Toward a Quantum Advantage in Deep Learning Architectures

Sergio Rodríguez Vidal

Mathematical Engineering and Artificial Intelligence
Comillas Pontifical University ICAI
Madrid, Spain
June 17, 2025

Contents

PREFACE

Unlike classical deep learning [1, 2], which emerged from decades of gradual refinement and broad consensus around its foundational ideas, **quantum neural networks (QNNs)** are currently in a thrilling yet nascent stage. **Quantum computing (QC)** presents itself as an essential next step to overcome critical limitations of current deep learning methodologies; particularly the escalating energy consumption and computational demands associated with increasingly large-scale models. To illustrate the magnitude of this challenge, projections [3] indicate that the United States alone could reach approximately 325 TWh of annual electricity usage from AI activities by 2028 (comparable to the total power consumption of Spain), highlighting the urgent need for more sustainable computational solutions.

Despite significant theoretical advances suggesting that quantum neural networks *could* outperform classical models in expressivity and efficiency [4], there is lack of a **well-packaged, end-to-end framework**. Today, newcomers must piece together scattered papers, code repositories, and tutorials, each using slightly different notation, training tricks, and tooling. A clearly organized "starter kit" that unifies core concepts, software, and best practices has yet to emerge.

The objective of this book is precisely to bridge these gaps. Our approach combines theory, practical tools, and hands-on guidance explicitly aimed at **AI practitioners, mathematicians, computer scientists, and engineers** who may not yet be familiar with quantum computing. We begin by introducing the core concepts of quantum mechanics; not in exhaustive detail, but sufficiently to grasp its unique properties, such as superposition and entanglement, and their potential for enhancing neural network architectures. Clearly outlining the sustainability and scalability problems faced by classical deep learning, we demonstrate how quantum neural networks could offer viable solutions by leveraging these quantum phenomena, potentially reducing energy usage and enabling the scaling of more complex models within practical computational and environmental constraints.

To facilitate practical engagement and experimentation, this book also serves as a guide to a complementary resource: **an accessible Python library**. The text provides readers with a clear, rigorous, and practical guide to the theoretical foundations, standardized frameworks, ongoing challenges, and the practical implementation of QNNs using this library. While familiarity with quantum physics and advanced mathematics is helpful—providing deeper insights into quantum mechanics and quantum gates—it is not strictly necessary to begin meaningful work with quantum neural networks. The text carefully introduces essential mathematical concepts and quantum phenomena,

emphasizing their key differences from classical neural network models and explaining how these differences translate into computational advantages.

The Python library developed alongside this book is explicitly designed to enable **PyTorch** users to experiment and seamlessly integrate quantum neural networks into their existing workflows without facing steep learning curves associated with quantum computing. Our library adheres closely to two core design principles inspired by PyTorch [5] itself:

- **Usability over Performance**: Prioritizing ease of experimentation and integration to encourage widespread adoption, even if at times sacrificing raw computational efficiency.
- **Simple Over Easy**: Ensuring explicitness and clarity over convenience, thus simplifying debugging, customization, and deeper understanding of QNN architectures.

Leveraging both **PyTorch and CUDA Quantum (CUDA-Q)**, our library provides a modular, transparent, and extendable framework, welcoming both newcomers who prefer predefined models and experts who desire extensive customization.

Ultimately, this project aims not merely to educate but also to engage. By clearly defining the critical challenges classical AI currently faces, detailing how QNNs address these issues, and providing practical tools to facilitate experimentation; we encourage collaboration, innovation, and meaningful interdisciplinary advancements. The pathway ahead is challenging yet immensely promising. Welcome to **quantum enhanced deep learning**, let's advance the frontier together.

QUANTUM MECHANICS

Quantum mechanics represents a **radical departure** from classical physics, revealing a realm governed by probabilities, wave-like behaviors, and the subtle interplay of measurement. Where classical physics outlines a deterministic view of reality, quantum mechanics instead posits that systems can exist in a blend of all possible states, described by a complex **wave function**. This wave function encodes every potential outcome of a **measurement**—an act that collapses these possibilities into a single, observed result.

These counterintuitive features (**superposition, interference and collapse**) have been harnessed in quantum computing, where information is encoded in **qubits** that can exist in superpositions of basis states and become entangled. In this chapter we'll introduce the wave function and its evolution under the **Schrödinger equation**, examine how measurements govern quantum behavior, and present the linear-algebraic framework and **Dirac bra–ket notation** that underlies it all. Equipped with these tools, you'll be ready to appreciate the power of Quantum Neural Networks.

1 Introduction

Quantum mechanics is not just about subatomic particles. It is our best fundamental description of nature to date, with successful predictions from the *microscopic* realm up to *mesoscopic* systems. **Classical physics** appears accurate when quantum effects become too subtle to notice, typically at larger masses or higher energies, allowing us to perceive a seemingly "classical" world. However, at microscopic or carefully engineered mesoscopic scales, intriguing **quantum phenomena** become prominent.

2 The Wave Function

Consider a particle of mass m constrained to move along the x axis, under the influence of some specified force F. Classically, we predict the particle's position $x(t)$ by applying **Newton's second law**,

$$F = m\frac{\mathrm{d}^2x}{\mathrm{d}t^2},$$

together with appropriate initial conditions (e.g., position and velocity at $t = 0$). From $x(t)$ this classical solution, we can determine **momentum** $p(t)$, kinetic energy $K(t)$, or any other physical variable of interest.

Quantum mechanics approaches the same problem rather differently. We look for the **particle's wave function**, $\Psi(x,t)$, by solving the **Schrödinger equation**:

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = \hat{H}\Psi(x,t), \tag{1}$$

where the **Hamiltonian operator** $\hat{H}$

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x). \tag{2}$$

encodes kinetic and potential energy. Here $i = \sqrt{-1}$, $\hbar$ is Planck's constant h divided by 2π and $V(x)$ represents the potential energy. This equation is logically analogous to Newton's law, in the sense that once you specify $\Psi(x,0)$ ("initial conditions"), the Schrödinger equation determines $\Psi(x,t)$ for all future times t. The derivative of the potential energy function $\frac{\partial V}{\partial x}$ yields the classical force F.

2.1 Probabilistic Interpretation

The wave function itself, however, does not directly correspond to a classical **observable** like position or momentum [6]. After all, a particle is localized at a point, whereas the wave function is spread out over space. Max Born's **statistical interpretation** resolves this:

$$Pr\{x \in [a,b]\} = \int_a^b |\Psi(x,t)|^2. \tag{3}$$

is the probability density of finding the particle between a and b at time t. Graphically, the probability of discovery between a and b is the area under $|\Psi(x,t)|^2$ from a to b.

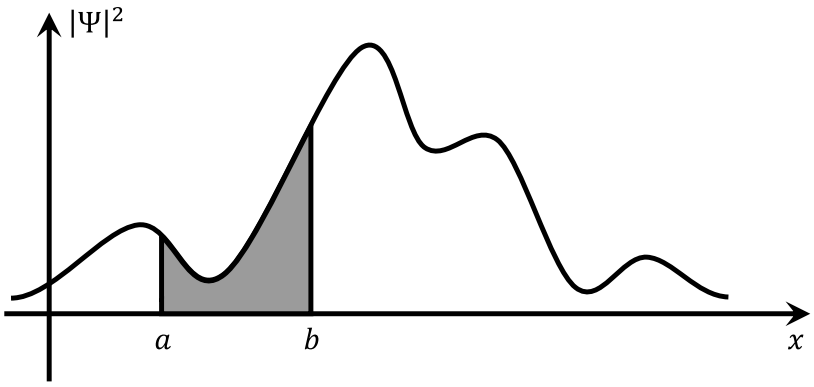


Figure 1: Max Born's Statistical Interpretation

Because of **Born's rule**, the wave function must be normalized so that all possibilities add up to probability 1. Mathematically, this means

$$\int_{-\infty}^{\infty} p(x,t)dx = \int_{-\infty}^{\infty} |\Psi(x,t)|^2 dx = 1, \tag{4}$$

for a single particle in one dimension. Any candidate Ψ must be normalized (or normalizable) to represent a valid physical state.

This probabilistic interpretation introduces fundamental **indeterminacy**: even with complete knowledge of Ψ, quantum mechanics can only predict statistical outcomes of position measurements. Upon measuring and finding the particle at a fixed arbitrary position x, two philosophical attitudes address "Where was it just before the click?"

1. **Realist.** The particle was at x all along; quantum mechanics is simply incomplete, lacking hidden variables that fix its true location.

2. **Copenhagen.** Prior to measurement the particle wasn't anywhere definite; the act of measurement forces it into a definite x. Observations "produce" properties, rather than merely revealing them.

Decades later, **Bell's theorem** showed this was experimentally testable, and experiments have sided with the Copenhagen picture: **Quantum systems**[1] do not have definite values for **physical observables** such as position or momentum until measured. **Measurement** outcomes are inherently probabilistic, determined by the amplitude squared of the wave function.

2.2 Measurement and Wave Function Collapse

Performing a measurement on a quantum system fundamentally alters its state, an event known as **wave function collapse**. Before measurement, the wave function evolves smoothly according to the Schrödinger equation, describing multiple possibilities simultaneously. Upon measurement, however, this smooth evolution abruptly and discontinuously changes (an instantaneous "collapse") into a single **eigenstate** consistent with the measurement outcome.

Intuitively, measurement in quantum mechanics is fundamentally different from classical observation. In classical physics, measuring a property of a system merely reveals a **pre-existing state**. In quantum mechanics, prior to measurement, properties like

[1] A **quantum system** is any physical entity or collection of entities whose behavior must be described by quantum mechanics rather than classical laws. Examples include single particles (electrons, photons), composite objects (atoms, molecules) or engineered qubit devices in quantum computing.

position or momentum do not have definite values; instead, the particle exists in a **superposition** of possibilities. The measurement itself forces the system to "choose" one possibility out of the superposition, collapsing the wave function into a single outcome.

The collapse is random yet statistically predictable: the probability of each outcome is determined by the squared magnitude of the corresponding amplitude.

The Double-slit Experiment

A practical illustration of wave function collapse is provided by the famous **double-slit experiment**. In this experiment, electrons are fired individually toward a barrier with two narrow slits. Classically, we expect electrons, treated as particles, to travel through one slit or the other, producing two bright bands directly behind the slits. However, quantum mechanically, each electron behaves as a wave that passes through both slits simultaneously, creating an interference pattern of multiple bright and dark fringes on a detection screen behind the barrier.

Remarkably, even when electrons are fired one at a time (ensuring only one electron is present in the apparatus at any given moment) the cumulative effect over many electrons still forms an interference pattern. Each electron leaves a single discrete spot on the detector, yet collectively, these spots build up the characteristic wave-like pattern. This demonstrates that each electron interferes with itself, traversing both slits simultaneously as described by its wave function.

However, if we attempt to measure which slit the electron passes through, the situation dramatically changes. Any attempt to detect the electron's path immediately destroys the interference pattern, leaving only two distinct bands aligned with each slit. This occurs because measuring the electron's position forces the wave function to collapse to a definite path; either through one slit or the other, but not both. The superposition necessary for interference is thus eliminated by the measurement.

It is worth noting that in most practical scenarios the true wave function of a quantum system is unknown; however, by repeatedly sampling measurement outcomes according to the system's probability distribution, one can employ continuous (**Monte Carlo**) sampling techniques to reconstruct or estimate the underlying wave function.

In **quantum computing** and **quantum machine learning**, wave function collapse has profound implications. Effective quantum algorithms carefully manage these probabilities, ensuring desirable outcomes have high probabilities and minimizing undesired ones.

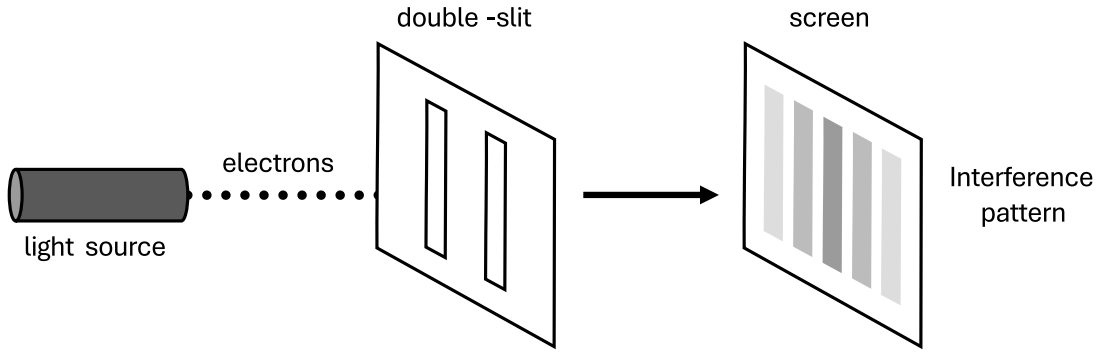


Figure 2: Double-slit Experiment without measurement

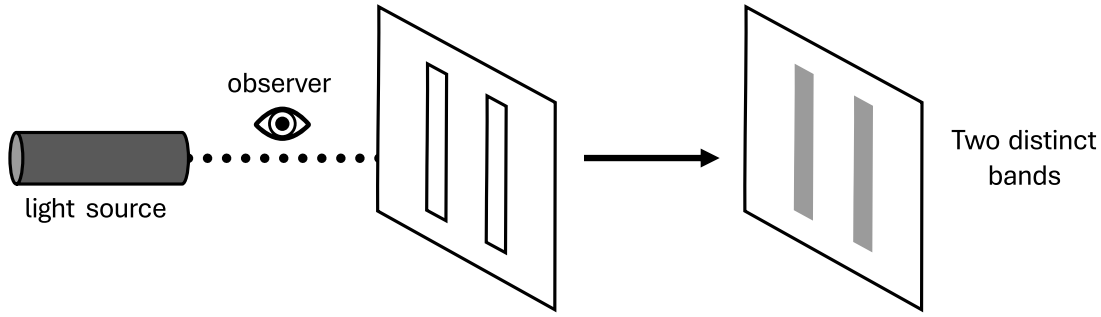


Figure 3: Double-slit Experiment with measurement

2.3 Continuous and Finite-Level Systems

Thus far we have treated quantum states by choosing the **continuous position-space basis** $\{|x\rangle\}$. Each basis vector $|x\rangle$ corresponds to a state perfectly localized at the point x and is represented by a column vector whose entries form the **Dirac delta function** $\delta(x' - x)$: zero everywhere except at $x' = x$, where it exhibits an infinitely sharp, unit-area spike. We then describe the state via the wave function

$$\Psi(x,t) = \langle x|\Psi(t)\rangle, \qquad \int_{-\infty}^{\infty} |\Psi(x,t)|^2\, dx = 1,$$

which evolves as

$$i\hbar\,\frac{\partial}{\partial t}\,\Psi(x,t) = \left[-\tfrac{\hbar^2}{2m}\nabla^2 + V(x)\right]\Psi(x,t).$$

Measuring $x \in \mathbb{R}$ yields outcomes with probability density $|\Psi(x,t)|^2$, and similarly for momentum using the momentum basis $\{|p\rangle\}$.

