Universiteit Leiden
The Netherlands

Bachelor Data Science and Artificial Intelligence

Weighing the Possibilities:

Scalable Simulation of Quantum Convolutional Neural Networks using Weighted Model Counting

Canavea Matei

Supervisors:
Thanos Dimitrios
Dr. Laarman Alfons

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/07/2026

Abstract

State-vector simulation of Quantum Convolutional Neural Networks (QCNNs) using simulators such as PennyLane becomes memory-infeasible beyond approximately 24 qubits on commodity hardware. The exponential simulation cost limits both the scale at which these circuits can be trained and the scale at which their behaviour can be validated. This thesis explores Weighted Model Counting (WMC) as an alternative approach to traditional circuit simulators via the Quokka# framework, with a quantitative comparison between Quokka# and PennyLane. We present the first implementation of QCNN circuits within Quokka#. Alongside the QCNN circuits, a hybrid pipeline is used for training and measuring both Quokka# and PennyLane. We perform two experiments, the first one measuring the scalability of the training process with a fixed 8-qubit architecture. The second experiment looks at how runtime scales with growing circuit size, starting from 4 qubits up to 128. On a single core, 8-qubit run, Quokka# requires roughly 10 times more wall time than PennyLane. However, once gradient computations are parallelised over 12 cores, the gap between Quokka# and PennyLane narrows to within roughly 1.4× of PennyLane. More importantly, runtime of Quokka# scales polynomially (with an empirically estimated exponent of approximately 1.4) for QCNNs of 20 qubits and above, allowing simulations of up to 128 qubit QCNNs within 5 seconds on standard hardware. Comparatively, state-vector simulations hit their limit at 24 qubits. Numerical consistency is validated through all experiments. These results demonstrate that WMC simulation with Quokka# enables scalable simulation of large-scale QCNN circuits, which were previously beyond the reach of conventional approaches.

Glossary

Term	Description	Type
θ	Trainable parameter vector	Vector
$\mathcal{L}(\theta)$	Cost function (Mean Squared Error)	Function
$\|\psi\rangle$	Quantum state	Vector
p_0	Probability of measuring the state $\|0\rangle$	Float

Abbreviations

AI	Artificial Intelligence
CNF	Conjunctive Normal Form
CNN	Convolutional Neural Network
CNOT	Controlled-NOT gate
CPU	Central Processing Unit
CRX	Controlled-R_X rotation gate
CRZ	Controlled-R_Z rotation gate
CZ	Controlled-Z gate
GPMC	Generic Probabilistic Model Counter (WMC solver)
GPU	Graphics Processing Unit
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
MPS	Matrix Product State
MSE	Mean Squared Error
NISQ	Noisy Intermediate-Scale Quantum
PCA	Principal Component Analysis
PQC	Parameterised Quantum Circuit
QASM	Quantum Assembly Language (OpenQASM)
QCNN	Quantum Convolutional Neural Network
RAM	Random Access Memory
SAT	Boolean Satisfiability
#SAT	Counting extension of SAT (model counting)
SSD	Solid State Drive
WMC	Weighted Model Counting

Contents

1 Introduction

Quantum computing is advancing rapidly, promising breakthroughs in the way we process information. Unlike classical computers that are restricted to binary bits, *quantum* bits, or *qubits*, exist in a *superposition* of both states simultaneously, allowing a system of n qubits to represent 2^n states at once. This so-called "quantum advantage" allows researchers to solve problems that are beyond the capabilities of classical hardware. [NC00, Pre18]

At the same time, Artificial Intelligence has undergone its own revolution. One of the biggest developments in the AI field was the introduction of Convolutional Neural Networks (CNNs), the foundation of modern image recognition, transforming what computers can learn from data. [KSH17]

It is natural, then, that researchers have begun blending these two domains, adapting the powerful hierarchical structure of CNNs for quantum hardware [CCL19]. Yet this promising combination faces an obstacle: fault-tolerant quantum computers remain difficult to scale, forcing researchers to often rely on classical simulation to develop and test their algorithms. Unfortunately, these quantum analogues quickly become too complex to simulate classically, as the computational resources required grow exponentially with the size of the system. [Fey82]

Typically, simulators such as **PennyLane** [BIS+18] or **Qiskit** [JATK+24] store a complete quantum state. This is an accurate model; however, the memory consumption is doubled every time another qubit is added, limiting simulation to approximately 24 qubits on commodity hardware. Mei et al. proposed a new simulator, **Quokka#** (pronounced "Quokka Sharp"). It uses **Weighted Model Counting** (WMC) to encode the quantum state as a logical formula, whose measurement probabilities are then computed using exact model counters [MBL24]. For specific, structured quantum circuits, Quokka# can match or outperform simulators such as PennyLane.

In this thesis, we examine Quantum Convolutional Neural Networks (QCNNs), a quantum analogue of CNNs. QCNNs have been previously explored, but their scalability has been a barrier, due to simulation and hardware limitations. Additionally, thanks to their highly structured circuit architecture, they are a prime candidate for simulation using Quokka#. Despite this, no implementation of QCNN circuits has previously existed for Quokka#, and no quantitative comparison between WMC-based and state-vector simulation has been performed for hierarchical quantum neural network architectures. This thesis addresses that gap directly, by implementing QCNN circuits ranging from 4 to 128 qubits and benchmarking their runtime performance against an equivalent PennyLane implementation.

1.1 Research Question

This work examines how Weighted Model Counting (WMC)-based simulation (Quokka#) can improve the analysis of Quantum Convolutional Neural Networks (QCNNs) compared to traditional state-vector simulators such as PennyLane [BIS+18]. In addition to runtime and scalability, the study also incorporated a correctness check. THis ensures that the Quokka# pipeline produces the same outputs as the PennyLane simulator under the same inputs. Thus, the research question is:

> *To what extent can Weighted Model Counting be employed for the scalable simulation and validation of hierarchical QCNNs, and how does its runtime and scalability compare to state-vector approaches?*

1.2 Study design

The primary contribution of this work is an implementation of Quantum Convolutional Neural Networks in the Quokka# simulator for the first time. To achieve this, QCNN circuits from 4 to 128 qubits were realised in OpenQASM format, based on the architecture presented by Hur et al. [HKP22] These circuits are computed by a modular quantum-classical training pipeline, where Quokka# and PennyLane are interchangeable simulation backends.
To evaluate this implementation, two experiments are presented. Experiment 1 measures training scalability at a fixed 8-qubit architecture, varying dataset size and epoch count. Experiment 2 measures architecture scalability by increasing the number of qubits from 4 to 128, a range inaccessible to state-vector simulators such as PennyLane. In both cases, runtime is the primary dependent variable. Additionally, a validation correctness check is run for every configuration, confirming that both backends (Quokka# and PennyLane) produce numerically equivalent outputs.

1.3 Thesis overview

This thesis firstly discusses essential concepts and definitions in Section 2, covering Machine Learning/Neural Networks, Quantum Computing and Logic. Section 3 presents a short literature review into Quantum Neural Networks and simulators. It then explains how the Weighted Model Counting simulator Quokka# processes quantum circuits and simulates them, using practical examples from logic and building up to Quantum Simulation. The main novel contributions of the thesis lie in Section 4, where the experimental setup, the experiments and their outcomes are described. Finally, section 5 discusses the main findings and potential future work and how the experiments could be improved. Appendices A and B, contain the results of the two experiments. All circuit implementations, training pipelines, comparison scripts, and the results derived from them are publicly available at `https://github.com/mateicanavea/QCNN-Quokka`.

2 Preliminaries

This section introduces the core concepts required for this thesis. The definitions are grouped into classical machine learning, quantum computing, and logic-based simulation. For a more comprehensive and accessible treatment of quantum computing and its formulation using Weighted Model Counting, see Quist et al. [QMCL25]. A similarly accessible introduction to quantum computing in the context of Quantum AI is presented by Klusch et al. [KLM+24].

2.1 Machine Learning

Convolutional Neural Network (CNN). An architecture of a feed-forward neural network, aimed at processing structured and grid-like inputs [LBBH98]. A CNN contains convolutional and pooling layers, alternately applying learned filters to input data in order to compute feature maps and their downsampling respectively. By layering these operations, the CNN learns hierarchical representations, ending up with a classifier of the problem at hand.
Binary Classification. A supervised learning problem, where each input gets labelled as one of two categories, labelled as $y \in \{-1, +1\}$ [Bis06].

Mean Squared Error (MSE). A cost function computing the average squared error between predicted labels $\hat{y}_i$ and true labels y_i, averaged over batch B [HTF09]:

$$\mathcal{L}(\theta) = \frac{1}{|B|} \sum_{i \in B} (\hat{y}_i - y_i)^2. \tag{1}$$

Principal Component Analysis (PCA). A technique for projecting high-dimensional data into a low-dimensional space by finding directions along which there is maximal variance [Jol02]. Applied in the current work to reduce MNIST images from 784 dimensions to 4 principal components.
Gradient. For a function $\mathcal{L}(\theta)$ with multiple parameters, the gradient generalises the single-variable derivative to vector-valued inputs: it is the vector of all partial derivatives of $\mathcal{L}$ with respect to each component of θ, denoted $\nabla_\theta \mathcal{L}(\theta)$ [GBC16].
Gradient Descent. An iterative optimisation technique that decreases $\mathcal{L}(\theta)$ by repeatedly moving θ a small step in the direction opposite its gradient, since this direction is guaranteed to reduce $\mathcal{L}$ for a sufficiently small step size [GBC16].
Epoch. One full pass of a training algorithm over the training dataset; training progress is commonly measured by the number of such passes [GBC16].
Adam Optimiser. An optimiser based on gradient descent, which uses adaptive learning rates for each parameter, computing their first and second moments [KB17]. It is used in training both classical and quantum-classical hybrid machine learning models.

