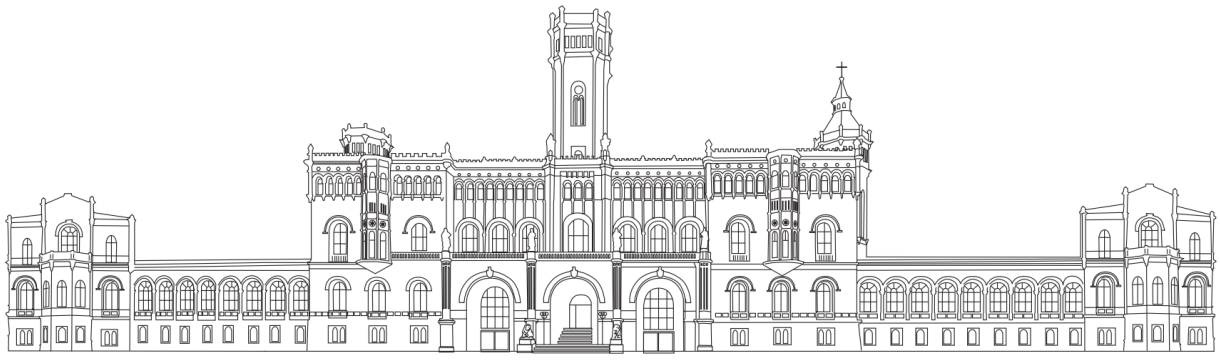


LEIBNIZ UNIVERSITY HANOVER
Quantum Information Group

QUANTUM MACHINE LEARNING OF GRAPH-STRUCTURED QUANTUM DATA

By

JULIUS JOHANNES KÖHLER

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

DECEMBER 2020

© Copyright by JULIUS JOHANNES KÖHLER, 2020
All Rights Reserved

QUANTUM MACHINE LEARNING OF GRAPH-STRUCTURED QUANTUM DATA

by Julius Johannes Köhler,
Leibniz University Hanover
December 2020

Abstract

Graph-structures are elementary models in natural science. We developed a quantum machine learning algorithm to execute unsupervised and semi-supervised training tasks on graph-structured quantum data with a quantum neural network. The algorithm teaches representations of partially labelled training sets of multi-qubit states to a quantum device with respect to a hidden graph-structure and the labels. Our classical simulation of the quantum learning algorithm shows excellent behaviour and offers new perspectives for future quantum computers.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter One

Introduction

In recent years, scientists have predicted 'quantum supremacy' or the superiority of quantum computers over classical computers [1, 2]. Their goal was to build a quantum device to solve a computational task that no conventional device can solve in a reasonable amount of time. In 1982, Richard Feynman claimed that quantum computers might be able to simulate a physical system which is not simulatable using a classical computer [3], and then in 2019, Google published a paper in Nature and described their newest quantum processor, which can reduce the processing time of sampling a quantum circuit from 10.000 years to 200 seconds. These processors can simulate 53 qubits or, in other words, can compute a 10^{16}-dimensional Hilbert space [4]. In reaction to Google's publication, IBM dampen expectations and argued that "an ideal simulation of the same task can be performed on a classical system in 2.5 days and with far greater fidelity"[5]. This disagreement between two eminent research groups might be due to the newness of the field of quantum computers. A priori, it is unhelpful to argue about 'quantum supremacy' in the presence of tasks which do not benefit from quantum devices [6]. Instead, the symbiosis of conventional and quantum devices should receive focus, since both profit from each other. For example machine learning algorithms can compress a quantum circuit for quantum computers [7] and Shor's algorithm boosts integer factorisation [8]. Moreover, further research in quantum information theory about quantum computing will bring new quantum algorithms as tools to exploit quantum devices and achieve the goal of quantum computers: to solve real-world problems. This master thesis thus concerns a new class of quantum algorithms, quantum machine learning of graph-structured quantum data using a quantum neural network. Hence, there is the following question: Is it possible to extend the ideas behind unsupervised and semi-supervised machine learning on graph-structured data and learn nearly identical tasks with a set of graph-structured multi-qubit states by using a quantum neural network [9]?

Machine learning (ML), a subset of artificial intelligence (AI), represents a current peak of computer science research. Machine learning algorithms aim to automatically improve their capability in solving a task through experience gained from 'training data' [10], and their applications have expanded from email filtering [11] to self-driven cars [12, 13]. Deep learning, a subset of ML, invokes artificial neural networks as learning architecture[14], whose learning process is inspired by biological systems. The layers of a neural network can evolve a deep inner structure of the training data, which primarily inspired the quantum neural network for this thesis. Deep learning algorithms are applied in audio, speech and image recognition [15, 16, 17, 18] as well as in video games [19], and a subfield of machine and deep learning regards graph-structured data [20, 21]. This subject is important since nature often produces measurable components in a correlated fashion, which can be represented by a directed or undirected graph.
ML provides three categories of learning tasks: unsupervised, supervised and semi-supervised learning. Recent unsupervised learning methods have trained embeddings of graph-structured data in lower dimensions or vertex representations, without distorting the graph's topological structure [22, 23, 24, 25]. Semi-supervised training use methods to classify nodes, for example by using graph convolutional networks (GCN) [26, 27, 28, 29].

Machine learning also includes a large range of applications in physics and especially quantum theory [30, 31, 32]. Quantum machine learning (QML) can be divided into three departments: to use classical machine learning to improve quantum tasks ('QC' ML) [33, 34, 35]; to hasten classical ML using quantum algorithms ('CQ' ML) [36, 37, 38, 39]; and to apply QML to quantum data ('QQ' ML) [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]. These departments all include learning tasks on graphs. Some publications have combined QML and graph structures, such as the quantum generalization of the random walk [50], convolutional networks [51, 52] and the quantum graph neural network, which is applied to graph-structured quantum processes [53]. Nevertheless, QML and graphs remain unexplored.

In this thesis, we focus on quantum neural networks to learn several unsupervised and semi-supervised representations and labels [54, 26] of quantum data with a hidden classical (generally physical) graph structure, which can thus be regarded as part of 'QQ' deep learning. The most promising neural network ansatz for this project is provided by the quantum neural network (QNN) [9], but other architectures should be considered [47, 55, 43, 45, 46, 48, 56].

This work begins with an overview of ML on graph-structured data, which will aid understanding the

construction of graph-structured quantum data and provide an outlook for GCNs. The next chapter comprises the main results, including the mathematical background of qubits and construction of the QNN; development of the graph-structured quantum data; and a QML task for this data for unsupervised and semi-supervised training. This includes calculating the updating rules for the QNN, implementing the training algorithm (in python) and discussing the results. The end of the chapter includes a numerical experiment to verify the model's generalisability, which is yet not achieved, however, due to the model's complexity and finding training data, which sensibly involves a graph structure.

Chapter Two

Machine learning of graph-structured data

This section introduces graphs in computer science and some applications of deep learning. It also provide basic notations for semi-supervised ML and some essential ML vocabulary. In addition, a vocabulary list can be found in the appendix (App. A).

2.1 Graphs

In Computer science, graphs are used to represent large datasets and their interior relations. A graph is a net build out of vertices and edges, and its mathematical way of representation is as follows:

Definition 1 (*graph*) *A graph G of N data points is a pair $G = (V, E)$ with a* ***vertex set***

$$V = \{v_1, \dots v_N\}$$

and an ***edge set***

$$E \subseteq \{\{v_i, v_j\} \mid v_i, v_j \in V \text{ and } v_i \neq v_j\}.$$

The vertex set only contains labels[1] of data points and does not specify the dataset, and thus a graph only

[1] Don't be confused with the labels of a training set.

shows relations of labels. One conventionally writes:

$$i \sim j \;\text{ if }\; \{v_i, v_j\} \in E,$$

which means that there is an edge between v_i and v_j. The structure of an unweighted graph can be summarised into a symmetric matrix A as:

$$[A]_{ij} = \begin{cases} 1, & \text{if } i \sim j \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

which is called the ***adjacency*** of the graph. For weighted graphs, each edge is equipped with a weight $w_{ij} = w_{ji} \in \mathbb{R}$, leading to the ***weight matrix*** W:

$$[W]_{ij} = \begin{cases} w_{ij} \in \mathbb{R}, & \text{if } i \sim j \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

The actual ***graph-structured data*** are a set of ***feature vectors*** $\vec{f}_i \in \mathbb{R}^K$ associated with a weighted or unweighted graph G, where $\mathbb{R}^K$ is the representation space for K features:

$$\textit{graph-structured data} = (\{\vec{f}_1, \ldots, \vec{f}_N\},\ G). \tag{2.3}$$

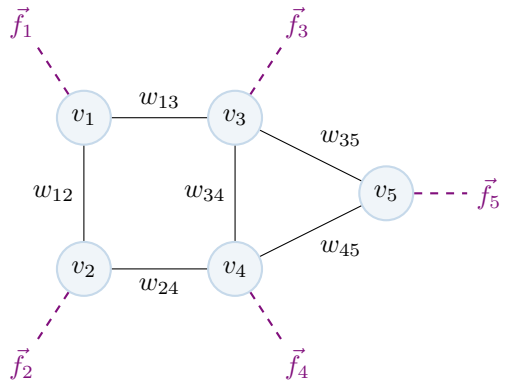
In other words, the i^{th} feature vector is assigned to the i^{th} vertex of the graph (as represented in (Figure 2.1)).

2.2 Machine learning tasks on graphs

This section considers an example of deep learning on graph-structured data, specifically a semi-supervised representation task with a partially labelled training set. The goal is to classify unlabelled data points of a test set into categories provided by the labels of the training set [57]. For this purpose, the input data[2] are mapped onto a new representation space, where each feature vector allows an identification to the given categories; however, this mapping should respect the graph's topological structure. The classification task can be viewed as preparatory work for GCNs. Some vocabulary first require clarification.

Labels $\vec{f}_i^{sv} \in \mathbb{R}^{K_{sv}}$ are assigned certain data points as targets in the training data. At the end of a training process a datapoint should match its label, and these labels specify the newest representation space, since K_{sv} can be regarded as the number of new features. It is not always possible to give the new features a useful meaning regarding the meaning of the input features. For example, if a cat has a feature vector describing its length and width, then the mapping can contain linear combinations of these attributes. ***Supervised***

[2]Set of data points without labels.



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad W = \begin{pmatrix} 0 & w_{12} & w_{13} & 0 & 0 \\ w_{12} & 0 & 0 & w_{24} & 0 \\ w_{13} & 0 & 0 & w_{34} & w_{35} \\ 0 & w_{24} & w_{34} & 0 & w_{45} \\ 0 & 0 & w_{35} & w_{45} & 0 \end{pmatrix}$$

Figure 2.1 Visualization of an example of graph-structured data consisting of 5 data points, its adjacency A and the corresponding weight matrix W.

learning generally uses fully labelled training data, unlike ***unsupervised learning***, which typically involves unlabelled data sets. ***Semi-supervised learning*** is a mixture of both, and thus the previously described semi-supervised learning uses a graph-structured dataset as in (Eq. 2.3), typically with a weighted graph, where $R \leq N$ feature vectors are labelled. This gives the ***semi-supervised graph-structured data***:

$$\textit{semi-supervised graph-structured data} = (\{(\vec{f}_1,\ \vec{f}_1^{\,sv}), \ldots, (\vec{f}_R,\ \vec{f}_R^{\,sv}), \vec{f}_{R+1}, \ldots, \vec{f}_N\},\ G). \tag{2.4}$$

The data is already sorted for simplicity, which leads to modifying the adjacency or weight matrix. An obvious method acquiring the classification of unlabelled data points is to display them in an output space $\mathbb{R}^{K_{sv}}$ so that connected features are close and labelled data points match their supervised destination (Figure 2.2). Such a task can be divided into supervised (matching the labels) and unsupervised graph-structured (learning the graph structure) parts. Hence, the data can also be divided into a supervised part:

$$\textit{supervised data} = \{(\vec{f}_1,\ \vec{f}_1^{\,sv}), \ldots, (\vec{f}_R,\ \vec{f}_R^{\,sv})\}, \tag{2.5}$$

which does not include the graph, and an unsupervised part:

$$\textit{unsupervised graph-structured data} = (\{\vec{f}_1, \ldots, \vec{f}_N\},\ G), \tag{2.6}$$

which does not include the labels. This distinction is important to formulating a cost function.

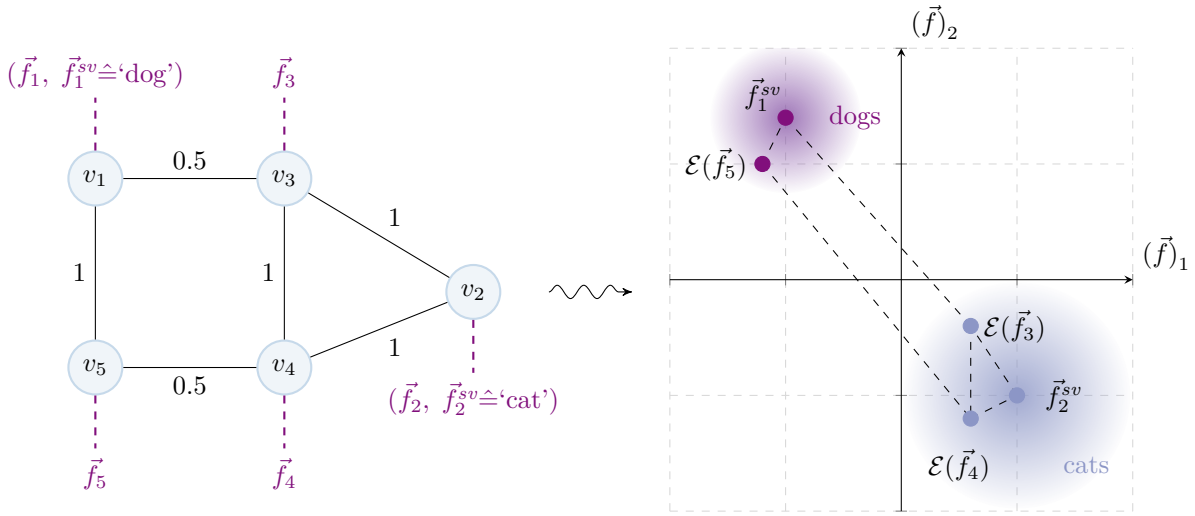


Figure 2.2 Shows the desired transition after training a neural network with a semi-supervised graph-structured dataset and a 2-feature output space. One can see the emergence of clusters of 'cats' and 'dogs' around the labels. It's important to mention, that the weights of the graph and the feature vectors influence the training output.

Solving a training task requires deriving a mathematical formulation of the task, which generally regards finding a map:

$$\mathcal{E} : \mathbb{R}^K \rightarrow \mathbb{R}^{K_{sv}} \tag{2.7}$$

which minimises or maximises a task-specific cost function. The ***cost function*** $\mathcal{L}$ is a measurement of how well the map $\mathcal{E}$ solves the training task. For the semi-supervised example above, this could be a combination of a supervised function $\mathcal{L}_{sv}$ and unsupervised function $\mathcal{L}_{usv}$:

$$\mathcal{L}_{ssv} = \mathcal{L}_{sv} + \mathcal{L}_{usv} = \sum_i ||\mathcal{E}(\vec{f}_i) - \vec{f}_i^{\,sv}||_{K_{sv}} + \sum_{i \sim j} W_{ij} ||\mathcal{E}(\vec{f}_i) - \mathcal{E}(\vec{f}_j)||_{K_{sv}}. \tag{2.8}$$

Deep learning however, uses an artificial neural network as map, which grants access to multiple set screws to control the training process. The neural network circumvents the issues in finding an exact analytic function.

2.3 Artificial neural networks

Similar to a directed graph, a ***fully connected neural network*** consists of a grid of ***neurons*** arranged in layers and weighted edges between every neuron of neighbouring layers. A single feature vector is then plugged into the left of the network, which produces an output on the right via layer transitions. The neurons carry a ***data type*** which is determined by the type of neuron and depends on the incoming data. The focus here lies on neurons, with each representing a real number or a bit (Figure 2.3).

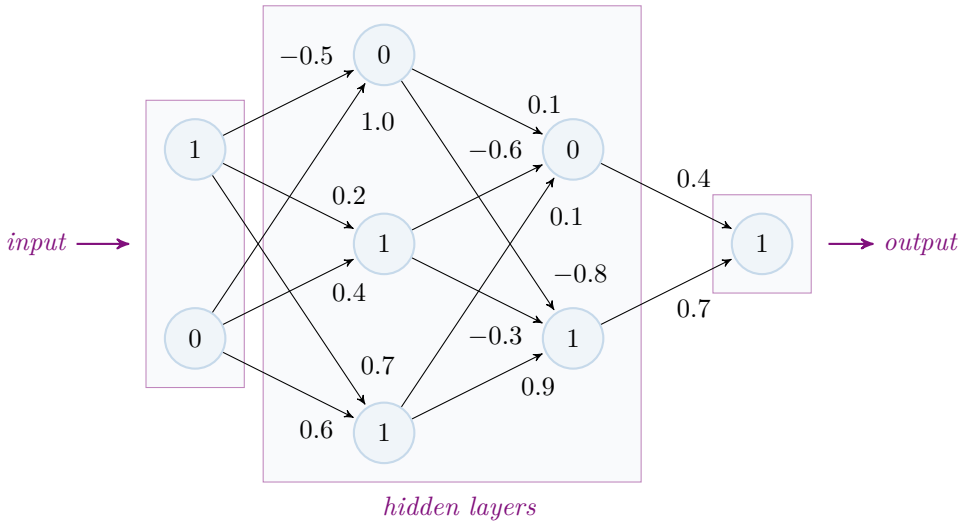


Figure 2.3 Depicts a fully connected $[2, 3, 2, 1]$-neural network, where each neuron is a bit. The layer propagation rule is given by (Eq. 2.10) with vanishing constant and the input vector is $\vec{f}^{in} = (1, 0)^{\mathrm{T}}$. The network weights are initialized randomly.

The structure of a neural network is called ***neural network architecture*** and can be written down as:

$$\textit{neural network architecture} = [n_{in}, n_1, \ldots, n_{L-1}, n_{out}]\,. \tag{2.9}$$

The architecture determines the number of neurons in each layer n_l[3] and the number of ***hidden layers*** $L - 1$. The hidden layers[4] (n_1 to n_{L-1}) mostly scale with the complexity and accuracy of the expected results, while the training task typically sets n_{in} and n_{out}. A mathematical description of the full network

[3] $n_{in} \equiv n_0$, $n_{out} \equiv n_L$.

[4] Often the output layer is included within the term 'hidden layers'.

requires assigning a vector of neuron values (depending on the data type):

$$\vec{h}^{(l)} = \begin{pmatrix} h_1^{(l)} \\ \vdots \\ h_{n_l}^{(l)} \end{pmatrix} \in \mathbb{R}^{n_l}$$

and a weight matrix:

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & \dots & w_{1n_{l-1}}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{n_l 1}^{(l)} & \dots & w_{n_l n_{l-1}}^{(l)} \end{pmatrix},$$

to every layer, where $w_{mj}^{(l)}$ corresponds to the edge between $h_j^{(l-1)}$ and $h_m^{(l)}$ (Figure 2.4).

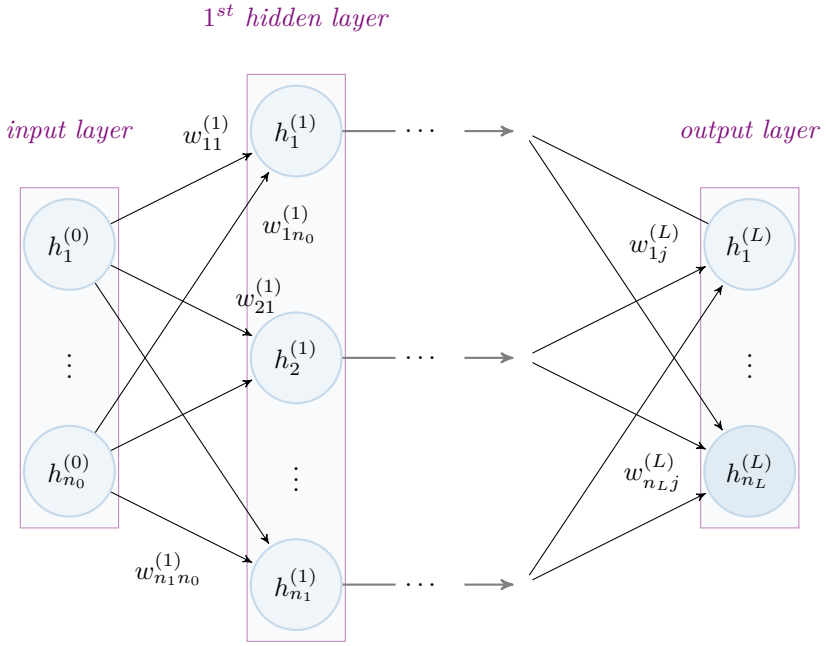


Figure 2.4 Illustrates the general structure of a neural network with corresponding weights and values of the perceptrons.

The hidden layer neurons do not take arbitrary values but are instead determined by objects called perceptrons. The j^{th} ***single-layer perceptron*** of the l^{th} layer is the map:

$$\mathcal{E}^{(l)}(\vec{h}^{(l-1)})_j = \Theta\left(\sum_i w_{ji}^{(l)} h_i^{(l-1)} + const.\right) = \begin{cases} 1, & \text{if } \sum_i w_{ji}^{(l)} h_i^{(l-1)} + const. > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

The output of perceptrons historically regarded a bit data type, and their original definition of a perceptron is made for neurons as bit. It is possible, however, to exchange the ***activation function*** (here the Heavyside step function) with another function[5] to generalise the idea of perceptrons to more adaptive neural networks. The arrangement of n_l perceptrons in each layer of the neural network gives the ***layer propagation rule***:

$$\mathcal{E}^{(l)} : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l} \text{ with } \mathcal{E}^{(l)}(\vec{h}^{(l-1)}) = \begin{pmatrix} \mathcal{E}^{(l)}(\vec{h}^{(l-1)})_1 \\ \vdots \\ \mathcal{E}^{(l)}(\vec{h}^{(l-1)})_{n_l} \end{pmatrix} \equiv \vec{h}^{(l)}. \tag{2.11}$$

The full neural network is then represented by:

$$\mathcal{E} : \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}} \text{ with } \mathcal{E}(\vec{h}^{(in)}) = \mathcal{E}^{(L)}(\dots \mathcal{E}^{(1)}(\vec{h}^{(in)}) \dots) \equiv \vec{h}^{(out)},$$

assuming that $h^{(in)}$ and the weights $W^{(l)}$ are provided[6]. In addition, the map (Eq. 2.7) states that the neural network requires the architecture:

$$\textit{neural network architecture} = [K, n_1, \dots, n_{L-1}, K_{sv}] .$$

The network thus separately transforms every feature vector of the graph-structured data into the output space determined by the given supervised data and the classical neural network remains the same for every vector (Figure 2.5).

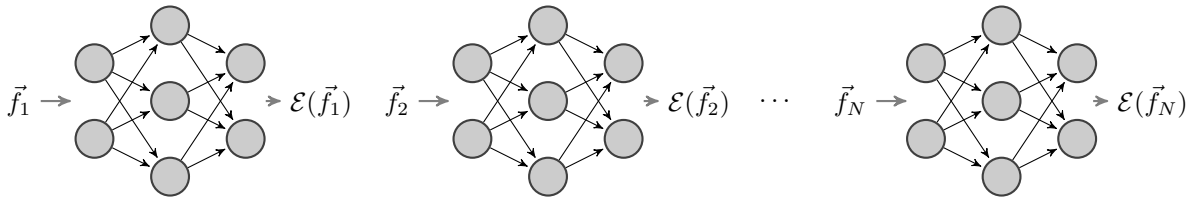


Figure 2.5 For semi-supervised machine learning on graph-structured data every feature vector is treated separately with the same neural network.

2.4 Graph convolutional networks

After defining the graph-structured data, formulating a semi-supervised learning task and introducing classical neural networks as learning instrument, the next step regards applying neural networks on (Eq. 2.8) and

[5] For example **ReLU** or **Sigmoid**.

[6] In general $W^{(l)}$ is initialized randomly and becomes determined during the training process.

portray the training process; however we skip this part and extend the idea to GCNs through introducing a mathematically compact formulation of the neural network using the neuron matrices:

$$H^{(l)} = \left(\vec{h}_1^{(l)} \quad \dots \quad \vec{h}_N^{(l)}\right),$$

where the input layer neuron matrix contains all input feature vectors:

$$H^{(0)} = \left(\vec{f}_1 \quad \dots \quad \vec{f}_N\right)$$

and the classical neural network devolves to a map:

$$\mathcal{E} : \mathbb{R}^{K\times N} \to \mathbb{R}^{K_{sv}\times N}.$$

The layer propagation rule then becomes:

$$\mathcal{E}^{(l)}(H^{(l-1)}) = \sigma\left(W^{(l)}H^{(l-1)} + const.\right), \tag{2.12}$$

which still separately handles every feature vector but executes the l^{th} transitions for all of them in a single step. Here, Θ is replaced by an arbitrary activation function σ.

Graph convolutional networks use an improved layer propagation:

$$\mathcal{E}^{(l)}(H^{(l-1)}) = \sigma\left(W^{(l)}H^{(l-1)}\tilde{D}^{-1}\tilde{A}\right), \tag{2.13}$$

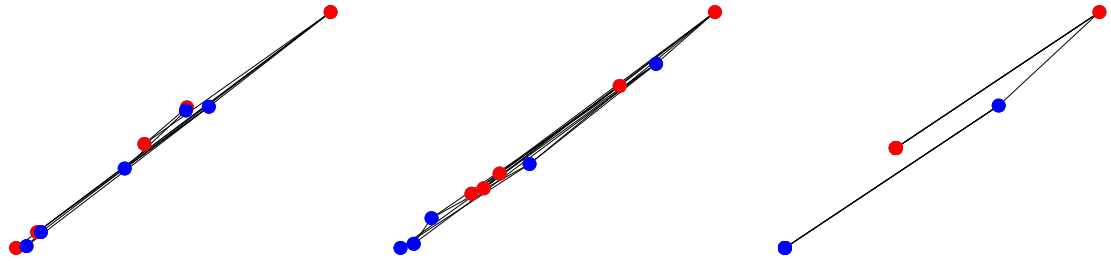
with:

$$\tilde{A} = A + \mathbb{I}_N,$$
$$\tilde{D}^{-1} = \operatorname{diag}\left(\sum_j \tilde{A}_{ij}\right).$$

where σ is an activation function, which works on every matrix entry separately. This new rule has a new propertie with respect to old one. Multiplying $W^{(l)}H^{(l-1)}$ with the self-looped adjacency $\tilde{A}$ integrates the graph structure into the layer propagation, and the connected feature vectors thus come closer together only after applying the neural network. Hence, GCNs no longer treat the feature vectors separately as before, and they accelerate the unsupervised learning of the graph structure. Here, $\tilde{D}^{-1}\tilde{A}$ normalizes $\tilde{A}$ thus the network output values do not become too large. The examples in (Figure 2.6) visualize this new attribute and shows that the GCN propagation rule clusters red to red vertices and blue to blue ones if they are more connected in the graph; however, this can fail if the random network weights are unlucky chosen.

The next step in training a ML task to a neural network is to calculate updating rules for the network weights. The training process then consists of four stages:

1. Applying the neural network $\mathcal{E}$ on the input data.



(a) 4 blue-blue, 4 red-red and 4 blue-red edges.

(b) 8 blue-blue, 8 red-red and 2 blue-red edges.

(c) 12 blue-blue, 12 red-red and 1 blue-red edges.

Figure 2.6 Shows the results of applying a $[10, 6, 2]$-GCN to three similar examples without training. The input data consits out of ten feature vectors (5 blue and 5 red) à ten features. The input feature matrix $H^{(0)}$ was initialized with the identity to exclude feature dependencies between the data points. Then the colours can be regarded as labels. The network weights are the same random weights for all examples. The graphs are varying in the number of edges between exclusively blue vertices, exclusively red vertices and between red and blue vertices. The activation function is **ReLU**. The pictures show how the layer propagation rule leads to building clusters of vertices if the number of edges between exclusively red and blue vertices increases from a) to c).

2. Calculating the cost function.

3. Updating the weights $w_{mj}^{(l)}$ via the updating rules.

4. Repeat step 1. to 3. until the cost function reaches its optimum.

But this won't be content of the thesis, instead the gained knowledge about semi-supervised deep learning on graph-structured data is transported to quantum theory in order to develop an ansatz of QML on graph-structured quantum data.

Chapter Three

Quantum machine learning of graph-structured quantum data

The previous chapter described the properties of a semi-supervised ML task on graph-structured data which are important to introduce its quantum version. The unsupervised part aimed to find new representations of an input data set to represent the graph structure. Additional labels add destinations to points in the training data and extend the unsupervised task to a semi-supervised one. Achieving a quantum analogue is aided by initially focusing on the unsupervised since the supervised part is solved in [9]. 'Finding a quantum version' concerns observing something in quantum theory whose classical limit shows all classical attributes. In this paper, however, QML regards extending ML tasks on graph-structured data to quantum data sets with a graph structure. Previous research provides no methods for finding such a quantum analogue, but it seems sensible to start with regarding the central object which manifests the borders of solvable tasks as the preferred type. It therefore make sense to introduce the QNN before researching a quantum analogue for the unsupervised task. The discussion of QNNs starts by introducing qubits, and systems of multiple qubits will act as the neurons' data type, which allows formulating the quantum theory analogue of a perceptron by introducing channels. The QNN is then described using the knowledge gained about quantum perceptrons, while all observations are reduced to cases, where each neuron is a qubit. A special case of QNN is then discussed, where each perceptron only affects a single output qubit. This specialised version of a QNN will be used for calculating the updating rules and the numerical experiments. The unsupervised learning task similar to (Eq. 2.8) is firstly discussed, including a description of the graph-structured quantum data which are compatible with the QNN (Figure 3.1).

The following method of creating a QNN of qubits is adopted from paper [9], which shows favourable results for the supervised task since they also tried to exploit quantum computing devices to execute ML tasks

on quantum data. All calculations regard a finite number of qubits but are extendable to a Hilbert space of infinite dimension without difficulty. The results are limited by the capacity of simulating a QML algorithm on classical devices and therefore unsuitable for real-world applications, but they provide an outlook for future research.

