

Quantum Machine Learning for Event Classification at the LHC

Bachelor's Thesis

Submitted to the Faculty of Mathematics, Computer Science and
Natural Sciences at RWTH Aachen University

presented by

Hendrik Kühne

under the supervision of

Prof. Dr. Michael Krämer

January 7, 2023

Contents

1. Introduction	1
2. (Quantum) Classifiers and Quantum Circuits	1
2.1. Quantum Classifiers	2
2.2. Quantum Circuits	3
2.3. Mathematical Representation	3
2.3.1. States of Qubits and their Tensor Products	3
2.3.2. Quantum Gates	4
3. Composing Gates	5
3.1. Composing Gates Acting in Series	6
3.2. Composing Gates Acting in Parallel	6
3.2.1. Two Wires and Single-Qubit-Gates	7
3.2.2. Arbitrary Number of Wires and Single-Qubit Gates	7
3.2.3. Double-Qubit Gates	8
4. Training Quantum Classifiers	10
5. Quantum Circuit Architecture	12
5.1. Structure of Quantum Circuits	12
5.2. Non-Linearity in Quantum Circuits	13
5.3. Layers in Quantum Circuits	14
5.4. The Use of Entanglement Gates	15
6. Implementing Quantum Classifiers	17
6.1. The Module pennylane	18
6.2. The numpy Implementation	19
6.3. Runtime Comparison	21
7. A Training Example	24
7.1. The Training Metrics	24
7.2. The Gradient of the Quantum Classifier	27
8. Dead parameters and Entanglement	28
8.1. The Cause of Dead Parameters	28
8.1.1. The parameters θ_7 , θ_8 and θ_9	29
8.1.2. The parameter θ_{12}	29
8.1.3. The parameter θ_6	30
8.2. Entanglement Comparison	30
9. Encoding Comparison	31
9.1. Angle encoding	31
9.2. Inverse angle encoding	32
9.3. Dense angle encoding	33
9.4. Double angle encoding	33
9.5. Wavefunction encoding	34
9.6. Encoding scheme comparison	35
10. Classical Preprocessing	37
10.1. Rotations	37

10.2. Translations	38
10.3. Composing rotations and translations	38
10.4. Comparison of preprocessing schemes	38
11. On Under- and Overfitting	42
12. Classifying the LHC Olympics Dataset	44
12.1. The Dataset	44
12.2. Feature Selection	45
12.3. Training a Neural Network and a Quantum Classifier	45
13. Conclusion & Outlook	48
A. Encoding Decision Boundaries	50
Acknowledgments	53
References	54

1. Introduction

Both computer science and particle physics currently undergo big changes.

Particle physics is at a threshold: The standard model of particle physics (SM) is complete, yet there are unsolved problems like the nature of dark matter and the origin of neutrino masses [1]. High-energy particle physics at CERN plays a pivotal role in the search for physics *beyond the standard model* (BSM). The search for signatures of BSM events is increasingly data-driven, with algorithms searching the vast amounts of data. The current dominant search paradigms are model-driven: Algorithms target specific signal models for which signal and background data are generated. An analysis strategy is developed and employed [1]. Machine learning is applied increasingly often when searching for signs of new physical processes in accelerator experiments. This is due to two reasons: Machine learning promises to enable a more model-independent search and scales well to the large amounts of data which are generated by the LHC.

Quantum technologies in computer science introduce paradigm shifts in the development of both hardware and software. The term *quantum computing* refers to computation being carried out on hardware which explicitly employs quantum mechanics to accelerate computation or make classically hard problems tractable. In recent years, quantum hardware has advanced from testing on a few qubits to operational quantum computers with over 10 qubits. In conjunction with advances in hardware, algorithms for near-term quantum hardware with practical applications have been proposed [2]. Since some authors [3] have now claimed to have achieved *quantum supremacy* (the completion of a task on a quantum computer which is not tractable on classical hardware), this field opens up the possibility of performance increases for applications like machine learning, probability distribution sampling [4] and simulation [5].

The potential of quantum computing and the demands of particle physics have kickstarted the field of quantum machine learning. It aims to implement machine learning tasks like supervised and unsupervised classification or regression on quantum hardware or in quantum-classical hybrid systems. Various authors have applied quantum machine learning techniques to particle physics [6, 7, 8].

In sections 2 to 4, we introduce the concept of supervised classification and the theoretical basics of quantum machine learning. This includes the building blocks of quantum classifiers and the method by which they are trained. We discuss design principles of quantum circuits in Section 5 and investigate how decision boundaries are constructed. We introduce an implementation in python which enables the simulation of quantum classifiers and use it to train quantum classifiers. We compare the python implementation with the pennylane library in python. Different methods of encoding the input data and different methods of preprocessing the dataset are compared in sections 9 and 10. We comment on the structure of the decision boundaries in Section 11. Finally, we apply the results of the previous sections to the LHC Olympics datasets and compare the performance of a quantum classifier and a neural network.

2. (Quantum) Classifiers and Quantum Circuits

The term *classifier* refers to a class of algorithms designed to separate data into classes. In *supervised classification* tasks, an algorithm is provided with *training data* $\{\vec{x}_t\} = T \subset \Gamma$ and the corresponding *labels* $\{y_t\}$. The set Γ contains all

possible data points. For d -dimensional input d , we refer to one component of \vec{x} as a *feature*. Throughout this work, we will deal with datasets where $d = 2$ and $\Gamma \subseteq \mathbb{R}^2$. For *binary classification*, the data is to be separated into two classes which we denote by 0 and 1, e.g. $y \in \{0, 1\}$. In analogy to high-energy physics nomenclature where the goal is to separate events with new physics from those without new physics, we will speak of a label of $y = 1$ being “positive” (or “signal”) and $y = 0$ being “negative” (or “background”).

Classifiers then are functions

$$Q : \Gamma \rightarrow \{0, 1\} \quad (2.1)$$

which take data $\vec{x} \in \Gamma$ and output a prediction for the data point (a label in the case of binary classification). Q may depend on an arbitrary number of parameters.

The parameters on which Q depends are changed in a process called *training* such that the output $Q(\vec{x}_t)$ of the classifier is, in an appropriate sense, “close” to the corresponding label y_t . “Closeness” is measured with the *loss function* $L : \Gamma^m \rightarrow \mathbb{R}$ where m is the number of training data points which are used for training. A good set of parameters minimizes L .

Since we are dealing with binary classification where $y \in \{0, 1\}$, the prediction $Q(\vec{x})$ of the classifier may be interpreted as the propability the classifier assigns to \vec{x} belonging to class 1. Because of this we use *cross-entropy*, defined as

$$\begin{aligned} L &= -\frac{1}{N} \sum_{\vec{x}_t \in T} \sum_i y_{t,i} \log_2(\hat{y}_i) \\ &= -\frac{1}{N} \sum_{\vec{x}_t \in T} (y_t \log_2(\hat{y}) + (1 - y_t) \log_2(1 - \hat{y})) \end{aligned} \quad (2.2)$$

as loss-function, where $\hat{y} = Q(\vec{x}_t)$ is the prediction of the classifier for the data point \vec{x}_t . \sum_i denotes a summation over the classes which the classifier is trained to distinguish, which amounts to two classes, hence the simplification. \hat{y}_i is the propability the classifier assigns to \vec{x} belonging to class i . The set $T \subset \Gamma$ contains all points used for training.

2.1. Quantum Classifiers

We investigate *quantum classifiers*, which are classifiers which are implemented on quantum hardware. We implemented algorithms of the form

$$Q(\vec{x}) = a(f(\vec{x}; \theta)) \quad (2.3)$$

where $f : \Gamma \rightarrow \chi$ is a *quantum circuit* (abbreviated as QC) and $a : \chi \rightarrow \{0, 1\}$ is the *activation function*. The set χ depends on the specific implementation. In this work, we choose $\chi = [-1, 1]$ and the sigmoid function

$$a(x) = \frac{1}{1 + e^{-2x}} \quad (2.4)$$

for a . We chose the factor 2 as scaling of the argument of the sigmoid function because this ensures that the loss function (defined below in Equation 2.2) is defined for every possible network output. We constructed the quantum classifier such that $Q(\vec{x}) = 1$ means that the classifier predicts that \vec{x} belongs to class 1.

2.2. Quantum Circuits

Quantum circuits are the center piece of a quantum classifier. They consist of quantum operators (hereafter called *gates* in analogy to classical computing) acting on two-level quantum systems or their tensor products. These two-level systems are called *qubits* in analogy to the bit of classical computation. The physical nature of these qubits is important in the design of quantum computers, however this work is concerned with the possible algorithms using this architecture and not their realization from a practical point of view. Thus, we will not discuss possible realizations of qubits¹.

The gates are unitary operators and act on one or multiple qubits. One qubit in a circuit is referred to as a *wire*, e.g. one may state that “gate X acts on wire j ”. The first wire is wire 0. The structure of a QC is often shown in a *QC diagram*. All wires are shown with the operators which are applied to them from left to right. An operator may be connected to two wires, meaning it acts on both of them (both qubits). Some wires may, at the end, terminate in a gate. These gates (or, to be precise, the corresponding operators) are the ones whose expectation value will be taken as the output of the QC. An example is shown in Figure 2.1.

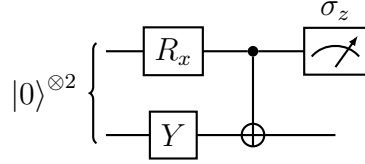


Figure 2.1: An example for a QC diagram: Both Qubits are initialized in the ground state $|0\rangle$. The gate R_x acts on wire 0 and Y acts on wire 1; afterwards, a CNOT-gate is applied with wire 0 being the control-qubit. The output is the expectation value of σ_z acting on wire 0. Specific quantum gates will be introduced in Section 2.3.2.

2.3. Mathematical Representation

2.3.1. States of Qubits and their Tensor Products

The state of a single qubit is given by

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle, \quad |\alpha|^2 + |\beta|^2 = 1, \quad |\psi\rangle \in \mathcal{H}_2. \quad (2.5)$$

where \mathcal{H}_2 is the Hilbert space of a single qubit and α, β are complex numbers. The state $|\psi\rangle$ of a system of N Qubits is contained in the *tensor product space*

$$\mathcal{H}_2^N = \bigotimes_{k=0}^{N-1} \mathcal{H}_2. \quad (2.6)$$

The state of a quantum system is often written in the *computational basis* as a column vector whose coefficients are the components in the tensor product basis: We write $|i_0, i_1, \dots, i_{N-1}\rangle$ for the basis vectors

¹For a discussion of physical realizations of quantum computers, see Ref. [9], Chapter 7: *Quantum Computers: physical realization*

$$|i_0, i_1, \dots, i_{N-1}\rangle = \bigotimes_{j=0}^{N-1} |i_j\rangle \quad \text{where } i_j \in \{0, 1\}. \quad (2.7)$$

A state in the computational basis then becomes

$$|\psi\rangle = \sum_{k=0}^{2^N-1} \alpha_{\pi_k} |\pi_k\rangle \doteq \begin{pmatrix} \alpha_{\pi_0} \\ \alpha_{\pi_1} \\ \vdots \\ \alpha_{\pi_{(2^N-1)}} \end{pmatrix} \quad (2.8)$$

where $\pi_k = (i_0, i_1, \dots, i_{N-1})$ is a permutation of the indices. We permute in such a way that $\pi_j = j$ when π_j is interpreted as a number in base two, with $0 \leq j \leq 2^N - 1$. The symbol “ \doteq ” means “is represented by”.

Considering for example $N = 2$, we have

$$|\psi\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle \doteq \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}. \quad (2.9)$$

The fact that there are 2^N permutations π_k highlights an important fact about simulating quantum circuits on classical computing devices: The state of the system and the gates acting on it become exponentially large when increasing the number N of qubits.

2.3.2. Quantum Gates

Gates act on qubits and change their state. They may be used to initialize the state of the system or they are applied during computation to change the system's state. They may be parameter-dependent. Since qubits may be written as vectors, gates take the form of matrices. With \mathcal{H}_2^N being 2^N -dimensional, the matrix representation of any gate A acting on N qubits is a $\mathbb{C}^{2^N \times 2^N}$ -matrix.

A multi-qubit-gate may be *separable* or *non-separable* depending on whether it is possible to write it as a tensor product of multiple gates. Mathematically, we call a gate U acting on \mathcal{H}_2^N separable, if

$$U = \bigotimes_{i=0}^{N-1} U_i \quad (2.10)$$

where each U_i acts on \mathcal{H}_2 .

Throughout this work, A will denote either the gate itself or its matrix representation based on the context.

Important examples of gates and their representations may be found in Table 2.1.

The Identity is of course a gate which does not change the state of the system. In tensor products, we will write identity gates explicitly as I . The X, Y, Z gates are the well-known Pauli spin operators. X is the quantum-mechanical analogue to a NOT-gate in classical computing as it swaps the amplitudes of the basis states $|0\rangle, |1\rangle$. We will write $\sigma_x, \sigma_y, \sigma_z$ instead to emphasize when these gates are used as physical observables in an expectation value (in the context of the output of the circuit). R_y is one of three often-used rotation gates. They obtain their names from





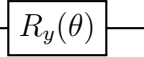
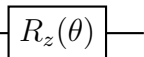
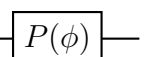
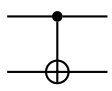
Identity		$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Pauli-X		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y		$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z		$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
y-Rotation		$\begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$
z-Rotation		$\begin{pmatrix} \exp(-i\theta/2) & 0 \\ 0 & \exp(i\theta/2) \end{pmatrix}$
Phase shift		$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$
Controlled-Not		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

Table 2.1: A list of gates with their circuit- and matrix representation.

the fact that the state of a single qubit may be visualised as a point on the three-dimensional unit sphere², which is then called the *Bloch Sphere*. Rotation gates then correspond to rotations of said sphere. The phase-shift gate $P(\phi)$ applies a phase-shift to the state $|1\rangle$. The Controlled-Not gate (abbreviated as CNOT) inverts the second qubit if and only if the first qubit is in state $|1\rangle$. CNOT is not separable.

We will very often use a more general rotation gate: The R -gate carries out a general single-qubit rotation on the Bloch Sphere [6] and is given by

$$R(\phi, \theta, \omega) = R_z(\omega)R_y(\theta)R_z(\phi) \doteq \begin{pmatrix} e^{-i(\phi+\omega)/2} \cos(\theta/2) & -e^{i(\phi-\omega)/2} \sin(\theta/2) \\ e^{-i(\phi-\omega)/2} \sin(\theta/2) & e^{i(\phi+\omega)/2} \cos(\theta/2) \end{pmatrix}. \quad (2.11)$$

The list in Figure 2.1 is not comprehensive; other gates are in use³ and in general, every unitary matrix may be interpreted as the representation of a quantum gate⁴.

3. Composing Gates

The state of a given Qubit is changed by applying a *quantum gate*. Since all quantum gates are represented by unitary matrices, a given quantum circuit may be seen as the composition of gates acting on the respective Qubit(s), and thus only as a linear transformation of the prepared state $|\psi_{\text{in}}\rangle = U_{\text{in}}(\vec{x})|0\rangle^{\otimes N}$. For implementing a quantum circuit, we need a firm grasp on how to apply quantum gates.

Gates may be applied in two ways: *in series* or *in parallel*.

²See Ref. [9], Chapter 1.2: *Quantum Bits*

³See Ref. [9], Chapter 4.2: *Single qubit operations*

⁴See Ref. [9], Chapter 1.3.1: *Single qubit gates*

3.1. Composing Gates Acting in Series

The composition of gates applied in series is straightforward: Applying two gates one after the other (in matrix-vector representation) means first multiplying the state vector by the first gate and then multiplying by the second gate. Composing the gates means multiplying the matrices, i.e.

$$U_1 (U_2 |\psi\rangle) = (U_1 U_2) |\psi\rangle. \quad (3.1)$$

This holds for gates acting on a single or multiple wires.

3.2. Composing Gates Acting in Parallel

Gates may be applied to only one of the N Qubits which make up the entire state of the system or different gates may be applied in parallel. As we will discuss in the next paragraph, we need a way to describe both actions as actions on the complete state of the system in the computational basis.

When applying a gate A to Qubit i , the operation should not affect the state of all qubits $j \neq i$ while applying A to i . We thus set the gate $A^{(i)}$ to be

$$A^{(i)} = \underbrace{I \otimes \cdots \otimes I}_{i-1 \text{ times}} \otimes A \otimes \underbrace{I \otimes \cdots \otimes I}_{N-i \text{ times}}. \quad (3.2)$$

This also enables us to apply gates in parallel: When applying $A \otimes B$, we may decompose this into

$$A \otimes B = (A \otimes I)(I \otimes B) = (I \otimes B)(A \otimes I) \quad (3.3)$$

Thus being able to write $A^{(i)}$ as a matrix acting on the computational basis is sufficient for building quantum circuits.

Writing expressions such as Equation 3.2 in the computational basis is necessary because of a phenomenon which is central to counter-intuitive processes in quantum information⁵: *entanglement*. We call a quantum state $|\psi\rangle \in \mathcal{H}_2^N$ entangled if it is not the result of a tensor product of states living in the factor Hilbert spaces. Mathematically, a state $|\psi\rangle$ is entangled if there exist no $\{|\psi_j\rangle\}, |\psi_j\rangle \in \mathcal{H}_2$, such that

$$|\psi\rangle = \bigotimes_{j=0}^{N-1} |\psi_j\rangle. \quad (3.4)$$

Otherwise, the state is *separable*.

Entangled states require us to compose the matrix representations of gates. To see why, consider the action of $U = A \otimes I$ on different states:

- The action of U on a separable state $|\psi\rangle = |\psi_0\rangle \otimes |\psi_1\rangle$ may be calculated directly:

$$U |\psi\rangle = (A \otimes I)(|\psi_0\rangle \otimes |\psi_1\rangle) = (A |\psi_0\rangle) \otimes |\psi_1\rangle, \quad (3.5)$$

using the known properties of the tensor product.

⁵See for example Ref. [9], Chapter 2.6: *EPR and the Bell Inequality*

- U acting on the state

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \neq |\psi_0\rangle \otimes |\psi_1\rangle \quad (3.6)$$

must be computed differently because $|\psi\rangle$ cannot be written as tensor product; it is entangled. Such states require us to find a way to express arbitrary U for arbitrary N as matrices $U \in \mathbb{C}^{2^N \times 2^N}$. These can then be applied to the state $|\psi\rangle$ written in the computational basis.

3.2.1. Two Wires and Single-Qubit-Gates

The complete gate acting on the computational basis is of course given by the tensor product. Consider the composition $A \otimes I$ where

$$A \doteq \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}. \quad (3.7)$$

The gate $A \otimes I$, which acts on $|\psi\rangle \in \mathcal{H}_2^2$, is then given by

$$A \otimes I \doteq \begin{pmatrix} a_{00}I^{2 \times 2} & a_{01}I^{2 \times 2} \\ a_{10}I^{2 \times 2} & a_{11}I^{2 \times 2} \end{pmatrix} = \begin{pmatrix} a_{00} & 0 & a_{01} & 0 \\ 0 & a_{00} & 0 & a_{01} \\ a_{10} & 0 & a_{11} & 0 \\ 0 & a_{10} & 0 & a_{11} \end{pmatrix}, \quad (3.8)$$

where $I^{n \times n}$ denotes the $n \times n$ identity matrix. Conversely, $I \otimes A$ is represented by

$$I \otimes A \doteq \begin{pmatrix} (I^{2 \times 2})_{00}A & (I^{2 \times 2})_{01}A \\ (I^{2 \times 2})_{10}A & (I^{2 \times 2})_{11}A \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{00} & a_{01} \\ 0 & 0 & a_{10} & a_{11} \end{pmatrix}. \quad (3.9)$$

This reflects the structure of the computational basis

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \doteq \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}, \quad (3.10)$$

where qubit 0 alternates between states $|0\rangle$ and $|1\rangle$ over two rows while qubit 1 alternates with every other row.