2.2 Quantum Computing

Qubit. The basic element of quantum information, similar to the classic bit, first defined within the scope of quantum computing in the landmark book "Quantum Computation and Quantum Information" by Nielsen and Chuang [NC00] - see Fig.1 for a visual representation. While a bit has a definite state in the set $\{0, 1\}$, a qubit can exist in a *superposition* of its basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1. \tag{2}$$

Here, the numbers α and β are called *probability amplitudes*, and on the detection of a qubit's state, it will collapse either into $|0\rangle$ with probability $|\alpha|^2$ or into $|1\rangle$ with probability $|\beta|^2$.
Quantum State. The n-qubit quantum state is a vector of a 2^n-dimensional complex Hilbert space $\mathcal{H}$ [NC00]. The *computational basis* is formed by the 2^n tensor products of qubits' basis states $|x\rangle = |x_0\rangle \otimes \cdots \otimes |x_{n-1}\rangle$ with each x_i being a member of the set $\{0, 1\}$. In general, an arbitrary pure state can be represented as a superposition over this basis:

$$|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle, \quad \sum_x |\alpha_x|^2 = 1. \tag{3}$$

An alternative representation is the *density matrix* $\rho = |\psi\rangle\langle\psi|$, which is useful when the state must be decomposed into a sum of operators, as is done in the Quokka# encoding [MBL24].
Measurement. An operation that collapses a quantum state to a classical outcome [NC00]. In this thesis, all measurements are *computational basis measurements*: qubit k is measured and returns 0 with probability $p_0 = \langle\psi|P_{k,0}|\psi\rangle$, where $P_{k,0}$ is the projector onto $|0\rangle$ on qubit k.

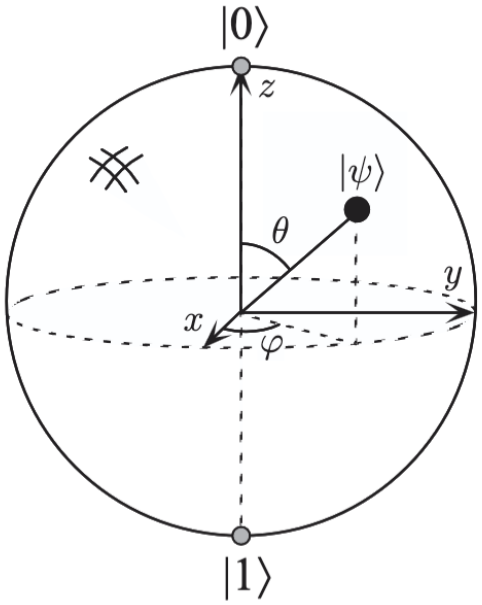

Figure 1: Bloch sphere representation of a qubit, as presented by Nielsen and Chuang

Pauli Operators. The four 2×2 matrices forming a basis for the space of single-qubit Hermitian operators [NC00]:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{4}$$

A **Pauli string** is a tensor product of Pauli operators, e.g. $X \otimes Z \otimes I$. Any density matrix can be written as a real linear combination of Pauli strings [Gay11], which is the foundation of the Quokka# encoding.

Quantum Gate. A reversible linear operation on one or more qubits, represented by a unitary matrix U satisfying $UU^\dagger = I$ [NC00]. A subcategory of quantum gates is the **Clifford group**: the set of quantum gates that transform Pauli operators (I, X, Y, Z) into other Pauli operators. Single-qubit gates of relevance to this thesis include:

- **Hadamard (H):** Creates an equal superposition, $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ [NC00].
- **Pauli rotations (R_X, R_Y, R_Z):** Parameterised rotations about the X, Y, or Z axis of the Bloch sphere by angle θ, used as trainable gates in the QCNN [NC00].
- **Pauli-X (X):** A bit-flip gate that inverts the state ($|0\rangle \leftrightarrow |1\rangle$); equivalent, up to a global phase, to a π rotation about the X-axis of the Bloch sphere [NC00].
- **T gate:** A $\pi/8$ phase rotation; a non-Clifford gate that, together with the Clifford group, forms a universal gate set for quantum computation [Got97].

Two-qubit gates used in this thesis include:

- **Controlled-Z (CZ):** two-qubit gate in quantum computing that applies a phase flip to the target qubit only when the control qubit is in the state $|1\rangle$. If the control qubit is in the state $|0\rangle$, the CZ gate does not affect the target qubit. [NC00].

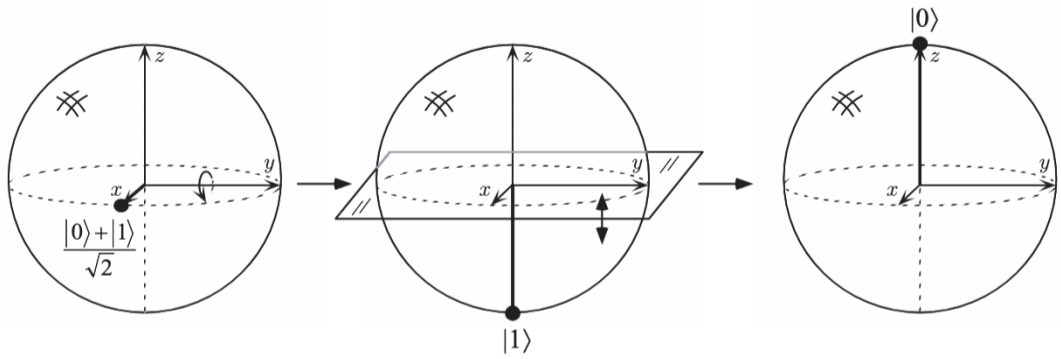

Figure 2: Visualisation of the Hadamard gate on the Bloch sphere

- **Controlled-R_Z (CRZ) and Controlled-R_X (CRX):** Parameterised two-qubit rotations used in the convolutional and pooling layers of the QCNN [HKP22].

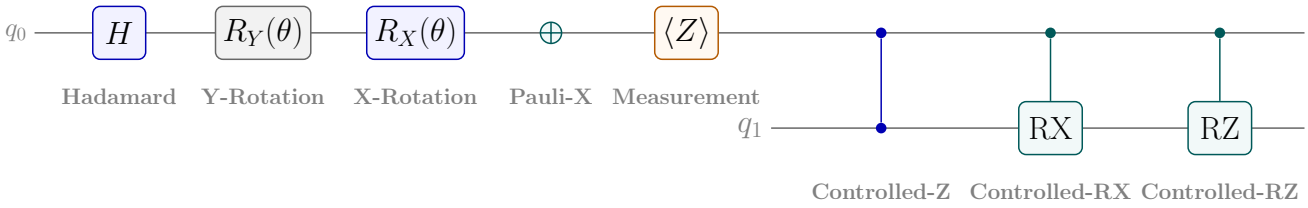

Figure 3: Single-qubit gates used in the experiments are on the left, two-qubit gates on the right.

Quantum Circuit. A sequence of quantum gates $C = G_0 G_1 \cdots G_{m-1}$ applied to an initial state. The circuit defines a unitary transformation $U = U_{m-1} \cdots U_0$ on the state space.
Parameterised Quantum Circuit (PQC). A quantum circuit in which some gate angles $\theta = (\theta_1, \ldots, \theta_k)$ are free parameters, optimised classically to minimise a loss function [FN18]. PQCs form the basis of variational quantum algorithms and are the quantum analogue of a neural network's weight vector.
Stabilizer. A quantum state $|\psi\rangle$ is a *stabilizer state* if it is the unique $+1$ eigenstate of a group of Pauli operators, called its *stabilizer group* [Got97]. Clifford circuits preserve this structure and can therefore be simulated classically in polynomial time. Quokka# extends this formalism to non-Clifford circuits via generalized stabilizer states [MBL24].
Quokka#. An open-source quantum circuit simulator based on Weighted Model Counting, developed at Leiden University [MBL24]. It encodes a quantum circuit given in OpenQASM format [CJAA+22] as a weighted CNF formula of size $\mathcal{O}(n+m)$, and delegates the model counting computation to a solver. In our experiments, the GPMC solver will be used [Has20].

2.3 Logic and SAT solvers

Boolean Satisfiability (SAT). Given a propositional formula F over Boolean variables $x_1, \ldots, x_n$, the SAT problem asks whether there exists an assignment $\alpha \in \{0,1\}^n$ such that $F(\alpha) = 1$ [Coo71]. Formulas are typically expressed in *Conjunctive Normal Form* (CNF), a conjunction of clauses where each clause is a disjunction of literals. SAT was the first problem proven to be NP-complete [Coo71].

#SAT. The counting extension of SAT: given a CNF formula F, compute the number of satisfying assignments $|\mathrm{SAT}(F)|$ [Val79]. The problem is #P-complete, meaning it is at least as hard as any problem whose solutions can be verified in polynomial time.

Weighted Model Counting (WMC). An extension of #SAT introduced in the context of probabilistic reasoning by [CD08], in which each literal x_i (or $\neg x_i$) is assigned a real-valued weight $W(x_i)$ (or $W(\neg x_i)$). The weighted model count is:

$$\mathrm{MC}_W(F) \triangleq \sum_{\alpha \in \mathrm{SAT}(F)} \prod_i W(\alpha(x_i)). \tag{5}$$

When weights are chosen to represent probabilities or quantum amplitudes, WMC computes expectation values or measurement probabilities directly.

3 Related Work

3.1 Quantum Neural Networks and Simulators

In recent years, several papers proposed the concept of **Quantum Neural Networks**. Farhi and Neven introduced the first Quantum Neural Network framework in 2018 [FN18]. In their paper, parameterised unitary transformations act on an input quantum state. For binary classification, a single Pauli operator is measured on a readout qubit.

Cong et al. introduced a specialised hierarchical architecture inspired by CNNs: **Quantum Convolutional Neural Networks** (QCNN) [CCL19]. For the convolutional layer, local unitaries are applied. For pooling, half of the qubits are measured, and the outcomes determine the qubit rotations. Hur et al. extended this into a fully parameterised QCNN for classical data classification, proposing multiple gate choices and a classical data encoding scheme benchmarked on MNIST [HKP22].

Both of the aforementioned papers use state-vector simulators, such as **PennyLane** [BIS+18] and **Qiskit** [JATK+24], which store the complete quantum state in terms of 2^n amplitudes. It is an accurate model; however, since the memory consumption for storing all amplitudes is doubled every time another qubit is added, it is limited to approximately 24 qubits for commodity hardware. Given that QCNNs need to measure the circuits thousands of times for training, the ceiling is reached quickly, and the QCNN literature has consistently reflected this issue [CCL19, HKP22].

Weighted Model Counting (WMC) is a logic-based method that offers a different approach. Rather than storing the full quantum state, Mei et al. showed that **Quokka#** encodes a quantum circuit as a weighted logical formula and delegates the probability computation to an exact model counter [MBL24]. For circuits with many rotation gates (precisely the architecture profile of QCNNs), Quokka# matches or outperforms traditional simulators, and does so without approximation error.

