the goal is to predict one or more *classes* from an input $\boldsymbol{x}$. In this setting, the target y is discrete and categorical, such as *color*, *dead/alive* and *type of animal*. 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 $\boldsymbol{x}$ is a vector consisting of elements $(x_1, \cdots, x_p)$ often called features and 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 $\boldsymbol{x} \in \mathbb{R}^p$.

2.1 Parametric Models

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

$$y = f(\boldsymbol{x}) + \epsilon,$$

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

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

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

where $\hat{y}$ is the predicted value, $f(\cdot;\cdot)$ defines a *family* of models, and $\boldsymbol{\theta}$ is a specific vector of parameters that defines a specific model from the family. Training the model involves finding the parameters $\boldsymbol{\theta}$ such that the model best reproduces the target 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:

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}), \tag{2.1}$$

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.

The choice of loss function is very problem dependent, and there is a vast collection of different choices in the machine learning literature Hastie, Tibshirani, and Friedman [2]. In this thesis we focus on the popular Mean Squared Error(MSE) when training supervised learning models. This loss function is suitable for regression problems

since it implements a natural distance measure between prediction and target. It is formulated as

$$MSE = \frac{1}{2N}\sum_{i=1}^{N}(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - y^{(i)})^2. \tag{2.2}$$

Fitting the model using MSE as loss function is often referred to as the *least squares approach*.

2.2 Optimization

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

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}). \tag{2.3}$$

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 2.3 does not generally have an analytical solution. More over, many popular parametric models result in non-convex optimisation problems, meaning the *loss landscape* exhibit several local minima. In practice, such optimization problems can't be solved efficiently [4]. However, it is important to realize that an exact, or close to exact, minimization of the loss function is seldom needed or even favourable. 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 with respect to the model parameters can be calculated as

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial \hat{y}^{(i)}} L(\hat{y}^{(i)}, y^{(i)}) \frac{\partial}{\partial \boldsymbol{\theta}_k} \hat{y}^{(i)} \tag{2.4}$$

where $\boldsymbol{\theta}_k$ is the k'th model parameter, and $\hat{y}^{(i)} = f(\boldsymbol{x}^{(i)};\boldsymbol{\theta})$. 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 $\boldsymbol{\theta}_k$, respectively. Notice 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:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \Big(\frac{\partial}{\partial \boldsymbol{\theta}_1} L(\boldsymbol{\theta}), \cdots, \frac{\partial}{\partial \boldsymbol{\theta}_p} L(\boldsymbol{\theta}) \Big) \tag{2.5}$$

The gradient Equation 2.5 can be geometrically interpreted as the direction at point $\boldsymbol{\theta}$ 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

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mu \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_{t-1}) \tag{2.6}$$

for $t = 1, \cdots, T$. Here, T is the total number of iterations, and μ is some small positive value, often called the *learning rate*. Usually, some initial choice of parameters $\boldsymbol{\theta}_0$ is chosen at random. The proceeding recalculation of the gradient and repeated adjustment of the parameters result in a gradual descent in the the loss landscape, analogous to walking down a mountain. 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 can have 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 is illustrated in Figure 2.2.1. Further more, the presence of high degree of distortions in certain directions in parameter space, so called *thin valleys*, can lead to oscillations and inefficient optimization. This is exemplified in Figure 2.2.2. We will discuss how the particularly popular *Adam optimizer*, which was used in this thesis, addresses these problems.

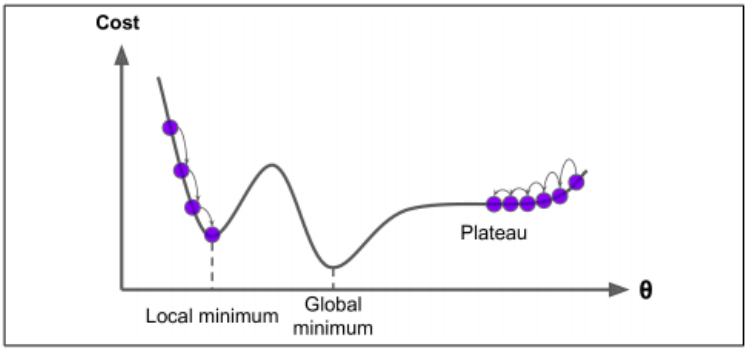


Figure 2.2.1: *One-dimensional representation of the loss landscape for a parameterized model, showcasing the phenomenon of getting stuck in local minima, and slow convergence induced by plateaus. The figure is retrieved from* **geÌĄron2017hands-on.**

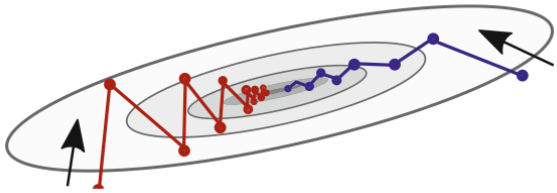


Figure 2.2.2: *Two-dimensional representation of the loss landscape for a parameterized model, illustrating a thin valley. The optimization steps in red showcase optimization without momentum. Optimization steps in blue implement momentum, showing dampened oscillation and better convergence. The figure is retrieved from Schuld and Petruccione [1].*

2.2.2 Adam Optimizer

Introduced by Kingma and Ba [5], the Adam algorithm implements a moving average of the gradient, called *momentum*, together with a rescaling. Replacing Equation 2.5 and Equation 2.6, Adam implements the following algorithm:

Algorithm 1: *Adam*, [5]. 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 θ_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)
  $\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end
return θ_t

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 tend to flip sign for certain directions, the averaging over previous iterations tends to dampen these oscillations. Likewise, directions of persistent sign tends to accumulate magnitude, making the optimisation gain "momentum" in these directions. This is a good property for overcoming thin valleys and plateaus. Also, the effect of momentum may also help avoid getting stuck in local minima by gracing over them. Further, the moving average of the gradient and its square is rendered unbiased as $\hat{m}_t = m_t / (1 - \beta_1)$ and $\hat{v}_t = v_t / (1 - \beta_2)$. Since the averages are initialized as zero, they are biased downward. Finally, the parameters are shifted by the quantity $\alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$. 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 hugely successful algorithm for optimizing machine learning models. In popular machine learning frames such as scikit-learn [6] and pyTorch [7], it is a default optimizer. Lately, Adam has also been used for optimizing quantum machine learning models [8] [9]. Pointed out by its authors, Adam requires very little tuning of hyper-parameters to be efficient, making 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(kilde?), artificial neural networks are powerful parameterized machine learning models that have proved to extremely useful for a vast number of applications. Over the years, a comprehensive collection of different network architectures have 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*, 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 non-linear transformation. The result of the first layer of the neural network can be formulated as

$$\boldsymbol{a}^1 = f^1(\boldsymbol{z}^1) = f^1(W^1\boldsymbol{x} + \boldsymbol{b}^1), \tag{2.7}$$

Here, $\boldsymbol{x} \in \mathbb{R}^p$ is a single sample of p features. $W^1 \in \mathbb{R}^{m\times p}$ and $\boldsymbol{b}^1 \in \mathbb{R}^m$ are a matrix and a vector of parameters called the *weights* and *biases*, respectively. The operations $W^1\boldsymbol{x} + \boldsymbol{b}^1$ applies 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 non-linear, applied element-wise on the derived features. This finally results in the output of the layer, $\boldsymbol{a}^1$, called the *activation*.

We will now generalize Equation 2.7 to an arbitrary layer. For a neural network with L layers, the feedforward procedure for layer l can be formulated as

$$\boldsymbol{a}^l = f^l(\boldsymbol{z}^l) = f^1(W^l\boldsymbol{a}^{\boldsymbol{l-1}} + \boldsymbol{b}^l), \tag{2.8}$$

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

$$\hat{y} = f_{DNN}(x; \boldsymbol{\theta}) = \boldsymbol{a}^L, \tag{2.9}$$

where $\boldsymbol{\theta} = [W^1, \boldsymbol{b}^1, \cdots, W^L, \boldsymbol{b}^L]$. This defines the whole forward procedure of a neural network, and also highlights the role of the functions $f^l(\cdot)$, often called the *non-linearities*. If set to identity, $f^l(x) = x$, the recursive application of Equation 2.8 would simply apply repeated affine transformations, which is an affine transformation in itself. 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 non-linear transformations is necessary to increase the flexibility of the neural network(kilde?).

2.3.2 Backpropagation

Assume that $f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$ is a dense neural network as described by Equation 2.8 and Equation 2.9. In order to use gradient-based methods, one needs to calculate the derivative of the loss-function Equation 2.4 for an arbitrary parameter $\boldsymbol{\theta}_k$, which could be any of the weights W^l or biases $\boldsymbol{b}^l$ in the various layers. This is not trivial given the sequential structure of the neural network. Often attributed to Rumelhart, Hinton, and Williams [10], the *backpropagation algorithm* calculates the gradient in a sequential manner, starting with the last layers first. Calculating on a single sample, the algorithm starts by calculating the *error* of the last layer

$$\delta_k^L = \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^L}, \tag{2.10}$$

where k indicates the node.

This error can be defined for any layer recursively by repeated application of the chain-rule:

$$\delta_j^l = \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_j^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^{l+1}} \frac{\partial \boldsymbol{a}_k^{l+1}}{\partial \boldsymbol{a}_j^l} = \sum_k \delta_k^{l+1} \frac{\partial \boldsymbol{a}_k^{l+1}}{\partial \boldsymbol{a}_j^l}. \tag{2.11}$$

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

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

$$\frac{\partial L(\hat{y}, y)}{\partial W_{ij}^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^l} \frac{\partial \boldsymbol{a}_k^l}{\partial W_{ij}^l} = \sum_k \delta_k^l f^{l\prime}(\boldsymbol{z}_k^l)\boldsymbol{a}_j^{l-1} I_{ik} = \delta_i^l f^{l\prime}(\boldsymbol{z}_i^l)\boldsymbol{a}_j^{l-1}, \tag{2.12}$$

and

$$\frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{b}_i^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^l} \frac{\partial \boldsymbol{a}_k^l}{\partial \boldsymbol{b}_i^l} = \sum_k \delta_k^l f^{l\prime}(\boldsymbol{z}_k^l)\boldsymbol{a}_j^{l-1} I_{ik} = \delta_i^l f^{l\prime}(\boldsymbol{z}_i^l). \tag{2.13}$$

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

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\hat{y}^{(i)}, y^{(i)}), \tag{2.14}$$

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

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 [11].

3.1 States in Quantum Mechanics

In quantum mechanics, isolated physical systems are completely described 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 $(z_1, \cdots, z_n)$ called *amplitudes*. Adopting Dirac notation, a state is denoted as

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

where ψ 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 spaces*, 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

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

where z^* indicates the complex conjugate, i.e. if $z = a + ib$, we have that $z^* = a - ib$. As a constraint on the amplitudes, state vectors describing physical systems unit norm, meaning

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

3.1.1 The Qubit

As is common in quantum computing, we will be focusing 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

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{3.4}$$

where α and β are complex numbers, and $|0\rangle$ and $|1\rangle$ are orthonormal states known as the *computational basis states* and are specially 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 α or β 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 very 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 composit systems by combining several smaller quantum systems. This can be used to construct systems of multiple qubits, whose state can be expressed, if the qubits are independent, as

$$|\psi_1\psi_2\cdots\psi_n\rangle \equiv |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots |\psi_n\rangle\,. \tag{3.5}$$

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 superpostion, 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

$$|\psi\rangle = \sum_{\boldsymbol{v}} c_{\boldsymbol{v}}\,|\boldsymbol{v}_1\rangle \otimes |\boldsymbol{v}_2\rangle \otimes \cdots |\boldsymbol{v}_n\rangle\,, \tag{3.6}$$

where $\boldsymbol{v} \in \{0,1\}^n$ 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 $c_{\boldsymbol{v}}$ to describe the state of n qubits in general. In other words, the information stored in the quantum state of n qubits in exponential in n, as opposed to the linear information of a 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 can not 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 probability $|\alpha|^2$ and $|\beta|^2$, respectively. For multiple qubits in a general state Equation 3.7, a measurement on all qubits will grant a state in the computational basis, i.e. $|\boldsymbol{v}_1\rangle \otimes |\boldsymbol{v}_2\rangle \otimes \cdots |\boldsymbol{v}_n\rangle$ for some binary string $\boldsymbol{v} \in \{0,1\}^n$, with probability $|c_{\boldsymbol{v}}|^2$. This motivates why states in quantum mechanics needs to have unity norm, i.e

$$\sum_{\boldsymbol{v}} |c_{\boldsymbol{v}}|^2 = 1, \tag{3.7}$$

as the probabilities of any outcome must sum to 1.