Some systems admit only a **finite set** of distinguishable outcomes; think of an electron's spin or an atom with N low-lying levels. In these cases we pick a discrete or-

thonormal basis e.g. $\{|n\rangle\}_{n=1}^{N} = \big\{(1,0,\ldots,0)^T, (0,1,\ldots,0)^T, \ldots, (0,0,\ldots,1)^T\big\}$, extract the amplitude of $|\Psi(t)\rangle$ along the n-th direction by projecting with $\langle n|$, giving

$$\Psi_n(t) = \langle n|\Psi(t)\rangle = (\underbrace{0,\ldots,0}_{n-1},\, 1,\, \underbrace{0,\ldots,0}_{N-n}) \begin{pmatrix} \Psi_1(t) \\ \Psi_2(t) \\ \vdots \\ \Psi_N(t) \end{pmatrix}, \quad \sum_{n=1}^{N} \big|\Psi_n(t)\big|^2 = 1.$$

Here $|\Psi_n(t)|^2$ gives the probability of outcome n, and for any observable Q with eigenvalues q_n in this basis,

$$\langle Q\rangle = \sum_{n=1}^{N} q_n\, |\Psi_n(t)|^2.$$

In their **abstract operator form**, both continuous-variable and finite-level quantum systems satisfy exactly the same Schrödinger equation,

$$i\hbar\, \frac{d}{dt}\, |\Psi(t)\rangle = \hat{H}\, |\Psi(t)\rangle\,, \tag{5}$$

which in a chosen basis becomes either a partial differential equation (continuous) or a system of ordinary differential equations (finite-level). In the discrete basis the matrix elements

$$H_{jk} = \langle j|\hat{H}|k\rangle = (\underbrace{0,\ldots,0}_{j-1},\, 1,\, \underbrace{0,\ldots,0}_{N-j})\; \hat{H}\; (\underbrace{0,\ldots,0}_{k-1},\, 1,\, \underbrace{0,\ldots,0}_{N-k})^T$$

give

$$i\hbar\, \dot{\Psi}_j(t) = \sum_{k=1}^{N} H_{jk}\, \Psi_k(t).$$

When the configuration space itself is discrete (e.g. an electron hopping on lattice sites labeled n[2]), we use the site basis $\{|n\rangle\}$. A tight-binding Hamiltonian with hopping amplitude t and on-site energies V_n has nonzero couplings $H_{n,n\pm1} = -t$ and $H_{n,n} = V_n$, yielding

$$i\hbar\, \dot{\Psi}_n(t) = -\,t\big[\Psi_{n+1}(t) + \Psi_{n-1}(t)\big] + V_n\, \Psi_n(t).$$

[2]A model in which the electron's position is restricted to discrete, equally spaced points in a regular array, each uniquely identified by the integer index n.

Example: Electron Spin in a Magnetic Field

An electron carries an internal binary property known as **spin**. If we measure the component of that spin along a fixed axis (say the z-axis), the result is always one of just two values: “up" and “down".

Therefore an electron’s spin along the z-axis lives in a two-dimensional space with basis $\{|\uparrow\rangle, |\downarrow\rangle\}$. We write

$$\Psi_\uparrow(t) = \langle\uparrow|\Psi(t)\rangle, \qquad \Psi_\downarrow(t) = \langle\downarrow|\Psi(t)\rangle, \qquad |\Psi_\uparrow|^2 + |\Psi_\downarrow|^2 = 1.$$

In a uniform magnetic field B along z, the Hamiltonian is

$$\hat{H} = -\gamma B\, S_z = \frac{\hbar\omega}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad \omega = \gamma B.$$

The Schrödinger equation

$$i\hbar\,\frac{d}{dt}\begin{pmatrix} \Psi_\uparrow \\ \Psi_\downarrow \end{pmatrix} = \begin{pmatrix} \frac{\hbar\omega}{2} & 0 \\ 0 & -\frac{\hbar\omega}{2} \end{pmatrix}\begin{pmatrix} \Psi_\uparrow \\ \Psi_\downarrow \end{pmatrix}$$

gives the solutions

$$\Psi_\uparrow(t) = \Psi_\uparrow(0)e^{-i\omega t/2}, \quad \Psi_\downarrow(t) = \Psi_\downarrow(0)e^{+i\omega t/2},$$

so that the populations $|\Psi_\uparrow|^2$, $|\Psi_\downarrow|^2$ stay fixed while their relative phase oscillates at frequency ω.

With this unified, single-section treatment, the *only* real difference between **continuous and finite-level quantum systems** is the nature of the index set ($\mathbb{R}$ versus a finite list); yet in both cases they obey the *same* Schrödinger equation, normalization condition, and probability postulates.

3 Time-Independent Schrödinger Equation

In most quantum-computing and quantum-neural-network applications the Hamiltonian $\hat{H}$ is taken to be **time-independent**, so that all nontrivial dynamics reduce to a global phase. One is then led directly to the **energy eigenproblem**, whose eigenvectors furnish a **stationary basis** ideal for encoding and processing quantum information.

3.1 Stationary States and the Energy Eigenproblem

When the Hamiltonian H is time-independent, the general Schrödinger equation allows the physicist's favorite trick, *separation of variables*, to turn the Schrödinger equation (5)

$$i\hbar \frac{d}{dt} \, |\Psi(t)\rangle = \hat{H} \, |\Psi(t)\rangle$$

into two ordinary differential equations. Because the left-hand side depends only on t and the action of H on $\Psi(t)$ depends only on the state-space structure, **linearity** of H guarantees we can write

$$|\Psi(t)\rangle = |\psi\rangle \, \varphi(t) \quad \Longrightarrow \quad i\hbar \, |\psi\rangle \, \frac{d\varphi}{dt} = \varphi \, H \, |\psi\rangle \implies i\hbar \, \frac{1}{\varphi} \frac{d\varphi}{dt} = \frac{1}{|\psi\rangle} \, H \, |\psi\rangle = E,$$

where E is a constant (the separation constant). Integrating the t–equation gives the ubiquitous "wiggle" factor,

$$\varphi(t) = e^{-\,iEt/\hbar},$$

while the remaining condition becomes the **time-independent Schrödinger equation** (or energy eigenproblem)

$$\hat{H} \, |\psi\rangle = E \, |\psi\rangle \, . \tag{6}$$

In general the Hamiltonian $\hat{H}$ admits a whole *spectrum* of solutions. Each stationary solution is

$$|\Psi_n(t)\rangle = |\psi_n\rangle \, e^{-\,iE_n t/\hbar},$$

with $\big|\psi_n\big\rangle$ an **eigenstate** of $\hat{H}$. These ***stationary states*** have two key properties:

1. **Time-independent probabilities.** Although $|\Psi_n(t)\rangle$ carries that phase $e^{-iE_n t/\hbar}$, the probability density $\big||\Psi_n(t)\rangle\big|^2 = \big||\psi_n\rangle\big|^2$ is constant in time.

2. **Definite energy.** One finds

   $$\langle H\rangle_n = \langle\psi_n|\hat{H}|\psi_n\rangle, \quad \big\langle(\Delta H)^2\big\rangle_n = \langle\psi_n|(\hat{H}-E_n)^2|\psi_n\rangle = 0,$$

   so every measurement of the total energy in state ψ_n returns E_n with certainty.

3.2 Superposition and Interference

Because the eigenstates $\{\psi_n\}$ form a complete basis (discrete or continuous), any initial wave function can be expanded as

$$|\Psi(0)\rangle = \sum_n c_n \, |\psi_n\rangle \quad \big(\text{or } |\Psi(0)\rangle = \int c(E) \, |\psi_E\rangle \, dE\big),$$

where the coefficients c_n are complex **probability amplitudes**, whose magnitudes relate to the likelihood of finding the system in one state or the other. Linearity of the full Schrödinger equation then guarantees the general time evolution

$$|\Psi(t)\rangle = e^{-\frac{i}{\hbar}\hat{H}t} \, |\Psi(0)\rangle = \sum_n c_n \, e^{-iE_n t/\hbar} \, |\psi_n\rangle \quad \big(\text{or } \int c(E) \, |\psi_E\rangle \, e^{-iEt/\hbar} \, dE\big).$$

Each term carries its own phase, and their weighted sum is precisely the quantum **superposition**.

A useful way to visualize superposition is to think of a **quantum state** as a "blend" of several classical possibilities, each weighted by a complex amplitude. Because the Schrödinger equation is a linear differential equation, whenever $|\psi_1\rangle$ and $|\psi_2\rangle$ are *energy eigenstates* of $\hat{H}$ (i.e. solutions of Eq.6), their **linear combination**

$$|\psi\rangle = c_1 \, |\psi_1\rangle + c_2 \, |\psi_2\rangle$$

is also a valid quantum state.

It should be noted that quantum amplitudes c_n can interfere *constructively* or *destructively*, leading to observable **interference patterns** when measurements are performed [7]. This interference of probability amplitudes has no classical analog and is one of the hallmarks of quantum behavior.

In the realm of quantum computing, superposition provides a powerful advantage. **Quantum computers** leverage superposition to represent and process a vast number of computational states simultaneously. Unlike a **classical bit** that exists in a definite state (0 or 1), a **qubit** can exist in a superposition of both states, enabling a form of parallelism that can greatly enhance computational speed for certain tasks.

4 Formalism

In the previous sections we have encountered wave functions, operators, and the Schrödinger equation in both continuous and discrete settings. To unify these ideas and prove the general theorems that underlie quantum behavior, we must cast the theory into the language of abstract vector spaces and linear operators. This abstract framework (built on **Hilbert spaces** and the **Dirac (bra–ket) notation**) provides the natural habitat for quantum states and observables, streamlines computations, and makes manifest the deep connection between quantum mechanics and linear algebra. In this section we introduce the key constructs and notation of this formalism.

4.1 Hilbert space

A **Hilbert space** is a real or complex vector space $\mathcal{H}$ equipped with an inner product, and complete in the norm induced by that inner product.

$$\langle \cdot , \cdot \rangle : \mathcal{H} \times \mathcal{H} \ \rightarrow \ \mathbb{K}, \quad \mathbb{K} = \mathbb{R} \text{ or } \mathbb{C},$$

In **quantum mechanics**, a "state" can be viewed as a complex vector in such a space:

- **Continuous systems:** The space of square-integrable functions on the real line,

$$L^2(\mathbb{R}) = \Big\{ \Psi : \mathbb{R} \to \mathbb{C} \ \Big| \ \int_{-\infty}^{\infty} |\Psi(x)|^2 \, \mathrm{d}x < \infty \Big\},$$

  with the inner product

$$\langle \Phi, \Psi \rangle = \int_{-\infty}^{\infty} \Phi^*(x) \, \Psi(x) \, \mathrm{d}x.$$

  where $\Phi^*(x)$ denotes the complex conjugate of $\Phi(x)$.

- **Finite-level systems:** The space $\mathbb{C}^N$ of N-component column vectors $\bar{\psi} = (\psi_1, \ldots, \psi_N)^T$, with the standard inner product

$$\langle \bar{\phi}, \bar{\psi} \rangle = \sum_{n=1}^{N} \phi_n^* \, \psi_n.$$

Normalization of a state ψ requires $\langle \bar{\psi}, \bar{\psi} \rangle = 1$, ensuring total probability unity. Two state vectors $\bar{\phi}$ and $\bar{\psi}$ are **orthogonal** if $\langle \bar{\phi}, \bar{\psi} \rangle = 0$, and an **orthonormal** set $\{e_n\}$ satisfies

$$\langle e_m, e_n \rangle = \delta_{mn}.$$

A **basis** is complete when any state in the Hilbert space can be expanded uniquely in that basis.

4.2 Dirac (bra–ket) notation

Although we have silently introduced **Dirac notation** in sections 2.3 and 3, it is worth pausing now to state its syntax and conventions clearly. In Dirac language a **quantum state** $\psi \in \mathcal{H}$ is written as a ***ket*** $|\psi\rangle$. In an N-dimensional orthonormal basis $\{|e_n\rangle\}_{n=1}^{N}$ its components are the *projections*

$$\psi_n = \langle e_n|\psi\rangle,$$

so the ket is represented by the column vector

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_N \end{pmatrix}.$$

Taking the Hermitian adjoint (conjugate transpose) of a ket $|\psi\rangle$ yields an element of the dual space $\mathcal{H}^*$, the ***bra***

$$\langle\psi| = \big(|\psi\rangle\big)^\dagger = \begin{pmatrix} \psi_1^* & \psi_2^* & \cdots & \psi_N^* \end{pmatrix},$$

and the **inner product** with another state $|\phi\rangle$ becomes

$$\langle\phi|\psi\rangle = \langle\phi|\ |\psi\rangle = \sum_{n=1}^{N} \phi_n^*\, \psi_n,$$

linear in the ket and conjugate-linear in the bra.

A convenient shorthand we shall use frequently is the **completeness relation**. Given any orthonormal basis $\{|e_n\rangle\}_{n=1}^{N}$ we have

$$\mathbb{I} = \sum_n |e_n\rangle\, \langle e_n|\,,$$

so that every state may be unpacked as

$$|\psi\rangle = \sum_n |e_n\rangle\langle e_n|\, \psi\rangle = \sum_n \langle e_n|\psi\rangle\ |e_n\rangle$$

Each operator $|e_n\rangle\langle e_n|$ acts as a **projector** onto the one-dimensional subspace spanned by $|e_n\rangle$, and the completeness relation simply states that these projectors together span the whole space.