3.2.2. Arbitrary Number of Wires and Single-Qubit Gates

Computing the tensor product of matrices of arbitrary dimension works analogously to what we presented in equation 3.8. In general, when computing a tensor product $A \otimes B$ where $A \in \mathbb{C}^{n \times n}$ and $B \in \mathbb{C}^{m \times m}$, the resulting matrix is a $(nm) \times (nm)$ -matrix. Every component of A is multiplied by the matrix B , i.e.

$$A \otimes B = \begin{pmatrix} a_{00}B & a_{01}B & \cdots & a_{0n}B \\ a_{10}B & a_{11}B & \cdots & a_{1n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n0}B & a_{n1}B & \cdots & a_{nn}B \end{pmatrix} \in \mathbb{C}^{(nm) \times (nm)}. \quad (3.11)$$

In this way, arbitrary tensor products of quantum gates may be evaluated iteratively; consider for example the composition of the X , Y and Z -gate

$$\begin{aligned}
X \otimes Y \otimes Z &\doteq \underbrace{\left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \right)}_{\in \mathbb{C}^{4 \times 4}} \otimes \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}}_{\in \mathbb{C}^{2 \times 2}} \\
&\underbrace{\hspace{10em}}_{\in \mathbb{C}^{8 \times 8}} \\
&= \begin{pmatrix} 0 \cdot Y \otimes Z & 1 \cdot Y \otimes Z \\ 1 \cdot Y \otimes Z & 0 \cdot Y \otimes Z \end{pmatrix} \\
&= \begin{pmatrix} 0 \cdot \begin{pmatrix} 0 \cdot Z & -i \cdot Z \\ i \cdot Z & 0 \cdot Z \end{pmatrix} & 1 \cdot \begin{pmatrix} 0 \cdot Z & -i \cdot Z \\ i \cdot Z & 0 \cdot Z \end{pmatrix} \\ 1 \cdot \begin{pmatrix} 0 \cdot Z & -i \cdot Z \\ i \cdot Z & 0 \cdot Z \end{pmatrix} & 0 \cdot \begin{pmatrix} 0 \cdot Z & -i \cdot Z \\ i \cdot Z & 0 \cdot Z \end{pmatrix} \end{pmatrix}.
\end{aligned} \tag{3.12}$$

Writing out everything yields

$$X \otimes Y \otimes Z \doteq \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -i & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & i \\ 0 & 0 & 0 & 0 & i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i & 0 & 0 \\ 0 & 0 & -i & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 0 & 0 & 0 \\ i & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -i & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{3.13}$$

3.2.3. Double-Qubit Gates

A double-qubit gate is a 4×4 -matrix, when written in the computational basis of the two wires it applies to. We will introduce the method by which we apply such gates using the example of the CNOT-gate.

The CNOT-gate where wire 0 serves as control and wire 1 may be inverted is represented by the matrix

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{3.14}$$

However, applying a CNOT to multi-qubit states where $N > 2$ and which are potentially entangled is not immediately possible. As with single-qubit gates, we need a way to evaluate the tensor product of a CNOT which is controlled by wire j and inverts wire k (which we will write as $\text{CNOT}(c = j, i = k)$) such that we can apply it to a state written in the computational basis.

The key realization for doing this is that the matrix in Equation 3.14 may be split up into submatrices, which correspond to the control-qubit being in state $|0\rangle$ or $|1\rangle$ in the input and output state. Let $|\psi^{\text{in}}\rangle$ denote the input and $|\psi^{\text{out}}\rangle$ the output. $|\psi_c^{\text{in/out}}\rangle$ is the state of the control-qubit in the input or output respectively; then, we may divide the CNOT-gate as follows (keeping in mind the structure of the computational basis and the way in which matrix rows and vectors align during matrix multiplication):

$$\begin{aligned}
|\psi_c^{\text{out}}\rangle = |0\rangle & \begin{pmatrix} \alpha_{00}^{\text{out}} \\ \alpha_{01}^{\text{out}} \\ \alpha_{10}^{\text{out}} \\ \alpha_{11}^{\text{out}} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha_{00}^{\text{in}} \\ \alpha_{01}^{\text{in}} \\ \alpha_{10}^{\text{in}} \\ \alpha_{11}^{\text{in}} \end{pmatrix} \\
|\psi_c^{\text{out}}\rangle = |1\rangle & \begin{pmatrix} \alpha_{00}^{\text{out}} \\ \alpha_{01}^{\text{out}} \\ \alpha_{10}^{\text{out}} \\ \alpha_{11}^{\text{out}} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha_{00}^{\text{in}} \\ \alpha_{01}^{\text{in}} \\ \alpha_{10}^{\text{in}} \\ \alpha_{11}^{\text{in}} \end{pmatrix}
\end{aligned} \quad (3.15)$$

These submatrices of course tell us how the components of the input state map to the output state. Consider the top-left submatrix: It corresponds to the action on the invert-wire when $|\psi_c^{\text{in}}\rangle = |\psi_c^{\text{out}}\rangle = |0\rangle$. The submatrices we find are explicitly written in Table 3.1.

CNOT _{sub}	$ \psi_c^{\text{in}}\rangle = 0\rangle$	$ \psi_c^{\text{in}}\rangle = 1\rangle$
$ \psi_c^{\text{out}}\rangle = 0\rangle$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
$ \psi_c^{\text{out}}\rangle = 1\rangle$	$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Table 3.1: The submatrices CNOT_{sub} of a CNOT-gate. These may be thought of as being single-qubit-gates which act on the inverted wire.

We can then think of these submatrices as being single-qubit gates which act on the invert-wire and can include them in the tensor product accordingly. Let us for example write CNOT($c = 0, i = 2$) as a gate which acts on a system with 3 qubits: We find the submatrices shown in Figure 3.1 and identify these as gates acting on wire 2 since $i = 2$. The action of CNOT on the control-wire is encoded in the submatrices, so the gate acting on wire 0 must not change any components of the complete gate. The matrix acting on wire 0 is $\mathbf{1}^{2 \times 2}$, accordingly ($\mathbf{1}^{n \times n}$ is a $n \times n$ matrix where $(\mathbf{1}^{n \times n})_{ij} = 1$ for all i, j). The complete tensor product for CNOT($c = 0, i = 2$) then reads as

$$\text{CNOT}(c = 0, i = 2) \doteq \mathbf{1}_{\text{CNOT}}^{2 \times 2} \otimes I \otimes \text{CNOT}_{\text{sub}}. \quad (3.16)$$

By writing $\mathbf{1}_{\text{CNOT}}^{2 \times 2}$, we emphasize that wire 0 serves as a control-wire. This tensor product may be evaluated as we discussed in Section 3.2.2, however, we need to substitute the submatrices of CNOT for CNOT_{sub} according to which state the control-wire is in in input and output. For this we may refer to the computational basis (see also Figure 3.1).

Writing out the tensor product in Equation 3.16 yields

$$\text{CNOT}(c = 0, i = 2) \doteq \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}. \quad (3.17)$$

$$|\psi\rangle \doteq \left(\begin{array}{c} \alpha_{000} \\ \alpha_{001} \\ \alpha_{010} \\ \alpha_{011} \\ \alpha_{100} \\ \alpha_{101} \\ \alpha_{110} \\ \alpha_{111} \end{array} \right) \left\{ \begin{array}{l} |\psi_2\rangle = |0\rangle \\ |\psi_2\rangle = |1\rangle \\ |\psi_2\rangle = |0\rangle \\ |\psi_2\rangle = |1\rangle \\ |\psi_2\rangle = |0\rangle \\ |\psi_2\rangle = |1\rangle \\ |\psi_2\rangle = |0\rangle \\ |\psi_2\rangle = |1\rangle \end{array} \right\} \left\{ \begin{array}{l} |\psi_1\rangle = |0\rangle \\ |\psi_1\rangle = |1\rangle \\ |\psi_1\rangle = |0\rangle \\ |\psi_1\rangle = |1\rangle \\ |\psi_1\rangle = |0\rangle \\ |\psi_1\rangle = |1\rangle \\ |\psi_1\rangle = |0\rangle \\ |\psi_1\rangle = |1\rangle \end{array} \right\} \left\{ \begin{array}{l} |\psi_0\rangle = |0\rangle \\ |\psi_0\rangle = |1\rangle \end{array} \right\}$$

Figure 3.1: An example for which rows in the computational basis correspond to which states of the respective qubits. The state of the j th qubit is written as $|\psi_j\rangle$. This applies analogously to the columns of a gate in matrix representation.

The bottom-left and top-right quadrants contain only zeros because the corresponding submatrices CNOT_{sub} contain zeros. The other submatrices are recognizable in the top-left and bottom-right quadrants. Their arrangement mirrors the fact that the identity acts on wire 1.

This rule for finding the matrix representation of $\text{CNOT}(i = j, c = k)$ is applicable to all double-qubit gates $U(j, k)$ which act on wires j, k where $j < k$. One must first divide U into submatrices (as we did in Equation 3.15) and may then write out U as a tensor product where $\mathbf{1}^{2 \times 2}$ acts on wire j and the submatrices U_{sub} act on wire k (analogous to Equation 3.16). Other gates may be incorporated in the tensor product structure as well by substituting the identity gate in Equation 3.16 with the respective gate.

4. Training Quantum Classifiers

Training a quantum classifier means finding good parameters such that a measure of performance is increased. The measure we chose is cross-entropy (see Section 2 and Equation 2.2).

We trained using two ways:

- Using the method `minimize` from the python module `scipy.optimize` with the optimizer COBYLA.
- Using gradient descent.

The first option is a black box, derivative-free method. It needs to be supplied with an objective function to be minimized which depends on the parameters that are optimized and an initial guess for said parameters. The algorithm then constructs a linear approximation of the objective function by evaluating it at the edges of a simplex. Based on this approximation, a better set of parameters is found that is again used to linearly approximate the objective function. This is repeated until improvement is no longer possible [10].

Gradient descent is a method in which the parameters are optimized by an iterative process. For a performance measure which needs to be minimized (e.g. a loss function), at every step, the gradient of the loss function L with respect to the parameters is calculated. The parameters are then iteratively updated by subtracting the gradient, i.e.

$$\vec{\theta} \mapsto \vec{\theta} - \eta \nabla L. \quad (4.1)$$

In this way, the algorithm converges to a local minimum of L in the parameter space. The parameter η is called the *learning rate*. After having tried different values for η , we set $\eta = 1$. This offers the best compromise between fast decrease of L and being able to converge to a minimum.

The key challenge with using gradient descent is to calculate the gradient of the quantum circuit. In classical machine learning, the gradient is calculated through backpropagation⁶, where at each step during calculation of a network output, the gradient of a single node is saved such that these gradients can later be chained to calculate the total gradient.

Calculating the gradient of quantum circuits is conventionally done in a different way [6, 12]. This is because conventionally used quantum gates are expressible as operator exponentials; consider for example the R_y -gate

$$R_y(\theta) \doteq \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \doteq \exp(-iY\theta/2). \quad (4.2)$$

This is because

$$Y \doteq \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \underbrace{\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ i & -i \end{pmatrix}}_{=:W} \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}}_{=:D} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ i & -i \end{pmatrix}^H \quad (4.3)$$

where W is unitary.

Equation 4.2 then follows directly from $\exp(-iY\theta/2) \doteq \exp(-iW D W^H \theta/2) = W \exp(-iD\theta/2) W^H$.

The authors of Ref. [13] showed the following: Given a function

$$f(\theta) = \langle \psi | B^\dagger U^\dagger(\theta) A U(\theta) B | \psi \rangle \quad (4.4)$$

where $U(\theta) = \exp(-ia\theta G)$ with G hermitian, the derivative of f is

$$\frac{\partial f}{\partial \theta} = r \left[f\left(\theta + \frac{\pi}{4r}\right) - f\left(\theta - \frac{\pi}{4r}\right) \right]. \quad (4.5)$$

For the phase shift gate P , the rotation gates R_x, R_y, R_z and hence the general rotation R as well, the normalization constant r amounts to $1/2$. This rule applies to quantum circuits; we may thus write down the gradient of the loss function, combining equations 2.2, 2.3, 2.4 and 4.5:

$$\begin{aligned} \frac{\partial L}{\partial \theta_j} &= -\frac{1}{N \ln 2} \sum_{\{\vec{x}_t\} \in T} \left(\frac{y_t}{\hat{y}} - \frac{1-y_t}{1-\hat{y}} \right) \frac{\partial a}{\partial x}(f(\vec{x}_t)) \frac{\partial f}{\partial \theta_j}(\vec{x}_t) \\ &= -\frac{1}{2N \ln 2} \sum_{\{\vec{x}_t\} \in T} \left(\frac{y_t}{\hat{y}} - \frac{1-y_t}{1-\hat{y}} \right) \frac{\partial a}{\partial x}(f(\vec{x}_t)) \\ &\quad \times \left[f\left(\vec{x}_t; \theta_j + \frac{\pi}{2}\right) - f\left(\vec{x}_t; \theta_j - \frac{\pi}{2}\right) \right] \end{aligned} \quad (4.6)$$

⁶See Ref. [11], in Chapter 10: *Multi-Layer Perceptron and Backpropagation*

Note that only the parameter θ_j which is changed is written explicitly as an argument of f for easier reading.

5. Quantum Circuit Architecture

In the following section, we'll introduce the structure of the quantum circuits we used. Furthermore, we'll discuss how exactly a QC is able to classify diverse data sets and what criteria must be met to reach this ability.

5.1. Structure of Quantum Qircuits

Since the aim of this thesis is to discuss machine learning on quantum hardware, the quantum classifiers we used are designed with the goal of high flexibility in mind. Flexibility refers to the ability of the quantum classifier to classify diverse data sets by fine-tuning parameters of the quantum circuit. This approach is contrary to quantum circuits which serve a more narrowly defined purpose. An example for this is the quantum adder proposed by Richard Feynman [14].

The resulting structure is similar to how classical neural networks are built because it is also composed of input encoding followed by layers (see Figure 5.1).

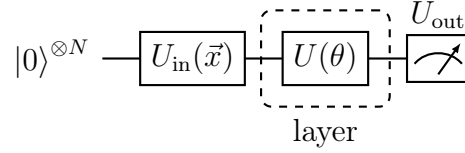


Figure 5.1: The typical structure of a quantum circuit: In the beginning, all qubits are in the ground state $|0\rangle$. The input \vec{x} is encoded in the state of the system using an *encoding gate* U_{in} . The parameterized gates $U(\theta)$ act on the state of the system and the final output of the quantum circuit f is the measurement of some observable through the measurement gate U_{out} .

In all circuits, we use the observable $\sigma_z^{(0)}$ as the quantum circuit output. The design of U_{in} is variable and will be introduced as necessary. $U(\theta)$ always consists of two parts: The *parameterized part* and the *entanglement part* (which will be referred to as ENT; see also Figure 5.2). The parameterized part consists of R -gates applied to every qubit. The entanglement part plays a vital role in the functioning of quantum circuits (as will be discussed in Section 5.4). It is built out of multi-qubit gates which have the ability to produce entangled states.

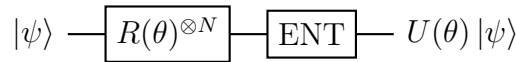


Figure 5.2: The typical structure of $U(\theta)$. First, a rotation gate is applied to every qubit (symbolised by $R(\theta)^{\otimes N}$), afterwards, the entanglement gate is applied (symbolised by ENT).

The ENT-part must be non-separable. We extensively use two examples in this work: *CNOT-layers* and *Ising time gates*. A CNOT-layer is a collection of CNOT-

gates in series where every qubit serves as control for every other qubit. Written in pseudocode, this reads

```
for j in range(n_QuBit):
    for k in range(n_QuBit):
        if j != k:
            CNOT(c=j, i=k)
```

where $\text{CNOT}(c=j, i=k)$ is a shorthand for applying said gate with the respective control and invert wires.

An Ising time gate applies the time evolution operator e^{-iHt} to the state of the system [12]. The Hamiltonian which is often used is the transverse Ising model on the 1D-chain with random coupling strenghts, i.e.

$$H = \sum_{j=0}^{N-1} X^{(j)} + \sum_{\langle j,k \rangle} J_{jk} Z^{(j)} Z^{(k)} \quad (5.1)$$

where $\sum_{\langle j,k \rangle}$ means summation over all adjacent chain locations. The coefficients are randomly (uniformly) sampled from the intervall $[-1, 1]$. We set the evolution time t randomly to 10 since it is not related to the gates' ability to entangle states. The Ising time gate $\text{ENT} = e^{-iHt}$ is thus a gate which acts on as many wires as desired; when used for entanglement, we apply it to all wires. It is not separable.

5.2. Non-Linearity in Quantum Circuits

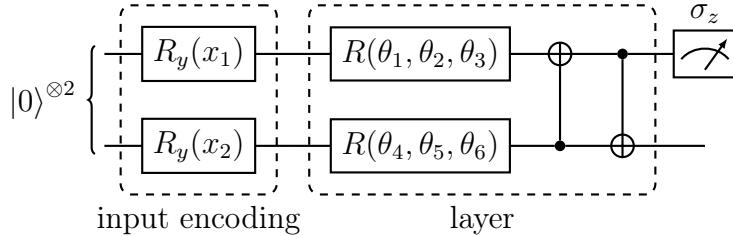


Figure 5.3: The quantum circuit used in Ref. [6]. Input encoding is done through R_y gates with the coordinates as input. The quantum state is processed by Rotation gates and CNOT-gates. Rotation and CNOT-gates together form a *layer* which is repeated multiple times. Ref. [6] uses two layers; we will investigate the effects of using more or fewer layers in Section 5.3. The output is the expectation value of σ_z applied to the 0th wire.

In order to understand how and under which requirements a quantum circuit is able to classify data, we'll write out a circuit in the density matrix formalism. Among others, we tested the design used in Ref. [6] (see Figure 5.3). The density matrix for a single qubit after encoding is

$$\begin{aligned} \rho &= |\psi_{\text{in}}(x)\rangle \langle \psi_{\text{in}}(x)| = R_y(x) |0\rangle \langle 0| R_y^\dagger(x) \\ &= \left[\cos\left(\frac{x}{2}\right) |0\rangle + \sin\left(\frac{x}{2}\right) |1\rangle \right] \left[\cos\left(\frac{x}{2}\right) \langle 0| + \sin\left(\frac{x}{2}\right) \langle 1| \right] \\ &= \frac{1}{2} [I + \sin(x)X + \cos(x)Z]. \end{aligned} \quad (5.2)$$

The complete density matrix then of course becomes

$$\rho_{\text{in}} = |\psi_{\text{in}}(x_1)\rangle \langle \psi_{\text{in}}(x_1)| \otimes |\psi_{\text{in}}(x_2)\rangle \langle \psi_{\text{in}}(x_2)|. \quad (5.3)$$

Writing this out results in a 4×4 -matrix whose components include the trigonometric functions. The output of the network is given by

$$f(\vec{x}; \theta) = \text{Tr} \left(\sigma_z^{(0)} U(\theta) \rho_{\text{in}} U^\dagger(\theta) \right). \quad (5.4)$$

At this point, a key difference between classical machine learning and quantum circuits used for machine learning becomes clear: classical neural networks are able to classify non-linear distributions because the output of every node is chained with an activation function. The output in Equation 5.4 however is the result of matrix multiplication and a trace operation; these are linear operations. Non-linearity is therefore only possible because the input is inserted into nonlinear functions during encoding (in this case the trigonometric functions).

This recontextualises the role of layers in a quantum circuit: In classical machine learning, layers are essential for the capability of the network to approximate nonlinear distributions. In quantum circuits, all non-linearity comes from the input encoding. In Section 5.3, we will investigate the advantages of using multiple layers despite this argument.

5.3. Layers in Quantum Circuits

To understand the role of layers, let us explicitly write out the density matrix of the circuit presented in Figure 5.3 after encoding, i.e.

$$\begin{aligned} \rho_{\text{in}} &= |\psi_{\text{in}}(x_1)\rangle \langle \psi_{\text{in}}(x_1)| \otimes |\psi_{\text{in}}(x_2)\rangle \langle \psi_{\text{in}}(x_2)| \\ &= \frac{1}{4} [I + \sin(x_1)X + \cos(x_1)Z] \otimes [I + \sin(x_2)X + \cos(x_2)Z] \\ &= \frac{1}{4} [I + \sin(x_2)X^{(1)} + \cos(x_2)Z^{(1)} \\ &\quad + \sin(x_1)X^{(0)} + \sin(x_1)\sin(x_2)X \otimes X + \sin(x_1)\cos(x_2)X \otimes Z \\ &\quad + \cos(x_1)Z^{(0)} + \cos(x_1)\sin(x_2)Z \otimes X + \cos(x_1)\cos(x_2)Z \otimes Z]. \end{aligned} \quad (5.5)$$

The tensor products of the gates can be computed using the rules introduced in Section 3. We find

$$\begin{aligned}
\rho_{\text{in}} = & \frac{1}{4} \begin{pmatrix} 1 + \cos(x_2) + \cos(x_1) + \cos(x_1) \cos(x_2) & 0 & 0 & 0 \\ \sin(x_2) + \cos(x_1) \sin(x_2) & 0 & 0 & 0 \\ \sin(x_1) + \sin(x_1) \cos(x_2) & 0 & 0 & 0 \\ \sin(x_1) \sin(x_2) & 0 & 0 & 0 \end{pmatrix} \\
& + \frac{1}{4} \begin{pmatrix} 0 & \sin(x_2) + \cos(x_1) \sin(x_2) & 0 & 0 \\ 0 & 1 - \cos(x_2) + \cos(x_1) - \cos(x_1) \cos(x_2) & 0 & 0 \\ 0 & \sin(x_1) \sin(x_2) & 0 & 0 \\ 0 & \sin(x_1) - \sin(x_1) \cos(x_2) & 0 & 0 \end{pmatrix} \\
& + \frac{1}{4} \begin{pmatrix} 0 & 0 & \sin(x_1) + \sin(x_1) \cos(x_2) & 0 \\ 0 & 0 & \sin(x_1) \sin(x_2) & 0 \\ 0 & 0 & 1 + \cos(x_2) - \cos(x_1) - \cos(x_1) \cos(x_2) & 0 \\ 0 & 0 & \sin(x_2) - \cos(x_1) \sin(x_2) & 0 \end{pmatrix} \\
& + \frac{1}{4} \begin{pmatrix} 0 & 0 & 0 & \sin(x_1) \sin(x_2) \\ 0 & 0 & 0 & \sin(x_1) - \sin(x_1) \cos(x_2) \\ 0 & 0 & 0 & \sin(x_2) - \cos(x_1) \sin(x_2) \\ 0 & 0 & 0 & 1 + \cos(x_2) - \cos(x_1) + \cos(x_1) \cos(x_2) \end{pmatrix}.
\end{aligned} \tag{5.6}$$

During computation, ρ is multiplied with unitary matrices from the left and the right (refer to Equation 5.4). Contrary to classical neural networks, the layer structure thus does not change the linearity or non-linearity of f ; the layers shuffle the components of ρ . The circuit output f is a weighted sum of the components of ρ_{in} .

