piQture: A Quantum Machine Learning Library for Image Processing

by

Saasha Joshi
B.E., University Institute of Engineering and Technology
Panjab University, 2021

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Saasha Joshi, 2024
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

piQture: A Quantum Machine Learning Library for Image Processing

by

Saasha Joshi
B.E., University Institute of Engineering and Technology
Panjab University, 2021

Supervisory Committee

Dr. Hausi A. Müller, Co-Supervisor
(Department of Computer Science)

Dr. Ulrike Stege, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Hausi A. Müller, Co-Supervisor
(Department of Computer Science)

Dr. Ulrike Stege, Co-Supervisor
(Department of Computer Science)

ABSTRACT

Quantum Machine Learning (QML) is a discipline of research at the intersection of quantum information and machine learning that leverages quantum mechanical properties to enhance computational capabilities. With its emergence, there is a need to integrate QML models into machine learning pipelines for real-life applications such as image processing. While standalone programs exist to demonstrate the performance of QML models, a well-defined model workflow is noticeably absent. This thesis thoroughly explores various existing QML models and their practical utility in image processing tasks, with the aim of constructing a robust QML library.

Throughout this thesis, we develop *piQture*, an open-source Python and Qiskit-based library that streamlines the development, training, and evaluation of QML models. Its design and structure prioritize usability among users familiar with classical machine learning without prior QML experience. Further, *piQture* is augmented with automated building, testing, and packaging workflows that enhance software reliability and reproducibility. Finally, we provide strategies to facilitate model management and storage within *piQture* for practical adoption and future analysis of pre-trained QML models.

Table of Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Glossary **xiii**

Acknowledgements **xv**

1 Introduction **1**

1.1 Motivation 2
1.2 Problem Definition and Research Questions 2
1.3 Contributions 3
1.4 Our Approach 3
1.5 Outline 4

2 Background and Related Work **5**

2.1 Data Embedding 5
2.1.1 Quantum Image Representations 6
2.2 QML Algorithms for Image Classification 13
2.2.1 Quantum Support Vector Machines 13
2.2.2 Kernel Methods 14
2.2.3 Variational Quantum Algorithms 14
2.2.4 Quantum Tensor Networks 15
2.2.5 Quantum Neural Networks 20

2.3 QML Workflows 24

3 Quantum Image Representation **27**
3.1 Overview 27
3.2 QIR Methods 29

4 Quantum Machine Learning **38**
4.1 Quantum Feature Maps and Kernels 38
4.1.1 Example: Pauli Feature Map 39
4.2 Kernel Methods 41
4.2.1 Support Vector Machines 41
4.2.2 Example: Quantum Kernel Estimator 45
4.3 Variational Quantum Algorithms 46
4.3.1 Example: Variational Quantum Classifier 47
4.4 Quantum Tensor Networks 49
4.4.1 Example: Hierarchical Quantum Classifier 50
4.5 Quantum Neural Networks 53
4.5.1 Example: Quantum Convolutional Neural Network 53

5 Introducing *piQture* **58**
5.1 Overview 59
5.2 Pipeline Design 61
5.2.1 Workflow Description 61
5.3 Pipeline Structure 65
5.3.1 Data Preprocessing 65
5.3.2 Quantum Circuit Preparation Stage 65
5.3.3 Quantum Circuit Processing Stage 71
5.3.4 Measurement 74

6 Advancing *piQture*: Strategies for CI/CD **75**
6.1 Overview 76
6.2 Building, Testing, and Packaging *piQture* 77
6.2.1 Setting Up the GitHub Repository 77
6.2.2 Building 80
6.2.3 Testing 86
6.2.4 Packaging 87

6.3 *piQture* in Production 89
6.3.1 Model Management 91
6.3.2 Prediction Service 94

7 Quick Start Guide: How to use *piQture*? 96
7.1 Getting Started 96
7.1.1 Setup 96
7.1.2 Installation 97
7.1.3 Installation from Source 97
7.2 Tutorials 98
7.2.1 Tutorial 1: Building an INEQR encoding 98
7.2.2 Tutorial 2: Building a TTN tensor network 99
7.2.3 Tutorial 3: Training a QCNN model 101

8 Conclusion and Future Work 104
8.1 Contributions 104
8.2 Future Work 105
8.2.1 Optimizing *piQture* Design and Structure 106
8.2.2 Workflow Management and Monitoring 106
8.2.3 Model Evaluation Techniques 107

Bibliography 108

List of Tables

2.1 Quantum Image Representation Methods 7

5.1 Mapping of various stages in the *piQture* workflow to their corresponding modules in the *piQture* library. 65

List of Figures

2.1 (a) A discriminative MPS tensor network with four inputs and a single output state. After each unitary operation, one of the qubits is measured and discarded. The output state of the discriminative MPS network is obtained by measuring the remaining qubit. (b) A generative MPS tensor network with two input and four output states. After each unitary operation, one of the qubits is measured and its result is recorded. All the qubits are initially prepared to a reference state of $|0\rangle$. Adapted from [41]. 17

2.2 Alternative two-qubit unitary gate parameterizations. Redrawn from [30]. 17

3.1 Four approaches to the underlying framework of QML. Redrawn from [102]. 28

3.2 A 2×2 grayscale image and its FRQI state representation. . . 29

3.3 A 2×2 colored image and its FRQCI state representation. . . . 30

3.4 A 2×2 colored image and its MCRQI circuit representation. . 31

3.5 A 2×2 grayscale image and its NEQR state representation. . . 33

3.6 A $2^h \times 2^w$ image box for a $H \times W$ image, displaying the redundant rows and columns. 33

3.7 A 2×2 colored image with its OCQR state and circuit representations. 35

3.8 Two 2×2 grayscale images with their QRMII state and circuit representations. 36

4.1 Circuit diagram for the Pauli feature map $\mathcal{U}_\Phi(x)$ with layers of Walsh-Hadamard transform and unitary operators $U_\phi(x)$. Redrawn from [38]. 40

4.2 Circuit diagram of the unitary $U_{\Phi(x)}$, from the family of Pauli-ZZ feature maps, with $|S| \leq 2$ qubit interactions as built with Qiskit. . 41

4.3 The circuit implemented to estimate the fidelity between two data points x_i and x_j. The unitary operators $U_\phi(x_i)$ and $U_\phi(x_j)$ are given by Equation 4.7. Redrawn from [38]. 45
4.4 Circuit diagram of the variational quantum classifier model that utilizes the Pauli feature map $\mathcal{U}_\Phi$ as a data embedding block, followed by a parameterized block $W(\theta)$. Redrawn from [38]. 48
4.5 Circuit diagram of the parameterized block $W(\theta)$, comprising a series of single-qubit rotation gates θ_n^i and entangling gate layer comprising of a sequence of controlled-phase (CZ) gates. The gates inside the dotted block are repeated l times. Adapted from [38]. 49
4.6 Circuit diagram of a classifier block $T(\theta)$ in the HC model, based on the TTN tensor network. The circuit consists of a series of layers of two-qubit unitary gates U_i, applied to adjacent qubits. 52
4.7 Circuit diagram of a classifier block $T(\theta)$ in the HC model, based on the MERA tensor network. The circuit consists of two layers of two-qubit unitary gates, D_i and U_i, applied to adjacent qubits. . . . 53
4.8 Alternative parameterizations for two-qubit unitary operations utilized in tensor network circuits. (a) A set of two arbitrary single-qubit unitary gates followed by a CNOT gate. (b) An arbitrary two-qubit unitary gate that can be decomposed into several optimal gate combinations. (c) An arbitrary three-qubit gate that utilizes an auxiliary qubit. Redrawn from [30]. 54
4.9 Alternative decomposition of a two-qubit unitary operation with a simple gate parameterization. (a) A real-valued decomposition with R_y gates. (b) A complex-valued decomposition with R_y and R_z gates. Redrawn from [58]. 54
4.10 A real-valued decomposition of an arbitrary two-qubit unitary gate with at most 12 single-qubit and 2 CNOT gates. In the circuit, $A, B = R_z(\alpha)R_y(\theta)R_z(\beta)$. Redrawn from [114]. 56
4.11 A complex-valued decomposition of an arbitrary two-qubit unitary gate with at most 15 single-qubit and 3 CNOT gates. In the circuit, $A_i = R_z(\alpha)R_y(\theta)R_z(\beta)$. Redrawn from [114]. 56

4.12 Circuit diagram for a QCNN model with stacked convolutional, pooling, and fully connected layers. The convolutional layer mirrors the structure of a MERA tensor network with unitary operations U_i. While the pooling layer performs mid-circuit measurements to apply classically conditioned unitary gates V_i on remaining qubits. The fully-connected layer is applied as a series of controlled-phase gates on the remaining qubits. Adapted from [18]. 57

5.1 Layout of (a) classical machine learning and (b) quantum machine learning workflows. Redrawn from [104]. 59

5.2 Structure of a classical machine learning workflow, highlighting the various tools involved in pipeline automation and model monitoring tasks. 60

5.3 Workflow for *piQture*, consisting of four main stages: Data Preprocessing, Quantum Circuit Preparation, Quantum Circuit Processing, and Measurement. The Quantum Circuit Preparation stage contains the Data Embedding and the Model Selection sub-stages. The Quantum Circuit Processing contains the Model Training and the Model Evaluation & Training sub-stages. 62

5.4 The input and output components at each stage of the *piQture* library, highlighting the flow of data through the workflow. Adapted from [127]. 63

5.5 Class diagram illustrating the layout of the `data_encoder` module in the *piQture* library. The diagram demonstrates the inheritance relationship between the `ImageEmbedding` base class and various QIR classes, such as `FRQI, NEQR,` and `INEQR`, as well as their composition with the `ImageMixin` class. 67

5.6 A class diagram of the `tensor_network_circuits` module within *piQture*. The diagram demonstrates the inheritance relationship between the various tensor network classes and the `BaseTensorNetwork` abstract class. 68

5.7 A class diagram of the `neural_networks` module within *piQture*. The diagram illustrates the inheritance relationship between the `QCNN` class and the `QuantumNeuralNetwork` base class. The `sequence` method in the `QuantumNeuralNetwork` class enables the stacking of various quantum layers in a QCNN circuit. 69

5.8 UML class diagram of the `layers` module in *piQture*. The diagram demonstrates the inheritance between various quantum layer structures and the `BaseLayer` class. 70
5.9 UML class diagram of the `neural_netwoorks` module in the *Qiskit Machine Learning* library [44]. The diagram demonstrates the inheritance between the `SamplerQNN`, `EstmatorQNN`, and the `NeuralNetworks` class. 72
5.10 Sequence diagram for evaluation and tuning sub-stage performed by the `NeuralNetworkClassifier` class from the *Qiskit Machine Learning* library. 73

6.1 The sequential development workflow of *piQture*, starting with the collection of source code into a central repository, followed by initiation of CI builds, and finally packaging *piQture* for release. 78
6.2 The directory structure of *piQture* with three essential directories: `piqture`, `tests`, and `graphics`. 79
6.3 The Issues section contains listings of bug reports, feature, and documentation requests related to *piQture*. 81
6.4 The Pull Requests section contains requests for new features and bug fixes. 81
6.5 The Milestones feature facilitates tracking version releases for *piQture*. 82
6.6 The GitHub Project assists in tracking progress and task prioritization for *piQture* via the Kanban-like Board view. 82
6.7 The GitHub Project assists in tracking progress, task prioritization, and version monitoring for *piQture* via the Table view. 83
6.8 A python package build triggered by a commit in *piQture*, executing jobs across different Python versions. 84
6.9 Code coverage for *piQture* after the latest commit, as provided by Coveralls. 88
6.10 Development and production workflows of the *piQture* pipeline. . . 90
6.11 The MLflow Tracking Server, locally hosted at a specified URI. . . . 92
6.12 The MLflow Model Registry with saved QCNN models, locally hosted at a specified URI. 92
6.13 Circuit Diagram artifact of a QML model logged on the MLflow Experiment Tracking Server. 93

6.14 Attributes and parameter values of a QML model logged on the MLflow Experiment Tracking Server. 94
6.15 A Prediction Service hosted on a local development server with Flask. The QCNN model is pre-trained, and accessed from a local MLflow registry. 95

7.1 INEQR circuit generated for a 2×2 grayscale MNIST image with pixel values $[[38, 49], [46, 41]]$, from the corresponding code given in Listing 7.2. Qubits q_0 and q_1 encode the pixel positions in the basis states with the help of Hadamard and X gates, whereas the CX gates encode the pixel color information in binary format. 98
7.2 A TTN tensor network circuit, corresponding to Listing 7.3, constructed with a simple gate parameterization and real gates. . . 100
7.3 A TTN tensor network circuit, corresponding to Listing 7.4, constructed with a general gate parameterization and real gates. . 100
7.4 Circuit diagram of the QCNN model corresponding to the Listing 7.5. The initial rotation gates (before the barrier) implement the Angle Encoding strategy, followed by unitary gates (R_y, CX, and CZ) that apply quantum convolutional and pooling layers. The final CZ gate represents a fully connected layer. 103
7.5 Model evaluation chart for the QCNN model corresponding to Listing 7.5, depicting a minimizing objective function value over 10 training and evaluation iterations. 103

Glossary

API *Application Programming Interface*. 89, 91, 94, 105

CI/CD *Continuous Integration and Delivery*. 1, 3, 4, 75, 76, 77, 80, 86, 105

CNN *Convolutional Neural Network*. 22, 23, 53

DIP *Digital Image Processing*. 2

HC *Hierarchical Quantum Classifier*. 19, 38, 50, 51, 54

HHL *Harrow–Hassidim–Lloyd*. 13, 14

HPC *High Performance Computing*. 25

HQC *Hybrid Quantum-Classical*. 1, 2, 5, 14, 24, 26, 46, 47

MERA *Multiscale-Entanglement Renormalization Ansatz*. 16, 17, 18, 19, 22, 50, 51, 53, 66, 69

MLOps *Machine Learning Operations*. 3, 75, 76, 91

MPS *Matrix Product States*. 16, 19, 49, 66

NISQ *Noisy Intermediate-Scale Quantum*. 15

PCA *Principal Component Analysis*. 18, 19, 22

PEPS *Projected Entangled Pair States*. 16

PQC *Parameterized Quantum Circuits*. 2, 14, 15, 21, 26, 46

PyPI *Python Package Index*. 89

QA *Quantum Autoencoder*. 21

QAOA *Quantum Approximate Optimization Algorithm*. 15, 24

QCNN *Quantum Convolutional Neural Networks*. 15, 19, 22, 23, 24, 38, 53, 54, 55, 66, 69, 71, 101, 104

QIR *Quantum Image Representation*. 5, 6, 12, 28, 66, 104

QML *Quantum Machine Learning*. iii, 1, 2, 3, 4, 5, 13, 24, 26, 27, 28, 38, 46, 58, 60, 61, 62, 64, 66, 71, 73, 74, 75, 79, 89, 91, 93, 94, 96, 99, 104, 105, 106, 107

QNN *Quantum Neural Networks*. 5, 19, 20, 21, 22, 23, 38, 104

QPU *Quantum Processing Unit*. 2

QSDLC *Quantum Software Development Lifecycle*. 3, 25

QSVM *Quantum Support Vector Machine*. 13, 14, 104

QTN *Quantum Tensor Networks*. 5, 15, 19, 38, 104

QuanvNN *Quanvolutional Neural Network*. 23, 104

SDK *Software Development Kit*. 1

SVM *Support Vector Machine*. 13, 14, 38, 41, 42, 43, 44, 45

TTN *Tree Tensor Networks*. 16, 17, 18, 19, 50, 51, 53, 66, 99

UCC *Unitary Coupled Clustered*. 15

UML *Unified Modelling Language*. 25

VQA *Variational Quantum Algorithms*. 5, 13, 14, 15, 38, 46, 104

VQC *Variational Quantum Classifier*. 19, 38, 47, 48, 99, 104

VQE *Variational Quantum Eigensolver*. 15, 24

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude towards the University of Victoria and my supervisors, Dr. Hausi Müller and Dr. Ulrike Stege. Their steadfast support and insightful guidance have been pivotal throughout my research journey.

I am greatly thankful to IBM Quantum for providing me with invaluable internship opportunities that have provided me with unforgettable experiences. Additionally, I am immensely grateful for being part of the NSERC CREATE Quantum BC and the vibrant Qiskit community, where I had the privilege of collaborating with exceptional minds in quantum computing. I also want to recognize the Rigi Research Group for their unwavering support, stimulating discussions, and passion for quantum computing, teaching, and outreach initiatives that have significantly shaped my personal and academic growth.

I extend my heartfelt thanks to my parents, Suparna and Vishal Joshi, for their unwavering support at every step of my life. Your outlook and life experiences have been a constant source of inspiration. You have been my guiding light and the pillars of support in my life. I consider myself incredibly fortunate to have you. I also want to thank my grandparents for their inspiring stories, which continue to motivate me every single day.

To my sibling, Sherron Joshi, your admiration and love for me gives me meaning and a sense of responsibility, for which I am truly grateful. I can't wait for the shenanigans we will get up to in Toronto! Also, I would like to extend my gratitude to my cousins, who made me feel at home in Canada or the US and were always there in need of help.

I want to thank my friends, including Sarthak Thakur, Sai Sree Laya Chukkapalli, Amit Bansal, Rohan Juneja, Shikhar Rana, and Pranav Kairon, for the meaningful conversations, inspiring messages, and an endless supply of memes that kept me going forward. To my colleagues-turned-friends—Sai Sree Laya Chukkapalli, Lijun Lang, Karan Taneja, Sumit Suresh Kale, Dhruv Srinivasan, and Abeer Vaishnav—thank you for the ridiculously entertaining office moments, weird conversations, Broadway shows, and trips to NYC; to my 36-220 crew, thank you for being the most inspiring, popular, and fun office mates. I will never forget our caffeine-induced borgor-filled memories. And to all those I haven't mentioned individually, your presence and support mean a great deal to me. Each of you has played a unique role in my

journey, and I deeply appreciate the collective support that has shaped my academic and personal growth.

Lastly, to the wonderful individuals I met at Victoria, Jennifer Munro, your Christmas dinners were a highlight; Sree Gayatri Talluri for your amazingly delicious baked food and travel inspiration; Aniket Mahindraker for being an enthusiastic travel mate and conversationalist; Solene, thank you for introducing me to the most captivating board games out there; Prashanti Priya Angara, your support and guidance were invaluable to me during my time in Victoria; and Angadh Singh for being a motivating lab mate. Thank you for being like family to me in Victoria.

Chapter 1

Introduction

Quantum Machine Learning (QML) is a discipline of research that lies at the intersection of quantum information and machine learning [104]. It leverages the distinctive properties of quantum systems to augment the computational capabilities of machine learning models in terms of speedup, complexity, simulation, and more [10]. This advantage occurs when computationally complex tasks, such as specific classical subroutines, predictive models, or optimization processes [104], are delegated to a dedicated quantum processing unit.

In our thesis, we primarily explore the *Hybrid Quantum-Classical* (HQC) QML algorithms and their practical applications in supervised machine learning, specifically for image classification tasks. Our primary objective is to construct a robust QML library, named *piQture*, that seamlessly integrates different stages of a QML workflow. *piQture* is an automated tool designed to accommodate users familiar with classical machine learning but without prior experience in QML, positioning itself as a user-friendly entry point into QML. We streamline the library by defining automated workflows within the GitHub Actions[1] platform, providing the necessary setup for its *Continuous Integration and Delivery* (CI/CD). To realize *piQture*, we use IBM's *Software Development Kit* (SDK), known as Qiskit [44], thereby solidifying our contributions to QML. The thesis concludes with a comprehensive evaluation of the contributions made by developing *piQture* and potential avenues for enhancements and extensions to it in the broader field of image processing.

This chapter, presents the underlying motivation behind our research, the problem statement, and the contributions this thesis makes to QML and image processing.

[1] https://github.com/features/actions

1.1 Motivation

The rapid evolution of quantum computing presents a compelling need to explore its practical applications. Simultaneously, the ongoing advancements in *Digital Image Processing* (DIP) highlight the need to explore its integration with emerging quantum technology. While DIP has presented unfathomable advancements in computer vision, remote sensing, security and surveillance, medical image analysis, and more, several challenges persist. These include issues related to the scalability of machine learning models to handle massive datasets, computational costs associated with processing images in high-dimensional feature spaces, and limitations in hardware capabilities [21, 54]. With the ability to support quantum mechanical processes, the advancement of quantum computing presents new avenues for improvement in image processing.

Quantum computers support HQC algorithms that execute computationally complex tasks on a *Quantum Processing Unit* (QPU) using *Parameterized Quantum Circuits* (PQC) with tunable unitary operations. These PQC contain trainable parameters that are subject to fine-tuning through a classical optimization loop [23, 80, 87, 92, 141]. Orchestrating these algorithms necessitates a dedicated workflow that manages the execution of quantum circuits on designated processors.

QML algorithms are HQC algorithms. While standalone programs exist to demonstrate the performance of QML models on image processing, a well-defined model workflow is noticeably absent [13, 127]. This absence of a structured and automated workflow hinders the seamless applicability of QML models in real-world applications, emphasizing the need for purpose-built and open-source pipelines [26]. Therefore, developing a library that integrates the HQC approach is a crucial step to exploring the potential of QML algorithms in real-world applications [127].

1.2 Problem Definition and Research Questions

The following research questions guide the research presented in our thesis:

RQ1: What are the most common QML models? How are these models applied to image processing?

RQ2: What is an effective architecture and workflow for a software library for implementing QML algorithms for image processing?

RQ3: How can the developed workflow be streamlined to facilitate continuous integration, packaging, and deployment of QML models?

1.3 Contributions

The contributions of our thesis are summarized as follows:

C1: A thorough exploration of existing QML models and their utilization in supervised machine learning tasks, focusing on image classification.

C2: The development of an open-source library, called *piQture* that seamlessly facilitates development, execution, and training of QML models for image processing tasks, prioritizing user-friendly functionality.

C3: The integration of various automation techniques into our *piQture* pipeline to ensure robust and automated testing, packaging, and deployment of QML models.

1.4 Our Approach

To address RQ1, we comprehensively study existing QML models and discern their architectural differences and implementation methodologies. This effort establishes the groundwork for effectively utilizing these models for image processing.

To answer RQ2, we design and develop an open-source Python and Qiskit-based library that streamlines the development, execution, and training of QML models for image processing tasks. We actively test and build the source code in this stage, utilizing GitHub Actions to establish a CI/CD framework.

For the third research question (RQ3), we focus on integrating the pipeline workflow with various automation techniques to ensure robust deployment of the QML models. We employ various *Machine Learning Operations* (MLOps) techniques to automate and manage the *Quantum Software Development Lifecycle* (QSDLC). Finally,

we facilitate model management and storage within *piQture* for practical adoption and future analysis of pre-trained QML models.

1.5 Outline

This chapter introduces the research undertaken in this thesis, shedding light on the underlying motivation, problem statements, and research contributions. The subsequent chapters establish the groundwork for formalizing a CI/CD QML pipeline tailored for image processing.

Starting with **Chapter 2**, we present the essential background and existing literature on QML models, laying the groundwork for subsequent chapters. **Chapter 3** explores the different data embedding and image representation techniques in detail. Moving forward, **Chapter 4** provides a detailed description of the various QML models primarily used for image classification. **Chapter 5** outlines the methodologies employed in designing and constructing the *piQture* library. **Chapter 6** discusses the strategies for automating and deploying the *piQture* library. **Chapter7** provides a quick start guide for utilizing the *piQture* library. Finally, **Chapter 8** summarizes and concludes this thesis and provides insights into possible future work.

Chapter 2

Background and Related Work

This chapter offers a comprehensive overview of the background and landscape of QML. This groundwork helps to set the stage for a thorough examination of QML models and workflows and their practical impact on advancing image processing capabilities using quantum computers. Our thesis primarily explores HQC QML algorithms and their applications in supervised machine learning, specifically for image classification tasks. We also discuss potential uses in the broader field of image processing.

We review various data embedding techniques, including *Quantum Image Representation* (QIR) methods in Section 2.1. Section 2.2, presents an extensive background on QML algorithms for image classification, including *Variational Quantum Algorithms* (VQA) (Subsection 2.2.3), *Quantum Tensor Networks* (QTN) (Subsection 2.2.4), and *Quantum Neural Networks* (QNN) (Subsection 2.2.5). Finally, we discuss the strategies for constructing QML workflows and highlight significant existing QML software frameworks in Section 2.3.

2.1 Data Embedding

QML algorithms rely on classical data inputs and thus must employ HQC workflows [92]. Aimeur et al. [4] introduced the encoding framework for using classical data for learning tasks on quantum devices. Subsequently, Schuld et al. [104] expanded and elaborated this framework. The hybrid approach sparked the development of sophisticated data embedding strategies that encode classical inputs effectively for quantum circuits. LaRose and Coyle [56] outlined several common approaches for en-

coding classical data, such as Basis Encoding, Amplitude Encoding, Angle Encoding, Hybrid Encoding, Time-Evolution Encoding, and Hamiltonian Encoding [56].

2.1.1 Quantum Image Representations

With the onset of quantum image processing research, various *Quantum Image Representation* (QIR) methods emerged. Each method is tailored to encode image data in a quantum framework, such as pixel position and color information. In 2003, Venegas-Andraca and Bose introduced the Qubit Lattice model [116], presenting a quantum mechanical technique capable of initializing qubits by detecting different electromagnetic waves. This model generated a quantum state representing an image across the entire spectrum of frequencies, extending beyond the visible spectrum. In 2005, Latorre et al. introduced the Real Ket method [57], leveraging superposition to represent images on a quantum computer. In 2010, Venegas-Andraca and Ball introduced the notion of an Entangled Image representation [115] that utilizes the quantum mechanical property of entanglement to represent digital images. After these seminal contributions, QIR techniques have followed the traditional gate-based representation of quantum operations. These operations constitute the crucial step of data embedding for encoding classical image data such as pixel position, color intensity, transparency, and more as a quantum state representation. Table 2.1 compares the existing QIR models based on their qubit requirements and type of color encoding. Some of these models are discussed below.

Flexible Representation of Quantum Images (FRQI): In 2011, Le et al. proposed the FRQI [59], enabling the representation of grayscale square images on quantum devices. FRQI utilized $n+1$ qubits to encode $2^n \times 2^n$ pixels and their intensity values. While the pixel positions were represented using the Walsh-Hadamard transform and encoded in the basis state of the quantum circuit, the pixel intensities were encoded in the probability amplitudes of the quantum state, notably through parameters such as the θ value of a controlled rotation gate like R_y.

Expansions on FRQI

Expanding on the FRQI method, in 2011, Sun et al. [112] proposed the Multi-Channel Representation of Quantum Images (MCRQI). MCRQI extended the framework to accommodate the RGB-α channels of an image, effectively encoding the primary

Table 2.1: Quantum Image Representation Methods

Authors	Year	QIR Method	Qubit Reqs.	Image Size	Intensity Range	Color Encoding
Le et al. [59]	2010	FRQI	$2n+1$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Sun et al. [112]	2011	MCRQI	$2n+3$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Zhang et al. [136]	2013	NEQR	$2n+q$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Zhang et al. [137]	2013	QUALPI	n_1+n_2+q	$2^{n_1} \times 2^{n_2}$	$[0, 2^q-1]$	Basis State
Li et al. [67]	2014	NASS	$2n+1$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Jiang and Wang [48]	2015	INEQR	n_1+n_2+q	$2^{n_1} \times 2^{n_2}$	$[0, 2^q-1]$	Basis State
Jiang et al. [47]	2015	GQIR	$h+w+q$	$H \times W$	$[0, 2^q-1]$	Basis State
Li et al. [69]	2016	FRQCI	$2n+1$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Sang et al. [100]	2016	NCQI	$2n+3q$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Abdolmaleky et al. [1]	2017	QMCR	$2n+3q$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Jiang et al. [46]	2017	QPC	$m_i+(m_x+m_y+m_z)+m_a+1$	$\langle X_i, Y_i, Z_i, A_i, f_i \rangle$	$[0, 2^q-1]$	Basis State
Liu et al. [74]	2018	OCQR	$2n+q+2$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Li et al. [65]	2018	BRQI	$2n+p+1$ or $2n+p+3$	$2^n \times 2^n$	$[0, 2^q-1]$ $p = \log q$	Basis State
Sahin and Yilmaz [99]	2018	QRMW	n_1+n_2+q+b	$2^{n_1} \times 2^{n_2}$	$[0, 2^q-1]$	Basis States
Zhou et al. [142]	2018	QRMMI	$2n+q+k$	$2^n \times 2^n$ 2^k images	$[0, 2^q-1]$	Basis State
Wang et al. [123]	2019	QRCI	$2n+p+3$	$2^n \times 2^n$	$[0, 2^q-1]$ $p = \log q$	Basis State
Li and Liu [68]	2019	Improved FRQI (FRQCI)	$2n+1$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Khan [53]	2019	IFRQI	$2n+\frac{q}{2}$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Wang et al. [120]	2019	QIIR	$2n+2p+q$	$2^n \times 2^n$ 2^p pallete	$[0, 2^q-1]$	Basis States
Li et al. [66]	2019	GNEQR	$2n+q+2$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Xu et al. [132]	2019	OQIM	$2n+2$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Xu et al. [132]	2019	mOQIM	$2n+k+2$	$2^n \times 2^n$ 2^k images	$[0, \frac{\pi}{2}]$	Probability Amplitude
Li et al. [66]	2019	QRDS	$2n+34$	$2^n \times 2^n$	32-bit float-point	Basis State
Li et al. [66]	2019	QCDSA	$2n+66$	$2^n \times 2^n$	Two 32-bit float-points	Basis State
Li et al. [66]	2019	QCDSE	$2n+66$	$2^n \times 2^n$	Two 32-bit float-points	Basis State
Liu et al. [75]	2019	QBIR	$2n+q$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis States
Grigoryan and Agaian [31]	2020	FTQR	n_1+n_2+1	$2^{n_1} \times 2^{n_2}$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Wang et al. [122]	2020	DQRCI	$2n+p+6$	$2^n \times 2^n$ 2 images	$[0, 2^q-1]$ $p = \log q$	Basis State
Nasr et al. [81]	2021	EFRQI	$2n+1$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$	Probability Amplitude
Nasr et al. [81]	2021	ENEQR	$2n+q+1$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Chen et al. [16]	2021	QIRHSI	$2n+q+2$	$2^n \times 2^n$	$[0, \frac{\pi}{2}]$ $I = [0, 2^q-1]$	Probability Amplitude (H, S), Basis State (I)
Haque et al. [36]	2023	SCMFRQI	$2n+1$	$2^n \times 2^n$	$[0, 2^q-1]$	Basis State
Levaillant [61]	2024	BRQMI	$2n+p+k+1$ or $2n+p+k+3$	$2^n \times 2^n$ 2^k images	$[0, 2^q-1]$ $p = \log q$	Basis State

colors (RGB) and pixel transparency α information. In 2016, Li et al. introduced watermarking capabilities to FRQI with the Flexible Representation of Quantum Color Images (FRQCI) method [69] by encoding a watermark image within probability amplitudes of the same quantum state representing the colored image.

The Improved FRQI method, proposed in 2018 by Li and Liu [68], upgrades the traditional FRQI [59] method by encoding different color channels within the phase of a qubit state. This method is different from FRQCI [69], as Improved FRQI specifically stores the Red channel in the θ phase, while the Green and Blue channels are stored in the ϕ phase of a qubit state.

In 2020, Nasr et al. [81] introduced Enhanced Flexible Representation of Quantum Images (EFRQI), an enhancement of FRQI that encodes grayscale values of an image in the amplitude of the quantum state using a partial negation operator.

Novel Enhanced Quantum Representation (NEQR): In 2013, Zhang et al. introduced the NEQR [136] method that allows the representation of squared colored images. NEQR necessitates the $2n + q$ qubit requirement to encode intensity values within the range of $[0, 2^q]$—typically $[0, 255]$—in the basis state of a quantum circuit.

Expansions on NEQR

Building upon NEQR, in 2015, Jiang and Wang [48] proposed the Improved Novel Enhanced Quantum Representation (INEQR) to represent non-squared images with dimensions $2^{n_1} \times 2^{n_2}$. INEQR maintains the $n_1 + n_2 + q$ qubit requirement to facilitate efficient representation of images of non-square formats.

Furthering these advancements, in 2015, Jiang et al. introduced the Generalized Quantum Image Representation (GQIR) [47] to represent images of arbitrary size $h \times W$. GQIR achieves this using a logarithmic number of qubits $h + w$, where $h = \lceil \log_2 H \rceil$ and $w = \lceil \log_2 W \rceil$. However, GQIR introduces redundancies of $(2^h - H)$ rows and $(2^w - W)$ columns resulting from the redundant binary representation of the pixel positions.

Following these developments, in 2019, Li et al. presented a comprehensive approach called Generalized Novel Enhanced Quantum Representation (GNEQR) [66]. GNEQR serves as a generalization of the NEQR [136], the INEQR [48], and the NCQI [100] representations. It utilizes $2n + q + 2$ qubits to efficiently represent pixel positions in $2n$ qubits, color channel index in 2 qubits, and grayscale magnitudes up to 2^q in q qubits.

In 2021, Nasr et al. introduced the Enhanced Novel Enhanced Quantum Representation (ENEQR) [81] method, building upon the NEQR approach. ENEQR enhances the encoding of grayscale values by encoding them directly into the basis state of the qubit. By introducing an auxiliary qubit, color information is encoded using one $2n$-CNOT gate to load pixel positions and q CNOT gates to encode grayscale values within a quantum circuit. This modification leads to a reduction in the overall quantum cost and time complexity of the encoding process.

Haque et al. [36] proposed the State Connection Modification FRQI (SCMFRQI) model that improves upon the ENEQR framework by minimizing the number of Toffoli gates for state preparation with reset gates, thereby streamlining the encoding process. Moreover, Haque et al. [35] proposed models, such as the Zero-discarded State Connection NEQR (ZSCNEQR) and the Non-Zero NEQR (NZNEQR), to reduce the need for Toffoli gates further.

Sang et al. [100] proposed the Novel Quantum Representation of Color Digital Images (NCQI) method by combining the MCQRI [112] and NEQR [136] techniques to encode RGB channels of colored images in the basis states of a qubit sequence. NCQI efficiently utilizes $3q$ qubits to store the three color channels. While limited to square images, NCQI achieves a quadratic reduction in time complexity for image preparation compared to MCRQI.

Xu et al. in 2019 [132] proposed the Order-Encoded Quantum Image Model (OQIM) that encodes pixels in an ascending order of their grayscale magnitude. OQIM utilizes $2n + 2$ qubits, where $2n$ qubits encode the sorted positions of the pixels in the basis state, while the remaining qubits encode the grayscale values and original pixel indices in the amplitude of the quantum state. Furthermore, the authors also introduce a variant termed mOQIM [132], that extends the capabilities of OQIM to encode multiple images. mOQIM necessitates an additional k qubits to encode 2^k images within a single representation.