Nothing changes in the passage from discrete indices to a **continuous** label. In the position basis $\{|x\rangle \mid x \in \mathbb{R}\}$ we replace sums by integrals, write $\Psi(x) = \langle x|\psi\rangle$, and trade the discrete completeness relation for

$$\int_{-\infty}^{\infty} |x\rangle\langle x| \, dx = \mathbb{I}.$$

All manipulations that work for finite vectors therefore carry over verbatim to continuous systems.

As discussed in previous sections, the state of a system is not known to be a single ket but rather a superposition of several eigenstates $\{|\psi_n\rangle\}_{n=1}^{N}$ with classical probabilities $\{c_n\}$. The correct quantum description is then the **density operator**

$$\rho = \sum_{n=1}^{N} c_n \, |\psi_n\rangle\langle\psi_n| \,, \qquad c_n \geq 0, \; \sum_n c_n = 1, \tag{7}$$

a **Hermitian**, positive-semidefinite matrix with unit trace. **Pure states**[3] appear as projectors, $\rho^2 = \rho$, while genuine mixtures satisfy $\rho^2 \neq \rho$.

Armed with kets, bras, the completeness relation and the density operator, we now possess a flexible vector-and-matrix toolkit that will underlie every formulation in the remainder of this book; from simple quantum computing algorithms to the construction and analysis of quantum neural network architectures.

4.3 Observables

In quantum theory a **physical observable**[4] is represented by a *Hermitian* operator $\hat{H} = \hat{H}^{\dagger}$ on the system's Hilbert space $\mathcal{H}$[8]. Hermiticity guarantees two things we demand of measurements:

(i) all eigenvalues are real numbers, so they can be recorded by an instrument, and

(ii) eigenvectors belonging to distinct eigenvalues are orthogonal, so they can be distinguished with certainty.

Thanks to the **spectral theorem** of functional analysis, any Hermitian observable admits the unique expansion

$$\hat{H} = \sum_i o_n \, \hat{P}_n, \qquad \hat{P}_n \equiv |e_n\rangle\langle e_n| \,, \qquad \sum_i \hat{P}_n = \mathbb{I},$$

[3] **Pure states** project onto a single *eigenstate* of ρ with eigenvalue 1, while **mixed states** decompose into *multiple orthogonal eigenstates*.

[4] **Measurable quantities** e.g. position $\hat{x}$, momentum $\hat{p}$, energy $\hat{H}$, spin $\hat{S}_z$.

where $\{|e_n\rangle\}$ is an orthonormal eigenbasis and the $\hat{P}_n$ are mutually orthogonal **projectors**. The set $\{\hat{P}_n\}$ is sometimes called a *projective measurement* because it both enumerates the possible outcomes $\{o_n\}$ and prescribes how the state updates once an outcome is registered.

Given a quantum system prepared in a (possibly mixed) state ρ, the **expectation value** and the corresponding **variance** of the observable $\hat{H}$ are

$$\langle \hat{H} \rangle = \mathrm{tr}(\rho\, \hat{H}), \qquad (\Delta O)^2 = \mathrm{tr}\big[\rho\, (\hat{H} - \langle \hat{H} \rangle)^2\big].$$

When two observables do not commute, $[\hat{A}, \hat{B}] \neq 0$, their variances obey the Robertson–Schrödinger uncertainty relation

$$\Delta A\, \Delta B \;\geq\; \frac{1}{2} \big| \langle [\hat{A}, \hat{B}] \rangle \big|,$$

a compact way to state, for instance, the familiar $\Delta x\, \Delta p \geq \hbar/2$, known as the **Heisenberg uncertainty principle**, where x denotes position and p momentum (not to be confused with the probability p used elsewhere).

4.4 Unitaries and time evolution

Exponentiating any Hermitian operator yields a **unitary** $\hat{U}(\lambda) = e^{\,i\lambda \hat{H}}$ with $\hat{U}^\dagger \hat{U} = \mathbb{I}$. Choosing $\lambda = -t/\hbar$ turns this into the time-evolution operator introduced in (5):

$$\hat{U}(t) = e^{-\,i\hat{H}t/\hbar}, \qquad |\Psi(t)\rangle = \hat{U}(t)\, |\Psi(0)\rangle\,, \qquad \rho(t) = \hat{U}(t) \rho(0) \hat{U}^\dagger(t).$$

Because unitaries preserve inner products, they **preserve *all* probabilities**. For later use in quantum computing it is helpful to remember that a finite product of elementary unitaries (generated, say, by local Hamiltonians) can approximate any desired global unitary to arbitrary precision.

The presentation so far has been cast entirely in the *Schrödinger picture*, where states carry the time dependence. Mathematically one may shift the time evolution onto the operators instead, defining the *Heisenberg picture* via

$$\hat{H}_H(t) = \hat{U}^\dagger(t) \hat{H}\, \hat{U}(t)$$

while ρ is kept fixed. Both pictures are equivalent; choosing one or the other is a matter of convenience.

4.5 Projective measurements

For the ideal ("**von Neumann**") measurement associated with the projector set $\{\hat{P}_n\}$ the probability of recording the specific outcome o_n is

$$p_n = \mathrm{tr}(\rho\, \hat{P}_n).$$

If that outcome is obtained the post-measurement state is

$$\rho \;\longrightarrow\; \frac{P_n \rho P_n}{p_m}.$$

A concrete illustration is the spin-$\frac{1}{2}$ example of Sec. 2.3. Measuring the **Pauli operator** σ_z corresponds to the pair of projectors

$$\hat{P}_\uparrow = |\uparrow\rangle\langle\uparrow|\,, \qquad \hat{P}_\downarrow = |\downarrow\rangle\langle\downarrow|\,,$$

and the corresponding outcome probabilities for a state $\rho = |\Psi\rangle\langle\Psi|$ are

$$p_\uparrow = |\psi_\uparrow|^2, \qquad p_\downarrow = |\psi_\downarrow|^2,$$

where $\psi_\uparrow = \langle\uparrow|\psi\rangle$ and $\psi_\downarrow = \langle\downarrow|\psi\rangle$.

Real laboratories, however, rarely achieve perfectly sharp projective measurements. The most general repeatable measurement is described instead by a collection of positive operators $\{E_m\}$ obeying the completeness relation which together form a **positive-operator-valued measure (POVM)**. Each effect operator can be expressed in terms of a *Kraus operator* A_m,

$$E_m \;=\; A_m^\dagger A_m,$$

a factorization that conveniently captures loss, detector inefficiency and other noise sources. When the pre-measurement state is ρ, the probability of obtaining outcome m is

$$p_m \;=\; \mathrm{tr}\big(\rho\, E_m\big),$$

and, conditioned on that outcome, the state updates according to

$$\rho \;\longrightarrow\; \frac{A_m \rho A_m^\dagger}{p_m}.$$

Projective measurements are recovered as the special case in which every E_m is itself a projector and the corresponding Kraus operators satisfy $A_m = A_m^\dagger = E_m$.

4.6 Composite Systems

Most practical settings involve several subsystems. If subsystem Σ_1 is described by $\mathcal{H}_1$ and Σ_2 by $\mathcal{H}_2$, the joint system $\Sigma = \Sigma_1 + \Sigma_2$ inhabits the **tensor-product space**

$$\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2, \qquad \dim \mathcal{H} = \dim \mathcal{H}_1 \cdot \dim \mathcal{H}_2.$$

An orthonormal basis for the composite space is conveniently generated by taking all possible tensor products of the single-system basis kets, $\{\left|e_j^{(1)}\right\rangle \otimes \left|e_k^{(2)}\right\rangle\}_{j,k}$.

A pure state is called **separable** (or a product state) when it can be expressed as an explicit tensor product of individual subsystem states,

$$|\psi\rangle = \left|\psi^{(1)}\right\rangle \otimes \left|\psi^{(2)}\right\rangle,$$

and **entangled** when no such decomposition exists. The distinction extends to mixed states: if $\rho = \rho_1 \otimes \rho_2$ the subsystems are uncorrelated, whereas any departure from that form signals correlations; classical, quantum, or both.

Operators inherit this structure. Given $\hat{H}_1$ on $\mathcal{H}_1$ and $\hat{H}_2$ on $\mathcal{H}_2$, the composite operator is $\hat{H} = \hat{H}_1 \otimes \hat{H}_2$. When the global state factorizes, expectation values factorize as well,

$$\langle \hat{H} \rangle = \mathrm{tr}\big((\hat{H}_1 \otimes \hat{H}_2)\, \rho_1 \otimes \rho_2\big) = \mathrm{tr}(\hat{H}_1 \rho_1)\ \mathrm{tr}(\hat{H}_2 \rho_2).$$

Frequently, however, one wishes to speak only about Σ_1; tracing over Σ_2 gives the **reduced state**

$$\rho_1 = \mathrm{tr}_2(\rho),$$

so that $\langle \hat{H}_1 \otimes \mathbb{I}_2 \rangle = \mathrm{tr}(\hat{H}_1 \rho_1)$. This procedure, **partial trace**, is the quantum analogue of marginalizing a joint probability distribution.

A canonical example of a composite system, widely used in quantum computing, is a **register of** n **qubits**: n identical two-level systems whose joint Hilbert space is $(\mathbb{C}^2)^{\otimes n} \cong \mathbb{C}^{2^n}$. How qubits exploit this tensor structure will become clear in the parts that follow.

QUANTUM COMPUTING

Quantum computing transfers the core principles of quantum mechanics into a computational paradigm. In this chapter, we develop the model of **quantum circuits** as the primary framework for **quantum computers**, directly analogous to how **Boolean logic circuits** serve classical machines. We begin by representing single **qubit** states on the **Bloch sphere**, then introduce **single-qubit gates** for state rotations and **controlled gates** to generate **entanglement** across multiple qubits. After covering the mechanics of **measurement**, we survey leading **quantum hardware** platforms—ion-trap, superconducting, neutral-atom, and photonic architectures—and show how to deploy and execute quantum circuits on them using NVIDIA's CUDA-Q toolkit [9]. Throughout the chapter, illustrative code snippets are shown to ground abstract concepts in hands-on practice.

5 Installing CUDA-Quantum

Before diving into quantum circuit construction, we first need to prepare our development environment. CUDA-Q is NVIDIA's hybrid *C++/Python framework* for crafting and executing **hybrid quantum–classical algorithms** via a concise, high-level API. In this book we adopt CUDA-Q because its **CUDA-powered simulators** let us tune precision, method, and scale on the fly, exploit NVIDIA GPUs for speed, and send the exact same code to a broad range of **cloud quantum hardware devices** (using **Amazon Braket**).

Step 1 – System Requirements. Because CUDA-Q binds to low-level libraries and (optionally) leverages GPU acceleration, it's important to start on a supported platform:

- **Linux**: Ubuntu 20.04 or 22.04 LTS (Debian-based distributions are best tested) or RHEL/CentOS 8 or 9. Ensure you have `build-essential`, `python3-dev`, and `git` installed.
- **macOS**: Version 12 Monterey or later, on either Apple Silicon or Intel hardware. On Apple Silicon, confirm Rosetta 2 is installed for x86 compatibility if needed.
- **Windows 11**: Use Windows Subsystem for Linux 2 (WSL2) with an Ubuntu 20.04 or 22.04 image. Install the `wsl` feature and set WSL2 as default: `wsl -install`.

A modern NVIDIA GPU (Ampère-class or newer) is *recommended* for high-performance simulation, but the CPU-only backend works perfectly well for early development and testing.

Step 2 – Create a Python Virtual Environment. To avoid conflicts between project dependencies and lock in compatible versions, we'll use a virtual environment.

```
python3 -m venv qenv
source qenv/bin/activate
```

Once activated, your prompt will prepend `(qenv)`, indicating that all subsequent Python and `pip` commands will target this isolated environment.

Step 3 – Install CUDA Quantum. With `qenv` active, upgrade `pip` and install the SeroviICAI CUDA-Quantum fork:

```
python -m pip install --upgrade pip
pip install cudaq
git clone https://github.com/SeroviICAI/cuda-quantum.git
python - <<'PY'
import importlib.util, pathlib, os, shutil
src = pathlib.Path("cuda-quantum/python/cudaq/kernel/
    quake_value.py").resolve()
dst = pathlib.Path(importlib.util.find_spec("cudaq.kernel.
    quake_value").origin)
dst.unlink()
os.symlink(src, dst)
PY
pip install torch-qu
```

This ensures you have both the standard CUDA-Q functionality and the extra hooks needed by the upcoming quantum neural network library.

Step 4 – Verify Installation. Let's run a minimal quantum program example:

```
import cudaq, math
kernel = cudaq.make_kernel()
q = kernel.qalloc(1)
kernel.rx(math.pi/4, q)
print(cudaq.sample(kernel, shots=64))
```

Run it with:

```
python kernel.py
```

You should see output similar to: `{ 0:53, 1:11 }`. The exact counts will vary each run, but the printed dictionary confirms that CUDA-Q is correctly installed and sampling from your quantum circuit is working as intended.

6 Quantum Circuits

Classical digital circuits process information via **bits** (0 or 1) and **Boolean logic gates** (AND, OR, NOT, etc.), typically realized by irreversible operations on voltage levels. In contrast, quantum circuits manipulate **qubits** using **quantum gates**. In analogy to classical gates building boolean circuits, quantum gates are the elementary building blocks of **quantum circuits**. However, unlike a classical AND gate which maps two bits to one (losing information), all quantum gates preserve information: mathematically they are described by **unitary matrices** (4.4) and hence are invertible.

Quantum circuits are often drawn with **wires** (horizontal lines) for each qubit and gates as symbols on those lines.

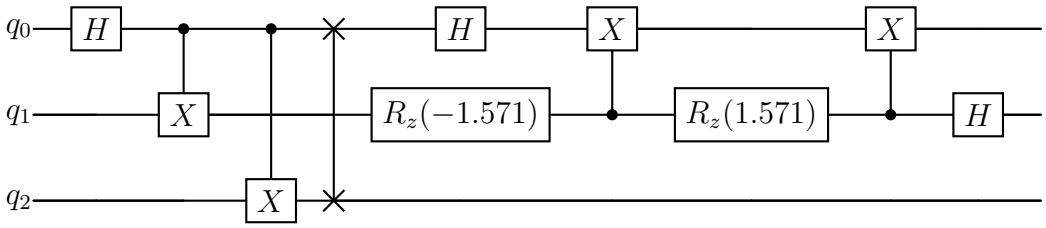


Figure 4: An example quantum circuit

Furthermore, qubits can exist in coherent **superpositions** of $|0\rangle$ and $|1\rangle$, and multiple qubits can be **entangled**, giving an exponentially larger state space than classical bits of the same size. This allows quantum circuits to exploit interference of amplitude phases, a feature absent in classical circuits.