3.2 Quokka# and Weighted Model Counting

Dealing with uncertain physical systems, such as quantum phenomena, requires probabilistic reasoning. For decades, SAT solvers have represented the go-to framework for solving such complex

tasks. A SAT solver aims to find a solution for the **Boolean Satisfiability (SAT) problem**. The satisfiability problem asks whether there exist assignments for each variable x_i such that the formula can be true (is satisfiable). To illustrate the problem, let us use an example. We will explore this logic formula in Conjunctive Normal Form (CNF):

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \tag{6}$$

SAT poses the question: *Given a CNF formula, does there exist an assignment that satisfies it?* In this example, one valid solution is setting $x_1, x_2, x_3 = 1$ (True), which satisfies the formula.
Once the **Decision Problem** of whether a formula is SAT is resolved, we can move on to the **Counting Problem**, which asks: *How many assignments satisfy the formula?* This is the primary concern of #**SAT** solvers. In the example above, $x_1, x_2, x_3 = 1$ is not the only solution; for instance, $x_1 = 1, x_3 = 1$, and $x_2 = 0$ also works. Running a #SAT solver would show that there are indeed four satisfying assignments for our formula:

$$\{(0, 1, 0), (0, 1, 1), (1, 0, 1), (1, 1, 1)\}$$

What makes SAT solvers work well in practice is that real-world problems have an exploitable structure. The performance of SAT and SAT# solvers has significantly improved over the past two decades due to **intelligent heuristics** [Ten23]:

- **Clause learning** [MSS96], where contradictions during search are recorded so that new clauses do not encounter similar mistakes in other branches of a search tree;
- **Component caching** [BP00], where disjoint components of a formula are cached so that they only have to be solved once. When combined with clause learning [SBB+04], this allows the solver to both prune the search space and avoid redundant computation across independent sub-problems;
- **Tree decompositions** [SS10], which exploit how "tree-like" a formula's structure is. Problems that are hard in general become tractable when the formula's variable interactions resemble a tree rather than a dense graph, a property measured by *treewidth*.

Because the hierarchical gate structure of QCNNs keeps treewidth low, a solver such as GPMC scales efficiently with increased qubit sizes. In the quantum context, the #SAT problem is not much different: variables x_i represent the basis states of the qubits, and a satisfying assignment is a computational basis state consistent with the circuit's gate constraints. However, unlike classical #SAT where every satisfying assignment contributes equally, quantum mechanics assigns each path a different probability amplitude, which calls for **Weighted Model Counting (WMC)**.
Here, each literal in a Boolean expression has a corresponding weight. If the literal in question is x_i, the corresponding weight is given by $W(x_i)$ if x_i is positive, otherwise $W(\neg x_i)$. Thus, the weight of an assignment is the product of the weights of its literals:

$$\mathrm{MC}_W(F) \triangleq \sum_{\alpha \in \mathrm{SAT}(F)} \prod_i W(\alpha(x_i)) \tag{7}$$

Coming back to the previous example, let us say that we have $W(x_1) = \frac{1}{2}$, $W(\neg x_1) = \frac{1}{2}$, and the remaining variables x_2, x_3 remain unbiased (weight 1 for both). What this shows is that WMC

can be used to define a probability distribution over the variable assignments. This is exactly why WMC is the ideal technique for simulating quantum circuits. When simulating a quantum circuit, we are interested not in the number of basis states satisfying the circuit, but *in the probability* of a certain measurement result.

The probability of a measurement outcome in a quantum circuit is computed by summing over all computational paths weighted by their amplitude, known as the *path sum*, which is precisely the form of a weighted model count [MML24]. The Quokka# framework therefore computes this exactly, rather than approximating it. It incorporates encoding proposed by Mei et al. [MBL24], which encodes a quantum circuit into a weighted CNF formula whose weighted model count is equal to the target measurement probability. This encoding can be broken down into quantum state encoding, gate operation encoding, and measurement encoding.

The idea behind this approach is that every quantum state $|\psi\rangle$ can be expressed as a linear combination of weighted Pauli strings using the density matrix of the state:

$$|\psi\rangle\langle\psi| = \sum_i \pm \left(\frac{1}{\sqrt{2}}\right)^{k_i} P_i, \qquad P_i \in \text{PAULI}_n \tag{8}$$

Here, each Pauli string $P_i = \sigma[x_0, z_0] \otimes \cdots \otimes \sigma[x_{n-1}, z_{n-1}]$ can be represented using $2n$ Boolean variables, where each qubit uses two Boolean variables (x_i, z_i), and an additional sign bit r, with $W(r) = -1$ and $W(\bar{r}) = 1$ to represent the $\pm$ factor without resorting to complex numbers, and weight $W(u) = \frac{1}{\sqrt{2}}$.

Each gate in the circuit now represents a Boolean function, representing how that gate acts on Pauli operators through conjugation. The Clifford gates H and CNOT result in small, deterministic constraints, where each solution corresponds to a unique solution after the gate. On the other hand, the T gates lead to branching, as the conjugation rule states that $TXT^\dagger = \frac{1}{\sqrt{2}}(X + Y)$, meaning that a single Pauli string before the gate results in two Pauli strings after the gate. This branching can be represented using an auxiliary variable u, which increases the number of solutions but halves their weight using the weight function $W(u) = \frac{1}{\sqrt{2}}$.

More importantly, the size of the formula stays **linear** in the number of qubits n and gates m, that is $\mathcal{O}(n + m)$, as each gate creates just a constant number of new Boolean variables. The measurement is represented by one more constraint, which keeps only the Pauli strings associated with the required outcome; for instance, to measure the state $|0\rangle$ of qubit k, we only consider assignments for which $z_k = 1$ and $x_j = 0$ for all j.

The weighted model counting of the formula multiplied by $\frac{1}{2^q}$ for q measured qubits gives the exact probability of the measurement. The Quokka# framework takes the input quantum circuit in the QASM format, generates the weighted CNF according to the previous description, and sends it to the chosen model counter GPMC [Has20].

4 Experiments

The experimental analysis consisted of two separate experiments, each focusing on one aspect of scalability in the hybrid quantum-classical pipeline. In both cases, the examined problems were the same: classification of a simplified MNIST set with digits 0 and 1, or synthetic data on the parity classification problem. Both experiments used the same hybrid Quantum-Classical pipeline for simulation and training. The main dependent variable in the experiments was **runtime**. The total elapsed time (including circuit construction, simulation, and gradient computation) was recorded via `time.perf_counter()`.

Experiment 01 (Training Scalability) An initial analysis of the Quokka# simulator with running a full training run. With a **fixed** network architecture, increases in runtime were analysed through numerous epochs and different training dataset sizes.

Experiment 02 (Architecture Scalability) In this experiment, architectural scalability was evaluated by limiting training to a single epoch. The key objective was to analyse the increase in computational overhead with respect to changes in architecture size by increasing the number of qubits. Thus, a quantitative comparison was made between Weighted Model Counting (WMC) and PennyLane as the Hilbert space increases exponentially.

4.1 Quantum-Classical pipeline for simulation and training

In this section, the architecture of a QCNN as well as its training method are discussed, using a 4-qubit QCNN as an example. This hybrid setup, illustrated in fig. 4, will be used initailly for 4.2. For the sweep experiment in 4.2 and for the entirety of 4.3, the 4-qubit QCNN needs to be scaled up to allow for an increasing number of qubits, but the architecture principles remain the same.
Firstly, there is a *classical* data loading and preprocessing technique, using either the MNIST dataset or synthetic data. Once the data is chosen, a randomly-parameterised QCNN is initiated. The typical CNN convolutional and pooling operations occur inside the *quantum* simulation, where the training data is processed by the parameterised QCNN, and thereafter a measurement evaluation occurs. Once the measurement is done, an expected value $\langle Z_{q_n-1} \rangle = 2p_0 - 1$ is computed classically, and the MSE values determines a numerical gradient. The optimiser update is then computed conventionally, outside the quantum simulator, and the loop calls upon the Quokka# QCNN for each new measurement once a new step in the gradient descent is taken.

4.1.1 Data pre-processing

MNIST For this experiment setup, there are two types of dataset that can be tested: the MNIST dataset, and the synthetic dataset on the parity classification problem.
As images have a much larger amount of features (or pixels) than the 4-qubit setup allows, the dimensionality of these images was reduced using Principal Component Analysis (PCA) [Jol02] to the required number of qubits. For quantum compatibility, a scaling of these features to the range $[0,\pi]$ is required, so that the data can be used as rotation angles for quantum gates. Finally, labels are converted to a ± 1 format and the dataset shuffled to ensure the model learns a balanced variety of digits during training.

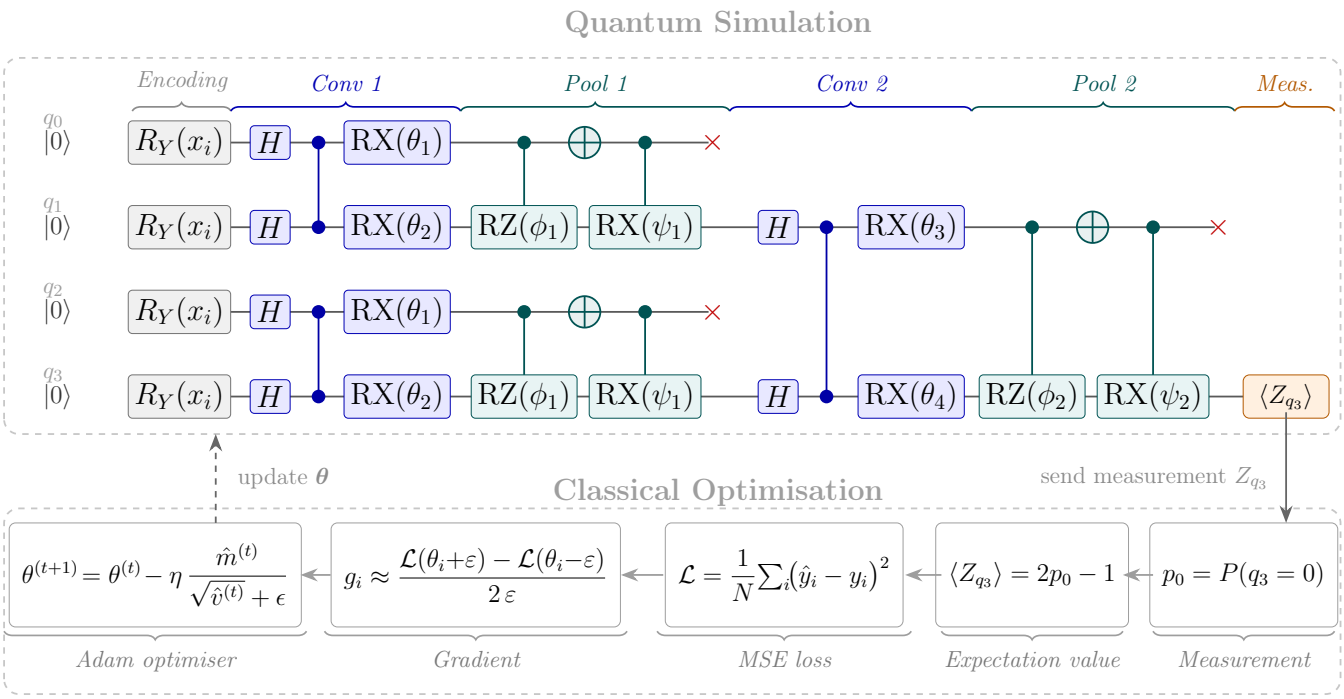

Figure 4: The QCNN architecture and training pipeline. The quantum simulator (top) processes four features from pre-processed data as qubits across a parameterised QCNN, producing a measurement. In the classical loop, the training occurs: an expectation value is computed, and then the MSE gradient is used for gradient descent via Adam optimiser to train the next QCNN parameters.