A quantum circuit thus behaves more like a series expansion than a neural network since f is a sum of functions rather than a composition of activation functions and weighted sums (as is the case with neural networks). In this sense, we will refer to the coefficients of the density matrix in tensor product form (Equation 5.5) as *basis functions*. The basis functions depend on the encoding scheme the quantum circuit uses (basis functions will be discussed in more detail in Section 9).

We investigate the impact of the amount of layers on the quality of shuffling empirically by comparing the accuracy of predictions for different numbers of layers. The *accuracy* is defined as the fraction of correct predictions out of all predictions. The quantum circuit has two wires and uses the encoding presented in Figure 5.3 while applying an Ising time gate as entanglement gate. The classifier was trained on the sklmoons dataset⁷ using the optimizer COBYLA. The results for all n_{layers} are displayed in Figure 5.4. Judging by Figure 5.4, we decide on using 4 layers. This encompasses an additional buffer for any unforeseen effects.

5.4. The Use of Entanglement Gates

The layers contain an entanglement gate. It's use is not immediately obvious; it does not contain parameters and thus cannot be trained. Entanglement is however essential to complex quantum circuits; to see why, consider a quantum circuit where there is no entanglement, e.g. all gates are separable (see Figure 5.5).

In such a circuit, the gates acting on all qubits but the one whose state is measured have no way of influencing the result since the gates (which are unitary) cancel out when the state is separable. It's expectation value is given by

⁷See Figure 9.1 for an introduction to the datasets we use.

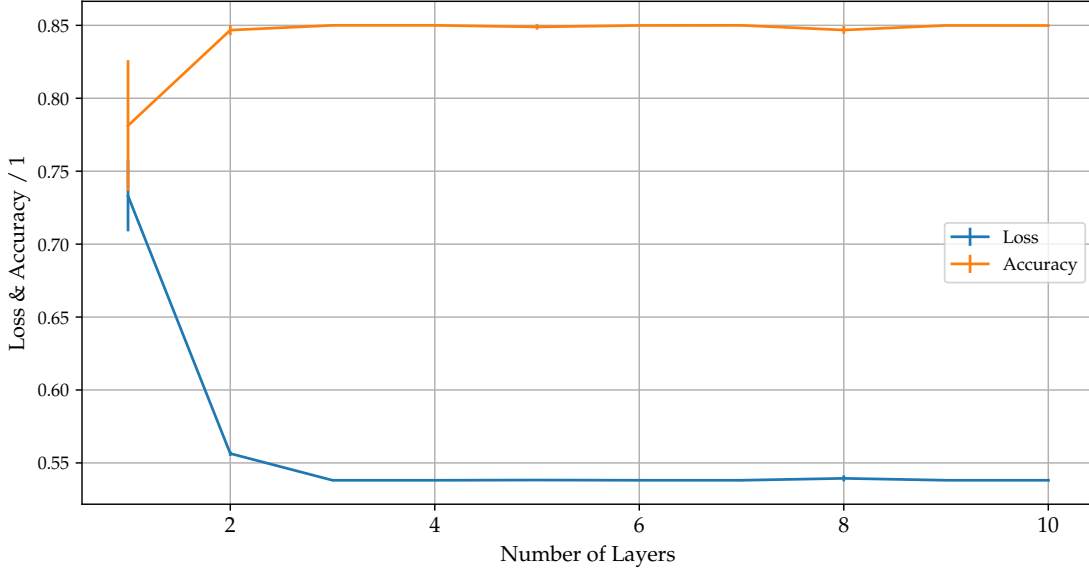


Figure 5.4: Loss and accuracy for different numbers of layers. For every n_{layers} , 30 classifiers were trained and the empirical standard deviations on loss and accuracy were calculated. Very low numbers of layers are clearly not enough as loss and accuracy fall respectively rise significantly for small numbers of layers. This is because one or two layers are not enough to shuffle the components of the density matrix sufficiently. Loss and accuracy reach plateaus with three layers.

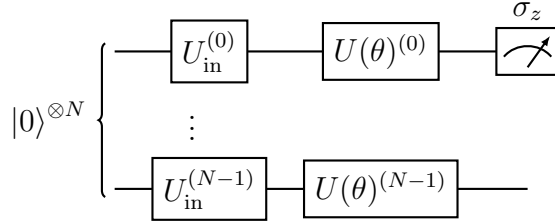


Figure 5.5: A schematic of a quantum circuit which is built out of separable qubits.

$$\begin{aligned}
f &= \langle 0|^{\otimes N} U_{\text{in}}^\dagger U(\theta)^\dagger \sigma_z^{(0)} U(\theta) U_{\text{in}} |0\rangle^{\otimes N} \\
&= \left[\bigotimes_{j=0}^{N-1} \langle 0| \left(U_{\text{in}}^{(j)} \right)^\dagger \left(U(\theta)^{(j)} \right)^\dagger \right] \sigma_z^{(0)} \left[\bigotimes_{j=0}^{N-1} U(\theta)^{(j)} U_{\text{in}}^{(j)} |0\rangle \right] \\
&= \prod_{j=0}^{N-1} \langle 0| \left(U_{\text{in}}^{(j)} \right)^\dagger \left(U(\theta)^{(j)} \right)^\dagger \sigma_0^{(0)} U(\theta)^{(j)} U_{\text{in}}^{(j)} |0\rangle \\
&= \langle 0| \left(U_{\text{in}}^{(0)} \right)^\dagger \left(U(\theta)^{(0)} \right)^\dagger \sigma_0^{(0)} U(\theta)^{(0)} U_{\text{in}}^{(0)} |0\rangle
\end{aligned} \tag{5.7}$$

where we have used that $(U^{(k)})^\dagger \sigma_z^{(j)} U^{(k)} = I$ for $k \neq j$ and unitary U since $\sigma_z^{(j)}$ applies the identity gate to qubit k .

Entanglement thus fulfills two roles:

- It enables access to the values stored in all qubits during encoding. Look back to Figure 5.3 for an example: during encoding, the value x_1 is stored in

qubit 0 and x_2 is stored in qubit 1. The circuit would have only had access to one of two features if it would have been separable.

- It increases the number of usable parameters. For an extreme example, refer to Figure 5.5: Although every wire passes through parameterized gates, all but the ones acting on wire 0 cancel out in the expectation value. We consider an ideal entanglement process to be one in which no parameter cancels out during the calculation of the expectation value.

6. Implementing Quantum Classifiers

We simulated quantum classifiers in Python. We used two implementations:

- The module `pennylane` by Xanadu Quantum Technologies Inc., a Python module which provides tools for simulating and training quantum circuits and interfaces with quantum hardware [15].
- A self-made implementation using `numpy`.

Both implementations are available in a class `predictor`. At initialization, it takes several keyword arguments:

- `implementation` with value `pennylane` or `scratch` (as in “from scratch”, for the `numpy`-implementation). Determines the implementation that will be used.
- `optimizer` with values `vanillaGD` (optimization through gradient descent) or `COBYLA`. Determines the optimizer.
- `n_Qubit` determines how many qubits the quantum circuit manipulates.
- `n_layers` determines the number of layers in the circuit.
- `learning_rate` sets the learning rate η . It defaults to 1 for the `numpy` or `pennylane` implementation.
- `information_density` determines how many features are encoded into one qubit (refer to Chapter 9 for examples of encoding schemes where multiple features are encoded into one qubit).

The `predictor` class saves the parameters of the quantum circuit it simulates and the metrics of the training process for every epoch internally.

The class `predictor` provides several methods:

- `new_params` sets the internal values of the parameters to new, random values. The rotation angles are uniform samples from $[0, 4\pi]$.
- `normalize` is used for normalization of the input data. On the first call, when provided with initial and new ranges $[x_{\min}, x_{\max}]$, $[\hat{x}_{\min}, \hat{x}_{\max}]$ of the input features, the normalization is defined. On subsequent calls, it is called by `predict` and normalizes the input of `predict` to the previously defined range.

- **predict** encodes the input in the N -qubit state of the quantum circuit and simulates computation. The user may provide parameters for the parameterized gates, but if no parameters are given, the internal values will be used. For simulating different quantum circuits, with `implementation==scratch`, the code in **predict** must be changed to implement the corresponding quantum circuit. If `implementation==pennylane`, the quantum function is defined in `__init__` of the predictor class and thus must be changed there.
- **fit** trains the classifier using a training set and the corresponding labels. These must be given as parameters. Calling **fit** while `optimizer==COBYLA` corresponds to finding optimal parameters for the provided training data. If `optimizer==vanillaGD`, one gradient descent iteration is performed. The method `parameter_update` is called which calculates the gradient ∇L of the loss function and updates the parameters. During calculation of ∇L , the method `grad_predictor` is called which calculates the gradient of the quantum circuit.

Both implementations accept a numpy array \mathbf{x} with shape

```
>>> print(x.shape)
[... , n_Qubits*information_density]
```

as input. “...” signifies that the first dimensions of \mathbf{x} are arbitrary. Only the last dimension must have the size `n_Qubits*information_density`, i.e. the last dimension of \mathbf{x} contains the input data $\vec{x} \in \Gamma$.

The quantum circuit is applied element-wise to the last dimension of the input, meaning **predict** outputs an array with shape [...]. The number of input data points will be referred to as *sample size*.

6.1. The Module pennylane

The simulation of quantum circuits in pennylane is done using objects called *quantum nodes* (QNode). The QNode computes the result of a quantum circuit and is usable like a standard function. To create it, one needs a *device* and a *quantum function*.

The device executes the computation. Devices can be simulations on classical hardware as well as interfaces to quantum hardware [16]. The devices we used are `default.qubit` and `lightning.qubit`. Both simulate quantum circuits on classical hardware.

The quantum function defines the quantum circuit that is to be executed. It is a python function that receives the device as decorator.

pennylane provides the gates we mentioned in Section 2.3.2 as classes such that we can write all quantum circuits as pennylane quantum functions. Ising time gates are also possible through the Trotter-Suzuki decomposition formula [18]

$$e^{A+B} = \lim_{n \rightarrow \infty} [e^{A/n} e^{B/n}]. \quad (6.1)$$

This is true for commuting as well as non-commuting operators A, B . The Ising Hamiltonian is a sum $H = \sum_j H_j = \sum_j \alpha_j \sigma_j$ (as discussed in Section 5) where $\alpha \in [-1, 1]$. Employing Equation 6.1 then yields

```

import pennylane as qml
from pennylane import numpy as np
    # to enable differentiation of quantum
    # circuits, pennylane provides an
    # interface to numpy

dev = qml.device('default.qubit',wires=2)

@qml.qnode(dev)
def quantum_circuit(x):
    qml.RZ(x, wires=0)
    qml.CNOT(wires=[0,1])
    qml.RY(x, wires=1)
    return qml.expval(qml.PauliZ(1))

result = quantum_circuit(0.543)

```

Figure 6.1: A code example taken from Ref. [17]. `dev` is the quantum device which uses the `default.qubit`-simulator. It enables computation of two-wire quantum circuits. `quantum_circuit` is the quantum function. It is decorated with the `qml.qnode` decorator, making it a `QNode` that nonetheless behaves like a normal python function.

$$e^{-iHt} \approx \prod_{k=1}^n \prod_j e^{-iH_j t/n}. \quad (6.2)$$

This becomes exact when the H_j commute. Using this ansatz, the Ising time gate e^{-iHt} is split up into multiple gates acting in series.

The input to the quantum circuit is passed as an argument to the encoding gates. The parameters are provided to the `QNode` as well and are used in the parameterized part of the quantum circuit. For a code example refer to Figure 6.1.

We implemented training using gradient descent and via COBYLA (refer to Section 4).

6.2. The numpy Implementation

We implemented quantum circuits from scratch since an implementation in numpy is, for certain circuit architectures, faster (this will be discussed in more detail in Section 6.3). This also greatly facilitated the process of understanding the working principles of quantum circuits. The center piece of this implementation is the state vector of the N -qubit system. It is saved in the computational basis representation as a numpy-array.

Gates are implemented as classes which have the following attributes:

- `n_Qubits`: The number of qubits the gate acts on.
- `wires`: The wires the gate acts on. If `n_Qubits` > 1, this attribute is array-like; otherwise, it is scalar.
- `n_channels`: The number of channels. If a gate depends on parameters, they must be provided during initialization. It is possible to provide an array for

the parameters instead of a scalar. In this case, we say that the gate has multiple channels: One for every parameter that is provided.

- **matrix**: The matrix the gate is represented by in the computational basis. This is a numpy-array with shape $(2^{n_{\text{Qubit}}}, 2^{n_{\text{Qubit}}})$ if `n_channels==1`, in other cases it has the shape $(n_{\text{channels}}, 2^{n_{\text{Qubit}}}, 2^{n_{\text{Qubit}}})$. Thus a gate is actually represented by multiple matrices; one for every channel.

The attributes `n_Qubits` and `wires` must be given at initialization.

Applying a gate of course means carrying out matrix multiplication. As with pennylane, gates may be applied in parallel. The rules for gate composition described in Section 3.2 are implemented in the function

`GateComposer(*gates, n_wires=0)`

which takes an arbitrary number of gates acting on one or two qubits and computes their tensor product. The number of wires will be set to the largest wire any of the gates in `gates` acts on or the value provided for the keyword `n_wires`. If there are any wires for which no gate is given, the identity will be assumed. Errors are raised if the number of channels do not match.

Applying gates in parallel is a two-step process: The gates have to be composed using `GateComposer` and matrix multiplication has to be carried out. These two steps can be done in two lines of code. The σ_z expectation value is calculated and returned at the end:

```
rot_gates = GateComposer(RX(wires=0, theta=np.pi/2),
                        RY(wires=1, theta=np.pi),
                        RZ(wires=2, theta=3*np.pi/2)
                        )
state = np.einsum("...jk,...k->...j",
                  rot_gates, state)

# computing the PauliZ expectation value:
PauliZ_gate = GateComposer(PauliZ(wires=0),
                           n_wires=self.n_Qubit-1)
PauliZ_state = np.einsum("...jk,...k->...j",
                          PauliZ_gate, state)
expval = np.einsum("...j,...j->...",
                  np.conjugate(state), PauliZ_state)

return expval
```

The function `np.einsum` provides a way to use einstein summation in python and is used here for matrix multiplication ⁸.

As in the pennylane-implementation, training is possible using the optimizers `COBYLA` or `vanillaGD`.

⁸We use `np.einsum` instead of `numpy.dot` because it can handle the arbitrary first dimensions of `x`.

6.3. Runtime Comparison

To investigate the behaviour of the different implementations and, in particular, compare the runtimes t_R , we simulated training for different configurations of `n_QuBit`, `n_layers` and the number n_{samples} of data points in the circuit input \mathbf{x} . We used the quantum circuit implemented in Figure 5.3 and simulated it for arbitrary `n_QuBit`-dimensional input. Extending the circuit to `n_QuBit`-dimensional input is done by adding wires as necessary in which the input is encoded with an R_y gate, similar to the other wires. We compared the numpy-implementation and pennylane where the latter used the quantum devices `default.qubit` (the default qubit device in pennylane) and `lightning.qubit` (a qubit device with C++ backend). The results may be found in Figure 6.2 to Figure 6.5.

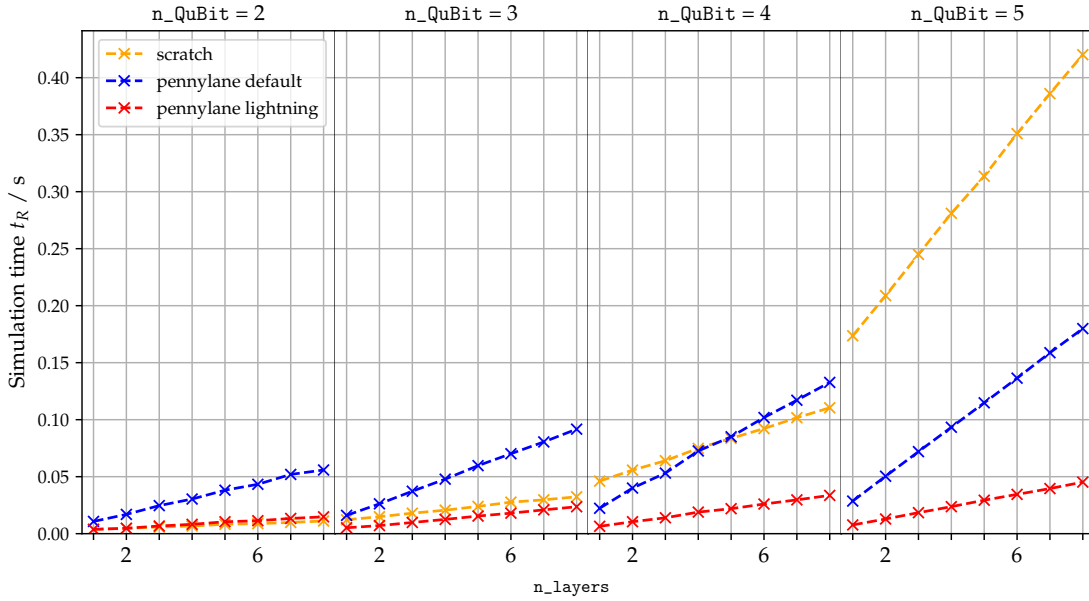


Figure 6.2: The runtimes for 10 samples.

We discuss the results separately for the numpy- and pennylane implementation:

numpy: For every sample size, t_R grows exponentially when the number of qubits is increased. This can be expected since the size of the computational basis vectors and the gate matrices also grows exponentially (recall that $|\psi\rangle \in \mathbb{C}^{(2^N)}$). For 10 or 100 samples and 4 or 5 qubits, the runtime grows linearly with the number of layers. We expected linear growth for all configurations with the number of layers since the number of arithmetic operations increases linearly with an increase in the number of layers (and thus the number of matrix multiplications). It may be visible only for large numbers of qubits because, in other cases, it is dominated by the exponential increase of t_R through a growing number of qubits.

pennylane: pennylane behaves differently to the numpy implementation. Using either `default.qubit` or `lightning.qubit` as device leads to t_R growing linearly with the number of qubits. In contrast to numpy, this is however not the dominating part of the growth in runtime: t_R increases strongly with an increase in the number of layers. Growth of t_R due to increases in the number

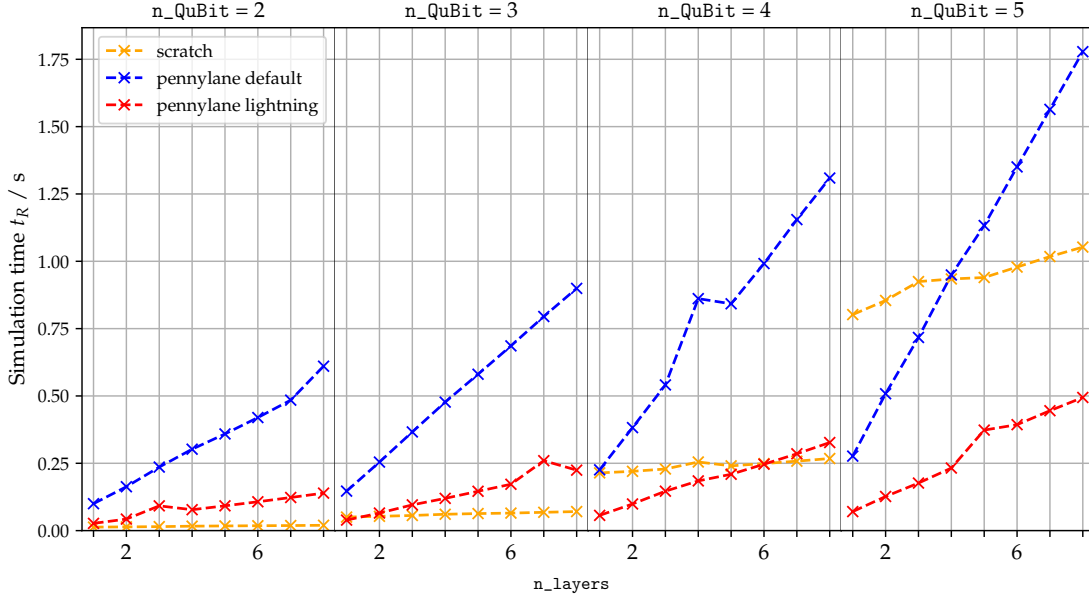


Figure 6.3: The runtimes for 100 samples.

of qubits or the number of layers is present for both devices, however it is much weaker using `lightning.qubit`.

pennylane scales more poorly with the sample size. Refer especially to the data obtained using four qubits: numpy is the slowest implementation for 10 samples while for 10 000 samples and four layers, it is the fastest.