The Quantum Block Image Representation (QBIR) method, as presented by Liu et al. [75], represents an image of size $2^n \times 2^n$ on $2n + q$ qubits. QBIR introduces a distinct process involving iterative encoding procedures representing sub-blocks of size $2^w \times 2^w$, where $w \leq n$. This decomposition strategy aims to facilitate the shuffling of sub-blocks within the original image, thereby enabling the creation of an encrypted image.

Quantum Log-Polar Image (QUALPI): In 2013, Zhang et al. [137] introduced the QUALPI representation method that stores and processes images with log-polar coordinates. In a log-polar representation of a digital image, each pixel has two attributes ρ and θ that denote log-radius and angular position, respectively. Inspired by the FRQI method, QUALPI requires $n_1 + n_2$ qubits to store an image with 2^{n_1} log-radius values and 2^{n_2} angular orientations. An additional q qubits store the color information, in the range $[0, 2^q - 1]$, in the basis state of the qubit sequence.

n-qubit Normal Arbitrary Superposition State (NASS) In 2014, Li et al. [67] introduced the NASS for image representation that encodes pixel positions as a superposition of the basis states. For a multi-dimensional image of size $2^n \times 2^n \times 2^n$, NASS requires $3n$ qubits to encode the binary pixel position information. The color information of the image is normalized in the range $\left[0, \frac{\pi}{2}\right]$ and encoded as the probability amplitude in the circuit.

Optimized Quantum Representation of Colored Images (OCQR): In a further optimization, Li et al. introduced the OCQR [74] method. OCQR requires approximately one-third of the qubits needed by NCQI, encoding RGB channel index and values in $q+2$ qubits. This includes a spare index representation for additional features like pixel transparency, resulting in a total requirement of $2n+q+2$ qubits. Additionally, in 2017, Abdolmaleky et al. proposed the Red-Green-Blue Multi-Channel Quantum Representation (QMCR) [1], a model similar to NCQI.

Quantum Point Cloud (QPC): In 2017, Jiang et al. presented the QPC [46] representation that enabled the representation of a $3D$ image model. QPC utilizes the point cloud information of a model in a 3-dimensional coordinate system and offers efficient quantum representation techniques for spatial data. In a point cloud P, each point is represented as a tuple of X, Y, and Z coordinates, point attributes A, and a boolean f indicating whether the point attribute is an original value or a difference arising from overlap with other points. A point p_i is expressed as,

$$p_i = \langle X_i, Y_i, Z_i, A_i, f_i \rangle \tag{2.1}$$

For representing m_i points in a point cloud with coordinates X_i, Y_i, and Z_i, QPC requires $m_i + (m_x + m_y + m_z) + m_a + 1$ qubits.

Bitplane Representation of Quantum Images (BRQI): Li et al. in 2018 [65] proposed the BRQI method that offers an innovative solution to employ binary color values $\in \{0, 1\}$ to encode images on a quantum device. Each pixel, with a grayscale value within the range of 2^q, can be decomposed into q binary color bits, called *bitplanes*. These bitplanes can be superimposed onto the same pixel index using $p = \log q$ qubits, while a single qubit can encode the color value. The pixel index representation aligns with that proposed in NEQR [136], requiring a total of $2n$ qubits for an image of size $2^n \times 2^n$. For colored images, BRQI introduces two additional qubits that differentiate between color channels, leaving a spare channel index. Consequently, the qubit requirement for BRQI is $2n + p + 1$ for grayscale images and $2n + p + 3$ for colored images.

Expansions on BRQI

Wang et al. introduced a model akin to BRQI [65], named Quantum Representation Model of Colored Digital Images (QRCI) [123]. QRCI also employs bitplane information to encode color images in a quantum circuit. However, QRCI diverges in its qubit utilization for storing color information. Unlike BRQI, which uses a single qubit for a binary color value, QRCI uses three qubits to store the RGB color values, respectively. Therefore, the total qubit requirements of QCRI is $2n + 6$, where $2n$ qubits encode pixel indices, three qubits represent RGB color values, and three qubits store the bitplane information.

In 2020, Wang et al. proposed the Double Quantum Color Images Representation (DQRCI) [122] building upon the QRCI [69] method. DQRCI enables the simultaneous encoding of two images in a superposition state on a quantum circuit. It requires $2n+p+6$ qubits to store an image of size $2^n \times 2^n$ and allocates an additional 3 qubits to store the color information of the second image.

Quantum Representation Model for Multiple Images (QRMMI): In 2018, Zhou et al. [142] introduced the QRMMI method, which was the first model to encode multiple images on a quantum computer. QRMMI utilizes $2n + q + k$ qubits to represent 2^k colored images. The pixel indices and color values are represented in the basis state, whereas image indices are encoded using an additional k qubits.

Quantum Signal Representation (QSR): In the same year, Li et al. [66] proposed the QSR method encompassing three models designed to store data, such as

signals as a sequence of discrete real or complex values.

The first model, Quantum Representation of Real-valued Digital Signals (QRDS), facilitates the representation of a single floating-point number on a quantum circuit. A 32-bit floating-point number has three parts of information: the sign (1 bit), exponent (8 bits), and mantissa (23 bits). QRDS utilizes 1, 8, and 23 qubits to represent these components, respectively. Moreover, $2n + 2$ qubits are needed to encode the position and color indices. Consequently, QRDS utilizes a total of $2n + 34$ qubits to represent a colored image of size $2^n \times 2^n$.

The second model in QSR is the Quantum Representation of Complex-valued Digital Signals with Algebra form (QCDSA), which enables the encoding of a complex number in the algebraic form $C(t) = iC_I(t) + C_R(t)$. Here, $C_I(t)$ and $C_R(t)$ are the 32-bit floating-point numbers representing the imaginary and real parts, respectively. QCDSA effectively combines multiple QRDS encodings to represent complex numbers.

The third model, Quantum Representation of Complex-valued Digital Signals with Exponential form (QCDSE), represents a complex number with exponential form, $C(t) = r(t)e^{i\theta(t)}$, on a quantum circuit. Here, the modulus $r(t)$ and the $\theta(t)$ are 32-bit floating-point numbers. The qubit requirement of QDCSE is equivalent to that of the QCDSA model.

Other QIR models

In 2018, Sahin and Yilmaz [99] proposed the Quantum Representation of Multi-Wavelength Images (QRMW) method with a novel approach to represent multi-spectral images on quantum computers. In multi-spectral images, different wavelengths of light are captured, forming stacks of n bands of light intensity data. QRMW encodes these spectral bands using $b = \lceil \log_2 n \rceil$ qubits, where each color value within the range 2^q uses q qubits for representation. Hence, QRMW requires $n_1 + n_2$ qubits to encode a multi-spectral image of size $2^{n_1} \times 2^{n_2}$. Remarkably, for larger image sizes, QRMW utilizes fewer qubits compared to MCQRI [112].

In 2022, Chen et al. proposed the Quantum Image Representation method based on the HSI color space model (QIRHSI) [16] model, specializing in encoding the Hue (H) and Saturation (S) values in the amplitude of a quantum state. Meanwhile, the Intensity (I), ranging within 2^q, is encoded in the basis state of a qubit. This encoding method requires $2n$ qubits for pixel position, q qubits for intensity, and 2 qubits for

hue and saturation values.

2.2 QML Algorithms for Image Classification

In 2014, several authors, including Schuld et al. [101] and Wittek [130], started to categorize the emerging QML algorithms based on the quantum subroutines they employed. Naturally, many of these new algorithms were directly adapted from classical machine learning algorithms. One key approach was to invoke subroutines on the quantum side. For example, the K-median and Hierarchical clustering algorithm by Aimeur et al. [5] and the Support Vector Machines algorithm by Anguita et al. [3] used Grover's search as a subroutine [32].

The *Harrow–Hassidim–Lloyd* (HHL) algorithm [37] solves a quantum system of linear equations by approximating the expectation value of the vector $\overrightarrow{x}$ satisfying $A\overrightarrow{x} = \overrightarrow{b}$. This algorithm inspired and facilitated the development of many current QML algorithms [19], including the *Quantum Support Vector Machine* (QSVM) algorithm [96].

2.2.1 Quantum Support Vector Machines

The QSVM is the quantum counterpart of a classical *Support Vector Machine* (SVM) that performs binary classification by producing a maximum-margin linear decision boundary in a higher dimensional space. However, this routine becomes tedious in higher dimensions, posing challenges in separating non-linear surfaces with a decision hyperplane. As a solution, SVM is also formulated using the Lagrangian method in the dual form [82] requiring a kernel method to calculate the distance between the data inputs.

The QSVM algorithm proposed by Rebentrost et al. [96] generates a linear decision boundary through an HHL subroutine and subsequently classifies the data points with a SWAP test [15]. However, Rebentrost et al. [96] suggest that with the natural ability of quantum computers to map data to higher dimensional spaces, evaluation of inner products, in the dual form of SVM, can be done faster on a quantum computer. Havelicek et al. [38] expand this idea by presenting *kernel methods* and *quantum feature maps* that non-linearly map data inputs to higher-dimensional quantum states, aiding the estimation of the required kernel function. These quantum feature maps are extremely useful in the application of VQA and are discussed subsequently in

Subsections 2.2.2 and 2.2.3.

In 2015, Li et al. [70] implemented the QSVM algorithm to classify handwritten digits. Their model implemented the HHL algorithm to acquire the hyperplane parameters, followed by a training-data oracle U, that constructs the quantum state $|\overrightarrow{x}\rangle$, and its inverse $U^\dagger$ to measure the fidelity between the states. Yang et al. [133] expanded this work by implementing an optimized version of the HHL algorithm and training-data oracle on Iris [27] and OCR [20] datasets. The model was run on an IBM Quantum device, namely IBMQX2, to achieve an accuracy of 98% and 99.5% on the respective datasets.

2.2.2 Kernel Methods

In 2018, both Havlicek et al. [38] and Schuld et al. [103] introduced a quantum kernel method that utilizes feature maps to estimate the fidelity between input data. A kernel approximated as a quantum sub-routine is then plugged into the dual form SVM to obtain a decision hyperplane for classification purposes. These kernel methods were dubbed as implicit models [45]. In 2021, Schuld [102] further extended the theory on quantum models formulated as kernel methods and compared their performance with *Variational Quantum Algorithms* (VQA).

Subsequently, in the same paper, Havlicek et al. [38] proposed a family of feature maps called the Pauli feature maps that are utilized as a quantum sub-routine in the kernel estimator models for classification purposes. Pauli feature maps are discussed in detail in Chapter 4. In 2020, Suzuki et al. [113] analyzed these Pauli feature maps to evaluate their performance in classifying the circles, moon, exponential, and xor datasets.

The kernel estimator model proposed by Havlicek et al. [38] was later benchmarked by Park et al. [88] on multiple datasets, including the XOR, Wines, Breast Cancer, Digits, and custom complex datasets. They utilized various feature maps to implement a quantum kernel, including the Pauli-ZZ and Pauli-YY feature maps. The kernel method showed an accuracy of $98 - 100\%$ with different feature maps.

2.2.3 Variational Quantum Algorithms

VQA are HQC algorithms that contain tunable unitary operations called *Parameterized Quantum Circuits* (PQC). These algorithms use a classical optimizer to train

these parameters, which is useful in the *Noisy Intermediate-Scale Quantum* (NISQ) era [93].

Two workhorses of quantum computing, the *Variational Quantum Eigensolver* (VQE) [92] and the *Quantum Approximate Optimization Algorithm* (QAOA) [23], are based on the VQA principle. QAOA is geared towards combinatorial optimization problems and utilizes a combination of problem-specific and mixer unitaries. Whereas, VQE focuses on finding the ground state energy of quantum systems through a tailored *ansatz*, an initial assumption of a quantum subroutine that approximates a solution to the problem. VQE and QAOA both employ PQC [107] as a part of their ansatz, along with classical optimization in a hybrid framework to minimize an objective function iteratively.

For example, Peruzzo et al. [92] introduced a *Unitary Coupled Clustered* (UCC) ansatz to obtain the ground state of an energy Hamiltonian using the VQE algorithm. In 2017, Kandala et al. [51] proposed a hardware-efficient ansatz that shortens the circuit depth by exploiting qubit symmetries on the hardware and brings correlated qubits closer. In 2019, Sim et al. [107] proposed several combinations of ansätze based on their ability to represent quantum states in the Hilbert Space. These ansätze were utilized by Hur et al. in 2022 [43] in building *Quantum Convolutional Neural Networks* (QCNN). Another approach to implementing such ansätze is employing *Quantum Tensor Networks* (QTN), as discussed in Section 2.2.4.

In 2018, Havlicek et al. [38] and Schuld et al. [103] proposed a variational classifier that utilizes a combination of data embedding techniques and a parameterized ansatz to perform classification. The parameterized ansatz comprises trainable unitary gates that classical optimization techniques can optimize to predict labels on unseen data. This model is discussed in detail in Chapter 4.

2.2.4 Quantum Tensor Networks

Tensor networks are mathematical structures designed to express computationally complex high-order tensors in terms of their tractable interconnected lower-rank tensor components [111]. Tensors, represented as the nodes within a tensor network, encapsulate data in multiple dimensions known as their rank. This arrangement of tensors in a tensor network facilitates complex computations.

A Hilbert Space, characterized by a fixed computational basis $|k\rangle$, effectively captures the networks and contractions of these tensors, making them adept at represent-

ing quantum states and quantum circuits [9,30,85]. Tensor networks also find application in quantum physics to produce classical representations of quantum systems [85]. Their graphical representation and versatile nature offer enhanced adaptability in constructing quantum circuits and developing quantum algorithms for practical quantum hardware [10]. Various tensor network architectures, such as *Matrix Product States* (MPS)) [86], *Tree Tensor Networks* (TTN) [106], *Projected Entangled Pair States* (PEPS) [117], *Multiscale-Entanglement Renormalization Ansatz* (MERA) [118], and others, form the diverse landscape of tensor networks, each with its unique strengths and applications.

Tensor networks gained traction for their performance in machine learning around 2015, with structures reminiscent of hierarchical neural networks but with a reduced resource requirement and complexity [83]. These tensor networks found applications in various supervised [84, 110] and unsupervised [34, 111] machine learning tasks, exhibiting behavior similar to that observed in resource-intensive neural networks.

Furthermore, a numerical experiment by Liu et al. [71] in 2019 supports the effectiveness of TTN and MERA structures in exhibiting abstraction capabilities similar to deep neural networks. Additionally, they highlight the capability of TTNs in displaying distinct quantum properties, such as entanglement and fidelity, which are instrumental in characterizing classical data like images. This observation aligns seamlessly with studies by Martyn et al. [79] in 2020 and by Liu et al. [76] in 2021.

Tensor Network Models

In 2019, Huggins et al. [41] employed tensor networks to construct models for both discriminative and generative machine learning tasks. A tensor network for discriminative tasks is designed to transform input data into a more compact representation to match the required output dimensions. The model, designed as a tree structure shown in Figure 2.1(a), parameterizes N-qubit input qubits to determine the model's prediction by discarding some qubits after the application of each unitary U. In contrast, a generative tensor network, depicted in Figure 2.1(b), adopts an inverse structure and takes a low-dimensional input to generate a higher-dimensional output. These discriminative and generative models trained with different hyperparameter values, achieved an average accuracy exceeding 95% in classifying grayscale images.

In 2018, Grant et al. [30] introduced tensor networks as quantum circuits for supervised classification tasks. They performed binary classification on datasets like

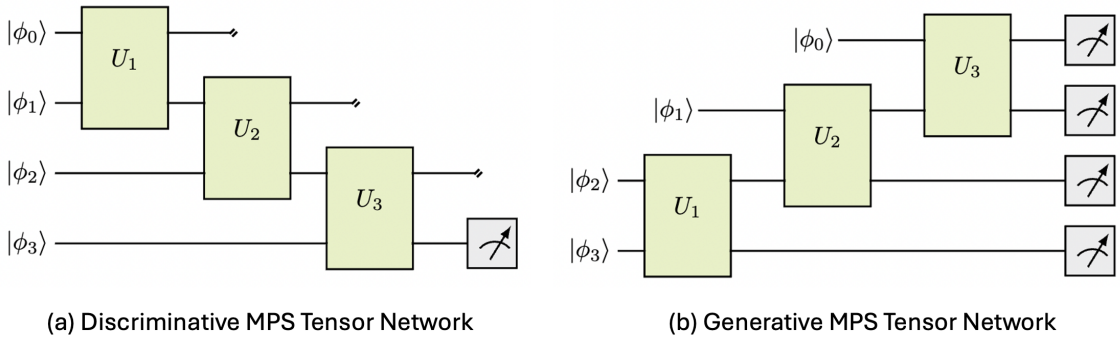


Figure 2.1: (a) A discriminative MPS tensor network with four inputs and a single output state. After each unitary operation, one of the qubits is measured and discarded. The output state of the discriminative MPS network is obtained by measuring the remaining qubit. (b) A generative MPS tensor network with two input and four output states. After each unitary operation, one of the qubits is measured and its result is recorded. All the qubits are initially prepared to a reference state of $|0\rangle$. Adapted from [41].

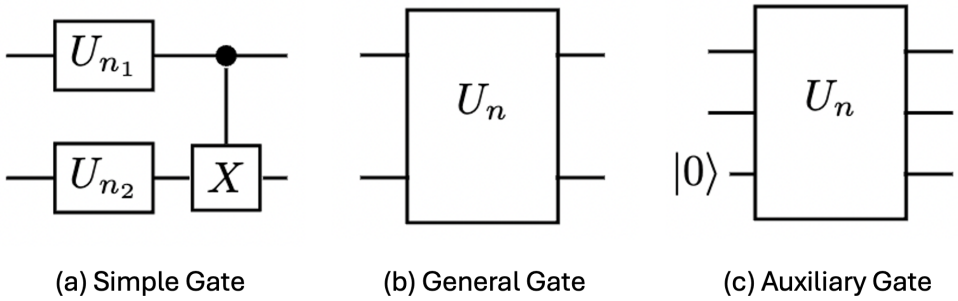


Figure 2.2: Alternative two-qubit unitary gate parameterizations. Redrawn from [30].

Iris [27] and MNIST handwritten digits [60] employing tensor network structures such as TTN, MERA, and a hybrid structure that combines MERA with a pre-trained TTN. These tensor networks feature three alternative gate parameterizations for two-qubit unitaries, as shown in Figure 2.2. Chapter 4 details these parameterizations.

Among the three proposed tensor network structures—TTN, MERA, and hybrid—with different gate parameterizations, the MERA architecture consistently outperforms TTN across various classification tasks. Overall, complex unitary parameterizations demonstrate greater accuracy than their real gate counterparts. However, it is noteworthy that a hybrid tensor network structure incorporating real gates within a general parameterization exhibits higher classification accuracy, surpassing other Hybrid or MERA configurations.

Inspired by the tensor networks proposed in [30, 41], Lazzarin et al. [58] propose models for multi-class classification purposes that implement a multi-readout strategy.

This strategy employs two methods: qubit decoding and amplitude decoding. In qubit decoding, the expectation values of N observables are measured, and the results are fed into a Softmax layer to generate a probability distribution for N classes. On the other hand, amplitude decoding involves sampling the N readout qubits to obtain probabilities for 2^N classes.

The TTN and MERA structures used in [58] incorporate a choice of gate parameterizations, discussed in Chapter 4; refer to Figures 4.9, 4.10, and 4.11. The models are applied to the MNIST dataset [60], with images pre-processed from a dimension of ($28\times$ to (2×2) using *Principal Component Analysis* (PCA) and a convolutional autoencoder, tailored to fit within the size constraints of quantum computers. When employing PCA and a convolutional autoencoder for image pre-processing, a TTN model with qubit encoding and a general gate parameterization with complex unitaries achieves a maximum accuracy of 81% and 83%, respectively. In contrast, the MERA structure performs better with amplitude encoding and a general gate parameterization, achieving an accuracy of 85% and 93% with PCA and convolutional autoencoder, respectively. Notably, a convolutional autoencoder consistently outperforms PCA in all experiments, allowing the model to learn features better during dimensionality reduction.

In 2020, Liu et al. [72] introduced a quantum-classical Hybrid Tensor Network (HTN) structure, combining tensor networks with classical neural networks in a uniform framework to overcome the inherent linearity limitations of tensor networks. The introduction of non-linearity in an HTN structure is achieved through the data encoding process and a classical neural network. HTN is built as a sequence of tensor network and classical neural network layers where a singular readout output from the tensor network is truncated into the classical structure. This model is trained through a standard backpropagation algorithm coupled with a gradient descent optimizer. The authors further adapt this configuration to build an autoencoder, where the tensor network layers function as an encoder, compressing quantum states to a low-dimensional intermediate state and the classical neural network layers operate as a decoder.

In 2021, Huang et al. [40] enhanced the HTN model [72] to introduce a Variational Quantum Tensor Network (VQTN) model designed for classification tasks on Iris [27] and MNIST datasets [60]. Akin to the HTN structure, VQTN leverages tensor networks to encode and process data inputs and feed the outputs to a classical neural network to produce the final classification results. The data encoding process is per-

formed with the help of a custom encoding strategy, named *kernel encoding*, wherein classical input data x is normalized based on a user-defined function $\phi(x)$. This normalization facilitates utilizing data as rotation angles of the feature map parameters during encoding. The feature map employs a dense angle encoding strategy with $\lceil \frac{N}{2} \rceil$ qubits to encode N data inputs. VQTN can be constructed with either TTN or MERA architectures, containing arbitrary two-qubit unitary gate parameterizations inspired by the Data-Driven Quantum Circuit Learning (DDQCL) model proposed by Benedetti et al. [7] in 2019.

Following the circuit modeling stage in the VQTN architecture, a multi-readout strategy is implemented to transfer a multiplex of feature descriptions of an input sample to the classical neural network, demonstrating better performance than a single readout. Finally, gradient descent optimization techniques are used to update the circuit parameters iteratively. VQTN has been shown to outperform other QTN, such as QNN [24], QCNN [18], and *Hierarchical Quantum Classifier* (HC) [30], owing to the new kernel encoding, circuit modeling, and multiple readout strategies. The architecture consistently achieves an accuracy above 95% across various classification categories.

In 2020, Chen et al. [17] proposed another hybrid model integrating MPS with *Variational Quantum Classifier* (VQC) for supervised learning tasks. Unlike many other models requiring pre-trained classical layers, MPS and VQC layers in this structure are trainable end-to-end. This unique characteristic allowed the model to exhibit high adaptability to available computational resources, such as adjusting the number of layers in MPS and VQC or the quantum-classical boundary [17], based on accessible quantum or classical computational power. While both the MPS and VQC layers have trainable parameters, MPS is additionally tunable through its bond dimension hyperparameter. In instances when the bond dimension of an MPS is $\chi \leq 2$, appending a VQC layer helps regularize the model to prevent overfitting.

The MPS-VQC model demonstrates a training and testing accuracy of 99% for a binary classification task on the MNIST dataset [60]. This accuracy surpasses an equivalent PCA-VQC model, which utilizes PCA for dimension reduction instead of a tensor network. While an MPS layer shows signs of overfitting with a bond dimension of $\chi = 2$, the performance of the MPS-VQC model attests that the VQC layer effectively mitigates overfitting by acting as a regularizer. However, the training becomes unstable again for bond dimensions $\chi > 2$. The authors attribute this behavior to the excessive representational power of an MPS for a binary classification

task, warranting further investigation.

In 2023, Guala et al. [33] demonstrated implementations of various tensor network circuits for classifying a custom bars and stripes image dataset using PennyLane [8]. Furthermore, they suggest that using circuit-cutting techniques [12,89] on these tensor networks could enhance the feasibility of executing multi-qubit circuits on smaller quantum devices. A circuit-cutting procedure involves partitioning a tensor network circuit into multiple sub-circuits. The initial sub-circuits can be measured in different bases, while subsequent sub-circuits are executed with various initial states, ultimately combining individual results to derive an outcome [89].

2.2.5 Quantum Neural Networks

The term quantum neural computing was first introduced in 1995 by Kak [50], describing it as a computational model that displays quantum mechanical behavior characterized by wavefunctions that can evolve or collapse based on the presence of observation in the system and have the capability to process information similar to an artificial neural network. Several authors followed this definition to introduce *Quantum Neural Networks* (QNN) [2, 22, 25, 91] and the idea of quantum neurons [52, 62, 108, 119]. This rise in prominence and the erratic use of the term was followed by Schuld et al. [105] proposing a set of minimum requirements for any quantum model to be classified as such. These requirements demanded QNN to ensure the existence of associative memory features, reflect some of the typical neural computing mechanisms, such as weighted attention dynamics, network structure, training rules, and make use of quantum mechanical properties, such as superposition, entanglement, and interference, to induce evolution in the circuit. The reason behind establishing such requirements is to ensure a successful generalization of QNN similar to that of classical neural networks.

In 2018, Farhi et al. [24] came up with the design of a QNN capable of performing supervised machine learning tasks such as the classification of handwritten digits. The QNN model, designed for near-term quantum devices, is constructed as a sequence of one or two-qubit unitary gate sequences, U_i, dependent on parameters θ_i. The authors classically simulate the QNN model for binary classification tasks by measuring one designated readout qubit. The overall goal of the model is to find an optimized set of parameters such that the readout, $+1$ or -1, corresponds to the true label of the input data. The model follows the training loop structure as described in [23,80,87,92,141],

along with a linear loss and stochastic gradient descent algorithm to consequently calculate the optimized parameters classically. However, the authors note the lack of non-linearity in this model due to the use of linear unitary gates only. This, on the other hand, prevents gradient blow-up in the model. The claim of non-linearity being introduced in such models through data encodings and measurement was put forth later by [18, 30, 38].

In 2019, Zhao et al. [139] extended the concept of QNN to build a quantum feedforward neural network with the help of a quantum neuron that processes both classical and quantum data. The authors realize all the inputs, outputs, weights, and activation functions as quantum states. As a result of this purely quantum structure, the training of a quantum feedforward neural network does not require any classical resources to record or store intermediate measurement results.

Similarly, Zhao et al. [138] introduced a Quantum Deep Neural Network (QDNN) structure in 2021, consisting of multiple Quantum Neural Network Layers (QNNL). These QNNL consist of PQC that are trained following the iterative classical optimization loop, as suggested in [80, 92, 141]. The layers consist of three parts, namely encoder, transformation, and output. The encoder encodes classical data onto a quantum circuit, hence introducing non-linearity in the process. Whereas, the linear transformation step applies unitary quantum gates on the encoded inputs and finally performs measurements to compute the expectation value of the output states. A QDNN is constructed as the composition of multiple such compatible QNNLs and classical DNN layers. This QDNN model is run to classify MNIST images [60] of size 8×8 to reach the training and test accuracies of 98.92% and 99.57% with increasing number of sampling shots. The model consists of three QNNLs—input, hidden, and output layers. The layers are constructed with trainable parameters in different ansatz structures that are hardware efficient, inspired from [51].

Quantum Autoencoders

In 2017, Romero et al. [98] built a *Quantum Autoencoder* (QA) to perform unsupervised machine learning tasks. QA, like a classical autoencoder, consists of an initial encoding $\mathcal{E}$ and a final decoding evolution $\mathcal{D}$. Some of the inputs are discarded and measured after the encoder is applied, aiding in dimension reduction. Formally, the learning task undertaken by a quantum autoencoder is to preserve the fidelity of the input state through the reduction of the data space. With a hybrid training loop

that undergoes classical optimization [23, 80, 87, 92, 141], the goal is to find suitable parameters for the encoder $\mathcal{E}$ and decoder $\mathcal{D}$ operations that maximize the average fidelity between the input and output state.

Quantum Convolutional Neural Networks

In 2019, Cong et al. [18] extended the concept of QNNs along with tensor networks to introduce a *Quantum Convolutional Neural Networks* (QCNN) structure for binary classification tasks. Akin to classical *Convolutional Neural Network* (CNN), QCNN consists of consecutive convolutional, pooling, and fully connected layers. The convolutional layer structure is constructed with the help of tensor network structures like MERA that are translationally invariant for a finite layer depth and the pooling layers reduce the structural dimension by measuring a fraction of the qubits with the help of mid-circuit measurements. These layers are applied alternatively or consecutively in the model until a desired output size is achieved. This model requires a total of $O(logN)$ variational parameters for N feature inputs, whereas the hyperparameters like the number of convolutional or pooling layers or their depth remain fixed.

This work was expanded to perform classification on classical data such as the MNIST [60] and Fashion MNIST [131] datasets by Hur et al. [43] in 2022. They achieved a best-case accuracy of 99% and 94% in respective datasets. While the schematic layout of their QCNN model resembles that presented in [18], the model is restricted to a two-qubit structure only. The convolutional and pooling layers use parameterized quantum gates that follow a hierarchical structure first suggested by Grant et al. [30] in 2018. The authors propose nine layouts for the convolutional layer inspired by work in [77, 107, 114]. The pooling layer is constructed as a block of two controlled rotations R_z and R_x. Their work also classically pre-processes the data using techniques like PCA, autoencoding, and bilinear interpolation. Another feature explored by Hur et al. is the boundary conditions of the proposed QCNN model where two-qubit unitary gates can be optionally applied on the first and the last qubit on the quantum circuit. Their open boundary QCNN, in which qubits contain nearest neighbor connectivity only, shows advantages with quantum devices with limited physical qubit connectivity [43].

In 2020, Henderson et al. [39] introduced a new transformational layer for QNN structures, namely the *quanvolutional layer*. This layer is a collection of multiple smaller quantum filters that can be applied to a local section of input data features,

resulting in a shallow circuit depth. These quantum filters consist of randomly generated quantum circuits, with no defined structure and can be applied anywhere in a CNN layer stack. The authors compare the performance of three models, a CNN, a QNN with a single quanvolutional layer at the beginning of the structure, and a RANDOM model with a random classical non-linear transformation layer as the first layer. The results show consistent performance, stated as statistically indistinguishable by the authors, by QNN and RANDOM models, hence recognizing that a quanvolutional transform does not necessarily show a quantum advantage. However, it is imperative to note that a *Quanvolutional Neural Network* (QuanvNN) model needs multiple measurements and data encoding steps if the quanvolutional layer is applied several times between the classical convolutional or pooling layers, thereby increasing the cost of the overall model.

In 2021, Liu et al. [73] built a hybrid Quantum-Classical Convolutional Neural Network (QCCNN) similar to [39] that contains a parameterized quantum convolution filter constructed with alternating layers of single and two-qubit gates. The single-qubit layer contains rotational gates with variable parameters and hence is trainable, whereas the two-qubit layer contains CNOT gates. These layers can be applied in a quantum convolution filter multiple times until a suitable correlation measurement is received. These measurements are subsequently fed into the pooling and fully connected layers. The model is tested on the Tetris dataset with 800 grayscale images of shape 3×3, and achieves a classification accuracy of almost 100%, even in the presence of T_1 and T_2 randomly chosen from normal distributions with mean 50 and 70 μs. However, details on the display of this resilient behavior, or the presence of any overfitting are yet to be investigated.

In [77], MacCormack et al. propose a branch QCNN (bQCNN) structure, inspired from the stacked QCNN by [18], that branches out into a different set of convolutional layers after a pooling layer. Unlike QCNN which focuses on local information obtained from the measurement of adjacent qubits in a pooling layer, bQCNN utilizes the global information gained from all of the measured qubits. It heavily relies on the outcomes received from mid-circuit measurements of the fraction of qubits after every pooling layer to "branch out" the model to a distinct set of convolutional layers. As a result, multiple branching sets of parameterized quantum circuits are defined for a single model which may not be used for training purposes. This branching effect increases the total number of trainable parameters in the model with no increase in the circuit depth as compared to QCNN. The advantage of using global information from all

the mid-circuit measurements is to enhance the expressibility of the bQCNN model. The authors show a significant decrease in the training time required for the bQCNN model that achieves an accuracy of 74.3% compared to QCNN's accuracy of only 70.6% that trains slower than bQCNN.

2.3 QML Workflows

In a QML workflow, there is dynamic switching between classical and quantum processing platforms to train and optimize the variational parameters in the QML model. These HQC QML models employ quantum circuits containing trainable classical parameters to perform learning. In 2016, McClean et al. [80] were the first to propose a generalized HQC optimization framework to train such models. This approach drew inspiration from Peruzzo et al.'s VQE [92], which utilized both quantum and classical computational resources for eigenvalues estimation and optimizations.

The variational methodology employed here splits an algorithm into three core stages—quantum state preparation from classical data inputs, expectation value determination through measurement, and finally, optimization of variational parameters with a classical optimizer—that iteratively run to perform convergence of a quantum state function towards the desired outcome. O'Malley et al. [87] successfully applied this methodology to simulate molecular energies in 2016, followed by Zhou et al. [141] in 2020, for implementing the QAOA algorithm. This HQC approach has hence become a cornerstone in most variational algorithms.

HQC QML algorithms may be more complex, typically requiring classical pre- and post-processing sub-routines. As Leymann et al. [64] highlight, a consequence of the increase in the complexity of an otherwise linear machine learning workflow may require a developer to rigorously understand the algorithm requirements and the interplay between heterogeneous computational platforms [13, 127].

Pattern Language

As a solution to the arising challenges from increasing complexity in HQC models, Leymann proposed a pattern language [63]. Like structured documents, these patterns offer abstract solutions to recurring routines in the algorithm. Leyman et al. tailored these patterns to the quantum computing landscape to define implementation details for the data embedding (state preparation), oracle construction, and

quantum-classical split steps.

In 2023, Buhler et al. [14] identified five recurring patterns in a quantum software development workflow. These include Quantum Module and Quantum Module Templates, which implement reusable quantum structures, followed by a Hybrid Module that combines quantum and classical sub-routines of the algorithm. Additionally, a Classical-Quantum Interface was recognized to facilitate the use of quantum algorithms by non-experts, along with a Quantum Circuit Translator that transpiles quantum circuits onto a quantum device. Pattern languages for quantum algorithms have further expanded to assist in quantum software development, data encoding, and error handling, as shown by contributions from various authors [6, 128, 129].

More recently, in 2023, Qiskit [44] introduced Qiskit Patterns [109], a set of steps that streamline code reuse, and workflow simplification and acceleration in remote environments such as *High Performance Computing* (HPC) or cloud infrastructures.

Quantum Software Development Lifecycle and Workflow Automation

In 2020, several works [29, 126, 140] introduced the concept of the *Quantum Software Development Lifecycle* (QSDLC) to design and conceptualize quantum software systems. Building upon this framework, Perez-Castillo et al. [90, 94] and Perez-Delgado et al. [95] extended software modeling practices by adapting *Unified Modelling Language* (UML) to cater quantum software systems, termed as Q-UML.

Expanding on the understanding of QSDLC, Weder et al. proposed strategies for automated quantum workflow management in 2020 [127] and 2021 [125]. Introducing the Quantum Modeling Extension (QUANTME), the authors developed a modeling extension tool that abstracts complex quantum computations, such as the classical pre and post-processing sub-routines, making them more accessible for non-experts. They utilize QUANTME modeling constructs tailored to quantum workflows to introduce a workflow management tool further called Quantum4BPMN, based on BPMN.[1]

In 2019, Zapata et al. [135] developed QMflows, a workflow automation tool for quantum chemistry applications. The QMflows package provides flexibility with limited maintenance requirements for beginners and experienced programmers with limited technical knowledge.