To make the discussion concrete we will adopt *CUDA-Q* as our working SDK[5]. A CUDA-Q quantum circuit is called a ***kernel***; created using `cudaq.make_kernel`.

```
import cudaq, math

kernel = cudaq.make_kernel()
qvec = kernel.qalloc(3)

kernel.h(qvec[0])
kernel.cx(qvec[0], qvec[1])
kernel.cx(qvec[0], qvec[2])
```

[5]**SDK** stands for **Software Development Kit**. It's a collection of tools, libraries, documentation, code samples, and sometimes APIs that developers use to build applications for a specific platform or framework.

```
kernel.swap(qvec[0], qvec[2])
kernel.h(qvec[0])
kernel.rz(-math.pi/2, qvec[1])
kernel.cx(qvec[1], qvec[0])
kernel.rz(math.pi/2, qvec[1])
kernel.cx(qvec[1], qvec[0])
kernel.h(qvec[1])
```

Listing 1: A CUDA-Q kernel that constructs the quantum circuit illustrated in Figure 4

6.1 The Qubit

A *qubit* is a quantum system with two basis states, conventionally denoted $|0\rangle$ and $|1\rangle$. A general pure state of a qubit is a normalized linear combination

$$|\psi\rangle = \alpha\ |0\rangle + \beta\ |1\rangle\,,$$

where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$. In the computational basis,

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

and $|\alpha|^2$ (resp. $|\beta|^2$) is the probability of measuring $|0\rangle$ (resp. $|1\rangle$).

To build geometric intuition, we **reparametrize** the two complex amplitudes by three real parameters (θ, φ) plus an unobservable global phase. First, enforce normalization by writing

$$|\alpha| = \cos\!\left(\tfrac{\theta}{2}\right), \quad |\beta| = \sin\!\left(\tfrac{\theta}{2}\right), \quad 0 \le \theta \le \pi.$$

so that $\cos^2(\frac{\theta}{2}) + \sin^2(\frac{\theta}{2}) = 1$. Next, note that any ***overall*** (global) phase $e^{i\gamma}$ multiplying the entire state leaves physical predictions unchanged, so we absorb it into the definition of $|\Psi\rangle$. We are therefore left with a single ***relative*** phase φ between α and β:

$$\alpha = \cos\!\left(\tfrac{\theta}{2}\right), \quad \beta = e^{i\varphi} \sin\!\left(\tfrac{\theta}{2}\right), \quad 0 \le \varphi < 2\pi.$$

Substituting back into the superposition gives the **Bloch-sphere** parametrization:

$$|\psi(\theta, \varphi)\rangle = \cos\!\left(\tfrac{\theta}{2}\right)|0\rangle + e^{i\varphi} \sin\!\left(\tfrac{\theta}{2}\right)|1\rangle, \quad (0 \le \theta \le \pi,\ 0 \le \varphi < 2\pi).$$

This form may at first glance appear more elaborate than the simple linear combination of $|0\rangle$ and $|1\rangle$, it offers a powerful geometric intuition: the two angles (θ, φ) map the state directly onto the surface of a unit sphere.

6.2 Bloch Sphere

The **Bloch sphere representation** (Figure 5) turns complex amplitudes α and β into concrete points in $\mathbb{R}^3$, making many single-qubit operations and visualizations far more transparent. Every point on the sphere's surface represents a distinct pure state, whereas interior points (not shown) represent mixed states.

By convention, the north pole of the Bloch sphere corresponds to $|0\rangle$ and the south pole to $|1\rangle$.

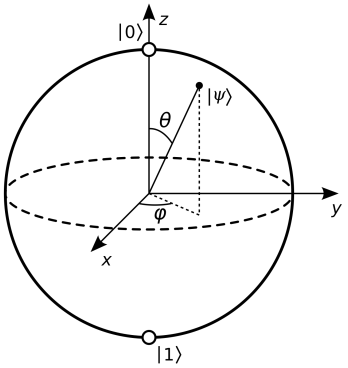


Figure 5: Bloch sphere

The mapping between the spherical angles (θ, φ) and the Bloch-sphere vector can be written in explicit **Cartesian form**, directly yielding the coordinates of the unit vector in $\mathbb{R}^3$:

$$(x, y, z) = \big(\sin\theta\cos\varphi,\ \sin\theta\sin\varphi,\ \cos\theta\big).$$

It's important to note the Bloch sphere is an **intuition tool** for one qubit only. There is no straightforward way to extend it to multi-qubit states, especially once entanglement comes into play. For one qubit, however, it gives a complete, visually appealing representation of the state, as every **single-qubit unitary operation** corresponds to rotations of this sphere-providing an intuitive picture for how gates transform qubit states.

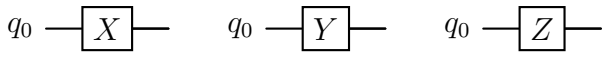
6.3 Single-Qubit Gates

As noted earlier, a quantum circuit transforms the joint state of qubits via **unitary gates** culminating in measurements that yield classical outcomes (0s and 1s). We classify quantum gates into single-qubit gates and controlled gates. **Single-qubit gates operate** on a single qubit at a time. Mathematically, any single-qubit gate can be represented by a 2×2 unitary matrix acting on the qubit's two-dimensional state vector [10]. Important examples include the Pauli and the Hadamard gates.

Pauli Gates. The Pauli gates are the simplest non-trivial single-qubit operations and form the basis for all single-qubit unitaries. They are defined by the following matrices, which correspond to π rotations about the Bloch-sphere axes:

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

In circuit diagrams, they are depicted as:



Each has a clear geometric and computational interpretation:

- **Bit-flip (X):** A rotation by π around the x–axis. In the computational basis it swaps $|0\rangle \mapsto |1\rangle$ and $|1\rangle \mapsto |0\rangle$, acting exactly like the classical NOT gate. On the Bloch sphere, it takes the north pole to the south pole.
- **Phase-flip (Z):** A π rotation around the z–axis. It preserves the populations of $|0\rangle$ and $|1\rangle$ but imparts a minus sign on $|1\rangle$, i.e. $Z\,|1\rangle = -\,|1\rangle$. Geometrically, Z flips the phase of any superposition in the equatorial plane.
- **Combined flip (Y):** A π rotation around the y–axis. Equivalently, $Y = i\,X\,Z$ (up to a global phase), so it implements both a bit-flip and a phase-flip simultaneously. Concretely, $Y\,|0\rangle = i\,|1\rangle$ and $Y\,|1\rangle = -\,i\,|0\rangle$.

These gates satisfy the Pauli algebra

$$X^2 = Y^2 = Z^2 = I, \qquad XY = i\,Z, \;\; YZ = i\,X, \;\; ZX = i\,Y,$$

and anticommute pairwise ($XY = -\,YX$, etc.), making them a representation of the Pauli algebra.

In the CUDA-Q API, you apply them as follows:

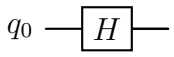
```
kernel = cudaq.make_kernel()
qubit = kernel.qalloc(1)  # allocate one qubit
kernel.x(qubit)  # apply Pauli-X (bit flip)
kernel.y(qubit)  # apply Pauli-Y (bit + phase flip)
kernel.z(qubit)  # apply Pauli-Z (phase flip)
```

Hadamard (H) Gate. One of the most important single-qubit gates is the Hadamard gate, often denoted H. The Hadamard gate places a qubit into an equal superposition of $|0\rangle$ and $|1\rangle$ (up to relative phase), making it an essential tool for creating superposition states. In matrix form,

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

which transforms the computational basis as $H|0\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $H|1\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$. This means that if a qubit is initially in state $|0\rangle$, applying H will yield the superposition $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ (often written as $|+\rangle$), and similarly $|1\rangle$ is mapped to $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ (the $|-\rangle$ state). The Hadamard thus creates a balanced superposition of basis states, a critical step in many quantum algorithms.

In circuit diagrams, the Hadamard gate is usually depicted as:

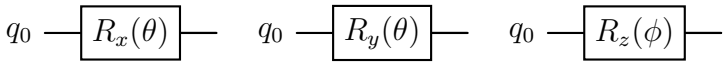


Below is a code snippet illustrating how to apply an H gate using CUDA-Q:

```
kernel = cudaq.make_kernel()
qubit = kernel.qalloc(1)  # allocate one qubit
kernel.h(qubit)  # apply Hadamard gate on the qubit
```

Rotation Gates (R_x, R_y, R_z). More generally, any single-qubit unitary can be expressed as rotations about the Bloch-sphere axes. They are denoted $R_x(\theta)$, $R_y(\theta)$, and $R_z(\phi)$ for rotations about the x-, y-, and z-axes by an angle θ, respectively. These gates allow continuous transformations of qubit states and are parameterized by the rotation angle. In general, one can express these rotations as exponentials of Pauli matrices:

$$R_x(\theta) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix} \quad R_z(\phi) = \begin{bmatrix} e^{-i\phi/2} & 0 \\ 0 & e^{i\phi/2} \end{bmatrix}$$



In code, rotation gates are invoked with a specified angle. For instance:

```
kernel = cudaq.make_kernel()
qubit = kernel.qalloc(1)  # allocate one qubit
theta, phi, lam = 1.5708, 0.7854, 2.094  # example angles
kernel.rx(theta, qubit)  # rotate around X-axis by theta
kernel.ry(phi, qubit)  # rotate around Y-axis by phi
kernel.rz(lam, qubit)  # rotate around Z-axis by lam
```

In summary, single-qubit gates allow us to prepare arbitrary superpositions and apply phase rotations to individual qubits. In practice, however, quantum algorithms (and specially quantum neural networks) operate on multi-qubit systems, and single-qubit operations alone cannot generate the entanglement that lies at the heart of their power. Instead, we make use of **controlled (multi-qubit) gates** to weave qubits together. Before diving into those entangling operations, we'll first pause to introduce the notation and structure of **multi-qubit states**.

6.4 Multi-Qubit States

A system of n-qubits lives in a 2^n-dimensional Hilbert space. The joint state of multiple qubits is given by the tensor (Kronecker) product of individual states. For two qubits, one can write the state as

$$|\psi\rangle = v_{00}\,|00\rangle + v_{01}\,|01\rangle + v_{10}\,|10\rangle + v_{11}\,|11\rangle\,,$$

whose vector representation is $\left[v_{00}, v_{01}, v_{10}, v_{11}\right]^T$.

In general the computational basis for n qubits consists of $|b_{n-1}\ldots b_0\rangle$ with each $b_i \in \{0,1\}$. Remember a multi-qubit state is **separable** (product) if it can be written as the tensor product of single-qubit states (e.g. $|\psi\rangle \otimes |\phi\rangle$), and **entangled** otherwise.

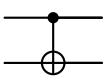
6.5 Controlled Gates

Controlled gates act on multiple qubits by applying a unitary to target qubits only if control qubits are in state $|1\rangle$. In general, a controlled-U (denoted $C(U)$) acting on two qubits has the block-diagonal:

$$C(U) \;=\; \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix},$$

where the upper-left block (action on $|0\rangle_{\text{ctrl}}$) is identity and the lower-right block (action on $|1\rangle_{\text{ctrl}}$) is U. Two commonly used controlled gates are the CNOT and CZ gates.

Controlled-NOT (CNOT) Gate. The CNOT gate (also called the **controlled-X gate**) flips the state of the target qubit if and only if the control qubit is $|1\rangle$. In the computational basis $\{|00\rangle\,, |01\rangle\,, |10\rangle\,, |11\rangle\}$ its matrix and circuit are:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{array}{l} q_0 \\ q_1 \end{array} \quad \text{  }$$

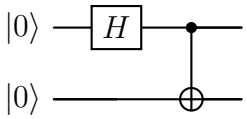
Under its action, the CNOT gate effects the following transformations:

$$|00\rangle \to |00\rangle\,, \quad |01\rangle \to |01\rangle\,, \quad |10\rangle \to |11\rangle\,, \quad |11\rangle \to |10\rangle\,.$$

If the control is in $|+\rangle$ and the target $|0\rangle$, then one of the **Bell states** is produced.

Example: Bell States

Two qubits both initialized to $|0\rangle$ form the product state $|00\rangle$. The following circuit (a Hadamard on the first qubit followed by a CNOT) produces a **maximally entangled Bell state**:



After this circuit, the qubits are in the state $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$, which cannot be factored into a tensor product of single-qubit states. Besides the "standard bell" state, the complete set of four orthonormal Bell states is:

$$|\Psi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}, |\Psi^-\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}, |\Phi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}, |\Phi^-\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}.$$

Such entangled states lie at the heart of quantum algorithms.

In **code**, one implements the CNOT gate via `cx(control,target)`.

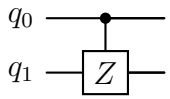
```
kernel = cudaq.make_kernel()
qvec = kernel.qalloc(2)  # allocate two qubits
kernel.cx(qvec[0], qvec[1])
```

Controlled-Z (CZ) Gate. The controlled-Z gate applies a Pauli-Z phase flip to the target iff the control is $|1\rangle$. Its matrix is diagonal:

$$\mathrm{CZ} = \mathrm{diag}(1,1,1,-1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

adding a (-1) phase to $|11\rangle$. When applied to $\frac{(|0\rangle+|1\rangle)^{\otimes 2}}{2}$, it produces an entangled phase state.

A circuit for CZ (control on q_0, target on q_1) is:



In CUDA-Q, one can implement CZ by `cz(control,target)`.

```
kernel = cudaq.make_kernel()
qvec = kernel.qalloc(2)  # allocate two qubits
kernel.cz(qvec[0], qvec[1])
```

Controlled gates are central to building complex multi-qubit operations. By combining single-qubit rotations and multi-qubit controlled gates like CNOT and CZ, we can construct arbitrary quantum circuits. In the context of quantum neural networks, these gates enable **entangling layers** and two-qubit interactions that are essential for the expressive power of the model by enabling **feature mixing** across qubits.

Before diving into more advanced topics, completing the exercises in the **TQADL Exercise Sheet** is recommended. These exercises reinforce the fundamentals of circuit construction and will prove useful when designing and testing your own quantum neural networks.