Parity Classification Problem While the PCA-compressed MNIST focuses on classifying real-world data that has continuous features which are locally correlated, the parity benchmark involves a dataset created to test the pipeline using a task that is completely non-local in nature, like those used for quantum kernel benchmarks in structural classification tasks [HBM+21].
For an n-qubit system, each sample is a binary number $x \in \{0,1\}^n$, whose angles are given by $\theta_i = \pi x_i$, while its label is $y = (-1)^{\sum_i x_i}$. The QCNN needs to reduce the global parity to a single readout qubit.

4.1.2 QCNN architecture

The architecture of the QCNN used in the experiments is a parameterised quantum circuit (PQC), inspired by classical CNN features. Compared to a reconstruction that exactly reproduces the arithmetic operations of CNNs for quantum computers, the PQC implementation is shallower in circuit depth, and more suitable for current Noisy Intermediate-Scale Quantum (NISQ) devices.
Compatibility with the Quokka # simulator is required for this experiment. Thus, the PQC was translated in the required OpenQASM format. The implementation in Fig. 5 consists of a 4-qubit QCNN, with 2 convolutional and 2 pooling layers, which will be used for the first experiment. However, larger constructions with more qubits do not differ from this example.
In Fig. 5, the Convolutional Neural Network consists of 8 parameters, presented in table 1, with 2 parameters per each layer. Reusing the same parameters helps reduce the execution time of the model counting simulator. This architecture is one of many that can be implemented.

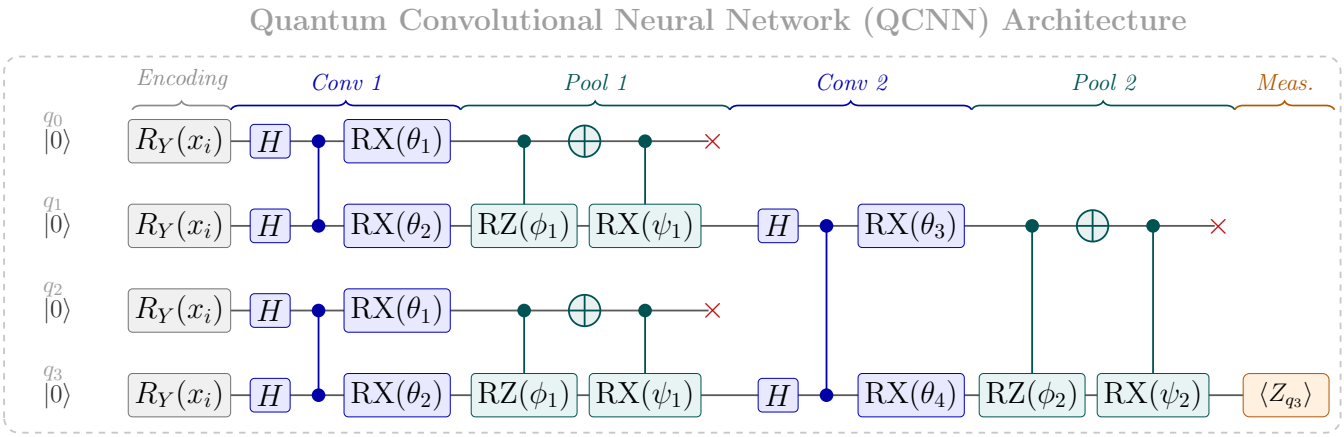

Figure 5: Quantum Convolutional Neural Network (QCNN) circuit diagram. The encoding block performs parallel RY(x_i) rotations. Each convolutional filter (the U_9 block) applies Hadamard gates to both qubits, an entangling Controlled-Z (CZ) gate, and parameterised RX rotations. Pooling layers apply Controlled-RZ (CRZ) and Controlled-RX (CRX) rotations to condense two qubits into one, reducing the circuit to a single measured qubit on wire 3. The output is determined by a single-qubit measurement at the end of the circuit.

Parameter	**Used in**	**Qubits**	**Gate**
c1_rx0	conv1	q[0], q[2]	RX
c1_rx1	conv1	q[1], q[3]	RX
p1_crz_angle	pool1	(0,1), (2,3)	CRZ
p1_crx_angle	pool1	(0,1), (2,3)	CRX
c2_rx0	conv2	q[1]	RX
c2_rx1	conv2	q[3]	RX
p2_crz_angle	pool2	(1,3)	CRZ
p2_crx_angle	pool2	(1,3)	CRX

Table 1: QCNN Parameter Mapping and Gate Operations

The convolutional block is a "U_9" block, consisting of two Hadamard gates and a Controlled-Z (CZ) gate, which create an entangled state between the two qubits, thereby allowing correlations between neighbours. Each qubit is then acted on by a parameterised R_X rotation. These are the adjustable parameters, which after training are tuned so that the states rotate towards the "True" position. The pooling block condenses the information down to one qubit instead of two, shifting the information from the control qubit onto the target qubit. In the end, there will be only one qubit, in the case of binary classification. This QCNN architecture and its convolutional and pooling ansatzes (i.e. parameterized quantum circuit templates) were compiled in a more comprehensive set by Hur et al. [HKP22], and the specific "U_9" block originates from Sim et. al [SJAG19].

4.1.3 Training Optimisation

The optimisation of the QCNN is achieved by randomly initializing the parameters θ_i, which are updated during training. The parameters are drawn from a uniform distribution $\mathcal{U}(-\pi/2, \pi/2)$. To update the parameters, the Adam Optimisation algorithm was used. Adam is a variant of stochastic gradient descent that adapts the parameters by taking into account past steps. In standard gradient descent, parameters are updated according to the current gradient. Adam updates the parameters taking into account past gradients and their changes.
The gradient is computed at each time step t as

$$g^{(t)} = \nabla_\theta \mathcal{L}(\theta^{(t)}). \tag{9}$$

The gradient is approximated using a finite difference method. The loss is computed at $\theta_i + \varepsilon$ and $\theta_i - \varepsilon$, where $\varepsilon = 0.01$, as

$$g_i^{(t)} \approx \frac{\mathcal{L}(\theta_i + \varepsilon) - \mathcal{L}(\theta_i - \varepsilon)}{2\varepsilon} \tag{10}$$

The loss is defined as the Mean Squared Error (MSE) where y_i is the label, and $\hat{y}_i$ is the predicted value, while performance is also monitored via accuracy:

$$\mathcal{L}(\theta) = \frac{1}{|B|} \sum_{i \in B} (\hat{y}_i - y_i)^2 \qquad \text{Accuracy} = \frac{1}{|B|} \sum_{i \in B} \mathbb{I}(\text{sign}(\hat{y}_i) = y_i) \tag{11}$$

Adam processes these gradients using two "moments": the first moment $m^{(t)}$ (the average direction/momentum) and the second moment $v^{(t)}$ (the squared gradient). These moments are computed as follows:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1 - \beta_1) g^{(t)} \qquad v^{(t)} = \beta_2 v^{(t-1)} + (1 - \beta_2)(g^{(t)})^2 \tag{12}$$

However, Adam avoids bias in the optimisation process using the following bias correction steps:

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1 - \beta_1^t} \qquad \hat{v}^{(t)} = \frac{v^{(t)}}{1 - \beta_2^t} \tag{13}$$

The update of the parameters $\theta^{(t)}$ at step $t + 1$ is computed using the following equation:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} \tag{14}$$

In our experiments, the values of $\eta = 0.05$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ are used. These values give a good balance between optimisation speed and stability.

4.2 Experiment 01 - Training Scalability

Focused on analysing the efficiency of the Quokka# simulator during a full training run, this experiment uses a fixed qubit architecture. An initial, single illustrative run examines Quokka# performance on a small-scale 4-qubit example: 30 epochs over 100 samples. A second, more structured experiment then fixes an 8-qubit architecture and varies the number of epochs and the training-set size, recording runtime and classification accuracy and comparing them against the PennyLane backend `lightning.qubit`.

4.2.1 Single Experiment

For an initial, one-off experiment, we use the exact setup explained in the previous section - a 4-qubit QCNN, and run complete training runs. In table 2 for example, a 30-epoch training with 100 training samples and 20 test samples was computed on a 16-core, GitHub Codespaces virtual machine with 64GBs of available RAM.

Label	Calls	Total (s)	Avg (s)	Min (s)	Max (s)
predict	48420	3667.42	0.076	0.029	0.204
run_quokka [total]	48420	3644.35	0.075	0.028	0.204
↘ run_quokka [wmc_p0]	48420	**2264.79**	0.047	0.023	0.183
↘ run_quokka [qasm_parse]	48420	297.78	0.006	0.000	0.082
epoch [wall]	30	315.80	10.527	10.031	11.134
numerical_gradient [epoch]	30	315.80	10.527	10.031	11.134
numerical_gradient	30	315.80	10.527	10.031	11.134
mse	480	315.79	0.658	0.572	0.818
build_circuit	48420	20.82	0.000	0.000	0.061
_render_qasm	48420	17.69	0.000	0.000	0.061
mse_and_accuracy	5	2.88	0.576	0.152	0.745

Table 2: Time cost analysis for QCNN training and inference (MNIST, n_train=100 n_test=20). Results represent a 30-epoch run with a final classification accuracy of 60.0%.