[1] https://www.bpmn.org/

QML Software Frameworks

Several gate-based quantum software frameworks developed in Python exist that provide an interface with classical machine learning libraries to accommodate computational heterogeneity in HQC algorithms. One notable framework is PennyLane[2] [8], which provides a differential programming environment for QML algorithms. PennyLane stands out for its flexibility in seamlessly integrating with classical libraries such as TensorFlow,[3] PyTorch,[4] JAX [11], and Autograd [78].

Another significant contribution to this domain is TorchQuantum,[5] developed by Wang et al. [121], built upon the PyTorch library. TorchQuantum enables training and deployment of PQC on quantum devices and offers users efficient conversion between PyTorch quantum circuits to Qiskit QuantumCircuit structures. It also boasts a 200× speedup compared to PennyLane.

In parallel, Broughton et al. [13] developed TensorFlow Quantum (TFQ),[6] a software framework for prototyping HQC QML models. TFQ combines the capabilities of Cirq[7] and TensorFlow, aiming to reduce the technical prerequisites for researchers and students while designing and training QML models.

Additionally, Qiskit [44] offers a separate Qiskit Machine Learning library[8] that provides Qiskit Runtime tools and algorithms for QML tasks.

[2]https://pennylane.ai/
[3]https://www.tensorflow.org/
[4]https://pytorch.org/
[5]https://hanruiwanghw.wixsite.com/torchquantum
[6]https://www.tensorflow.org/quantum
[7]https://quantumai.google/cirq
[8]https://github.com/qiskit-community/qiskit-machine-learning

Chapter 3

Quantum Image Representation

3.1 Overview

The underlying framework of any QML algorithm comprises four approaches, initially introduced by Aimeur et al. [4] and subsequently expanded upon by Schuld et al. [104], which combine quantum and classical paradigms to facilitate interfacing of data with different information processing devices, refer Figure 3.1. In this context, classical data is information represented by bits, and quantum data by qubits.

CC: This approach utilizes classical data (C) to perform processing on a classical device (C), akin to traditional machine learning (ML) or quantum-inspired algorithms.

QC: Here, classical devices (C) process, define, or learn from provided quantum data (Q).

CQ: Often referred to as QML, this approach employs classical data (C) to perform learning tasks on quantum devices (Q).

QQ: This approach leverages quantum devices (Q) to process quantum data (Q), exploring quantum-native algorithms.

QML algorithms based on the hybrid quantum-classical approach [92] are synonymous with the CQ approach, wherein classical data features and observations, such as images, text, or probability distributions, are used as inputs to a quantum device that can undertake any learning tasks, including image processing.

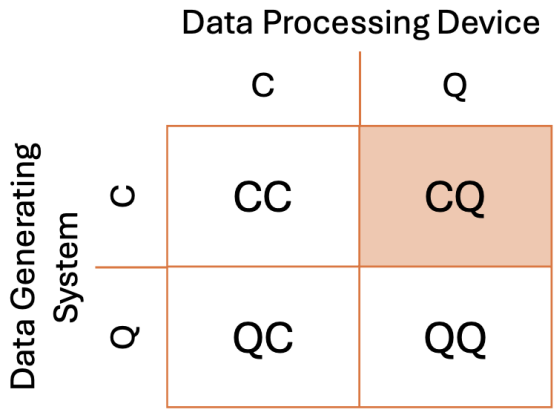


Figure 3.1: Four approaches to the underlying framework of QML. Redrawn from [102].

Data Embedding

Data embedding techniques play a crucial role in providing an interface between classical and quantum platforms by representing classical data on quantum devices, enabling their role in hybrid quantum-classical algorithms like QML. In addition to initializing quantum states, these techniques function as feature maps, effectively mapping classical inputs to a Hilbert Space [104]. The selection of data embedding techniques critically impacts the performance, in terms of space and time complexity, of any QML algorithm.

The role of data embedding techniques also extends to representing digital images on quantum devices. This application, often called *Quantum Image Representation* (QIR), consists of methods that capture pixel position and color information for efficient image representation. Moreover, these techniques go beyond mere representation to present ways for image transformation and compression, offering advancements in fields ranging from computer vision to data compression [124]. In summary, QIR is a powerful tool for encoding digital images on quantum devices, enabling advanced image processing within quantum computing.

While this chapter comprehensively discusses essential QIR techniques for embedding digital images on a quantum device, the role of data encodings as feature maps is discussed subsequently in Chapter 4.

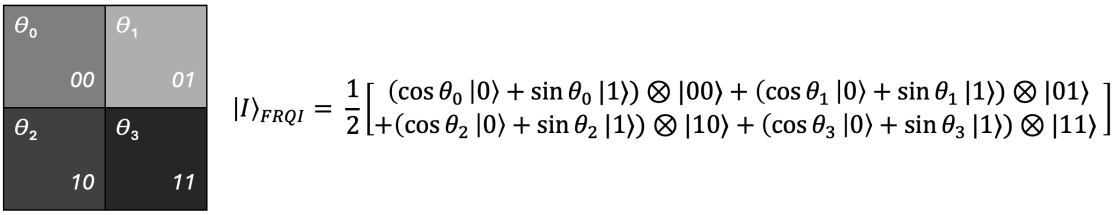


Figure 3.2: A 2×2 grayscale image and its FRQI state representation.

3.2 QIR Methods

Flexible Representation of Quantum Images (FRQI)

FRQI [59] is a technique for representing grayscale images on a quantum computer inspired by classical pixel representation methods. Employing unitary operations like Hadamard and controlled rotations, R_y, FRQI captures pixel position and color information. The quantum state representing a $2^n \times 2^n$ image using FRQI is expressed as,

$$|I(\theta)\rangle = \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} (\cos\theta_i|0\rangle + \sin\theta_i|1\rangle) \otimes |i\rangle \tag{3.1}$$

Here,

$$\theta \in \left[0, \frac{\pi}{2}\right] \quad \text{and} \quad i = 0, 1, ..., 2^{2n} - 1$$

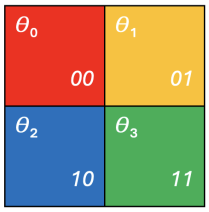
Color values C in range $[0, 2^q - 1]$ can be mapped to θ in range $[0, \frac{\pi}{2}]$ with a simple transform,

$$\theta = \frac{\pi}{2} \frac{C}{2^q - 1} \tag{3.2}$$

The proposed *Polynomial Preparation Theorem* (PPT) for FRQI provides a constructive process to implement image representation on a quantum circuit. It states that a unitary transformation $\mathcal{P}$ can effectively map qubits from an initial state to the FRQI state given in the above Equation 3.1 such that,

$$|0\rangle^{\otimes 2n+1} \xrightarrow{\mathcal{P}} |I(\theta)\rangle \tag{3.3}$$

This unitary transformation involves multiple sequential steps. First, a Walsh-Hadamard transform is applied, which uses superposition to encode pixel positions across $n + n$ qubits for x and y-coordinates in the computational basis states. Subse-



$$|I\rangle_{FRQCI} = \frac{1}{2}\left[\begin{array}{l}(\cos\frac{\theta_0}{2}|0\rangle + e^{i\phi_0}\sin\frac{\theta_0}{2}|1\rangle)\otimes|00\rangle + (\cos\frac{\theta_1}{2}|0\rangle + e^{i\phi_1}\sin\frac{\theta_1}{2}|1\rangle)\otimes|01\rangle \\ +(\cos\frac{\theta_2}{2}|0\rangle + e^{i\phi_2}\sin\frac{\theta_2}{2}|1\rangle)\otimes|10\rangle + (\cos\frac{\theta_3}{2}|0\rangle + e^{i\phi_3}\sin\frac{\theta_3}{2}|1\rangle)\otimes|11\rangle\end{array}\right]$$

Figure 3.3: A 2×2 colored image and its FRQCI state representation.

quently, a controlled rotation transform is applied, employing controlled-$R_y(2\theta)$ gates to encode the color information in the probability amplitude of the quantum state initialized during the first step.

$$|0\rangle^{\otimes 2n+1} \xrightarrow{H} |H\rangle \xrightarrow{\mathcal{R}} |I(\theta)\rangle \tag{3.4}$$

FRQI effectively leverages quantum mechanical principles such as superposition to store normalized images. However, it has constraints in handling colored images. Additionally, since the color information is stored in the probability amplitude of a quantum state, image retrieval for an image embedded using the FRQI method is probabilistic. In response to these limitations, several extensions of the FRQI method have been proposed. These include FRQCI and MCRQI, explained in the subsequent sections.

Flexible Representation of Quantum Color Images (FRQCI)

FRQCI [69] extends the foundational principles of the FRQI method to support the representation of colored images. In the FRQCI method, the color information of a $2^n \times 2^n$ image is encoded in the phase θ_k of a quantum state given as,

$$|I(\theta,\phi)\rangle = \frac{1}{2^n}\sum_{k=0}^{2^{2n}-1}\left(\cos\theta_k|0\rangle + e^{i\phi_k}\sin\theta_k|1\rangle\right)\otimes|k\rangle \tag{3.5}$$

$$\theta_k = \frac{(c_k^R \times 2^{16} + c_k^G \times 2^8 + c_k^B)\times\pi}{2^{24}-1} \quad \text{and} \quad \phi_k = 0$$

Where,

$$\theta \in \left[0, \frac{\pi}{2}\right] \quad \text{and} \quad k = 0, 1, ..., 2^{2n}-1$$

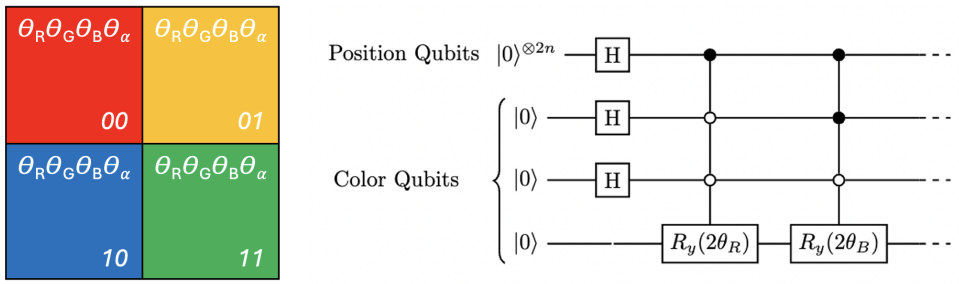


Figure 3.4: A 2×2 colored image and its MCRQI circuit representation.

The remaining phase ϕ_k is optionally used to encode a watermark image in the quantum circuit, providing additional functionality. FRQCI utilizes $2n$ qubits to encode the x and y-coordinates of the pixel positions with an additional qubit for encoding the color information. Akin to the FRQI method, image retrieval in FRQCI is probabilistic.

Multi-Channel Representation of Quantum Images (MCRQI)

MCRQI [112] is another image representation method that expands the FRQI method to capture the RGBα channel space to encode color spectrum (RGB) and opacity (α) information. The quantum state of a $2^n \times 2^n$ image embedded with MCRQI is represented as,

$$|I(\theta)\rangle = \frac{1}{2^{n+1}} \sum_{i=0}^{2^{2n}+1} |c^i_{RGB\alpha}\rangle \otimes |i\rangle \tag{3.6}$$

Where $|c^i_{RGB\alpha}\rangle$ is the color information that encodes the $RGB\alpha$ channels as,

$$\begin{aligned} |c^i_{RGB\alpha}\rangle = &(\cos(\theta)_{Ri}|0\rangle + \sin(\theta)_{Ri}|1\rangle) \otimes |00\rangle \\ &+ (\cos(\theta)_{Gi}|0\rangle + \sin(\theta)_{Gi}|1\rangle) \otimes |01\rangle \\ &+ (\cos(\theta)_{Bi}|0\rangle + \sin(\theta)_{Bi}|1\rangle) \otimes |10\rangle \\ &+ (\cos(\theta)_{\alpha i}|0\rangle + \sin(\theta)_{\alpha i}|1\rangle) \otimes |11\rangle \end{aligned} \tag{3.7}$$

with

$$\theta \in \left[0, \frac{\pi}{2}\right] \quad \text{and} \quad i = 0, 1, ..., 2^{2n} - 1$$

As per the *Polynomial Preparation Theorem* for the MCRQI method, the Walsh-

Hadamard transform helps store the pixel positions in their computational basis states. Control qubits for $RGB\alpha$ channels are encoded in two extra qubits using the same transform. Subsequently, the control rotation transform encodes the color and opacity information in the probability amplitude of the quantum state. This procedure requires a total of $2n + 3$ qubits to store pixel position and $RGB\alpha$ channels, respectively.

Novel Enhanced Quantum Representation (NEQR)

NEQR [136] method offers a distinct approach to image representation compared to FRQI. Unlike FRQI, where information is encoded in the probability amplitude, NEQR uses the basis states of the qubits to represent color information, resulting in a deterministic image retrieval process. However, NEQR can only represent grayscale images.

In this method, two entangled qubit sequences of $2n + q$ qubits store the position and color information of the entire image. The grayscale color information in the range $[0, 2^q - 1]$ is converted to a binary format to be stored in q qubits. For a $2^n \times 2^n$ image, denoted by (Y, X) pixel position, NEQR representation is given as,

$$|I\rangle = \frac{1}{2^n} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} |f(Y,X)\rangle |YX\rangle = \frac{1}{2^n} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} \bigotimes_{i=0}^{q-1} |C_{YX}^i\rangle |YX\rangle \tag{3.8}$$

Where,

$$f(Y,X) = C_{YX}^0 C_{YX}^1 \dots C_{YX}^{q-1} \tag{3.9}$$

with

$$C_{YX}^k \in \{0,1\} \quad \text{and} \quad f(Y,X) \in [0, 2^q - 1]$$

The NEQR method is restricted to storing square images, presenting a significant limitation. However, alternative methods like INEQR and GQIR offer image representations to overcome this limitation.

Improved Novel Enhanced Quantum Representation (INEQR)

Improved Novel Enhanced Quantum Representation (INEQR) [47] improves the NEQR method to represent non-square images, enabling support for image resizing and scaling. A $2^{n_1} \times 2^{n_2}$ grayscale image can be represented with INEQR as,

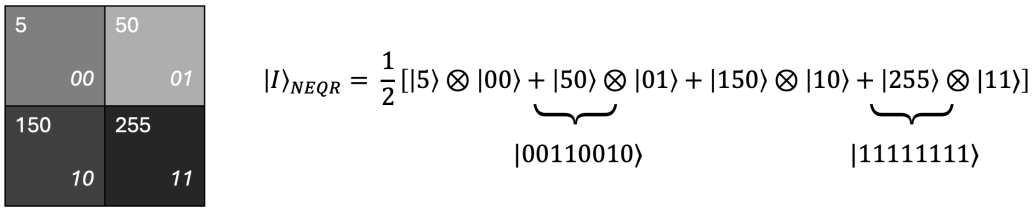


Figure 3.5: A 2×2 grayscale image and its NEQR state representation.

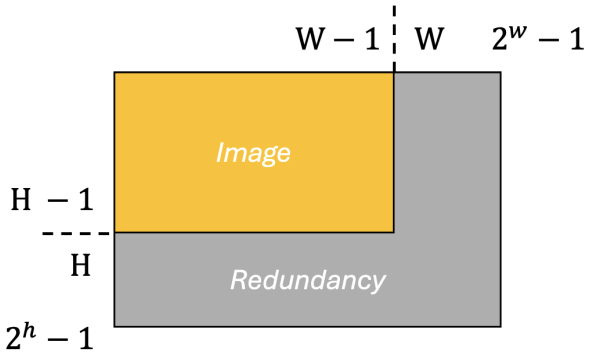


Figure 3.6: A $2^h \times 2^w$ image box for a $H \times W$ image, displaying the redundant rows and columns.

$$|I\rangle = \frac{1}{2^{\frac{n_1+n_2}{2}}} \sum_{Y=0}^{2^{n_1}-1} \sum_{X=0}^{2^{n_2}-1} |f(Y,X)\rangle |YX\rangle = \frac{1}{2^{\frac{n_1+n_2}{2}}} \sum_{Y=0}^{2^{n_1}-1} \sum_{X=0}^{2^{n_2}-1} \bigotimes_{i=0}^{q-1} |C_{YX}^i\rangle |YX\rangle \quad (3.10)$$

Where,

$$|YX\rangle = |y_0 y_1 \dots y_{n_1-1}\rangle |x_0 x_1 \dots x_{n_2-1}\rangle \quad \text{and} \quad y_i, x_i \in \{0,1\} \quad (3.11)$$

INEQR requires $n_1 + n_2$ qubits to encode pixel positions for non-square images with unequal horizontal (2^{n_1}) and vertical (2_{n_2}) dimensions. However, like NEQR, INEQR has limitations in supporting colored images.

Generalized Quantum Image Representation (GQIR)

Like INEQR, the Generalized Quantum Image Representation (GQIR) method can encode images of arbitrary dimensions $H \times W$. This method utilizes $h = \lceil log_2 H \rceil$ qubits to encode y-coordinates and $w = \lceil log_2 W \rceil$ to encode x-coordinates of the pixel positions. A $H \times W$ grayscale image encoded with GQIR can be represented as,

$$|I\rangle = \frac{1}{2^{\frac{H+W}{2}}} \sum_{Y=0}^{H-1} \sum_{X=0}^{W-1} \bigotimes_{i=0}^{q-1} |C_{YX}^{i}\rangle |YX\rangle \tag{3.12}$$

Where,

$$h = \begin{cases} \lceil \log_2 H \rceil, & \text{if } H > 1 \\ 1, & \text{if } H = 1 \end{cases}$$

$$w = \begin{cases} \lceil \log_2 W \rceil, & \text{if } W > 1 \\ 1, & \text{if } W = 1 \end{cases} \tag{3.13}$$

Furthermore, GQIR can be extended to support colored images by introducing additional color qubits to encode a color range $[0, 2^q - 1]$, where $q = 2$ for binary, $q = 8$ for grayscale, and $q = 24$ for colored images. However, this image representation method produces inevitable redundancies due to the binary representation of the pixel positions. These redundancies account for a total of $2^h - H$ rows and $2^w - W$ columns within a $2^h \times 2^w$ image box, resultant to the GQIR representation on $h + w$ qubits, as illustrated in Figure 3.6.

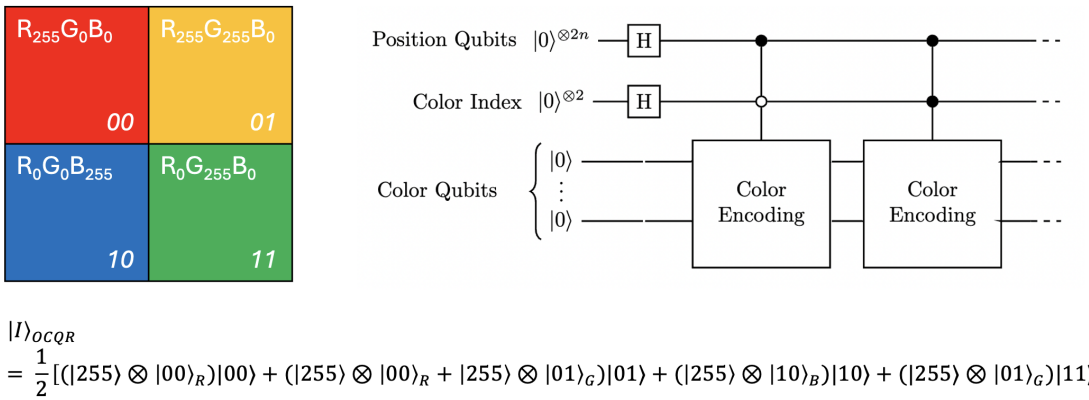
Optimized Quantum Representation for Colored Digital Images (OCQR)

Optimized Quantum Representation for Colored Digital Images (OCQR) [74] introduces the concept of indices for different color channels. This model utilizes three-dimensional qubit sequences to store the position information, color information, and corresponding channel indices. The position information is stored in $2n$ qubits, while color information and index require $q + 2$ qubits. A color channel index helps to identify the RGB color channels, where $|00\rangle$ is red, $|01\rangle$ is green, and $|10\rangle$ is blue channel index, whereas $|11\rangle$ is a spare channel.

A $2^n \times 2^n$ image in OCQR encoding can be represented as

$$|I\rangle = \frac{1}{2^{n+1}} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} |C(Y,X)\rangle |C_{\text{index}}|\rangle |YX\rangle \tag{3.14}$$

The remaining spare channel index can be utilized to store image opacity information. A significant limitation of OCQR is that it can only represent square images.



$$|I\rangle_{OCQR} = \frac{1}{2}[(|255\rangle \otimes |00\rangle_R)|00\rangle + (|255\rangle \otimes |00\rangle_R + |255\rangle \otimes |01\rangle_G)|01\rangle + (|255\rangle \otimes |10\rangle_B)|10\rangle + (|255\rangle \otimes |01\rangle_G)|11\rangle]$$

Figure 3.7: A 2×2 colored image with its OCQR state and circuit representations.

Bitplane Representation of Quantum Images (BRQI)

The Bitplane Representation of Quantum Images (BRQI) [65] method uniquely helps in representing images through binary color values $\in \{0, 1\}$.

According to BRQI encoding, a grayscale image can be interpreted as a composition of eight bitplanes. That is, each grayscale color value in the range $[0, 2^q - 1]$ can be decomposed into q binary color bits in the set $\{0, 1\}$. These bits collectively form q bitplanes that can be embedded as a superposition on the corresponding pixel index using $p = \log q$ qubits. Here, each binary color value uses a single qubit for encoding. The pixel index representation remains the same as proposed in NEQR [136] that requires a total of $2n$ qubits for an image of size $2^n \times 2^n$.

Each bitplane can be encoded using a GQIR method as in Equation 3.12 such that,

$$|\Psi_m^q\rangle = |I^{n_1+n_2}\rangle_{\text{GQIR}} \tag{3.15}$$

Where, q denotes the q-th bitplane of m-th binary color bit, such that,

$$q = n_1 + n_2 = 8, \quad m = \{0, 1\}$$

An RGB image, on the other hand, can be viewed as a composition of three grayscale images. For the BRQI representation of colored images, an additional two qubits are used to map different color channels through their indices, leaving a spare channel with an index of 00. The BRQI state of a $2^n \times 2^n$ colored image, therefore, can be given as,

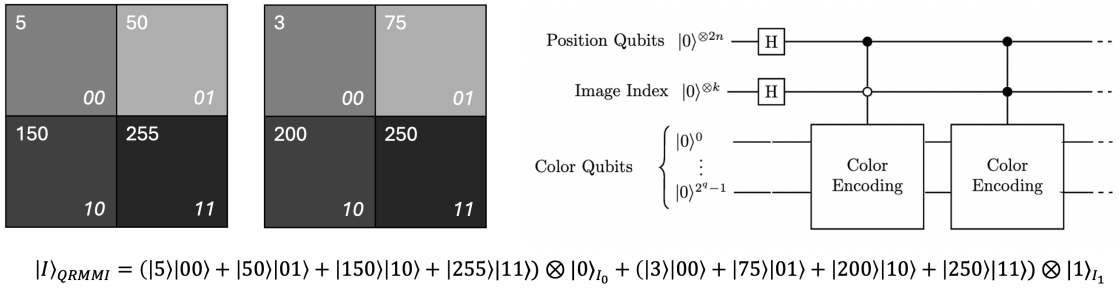


Figure 3.8: Two 2×2 grayscale images with their QRMII state and circuit representations.

$$|\Psi\rangle = \frac{1}{2^{\frac{2n+p}{2}}} \sum_{l=0}^{q-1} |\Psi_m^{2n}\rangle |l\rangle = \frac{1}{2^{\frac{2n+p}{2}}} \sum_{l=0}^{q-1} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} |C(Y,X)\rangle |YX\rangle |l\rangle \tag{3.16}$$

where,

$$|C(Y,X)\rangle \in \{0,1\} \quad \text{and} \quad p = \log q$$

Hence, the qubit requirement for the BRQI method is $2n+p+1$ for grayscale images and $2n+p+3$ for colored images.

Quantum Representation Model for Multiple Images (QRMMI)

Quantum Representation Model for Multiple Images (QRMMI) [140] introduces a method to encode multiple colored images leveraging the principle of quantum superposition. QRMMI utilizes two entangled qubit sequences to represent multiple images in their basis states, leading to a deterministic image retrieval process. The first sequence encodes the color information, while the second sequence stores the pixel position and an index marking the image. A total of 2^k grayscale images of size $2^n \times 2^n$ can be represented with QRMMI as,

$$|I\rangle = \frac{1}{\sqrt{2}^{2n+k}} \sum_{K=0}^{2^k-1} \sum_{Y=0}^{2^n-1} \sum_{X=0}^{2^n-1} |f_K(Y,X)\rangle |KYX\rangle \tag{3.17}$$

Where the color information and range is given as,

$$f_K(Y,X) = C_{KYX}^{q-1} C_{KYX}^{q-2} \dots C_{KYX}^0, \quad C_{KYX}^q \in \{0,1\}, \quad \text{and} \quad f_K(Y,X) \in [0, 2^q - 1] \tag{3.18}$$

Therefore, the total qubit requirement to store 2^k images with QRMMI is $2n+q+k$ qubits. These qubits are allocated for encoding pixel position, color information, and image index, respectively. Additionally, QRMMI can be extended to represent colored images by increasing the number of q color qubits for a color range of $[0, 2^q - 1]$.

Chapter 4

Quantum Machine Learning

In this chapter, we examine the theoretical foundations of selective QML algorithms for image classification. Section 4.1 introduces Quantum Feature Maps and Kernels, providing an example of a Pauli Feature Map in Subsection 4.1.1. Following this, in Section 4.2 we discuss Kernel Methods, including SVM in Subsection 4.2.1. Based on this, we discuss the steps to build a Kernel Estimator in Subsection 4.2.2. Subsequently, VQA are detailed in Section 4.3, along with an example of a *Variational Quantum Classifier* (VQC) in Subsection 4.3.1. Finally, we discuss various circuit architectures and unitary parameterizations used for building QTN and QNN in Sections 4.4 and 4.5. Additionally, we provide examples of a *Hierarchical Quantum Classifier* (HC) in Subsection 4.4.1, and a QCNN in Subsection 4.5.1, offering a practical understanding of the theoretical constructs for QML.

4.1 Quantum Feature Maps and Kernels

We define feature maps ϕ as functions that map classical data inputs x from an input set χ to a feature space $\mathcal{F}$. A feature map is given as,

$$\phi : x \in \chi \mapsto |\phi(x)\rangle \in \mathcal{F}, \tag{4.1}$$

Such a feature space necessitates the definition of an inner product, given as a bivariate kernel κ,

$$\kappa(x, x') = \langle \phi(x') | \phi(x) \rangle \tag{4.2}$$

Kernel functions effectively capture non-linear relationships between data features

as if mapped to a higher-dimensional feature space.

Quantum Kernel

In Chapter 3, we discussed various data embedding techniques that encode images on a quantum device. Such data embeddings map the classical image features x from an input space χ to the quantum state space $\mathcal{F}$ [103, 104]. These encodings can be given as a *quantum feature map* ρ such that,

$$\rho : x \mapsto \rho(x) \tag{4.3}$$

where, $\rho(x)$ is the density matrix defined as,

$$\rho(x) = |\phi(x)\rangle\langle\phi(x)| \tag{4.4}$$

Given a quantum feature map ρ, a corresponding *quantum kernel* [104] is expressed as

$$\kappa_q(x, x') = |\langle\phi(x')|\phi(x)\rangle|^2 \tag{4.5}$$

Additional information about quantum kernels is covered in Sections 4.2 and 4.4.

4.1.1 Example: Pauli Feature Map

Havlicek et al. [38] proposed a family of feature maps, called the Pauli Feature Map, that implements a kernel function given in Equation 4.5. The Pauli feature map encodes data $x \in \chi, \chi \in \mathbb{R}^n$ in the quantum feature space $\mathcal{F}$. It utilizes n qubits to have the following representation,

$$\mathcal{U}_\Phi(x) = U_{\Phi(x)} H^{\otimes n} U_{\Phi(x)} H^{\otimes n} \tag{4.6}$$

where $H^{\otimes n}$ is a sequence of Walsh-Hadamard transforms applied to the n qubits, and,

$$U_{\Phi(x)} = \exp\left(i \sum_{S \subseteq [n]} \phi^S(x) \prod_{i \in S} P_i \right) \tag{4.7}$$

Here, the unitary operator $U_{\Phi(x)}$ is expressed as a diagonal in a Pauli basis P_i, where $i = \{I, X, Y, Z\}$ [113].

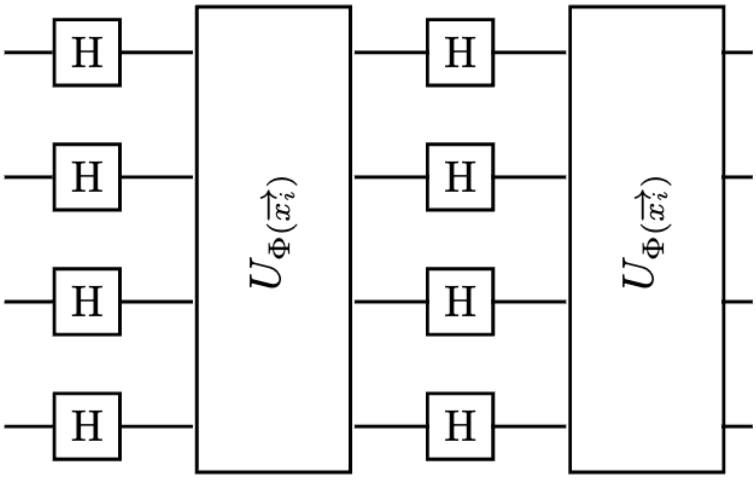


Figure 4.1: Circuit diagram for the Pauli feature map $\mathcal{U}_\Phi(x)$ with layers of Walsh-Hadamard transform and unitary operators $U_\phi(x)$. Redrawn from [38].

Hence, the quantum state produced from the application of the Pauli feature map, Equation 4.6, on a set of initial states in state $|0\rangle^{\otimes n}$ can be given as,

$$|\Phi(x)\rangle = \mathcal{U}_\Phi(x)|0\rangle^{\otimes n} \tag{4.8}$$

There exist 2^n non-linear feature maps $\phi^S(x) \in \mathbb{R}$ that can be expressed with $|S|$ degrees of interactions between the individual qubits, where $S \in \{\binom{n}{k}$ combinations, $k = 1, \ldots n\}$. For $|S| \leq 2$ interactions, the local feature maps are governed by the following data mapping strategy[1],

$$\phi^S_{\{k\}}(x) = \begin{cases} x_0 & \text{if } k = 1 \\ \prod_{j=0}^{k-1}(\pi - x_j) & \text{otherwise} \end{cases} \tag{4.9}$$

These interactions can be carefully selected based on the connectivity graphs of the hardware being used to create short-depth quantum circuits [38].

For example, unitary operator $U_{\Phi(x)}$ with $|S| \leq 2$ from the family of Pauli-ZZ feature maps can be represented as,

$$\begin{aligned} U_{\phi\{k\}(x)} &= \exp\left(i\phi_{\{k\}}(x)Z_k\right) && \text{for } |S| = 1 \\ U_{\phi\{l,m\}(x)} &= \exp\left(i\phi_{\{l,m\}}(x)Z_lZ_m\right) && \text{for } |S| = 2 \end{aligned} \tag{4.10}$$

Using the mapping strategy defined in Equation 4.9, the following local feature

[1]https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.PauliFeatureMap

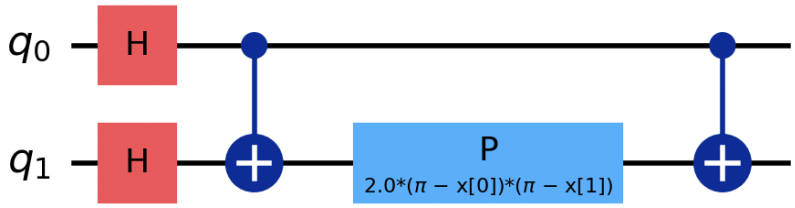


Figure 4.2: Circuit diagram of the unitary $U_{\Phi(x)}$, from the family of Pauli-ZZ feature maps, with $|S| \leq 2$ qubit interactions as built with Qiskit.

maps can be constructed,

$$\phi_{\{0\}}(x) = x_0 \qquad \phi_{\{0,1\}}(x) = (\pi - x_0)(\pi - x_1) \tag{4.11}$$

Therefore, the resulting unitary can be given as,

$$U_{\Phi(x)} = \exp\left(ix_0 Z_0 + ix_1 Z_1 + i(\pi - x_0)(\pi - x_1) Z_0 Z_1\right) \tag{4.12}$$

This unitary $U_{\Phi(x)}$ can be built in Qiskit, as shown in Figure 4.2.

4.2 Kernel Methods

Kernel methods are algorithms that provide the similarity measure between data features [104]. This similarity measure is expressed as an inner product of the data features, called the kernel, as given in Equation 4.2. Akin to classical kernel methods, quantum kernel methods can map data features into higher-dimensional feature spaces and classify them with a linear decision hyperplane defined via the process of measurement [38, 102]. Consequent to this property, the feature space of a quantum state space should be defined with a quantum kernel, as represented in Equation 4.5.

4.2.1 Support Vector Machines

An SVM performs data classification by producing a maximum-margin decision hyperplane in a high dimensional space. Let us consider a set of data attributes $\{(x_i, y_i)\,; i = 1, ..., n\}$, with x_i as the input features and y_i as the output label of i^{th} data. For a binary classification problem, with $y_i \in \{-1, +1\}$, the SVM algorithm non-linearly maps the input features from a low dimensional input space $\chi \in \mathbb{R}^d$ to a high dimensional feature space $\mathcal{F} \in \mathbb{R}^p$, where $p > d$. This mapping can be

represented as,

$$\phi : x \in \mathbb{R}^d \mapsto \phi(x) \in \mathbb{R}^p \tag{4.13}$$

Subsequently, the classifier decision hyperplane can be represented as a non-linear function,

$$h_{w,b}(x_i) = g(w^T \phi(x_i) + b) \tag{4.14}$$

Where $w \in \mathbb{R}^p$ is the set of weights in the mapping function, and $w_0 = b$ is a bias term. The corresponding threshold function g is governed by the rule,

$$g(z) = \begin{cases} 1 & \text{if } z \geq 1 \\ -1 & \text{otherwise} \end{cases} \tag{4.15}$$

Since SVM constructs a maximum-margin decision hyperplane, the central task of the model is to find a confident prediction such that the data features in either set of a binary classification problem have the largest possible distance from the decision hyperplane. This distance is referred to as the functional margin, given as $\gamma_i \quad \forall \quad i$. A confident prediction results in $y_i = 1$ when $w^T \phi(x_i) + b \gg 0$, and $y_i = -1$ when $w^T \phi(x_i) + b \ll 0$. Hence, a large functional margin corresponds to the correctness and confidence of the prediction [82].