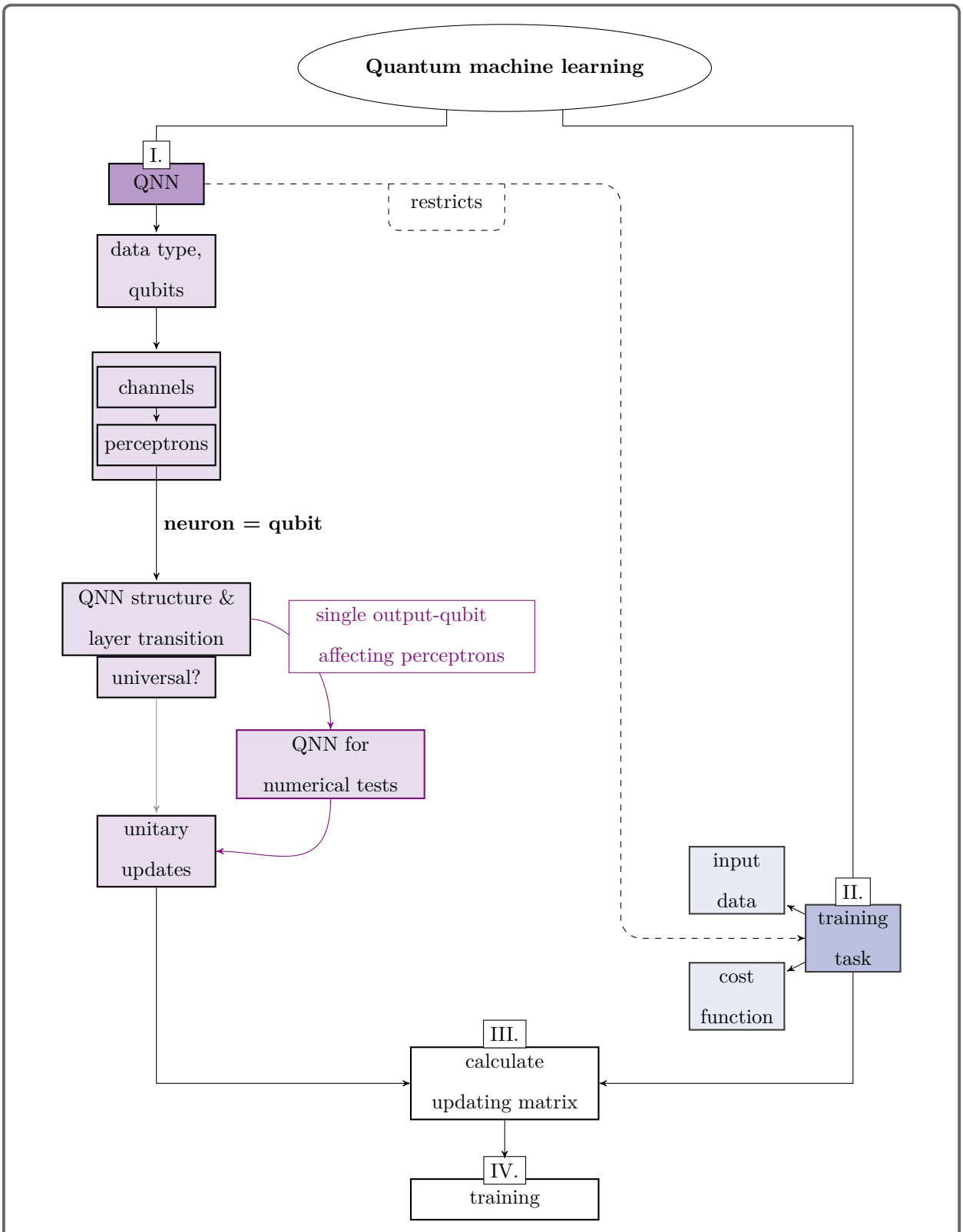


Figure 3.1 Depicts how this chapter is build up.

3.1 Quantum neural networks

An ML task is always applied to a certain type of data, such as vectors of real or complex numbers, or vectors of zeros and ones, i.e multiple bits. The ML task's chosen data type determines the neurons of the neural network, the input data and the cost functions. In classical computer science, the basic data type are bits, which provide the advantage of being easily build as hardware, for example with a flip-flop circuit, and a set of n bits can represent 2^n different states, for example 2^n different numbers (binary system). This leads to many practicable applications for bits as data type. It is thus sensible to regard something similar to bits in quantum theory to develop a new theory, hence the introduction of the quantum bit called qubit, which extends the classical characterization of a bit, representing two different states (0 and 1), to quantum theory.

Data type, qubits

The mathematical description of qubits is identical to wave functions in the Hilbert space[1] $\mathcal{H} \cong \mathbb{C}^2$. Therefore, a pure qubit state describes an element in the two dimensional complex vector space $\mathbb{C}^2$ and regards a superposition of two base states $|0\rangle$ and $|1\rangle$:

$$|\Psi\rangle = c_1 |0\rangle + c_2 |1\rangle \in \mathbb{C}^2,$$

with the common choice of basis:

$$B = \Big\{ |0\rangle \cong \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle \cong \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Big\},$$

where the complex numbers c_1 and c_2 satisfy the normalization condition of wave functions: $\langle \Psi, \Psi \rangle = 1$. Their density matrix representation:

$$\rho = |\Psi\rangle \langle \Psi| \in \mathcal{D}(\mathbb{C}^2)$$

fulfils the property $\rho^2 = \rho$. Here $\mathcal{D}(\mathcal{H})$ denotes the space of density operators on the Hilbert space $\mathcal{H}$. Mixed qubit states cannot be described by a single Ket-vector but rather are described by a linear combination of pure density operators:

$$\rho = \sum_i p_i |\Psi_i\rangle \langle \Psi_i| \in \mathcal{D}(\mathbb{C}^2),$$

where $|\Psi_i\rangle$ is a pure qubit state and $p_i \in (0, 1]$ with $\sum_i p_i = 1$ are the relative frequencies of creating the state $|\Psi_i\rangle$ during the preparation of the mixed state. A pure state regards a special case of a mixed state.

[1] See definition of a Hilbert space in (App. B).

Furthermore, mixed density operators are hermitian:

$$\rho^\dagger = \rho$$

and satisfy:

$$\rho^2 \leq \rho. \tag{3.1}$$

Rather than consisting of multiple single qubits, the training data should regard a single data point as a system of multiple qubits. The Hilbert space of multiple particles is described via the tensor product of single-particle Hilbert spaces:

$$\mathcal{H}_{multi\text{-}qubit} = \mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2.$$

A system of N qubits is then represented by the density operator:

$$\rho \in \mathcal{D}\left(\bigotimes_{i=1}^{N} \mathbb{C}^2\right) \cong \mathcal{D}(\mathbb{C}^{2N}).$$

This sets the general data type of the model as a multi-qubit system. It might be also interesting to use qudits[2] instead of qubits, but qubits provide sufficient flexibility. The question then becomes, how can information be transferred between two multi-qubit systems? The classical counterpart uses perceptrons which can learn any function from a set of input bits to a set of output bits. In this regard, a quantum perceptron should be able to perform general physical operation on a multi-qubit density operator, since an application or implementation on a quantum device is restricted to these operations. The goal is to transfer information between two physical systems obeying the rules of quantum theory. Constructing such a map is aided by introducing the abstract state/effect formalism, where physical systems are regarded as algebras (see [59] and [60]).

Abstract state/effect formalism and channels

Let $\mathcal{H}$ be a finite dimensional Hilbert space. In quantum theory a state is assigned to an operator ρ on $\mathcal{H}$ with:

$$\rho \geq 0,\ \mathrm{tr}(\rho) = 1$$

and an effect to an operator E with:

$$0 \leq E \leq \mathbb{I}\,.$$

Both objects together form the complete structure of a finite dimensional quantum theory. The definition is similar concerning algebras:

[2] One can check out qudits in quantum computing in [58].

Definition 2 (states and effects) *Let $\mathcal{A}$ be a unital $*$-algebra[a]. An **effect** e is defined as:*

$$e \in \mathcal{A} \;\; with \;\; 0 \leq e \leq \mathbb{I}\,.$$

*A **state** is defined as a linear functional on $\mathcal{A}$:*

$$f : \mathcal{A} \to \mathbb{C},$$

which is positive ($f(a) \geq 0$ for all $a \geq 0$) and unital ($f(\mathbb{I}) = 1$).

[a] All important definitions to introduce $*$-algebras can be found in (App. B.3).

Using the following theorem, shows the connection to quantum theory.

Theorem 1 *Suppose that $\mathcal{A}$ is a $*$-subalgebra of the $n \times n$ complex matrices $\mathcal{M}_n$. Then for every linear functional $f : \mathcal{A} \to \mathbb{C}$ there exists $R \in M_n(\mathbb{C})$ such that:*

$$f(A) = Tr(RA)\ \forall A \in \mathcal{A}.$$

Hence, it is possible to identify every state with an operator R. When consider the Hilbert space $\mathcal{H} = \mathbb{C}^d$ and unital $*$-algebra $\mathcal{A} = B(\mathcal{H}) = \mathcal{M}_d$ on $\mathcal{H}$, their states and effects yields quantum theory. The effects are the same as in quantum theory, and the states correspond to density matrices (Theorem 1).

Regarding the abstract states/effects formalism, the general method of transferring information from a system represented by an algebra $\mathcal{A}_1$ to another system $\mathcal{A}_2$ must preserve the states/effects structure, which means mapping states to states (Schrödinger picture) or effects to effects (Heisenberg picture). Such a map:

$$\mathcal{E} : \mathcal{A}_1 \to \mathcal{A}_2$$

is called a channel, while a general definition follows via the adjoint channel in the Heisenberg picture:

Definition 3 (channel) *A channel transferring information between two systems defined by the algebras $\mathcal{A}_1$ and $\mathcal{A}_2$ is represented by a linear completely positive map (applied to effects):*

$$\mathcal{E}^\dagger : \mathcal{A}_2 \to \mathcal{A}_1,$$

satisfying $\mathcal{E}^\dagger(\mathbb{I}) = \mathbb{I}$.

Definition 4 (completely positive) *A linear map $\mathcal{E} : \mathcal{A}_1 \to \mathcal{A}_2$ is completely positive, iff :*

$$\mathcal{E} \otimes \mathbb{I}_n : \mathcal{A}_1 \otimes \mathcal{B}(\mathbb{C}^n) \to \mathcal{A}_2 \otimes \mathcal{B}(\mathbb{C}^n)$$

is positive for all $n \in \mathbb{N}$.

Linearity preserves convex combinations of effects, while positivity and unitality map effects to effects. Furthermore, one demands a channel to be completely positive, since it should respect these conditions as being part of a larger system. This represents the most general method to transfer information between two systems while obeying the state/effect structure. Similar yields, a channel in the Schrödinger picture is a linear, completely positive, trace-preserving map (CPTP) on matrix algebras applied to states (see Theorem 1).

Using the **Stinespring theorem** leads to a method of constructing a channel.

Theorem 2 (Stinespring dilaton) *A linear map $\mathcal{E} : \mathcal{B}(\mathcal{H}_A) \to \mathcal{B}(\mathcal{H}_B)$ is completely positive if and only if there exists a Hilbert space $\mathcal{H}_E$ and a linear map $V : \mathcal{H}_A \to \mathcal{H}_B \otimes \mathcal{H}_E$, such that for all states ρ:*

$$\mathcal{E}(\rho) = \operatorname*{tr}_{\mathcal{H}_E} (V \rho V^\dagger) \tag{3.2}$$

and it is only trace-preserving iff V is unitary.

The perceptron specifies this abstract formulation to neural networks. Using channels does not necessarily mean that a QNN comprised of quantum perceptrons can approximate every function or algorithm on the input state. This issue of universality is investigated at the end of this section.

Quantum perceptron

A quantum perceptron should represent the most general physical transformation between a multi-qubit space $\mathcal{H}_{in}$ and multi-qubit space $\mathcal{H}_{out}$.

Consider an input space $\mathcal{H}_{in}$ of m qubits[3] and output space $\mathcal{H}_{out}$ of n qubits, let $\rho^{in} \in \mathcal{D}(\mathcal{H}_{in})$ be the input state. A **single-layer quantum perceptron** (Figure 3.2a) is represented by a unitary $U \in \mathcal{U}(\mathcal{H}_{in} \otimes \mathcal{H}_{out})$ acting on $m + n$ qubits and produces a n-qubit output. A method of realizing such a perceptron follows from (Theorem 2) (see [9]):

$$\rho^{out} = \operatorname*{tr}_{\mathcal{H}_{in}} \left(U \left(\rho^{in} \otimes |0, \ldots, 0\rangle_{out} \langle 0, \ldots, 0| \right) U^\dagger \right). \tag{3.3}$$

[3]Works also for qudits.

Then, the output of s **perceptrons in the same layer** (Figure 3.2b) is written as:

$$\rho^{out} = \operatorname*{tr}_{\mathcal{H}_{in}} \left(\prod_{i=s}^{1} U_i \left(\rho^{in} \otimes |0,\ldots,0\rangle_{out} \langle 0,\ldots,0| \right) \prod_{i=1}^{s} {U_i}^{\dagger} \right), \tag{3.4}$$

where $U = \prod_{i=s}^{1} U_i$ is a product of unitaries $U_i \in \mathcal{U}\left(\mathcal{H}_{in} \otimes \mathcal{H}_{out}\right)$ (generally non-commuting), each representing a perceptron, acting on the same input and output qubits. This layer transition is denoted as:

$$\rho^{out} = \mathcal{E}(\rho^{in}).$$

The implementation focuses on a special case of perceptrons: They should affect all m input qubits and only a single output qubit but still act on $m+n$ qubits. This requires n perceptrons, each affecting a different output qubit, to integrate all n output qubits into the layer transition:

$$\rho^{out} = \mathcal{E}(\rho^{in}) = \operatorname*{tr}_{\mathcal{H}_{in}} \left(\prod_{i=n}^{1} U_i \left(\rho^{in} \otimes |0,\ldots,0\rangle_{out} \langle 0,\ldots,0| \right) \prod_{i=1}^{n} {U_i}^{\dagger} \right) \tag{3.5}$$

Here, U_i should only affect the i^{th} output qubit as visualized in (Figure 3.2c), and U_i must remain an element of $\mathcal{U}\left(\mathcal{H}_{in} \otimes \mathcal{H}_{out}\right)$. Such an arbitrary unitary can be constructed by first taking the tensor product of an arbitrary unitary $\tilde{U} \in \mathcal{U}\left(\mathcal{H}_{in} \otimes \mathbb{C}^2\right)$ acting on the $m+1$ qubits and $n-1$ identities on $\mathbb{C}^2$:

$$\begin{aligned} \textit{construct} &= \tilde{U} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n-1} \\ &= \sum_{\substack{k_{1,\ldots,m+1} \\ l_{1,\ldots,m+1}}} \in \{0,1\} \tilde{u}_{k_{1,\ldots,m+1}, l_{1,\ldots,m+1}} \left(\underbrace{|k_1\rangle \langle l_1| \otimes \ldots \otimes |k_{m+1}\rangle \langle l_{m+1}| \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n-1}}_{(*)} \right). \end{aligned}$$

To assign the perceptron to the i^{th} output qubit, swap the $(m+1)^{th}$ element with the $(m+i)^{th}$ element in $(*)$:

$$\begin{aligned} U_i &= \sum_{\substack{k_{1,\ldots,m+1} \\ l_{1,\ldots,m+1}}} \in \{0,1\} \tilde{u}_{k_{1,\ldots,m+1}, l_{1,\ldots,m+1}} \\ &\quad \cdot \left(|k_1\rangle \langle l_1| \otimes \ldots \otimes |k_m\rangle \langle l_m| \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{i-1} \otimes |k_{m+1}\rangle \langle l_{m+1}| \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \right). \end{aligned} \tag{3.6}$$

In addition, one can extend the two-qubit gate **SWAP**

$$\mathbf{SWAP} = |00\rangle \langle 00| + |10\rangle \langle 01| + |01\rangle \langle 10| + |11\rangle \langle 11| \tag{3.7}$$

to K qubits, thus it swaps the i^{th} and j^{th} element of a state in $\bigotimes_{i=1}^{K} \mathbb{C}^2$:

$$\mathbf{SWAP}_K[i,j] = \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|0\rangle \langle 0|}_{i} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|0\rangle \langle 0|}_{j} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2$$

$$
\begin{aligned}
&+ \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|1\rangle\langle 0|}_{i} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|0\rangle\langle 1|}_{j} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \\
&+ \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|0\rangle\langle 1|}_{i} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|1\rangle\langle 0|}_{j} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \\
&+ \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|1\rangle\langle 1|}_{i} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \otimes \underbrace{|1\rangle\langle 1|}_{j} \otimes \mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2 \,. \qquad (3.8)
\end{aligned}
$$

Thus, (Eq. 3.6) can also be written as:

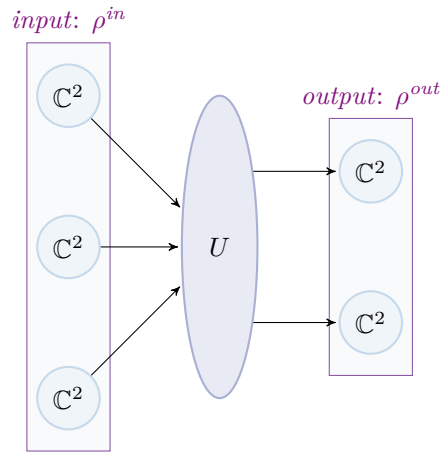
$$
U_i = \mathbf{SWAP}_{m+n}[m+1, m+i] \left(\tilde{U} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n-1} \right) \mathbf{SWAP}_{m+n}[m+1, m+i]. \qquad (3.9)
$$

Such a unitary operator leaves all output qubits unchanged, except for the i^{th} one. It is unclear, however, which channels can be implemented by these restricted perceptrons, since (Eq. 3.9) does not cover the full range of unitaries.

The next step is to derive the structure of a QNN, where each neuron is a qubit.

single perceptron

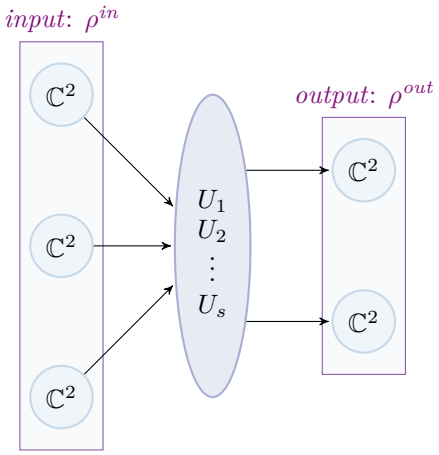
$$\rho^{out} = \operatorname*{tr}_{\mathcal{H}_{in}} \left(U \left(\rho^{in} \otimes |0,0\rangle \langle 0,0| \right) U \right)$$



(a) Sketches the structure of a single-layer perceptron with three input qubits and two output qubits.

s × perceptrons in one layer

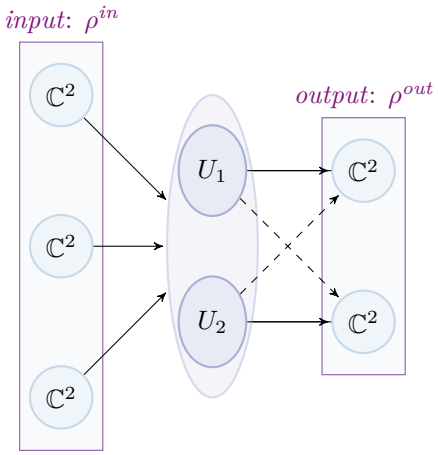
$$\rho^{out} = \operatorname*{tr}_{\mathcal{H}_{in}} \left(\prod_{i=s}^{1} U_i \left(\rho^{in} \otimes |0,0\rangle \langle 0,0| \right) \prod_{i=1}^{s} U_i{}^{\dagger} \right)$$



(b) Sketches s single-layer perceptrons in the same layer, while each acts on the same three input and two output qubits.

2 × single output perceptrons in one layer

$$\rho^{out} = \operatorname*{tr}_{\mathcal{H}_{in}} \left(U_2 U_1 \left(\rho^{in} \otimes |0,0\rangle \langle 0,0| \right) U_1{}^{\dagger} U_2{}^{\dagger} \right)$$



(c) Sketches two special perceptron, each assigned to a output qubit. We draw the dashed lines to mention, that the perceptrons still act on the full output space.

Figure 3.2 Shows the three types of layer transitions via perceptrons for three input qubits and two output qubits.

QNN structure

A QNN possesses a structuring identical to a conventional neural network via the architecture:

$$\textit{quantum neural network architecture} = [n_{in}, n_1, \ldots, n_L].$$

Thus, the network has an input layer with n_{in} neurons, an output layer with $n_{out} = n_L$ neurons and $L-1$ hidden layers with n_i neurons. Every neuron should represent a qubit, and thus n_{in} determines the number of qubits of the input state and n_L the number of output qubits.

A state consisting of n_{in} qubits is described by a mixed density operator on the tensor product of n_{in} single qubit Hilbert spaces:

$$\rho^{in} \in \mathcal{D}\big(\bigotimes_{i=1}^{n_{in}} \mathbb{C}^2\big) =: \mathcal{D}\left(\mathcal{H}_{in}\right).$$

Then, every layer l should also be a multi-particle system of n_l qubits, i.e. the l^{th} layer is represented by a mixed-state density operator:

$$\rho^{(l)} \in \mathcal{D}\big(\bigotimes_{i=1}^{n_l} \mathbb{C}^2\big) =: \mathcal{D}\left(\mathcal{H}_l\right)^{4}.$$

These $\rho^{(l)}$ are not arbitrary, but are given by the layer propagation rule applied to the input data.

QNN layer transition

The layer propagation rule applies perceptrons to the QNN's architecture. An arbitrary number of perceptrons can propagate between the $(l-1)^{th}$ and l^{th} layer of a neural network (Eq. 3.4), but a QNN restricts the number of perceptrons to the number of output qubits n_l:

$$\begin{aligned}
\mathcal{E}^{(l)}(\rho^{(l-1)}) = \rho^{(l)} &= \operatorname*{tr}_{\mathcal{H}_{l-1}} \left(U^l \left(\rho^{(l-1)} \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0| \right) {U^l}^\dagger \right) \\
&= \operatorname*{tr}_{\mathcal{H}_{l-1}} \left(\prod_{m=n_l}^{1} U_m^l \left(\rho^{(l-1)} \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0| \right) \prod_{m=1}^{n_l} {U_m^l}^\dagger \right),
\end{aligned} \tag{3.10}$$

where U_m^l acts on n_{l-1} input qubits and n_l output qubits. This expression is called the layer transition or layer propagation rule. Although the number of perceptrons relates the layer transition to the given neural network but the unitaries do not necessarily have the form of (Eq. 3.9). A full propagation through the QNN is given by:

$$\begin{aligned}
\rho^{out} = \mathcal{E}(\rho^{in}) &= \mathcal{E}^{(L)}(\mathcal{E}^{(L-1)}(\ldots \mathcal{E}^{(2)}(\mathcal{E}^{(1)}(\rho^{in}))\ldots)) \\
&= \operatorname*{tr}_{\mathcal{H}_{L-1}} \big\{ U^L \Big(\operatorname*{tr}_{\mathcal{H}_{L-2}} \big\{ \ldots \operatorname*{tr}_{\mathcal{H}_{in}} \big\{ U^1 \left(\rho^{in} \otimes |0,\ldots,0\rangle_1 \langle 0,\ldots,0| \right) {U^1}^\dagger \big\} \ldots \big\} \\
&\quad \otimes |0,\ldots,0\rangle_L \langle 0,\ldots,0| \Big) {U^L}^\dagger \big\}.
\end{aligned} \tag{3.11}$$

[4] $\rho^{out} \equiv \rho^{(L)}$, $\mathcal{H}_{out} \equiv \mathcal{H}_L$

Furthermore, a more compact form of (Eq. 3.11), can simplify the calculations for the updating rules by extending all U^l to the full Hilbert space:

$$\mathcal{H}_{full} = \bigotimes_{l=0}^{L} \mathcal{H}_l$$

via identities:

$$U^l \in \mathcal{U}(\mathcal{H}_l) \longmapsto U^l := \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{in}+\ldots+n_{l-2}} \otimes U^l \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{l+1}+\ldots+n_L} \in \mathcal{U}(\mathcal{H}_{full}) \tag{3.12}$$

which compresses (Eq. 3.11) to:

$$\begin{aligned} \mathcal{E}(\rho^{in}) &= \operatorname*{tr}_{\{\mathcal{H}_0,\ldots,\mathcal{H}_{L-1}\}} \left(\prod_{l=L}^{1} U^l (\rho^{in} \otimes |0,\ldots,0\rangle_{\{1,\ldots,L\}} \langle 0,\ldots,0|) \prod_{l=1}^{L} U^{l\dagger} \right) \\ &= \operatorname*{tr}_{\{in,\ hidden\}} \left(\prod_{l=L}^{1} U^l (\rho^{in} \otimes |0,\ldots,0\rangle_{\{hidden,\ out\}} \langle 0,\ldots,0|) \prod_{l=1}^{L} U^{l\dagger} \right). \end{aligned} \tag{3.13}$$

A classical neural network is universal, i.e., it can approximate every function up to arbitrary accuracy, and it would be advisable that the QNN has a similar property. In quantum theory universality is defined as follows:

Definition 5 *A set of quantum gates*[a] *is said to be universal if a quantum circuit, build from these gates, can approximate every unitary operation to arbitrary accuracy.*

[a] Quantum gates are the elemental operations on qubits executed by a quantum device.

For more information check [60].

Hence, to show that the QNN is universal, it must be an arbitrary circuit built from a set of universal quantum gates. Consider the case of an unitary V acting on two qubits:

$$V \rho^{in} V^\dagger = \rho^{out}$$

The appropriate QNN should be a $[2,2]$-QNN with two perceptrons U_1^1 and U_2^1, see (Figure 3.3). Choosing U_1^1 as:

$$U_1^1 = \mathbf{SWAP}_4 [1,3] (V \otimes \mathbb{I}_2 \otimes \mathbb{I}_2) \tag{3.14}$$

and U_2^1 as:

$$U_2^1 = \mathbf{SWAP}_4 [2,4], \tag{3.15}$$

leads to the following layer transition:

$$\begin{aligned} \mathcal{E}(\rho^{in}) = \operatorname*{tr}_{\mathcal{H}_{in}} \Big(& \mathbf{SWAP}_4 [2,4] \, \mathbf{SWAP}_4 [1,3] (V \otimes \mathbb{I}_2 \otimes \mathbb{I}_2) \left(\rho^{in} \otimes |0,0\rangle \langle 0,0| \right) (V \otimes \mathbb{I}_2 \otimes \mathbb{I}_2)^\dagger \\ & \cdot \mathbf{SWAP}_4 [1,3] \, \mathbf{SWAP}_4 [2,4] \Big) \end{aligned}$$

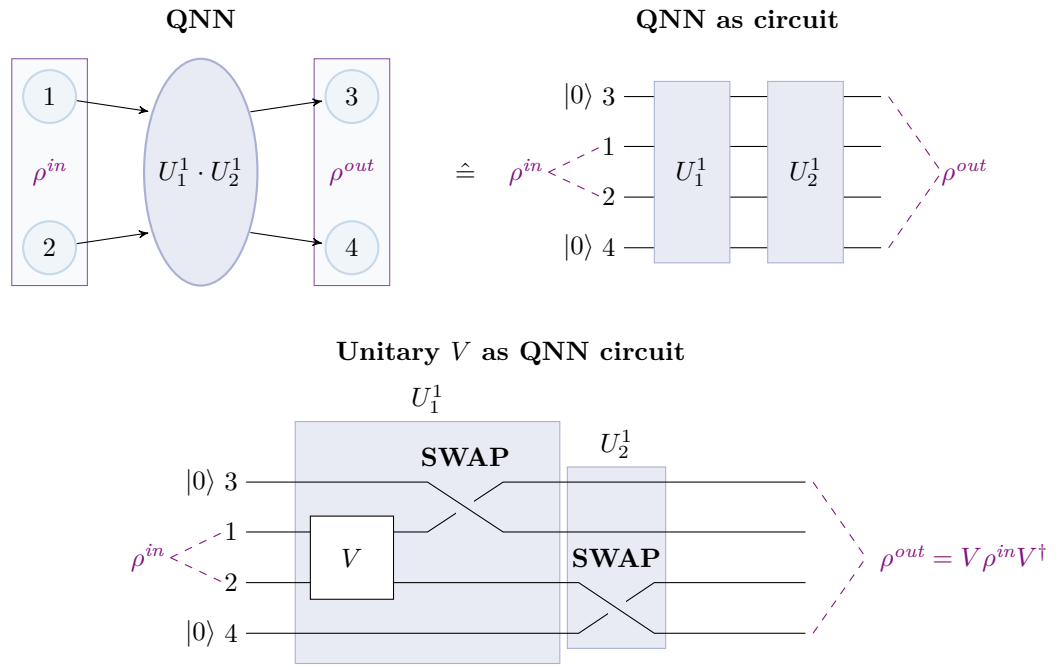


Figure 3.3 Shows a $[2,2]$-QNN with two perceptrons and its circuit representation. Choosing U_1^1 and U_2^1 as in (Eq. 3.14) and (Eq. 3.15) leads to the circuit below, which acts as an unitary V on a two qubit state.

$$
\begin{aligned}
&= \operatorname{tr}_{\mathcal{H}_{in}} \left(\mathbf{SWAP}_4 [2,4] \, \mathbf{SWAP}_4 [1,3] \left(V \rho^{in} V^\dagger \otimes |0,0\rangle \langle 0,0| \, \mathbf{SWAP}_4 [1,3] \, \mathbf{SWAP}_4 [2,4] \right) \right) \\
&= \operatorname{tr}_{\mathcal{H}_{in}} \left(|0,0\rangle \langle 0,0| \otimes V \rho^{in} V^\dagger \right) \\
&= V \rho^{in} V^\dagger = \rho^{out}.
\end{aligned}
$$

The QNN can thus represent a unitary V. Since **SWAP** and V are two-qubit quantum gates, the QNN above represents a circuit built from these gates (Figure 3.3), and two-qubit quantum gates are universal [60], then all QNNs built from these two in and out qubit blocks are universal. Similar one can proceed for unitaries acting on more then two qubits. Therefore, the QNN is said to be universal, which results from constructing perceptrons as channels.