3.2 Quantum Circuits

We have until now looked into how quantum states can encode information, and how information can be interacted with via measurement. How then can quantum mechanics be used for computation? To do computations, it is necessary to introduce some dynamical transformation of the quantum state. In quantum mechanics, transformations can be formulated as

$$|\phi\rangle = U|\psi\rangle, \tag{3.8}$$

where U is a *unitary* operator that acts on the vector space where $|\psi\rangle$ and $|\phi\rangle$ lives. 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:

$$\langle\phi|\phi\rangle = \langle\psi|\underbrace{U^\dagger U}_{I}|\psi\rangle = \langle\psi|\psi\rangle = 1. \tag{3.9}$$

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 transforms the state in a desired way, often called a *quantum circuit*. Typical operators used in quantum computing, often called quantum gates, act on either on one or multiple qubits, and bear resemblance to logical operations in the classical context.

Figure 3.2.1 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 seen in this schematic, and is highly dependent on the implementation of physical hardware. The horizontal lines, called wires, each symbolize a qubit. From the notation on the left hand, it can be seen that all qubits are initialized in the zero state. 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.

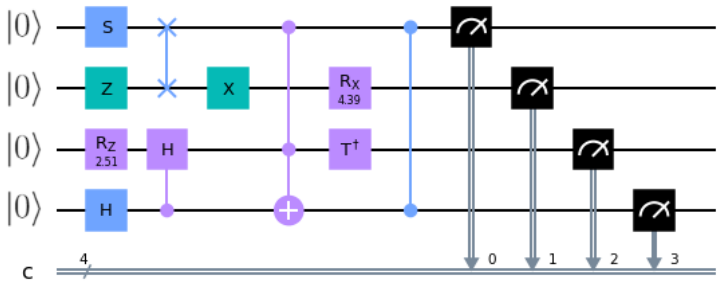


Figure 3.2.1: *Example circuit consisting of 4 qubits initialized to* $|0\rangle$*. A random selection of quantum gates acting on one, two and three qubits are then applied. Finally, all qubits are measured in the computational basis and stored in a classical register.*

3.2.1 Single Qubit Operations

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

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \equiv \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \tag{3.10}$$

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

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}, \tag{3.11}$$

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

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \text{—}\boxed{H}\text{—}. \tag{3.12}$$

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

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

which will lead to a 50% chance to yield either 0 or 1 upon measuring. In a sense, this is the quantum mechanical equivalent to a coin toss. However, it is actually much more interesting. If applied a second time, we actually return to the original state, "unscrambling" the coin, or qubit:

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

Pointed out in Schuld and Petruccione [1], this 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), allowing for 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 exponential 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:

$$\begin{aligned} X = \sigma_x &= \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} = \text{—}\boxed{X}\text{—}\\ Y = \sigma_y &= \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} = \text{—}\boxed{Y}\text{—}\\ Z = \sigma_z &= \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} = \text{—}\boxed{Z}\text{—} \end{aligned} \tag{3.14}$$

To visualize what these operators do, it is useful to introduce a graphical picture called the *Bloch sphere*, illustrated in Figure 3.2.2. Rewriting the state of a qubit to

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

the new parameters θ and ϕ 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.

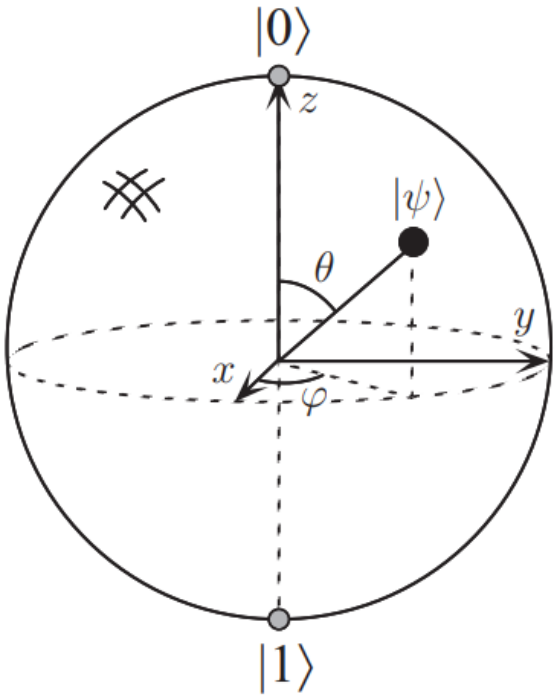


Figure 1.3. Bloch sphere representation of a qubit.

Figure 3.2.2: *Graphical picture visualizing the state of a qubit, called the Bloch sphere. The figure is retrived from [11].*

The Pauli-gates represent 180° rotation of the state around the corresponding axis. Particlularly, the X gate, often called the *flip gate*, acts on the basis states as

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

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

Another set of gates extremely important 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:

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

.

These action of these gates on any state is to rotate it on the Bloch sphere around the corresponding axis, for an amount of θ radians. As θ can be any real number, these gates can be though of as being quantum gates parameterized by θ, 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 operates 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

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

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:

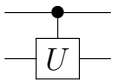
$$CNOT\,|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes I\,|0\rangle + |1\rangle \otimes X\,|0\rangle) = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle). \quad (3.20)$$

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 also 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 states possible to access.

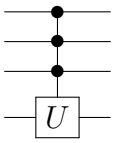
Multi-controlled Controlled Gate

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

$$|0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes U = \quad , \quad (3.21)$$

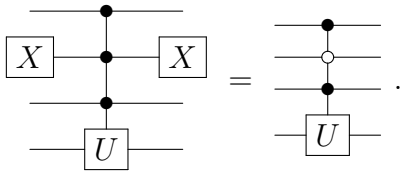


where U is any single qubit gate. Moreover, there also exists *multi-controlled gates* on the form



. (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 $|1\rangle$. It is also possible to condition on the control qubits being in state $|0\rangle$ rather that $|1\rangle$. This can be done by applying an X gate before and after the controlled operation to the qubit one wishes to invert, such as



. (3.23)

Here, the conditioning on state $|0\rangle$ is indicated by a white dot. Multi-controlled gates will be relevant when we later talk about *amplitude encoding*, which is a technique for encoding information onto the amplitudes of a quantum state.

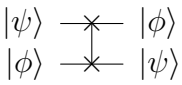
SWAP gate

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

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \quad , \tag{3.24}$$

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

$$\begin{array}{l} |\psi\rangle \\ |\phi\rangle \end{array} \quad \begin{array}{c} \\ \end{array} \quad \begin{array}{l} |\phi\rangle \\ |\psi\rangle \end{array} \tag{3.25}$$



3.2.3 Observables

In subsection 3.1.3, we shortly 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

$$\hat{O} = \sum_m m P_m, \tag{3.26}$$

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

$$p(m) = \langle\psi| P_m |\psi\rangle , \tag{3.27}$$

and original state will be projected onto P_m yielding

$$|\psi\rangle \to \frac{P_m |\psi\rangle}{\sqrt{\langle\psi| P_m |\psi\rangle}}, \tag{3.28}$$

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

$$\sigma_z = |0\rangle\langle 0| - |1\rangle\langle 1| \tag{3.29}$$

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

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

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 subsection 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

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

$\langle\psi|\hat{O}|\psi\rangle$ is a recurring expression in quantum mechanics, often denoted simply as $\langle\hat{O}\rangle$. The expectation value is very important as it serves as a method for extracting a deterministic value from a quantum state. Whereas the quantum information of a state is inaccessible to us as a whole, estimating the expectation value for some desired observable is possible for retrieving an output of a quantum algorithm. Hopefully, this output serves as a solution to the problem we wished to solve. Looking at the expected value Equation 3.34, 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 $|\psi\rangle \to e^{i\gamma}|\psi\rangle$? 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 shows that whatever measurement we do on the state, we can not determine if the global phase is present or not. Therefore, we can assume two state 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 that will be relevant in this thesis, we are able to express them as a spectral

decomposition of the computational basis, meaning we can write them in the form

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

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

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

Given that we know the eigenvalues λ_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 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 measure it in the computational basis, called performing several *shots*, one can gather S such samples $\{y_{i1}, \cdots y_{iS}\}$. As pointed out in Schuld and Petruccione [1], $|\alpha_i|^2$ can be estimated with a *frequentist estimator* $\hat{p}_i$ given by

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

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

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

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 need 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

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

From this expression, it can be seen that also the error of the expectation value goes as $O(1/\sqrt{S})$. This is a very costly 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 finite number of shots. In the case the quantity one tries to estimate is very small, the number of shots required to overcome bad signal-to-noise ratio can become prohibitively large.

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 in practice on quantum hardware, and what potential drawbacks such implementation might bring. Even in the ideal case, we saw in subsection 3.2.5 that outputs of quantum circuits are noisy as a result of finite shots. Quantum computing on near-term quantum hardware, so-called *noisy intermediate-scale quantum computing*(NISQ)[12], is characterized by few available qubits, low-fidelity computations and other restrictions. These are aspects that 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 [1] and [12].

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 implementation of qubits, *super-conducting qubits*, uses pulses of microwaves to control the qubits. Using this technique, Barends et al. [13] is able to implement quantum gates with as low as 1% measurement error probability, but oftentimes also 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 accumulation of error that will render the outcome useless[12]. Because of this, the number of gates should be kept low in order to minimize error of the quantum algorithm.

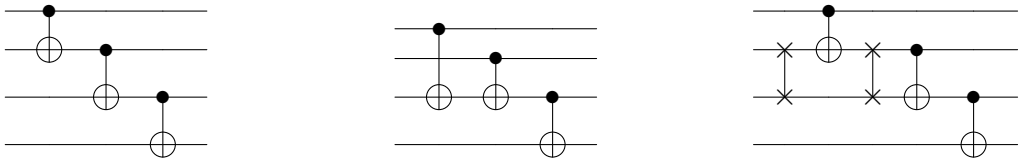
3.3.2 Decoherence

In addition to the error introduced by 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 practise, quantum computers are only approximately isolated, as vibrations and external fields tends to leak into the system, degrading

the information stored in the state(kilde?). This effect tends to worsen 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 specifics 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*[14]. This means two-qubit gates may only be applied on neighboring qubits, i.e. they are linearly connected. Figure 3.3.1 gives examples on quantum circuits that either respect or violate the linear connectivity of qubits.



(a) *Respecting the linear connectivity of qubits.*

(b) *Violating the linear connectivity of qubits.*

(c) *Restoring the linear connectivity using SWAP.*

Figure 3.3.1: *Different quantum circuit that either respect or violate the linear connectivity of qubits.*

Figure 3.3.1a applies CNOT gate only on neighboring qubits, which is allowed on a linear architecture. In contrast, Figure 3.3.1b can be seen to violate it. However, the circuit in Figure 3.3.1c have 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.

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[15], which promises to factor integers in polynomial time. This is 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 and inaccuracy of the gates is corrected for. Also, their implementation requires often 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 some interesting candidate algorithms for useful near-term applications? A promising family of algorithms are *parameterized quantum circuits*(PQC), which are quantum circuits comprised of fixed gates, such as CNOT gates, and adjustable gates, such as Pauli rotations[16]. 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 needs 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 though 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[17]. 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 hard in practice because of how non-intuitive quantum computing can be.

Lately, there has been a lot of research on the use of PQC as machine learning models, often called *Quantum Neural Network*(QNN)[8][16]. In general, the typical structure of QNNs can be broken up into three stages: *feature encoding*, *processing* and *measurement* with potential post-processing. This general procedure is summarized in Figure 4.0.1.

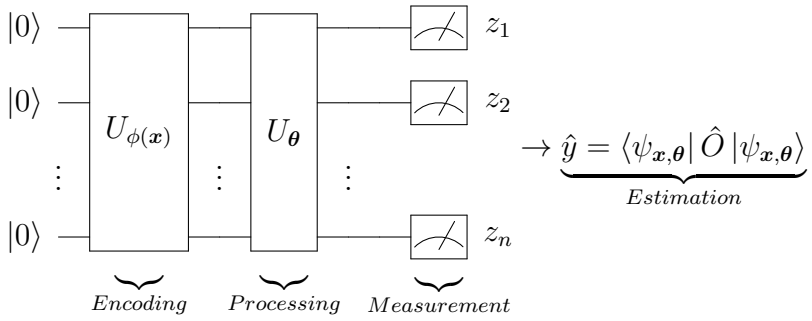


Figure 4.0.1: *The general structure of a Quantum Neural Network(QNN). The procedure consists of three steps: A routine $|\psi_{\boldsymbol{x}}\rangle = U_{\phi(\boldsymbol{x})} |0\rangle$ for encoding a feature vector onto an n-qubit Hilbert space. Next, $|\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle = U_{\boldsymbol{\theta}}$ applies a unitary transformation parameterized by $\boldsymbol{\theta}$ $|\psi_{\boldsymbol{x}}\rangle$, transforming the state in Hilbert space. Lastly, the expectation value of some appropriate observable $\hat{O}$ is estimated for the resulting state, yielding in a model output $\hat{y} = \langle\psi_{\boldsymbol{x},\boldsymbol{\theta}}| \hat{O} |\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle$.*

The QNN starts by encoding a feature vector $\boldsymbol{x}$ onto a n-qubit Hilbert space by using a unitary transformation

$$|\psi_{\boldsymbol{x}}\rangle = U_{\phi(\boldsymbol{x})} |\mathbf{0}\rangle , \tag{4.1}$$

often called a *quantum feature map*. Next, a circuit parameterized by $\boldsymbol{\theta} = (\theta_1, \cdots, \theta_{n_\theta})$ is applied, resulting in

$$|\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle = U_{\boldsymbol{\theta}} |\psi_{\boldsymbol{x}}\rangle . \tag{4.2}$$

In this context, this circuit $U_{\boldsymbol{\theta}}$ is often called an *ansatz* and serves as a general way of transforming the state $|\psi_{\boldsymbol{x}}\rangle$ encoding the data in Hilbert space. Lastly, the expectation value of some appropriate observable $\hat{O}$ is estimated for this state, resulting in a model output

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

In the standard approach of supervised learning, the parameters $\boldsymbol{\theta}$ must be adjusted in order to minimize some loss function $L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} L(\hat{y}_i, y_i)$, as explained in section 2.1.