Given the data inputs, the functional margin for a training input can be formalized as,

$$\hat{\gamma_i} = y_i(w^T \phi(x_i) + b) \tag{4.16}$$

We further normalize the functional margin, making it invariant to the rescaling of the parameters. This margin is called the geometric margin and can be given as,

$$\hat{\gamma_i} = \|w\| \gamma_i \tag{4.17}$$

Which for training input can be expanded as,

$$\gamma_i = y_i \left(\left(\frac{w}{\|w\|} \right)^T \phi(x_i) + \left(\frac{b}{\|w\|} \right) \right) \tag{4.18}$$

Now, the maximum margin classifier problem can be formulated as an optimization problem, given as,

$$\begin{aligned} &\max_{\gamma,w,b} \quad \gamma \\ &\text{s.t.} \quad y_i(w^T\phi(x_i)+b) \geq \gamma, \quad i=1,...,n \quad \text{and} \quad \|w\|=1 \end{aligned} \tag{4.19}$$

Primal Form SVM

The SVM algorithm provides two forms of optimizations - primal and dual. The primal form transforms the Equation 4.19 to remove the constraint of $\|w\| = 1$, resulting in

$$\begin{aligned} &\max_{\hat{\gamma}_i,w,b} \quad \frac{\hat{\gamma}_i}{\|w\|} \\ &\text{s.t.} \quad y_i(w^T\phi(x_i)+b) \geq \hat{\gamma}_i, \quad i=1,...,n \end{aligned} \tag{4.20}$$

To simplify the objective function $\frac{\hat{\gamma}_i}{\|w\|}$, a scaling constraint is introduced such that $\hat{\gamma}_i = 1$. The optimization problem finally results in,

$$\begin{aligned} &\min_{w,b} \quad \frac{1}{2}\|w\|^2 \\ &\text{s.t.} \quad y_i(w^T\phi(x_i)+b) \geq 1, \quad i=1,...,n \end{aligned} \tag{4.21}$$

Dual Form SVM

Further, SVM can be represented as the Lagrangian of the primal form, called the dual form, to further derive an efficient representation of the optimization problem.

First, the constraints of Equation 4.21 are rewritten as,

$$y_i(w^T\phi(x_i)+b) - 1 \geq 0 \tag{4.22}$$

Subsequently, the Lagrangian of the primal optimization problem, as given in Equation 4.21, can be written as,

$$\mathcal{L}(w,b,\alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i \left[y_i \left(w^T\phi(x_i)+b \right) - 1 \right] \tag{4.23}$$

Where α_i is the Lagrange multiplier for the inequality constraint in Equation 4.22. Subsequently, we take the derivative of this Lagrangian with w and b, respectively.

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{n} \alpha_i y_i \phi(x_i) = 0 \tag{4.24}$$

and,

$$\nabla_b \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{4.25}$$

Which implies,

$$w = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i) \tag{4.26}$$

Plugging the Equation 4.26 in Equation 4.23, the Lagrangian of the dual form can be written as,

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \phi(x_i)^T \phi(x_j) - b \sum_{i=1}^{n} \alpha_i y_i \tag{4.27}$$

Now, plugging the Equation 4.25 in Equation 4.27,

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \phi(x_i)^T \phi(x_j) \tag{4.28}$$

Finally, the dual form of the SVM optimization problem can be given as,

$$\begin{aligned} \max_{\alpha} \quad & \mathcal{L}_D(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \langle \phi(x_i), \phi(x_j) \rangle \\ \text{s.t.} \quad & \alpha_i \geq 0 \quad i = 1, ..., n \quad \text{and} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{4.29}$$

In this formulation, $\langle \phi(x_i), \phi(x_j) \rangle$ is a kernel function representing the inner product of the feature vectors $\phi(x_i)$ and $\phi(x_j)$. This implies that the maximum-margin decision hyperplane can be determined using a kernel trick, bypassing the mapping of feature vectors in a high-dimensional space. By leveraging the kernel trick, the SVM is well-suited for implementation on a quantum computer.

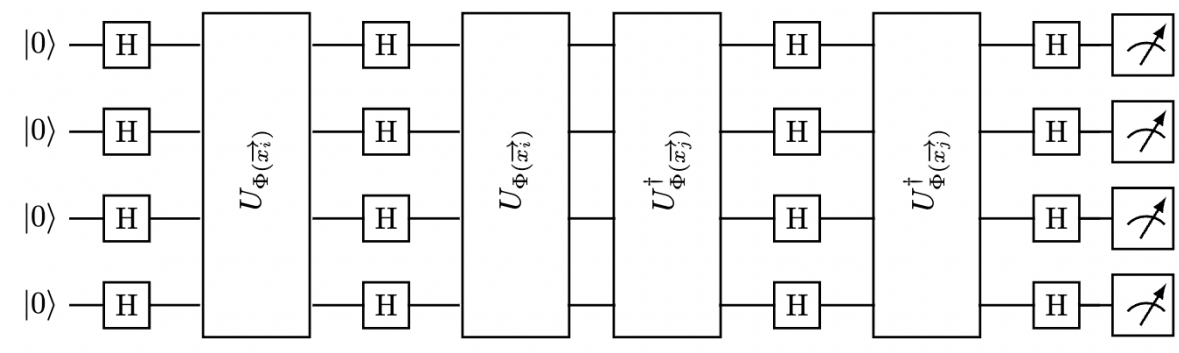


Figure 4.3: The circuit implemented to estimate the fidelity between two data points x_i and x_j. The unitary operators $U_\phi(x_i)$ and $U_\phi(x_j)$ are given by Equation 4.7. Redrawn from [38].

4.2.2 Example: Quantum Kernel Estimator

Havlicek et al. [38] proposed a quantum kernel estimation model that classifies data feature inputs $x \in \mathbb{R}$ by estimating a kernel $\kappa(x_i, x_j)$ on a quantum computer. The proposed algorithm consists of three steps:

Step 1: Estimate the kernel

To implement the kernel estimator, a circuit is constructed by applying the Pauli feature map $\mathcal{U}_\Phi(x_i)$ on data input x_i, as detailed in Subsection 4.1.1. This is followed by applying the inverse of the same feature map $\mathcal{U}_\Phi(x_j)$ to another data input x_j. The circuit is then measured in the Pauli-Z basis, yielding the outcome representing the overlap between the two data features. This circuit is shown in Figure 4.3.

Step 2: Maximize the dual form SVM

Once the kernel is estimated for all the pairs of training data, the optimization problem in the dual form of SVM is formulated to obtain the optimal decision hyperplane. This involves finding the solution to Equation 4.29.

Step 3: Estimate the kernel for unseen data

Following the training phase, where the kernel and support vectors were identified, kernel estimation on new unseen data $s \in \mathcal{S}$ is performed using $\kappa(x_i, s)$.

4.3 Variational Quantum Algorithms

VQA are HQC algorithms that contain tunable unitary operations called PQC. These algorithms outsource complex subroutines in an algorithm, like QML algorithms, to quantum computational resources while performing classical pre and post-processing routines. The general workflow for any quantum variational algorithm [23, 80, 87, 92, 141] is split into three core stages—quantum state preparation from classical data inputs, expectation value determination through measurement, and finally, optimization of parameters with a classical optimizer—that iteratively run to perform convergence of a quantum state function towards the desired outcome.

The circuits $U(x, \theta)$ for quantum variational algorithms depend on data inputs x, and some ansatz, a sequence of parameterized and fixed gates, that define the architecture of the algorithm [104]. While the circuit can have an arbitrary internal structure, a popular architecture is a stacking of a data embedding block $S(x)$ and a parameterized block $W(\theta)$.

$$U(x, \theta) = W(\theta)S(x) \tag{4.30}$$

Such that, a n-qubit state prepared by $U(x, \theta)$ can be given as,

$$|\Phi(x)\rangle = U(x, \theta)|0\rangle^{\otimes n} \tag{4.31}$$

The data embedding block in Equation 4.30 serves as a feature map for mapping data features $x \in \chi$ onto a quantum state space $\mathcal{F}$. This step can be performed by data embedding techniques, as discussed in Chapter 3, or feature maps such as the Pauli feature map $\mathcal{U}_\Phi$ given in Equation 4.6.

The parameterized block $W(\theta)$ on the other hand depends on some variational parameters θ and is subsequently applied to the quantum state prepared by the data embedding block. The circuit is finally measured through a measurement operator $\mathcal{M}$, such that the circuit function can be represented as,

$$\begin{aligned} f_\theta(x) &= \langle \phi(x)|W^\dagger(\theta)\mathcal{M}W(\theta)|\phi(x)\rangle \\ &= \langle \Phi(x, \theta)|\mathcal{M}|\Phi(x, \theta)\rangle \end{aligned} \tag{4.32}$$

Here, the measurement operator $\mathcal{M}$ is a diagonal operator $\mathcal{M}$ in any Pauli-α basis. For example, in Pauli-Z basis $\mathcal{M}$ can be represented as,

$$\mathcal{M} = \sum_{z \in 0,1^n} f(z)|z\rangle\langle z| \tag{4.33}$$

Applying Equation 4.33 in Equation 4.32, we get

$$f_\theta(x) = \sum_z f(z)|\langle z|\Phi(x,\theta)\rangle|^2 = \sum_z f(z)p(z) \tag{4.34}$$

Where, $p(z) = |\langle z|\Phi(x,\theta)\rangle|^2$ is the probability of measuring the outcome z. For a Pauli-Z basis measurement, it can be expanded as,

$$f_\theta(x) = |\langle 0|\Phi(x,\theta)\rangle|^2 - |\langle 1|\Phi(x,\theta)\rangle|^2 = p(0) - p(1) \tag{4.35}$$

Additionally, the measurement results from Equation 4.34 can be mapped to obtain the eigenvalues with the help of predefined functions. For example, in a binary classification task, a boolean function can be defined as,

$$b : \{0,1\}^n \mapsto \{+1,-1\} \tag{4.36}$$

The variational circuit $f_\theta(x)$ goes through the HQC training loop [80] to learn the optimal variational parameters in the parameterized block $W(\theta)$. This involves performing M measurements and averaging over the results to obtain,

$$\hat{f}(x) = \frac{1}{M}\sum_{m=1}^{M} f(z)^m \tag{4.37}$$

Where $f(z)^m$ is the eigenvalue obtained from each measurement outcome.

4.3.1 Example: Variational Quantum Classifier

Havlicek et al. [38] introduced a variational quantum classifier model for binary classification tasks on a dataset $x \in \mathbb{R}$. The *Variational Quantum Classifier* (VQC) circuit consists of a data embedding step $S(x)$, constructed with the help of the Pauli feature map $\mathcal{U}_\Phi(x)$, as detailed in Section 4.1.1. This embedding process encodes the classical data inputs into a quantum state suitable for subsequent processing.

Following the data embedding block a parameterized block $W(\theta)$ is implemented. It is constructed as a series of single-qubit unitaries and entangling gate layers. The structure of the parameterized block, with l repetitions of the entangling layer, is

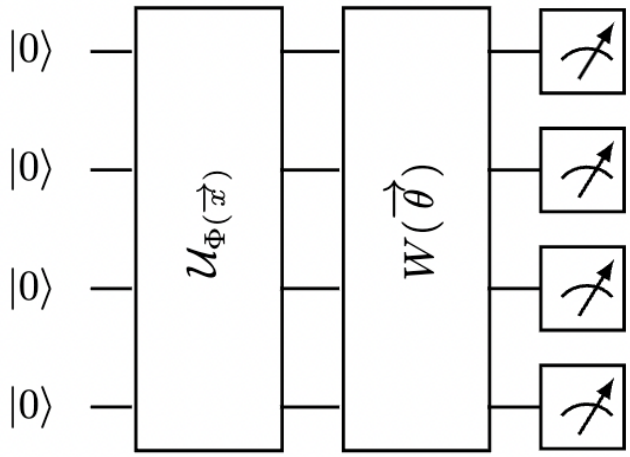


Figure 4.4: Circuit diagram of the variational quantum classifier model that utilizes the Pauli feature map $\mathcal{U}_\Phi$ as a data embedding block, followed by a parameterized block $W(\theta)$. Redrawn from [38].

given as,

$$W(\theta) = U^l_{\text{loc}}(\theta_l)U_{\text{ent}} \ldots U^2_{\text{loc}}(\theta_2)U_{\text{ent}}U^1_{\text{loc}}(\theta_1)U_{\text{ent}} \tag{4.38}$$

The local single-qubit unitaries U_{loc} are implemented as rotation gates, such as R_y or R_z, and are parameterized with angles θ. These local unitaries can be given as,

$$\begin{aligned} U^t_{\text{loc}}(\theta_t) &= \bigotimes_{m=1}^{M} U(\theta_{m,t}) \\ \text{s.t.} \quad U(\theta_{m,t}) &= e^{i\frac{1}{2}\theta^z_{m,t}Z_m} e^{i\frac{1}{2}\theta^y_{m,t}Y_m} \end{aligned} \tag{4.39}$$

While the entangling layer comprises a sequence of controlled-phase gates, given as,

$$U_{\text{ent}}(\theta) = \prod_{(i,j)\in E} CZ(i,j) \tag{4.40}$$

Where, (i, j) are the entangling interactions between qubits i and j according to the connectivity graph, $G = (V, E)$, of quantum device.

The central aim of the VQC is to find optimal parameter values for $W(\theta)$, with the help of an optimizer, that can effectively classify data inputs with distinct labels. The resulting circuit, as shown in Figure 4.4, is executed and sampled multiple times

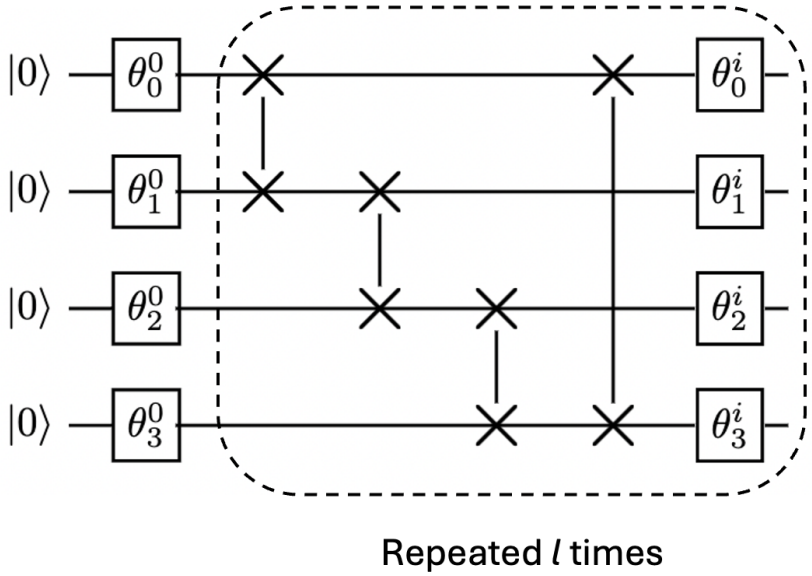


Figure 4.5: Circuit diagram of the parameterized block $W(\theta)$, comprising a series of single-qubit rotation gates θ_n^i and entangling gate layer comprising of a sequence of controlled-phase (CZ) gates. The gates inside the dotted block are repeated l times. Adapted from [38].

to estimate the expectation value of the circuit function, as given in Equation 4.32. The measurement outcomes are mapped to corresponding labels $\{+1, -1\}$ according to the boolean mapping given in Equation 4.36.

After obtaining the optimal parameter values, the circuit is subsequently executed with a fixed set of optimal parameters to determine the classification labels for unseen data $s \in \mathcal{S}$.

4.4 Quantum Tensor Networks

Tensor networks are mathematical structures that can approximate intractable high-dimensional vectors as a contracted sequence of low-dimensional tensors. Stoudenmire et al. [111] carefully explain the approach to using tensor networks, such as MPS, in supervised classical machine learning to approximate a higher-order feature map $\Phi(x)$ applied on data $x = (x_1, x_2, \ldots, x_N)$ and $x \in \mathbb{R}^n$.

$$f(x) = w \cdot \Phi(x) \tag{4.41}$$

where w is the weight vector. The MPS tensor network captures one-dimensional correlations in an order-N (d^N-dimensional) tensor by decomposing it into a chain of

N lower-order tensors. This high-order feature map $\Phi(x)$ can be represented as the product of lower-order local feature maps as,

$$\Phi_{s_j}(x) = \bigotimes_{j=0}^{N-1} \phi_{s_j}(x_j) = \phi_{s_0}(x_0) \otimes \phi_{s_2}(x_2) \otimes ... \otimes \phi_{s_{N-1}}(x_{N-1}) \tag{4.42}$$

Where $\phi_{s_j}(x_j)$ is a local feature map applied to a single feature x_j of the N-dimensional data vector x. s_j represents the indices of the local tensors, ranging from $[1, d]$ where d is the local dimension.

For example, a local map $\phi_{s_j}(x_j)$ can be chosen such that,

$$\phi_{s_j}(x_j) = \left[\cos\left(\frac{\pi}{2}x_j\right), \sin\left(\frac{\pi}{2}x_j\right)\right] \tag{4.43}$$

In a quantum setting, a quantum state can be representative of a 2^n-dimensional tensor in a quantum state space $\mathcal{F} \subseteq \mathcal{H}_2$, and can be represented by the tensor product of n individual qubits [38]. Consequently, the local feature maps $\phi_{s_j}(x_j)$ represent any data embedding strategy applied to qubits as a sequence of single or multiple qubit gates. Such a feature map can be represented as,

$$\Phi : x \mapsto |\Phi(x)\rangle\langle\Phi(x)| = \bigotimes_{i=1}^{n} |\phi_i(x)\rangle\langle\phi_i(x)| \tag{4.44}$$

Such that, Equation 4.42 can be written as [41],

$$\Phi_N(x) = \begin{bmatrix}\cos\left(\frac{\pi}{2}x_0\right) \\ \sin\left(\frac{\pi}{2}x_0\right)\end{bmatrix} \otimes \begin{bmatrix}\cos\left(\frac{\pi}{2}x_2\right) \\ \sin\left(\frac{\pi}{2}x_2\right)\end{bmatrix} \otimes \cdots \otimes \begin{bmatrix}\cos\left(\frac{\pi}{2}x_{N-1}\right) \\ \sin\left(\frac{\pi}{2}x_{N-1}\right)\end{bmatrix} \tag{4.45}$$

The feature map in Equation 4.45 is also known as the product state feature map [38].

4.4.1 Example: Hierarchical Quantum Classifier

Grant et al. [30] proposed a *Hierarchical Quantum Classifier* (HC) model based on the TTN and MERA architectures, that enables supervised machine learning tasks on classical and quantum data. In the classifier circuit, classical data inputs $x \in \chi$ undergo data embedding, while quantum data, assumed to originate from another quantum device, doesn't require encoding. This is followed by a classifier block $T(\theta)$ containing a series of two-qubit unitary gates $U(\theta_i)$ with trainable parameters θ.

For a binary classification task on a dataset $(x^d, y^d)_{d=1}^D$, where $x^d \in \mathbb{R}^N$ are N-dimensional data features and $y^d \in \{0, 1\}$ are the corresponding class labels. The classifier circuit with data embedding $S(x^d)$ and classifier blocks $T(\theta)$ can be represented as,

$$\mathcal{U}(x^d, \theta) = T(\theta)S(x^d) \tag{4.46}$$

The circuit is finally measured with a measurement operator $\mathcal{M}$, such that the circuit function can be expressed as,

$$\begin{aligned} f_\theta(x^d) &= \langle \phi(x^d)|T^\dagger(\theta)\mathcal{M}T(\theta)|\phi(x^d)\rangle \\ &= \langle \Phi(x^d, \theta)|\mathcal{M}|\Phi(x^d, \theta)\rangle \end{aligned} \tag{4.47}$$

where $\Phi(x^d, \theta)$ is the quantum state prepared by the classifier circuit $\mathcal{U}(x^d, \theta)$. Here, $\mathcal{M}$ is a single-qubit measurement operator whose expectation value is a binary output that classifies data inputs into corresponding classes.

Classifier architecture

The classifier block $T(\theta)$ can be modeled to replicate the tree-like structure of the TTN and MERA tensor networks. In a TTN circuit, a set of two-qubit unitary gates is applied to adjacent qubits. These connections allow TTN to capture local quantum correlations, such as entanglement, between a group of tensor nodes [97]. After each layer, one of the qubits from each gate is discarded, or left unmeasured. This aids in downsampling the data inputs to a single qubit output for binary classification purposes. A TTN circuit is depicted in Figure 4.6.

Similar to TTN, a MERA circuit has a tree-like structure with an additional layer of unitary gates $D(\theta)$ that connects the different branches of a tensor network. This allows MERA to capture long-range quantum correlations in tensor nodes [30, 97]. A MERA circuit is shown in Figure 4.7.

Unitary parameterization

The unitary operations U_i and D_i in the HC circuit are arbitrary two-qubit unitary gates, parameterized to include trainable parameters. These unitaries can be either real-valued, containing the R_y rotation gates, or complex-valued, containing a combination of R_y or R_z gates [58]. Utilizing complex-valued unitary operation prevents

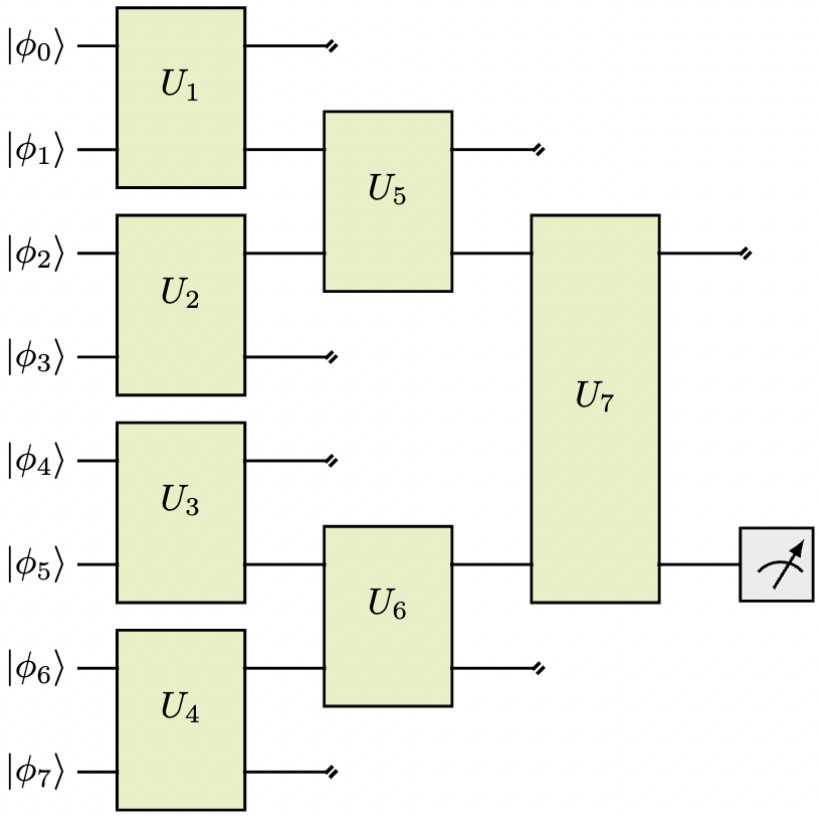


Figure 4.6: Circuit diagram of a classifier block $T(\theta)$ in the HC model, based on the TTN tensor network. The circuit consists of a series of layers of two-qubit unitary gates U_i, applied to adjacent qubits.

the optimizer from getting stuck in local minima during optimization [30].

Further, the two-qubit unitary gates can be decomposed into three alternative parameterizations, as illustrated in Figure 4.8. In a simple parameterization, Figure 4.8(a), a set of two arbitrary single-qubit unitary gates (U_{n_1}, U_{n_2}) and a CNOT gate constitute the two-qubit unitary operation.

The general gate structure, Figure 4.8(b), consists of an arbitrary two-qubit gate that can be decomposed into several combinations of single-qubit rotations and CNOT gates. Some optimal decompositions for real and complex-valued general gate structures are provided by Vatan et al. [114], as illustrated in Figures 4.10 and 4.11. Lastly, the third parameterization utilizes an arbitrary three-qubit gate with an auxiliary qubit, as shown in Figure 4.8(c).

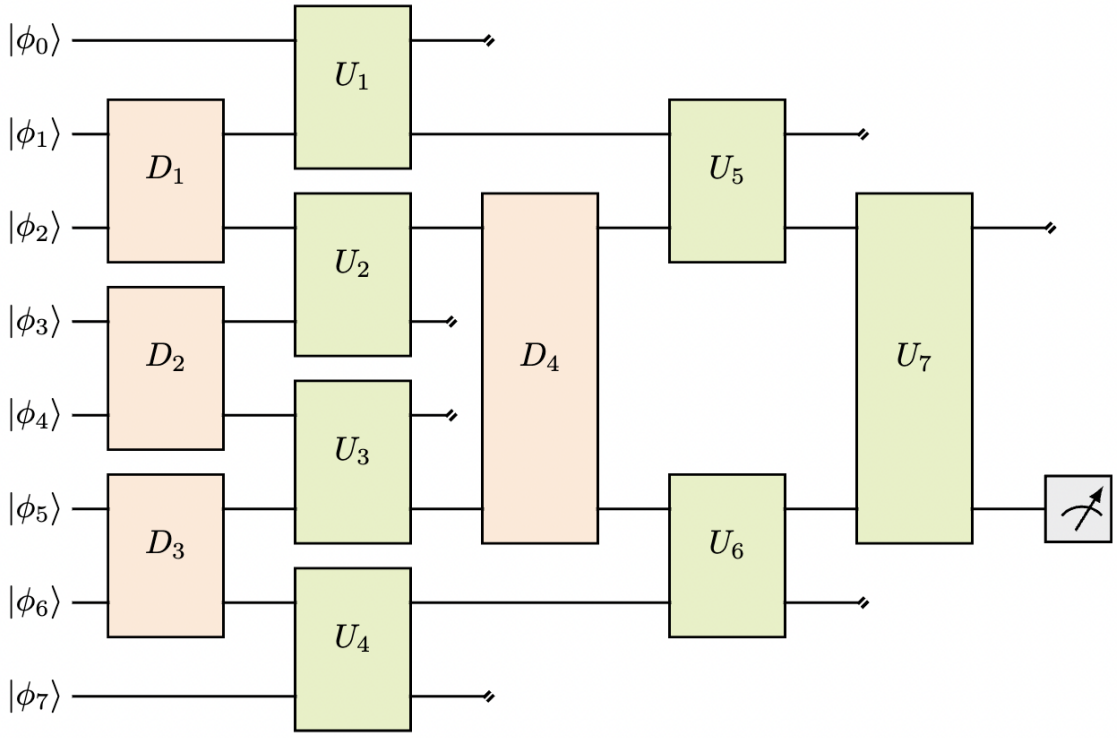


Figure 4.7: Circuit diagram of a classifier block $T(\theta)$ in the HC model, based on the MERA tensor network. The circuit consists of two layers of two-qubit unitary gates, D_i and U_i, applied to adjacent qubits.

4.5 Quantum Neural Networks

4.5.1 Example: Quantum Convolutional Neural Network

Cong et al. [18] introduced the QCNN, a quantum counterpart of the CNN. Akin to a CNN, a QCNN stacks convolutional, pooling, and fully connected layers to extract essential features from the input data. The circuit assumes an input of an N-qubit quantum state ϕ that it classifies into a fixed number of classes. From the sections discussed above, we know that any data embedding block $S(x)$ can encode classical data inputs x to a quantum state $\phi(x)$.

QCNN Layers

The circuit for a convolutional layer consists of a series of unitary operations U_i that are applied to the input features in a translationally invariant manner. Many architectural layouts like TTN and MERA, and unitary parameterizations, such as

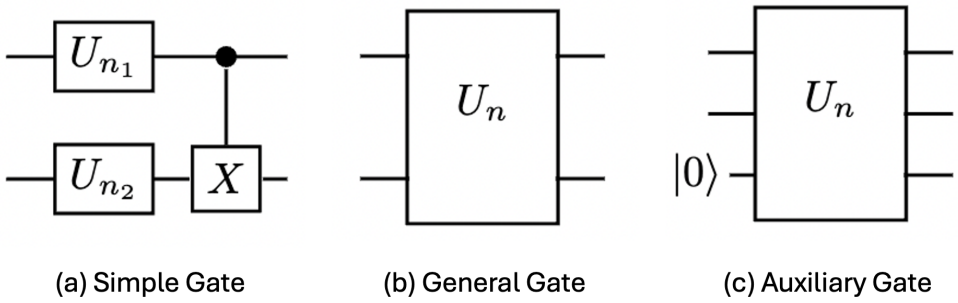


Figure 4.8: Alternative parameterizations for two-qubit unitary operations utilized in tensor network circuits. (a) A set of two arbitrary single-qubit unitary gates followed by a CNOT gate. (b) An arbitrary two-qubit unitary gate that can be decomposed into several optimal gate combinations. (c) An arbitrary three-qubit gate that utilizes an auxiliary qubit. Redrawn from [30].

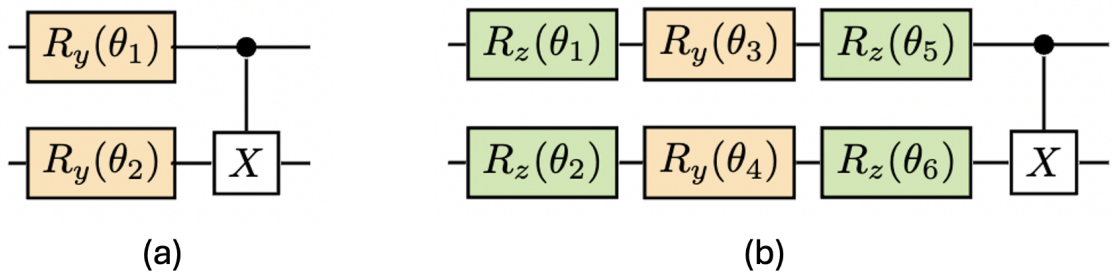


Figure 4.9: Alternative decomposition of a two-qubit unitary operation with a simple gate parameterization. (a) A real-valued decomposition with R_y gates. (b) A complex-valued decomposition with R_y and R_z gates. Redrawn from [58].

simple and general gate parameterizations, were mentioned while discussing the HC model in Subsection 4.4.1 that can be applied to construct the convolutional layer. The trainable parameters in the unitary operations are learned during the training process of the QCNN model.

Subsequently, a fraction of the qubits output from the convolutional layer are measured in the pooling layer. This downsampling can be performed by mid-circuit measurements, following which the results are either discarded or utilized to apply classically conditioned operations V_i on the remaining qubits [77]. In cases where mid-circuit measurements cannot be performed, the operation can be replaced by controlled unitary gates and measurement can be deferred until the end of the QCNN layer sequence. The convolutional and pooling layers are stacked for a finite depth, typically alternatively, until the desired output size is achieved in the model.

Finally, a fully connected layer is applied to the remaining qubits. This layer is

constructed with the controlled-phase gates, followed by a measurement operator in a Pauli-α basis.

The QCNN model requires a total of $O(\log N)$ variational parameters to classify an N-qubit input state, while the hyperparameters like the number of convolutional or pooling layers and their depth remain fixed. A sample QCNN model is shown in Figure 4.12.

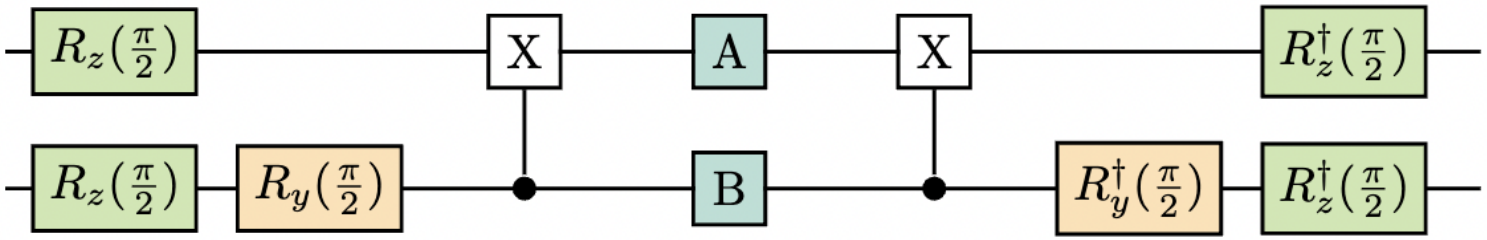


Figure 4.10: A real-valued decomposition of an arbitrary two-qubit unitary gate with at most 12 single-qubit and 2 CNOT gates. In the circuit, $A, B = R_z(\alpha)R_y(\theta)R_z(\beta)$. Redrawn from [114].

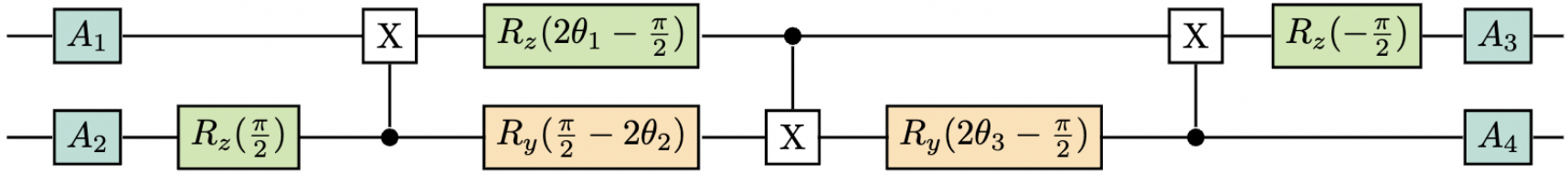


Figure 4.11: A complex-valued decomposition of an arbitrary two-qubit unitary gate with at most 15 single-qubit and 3 CNOT gates. In the circuit, $A_i = R_z(\alpha)R_y(\theta)R_z(\beta)$. Redrawn from [114].