Unitary updates

During the training process, a training step number s dependency is added to all objects: $\mathcal{E}_s$, $U_m^l(s)$. Furthermore the network unitaries are updated via:

$$U_m^l(s) \in \mathcal{U}\left(\mathcal{H}_{l-1} \otimes \mathcal{H}_l\right) \longmapsto U_m^l(s+\epsilon) = e^{i\epsilon K_m^l(s)} U_m^l(s),$$

where s marks the current training step and ϵ the step size. The **updating matrix** $K_m^l(s)$ should be a hermitian operator acting on $\mathcal{H}_{l-1} \otimes \mathcal{H}_l$, and thus $e^{i\epsilon K_m^l(s)}$ is unitary and $U_m^l(s+\epsilon)$ remains as well.

QNN for numerical tests

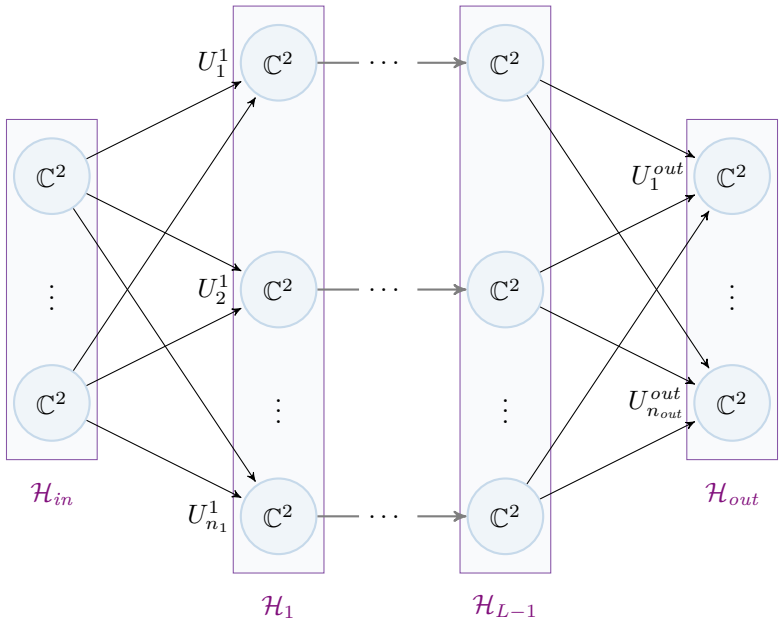


Figure 3.4 Represents the structure of a quantum neural network for arbitrary architectures with single output qubit affecting perceptrons.

The implementation's focus lies on perceptrons which only affect a single output qubit, and thus $U_m^l \in \mathcal{D}\left(\mathcal{H}_{l-1} \otimes \mathcal{H}_l\right)$ takes the form:

$$U_m^l = \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \left(\tilde{U} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l - 1} \right) \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \,,$$
$$\tilde{U} \in \mathcal{U}\left(\mathcal{H}_{l-1} \otimes \mathbb{C}^2\right), \tag{3.16}$$

see (Eq. 3.9). This special kind of QNN is represented in (Figure 3.4), where every qubit in the hidden and output layer can be identified with a unitary. To maintain this structure during the training process the updating matrices K_m^l must be regarded as a hermitian operator which only affects the m^{th} output qubit. Such a network allows a more efficient training process, since every hidden qubit can be separately adjust by modifying its assigned unitary.

3.2 Quantum version of the unsupervised learning task

This section describes the development of the quantum analogue of the unsupervised ML task as in (Eq. 2.8) on unweighted graph-structured data. Because the training task is applied to the given QNN from the previous section, the input data are restricted to multi-qubit states. A quantum version of graph-structured data could be:

$$\textit{graph structured quantum data} = (\{\rho_1^{in}, \ldots, \rho_N^{in}\}, G) \tag{3.17}$$

with:

$$\rho_i^{in} \in \mathcal{D}(\mathbb{C}^{2K}).$$

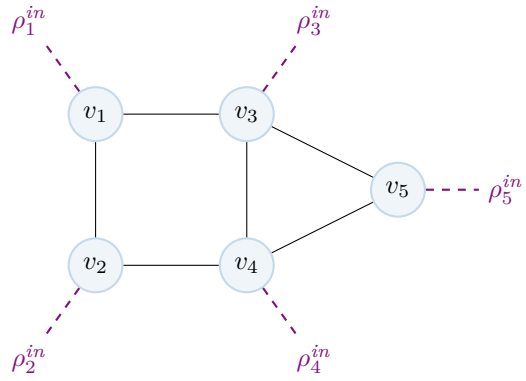


Figure 3.5 Visualizes graph-structured quantum data consisting of 5 data points. The graph remain classical, but the feature vectors are replaced by quantum states.

The vector of K features are exchanged with a K-qubits density operator. This type of data set is called graph-structured quantum data, which has a classical part of the graph structure and a quantum part of the states/density operators (multi-particle systems of qubits) (Figure 3.5). The abstract training task is to map connected data points close to each other, which requires comparing two given density operators regarding their closeness. Two common closeness-measures of a Hilbert space are as follows:
Two states ρ_1 and ρ_2 of a system described by the Hilbert space $\mathcal{H}$ are said to be close if the **Fidelity** [16]:

$$\mathcal{F} : \mathcal{D}(\mathcal{H}) \times \mathcal{D}(\mathcal{H}) \to [0, 1]$$
$$\mathcal{F}(\rho_1, \rho_2) = \operatorname{tr} \sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}$$

is close to 1. If one of the states is pure, the Fidelity simplifies to

$$\mathcal{F}(|\Psi_1\rangle\langle\Psi_1|, \rho_2) = \operatorname{tr}(|\Psi_1\rangle\langle\Psi_1|\rho_2) = \langle\Psi_1, \rho_2\Psi_1\rangle = \langle\Psi_1|\rho_2|\Psi_1\rangle, \tag{3.18}$$

which is the expectation value of the operator ρ_2 applied to a system in state $|\Psi_1\rangle$. The Fidelity leads to complicated calculations in cases of mixed states, however, and thus it is useful to introduce a second distance measure. Two states are close according to the **Hilbert-Schmidt distance** if:

$$d_{HS} : \mathcal{D}(\mathcal{H}) \times \mathcal{D}(\mathcal{H}) \to [0,1],$$
$$d_{HS}(\rho_1, \rho_2) = \mathrm{tr}\left((\rho_1 - \rho_2)(\rho_1 - \rho_2)^\dagger\right) \equiv \mathrm{tr}\left((\rho_1 - \rho_2)^2\right)$$

is close to 0. Although the Fidelity measurement is generally used, the Hilbert-Schmidt distance involves simpler calculations when both states are mixed. With this in mind the unsupervised learning task can be formulated via the following cost function:

$$\mathcal{L}_{usv}(s) = \sum_{i\sim j}[A]_{ij} d_{HS}(\mathcal{E}_s(\rho_i^{in}), \mathcal{E}_s(\rho_j^{in})) = \sum_{i\sim j}[A]_{ij}\,\mathrm{tr}\left\{(\mathcal{E}_s(\rho_i^{in}) - \mathcal{E}_s(\rho_j^{in}))^2\right\}, \tag{3.19}$$

which must be minimised towards 0. There is currently no quantum version of a GCN, and hence each data point will be treated separately and only the network updates will involve the graph structure.

A closer examination of the QNN will lead to the optimal map $\mathcal{E}$. If the network unitaries can be decomposed into two unitaries: one on $\mathcal{H}_{in}$ and the other on the remaining space (for example the identity in $\mathcal{H}_{full}$), this leads to:

$$\begin{aligned}
\mathcal{E}_s(\rho_i^{in}) &= \operatorname*{tr}_{\{in,\ hidden\}}\Big\{\prod_{l=L}^{1} U_{in}^{(l)}(s)\otimes U_{hidden,\ out}^{(l)}(s)(\rho_i^{in}\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|) \\
&\quad\cdot\prod_{l=1}^{L} {U_{in}^{(l)}}^\dagger(s)\otimes {U_{hidden,\ out}^{(l)}}^\dagger(s)\Big\} \\
&= \operatorname*{tr}_{\{in,\ hidden\}}\Big\{\Big(\prod_{l=L}^{1} U_{in}^{(l)}(s)\rho_i^{in}\prod_{l=1}^{L} {U_{in}^{(l)}}^\dagger(s)\Big)\otimes\Big(\prod_{l=L}^{1} U_{hidden,\ out}^{(l)}(s)\,|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0| \\
&\quad\cdot\prod_{l=1}^{L} {U_{hidden,\ out}^{(l)}}^\dagger(s)\Big)\Big\} \\
&= \underbrace{\operatorname*{tr}_{\{in\}}\Big\{\prod_{l=L}^{1} U_{in}^{(l)}(s)\rho_i^{in}\prod_{l=1}^{L} {U_{in}^{(l)}}^\dagger(s)\Big\}}_{=1}\operatorname*{tr}_{\{hidden\}}\Big\{\prod_{l=L}^{1} U_{hidden,\ out}^{(l)}(s)\,|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0| \\
&\quad\cdot\prod_{l=1}^{L} {U_{hidden,\ out}^{(l)}}^\dagger(s)\Big\} \\
&= \operatorname*{tr}_{\{hidden\}}\Big\{\prod_{l=L}^{1} U_{hidden,\ out}^{(l)}(s)(|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|)\prod_{l=1}^{L} {U_{hidden,\ out}^{(l)}}^\dagger(s)\Big\},
\end{aligned} \tag{3.20}$$

which is independent of ρ_i^{in}, thus the cost function vanishes to zero. Such a network maps all inputs onto the same output, which is a trivial solution for the unsupervised training task. A more favourable solution is a map which can distinguish between connected sets of vertices. We will see in the numerics that a QNN is

able to find such a map. Nevertheless, the next step is to calculate the updating rule and check whether the QNN and its implementation also train for random initial unitaries, since the semi-supervised task should provide solutions which are not trivial.

3.3 Updating rule for the unsupervised learning task

The goal of the updating rule is to find $K_m^l(s)$, and thus every training step leads to decreasing of (Eq. 3.19). As mentioned in Section 3.1, the unitaries should only affect a single output qubit in the network implementation and therefore the updating matrix $K_m^l(s)$ should have the same property as in (Eq. 3.9). If the network leads to an increase in one training step, then control is lost, preventing predictions concerning a large amount of training steps. This problem is solved by beginning with the derivative of the cost function:

$$\frac{d\mathcal{L}_{usv}(s)}{ds} = \lim_{\epsilon \to 0} \frac{\mathcal{L}_{usv}(s+\epsilon) - \mathcal{L}_{usv}(s)}{\epsilon} \tag{3.21}$$

and finding $K_m^l(s)$, and thus $d_s\mathcal{L}_{usv}(s)$ becomes negative for all s. Beforehand, it is useful to expend $\mathcal{E}_{s+\epsilon}(\rho_i^{in})$ in first order ϵ, so ϵ needs to be small enough:

$$\begin{aligned}
\mathcal{E}_{s+\epsilon}(\rho_i^{in}) \quad = \quad & \operatorname*{tr}_{\{in,\ hidden\}} \Big\{ e^{i\epsilon K_{n_L}^L(s)} U_{n_L}^L(s) \dots e^{i\epsilon K_1^1(s)} U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) \\
& \cdot {U_1^1}^\dagger(s)\, e^{-i\epsilon K_1^1(s)} \dots {U_{n_L}^L}^\dagger(s)\, e^{-i\epsilon K_{n_L}^L(s)} \Big\}.
\end{aligned}$$

With $e^{i\epsilon K_j^i(s)} = \mathbb{I} + i\epsilon K_j^i(s) + \mathcal{O}\left(\epsilon^2\right)$ follows:

$$\begin{aligned}
\mathcal{E}_{s+\epsilon}(\rho_i^{in}) \quad = \quad & \operatorname*{tr}_{\{in,\ hidden\}} \left\{ U_{n_L}^L(s) \dots U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) \dots {U_{n_L}^L}^\dagger(s) \right\} \\
& + i\epsilon \operatorname*{tr}_{\{in,\ hidden\}} \left\{ K_{n_L}^L(s) U_{n_L}^L(s) \dots U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) \dots {U_{n_L}^L}^\dagger(s) \right. \\
& - U_{n_L}^L(s) \dots U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) \dots {U_{n_L}^L}^\dagger(s) K_{n_L}^L(s) \\
& + \dots \\
& + U_{n_L}^L(s) \dots K_1^1(s) U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) \dots {U_{n_L}^L}^\dagger(s) \\
& \left. - U_{n_L}^L(s) \dots U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) K_1^1(s) \dots {U_{n_L}^L}^\dagger(s) \right\} + \mathcal{O}(\epsilon^2) \\
= \quad & \rho_i^{out}(s) \\
& + i\epsilon \operatorname*{tr}_{\{in,\ hidden\}} \left\{ \left[K_{n_L}^L(s), U_{n_L}^L(s) \dots U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) \dots {U_{n_L}^L}^\dagger(s) \right] \right. \\
& + \dots \\
& \left. + U_{n_L}^L(s) \dots U_2^1(s) \left[K_1^1(s), U_1^1(s) \left(\rho_i^{in} \otimes |0,\dots,0\rangle_{\{hidden,\ out\}} \langle 0,\dots,0| \right) {U_1^1}^\dagger(s) \right] {U_2^1}^\dagger(s) \dots {U_{n_L}^L}^\dagger(s) \right\} \\
& + \mathcal{O}(\epsilon^2).
\end{aligned}$$

For simplicity we set:

$$
\begin{aligned}
X(s,\rho_i^{in}) \;:=\; & \operatorname*{tr}_{\{in,\ hidden\}} \Big\{ \Big[K_{n_L}^L(s), U_{n_L}^L(s)\dots U_1^1(s)\Big(\rho_i^{in}\otimes|0,\dots,0\rangle_{\{hidden,\ out\}}\langle 0,\dots,0|\Big)U_1^{1\dagger}(s)\dots U_{n_L}^{L\ \dagger}(s)\Big] \\
& + \dots \\
& + U_{n_L}^L(s)\dots U_2^1(s)\Big[K_1^1(s), U_1^1(s)\Big(\rho_i^{in}\otimes|0,\dots,0\rangle_{\{hidden,\ out\}}\langle 0,\dots,0|\Big)U_1^{1\dagger}(s)\Big]U_2^{1\dagger}(s)\dots U_{n_L}^{L\ \dagger}(s)\Big\},
\end{aligned}
\tag{3.22}
$$

thus the previous expression becomes:

$$
\mathcal{E}_{s+\epsilon}(\rho_i^{in}) = \rho_i^{out}(s) + i\epsilon X(s,\rho_i^{in}) + \mathcal{O}(\epsilon^2). \tag{3.23}
$$

Hence, the derivative holds as:

$$
\begin{aligned}
\frac{d\mathcal{L}_{usv}(s)}{ds} &= \lim_{\epsilon\to 0}\frac{\mathcal{L}_{usv}(s+\epsilon)-\mathcal{L}_{usv}(s)}{\epsilon} \\
&= \lim_{\epsilon\to 0}\frac{\sum\limits_{i\sim j}[A]_{ij}\operatorname{tr}\{\big(\mathcal{E}_{s+\epsilon}(\rho_i^{in})-\mathcal{E}_{s+\epsilon}(\rho_j^{in})\big)^2\}-\sum\limits_{i\sim j}[A]_{ij}\operatorname{tr}\{\big(\mathcal{E}_{s}(\rho_i^{in})-\mathcal{E}_{s}(\rho_j^{in})\big)^2\}}{\epsilon} \\
&= \lim_{\epsilon\to 0}\sum_{i\sim j}\frac{[A]_{ij}}{\epsilon}\operatorname{tr}\{\big(\mathcal{E}_{s+\epsilon}(\rho_i^{in})-\mathcal{E}_{s+\epsilon}(\rho_j^{in})\big)^2-\big(\mathcal{E}_{s}(\rho_i^{in})-\mathcal{E}_{s}(\rho_j^{in})\big)^2\} \\
&= \lim_{\epsilon\to 0}\sum_{i\sim j}\frac{[A]_{ij}}{\epsilon}\operatorname{tr}\{\rho_i^{out}(s)^2-\rho_i^{out}(s)\rho_j^{out}(s)-\rho_j^{out}(s)\rho_i^{out}(s)+\rho_j^{out}(s)^2 \\
&\quad + i\epsilon\big(\rho_i^{out}(s)X(s,\rho_i^{in})-\rho_i^{out}(s)X(s,\rho_j^{in})+X(s,\rho_i^{in})\rho_i^{out}(s)-X(s,\rho_i^{in})\rho_j^{out}(s) \\
&\quad + \rho_j^{out}(s)X(s,\rho_j^{in})-\rho_j^{out}(s)X(s,\rho_i^{in})+X(s,\rho_j^{in})\rho_j^{out}(s)-X(s,\rho_j^{in})\rho_i^{out}(s)\big)+\mathcal{O}\left(\epsilon^2\right) \\
&\quad - \rho_i^{out}(s)^2+\rho_i^{out}(s)\rho_j^{out}(s)+\rho_j^{out}(s)\rho_i^{out}(s)+\rho_j^{out}(s)^2\} \\
&= i\sum_{i\sim j}[A]_{ij}\operatorname{tr}\{2\rho_i^{out}(s)X(s,\rho_i^{in})-2\rho_i^{out}(s)X(s,\rho_j^{in})+2\rho_j^{out}(s)X(s,\rho_j^{in})-2\rho_j^{out}(s)X(s,\rho_i^{in})\} \\
&\quad + \lim_{\epsilon\to 0}\mathcal{O}\left(\epsilon\right) \\
&= 2i\sum_{i\sim j}[A]_{ij}\operatorname{tr}\left\{\ \big(\rho_i^{out}(s)-\rho_j^{out}(s)\big)\big(X(s,\rho_i^{in})-X(s,\rho_j^{in})\big)\ \right\}
\end{aligned}
$$

After inserting the expression for $X(s,\rho_i^{in})$, one can expand $\big(\rho_i^{out}(s)-\rho_j^{out}(s)\big)$ with identities $\mathbb{I}_{\{in,\ hidden\}}$ to the full Hilbert space and pull out the partial trace of $X(s,\rho_i^{in})$:

$$
\begin{aligned}
\frac{d\mathcal{L}_{usv}(s)}{ds} &= 2i\sum_{i\sim j}[A]_{ij}\operatorname{tr}\left\{\ \big(\mathbb{I}_{\{in,\ hidden\}}\otimes\big(\rho_i^{out}(s)-\rho_j^{out}(s)\big)\big)\right. \\
&\quad \cdot\Big(\Big[K_{n_L}^L(s), U_{n_L}^L(s)\dots U_1^1(s)\Big(\big(\rho_i^{in}-\rho_j^{in}\big)\otimes|0,\dots,0\rangle_{\{hidden,\ out\}}\langle 0,\dots,0|\Big)U_1^{1\dagger}(s)\dots U_{n_L}^{L\ \dagger}(s)\Big] \\
&\quad + \dots \\
&\quad + U_{n_L}^L(s)\dots U_2^1(s)\Big[K_1^1(s), U_1^1(s)\Big(\big(\rho_i^{in}-\rho_j^{in}\big)\otimes|0,\dots,0\rangle_{\{hidden,\ out\}}\langle 0,\dots,0|\Big)U_1^{1\dagger}(s)\Big] \\
&\quad \left.\cdot\, U_2^{1\dagger}(s)\dots U_{n_L}^{L\ \dagger}(s)\Big)\right\} \\
&= 2i\sum_{i\sim j}[A]_{ij}\operatorname{tr}\Big\{\Big[U_{n_L}^L(s)\dots U_1^1(s)\Big(\big(\rho_i^{in}-\rho_j^{in}\big)\otimes|0,\dots,0\rangle_{\{hidden,\ out\}}\langle 0,\dots,0|\Big)U_1^{1\dagger}(s)\dots U_{n_L}^{L\ \dagger}(s),
\end{aligned}
$$

$$\left(\mathbb{I}_{\{in,\ hidden\}} \otimes \left(\rho_i^{out}(s) - \rho_j^{out}(s)\right)\right)\Big] K_{n_L}^L(s) + \ldots$$
$$+ \Big[U_1^1(s)\left(\left(\rho_i^{in} - \rho_j^{in}\right) \otimes |0,\ldots,0\rangle_{\{hidden,\ out\}} \langle 0,\ldots,0|\right) U_1^{1\dagger}(s), U_2^{1\dagger}(s) \ldots U_{n_L}^{L\ \dagger}(s)$$
$$\cdot \left(\mathbb{I}_{\{in,\ hidden\}} \otimes \left(\rho_i^{out}(s) - \rho_j^{out}(s)\right)\right) U_{n_L}^L(s) \ldots U_2^1(s)\Big] K_1^1(s)\Big\}.$$

This compresses to:

$$\frac{d\mathcal{L}_{usv}(s)}{ds} = 2i \sum_{i\sim j} [A]_{ij} \operatorname{tr}\left\{M_{n_L\{i,j\}}^L(s) K_{n_L}^L(s) + \ldots + M_{1\{i,j\}}^1(s) K_1^1(s)\right\} \tag{3.24}$$

with:

$$M_{m\{i,j\}}^l(s) = \Big[U_m^l(s) \ldots U_1^1(s)\left(\left(\rho_i^{in} - \rho_j^{in}\right) \otimes |0,\ldots,0\rangle_{\{hidden,\ out\}} \langle 0,\ldots,0|\right) U_1^{1\dagger}(s) \ldots U_m^{l\ \dagger}(s), \tag{3.25}$$
$$U_{m+1}^{l\ \dagger}(s) \ldots U_{n_L}^{L\ \dagger}(s) \left(\mathbb{I}_{\{in,\ hidden\}} \otimes \left(\rho_i^{out}(s) - \rho_j^{out}(s)\right)\right) U_{n_L}^L(s) \ldots U_{m+1}^l(s)\Big].$$

It then becomes useful to expand the hermitian matrices $K_m^l(s)$ via Pauli matrices:

$$K_m^l(s) = \sum_{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta} K_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}^l(s) \left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta}\right), \tag{3.26}$$

while:

$$K_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}^l(s) \in \mathbb{R}$$

and $\left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta}\right)$ is meant as an extended version to the full Hilbert space via identities (see (Eq. 3.16) and (Eq. 3.12)):

$$\left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta}\right) := \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{in}+\ldots+n_{l-2}} \otimes \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m]$$
$$\cdot \left(\left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta}\right) \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l-1}\right) \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{l+1}+\ldots+n_L}$$

$$= \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{in}+\ldots+n_{l-2}} \otimes \sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{m-1} \otimes \sigma^{\beta} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l-m+n_{l+1}+\ldots+n_L}. \tag{3.27}$$

Then, $K_m^l(s)$ affects a single output qubit and $d_s\mathcal{L}_{usv}(s)$ is linear in the coefficients $K_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}^l(s)$. One would expect that setting $K_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}^l(s)$ to zero directly leads to the minimum, but then $\mathcal{L}_{usv}(s)$ is constant in s and the global minimum cannot be achieved for all possible network unitaries. This implies a need to stepwise approach the minimum, but this process can be accelerated by minimising $d_s\mathcal{L}_{usv}(s)$ with respect to the coefficients. Since this is a linear function in $K_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}^l(s)$ up to order ϵ, one would expect the minimum at infinity, but a finite solution for the training process is desirable. A large range of solutions have a negative $d_s\mathcal{L}_{usv}(s)$ for some $K_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}^l(s)$ depending on $M_{m\{i,j\}}^l(s)$, and thus 'the best'

finite solutions must be found to ensure that the training process terminates, i.e., reaches $d_s\mathcal{L}_{usv}(s) = 0$ for s going to infinity. This is achievable by introducing a Lagrange parameter λ and minimising the derivative via the Lagrange method with the next condition:

$$\sum_{\alpha_1,\ldots,\beta} K^l_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}(s)^2 = 0.$$

Hence, one must solve:

$$\min_{K^a_{b,\mu_1,\ldots,\mu_{n_{l-1}},\nu}} \left(\frac{d\mathcal{L}_{usv}(s)}{ds} - \lambda \sum_{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta} K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s)^2 \right) \tag{3.28}$$

$$\Leftrightarrow \partial_{K^a_{b,\mu,\ldots,\mu_{n_{l-1}},\nu}} \left(\frac{d\mathcal{L}_{usv}(s)}{ds} - \lambda \sum_{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta} K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s)^2 \right) \stackrel{!}{=} 0. \tag{3.29}$$

With (Eq. 3.24) follows:

$$\partial_{K^a_{b,\mu_1,\ldots,\mu_{n_{l-1}},\nu}} \Big(2i \sum_{i\sim j} [A]_{ij} \operatorname{tr}\left\{ M^L_{n_L\{i,j\}}(s) K^L_{n_L}(s) + \ldots + M^1_{1\{i,j\}}(s) K^1_1(s) \right\} - \lambda \sum_{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta} K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s)^2 \Big)$$
$$\stackrel{!}{=} 0.$$

Inserting (Eq. 3.26) for $K^l_m(s)$ and using the linearity of the trace leads to:

$$\partial_{K^a_{b,\mu_1,\ldots,\mu_{n_{l-1}},\nu}} \Big(2i \sum_{i\sim j} [A]_{ij} \Big(\sum_{\alpha_1,\ldots,\alpha_{n_{L-1}},\beta} K^L_{n_L,\alpha_1\ldots,\alpha_{n_{L-1}},\beta}(s) \operatorname{tr}\left\{ M^L_{n_L\{i,j\}}(s) \left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{L-1}}} \otimes \sigma^\beta\right) \right\} + \ldots$$
$$+ \sum_{\alpha_1,\ldots,\alpha_{n_{in}},\beta} K^1_{1,\alpha_1\ldots,\alpha_{n_{in}},\beta}(s) \operatorname{tr}\left\{ M^1_{1\{i,j\}}(s) \left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{in}}} \otimes \sigma^\beta\right) \right\} \Big) - \lambda \sum_{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta} K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s)^2 \Big)$$
$$\stackrel{!}{=} 0.$$

Taking the derivative with respect to $K^a_{b,\mu_1,\ldots,\mu_{n_{l-1}},\nu}$ results in:

$$2i \sum_{i\sim j} [A]_{ij} K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s) \operatorname{tr}\left\{ M^l_{m\{i,j\}}(s) \left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^\beta\right) \right\} - \lambda 2 K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s) \stackrel{!}{=} 0.$$

It is then possible to divide the trace into a part, which only acts on $\mathcal{H}_{n_{l-1}}$ and the m^{th} qubit in the l^{th} layer denoted as $\operatorname{tr}_{\{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta\}}$, and another part acting on the extant Hilbert space, denoted as tr_{rest}:

$$\begin{aligned} K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta} &= \frac{i}{\lambda} \sum_{i\sim j} [A]_{ij} \operatorname{tr}\left\{ M^l_{m\{i,j\}}(s) \left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^\beta\right) \right\} \\ &= \frac{i}{\lambda} \sum_{i\sim j} [A]_{ij} \operatorname{tr}_{\{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta\}} \left\{ \operatorname{tr}_{rest} \left\{ M^l_{m\{i,j\}}(s) \left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^\beta\right) \right\} \right\} \end{aligned}$$

This term is simplified by writing out $\left(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^\beta\right)$ as in (Eq. 3.27):

$$K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta} = \frac{i}{\lambda} \sum_{i\sim j} [A]_{ij} \operatorname{tr}_{\{\alpha_1,\ldots,\alpha_{n_{l-1}},\beta\}} \left\{ \operatorname{tr}_{rest} \left\{ M^l_{m\{i,j\}}(s) \Big(\underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{in}+\ldots+n_{l-2}} \right.\right.$$

$$\otimes\, \sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{m-1} \otimes \sigma^{\beta} \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l - m + n_{l+1} + \ldots + n_L} \Big) \Big\} \Big\}$$

$$= \frac{i}{\lambda} \sum_{i \sim j} [A]_{ij} \operatorname*{tr}_{\{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta\}} \Big\{ \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\} \big(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta} \big) \Big\}.$$

One can then expand $\mathrm{tr}_{rest} \big\{ M^l_{m\{i,j\}} \big\}$ in terms of Pauli matrices:

$$\begin{aligned}
K^l_{m,\alpha_1 \ldots, \alpha_{n_{l-1}}, \beta} &= \frac{i}{\lambda} \sum_{i \sim j} [A]_{ij} \operatorname*{tr}_{\{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta\}} \Big\{ \sum_{\mu_1, \ldots, \mu_{n_{l-1}}, \nu} \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\}_{\mu_1, \ldots, \mu_{n_{l-1}}, \nu} \big(\sigma^{\mu_1} \otimes \ldots \otimes \sigma^{\mu_{n_{l-1}}} \otimes \sigma^{\nu} \big) \\
&\qquad \cdot \big(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta} \big) \Big\} \\
&= \frac{i}{\lambda} \sum_{i \sim j} [A]_{ij} \sum_{\mu_1, \ldots, \mu_{n_{l-1}}, \nu} \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\}_{\mu_1, \ldots, \mu_{n_{l-1}}, \nu} \\
&\quad \cdot \operatorname*{tr}_{\{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta\}} \big\{ \sigma^{\mu_1} \sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\mu_{n_{l-1}}} \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\nu} \sigma^{\beta} \big\} \\
&= \frac{i}{\lambda} \sum_{i \sim j} [A]_{ij} \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\}_{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta} 2^{n_{l-1}+1} \\
&= \frac{2^{n_{l-1}+1} i}{\lambda} \sum_{i \sim j} [A]_{ij} \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\}_{\{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta\}}.
\end{aligned}$$

This yields the full matrix:

$$\begin{aligned}
K^l_m(s) &= \frac{2^{n_{l-1}+1} i}{\lambda} \sum_{i \sim j} [A]_{ij} \sum_{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta} \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\}_{\{\alpha_1, \ldots, \alpha_{n_{l-1}}, \beta\}} \big(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta} \big) \\
&= \frac{2^{n_{l-1}+1} i}{\lambda} \sum_{i \sim j} [A]_{ij} \operatorname*{tr}_{rest} \big\{ M^l_{m\{i,j\}}(s) \big\}. \qquad (3.30)
\end{aligned}$$

It is again necessary to extend this matrix to the full Hilbert space to be convenient with earlier expressions:

$$\begin{aligned}
K^l_m(s) &:= \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{in} + \ldots + n_{l-2}} \otimes \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \left(K^l_m(s) \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l - 1} \right) \qquad (3.31) \\
&\quad \cdot \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_{l+1} + \ldots + n_L}.
\end{aligned}$$

It must be clarified how to choose λ thus $d_s \mathcal{L}_{usv}(s) < 0$ by regarding (Eq. 3.24) once again as a real, linear function in $K^l_{m,\alpha_1 \ldots, \alpha_{n_{l-1}}, \beta}$:

$$\frac{d\mathcal{L}_{usv}(s)}{ds} = \Big(\sum_{\alpha_1, \ldots, \alpha_{n_{L-1}}, \beta} a^L_{n_L, \alpha_1, \ldots, \alpha_{n_{L-1}}, \beta} K^L_{n_L, \alpha_1 \ldots, \alpha_{n_{L-1}}, \beta}(s) + \ldots + \sum_{\alpha_1, \ldots, \alpha_{n_{in}}, \beta} a^1_{1, \alpha_1, \ldots, \alpha_{n_{in}}, \beta} K^1_{1, \alpha_1 \ldots, \alpha_{n_{in}}, \beta}(s) \Big) \quad (3.32)$$

with:

$$a^l_{m, \alpha_1, \ldots, \alpha_{n_{l-1}}, \beta} = 2i \sum_{i \sim j} [A]_{ij} \operatorname{tr} \{ M^l_{m\{i,j\}}(s) \big(\sigma^{\alpha_1} \otimes \ldots \otimes \sigma^{\alpha_{n_{l-1}}} \otimes \sigma^{\beta} \big) \}$$

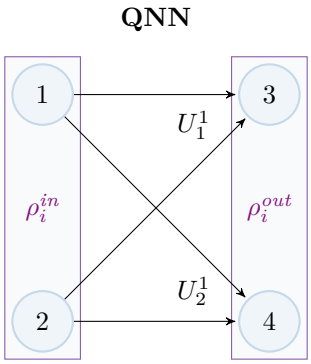


Figure 3.6 Visualizes a $[2,2]$-QNN. Here the neurons are numbered consecutively.

and applying the same Lagrange formalism leads to:

$$K^l_{m,\alpha_1\ldots,\alpha_{n_{l-1}},\beta}(s) = \frac{a^l_{m,\alpha_1,\ldots,\alpha_{n_{l-1}},\beta}}{2\lambda}.$$

Inserting this solution into (Eq. 3.32) gives:

$$\frac{d\mathcal{L}_{usv}(s)}{ds} = \frac{1}{2\lambda}\Big(\sum_{\alpha_1,\ldots,\alpha_{n_{L-1}},\beta}\Big(a^L_{n_L,\alpha_1,\ldots,\alpha_{n_{L-1}},\beta}\Big)^2 + \ldots + \sum_{\alpha_1,\ldots,\alpha_{n_{in}},\beta}\Big(a^1_{1,\alpha_1,\ldots,\alpha_{n_{in}},\beta}\Big)^2\Big),$$

which is smaller than zero if $\lambda < 0$.