In the sections to come, we will discuss specific choices of feature encoding, ansätze, and expectation values.

4.1 Feature Encoding

In this section, we will introduce three ways of doing feature encoding, namely *qubit encoding*, *RZZ encoding* and *amplitude encoding*.

4.1.1 Qubit Encoding

One popular approach for encoding a feature vector to a quantum state is the often-called method *qubit encoding*[16]. 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, the encoding works by performing a Pauli-rotation on each qubit with a rotational angle equal to the corresponding (pre-processed) feature. Figure 4.1.1 shows how qubit encoding is implemented using R_x, R_y and R_z rotation.

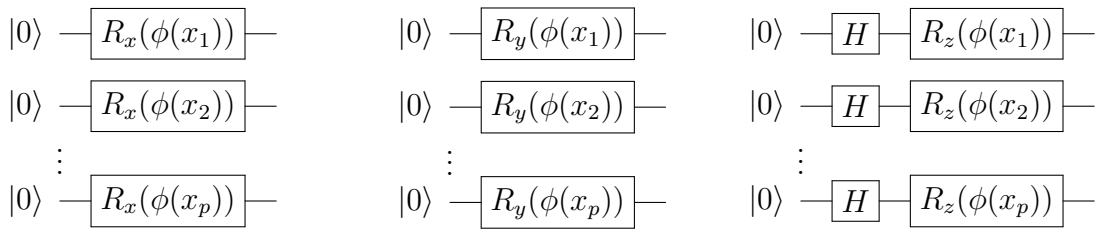


Figure 4.1.1: *Qubit encoding using R_x, R_y and R_z rotations to encode p features, left to right. $\phi(\cdot)$ applies some kind of pre-processing to the samples, e.g. scaling.*

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

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

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 very simple, which may limit the overall expressive power of the final model. In the next subsection, 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 circumstances.

4.1.2 RZZ Encoding

Abbas et al. [8] implemented a quantum feature map for encoding quantum features up to second order, meaning the embedding 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 $i \in [1, \cdots, p-1]$ and $j \in [i+1, \cdots, p]$. 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 4.1.2

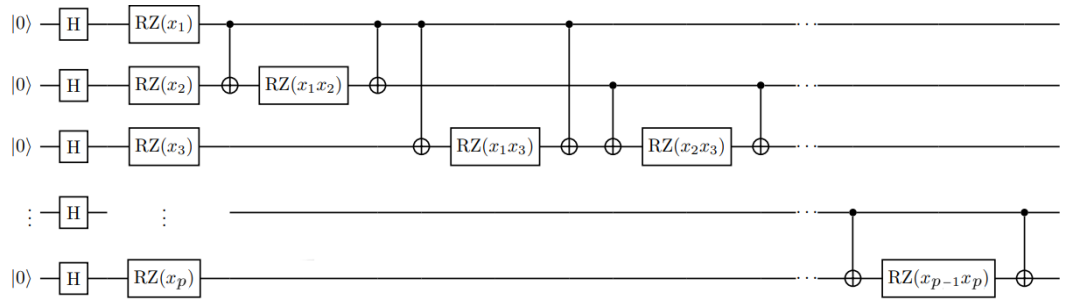


Figure 4.1.2: *Circuit visualizing implementation of RZZ encoding of p features. The figure is retrieved from [8] and adapted to fit our notation.*

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. [8] conjecture that this feature map is difficult to simulate classically for depth ≥ 2 and increasing number of features. Since it requires $\mathcal{O}(p^2)$ operations on a quantum computer, its time complexity is polynomial and hence efficient in comparison. It was showed by the same authors that RZZ encoding produce 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.

4.1.3 Amplitude Encoding

Amplitude encoding is a way of encoding features directly as the amplitudes of a quantum state. Given a data set $\boldsymbol{x} = (x_1, \cdots, x_p)$, where $p = 2^n$, amplitude encoding involves preparing a state $|\psi_{\boldsymbol{x}}\rangle = \sum_{i=1}^{p} x_i |i\rangle$ on n qubits. Because of the normalization of quantum states, it is necessary to scale the data to ensure that $\sum_{i=1}^{p} |x_i|^2 = 1$ holds. The main advantage of amplitude encoding stems from the fact that the number of amplitudes is exponential in the number of qubit. Hence, we only need $\log_2(p)$ qubits to encode p features, enabling us to encode an exponential amount of information as the number of qubits increase. However, there is not a general way of preparing a state for any given $\boldsymbol{x} = (x_1, \cdots, x_p)$ in an efficient way, i.e. with time-complexity polynomial in the number of qubits.

Here, we will present a method for preparing an arbitrary state $|\psi\rangle$ with real-valued amplitudes starting from the initial state $|0\rangle$. The method is based on the description of [1], which is in turn based on the work of [18]. The method prepares the state by iterativly *branching* the state by using multi-controlled rotations. Consider a general system of n qubits. Starting in the state $|00\cdots0\rangle$ the algorithm applies an R_y rotation with rotation angle β_1^1 on the first qubit

$$|\psi_1\rangle = R_y^1(\beta_1^1)\,|000\rangle = a_1\,|00\cdots0\rangle + a_2\,|10\cdots0\rangle\,, \tag{4.5}$$

where the superscript of the gate indicates which qubit it acts on. This creates two different branches where the first qubit is in state $|0\rangle$ and $|1\rangle$, respectively. Then, for each branch, we apply a different rotation on the second qubit. This can be done by conditioning the rotation on the first qubit, resulting in

$$\begin{aligned} |\psi_2\rangle = R_y^{2|q_1=0}(\beta_1^2)R_y^{2|q_1=1}(\beta_2^2)\,|\psi_1\rangle = \\ a_1(b_1\,|00\cdots0\rangle + b_2\,|01\cdots0\rangle) + a_2(b_3\,|10\cdots0\rangle + b_4\,|11\cdots0\rangle). \end{aligned} \tag{4.6}$$

Here, the superscript "$2|q_1 = 0$" means that the gate is applied on the second qubit, conditioned on the first qubit being in state $|0\rangle$ ($q_1 = 0$). In this way, each branch is broken into two new branches. This procedure is continued until all possible branches has been created. In general, we use multi-controlled rotations on qubit i conditioned on all possible states of the preceding qubits, i.e.

$$R_y^{i|q_1,\cdots,q_{i-1}}(\beta_j^i)\,|q_1\cdots q_{i-1}\rangle \otimes |0\rangle\,, \tag{4.7}$$

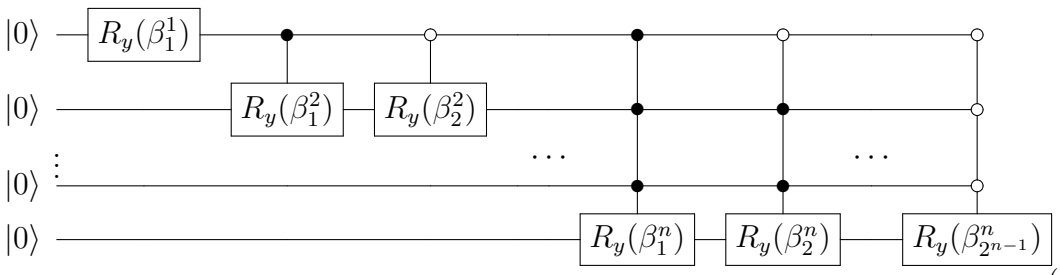
where j enumerates the 2^{i-1} different branches of $|q_1\cdots q_{i-1}\rangle$. To ensure that we arrive at the desired amplitude after all the branches have been created, we must choose rotation angles

$$\beta_j^i = 2\arcsin\frac{\sqrt{\sum_{l=1}^{2^{n-i}}|x_{(2j-1)2^{n-i}+l}|^2}}{\sqrt{\sum_{l=1}^{2^{n-i+1}}|x_{(j-1)2^{n-i+1}+l}|^2}}, \tag{4.8}$$

with the special case for $i = n$

$$\beta_j^n = 2\arcsin\frac{x_{(2j-1)+l}}{\sqrt{\sum_{l=1}^{2}|x_{2(j-1)+l}|^2}}. \tag{4.9}$$

The full procedure is visualized in the following circuit:



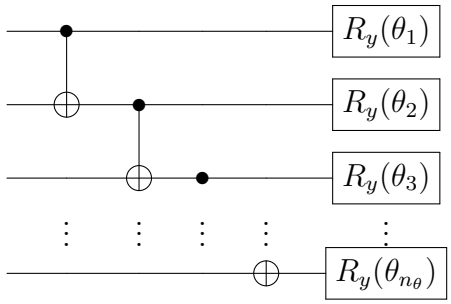
(4.10)

4.2 Ansätze

What kind of unitary transformation are interesting as ansätze used for processing information embedded 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*[19]. In this context, universality means that for any unitary operator, there exists a sufficiently deep ansatz that approximates the operator to an arbitrary accuracy. However, such approximations are often exponentially deep[11], 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 exists reasonably shallow ansatz that are useful for constructing powerful machine learning models. Many of these ansätze are also believed to be classically hard to simulate, eluding to a possible quantum advantage for quantum machine learning[20].

In this thesis, we will investigate an ansatz that respect limitations of near-team quantum computers, focusing on circuit depth that scales linearly with the number of qubits and linear connectivity between qubits. We will refer to this ansatz as the *simple ansatz*(SA). It can be visualised as

$$U_{SA}(\boldsymbol{\theta}) = \quad . \tag{4.11}$$



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 subsection 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, we may repeat the simple ansatz d number of times with independent parameter as such:

$$|\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle = U_{SA}(\boldsymbol{\theta}^r)\cdots U_{SA}(\boldsymbol{\theta}^2)U_{SA}(\boldsymbol{\theta}^1)\,|\psi_{\boldsymbol{x}}\rangle\,, \tag{4.12}$$

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

4.3 Inference

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. $\hat{y} = \langle \psi_{\boldsymbol{x},\boldsymbol{\theta}} | \hat{O} | \psi_{\boldsymbol{x},\boldsymbol{\theta}} \rangle$. In the same manner as [8], we use the *parity* of the state to derive a model output for inference. The parity of an n qubit state can be formulated as an operator as

$$P = \frac{1}{2}(I^{\otimes n} + \bigotimes_{i=1}^{n} \sigma_z). \tag{4.13}$$

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

$$P |v_1 \cdots v_n\rangle = \frac{1}{2}(|v_1 \cdots v_n\rangle - (-1)^{\sum_{i=1}^{n} v_i} |v_1 \cdots v_2\rangle) = \bigoplus_{i=1}^{n} v_i |v_1 \cdots v_n\rangle , \tag{4.14}$$

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 the state is 0 if the number of qubits in state $|1\rangle$ is even, and is otherwise 1. This is called even and odd parity, respectively.

To estimate the expected parity $\langle \psi_{\boldsymbol{x},\boldsymbol{\theta}} | P | \psi_{\boldsymbol{x},\boldsymbol{\theta}} \rangle$, we use the technique described in subsection 3.2.5 by preparing the state repeatedly and measure it in the computational basis. The expected parity can then be estimated as

$$\langle \psi_{\boldsymbol{x},\boldsymbol{\theta}} | P | \psi_{\boldsymbol{x},\boldsymbol{\theta}} \rangle \approx \frac{1}{S} \sum_{j=1}^{S} p_j, \tag{4.15}$$

where S is the number of shots used, and p_j is the parity resulting from measurement ij.

4.4 Optimization of PQC

A key component of hybrid methods such as PQC is the optimization of the parameters $\boldsymbol{\theta}$ entering the ansatz. These parameters are usually optimized with respect to some objective function in order to solve a given problem [16]. In the context of machine learning, one seeks to minimize the loss function Equation 2.1 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:

$$\frac{\partial}{\partial \theta_i} L(\boldsymbol{\theta}) \approx \frac{L(\theta_1, \cdots, \theta_i + \epsilon, \cdots \theta_{n\theta}) - L(\theta_1, \cdots, \theta_i, \cdots \theta_{n\theta})}{\epsilon}, \tag{4.16}$$

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 practise. Recently, analytical techniques have been proved to be very efficient for calculating the gradient on quantum computers [8] [16]. In the next section, we will detail how this gradient can be calculated.