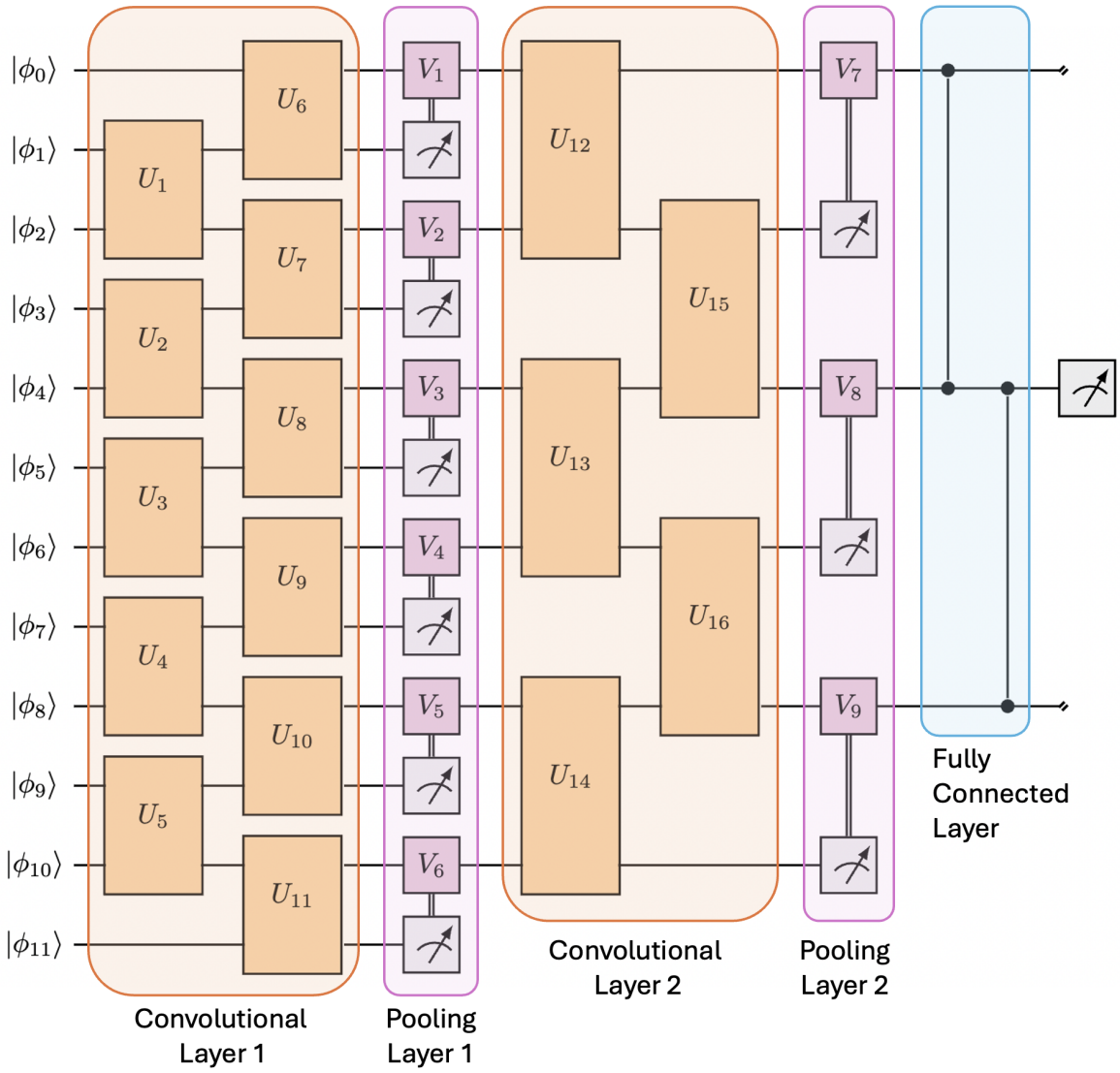


Figure 4.12: Circuit diagram for a QCNN model with stacked convolutional, pooling, and fully connected layers. The convolutional layer mirrors the structure of a MERA tensor network with unitary operations U_i. While the pooling layer performs mid-circuit measurements to apply classically conditioned unitary gates V_i on remaining qubits. The fully-connected layer is applied as a series of controlled-phase gates on the remaining qubits. Adapted from [18].

Chapter 5

Introducing *piQture*

QML is emerging as a viable technique for experimenting with machine learning to tackle complex scientific and engineering problems. Thus, introducing the integration of QML models into machine learning pipelines for real-life applications. While standalone programs exist to demonstrate the performance of QML models for image processing, there is a notable absence of a well-defined model workflow [13, 127]. The lack of such a workflow adds complexity to integrating quantum sub-routines into the classical landscape of image processing. With the current quantum hardware, it is crucial to emphasize the need for accessible and user-friendly tools.

Furthermore, employing QML models in real-world image processing applications presents challenges requiring a nuanced understanding of domain-specific knowledge. This requirement is a barrier for individuals looking to experiment and implement QML algorithms without significant expertise [13].

In this chapter, we introduce *piQture*, a Python and Qiskit-based [44] QML library tailored for image processing applications. *piQture* is designed to streamline QML workflows to accommodate users familiar with classical machine learning but without prior experience in QML. The overarching design of the library provides a user-friendly entry point into QML, fostering an environment instrumental for effective collaboration among researchers, developers, and students. This chapter provides an overview of the design and structure of the *piQture* library.

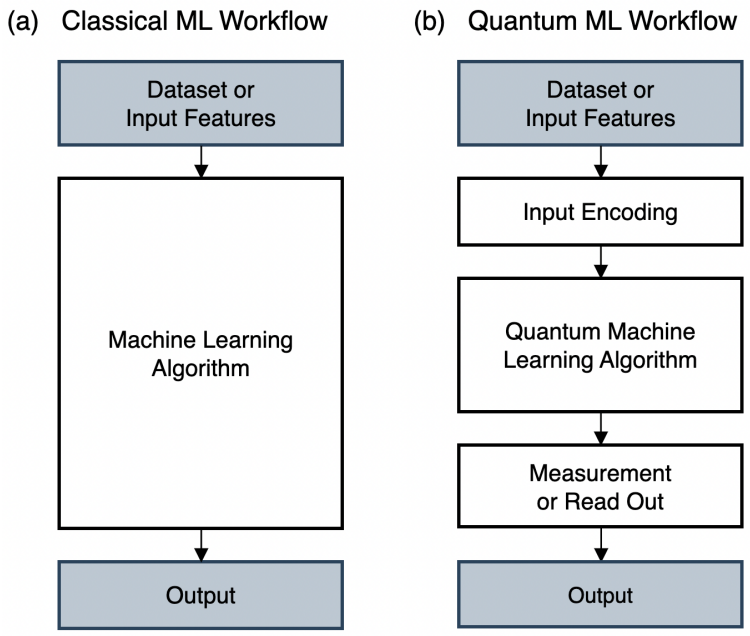


Figure 5.1: Layout of (a) classical machine learning and (b) quantum machine learning workflows. Redrawn from [104].

5.1 Overview

Classical Machine Learning Workflows

A machine learning workflow is a structured framework that lays out a definitive set of rules governing data transition through distinct stages. Each stage contributes to the development and refinement of a predictive model. As illustrated in Figure 5.1(a), a classical machine learning workflow involves utilizing input data features to train any machine learning algorithm, ultimately generating trained models that can be saved and employed for making accurate predictions on previously unseen data. Contrary to its linear depiction, a machine learning workflow is more complex, comprising multiple interconnected steps iteratively triggered for model training and evaluation. These steps may include data processing, feature selection, model training, hyperparameter tuning, and model evaluation. Overall, a machine learning workflow is dynamic and involves multiple feedback loops across its stages to refine and improve model performance throughout its training period.

Various automation tools have been developed and widely used to navigate these iterative steps inherent in a machine learning workflow. Examples include TensorFlow

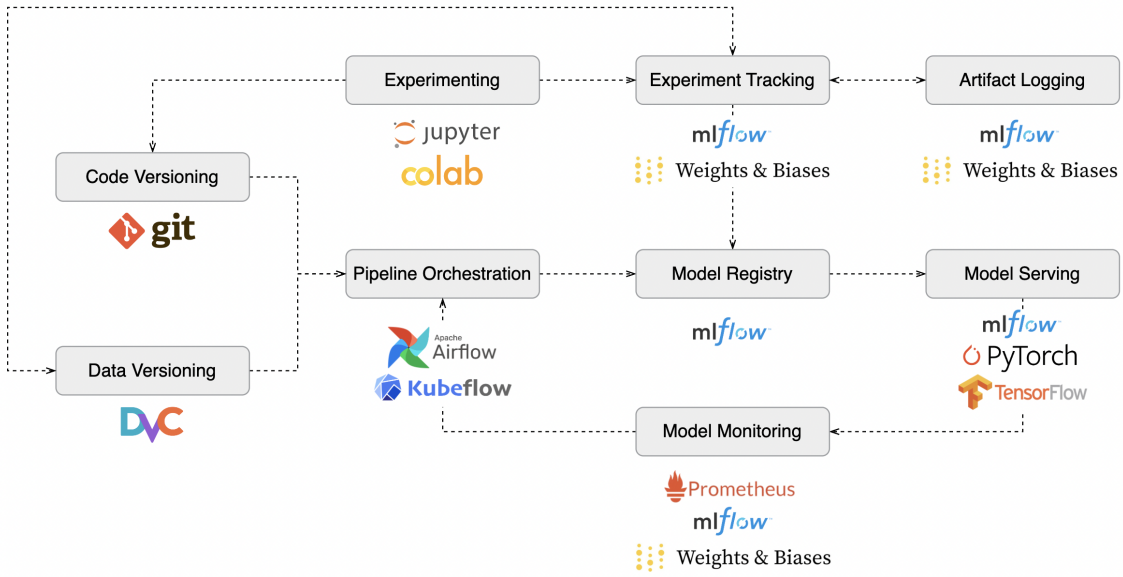


Figure 5.2: Structure of a classical machine learning workflow, highlighting the various tools involved in pipeline automation and model monitoring tasks.

Extended (TFX)[1] and GitHub Actions[2] for workflow automation, Apache Airflow[3] and Kubeflow[4] for pipeline orchestration, MLflow[5] [134] and Weights and Biases[6] for model packaging, registry, and deployment, and Data Version Control (DVC)[7] for data versioning, and Git for version control. However, these examples are not exhaustive, as additional tools are utilized in machine learning workflows. Figure 5.2 provides an overview of some of these tools and their functionalities. These tools streamline various stages in a machine learning pipeline, mitigating the need for manual intervention and reducing effort across multiple stages.

Quantum Machine Learning Workflows

A QML workflow is a hybrid quantum-classical workflow, as illustrated in Figure 5.1(b), that employs a variational methodology [80]. This methodology splits any QML algorithm into three core stages—(1) quantum state preparation from classical data inputs, (2) expectation value determination through measurement, and (3) optimization

[1]https://www.tensorflow.org/tfx
[2]https://github.com/features/actions
[3]https://airflow.apache.org/
[4]https://www.kubeflow.org/
[5]https://mlflow.org/
[6]https://wandb.ai/site
[7]https://dvc.org/

of variational parameters with a classical optimizer—iteratively run to perform convergence of a quantum circuit function towards the desired outcome. These stages are executed on quantum and classical computational platforms, resulting in a more complex QML workflow requiring several classical pre- and post-processing sub-routines.

Managing the complexity arising from heterogeneous computational routines and platforms presents unique challenges in orchestrating QML workflows while maintaining data portability between different platforms. Consequently, this requires an individual to understand the algorithmic requirements and the interplay between heterogeneous computational platforms [13, 64, 127], limiting their ability to transition, learn, or experiment with QML algorithms in real-world applications [26]. Hence, defining optimal workflows for QML becomes an essential and critical skill as the field advances.

5.2 Pipeline Design

To address the requirement for QML workflows, we present *piQture*—an open-source Python and Qiskit-based library that streamlines the development, execution, and training of QML models tailored for image processing tasks. Within the scope of this thesis, *piQture* primarily focuses on classifying image data that includes attributes such as image size and color values. Within its development environment, *piQture* consists of quantum state preparation, model selection and training, parameter tuning, and evaluation processes.

5.2.1 Workflow Description

The orchestrated workflow of *piQture* progresses sequentially through the stages of Data Preprocessing, Quantum Circuit Preparation, Quantum Circuit Processing, and Measurement, as illustrated in Figure 5.3. In the workflow, the processes constantly transition between classical and quantum processing platforms. Additionally, these stages exhibit significant distinctions like the type of input data received and the outputs produced. Figure 5.4 depicts the flow of data, detailing the inputs accepted and outputs generated at each stage throughout the workflow.

The workflow begins with the **Data Processing** stage, where raw input data undergoes cleaning and preparation. The primary objective of this stage is to ensure that the raw input data is suitably prepared for the subsequent stages by addressing

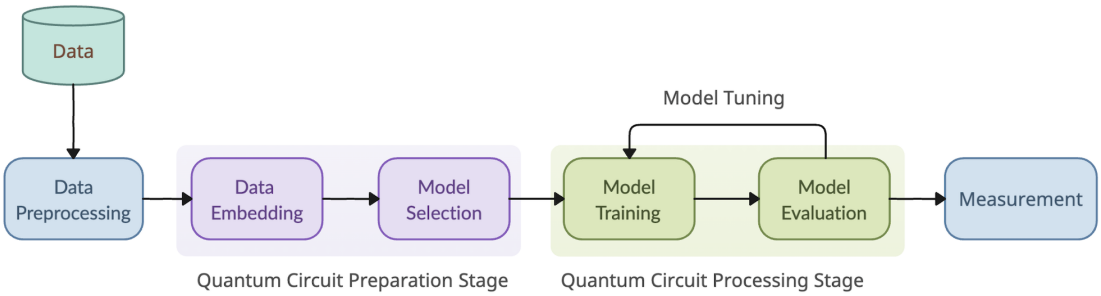


Figure 5.3: Workflow for *piQture*, consisting of four main stages: Data Preprocessing, Quantum Circuit Preparation, Quantum Circuit Processing, and Measurement. The Quantum Circuit Preparation stage contains the Data Embedding and the Model Selection sub-stages. The Quantum Circuit Processing contains the Model Training and the Model Evaluation & Training sub-stages.

various challenges in refining and enhancing its quality. The process includes cleaning noisy data, transforming data into a suitable format for analysis, reducing dimensionality to focus on essential features, and normalizing data to ensure consistency across different scales. Furthermore, data preprocessing may include techniques for handling incomplete data, missing attributes, and detecting and handling outliers.

After the Data Preprocessing stage, the processed classical data transitions to the **Quantum Circuit Preparation** stage. Here, the focus shifts towards constructing a quantum circuit for a QML model. This stage comprises two sub-stages: Data Embedding and Model Selection, both of which are classical sub-routines. The resulting output of this stage is a quantum circuit with data embedding and parameterized routines.

The **Data Embedding** sub-stage facilitates the representation of the preprocessed classical data on quantum devices, establishing an interface between classical and quantum platforms. With the help of data encoding techniques and feature maps discussed in Chapters 3 and 4, this sub-stage maps data to a quantum state space, effectively embedding classical information into a quantum state for subsequent use. Since this stage involves selecting an appropriate data embedding technique, only, it is categorized as a classical sub-routine.

The **Model Selection** sub-stage facilitates the creation of a desired QML model as a quantum circuit. During this stage, users can select from an array of QML models, similar to those discussed in Chapter 4, each offering a unique approach to solving an image processing task. While the *piQture* library has default options for constructing a basic QML model structure, users can make informed decisions

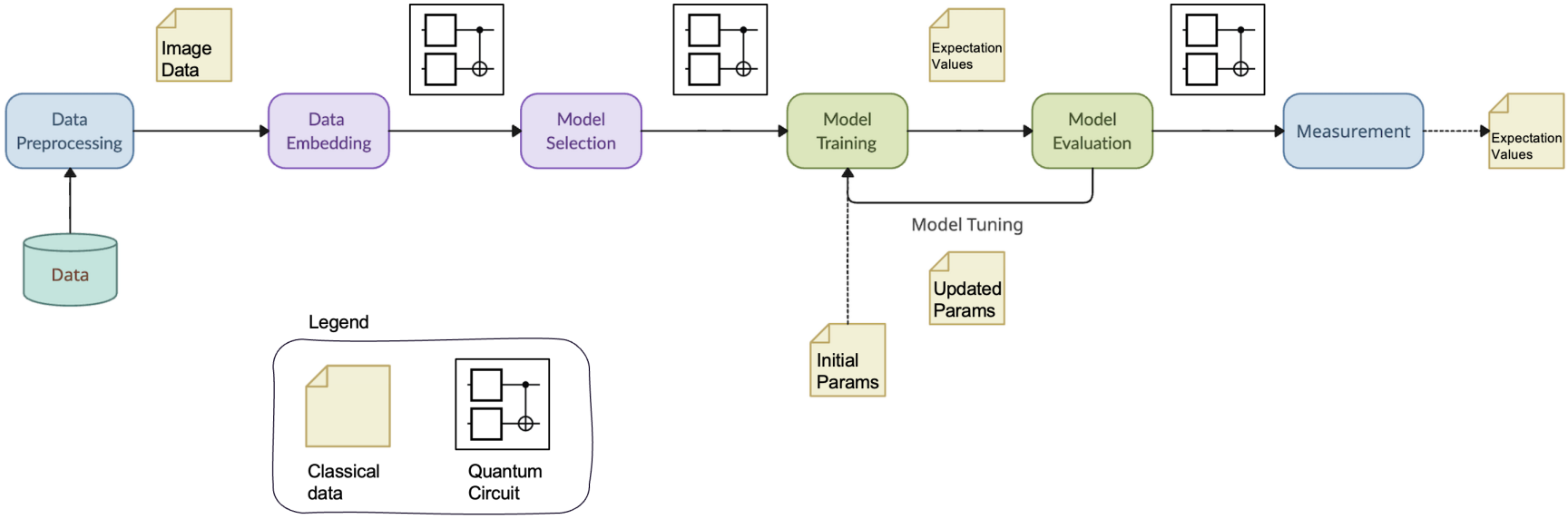


Figure 5.4: The input and output components at each stage of the *piQture* library, highlighting the flow of data through the workflow. Adapted from [127].

and tailor the architecture to their needs and preferences. As the name suggests, the Model Selection sub-stage deals with choosing an appropriate QML model only. Hence this stage is categorized as a classical sub-routine in the workflow.

Once the QML circuit is prepared, the workflow transitions to the **Quantum Circuit Processing** stage, which facilitates the training and evaluation of the constructed circuits. This stage consists of two sub-stages: Model Training and Model Evaluation & Tuning. QML models, such as those implemented as kernel methods or variational algorithms as discussed in Chapter 4, consist of an ansatz with trainable parameters. During this stage, an iterative loop between the two sub-stages optimizes the trainable parameters utilizing an optimizer and a loss function. The resulting output of this stage is a trained QML model with optimal parameter values.

In the **Model Training** sub-stage, the prepared QML circuit from the Quantum Circuit Preparation stage is executed on a backend. This sub-stage mimics the forward and backward passes of a classical machine learning model, which in a quantum setting, refers to the execution of the circuit with a fixed set of parameters. Initially, a QML circuit is run with a random set of parameter values. While subsequent calls for this sub-stage circuit utilize the updated parameters obtained from the Model Evaluation & Tuning stage. The forward and backward passes directly schedule circuit execution jobs on a quantum backend. Hence this sub-stage is categorized as a quantum sub-routine in the *piQture* workflow.

The **Model Evaluation** & **Tuning** sub-stage facilitates the optimization of trainable parameters in the QML circuit by triggering calls to a classical optimizer function. The expectation values obtained from the Model Training sub-stage, with the help of a predefined loss function, contribute to shaping a cost function that evaluates the performance of the quantum circuit function across different sets of parameters. The optimizer aims to minimize the cost function by adjusting the parameter values in the quantum circuit. Such an iterative optimization process enhances the performance of the quantum machine learning model. This sub-stage may occasionally trigger circuit execution on a backend, thereby categorizing it as a quantum sub-routine.

The iterative loop between the Model Training and the Model Evaluation & Tuning sub-stages continues until the circuit function converges.

Finally, in the **Measurement** stage, the trained QML model obtained from the Quantum Circuit Processing stage is executed on a backend with a fixed set of optimal parameters. This trained model can also be executed on new and unseen data, resulting in corresponding expectation values or quasi-probability distributions.

Table 5.1: Mapping of various stages in the *piQture* workflow to their corresponding modules in the *piQture* library.

Stage	Sub-stage	Module
Data Preprocessing		`data_loader`
Quantum Circuit Preparation	Data Embedding	`data_encoder` `image_representations`
	Model Selection	`tensor_network_circuits` `neural_networks`
Quantum Circuit Processing		Relies on `algorithms` and `neural_networks` modules in the *Qiskit Machine Learning* library
Other Supporting Modules		`gates`, `mixin`

5.3 Pipeline Structure

The mapping of stages in the *piQture* workflow to their corresponding modules in the library is presented in Table 5.1.

5.3.1 Data Preprocessing

The Data Preprocessing stage operates to refine raw data for compatibility with the subsequent Data Embedding sub-stage. This stage may perform cleaning, transformation, dimensionality reduction, or normalization tasks, ensuring the classical image inputs are suitably prepared for upcoming stages.

A classical image dataset must include attributes such as:

image size A two-dimensional tuple `(int, int)` expressing the width and height of the images in the dataset.

pixel values Multi-dimensional arrays describing the color intensity of each pixel in the image, with distinct arrays for RGB channels.

The *piQture* library provides a utility function called `load_mnist_dataset`, designed to import an MNIST [60] dataset using PyTorch's data primitives, `Dataset` and `DataLoader`. This function wraps processes such as batching, resizing, and normalization for the MNIST dataset, which serves as the default dataset in this thesis.

5.3.2 Quantum Circuit Preparation Stage

The Quantum Circuit Preparation Stage comprises two sub-stages: Data Embedding and Model Selection. These sub-stages facilitate the construction of a quantum circuit

corresponding to a QML model.

Data Embedding

Within *piQture*, the Data Embedding module encompasses the `ImageEncoding` and `ImageMixin` classes, offering abstractions for encoding classical image attributes, such as pixel position and color values, onto a quantum state. The data encoding techniques like Angle or Amplitude Encoding, and QIR methods, as detailed in Chapter 3, inherit from these base classes to outline the circuit preparation process for images of varied dimensions and color schemes. The class diagram for this module is depicted in Figure 5.5. This circuit generated by the `data_encoder` module subsequently serves as an input for the Model Selection sub-stage.

Model Selection

The Model Selection sub-stage provides the user with a factory of QML model implementations which include various tensor and neural network architectures.

Tensor Network Circuits: The `tensor_network_circuits` module within *piQture* provides implementations for tensor network structures such as MPS, TTN, and MERA. A `BaseTensorNetwork` class provides abstractions for laying out the structure of a tensor network circuit, inherited by the `MPS, TTN,` and `MERA` classes. The underlying architecture of the tensor networks consists of a series of two-qubit unitary gates, as discussed in Chapter 4, built with arbitrary unitary parameterizations [30]. A `TwoQubitUnitary` class within the `gates` module of *piQture* lays out these gate parameterizations that are utilized by the individual tensor network classes. All the tensor network circuits follow the same functional structure of building unitary gate layers with simple or general and real or complex parameterizations of the two-qubit unitary blocks. Within each `MPS`, `TTN`, and `MERA` class, a `backbone` function further defines the structure of the respective tensor network. Figure 5.6 presents a class diagram for the `tensor_network_circuits` module.

Quantum Neural Networks: *piQture* offers implementation for QCNN, including quantum layer implementations, such as the convolutional and quanvolutional layers, within the `neural_networks` module.

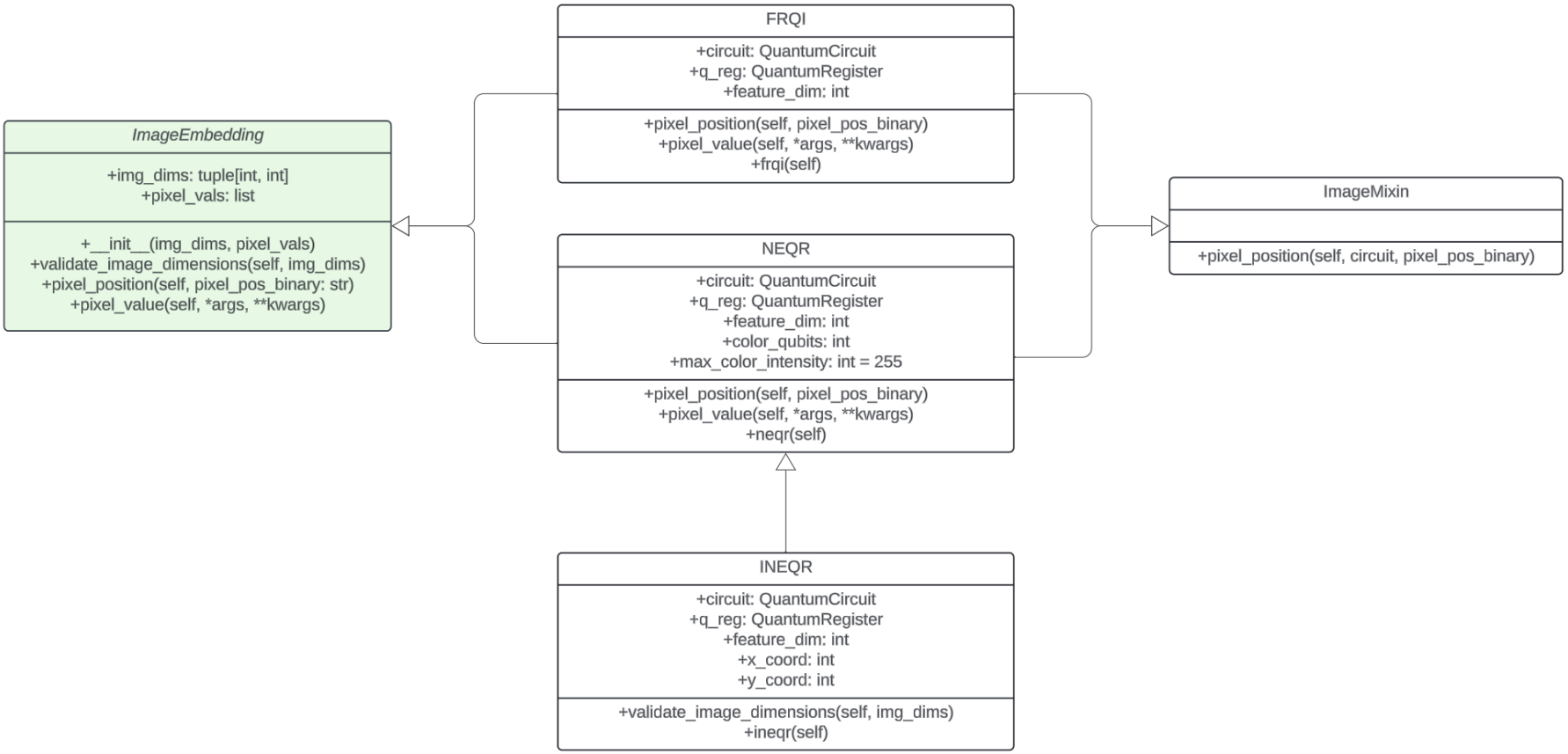


Figure 5.5: Class diagram illustrating the layout of the `data_encoder` module in the *piQture* library. The diagram demonstrates the inheritance relationship between the `ImageEmbedding` base class and various QIR classes, such as `FRQI, NEQR,` and `INEQR`, as well as their composition with the `ImageMixin` class.

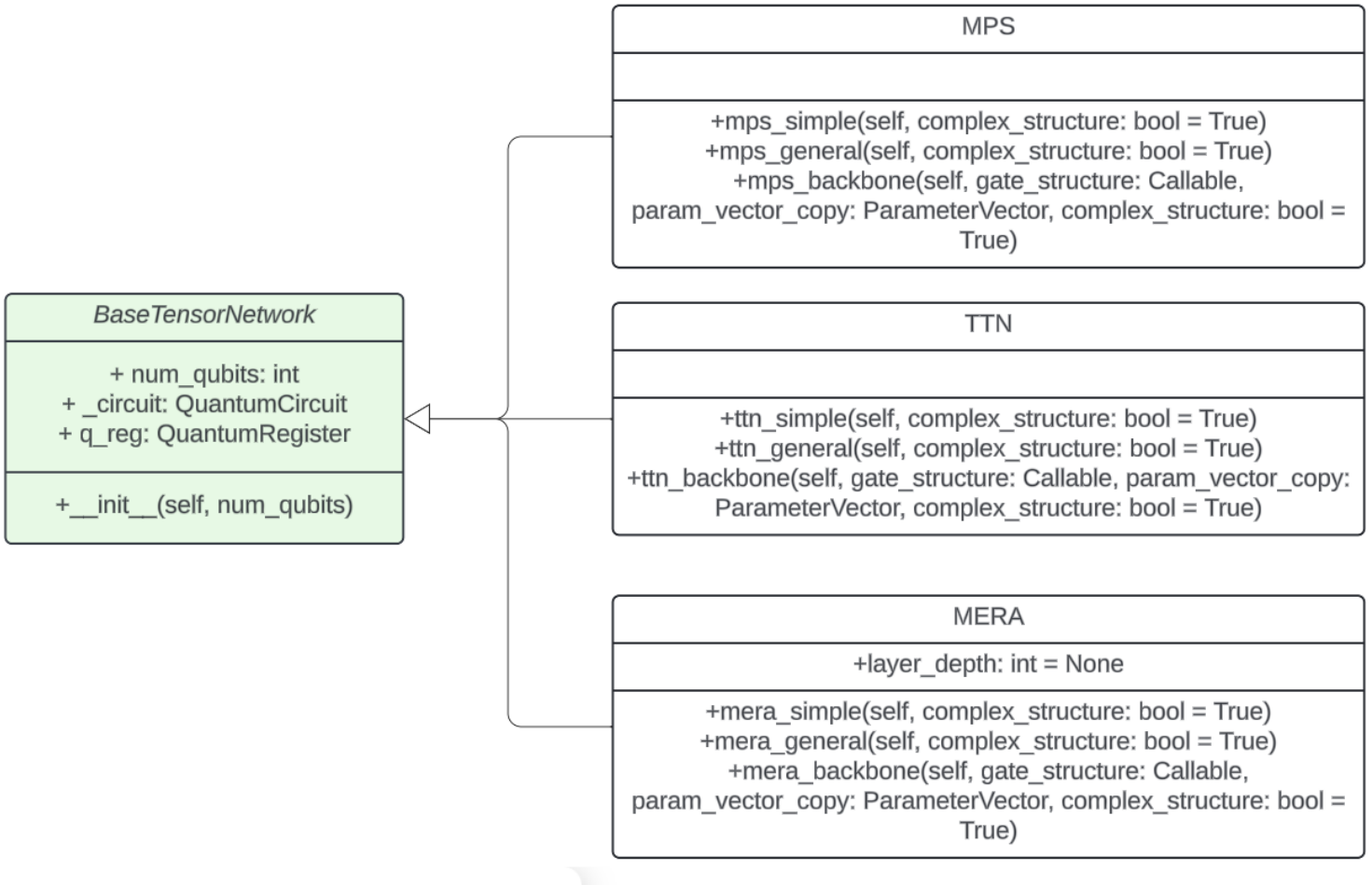


Figure 5.6: A class diagram of the `tensor_network_circuits` module within *piQture*. The diagram demonstrates the inheritance relationship between the various tensor network classes and the `BaseTensorNetwork` abstract class.

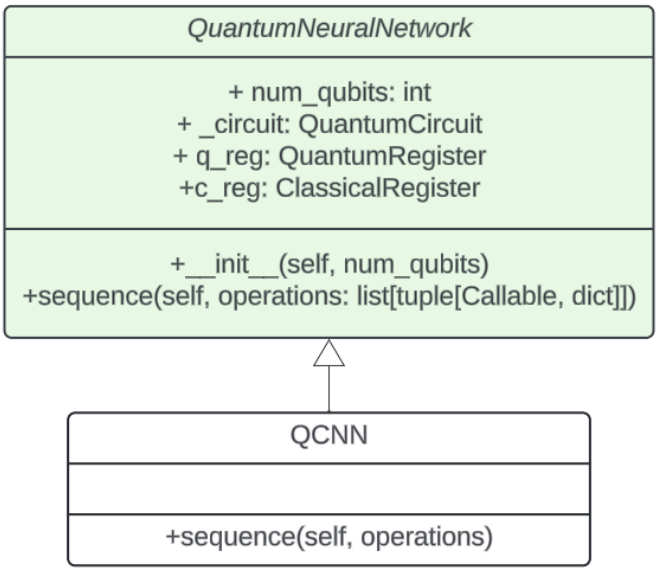


Figure 5.7: A class diagram of the `neural_networks` module within *piQture*. The diagram illustrates the inheritance relationship between the `QCNN` class and the `QuantumNeuralNetwork` base class. The `sequence` method in the `QuantumNeuralNetwork` class enables the stacking of various quantum layers in a QCNN circuit.

A QCNN circuit is constructed by sequentially stacking the Convolutional, Pooling, and Fully Connected layers. Within *piQture* implementations for these layers are provided through classes such as `QuantumConvolutionalLayer`, `QuantumPoolingLayer2`, `QuantumPoolingLayer3`, and `QuantumConvolutionalLayer`, all inheriting from the abstract class `BaseLayer`. This inheritance relationship is depicted in Figure 5.8. These classes instantiate quantum circuits for a specified number of qubits while accounting for unmeasured qubits as the layers are stacked. This tracking of unmeasured qubits is essential especially when the preceding layers perform mid-circuit measurements, as in the case of `QuantumPoolingLayer2` and `QuantumPoolingLayer3`, reducing the number of available qubits for the subsequent layers.

Quantum Layers: The `QuantumConvolutionalLayer` class lays out a sequence of unitary gates, dictated by the structure of a MERA tensor network, as discussed in Chapter 4. Consequently, this class requires relevant arguments to call the `MERA` class internally—a `layer_depth` integer to specify the desired depth of the convolutional layer, a `mera_instance` integer that maps the convolutional layer to a particular instance of the MERA tensor network, simple or generalized, and a `complex_structure`

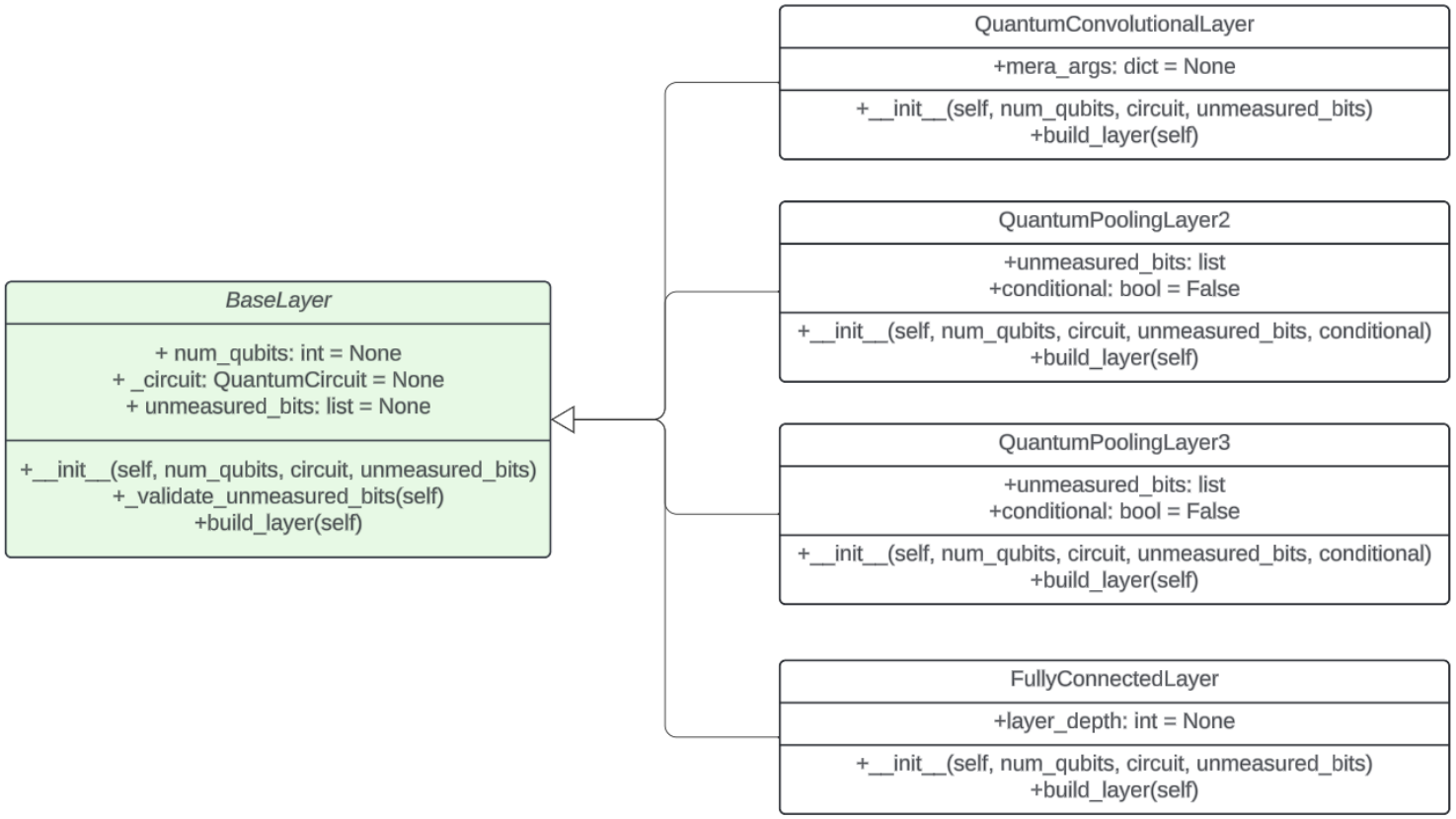


Figure 5.8: UML class diagram of the `layers` module in *piQture*. The diagram demonstrates the inheritance between various quantum layer structures and the `BaseLayer` class.

boolean to specify real or complex the unitary gate parameterizations, thereby introducing an added layer of sophistication to a quantum convolutional layer.