These results lead to us using the numpy implementation often since for two to four qubits it is often the fastest implementation or offers performance comparable to pennylane using the `lightning.qubit`-device. A significant advantage of the numpy implementation is that Ising time gates are much less time-consuming: As mentioned in Section 6.1, e^{-iHt} is implemented by splitting it up into many gates applied in series. Since the ENT-part (which is an Ising time gate) is applied to all qubits, it functions like an extra layer in the quantum circuit. pennylane however scales very poorly with an increasing number of layers. In the numpy implementation, the matrix exponential e^{-iHt} may be directly calculated using `scipy.linalg.expm` and is thus only one more gate that needs to be applied. Therefore, we use the numpy implementation for most simulations.

Both implementations are, of course, identical in output. The differences are only in runtime and, in cases like the Ising time gate, numerical precision.

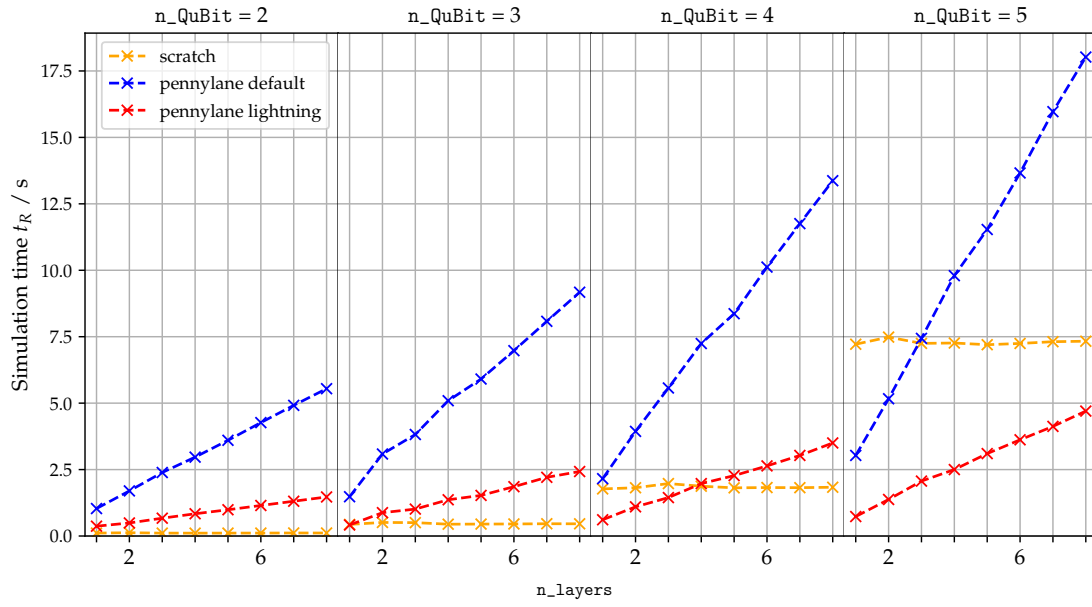


Figure 6.4: The runtimes for 1000 samples.

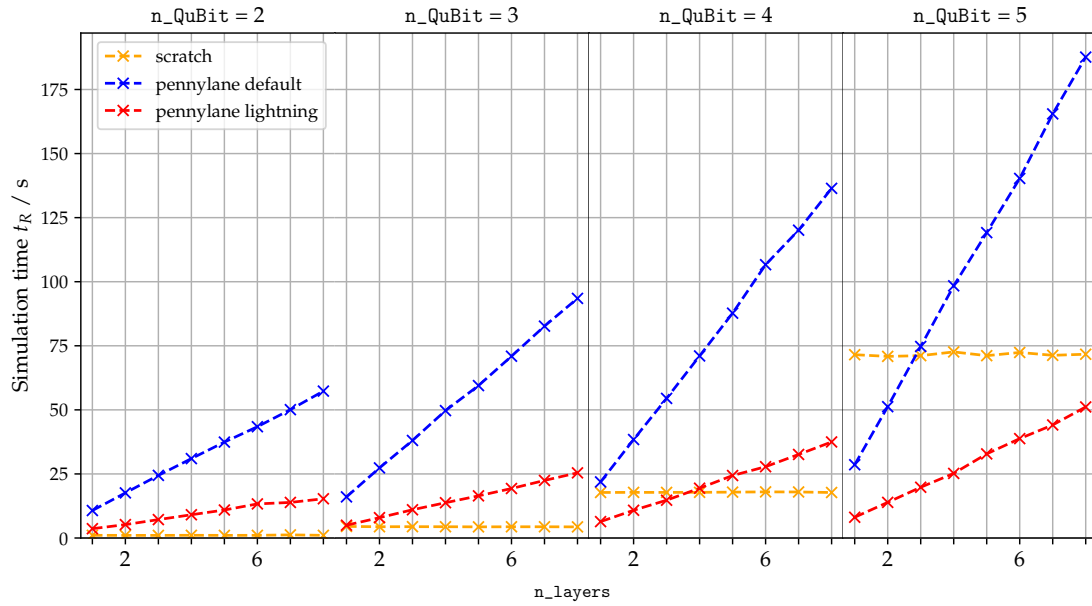


Figure 6.5: The runtimes for 10 000 samples.

7. A Training Example

To see the content of the previous sections in action and introduce the topic “Dead Parameters” of Section 8, we train a quantum circuit on an example dataset.

We choose the `skblobs` dataset from the `sklearn.datasets` module of the SciKit-learn library [19]. It consists of two gaussian blobs centered around given coordinates in arbitrary dimensions with variable noise. We generated 1000 samples with two features (e.g. x_1 and x_2 coordinates) centered around $(-1, -1)^T$ respectively $(0, 2)^T$ with a standard deviation of 0.7 and `random_state=42` (initialization of the random number generator). We split the dataset in half to obtain a training set and a validation set. Validation and Training set contain equally many points between themselves and of noth labels. The dataset is shown in Figure 7.1.

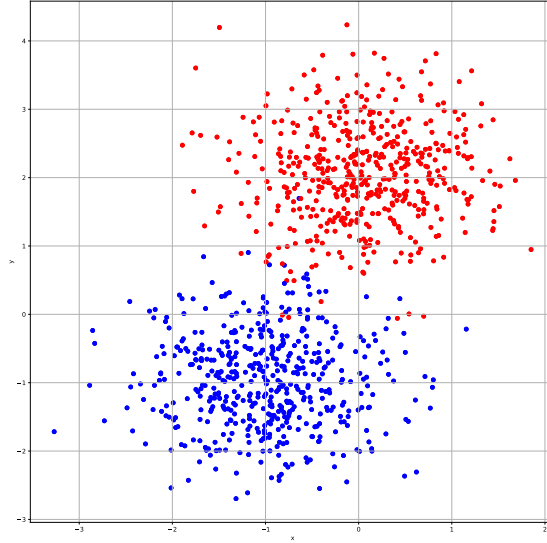


Figure 7.1: The skblobs dataset used for training.

We used the circuit presented in Figure 5.3. To gain a better overview of the parameter space, we set $n_{\text{layers}} = 2$ instead of 4. The data is normalized to the interval $[0, 2\pi]$. Optimization is done through gradient descent with 30 epochs. We split the dataset into two sets used for training and validation respectively.

We observe the training process through plots called *decision boundary plots*. These depict the classifier output $Q(\vec{x})$ for every point on a grid containing the dataset. The *decision boundary* in particular are regions in the plot where $Q(\vec{x}) = 1/2$, which means the classifier assigns a probability of 50 % to the point belonging to either class. These plots are generated for every epoch (see also Figure 7.2).

7.1. The Training Metrics

We monitor the progress of the training process through four metrics which are often used for measuring the performance of classifiers: The value of the loss function L as well as *accuracy*, *precision* and *recall*. They are defined as⁹

⁹See Ref. [11], Chapter 3: *Performance Measures*

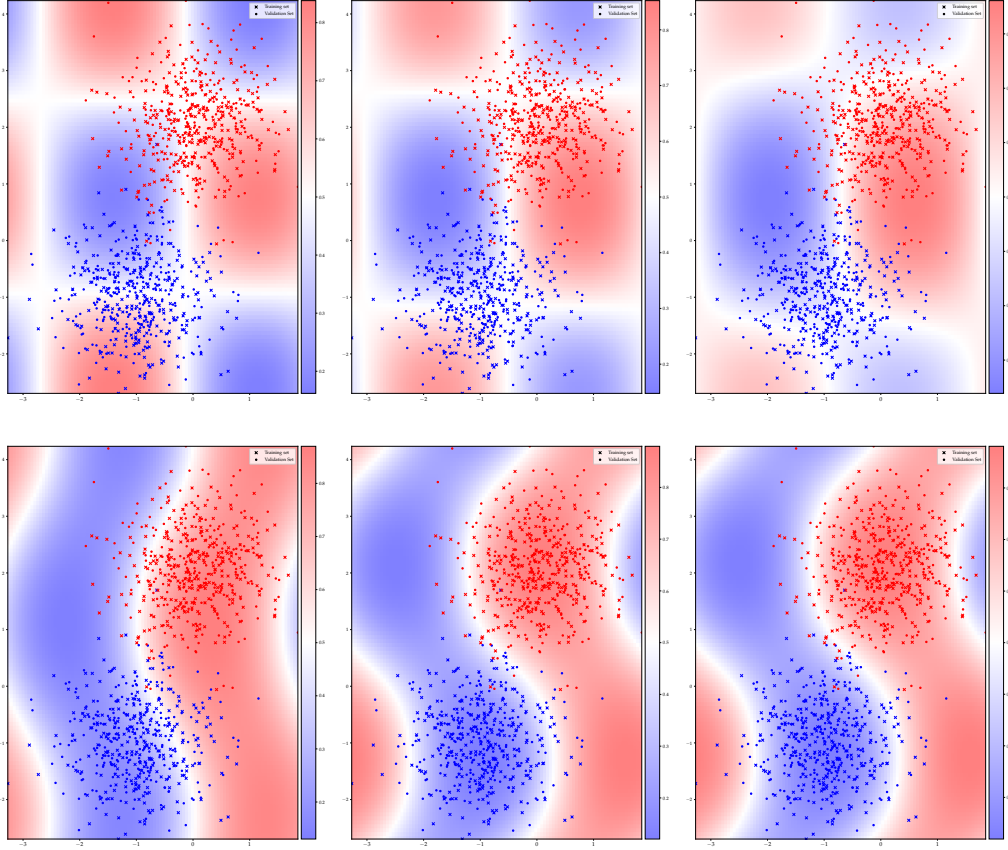


Figure 7.2: In the top row, the decision boundaries for the first three epochs are shown. The bottom row shows epochs 9, 19 and 29. The labels correspond to different colours: Red points respectively regions shaded in red denote labels $y = 1$ respectively regions which are classified as positive (see also the colormaps of the plots). Points belonging to the training set are marked with crosses, the validation set is marked with dots. In the top left (Epoch 0), the random initialization of the parameters means that the decision boundaries do not match the dataset very well. The biggest leap in changing the decision boundaries is taken between epochs 0 and 1. In subsequent epochs, smaller adjustments take place.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (7.1)$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (7.2)$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (7.3)$$

where TP (true positive) is the number of positive predictions which were true, FP (false positive) is the number of positive predictions which were false, TN (true negative) is the number of negative predictions which were true and FN is the number of negative predictions which were false.

These metrics may be interpreted accordingly:

accuracy is the fraction of correct predictions relative to all predictions.

precision is the fraction of correct and positive predictions relative to all instances where the classifier made a “positive” prediction. More shortly, it is the fraction of positive predictions which were correct.

recall is the fraction of positive labels which were correctly identified by the classifier.

The metrics for the training example are shown in Figure 7.3.

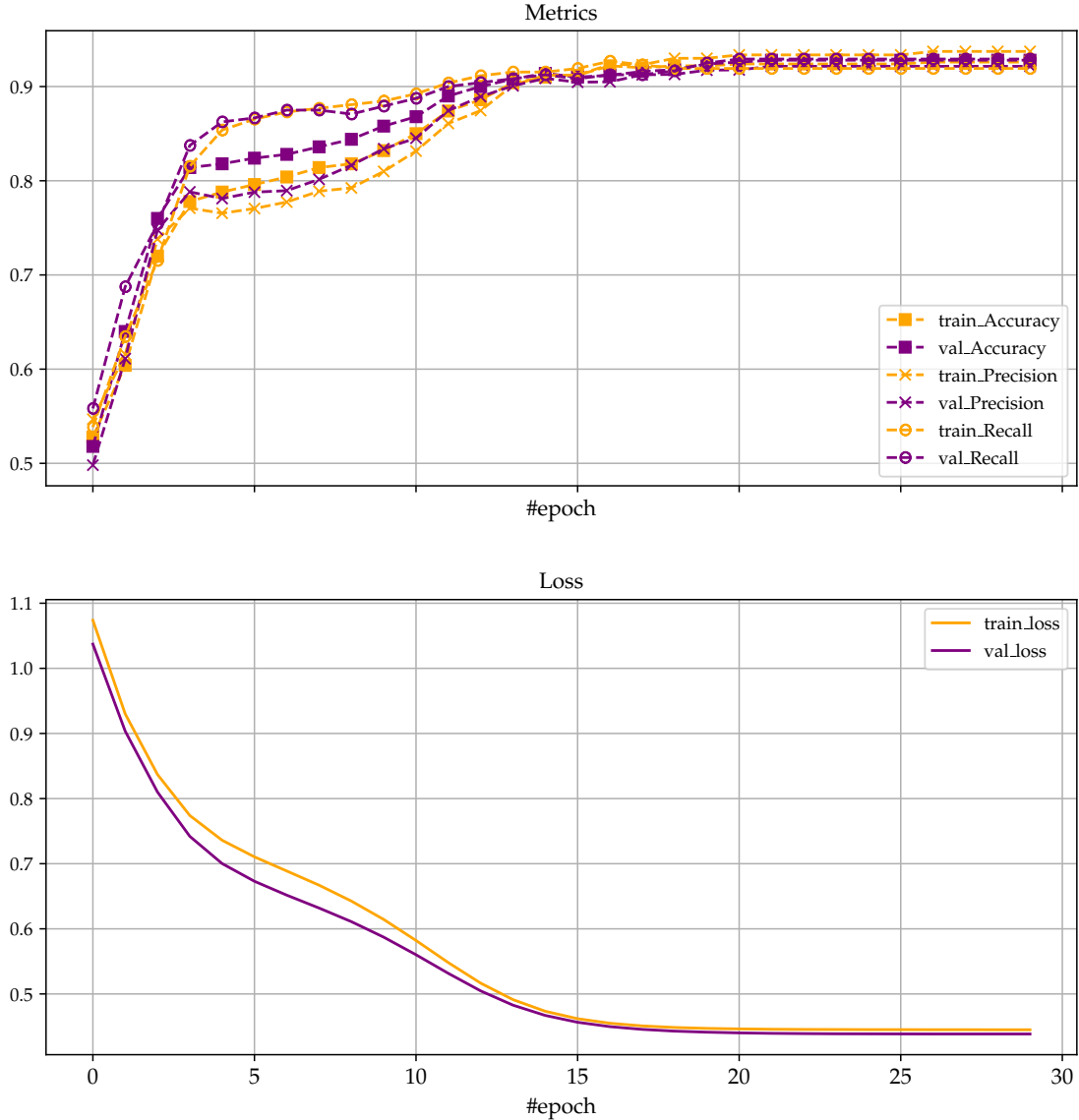


Figure 7.3: The metrics for the training example. The stagnation of the metrics and L show that the optimization process converged to a local minimum. This is of course also visible in $|\nabla L|$ converging to 0.

We make two observations which are unexpected from the perspective of classical machine learning: recall and precision are very close once training is complete and loss is almost equal for the training and validation sets. Note especially that validation loss is always (slightly) smaller than training loss although the classifier was never trained on the validation set.

precision and recall being almost equal hints to the fact that our network has no *bias*, e.g. a trainable parameter which is added to the value f of the quantum

circuit before the activation function is applied. A bias would enable us to manually set a higher threshold for a data point to be interpreted as positive or negative. This would enable us to increase precision at the cost of lower recall or vice versa; a tradeoff known as the *precision/recall tradeoff*¹⁰.

The fact that validation loss is slightly lower than training loss is more subtle. In classical machine learning, this would be interpreted as a lack of *overfitting*. Overfitting describes a scenario in which a classifier is complex enough to learn patterns in the training set that are not present in the validation set. Since the training set is used for training, ongoing training leads to the classifier learning unwanted patterns which are specific to the training set that is used; this means that the training loss value becomes smaller than validation loss. The opposite scenario is called *underfitting*, where a classifier is not complex enough to learn the patterns necessary for successful classification¹¹.

Validation loss being slightly less than training loss may occur by chance due to the specific training and validation sets. Validation loss and training loss being roughly equal when training converges would be interpreted as an ideal fit or underfitting, depending on how good the metrics are. The question if a quantum classifier is not prone to overfitting, if it is just ideally suited for the sklblobs dataset or, conversely, if it is prone to underfitting, will be addressed in Section 11.

7.2. The Gradient of the Quantum Classifier

The components of the gradient are shown on a logarithmic scale in Figure 7.4.

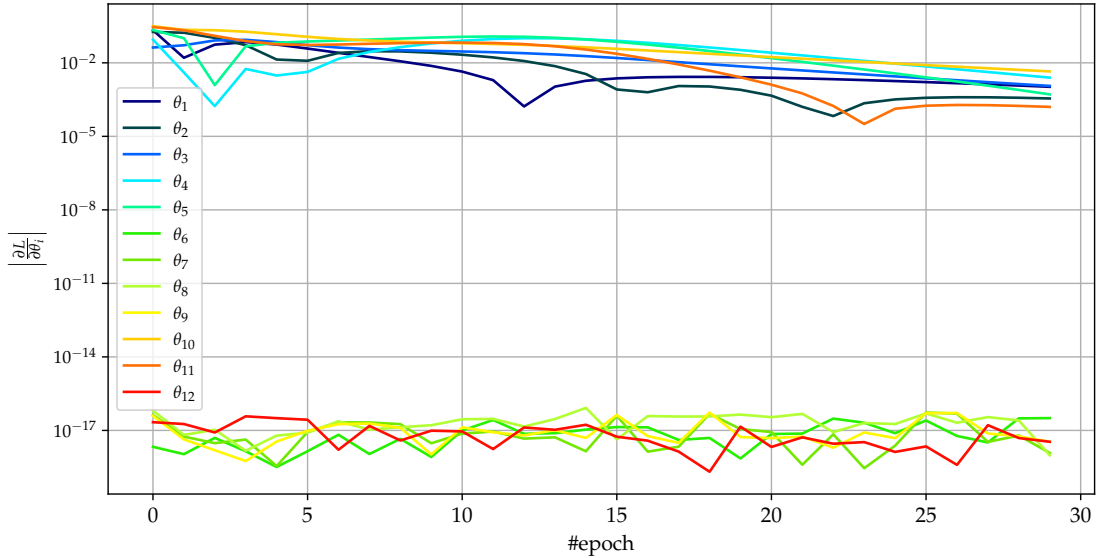


Figure 7.4: The components of the gradient for our training example. The θ_i are the parameters of the Rot-gates. Since one gate takes three parameters, θ_1 to θ_6 belong to the first layer while θ_7 to θ_{12} belong to the second layer.

The plot shows that the derivatives with respect to θ_1 to θ_5 , θ_{10} and θ_{11} decrease in magnitude, as we would expect if the algorithm converges to a local minimum. θ_6 to θ_9 and θ_{12} however do not follow this pattern: Their respective derivatives

¹⁰See Ref. [11], Chapter 3: *Performance Measures*

¹¹See Ref. [11], in Chapter 1: *Overfitting the Training Data and Underfitting the Training Data*

are many orders of magnitude smaller and do not decrease consistently. Their magnitude raises the question if these derivatives have non-zero values because machine precision is reached. Since the parameters themselves take values $\theta_j \approx 1$, these parameters are, effectively, not trained during gradient descent. We will refer to parameters which are not trained as *dead parameters*. In relation to the size of the quantum circuit, the number of dead parameters in this example is large: almost half of our parameters and thus a lot of flexibility of the quantum circuit is lost (refer to Figure 7.5 for details). The cause of dead parameters will be investigated in Section 8.

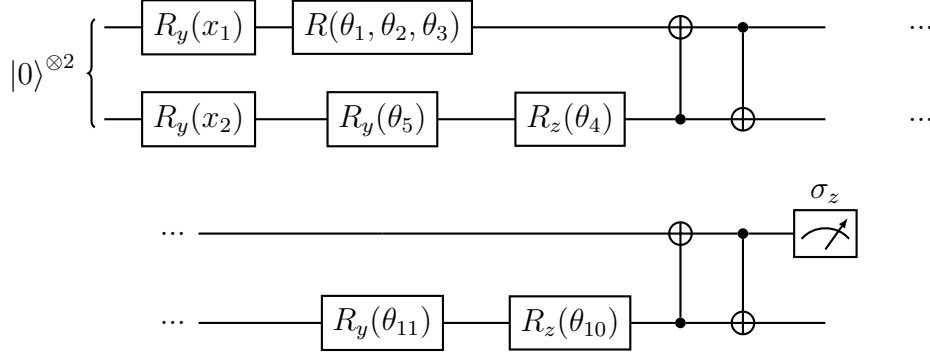


Figure 7.5: The example circuit. Since θ_6 , θ_7 , θ_8 , θ_9 and θ_{12} are dead, the circuit that is effectively trained is not the one shown in Figure 5.3 but the one given here (this follows from the decomposition of R -gates presented in Equation 2.11).