6.6 Parametrized Quantum Circuits

Most near-term quantum algorithms, including **quantum neural networks** (which we will meet in Part IV), rely on **parametrized quantum circuits** (PQCs). A PQC is a unitary[6]

$$U(\boldsymbol{\theta}) \;=\; U_M(\theta_M)\,U_{M-1}(\theta_{M-1})\cdots U_1(\theta_1),$$

where each gate $U_m(\theta_m)$ is typically a **parametrized rotation** (6.3) whose angle θ_m is a *trainable* real number and $M = |\boldsymbol{\theta}|$ is the total parameter count.

In CUDA-Q a circuit becomes parametrized as soon as you declare one or multiple arguments in the kernel's signature `cudaq.make_kernel(*args)`. The following code snippet illustrates this idea for three qubits.

[6] A quantum circuit, parametrized (e.g. QNNs) or not, acts on the register as a ***single global unitary***. The reason is elementary: each gate is unitary and the set of unitaries is closed under multiplication. Indeed, if U and V are unitary ($U^\dagger U = VV^\dagger = \mathbb{I}$), then $(UV)^\dagger(UV) = V^\dagger U^\dagger UV = V^\dagger V = \mathbb{I}$; hence the **product** UV is itself unitary. This mirrors the classical picture in which a multi-layer feed-forward neural network can be treated mathematically as a single function.

```python
kernel, thetas = cudaq.make_kernel(list[float])
qvec = kernel.qalloc(3)

# Rotation Block
for i in range(3):
    kernel.ry(thetas[i], qvec[i])

# Entanglement Block
kernel.cx(qvec[0], qvec[1])
kernel.cx(qvec[1], qvec[2])
kernel.cx(qvec[2], qvec[0])
```

Listing 2: Three-qubit parametrised kernel $U(\boldsymbol{\theta})$

CUDA-Q also lets you define a parametrized kernel as a regular Python function decorated with `@cudaq.kernel`. The parameters are passed in as plain Python values, so they can flow straight into any ordinary function and native arithmetic without wrestling with symbolic placeholders[7] (unlike with `cudaq.make_kernel`). The main downside this method has is **reduced modularity**, which clashes with the object-oriented coding style common in libraries such as `PyTorch`. The kernel body cannot touch non-local variables (e.g. `self` attributes) and it is harder to embed sub-kernels.

```python
@cudaq.kernel
def kernel(thetas: list[float]):
    qvec = cudaq.qvector(3)  # allocate 3 qubits

    for i in range(3):
        ry(thetas[i], qvec[i])

    x.ctrl(qvec[0], qvec[1])
    x.ctrl(qvec[1], qvec[2])
    x.ctrl(qvec[2], qvec[0])
```

Listing 3: Alternative definition for the same three-qubit kernel of Listing 2

Throughout the book we will adopt the `cudaq.make_kernel` style, as its class-friendly scope rules outweigh the extra boilerplate when building large, modular QNNs.

[7] CUDA-Q lowers kernel bodies to the MLIR **Quake** dialect, where every argument/parameter or operation result is a **QuakeValue**. A `QuakeValue` is a *symbolic placeholder*; an SSA (*Static Single Assignment*) handle that represents one definition in the compiler's *intermediate representation* (IR). It can stand for scalars (`int`, `float`), containers (`list`, `qvector`), or quantum objects (`qubit`) and overloads basic arithmetic and indexing so the compiler can trace data-flow for optimization.

6.7 Executing Quantum Circuits

Up to this point we have treated circuits as *abstract sequences of unitary gates* that map an initial product state $|0\rangle^{\otimes n}$ into a final many-qubit wave function $|\Psi_{\text{out}}\rangle$. The moment we want *numbers* (e.g. probabilities, expectation values, gradients) we must invoke the **measurement and observable postulates** introduced in Section 4.

CUDA-Q exposes **three complementary entry-points** that correspond almost one-to-one with the three most common laboratory tasks:

- `sample`: performs **projective measurements** in the computational (Z) basis for a user-specified number of shots, returning the raw bit-string histogram. This is the routine we will rely on throughout the Quantum Neural Network chapter.
- `observe`: evaluates the **expectation value** $\langle \hat{H} \rangle = \mathrm{tr}(\rho\, \hat{H})$ of a Hermitian operator $\hat{H}$ by automatically rotating to appropriate Pauli bases and re-assembling the weighted sum.
- `get_state`: returns the **full state vector** (simulation backend only).

1. Sampling: projective read-out statistics The `sample` routine executes the *textbook projective measurement* of Sec. 4.5. For *each shot* CUDA-Q

(i) projects $|\psi_{\text{out}}\rangle$ onto the computational (Z) basis,

(ii) collapses the *entire* n-qubit register to a single basis state $|b_{n-1} \dots b_0\rangle$,

(iii) returns the corresponding classical bit-string $b_{n-1} \dots b_0$.

By repeating this process for a user-defined number of shots, `sample` builds up a histogram of outcomes whose relative frequencies converge to the circuit's Born probabilities.

Using the same parametrized circuit from Listing 2:

```python
import cudaq
from math import pi

# Sample with thetas = [pi, pi/2, pi/3]
counts = cudaq.sample(
    kernel, [pi, pi/2, pi/3], shots_count=1024
)
print(counts)
```

Expected output (device- and noise-dependent):

```
{ 001:111 011:389 100:386 110:138 }
```

Because a simulator retains the full wave function in memory, it can draw all 1000 bit-strings from *one* state-vector. A real device, in contrast, must *re-prepare* its state for *every* shot; a direct, experimental manifestation of wave-function collapse.

2. Observing: expectation values of Hamiltonians Many quantum algorithms hinge on the **average energy** $\langle \hat{H} \rangle$ of some Hermitian operator $\hat{H}$ (Sec. 4.3). CUDA-Q's `observe` call internally

(i) decomposes $\hat{H}$ into tensor products of Pauli operators,

(ii) measures each term in the appropriate rotated basis, and

(iii) re-assembles the weighted sum.

By default `observe` performs *analytical* expectation-value evaluation on a simulator (one circuit run suffices), but you can switch to a *shot-based* estimate (identical in spirit to `sample`) by providing a `shots_count` parameter. This is the setting one would use on real hardware or when wanting to benchmark finite-sampling noise in a simulation. Consider the following two-qubit GHZ-state kernel:[8]

```python
from cudaq import spin  # helper module for Pauli algebra

@cudaq.kernel
def ghz_kernel(num_qubits: int):
    qvec = cudaq.qvector(num_qubits)
    h(qvec[0])
    for i in range(1, num_qubits):
        x.ctrl(qvec[0], qvec[i])

# Pauli Hamiltonian H = Z_0 + Y_1 + X_0 + Z_1
H = spin.z(0) + spin.y(1) + spin.x(0) * spin.z(1)

# Observe with num_qubits = 2
expect = cudaq.observe(ghz_kernel, H, 2, shots_count = 1024).
    expectation()
print(rf"<H> = {expect:.3f}")
```

For the ideal two-qubit GHZ state $\big(|00\rangle + |11\rangle\big)/\sqrt{2}$ this Hamiltonian yields $\langle \hat{H} \rangle = 0$, in perfect agreement with the *ab-initio* calculation $\mathrm{tr}(\rho\, \hat{H}) = 0$.

[8] The **Greenberger–Horne–Zeilinger (GHZ) state** $(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})/\sqrt{2}$ is a **maximally entangled** n-qubit state. The maximally entangled Bell state is the simplest, 2-qubit, instance of a GHZ state

3. Accessing the full wave function (simulation only) Finally, `get_state` exposes the entire state-vector in *big-endian* ordering $|b_{n-1}\dots b_0\rangle \longleftrightarrow 2^{n-1}b_{n-1}+\cdots+2^0 b_0$. This is a *diagnostic* tool, nothing in genuine quantum hardware can reveal global amplitudes without exponential effort; but it comes in handy for debugging and experimenting. (Chap. IV).

```python
import numpy as np

# Get state with num_qubits = 3
state = cudaq.get_state(ghz_kernel, 3)
print(np.array(state))
# [0.70710677+0.j 0.+0.j ... 0.+0.j 0.70710677+0.j]
```

The two non-zero components correspond to $|000\rangle$ and $|111\rangle$, exactly the branches predicted using the equations in Sec. 4.4.

Even though our **Quantum Neural Network** workflows rely exclusively on `sample` and learn directly from shot statistics, it is still worth keeping `observe` and `get_state` in view. In some of the recent literature QNN *outputs* are defined as **expectation values** of a read-out observable, and many simulation studies benchmark new ansätze by comparing the *full* state vector against a target. Knowing how these two API calls realise the measurement postulates and unitary dynamics reviewed in **Part II** will therefore prove useful whenever you compare our sampling-based approach with alternative formulations.

7 Quantum Processing Units

Thus far, we've described quantum computing in an abstract circuit model. In reality, executing quantum circuits requires specialized **quantum hardware (Quantum Processing Units or QPUs)**. There are several leading physical implementations of qubits, each with pros and cons. CUDA-Q is designed to be hardware-agnostic and connects to multiple types of QPUs via cloud providers. Here we overview four major quantum hardware paradigms and discuss how to run circuits on actual devices.

7.1 Ion-Trap QPUs

Trapped-ion computers confine a linear string of identical ions (Yb^+, Ba^+,...) in an electromagnetic potential. The qubit states sit in two long-lived electronic levels, manipulated by coherent laser pulses; shared vibrational modes in the chain mediate entangling operations, giving a naturally *all-to-all* connectivity. Because these ions are genuine atomic clones, **gate fidelities exceed 99.9 %** and **coherence times run into seconds**, making trapped ions the accuracy champions of today's hardware. The price is speed: single-qubit rotations last microseconds and two-qubit gates tens of microseconds, so wall-clock runtime becomes the limiting resource for deep circuits. Current commercial systems top out near one hundred ions in a single trap; larger processors are expected to network multiple traps with photonic links. Notable platforms include **IonQ's Aria** and **Quantinuum's H-series**. CUDA-Q connects to both through the same call:

```
# NOTE: Credentials must be set before running this program.
cudaq.set_target("ionq", qpu="qpu.aria-1")  # IonQ Aria ion trap
cudaq.set_target("quantinuum", machine="H1-2")  # Quantinuum H1-2
```

7.2 Superconducting QPUs

Superconducting processors pattern aluminium–on–silicon *transmons*, artificial atoms whose two lowest microwave levels form a qubit. Chips are cooled to a frigid $\sim$10 mK so that thermal noise is frozen out, and nanosecond-scale microwave pulses effect rotations, while dedicated couplers supply two-qubit gates on neighbouring sites. Here the virtues are **speed** (tens of nanoseconds per gate) and **foundry manufacturability**; the drawbacks are **modest coherence** (tens–hundreds of microseconds) and **planar-lattice topology** that forces extra SWAP operations. CUDA-Q already ships drivers for Anyon Technologies, IQM, and Oxford Quantum Circuits (OQC):

```
# NOTE: Credentials must be set before running this program.
cudaq.set_target("anyon", machine="telegraph-8q")  # 8-qubit
    ring
cudaq.set_target("iqm", url="https://<server>/cocos",
                 **{"qpu-architecture":"Adonis"})  # 20-qubit
     lattice
cudaq.set_target("oqc", machine="lucy")  # 8-qubit Lucy
```

7.3 Neutral-Atom QPUs

Neutral-atom machines trap individual Rb or Cs atoms in reconfigurable arrays of optical tweezers. A brief promotion to a highly excited *Rydberg state* switches on a strong dipole–dipole interaction, so pairs of tweezers act as controllable two-qubit gates. These platforms combine **programmable geometry** with sub-microsecond gate times and can already reach a few hundred qubits, albeit with coherence limited by the Rydberg lifetime. CUDA-Q speaks natively to Infleqtion's gate-based hardware, Pasqal's analog "Fresnel" device, and QuEra's 256-qubit *Aquila* via Amazon Braket:

```
# NOTE: Credentials must be set before running this program.
cudaq.set_target("infleqtion", machine="cq_sqale_qpu")  #
    digital Rydberg
cudaq.set_target("pasqal", machine="FRESNEL")  # analog array
cudaq.set_target("quera")  # QuEra Aquila
```

7.4 Photonic QPUs

In photonic processors the information carrier is the photon itself, encoded for instance in dual rails, polarization, or time bins. Photons travel essentially decoherence-free at room temperature, so the chief engineering challenges are **on-demand single-photon sources**, **low-loss interference**, and **high-efficiency detectors**. ORCA Computing's PT-series adopts a time-bin interferometer that naturally fits boson-sampling workloads and continuous-variable algorithms. A single environment variable is all CUDA-Q needs:

```
import os, cudaq
orca_url = os.getenv("ORCA_ACCESS_URL", "https://pt1.orca.com
    ")
cudaq.set_target("orca", url=orca_url)
```

With these four paradigms in hand you can select the fabric that best suits a given algorithm, all without changing a single line of quantum-kernel code.

7.5 Simulation Backends

At the time of writing, public quantum processors rarely exceed one hundred physical qubits and are still limited by coherence times and gate fidelity. For that reason a *serious* workflow in quantum-enhanced deep learning always begins on classical hardware, where kernels can be tested, tuned, and even fully trained long before the first token of cloud credit is spent on a QPU. **CUDA-Q** answers that practical need with a family of simulators that preserve one guiding promise: whatever you intend to run on hardware is *exactly* what you run in-silico. Whether the wave function lives in CPU memory, on a single GPU, is sharded across a cluster, or is hidden inside a tensor network, a single target string controls the migration and nothing else in your code has to change.