4.4.1 Analytical Gradient-Based Optimization

Based on the derivation presented by [21], 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 $\boldsymbol{\theta}$ that prepares a state $|\psi_{\boldsymbol{\theta}}\rangle = U_{\boldsymbol{\theta}} |0\rangle$. The expectation value of some observable $\hat{O}$ can be formulated as

$$a = \langle\psi_{\boldsymbol{\theta}}| \hat{O} |\psi_{\boldsymbol{\theta}}\rangle = \langle 0| U_{\boldsymbol{\theta}}^{\dagger} \hat{O} U_{\boldsymbol{\theta}} |0\rangle . \tag{4.17}$$

Assume for simplicity that any parameter θ_i affects only a single gate. Then, since any circuit can be decomposed into a sequence of gates, we can decompose the circuit as $U_{\boldsymbol{\theta}} |0\rangle = AG(\theta_i)B$, where G is the only gate dependent on θ_i, and A and B is the rest of the circuit. This allows us to rewrite the expectation value as

$$a = \langle\psi'| G(\theta_i)^{\dagger} \hat{O}' G(\theta_i) |\psi'\rangle , \tag{4.18}$$

where $|\psi'\rangle = B |0\rangle$ and $\hat{O}' = A^{\dagger} \hat{O} A$. Starting from this expression, it is easy to compute the derivative of the expectation value:

$$\partial_{\theta_i} a = \langle\psi'| G(\theta_i)^{\dagger} \hat{O}' (\partial_{\theta_i} G(\theta_i)) |\psi'\rangle + h.c., \tag{4.19}$$

where h.c. refers to the hermitian conjugate. In its current form, the terms of 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

$$\begin{aligned} \partial_{\theta_i} a = \frac{1}{4} (&\langle\psi'| [G(\theta_i) + 2\partial_{\theta_i} G(\theta_i)]^{\dagger} \hat{O}' [G(\theta_i) + 2\partial_{\theta_i} G(\theta_i)] |\psi'\rangle - \\ &\langle\psi'| [G(\theta_i) - 2\partial_{\theta_i} G(\theta_i)]^{\dagger} \hat{O}' [G(\theta_i) - 2\partial_{\theta_i} G(\theta_i)] |\psi'\rangle). \end{aligned} \tag{4.20}$$

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 will 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

$$G(\theta_i) \pm 2\partial_{\theta_i} G(\theta_i) = \underbrace{(I \mp i\sigma_j)}_{\sqrt{2} G(\pm \frac{\pi}{2})} G(\theta_i) = \sqrt{2} G(\theta_i \pm \frac{\pi}{2}), \tag{4.21}$$

where the relation $R_j(a)R_j(b) = R_j(a+b)$ was used in the last step. Inserting this result back into Equation 4.20, we get the final expression

$$\begin{aligned}\partial_{\theta_i} a = \frac{1}{2}(&\langle\psi'|\, G(\theta_i + \frac{\pi}{2})^\dagger \hat{O}' G(\theta_i + \frac{\pi}{2})\,|\psi'\rangle - \\ &\langle\psi'|\, G(\theta_i - \frac{\pi}{2})\hat{O}' G(\theta_i - \frac{\pi}{2})\,|\psi'\rangle).\end{aligned} \tag{4.22}$$

The form of the above expression 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 a very 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 $\boldsymbol{x}$ enter the state $|\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle$ in the same way as the parameters $\boldsymbol{\theta}$ if qubit encoding is used(see subsection 4.1.1), 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 features, i.e. $\partial_{x_i} a$. This will be relevant when we later introduce models consisting of multiple circuits.

4.4.2 Barren Plateus in QNN Loss Landscape

While recent studies have shown several promising characteristics of QNNs, such as faster training and greater flexibility[8], these studies have been largely focused on smaller systems and heuristic measures. As such, few scaling relations of QNNs have been rigorously proven. Recently, McClean et al. [22] established an important result connecting the magnitude of the gradient for a large class of PQCs to the number of qubits. They found that the variance of the output of randomly initialized PQCs vanish exponentially in the number of qubits, with a worsening effect the deeper the circuit is. This is because the PQC essentially approached a random circuit, leading to a concentration of outputs around an the average output over all possible initialization. In particular, it was showed that the gradient of PQCs center around zero, meaning the gradient vanishes exponentially fast as the circuits grows wider and deeper.

The vanishing of PQCs gradients manifests itself as loss landscapes that are extremely flat in most of parameter space, reminiscent of the vanishing gradient phenomenon of classical neural networks as the depth increase[23]. The exponential vanishing of the gradient means that exponentially many shots are required in order to obtain a sufficient signal-to-noise ratio, as explained in subsection 3.2.5. This may render the training of larger QNN models intractable as the number of qubits is increased in order to handle harder learning problems.

4.5 Quantum Circuit Network

In order to extend the QNN framework discussed so far, we will implement multi-circuit models that utilize several such QNNs. These models exhibit a network-like structure, consisting of layers of several circuits. The layers of circuits transform feature vectors in a sequential manner until a model prediction is obtained. To avoid confusion with "quantum neural networks", we have opted to call the multi-circuit model a *quantum circuit network*(QCN). This type of architecture was explored by **b** and found to be able to sufficiently fit a nonlinear function in one dimension when optimized with Nelder-Meads algorithm, a gradient-free optimization algorithm.

4.5.1 Feed-Forward

Figure 4.5.1 illustrates the general structure of a quantum circuit network, which exhibits 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 $\boldsymbol{\theta}^{[l,j]}$, 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 $\boldsymbol{a}^{(l-1)}$ resulting from the previous layer(with the special case that $\boldsymbol{a}^{(0)} = \boldsymbol{x}$). For each node j, an output $a_k^{(l)}$ is produced, i.e.

$$a_j^{(l)} = f_{QNN}^{(l)}(\boldsymbol{a}^{(l-1)}; \boldsymbol{\theta}^{[l,j]}) \tag{4.23}$$

These outputs are then concatenated to make a new feature vector $\boldsymbol{a}^{(l)}$. This is then repeated for each layer 1 through L. Finally, the output of the last is identified as the model output

$$\hat{y} = f_{QCN}(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{a}^{(L)}, \tag{4.24}$$

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

The sequential transformation resulting from the multi-circuit architecture of QCNs offer several interesting properties compared to the single circuit QNN models. First, as the circuits are evaluated one at a time, it is possible to compute very large models of many parameters without the need for quantum computers that can sustain a very long coherence time. While QNNs of many paramaters generally have high circuits depths and consist of many qubits, QCN can on the other hand consist of many more shallow circuits of few qubits, much more suitable for near-term quantum computers.

Further, the repeated embedding and estimation of outputs of intermediate layers may alter the space of functions possible to compute in an interesting way. This is because each of these steps introduce a nonlinear transformation of the data. For classical neural networks, we know from section 2.3 that repeated nonlinear transformation is key for their great expressive power. Switching over to the single-circuit QNN model, nonlinearity is introduced only twice: First when the features are encoded as a quantum state using rotations such as $R_j(x_i)\,|0\rangle = \cos\left(\frac{x_i}{2}\right)|0\rangle -$

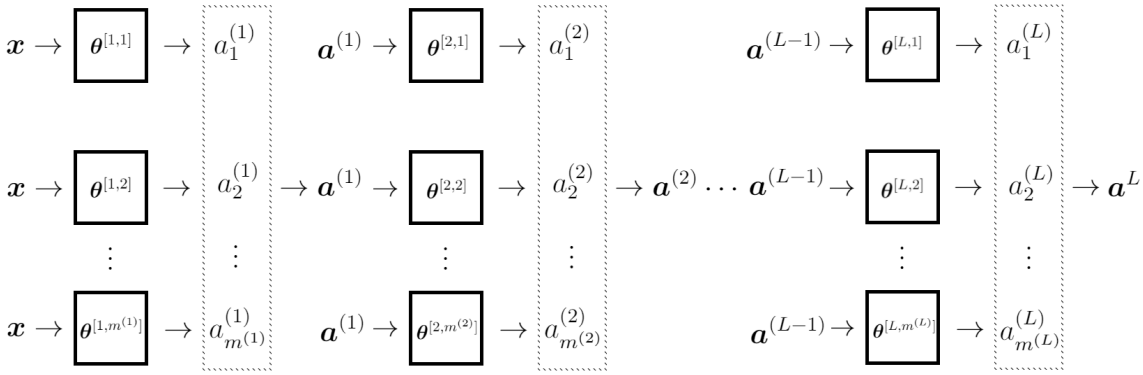


Figure 4.5.1: *General structure of a quantum circuit network. Each node, indicated by a box, is a QNN model parameterized by* $\boldsymbol{\theta}^{[l,j]}$*, where* l *is the layer and* j *is the node in the current layer.* $m^{(l)}$ *is the number of nodes in layer* l*. For layer* l*, each node receives the feature vector* $\boldsymbol{a}^{(l-1)}$ *resulting from the previous layer(with the special case that* $\boldsymbol{a}^{(0)} = \boldsymbol{x}$*). For each node* j*, an output* $a_j^{(l)}$ *is produced, which is then concatenated to make a new feature vector* $\boldsymbol{a}^{(l)}$*. This is then repeated for each layer* 1 *through* L*. Finally, the output of the last layer is identified as the model output* $\hat{y} = \boldsymbol{a}^{(L)}$*.*

$i \sin\left(\frac{x_i}{2}\right)\sigma_j |0\rangle$. The second time is when an output is derived by estimating an expectation value, which relates to the modulo square of the amplitudes of the state, i.e. $|\alpha_k|^2$. All other processing of the data is performed by the ansatz, which by definition is unitary and therefore linear in nature. In this sense, it could be that the formulation of QNNs as shown in Figure 4.0.1 could be a bit constrained with respect to what functions it can compute. By combining several circuits, nonlinearity is introduced with each layer, possibly creating a more powerful model able to fit more complicated functions.

4.5.2 Backward Propagation

Comparing the formulation of QCN Equation 4.24 and classical neural network Equation 2.9, it can be seen that they are structurally the same down to the mathematical operations happening inside each node. For the QCN each node implements a QNN model, while the classical neural network implements an affine transformation, followed by a nonlinear activation. Using this oberservation, it is possible to implement a slightly modified backpropagation, earlier described in subsection 2.3.2, for QCN. This assumes that we are able to calculate the derivative of the outputs of each QNN model. As described in section 2.1, this is indeed possible using the parameter shift rule.

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

$$\delta_k^L = \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^L}, \tag{4.25}$$

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

$$\delta_j^l = \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_j^l} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^{l+1}} \frac{\partial \boldsymbol{a}_k^{l+1}}{\partial \boldsymbol{a}_j^l} = \sum_k \delta_k^{l+1} \frac{\partial \boldsymbol{a}_k^{l+1}}{\partial \boldsymbol{a}_j^l}. \tag{4.26}$$

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

$$\frac{\partial L(\hat{y}, y)}{\partial \theta_n^{[l,j]}} = \sum_k \frac{\partial L(\hat{y}, y)}{\partial \boldsymbol{a}_k^l} \frac{\partial \boldsymbol{a}_k^l}{\partial \theta_n^{[l,j]}} = \delta_j^l \frac{\partial \boldsymbol{a}_j^l}{\partial \theta_n^{[l,j]}}, \tag{4.27}$$

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

The terms

$$\frac{\partial \boldsymbol{a}_j^l}{\partial \theta_n^{[l,j]}}, \frac{\partial \boldsymbol{a}_k^{l+1}}{\partial \boldsymbol{a}_j^l} \tag{4.28}$$

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 subsection 4.4.1, the local gradients can be calculated analytically using the parameter shift rule with respect to the parameters $\theta_n^{[l,j]}$ and the inputs $\boldsymbol{a}_j^l$. By first performing a forward pass to calculate $\boldsymbol{a}^l$ for all the layers l, the local gradients can then estimated and stored one at a time. Finally, Equation 4.27 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 $\boldsymbol{x}$. Repeating this for all samples $\boldsymbol{x}^{(i)}$, we can calculate the average total gradient

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\hat{y}^{(i)}, y^{(i)}). \tag{4.29}$$

The calculation of the gradient allows us to leverage more information of the loss function and enables for gradient-based optimisation. Potentially, this could mean faster optimization relative to derivative-free optimization, such as Nelder-Mead's algorithm which was originally used to optimize QCN(kilde). However, as explained in subsection 2.2.1, gradient-based methods such as gradient descent are prone to getting stuck in local minima, potentially causing slow optimization.