It can be observed that the main bottleneck occurs when the Quokka simulation is run: the 'run_quokka' function, more specifically in the 'wmc_p0' process, looks as follows:

```
"run_quokka [wmc_p0]":
    cnf0 = qk.encoding.QASM2CNF(circuit, computational_basis=True,weighted=True)
    readout = n_qubits - 1
    cnf0.add_measurement({readout: 0})
    p0 = qk.Simulate(cnf0, cnf_file_root=td)
```

Therefore, the long computation can be attributed to the encoding and GPMC model counting subprocesses that start with the simulation when measuring $p(q_{n-1} = 0)$. In this process alone, one call averages 0.047 seconds. And for a 30 epoch run such as the one presented above, it means that there will be tens of thousands of simulations: each QCNN is trained on 100 samples, with an 8 parameter circuit that has to run 2 times, for 30 epochs $\Rightarrow 100 * 8 * 2 * 30 = 48000$, as well as testing during epoch prints $(4 * 100 = 400)$, and during testing (20 runs), resulting in the 48420 calls of run_quokka.

4.2.2 Sweep Experiment

In order to study the effects of this computational bottleneck in depth, a structured, scaled up experiment was run: an 8-qubit QCNN architecture with varying training data set sizes $(n \in \{10, 30, 50, 70, 90\})$ and number of epochs (epochs $\in \{10, 20, 30, 40, 50\}$). We compare the exact model counting runtime of Quokka versus the traditional simulation of PennyLane. The

results are plotted below in Figure 6, and available in full in appendix A. This experiment was run locally, on an AMD Ryzen 5 3600X CPU (6 cores), 48 GB of RAM, an NVIDIA GeForce RTX 2070 GPU, and a 500 GB SSD configuration.

The left figure in Figure 6 illustrates the time-scaling trends with respect to the number of training samples (on a log-log scale) while the right plot demonstrates the scaling behaviour with respect to the number of epochs (also on a log-log scale). The trends observed are linear (seen as two parallel lines), meaning that:

$$\text{Workload} \propto n_{\text{train}} \times n_{\text{parameters}} \times \text{epochs} \tag{15}$$

The performance disparity is observed in all considered configurations. The Quokka simulator (solid lines) needs approximately an order of magnitude more time than the PennyLane backend `lightning.qubit` (dashed lines) for each configuration. For example, the workload ($n = 90$, 50 epochs), takes around 200 seconds to train with PennyLane, while Quokka needs more than 1,000 seconds. Although both simulations show a relatively linear scaling with respect to the absolute theoretical workload growth, the vertical separation between the solid and dashed lines is clearly visible. This implies that the fundamental bottleneck found in Table 2 i.e., the repetitive overhead involved in translating QASM into CNF format (`qasm_parse`) and running the model counting processes (`wmc_p0`), results in a constant cost that increases with the data size and training epochs.

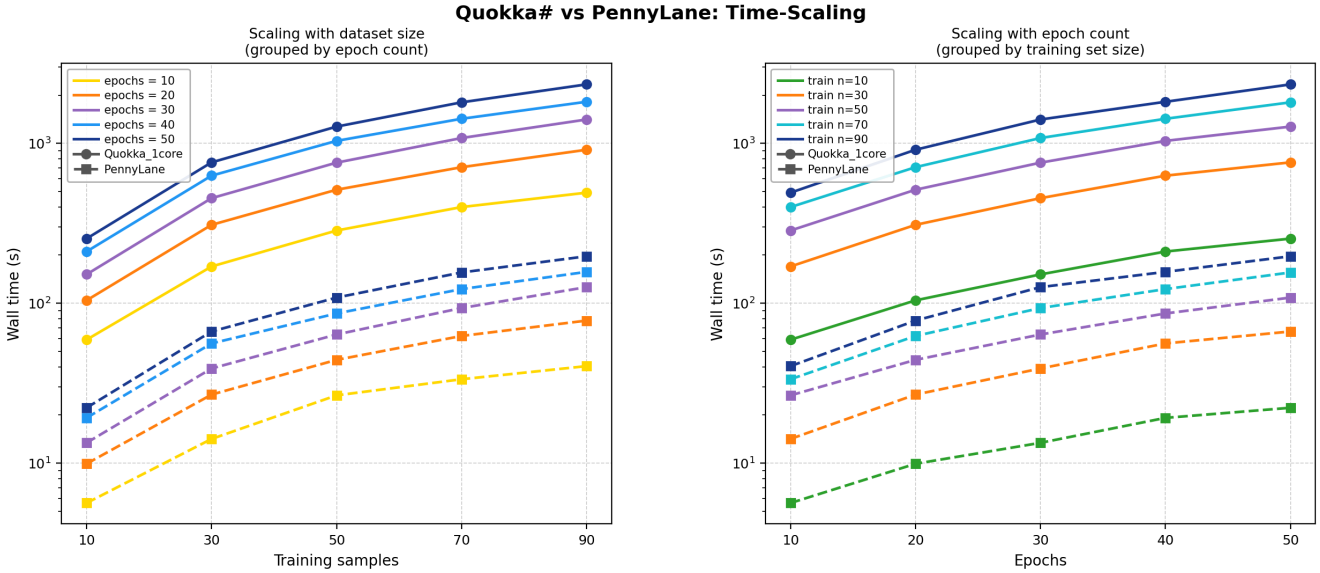

Figure 6: Time scaling comparison between the Quokka and PennyLane backends across varying training set sizes (10, 30, 50, 70, 90 samples) and epoch counts (10, 20, 30, 40, 50). Both axes use a log scale. *Left:* runtime as a function of training set size, grouped by epoch count. *Right:* runtime as a function of epoch count, grouped by training set size. Solid lines denote Quokka; dashed lines denote PennyLane. Quokka's exact numerical computations result in higher execution times across all configurations.

4.2.3 Parallelisation Improvement

For each epoch of training, the gradient of the loss function needs to be computed for all eight parameters in the four-qubit QCNN model (refer to Table 1 and Figure 5). This requires two circuit simulations using the central finite difference scheme, where for parameter Θ_j we need to evaluate the circuits at $\mathbf{\Theta} + \varepsilon \mathbf{e}_j$ and $\mathbf{\Theta} - \varepsilon \mathbf{e}_j$. In general, there are $2d = 16$ simulations for the Quokka# circuits per epoch using N training samples since $d = 8$. For example, in the four qubit run from Table 2 with $N = 100$ samples over 30 epochs, this amounts to 48,000 evaluations in total.
The feature exploited by parallelisation is the fact that no evaluation depends on any other: each of these is an independent run of the QCNN circuit using the same parameter vector but a different input sample. The calculation of the gradient can only proceed after all the $2dN$ values have been obtained, and hence the entire process per epoch reduces to a set of independent tasks. This scenario is an *embarrassingly parallel* problem, a term initially coined in a 1986 book on multiprocessors, used here to refer to parallelisation problems which are "embarrassingly easy" [Mol86].
Thus, all of the $16N$ computations were grouped into a list at the beginning of each epoch, which is executed by passing it to `multiprocessing.Pool.map` in one call. Each process works individually on creating the string representation of the parameterised QASM circuit for its task (parameter vector and input sample), executing the GPMC algorithm, and returning the expected value. After all processes complete their work, the parent process calculates the gradient and performs an Adam step. Figure 7 shows the workflow. When running a new experiment on a parallelised 8-qubit circuit, the results make Quokka# more competitive with PennyLane, if slower still, as reflected in fig. 8.

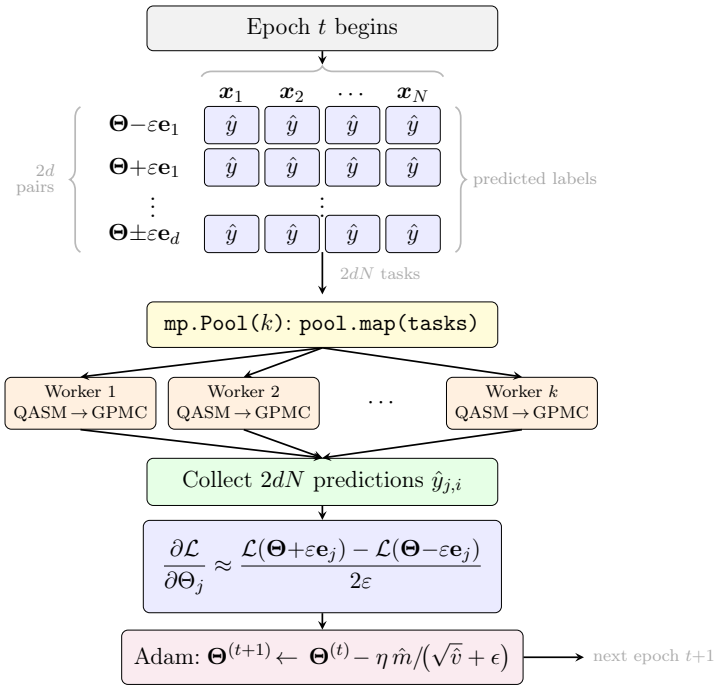

Figure 7: Data flow for one training epoch under Quokka's parallel gradient computation. At the start of each epoch, all circuit evaluations are assembled into a task list and sent via `multiprocessing.Pool.map`. Each worker independently builds the circuit, calls the solver, and returns the expectation value as a scalar. After all workers report their output, the gradient is assembled via the forward finite-difference formula and the parameters are updated with Adam.