The pooling layer, on the other hand, incorporates mid-circuit measurements in the quantum circuit. Multiple implementations of this layer are available in *piQture*. In the `QuantumPoolingLayer2` class, one of the qubits in a pair of adjacent qubits is measured, whereas, the `QuantumPoolingLayer3` class performs measurements on two from a set of three adjacent qubits. Measurements in both the layers are accompanied with classically conditioned phase gates to the unmeasured qubits, as detailed in Chapter 4. This implementation approach contributes to the adaptability and the expressiveness of the QCNN model.

The `FullyConnectedLayer` class connects the remaining unmeasured qubits with controlled-phase gates and performs final measurements in a Pauli-basis.

5.3.3 Quantum Circuit Processing Stage

The Quantum Circuit Processing Stage encompasses training, evaluating, and tuning the prepared QML circuit. This stage consists of two sub-stages: Model Training and Model Evaluation & Tuning, which iteratively optimize the trainable parameters in the circuit. QML circuits with different parameter values are executed on a simulator or a quantum device in each iteration, generating some expectation values.

Model Training

The Model Training sub-stage takes the prepared QML circuit from the Quantum Circuit Preparation stage and initiates an iterative training process. In the context of a supervised machine learning procedure, the trainable parameters in the circuit are optimized to minimize the cost associated with the cost function of the QML model. Currently, *piQture* depends on the `SamplerQNN` and `EstimatorQNN` classes, present inside the `neural_networks` module in the *Qiskit Machine Learning*[8] library.

During a training step, either a `SamplerQNN` or `EstimatorQNN` provides a forward and backward pass method that triggers the execution of the QML circuit with a specified set of input data and parameters on a backend. The output from a forward pass is generated in the form of quasi-probability distributions or expectation values that are interpreted as the predictive outcome of the QML model, with respect to the provided set of parameter values. A backward pass executes the same quantum

[8]https://github.com/qiskit-community/qiskit-machine-learning

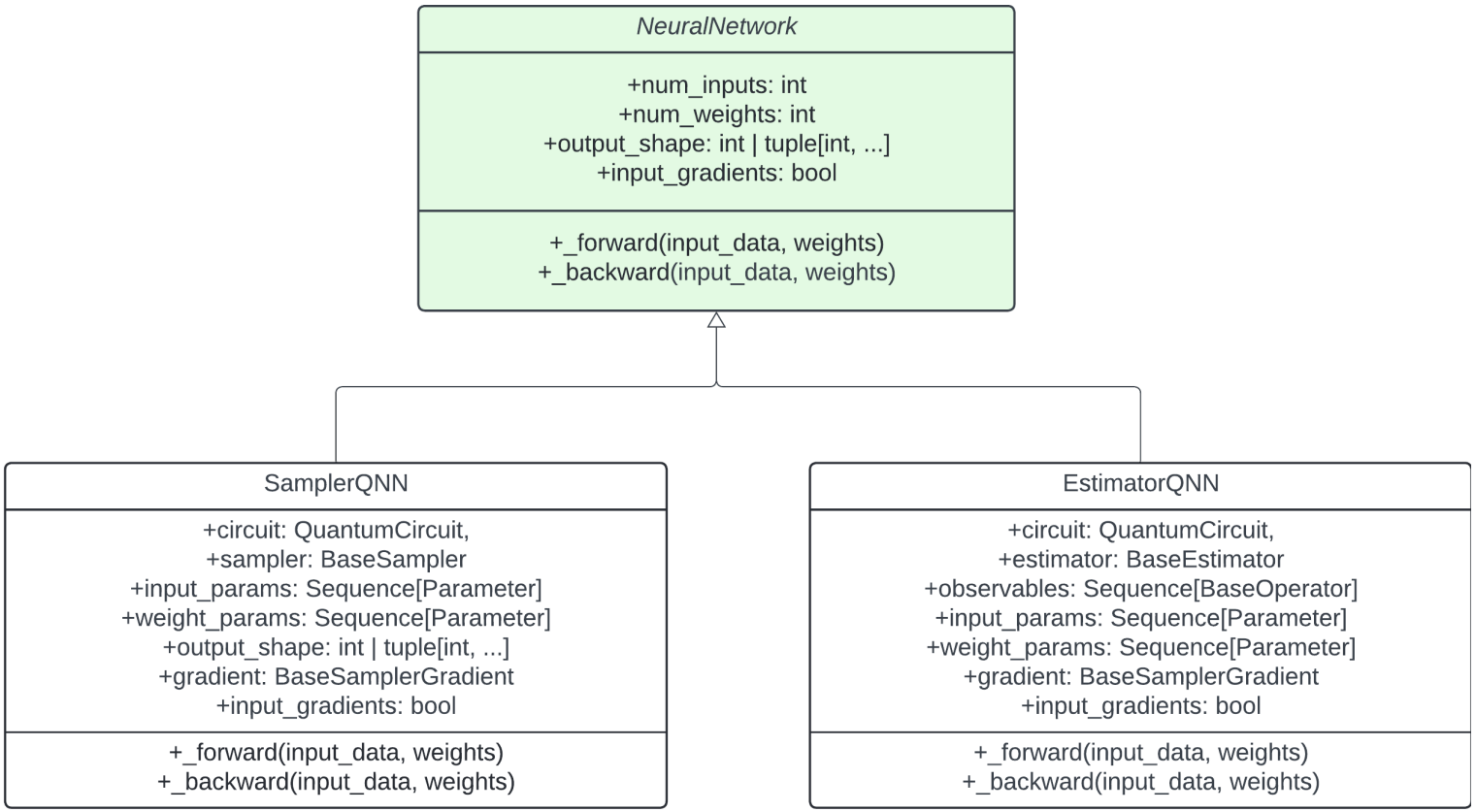


Figure 5.9: UML class diagram of the `neural_netwoorks` module in the *Qiskit Machine Learning* library [44]. The diagram demonstrates the inheritance between the `SamplerQNN`, `EstmatorQNN`, and the `NeuralNetworks` class.

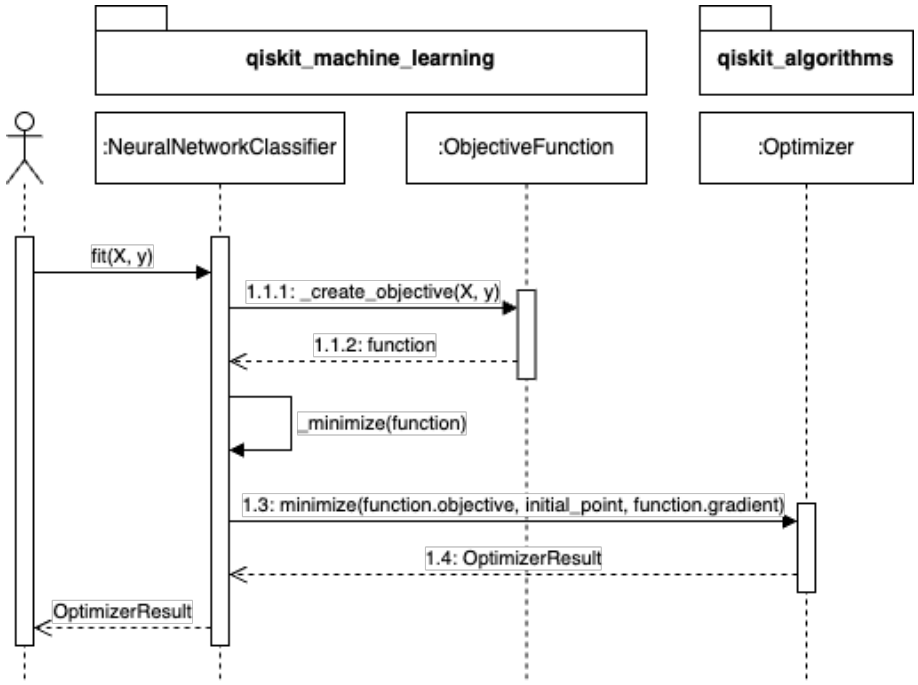


Figure 5.10: Sequence diagram for evaluation and tuning sub-stage performed by the `NeuralNetworkClassifier` class from the *Qiskit Machine Learning* library.

circuit to compute the gradient values of the cost function that are pivotal in implementing optimizer functions in the subsequent Model Evaluation & Tuning sub-stage. Figure 5.9 shows the structure of the `neural_networks` module and the `SamplerQNN` and `EstimatorQNN` classes, within the *Qiskit Machine Learning* library.

While currently, the Model Training stage depends on the *Qiskit Machine Learning* library, proposals for suitable changes to the implementation of the `SamplerQNN` and `EstimatorQNN` classes are outlined in Chapter 8.

Model Evaluation & Tuning

In the Model Evaluation & Tuning sub-stage, a predefined classical optimizer and loss function update the values of trainable parameters in the QML circuit. The expectation values obtained from the preceding Model Training sub-stage and the loss function shape the cost landscape of the circuit function while the optimizer aims to achieve a minimum. In cases where a gradient-based optimizer is defined, the cost function gradients obtained from a backward pass of the circuit are utilized by the optimizer function to minimize the overall cost of the circuit function.

This optimization process runs iteratively between the Model Training and Model

Evaluation & Tuning stages for a specified number of iterations. A callback function may be introduced to halt the training upon reaching a particular cost threshold. The outcome of the Model Evaluation & Tuning sub-stage is a set of optimized parameter values that are either fed back into the Model Training sub-stage for further optimization or utilized by the subsequent Measurement stage to produce the final trained QML model.

Currently, *piQture* utilizes the optimizer functions defined in the *Qiskit Algorithms*[9] library. A `NeuralNetworkClassifier` wrapper class from the *Qiskit Machine Learning*[10] library is called to trigger the iterative loop between Model Training and Model Evaluation & Tuning sub-stages. This evaluation and tuning loop is depicted in a sequence diagram in Figure 5.10.

5.3.4 Measurement

In the Measurement stage, the optimal parameter values received from the Model Evaluation & Tuning stage are utilized to execute a final forward pass of the QML circuit, producing the conclusive results of the entire QML model. The results obtained from this stage are either quasi-probability distributions or expectation values, depending on the `SamplerQNN` and `EstimatorQNN` class being called.

Further, the trained model can be executed on unseen datasets to make predictions or may be stored in a model registry for easy retrieval or future analysis. This process of model management and storage is discussed in the next chapter.

[9]https://github.com/qiskit-community/qiskit-algorithms
[10]https://github.com/qiskit-community/qiskit-machine-learning

Chapter 6

Advancing *piQture*: Strategies for CI/CD

In Chapter 5, we introduced the *piQture* library for building and training QML models for image processing tasks. The user-friendly design of the library aims to enhance accessibility and ease of use for researchers, developers, and students. Nevertheless, ensuring the reliability and robustness of *piQture* demands rigorous testing procedures. This necessity is especially imperative to quantum software development, where the functionality must be accurate across heterogeneous computational platforms.

In this chapter, we discuss the critical strategies for transitioning *piQture* from development to production environments through *Continuous Integration and Delivery* (CI/CD). We explore various practices, from testing methodologies to building and packaging techniques essential for seamless software deployment. To streamline our CI/CD workflow, we leverage tools like GitHub Actions,[1] while MLflow[2] [134] facilitates QML lifecycle management. Together, these tools enable us to implement *Machine Learning Operations* (MLOps), ensuring a robust deployment workflow for the *piQture* library with suitable testing and workflow automation techniques.

[1] https://github.com/features/actions
[2] https://mlflow.org/

6.1 Overview

Machine Learning Operations

DevOps is a set of practices and principles that aid in continuously integrating and automating software development and related operations [28]. MLOps is an extension of DevOps principles that enables software engineers to automate and monitor machine learning models and workflows [55].

We apply MLOps concepts to automate and streamline the end-to-end process of developing, deploying, and maintaining the *piQture* library, aiding in its seamless transition from development to production environments. This process involves several steps, including continuous source code integration into a shared central repository and automated testing of each change, triggering a build process, and packaging the entire library with version control support [55].

Continuous Integration and Delivery

CI/CD is a software development practice that optimizes the delivery pipeline within the SDLC by streamlining software pipeline building, testing, and deployment. Implementing CI/CD processes facilitates collaboration among developers by minimizing manual errors and accelerating feedback loops. At its core, CI/CD comprises two essential practices: Continuous Integration (CI) and Continuous Delivery (CD).

Continuous Integration (CI): CI accommodates frequent integration of code updates into a shared central repository. Each integration triggers an automated build and test workflow, ensuring seamless merging of new code with the existing codebase [28]. This process aids in identifying bugs early in the development cycle. By encouraging developers to make smaller and more frequent code changes, CI promotes continuous improvement in code quality throughout the development process [49].

Continuous Delivery (CD):	CD extends CI capabilities by automating the packaging process and releasing deployment-ready code to end-users [42]. While CI focuses on integrating code changes and running automated tests, CD takes it further by preparing and packaging code changes that pass the tests in CI for deployment. It emphasizes the packaging and deployment processes, ensuring swift delivery of software updates to end-users while maintaining the quality of the source code [42].

CI/CD processes work together to streamline the release cycle of a software library, enabling teams to promptly address integration issues and bug fixes while delivering high-quality software.

6.2 Building, Testing, and Packaging *piQture*

To facilitate the transition from development to production environments through CI/CD, *piQture* follows a series of systematic steps. These steps include creating a shared central repository that multiple users can use for the integration of code updates or collaboration on development ideas, aided by automatic triggering of build and testing workflows.

The steps for building, testing, and packaging are discussed below.

6.2.1 Setting Up the GitHub Repository

GitHub[3] is a centralized developer platform built upon Git,[4] a widely used distributed version control system. GitHub offers a collaborative environment where developers can host, share, and manage their code repositories. It also offers tools for code review, issue tracking, and project management.

We begin the development cycle of *piQture* by setting up a GitHub repository that helps us store source code, scripting files, documentation, and other project assets. We establish an organized directory structure based on the pipeline design and structure discussed in Chapter 5. This structure is foundational for efficient design and utilization among developers.

[3]https://github.com/
[4]https://git-scm.com/

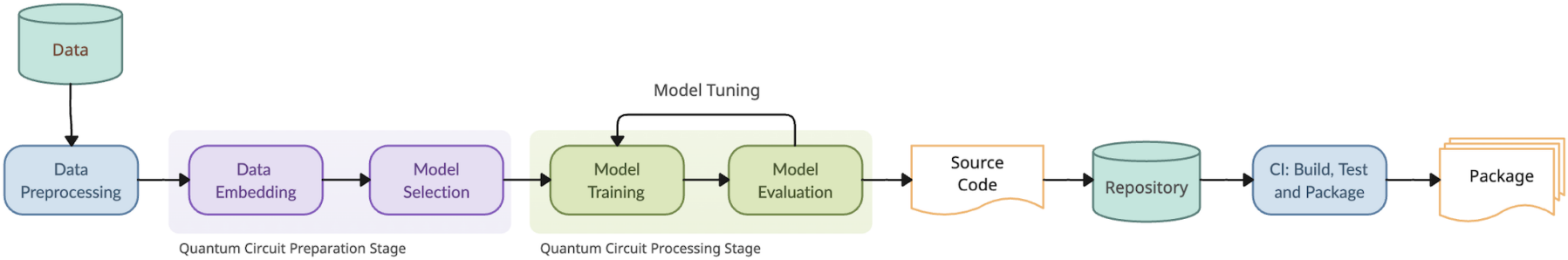


Figure 6.1: The sequential development workflow of *piQture*, starting with the collection of source code into a central repository, followed by initiation of CI builds, and finally packaging *piQture* for release.

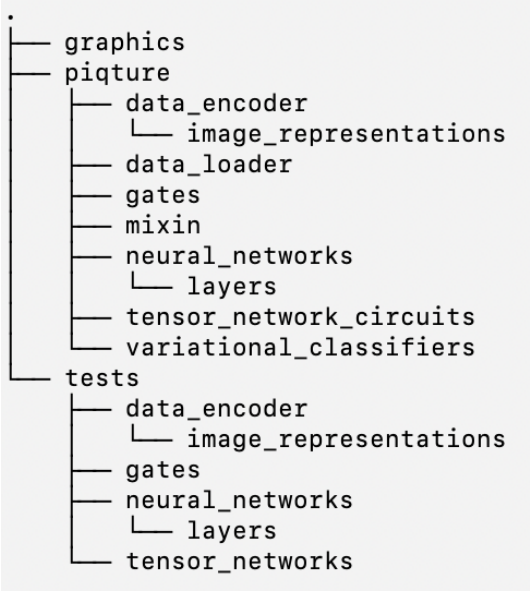


Figure 6.2: The directory structure of *piQture* with three essential directories: `piqture`, `tests`, and `graphics`.

Directory Structure

We structure *piQture* with a well-organized categorization, where each directory contains code implementations essential for constructing a QML workflow.

At the core of the library is the `piqture` directory, containing modules such as `data_encoder`, `data_loader`, `gates`, `mixin`, `neural_networks`, `tensor_network_circuits`, and `variational_classifiers` that provide essential code implementations to load and encode data, and construct a QML model.

Notably, each module in the `piqture` directory is complemented by its counterpart in the `tests` directory. The `tests` directory mirrors the structure of modules in `piqture` to host unit tests tailored to validate their functionalities correspondingly. Such an organization segregates tests from application code, allowing a scalable approach to test management.

On the other hand, the `graphics` directory stores essential graphical assets and resources used for visualization and documentation.

Issues and Pull Requests

One of the key attributes of GitHub is its support for features like Issues and Pull Requests that enable developers to propose changes, review code, and merge contri-

butions in the central repository. The Issues feature helps developers prioritize tasks, report bugs, and submit feature requests. This comprehensive documentation of issues and tracking their scope within the repository ensures that all the team members align with the project priorities and progress. The Pull Requests feature promotes code review and collaboration, ensuring new features meet quality standards before merging into the main codebase.

Additionally, GitHub offers tagging and commit history features that enable developers to annotate significant milestones, releases, and changes within the source code. These milestones allow developers to group issues and pull requests together to prioritize completing critical tasks and manage project timelines. Figure 6.5 shows how we utilize the milestone feature to monitor version releases for *piQture*.

GitHub Projects

Furthermore, GitHub's support for Kanban projects offers a tool for visualizing and managing projects. With the help of GitHub Projects, we created custom boards and cards to organize tasks into To Do, In Progress, and Done categories, facilitating tracking of progress, priorities, and assignments for *piQture*. This feature aids in ensuring that our project stays on track and meets deadlines.

GitHub Actions

A standout feature of GitHub is GitHub Actions,[5] an automation tool that enables developers to define customized workflows that automate CI/CD tasks like code compilation, testing, and deployment. This automation streamlines the development workflow, improves efficiency, and ensures consistent code quality across projects while reducing manual effort at each stage. We utilize GitHub Actions to streamline the building, testing, and packaging workflows for *piQture*. These workflows are explained in Subsections 6.2.2, 6.2.3, and 6.2.4 below.

6.2.2 Building

With the GitHub repository in place, our focus shifts to configuring automated configurations that govern the code compilation, dependency resolution, and testing processes for the *piQture* library. Automating these tasks mitigates the risk of manual

[5]https://github.com/features/actions

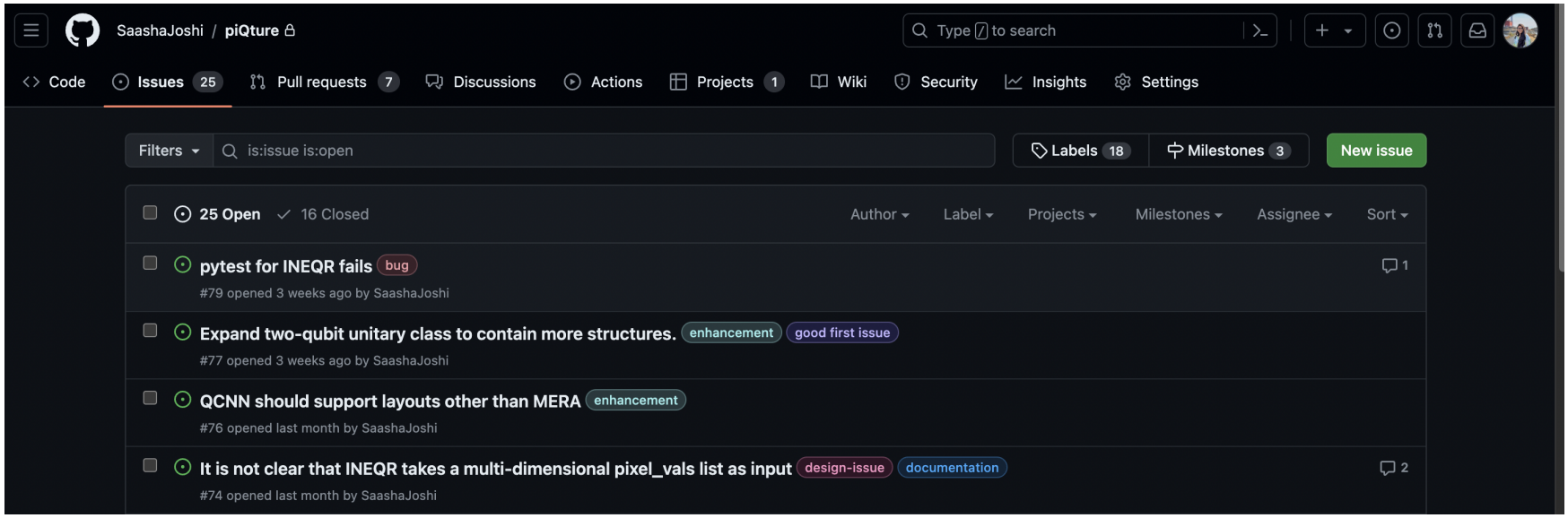


Figure 6.3: The Issues section contains listings of bug reports, feature, and documentation requests related to *piQture*.

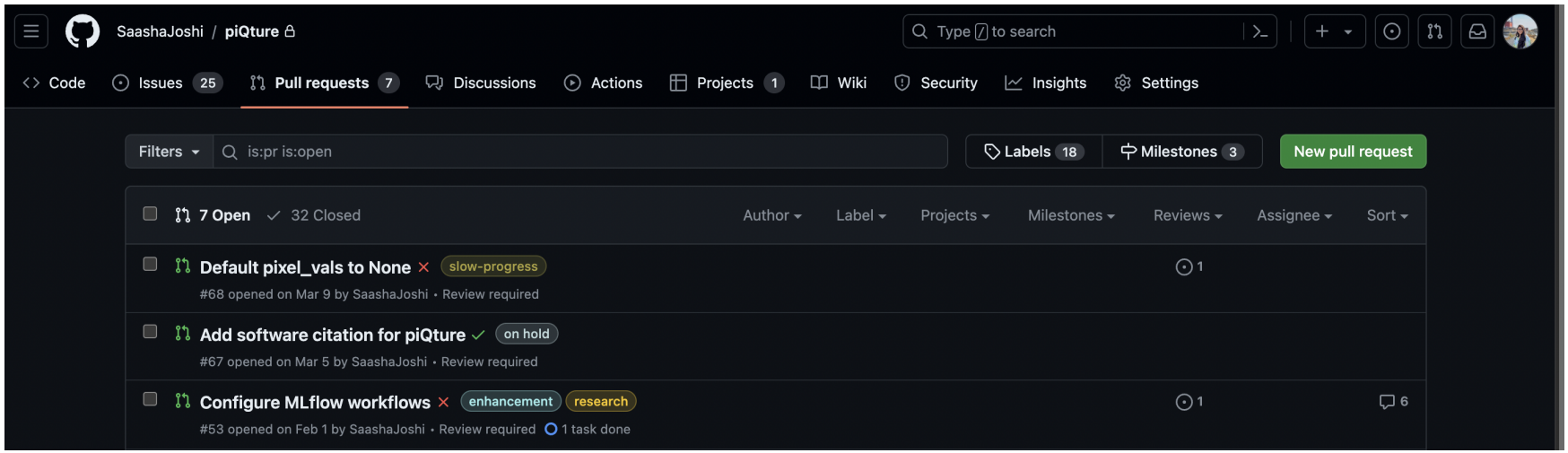


Figure 6.4: The Pull Requests section contains requests for new features and bug fixes.

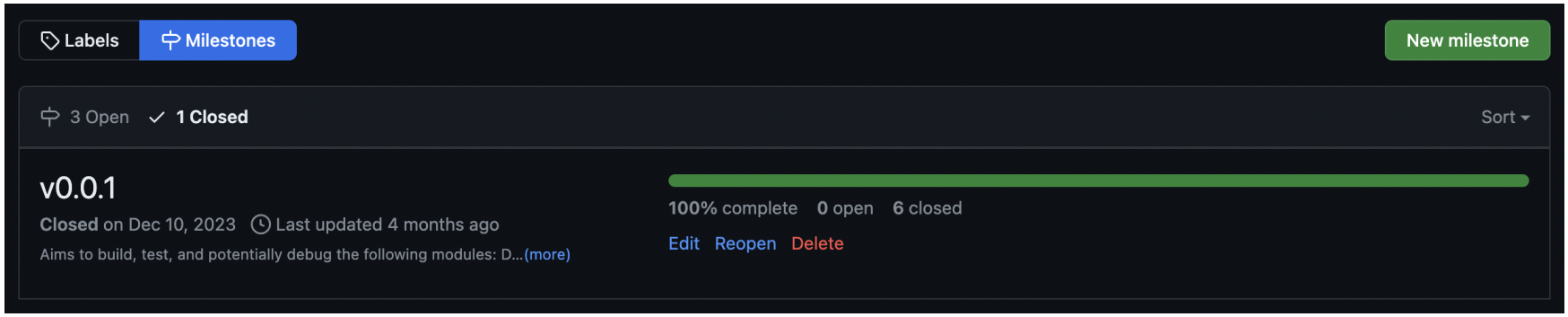


Figure 6.5: The Milestones feature facilitates tracking version releases for *piQture*.

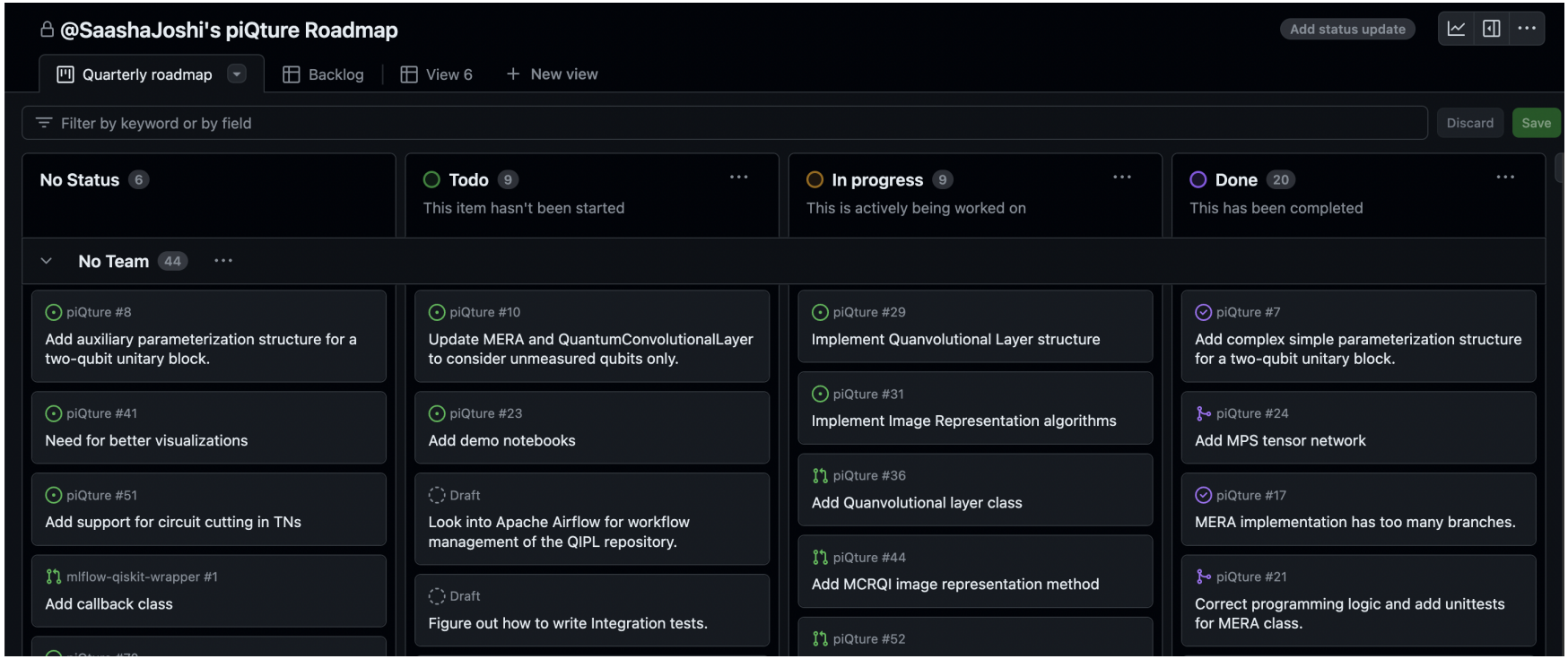


Figure 6.6: The GitHub Project assists in tracking progress and task prioritization for *piQture* via the Kanban-like Board view.

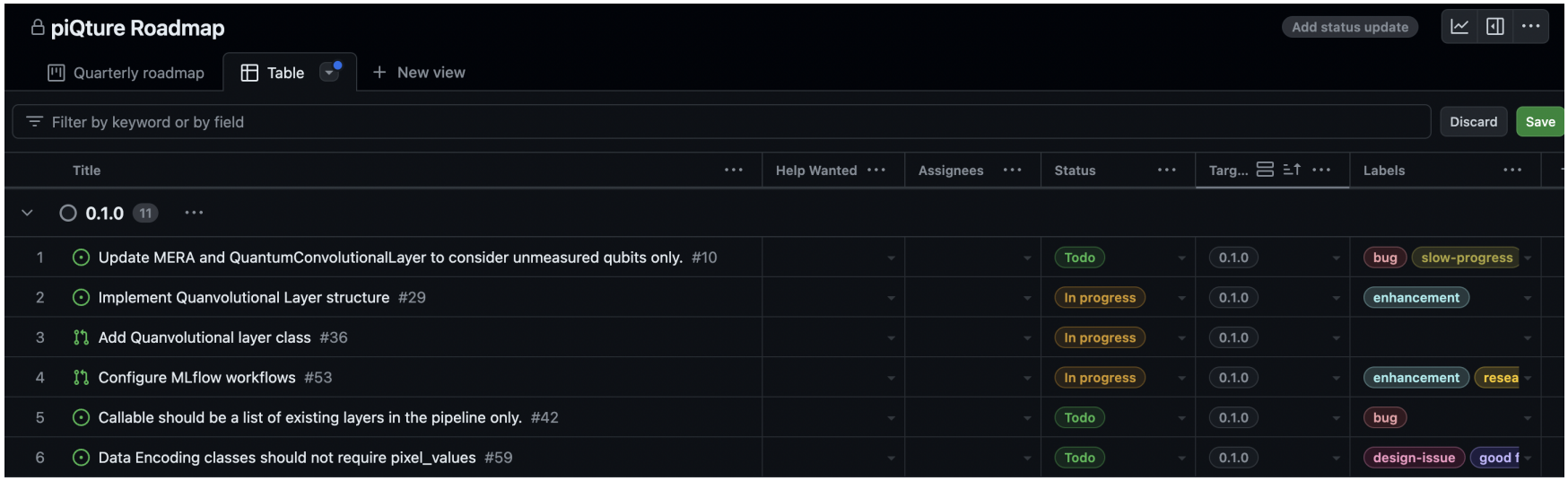


Figure 6.7: The GitHub Project assists in tracking progress, task prioritization, and version monitoring for *piQture* via the Table view.

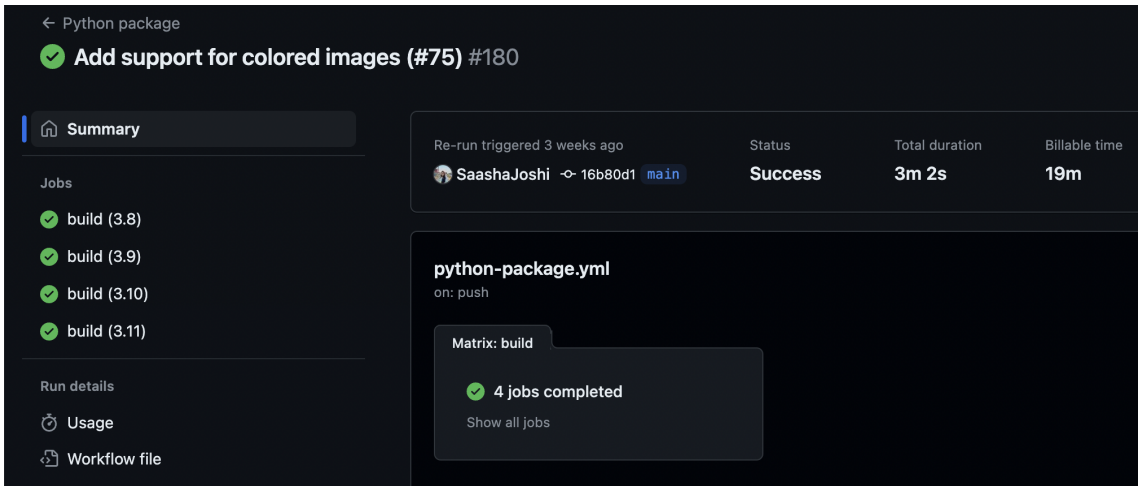


Figure 6.8: A python package build triggered by a commit in *piQture*, executing jobs across different Python versions.

errors and inconsistencies, ensuring the library is compiled consistently across different software environments, including various Python versions and operating systems.