4.6 Recursive Circuit Optimisation

5

Tools for Analysis

5.1 Trainability

In machine learning, *trainability* refers to how easily a particular model can be trained under different conditions [8]. A common way of exploring the trainability is by exploring the geometry of the loss landscape. For example, the loss function of dense neural networks exhibit local flatness for most directions in parameter space, and strong distortion in others [24]. In a loss landscape that is mostly flat, the gradient of the model will tend to diminish, known as *vanishing gradient*, making it difficult to train the model using gradient-based methods. This is also 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 subsection.

5.1.1 Hessian Matrix

Let $f(\boldsymbol{x}^{(k)}; \boldsymbol{\theta})$ be a parameterized and differentiable model, where $\boldsymbol{x}^{(k)} \in \mathbb{R}^p$ are p features, and $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$ are n_θ model parameters. For a general loss function on the form Equation 2.1, $L(\boldsymbol{\theta}) = \sum_{k=1}^{N} L(f(\boldsymbol{x}^{(k)}; \boldsymbol{\theta}), y^{(k)})$, where N is the number of samples in the data set, the hessian matrix of the loss function is given by

$$H_{ij} = \frac{\partial^2 L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_i \partial \boldsymbol{\theta}_j}. \tag{5.1}$$

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 a much-studied quantity in the machine learning community, and it has been used to study the loss landscape both for classical and quantum mechanical machine learning models [25] [26]. In particular, it's 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")[25]. This indicated the that loss landscape is mostly flat, with huge distortions in a few directions, which in turn causes slow optimization as discussed earlier.

5.1.2 Empirical Fisher Information Matrix

An apparent shortcoming of the Hessian matrix Equation 5.1 is the huge 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)[24], can be calculated using $\mathcal{O}(n_\theta)$ first order derivatives, which is much more suited for big models. We will now derive the EFIM and relate it to the Hessian.

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

$$H_{ij} = F_{ij} - \frac{1}{N}\sum_{k=1}^{N}(y^{(k)} - f(\boldsymbol{x}^{(k)};\boldsymbol{\theta}))\frac{\partial^2 f(\boldsymbol{x}^{(k)};\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_i\partial\boldsymbol{\theta}_j}, \tag{5.2}$$

where F is identified as the EFIM, given by

$$F_{ij} = \frac{1}{N}\sum_{k=1}^{N}\frac{\partial f(\boldsymbol{x}_k;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_i}\frac{\partial f(\boldsymbol{x}_k;\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_j}. \tag{5.3}$$

From Equation 5.2, the EFIM can been seen to coincide with the Hessian matrix if $f(\boldsymbol{x}^{(k)};\boldsymbol{\theta}) = y^{(k)}$, since the terms in the last sum vanishes. This is the case if the model manage to perfectly replicate the targets from the inputs, which is approximately true for well-trained models what 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 [24] [8]. 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 mathematical 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 the model affects the space of functions it can compute. Put more simply, expressivity measures how flexible and complex the model is. The first attempts to measure expressivity of neural networks took a highly theoretical approach, such as Bartlett, Maiorov, and Meir [27] calculating of the VC dimension of shallow neural networks. The VC dimension, or *Vapnik–Chervonenkis* dimension[2], is a well established measure of complexity. However, it is know to be hard to compute in practise for a variety of models[8].

5.2.1 Trajectory Length

In order to explore the expressivity of deep neural networks, Raghu et al. [28] 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 grows as it is passed through the various layers of the model.

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

$$l(\boldsymbol{x}(t)) = \int_t \left\| \frac{\boldsymbol{x}(t)}{dt} \right\| dt \tag{5.4}$$

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

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

By making an appropriate trajectory $\boldsymbol{x}(t_i)$ 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 $l(\boldsymbol{a}^l(t_i))$, where $\boldsymbol{a}^l(t_i)$ are the outputs of layer l resulting from the input $\boldsymbol{x}(t_i)$ for some neural network. As an example, one can make a trajectory $\boldsymbol{x}(t_i) \in \mathbb{R}^2$ in the shape of a circle. By projecting $\boldsymbol{a}^l(t_i)$ down to 2D, it is possible to visualize how each layer of the neural network distorts the trajectory. This has been exemplified in Figure 5.2.1.

Figure 5.2.1: *Picture 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. [28].*

Figure 5.2.1 shows that the inputs gets transformed in a highly non-linear way as it is being transformed by each layer. Especially, neighboring points in the input trajectory gets mapped further and further apart for each transformation, indicating that small perturbations in the input grows for each layer. Raghu et al. [28] showed that the trajectory length of trained neural networks increase exponentially with depth, suggesting a capacity to compute exponentially complex functions as the number of layers increase. On the other hand, randomly initialized neural networks was shown to not exhibit this exponential growth regime.

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 elaborate frame work for doing machine learning, capable of implementing dense neural networks(DNN, Equation 2.9) and quantum circuit networks(QCN, Equation 4.24). In addition, various numerical tools for analysing the models are available. The code base is object-orientated using Python, focusing on flexibility. This grants huge 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 QNC layers. To implement quantum machine learning, the frame work is built around Qiskit[29], 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**.

6.1 Qiskit

Qiskit[29] 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 among the import 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:

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

Here, each of the registers **q_reg_1** and **q_reg_2** contains two qubits. By default, they are each initialized in the state $|0\rangle$. Thus, total circuit can be written as

$$|00\rangle\,|00\rangle\,, \tag{6.1}$$

where is *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 may 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:

```
circuit.h(q_reg_1[0])
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**:

```
circuit.cx(q_reg_1[0], q_reg_2[0])
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()**:

```
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*:

```
job = qk.execute(circuit,
                 backend =
                 qk.Aer.get_backend("qasm_simulator"),
                 shots = 1000)
result = job.result().get_counts(circuit)
print(result)
```

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

Resulting from this is a python dictionary whose keys are strings indicating the different states that was 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 being able to access state resulting from some computation directly and calculate exact expectation values. As explained in subsection 3.1.3, this is not possible when using real quantum computers. Have 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:

```
job = qk.execute(circuit,
                 backend =
                 qk.Aer.get_backend("statevector_simulator"))
result = job.result().get_counts(circuit)
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

6.2 QNN Example

We will now present a possible way of implementing a QNN Equation 4.3 for fitting the popular Iris data set(kilde). The model will be optimized using the parameter shift rule and gradient descent.

6.2.1 Encoding

To encode the data, 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 subsection 4.1.1, can be implemented as the following function:

```
def qubit_encoder(circuit, data_reg, data):
    for i, x in enumerate(data):
        circuit.rx(x, data_register[i])

    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, reps)**, where **theta** is a numpy array containing the parameters. The "simple ansatz" detailed in section 4.2 can be implemented as a python function:

```
def simple_ansatz(circuit, data_reg, theta):
    n_qubits = data_reg.size()

    for i range(n_qubits - 1):
        circuit.cx(data_reg[i], data_reg[i+1])

    for i, w in enumerate(theta):
        circuit.ry(w, data_register[i])

    return circuit
```

Here, **data_reg.size()** was used to retrieve the number of qubits present in the register. The first for-loop ensures to entangle 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 Inference

To derive an inference from the model, we must estimate an expectation value as explained in section 4.3. 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 subsection 6.1.3 and subsection 6.1.4. We can implement a function estimating the parity(see section 4.3) in the following way:

```
def parity(counts):
    shots = sum(counts.values())

    output = 0
    for bitstring, samples in counts.items():
        if parity_of_bitstring(bitstring) == 1:
            output += samples

    output = output / shots

    return output
```

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

```
def parity_of_bitstring(bitstring):
    binary = [int(i) for i in bitstring]
    parity = sum(binary) % 2

    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 performe the whole inference, 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

```
def qnn(x, theta):
    n_qubits = len(x)
    reps = 2
    data_reg = qk.QuantumRegister(n_qubits)
    clas_reg = qk.ClassicalRegister(n_qubits)
    circuit = qk.QuantumCircuit(data_reg, clas_reg)

    circuit = qubit_encoder(circuit, data_reg, x)
    circuit = simple_ansatz(circuit, theta, reps)

    job = qk.execute(circuit,
                     backend =
                     qk.Aer.get_backend("qasm_simulator"),
                     shots = 1000)
    counts = job.result().get_counts(circuit)
    y_pred = parity(counts)

    return y_pred
```

6.2.4 Gradient

Having a function that implements a QNN that performs inference, we can apply the parameter shift rule described in subsection 4.4.1 to calculate the gradient of the output with respect to the parameters:

```
def gradient(x, theta):
    deriv_plus = np.zeros(len(theta))
    deriv_minus = np.zeros(len(theta))

    for i in range(len(theta)):
        theta[i] += np.pi/2 #parameter shifted forward
        deriv_plus[i] = qnn(x, theta)

        theta[i] -= np.pi   #parameter shifted backwards
        deriv_minus[i] = qnn(x, theta)

        theta[i] += np.pi/2 #parameter reset

    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 2.14 resulting from the samples and targets **x_list** and **y_list**, using MSE loss. The function then updates the parameters **theta** iterativly(**epochs** number of times) 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)       #inference
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 += (y_pred - y)*gradient(x, theta)** accumulates the gradient of the MSE loss function with respect to the parameter, i.e. Equation 2.4. The parameters are then updated at line 14.

6.2.6 Putting It All Together

Stuff

6.3 Quantum Circuit Network

Stuff

6.3.1 Encoders, Ansätze and Samplers

In the QCN framework, we implement encoders, ansätze and samplers as callable python classes with much the same functionality as described in section 6.2. We will now go thorugh the use of the most important classes.

QubitEncoder

The encoder class **QubitEncoder** can be instantiated as

```
from encoders import QubitEncoder

encoder = QubitEncoder(mode)
```

Here, **mode** is a string that specifies the rotation used for encoding, either **"x"**, **"y"** or **"z"**. See subsection 4.1.1 for details.

Ansatz

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

```
from ansatzes import Ansatz

ansatz = Ansatz(block, reps)
```

Here, **block** is a python list containing strings that specify gates that are 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 4.2. **reps** specifies the number of times the ansatz is then repeated.

Parity

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

```
from samplers import Parity

sampler = Parity()
```

This class implements the same functionality as the **parity** function described in section 6.2.

6.3.2 QLayer

Our framework for making QCN models implements layers consisting of QNN models as nodes, as explained in section 4.5. In the framework, a QCN layer can be created as a python object of the type **QLayer** in the following way:

```
from layers import QLayer

qlayer = QLayer(n_qubits,   #number of qubits in each QNN
                n_features, #number of input features
                n_targets,  #number of outputs, i.e. nodes
                scale,      #scaling of output
                encoder,
                ansatz,
                sampler,
                backend,
                shots)
```

The arguments **encoder**, **ansatz** and **sampler** defines the architecture of each QNN in the layer. Examples of possible choices are described in subsection 6.3.1.

As an example, a **QLayer** can be instantiated and used on a data in the following way:

```
import qiskit as qk
from layers import QLayer
backend = qk.Aer.get_backend("qasm_simulator")

x = np.random.normal((4,3))

qlayer = QLayer(n_qubits = 3,
                n_features = 3,
                n_targets = 2,
                scale = 2*np.pi,
                encoder = QubitEncoder(mode = "x"),
                ansatz = Ansatz(blocks=["entangle", "y"],
                reps = 2),
                sampler = Parity(),
                backend = backend,
                shots=1000)

y_pred = qlayer(x)
print(y_pred)
```

(skriv output her)

Here, **x** is a dataset containing four samples of three features each. By specifying **n_targets=2**, the layer consists of two nodes and produces thus two output targets. The layer is callable, and performs inference 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 subsection 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:

```
from neuralnetwork import NeuralNetwork

x = np.random.normal(0, 1, (4,3))

#unspecified arguments assumes default values
layer1 = QLayer(n_qubits=3,
```

```
                  n_features=3,
                  n_targets=4)

layer2 = QLayer(n_qubits=4,
                  n_features=4,
                  n_targets=1)

network = NeuralNetwork(layers = [layer1, layer2],
                        cost = MSE(),
                        optimizer = Adam(lr=0.1))
y_pred = network.predict(x)
print(y_pred)
```

In the above code, the **NeuralNetwork** class stores the layer objects in a python list **self.layers**. When doing inference, the class implements feed forward, as described in subsection 4.5.1, using a **__call__** method:

```
def __call__(self, x):
        self.a = []
        self.a.append(x)
        for layer in self.layers:
            x = layer(x)
            self.a.append(x)
```

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

6.3.4 Back Propagation

The class **NeuralNetwork** performs back propagation, as described in subsection 4.5.2, using a class method **network.backward(x,y)**. In simplified terms, the method is implemented as:

```
def backward(self, x, y):
    self(x)                  #feed forward
    y_pred = self.a[-1]      #inference
    delta = self.cost.derivative(y_pred, y)