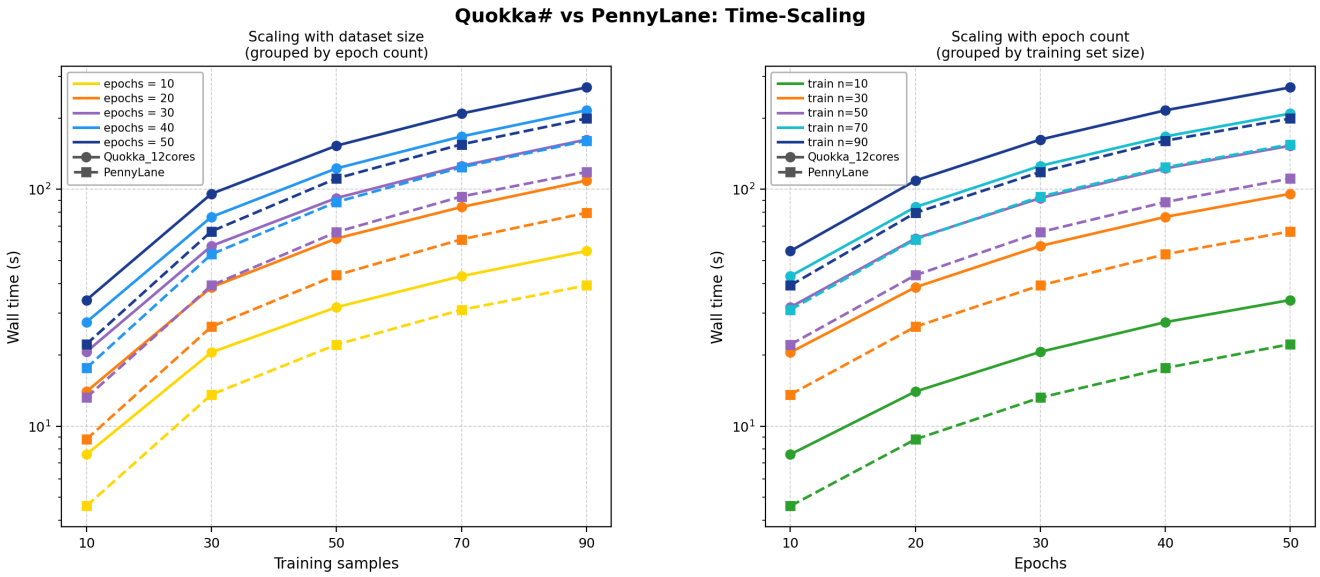

Figure 8: Time scaling comparison between the Quokka and PennyLane backends across varying training set sizes (10, 30, 50, 70, 90 samples) and epoch counts (10, 20, 30, 40, 50) on a 8-qubit circuit. Both axes use a log scale. *Left:* runtime as a function of training set size, grouped by epoch count. *Right:* runtime as a function of epoch count, grouped by training set size. Solid lines denote Quokka; dashed lines denote PennyLane. Quokka's newly parallelised execution tasks make it competitive and comparable to PennyLane.

The use of parallelised gradient calculation by means of `multiprocessing.Pool` results in a considerable decrease in wall-clock time across all configurations. In the single-core regime, Quokka# requires between 10× and 12× more time than PennyLane `lightning.qubit`; for instance, at $n_{\text{train}} = 90$ and 50 epochs, the single-core execution takes 2,337 seconds compared to PennyLane's 197 seconds. With 12 workers, the same configuration completes in 270 seconds, reducing the overhead ratio from 11.9× to 1.37×, a speed-up of approximately 9 times attributable solely to parallelisation. This brings Quokka# to a consistent overhead of roughly 1.4× over PennyLane across the sweep: parallelisation makes the two backends comparable but does not reach parity, as the per-call CNF encoding and GPMC subprocess overhead cannot be parallelised away.

4.3 Experiment 02 – Architecture Scalability

The second experiment examines Quokka#'s main advantage: efficient quantum circuit simulation at scale, via Weighted Model Counting. Mei et al. found that model counting often outperforms state-of-the-art simulation techniques based on the ZX calculus and decision diagrams when tested against an increasing number of qubits [MBL24]. That advantage is especially clear for circuits containing a large number of rotation gates, a feature shared by the QCNN architecture used here, where both convolutional and pooling layers feature parameterised R_X, CRZ, and CRX gates throughout (see Section 4.1.2).

Instead of running a full training pipeline, this experiment fixes the training to a single epoch with one training sample and one test point. The goal is not to obtain a mature classifier but to calculate the cost of one complete gradient-descent step, with its circuit construction, forward-pass evaluations, and parameter update, as a function of qubit count n. The number of qubits is varied as $n \in \{4, 8, 12, \ldots, 128\}$, and for each value the full hybrid quantum-classical loop is executed once under each backend. The experiment was performed locally on a system equipped with an AMD Ryzen 5 3600X CPU (6 cores), 48 GB of RAM, an NVIDIA GeForce RTX 2070 GPU, and a 500 GB SSD, and the results below (available in full in appendix B) reflect those hardware constraints.

4.3.1 Experimental Pipeline

The parameterised QCNN is constructed with n qubits following the hierarchical architecture described in Section 4.1.2: convolutional and pooling layers are scaled proportionally to n, and the circuit is exported to OpenQASM format for Quokka# compatibility. For each backend (Quokka#, `default.qubit`, and `lightning.qubit`), the hybrid loop executes one epoch under the same conditions: a randomly drawn parameter vector $\theta \sim \mathcal{U}(-\pi/2, \pi/2)$, a single training sample and the finite-difference gradient with $\varepsilon = 0.01$. Wall time is recorded via `time.perf_counter()` around the full epoch, including circuit construction, all simulator calls, and the parameter update.

A notable constraint applies to the PennyLane state-vector backends. Both `default.qubit` and `lightning.qubit` represent the full quantum state as a complex vector of 2^n amplitudes, requiring $2^n \times 16$ bytes of memory. At $n = 20$ qubits this amounts to approximately 16 MB, but the gradient computation requires $2n_{\text{params}}$ such evaluations per epoch, each loading the full state-vector, and memory consumption becomes unattainable before $n = 28$ qubits on the local hardware. Thus, both backends are limited to $n = 24$ qubits in this experiment.

An alternative would be the `lightning.tensor` backend, which approximates the quantum state as a Matrix Product State with `max_bond_dim`= 64 using NVIDIA's cuQuantum SDK [ADP+24], and thus avoids this exponential memory wall and could run up to $n = 128$ qubits. Due to time constraints, this backend was not benchmarked in the current thesis.

4.3.2 Single-core Experiment Results

Figure 9 presents wall-time results for Quokka# running single-threaded ($n_{\text{workers}} = 1$) alongside PennyLane backends `default.qubit` and `lightning.qubit`. Three regimes are visible.

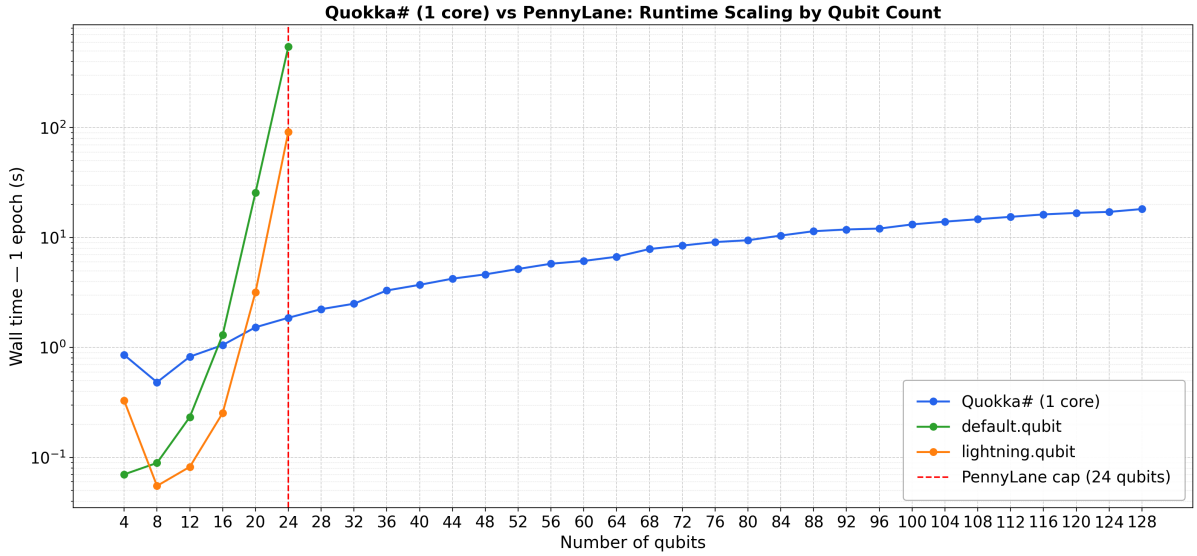

Figure 9: Timing log-scaling of Quokka# against PennyLane simulators (`default.qubit` and `lightning.qubit`) for QCNN wall time per epoch, single-threaded Quokka# ($n_{\text{workers}} = 1$). All benchmarks were performed on a system with an AMD Ryzen 5 3600X, 48 GB RAM, NVIDIA RTX 2070, and 500 GB SSD. PennyLane state-vector simulation is infeasible beyond 24 qubits due to exponential memory growth. Quokka# exhibits higher overhead at small qubit counts but crosses below PennyLane at $n \approx 16$ qubits and then scales polynomially (with an empirically estimated exponent of approximately 1.4) to 128 qubits.

Low-qubit regime ($n < 16$) For architectures with less than 16 qubits, Quokka# is slower than PennyLane's state-vector simulation by a significant margin: at $n = 4$ qubits, Quokka# takes approximately 0.85 s compared to 0.07 s for `default.qubit`, an approximate 12× overhead. This cost is not attributable to the WMC computation itself, which at 4 qubits is trivially fast, but rather to the **fixed overhead** of CNF encoding, file I/O to the GPMC subprocess, and solver initialisation, all of which are paid per simulation call regardless of circuit size.

Crossover point ($n = 16$) At $n = 16$, Quokka# and PennyLane `default.qubit` show relatively equal wall times (roughly 1.0-1.5 s). For $n > 24$, PennyLane's state-vector backends use more than the system available memory and is skipped to avoid a system crash, while Quokka# keeps running.

High-qubit regime ($n > 16$) In the high-qubit regime, Quokka# increases from roughly 2.5 s at $n = 32$ qubits up to approximately 18.2 s at $n = 128$ qubits: a 7.3× increase for a fourfold increase in the number of QCNN parameters and a huge increase in Hilbert space size (2^{96}).