8. Dead parameters and Entanglement

To see why, as shown in Section 7.2, some parameters of a quantum circuit may die, we will investigate the mathematical structure of entanglement which uses CNOT gates. We will furthermore compare entanglement using either CNOT layers or Ising time gates.

8.1. The Cause of Dead Parameters

The output of the circuit is given by the expectation value

$$\begin{aligned}
 f &= \langle \sigma_z^{(0)} \rangle = \langle 0|^{\otimes N} \text{QC} |0\rangle^{\otimes N} \\
 &= \langle 0|^{\otimes N} \left[U_{\text{in}}^\dagger(\vec{x}) \left(\prod_{\text{layers}} U(\theta) \right)^\dagger \right] \sigma_z^{(0)} \left[\left(\prod_{\text{layers}} U(\theta) \right) U_{\text{in}}(\vec{x}) \right] |0\rangle^{\otimes N} \quad (8.1)
 \end{aligned}$$

where QC denotes the quantum circuit and

$$U(\theta) = R^{\otimes N} \text{ENT}. \quad (8.2)$$

We substitute a CNOT layer for ENT and set $N = 2$ to investigate the circuit used in Section 7 and prove why the aforementioned parameters die.

8.1.1. The parameters θ_7 , θ_8 and θ_9

The last gates in the circuit are the entanglement gates in the last layer. The corresponding ENT gate is given by

$$\begin{aligned} \text{ENT} &= \text{CNOT}(c=0, i=1) \text{CNOT}(c=1, i=0) \\ &\doteq \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 1 & & \\ & & 1 \\ & 1 & \end{pmatrix} = \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix}. \end{aligned} \quad (8.3)$$

Writing out the application of this ENT gate to the N -qubits state in terms of Equation 8.1 (“from the inside out”) results in

$$\begin{aligned} &\text{ENT}^\dagger (\sigma_z \otimes I) \text{ENT} \\ &= \text{CNOT}(0, 1)^\dagger \text{CNOT}(1, 0)^\dagger (\sigma_z \otimes I) \text{CNOT}(1, 0) \text{CNOT}(0, 1) \\ &\doteq \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 1 & & \\ & 1 & \\ & & -1 \end{pmatrix} \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & & \\ & -1 & \\ & & 1 \\ & & & -1 \end{pmatrix} \doteq I \otimes \sigma_z, \end{aligned} \quad (8.4)$$

which is separable. This means that the last rotation gate on wire 0 will cancel out since quantum gates are unitary: The application of the last layer results in

$$\begin{aligned} U^\dagger (\sigma_z \otimes I) U &= (\text{ENT} R^{\otimes 2})^\dagger (\sigma_z \otimes I) (\text{ENT} R^{\otimes 2}) \\ &= (R^{\otimes 2})^\dagger \underbrace{\text{ENT}^\dagger (\sigma_z \otimes I) \text{ENT}}_{=I \otimes \sigma_z} R^{\otimes 2} \\ &= (R_{7:9}^\dagger \otimes R_{10:12}^\dagger) (I \otimes \sigma_z) (R_{7:9} \otimes R_{10:12}) \\ &= (R_{7:9}^\dagger I R_{7:9}) \otimes (R_{10:12}^\dagger \sigma_z R_{10:12}) \\ &= I \otimes (R_{10:12}^\dagger \sigma_z R_{10:12}) =: I \otimes \tilde{\sigma} \end{aligned} \quad (8.5)$$

where $R_{i:j}$ denotes the rotation gate $R(\theta_i, \dots, \theta_j)$. This explains why one rotation gate is dead (see again Figure 7.5).

8.1.2. The parameter θ_{12}

The parameter θ_{12} dies because the expression $\tilde{\sigma} = R_{10:12}^\dagger \sigma_z R_{10:12}$ in Equation 8.5 does not depend on θ_{12} . Evaluating it using the definition of the rotation gate (Equation 2.11) and substituting an arbitrary diagonal matrix D for σ_z yields

$$\begin{aligned} \tilde{D} &= R_{10:12}^\dagger \begin{pmatrix} D_i & \\ & D_j \end{pmatrix} R_{10:12} \\ &\doteq \begin{pmatrix} D_i \cos^2\left(\frac{\theta_{11}}{2}\right) + D_j \sin^2\left(\frac{\theta_{11}}{2}\right) & e^{i\theta_{10}} \cos\left(\frac{\theta_{11}}{2}\right) \sin\left(\frac{\theta_{11}}{2}\right) (D_j - D_i) \\ e^{-i\theta_{10}} \cos\left(\frac{\theta_{11}}{2}\right) \sin\left(\frac{\theta_{11}}{2}\right) (D_j - D_i) & D_j \cos^2\left(\frac{\theta_{11}}{2}\right) + D_i \sin^2\left(\frac{\theta_{11}}{2}\right) \end{pmatrix}. \end{aligned} \quad (8.6)$$

8.1.3. The parameter θ_6

The parameter θ_6 dies in the same way θ_{12} dies. This becomes clear when we write out the first layer in the quantum circuit. We use the result for the last layer (Equation 8.5) and evaluate the complete quantum circuit

$$\begin{aligned} \text{QC} &= (\text{ENT} R_{1:3} \otimes R_{4:6})^\dagger (I \otimes \tilde{\sigma}) (\text{ENT} R_{1:3} \otimes R_{4:6}) \\ &= (R_{1:3} \otimes R_{4:6})^\dagger \text{ENT}^\dagger (I \otimes \tilde{\sigma}) \text{ENT} (R_{1:3} \otimes R_{4:6}). \end{aligned} \quad (8.7)$$

Using

$$\text{ENT}^\dagger (I \otimes \tilde{\sigma}) \text{ENT} \doteq \begin{pmatrix} \tilde{\sigma}_{00} & 0 & \tilde{\sigma}_{01} & 0 \\ 0 & \tilde{\sigma}_{11} & 0 & \tilde{\sigma}_{10} \\ \tilde{\sigma}_{10} & 0 & \tilde{\sigma}_{11} & 0 \\ 0 & \tilde{\sigma}_{01} & 0 & \tilde{\sigma}_{00} \end{pmatrix} \quad (8.8)$$

and

$$R_{1:3} \otimes R_{4:6} \doteq \begin{pmatrix} r_{00}I & r_{01}I \\ r_{10}I & r_{11}I \end{pmatrix} \begin{pmatrix} R_{4:6} & 0 \\ 0 & R_{4:6} \end{pmatrix} \quad (8.9)$$

(where we omit the index $_{1:3}$ and write I for the unit matrix $I^{2 \times 2}$ for brevity), we find

$$\begin{aligned} \text{QC} &\doteq \begin{pmatrix} R_{4:6}^H & 0 \\ 0 & R_{4:6}^H \end{pmatrix} \begin{pmatrix} r_{00}^*I & r_{10}^*I \\ r_{01}^*I & r_{11}^*I \end{pmatrix} \begin{pmatrix} \tilde{\sigma}_{00} & 0 & \tilde{\sigma}_{01} & 0 \\ 0 & \tilde{\sigma}_{11} & 0 & \tilde{\sigma}_{10} \\ \tilde{\sigma}_{10} & 0 & \tilde{\sigma}_{11} & 0 \\ 0 & \tilde{\sigma}_{01} & 0 & \tilde{\sigma}_{00} \end{pmatrix} \\ &\quad \times \begin{pmatrix} r_{00}I & r_{01}I \\ r_{10}I & r_{11}I \end{pmatrix} \begin{pmatrix} R_{4:6} & 0 \\ 0 & R_{4:6} \end{pmatrix} \\ &= \begin{pmatrix} R_{4:6}^H & 0 \\ 0 & R_{4:6}^H \end{pmatrix} \begin{pmatrix} D_0 & 0 & D_4 & 0 \\ 0 & D_2 & 0 & D_6 \\ D_1 & 0 & D_5 & 0 \\ 0 & D_3 & 0 & D_7 \end{pmatrix} \begin{pmatrix} R_{4:6} & 0 \\ 0 & R_{4:6} \end{pmatrix} \\ &= \begin{pmatrix} R_{4:6}^H \begin{pmatrix} D_0 & 0 \\ 0 & D_2 \end{pmatrix} R_{4:6} & R_{4:6}^H \begin{pmatrix} D_4 & 0 \\ 0 & D_6 \end{pmatrix} R_{4:6} \\ R_{4:6}^H \begin{pmatrix} D_1 & 0 \\ 0 & D_3 \end{pmatrix} R_{4:6} & R_{4:6}^H \begin{pmatrix} D_5 & 0 \\ 0 & D_7 \end{pmatrix} R_{4:6} \end{pmatrix} \end{aligned} \quad (8.10)$$

where $D_i = \sum \tilde{\sigma}_{jk} r_{lm}$.

In Equation 8.6, we showed that expressions of the form $R_{i:j}^H D R_{i:j}$ do not depend on the last parameter θ_j ; thus the quantum circuit does not depend on θ_6 .

8.2. Entanglement Comparison

In the previous section, we showed that 5 out of 12 parameters of the quantum circuit presented in Figure 5.3 are dead parameters. This is due to the CNOT-gate being represented by a sparse matrix; it contains only one component in every column which is 1. Because of this, effects like $\text{ENT}^\dagger(I \otimes \sigma_z)\text{ENT}$ being separable can occur.

For larger N , the products of ENT-gates and the σ_z -expectation values of various Qubits are separable as well; consider for example

- $N = 3$:

$$\text{ENT}^\dagger (\sigma_z \otimes I \otimes I) \text{ENT} = I \otimes \sigma_z \otimes I \quad (8.11)$$

$$\text{ENT}^\dagger (I \otimes \sigma_z \otimes I) \text{ENT} = \sigma_z \otimes I \otimes \sigma_z \quad (8.12)$$

$$\text{ENT}^\dagger (I \otimes I \otimes \sigma_z) \text{ENT} = \sigma_z \otimes \sigma_z \otimes I \quad (8.13)$$

- $N = 4$:

$$\text{ENT}^\dagger (\sigma_z \otimes I \otimes I \otimes I) \text{ENT} = \sigma_z \otimes \sigma_z \otimes I \otimes I \quad (8.14)$$

$$\text{ENT}^\dagger (I \otimes \sigma_z \otimes I \otimes I) \text{ENT} = (\sigma_z R_y(2\pi)) \otimes I \otimes \sigma_z \otimes I \quad (8.15)$$

$$\text{ENT}^\dagger (I \otimes I \otimes \sigma_z \otimes I) \text{ENT} = \sigma_z \otimes I \otimes I \otimes \sigma_z \quad (8.16)$$

$$\text{ENT}^\dagger (I \otimes I \otimes I \otimes \sigma_z) \text{ENT} = \sigma_z \otimes I \otimes I \otimes I \quad (8.17)$$

Therefore, we use Ising time gates e^{-iHt} for entanglement. These are not sparse and thus make dead parameter significantly more unlikely. For an example, see Figure 8.1.

$$e^{-iHt} = \begin{pmatrix} -0.46 - 0.16i & -0.00 + 0.57i & -0.00 + 0.37i & 0.53 - 0.17i \\ 0.00 + 0.57i & -0.46 + 0.16i & 0.53 + 0.17i & -0.00 + 0.37i \\ 0.00 + 0.37i & 0.53 + 0.17i & -0.46 + 0.16i & 0.00 + 0.57i \\ 0.53 - 0.17i & -0.00 + 0.37i & 0.00 + 0.57i & -0.46 - 0.16i \end{pmatrix}$$

Figure 8.1: An example Ising time gate for two qubits where $t = 10$. It is not sparse and hence less likely to result in separable ENT-layers.

9. Encoding Comparison

As we discussed in Section 5.2, all non-linearity in a quantum circuit depends on how the input data is encoded in the state of the N -qubit system. In this section, we compare different methods of encoding the input data in the quantum circuit based on the decision boundaries they converge to and the corresponding performance metrics. We test each variant using a quantum circuit with four layers and Ising time gates for entanglement, acting on two or four qubits. Each variant is trained on the sklblobs, sklmoons, sklcircles and lines datasets (see Figure 9.1). We will train 30 classifiers using COBYLA.

We will introduce all encoding methods first and then compare them. Our list of possible encoding schemes is not comprehensive; other authors have proposed similar as well as more complicated methods of encoding data into a quantum circuit. Examples may be found in Refs. [7, 8, 20].

9.1. Angle encoding

The *angle encoding* has already been used in Figure 5.3: The input features are used as rotation angles for R_y gates. The density matrix for a single qubit after encoding is given by

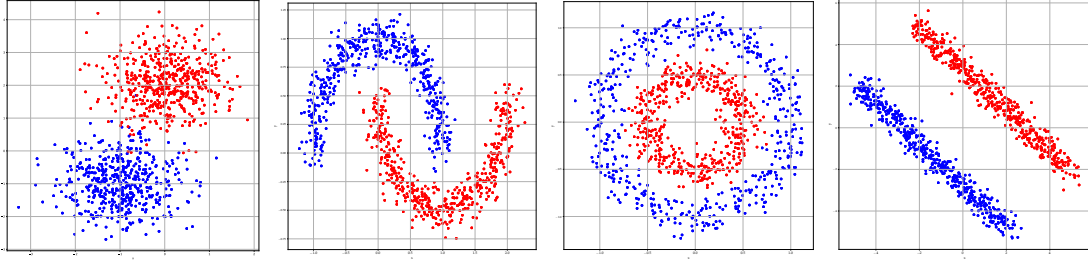


Figure 9.1: The datasets we use for comparing the different encoding schemes from left to right: sklblobs, sklmoons, sklcircles and lines. The first three are generated using the `sklearn.datasets` module of the SciKit-learn library [19]. The lines dataset is self-made.

$$\begin{aligned}
\rho_{\text{in}}^{(j)} &= |\psi_{\text{in}}(x)\rangle \langle \psi_{\text{in}}(x)| = R_y(x) |0\rangle \langle 0| R_y^\dagger(x) \\
&= \left[\cos\left(\frac{x}{2}\right) |0\rangle + \sin\left(\frac{x}{2}\right) |1\rangle \right] \left[\cos\left(\frac{x}{2}\right) \langle 0| + \sin\left(\frac{x}{2}\right) \langle 1| \right] \\
&= \frac{1}{2} [I + \sin(x)X + \cos(x)Z].
\end{aligned} \tag{9.1}$$

The output of the circuit is thus non-linear in the input features x_1, x_2 because they are encoded using trigonometric functions. Because the complete density matrix of an N -qubit system is given by

$$\rho_{\text{in}} = \bigotimes_{j=0}^{N-1} \rho_{\text{in}}^{(j)} = \bigotimes_{j=0}^{N-1} \frac{1}{2} [I + \sin(x_j)X + \cos(x_j)Z], \tag{9.2}$$

the quantum circuit behaves like a series expansion in products of trigonometric functions.

The corresponding quantum circuit is shown in Figure 9.2. This scheme is extended to four-qubit systems by encoding x_1 in wire 2 and x_2 in wire 3.

Before encoding, the input data is normalized to a range of $[0, 2\pi]$.

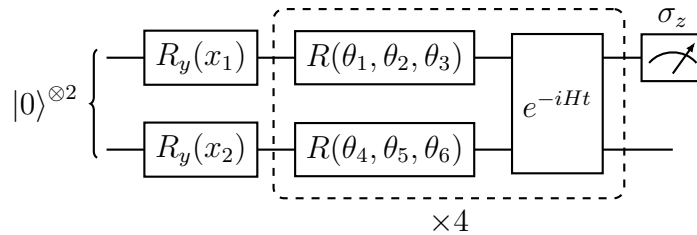


Figure 9.2: The quantum circuit we use for testing the angle encoding. It is inspired by Ref. [6]. As discussed in sections 5.3 and 5.4, we use four layers and substitute the CNOT-layer with an Ising time gate.

9.2. Inverse angle encoding

Inverse angle encoding is similar to angle encoding, in that it also uses R_y gates as encoding gates. The input features are however not used as rotation angles θ for the gates; instead, we set $\theta = \arccos(x)$ [12]. The density matrix is

$$\begin{aligned}
\rho_{\text{in}}^{(j)} &= R_y(\arccos(x)) |0\rangle \langle 0| (R_y)^\dagger(\arccos(x)) = \frac{1}{2} [I + xZ + \sqrt{1-x^2}X] \\
\Rightarrow \rho_{\text{in}} &= \bigotimes_{j=1}^{N-1} \frac{1}{2} [I + x_j Z + \sqrt{1-x_j^2} X].
\end{aligned} \tag{9.3}$$

The circuit thus behaves like a series expansion in powers of x . As with angle encoding, we simulate a 4-qubit circuit by encoding x_1 in wire 2 and x_2 in wire 3. Using such a setup, the circuit is theoretically able to approximate every polynomial for large enough N [12]. Consider for example encoding of one-dimensional input x : If $N = 3$, Equation 9.3 contains terms up to the third order in x .

Before encoding, the input data is normalized to a range of $[-1, 1]$.

9.3. Dense angle encoding

The previous encoding schemes do not exploit every degree of freedom a qubit provides. Since $\mathcal{H}_2 = \mathbb{C}^2$, the coefficients of $|\psi_{\text{in}}\rangle$ may be complex numbers; this is not the case for angle encoding or inverse angle encoding. *Dense angle encoding* uses this degree of freedom to encode two features in one qubit [20] by applying an R_y gate and a P gate during preparation (see Figure 9.3).

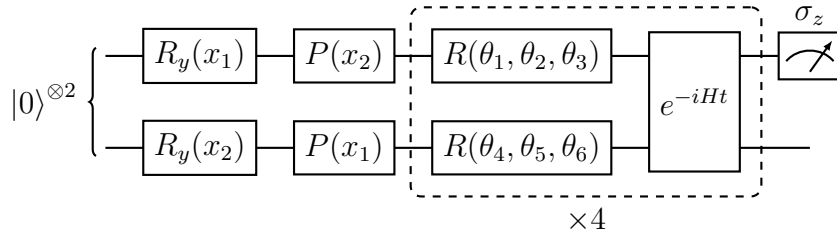


Figure 9.3: The quantum circuit we use for testing the dense angle encoding.

Calculating the density matrix results in

$$\begin{aligned}
\rho_{\text{in}}^{(j)} &= P(x_2) R_y(x_1) |0\rangle \langle 0| R_y(x_1)^\dagger P^\dagger(x_2) \\
&= \frac{1}{2} [I + \sin(x_1) \cos(x_2) X + \sin(x_1) \sin(x_2) Y + \cos(x_1) Z] \\
\Rightarrow \rho_{\text{in}} &= \bigotimes_{j=0}^{N-1} \frac{1}{2} [I + \sin(x_j) \cos(x_j) X + \sin(x_j) \sin(x_j) Y + \cos(x_j) Z].
\end{aligned} \tag{9.4}$$

Extending this encoding scheme to 4 qubits is done analogously to the other encoding schemes.

Before encoding, the input data is normalized to a range of $[0, 2\pi]$.

9.4. Double angle encoding

The authors of Ref. [12] present an encoding scheme which we call *double angle encoding*. Only one feature is encoded into one qubit but, by using two gates for one feature, double angle encoding creates a density matrix which contains the

input feature up to its third power. The circuit is shown in Figure 9.4. This setup yields the density matrix

$$\begin{aligned}
\rho_{\text{in}}^{(j)} &= R_z(\arccos(x^2)) R_y(\arcsin(x)) |0\rangle \langle 0| R_y^\dagger(\arcsin(x)) R_z^\dagger(\arccos(x^2)) \\
&= \frac{1}{2} \left[I + \sqrt{1-x^2} Z + x^3 X - ix\sqrt{1-x^2} Y \right] \\
\Rightarrow \rho_{\text{in}} &= \bigotimes_{j=0}^{N-1} \frac{1}{2} \left[I + x_j^3 X - ix_j \sqrt{1-x_j^2} Y + \sqrt{1-x_j^2} Z \right].
\end{aligned} \tag{9.5}$$

This encoding scheme is thus similar to inverse angle encoding, it however provides different basis functions for the series which is the circuit output.

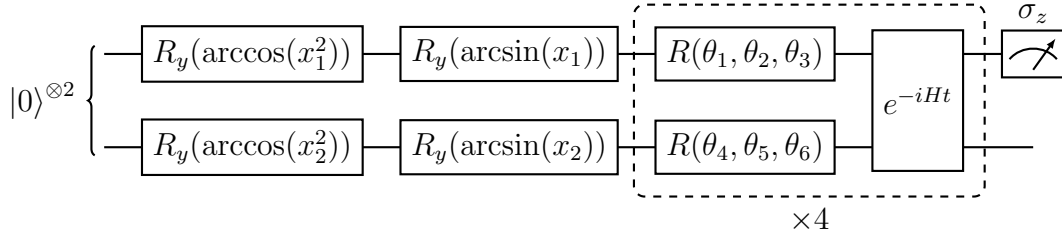


Figure 9.4: The quantum circuit we use for testing the double angle encoding.