Example $[2,2]$-QNN

A simple example can display the full formalism above. Consider a $[2,2]$-QNN, see (Figure 3.6) with an arbitrary graph. The neurons in (Figure 3.6) are consecutively numbered, which allows identifying the partial trace with the associated Hilbert space.

The QNN consists of an input state:

$$\rho_i^{in} \in \mathcal{D}(\mathbb{C}^4),$$

with two features, an output state:

$$\rho_i^{out} \in \mathcal{D}(\mathbb{C}^4),$$

with two features and two perceptrons, represented by the unitaries:

$$\begin{aligned} U_1^1 &= \tilde{U}_1^1 \otimes \mathbb{I}_2 \in \mathcal{U}\left(\mathcal{H}_{full}\right) \text{ with } \tilde{U}_1^1 \in \mathcal{U}\left(\mathcal{H}_{in} \otimes \mathbb{C}^2\right) \\ U_2^1 &= \mathbf{SWAP}_4[3,4]\left(\tilde{U}_2^1 \otimes \mathbb{I}_2\right)\mathbf{SWAP}_4[3,4] \in \mathcal{U}\left(\mathcal{H}_{full}\right) \text{ with } \tilde{U}_2^1 \in \mathcal{U}\left(\mathcal{H}_{in} \otimes \mathbb{C}^2\right). \end{aligned}$$

A full network transition is given by:

$$\mathcal{E}_s(\rho_i^{in}) = \rho_i^{out}(s) = \underset{\mathcal{H}_{in}}{\text{tr}} \left\{ U_2^1(s) U_1^1(s) \left(\rho_i^{in} \otimes |00\rangle \langle 00| \right) {U_1^1}^\dagger(s) {U_2^1}^\dagger(s) \right\}.$$

This leads to the following expression for the derivative of the cost function (see (Eq. 3.24)):

$$\frac{d\mathcal{L}_{usv}(s)}{ds} = 2i \sum_{i \sim j} [A]_{ij} \, \text{tr} \left\{ M^1_{2\{i,j\}}(s) K_2^1(s) + M^1_{1\{i,j\}}(s) K_1^1(s) \right\}$$

with:

$$\begin{aligned} M^1_{1\{i,j\}}(s) &= \left[U_1^1(s) \left(\left(\rho_i^{in} - \rho_j^{in} \right) \otimes |00\rangle \langle 00| \right) {U_1^1}^\dagger(s), {U_2^1}^\dagger(s) \left(\mathbb{I}_2 \otimes \mathbb{I}_2 \otimes \left(\rho_i^{out}(s) - \rho_j^{out}(s) \right) \right) U_2^1(s) \right] \\ M^1_{2\{i,j\}}(s) &= \left[U_2^1(s) U_1^1(s) \left(\left(\rho_i^{in} - \rho_j^{in} \right) \otimes |00\rangle \langle 00| \right) {U_1^1}^\dagger(s) {U_2^1}^\dagger(s), \mathbb{I}_2 \otimes \mathbb{I}_2 \otimes \left(\rho_i^{out}(s) - \rho_j^{out}(s) \right) \right]. \end{aligned}$$

The updating matrices become (Eq. 3.30):

$$\begin{aligned} K_1^1(s) &= \frac{2^3 i}{\lambda} \sum_{i \sim j} [A]_{ij} \underset{4}{\text{tr}} \left\{ M^1_{1\{i,j\}}(s) \right\} \\ K_2^1(s) &= \frac{2^3 i}{\lambda} \sum_{i \sim j} [A]_{ij} \underset{3}{\text{tr}} \left\{ M^1_{2\{i,j\}}(s) \right\}, \end{aligned}$$

where tr_3 is the trace over the third qubit and tr_4 over the fourth.

The next step is to implement an algorithm which can simulate the QNN on a classical or quantum device.

3.4 Implementation of the unsupervised training process

This section aims to formulate an efficient implementation of the QNN and an algorithm which calculates all updating matrices for unsupervised training the QNN. Conventions need to be chosen, regarding how to represent the input data and QNN and how to generate the training data.

First, the input data regard a 1-dimensional list of density matrices on $\mathbb{C}^{2K}$:

$$\textit{initialStates} = \left[\rho_1^{in}, \ldots, \rho_N^{in} \right]$$

The QNN architecture will be represented by a 1-dimensional list as well:

$$\textit{qnnArch} = [n_{in}, \ldots, n_{out}]$$

A suitable choice for the initial network unitaries is a 2-dimensional list with an empty zeroth element, to ensure that the indices begin at the correct position:

$$\textit{initialUnitaries} = \left[[\,], \left[U_1^1, \ldots, U_{n_1}^1 \right], \ldots, \left[U_1^L, \ldots, U_{n_L}^L \right] \right]$$

Furthermore, the full-Hilbert-space-version (Eq. 3.13) of our QNN will not be used to store memory, but instead $U_m^l \in \mathcal{U}(\mathcal{H}_{l-1} \otimes \mathcal{H}_l)$ is chosen as in (Eq. 3.10), and therefore, it will prove useful to store each $\mathcal{E}^{(l)}(\rho^{(l-1)})$. For test purposes, the training uses random mixed initial states created from a normal distribution, as well as the random initial unitaries (App. C.1.2).

The training algorithm can be divided into three sub algorithms, where the first (**Feedforward**) implements the neural network and grants access to a list containing layer-wise outputs for every input state:

Algorithm 1 Feedforward

procedure FEEDFORWARD(*initialStates*, *qnnArch*, *currentUnitaries*)
    storedStates ← [];
    for i in $\{1, \ldots, N\}$ **do**
        state ← ρ_i^{in}
        layerwiseList ← [*state*]
        for l in $\{1, \ldots, L\}$ **do**
            state ← $\mathcal{E}^{(l)}(state) = \mathrm{tr}_{\mathcal{H}_{l-1}}\left(\prod_{m=n_l}^{1} U_m^l \left(state \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0|\right) \prod_{m=1}^{n_l} U_m^{l\,\dagger}\right)$
            layerwiseList **append** *state*
        storedStates **append** *layerwiseList*;
        return *storedStates*;

Then, the *storedStates* list takes the form:

$$storedStates = [[\rho_1^{in}, \rho_1^{(1)}, \ldots, \rho_1^{out}], \ldots, [\rho_N^{in}, \rho_N^{(1)}, \ldots, \rho_N^{out}]].$$

The second algorithm (**unsupervised update matrix**) calculates the updating matrix for the given indices (l, m) using the *storedStates* list from the previous algorithm:

Algorithm 2 Unsupervised update matrix

1: **procedure** UsvUpdatematrix(*storedStates*, *qnnArch*, *currentUnitaries*, λ, ϵ, A, l, m)

2: $K_m^l \leftarrow 0$;

3: **for** i in $\{1, \ldots, N\}$ **do**

4: **for** j in $\{1, \ldots, N\}$ **do**

5: **if** $[A]_{ij} \neq 0$ **then**

6: $M^l_{m\{i,j\}} \leftarrow \Bigg[\prod_{\alpha=m}^{1} U_\alpha^l \left(\left(\rho_i^{(l-1)} - \rho_j^{(l-1)}\right) \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0|\right) \prod_{\alpha=1}^{m} U_\alpha^{l\,\dagger},$

7: $\prod_{\alpha=m+1}^{n_l} U_\alpha^l \left(\mathbb{I}_{l-1} \otimes \left(\sigma^{(l)}(\rho_i^{out}) - \sigma^{(l)}(\rho_j^{out})\right)\right) \prod_{\alpha=n_l}^{m+1} U_\alpha^{l\,\dagger}\Bigg]$;

8: $K_m^l += [A]_{ij} M^l_{m\{i,j\}}$;

9: $K_m^l \leftarrow \frac{2^{n_{l-1}+1} i}{\lambda} \operatorname{tr}_{rest}\left\{K_m^l\right\}$;

10: $K_m^l \leftarrow \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \left(K_m^l \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l - 1}\right) \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m]$;

11: $update \leftarrow e^{i\epsilon K_m^l} U_m^l$;

12: return *update*;

In line 6, parts of tr_{rest} are already applied, and thus $\rho_i^{(l-1)}$ can be used as well as the adjoint channel $\mathcal{G}$. The following expression illustrates this connection:

$$
\begin{aligned}
\operatorname*{tr}_{rest}\left\{M^l_{m\{i,j\}}\right\} &= \operatorname*{tr}_{rest}\Big\{\Big[U_m^l \ldots U_1^1 \left(\left(\rho_i^{in} - \rho_j^{in}\right) \otimes |0,\ldots,0\rangle_{\{hidden,\ out\}} \langle 0,\ldots,0|\right) U_1^{1\,\dagger} \ldots U_m^{l\,\dagger},\\
&\quad U_{m+1}^{l\,\dagger} \ldots U_{n_L}^{L\,\dagger} \left(\mathbb{I}_{\{in,\ hidden\}} \otimes \left(\rho_i^{out} - \rho_j^{out}\right)\right) U_{n_L}^L \ldots U_{m+1}^l\Big]\Big\}\\
&= \operatorname*{tr}_{\{rest, l+1,\ldots,L\}}\Big\{\Big[\prod_{\alpha=m}^{1} U_\alpha^l \operatorname*{tr}_{\{1,\ldots,l-2\}}\Big\{U_{n_{l-1}}^{l-1} \ldots U_1^1 \left(\left(\rho_i^{in} - \rho_j^{in}\right) \otimes |0,\ldots,0\rangle_{\{1,\ldots,l-1\}} \langle 0,\ldots,0|\right) \cdot\\
&\quad U_1^{1\dagger} \ldots U_{n_{l-1}}^{l-1\,\dagger}\Big\} \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0| \prod_{\alpha=1}^{m} U_\alpha^{l\,\dagger} \otimes |0,\ldots,0\rangle_{\{l+1,\ldots,L\}} \langle 0,\ldots,0|,\\
&\quad U_{m+1}^{l\,\dagger} \ldots U_{n_L}^{L\,\dagger} \left(\mathbb{I}_{\{l-1,\ldots,L-1\}} \otimes \left(\rho_i^{out} - \rho_j^{out}\right)\right) U_{n_L}^L \ldots U_{m+1}^l\Big]\Big\}\\
&= \operatorname*{tr}_{rest}\Big\{\Big[\prod_{\alpha=m}^{1} U_\alpha^l \left(\left(\rho_i^{(l-1)} - \rho_j^{(l-1)}\right) \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0|\right) \prod_{\alpha=1}^{m} U_\alpha^{l\,\dagger}, \prod_{\alpha=m+1}^{n_l} U_\alpha^l \Big(\mathbb{I}_{l-1} \otimes\\
&\quad \operatorname*{tr}_{\{l+1,\ldots,L\}}\Big\{\mathbb{I}_l \otimes |0,\ldots,0\rangle_{\{l+1,\ldots,L\}} \langle 0,\ldots,0| U_{n_{l+1}}^{l+1\,\dagger} \ldots U_{n_L}^{L\,\dagger} \cdot\\
&\quad \left(\mathbb{I}_{\{l,\ldots,L-1\}} \otimes \left(\rho_i^{out} - \rho_j^{out}\right)\right) U_{n_L}^L \ldots U_{n_{l+1}}^{l+1}\Big\}\Big) \prod_{\alpha=n_l}^{m+1} U_\alpha^{l\,\dagger}\Big]\Big\}\\
&= \operatorname*{tr}_{rest}\Big\{\Big[\prod_{\alpha=m}^{1} U_\alpha^l \left(\left(\rho_i^{(l-1)} - \rho_j^{(l-1)}\right) \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0|\right) \prod_{\alpha=1}^{m} U_\alpha^{l\,\dagger},\\
&\quad \prod_{\alpha=m+1}^{n_l} U_\alpha^l \left(\mathbb{I}_{l-1} \otimes \left(\sigma^{(l)}(\rho_i^{out}(s)) - \sigma^{(l)}(\rho_j^{out}(s))\right)\right) \prod_{\alpha=n_l}^{m+1} U_\alpha^{l\,\dagger}\Big]\Big\}
\end{aligned}
$$

Here, $\sigma^{(l)}$ can be derived via the adjoined channel $\mathcal{G}^{(l)}$ of $\mathcal{E}^{(l)}$ using the **Kraus theorem**.

Theorem 3 (Kraus theorem) *Consider two Hilbert spaces $\mathcal{H}_{in}$ of dimension n and $\mathcal{H}_{out}$ of dimension m. Let $\mathcal{E}$ be a channel from $\mathcal{H}_{in}$ to $\mathcal{H}_{out}$. Then there are matrices:*

$$A_\alpha \text{ with } \alpha \in \{1, \ldots, nm\}$$

mapping from $\mathcal{H}_{in}$ to $\mathcal{H}_{out}$ and satisfying:

$$\sum_\alpha A_\alpha^\dagger A_\alpha \leq 1,$$

thus:

$$\mathcal{E}(\rho) = \sum_\alpha A_\alpha \rho A_\alpha^\dagger.$$

A_α are called Kraus operators.

The adjoint $\mathcal{G}$ of this channel is defined as:

$$\mathcal{G}(\rho) = \sum_\alpha A_\alpha^\dagger \rho A_\alpha,$$

where ρ must be a density operator on $\mathcal{H}_{out}$. Consider the Kraus representation of the channel between the $(l-1)^{th}$ and l^{th} layers, applied to an arbitrary operator:

$$\mathcal{E}^{(l)}(X^{(l-1)}) = \sum_\alpha A_\alpha X^{(l-1)} A_\alpha^\dagger$$

and its adjoint channel:

$$\mathcal{G}^{(l)}(X^{(l)}) = \sum_\alpha A_\alpha^\dagger X^{(l)} A_\alpha. \tag{3.33}$$

Furthermore, let $|m\rangle, |n\rangle$ be arbitrary vectors in $\mathcal{H}_{l-1}$ and $|i\rangle, |j\rangle$ in $\mathcal{H}_l$. One then determines A_α via:

$$\begin{aligned} \langle i| \mathcal{E}^{(l)}(|m\rangle\langle n|)|j\rangle &= \langle i| \operatorname*{tr}_{\mathcal{H}_{l-1}} \left(U^l (|m\rangle\langle n| \otimes |0, \ldots, 0\rangle_l \langle 0, \ldots, 0|) U^{l\dagger} \right) |j\rangle \\ &= \langle i| \sum_\alpha \langle \alpha| U^l |m, 0, \ldots, 0\rangle_l \langle n, 0, \ldots, 0| U^{l\dagger} |\alpha\rangle |j\rangle, \end{aligned}$$

thus:

$$\langle i| A_\alpha |j\rangle = \langle i| \langle \alpha| U^l |m, 0, \ldots, 0\rangle_l,$$

where $|\alpha\rangle$ is an orthonormal basis of $\mathcal{H}_{l-1}$. This yields the following expression for the adjoint channel:

$$\begin{aligned} \langle m| \mathcal{G}^{(l)}(|i\rangle\langle j|)|n\rangle &= \sum_\alpha \langle m| A_\alpha^\dagger |i\rangle \langle j| A_\alpha |n\rangle \\ &= \sum_\alpha \langle m, 0, \ldots, 0| U^{l\dagger} |\alpha, i\rangle \langle \alpha, j| U^l |n, 0, \ldots, 0\rangle \end{aligned}$$

$$
\begin{aligned}
&= \langle m,0,\ldots,0|\, {U^l}^\dagger\, \mathbb{I}_{l-1}\otimes|i\rangle\,\langle j|\, U^l\,|n,0,\ldots,0\rangle \\
&= \langle m|\operatorname*{tr}_l\left(\mathbb{I}_{l-1}\otimes|0,\ldots,0\rangle_l\,\langle 0,\ldots,0|\, {U^l}^\dagger\,\mathbb{I}_{l-1}\otimes|i\rangle\,\langle j|\, U^l\right)|n\rangle\,.
\end{aligned}
$$

It is then possible to write (Eq. 3.33) as:

$$
\mathcal{G}_s^{(l)}(X^{(l)}) = \operatorname*{tr}_l\left(\mathbb{I}_{l-1}\otimes|0,\ldots,0\rangle_l\,\langle 0,\ldots,0|\, {U^l}^\dagger(s)\,\mathbb{I}_{l-1}\otimes X^{(l)}U^l(s)\right). \tag{3.34}
$$

This allows expressing $\sigma^{(l)}$ via the adjoined channel:

$$
\begin{aligned}
\sigma_s^{(l)}(\rho_i^{out}) &= \operatorname*{tr}_{\{l+1,\ldots,L\}}\big\{\,\mathbb{I}_l\otimes|0,\ldots,0\rangle_{\{l+1,\ldots,L\}}\,\langle 0,\ldots,0|\, {U_{n_{l+1}}^{l+1}}^\dagger\ldots{U_{n_L}^L}^\dagger\left(\mathbb{I}_{\{l,\ldots,L-1\}}\otimes\rho_i^{out}\right) \\
&\quad\cdot U_{n_L}^L\ldots U_{n_{l+1}}^{l+1}\big\} \\
&= \mathcal{G}_s^{(l+1)}(\mathcal{G}_s^{(l+2)}(\ldots\mathcal{G}_s^{(L)}(\rho_i^{out}))),
\end{aligned}
$$

and using U_m^l on $\mathcal{H}_{l-1}\otimes\mathcal{H}_l$ during the full calculation, meaning they do not have to be extended to the full Hilbert space.

The third sub algorithm (**network training**) combines the previous ones to a training process:

Algorithm 3 Network training

```
1:  procedure NetworkTraining(initialStates, qnnArch, initialUnitaries, λ, ϵ, A, trainingSteps)
2:      currentUnitaries ← initialUnitaries;
3:      s ← 0;
4:      storedStates ← Feedforward(initialStates, qnnArch, currentUnitaries);
5:      costFunction ← [L_usv(storedStates)];
6:      for s in {1, ..., trainingSteps} do
7:          for l in {1, ..., L} do
8:              for , in {1, ..., n_l} do
9:                  currentUnitaries[l][m] ← Updatematrix(storedStates, qnnArch, currentUnitaries, λ, ϵ, A, l, m);
10:         storedStates ← Feedforward(initialStates, qnnArch, currentUnitaries);
11:         costFunction append L_usv(storedStates);
12:     return costFunction;
```

A full python implementation is given in (App. C.1.3) and (App. C.1.4). These implementations and preparatory work enable observing results for the unsupervised training and then continuing to more difficult tasks.

3.5 Results for unsupervised training

This section includes the results of unsupervised training a QNN for the task in (Eq. 3.19) with different graphs and parameters, beginning with overviewing a simple type of graph and a small network. The graph is described by the adjacency:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \tag{3.35}$$

which regards a chain of three vertices. The training data should consist of two features, and thus one of the simplest possible QNN (Figure 3.7) is:

$$qnnArch = [2, 2]\,. \tag{3.36}$$

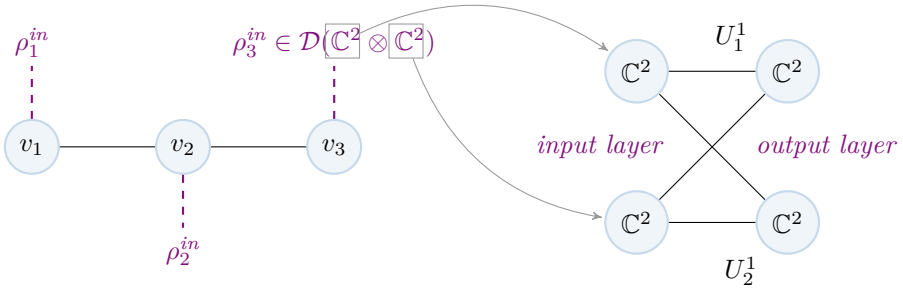


Figure 3.7 Visualizes the graph with adjacency (Eq. 3.35) and a QNN with equal in-and-output space (Eq. 3.36).

In the results, only ϵ is varied while λ is set to -1. (Figure 3.8) shows that the implemented algorithm trains the neural network close to zero within 40 steps. If ϵ is sufficiently small, the cost function can be expected to reach zero within infinite training steps.

However, if ϵ is too large, $d_s\mathcal{L}_{usv}$ does not remain constantly negative and control of the training process becomes lost (Figure 3.9). Additional results for different graph types and QNNs are found in the appendix (Figure D.1) and (Figure D.2).

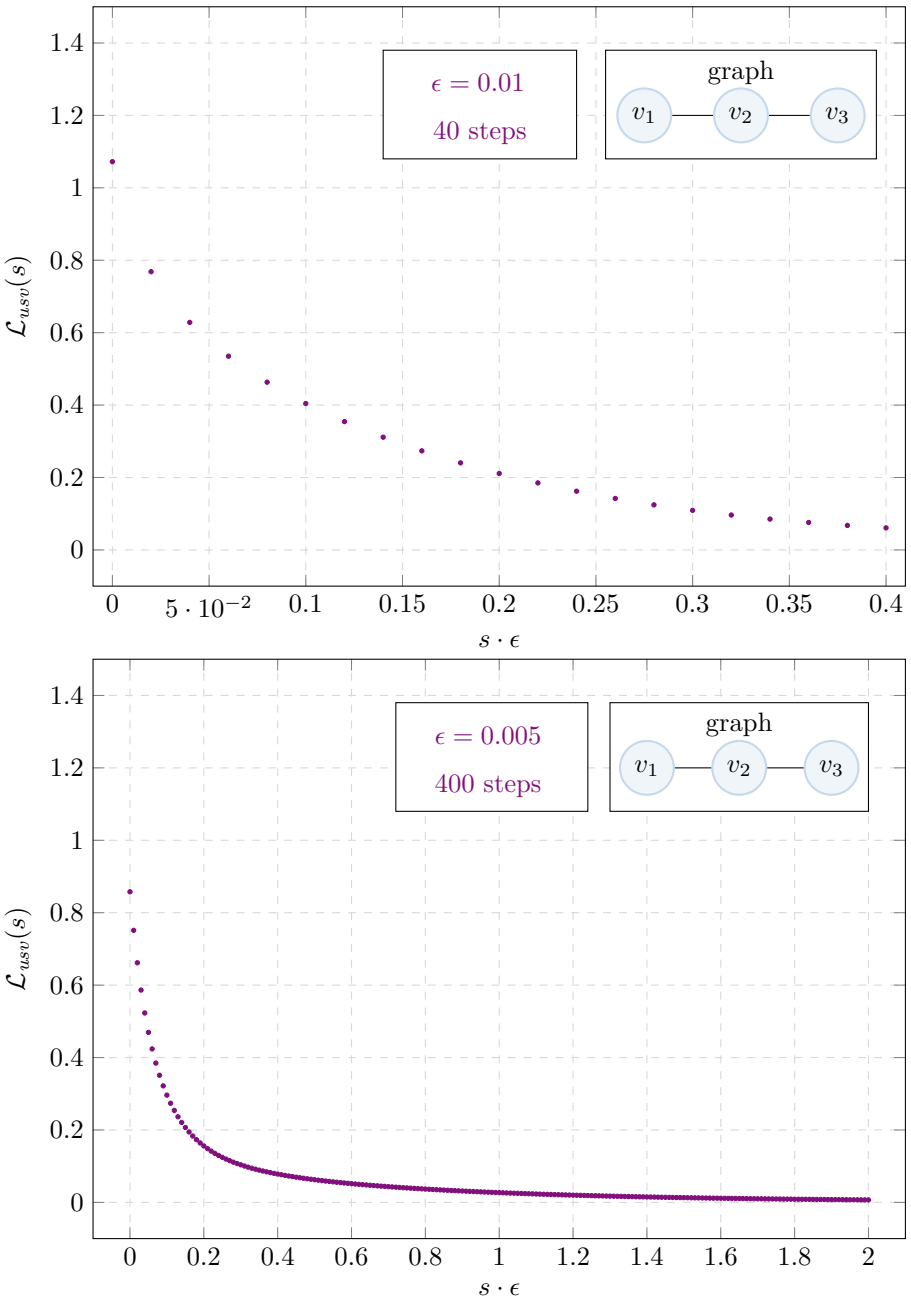


Figure 3.8 Shows the cost function values for unsupervised training (Eq. 3.19) of a QNN (Eq. 3.36) with two feature graph-structured quantum data ((Eq. 3.35), (Figure 3.2)) for two different values of ϵ.

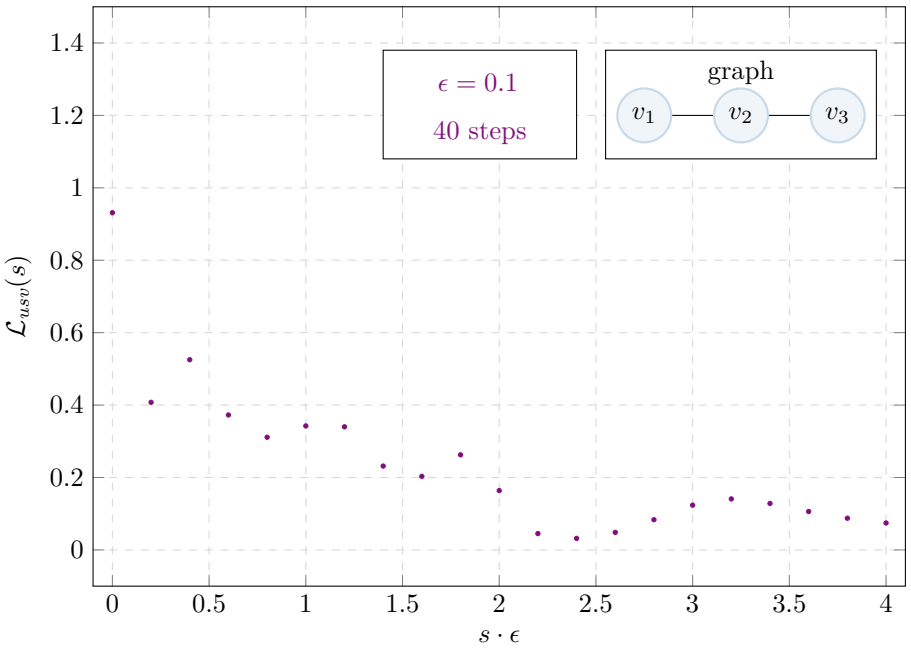


Figure 3.9 Shows the same training process as in (Figure 3.8) but here ϵ was chosen too large.

A more complex case offers the following graph:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{3.37}$$

which is not fully connected, but instead there are two disconnected pieces. The minimum requirement for minimising the cost function is mapping v_1 and v_2 onto the same state as well as v_3 and v_4 respectively (Figure 3.10). This could provide an answer to the question in Section 3.2: Does the network distinguish between unconnected sets in the graph? The answer is yes in this case, as it can distinguish between the pairs of vertices. To illustrate, consider the Hilbert-Schmidt distance between the second and third vertices during the training process. If:

$$d_{HS}(\mathcal{E}(\rho_2^{in}), \mathcal{E}(\rho_3^{in})) \neq 0, \tag{3.38}$$

then the network does not choose the trivial solution of mapping all states to the same output but rather can distinguish between disconnected graph pieces, as shown in (Figure 3.10). One generally cannot exclude cases where the network maps all states onto the same output by randomly choosing the network unitaries as

described in (Eq. 3.20). Additional results about this feature for another disconnected graph and different QNNs are found in the appendix (Figure D.3), (Figure D.4) and (Figure D.5).