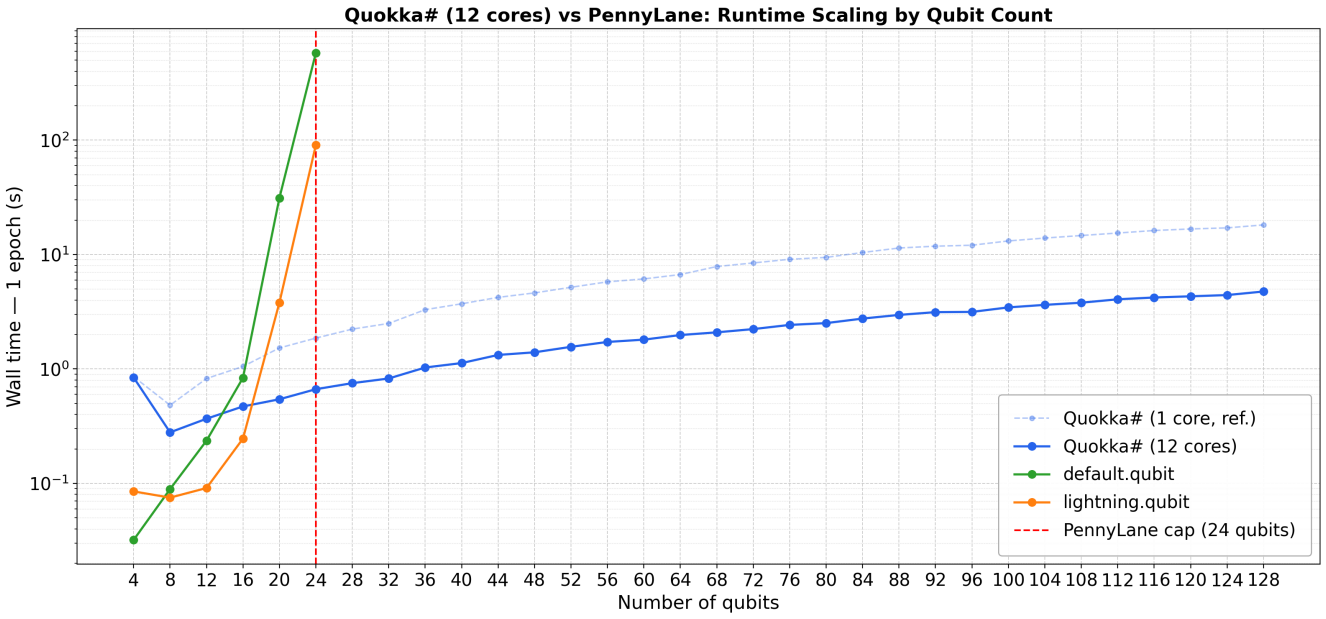

Figure 10: Timing log-scaling of Quokka# with 12-core parallelisation ($n_{\text{workers}} = 12$), same hardware as Figure 9. Parallelisation yields a $\approx 2.8\times$ speed-up for $n \leq 20$, increasing to ≈ 3.8 at $n = 128$.

4.3.3 Parallelisation Experiment Results

Figure 10 shows the time per epoch for a 12-core parallel Quokka# execution across qubit counts from $n = 4$ to $n = 128$. At small qubit counts ($n \leq 20$), the 12-core variant reduces wall time by a factor of approximately $2.8\times$; for example, at $n = 20$ the single-core execution takes around 1.5 s compared to 0.5 s with 12 workers. This speed-up is consistent with the embarrassingly parallel structure of the gradient computation, where all $2dN$ circuit evaluations are dispatched simultaneously and collected before the parameter update. As the qubit count increases, both curves exhibit sub-exponential, mildly super-linear growth and the relative speed-up increases: at $n = 128$, the single-core run takes approximately 18.2 s while the 12-core run completes in under 4.8 s, corresponding to a speed-up of roughly $3.8\times$. The convergence of the two curves at smaller n reflects the fixed cost of worker initialisation and dispatch overhead, which dominates over the comparatively cheap individual GPMC calls and reduces the marginal benefit of parallelism. As n increases, this overhead becomes negligible relative to the per-call computation, and the speed-up grows toward the 12-core ideal. Nonetheless, the 12-core configuration remains strictly faster across all tested qubit counts, confirming that parallelisation of gradient computation provides a consistent and practically significant reduction in training time for Quokka#.

Therefore, the primary result of Experiment 02 is the following: Weighted Model Counting enables practical and exact QCNN simulation at qubit counts where traditional approaches are memory-infeasible, achieving polynomial wall-time growth from $n = 32$ to $n = 128$ qubits on commodity hardware. This polynomial (exponent ≈ 1.4) behaviour arises from the GPMC solver's exploitation of the repetitive gate structure of the QCNN. It motivates WMC-based simulation as a tool for analysis tasks, such as estimating misclassification probabilities or studying circuit behaviour under parameter perturbations, which would be inaccessible to state-vector simulators at these scales.

4.4 Validation and Accuracy

Due to the different components and dependencies involved in simulating Quantum Convolutional Neural Networks with the PennyLane and Quokka# environments, validation is essential before any runtime comparison can be considered meaningful. Thus, for each new configuration in both experiments, a **correctness check** is run prior to collecting benchmark data.
This check simulates a fixed set of test cases: all-zero parameters $\theta = \mathbf{0}$, a uniform initialisation $\theta_i = \pi/4$ for all i, and five independent draws from $\mathcal{U}(-\pi/2, \pi/2)$. Each case is run through both backends on the same QCNN architecture. The check first confirms that both implementations designate the same readout qubit (e.g. `q[n-1]`) to guard against architecture-level bugs introduced when scaling the circuit. Once the readout qubit is confirmed, the expectation value $\langle Z \rangle = 2p_0 - 1$ is computed from both backends and the two values are compared with a numerical tolerance of 1×10^{-6}. A failure at any test case halts the benchmark sweep and reports the offending parameter vector, preventing contaminated runtime measurements.
We adapt the random-stimulus verification methodology of Burgholzer et al. as a validation (equivalence) check between our two backends [BKW21]. According to Burgholzer et al. even using a few inputs randomly selected will help in detecting bugs in quantum circuit implementations with high accuracy. Seven different examples have been considered: these include the zero gradient initialization (which makes use of the identity property of the gate at $\theta = 0$), the uniform example (which considers all the parameterised rotations as equally likely), and random examples that are drawn from the training initialization distribution.
For all experiments involving both Quokka # and Pennylane QCNNs conducted on both a single-core and 12-core configuration, the correctness test results showed that the deviation of the expectation values between the two was $\Delta\langle Z \rangle < 1 \times 10^{-6}$. This validates that the OpenQASM and CNF encodings using Pauli strings, and the QCNN class in Python share the same circuit semantics.

5 Conclusions and Further Research

This thesis investigated the extent to which Weighted Model Counting can improve the scalable simulation of Quantum Convolutional Neural Networks compared to state-vector approaches, and whether the two simulators produce numerically equivalent outputs. Two experiments were conducted using a hybrid quantum-classical pipeline: Experiment 01 examined training scalability at fixed 8-qubit architecture across varying epoch counts and dataset sizes, and Experiment 02 looked at architecture scalability by increasing the number of input qubits from 4 to 128 at a fixed one-epoch training budget.

5.1 Conclusions

Experiment 01 Experiment 01 demonstrates that training a QCNN via Quokka# is a computationally viable method, though slower per simulation call than PennyLane. For an illustrative, 4-qubit run, Quokka# averages 0.047 s per `wmc_p0` call, versus sub-millisecond per call for `lightning.qubit`, resulting in a $\sim 12\times$ overhead on individual invocations. However, when parallelisation was applied across the $2n_{\text{params}}$ gradient evaluations within each epoch, runtime becomes comparable for small training budgets. This overhead is expected with the WMC approach: the CNF encoding, file I/O to the GPMC subprocess, and solver initialisation costs dominate at the 8-qubit scale where the

model counting problem itself is trivially small. As shown in Experiment 02, this overhead inverts at larger qubit counts, where GPMC efficiently uses the circuit's repetitive convolutional structure and scales gracefully while state-vector simulation fails entirely past a certain number of qubits.

Experiment 02 Experiment 02 provides the primary quantitative result of this thesis. In single-threaded execution, Quokka# matches PennyLane's state-vector runtime at $n \approx 16$ qubits and surpasses it beyond $n = 20$ qubits, and at $n = 24$ both `default.qubit` and `lightning.qubit` exhaust available memory and time constraints. Beyond this crossover, Quokka# exhibits polynomial (exponent ≈ 1.4) growth: wall time increases from approximately 2.5 s at $n = 32$ qubits to approximately 18.2 s at $n = 128$ qubits: a factor of 7.3× over a 4× expansion in the number of QCNN parameters and a 2^{96}-fold expansion in Hilbert space dimension. This behaviour arises because the WMC solver exploits the repetitive gate structure of the QCNN: identical convolutional and pooling blocks share CNF structure, and the GPMC solver's caching and component decomposition take advantage of this regularity.

Validation The correctness check described in Section 4 passed for all configurations tested, with $\Delta\langle Z\rangle < 1 \times 10^{-6}$ between Quokka# and `default.qubit` across tested qubits up to $n = 24$, the limit of the PennyLane backends. This confirms that the WMC-based encoding is numerically equivalent to state-vector simulation for the QCNN circuits studied here and that the runtime advantage of Quokka# is not achieved at the cost of approximation error.

Final Conclusion Together, these findings establish a clear and quantitatively supported answer to the research question: Weighted Model Counting can substantially improve the scalable simulation of hierarchical QCNNs at circuit sizes where traditional state-vector approaches are memory-infeasible, and it does so with exact probability estimation. The practical barrier for state-vector simulation at approximately 24 qubits has been a consistent obstacle across the QCNN literature [HKP22, CCL19], and the results of Experiment 02 demonstrate that this barrier does not apply to the WMC approach. Circuits up to 128 qubits were simulated in under 19 s single-core/5 s parallelised per epoch on local mid-range hardware (AMD Ryzen 5 3600X, 48 GB RAM), with no approximation introduced into the probability estimates. The research question can thus be positively answered: the WMC method makes scalable exact emulation of hierarchical QCNNs possible at qubit numbers where even state-vector methods become completely unfeasible.

5.2 Further research

This thesis demonstrates that WMC-based simulation is a viable and scalable alternative to state-vector approaches for QCNNs; the following directions build on this foundation and suggest opportunities for further research.

WMC Solver Firstly, the performance of the WMC simulator depends heavily on its solver. GPMC was used as the back-end counter. However, more recent counters show better performance. For example, Soos and Meel introduced Ganak2 [SM25], a re-engineered version that solves more benchmark instances than the previous baselines within the same time limit [SM25]. Using Ganak as the back-end solver for Quokka# is thus an improvement that should show significant speed-ups.