    #work thru layers in reverse
    for i, layer in reversed(list(enumerate(self.layers))):
        weight_gradient, delta = layer.grad(self.a[i],
                                            delta)
        self.weight_gradient_list.append(weight_gradient)

    self.weight_gradient_list.reverse()
```

network.backward(x,y) starts by performing feed forward and inference. In line 4, the error of the last layer Equation 4.25 is initiated as **delta**. For each

layer, starting with the last first, the gradient is calculated and the error **delta** is updated. This is done using Equation 4.27 and Equation 4.26, 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

```
def train(self, x, y):
    self.loss = []
    for i in range(epochs):
        self.backward(x, y)
        self.step()

        y_pred = self.a[-1]
        self.loss.append(self.cost(y_pred, y))

    y_pred = self.predict(x)
    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 2.6, 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:

```
layer = QLayer(n_qubits = 4,
               n_features = 4,
               n_targets = 1,
               encoder = RZZEncoder(),
               ansatz = Ansatz(blocks=["entangle", "ry"],
                               reps=2),
               sampler = Pairty(),
               backend = backend,
               shots = 1000)


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 subsection 4.4.1 and subsection 6.2.4, fails. Possible choices is RZZ encoding(as used in the above code) and amplitude encoding.

6.3.7 Hybrid Networks

In addition to **QLayer** layers, the neural network framework also implements **Dense** layers, which are densely connected, classical layers as defined in section 2.3. It can be instantiated in the following way:

```
from layers import Dense, Sigmoid

dense = Dense(n_features = 4,
              n_targets = 3,
              scale = 1,
              activation = Sigmoid(),
              bias = True)
```

Here, **activation** specifies the activation 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. 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 of subsection 6.3.5.

6.4 Tools for Analysis

6.4.1 Magnitude of Gradient

6.4.2 Empirical Fisher Information

To investigate the loss landscape of the QCN model, we calculate the EFIM Equation 5.3 and its eigen value spectrum using a class **EFIM**. Given some **network** and data set **x**, it can be used in the following way:

```
from analysis import EFIM

efim = EFIM(network)
efim.fit(x)
```

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

```
def fit(self, x):
    n_samples = x.shape[0]

    self.model.backward(x, samplewise=True)
    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,
        axis=1)
12
13      self.fim = 1 / n_samples * gradient_flattened.T @
        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_θ), 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.efim**

After calculating the EFIM, its eigenvalue spectrum can be calculated as

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

This performs a simple eigenvalue decomposition of **self.efim** using numpy's **linalg.eigh** to extract the eigenvalues. Because the EFIM is often very 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 $\boldsymbol{x}(t_i)$, as described in subsection 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 a two python list: **traj_len** contains the trajectory length of the layer outputs. **traj_proj** contains the layer outputs themselves, projected down onto 2D for visualization. The projection was done using scikit-learn's(kilde) PCA decomposition. (Describe code?)

Part III

Results & Discussion

7

Results and Discussion

In this chapter, we will detail how various models presented in previous chapters were configured and trained. The models will also be characterized using aforementioned numerical methods, and the results will be compared to earlier research.

The vanishing gradient phenomenon will be investigated for QNNs, QCNs and DNNs by studying the magnitude of their gradients (see subsection 4.5.2 and subsection 2.3.2). This will be done for different architectures, especially different number of layers.

We will characterize the geometry of the loss landscape of QCNs(see section 4.5) and other models by studying the EFIM presented in subsection 5.1.2. The result will be used to asses the trainability of different models and predict how architecture affects training.

To assess the expressivity of QCNs and compare them to other models, we will use trajectory length as presented in subsection 5.2.1. This will be done for both trained and untrained models.

In order to test models in a practical setting, and give support to previous results and analyses in this thesis, the models will be trained to fit gaussian data. This will be done both using idealised simulation(see subsection 6.1.4), and simulated, noisy hardware(see).

7.1 Vanishing Gradient Phenomenon

7.1.1 Vanishing Gradient in QNNs

We start by investigating the magnitude of the gradient for QNNs for different number of qubits and repetitions of the ansatz. To construct the QNNs, we use qubit encoding with R_x rotations together with the simple ansatz. To derive an output, we estimate the parity of the state exactly using the methods outlined in see subsection 6.1.4 and section 4.3. Instead of investigating the gradient of some loss function, i.e. Equation 2.4, we will investigate the gradient of the model output itself, $\frac{\partial f(\boldsymbol{x}^{(i)};\boldsymbol{\theta})}{\partial \theta_j}$. If this quantity vanishes, it follows that also Equation 2.4 vanishes, for any loss function and labels $y^{(i)}$. To be concrete, we seek to compute the magnitude of the gradient averaged over the parameters and samples, i.e.

$$\frac{1}{Nn_\theta}\sum_{i,j}\left|\frac{\partial f(\boldsymbol{x}^{(i)};\boldsymbol{\theta})}{\partial \theta_j}\right| \tag{7.1}$$

To get a result representative of a realistic feature space, we will sample features $\mathcal{X} = \{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(N)}\}$ uniformly as $\mathcal{X} \sim U(-\frac{\pi}{2}, \frac{\pi}{2})^{[N,p]}$. Here, we will use $N = 100$ samples, and the number of features p will be set equal to the number of qubits for each QNN. The QNNs will also be 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

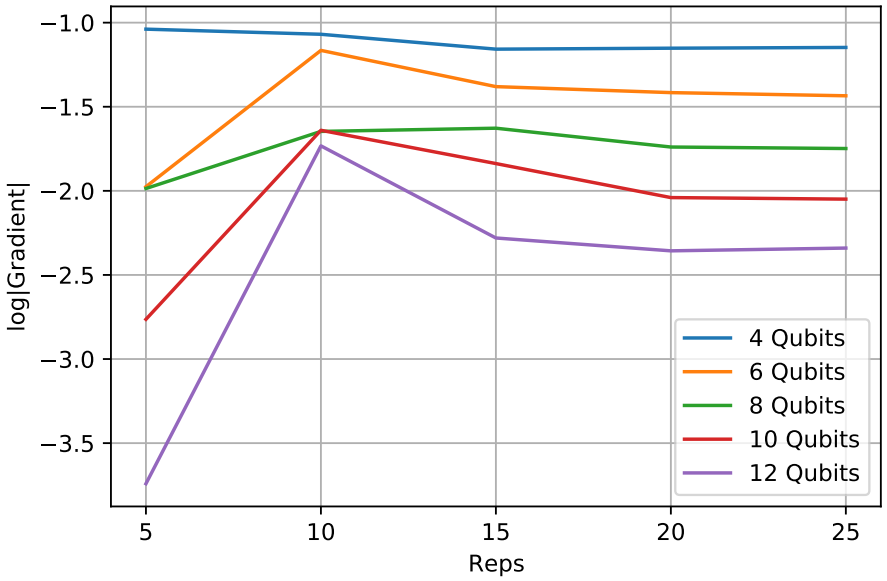


Figure 7.1.1

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 worse the more repetitions of the ansatz is used.

This behaviour can be explained by the fact that QNNs are a special case of PQCs. By encoding random inputs and randomly initializing the parameters, the QNN approaches essentially a random circuit. As shown by McClean et al. [22], randomly initialized PQCs tend to produce gradients closely centered around zero as the number of qubits are increased. This is also consistent with our analysis of QNNs. As explained in subsection 4.4.2, it is necessary to use exponentially many shots in order to obtain a good signal-to-noise ratio for wide circuits, which in turn may make training intractable.

7.1.2 Vanishing Local Gradient in QCNs

A potentially promising feature of QCNs is the ability to scale up the model by introducing more circuits, rather than wider and deeper ones. As the gradient of QNNs tend to vanish for a high number of qubits, we want to investigate how the gradients of smaller QNNs behave when they enter as nodes in a QCN architecture.

The QNNs used to contruct QCNs here are set up and initialized in the same manner as in subsection 7.1.1, with two repetitions of the simple ansatz. Also, we sample input data in the same manner, i.e. uniformly as $\mathcal{X} \sim U(-\frac{\pi}{2}, \frac{\pi}{2})^{[N,p]}$, with $N = 100$ and the number of features p set to the number of qubits used in the QNNs. In this section, all QCNs have 8 hidden layers an. Each hidden layer will utilize the number of nodes, d, which will range from 4 to 8. Also, the number of qubits in each node will be set to d as well. 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. Figure 7.1.2 shows the magnitude of the local gradients Equation 4.28 of the nodes, averaged over each layer, the samples and 10 different realizations of the models. In addition, the standard deviation of this magnitude is estimated over the different realizations, yielding a confidence interval as seen in the figure.

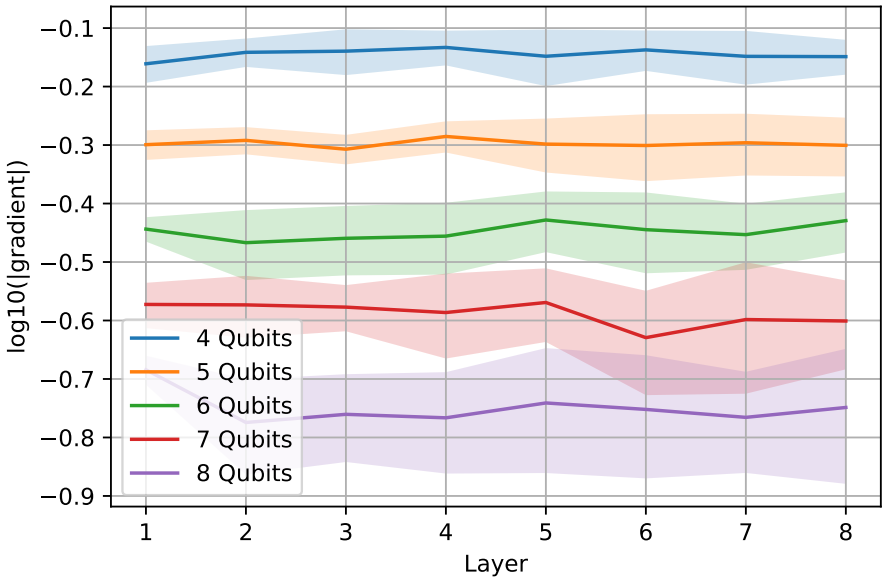


Figure 7.1.2

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 QNNs entering the QCN model tend to vanish exponentially in the number of qubits, as with the single-circuit QNNs seen in Figure 7.1.1. However, the relative position of the QNNs along the depth of the QCN does not seem to affect the magnitude. This can be seen from the overlapping confidence intervals of the magnitudes for each number of qubits. Any variation of the average magnitude between the layers is thus likely just noise induced by the low number of 10 parameter realizations, and does not indicate significant differences. This suggests that QCNs can be scaled up by making them deeper, with a computational cost linear in the number of layers, 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 single-to-noise ratio.

7.1.3 Vanishing Total Gradient in QCNs

In the previous section, it was shown that scaling up QCNs by adding more layers did not effect the number of shots needed to sufficiently estimate the local gradients of each QNN. This is in opposition to single-QNN models, whose gradient vanishes as both the number of qubits and repetitions are increase, as seen in Figure 7.1.1. However, we need to investigate how the magnitude of the total gradient Equation 4.27 behaves as the number of layers is increased. In this section, the total gradient is calculated using the local gradients from the same numerical experiment as in subsection 7.1.2. As previously, the magnitude of the total gradient is averaged over each layer, the samples and 10 realisations of the parameters. Figure 7.1.3 shows the average total gradient QCNs for each layer, for

different number of qubits. For comparison, it also shows the magnitude of the total gradient of a DNN with the same number of layers and similar number of parameters as the biggest QCN.

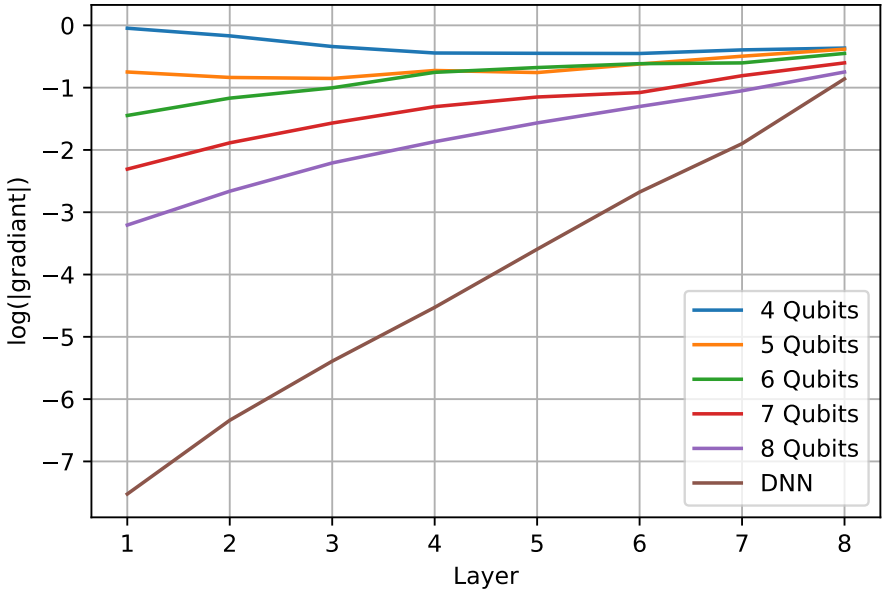


Figure 7.1.3

From Figure 7.1.2, we see that the total gradient for a given layer of the DNN tends to vanish exponentially in the number of layers after it. This is a well-known phenomenon for classical neural networks, often explained by the saturation of the activation function during feed forward[23]. As the activations tend to be very flat for when saturated, as for the sigmoid function used here, the gradient tends vanish during back propagation.