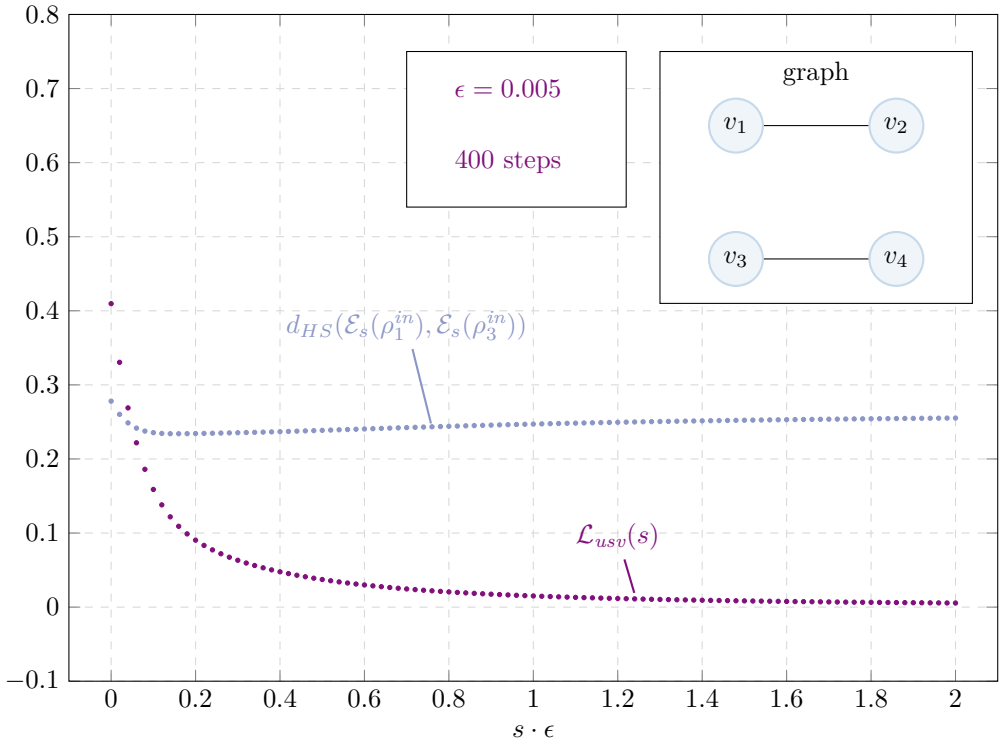


Figure 3.10 Shows the cost function $\mathcal{L}_{usv}$ and the distance between v_1 and v_3 for unsupervised training a $[2, 2]$-QNN using the graph (Eq. 3.37) with $\epsilon = 0.005$ and 400 training steps.

3.6 Semi-supervised learning task and results

This section extends the previous unsupervised learning task to a semi-supervised one, whose classical correspondence was described in Section 2.2. Instead of training a QNN with solely unlabelled training data, there are additional labels:

$$\rho_i^{sv} \in \mathcal{D}\left(\mathbb{C}^{2K_{sv}}\right)$$

for certain data points as targets for the training process. The sorted semi-supervised graph-structured quantum data then takes the form:

$$\textit{semi-supervised graph-structured quantum data} = \left(\{(\rho_1^{in}, \rho_1^{sv}), \ldots, (\rho_R^{in}, \rho_R^{sv}), \rho_{R+1}^{in}, \ldots, \rho_N^{in}\}, G\right).$$

The Fidelity (Eq. 3.18) represents a suitable choice for a supervised cost function:

$$\mathcal{L}_{sv}(s) = \frac{1}{R}\sum_{i=1}^{R} \operatorname{tr}\left\{\mathcal{E}_s(\rho_i^{in})\rho_i^{sv}\right\}, \tag{3.39}$$

with R representing the number of supervised training pairs. Hereafter, all observations will be restricted to pure labels ρ_i^{sv}, thereby validating the previous cost function choice. The QNN best matches the supervised data if $\mathcal{L}_{sv}$ maximises to 1, and thus a full mathematical description of the semi-supervised task is achieved via:

$$\mathcal{L}_{ssv}(s) = \mathcal{L}_{sv}(s) + \gamma\mathcal{L}_{usv}(s) = \frac{1}{R}\sum_{i=1}^{R} \operatorname{tr}\left\{\mathcal{E}_s(\rho_i^{in})\rho_i^{sv}\right\} + \gamma\sum_{i\sim j}[A]_{ij}\operatorname{tr}\left\{(\mathcal{E}_s(\rho_i^{in}) - \mathcal{E}_s(\rho_j^{in}))^2\right\}, \tag{3.40}$$

where γ is an additional parameter. γ controls the importance of the unsupervised training and fixes, that $\mathcal{L}_{sv}$ has to be maximised and $\mathcal{L}_{usv}$ minimised by choosing $\gamma < 0$. Then, $\gamma\mathcal{L}_{usv}$ should be maximised and the overall training task becomes maximising $\mathcal{L}_{ssv}$ to 1.

The procedure to calculate the updating matrices matches that in Section 3.3, and all operations used from (Eq. 3.21) to (Eq. 3.30) are linear in the cost function. Hence, the remaining task is to determine the updating matrices for $\mathcal{L}_{sv}$. Since the neural network and $\mathcal{E}_s$ remain the same as in Section 3.3, (Eq. 3.23) and (Eq. 3.22) can be used to get:

$$\begin{aligned}
\frac{d\mathcal{L}_{sv}(s)}{ds} &= \lim_{\epsilon\to 0}\frac{\mathcal{L}_{sv}(s+\epsilon) - \mathcal{L}_{sv}(s)}{\epsilon} \\
&= \lim_{\epsilon\to 0}\frac{\sum_i^R \operatorname{tr}\left\{\mathcal{E}_{s+\epsilon}(\rho_i^{in})\rho_i^{sv}\right\} - \sum_i^R \operatorname{tr}\left\{\mathcal{E}_s(\rho_i^{in})\rho_i^{sv}\right\}}{R\epsilon} \\
&= \lim_{\epsilon\to 0}\frac{\sum_i^R \operatorname{tr}\left\{\mathcal{E}_s(\rho_i^{in})\rho_i^{sv}\right\} + i\epsilon\sum_i^R \operatorname{tr}\left\{X(s,\rho_i^{in})\rho_i^{sv}\right\} + \mathcal{O}(\epsilon^2) - \sum_i^R \operatorname{tr}\left\{\mathcal{E}_s(\rho_i^{in})\rho_i^{sv}\right\}}{R\epsilon} \\
&= \frac{i}{R}\sum_i \operatorname{tr}\left\{\rho_i^{sv}X(s,\rho_i^{in})\right\} + \lim_{\epsilon\to 0}\mathcal{O}(\epsilon) \\
&= \frac{i}{R}\sum_i \operatorname{tr}\Big\{ \left(\mathbb{I}_{\{in,\ hidden\}}\otimes\rho_i^{sv}\right) \\
&\quad \cdot \Big(\Big[K_{n_L}^L(s), U_{n_L}^L(s)\ldots U_1^1(s)\left(\rho_i^{in}\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|\right)U_1^{1\dagger}(s)\ldots U_{n_L}^{L\ \dagger}(s)\Big] \\
&\quad + \ldots \\
&\quad + U_{n_L}^L(s)\ldots U_2^1(s)\left[K_1^1(s), U_1^1(s)\left(\rho_i^{in}\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|\right)U_1^{1\dagger}(s)\right]U_2^{1\dagger}(s)\ldots U_{n_L}^{L\ \dagger}(s)\Big)\Big\} \\
&= \frac{i}{R}\sum_i \operatorname{tr}\Big\{ \Big[U_{n_L}^L(s)\ldots U_1^1(s)\left(\rho_i^{in}\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|\right)U_1^{1\dagger}(s)\ldots U_{n_L}^{L\ \dagger}(s), \\
&\qquad \left(\mathbb{I}_{\{in,\ hidden\}}\otimes\rho_i^{sv}\right)\Big]K_{n_L}^L(s) + \ldots \\
&\quad + \Big[U_1^1(s)\left(\rho_i^{in}\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|\right)U_1^{1\dagger}(s), U_2^{1\dagger}(s)\ldots U_{n_L}^{L\ \dagger}(s) \\
&\qquad \cdot \left(\mathbb{I}_{\{in,\ hidden\}}\otimes\rho_i^{sv}\right)U_{n_L}^L(s)\ldots U_2^1(s)\Big]K_1^1(s)\Big\}
\end{aligned}$$

$$= \frac{i}{R}\sum_i \mathrm{tr}\left\{M^L_{n_L\{i\}}(s)K^L_{n_L}(s)+\ldots+M^1_{1\{i\}}(s)K^1_1(s)\right\}.$$

with:

$$M^l_{m\{i\}}(s) = \Big[U^l_m(s)\ldots U^1_1(s)\left(\rho^{in}_i\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|\right)U_1^{1\,\dagger}(s)\ldots U_m^{l\,\dagger}(s), \quad (3.41)$$
$$U^l_{m+1}{}^{\dagger}(s)\ldots U^L_{n_L}{}^{\dagger}(s)\left(\mathbb{I}_{\{in,\ hidden\}}\otimes\rho^{sv}_i\right)U^L_{n_L}(s)\ldots U^l_{m+1}(s)\Big].$$

Applying the Lagrange method leads to:

$$K^l_m(s) = \frac{2^{n_{l-1}}i}{R\lambda}\sum_i \underset{rest}{\mathrm{tr}}\left\{M^l_{m\{i\}}(s)\right\}, \quad (3.42)$$

where maximising the cost function means $d_s\mathcal{L}_{sv}(s)>0$, and thus the Lagrange parameter λ must be greater then zero, which is consistent with the previous choice of γ in (Eq. 3.40). The notable difference between the updating rules for supervised and unsupervised training regards the M-matrices (Eq. 3.25) and (Eq. 3.41). The supervised matrix replaces $(\rho^{in}_i-\rho^{in}_j)$ with ρ^{in}_i and $(\rho^{out}_i-\rho^{out}_j)$ with ρ^{sv}_i. This result is obvious since the supervised task does not compare two states in the input data; however, why does the unsupervised task involve the network output $\rho^{out}_i(s)$ while the supervised task does not? This is no contradiction since there is no further information stored in $\rho^{out}_i(s)$, which is not contained in the unitaries. Using the Hilbert-Schmidt distance in the supervised cost function:

$$\mathcal{L}^{alt.}_{sv}(s) = \frac{1}{R}\sum_i^R d_{HS}\left(\mathcal{E}(\rho^{in}_i),\rho^{sv}_i\right), \quad (3.43)$$

results in:

$$K^l_m(s) = \frac{2^{n_{l-1}+1}i}{R\lambda}\sum_i \underset{rest}{\mathrm{tr}}\left\{M^{alt.,l}_{m\{i\}}(s)\right\}$$

with:

$$M^{alt.,l}_{m\{i\}}(s) = \Big[U^l_m(s)\ldots U^1_1(s)\left(\rho^{in}_i\otimes|0,\ldots,0\rangle_{\{hidden,\ out\}}\langle 0,\ldots,0|\right)U_1^{1\,\dagger}(s)\ldots U_m^{l\,\dagger}(s),$$
$$U^l_{m+1}{}^{\dagger}(s)\ldots U^L_{n_L}{}^{\dagger}(s)\left(\mathbb{I}_{\{in,\ hidden\}}\otimes\ \left(\rho^{out}_i(s)-\rho^{sv}_i\right)\right)U^L_{n_L}(s)\ldots U^l_{m+1}(s)\Big].$$

Hence, the inclusion of $\rho^{out}_i(s)$ in the M-matrices is an artefact of the Hilbert-Schmidt distance. (Eq. 3.43) also allows using mixed labels, but we will restrict our observation to pure ones.

Matching both updating matrices for supervised (Eq. 3.42) and unsupervised (Eq. 3.30) training leads to the updating matrix for semi-supervised training

$$K^l_m(s) = \frac{2^{n_{l-1}}i}{R\lambda}\sum_i \underset{rest}{\mathrm{tr}}\left\{M^l_{m\{i\}}(s)\right\}+\gamma\frac{2^{n_{l-1}+1}i}{\lambda}\sum_{i\sim j}[A]_{ij}\underset{rest}{\mathrm{tr}}\left\{M^l_{m\{i,j\}}(s)\right\},\ \lambda>0>\gamma. \quad (3.44)$$

The network implementation is similar to the one in Section 3.4, but the algorithm for the updating matrix requires modification. Firstly, another list contains all supervised targets:

$$trainingData = [(\rho_1^{in}, \rho_1^{sv}), \ldots, (\rho_R^{in}, \rho_R^{sv})].$$

The update matrix algorithm then becomes

Algorithm 4 Semi-supervised update matrix

procedure SSVUPDATEMATRIX(*storedStates*, *trainingData*, *qnnArch*, *currentUnitaries*, λ, ϵ, γ, A, l, m)

$K^l_{m,usv} \leftarrow 0$;

$K^l_{m,sv} \leftarrow 0$;

for i in $\{1, \ldots, N\}$ **do**

for j in $\{1, \ldots, N\}$ **do**

if $[A]_{ij} \neq 0$ **then**

$$M^l_{m\{i,j\}} \leftarrow \left[\prod_{\alpha=m}^{1} U^l_\alpha \left(\left(\rho_i^{(l-1)} - \rho_j^{(l-1)} \right) \otimes |0, \ldots, 0\rangle_l \langle 0, \ldots, 0| \right) \prod_{\alpha=1}^{m} U^{l\,\dagger}_\alpha, \right.$$

$$\left. \prod_{\alpha=m+1}^{n_l} U^l_\alpha \left(\mathbb{I}_{l-1} \otimes \left(\sigma^{(l)}(\rho_i^{out}) - \sigma^{(l)}(\rho_j^{out}) \right) \right) \prod_{\alpha=n_l}^{m+1} U^{l\,\dagger}_\alpha \right];$$

$K^l_{m,usv} += [A]_{ij} M^l_{m\{i,j\}}$;

$K^l_{m,usv} \leftarrow \frac{2^{n_{l-1}+1} i}{\lambda} \operatorname{tr}_{rest}\left\{K^l_{m,usv}\right\}$;

for i in $\{1, \ldots, R\}$ **do**

$$M^l_{m\{i\}} \leftarrow \left[\prod_{\alpha=m}^{1} U^l_\alpha \left(\rho_i^{(l-1)} \otimes |0, \ldots, 0\rangle_l \langle 0, \ldots, 0| \right) \prod_{\alpha=1}^{m} U^{l\,\dagger}_\alpha, \prod_{\alpha=m+1}^{n_l} U^l_\alpha \left(\mathbb{I}_{l-1} \otimes \sigma^{(l)}(\rho_i^{sv}) \right) \prod_{\alpha=n_l}^{m+1} U^{l\,\dagger}_\alpha \right];$$

$K^l_{m,sv} += M^l_{m\{i,j\}}$;

$K^l_{m,sv} \leftarrow \frac{2^{n_l-1} i}{R\lambda} \operatorname{tr}_{rest}\left\{K^l_{m,sv}\right\}$;

$K^l_m \leftarrow K^l_{m,sv} + \gamma K^l_{m,usv}$;

$$K^l_m \leftarrow \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m] \left(K^l_m \otimes \underbrace{\mathbb{I}_2 \otimes \ldots \otimes \mathbb{I}_2}_{n_l - 1} \right) \mathbf{SWAP}_{n_{l-1}+n_l}[n_{l-1}+1,\ n_{l-1}+m];$$

$update \leftarrow e^{i\epsilon K^l_m} U^l_m$;

return *update*;

The python implementation can be found in (App. C.1.5).

Multiple examples must be checked, including multiple settings for γ and graph types. The first example will be as simple as the first example for unsupervised training, and therefore we choose the same graph and QNN ($K = K_{sv} = 2$) and add random pure labels to the vertices v_1 and v_3. (Figure 3.11) shows the training results for $\gamma = -1$. As expected, it is not possible to reach the optimal cost function value of 1 if $\rho_1^{sv} \neq \rho_3^{sv}$. Instead, the network reaches a compromise between unsupervised and supervised tasks which map ρ_2^{in} to

the same distance to both supervised states (Figure 3.11). Furthermore, decreasing γ separates the states, and thus the supervised task is more fulfilled (Figure 3.12) and the distance between ρ_1^{out} and ρ_2^{out} (ρ_2^{out} and ρ_3^{out} respectively) drastically increases. (Figure 3.13) shows the influence of a different graph type and that solving the task for such a graph becomes more difficult, and thus the cost function is not nearly close to 1. Further results for different labels can be found in (Figure E.2), for different QNN architectures in (Figure E.2) and (Figure E.1), different values of γ in (Figure E.3) and (Figure E.4) and another graph in (Figure E.4). Each of them shows the same results as above.

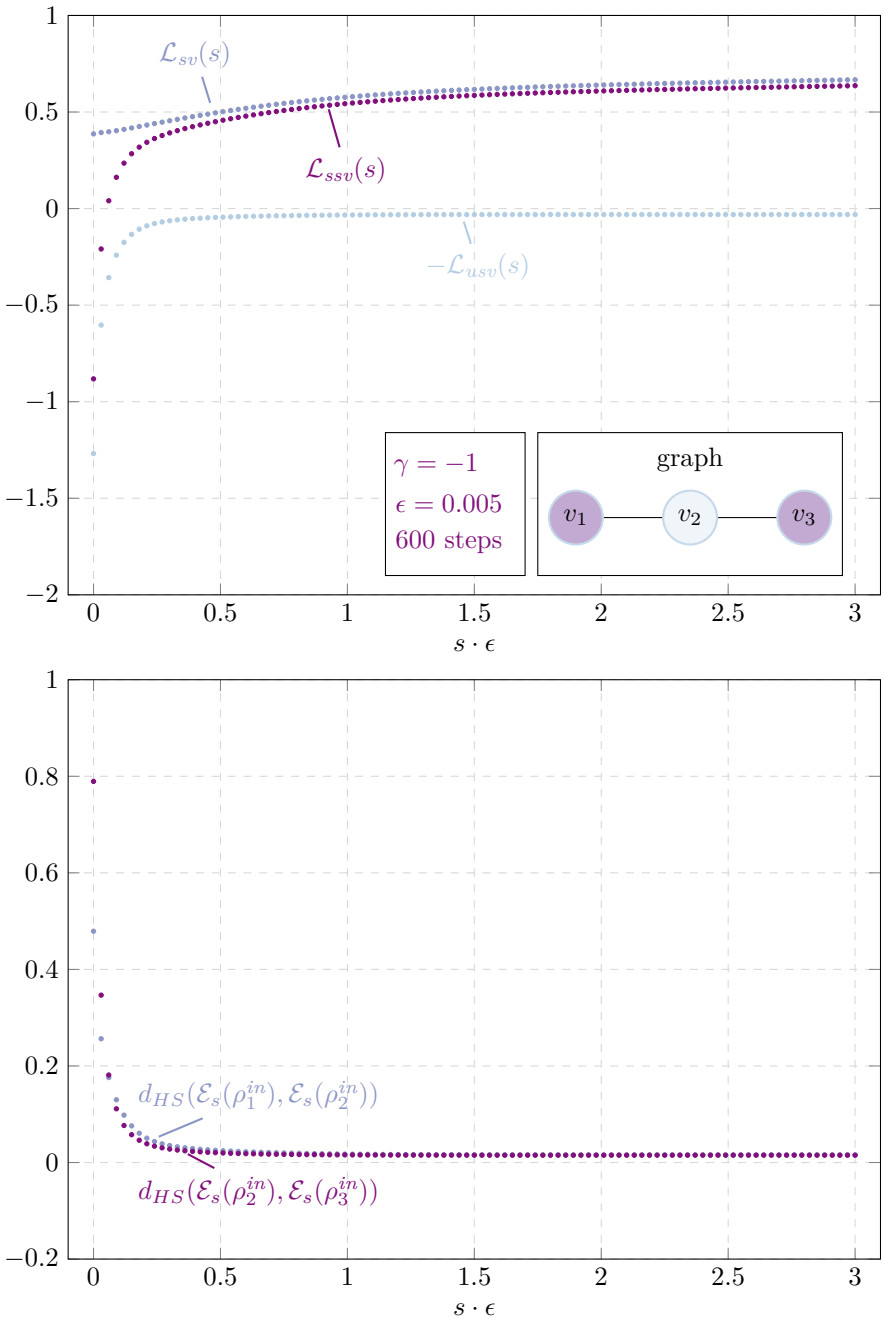


Figure 3.11 Values of the cost function for semi-supervised training of a $[2, 2]$-QNN. Here $\epsilon = 0.005$, $\lambda = 1$ and $\gamma = -1$. The labeled vertices are indicated in purple. The second graph shows the distance of the left and right data point in the graph to the point in the center during the training process.

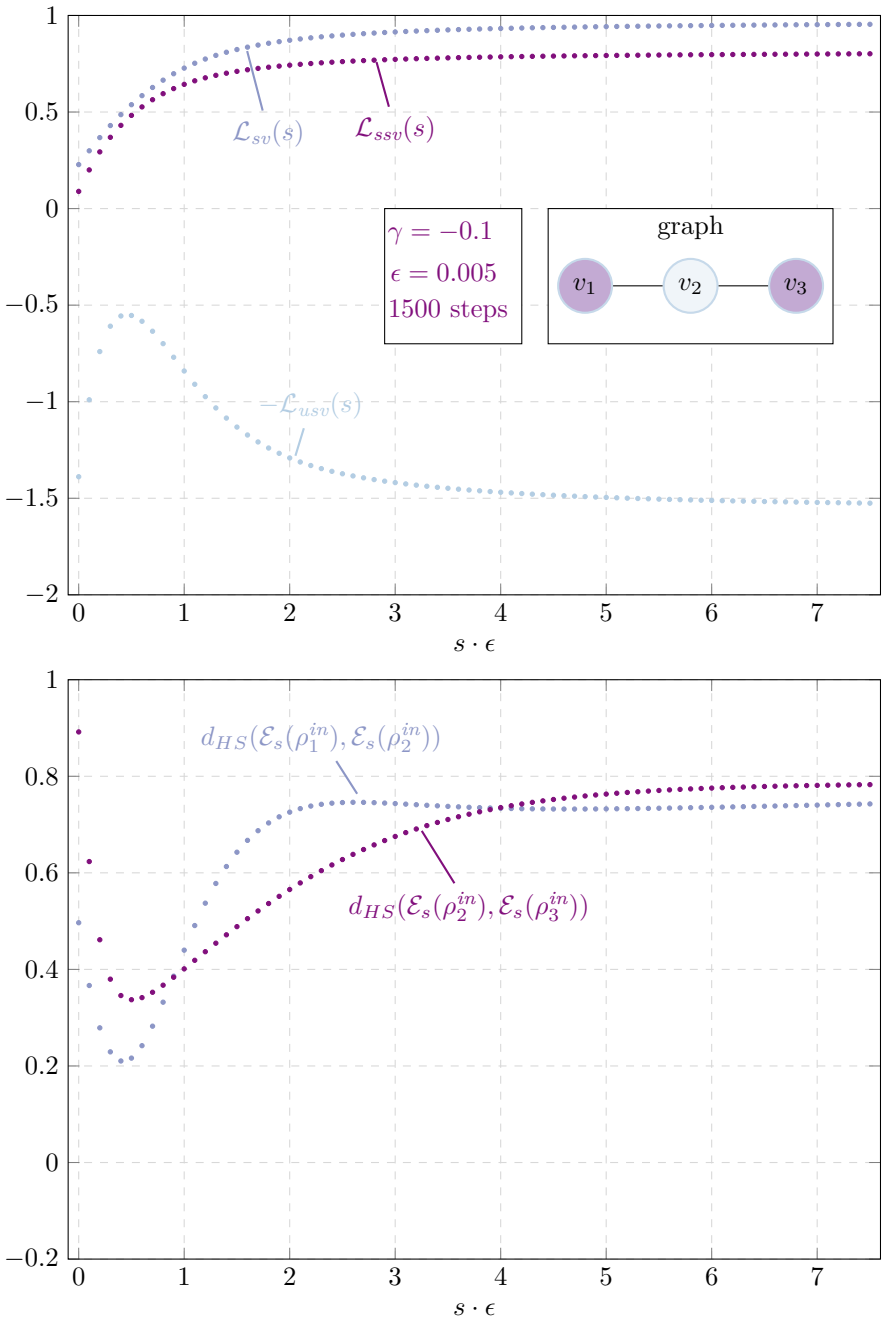


Figure 3.12 Values of the cost function for semi-supervised training of a $[2, 2]$-QNN. Here $\epsilon = 0.005$, $\lambda = 1$ and $\gamma = -0.1$. The labeled vertices are indicated in purple. The second graph shows the distance of the left and right data point in the graph to the point in the center during the training process.

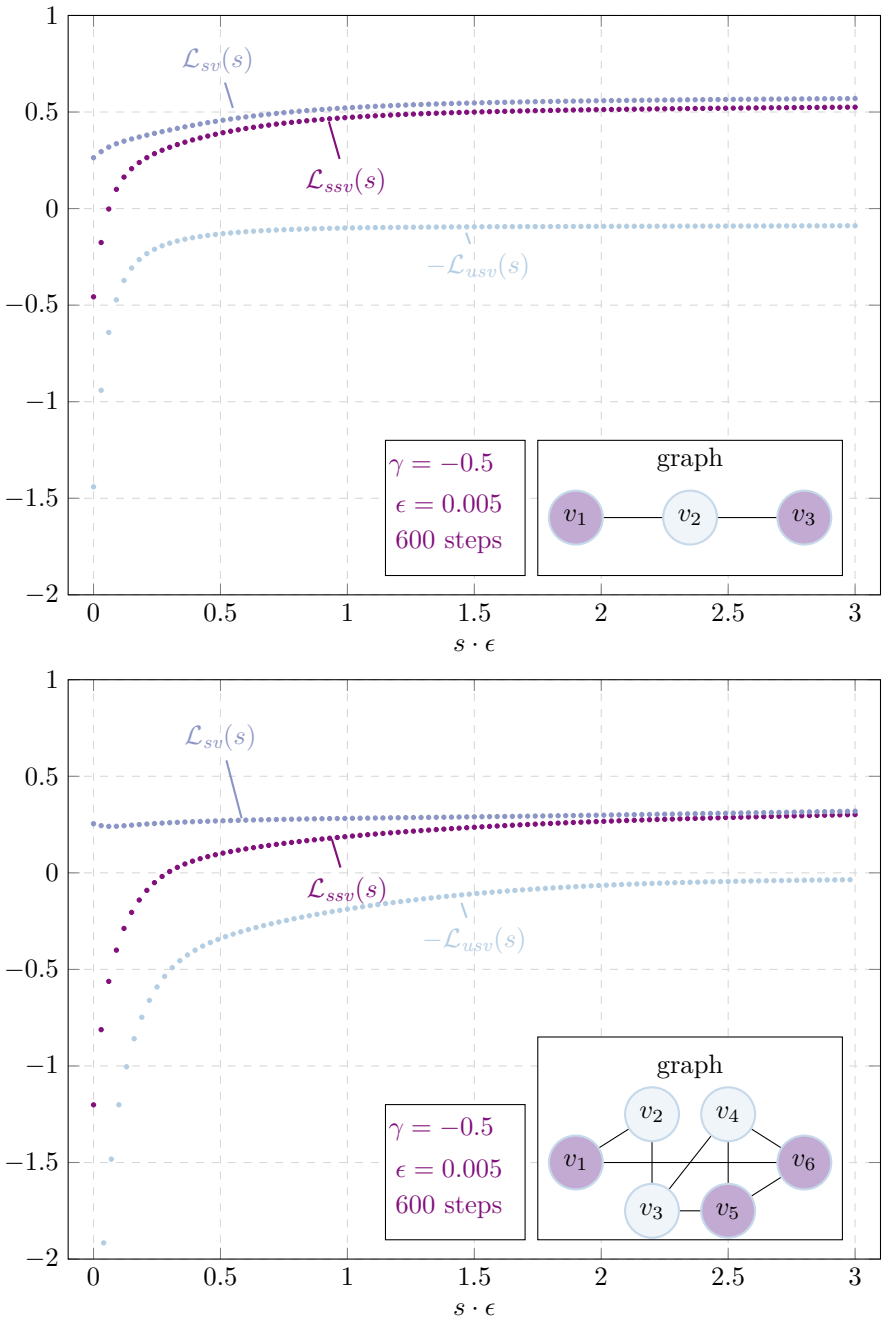


Figure 3.13 Values of the cost function for semi-supervised training of a $[2, 2]$-QNN. Here we observed two different graph types with $\epsilon = 0.005$, $\lambda = 1$ and $\gamma = -0.5$. The labeled vertices are indicated in purple.