Larger QCNN trainings Additionally, a fully trained QCNN at larger qubit counts (such as 20 or more qubits) was not benchmarked against PennyLane due to time constraints, as it required more than 30-epochs for training. Experiment 02 demonstrates that Quokka# can simulate circuits up to 128 qubits in reasonable time, but the accuracy of those models was not the focus. A complete training pipeline at 12 qubits run to convergence and compared against PennyLane's `default.qubit` would create a more complete picture of whether WMC-based simulation reproduces the same decision model and final accuracy as state-vector simulation.

Accuracy Moreover, classification accuracy was not the primary objective of this thesis, which dealt with the viability of WMC simulators through benchmarking at runtime. Nevertheless, classification accuracy is obviously a critical part of any coherent approach to QCNNs, and more research is needed into this issue. Recently, it has been demonstrated that even small-scale circuits with as few as 12 to 51 parameters are capable of providing good accuracy in classification tasks on common data sets like MNIST and Fashion-MNIST [HKP22]. Performing a comprehensive analysis involving variation in the ansatz architecture, the encoding scheme, and the amount of training data via Quokka# simulator could help determine whether the accuracy performance of WMC-based learning is comparable to the results obtained via state-vector simulators.

Formal Verification The correctness check used in this thesis followed the approach by Burgholzer et al. [BKW21], but there is no guarantee on equivalence. Formal equivalence checking guarantees the two implementations return the same output with respect to all possible inputs. Thanos et al. have recently devised an efficient equivalence checker for Clifford circuits [TCL23]. Using such a verification technique to check the correctness of the Quokka# QCNN against PennyLane could replace the current probabilistic correctness check with an exact one.

Memory consumption There is also an additional aspect of resource utilization that has not been explored here - memory consumption. Runtime was selected as the main dependent variable because of its obvious connection to computational feasibility of the problem under investigation. However, memory usage for both the encoding process and for the GPMC solver itself increases along with increasing circuit complexity, but exact characteristics of this growth have yet to be discovered specifically for QCNN tasks. state-vector simulators like `default.qubit` and `lightning.qubit` face hard limits at around 24 qubits because of the necessity to store the $2^n \times 16$ byte state-vector [ADP+24]. A study focusing on the memory consumption of Quokka# in addition to wall times across the same qubit range investigated in experiment 02 would provide an additional perspective on the trade-off between memory and computing requirements.

GPU enhancement Finally, GPU-assisted simulation using the `lightning.tensor` [ADP+24] approach provides an excellent alternative to compare against, which was briefly considered in this research but should be further investigated. The `lightning.tensor` device implements quantum states via their approximation as a Matrix Product State (MPS) model and thereby scales polynomially rather than exponentially in terms of memory usage. Given how tensor networks have been the method of choice for GPU-assisted quantum ML in recent years, finding out the crossover between MPS simulation and WMC simulation in terms of entanglement and qubit number would be of high practical importance.

References

[ADP+24] Ali Asadi, Amintor Dusko, Chae-Yeun Park, Vincent Michaud-Rioux, Isidor Schoch, Shuli Shu, Trevor Vincent, and Lee James O'Riordan. Hybrid quantum programming with pennylane lightning on hpc platforms, 2024.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[BIS+18] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Carsten Blank, Keri McKiernan, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[BKW21] Lukas Burgholzer, Richard Kueng, and Robert Wille. Random stimuli generation for the verification of quantum circuits. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.

[BP00] Roberto J. Bayardo and Joseph D. Pehoushek. Counting models using connected components. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI)*, pages 157–162, 2000.

[CCL19] Iris Cong, Soonwon Choi, and Mikhail D. Lukin. Quantum convolutional neural networks. *Nature Physics*, 15(12):1273–1278, 2019.

[CD08] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008.

[CJAA+22] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[Coo71] Stephen A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[Fey82] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6–7):467–488, 1982.

[FN18] Edward Farhi and Hartmut Neven. Classification with quantum neural networks on near term processors, 02 2018.

[Gay11] Simon J. Gay. Stabilizer states as a basis for density matrices. *arXiv preprint arXiv:1112.2156*, 2011.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[Got97] Daniel Gottesman. *Stabilizer Codes and Quantum Error Correction*. PhD thesis, California Institute of Technology, 1997.

[Has20] Kenji Hashimoto. GPMC. https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC, 2020.

[HBM+21] Hsin-Yuan Huang, Michael Broughton, Masoud Mohseni, Ryan Babbush, Sergio Boixo, Hartmut Neven, and Jarrod R. McClean. Power of data in quantum machine learning. *Nature Communications*, 12(1):2631, 2021.

[HKP22] Tak Hur, Leeseok Kim, and Daniel K Park. Quantum convolutional neural network for classical data classification. *Quantum Machine Intelligence*, 4(1):3, 2022.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, New York, 2nd edition, 2009.

[JATK+24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.

[Jol02] Ian T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York, 2 edition, 2002.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KLM+24] Matthias Klusch, Jörg Lässig, Daniel Müssig, Antonio Macaluso, and Frank K. Wilhelm. Quantum artificial intelligence: A brief survey. *KI - Künstliche Intelligenz*, 38(4):257–276, November 2024. Open access.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[MBL24] Jingyi Mei, Marcello Bonsangue, and Alfons Laarman. Simulating quantum circuits by model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 555–578, Cham, 2024. Springer Nature Switzerland.

[MML24] Jingyi Mei, Jan Martens, and Alfons Laarman. Disentangling the gap between quantum and #sat. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 17–40, Cham, 2024. Springer Nature Switzerland.

[Mol86] Cleve Moler. Matrix computation on distributed memory multiprocessors. In Michael T. Heath, editor, *Hypercube Multiprocessors*, Philadelphia, 1986. Society for Industrial and Applied Mathematics.

[MSS96] João P. Marques-Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, 1996.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, 2000.

[Pre18] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[QMCL25] Arend-Jan Quist, Jingyi Mei, Tim Coopmans, and Alfons Laarman. Advancing quantum computing with formal methods. In Andre Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods*, pages 420–446, Cham, 2025. Springer Nature Switzerland.

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Theory and Applications of Satisfiability Testing (SAT)*, 2004.

[SJAG19] Sukin Sim, Peter D. Johnson, and Alán Aspuru-Guzik. Expressibility and entangling capability of parameterized quantum circuits for hybrid quantum-classical algorithms. *Advanced Quantum Technologies*, 2(12):1900070, 2019.

[SM25] Mate Soos and Kuldeep S. Meel. Engineering an efficient probabilistic exact model counter. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, volume 15933 of *Lecture Notes in Computer Science*. Springer, Cham, 2025.

[SS10] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.

[TCL23] Dimitrios Thanos, Tim Coopmans, and Alfons Laarman. Fast equivalence checking of quantum circuits of clifford gates. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 199–216, Cham, 2023. Springer Nature Switzerland.

[Ten23] Shang-Hua Teng. "intelligent heuristics are the future of computing". *ACM Trans. Intell. Syst. Technol.*, 14(6), November 2023.

[Val79] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

A Experiment 1 Runtime Results

Wall-clock runtimes (seconds) for Experiment 1 across all training configurations. Quokka# is shown under single-core and 12-core parallel execution; the higher performing `lightning.qubit` PennyLane backend is used.

Epochs	n_{train}	**Quokka# (1 core)** (s)	**Quokka# (12 cores)** (s)	**lightning.qubit** (s)
10	10	59.1	7.6	5.6
	30	169.8	20.5	14.2
	50	284.9	31.8	26.5
	70	399.4	43.0	33.5
	90	491.4	54.9	40.4
20	10	104.2	14.0	9.9
	30	309.2	38.6	26.8
	50	512.7	62.0	44.2
	70	709.3	84.2	62.4
	90	911.8	108.9	77.9
30	10	151.5	20.6	13.4
	30	454.1	57.7	39.1
	50	757.2	91.9	63.7
	70	1,079.6	125.7	93.3
	90	1,409.7	162.3	126.1
40	10	210.5	27.5	19.2
	30	628.0	76.5	56.1
	50	1,036.2	122.6	86.3
	70	1,424.7	167.3	122.6
	90	1,820.0	215.7	157.1
50	10	253.6	34.1	22.2
	30	760.4	95.8	66.6
	50	1,272.6	152.9	108.5
	70	1,805.4	209.0	155.8
	90	2,337.0	269.5	196.7

B Experiment 2 Runtime Results

Wall-clock runtimes (seconds) for Experiment 2 across all qubit counts. Quokka# is shown under single-core and 12-core parallel execution. PennyLane `default.qubit` and `lightning.qubit` were skipped beyond 24 qubits due to time and memory constraints.

Qubits	**Quokka# (1 core)** (s)	**Quokka# (12 cores)** (s)	**default.qubit** (s)	**lightning.qubit** (s)
4	0.855	0.842	0.070	0.328
8	0.482	0.279	0.089	0.055
12	0.824	0.367	0.233	0.082
16	1.051	0.470	1.303	0.253
20	1.525	0.543	25.606	3.185
24	1.860	0.665	544.710	91.121
28	2.228	0.751	n/a	n/a
32	2.498	0.825	n/a	n/a
36	3.296	1.029	n/a	n/a
40	3.710	1.125	n/a	n/a
44	4.221	1.325	n/a	n/a
48	4.618	1.397	n/a	n/a
52	5.172	1.560	n/a	n/a
56	5.770	1.721	n/a	n/a
60	6.119	1.803	n/a	n/a
64	6.682	1.978	n/a	n/a
68	7.854	2.088	n/a	n/a
72	8.441	2.229	n/a	n/a
76	9.086	2.425	n/a	n/a
80	9.437	2.514	n/a	n/a
84	10.414	2.754	n/a	n/a
88	11.402	2.966	n/a	n/a
92	11.828	3.135	n/a	n/a
96	12.054	3.150	n/a	n/a
100	13.145	3.450	n/a	n/a
104	13.928	3.630	n/a	n/a
108	14.656	3.791	n/a	n/a
112	15.409	4.060	n/a	n/a
116	16.211	4.210	n/a	n/a
120	16.726	4.312	n/a	n/a
124	17.092	4.421	n/a	n/a
128	18.147	4.750	n/a	n/a