Before encoding, the input data is normalized to a range of $[-1, 1]$.

9.5. Wavefunction encoding

Inspired by Ref. [20], we introduce an encoding scheme called *wavefunction encoding*. It encodes data points \vec{x} with d features into a d -level quantum system by preparing the state

$$|\psi_{\text{in}}\rangle = \frac{1}{|\vec{x}|} \sum_{j=0}^{d-1} x_j |j\rangle. \tag{9.6}$$

This differs from the definition of wavefunction encoding of Ref. [20] in the normalization of $|\psi_{\text{in}}\rangle$: The authors of Ref. [20] choose a normalization factor of $1/|\vec{x}|^2$ whereas we chose $1/|\vec{x}|$. We made this choice because we did not include a bias in our quantum classifier, therefore we lack a degree of freedom to adapt to a varying magnitude of $|\psi_{\text{in}}\rangle$.

In our case, we set $d = 2$ and find the density matrix

$$\rho_{\text{in}} = \frac{1}{|\vec{x}|^2} (x_1^2 |0\rangle \langle 0| + x_1 x_2 X + x_2^2 |1\rangle \langle 1|) \doteq \frac{1}{|\vec{x}|^2} \begin{pmatrix} x_1^2 & x_1 x_2 \\ x_1 x_2 & x_2^2 \end{pmatrix}. \tag{9.7}$$

The basis functions correspond immediately to the features of the input, however, the encoding is independent of the magnitude of \vec{x} . The consequences of this are visible in the decision boundaries (for example Figure A.2).

9.6. Encoding scheme comparison

The different encoding methods are compared in Figure 9.5. A few comments on the results:

- Not every encoding scheme profits from an increasing number of qubits. Double angle encoding increases only lightly in accuracy when classifying sklblobs and the performance of wavefunction encoding does not change. Angle encoding and inverse angle encoding notably improve their accuracy for sklblobs. A possible explanation for this is the limitation due to the available basis functions: When the number of qubits is increased, The decision boundaries naturally become more complex. They are however limited in orientation. Section 11 goes into more detail on this topic.
- Angle encoding performs very well despite its simplicity relative to the other encoding schemes we present. It also has a short runtime. For four qubits, one epoch using COBYLA takes approximately 15 min using angle encoding but approximately 30 min using dense angle encoding. For higher numbers of qubits, angle encoding may thus be preferable when computation time becomes prohibitive for other quantum circuit architectures.
- Wavefunction encoding is not able to classify the datasets as it only reaches 50 % accuracy for datasets other than sklblobs. Here, it still underperforms with approximately 70 % accuracy. The reason for this is that only the polar angle of a data point $\vec{x} \in \mathbb{R}^2$ may be encoded and not its radius (representing \vec{x} in polar coordinates). This flaw in wavefunction encoding may be countered by normalizing with the denominator $|\vec{x}|^2$ (see Equation 9.6), however the quantum classifier then needs a trainable bias.

Decision boundary plots for various encoding schemes and datasets are shown in Appendix A.

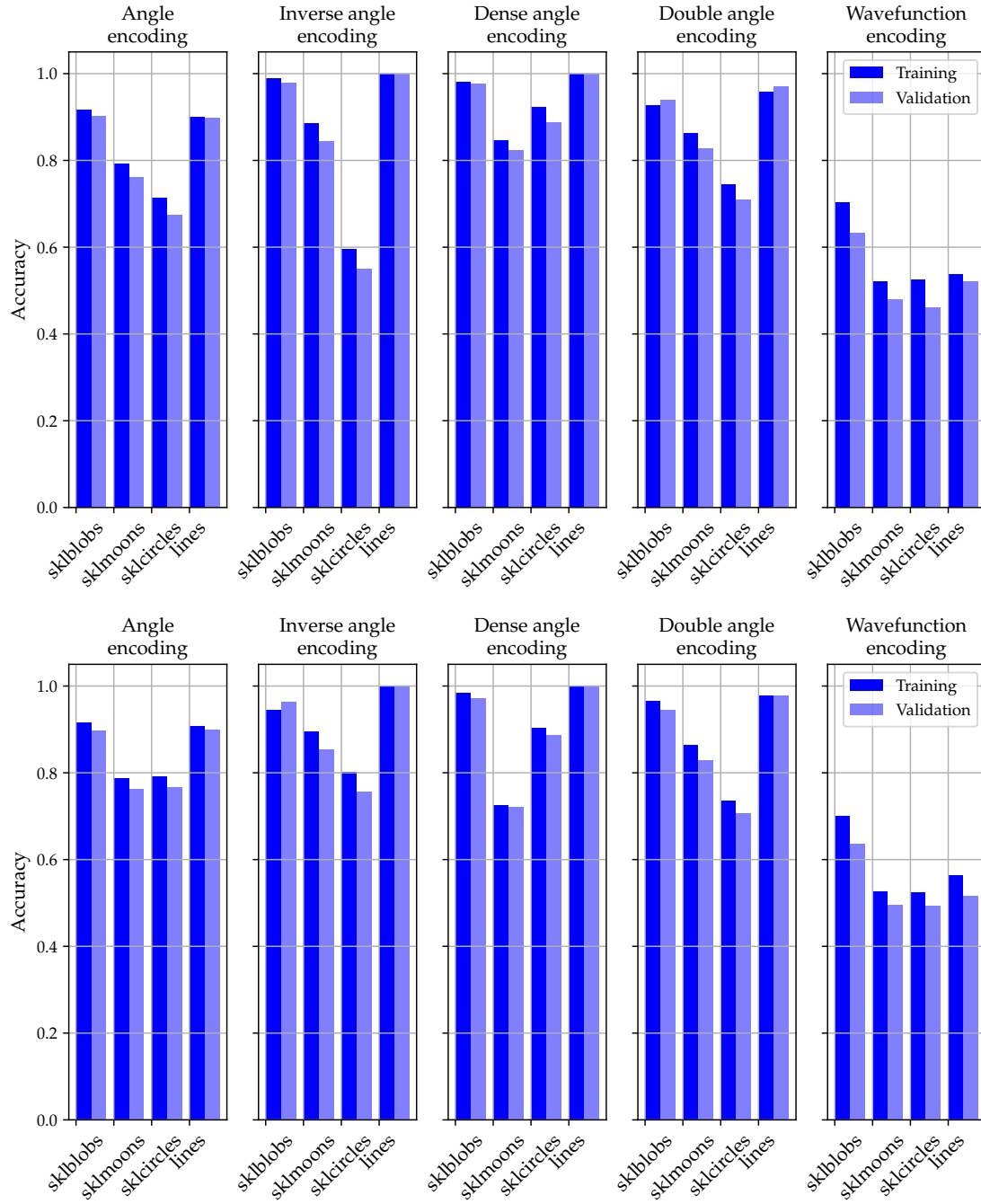


Figure 9.5: The comparison of encoding methods for two (top) and four (bottom) qubits. The lightly colored bars are the metrics of the validation set.

10. Classical Preprocessing

Until now, the modifications we made in order to reach better classification results were internal to the quantum circuit. We investigated the best method of entanglement, discussed how many layers are necessary and tested different methods of encoding. In this section, we introduce a method which is external to the quantum circuit: *classical preprocessing*. By this we refer to a transformation

$$p : \Gamma \rightarrow \Gamma \quad (10.1)$$

of the input data which is executed on classical hardware. p depends on trainable parameters. Such a transformation will of course not change the decision boundaries the quantum circuit is able to implement; instead, it enables us to transform the dataset such that it better fits the given quantum circuit and its decision boundaries. We will introduce three preprocessing transformations: rotations, translations and their composition.

We did not investigate feature engineering, which could be interpreted as preprocessing. Carefully chosen features could have a large impact on the performance of quantum classifiers since the quantum circuits we investigated are not complex enough to carry out feature engineering internally. The performance of quantum classifiers with more features is better such that feature engineering is another promising approach.

10.1. Rotations

One possible way is to introduce rotations of the dataset. We choose this approach because the decision boundaries a quantum circuit is able to implement are restricted by the basis functions the chosen encoding provides. The encoding schemes provide diverse decision boundaries, however they are often restricted to specific orientations in the two-dimensional feature space $\Gamma \subset \mathbb{R}^2$.

To implement a rotation independent of the coordinate system of the dataset, we rotate around the mean value of the input, i.e.

$$p_{\text{rot}} : \vec{x} \mapsto \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \vec{s} + R(\vec{x} - \vec{s}) \quad (10.2)$$

where $\vec{s} = \frac{1}{m} \sum \vec{x}$ is the mean of the input and m is the number of input data points. R is a rotation matrix whose corresponding angle φ is a trainable parameter of the quantum circuit. In our implementation, this is realized by always passing the complete training- or validation set as input to the quantum circuit; \vec{s} is calculated for the whole input.

The derivative of the quantum classifier with respect to φ is calculable through the parameter-shift rule. During calculation of $\partial L / \partial \varphi$, through the chain rule, the term $\partial f / \partial x_j$ arises. The conditions for applying the parameter-shift rule (which are discussed in Section 4) are fulfilled here: x_j is used as a parameter for gates which are expressible as operator exponentials, therefore, the derivative of L with respect to φ is (remembering the notation introduced in Section 2)

$$\begin{aligned}
\frac{\partial L}{\partial \varphi} &= \frac{1}{N \ln 2} \sum_{\{\vec{x}_t\} \in T} \left(\frac{y_t}{\hat{y}} - \frac{1 - y_t}{1 - \hat{y}} \right) \frac{\partial a}{\partial f} \frac{\partial f}{\partial \varphi} \\
&= \frac{1}{N \ln 2} \sum_{\{\vec{x}_t\} \in T} \left(\frac{y_t}{\hat{y}} - \frac{1 - y_t}{1 - \hat{y}} \right) \frac{\partial a}{\partial f} \left(\frac{\partial f}{\partial p_1} \frac{\partial p_1}{\partial \varphi} + \frac{\partial f}{\partial p_2} \frac{\partial p_2}{\partial \varphi} \right)
\end{aligned} \tag{10.3}$$

where $\partial f / \partial p_j$ is calculable through the parameter-shift rule and

$$\begin{aligned}
\frac{\partial p_1}{\partial \varphi} &= -\sin(\varphi)x_1 + \cos(\varphi)x_2 \\
\frac{\partial p_2}{\partial \varphi} &= -\cos(\varphi)x_1 - \sin(\varphi)x_2.
\end{aligned} \tag{10.4}$$

In our implementation of quantum classifiers, preprocessing using a rotation is enabled by passing `prerot` as value for the `implementation` keyword at initialisation of the `predictor` object.

10.2. Translations

Translations of the dataset are of course given by

$$p_{\text{trans}} : \vec{x} \mapsto \vec{x} + \vec{\eta} \tag{10.5}$$

where $\vec{\eta} = \begin{pmatrix} \eta_1 & \eta_2 \end{pmatrix}^T$ and η_1, η_2 are trainable parameters.

In our implementation of quantum classifiers, preprocessing using a translation is enabled by passing `pretrans` as value for the `implementation` keyword at initialisation of the `predictor` object. The gradient of L may be calculated in a similar fashion to Equation 10.3.

10.3. Composing rotations and translations

The most general transformation is of course a composition of rotation and translation. We choose to define the composed transformation as

$$p_{\text{comp}} = p_{\text{trans}} \circ p_{\text{rot}} : \vec{x} \mapsto \vec{s} + R(\vec{x} - \vec{s}) + \vec{\eta}. \tag{10.6}$$

The order of the transformations in the composition is not important. Rotations and translations do not commute, however, the same transformation can be expressed as either a rotation followed by a translation or vice versa. We chose $p_{\text{comp}} = p_{\text{trans}} \circ p_{\text{rot}}$ because this facilitates the calculation of the gradient.

In our implementation of quantum classifiers, preprocessing using a composition of rotation and translation is enabled by passing `pregalilei` as value for the `implementation` keyword at initialisation of the `predictor` object. The gradient of L may be calculated in a similar fashion to Equation 10.3.

10.4. Comparison of preprocessing schemes

We test all preprocessing schemes using quantum circuits which have four layers. Entanglement is done with Ising time gates. The initial value for φ is a random sample from $[0, 2\pi]$, the initial value for $\vec{\eta}$ is 0. We train the quantum circuits using COBYLA.

We test every preprocessing scheme using both angle encoding and dense angle encoding. We evaluate the different encoding schemes based on the accuracy they reach. Since COBYLA reaches the same minimum of the parameter space very consistently, only the mean value of the accuracy and not its standard deviation is given. The results are shown in figures 10.1 to 10.4.

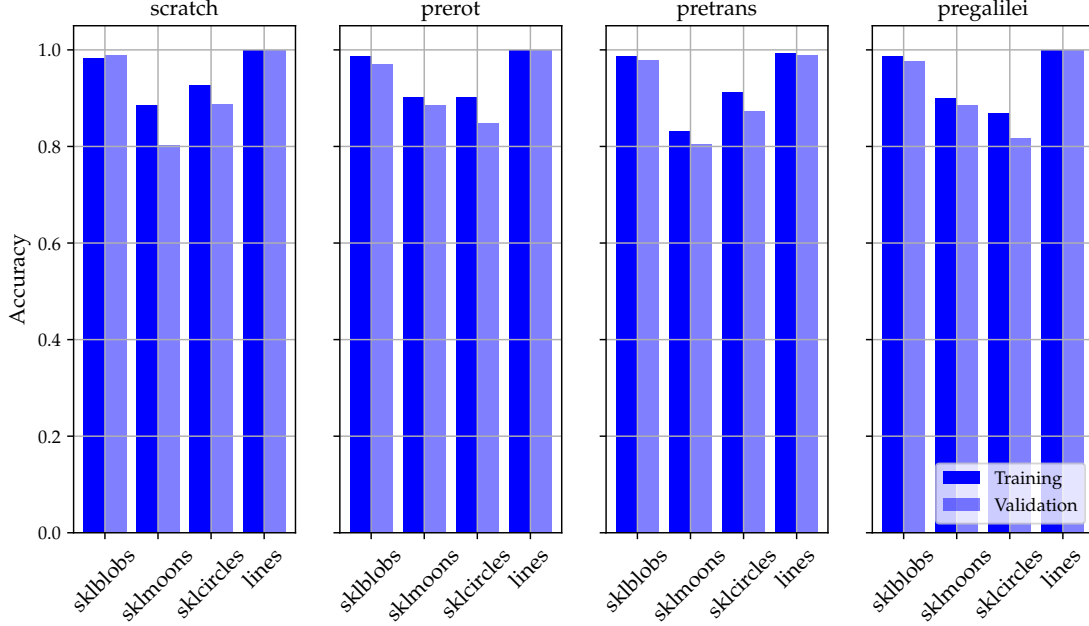


Figure 10.1: The comparison of preprocessing methods with angle encoding for two qubits. The lightly colored bars are the metrics of the validation set.

A few comments on the results:

- The change from two to four qubits increases accuracy in all preprocessing schemes and for all datasets whereas changing the preprocessing scheme did not increase accuracy for every dataset. This suggests that increasing the number of qubits is the most effective way to increase accuracy through increased flexibility and is in contrast to what we concluded in Section 9.6 where adding more qubits did not necessarily have a great impact. The difference to the previous comparison is that here, preprocessing enables the quantum circuit to adapt the dataset to take advantage of the more complex decision boundaries, leading to increased performance.
- Preprocessing through translations rarely offered an advantage (Figure 10.1), with the only exception being a minor increase for the skblobs dataset. A possible explanation is that the encodings we use in this comparison are periodic in the input data, meaning the ability to translate the dataset does not significantly change the decision boundaries which are applied.
- Preprocessing through composition of rotation and translation, which we regarded as the most general type of encoding, did not perform significantly better than pre-rotations and in some cases even performed worse than pre-rotations (Figure 10.2). It improves the accuracy for two qubits

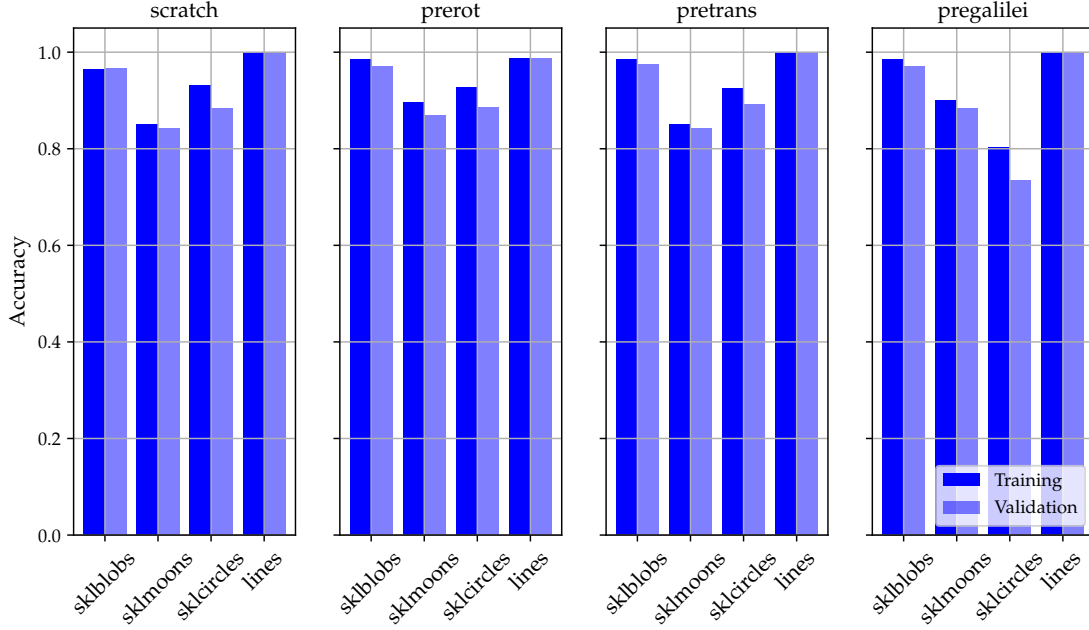


Figure 10.2: The comparison of preprocessing methods with dense angle encoding for two qubits. The lightly colored bars are the metrics of the validation set.

on sklmoons but decreases it on sklcircles (figures 10.1 and 10.2). The inability to increase performance follows from the previous point concerning pre-translations. One possible explanation for worse performance is that more trainable parameters create more local minima of L in the parameter space. The worsening of accuracy is more pronounced for two qubits; for four qubits, the accuracy gains are comparable to the gains through pre-rotations.

- Preprocessing through rotations never significantly decreased performance on a dataset but instead enabled the quantum circuit to reach high accuracy on all datasets for four qubits. For two qubits, the gain in accuracy is most significant for the lines dataset when using angle encoding. In all scenarios we tested, pre-rotations offered the best or comparable performance on every dataset (with the only exception being four qubits with angle encoding for the sklmoons dataset). Preprocessing with rotations thus seems to offer the highest flexibility.
- The sklcircles dataset stands out as being challenging because no preprocessing scheme was able to significantly improve the performance of the **scratch**-implementation. Pre-rotations are a priori unable to improve performance in this case because sklcircles is cylindrically symmetric. Pre-translations can only improve performance if rotationally symmetric decision boundaries are present somewhere in the periodic decision boundary plots; this does not seem to be the case.

Our list of preprocessing methods is of course not complete. Another promising method is the scaling of the dataset. Since some encoding schemes are periodic, this would allow the characteristic length of the decision boundaries (the scale on

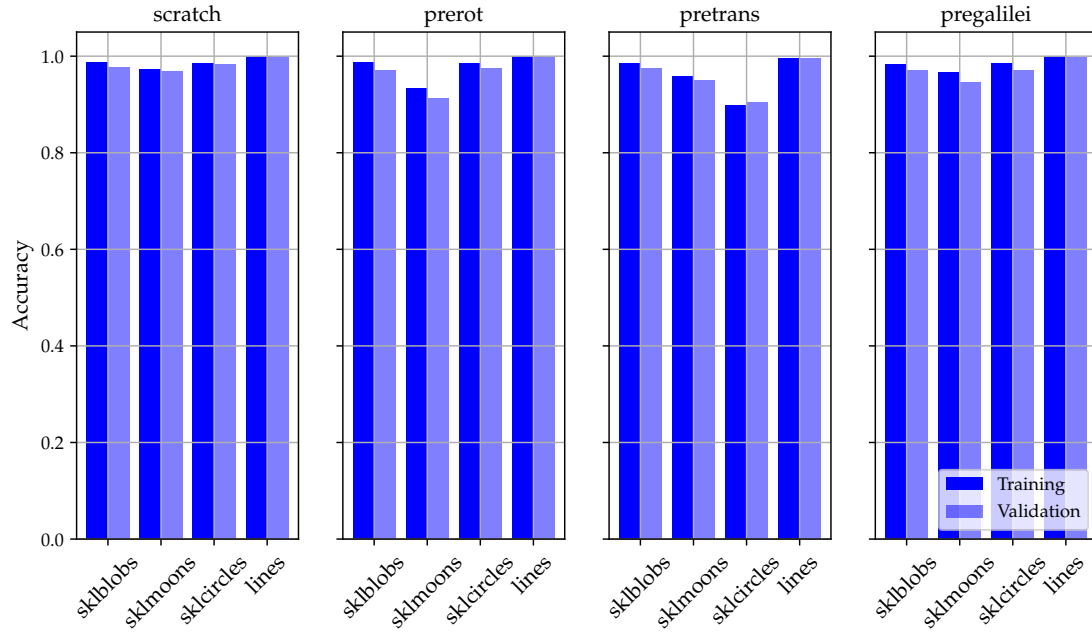


Figure 10.3: The comparison of preprocessing methods with angle encoding for four qubits. The lightly colored bars are the metrics of the validation set.

which decision boundaries vary in the input space Γ) to become smaller relative to the dataset. This could enable the quantum classifier to classify more complex datasets.