3.7 Generalisation

We then attempt to use a specific task to indicate the usefulness of the new model. A semi-supervised learning task can also be regarded as interpolating unlabelled data points alongside the topological structure of the graph manifested by a set of labels in the output space. This section provides a numerical test for this proposal, which involves a set of five pure, single-qubit input states representing a straight line on the surface from the north to south poles on the Bloch sphere:

$$\textit{input states} = \{|\phi^{in}\rangle = |0\rangle\,, |\phi_2^{in}\rangle\,, |\phi_3^{in}\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), |\phi_4^{in}\rangle\,, |\phi_5^{in}\rangle = |0\rangle\},$$

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

The labels are therefore generated by a unitary operator U representing a rotation around the z axis. This leads to a correlation between input data and labels, but one forgets the outputs of the second and fourth states to achieve a semi-supervised interpolating task:

$$\textit{training data} = \{(|\phi_1^{in}\rangle\,, |\phi_1^{sv}\rangle = U\,|\phi_1^{in}\rangle), (|\phi_3^{in}\rangle\,, |\phi_3^{sv}\rangle = U\,|\phi_1^{in}\rangle), (|\phi_4^{in}\rangle\,, |\phi_4^{sv}\rangle = U\,|\phi_4^{in}\rangle)\}.$$

To control the importance of the different training data points, two parameters $\gamma_1, \gamma_2 < 0$ in the cost function can be used:

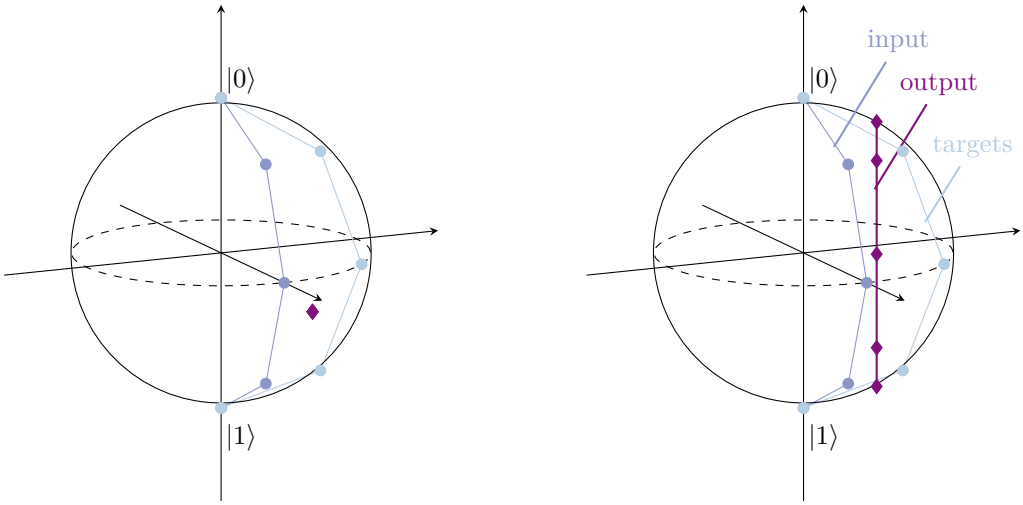
$$\mathcal{L}(s) = \sum_{i\sim j}[A]_{ij} d_{HS}(\mathcal{E}_s(\rho_i^{in}), \mathcal{E}_s(\rho_j^{in})) + \gamma_1(\langle\phi_1^{sv}|\,\mathcal{E}_s(\rho_1^{in})\,|\phi_1^{sv}\rangle + \langle\phi_5^{sv}|\,\mathcal{E}_s(\rho_5^{in})\,|\phi_5^{sv}\rangle) + \gamma_2\,\langle\phi_3^{sv}|\,\mathcal{E}_s(\rho_3^{in})\,|\phi_3^{sv}\rangle\,.$$

The updating rules are calculated similarly as before, where one attempts to minimise $\mathcal{L}$ with $\lambda < 0$. This results in the following expression for K_m^l:

$$\begin{aligned} K_m^l(s) \;=\; & \frac{2^{n_{l-1}+1} i}{\lambda} \sum_{i\sim j}[A]_{ij} \operatorname*{tr}_{rest}\left\{M^l_{m\{i,j\}}(s)\right\} + \gamma_1 \frac{2^{n_{l-1}} i}{\lambda}\left(\operatorname*{tr}_{rest}\left\{M^l_{m\{1\}}(s)\right\} + \operatorname*{tr}_{rest}\left\{M^l_{m\{5\}}(s)\right\}\right) \\ & + \gamma_2 \frac{2^{n_{l-1}} i}{\lambda} \operatorname*{tr}_{rest}\left\{M^l_{m\{3\}}(s)\right\}. \end{aligned}$$

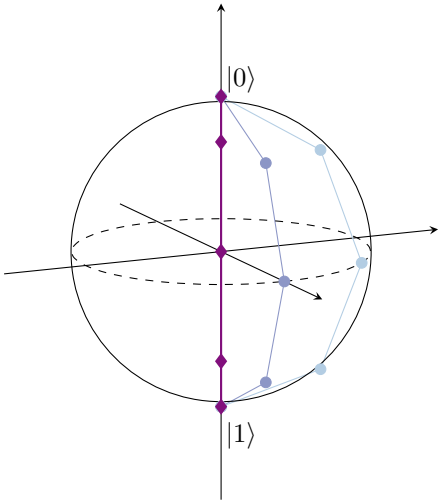
The training results are shown in (Figure 3.14) for different values of γ_1 and γ_2, where case (a) shows the result of unsupervised training a fully connected data set. Case (b) shows how the QNN tries to satisfy supervising the north and south pole states of the Bloch sphere, as well as the unsupervised task, while in (c) the first supervised term begins to dominate. In addition, activating the second supervised term

(d) approximates the unitary rotation as well as for the unsupervised states; however, if the supervised task becomes much more important than the unsupervised one, then the unitary rotation is seemingly best approximated. The results from [9], show that a QML can learn a unitary rotation with solely supervised training. A graph structure is thus not needed to solve the task, but instead makes this more difficult for the QNN. This example only verifies the generalisability of the supervised part and the lack of conflict with the unsupervised part; however, the generalisability for the whole semi-supervised task is not yet shown, which would require an example, where the graph somehow influences the labels.

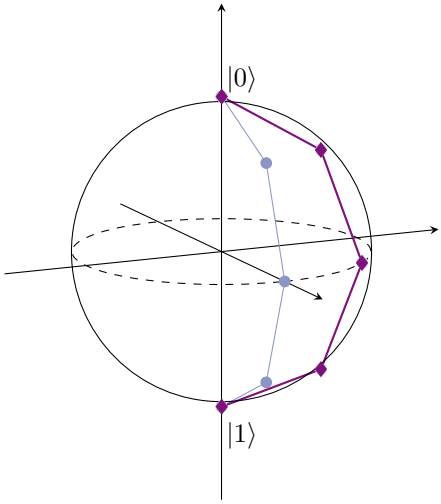


(a) $\gamma_1 = 0, \gamma_2 = 0$

(b) $\gamma_1 = -1, \gamma_2 = 0$



(c) $\gamma_1 = -2, \gamma_2 = 0$



(d) $\gamma_1 = -6, \gamma_2 = -4$

Figure 3.14 Shows the results of training a $[1, 1]$-QNN with input data (Eq. 3.45) and training data (Eq. 3.45) represented on the Bloch sphere, while U is a rotation by 45°. (Eq. 3.45) was taken as cost function with adjacency (Eq. 3.45). The network output, after training, is depicted in purple, while the input data is dark blue and the target data light blue. Here we vary γ_1 and γ_2.

Chapter Four

Conclusion

In summary, we firstly observed semi-supervised deep learning tasks on graph-structured data and GCNs, to aid formulating a quantum version. The QNN must be described before formulating a deep learning quantum task. The QNN can handle multi-qubit input data and is universal, while the modified version for the implementation probably violates this universality. Nevertheless, the QNN does not violate quantum theory due to constructing the perceptron as channel. Then we formulate an unsupervised QML task for graph-structured quantum data, where the training data consist of multi-qubit states and a graph showing relations between the data points. The QML task's cost function involves the Hilbert-Schmidt distance due to the need to measure the closeness of two connected outputs of the QNN, which are generally mixed. Calculating the updating matrix and implementing the training process into python enabled observing training examples. The numerics confirmed the expected results, i.e., the network solves the unsupervised task via mapping connected input data onto the same output and disconnected pieces of the graph to different outputs.

The extension to semi-supervised training involved the results from [9]. We add several labels to the graph-structured quantum data and an additional supervised term to the cost function using the Fidelity. Hence, the new updating matrix represents a linear combination of the unsupervised and semi-supervised updating matrices. The numerics included different graph types, weights and parameter variations to check whether the network properly performs, which resulted in the QNN again passing all tests. These results lead to the supposition that a numerical proof of generalisability would confirm the model's uesfulness; however, the experiment was less meaningful than expected, and thus its generalisability was not confirmed, but we showed that it is possible to extend unsupervised and semi-supervised ML tasks on graph-structured data to QML.

A generalisable model can also perform well on an unknown data set after training with a training set, and this ability is key to being a useful ML model. But generalisability is difficult to check, however, since there are probably infinite data sets with unknown correlations, a trained neural network can behave strangely, and sometimes there is no 'good' training data found until now. But one also gains confidence that a model performs well for unknown sets if its results are predictable for a basis of default sets. The training examples showed that the QNN could maximise or minimise the cost function and therefore perfectly solved the tasks. Furthermore the supervised part is still generalisable, which suggests the same for the unsupervised part. This does not present a proof of generalisation, but probably provides sufficient evidence to assume correct behaviour, in the absence of observing a meaningful example. Such an example could include large sets of input data, where several states are close to each other while all other are far away, in which case the graph could seemingly aid finding unknown labels.

These results form a foundation for further observations to understand which types of tasks can be solved by a QNN. Therefore, it might be interesting to observe a more difficult unsupervised task, which forces two connected vertices in the graph to be far away from each other, or to examine more physical examples, i.e., a physical correlation between input and supervised data described by a Hamiltonian. We can also attempt to implement the abilities of GCNs into our QNN.
If these problems are solvable by QML on grap-structured quantum data, this could form a powerful tool for real-world problems. .

REFERENCES

[1] John Preskill. *Quantum computing and the entanglement frontier*. 2012. arXiv: 1203.5813 [quant-ph].

[2] John Preskill. "Quantum Computing in the NISQ Era and Beyond". In: *Quantum* 2 (2018), p. 79. arXiv: 1801.00862.

[3] R. P. Feynman. "Simulating physics with computers". In: *Int. J. Theor. Phys.* 21 (1982), pp. 467–488.

[4] F. Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510.

[5] E. Pednault et al. "On Quantum Supremacy". In: *IBM Research Blog* (Oct. 2019). URL: https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/.

[6] S. Aaronson. "The limits of quantum computers". In: *Scientific American* (2008).

[7] S. Pakin and P. Coles. "The Problem with Quantum Computers". In: *Scientific American* (June 2019).

[8] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[9] Kerstin Beer et al. *Efficiant Learning for Deep Quantum Neural Netowrks*. 2019. URL: https://arxiv.org/abs/1902.10445.

[10] Tom Mitchelli. *Machine Learning*. New York: McGraw Hill, 1997.

[11] Emmanuel Gbenga Dada et al. "Machine learning for email spam filtering: review, approaches and open research problems". In: *Heliyon* 5.6 (2019), e01802. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2019.e01802. URL: http://www.sciencedirect.com/science/article/pii/S2405844018353404.

[12] S. Ramos et al. "Detecting Unexpected Obstacles for Self-Driving Cars: Fusing Deep Learning and Geometric Modeling". 2017. eprint: arXiv:1612.06573.

[13] J. Stilgoe. "Machine learning, social learning and the governance of self-driving cars". In: *Soc. Stud. Sci.* 48.1 (2018), pp. 25–56.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[15] C. Bishop. *Pattern Recognition and Machine Learning, Information Science and Statistics*. Springer-Verlag, 2006.

[16] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 . URL: http://neuralnetworksanddeeplearning.com.

[17] M. I. Jordan and T. M. Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (2015), pp. 255–260.

[18] T. Joutou and K. Yanai. "A food image recognition system with Multiple Kernel Learning". In: *ICIP'09 Proceedings of the 16th IEEE international conference on Image processing*. 2009, pp. 285–288.

[19] J.S. Zhen and I. Watson. *Neuroevolution for Micromanagement in the Real-Time Strategy Game Starcraft: Brood War*. 2013.

[20] Megha Khosla, Vinay Setty, and Avishek Anand. "A Comparative Study for Unsupervised Network Representation Learning". In: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[21] Z. Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), pp. 1–21.

[22] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: Online Learning of Social Representations". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2014, pp. 701–710.

[23] J. Qiu et al. "Network Embedding As Matrix Factorization: Unifying DeepWalk, LINE, PTE, and Node2Vec". In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. Association for Computing Machinery, 2018, pp. 459–467.

[24] Shengchao Liu, Mehmet F Demirel, and Yingyu Liang. "N-Gram Graph: Simple Unsupervised Representation for Graphs, with Applications to Molecules". In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 8466–8478.

[25] Megha Khosla et al. "Node Representation Learning for Directed Graphs". In: *Machine Learning and Knowledge Discovery in Databases*. Springer International Publishing, 2020, pp. 395–411.

[26] Max Welling Thomas N. Kipf. *Semi-Supervised Classification with Graph Convolutional Networks*. 2016. URL: https://arxiv.org/abs/1609.02907.

[27] Petar Velikovi et al. "Graph Attention Networks". In: *International Conference on Learning Representations*. 2018.

[28] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Advances in neural information processing systems*. 2017, pp. 1024–1034.

[29] Keyulu Xu et al. "How Powerful are Graph Neural Networks?" In: *International Conference on Learning Representations*. 2019.

[30] J. Biamonte et al. "Quantum machine learning". In: *Nature* 549.7671 (2017), pp. 195–202.

[31] Carlo Ciliberto et al. "Quantum Machine Learning: A Classical Perspective". In: *Proc. Roy. Soc. A* 474.2209 (2018), p. 20170551.

[32] Maria Schuld and Francesco Petruccione. "Quantum Machine Learning". In: *Encyclopedia of Machine Learning and Data Mining*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2017, pp. 1034–1043.

[33] N. B. Lovett et al. "Differential Evolution for Many-Particle Adaptive Quantum Metrology". In: *Phys. Rev. Lett.* 110.22 (2013), p. 220501. DOI: 10.1103/PhysRevLett.110.220501.

[34] Giuseppe Carleo and Matthias Troyer. "Solving the Quantum Many-Body Problem with Artificial Neural Networks". In: *Science* 355.6325 (2017), pp. 602–606.

[35] M. Tiersch, E. J. Ganahl, and H. J. Briegel. "Adaptive Quantum Computation in Changing Environments Using Projective Simulation". In: *Sci. Rep.* 5.1 (2015), p. 12874.

[36] Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. "Quantum Speed-up for Unsupervised Learning". In: *Machine Learning* 90.2 (Feb. 2013), pp. 261–287.

[37] Giuseppe Davide Paparo et al. "Quantum Speedup for Active Learning Agents". In: *Phys. Rev. X* 4.3 (2014), p. 031002.

[38] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. "The Quest for a Quantum Neural Network". In: *Quant. Inf. Proc.* 13.11 (2014), pp. 2567–2586.

[39] Kunal Sharma et al. "Trainability of Dissipative Perceptron-Based Quantum Neural Networks". In: *arXiv:2005.12458 [quant-ph]* (May 2020). arXiv: 2005.12458 [quant-ph].

[40] Masahide Sasaki and Alberto Carlini. "Quantum Learning and Universal Quantum Matching Machine". In: *Phys. Rev. A* 66.2 (2002), p. 022303.

[41] Sébastien Gambs. "Quantum Classification". In: *arXiv:0809.0444* (2008).

[42] G. Sentís et al. "Quantum learning without quantum memory". In: *Sci. Rep.* 2.1 (2012). DOI: 10.1038/srep00708.

[43] V. Dunjko, J. M. Taylor, and H. J. Briegel. "Quantum-Enhanced Machine Learning". In: *Phys. Rev. Lett.* 117.13 (2016). DOI: 10.1103/physrevlett.117.130501.

[44] Alex Monràs, Gael Sentís, and Peter Wittek. "Inductive Supervised Quantum Learning". In: *Phys. Rev. Lett.* 118.19 (2017), p. 190503.

[45] U. Alvarez-Rodriguez et al. "Supervised Quantum Learning without Measurements". en. In: *Sci. Rep.* 7.1 (Oct. 2017), p. 13645.

[46] M. H. Amin et al. "Quantum Boltzmann Machine". In: *Phys. Rev. X* 8.2 (2018), p. 021050.

[47] Y. Du et al. "The Expressive Power of Parameterized Quantum Circuits". In: *arXiv:1810.11922* (Oct. 2018). URL: http://arxiv.org/abs/1810.11922 (visited on 01/31/2019).

[48] G. Sentís et al. "Unsupervised classification of quantum data". In: *Phys. Rev. X* 9 (2019), p. 041029.

[49] Guillaume Verdon, Jason Pye, and Michael Broughton. "A Universal Training Algorithm for Quantum Deep Learning". In: *arXiv:1806.09729* (2018).

[50] Stefan Dernbach et al. "Quantum walk neural networks with feature dependent coins". In: *Applied Network Science* 4.1 (2019), p. 76.

[51] Srinivasan Arunachalam and Ronald de Wolf. "Guest Column: A Survey of Quantum Learning Theory". In: *SIGACT News* 48.2 (2017), pp. 41–67.

[52] Iris Cong, Soonwon Choi, and Mikhail D Lukin. "Quantum convolutional neural networks". In: *Nature Physics* 15.12 (2019), pp. 1273–1278.

[53] Guillaume Verdon et al. "Quantum Graph Neural Networks". In: *arXiv preprint arXiv:1909.12264* (2019).

[54] Partha Niyogi Mikhail Belkin Irina Matveeva. *Regularization and Semi-supervised Learning on Large Graphs*. The University of Chicago, Department of Computer Science. URL: http://people.cs.uchicago.edu/~niyogi/papersps/reg_colt.pdf.

[55] E. Farhi et al. "Quantum Algorithms for Fixed Qubit Architectures". In: (2017). eprint: arXiv:1703.06199.

[56] Vedran Dunjko and Hans J Briegel. "Machine learning & artificial intelligence in the quantum domain: a review of recent progress". In: *Rep. Prog. Phys.* 81.7 (2018), p. 074001. DOI: 10.1088/1361-6633/aab406.

[57] Jure Leskovec William L. Hamilton Rex Ying. *Representation Learning on Graphs: Methods and Applications*. 2017. URL: https://www-cs.stanford.edu/people/jure/pubs/graphrepresentation-ieee17.pdf.

[58] Yuchen Wang et al. *Qudits and high-dimensional quantum computing*. 2020. URL: https://arxiv.org/abs/2008.00959.

[59] C. Bény and F. Richter. *Algebraic approach to quantum theory: a finite-dimensional guide*. 2015. URL: https://arxiv.org/abs/1505.03106.

[60] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press, 2000.

[61] Marvin Schwiering. "Classical Simulation of Quantum Feed-forward Neural Networks". unpublished thesis. 2020.

APPENDIX

Appendix A

Machine learning vocabulary

word	meaning	(equation)
graph G	see definition 1	n.a.
adjacency A	shows the relations of a graph in a symmetric matrix	(Eq. 2.1)
weight matrix W	shows the weights of a graph in a symmetric matrix	(Eq. 2.2)
feature vector $\vec{f}$	represents a data point of K features in $\mathbb{R}^K$	n.a.
graph-structured data	data set containing a graph and associated feature vectors	(Eq. 2.3)
label $\vec{f}^{sv}$	vector targets for certain points in a training set	n.a.
supervised learning	machine learning task with fully labeled training data	n.a.
unsupervised learning	machine learning task with unlabeled training data i.e. graph structured data	n.a.
semi-supervised learning	machine learning task with partial labeled training data	n.a.
semi-supervised graph structured data	partial labeled graph-structured data	(Eq. 2.4)
cost function $\mathcal{L}$	measures how well a given output set fits the training task	example (Eq. 2.8)

word	meaning	(equation)
fully connected neural network	net of layers of neurons, each neuron is connected to each neuron in the next layer via a weight	n.a.
neuron	elementary building block of a neural network, carries a data type	n.a.
data type	specifies the neuron, value or vector assigned to the neuron	n.a.
neural network architecture	shows how many neurons are in each layer	(Eq. 2.9)
hidden layers	layers between the input and the output layer of a neural network	n.a.
single-layer perceptron	assigns a bit value to a neuron, depending on all neurons in the previous layer	(Eq. 2.10)
activation function	confines the output of a perceptron or the layer propagation rule	n.a.
layer propagation rule	applies a certain number of perceptrons to get the values of the next layer	(Eq. 2.11)
graph convolutional networks	neural network with a special type of layer propagation rule which doesn't treat every feature vector separately	(Eq. 2.13)

Appendix B

Additional mathematics

This section provides a mathematical description of the Hilbert space, the representation of a single-qubit state in the Bloch ball and some necessary definitions to introduce $*$-algebras .

B.1 Hilbert space

All observations in this thesis and in quantum mechanics generally involve the Hilbert space as a central mathematical object.

Definition 6 (Hilbert space) *A Hilbert space is a complex (or real) vector space $\mathcal{H}$ with an inner product:*

$$\langle \cdot, \cdot \rangle : \mathcal{H} \times \mathcal{H} \to \mathbb{C},$$

while $\mathcal{H}$ is complete under the norm induced by the inner product.

Let $x, y \in \mathcal{H}$, then the inner product satisfies the following properties:

1. The inner product is conjugate symmetric: $\langle x, y \rangle = \langle y, x \rangle^*$.
2. The inner product is linear in the first argument: $\langle c_1 x_1 + c_2 x_2, y \rangle = c_1 \langle x_1, y \rangle + c_2 \langle x_2, y \rangle$, $\forall c_1, c_2 \in \mathbb{C}$.
3. The inner product is positive definite: $\langle x, x \rangle > 0$ for $x \neq 0$ and $\langle x, x \rangle = 0$ for $x = 0$.

Such an inner product induces a norm on $\mathcal{H}$:

$$\|x\| = \sqrt{\langle x, x \rangle}$$

and a distance measure between two points in $\mathcal{H}$:

$$d(x, y) = \|x - y\| = \sqrt{\langle x - y, x - y \rangle}.$$

Furthermore the inner product satisfies the Cauchy-Schwarz inequality:

$$|\langle x, y\rangle| \leq \|x\| \, \|y\| \, .$$

B.2 Bloch Ball

In case of single qubits it is sometimes useful to express them via Pauli matrices:

$$\rho \cong \begin{pmatrix} \rho_{00} & \rho_{01} \\ \rho_{10} & \rho_{11} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1+z_3 & z_1 - iz_2 \\ z_1 + iz_2 & 1-z_3 \end{pmatrix} = \frac{1}{2} \left(\mathbb{I} + z_1\sigma_1 + z_2\sigma_2 + z_3\sigma_3 \right) .$$

Here:

$$\{\rho_{ij} | i, j \in \{0,1\}, \rho_{ij} \in \mathbb{C}\} \tag{B.1}$$

are the complex entries of the matrix representation of a single-qubit density operator. Their hermitian condition leads to real numbers $\{z_i | i \in \{1,2,3\}, z_i \in \mathbb{R}\}$ and condition (Eq. 3.1) is equal to:

$$\operatorname{tr} \rho^2 = \frac{1}{2} \left(1 + |\vec{z}|^2 \right) \leq 1 \Leftrightarrow |\vec{z}| \leq 1.$$

We call $\vec{z} = (z_1, z_2, z_3) \in \mathbb{R}^3$ the Bloch vector of a state $\rho \in \mathcal{D}\left(\mathbb{C}^2\right)$, which identifies a single qubit density operator with a point in the Bloch ball (Figure B.1).

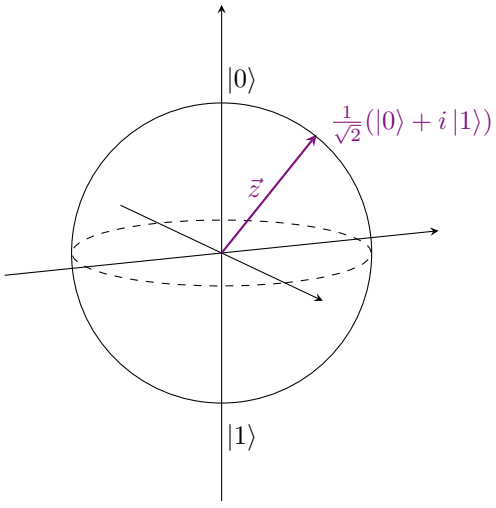


Figure B.1 Shows the Bloch ball. Here for example $\frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle) \cong \vec{z} = (0, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$.

B.3 Algebras

This paragraph provides some important definitions for $*$-algebras.

Definition 7 (Algebra) *An algebra $\mathcal{A}$ over $\mathbb{C}$ is a complex vector space with a multiplication:*

$$\cdot : \mathcal{A} \times \mathcal{A} \to \mathcal{A}, \tag{B.2}$$

which is compatible with vector addition: $a \cdot (b + c) = a \cdot b + a \cdot c$.

Definition 8 *An algebra $\mathcal{A}$ is called unital if it contains the identity $\mathbb{I}$ for the multiplication $\mathbb{I} \cdot a = a \cdot \mathbb{I} = a$.*

Definition 9 ($*$-algebra) *A $*$-algebra $\mathcal{A}$ is an associative algebra over $\mathbb{C}$ with an anti-linear map:*

$$* : \mathcal{A} \to \mathcal{A} \tag{B.3}$$

satisfying:

$$(a^*)^* = a \tag{B.4}$$

$$(a \cdot b)^* = b^* \cdot a^* \tag{B.5}$$

for all $a, b \in \mathcal{A}$.

Definition 10 *A subset $\mathcal{A} \subset \mathcal{B}$ of a $*$-algebra $\mathcal{B}$ is called $*$-subalgebra if it satisfies:*

$$a, b \in \mathcal{A}, \alpha, \beta \in \mathbb{C} \Rightarrow \alpha a + \beta b \in \mathcal{A} \tag{B.6}$$

$$a, b \in \mathcal{A} \Rightarrow a \cdot b \in \mathcal{A} \tag{B.7}$$

$$a \in \mathcal{A} \Rightarrow a^* \in \mathcal{A}. \tag{B.8}$$

Definition 11 *A C^*-algebra is a $*$-algebra which is also a Banach algebra (complete w.r.t. a norm satisfying $\|xa\| \leq \|x\| \, \|y\| \;\; \forall x, y \in \mathcal{A}$) that satisfies:*

$$\|x \cdot x^*\| = \|x\| \, \|x^*\| = \|x\|^2 \;\; \forall x \in \mathcal{A}. \tag{B.9}$$

Definition 12 *Let $\mathcal{A}$ be a $*$-algebra, then we say $a \in \mathcal{A}$ to be positive ($a \geq 0$) if there are finite elements $b_i \in \mathcal{A}$ such that:*

$$a = \sum_i b_i^* b_i. \tag{B.10}$$

This introduces a partial order on $\mathcal{A}$, while $a \geq b$ if $b - a \geq 0$.

Appendix C

Source code for training the QNN in python

C.1 Unsupervised learning tasks on graphs

This section explains the developed python code for the unsupervised learning tasks. Most of the algorithms were taken from [61], especially for the supervised task.

C.1.1 Package-imports and conventions

Packages

The following packages have been used in the code:

```
#math related packages
import scipy as sc;
import qutip as qt;
#further packages
import matplotlib.pyplot as plt;
```

Qubit-states

We also declare the basis for qubit states, due to their repetitive application:

```
#qubit states
qubit0 = qt.basis(2,0);
qubit1 = qt.basis(2,1);
#qubit density matrices
qubit0mat = qubit0 * qubit0.dag();
```

```
6 qubit1mat = qubit1 * qubit1.dag();
```

Conventions

Before implementing the algorithm, it is necessary to clarify how the input data sets are represented in Python. Our input data are a 1-dimensional list of random density matrices on $\bigotimes_{i=1}^{K} \mathbb{C}^2$:

$$initialStates = \left[\rho_1^{in}, \ldots, \rho_N^{in}\right]$$

The QNN architecture is represented by a 1-dimensional list:

$$qnnArch = [n_{in}, \ldots, n_{out}]$$

The initial network unitaries are arranged in a 2-dimensional list with an empty zeroth element, and thus the indices begin at the correct position:

$$initialUnitaries = \left[[\,], \left[U_1^1, \ldots, U_{n_1}^1\right], \ldots, \left[U_1^L, \ldots, U_{n_L}^L\right]\right],$$

while $U_m^l \in \mathcal{U}(\mathcal{H}_{n_{l-1}} \otimes \mathcal{H}_{n_l})$ should be unitary as in (Eq. 3.16).
The adjacency is given by a 2-dimensional list:

$$adjMatrix = [[[A]_{11}, \ldots, [A]_{1N}], \ldots, [[A]_{N1}, \ldots, [A]_{NN}]] \tag{C.1}$$

For sake of clarity, it is also useful to specify further inputs. The 'current QNN' is always meant as the architecture $[n_{in}, \ldots, n_{out}]$ and current list of unitaries:

$$currentUnitaries = \left[[\,], \left[U_1^1(s), \ldots, U_{n_1}^1(s)\right], \ldots, \left[U_1^L(s), \ldots, U_{n_L}^L(s)\right]\right], \tag{C.2}$$

updated up to the current training step. The current output is similar:

$$currentOutput = \left[\rho_1^{out}(s), \ldots, \rho_N^{out}(s)\right]. \tag{C.3}$$

C.1.2 Utility code and helper functions

It is also useful to introduce helper functions to generate the data sets and some recurring trace operations. In the following, every input variable of an algorithm mentioned for the first time in the commentary is indicated by square brackets.

Trace operations

partialTraceRem takes a quantum object (qt.Obj) [*obj*] on $\mathcal{H}_0 \otimes \ldots \otimes \mathcal{H}_{m-1}$ and a list of integers between 0 and $m-1$ [*rem*] to build the partial trace over all Hilbert spaces with index:

$$\{0, 1, \ldots, m-1\} \setminus rem.$$

```
def partialTraceRem(obj, rem):
    #prepare keep list
    rem.sort(reverse=True);
    keep = list(range(len(obj.dims[0])));
    for x in rem:
        keep.pop(x);
    res = obj;
    #take partial trace if there are more qubits in obj then indices in keep
    if len(keep) != len(obj.dims[0]):
        res = obj.ptrace(keep);
    #return partial trace or object
    return res;
```

partialTraceKeep takes a quantum object [*obj*] on $\mathcal{H}_0 \otimes \ldots \otimes \mathcal{H}_{m-1}$ and a list of integers between 0 and $m-1$ [*keep*] to build the partial trace over all Hilbert spaces with index in *keep*.

```
def partialTraceKeep(obj, keep):
    res = obj;
    #take partial trace if there are more qubits in obj then indices in keep
    if len(keep) != len(obj.dims[0]):
        res = obj.ptrace(keep);
    #return partial trace or object
    return res;
```

If $\{0, 1, \ldots, m-1\} = keep$ or *rem* booth functions return *obj*. This will be important for single qubit layers in the QNN.