As with the DNN, also the QCNs exhibits vanishing total gradient with increasing number of layers, with a strong dependence on the number of qubits in each node. Seen in Figure 7.1.2, the total 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 of the QCN is propagated backwards using Equation 4.26, it accumulates the local gradients $\frac{\partial \boldsymbol{a}_k^{l+1}}{\partial \boldsymbol{a}_j^l}$ as factors. In the case that these factors are large, the error will tend to decrease slowly, and hence also the total gradient by Equation 4.27. This is the case for architectures with few qubits per node, as discussed in subsection 7.1.2. However, as the number of qubits increase, the local gradients will tend to decrease. Accumulating small factors will causes the error to decrease faster, exponentially so for each layer. In a sense, this vanishing of local gradients with increasing number of qubits is analogous to the saturation of the activations for classical networks.

Even though the magnitude of local gradients of QCNs tends to stay constant in the number of layers, we saw that back propagation induce an exponentially vanishing gradient. Although this behaviour is similar to that of DNNs, it is not as severe for the architectures used in this section: For the largest QCN of 100 parameters, the magnitude of the total gradient for the first layer is of order 10^{-3}, as opposed to $10^{-7.5}$ for the DNN. Obviously, this difference will tend to decrease when increasing the number of qubits.

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. This is true, however, for single-QNN models as discussed in subsection 4.4.2. Put another way, QCNs' use of several smaller circuits, rather than one big, moves the estimation of vanishing quantities (the gradient) from quantum expectation values to classical computation.

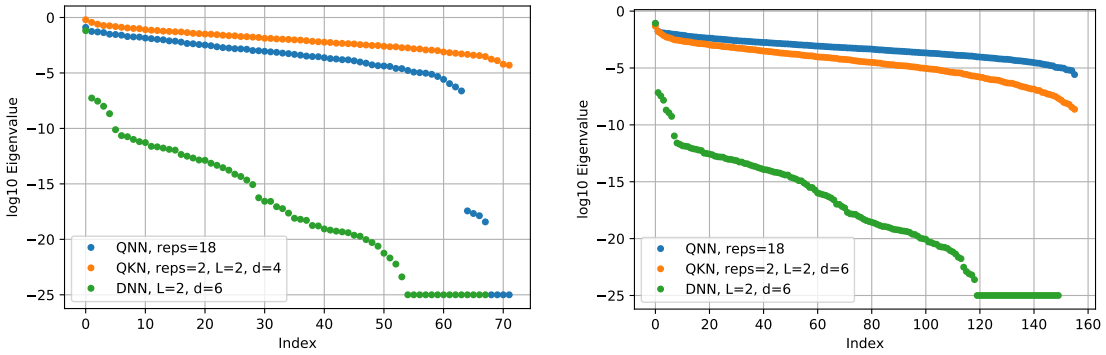
7.2 Investigating the Loss Landscape

We explore the geometry of the loss landscape of various models by studying the eigenvalue spectrum of the EFIM. Looking Equation 5.3, we see that the EFIM, unlike the Hessian, is independent of targets $y^{(i)}$. This makes the analysis problem independent, and serves to characterises architectures only. The input data $\mathcal{X} = \{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(N)}\}$, used for calculating the EFIM, is sampled randomly from a standard normal distribution as $\mathcal{X} \sim N(0,1)^{(N,p)}$. This ensures that the input is evenly sampled from feature space and is consistent with the analysis of Abbas et al. [8]. Here, we use $N = 200$ samples and either $p = 4$ or $p = 6$ inputs, depending on the model. For each unique model, the EFIM is calculated 10 times for different random initializations of the parameters(see stuff). The resulting spectrum is then averaged over the 10 initializations to produce more significant results. For a complete description of the models analysed in this section, see Table 7.2.1.

Model	Type	Qubits	Reps	Layers	Width	Encoder	n_θ
A	QNN	4	18	1	4	RZZ Encoding	72
B	QCN	4	3	2	4	Qubit Encoding	60
C	QCN	4	2	3	4	Qubit Encoding	72
D	QCN	4	1	5	4	Qubit Encoding	68
E	DNN	NA	NA	3	6	NA	79
F	QNN	6	26	1	6	RZZ Encoding	156
G	QCN	6	4	2	6	Qubit Encoding	168
H	QCN	6	2	3	6	Qubit Encoding	156
I	QCN	6	1	5	6	Qubit Encoding	150
J	DNN	NA	NA	3	9	NA	163

Table 7.2.1: *Description of the architecture of the models analysed in this section. The QNN and QCN models use exact evaluation of parity to derive outputs (see subsection 6.1.4 and section 4.3). The DNN models uses sigmoid activation in all layers. The parameters of the models are appropriately initialized as presented in (stuff).*

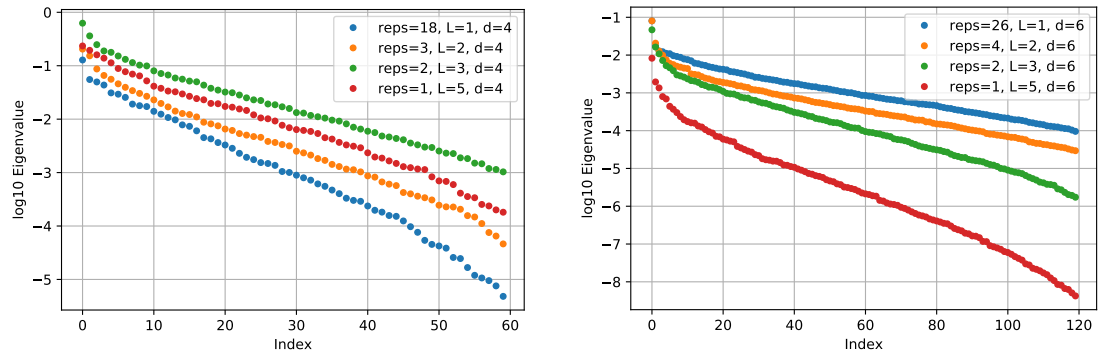
Figure 7.2.1 compares the EFIM spectrum of QNNs, QCNs and DNNs. Their architectures are chosen so that the models have approximately equal number of parameters. This is to ensure a fair comparison. Looking at the spectra of the DNN models, we see the characteristic result of a singular large eigenvalue, with the rest sitting close to zero. This indicates that DNN models exhibit a loss landscape that is very flat in all but one direction, where it is extremely distorted. This result is consistent with the findings of Abbas et al. [8] and Karakida, Akaho, and Amari [24]. As explained in section 2.2, this geometry of the loss landscape is often associated with slower optimization. Further, we see that the spectra of the our QNN models 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. Abbas et al. [8] came to the same conclusion for their QNN models, and argued that this uniformity of the spectrum meant that landscape was more well-condition for optimization. They strengthened this hypothesis by showing numerically that QNNs trained faster than DNNs.



(a) *Lorem ipsum* **(b)** *Lorem ipsum*

Figure 7.2.1: *Caption place holder*

Returning to Figure 7.2.1, we see that the spectra of the three layer QCNs exhibit much the same uniformity as the QNNs. A more thorough comparison between different QCNs can be seen in Figure 7.2.2. In this figure, we vary the number of layers of the QNCs 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. In doing this, we get to precisely shift how much of the complexity of a given QCN results from the complexity of each node or overall structure of the network. Figure 7.2.2a shows that, for 4 qubit circuits, the spectra of the QNN and different QCNs exhibit roughly the same uniformity. Going up to 6 qubits, Figure 7.2.2b shows that the spectrum tends to concentrate more around zero the more layers the QCN have. This is likely related to the vanishing of the gradient induced by back propagation. For 4 qubits, this is not as big of a problem since the local gradients are relatively big. However, for 6 qubits, the local gradients are smaller. This results in the gradient vanishing faster when increasing the number of layers, which in turn results in a flatter landscape. At any rate, even for the 5 layer QCN, the landscape is not as badly distorted as for the 5 layer DNN. Further, the few layered QCNs, in particular(ost).



(a) *Lorem ipsum* **(b)** *Lorem ipsum*

Figure 7.2.2: *Caption place holder*

7.3 Expressivity

We will investigate the expressivity QCN using the trajectory length method of Raghu et al. [28], as described in subsection 5.2.1. The trajectory length will be first studied for randomly initialized QCNs for 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 2D gaussian data. The results in both cases will be compared to similar DNNs, with approximately the same number of parameters for fair comparison. The input trajectory $\boldsymbol{x}(t_i)$ used will be circle in $\mathbb{R}^2$ with radius $\frac{\pi}{2}$ and centered around 0, divided up into 1000 equally spaced point.

7.3.1 Untrained Models

Figure 7.3.1 shows the trajectory length resulting from each layer up to the eighth laye, for several different QCNs. All layers have d number of nodes, and all nodes have d number of qubits. d ranges from 4 to 8. For comparison, the trajectory length is also calculated for an 8-layer DNN with approximately the same number of parameters as the biggest QCN. The QCNs and the DNN is randomly initialized, as in section 7.1.

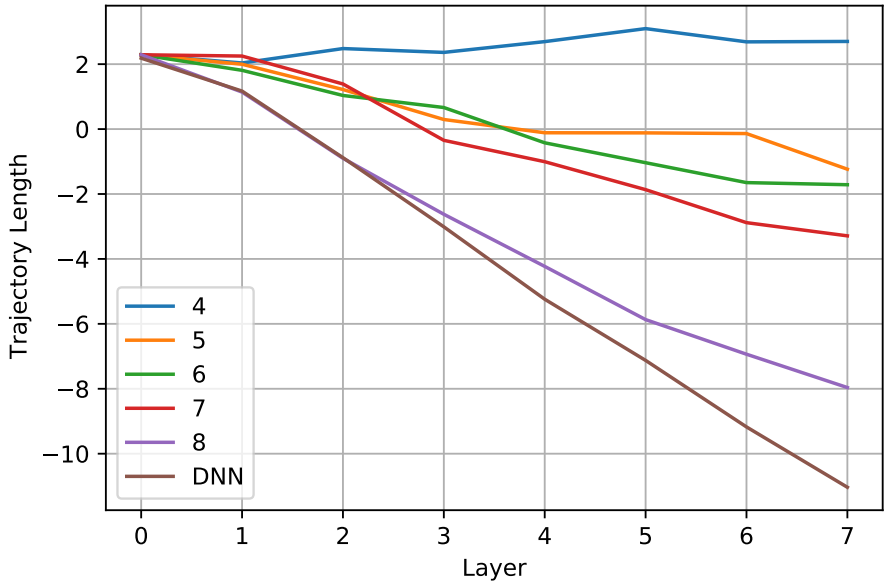


Figure 7.3.1

Figure 7.3.2, accompanying Figure 7.3.1, shows the trajectories of selected models and layers, projected onto 2D. The rows correspond to the 4 qubit QCN, 8 qubit QCN and DNN, from top to bottom. The columns correspond to the first layer, second layer, third layer and last layer, left to right.