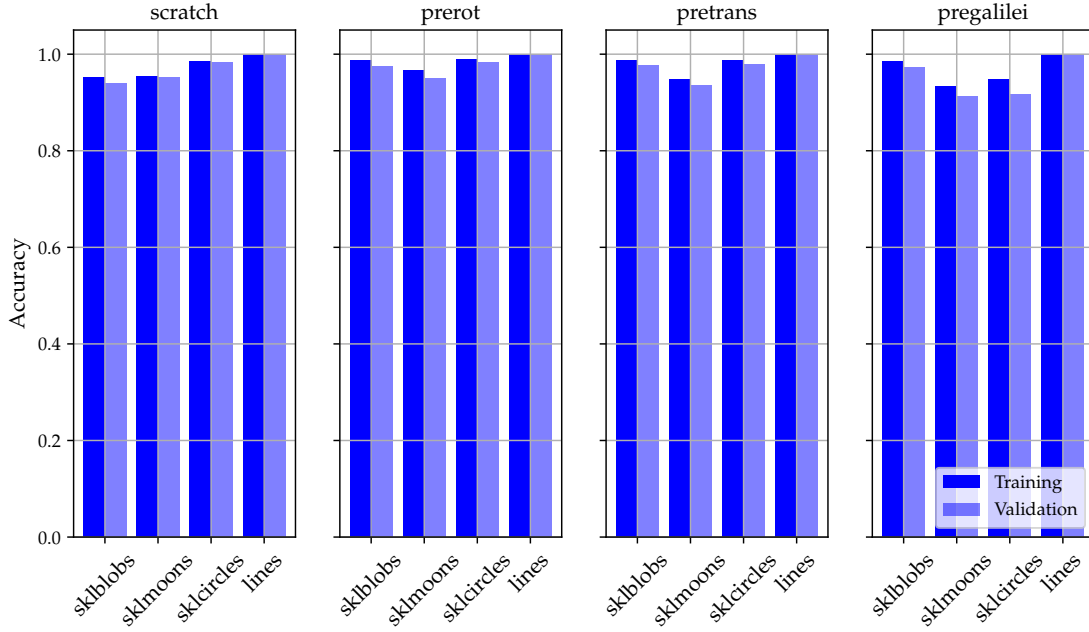


Figure 10.4: The comparison of preprocessing methods with dense angle encoding for four qubits. The lightly colored bars are the metrics of the validation set.

11. On Under- and Overfitting

As mentioned in previous sections, in the following section we discuss the limitations of possible decision boundaries and if a quantum classifier is prone to over- or underfitting. We do this using *decision boundary sampling*: We initialize the parameters of the quantum circuit randomly and simulate it for a grid on which we plot the decision boundaries. We simulate evenly distributed points $\vec{x} \in [0, 2\pi] \times [0, 2\pi]$ on a 200×200 -grid. Results are shown in figures 11.1 and 11.2.

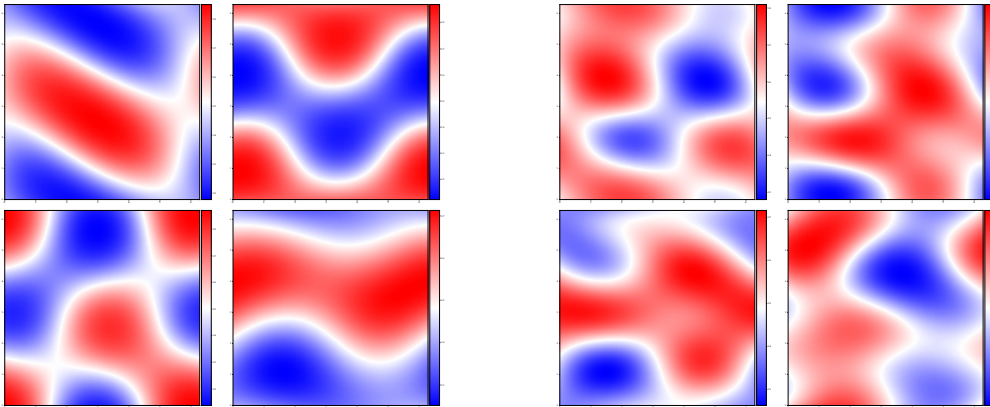


Figure 11.1: Four decision boundaries for angle encoding using two (left) respectively four (right) qubits.

These decision boundaries differ significantly from simple models of classical classifiers. The simplest classifiers on classical hardware are *linear classifiers*, which are only capable of finding linear decision boundaries. Another classical model are

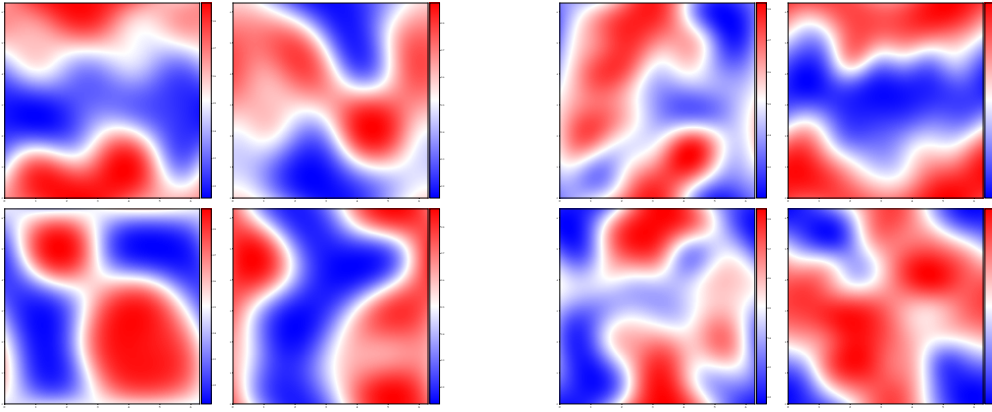


Figure 11.2: Four decision boundaries for dense angle encoding using two (left) respectively four (right) qubits. We want to emphasize that the corresponding classifiers have not been trained; these plots give an overview of the decision boundaries that are possible, not the edge cases which are useful for classifying the datasets we are interested in.

decision trees, whose decision boundaries are drawn by iteratively bisecting existing regions where the model makes a certain prediction¹². The decision boundaries of a quantum classifier are very different because it behaves more like a series expansion than a neural network (as discussed in Section 5.3): For (dense) angle encoding, the basis functions are trigonometric and hence the decision boundaries have trigonometric features. The available decision boundaries are complex compared to classical classifiers, they are however limited in other aspects. For example, it is not easily possible for a quantum classifier to rotate its decision boundaries. This is of course due to the available basis functions: only such decision boundaries can be achieved that can be reached by shuffling the components of the density matrix ρ .

For this reason, our quantum classifiers are not prone to overfitting despite classifying the datasets we investigated well. The output of a quantum classifier is per construction well suited for smooth datasets because the basis functions reflect their structure. The output $f = \text{Tr}(\sigma_z^{(0)} U(\theta) \rho_{\text{in}} U^\dagger(\theta))$ however does not consist of many terms (refer to Section 5.3: For a two-qubit system, $\rho \in \mathbb{C}^{4 \times 4}$). The classifier is simply not complex enough to overfit the data¹³.

The quantum classifiers we studied also offer advantages concerning the size of the dataset used for training. Since the decision boundaries that are possible fit the datasets by construction, the classifier is still able to learn the desired patterns using small dataset. An example is shown in Figure 11.3.

Underfitting is present when training the sklmoons dataset; refer for example to Figure A.1: The classifier trained on sklmoons was not able to learn the patterns in the sklmoons dataset. The other datasets were rarely underfitted.

The way to alleviate underfitting for quantum classifiers is different to classical classifiers. In classical models, the model complexity can often easily be increased. This involves increasing the number of layers of a neural networks or the depth

¹²See [11], Chapter 6: *Decision Trees*

¹³The question whether quantum classifiers are prone to overfitting is addressed in Ref. [12] for inverse angle encoding. The authors argue that the unitarity of the transformations applied to ρ_{in} prevents overfitting. This may be a contributing factor to the other encoding schemes as well as it functions as a regularization constraint. We suspect however that it becomes more important for larger numbers of qubits when ρ_{in} becomes more complex.

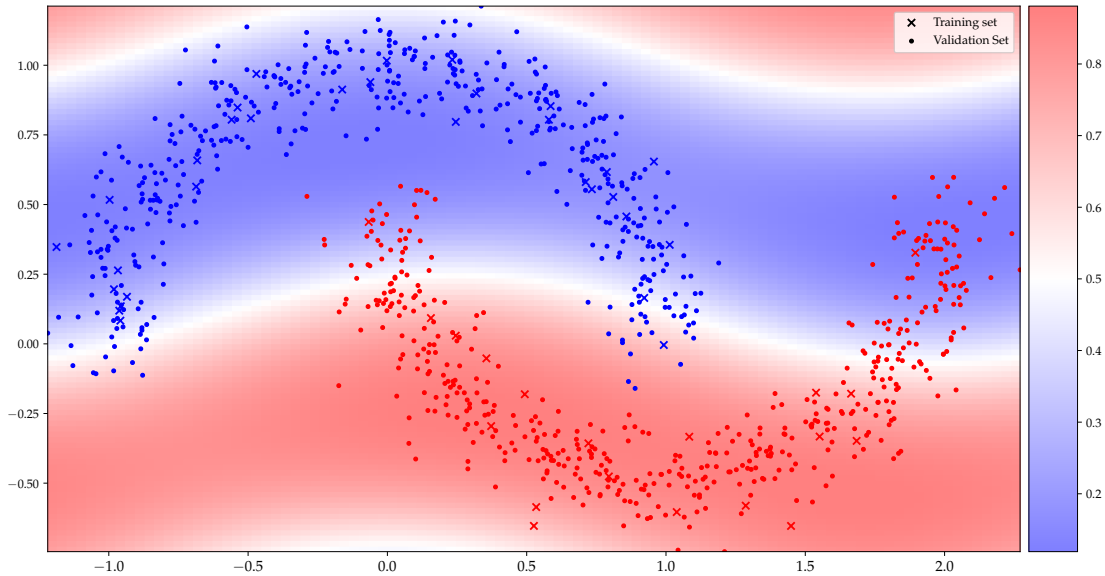


Figure 11.3: An example where a quantum classifier is trained using a small number of data points. The above classifier uses two qubits, four layers and angle encoding. It is trained on 30 training points and validated using 970 data points. The figure shows that the classifier still finds the same decision boundaries and reaches a training accuracy of 0.933. The validation accuracy is 0.837; hence it does not overfit much as, contrary to the expectation for such a small training set compared to the validation set. The decision boundary shows that this design (for quantum circuits) is able to reach high accuracies even for very small datasets.

of a decision tree. We showed in Section 5.3 that the number of layers has little influence on the performance beyond $n_{\text{layers}} = 4$. Using the qubit numbers we simulated, underfitting must be countered with different encoding methods or preprocessing. As briefly discussed in Section 9.2, another way of countering underfitting is to incorporate more qubits in the quantum circuit. An interesting topic for future research is the question if quantum classifiers with increasingly more qubits always profit from different encoding or preprocessing (as we found in Section 10.4 for two or four qubits) or if the growing complexity of ρ_{in} makes this obsolete.

12. Classifying the LHC Olympics Dataset

In this section, we introduce the LHC Olympics (LHCo) dataset and the features we use during training of a quantum classifier. We then compare the results of a quantum classifier and a neural network with comparable complexity which are both trained on the same data.

12.1. The Dataset

We apply the results of the previous sections by training a quantum classifier with a dataset which is relevant to physics research. We choose the *LHC Olympics 2020*

dataset. The LHC Olympics [1] are a community challenge for analysing simulated datasets with the goal of finding BSM events in an SM background. One R&D dataset and three black box dataset are provided. For evaluating the performance of quantum classifiers, we focused on the R&D dataset [21].

The R&D dataset consists of 1M SM events and 100 000 BSM events. In the SM events, two jets were produced through the strong interaction. The BSM events are $pp \rightarrow Z' \rightarrow XY$ events where X and Y decay into quark-antiquark pairs. The particles Z' , X and Y are BSM particles with masses 3.5 TeV, 500 GeV and 100 GeV, respectively. The events were generated using the tools `pythia`, version 8.219 [22, 23] and `DELPHES` [24, 25, 26]. `pythia` is a tool for generating high-energy physics collision events. `DELPHES` simulates the detector response to a collision.

The complete dataset contains detector coordinates (p_T, η, ϕ) for 700 particles for every collision and the corresponding label (“1” for signal or “0” background) in an array with size (1.1 M, 2101). p_T is the transverse momentum, η is the pseudo-rapidity and ϕ is the polar angle in the transverse plane. Because of the size of the feature space in the complete dataset, we use the corresponding high-level features dataset. It is generated by clustering every event into jets. For the two jets with the highest transverse momentum, the quantities

$$\left(p_x^{\text{ji}}, p_y^{\text{ji}}, p_z^{\text{ji}}, m^{\text{ji}}, \tau_1^{\text{ji}}, \tau_2^{\text{ji}}, \tau_3^{\text{ji}}, y\right) \quad (12.1)$$

are given [21]. p_x, p_y, p_z are the 3-momenta, m is the invariant mass and τ_1, τ_2, τ_3 are the n -jettiness variables. y is the label. The superscript “ji” denotes the number of the jet, i.e. “j1” for jet 1 and “j2” for jet 2. We again do a 50/50 training-validation split of the dataset.

12.2. Feature Selection

Since the quantities given above are not independent of each other, we limit ourselves to combinations of m^{ji} and τ_k^{ji} as features. Scatter plots of all combinations are shown in Figure 12.1.

A well-chosen combination of features yields signal and background data points which do not overlap and which are easily separable. Based on these criteria, we choose m^{j2} and τ_1^{j1} as features which we will use to train quantum classifiers.

12.3. Training a Neural Network and a Quantum Classifier

We train multiple classifiers to compare the performance of quantum classifiers and classical classifiers.

Based on the results of sections 5, 9 and 10, we choose a quantum classifier which uses four qubits and four layers. The features of the dataset are normalized to the interval $[0, 2\pi]$ before encoding. Entanglement is done through Ising time gates. We pre-process the dataset using a rotation. The quantum classifier therefore has 49 trainable parameters. We train it using COBYLA.

As a classical counterpart, we set up a neural network with four fully connected layers. The first three have four nodes while the last one has three. We lastly add one output layer. Before being fed into the network, the features are individually normalized to mean 0 and a standard deviation of 1. We use the Tanh activation function for the hidden layers and Sigmoid as the output activation function. The input layer consists of two nodes which receive the input features $(m^{\text{j2}}, \tau_1^{\text{j1}})$. We

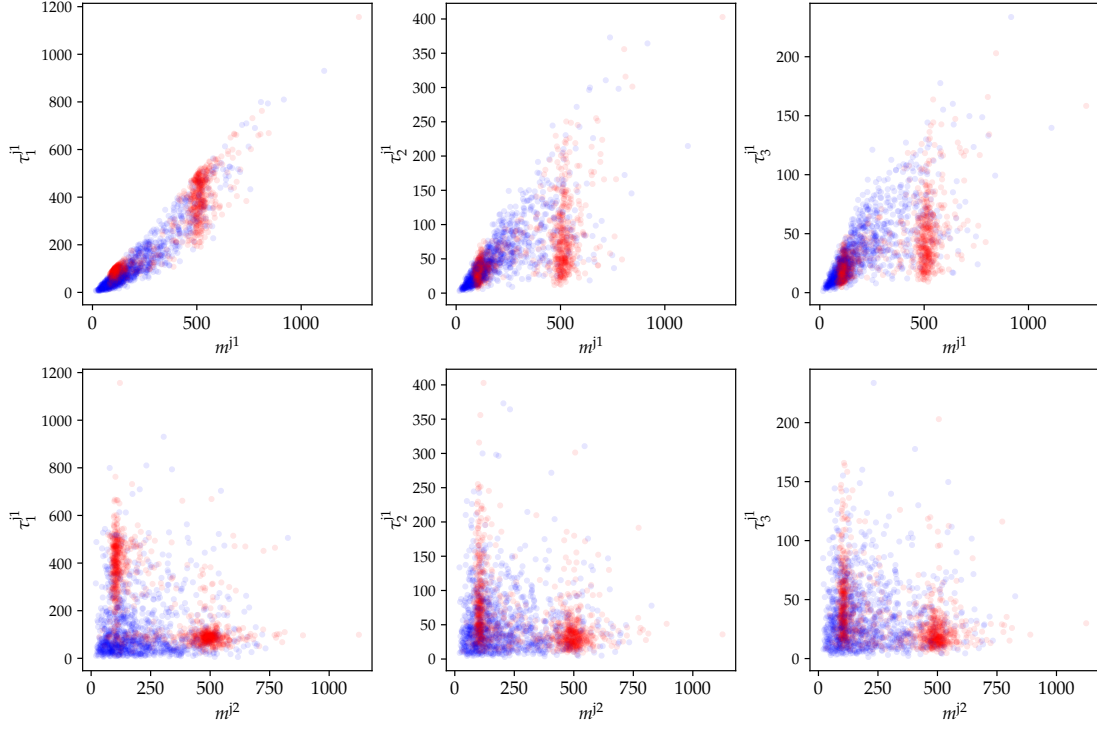


Figure 12.1: The combinations of m^{ji} and τ_k^{ji} . Signal data points are shaded red while the background is shaded blue.

choose this network structure because, when neglecting one of the four-node layers, this neural network has 51 parameters, which is close to the 49 parameters of the quantum classifier. The remaining four-node layer is added to compensate for the fact that the quantum classifier uses a pre-rotation. The network is shown in Figure 12.2. We implement the neural network using keras [27]. It is trained using stochastic gradient descent¹⁴ with a learning rate of $\eta = 0.05$.

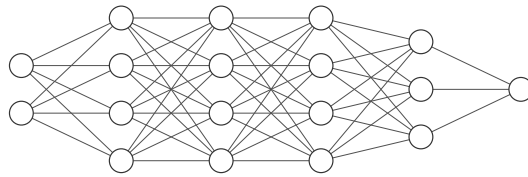


Figure 12.2: The structure of the neural network used for classifying the LHCo dataset.

The performance of the classifiers is shown in Figure 12.3.

The two classifiers perform very similarly on the LHCo dataset. The accuracy of the quantum classifier on the training set is slightly better than on the validation set, which is to be expected since the validation set is not used for training. The neural network actually performs slightly better on the validation set. This may be due to the specific distribution of data points in the dataset.

The quantum classifier and the neural network reach similar loss, however there is a discrepancy in the loss for the validation set: The Neural network performs

¹⁴See Ref. [11], in Chapter 4: *Stochastic Gradient Descent*

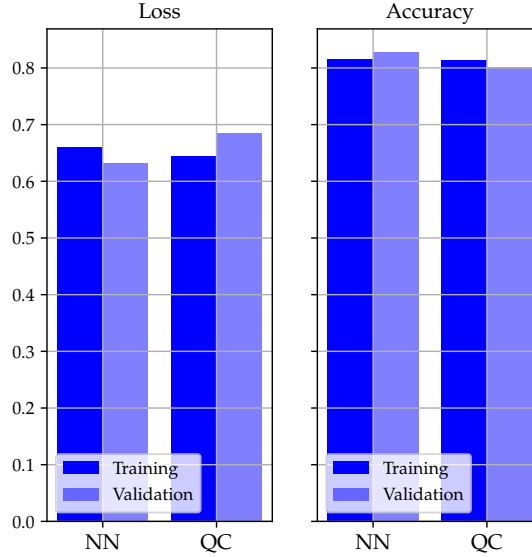


Figure 12.3: The results of the training processes for the LHCo data.

comparably well in terms of loss on the validation set whereas the quantum classifier has higher loss on the validation set. This mirrors the fact that validation accuracy is higher than training accuracy for the neural network and vice versa for the quantum classifier.

Example decision boundaries are shown in Figure 12.4.

We have thus demonstrated that a quantum classifier reaches similar performance on the LHCo dataset, when compared with a neural network of similar complexity. We contend that this is due to the versatility of the quantum classifier, paired with preprocessing of the dataset: After being able to rotate the dataset, the quantum classifier fits its decision boundaries according to basis functions determined by the encoding. These are continuous and differentiable and hence mirror the boundaries of many real-world datasets.

We acknowledge that the notion of “similar complexity” of the quantum classifier compared to the neural network is disputable. We substituted the pre-rotation of the quantum classifier with 20 additional parameters by adding a fully connected dense layer to the neural network. It is however not easily interpretable if these additional degrees of freedom enable transformations similar to a pre-rotation.