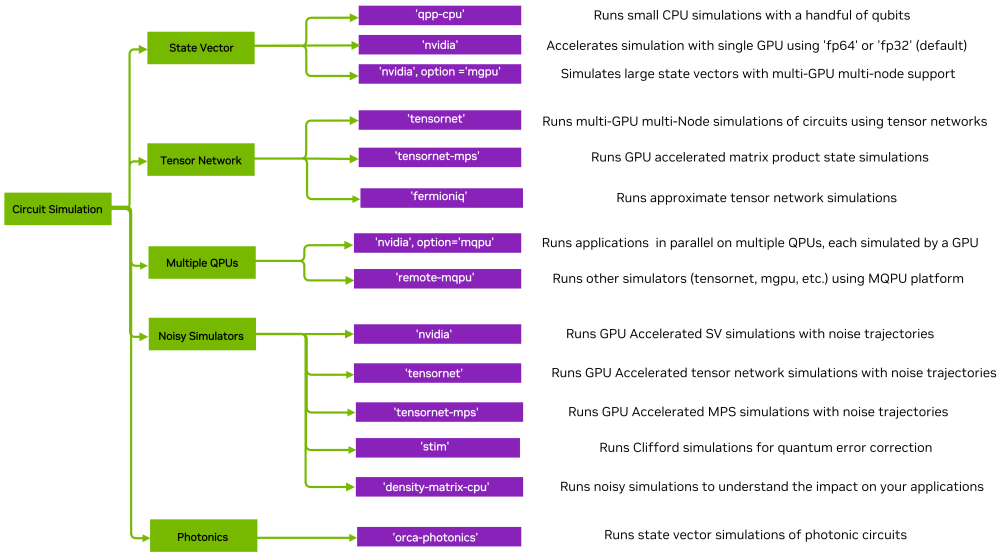


Figure 6: CUDA-Q's circuit simulation landscape. Purple labels are valid arguments to `cudaq.set_target`. Green branches are the paths most relevant to quantum neural networks.

We start with the **state vector engines** because they behave like a perfect Schrödinger laboratory: the global amplitude of every basis state is available and the only limitation is memory. On a laptop without CUDA the `qpp-cpu` target is the default; it is OpenMP[9] threaded, double precision, and remains comfortable until roughly twenty eight qubits.

[9] **OpenMP** (Open Multi-Processing) is a portable, directive-based API that lets C, C++, and Fortran programs exploit shared-memory parallelism on multicore CPUs.

```
cudaq.set_target("qpp-cpu")  # CPU, double precision
hist = cudaq.sample(kernel, shots_count=1024)
print(hist)
```

When a single NVIDIA GPU is present, the target named `nvidia` silently switches over to *cuStateVec*: the entire wave function is placed in device memory, gates are fused on the fly, and CUDA cores power through millions of amplitudes per clock tick. On a 64 GB H100 the single precision flavour fits thirty three qubits; choosing double precision (`fp64`) sacrifices a few qubits for numerical stability, something you may need if your loss landscape is particularly steep or your QNN uses large amplitude rotations.

```
cudaq.set_target("nvidia", option="fp32")  # fastest, ~33
    qubits on H100
loss = cudaq.observe(kernel, H).expectation()
```

Large data batches, hyperparameter grids, or simply bigger models eventually overrun the RAM of a single card. Appending the option `mgpu` distributes the state vector across all visible GPUs and, under MPI, across nodes as well.

```
cudaq.set_target("nvidia", option="mgpu,fp64")
# mpiexec -np 4 python3 train.py --target nvidia --target-
    option mgpu,fp64
```

Where state vector methods run out of steam, **tensor networks** take over. The target `tensornet` rewrites the entire circuit as an exact tensor graph and contracts it with *cuTensorNet*. Because contraction order is chosen automatically, wide but shallow circuits (some quantum neural network architectures) scale to thousands of qubits before memory becomes an issue.

```
cudaq.set_target("tensornet")  # fp64 default
hist = cudaq.sample(kernel, shots_count=1024)
```

If depth grows while entanglement stays local, an approximate matrix product state picture is often faster. Switching to `tensornet-mps` keeps only the largest singular values. Hundreds of qubits on a single GPU become routine.

```
import os
cudaq.set_target('tensornet-mps", option='fp32')
samples = cudaq.sample(kernel, shots_count=4096)
```

7.5 SIMULATION BACKENDS

Not every workload is a single, monolithic circuit. Optimizing a QNN often means evaluating dozens of parameter sets in parallel. Adding the **Multi-QPU simulation option**, `mqpu`, instantiates one virtual QPU per GPU and lets you launch asynchronous jobs with `sample_async` or `observe_async`. CUDA-Q hides the thread pool or MPI ranks; you simply collect futures when you need the numbers back.

```
cudaq.set_target("nvidia", option="mqpu")
futs = [
    cudaq.sample_async(qnn, p) for p in parameter_grid
]
grads = [f.get() for f in futs]  # histogram per param set
```

Real devices suffer noise, and so should our simulations. Any GPU backend-state vector or tensor network enters the Monte Carlo trajectory regime as soon as a **Noise model** is supplied. Each gate is wrapped in Kraus operators (4.5), trajectories are sampled shot-by-shot, and statistical error shrinks like $1/\sqrt{N_{\text{traj}}}$. The call signature is unchanged.

```
bit_flip = cudaq.BitFlipChannel(0.01)
noise = cudaq.NoiseModel()
noise.add_all_qubit_channel("x", bit_flip)
cudaq.set_target("tensornet")  # any GPU back-end works
exp = cudaq.observe(
    kernel, H, noise_model=noise, num_trajectories=4096
).expectation()
```

In practice you will jump between these backends oftenly: `qpp-cpu` for quick assertions, `nvidia fp32` for fast gradient loops, `mgpu` when the model grows, `tensornet` for thousand qubit experiments, and `mqpu` whenever a parameter grid appears. The kernels never notice the change, you merely point them at a new backend and execute them.

QUANTUM NEURAL NETWORKS

Artificial Intelligence seeks to empower computers to perform tasks that defy straightforward human explanation. These could be processes whose detailed mechanisms remain unknown or tasks we understand intuitively but struggle to define clearly at the low-level detail a computer requires. For instance, consider recognizing a dog in an image: humans effortlessly identify characteristic features—fur, tail, paws—that constitute "dogness". Yet, for a computer, an image is merely a numerical array of pixel values devoid of intrinsic meaning. No direct computational rule encodes concepts like fur or paws within these numbers, highlighting a fundamental gap between human intuition and machine interpretation.

Classical AI addresses this challenge through learning-based models, particularly **Artificial Neural Networks (ANNs)**. Rather than relying on explicit instructions, these models autonomously identify patterns by adjusting parameters based on extensive labeled datasets. Consequently, ANNs uncover subtle and distributed features in data that would be impractical or impossible to explicitly program.

Despite the *empirical success of deep learning*, classical methods are reaching **significant limitations** due to their rising computational and energy demands. Quantum computing, though still in its exploratory youth, offers a radically different computational paradigm by leveraging the principles of **quantum mechanics**. Unlike classical computing, quantum computing exploits these phenomena to process information in parallel and with potentially exponential speedups for certain types of problems. Specifically, **parametrized quantum circuits** (Sec. 6.6) present themselves as promising contenders to classical ANNs, potentially addressing current computational constraints.

8 Installing Torch-Quantum

Torch-Q is a quantum-native deep-learning library developed alongside this book. We'll use it in this chapter to build and test custom quantum neural networks within the familiar **PyTorch ecosystem**. Torch-Q provides drop-in `nn.Module` layers, autograd-compatible parameter-shift gradients and other research-friendly utilities [11].

Before installing Torch-Q, ensure you have already completed the steps in Section 5 to install **CUDA-Q** and the required **extra hooks**.

```
pip install torch-qu
```

You're now equipped to follow the examples in the following sections and integrate Torch-Q into your quantum machine-learning experiments and research workflows.

9 Architecture and Forward Pass

It has already been spoiled that a **quantum neural network (QNN)** is, in essence, just a parametrized quantum circuit (PQC); not a neural network in the classical sense [12]. We adopt the familiar terminology to help newcomers recognize the analogy, but in the literature you may also encounter names such as *variational quantum classifiers*. Regardless of name, the structure is the same.

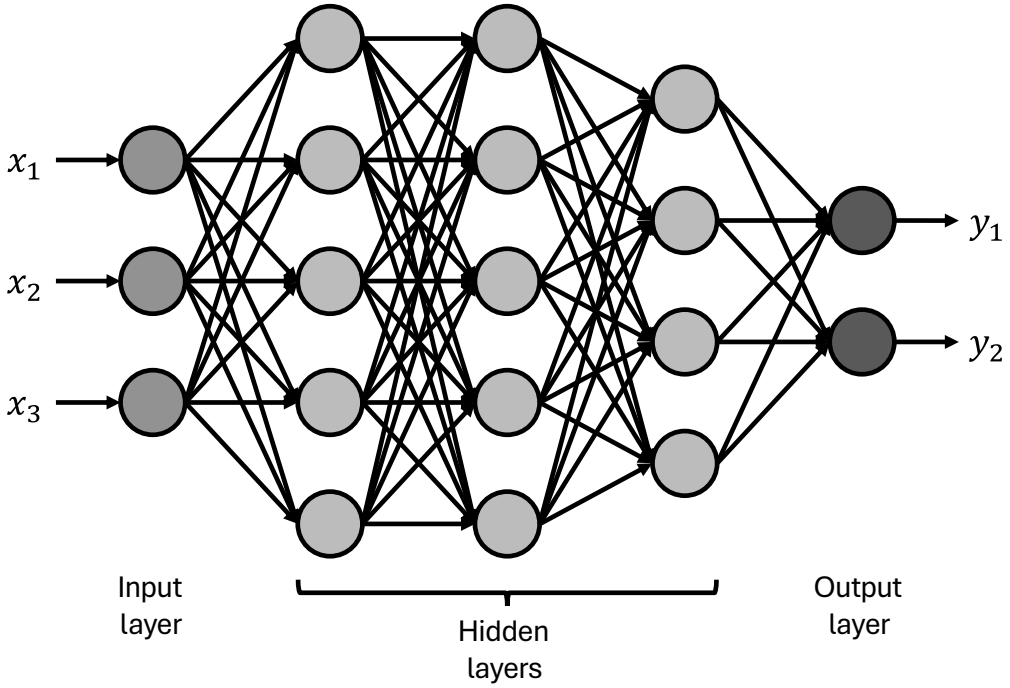


Figure 7: Classical Neural Network

Just as a classical neural network decomposes into an input layer, one or more hidden layers, and an output layer (Fig. 7), a QNN naturally splits into **three building blocks**:

- **Feature map.** Encodes the classical input data into a quantum state.
- **Variational form.** Applies a trainable sequence of gates, whose parameters will be optimized.
- **Output measurement.** Maps the final quantum state back to a set of classical values (probabilities, expectation values, bit-string counts).

In code, and following our modular design, we implement each of these blocks as a *subkernel* and then "stitch" them together. For example:

```
qnn, x, thetas = cudaq.make_kernel(List[float], List[float])
qvec = kernel.qalloc(in_features)

# Feature map subkernel, passing in input data x
kernel.apply_call(feature_map.kernel, qvec, x)

# Variational form subkernel, passing in trainable parameters
kernel.apply_call(var_form.kernel, qvec, thetas)
```

Listing 4: A Quantum Neural Network abstract kernel example

Here, `kernel.apply_call` invokes another kernel (our feature map or variational form) on the same quantum register `qvec`, forwarding the appropriate arguments. This approach keeps each block encapsulated, mirroring the **block structure** familiar from standard deep-learning tooling (e.g. PyTorch) and makes it easy to swap in new feature maps or variational forms without rewriting the QNN's outer loop.

The final piece, *output measurement*, is performed at execution time; once you invoke `cudaq.sample`. In the sections that follow we will unpack each of these components in depth: Feature maps, Variational forms and Output measurements.

9.1 Feature Maps

A classical feed–forward network accepts its input $\mathbf{x} \in \mathbb{R}^m$ by *directly* evaluating a trainable function $f_{\boldsymbol{\theta}} \colon \mathbf{x} \mapsto \mathbf{y}$. Quantum circuits do *not* enjoy this luxury: the register begins in a fixed reference state, usually $|0\rangle^{\otimes n}$, and *only unitary gates* may act on it. Consequently the classical data must first be *encoded* into a quantum state by a **feature-map unitary**

$$U_\Phi(x) \;:\; |0\rangle^{\otimes m} \;\longmapsto\; |\psi_x\rangle \;=\; U_\Phi(x)\,|0\rangle^{\otimes m},$$

whose image $\{|\psi_x\rangle\}_{\mathbf{x}\in\mathbb{R}^m} \subset \mathcal{H}$ plays the role of a high-dimensional *quantum feature space*. Only *after* this embedding layer does the variational circuit $\mathcal{G}_\theta$ act and learning proper begins (Fig. 8). Because each qubit can carry at most one real degree of freedom[10], it is customary to choose the number of qubits equal to the input dimension, m, so every entry x_i has its "own" qubit.

[10]The global phase is physically irrelevant.

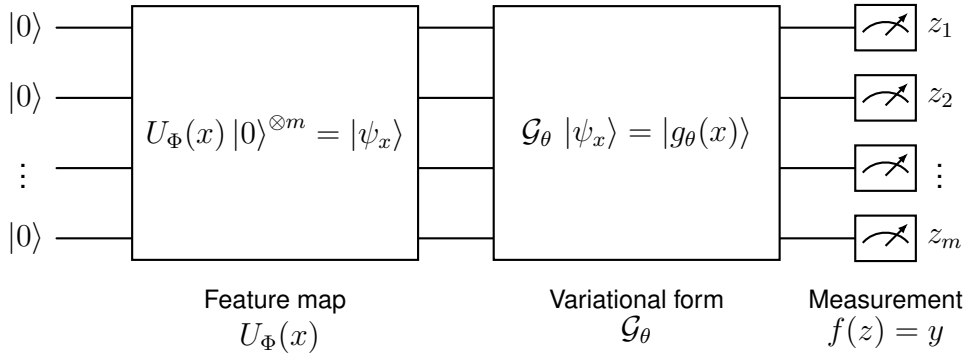


Figure 8: Quantum neural network viewed as a three-stage pipeline. A classical datum x is first embedded by the feature-map unitary $U_\Phi(x)$ (*left block*), producing the state $|\psi_x\rangle$. A trainable variational circuit $\mathcal{G}_{\boldsymbol{\theta}}$ then maps $|\psi_x\rangle$ to $|g_{\boldsymbol{\theta}}(x)\rangle$ (*centre block*). Finally, projective measurements in the computational basis yield a bit-string $z = (z_1, \ldots, z_m)$, which is fed into a classical post-processing function $f(z) = y$.