Figure 7.3.1, we see that the untrained DNN exhibits an exponential decrease in trajectory length as it is being transformed by each layer. From Figure 7.3.2,

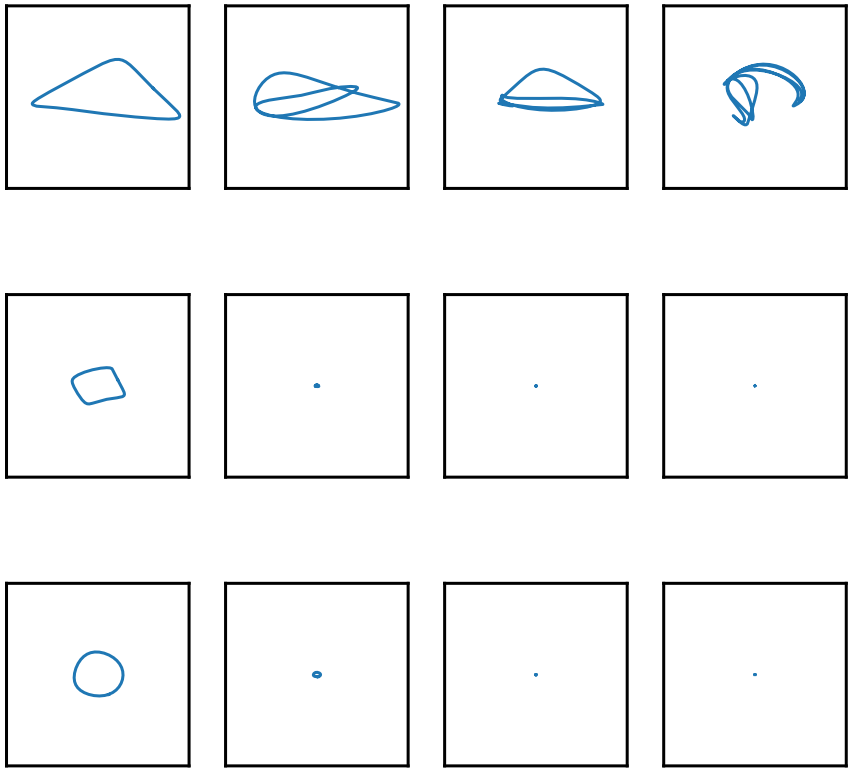


Figure 7.3.2

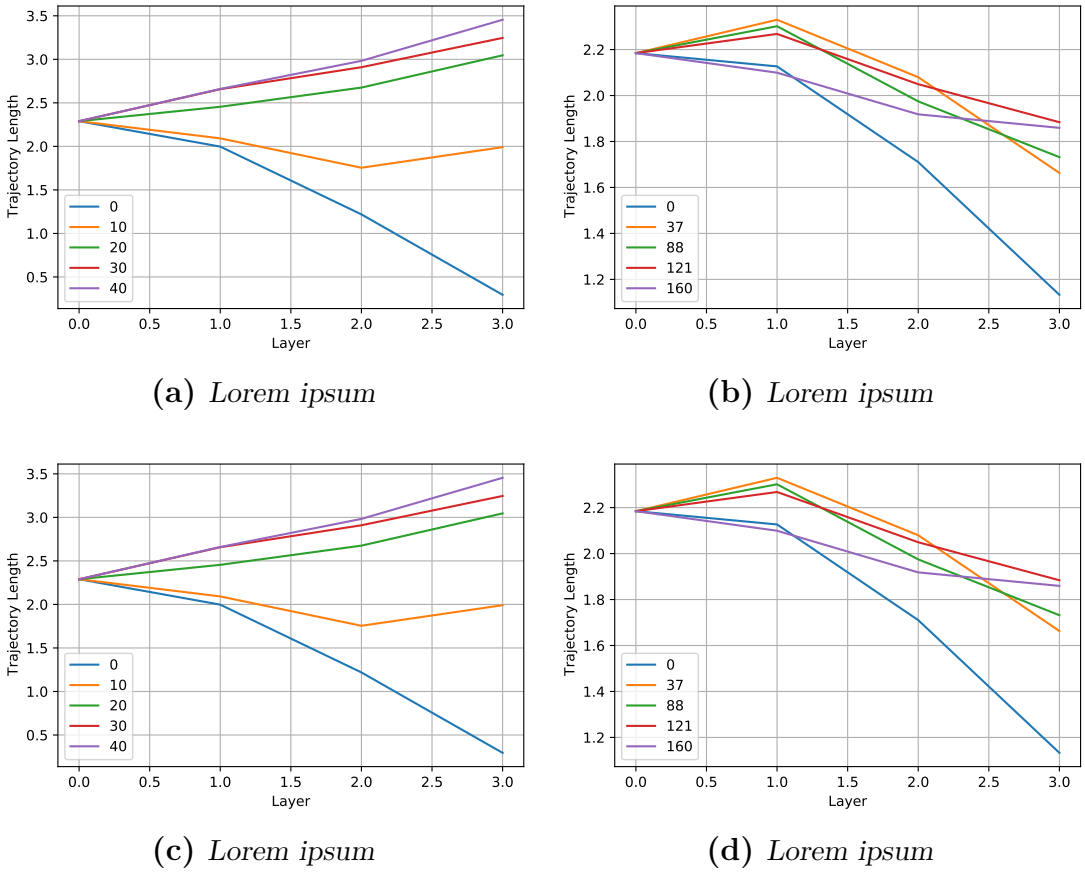
we see this manifesting itself as the trajectory concentrating around some mean, progressively more for each layer. This shows that randomly initialized DNNs can tend to compute functions which are not very sensitive to the input, which is consistent with the result of Raghu et al. [28].

From the same figures we see that the 4-qubit QCNs exhibits a trajectory length that increases approximately exponentially in the number of layers. Figure 7.3.2 shows how the trajectory changes after each layer for the same model, becoming increasingly more distorted and complex. This is in opposition of the result for the DNN, which rather showed an exponential decrease of the trajectory length, concentrating its trajectory around a mean. However, as the number of qubits in each node increase, the QCNs tend to produce trajectory lengths that decrease exponentially in the number of layers. This effect is more severe the more qubits are used. This effect can be explained by the concentration of output values of PQCs and QNNs and their mean, for increasing number of qubits. As QCNs create new features for each layer using QNNs, this concentration around the mean is applied multiple times, mapping the inputs closer and closer.

For 4 though 8 qubits, the previous result indicates that QCNs generally compute more complex functions than the DNN when randomly initialized. A possible explanation for this is that the mathematical transformation applied by each QNN in each nodes are in a way more powerful and "interesting" compared to DNNs. As the information is represented as an entangled state in an exponentially large Hilbert space, each Pauli rotation transforms an exponential number of amplitudes. This constitutes a more complex computation than the affine transformation happening in the DNN, for the same number of parameters. However, this results only in a more interesting output for a low number of qubits, as the increased dimension of the Hilbert space tends to concentrate the output around the mean.

7.3.2 Trained Models

Figure 7.3.3 shows how the trajectory length changes as various models are incrementally trained to fit 2D Gaussian data(see). The models being investigated here are two QCNs with 5 and 6 qubits, respectively, with each node QNN having the same architecture as in subsection 7.3.1. These will be compared with DNNs with approximately same number of parameters. For more information about the models, see Table 7.3.1. All the models are trained using Adam optimizer with the standard hyperparameters and a learning rate of 0.1. The QCNs are trained for a total of 40 epochs, in increments of 10. In order to produce a fair comparison, the DNNs will not be trained for the same increments of epochs. Rather, they will trained until they achieve approximately the same MSE on the training set as the QCNs, for each increment. In this way, we obtain QCNs and DNNs that fit the data to an equal degree, which we then deem comparable.



(a) *Lorem ipsum* **(b)** *Lorem ipsum*

(c) *Lorem ipsum* **(d)** *Lorem ipsum*

Figure 7.3.3: *Caption place holder*

From Figure 7.3.3, we see that training the QCNs and DNNs progressively increases the trajectory length of the models. After only 10 to 20 epochs, the trajectory length of the 5 qubit QNC enters the exponential growth regime. This demonstrates that randomly initialized QCNs can be made to produce complex functions through training, even though outputs initially tended to concentrate around a mean, as shown in Figure 7.3.1. A likely explanation is that the training updates the parameters in such a way that brings structure to each of the node QNNs, moving them away from being random circuits. In this way, the outputs no longer concentrate around the mean, which in turn lets the model compute more complex functions. As seen in (fig), 6-qubit is also eventually brought into the exponential growth regime after 60 epochs, which is significantly more than what was required for 5 qubits. 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 are required to significantly change the parameters such that the QNN nodes no longer resemble random circuits.

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, are more expressive

than DNNs for the same number of parameters. Using the same argument as in subsection 7.3.1, the increased expressivity of QCNs relative to DNNs same number of parameters

Model	Type	Qubits	Layers	Width	n_θ
A	QCN	5	3	5	168
B	QCN	6	3	6	228
C	DNN	NA	3	8	177
D	DNN	NA	3	9	217

Table 7.3.1: *Stuff.*

7.4 Training Models on Gaussian Data

Part IV

Conclusion & Future Research

8

Conclusion & Future Research

8.1 Conclusion

8.2 Future Research

Appendices

A

Amplitude Encoding

Stuff

B

Data Generation

Stuff

References

[1] Maria Schuld and Francesco Petruccione. *Supervised Learning with Quantum Computers*. 1st. Springer Publishing Company, Incorporated, 2018. ISBN: 3319964232 (cit. on pp. 3, 7, 15, 21, 22, 28).

[2] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001 (cit. on pp. 3, 4, 38).

[3] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksanddeeplearning.com/ (cit. on p. 3).

[4] S. A. Vavasis. *Nonlinear Optimization: Complexity Issues*. New York: Oxford University Press, 1991 (cit. on p. 5).

[5] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on p. 8).

[6] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 8).

[7] 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 et al. 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 p. 8).

[8] Amira Abbas et al. *The power of quantum neural networks*. 2020. arXiv: 2011.00027 [quant-ph] (cit. on pp. 8, 25, 27, 30–32, 36–38, 60, 61).

[9] Andrea Skolik et al. *Layerwise learning for quantum neural networks*. 2020. arXiv: 2006.14904 [quant-ph] (cit. on p. 8).

[10] 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. ISBN: 026268053X (cit. on p. 10).

[11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176 (cit. on pp. 11, 16, 29).

[12] John Preskill. "Quantum Computing in the NISQ era and beyond." In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: http://dx.doi.org/10.22331/q-2018-08-06-79 (cit. on p. 22).

[13] R. Barends et al. "Superconducting quantum circuits at the surface code threshold for fault tolerance." In: *Nature* 508.7497 (Apr. 2014), pp. 500–503. ISSN: 1476-4687. DOI: 10.1038/nature13171. URL: http://dx.doi.org/10.1038/nature13171 (cit. on p. 22).

[14] Adam Holmes et al. "Impact of qubit connectivity on quantum algorithm performance." In: *Quantum Science and Technology* 5.2 (Mar. 2020), p. 025009. ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab73e0. URL: http://dx.doi.org/10.1088/2058-9565/ab73e0 (cit. on p. 23).

[15] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 1095-7111. DOI: 10.1137/s0097539795293172. URL: http://dx.doi.org/10.1137/S0097539795293172 (cit. on p. 24).

[16] Marcello Benedetti et al. "Parameterized quantum circuits as machine learning models." In: *Quantum Science and Technology* 4.4 (Nov. 2019), p. 043001. ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab4eb5. URL: http://dx.doi.org/10.1088/2058-9565/ab4eb5 (cit. on pp. 24–26, 30, 31).

[17] M. Cerezo et al. "Cost function dependent barren plateaus in shallow parametrized quantum circuits." In: *Nature Communications* 12.1 (Mar. 2021). ISSN: 2041-1723. DOI: 10.1038/s41467-021-21728-w. URL: http://dx.doi.org/10.1038/s41467-021-21728-w (cit. on p. 24).

[18] Mikko Möttönen et al. "Transformation of quantum states using uniformly controlled rotations." In: *Quantum Information Computation* 5 (Sept. 2005), pp. 467–473. DOI: 10.26421/QIC5.6-5 (cit. on p. 28).

[19] Seth Lloyd. *Quantum approximate optimization is computationally universal*. 2018. arXiv: 1812.11075 [quant-ph] (cit. on p. 29).

[20] Seth Lloyd et al. *Quantum embeddings for machine learning*. 2020. arXiv: 2001.03622 [quant-ph] (cit. on p. 29).

[21] Maria Schuld et al. "Evaluating analytic gradients on quantum hardware." In: *Physical Review A* 99.3 (Mar. 2019). ISSN: 2469-9934. DOI: 10.1103/physreva.99.032331. URL: http://dx.doi.org/10.1103/PhysRevA.99.032331 (cit. on p. 31).

[22] Jarrod R. McClean et al. "Barren plateaus in quantum neural network training landscapes." In: *Nature Communications* 9.1 (Nov. 2018). ISSN: 2041-1723. DOI: 10.1038/s41467-018-07090-4. URL: http://dx.doi.org/10.1038/s41467-018-07090-4 (cit. on pp. 32, 57).

[23] Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. *Failures of Gradient-Based Deep Learning*. 2017. arXiv: 1703.07950 [cs.LG] (cit. on pp. 32, 59).

[24] 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. 36, 37, 61).

[25] Yann A. LeCun et al. "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. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3 (cit. on p. 37).

[26] Patrick Huembeli and Alexandre Dauphin. "Characterizing the loss landscape of variational quantum circuits." In: *Quantum Science and Technology* 6.2 (Feb. 2021), p. 025011. ISSN: 2058-9565. DOI: 10.1088/2058-9565/abdbc9. URL: http://dx.doi.org/10.1088/2058-9565/abdbc9 (cit. on p. 37).

[27] 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. Solla, and D. Cohn. Vol. 11. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper/1998/file/bc7316929fe1545bf0b98d114ee3ecb8-Paper.pdf (cit. on p. 38).

[28] Maithra Raghu et al. *On the Expressive Power of Deep Neural Networks*. 2017. arXiv: 1606.05336 [stat.ML] (cit. on pp. 38, 39, 63, 65).

[29] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. DOI: 10.5281/zenodo.2562110 (cit. on p. 41).