A build refers to the process of compiling the source code, resulting in the generation of outputs such as executables, libraries, or intermediate artifacts used for testing or deployment. We utilize GitHub Actions in our library to facilitate the build step by writing a few configuration files. Since *piQture* is a Python-based library, we build the following configuration files that support different Python versions.

Build Configuration Files

`requirements.txt`	This text file catalogs all the library dependencies and Python packages, along with their version constraints. Its primary function is to ensure the library's reproducibility across various environments.
`setup.py`	This script file defines metadata such as library name, version, and other relevant details, facilitating the packaging of the library for distribution.
`python-package.yml`	This YAML file is a configuration file tailored for GitHub Actions. This file defines a sequence of steps to orchestrate the build and test processes.

The YAML file has components that trigger automatic build and testing processes. These components include,

Trigger: A trigger specifies an event that initiates the workflow, such as pushes made to specific branches, pull requests, or scheduled intervals.

Listing 6.1: Push and Pull Request trigger events defined on the main branch of *piQture*.

```yaml
name: Python package
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

Jobs: Jobs are the backbone of the workflow, outlining specific tasks to be executed within a defined environment. Each job is specified on a virtual machine or container environment, such as Ubuntu, Windows, or macOS, in which it will run.

Listing 6.2: A job with a defined container environment, strategies to manage test failures, and multiple Python versions for testing.

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: true
      matrix:
        python-version: ["3.8", "3.9", "3.10", "3.11"]
```

Each job also includes a sequence of steps that are defined and executed sequentially. These steps may include tasks like installing dependencies, examining code quality through the linting process, running unit tests, and more. Moreover, jobs are equipped with strategies that handle contingencies in case of build or test failures, enabling swift remediation actions.

Listing 6.3: A job with sequential steps for setting up a Python environment with GitHub Actions, installing dependencies, linting operations, and pytest testing.

```yaml
jobs:
    steps:
```

```
- uses: actions/checkout@v3
- name: Set up Python ${{ matrix.python-version }}
  uses: actions/setup-python@v3
  with:
    python-version: ${{ matrix.python-version }}
- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
- name: Lint with pylint
  run: |
    pylint $(git ls-files '*.py')
- name: Test with pytest
  run: |
    python -m pytest --cov=quantum_image_processing
        --cov-report=html:.pytest_cache/coverage_report
```

With these build configurations in place, automated CI/CD processes orchestrate the building process for the *piQture* library. GitHub Actions can parse these files to fetch necessary dependencies, execute commands, and streamline the development workflow, ensuring the readiness of the library for deployment.

6.2.3 Testing

In the subsequent testing phase, we utilize the pytest framework to write unit tests for the *piQture* library. These tests verify the functionality and behavior of each function in each module, ensuring the quality and validation of the codebase.

To automate the testing process, we utilize the Python configuration files, particularly `pythonpackage.yml` to orchestrate a GitHub Actions workflow. This workflow triggers the execution of unit tests with each commit and pull request created on the GitHub repository. These tests are performed across multiple Python versions, mentioned in the configuration file, enabling us to verify the compatibility and stability of the library across different environments.

This proactive approach to testing enhances the overall robustness of the *piQture* library, allowing developers to identify and address issues early in the development cycle.

Test Coverage

Code coverage is a metric used in software development to measure the proportion of code tested against the existing unit tests. It is expressed as a percentage, reflecting the ratio of lines of code executed during testing to the total lines of code in the codebase. Higher code coverage indicates a more thorough testing of the codebase, helping identify areas that may require additional testing and other potential gaps in the test suite.

For *piQture*, we commit to maintaining a high code coverage standard throughout the development workflow. We continuously track our code coverage metrics, ensuring that the testing metrics effectively cover the breadth of the codebase. Presently, the code coverage for *piQture* is 91%. This percentage is provided by Coveralls,[6] a web-based service that provides code coverage statistics.

6.2.4 Packaging

Once the codebase has been built and tested, the final step is to package *piQture* for distribution and deployment. Packaging consists of bundling the code and its dependencies and configuration files into a distributable format like a Docker container or a package archive, such as `.tar, .tar.gz`. The resulting package, also known as a release package, can contain documentation files, release notes, license information, installation instructions, and other metadata like version details. This comprehensive package ensures the end-users have all the necessary resources to deploy and utilize the *piQture* library. Furthermore, the packaging process can be automated using GitHub Actions. Listing 6.4 illustrates a YAML configuration file that orchestrates this automation.

Listing 6.4: A YAML configuration file for automating the packaging process and deploying it to PyPI.

```yaml
name: Deploy to PyPI
on:
  push:
    branches:
      - main
jobs:
  deploy:
```

[6] https://coveralls.io/

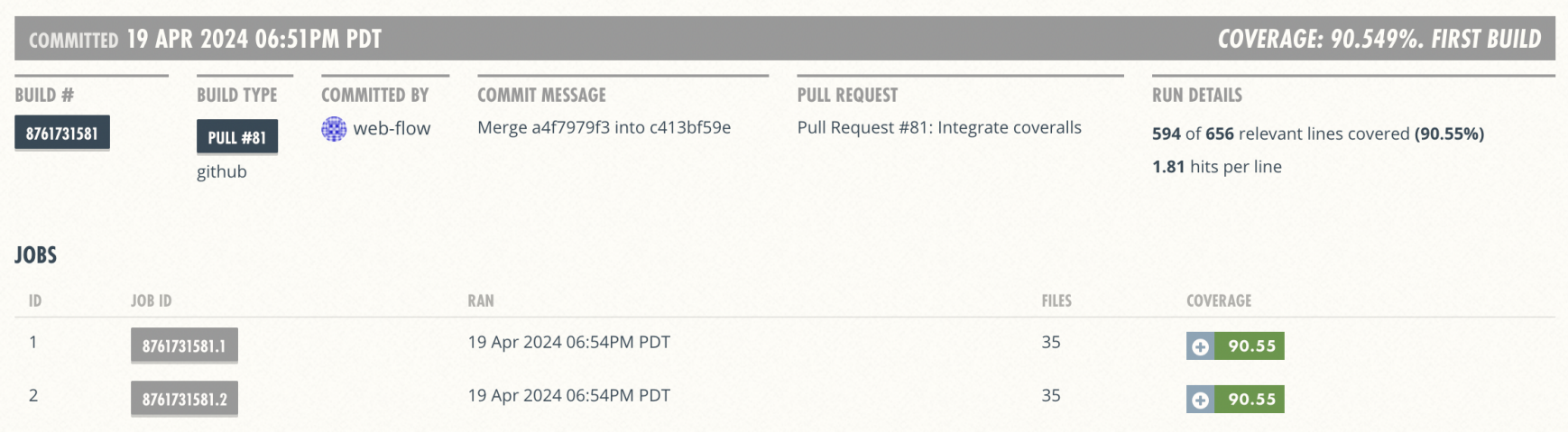


Figure 6.9: Code coverage for *piQture* after the latest commit, as provided by Coveralls.

```
runs-on: ubuntu-latest
strategy:
  matrix:
    python-version: [ "3.8", "3.9", "3.10", "3.11" ]
steps:
  - name: Build package
    run: python setup.py sdist bdist_wheel
  - name: Upload package to PyPI
    env:
      USERNAME: __token__
      PASSWORD: ${{ secrets.PYPI_API_TOKEN }}
    run: |
      python -m twine upload dist/*
```

6.3 *piQture* in Production

Upon completion of the testing and packaging stages, *piQture* transitions into a production workflow where workflow automation tools manage the execution and training of QML models sourced from the packaged source code. Tools like MLflow[7] [134] provide essential APIs for tracking experiments, optimizing models, and managing model versions. The trained QML models from the library are stored in a Model Registry, serving as a data hub accessible to users through a web API as a Prediction Service. This service allows users to interact with *piQture's* QML models, making predictions based on unseen input data.

At its current stage, deploying *piQture* into a production workflow represents a preliminary implementation. Although automated scheduling and model monitoring processes are not yet fully developed, manual intervention ensures the tracking of experiments and fine-tuning of QML models. Additionally, the current workflow is structured as a monolithic software, where various functionalities and components of *piQture* are integrated into a single unit, deployed either on PyPI[8] or as a web service. We make these decisions to optimize the time, simplicity, and efficiency of developing the initial phases of *piQture*. The following sections discuss the above processes, followed by a brief discussion of future expansion possibilities.

[7] https://mlflow.org/
[8] https://pypi.org/

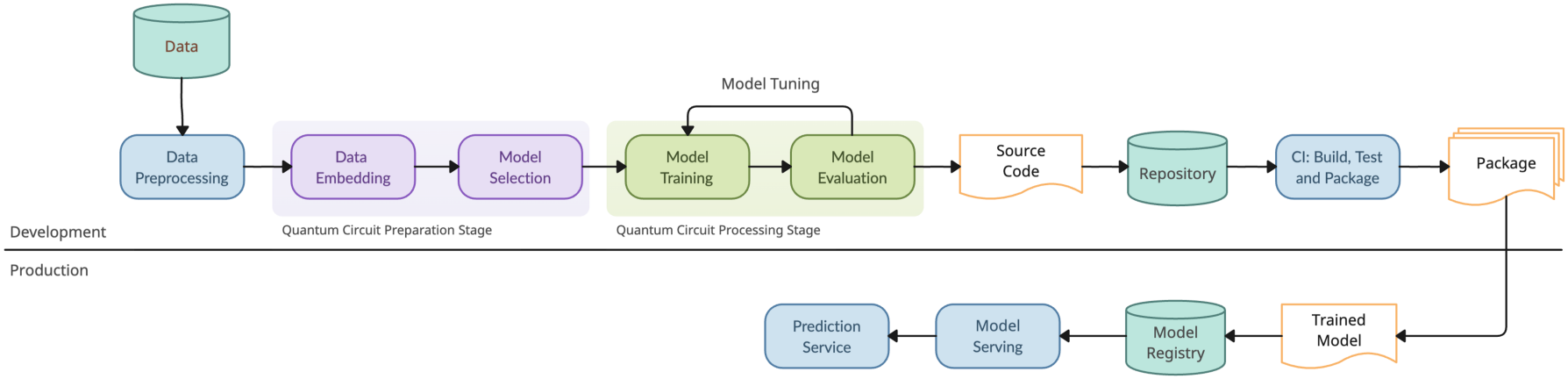


Figure 6.10: Development and production workflows of the *piQture* pipeline.

6.3.1 Model Management

In the production workflow following MLOps, robust model registry, artifact logging, and experiment tracking are essential for model management purposes.

Experiment Tracking

Experiment Tracking refers to the documentation and management of a project workflow. It involves capturing a wide range of experiment details, including training datasets, parameters and hyperparameters, model architectures, evaluation metrics, and outcomes obtained during the training and evaluation of a QML model. This comprehensive logging of relevant metadata from the experiments enables developers and users to trace the evolution and performance of a model while identifying strategies and areas for improvement. Figure 6.11 displays the interface of an MLflow Tracking server hosted locally on a set URI: http://127.0.0.1:5000/.

Artifact Logging

Artifact Logging involves systematically recording and storing various artifacts associated with each experiment. These artifacts include model binaries, hyperparameters, and evaluation metrics, including elements specific to QML models, such as circuit diagrams, transpiled circuit data, and variational parameters, as shown in Figures 6.13 and 6.14. Within *piQture*, we leverage MLflow's APIs during model script execution to log artifacts onto the MLflow Experiment Tracking Server. This meticulous process of logging ensures the availability of artifacts for future reference and analysis, which is crucial for the reproducibility of models and insights into their performance.

Model Registry

A Model Registry is a centralized database for storing, versioning, and managing trained QML models. Within *piQture*, we utilize Model Registry to catalog trained QML models, facilitating their easy retrieval and deployment. We utilize MLflow's Model Registry APIs to streamline the process of model registration, versioning, and even creating model aliases for easy reference, as depicted in Figure 6.12. Building a Model Registry ensures that the trained QML models are systematically organized and readily accessible for deployment. Furthermore, MLflow's Model Registry synchronizes the experiment data with model artifacts, providing a common platform for

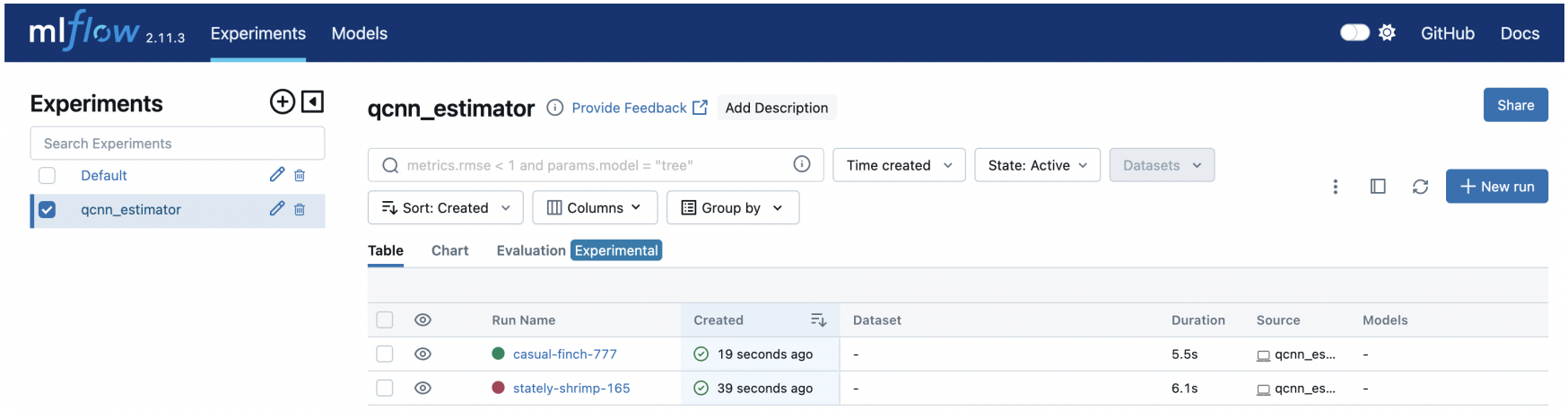


Figure 6.11: The MLflow Tracking Server, locally hosted at a specified URI.

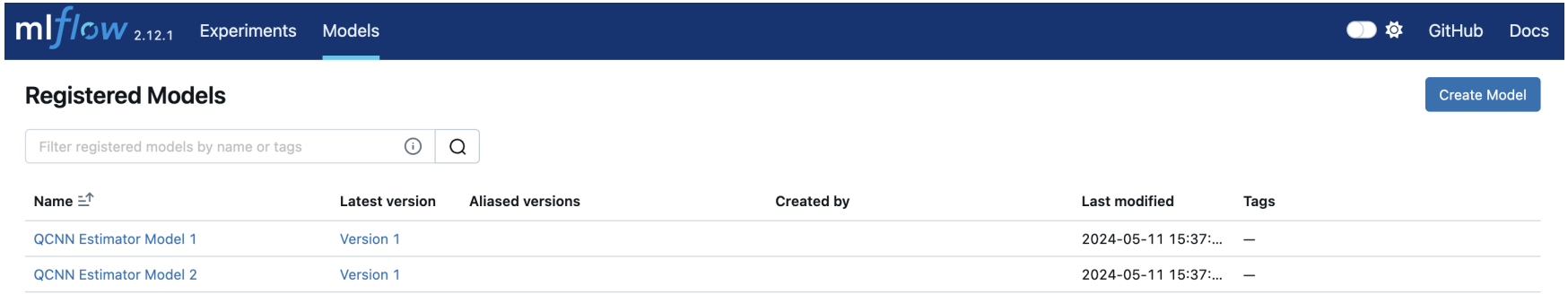


Figure 6.12: The MLflow Model Registry with saved QCNN models, locally hosted at a specified URI.

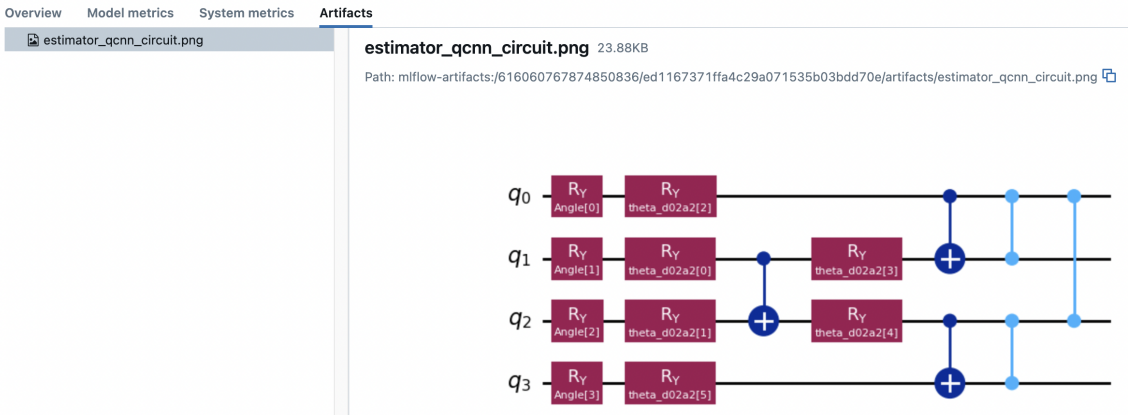


Figure 6.13: Circuit Diagram artifact of a QML model logged on the MLflow Experiment Tracking Server.

managing experiments and models.

To ensure seamless integration of QML models based on Qiskit within MLflow, we defined a custom Python function model, also referred to as a PyFunc. This PyFunc encapsulates the logic required for loading and executing QML utilizing Qiskit functionalities. Given that MLflow lacks pre-defined model flavors specific to Qiskit-based models, custom PyFuncs are indispensable for enabling the incorporation of QML models into MLflow's infrastructure. Listing 6.5 shows the custom PyFunc model designed to support Qiskit-based QML models from *piQture*.

Listing 6.5: A custom PyFunc that supports incorporation of QML model, based on Qiskit, into MLflow's infrastruture.

```
class QuantumModel(PythonModel, metaclass=ABCMeta):
    """
    Subclass of PythonModel class that wraps various
    quantum models as Python models.
    """
    def __init__(self, model: TrainableModel):
        self.model = model

    def predict(
        self, context=None, model_input=None, params:
            Optional[dict[str, Any]] = None
    ):
        return self.model.predict(X=model_input)
```

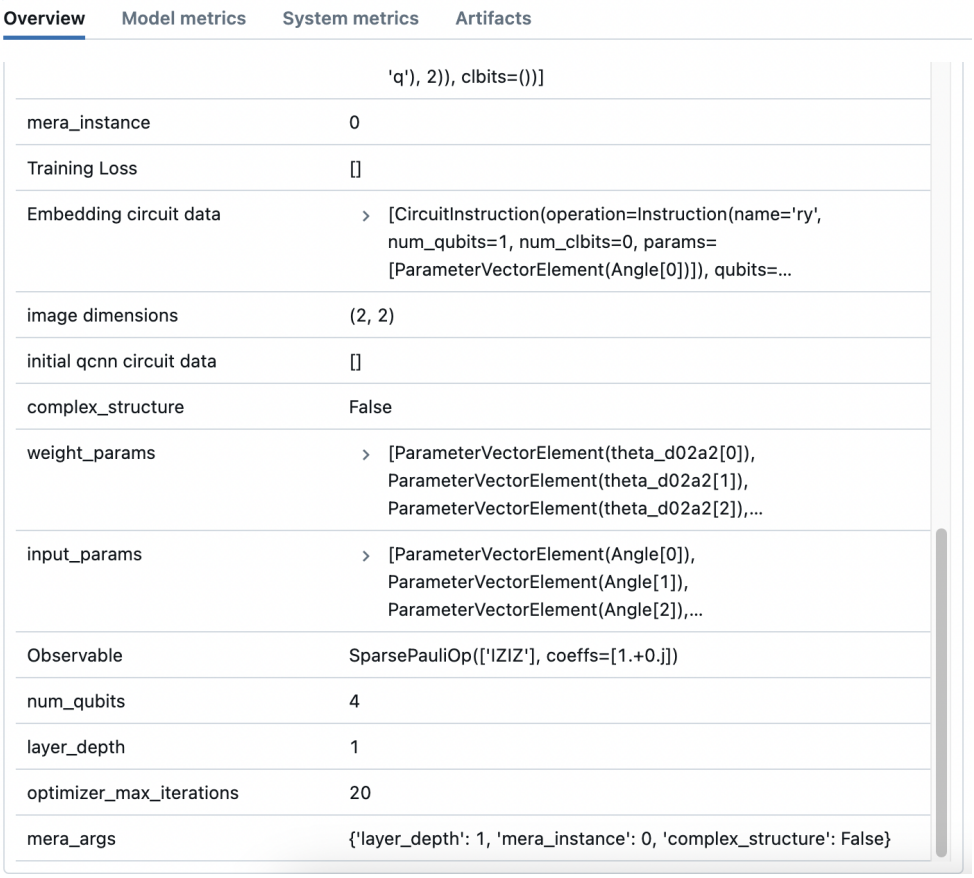


Figure 6.14: Attributes and parameter values of a QML model logged on the MLflow Experiment Tracking Server.

6.3.2 Prediction Service

Finally, we leverage the capabilities and functionalities of *piQture* to build a Prediction Service using Flask. This service establishes an API, delivering a user-friendly interface for accessing trained QML models stored in the Model Registry. Through Flask's lightweight framework, the Prediction Service handles incoming prediction requests and retrieves the necessary model from the registry. Subsequently, it generates new predictions based on the trained model. By deploying *piQture* as a web service, the registered and trained models are directly accessible to the users, enhancing interaction and usability of the library as a service.

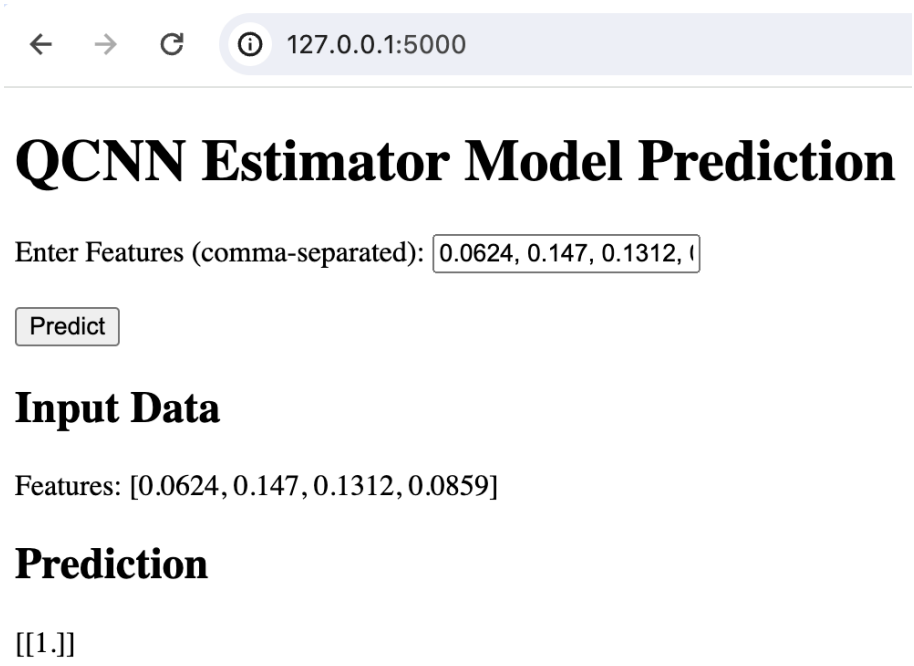


Figure 6.15: A Prediction Service hosted on a local development server with Flask. The QCNN model is pre-trained, and accessed from a local MLflow registry.

Chapter 7

Quick Start Guide: How to use *piQture*?

In this chapter, we provide a comprehensive quick start guide for utilizing the *piQture* library, a tool for applying QML models in image processing tasks. Section 7.1 lays out the necessary steps to create a setup required for installing the *piQture* library, for users and developers. Subsequently, Section 7.2 presents three tutorials, each demonstrating the utilization of distinct modules in the library: `image_representation`, `tensor_network_circuits`, and `neural_networks`.

7.1 Getting Started

7.1.1 Setup

Begin by creating a new Python environment or activating an existing one for working with the *piQture* library. Set up a Python virtual environment (venv) or a Conda environment and use pip or conda to install *piQture* package. Here is how to create a conda environment and manage a Python environment:

Listing 7.1: CLI commands to create and activate a Conda environment.

```
# Create a new conda environment
conda create -n piqture_environment python=3.x

# Activate the conda environment
conda activate piqture_environment
```

7.1.2 Installation

Once the Python environment is activated, the required *piQture* package can be installed using pip. You can install the latest version directly from PyPI.

```
pip install piqture
```

7.1.3 Installation from Source

To set up a development environment and install *piQture* from source, follow these steps:

- **Clone the *piQture* repository:** Start by cloning the *piQture* repository from GitHub.

  ```
  # Clone the GitHub repository.
  git clone https://github.com/SaashaJoshi/piQture.git
  ```

- Activate the Python environment and navigate to the `piQture` repository directory. Then, inside the Python environment, install the required dependencies from the `requirementstxt` configuration file using the following commands:

  ```
  # Install the required dependencies
  pip install -r requirements.txt
  ```

- **Install *piQture* in editable mode:** Installing a package in editable mode allows you to make changes to the source code. To install *piQture* from source in editable mode, run:

  ```
  # Install from source in editable mode
  pip install -e .
  ```

Now your development environment is set up, and *piQture* is installed from source. You can now start making changes to the code, running tests, and contributing to the project as a developer.

7.2 Tutorials

7.2.1 Tutorial 1: Building an INEQR encoding

In Chapter 3, we explored the Improved Novel Enhanced Quantum Representation (INEQR). This image encoding strategy embeds a non-square grayscale image onto a quantum circuit. In this tutorial, we utilize the `data_loader` and `data_encoder` modules in the *piQture* to build an INEQR circuit for a grayscale MNIST image.

First, we utilize the `load_mnist_dataset`, a PyTorch wrapper function, from the `data_loader` module to load an MNIST dataset. Next, we resize the images in the dataset to dimensions 2×2, and convert the pixel values to integral values, as per the requirements of the INEQR method. Finally, we use the `INEQR` method, present in the `image_representations` module, to generate an encoding circuit. The `INEQR` class accepts parameters such as `image_size` and a list of `pixel_val`. The circuit produced from executing code in Listing 7.2 is shown in Figure 7.1.

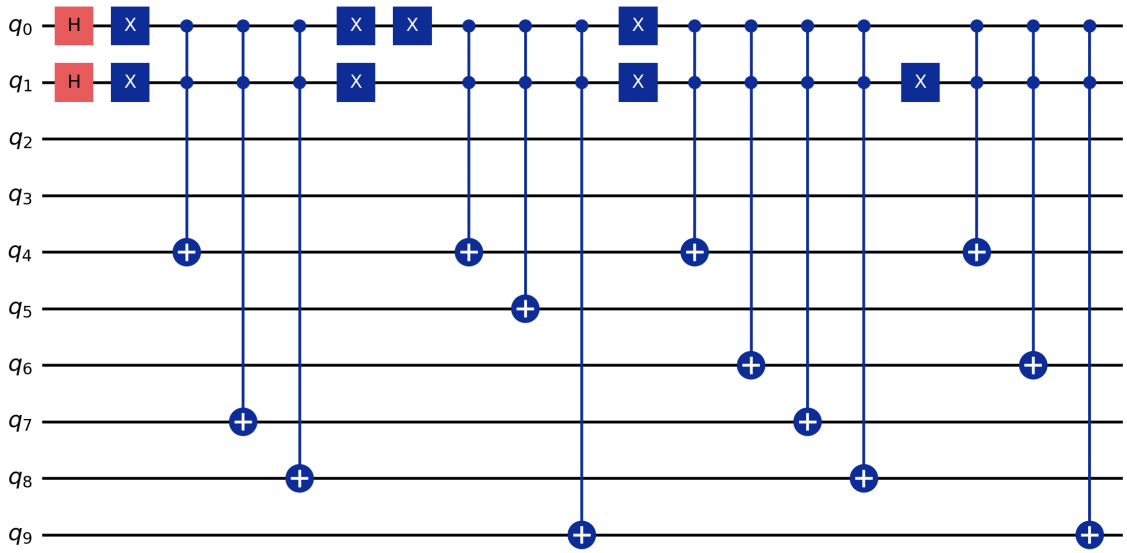


Figure 7.1: INEQR circuit generated for a 2×2 grayscale MNIST image with pixel values $[[38, 49], [46, 41]]$, from the corresponding code given in Listing 7.2. Qubits q_0 and q_1 encode the pixel positions in the basis states with the help of Hadamard and X gates, whereas the CX gates encode the pixel color information in binary format.

Listing 7.2: Constructing an INEQR image encoding with *piQture*.

```
# Imports
import torch.utils.data
```

```
from piqture.data_loader.mnist_data_loader import load_mnist_dataset
from piqture.data_encoder.image_representations.ineqr import INEQR

# Load MNIST dataset
train_dataset, test_dataset = load_mnist_dataset(img_size=2)

# Retrieve a single image from the dataset
image, label = train_dataset[0]
image_size = tuple(image.squeeze().size())

embedding = INEQR(image_size, image).ineqr()
```

7.2.2 Tutorial 2: Building a TTN tensor network

In this tutorial, we explore the `tensor_network_circuits` module in the *piQture* to build a TTN tensor network circuit. A TTN structure can be utilized as a parameterized ansatz within various QML algorithms, such as a VQC.

To initialize a TTN circuit, we utilize the `TTN` class from the `tensor_network_circuits` module. Once the circuit is instantiated, the user can call a class method tailored to the desired gate parameterization in the TTN structure. The available methods consist of two main structures: `ttn_simple` and `ttn_general`, each offering variations with complex and non-complex gates. The circuits produced from `ttn_simple` and `ttn_general` with non-complex gates, given in Listings 7.3 and 7.4, are shown in Figures 7.2 and 7.3.

Listing 7.3: Constructing TTN tensor network structure with simple gate parameterization and real gates.

```
# Imports
from piqture.tensor_network_circuits import TTN

# Instantiate a TTN circuit
num_qubits = 4
ttn = TTN(num_qubits)

# Specify the gate parameterization.
ttn = ttn.ttn_simple(complex_structure=False)
```

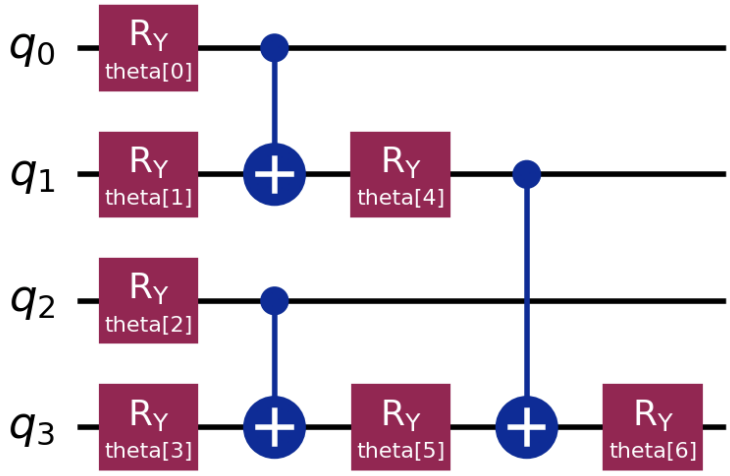


Figure 7.2: A TTN tensor network circuit, corresponding to Listing 7.3, constructed with a simple gate parameterization and real gates.

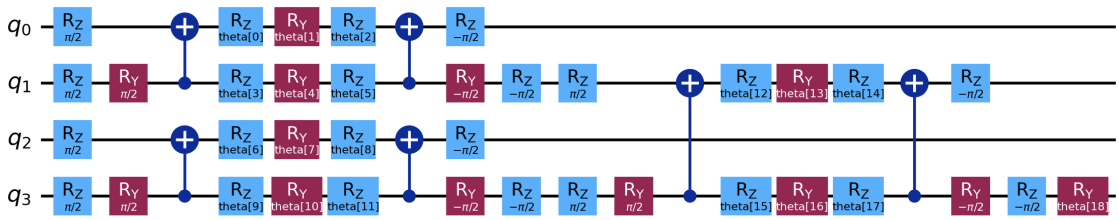


Figure 7.3: A TTN tensor network circuit, corresponding to Listing 7.4, constructed with a general gate parameterization and real gates.

Listing 7.4: Constructing TTN tensor network structure with general gate parameterization and real gates.

```
# Imports
from piqture.tensor_network_circuits import TTN

# Instantiate a TTN circuit
num_qubits = 4
ttn = TTN(num_qubits)

# Specify the gate parameterization.
ttn = ttn.ttn_general(complex_structure=False)
```

7.2.3 Tutorial 3: Training a QCNN model

In this tutorial, we construct a QCNN model, as detailed in Chapter 4, with the help of convolutional, pooling, and fully connected layers. To facilitate this, we utilize the `neural_networks` module in the *piQture* library to construct a layer sequence for our QCNN model.

The initial data encoding is performed by the `AngleEncoding` class inside the `data_encoding` module. Subsequently, we train and evaluate the QCNN model with the `EstimatorQNN` and `NeuralNetworkClassifier` classes from the *Qiskit Machine Learning* library. The resulting QCNN circuit, as generated by the code written in Listing 7.5, is shown in Figure 7.4. Whereas, Figure 7.5 illustrates the minimization of the objective function value over ten training and evaluation iterations.