The simplest possible embedding is to write each real feature value into the phase of its qubit by a single Z-rotation,

$$U_Z(\mathbf{x}) \;=\; \bigotimes_{i=1}^{m} R_z\big(x_i\big), \qquad R_z(\varphi) \;=\; \exp\Big(-\tfrac{i}{2}\,\varphi Z\Big) \;=\; \cos\big(\tfrac{\varphi}{2}\big)\,I - i\sin\big(\tfrac{\varphi}{2}\big)\,Z.$$

On the computational vacuum, each qubit becomes $|0\rangle \mapsto \cos(\frac{x_i}{2})\ |0\rangle - i\sin(\frac{x_i}{2})\ |0\rangle = e^{-ix_i/2}\,|0\rangle$, so the global state acquires a phase $|\Phi_Z(\mathbf{x})\rangle = e^{-\frac{i}{2}\sum_i x_i}\,|0\rangle^{\otimes m}$. Although this phase is *global* and may appear trivial, its importance is revealed once the variational circuit entangles the qubits: interference converts the phase factors into measurable *bilinear* trigonometric functions of the inputs. Indeed, expanding each $R_z(x_i)$ in the Pauli basis,

$$R_z(x_i) \;=\; \tfrac{1}{2}\Big[\big(1+\cos x_i\big)I - i\sin x_i\, Z\Big],$$

and taking tensor products, one finds that the expectation value of any **Pauli word**[11] $P \in \{I, Z\}^{\otimes m}$ is a product of either $\cos x_i$ or $\sin x_i$ factors; hence a *bilinear form* in the features after the trigonometric identity $\sin x \cos y = \frac{1}{2}\big[\sin(x+y) + \sin(x-y)\big]$. This proves that the Z-rotation embedding already lifts the data into a non-linear feature space.

[11] A **Pauli word** is any tensor product $P = P_1 \otimes P_2 \otimes \cdots \otimes P_m$ of single-qubit Pauli operators $P_j \in \{I, X, Y, Z\}$. For the R_z-only feature map we need only the commuting subset $\{I, Z\}^{\otimes m}$, which is diagonal in the computational basis.

Within the **Torch-Q** framework, one can implement a minimal, reusable feature map layer that realises $U_Z(\mathbf{x})$ which may be expressed in code as follows:

```python
import cudaq, torchq.quantum as qu

class MyFeatureMapLayer(qu.Ansatz):
    def __init__(self, num_qubits: int) -> None:
        # Add code before super().__init__ call
        super().__init__(num_qubits)

    def build_kernel(self) -> cudaq.Kernel:
        kernel, qvec, x = cudaq.make_kernel(cudaq.qvector, list[float])
        for i in range(self.num_qubits):
            kernel.rz(x[i], qvec[i])  # <-- the encoding gate
        return kernel

    def get_parameter_count(self) -> int:
        return self.num_qubits
```

Stacking L such layers yields the full feature map

```python
import torchq.quantum.feature_maps as qfm

class MyFeatureMap(qfm.FeatureMap):
    def __init__(self, in_features: int, num_layers: int) -> None:
        # Add code before super().__init__ call
        super().__init__(in_features, num_layers)

    def build_kernel(self) -> cudaq.Kernel:
        kernel, qvec, x = cudaq.make_kernel(cudaq.qvector, list[float])
        for l in range(self.num_layers):
            layer = MyFeatureMapLayer(self.num_qubits)
            self.layers[f"layer_{l}"] = layer
            kernel.apply_call(layer.kernel, qvec, x)
        return kernel
```

This single-qubit R_z embedding is, in fact, the *simplest member* of the broader **Pauli (Havlíček) feature-map family** introduced in [13]. More expressive models enlarge the Pauli set $\mathcal{P}$ beyond the one-body Z operators to include (Pauli) words such as ZZ, XYZ, etc., thereby injecting higher-order products of the input features into the phase factors. Given a set $\mathcal{P} \subset \{X, Y, Z, I\}^{\otimes m}$ of Pauli strings that exclude the identity, define

$$U_{\mathcal{P}}(\mathbf{x}) \;=\; \prod_{P \in \mathcal{P}} \exp\!\Big(i\, \alpha_P \prod_{j \in \mathrm{supp}(P)} x_j \; P \Big),$$

where $\alpha_P \in \mathbb{R}$ is a tunable scale and $\mathrm{supp}(P)$ denotes the qubit indices on which P is *not* the identity.

Circuit-wise Eq. (9.1) is diagonal in the computational basis, and each exponential can be realised by

(i) single-qubit basis changes (H or $R_x(\pi/2)$ for X/Y);

(ii) a *CNOT ladder* that successively XORs the parity of those qubits onto the last one (this step **entangles** the specified qubits whenever $|\mathrm{supp}(P)| > 1$);

(iii) a single, data-dependent rotation $R_z\big(\alpha_P \prod_{j\in\mathrm{supp}(P)}(\pi - x_j)\big)$ on that last qubit;

(iv) the reverse CNOT ladder followed by the **inverse basis changes**, which disentangles the register and restores the original computational basis.

Because Eq. 9.1 multiplies *products* of features into the exponent, the kernel contains **polynomial terms** of arbitrarily high degree. In fact, Havlíček *et al.* showed that, for suitable $\mathcal{P}$, estimating K up to additive error is #P-hard, making the corresponding SVM kernel *classically intractable*. This is precisely the regime where a quantum computer can enjoy an **advantage**.

Torch-Q's quantum module exposes a `PauliFeatureMap` class whose constructor takes nothing more exotic than

```
PauliFeatureMap(in_features, num_layers, paulis, entanglement
    ="full", alpha=2.0)
```

where `paulis` is a *list of Pauli strings* such as $\{$`"Z"`, `"ZZ"`, `"XYZ"`$\}$. Under the hood the call chain `PauliFeatureMap` → `PauliFeatureMapLayer` → `PauliBlock` expands Eq. (9.1) into basis-change gates, CNOT ladders, and a single data-driven R_z per Pauli word, then stacks everything for as many layers as you asked for.

A quick way to *see* what any choice of $\mathcal{P}$ does is simply to draw the circuit:

```
import cudaq, torchq.quantum.feature_maps as qfm

m = 3
paulis = ["Z", "YY", "ZXZ"]  # try your own list!
fm = qfm.PauliFeatureMap(
    in_features=m,
    num_layers=1,
    paulis=paulis
)

```

```
kernel, x = cudaq.make_kernel(list[float])
qvec = kernel.qalloc(m)
kernel.apply_call(fm.kernel, qvec, x)
print(cudaq.draw(kernel, [0.1, 0.2, 0.3]))
```

Tweak the `paulis` list, run the code, and you will immediately recognise how higher-weight words increase entanglement depth and parameter coupling.

Because the two Pauli sets $\mathcal{P}_Z = \{Z_i\}_1^m,\ \mathcal{P}_{ZZ} = \{Z_1, \ldots, Z_d,\ Z_1Z_2, \ldots, Z_{m-1}Z_d\}$ appear in almost every QML paper, Torch-Q provides them as convenience wrappers:

```
Z_map = qfm.ZFeatureMap (in_features=2, num_layers=1)
ZZ_map = qfm.ZZFeatureMap(in_features=2, num_layers=1)
```

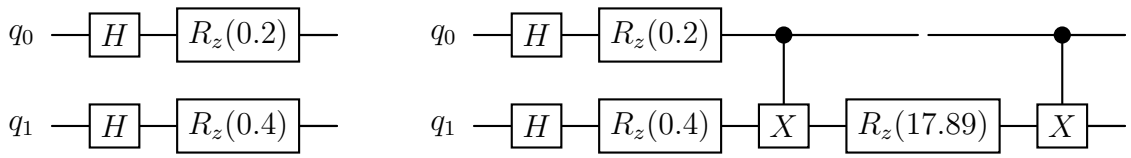


Figure 9: One-layer (a) Z and (b) ZZ feature maps.

Either can be dropped into a QNN exactly like the generic `PauliFeatureMap`, they are simply pre-configured instances with defaults `entanglement="full"` and $\alpha = 2$.

9.2 Variational Forms

In feed–forward networks the trainable weights appear *before* the non–linearity, so every neuron can mix the full input vector via the simple, but enormously powerful, linear combination

$$h_j^{(\ell)} = \sigma\Big(\sum_i w_{ji}^{(\ell)}\, h_i^{(\ell-1)} + b_j^{(\ell)}\Big).$$

By contrast, a quantum circuit evolves by *unitaries*, and the only mechanism able to **fuse information** carried by different qubits is **entanglement**. A variational form (sometimes called “ansatz”) therefore interleaves *local* parameterised rotations with an *entangling* pattern of multi–qubit gates, exactly mimicking the **linear to non-linear** alternation that drives classical deep learning.

Similar to the simple Pauli-Z feature map we studied in Sec. 9.1, the absolute simplest choice places a single $R_z(\theta_i)$ rotation on every qubit,

$$\mathcal{G}_{\boldsymbol{\theta}}^{\text{naive}} = \bigotimes_{i=1}^{n} R_z(\theta_i), \qquad R_z(\theta) = e^{-\frac{i}{2}\theta Z}.$$

While perfectly legitimate, this ansatz is *ill-posed*: because every qubit is acted on independently, no entangling mechanism exists to fuse information across qubits. Each wire merely stores a unary feature, the joint bit-string read-out factorises into independent marginals, and the model offers no way to capture correlations; reducing it to little more than a per-qubit lookup table.

The obvious cure is to insert two–qubit gates that *propagate* phase information across the register. The lightest variant is a "*linear chain*" (nearest–neighbour) of CNOTs:

$$\mathcal{G}_{\boldsymbol{\theta}}^{(1)} = \left[C_X^{(n-1,n)} \cdots C_X^{(2,3)} C_X^{(1,2)}\right] \qquad \bigotimes_{i=1}^{n} R_z(\theta_i)$$

where $C_X^{(i,i+1)}$ flips qubit $i{+}1$ conditioned on qubit i. The single layer above already suffices to couple *all* θ_i's via many–body Pauli words generated by conjugation, $C_X\, Z_i\, C_X = Z_i Z_{i+1}$.

The following code demonstrates how this lean variational layer is implemented within the Torch-Q framework:

```python
import cudaq, torchq.quantum as qu

class MyVariationalFormLayer(qu.Ansatz):
    def __init__(self, num_qubits: int) -> None:
        # Add code before super().__init__ call
        super().__init__(num_qubits)

    def build_kernel(self) -> cudaq.Kernel:
        kernel, qvec, thetas = cudaq.make_kernel(cudaq.
    qvector, list[float])
        for i in range(self.num_qubits):
            kernel.rz(thetas[i], qvec[i])
        for i in range(self.num_qubits - 1):
            kernel.cx(qvec[i], qvec[i + 1])
        return kernel

    def get_parameter_count(self) -> int:
        return self.num_qubits
```

Stacking L identical layers yields the full variational form:

```python
import torchq.quantum.variational_forms as qvf

class MyVariationalForm(qvf.VariationalForm):
    def __init__(self, num_qubits: int, num_layers: int) -> None:
        # Add code before super().__init__ call
        super().__init__(num_qubits, num_layers)

    def build_kernel(self) -> cudaq.Kernel:
        kernel, qvec, thetas = cudaq.make_kernel(cudaq.qvector, List[float])
        ptr = 0
        for layer in range(self.num_layers):
            layer = MyVariationalFormLayer(self.num_qubits)
            self.layers[f"layer_{l}"] = layer
            cnt = layer.get_parameter_count()
            kernel.apply_call(layer.kernel, qvec, thetas[ptr : ptr + cnt])
            ptr += cnt
        return kernel

    def get_parameter_count(self) -> int:
        return self.num_layers * self.num_qubits
```

The pattern *rotation* $\rightarrow$ *entanglement* is so ubiquitous in the QNN literature that it is commonly formalized under the umbrella term **NLocal** [14]. An NLocal ansatz repeats

$$\big[\text{rotation block}\big] \;\rightarrow\; \big[\text{entanglement block}\big]$$

for a user-chosen number of layers, optionally closing with an extra rotation stage. Rotation blocks accept an *ordered list* of single-qubit gates (e.g. `"ry"`, `"rz"`), while the entanglement block specifies *(a)* the multi–qubit gate (`"cx"`, `"cz"`, ...) and *(b)* the coupling graph: `"full"`, `"linear"`, `"reverse_linear"`, or an explicit list of pairs.

Three celebrated instances, widely used in QML benchmarks, fit seamlessly into this scheme.

- **RealAmplitudes**: Rotation$=\mathrm{RY}$, Entanglement$=\mathrm{CX}$ (all layers identical). Suitable for real-valued wave-functions and the go-to baseline in variational classifiers.
- **EfficientSU2**: Alternating RY–RZ local gates wrapped by CNOTs. Provides a universal gate set per layer and approximates arbitrary unitaries with $\mathcal{O}(nL)$ parameters.

- **PauliTwoDesign**: Randomly samples one Pauli rotation per qubit and entangles with a pair-patterned CZ network, approaching a unitary 2-design in depth $\sim \log n$ [15]. Excellent for "hardware-efficient" explorations.

Each of these classes is nothing more than a pre-configured `NLocal` with a specific choice of rotation list, entanglement gate, and connectivity graph:

```
ansatz1 = qvf.RealAmplitudes(num_qubits=4, num_layers=1,
    entanglement="full")
ansatz2 = qvf.EfficientSU2(num_qubits=4, num_layers=1,
    su2_gates=["ry", "rz"])
ansatz3 = qvf.PauliTwoDesign(num_qubits=4, num_layers=1, seed
    =2025)
```

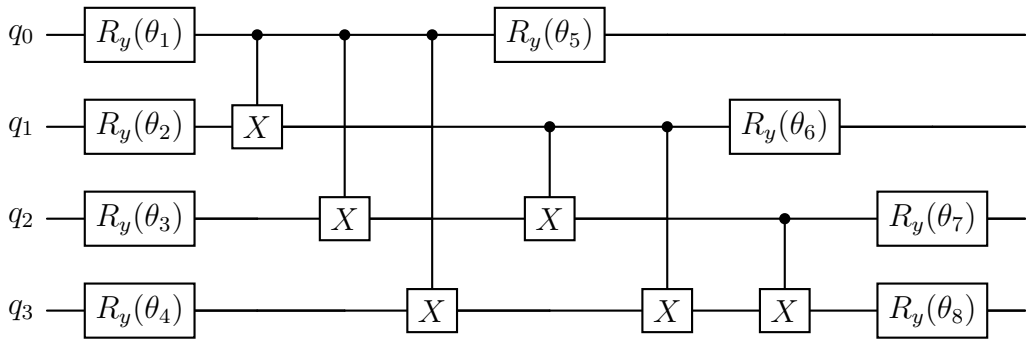


Figure 10: Real Amplitudes Ansatz.