The good performance of the quantum classifier may also owe to the fact that the LHCo dataset is relatively easily separable: The signal data points are mostly concentrated in two dense regions. It remains to examine how well a quantum classifier performs in more noisy datasets where the boundaries between signal and background regions are less clear.

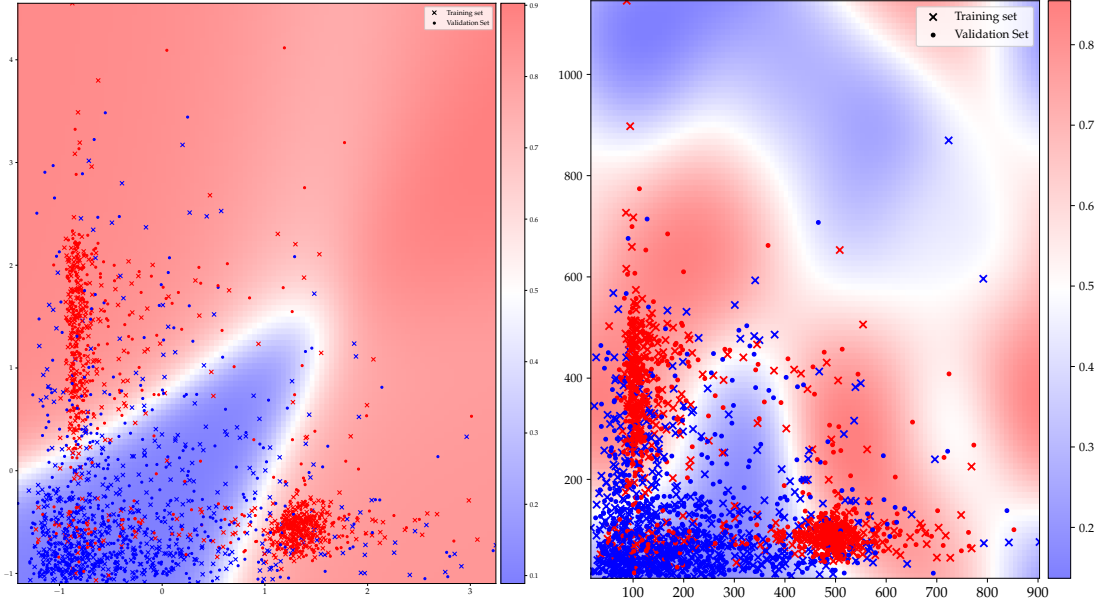


Figure 12.4: Example decision boundaries for the LHCo dataset achieved by a neural network (left) and a quantum classifier (right). The rotation angle is $\varphi = 25.78^\circ$. The dataset shown with the neural network contains outliers the QC-dataset does not contain; this is because these plots were generated using different, randomly chosen samples of the complete LHCo dataset. Accuracy and loss were however consistent for different random samples of the dataset, hence the comparison of these two decision boundary plots.

13. Conclusion & Outlook

We introduced the concept of binary classification and its realization on quantum hardware in sections 2 to 6. This includes training of quantum circuits and important design principles as well as their implementation. In particular, we found that the decision boundaries largely depend on the encoding method. We showed that proper entanglement is vital to quantum circuits and that 4 layers are sufficient for shuffling the components of ρ . We introduced the pennylane- and our own numpy implementation and compared them in terms of runtime.

In sections 7 and 8, we trained a quantum circuit on an example dataset. We demonstrated that some parameters die and explained the cause of dead parameters, namely entanglement using CNOT layers. We thus use Ising time gates in all quantum circuits.

In Section 9, we compared different methods of encoding by training corresponding classifiers on test datasets. We found that dense angle encoding offers the most flexibility on these datasets.

In Section 10, we compared different methods of preprocessing the datasets in order to increase performance. Pre-rotations lead to accuracy gains such that dense angle encoding is able to classify the studied datasets almost perfectly when using four qubits. Pre-translations offered no significant accuracy gains and hence the composition of translations and rotations did not perform better than rotations alone. We however also found that increasing the number of qubits has the largest impact on the performance of a quantum circuit.

In Section 11, we discussed how a quantum classifier constructs decision bound-

aries and emphasized that they are highly dependent on the encoding that is chosen. The classifiers we tested are not prone to overfitting since they are not complex enough. Instead, the encoding methods we introduced enable good fits on the datasets.

In Section 12, we applied our methods to the LHC Olympics dataset. This dataset consists of simulated SM and BSM data used to evaluate the performance of different machine learning methods on accelerator data. We compared the performance of a classifier using either a neural network or a quantum circuit with comparable complexity and found that they reach similar accuracy.

This is a promising result as it suggests that model-independent searches at the LHC could be further diversified by deploying quantum machine learning alongside other tools that are already in use.

There is, of course, much still left to learn and test concerning quantum classifiers:

- Can quantum classifiers be more efficiently or more consistently trained? The authors of Ref. [6] implement a variation of gradient descent they call *quantum gradient descent* which trains the quantum classifier while using the quadratic distance norm $L = \frac{1}{n} \sum (y - \hat{y})^2$ as loss function. The authors include the *Fubini-Study-Metric* to transform the gradient. Is this a more efficient or more consistent training method than our methods, e.g. gradient descent or COBYLA?
- We investigated quantum classifiers whose structure mirrors the structure of neural networks. It is an open question whether this is optimal; are other structures better at achieving entanglement and shuffling the components of the density matrix ρ_{in} ? For an example of a different quantum circuit design, refer to Ref. [28]. For an example of a different entangling method, refer to Ref. [8].
- Are there other encoding methods that provide better decision boundaries? We compared five different methods but there is no limit to the complexity of encoding methods. Ref. [7], for example, introduces a more complex encoding scheme through their *Quantum Circuit Learning* design. For a thorough analysis of different encoding schemes, refer to Ref. [20].
- Why are the performance gains through preprocessing so small? Do these methods only offer significant improvements for more complex datasets? What other preprocessing methods are there (see the example of scaling the dataset from the end of Section 10.4)?
- Do preprocessing and especially encoding still play pivotal roles in quantum circuits with more qubits? Recall for example that, in Section 9.2, we argued that arbitrary polynomials may be approximated for high-enough numbers of qubits. Is this a feature only of inverse angle encoding or of other encodings as well?
- Does combining neural networks and quantum circuits yield higher performance? One approach could be to let a neural network act as preprocessing for a quantum circuit or vice-versa.

A. Encoding Decision Boundaries

In this section, we show the decision boundaries of various encoding schemes and for various datasets.

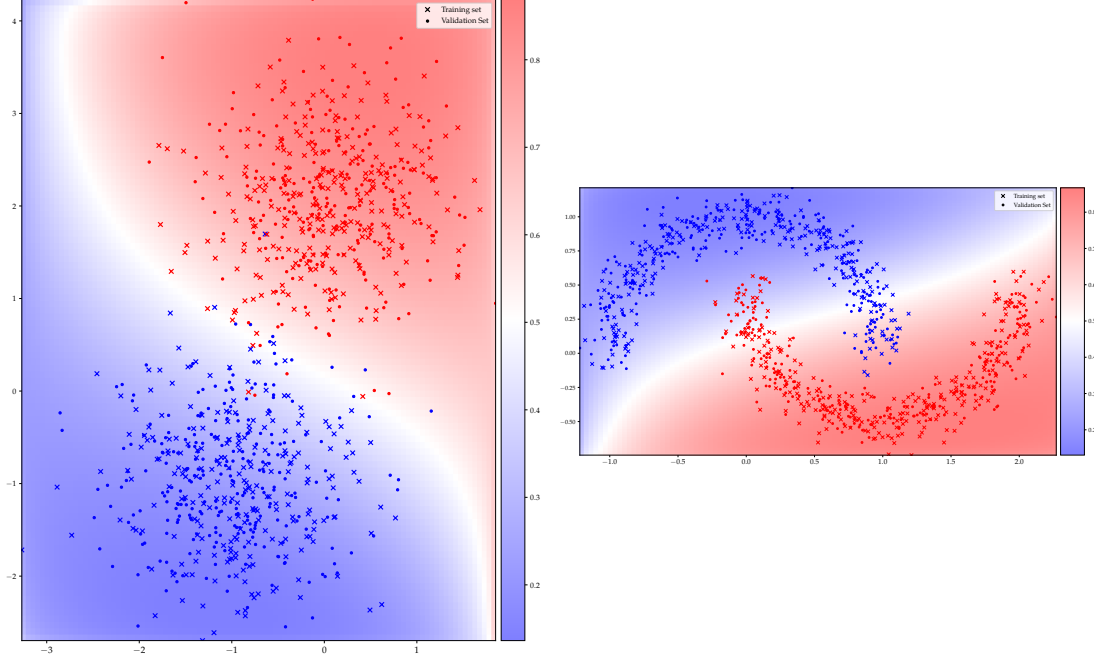


Figure A.1: Inverse angle encoding using four qubits. Since the basis functions are polynomial (Equation 9.3), a nearly linear decision boundary is possible for sklblobs. This is a good fit for the sklblobs dataset which is (neglecting outliers) linearly separable. The plots show decision boundaries which are polynomial and not trigonometric. This encoding is however not able to find ideal decision boundaries for sklmoons.

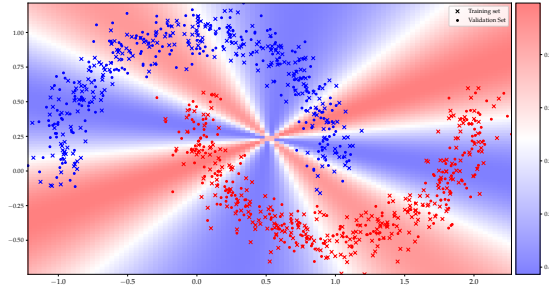


Figure A.2: Wavefunction encoding using four qubits. As mentioned in Section 9.5, the wavefunction encoding does not depend on the magnitude of the input \vec{x} , hence the decision boundaries are radial.

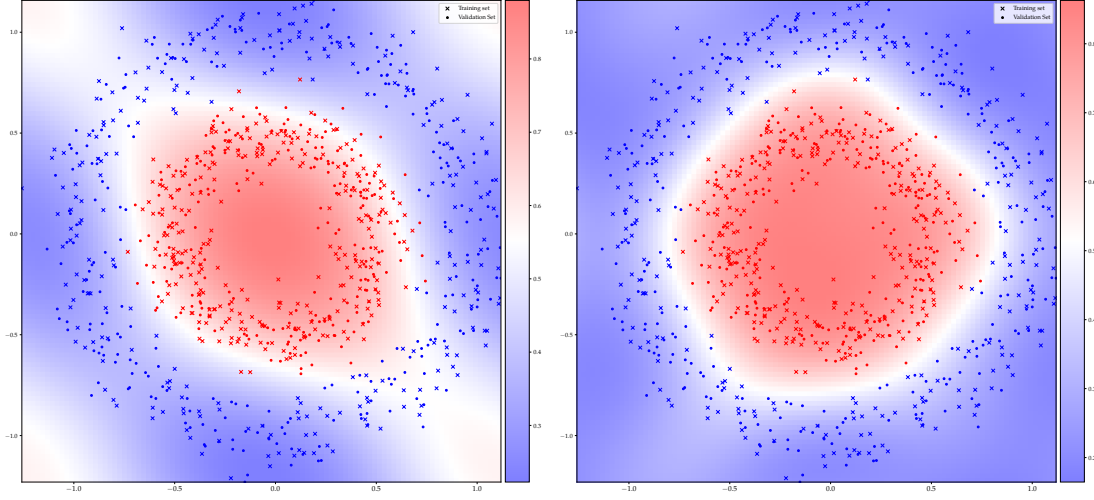


Figure A.3: Angle encoding using two (left) respectively four (right) qubits. Despite angle encoding being the simplest encoding scheme, it produces suitable decision boundaries, especially when using four qubits. This reflects a result from Section 10.4: The number of qubits has the biggest influence on the performance of a quantum classifier.

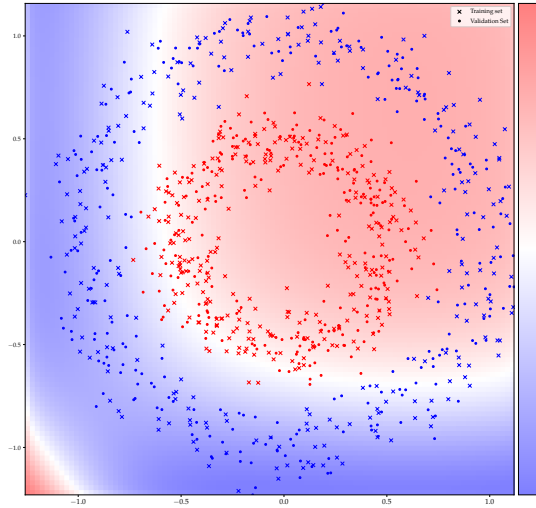


Figure A.4: Double angle encoding for two qubits. Similar decision boundaries are found using four qubits. This suggests that double angle encoding, despite its complexity relative to the other encoding schemes we presented, lacks flexibility.

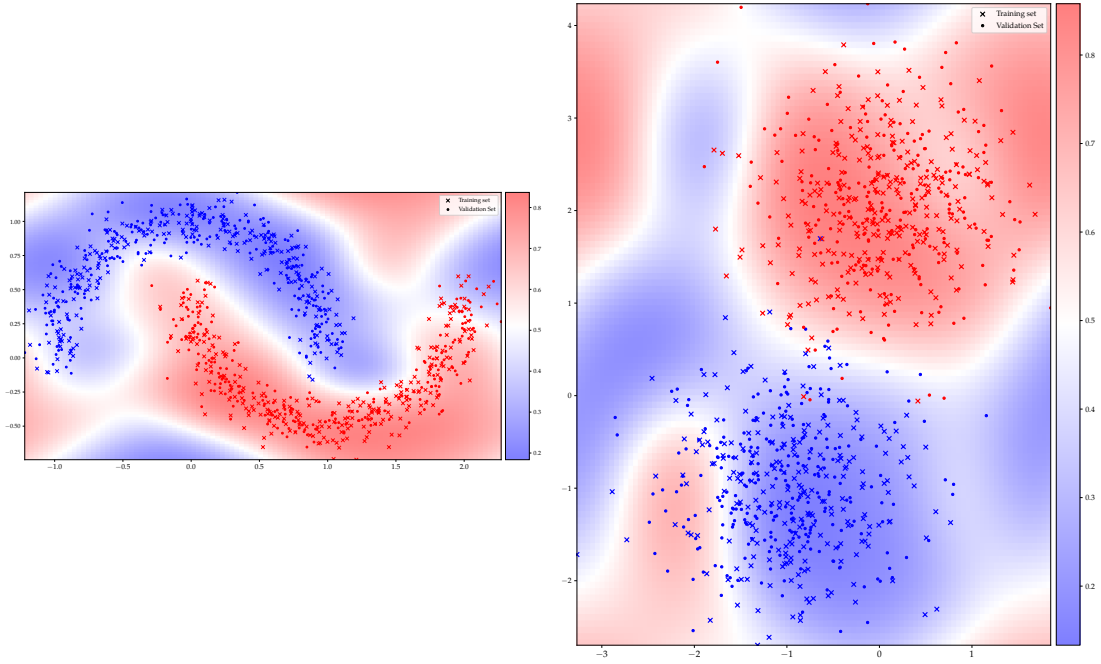


Figure A.5: Dense angle encoding using four qubits. These decision boundary plots suggest that dense angle encoding offers the greatest flexibility in terms of the decision boundaries it is able to provide. Compare that to inverse angle encoding (Figure A.1), which is able to classify sklblobs very well but which does not mirror the structure of the dataset when applied to sklmoons.

Acknowledgments

Last but not least, I would like to thank the people without whom I could not have completed this work.

I would like to thank my supervisor Michael Krämer for enabling me to work on such an interesting topic.

I would like to thank Alexander Mück for the many meetings spent discussing the topic and the time he devoted to helping me researching and writing this thesis.

I would like to thank my family and my friends for supporting and encouraging me during these busy times.

References

- [1] Gregor Kasieczka et al. “The LHC Olympics 2020 a community challenge for anomaly detection in high energy physics”. In: *Reports on Progress in Physics* 84.12 (Dec. 2021). arXiv: 2101.08320v1 [hep-ph].
- [2] Alejandro Perdomo-Ortiz et al. “Opportunities and challenges for quantum-assisted machine learning in near-term quantum computers”. In: *Quantum Science and Technology* 3.3 (June 2018). arXiv: 1708.09757v2 [quant-ph].
- [3] Frank Arute et al. “Quantum supremacy using a programmable superconducting processor”. In: *Nature* 574.7779 (Oct. 1, 2019). arXiv: 1910.11333v2 [quant-ph].
- [4] A. P. Lund, Michael J. Bremner, and T. C. Ralph. “Quantum sampling problems, BosonSampling and quantum supremacy”. In: *npj Quantum Information* 3.1 (Apr. 2017). arXiv: 1702.03061v1 [quant-ph].
- [5] T. H. Johnson, S. R. Clark, and D. Jaksch. “What is a quantum simulator?”. In: *EPJ Quantum Technology* (2014). arXiv: 1405.2831v1 [quant-ph].
- [6] Andrew Blance and Michael Spannowsky. “Quantum machine learning for particle physics using a variational quantum classifier”. In: *Journal of High Energy Physics* 2021.2 (Feb. 2021). arXiv: 2010.07335v1 [quant-ph].
- [7] Sulaiman Alvi, Christian Bauer, and Benjamin Nachman. *Quantum Anomaly Detection for Collider Physics*. 2022. arXiv: 2206.08391v1 [hep-ph].
- [8] Koji Terashi et al. “Event Classification with Quantum Machine Learning in High-Energy Physics”. In: *Computing and Software for Big Science* 5.1 (Jan. 3, 2021). arXiv: 2002.09935v2 [physics.comp-ph].
- [9] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.
- [10] M. J. D. Powell. “A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation”. In: *Advances in Optimization and Numerical Analysis*. Springer Netherlands, 1994. ISBN: 978-94-015-8330-5. DOI: 10.1007/978-94-015-8330-5_4.
- [11] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O’Reilly, June 2019.
- [12] K. Mitarai et al. “Quantum circuit learning”. In: *Phys. Rev. A* 98 (3 Sept. 2018). arXiv: 1803.00745v3 [quant-ph].
- [13] Gavin E. Crooks. *Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition*. 2019. arXiv: 1905.13311v1 [quant-ph].
- [14] Richard P. Feynman. “Quantum Mechanical Computers”. In: *Foundations of Physics* 16.6 (1986). DOI: 10.1007/2Fbf01886518.
- [15] Ville Bergholm et al. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2018. arXiv: 1811.04968v4 [quant-ph].
- [16] Xanadu Quantum Technologies Inc. *PENNYLANE Plugins and Ecosystems*. 2019. URL: <https://pennylane.ai/plugins.html> (visited on 08/26/2022).
- [17] Xanadu Quantum Technologies Inc. *PENNYLANE Documentation: Quantum Circuits*. 2019. URL: <https://docs.pennylane.ai/en/stable/introduction/circuits.html> (visited on 08/26/2022).

- [18] Xanadu Quantum Technologies Inc. *PENNYLANE Documentation: class ApproxTimeEvolution*. 2019. URL: <https://docs.pennylane.ai/en/stable/code/api/pennylane.ApproxTimeEvolution.html> (visited on 08/26/2022).
- [19] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011). arXiv: 1201.0490v4 [cs.LG].
- [20] Ryan LaRose and Brian Coyle. “Robust data encodings for quantum classifiers”. In: *Physical Review A* 102.3 (Sept. 2020). arXiv: 2003.01695v1 [quant-ph].
- [21] Gregor Kasieczka, Ben Nachman, and David Shih. *R&D Dataset for LHC Olympics 2020 Anomaly Detection Challenge*. Version v5. Zenodo, Apr. 2019. DOI: 10.5281/zenodo.6466204.
- [22] Torbjörn Sjöstrand et al. “An introduction to PYTHIA 8.2”. In: *Computer Physics Communications* 191 (2015). ISSN: 0010-4655. arXiv: 1410.3012v1 [hep-ph].
- [23] Torbjörn Sjöstrand, Stephen Mrenna, and Peter Skands. “PYTHIA 6.4 physics and manual”. In: *Journal of High Energy Physics* 2006.05 (May 2006). arXiv: hep-ph/0603175v2 [hep-ph].
- [24] J. de Favereau et al. “DELPHES 3: a modular framework for fast simulation of a generic collider experiment”. In: *Journal of High Energy Physics* 2014.2 (Feb. 13, 2014). arXiv: 1307.6346v3 [hep-ex].
- [25] Alexandre Mertens. “New features in Delphes 3”. In: *Journal of Physics: Conference Series* 608 (May 2015). DOI: 10.1088/1742-6596/608/1/012045.
- [26] Michele Selvaggi. “DELPHES 3: A modular framework for fast-simulation of generic collider experiments”. In: *Journal of Physics: Conference Series* 523 (June 2014). DOI: 10.1088/1742-6596/523/1/012033.
- [27] François Chollet et al. *Keras*. <https://keras.io>. 2015.
- [28] Maria Schuld et al. “Circuit-centric quantum classifiers”. In: *Physical Review A* 101.3 (Mar. 2020). arXiv: 1804.00633v1 [quant-ph].