Listing 7.5: Training a QCNN model with *piQture* and *Qiskit Machine Learning* libraries.

```
# Imports
from piqture.neural_networks import (
    QCNN,
    QuantumConvolutionalLayer,
    QuantumPoolingLayer2,
    FullyConnectedLayer,)
from qiskit_machine_learning.neural_networks import EstimatorQNN
from qiskit_machine_learning.algorithms import NeuralNetworkClassifier
from qiskit.quantum_info import SparsePauliOp
from qiskit_algorithms.optimizers.cobyla import COBYLA

# Initializing a Data Embedding circuit for 2x2 images.
embedding = angle_encoding(img_dim=(2, 2))

# Initializing a QCNN circuit for 2x2 images.
qcnn_circuit = QCNN(num_qubits=4)

# Gathering parameters for layer objects.
mera_params = {"layer_depth": 1, "mera_instance": 0,
    "complex_structure": False}
convolutional_params = {"mera_args": mera_params}

# Building the QCNN circuit with layers.
```

```
qcnn_circuit = qcnn_circuit.sequence([
    (QuantumConvolutionalLayer, convolutional_params),
    (QuantumPoolingLayer2, {}),
    (FullyConnectedLayer, {})
])

# Utilizing EstimatorQNN and NeuralNetworkClassifier to train and
    evaluate the circuit.
estimator_qcnn = EstimatorQNN(
    circuit=qcnn_circ,
    observables=SparsePauliOp(["IZIZ"]),
    input_params=embedding.params,
    weight_params=qcnn_circuit.parameters,
)

weights = algorithm_globals.random.random(estimator_qcnn.num_weights)
initial_point = np.random.random((len(qcnn_circuit.parameters),))

classifier = NeuralNetworkClassifier(
    estimator_qcnn,
    optimizer=COBYLA(maxiter=20),
    callback=callback_graph,
    initial_point=initial_point,
)

# Fitting the QCNN model on train dataset.
classifier.fit(train_img, train_labels)
score = classifier.score(train_img, train_labels)
```

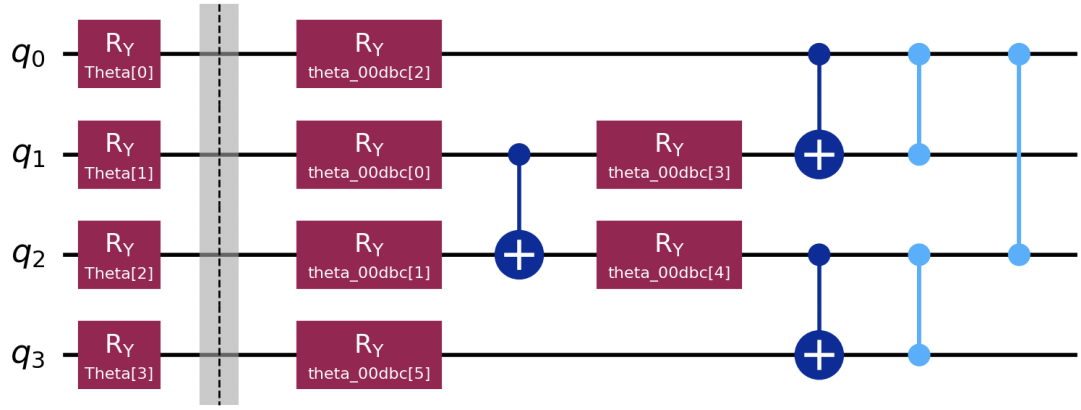


Figure 7.4: Circuit diagram of the QCNN model corresponding to the Listing 7.5. The initial rotation gates (before the barrier) implement the Angle Encoding strategy, followed by unitary gates (R_y, CX, and CZ) that apply quantum convolutional and pooling layers. The final CZ gate represents a fully connected layer.

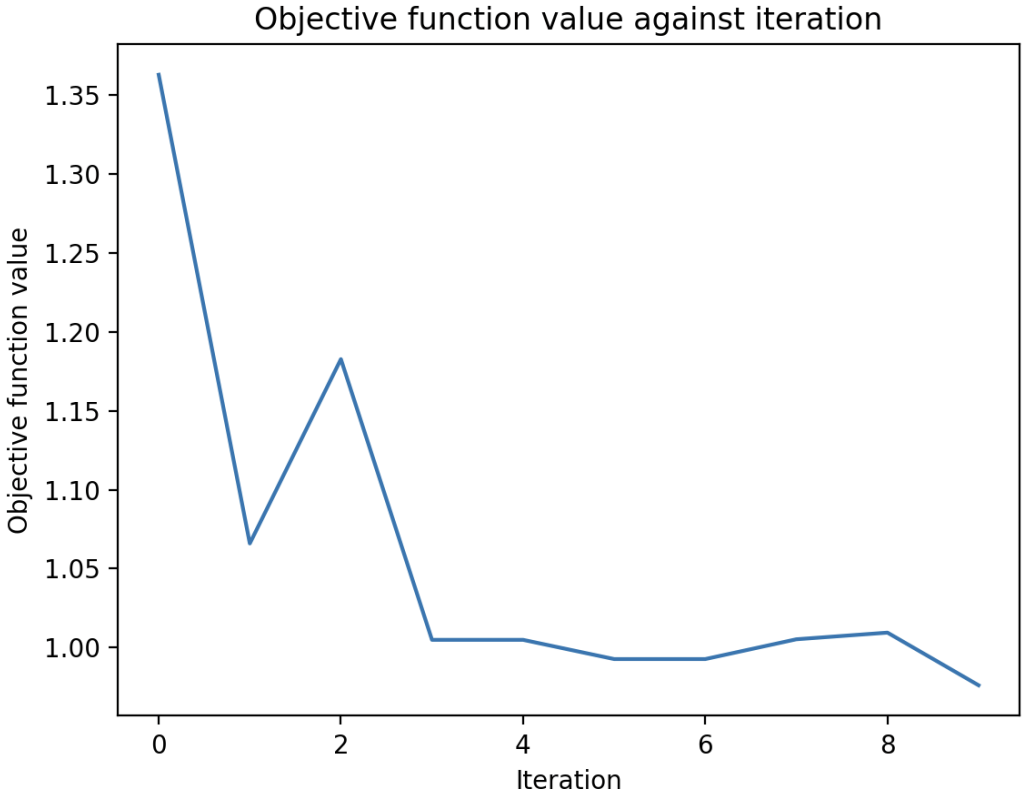


Figure 7.5: Model evaluation chart for the QCNN model corresponding to Listing 7.5, depicting a minimizing objective function value over 10 training and evaluation iterations.

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the contributions of our thesis and outline the avenues for future work and development of *piQture*.

8.1 Contributions

RQ1: What are the most common QML models? How are these models applied to image processing?

- We extensively reviewed QML models, primarily focusing on their applicability to image classification tasks.
- In Chapter 2, we presented a comprehensive background and literature review on Data Embedding, QIR methods, QML models such as QSVM, VQA including VQC and kernel methods, QTN, and QNN such as QCNN and QuanvNN, and QML workflows.
- Chapters 3 and 4 discussed the theory behind selective QIR methods and QML models, such as Quantum Kernel Estimator, VQC, QTN, and QCNN that are utilized for classification tasks.

RQ2: What is an effective architecture and workflow for a software library for implementing QML algorithms for image processing?

- We developed *piQture* library, a Python and Qiskit-based toolkit that facilitates the implementation, training, and deployment of QML models for image processing tasks.

- In Chapter 5, we presented the design and structural decisions behind the development of *piQture*, including modularized classes and functions, built to construct an effective and streamlined QML workflow.
- These decisions made *piQture* accessible to users familiar with classical machine learning without prior QML experience.
- We ensured the reliability, robustness, and reproducibility of *piQture* as software by employing CI/CD techniques, unit testing, and code coverage analysis.
- We open-sourced *piQture* to provide access to a wider community of users.
- We provided a concise quick-start guide to using *piQture* in Chapter 7.

RQ3: How can the developed workflow be streamlined to facilitate continuous integration, packaging, and deployment of QML models?

- In Chapter 6, we presented strategies to integrate automation techniques to streamline the integration and deployment processes for *piQture*.
- We utilized GitHub Actions to automate the building, testing, packaging, and deployment workflows within *piQture*.
- We facilitated model management within *piQture* by utilizing MLflow to track experiments, log model and experiment artifacts, and register models in a database for easy retrieval and analysis of trained QML models.
- We facilitated the practical adoption of pre-trained QML models by developing a Flask-based API to build a web-based prediction service.

8.2 Future Work

piQture has the potential to become an experimental platform for students and developers to explore and implement QML models. This section introduces the future research and development avenues for *piQture*.

8.2.1 Optimizing *piQture* Design and Structure

One avenue for future research and development is to optimize the design and structure of the *piQture* library by identifying areas of improvement. In particular, the following feature enhancements would be worth investigating: extending the `TwoQubitUnitary` class to support more gate parameterizations and decompositions, introducing better visualizations for the `neural_networks` module, implementing color and geometric transformations and compression on images, and adding support for circuit cutting. These potential enhancements are also listed in the Issues section of the *piQture* library on GitHub.

Another future research avenue is to address the reliance of *piQture's* Quantum Circuit Processing Stage on the `neural_networks` module in the *Qiskit Machine Learning* library, as discussed in Chapter 5. Upon experimenting, we realized that the current implementations of `SamplerQNN` and `EstimatorQNN` classes are not flexible enough to support parallel or distributed execution of multiple quantum circuits that may result from particular workflows, such as while implementing quantum circuit cutting. Introducing a custom `training` module to *piQture* would help to solve this problem.

8.2.2 Workflow Management and Monitoring

Despite the significant progress made in developing *piQture*, few areas of potential enhancements remain. One key area of focus is workflow management and monitoring using tools like Apache Airflow.[1] Such management would help to optimize the execution, scheduling, and monitoring of QML workflows to reflect the frequent changes in data or package dependencies of *piQture*.

Another useful enhancement would be to expand *piQture* to support cloud-based model management and deployment for broader accessibility. Currently, the model registry and web-based prediction services are hosted locally. Having cloud-based support is beneficial for extending the reach of *piQture* to a broader community as well as for providing *piQture* as a service to users.

[1] https://airflow.apache.org/

8.2.3 Model Evaluation Techniques

Model evaluation is essential when assessing the performance of trained models. It includes metrics such as accuracy, precision, recall, F1-score, and ROC-AUC score, offering insights into a model's performance on unseen data. It would be benficial to integrate model evaluation techniques into *piQture*, tailored to QML and image processing tasks. This integration would allow users to prioritize evaluation strategies within a project workflow, ensuring a comprehensive assessment of model performance and benchmarking.

Bibliography

[1] Mona Abdolmaleky, Mosayeb Naseri, Josep Batle, Ahmed Farouk, and Li-Hua Gong. Red-Green-Blue multi-channel quantum representation of digital images. *Optik*, 128:121–132, 2017.

[2] M.V. Altaisky. Quantum neural network, 2001. arXiv:quant-ph/0107012.

[3] Davide Anguita, Sandro Ridella, Fabio Rivieccio, and Rodolfo Zunino. Quantum optimization for training support vector machines. *Neural Networks*, 16(5):763–770, 2003.

[4] Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. Machine Learning in a Quantum World. In Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Ahmed Y. Tawfik, and Scott D. Goodwin, editors, *Advances in Artificial Intelligence*, volume 3060, pages 431–442. Springer, 2006.

[5] Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. Quantum speed-up for unsupervised learning. *Machine Learning*, 90(2):261–287, 2013.

[6] Martin Beisel, Johanna Barzen, Frank Leymann, Felix Truger, Benjamin Weder, and Vladimir Yussupov. Patterns for Quantum Error Handling. In *Proceedings of the 14th International Conference on Pervasive Patterns and Applications (PATTERNS 2022)*, pages 22–30, 2022.

[7] Marcello Benedetti, Delfina Garcia-Pintos, Oscar Perdomo, Vicente Leyton-Ortega, Yunseong Nam, and Alejandro Perdomo-Ortiz. A generative modeling approach for benchmarking and training shallow quantum circuits. *npj Quantum Information*, 5(1), 2019.

[8] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. Akash-Narayanan, Ali Asadi, Juan Miguel Arrazola, Utkarsh Azad, Sam Banning, Carsten Blank, Thomas R. Bromley, Benjamin A. Cordier, Jack Ceroni, Alain Delgado, Olivia Di Matteo, Amintor Dusko, Tanya Garg, Diego Guala, Anthony Hayes, Ryan Hill, Aroosa Ijaz, Theodor Isacsson, David Ittah, Soran Jahangiri, Prateek Jain, Edward Jiang, Ankit Khandelwal, Korbinian Kottmann, Robert A. Lang, Christina Lee, Thomas Loke, Angus Lowe, Keri McKiernan, Johannes Jakob Meyer, J. A. Montañez-Barrera, Romain Moyard, Zeyue Niu, Lee James O'Riordan, Steven Oud, Ashish Panigrahi, Chae-Yeun Park, Daniel Polatajko, Nicolás Quesada, Chase Roberts, Nahum Sá, Isidor Schoch, Borun Shi, Shuli Shu, Sukin Sim, Arshpreet Singh, Ingrid Strandberg, Jay Soni, Antal Száva, Slimane Thabet, Rodrigo A. Vargas-Hernández, Trevor Vincent, Nicola Vitucci, Maurice Weber, David Wierichs, Roeland Wiersema, Moritz Willmann, Vincent Wong, Shaoming Zhang, and Nathan Killoran. PennyLane: Automatic differentiation of hybrid quantum-classical computations, 2022. arXiv:1811.04968.

[9] Jacob Biamonte. Lectures on Quantum Tensor Networks, 2020. arXiv:1912.10049.

[10] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. `http://github.com/google/jax`, 2018.

[12] Lukas Brenner, Christophe Piveteau, and David Sutter. Optimal wire cutting with classical communication, 2023. arXiv:2302.03366.

[13] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David

Von Dollen, Hongxiang Chen, Shuxiang Cao, Roeland Wiersema, Hsin-Yuan Huang, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, and Masoud Mohseni. TensorFlow Quantum: A Software Framework for Quantum Machine Learning, 2021. arXiv:2003.02989.

[14] Fabian Bühler, Johanna Barzen, Martin Beisel, Daniel Georg, Frank Leymann, and Karoline Wild. Patterns for Quantum Software Development. In *Proceedings of the 15th International Conference on Pervasive Patterns and Applications (PATTERNS 2023)*, pages 30–39, 2023.

[15] Harry Buhrman, Richard Cleve, John Watrous, and Ronald de Wolf. Quantum fingerprinting. *Physical Review Letters*, 87(16):167902, 2001.

[16] Guang-Long Chen, Xian-Hua Song, Salvador E. Venegas-Andraca, and Ahmed A. Abd El-Latif. QIRHSI: novel quantum image representation based on HSI color space model. *Quantum Information Processing*, 21(1):5, 2022.

[17] Samuel Yen-Chi Chen, Chih-Min Huang, Chia-Wei Hsing, and Ying-Jer Kao. Hybrid quantum-classical classifier based on tensor network and variational quantum circuit, 2020. arXiv:2011.14651.

[18] Iris Cong, Soonwon Choi, and Mikhail D. Lukin. Quantum Convolutional Neural Networks. *Nature Physics*, 15(12):1273–1278, 2019. arXiv:1810.03787.

[19] Bojia Duan, Jiabin Yuan, Chao-Hua Yu, Jianbang Huang, and Chang-Yu Hsieh. A survey on HHL algorithm: From theory to application in quantum machine learning. *Physics Letters A*, 384(24):126595, 2020.

[20] David Millán Escrivá. Basic optical character recognition tutorial. `https://github.com/damiles/basicOCR`, July 2012.

[21] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming Transformers for High-Resolution Image Synthesis. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12868–12878. IEEE, 2021.

[22] Alexandr A. Ezhov and Dan Ventura. Quantum Neural Networks. In Nikola Kasabov, editor, *Future Directions for Intelligent Systems and Information Sciences: The Future of Speech and Image Technologies, Brain Computers, WWW, and Bioinformatics*, pages 213–235. Physica-Verlag HD, 2000.

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

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

[25] Li Fei and Zheng Baoyu. A study of quantum neural networks. In *International Conference on Neural Networks and Signal Processing, 2003. Proceedings of the 2003*, volume 1, pages 539–542, 2003.

[26] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. Open source software in quantum computing. *PLOS ONE*, 13(12):e0208561, 2018.

[27] R.A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: `https://doi.org/10.24432/C56C76`.

[28] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[29] Ilie-Daniel Gheorghe-Pop, Nikolay Tcholtchev, Tom Ritter, and Manfred Hauswirth. Quantum DevOps: Towards Reliable and Applicable NISQ Quantum Computing. In *2020 IEEE Globecom Workshops*, pages 1–6, Taipei, Taiwan, 2020. IEEE.

[30] Edward Grant, Marcello Benedetti, Shuxiang Cao, Andrew Hallam, Joshua Lockhart, Vid Stojevic, Andrew G. Green, and Simone Severini. Hierarchical quantum classifiers. *npj Quantum Information*, 4(1):65, 2018.

[31] Artyom M. Grigoryan and Sos S. Agaian. New look on quantum representation of images: Fourier transform representation. *Quantum Information Processing*, 19(5):148, 2020.

[32] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, pages 212–219, Philadelphia, Pennsylvania, United States, 1996. ACM Press.

[33] Diego Guala, Shaoming Zhang, Esther Cruz, Carlos A. Riofrío, Johannes Klepsch, and Juan Miguel Arrazola. Practical overview of image classification with tensor-network quantum circuits. *Scientific Reports*, 13(1):4427, 2023.

[34] Zhao-Yu Han, Jun Wang, Heng Fan, Lei Wang, and Pan Zhang. Unsupervised Generative Modeling Using Matrix Product States. *Physical Review X*, 8(3):031012, 2018.

[35] Md Ershadul Haque, Manoranjan Paul, Anwaar Ulhaq, and Tanmoy Debnath. Efficient quantum image representation and compression circuit using zero-discarded state preparation approach, 2023. arXiv:2306.12634.

[36] Md Ershadul Haque, Manoranjan Paul, Anwar Ulhaq, and Tanmoy Debnath. A Novel State Connection Strategy for Quantum Computing to Represent and Compress Digital Images. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2023.

[37] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters*, 103(15):150502, 2009. Publisher: American Physical Society.

[38] Vojtech Havlicek, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum enhanced feature spaces. *Nature*, 567(7747):209–212, 2019.

[39] Maxwell Henderson, Samriddhi Shakya, Shashindra Pradhan, and Tristan Cook. Quanvolutional neural networks: powering image recognition with quantum circuits. *Quantum Machine Intelligence*, 2(1):2, 2020.

[40] Rui Huang, Xiaoqing Tan, and Qingshan Xu. Variational quantum tensor networks classifiers. *Neurocomputing*, 452:89–98, 2021.

[41] William Huggins, Piyush Patil, Bradley Mitchell, K. Birgitta Whaley, and E. Miles Stoudenmire. Towards quantum machine learning with tensor networks. *Quantum Science and Technology*, 4(2):024001, 2019.

[42] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[43] Tak Hur, Leeseok Kim, and Daniel K. Park. Quantum convolutional neural network for classical data classification. *Quantum Machine Intelligence*, 4(1), 2022. arXiv:2108.00661.

[44] IBM Quantum. Qiskit 1.0.2. `https://www.ibm.com/quantum/qiskit`, 2024.

[45] Sofiene Jerbi, Lukas J. Fiderer, Hendrik Poulsen Nautrup, Jonas M. Kübler, Hans J. Briegel, and Vedran Dunjko. Quantum machine learning beyond kernel methods. *Nature Communications*, 14(1):517, January 2023.

[46] Nan Jiang, Hao Hu, Yijie Dang, and Wenyin Zhang. Quantum Point Cloud and its Compression. *International Journal of Theoretical Physics*, 56(10):3147–3163, 2017.

[47] Nan Jiang, Jian Wang, and Yue Mu. Quantum image scaling up based on nearest-neighbor interpolation with integer scaling ratio. *Quantum Information Processing*, 14(11):4001–4026, 2015.

[48] Nan Jiang and Luo Wang. Quantum image scaling using nearest neighbor interpolation. *Quantum Information Processing*, 14(5):1559–1571, 2015.

[49] Miguel Jiménez. *An infrastructure for autonomic and continuous long-term software evolution*. PhD thesis, University of Victoria, 2022.

[50] Subhash C. Kak. Quantum Neural Computing. In Peter W. Hawkes, editor, *Advances in Imaging and Electron Physics*, volume 94, pages 259–313. Elsevier, 1995.

[51] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, 2017.

[52] Ashish Kapoor, Nathan Wiebe, and Krysta Svore. Quantum Perceptron Models. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[53] Rabia Amin Khan. An improved flexible representation of quantum images. *Quantum Information Processing*, 18(7):201, 2019.

[54] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in Vision: A Survey. *ACM Computing Surveys*, 54(10s):1–41, 2022.

[55] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE access*, 2023.

[56] Ryan LaRose and Brian Coyle. Robust data encodings for quantum classifiers. *Physical Review A*, 102(3):032420, 2020.

[57] Jose I. Latorre. Image compression and entanglement, 2005. arXiv:quant-ph/0510031.

[58] Marco Lazzarin, Davide Emilio Galli, and Enrico Prati. Multi-class quantum classifiers with tensor network circuits for quantum phase recognition. *Physics Letters A*, 434:128056, 2022.

[59] Phuc Q. Le, Fangyan Dong, and Kaoru Hirota. A flexible representation of quantum images for polynomial preparation, image compression, and processing operations. *Quantum Information Processing*, 10(1):63–84, 2011.

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

[61] Claire I. Levaillant. Quantum multiple gray scale images encryption scheme in the bit plane representation model, 2024. arXiv:2401.00787.

[62] M. Lewenstein. Quantum Perceptrons. *Journal of Modern Optics*, 1994. Publisher: Taylor & Francis Group.

[63] Frank Leymann. Towards a Pattern Language for Quantum Algorithms. In Sebastian Feld and Claudia Linnhoff-Popien, editors, *Quantum Technology and Optimization Problems*, volume 11413, pages 218–230. Springer, Cham, 2019. LNCS.

[64] Frank Leymann and Johanna Barzen. The bitter truth about gate-based quantum algorithms in the NISQ era. *Quantum Science and Technology*, 5(4):044007, 2020.

[65] Hai-Sheng Li, Xiao Chen, Haiying Xia, Yan Liang, and Zuoshan Zhou. A Quantum Image Representation Based on Bitplanes. *IEEE Access*, 6:62396–62404, 2018.

[66] Hai-Sheng Li, Ping Fan, Hai-Ying Xia, Huiling Peng, and Shuxiang Song. Quantum Implementation Circuits of Quantum Signal Representation and Type Conversion. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(1):341–354, 2019.

[67] Hai-Sheng Li, Qingxin Zhu, Ri-Gui Zhou, Ming-Cui Li, Lan Song, and Hou Ian. Multidimensional color image storage, retrieval, and compression based on quantum amplitudes and phases. *Information Sciences*, 273:212–232, 2014.

[68] Panchi Li and Xiande Liu. Color image representation model and its application based on an improved FRQI. *International Journal of Quantum Information*, 16(01):1850005, 2018.

[69] Panchi Li, Hong Xiao, and Binxu Li. Quantum representation and watermark strategy for color images based on the controlled rotation of qubits. *Quantum Information Processing*, 15(11):4415–4440, 2016.

[70] Zhaokai Li, Xiaomei Liu, Nanyang Xu, and Jiangfeng Du. Experimental Realization of a Quantum Support Vector Machine. *Physical Review Letters*, 114(14):140504, 2015.

[71] Ding Liu, Shi-Ju Ran, Peter Wittek, Cheng Peng, Raul Blázquez García, Gang Su, and Maciej Lewenstein. Machine learning by unitary tensor network of hierarchical tree structure. *New Journal of Physics*, 21(7):073059, 2019.

[72] Ding Liu, Zekun Yao, and Quan Zhang. Quantum-Classical Machine learning by Hybrid Tensor Networks, 2020. arXiv:2005.09428.

[73] Junhua Liu, Kwan Hui Lim, Kristin L. Wood, Wei Huang, Chu Guo, and He-Liang Huang. Hybrid quantum-classical convolutional neural networks. *Science China Physics, Mechanics & Astronomy*, 64(9):290311, 2021.

[74] Kai Liu, Yi Zhang, Kai Lu, Xiaoping Wang, and Xin Wang. An Optimized Quantum Representation for Color Digital Images. *International Journal of Theoretical Physics*, 57(10):2938–2948, 2018.

[75] Xingbin Liu, Di Xiao, Wei Huang, and Cong Liu. Quantum Block Image Encryption Based on Arnold Transform and Sine Chaotification Model. *IEEE Access*, 7:57188–57199, 2019.

[76] Yuhan Liu, Wen-Jun Li, Xiao Zhang, Maciej Lewenstein, Gang Su, and Shi-Ju Ran. Entanglement-Based Feature Extraction by Tensor Network Machine Learning. *Frontiers in Applied Mathematics and Statistics*, 7:716044, 2021.

[77] Ian MacCormack, Conor Delaney, Alexey Galda, Nidhi Aggarwal, and Prineha Narang. Branching quantum convolutional neural networks. *Physical Review Research*, 4(1):013117, 2022.

[78] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

[79] John Martyn, Guifre Vidal, Chase Roberts, and Stefan Leichenauer. Entanglement and Tensor Networks for Supervised Image Classification, July 2020. arXiv:2007.06082.

[80] Jarrod R. McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, February 2016.

[81] Norhan Nasr, Ahmed Younes, and Ashraf Elsayed. Efficient representations of digital images on quantum computers. *Multimedia Tools and Applications*, 80(25):34019–34034, 2021.

[82] Andrew Ng. CS229 Lecture Notes. We need much more info here!!, 2000.

[83] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. Tensorizing Neural Networks. *Advances in Neural Information Processing Systems (NIPS 2015)*, 2015.

[84] Alexander Novikov, Mikhail Trofimov, and Ivan Oseledets. Exponential Machines, 2017. arXiv:1605.03795.

[85] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[86] I.V. Oseledets. Tensor-Train Decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[87] P.J.J. O'Malley, R. Babbush, I.D. Kivlichan, J. Romero, J.R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A.G. Fowler, E. Jeffrey, E. Lucero, A. Megrant, J.Y. Mutus, M. Neeley, C. Neill, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T.C. White, P.V. Coveney, P.J. Love, H. Neven, A. Aspuru-Guzik, and J.M. Martinis. Scalable Quantum Simulation of Molecular Energies. *Physical Review X*, 6(3):031007, 2016.

[88] Jae-Eun Park, Brian Quanz, Steve Wood, Heather Higgins, and Ray Harishankar. Practical application improvement to Quantum SVM: theory to practice, 2020. arXiv:2012.07725.

[89] Tianyi Peng, Aram W. Harrow, Maris Ozols, and Xiaodi Wu. Simulating Large Quantum Circuits on a Small Quantum Computer. *Physical Review Letters*, 125(15):150504, 2020.

[90] Ricardo Perez-Castillo, Luis Jimenez-Navajas, and Mario Piattini. Modelling Quantum Circuits with UML. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 7–12, Madrid, Spain, 2021. IEEE.

[91] Mitja Perus. Neural Networks as a Basis for Quantum Associative Networks. `https://api.semanticscholar.org/CorpusID:2511138`, 2004.

[92] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a quantum processor. *Nature Communications*, 5(1):4213, 2014.

[93] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[94] Ricardo Pérez-Castillo and Mario Piattini. Design of classical-quantum systems with UML. *Computing*, 104(11):2375–2403, 2022.

[95] Carlos A. Pérez-Delgado and Hector G. Perez-Gonzalez. Towards a Quantum Software Modeling Language. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 442–444, Seoul Republic of Korea, 2020.

[96] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. Quantum Support Vector Machine for Big Data Classification. *Physical Review Letters*, 113(13):130503, 2014.

[97] Hans-Martin Rieser, Frank Köster, and Arne Peter Raulf. Tensor networks for quantum machine learning. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 479(2275):20230218, 2023. arXiv:2303.11735.

[98] Jonathan Romero, Jonathan P. Olson, and Alan Aspuru-Guzik. Quantum autoencoders for efficient compression of quantum data. *Quantum Science and Technology*, 2(4):045001, 2017.

[99] Engin Şahin and İhsan Yilmaz. QRMW: quantum representation of multi wavelength images. *Turkish Journal of Electrical Engineering & Computer Sciences*, 26(2):768–779, 2018.

[100] Jianzhi Sang, Shen Wang, and Qiong Li. A novel quantum representation of color digital images. *Quantum Information Processing*, 16(2):42, 2017.

[101] M. Schuld, I. Sinayskiy, and F. Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185, 2015. arXiv:1409.3097.

[102] Maria Schuld. Supervised quantum machine learning models are kernel methods, 2021. arXiv:2101.11020 [quant-ph, stat].

[103] Maria Schuld and Nathan Killoran. Quantum machine learning in feature Hilbert spaces. *Physical Review Letters*, 122(4):040504, 2019. arXiv:1803.07128.

[104] Maria Schuld and Francesco Petruccione. *Machine Learning with Quantum Computers*. Quantum Science and Technology. Springer, 2021.

[105] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. The quest for a Quantum Neural Network. *Quantum Information Processing*, 13(11):2567–2586, 2014.

[106] Y-Y Shi, L-M Duan, and Guifre Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Physical review a*, 74(2):022320, 2006.

[107] Sukin Sim, Peter D. Johnson, and Alán Aspuru-Guzik. Expressibility and Entangling Capability of Parameterized Quantum Circuits for Hybrid Quantum-Classical Algorithms. *Advanced Quantum Technologies*, 2(12):1900070, 2019.

[108] Michael Siomau. A quantum model for autonomous learning automata. *Quantum Information Processing*, 13(5):1211–1221, 2014.

[109] Iskandar Sitdikov, David Garcia Valinas, Francisco Martin Fernandez, Iulia Zidaru, Paul Schweigert, and Akihiko Kuroda. Quantum Serverless. `https://github.com/Qiskit-Extensions/quantum-serverless`, February 2023. original-date: 2022-09-15T18:33:43Z.

[110] Edwin Stoudenmire and David J. Schwab. Supervised learning with tensor networks. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

[111] E.M. Stoudenmire. Learning Relevant Features of Data with Multi-scale Tensor Networks. *Quantum Science and Technology*, 3(3):034003, July 2018.

[112] Bo Sun, Phuc Q. Le, Abdullah M. Iliyasu, Fei Yan, J. Adrian Garcia, Fangyan Dong, and Kaoru Hirota. A Multi-Channel Representation for images on quantum computers using the RGBα color space. In *2011 IEEE 7th International Symposium on Intelligent Signal Processing*, pages 1–6. IEEE, 2011.

[113] Yudai Suzuki, Hiroshi Yano, Qi Gao, Shumpei Uno, Tomoki Tanaka, Manato Akiyama, and Naoki Yamamoto. Analysis and synthesis of feature map for kernel-based quantum classifier. *Quantum Machine Intelligence*, 2(1):9, 2020. arXiv:1906.10467.

[114] Farrokh Vatan and Colin Williams. Optimal quantum circuits for general two-qubit gates. *Physical Review A*, 69(3):032315, 2004.

[115] Salvador E. Venegas-Andraca and J.L. Ball. Processing images in entangled quantum systems. *Quantum Information Processing*, 9(1):1–11, 2010.

[116] Salvador E. Venegas-Andraca and Sougato Bose. Storing, processing, and retrieving an image using quantum mechanics. In *Quantum Information and Computation*, volume 5105 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 137–147, 2003.

[117] Frank Verstraete and J Ignacio Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv preprint cond-mat/0407066*, 2004.

[118] Guifre Vidal. Entanglement renormalization: an introduction. *arXiv preprint arXiv:0912.1651*, 2009.

[119] Kwok Ho Wan, Oscar Dahlsten, Hlér Kristjánsson, Robert Gardner, and M.S. Kim. Quantum generalisation of feedforward neural networks. *npj Quantum Information*, 3(1):36, 2017.

[120] Bing Wang, Meng-qi Hao, Pan-chi Li, and Zong-bao Liu. Quantum Representation of Indexed Images and its Applications. *International Journal of Theoretical Physics*, 59(2):374–402, 2020.

[121] Hanrui Wang, Yongshan Ding, Jiaqi Gu, Zirui Li, Yujun Lin, David Z. Pan, Frederic T. Chong, and Song Han. QuantumNAS: Noise-Adaptive Search for Robust Quantum Circuits. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 692–708, April 2022.

[122] Ling Wang, Qiwen Ran, and Jing Ma. Double quantum color images encryption scheme based on DQRCI. *Multimedia Tools and Applications*, 79(9-10):6661–6687, 2020.

[123] Ling Wang, Qiwen Ran, Jing Ma, Siyuan Yu, and Liying Tan. QRCI: A new quantum representation model of color digital images. *Optics Communications*, 438:147–158, 2019.

[124] Zhaobin Wang, Minzhe Xu, and Yaonan Zhang. Review of Quantum Image Processing. *Archives of Computational Methods in Engineering*, 29(2):737–761, 2022.

[125] Benjamin Weder, Johanna Barzen, Frank Leymann, and Marie Salm. Automated Quantum Hardware Selection for Quantum Workflows. *Electronics*, 10(8):984, 2021.

[126] Benjamin Weder, Johanna Barzen, Frank Leymann, Marie Salm, and Daniel Vietz. The Quantum software lifecycle. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software*, pages 2–9, Virtual USA, 2020. ACM.

[127] Benjamin Weder, Uwe Breitenbucher, Frank Leymann, and Karoline Wild. Integrating Quantum Computing into Workflow Modeling and Execution. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 279–291, 2020.

[128] ShiJie Wei, YanHu Chen, ZengRong Zhou, and GuiLu Long. A quantum convolutional neural network on NISQ devices. *AAPPS Bulletin*, 32(1):2, 2022.

[129] Manuela Weigold, Johanna Barzen, Frank Leymann, and Daniel Vietz. Patterns for Hybrid Quantum Algorithms. In Johanna Barzen, editor, *Service-Oriented Computing*, pages 34–51, Cham, 2021. Springer International Publishing.

[130] Peter Wittek. *Quantum Machine Learning: What Quantum Computing Means to Data Mining*. Academic Press, 2014.

[131] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017. arXiv:1708.07747.

[132] Guanlei Xu, Xiaogang Xu, Xun Wang, and Xiaotong Wang. Order-encoded quantum image model and parallel histogram specification. *Quantum Information Processing*, 18(11):346, 2019.

[133] Jiaying Yang, Ahsan Javed Awan, and Gemma Vall-Llosera. Support Vector Machines on Noisy Intermediate Scale Quantum Computers. *arXiv:1909.11988*, 2019.

[134] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.

[135] Felipe Zapata, Lars Ridder, Johan Hidding, Christoph R. Jacob, Ivan Infante, and Lucas Visscher. QMflows: A Tool Kit for Interoperable Parallel Workflows in Quantum Chemistry. *Journal of Chemical Information and Modeling*, 59(7):3191–3197, 2019.

[136] Yi Zhang, Kai Lu, Yinghui Gao, and Mo Wang. NEQR: A novel enhanced quantum representation of digital images. *Quantum Information Processing*, 12(8):2833–2860, 2013.

[137] Yi Zhang, Kai Lu, Yinghui Gao, and Kai Xu. A novel quantum representation for log-polar images. *Quantum Information Processing*, 12(9):3103–3126, 2013.

[138] Chen Zhao and Xiao-Shan Gao. QDNN: Deep neural networks with quantum layers. *Quantum Machine Intelligence*, 3(1):15, 2021.

[139] Jian Zhao, Yuan-Hang Zhang, Chang-Peng Shao, Yu-Chun Wu, Guang-Can Guo, and Guo-Ping Guo. Building quantum neural networks based on a swap test. *Physical Review A*, 100(1):012334, 2019.

[140] Jianjun Zhao. Quantum Software Engineering: Landscapes and Horizons, 2021. arXiv:2007.07047.

[141] Leo Zhou, Sheng-Tao Wang, Soonwon Choi, Hannes Pichler, and Mikhail D. Lukin. Quantum Approximate Optimization Algorithm: Performance, Mechanism, and Implementation on Near-Term Devices. *Physical Review X*, 10(2):021067, 2020.

[142] Nanrun Zhou, Xingyu Yan, Haoran Liang, Xiangyang Tao, and Guangyong Li. Multi-image encryption scheme based on quantum 3D Arnold transform and scaled Zhongtang chaotic system. *Quantum Information Processing*, 17(12):338, 2018.