Basic operations

swappedOp takes a quantum object $[obj]$ on $\mathcal{H}_1 \otimes \ldots \otimes \mathcal{H}_m$ and swaps the i^{th} operator with the j^{th}

$$A_1 \otimes \ldots \otimes A_i \otimes \ldots \otimes A_j \otimes \ldots \otimes A_m \mapsto A_1 \otimes \ldots \otimes A_j \otimes \ldots \otimes A_i \otimes \ldots \otimes A_m.$$

```
def swappedOp(obj, i, j):
    numberOfQubits = len(obj.dims[0]);
    #create a ordered list of Hilbert space indices
    permute = list(range(numberOfQubits));
    #exchange i and j in permute
    permute[i], permute[j] = permute[j], permute[i];
    #return permuted operators in tensor product of obj
    return obj.permute(permute);
```

This operation is equivalent to:

$$obj \mapsto \mathbf{SWAP}_K[i,j] \cdot obj \cdot \mathbf{SWAP}_K[i,j],$$

with the number of qubits in obj K.

Helper functions

tensoredId tensores N times the identity on $\mathbb{C}^2$.

```
def tensoredId(N):
    #make Identity matrix
    res = qt.qeye(2**N);
    #make dims list
    dims = [2 for i in range(N)];
    dims = [dims.copy(), dims.copy()];
    res.dims = dims;
    #return
    return res;
```

tensoredQubit0 tensores N times *qubit0mat*.

```
def tensoredQubit0(N):
    #make Qubit matrix
    res = qt.fock(2**N).proj();
    #make dims list
    dims = [2 for i in range(N)];
    dims = [dims.copy(), dims.copy()];
    res.dims = dims;
    #return
    return res;
```

unitariesCopy copies the list of unitaries [*unitaries*].

```
def unitariesCopy(unitaries):
    #create copy of unitaries
    newUnitaries = [];
    for layer in unitaries:
        newLayer = [];
        for unitary in layer:
            newLayer.append(unitary.copy());
        newUnitaries.append(newLayer);
    #return
    return newUnitaries;
```

randumQubitUnitary returns a unitary $2^{numQubits} \times 2^{numQubits}$ matrix as a quantum object on $\bigotimes_{i=1}^{numQubits} \mathbb{C}^2$ out of a normal distribution.

```
def randomQubitUnitary(numQubits):
    dim = 2**numQubits;
    #make unitary matrix
    res = sc.random.normal(size=(dim,dim))+ 1j * sc.random.normal(size=(dim,dim));
    res = sc.linalg.orth(res);
    res = qt.Qobj(res);
    #make dims list
    dims = [2 for i in range(numQubits)];
```

```
    dims = [dims.copy(), dims.copy()];
    res.dims = dims;
    #return
    return res;
```

randomQubitDens returns a density operator for a random pure state in $\bigotimes_{i=1}^{numQubits} \mathbb{C}^2$ out of a normal distribution.

```
def randomQubitDens(numQubits):
    dim = 2**numQubits
    #make normalized state
    res = sc.random.normal(size=(dim,1)) + 1j * sc.random.normal(size=(dim,1));
    res = (1/sc.linalg.norm(res)) * res;
    res = qt.Qobj(res);
    #make dims list
    dims1 = [2 for i in range(numQubits)];
    dims2 = [1 for i in range(numQubits)];
    dims = [dims1, dims2];
    res.dims = dims;
    #return density operator of normalized state
    return res * res.dag();
```

randomMixedQubitDens returns a density operator for a random mixed state in $\bigotimes_{i=1}^{numQubits} \mathbb{C}^2$ out of a pure density operator on $\bigotimes_{i=1}^{numQubits+1} \mathbb{C}^2$.

```
def randomMixedQubitDens(numQubits):
    #create random pure density operator of numQubits+1 qubits
    res=randomQubitDens(numQubits+1);
    #return partial trace over single qubit of res
    return(partialTraceRem(res,[0]));
```

randimInitialNetworkUnitaries takes an architecture [*qnnArch*] and returns a 2-dimensional list of random unitaries as in *initalUnitaries*.

```
def randomInitialNetworkUnitaries(qnnArch):
```

```
    unitaries = [[]];
    for l in range(1, len(qnnArch)):
        numInputQubits = qnnArch[l-1];
        numOutputQubits = qnnArch[l];
        layer = [];
        #create inital unitaries of layer l
        for m in range(numOutputQubits):
                #create unitary on numInputQubits + 1 qubits
            unitary = randomQubitUnitary(numInputQubits + 1);
            if numOutputQubits-1 != 0:
                #extend unitary to numInputQubits+numOutputQubits qubits
                unitary = qt.tensor(unitary, tensoredId(numOutputQubits-1));
                #only effect the mth output qubit
                unitary = swappedOp(unitary, numInputQubits, numInputQubits + m);
            layer.append(unitary);
        #add layer to list of unitaries
        unitaries.append(layer);
    #return
    return(unitaries);
```

C.1.3 QNN-Code

The next subsection contains the implementation of the QNN layer propagation, calculation of the cost function and update matrix.

Cost function

costFunktionUsv takes a list of density matrices [*currentOutput*] assigned to a graph adjacency [*adjMatrix*] and calculates the unsupervised cost function (Eq.3.19).

```
def costFunctionUsv(adjMatrix, currentOutput):
    costSum = 0;
    for i in range(len(adjMatrix[0])):
        for j in range(len(adjMatrix[0])):
```

```
            if adjMatrix[i][j] != 0:
                #calculate unsupervised cost function
                costSum += adjMatrix[i][j]*((currentOutput[i]-currentOutput[j])
                                            *(currentOutput[i]-currentOutput[j]));
    #return trace of costSum
    return costSum.tr();
```

Layer-Channel

makeLayerChannel takes the current QNN [*qnnArch*, *currentUnitaries*] and a density matrix[1] [*state*] on $\mathcal{H}_{l-1}$ and applies the l^{th} layer transition $\mathcal{E}^{(l)}$ (Eq.3.10) with respect to the given QNN.

```
def makeLayerChannel(qnnArch, currentUnitaries, l, state):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];
    #extend input state to numInputQubits+numOutputQubits qubits
    stateExt = qt.tensor(state, tensoredQubit0(numOutputQubits));
    #calculate layer unitary
    layerUni = currentUnitaries[l][0].copy();
    for i in range(1, numOutputQubits):
        layerUni = currentUnitaries[l][i] * layerUni;
    #return layer output
    return partialTraceRem(layerUni * stateExt * layerUni.dag(),
                                            list(range(numInputQubits)));
```

makeAdjointLayerChannel takes the current QNN [*qnnArch*, *currentUnitaries*] and a density matrix[2] [*state*] on $\mathcal{H}_l$ and applies the l^{th} adjoint layer channel (Eq.3.34) with respect to the given QNN:

$$\mathcal{G}_s^{(l)}(\rho^{(l)}(s)) = \operatorname*{tr}_l \left\{ (\mathbb{I}_{l-1} \otimes |0,\ldots,0\rangle_l \langle 0,\ldots,0|)\, {U^{(l)}}^{\dagger}(s)(\mathbb{I}_{l-1} \otimes \rho^{(l)}(s))U^{(l)}(s) \right\}.$$

```
def makeAdjointLayerChannel(qnnArch, currentUnitaries, l, state):
    numInputQubits = qnnArch[l-1];
```

[1] in general the output of the $(l-1)^{th}$ layer transition from the current training step $\rho^{(l-1)}(s)$

[2] in general the output of the $(l)^{th}$ layer transition from the current training step $\rho^{(l)}(s)$

```
    numOutputQubits = qnnArch[l];
    #prepare both states in the adjoint channel
    inputId = tensoredId(numInputQubits);
    state1 = qt.tensor(inputId, tensoredQubit0(numOutputQubits));
    state2 = qt.tensor(inputId, state);
    #calculate layer unitary
    layerUni = currentUnitaries[l][0].copy();
    for i in range(1, numOutputQubits):
        layerUni = currentUnitaries[l][i] * layerUni;
    #return adjoined layer output
    return partialTraceKeep(state1 * layerUni.dag() * state2 * layerUni,
                                                    list(range(numInputQubits)));
```

Feedforward

feedforward takes the current QNN [*qnnArch*, *currentUnitaries*] and the list of initial states [*initalStates*] and calculates the output for every layer $\rho^{(l)}(s)$.

```
def feedforward(qnnArch, currentUnitaries, initialStates):
    storedStates = [];
    for x in range(len(initialStates)):
        state = initialStates[x];
        layerwiseList = [state];
        #applys the lth layer channel on the xth input state
        for l in range(1, len(qnnArch)):
            state = makeLayerChannel(qnnArch, currentUnitaries, l, state);
            layerwiseList.append(state);
        #add the layer output to storedStates
        storedStates.append(layerwiseList);
    #return
    return storedStates;
```

The output is a 2-dimensional list consisting of every layer output $\rho_i^{(l)}(s)$ for each input state:

$$storedStates = \left[\left[\rho_1^{in}, \rho_1^{(1)}(s), \ldots, \rho_1^{(L)}(s)\right], \ldots, \left[\rho_N^{in}, \rho_N^{(1)}(s), \ldots, \rho_N^{(L)}(s)\right]\right]$$

Update matrix

makeUpdateMatrixUsv takes the current QNN [*qnnArch*, *currentUnitaries*], the Lagrange parameter λ [*lda*], the current output [*currentOutput*], the graph adjacency [*adjMatrix*] and the current list of stored states from Feedforward [*storedStates*] and calculates $K_m^l(s)$ on $\mathcal{H}_{l-1} \otimes \mathbb{C}^2$ (Eq.3.30).

```
def makeUpdateMatrixUsv(qnnArch, currentUnitaries, lda, currentOutput,
                                            adjMatrix, storedStates, l, m):
    numInputQubits = qnnArch[l-1];
    sumUpdate = 0;
#calculate the update matrix by using updateMatrixFirstPart and updateMatrixSecondPart
    for i in range(len(adjMatrix[0])):
        for j in range(len(adjMatrix[0])):
            if adjMatrix[i][j] !=0:
                                #calculate the commutator
                firstPart = updateMatrixFirstPart(qnnArch, currentUnitaries,
                                    storedStates[i][l-1]-storedStates[j][l-1], l, m);
                secondPart = updateMatrixSecondPart(qnnArch, currentUnitaries,
                                            currentOutput[i]-currentOutput[j], l, m);
                mat = qt.commutator(firstPart, secondPart);
                #trace over rest-Hilbert space
                keep = list(range(numInputQubits));
                keep.append(numInputQubits + m);
                mat = partialTraceKeep(mat, keep);
                #add up all parts of K
                sumUpdate = sumUpdate + (adjMatrix[i][j] * mat);
    #multiply the prefactors of the K-matrix
    sumUpdate = ((0 +1j) * (2**(numInputQubits+1))/lda) * sumUpdate;
    #return
    return sumUpdate;
```

updateMatrixFirstPart takes the current QNN [*qnnArch*, *currentUnitaries*] and a density operator [*state*] on $\mathcal{H}_{l-1}$ and calculates

$$U_m^l(s) \dots U_1^l(s) \left((state) \otimes |0, \dots, 0\rangle_l \langle 0, \dots, 0| \right) U_1^{l\dagger}(s) \dots U_m^{l\ \dagger}(s).$$

```
def updateMatrixFirstPart(qnnArch, currentUnitaries, state, l, m):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];
    #extend the state to numInputQubits + numOutputQubits qubits
    stateExt = qt.tensor(state, tensoredQubit0(numOutputQubits));
    #calculate product of unitaries
    productUni = currentUnitaries[l][0].copy();
    for x in range(1, m+1):
        productUni = currentUnitaries[l][x] * productUni;
    #multiply and return
    return productUni * stateExt * productUni.dag();
```

updateMatrixSecondPart takes the current QNN [*qnnArch*, *currentUnitaries*] and a density operator *state* on $\mathcal{H}_L$ and calculates

$$U_{m+1}^{l}{}^{\dagger}(s)\dots U_{n_l}^{l}{}^{\dagger}(s)\left(\mathbb{I}_{l-1}\otimes\sigma_s^{(l)}(state)\right)U_{n_l}^{l}(s)\dots U_{m+1}^{l}(s)$$

by using makeAdjointLayerChannel.

```
def updateMatrixSecondPart(qnnArch, currentUnitaries, state, l, m):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];
    #calculate sigma
    stateExt = state;
    for y in range(len(qnnArch)-1,l,-1):
        stateExt = makeAdjointLayerChannel(qnnArch, currentUnitaries, y, stateExt);
    stateExt = qt.tensor(tensoredId(numInputQubits), stateExt);
    #calculate product of unitaries
    productUni = tensoredId(numInputQubits + numOutputQubits);
    for x in range(m+1, numOutputQubits):
        productUni = currentUnitaries[l][x] * productUni;
    #multiply and return
    return productUni.dag() * stateExt * productUni;
```

addUpdateMatrixUsv takes the current QNN [*qnnArch*, *currentUnitaries*], the Lagrange parameter λ [*lda*], the step size ϵ [*ep*], the current output [*currentOutput*], the graph adjacency [*adjMatrix*] and the list of stored states [*storedStates*] from feedforward and calculates $e^{i\epsilon K_m^l(s)}$ on $\mathcal{H}_{l-1} \otimes \mathbb{C}^2$.

```
def addUpdateMatrixUsv(qnnArch, currentUnitaries, lda, ep, currentOutput,
                                                   adjMatrix, storedStates, l, m):
    res = makeUpdateMatrixUsv(qnnArch, currentUnitaries, lda, currentOutput,
                                                   adjMatrix, storedStates, l, m);
    #return
    return ((0 + 1j)* ep * res).expm();
```

makeUpdateMatrixTensored takes an architecture [*qnnArch*] and the updating matrix [*updateMatrix*] from AddUpdateMatrixUsv and extends $e^{i\epsilon K_m^l(s)}$ to $\mathcal{H}_{l-1} \otimes \mathcal{H}_l$.

```
def makeUpdateMatrixTensored(updateMatrix, qnnArch, l, m):
    numInputQubits = qnnArch[l-1];
    numOutputQubits = qnnArch[l];
    #extend to numInputQubits+numOutputQubits qubits
    if numOutputQubits-1 != 0:
        updateMatrix = qt.tensor(updateMatrix, tensoredId(numOutputQubits-1));
    #return updateMatrix effecting the mth output qubit
    return swappedOp(updateMatrix, numInputQubits, numInputQubits + m);
```

C.1.4 Training of the QNN

qnnTrainingUsvtakes the initial QNN [*qnnArch*, *initialUnitaries*], the Lagrange parameter λ [*lda*], the step size ϵ [*ep*], the initial states [*initialStates*], the graph adjacency [*adjMatrix*] and the number of training rounds [*trainingRounds*] and teaches the unsupervised training task to a QNN.

```
def qnnTrainingUsv(qnnArch, lda, ep, initialStates, adjMatrix,
                                        initialUnitaries, trainingRounds):
    #set training step
    s = 0;
    #feedforward for given unitaries
    currentUnitaries = initialUnitaries;
```

```
    storedStates = feedforward(qnnArch, currentUnitaries, initialStates);
    #calculation of cost function for initial unitaries
    outputStates = [];
    for x in range(len(storedStates)):
        outputStates.append(storedStates[x][-1]);
    costValueUsv = costFunctionUsv(adjMatrix, outputStates);
    plotList = [[s, costValueUsv]];
    #store costValueUsv for bug check
    prevValUsv = costValueUsv;
    #begin training of the QNN
    for n in range(trainingRounds):
        print(n);
        newUnitaries = unitariesCopy(currentUnitaries);
        #loop over layers:
        for l in range(1, len(qnnArch)):
            numInputQubits = qnnArch[l-1];
            numOutputQubits = qnnArch[l];
            #loop over perceptrons
            for m in range(numOutputQubits):
                updateMatrix = AddUpdateMatrixUsv(qnnArch, currentUnitaries,
                            lda, ep, outputStates, adjMatrix, storedStates, l, m);
                newUnitaries[l][m] = makeUpdateMatrixTensored(updateMatrix,
                                        qnnArch, l, m) * currentUnitaries[l][m];
        #update s
        s = s + ep;
        #feedforward for current unitaries
        currentUnitaries = newUnitaries;
        storedStates = feedforward(qnnArch, currentUnitaries, initialStates);
        #calculation of cost function for current unitaries
        outputStates = [];
        for x in range(len(storedStates)):
            outputStates.append(storedStates[x][-1]);
```

```
        costValueUsv = costFunctionUsv(adjMatrix, outputStates);
        plotList.append([s, costValueUsv]);
        #bug check, break if cost function increases
        if costValueUsv > prevValUsv:
            break;
        prevValUsv = costValueUsv;
    #return cost function values, current unitaries and output states
    return [plotList, currentUnitaries, outputStates];
```

C.1.5 Additional algorithms for semi-supervised training

This section provides the additional algorithms needed to train a QNN for a semi-supervised task. The additional labelled training set is:

$$trainingData = \left[\left[\rho_1^{in}, \rho_1^{sv}\right], \ldots, \left[\rho_R^{in}, \rho_R^{sv}\right]\right]. \tag{C.4}$$

So in terms of semi-supervised learning tasks on graphs, we assume that the supervised training data is given for the first R vertices of the graph.

Cost function

costFunktionSv takes a list of density matrices [*currentOutput*] and the training data [*trainingData*] to calculate the supervised cost function (Eq.3.39).

```
def costFunctionSv(trainingData, currentOutput):
    lossSum = 0;
    #calculate supervised cost function
    for i in range(len(trainingData)):
        lossSum += trainingData[i][1].dag() * currentOutput[i] * trainingData[i][1];
    #return trace of costSum
    return lossSum.tr()/len(trainingData);
```

Update matrix for supervised and semi-supervised training

makeUpdateMatrixSv takes the current QNN [*qnnArch*, *currentUnitaries*], the Lagrange parameter λ [*lda*], the current output [*currentOutput*], the training Data [*trainingData*] and the stored states [*storedStates*] from feedforward to calculate $K_m^l(s)$ on $\mathcal{H}_{l-1} \otimes \mathbb{C}^2$ (Eq.3.42).

```
def makeUpdateMatrixSv(qnnArch, currentUnitaries, lda, trainingData, storedStates,
                                                                        l, m):
    numInputQubits = qnnArch[l-1]
    sumUpdate = 0;
#calculate the update matrix by using updateMatrixFirstPart and updateMatrixSecondPart
    for x in range(len(trainingData)):
        #calculate the commutator
        firstPart = updateMatrixFirstPart(qnnArch, currentUnitaries,
```

```
                                                storedStates[x][l-1], l, m);
        secondPart = updateMatrixSecondPart(qnnArch, currentUnitaries,
                                                trainingData[x][1], l, m);
        mat = qt.commutator(firstPart, secondPart);
        #trace over rest-Hilbert space
        keep = list(range(numInputQubits));
        keep.append(numInputQubits + m);
        mat = partialTraceKeep(mat, keep);
        #add up all parts of K
        sumUpdate = sumUpdate + mat;
    #multiply the prefactors of the K-matrix
    sumUpdate = ((0 + 1j) * 2**numInputQubits)/(lda *len(trainingData)) * sumUpdate;
    #return
    return sumUpdate;
```

AddUpdateMatrixSsv takes the current QNN [*qnnArch*, *currentUnitaries*], the Lagrange parameter λ [*lda*], the step size ϵ [*ep*], the current output [*currentOutput*], the graph adjacency [*adjMatrix*], the training data [*trainingData*] and the stored states [*storedStates*] from feedforward to calculate the unsupervised and supervised update matrix and combine them into the semi-supervised update $e^{i\epsilon K_m^l(s)}$ on $\mathcal{H}_{l-1} \otimes \mathbb{C}^2$ (Eq.3.44).

```
def AddUpdateMatrixSsv(qnnArch, currentUnitaries, lda, ep, gamma, currentOutput,
                                    adjMatrix, trainingData, storedStates, l, m):
    #calculate supervised and unsupervised update matrix
    res = makeUpdateMatrixSv(qnnArch, currentUnitaries, lda, trainingData,
    storedStates, l, m) + gamma * makeUpdateMatrixUsv(qnnArch, currentUnitaries,
                        lda, currentOutput, adjMatrix, storedStates, l, m);
    #return
    return ((0 + 1j)* ep * res).expm();
```

Training algorithm for semi-supervised learning

qnnTrainingSsv takes the initial QNN [*qnnArch*, *initalUnitaries*], the Lagrange parameter λ [*lda*], the step size ϵ [*ep*], the initial states [*initialStates*], the graph adjacency [*adjMatrix*], the training data [*trainingData*]

and the number of training rounds [*trainingRounds*] and teaches the semi-supervised training task to a QNN.

```
def qnnTrainingSsv(qnnArch, lda, ep, gamma, initialStates, adjMatrix,
                           initialUnitaries, trainingData, trainingRounds):
    #set training step
    s = 0;
    #feedforward for given unitaries
    currentUnitaries = initialUnitaries;
    storedStates = feedforward(qnnArch, currentUnitaries, initialStates);
    #calculation of cost function for initial unitaries
    outputStates = [];
    for x in range(len(storedStates)):
        outputStates.append(storedStates[x][-1]);
    #supervised cost function
    costValueSv = costFunctionSv(trainingData, outputStates);
    #unsupervised cost function
    costValueUsv = costFunctionUsv(adjMatrix, outputStates);
    #semi-supervised cost function
    costValueFull = costValueSv + gamma * costValueUsv;
    plotListFull = [[s,costValueFull, costValueSv, gamma * costValueUsv]];
    #store costValueFull for bug check
    prevValFull = costValueFull;
    #begin training the QNN
    for n in range(trainingRounds):
        newUnitaries = unitariesCopy(currentUnitaries);
        #loop over layers:
        for l in range(1, len(qnnArch)):
            numInputQubits = qnnArch[l-1];
            numOutputQubits = qnnArch[l];
            #loop over perceptrons
            for m in range(numOutputQubits):
                updateMatrix = AddUpdateMatrixSsv(qnnArch, currentUnitaries,
```

```
                lda, ep, gamma, outputStates, adjMatrix, trainingData,
                                                        storedStates, l, m);

                newUnitaries[l][m] = makeUpdateMatrixTensored(updateMatrix,
                                    qnnArch, l, m) * currentUnitaries[l][m];
        #update s
        s = s + ep;
        #feedforward for current unitaries
        currentUnitaries = newUnitaries;
        storedStates = feedforward(qnnArch, currentUnitaries, initialStates);
        #calculation of cost function for current unitaries
        outputStates = [];
        for x in range(len(storedStates)):
            outputStates.append(storedStates[x][-1]);
        #supervised cost function
        costValueSv = costFunctionSv(trainingData, outputStates);
        #unsupervised cost function
        costValueUsv = costFunctionUsv(adjMatrix, outputStates);
        #semi-supervised cost function
        costValueFull = costValueSv + gamma * costValueUsv;
        plotListFull.append([s, costValueFull,  costValueSv, gamma * costValueUsv]);
        #bug check, break if semi-supervised cost function decreases
        if costValueFull < prevValFull:
            break;
        prevValFull = costValueFull;
    #return all cost function values, current unitaries and output states
    return [plotListFull, currentUnitaries, outputStates];
```

C.1.6 Additional algorithms for the generalization task

This section provides the additional algorithms needed to train a QNN for the generalization task in section 3.7. In this case we do not sort the training data according to the labels, instead we split the training data in to a γ_1 and a γ_2 set and add a list of indices to which vertex of the graph they belong:

$$trainingData1 = [\underbrace{[0,4]}_{\text{list of indeces}}, [[\rho_1^{in}, \rho_1^{sv}], \underbrace{0,0,0}_{\text{placeholder}}, [\rho_5^{in}, \rho_5^{sv}]]], \tag{C.5}$$

$$trainingData2 = [\underbrace{[2]}_{\text{list of indeces}}, [0, 0, [\rho_3^{in}, \rho_3^{sv}], 0, 0]]. \tag{C.6}$$

Furthermore, we have a third set of labeled data, which also contains the 'forgotten' labels

$$expect = [\underbrace{[0,1,2,3,4]}_{\text{list of indeces}}, [[\rho_1^{in}, \rho_1^{sv}], [\rho_2^{in}, \rho_2^{sv}], [\rho_3^{in}, \rho_3^{sv}], [\rho_4^{in}, \rho_4^{sv}], [\rho_5^{in}, \rho_5^{sv}]]]. \tag{C.7}$$

Then we have to modify the algorithms for the supervised cost function, the update matrix and the training algorithm.

Cost function

costFunktionSvGen takes a list of density matrices [*currentOutput*] and the training data according to a γ [*trainingData*] to calculate the supervised cost function (Eq. 3.45) for each state in assigned to an index in the list of indices.

```
1 def costFunctionSvGen(trainingData, currentOutput):
2     costSum = 0;
3     #calculates the cost function for each state in the index set
4     for i in trainingData[0]:
5         costSum += trainingData[1][i][1].dag() * currentOutput[i]
6                                               * trainingData[1][i][1];
7     #return
8     return costSum.tr();
```

Update matrix generalization task

makeUpdateMatrixSvGen takes the current QNN [*qnnArch*, *currentUnitaries*], the Lagrange parameter λ [*lda*], the current output [*currentOutput*], the training Data [*trainingData*] and the stored states [*storedStates*] from feedforward to calculate $K_m^l(s)$ on $\mathcal{H}_{l-1} \otimes \mathbb{C}^2$ (Eq.3.42).

```
def makeUpdateMatrixSvGen(qnnArch, currentUnitaries, lda, trainingData, storedStates,
                                                                                l, m):
    numInputQubits = qnnArch[l-1]
    sumUpdate = 0;
#calculate the update matrix by using updateMatrixFirstPart and updateMatrixSecondPart
    #only for the states assigned to an index in the list of indices
    for x in trainingData[0]:
        #calculate the commutator
        firstPart = updateMatrixFirstPart(qnnArch, currentUnitaries,
                                          storedStates[x][l-1], l, m);
        secondPart = updateMatrixSecondPart(qnnArch, currentUnitaries,
                                         trainingData[1][x][1], l, m);
        mat = qt.commutator(firstPart, secondPart);
        #trace over rest-Hilbert space
        keep = list(range(numInputQubits));
        keep.append(numInputQubits + m);
        mat = partialTraceKeep(mat, keep);
        #add up all parts of K
        sumUpdate = sumUpdate + mat;
    #multiply the prefactors of the K-matrix
    sumUpdate = ((0 + 1j) * 2**numInputQubits)/(lda) * sumUpdate;
    #return
    return sumUpdate;
```

AddUpdateMatrixGen takes the current QNN [*qnnArch*, *currentUnitaries*], the Lagrange parameter λ [*lda*], the step size ϵ [*ep*], the current output [*currentOutput*], the graph adjacency [*adjMatrix*], the two training data sets [*trainingData1*, *trainingData2*], the two parameters γ_1, γ_2 [*gamma1*,*gamma2*] and the stored states [*storedStates*] from feedforward to calculate the update matrix for the generalization task $e^{i\epsilon K^l_m(s)}$ on $\mathcal{H}_{l-1} \otimes \mathbb{C}^2$ (Eq.??).

```
def AddUpdateMatrixGen(qnnArch, currentUnitaries, lda, ep, gamma1, gamma2,
    currentOutput, adjMatrix, trainingData1, trainingData2, storedStates, l, m):
    #calculate supervised and unsupervised update matrix
```

```
    res = makeUpdateMatrixUsv(qnnArch, currentUnitaries, lda, currentOutput,
    adjMatrix, storedStates, l, m) + gamma1 * makeUpdateMatrixSvGen(qnnArch,
    currentUnitaries, lda, trainingData1, storedStates, l, m) + gamma2
    * makeUpdateMatrixSvGen(qnnArch, currentUnitaries, lda, trainingData2,
    storedStates, l, m);
    #return
    return ((0 + 1j) * ep * res).expm();
```

Training algorithm for generalization task

qnnTrainingGen takes the initial QNN [*qnnArch*, *initalUnitaries*], the Lagrange parameter λ [*lda*], the step size ϵ [*ep*], the initial states [*initialStates*], the graph adjacency [*adjMatrix*], the two training data sets [*trainingData1*, *trainingData2*], the two parameters γ_1, γ_2 [*gamma1*,*gamma2*] and the number of training rounds [*trainingRounds*] and teaches the semi-supervised generalization task to a QNN. The algorithm also uses the set containing the 'forgotten' labels [*expect*] to check how well the generalization task is solved by the QNN.

```
def qnnTrainingSsv(qnnArch, lda, ep, gamma1, gamma2, initialStates, adjMatrix,
                               initialUnitaries, trainingData1, trainingData2,
                                                        trainingRounds, expect):


    #set training step
    s = 0;
    #feedforward for given unitaries
    currentUnitaries = initialUnitaries;
    storedStates = feedforward(qnnArch, currentUnitaries, initialStates);
    #calculation of cost function for initial unitaries
    outputStates = [];
    for x in range(len(storedStates)):
        outputStates.append(storedStates[x][-1]);
    #supervised cost function for gamma1
    costValueSv1 = costFunctionSvGen(trainingData1, outputStates);
    #supervised cost function for gamma2
```

```
    costValueSv2 = costFunctionSvGen(trainingData2, outputStates);
    #unsupervised cost function
    costValueUsv = costFunctionUsv(adjMatrix, outputStates);
    #semi—supervised cost function
    costValueFull = costValueUsv + gamma1 * costValueSv1 + gamma2 * costValueSv2;
    #check generalization
    genCost = costFunctionSvGen(expect, outputStates)
    plotListFull = [[s,costValueFull,costValueUsv, costValueSv1, costValueSv2,
                                                                        genCost]];
    #store costValueFull for bug check
    prevValFull = costValueFull;
    #begin training the QNN
    for n in range(trainingRounds):
        newUnitaries = unitariesCopy(currentUnitaries);
        #loop over layers:
        for l in range(1, len(qnnArch)):
            numInputQubits = qnnArch[l—1];
            numOutputQubits = qnnArch[l];
            #loop over perceptrons
            for m in range(numOutputQubits):
                updateMatrix = AddUpdateMatrixGen(qnnArch, currentUnitaries, lda, ep,
                gamma1, gamma2, currentOutput, adjMatrix, trainingData1, trainingData2,
                                                                storedStates, l, m);
                newUnitaries[l][m] = makeUpdateMatrixTensored(updateMatrix, qnnArch,
                                                        l, m)* currentUnitaries[l][m];
        #update s
        s = s + ep;
        #feedforward for current unitaries
        currentUnitaries = newUnitaries;
        storedStates = feedforward(qnnArch, currentUnitaries, initialStates);
        #calculation of cost function for current unitaries
        outputStates = [];
```

```
49          for x in range(len(storedStates)):
50              outputStates.append(storedStates[x][-1]);
51          #supervised cost function for gamma1
52          costValueSv1 = costFunctionSvGen(trainingData1, outputStates);
53          #supervised cost function for gamma2
54          costValueSv2 = costFunctionSvGen(trainingData2, outputStates);
55          #unsupervised cost function
56          costValueUsv = costFunctionUsv(adjMatrix, outputStates);
57          #semi-supervised cost function
58          costValueFull = costValueUsv + gamma1 * costValueSv1 + gamma2 * costValueSv2;
59          #check generalization
60          genCost = costFunctionSvGen(expect, outputStates)
61          plotListFull.append([s,costValueFull,costValueUsv, costValueSv1, costValueSv2,
62                                                                                  genCost]);
63          #bug check, break if semi-supervised cost function decreases
64          if costValueFull < prevValFull:
65              break;
66          prevValFull = costValueFull;
67      #return all cost function values, current unitaries and output states
68      return [plotListFull, currentUnitaries, outputStates];
```

Appendix D

Further results unsupervised training task

The following figures contain further test for unsupervised training a QNN. The results are discussed in Section 3.5. All test behaved similar, while the cost function is trained in all cases to zero.

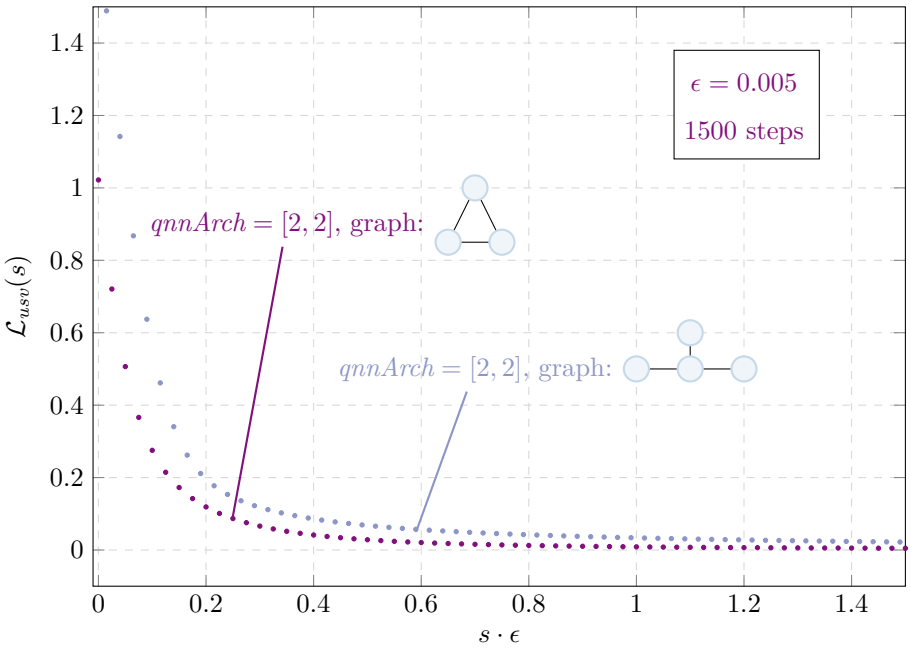


Figure D.1 Shows the cost function values for unsupervised training (Eq. 3.19) a $[2, 2]$-QNN with two feature graph-structured quantum data for two different types of graphs.

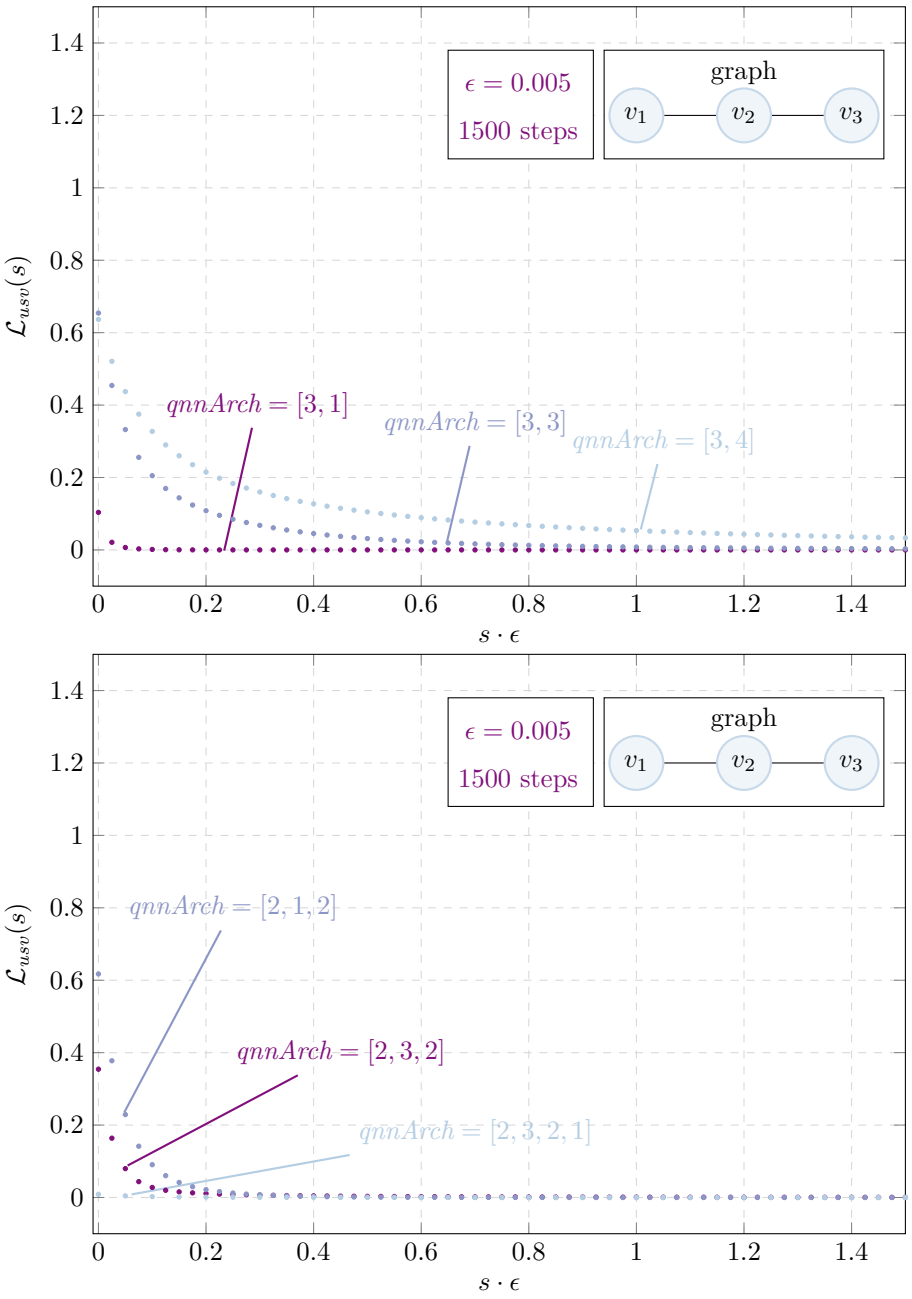


Figure D.2 Shows the cost function values for unsupervised training (Eq. 3.19) different QNNs (Eq. 3.36). The graph-structured quantum data in the first picture is equipped with three features and in the second with two features.

The QNN distinguishes between disconnected pieces of the graph:

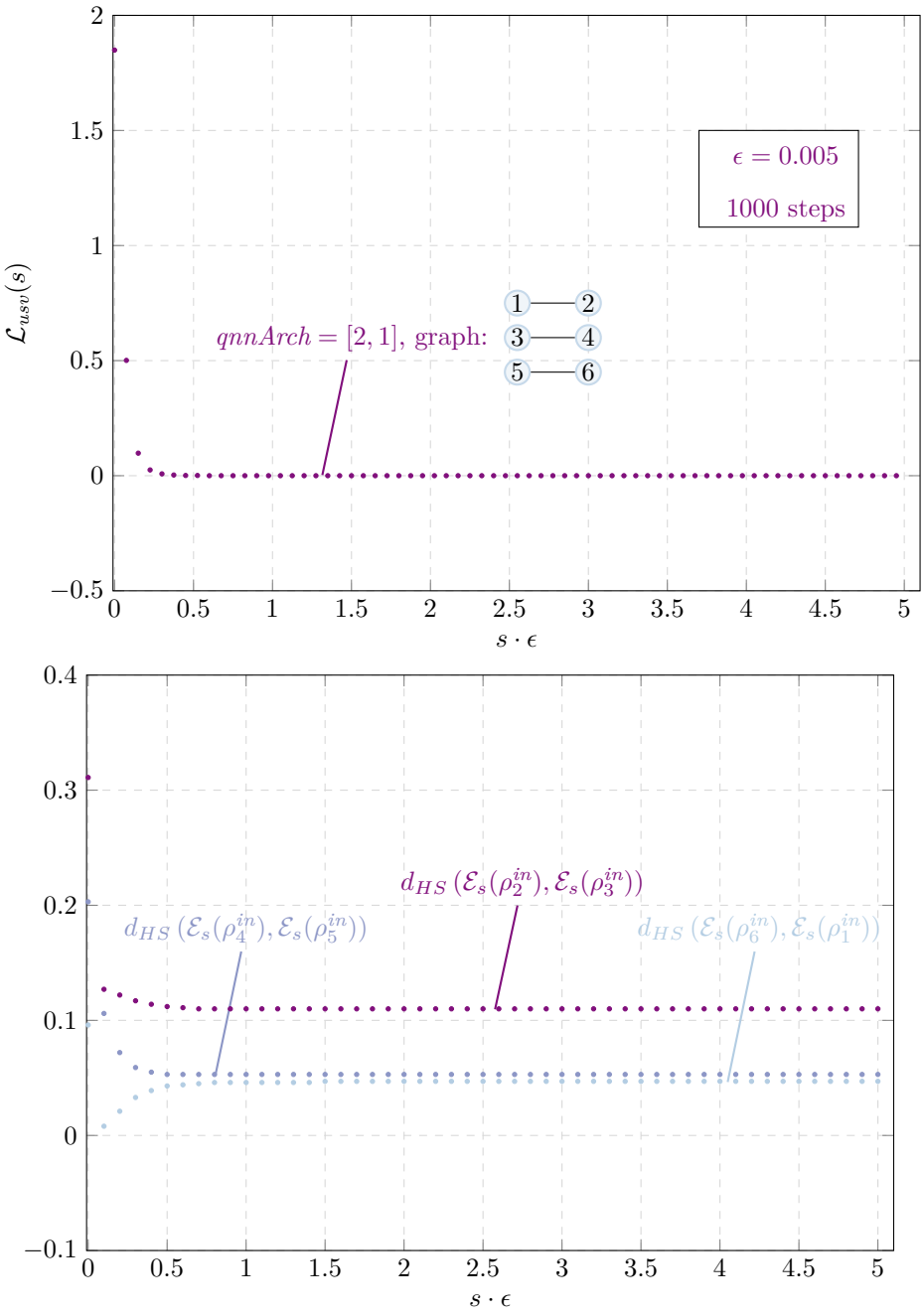


Figure D.3 Shows the cost function values for unsupervised training (Eq. 3.19) a $[2, 1]$-QNN for two features. The second picture shows the distances between the disconnected pieces of the graph.

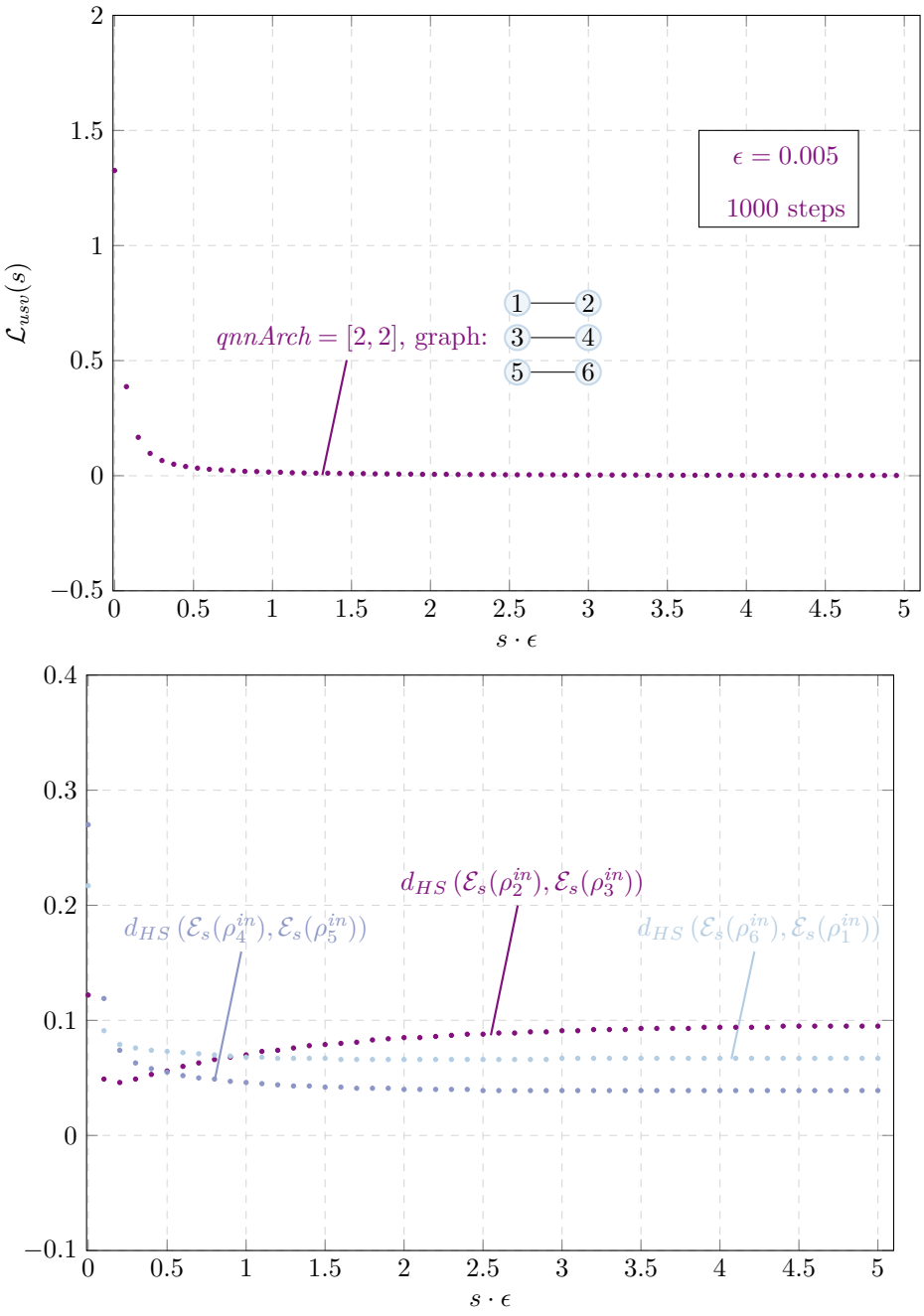


Figure D.4 Shows the cost function values for unsupervised training (Eq. 3.19) a $[2, 2]$-QNN for two features. The second picture shows the distances between the disconnected pieces of the graph.

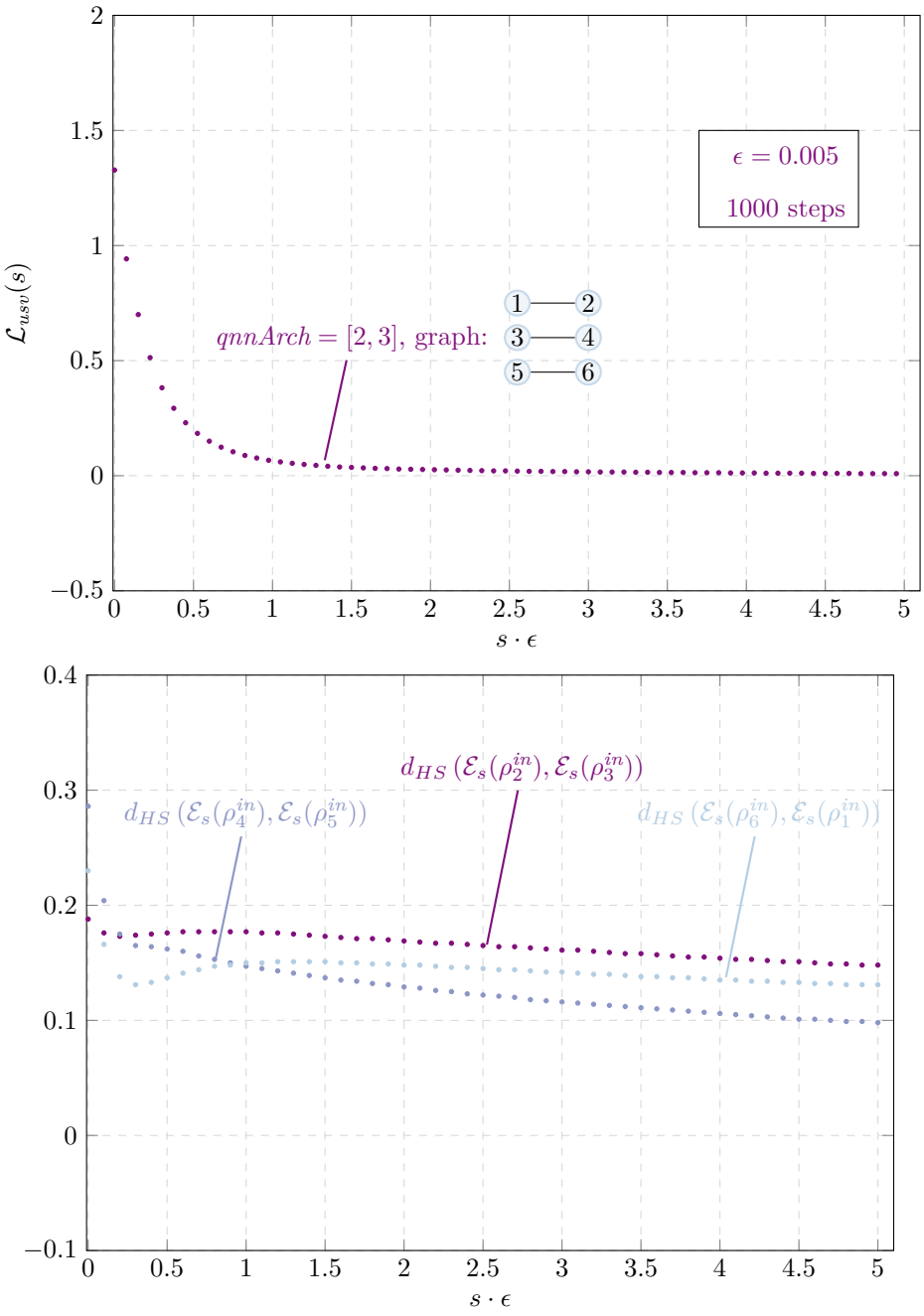


Figure D.5 Shows the cost function values for unsupervised training (Eq. 3.19) a $[2, 3]$-QNN for two features. The second picture shows the distances between the disconnected pieces of the graph.

Appendix E

Further results semi-supervised training task

The following figures contain further test for semi-supervised training a QNN. The results are discussed in Section 3.6. All test behaved similar, while the cost function was tired to be maximized. The conflict between supervised and unsupervised task prevent the cost function from being maximized to 1.

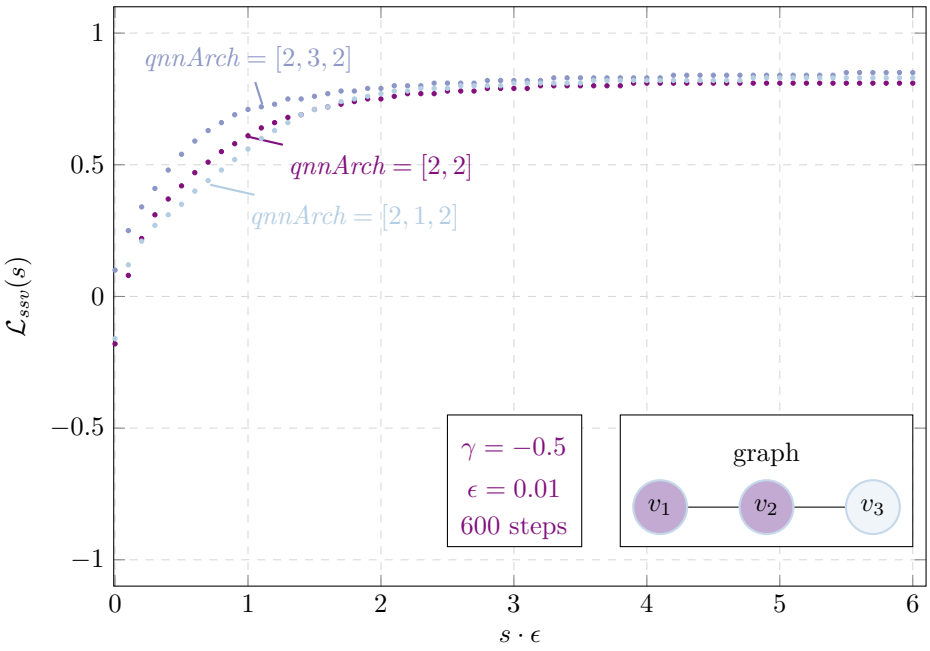


Figure E.1 Shows the values of the semi-supervised cost function (Eq. 3.40) for training three different QNNs with two feature input data and pure two feature labels.

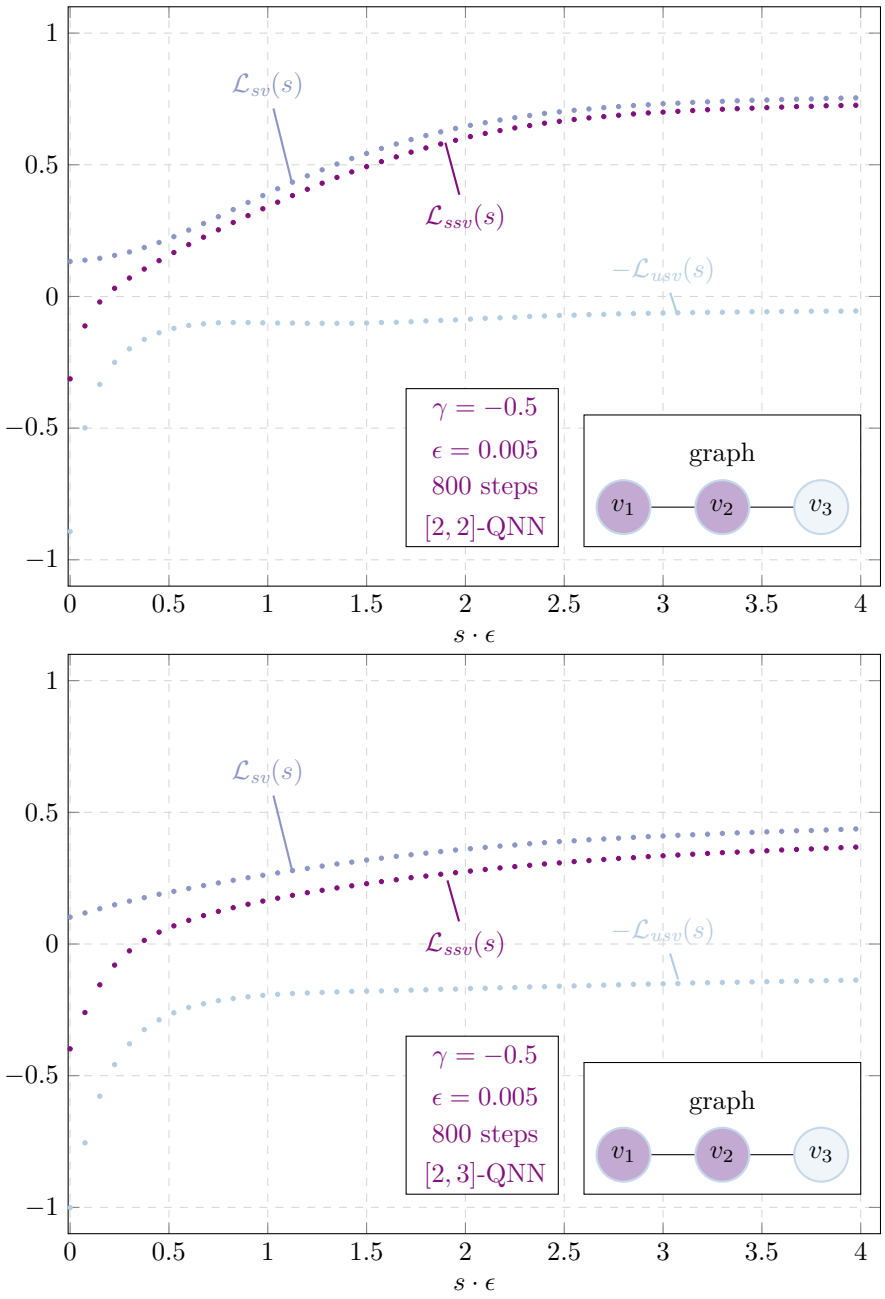


Figure E.2 Shows the cost function values for semi-supervised training (Eq. 3.40) a $[2, 2]$-QNN with two feature input data and pure two feature labels and training a $[2, 3]$-QNN with two feature input data and pure three feature labels.

The parameter γ controls the importance of the unsupervised task. Decreasing this parameter leads to an increase of the cost function:

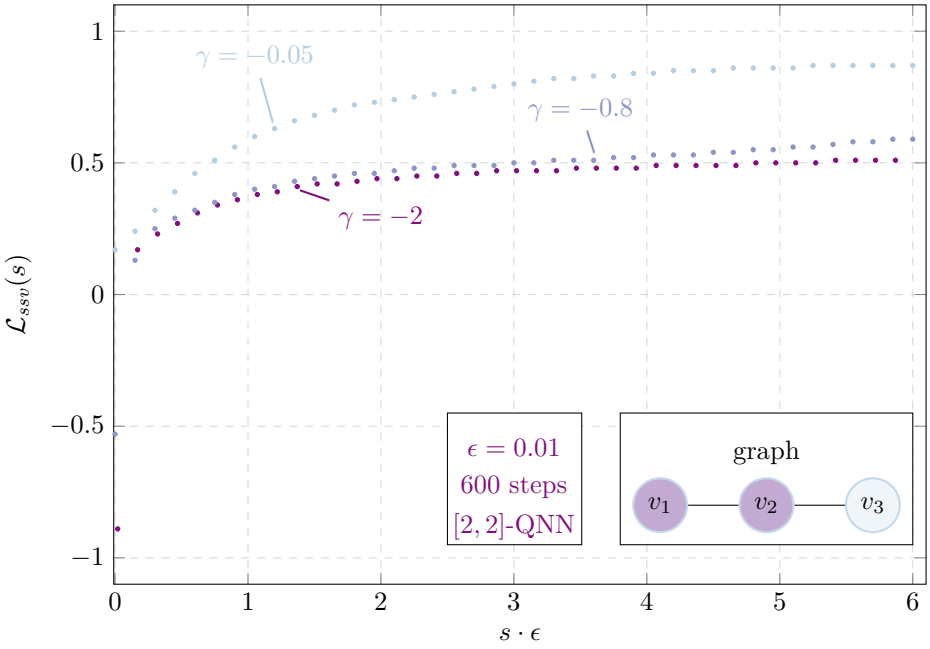


Figure E.3 Shows the values of the semi-supervised cost function (Eq. 3.40) training a $[2, 2]$-QNN with two feature input data and pure two feature labels for three different values of γ.

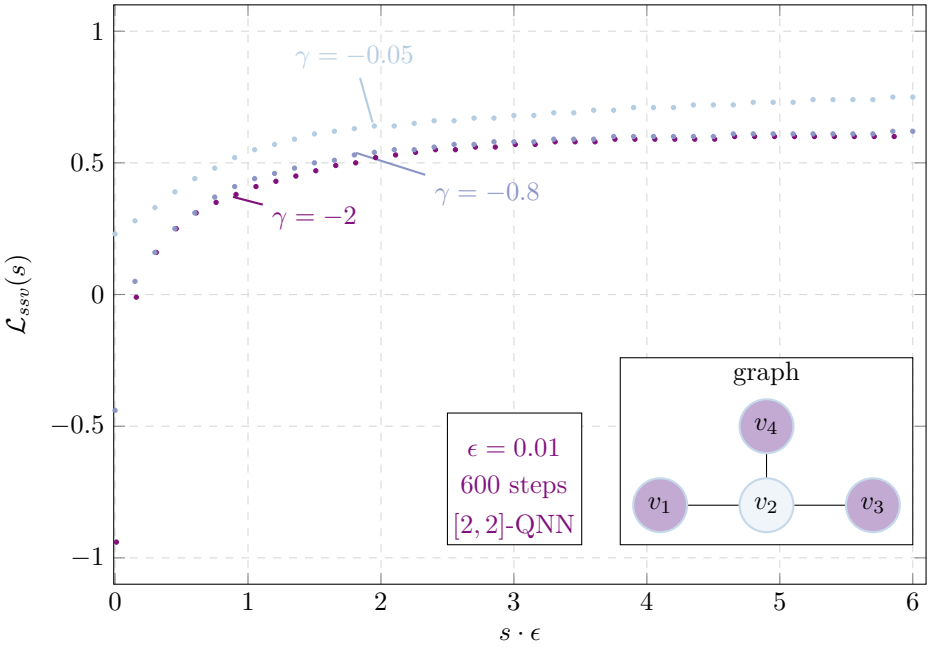


Figure E.4 Shows the values of the semi-supervised cost function (Eq. 3.40) for training a $[2, 2]$-QNN with two feature input data and pure two feature labels for three different values of γ.