Combined with the feature maps seen in Sec. 9.1, the variational form just described already closes the **unitary part** of a QNN; everything that follows lives *outside* the circuit and consists solely of the final computational-basis measurement and classical post-processing that turns amplitudes into classical data.

9.3 Output Measurements

Among the many problems tackled by modern AI, **classification** is the work-horse task that powers computer-vision pipelines, large-language models, speech recognition, fraud detection, medical triage, and countless other deep-learning applications. It will therefore be *our overriding objective* in the remainder of this part: convert quantum amplitudes into *class scores*.

9.3 OUTPUT MEASUREMENTS

After the variational form has acted, we project the register onto the computational basis[12]. A single shot returns a bit-string

$$z = (z_1, \ldots, z_m) \in \{0,1\}^m.$$

Repeating the circuit S times produces a multiset $\mathcal{Z} = \{z^{(1)}, \ldots, z^{(S)}\}$ whose empirical frequencies converge, in the $S \to \infty$ limit, to the Born probabilities $P(z) = \|\langle z| U(x,\theta) |\psi_x\rangle\|^2$.

Because our objective is classification, we can map every shot to one of C classes. Let

$$g : \{0,1\}^m \longrightarrow \{0, \ldots, C-1\}$$

denote a **hash** that assigns each string to a class index. The simplest and most hardware-friendly choice interprets the bit-string as an *unsigned integer* and then reduces modulo C:

$$c = g(z) := \Big[\underbrace{\sum_{i=1}^{m} 2^{m-i} z_i}_{\text{binary-to-int}} \Big] \bmod C. \tag{8}$$

Accumulating hash outcomes over all shots yields the **count** vector $\mathbf{n} = (n_0, \ldots, n_{C-1})^\top$, $n_c = \#\{ z \in \mathcal{Z} \mid g(z) = c\}$, and the **empirical class probabilities**

$$\hat{p}_c = \frac{n_c}{S}, \qquad \hat{\mathbf{p}} = \frac{\mathbf{n}}{\|\mathbf{n}\|_1} \in \Delta^{C-1}, \tag{9}$$

where Δ^{C-1} is the $(C-1)$-simplex. In mini-batch training we repeat this procedure for every sample in the batch, producing the tensor `results` shown in Listing 5. Those probabilities can be later fed directly to a classical loss (e.g. cross-entropy).

```
batch_size: int = x.shape[0]
num_classes: int = self.out_features
device = theta_vals.device
results = torch.zeros(batch_size, num_classes, device=device)
theta_list = theta_vals.detach().tolist()

for i in range(batch_size):
    feature_list = x[i].detach().tolist()
    shots = cudaq.sample(self.kernel,
                         feature_list,
                         theta_list,
```

[12]Measurements in rotated bases or using POVMs are possible, but the Z basis keeps hardware demands and exposition minimal.

```
                        shots_count=self.shots)
    # hash each bit-string via modulo-C rule
    class_ids = torch.tensor(
        list(
            map(
                lambda bs: int(bs, 2) % num_classes,
                sample_results.get_sequential_data(),
            )
        ),
        dtype=torch.long,
        device=device,
    )
    counts = torch.bincount(class_ids, minlength=num_classes)
    results[i] = counts / counts.sum()
```

Listing 5: Forward pass

Although the **hash-and-count rule** is appealingly simple, its expressive power is limited by the combinatorics of Eq. (8). A more flexible (and **hybrid**) alternative first extracts the *raw marginal statistics* $\hat{\mathbf{p}} \in \Delta^{2^m-1}$ of *all* observed bit-strings[13] and then applies a *classical* linear head

$$\mathbf{y} = \mathrm{softmax}\big(W\hat{\mathbf{p}} + \mathbf{b}\big), \qquad W \in \mathbb{R}^{C\times 2^m},\ \mathbf{b} \in \mathbb{R}^C. \tag{10}$$

Because W and $\mathbf{b}$ live entirely on the CPU/GPU, they can be trained together with the quantum parameters or fine-tuned afterwards, turning the QNN into a feature extractor and delegating the final decision boundary to a lightweight neural layer.

Although the hybrid head in Eq.,(10) offers greater expressivity, it also introduces significant **practical challenges**:

(i) The addition of a full 2^m-dimensional weight matrix drastically increases the number of trainable parameters, which in turn magnifies the stochastic noise from finite-shot sampling.

(ii) Coupling this large classical layer with the quantum circuit creates an especially rugged, non-convex **optimization landscape**, making training more difficult.

As a consequence, the much simpler modulo hash approach frequently achieves comparable accuracy using fewer training epochs and far lower shot counts.

With the **forward pass** now complete (feature map, variational form, measurement and post-processing) all that remains is to differentiate the circuit outputs with respect to the parameters $\boldsymbol{\theta}$. The following section develops the parameter-shift rule and other backward methods.

[13] In practice we restrict to the strings that actually appear in the shot record, padding with zeros so the vector length is constant.

10 Backward Pass

In classical deep learning, efficient training hinges on **backpropagation**, which computes all gradients in a single reverse pass through the network. Backpropagation relies on two key ideas:

(i) **Storing intermediate activations.** During the forward pass, every hidden-layer output is saved.

(ii) **Reusing those activations via the chain rule.** In the backward pass, we reuse the saved values to propagate gradients without re-running each layer from scratch.

As a result, computing the full gradient of a network with d parameters costs only about the same (up to log factors) as a single forward evaluation. This "backpropagation scaling" makes it possible to train very large models efficiently.

A quantum neural network (QNN), however, follows a very different paradigm. Once the forward circuit has run, the hardware holds *one copy* of the final m-qubit state $|g_\theta(x)\rangle$. Any projective measurement collapses that state, erasing its amplitude information. Because of the **No-Cloning theorem**, we cannot create additional perfect copies of $|g_\theta(x)\rangle$ to reuse in a backward pass. Consequently, we cannot "save" intermediate quantum states and later "rewind" through them.

No-Cloning Theorem (review of Secs. 3 and 4.6)

There is no physical operation (unitary) that, given an unknown quantum state $|\psi\rangle$, produces two identical copies $|\psi\rangle \otimes |\psi\rangle$. Formally, no unitary U on $\mathcal{H} \otimes \mathcal{H}$ can satisfy

$$U\big(|\psi\rangle \otimes |0\rangle\big) \;=\; |\psi\rangle \otimes |\psi\rangle \quad \text{for all } |\psi\rangle.$$

Simple proof sketch. Suppose, for contradiction, there were a unitary U and a fixed "blank" state $|0\rangle$ such that

$$U\big(|\psi\rangle \otimes |0\rangle\big) = |\psi\rangle \otimes |\psi\rangle \quad \text{for every unknown } |\psi\rangle.$$

Pick two distinct, non-orthogonal states $|\phi\rangle$ and $|\chi\rangle$. Their inner product is $\langle\phi \mid \chi\rangle = s \neq 0, 1$. After cloning, the inner product of the outputs would be $\langle\phi \otimes \phi \mid \chi \otimes \chi\rangle = s^2$. But a unitary must preserve inner products. We would require $s = s^2$, which holds only if $s = 0$ or $s = 1$. That contradiction shows cloning is impossible.

Because of No-Cloning, any "quantum backpropagation" must *reprepare* the state $|g_\theta(x)\rangle$ from scratch at each needed parameter configuration. In practice, to estimate partial derivative of the loss with respect to the k-th parameter, $\partial_{\theta_k} L$, we run two new circuits with parameters $\theta_k \pm \Delta$. Below we examine two such methods.

10.1 Finite-Difference Gradients

A naive and straightforward way to approximate the derivative $\partial_\theta L(\theta)$, without relying on any specialized quantum gradient techniques, is to employ the **centered finite-differences**:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + s) - L(\theta - s)}{2s}, \quad \text{error} = \mathcal{O}(s^2).$$

Here $L(\theta)$ is obtained by running the circuit at θ and measuring an observable multiple times. To estimate $L(\theta \pm s)$, we must run two independent forward circuits. In a QNN with d parameters, each small step s costs two full circuit executions. Moreover:

- If s is too small, shot noise (from finite sampling) dominates the numerator $L(\theta + s) - L(\theta - s)$.
- If s is too large, the $\mathcal{O}(s^2)$ error becomes significant.

Thus one must carefully choose s, and still pay $2M$ circuit calls per data point. For even moderate d, this quickly becomes infeasible on noisy, near-term hardware.

10.2 Parameter-Shift Rule

The parameter-shift rule can be viewed as a special case of the centered finite-difference formula with step $s = \frac{\pi}{2}$. In a generic finite-difference approximation, one incurring an $\mathcal{O}(s^2)$ truncation error. However, choosing $s = \frac{\pi}{2}$ makes that truncation error vanish identically. This choice is not merely convenient, $\pi/2$ is the unique step size that yields an exact derivative for the loss function derivative.

A key feature of the variational forms introduced in Secs. 6.6 and 9.2 is that each trainable gate acts as a **rotation** about a Pauli operator (or Pauli string) P with $P^2 = I$. Concretely, any single-parameter gate in our QNN can be written as

$$R_P(\theta) = \exp\Big(-\tfrac{i}{2}\,\theta\,P\Big) = \cos\big(\tfrac{\theta}{2}\big)\,I - i\,\sin\big(\tfrac{\theta}{2}\big)\,P, \qquad P^2 = I.$$

Since every other gate in the network is *independent* of θ, any scalar loss function $L(\theta)$ produced by the circuit (for example, a measured probability, an expectation value,

or a loss) must be a linear combination of $\cos(\theta/2)$ and $\sin(\theta/2)$. In other words, there exist real constants A and B such that

$$L(\theta) = A\,\cos\!\left(\tfrac{\theta}{2}\right) + B\,\sin\!\left(\tfrac{\theta}{2}\right), \quad \frac{dL}{d\theta} = -\tfrac{1}{2}\,A\,\sin\!\left(\tfrac{\theta}{2}\right) + \tfrac{1}{2}\,B\,\cos\!\left(\tfrac{\theta}{2}\right).$$

On the other hand, notice that shifting the argument θ by $\frac{\pi}{2}$ effectively rotates the half-angle $\frac{\theta}{2}$ by $\frac{\pi}{4}$. In particular, one has

$$\cos\!\left(\tfrac{\theta+\frac{\pi}{2}}{2}\right) = \cos\!\left(\tfrac{\theta}{2}+\tfrac{\pi}{4}\right) = \tfrac{1}{\sqrt{2}}\left[\cos\!\left(\tfrac{\theta}{2}\right) - \sin\!\left(\tfrac{\theta}{2}\right)\right],$$

$$\sin\!\left(\tfrac{\theta+\frac{\pi}{2}}{2}\right) = \sin\!\left(\tfrac{\theta}{2}+\tfrac{\pi}{4}\right) = \tfrac{1}{\sqrt{2}}\left[\sin\!\left(\tfrac{\theta}{2}\right) + \cos\!\left(\tfrac{\theta}{2}\right)\right],$$

and similar goes for $\theta - \frac{\pi}{2}$. A short trigonometric check then shows

$$\frac{L\left(\theta+\frac{\pi}{2}\right) \;-\; L\left(\theta-\frac{\pi}{2}\right)}{2} \;=\; -\tfrac{1}{2}\,A\,\sin\!\left(\tfrac{\theta}{2}\right) \;+\; \tfrac{1}{2}\,B\,\cos\!\left(\tfrac{\theta}{2}\right) \;=\; \frac{dL}{d\theta}.$$

Hence, for any gate of the form $R_P(\theta)$ with $P^2 = I$ and $\forall\,\theta$, the derivative of the circuit output with respect to θ is given *exactly* by

$$\frac{dL(\theta)}{d\theta} \;=\; \frac{L\left(\theta+\frac{\pi}{2}\right) \;-\; L\left(\theta-\frac{\pi}{2}\right)}{2}, \quad \text{error} = 0.$$

Below is a PyTorch sketch of how one might implement parameter-shift where `quantum_circuit.run` is the forward pass of Listing 5.

```
theta, x, _ = ctx.saved_tensors
gradients: torch.Tensor = torch.zeros_like(theta)
for i in range(theta.numel()):
    theta_plus: torch.Tensor = theta.clone()
    theta_plus[i] += ctx.shift  # Shift forward by +pi/2
    y_plus: torch.Tensor = ctx.quantum_circuit.run(theta_plus
   , x)

    theta_minus: torch.Tensor = theta.clone()
    theta_minus[i] -= ctx.shift  # Shift forward by -pi/2
    y_minus: torch.Tensor = ctx.quantum_circuit.run(
   theta_minus, x)

    diff: torch.Tensor = (y_plus - y_minus) / (2 * ctx.shift)
    gradients[i] = torch.sum(grad_output * diff)
```

Listing 6: Backward pass (with parameter-shift rule)

Because there are d parameters, each requiring two circuit executions, and each execution typically involves S measurement shots, the total **cost** per data point is $2M \times S$ executions. Even for tens of parameters, this becomes prohibiting on current hardware. In the following sections, we will therefore explore alternative "quantum backpropagation" strategies that can mitigate this overhead and make gradient estimation more efficient.

10.3 Other Backward Methods

Recent work has introduced two distinct paradigms as potential routes to recover true **backpropagation scaling** (i.e. a gradient-to-cost overhead logarithmic in the number of parameters) [16]. In what follows, we sketch the main ideas of these methods. We first recall that our QNN is a unitary circuit

$$U(\mathbf{x},\,\theta)\;:\;|0\rangle^{\otimes m}\;\longmapsto\;|\psi_{\mathbf{x},\,\theta}\rangle\;=\;U(\mathbf{x},\,\theta)\;|0\rangle^{\otimes m},$$

and every qubit is measured in the Z basis at the end of the forward pass. The classical loss function

$$L(\mathbf{x},\,\theta)\;=\;\ell\Big(\{\,z\in\{0,1\}^m\}_{\text{shots}}\Big)$$

is computed from the bit-string frequencies $\hat{p}(z)$; in particular, for analytic gradient formulas we often think of L as the expectation value of some Pauli-string operator on the final state $|\psi_{\mathbf{x},\,\theta}\rangle$. Our goal is to estimate all d partial derivatives $\partial L/\partial\theta_k$ with only $O(\log d)$ extra "depth" rather than $O(d)$.

1. Shadow-tomography approach. At each parameter index k, one can rewrite the kth gradient component in the form

$$\frac{\partial L}{\partial\theta_k}\;=\;2\,\Im\Big\langle\,0^{\otimes m}\;\Big|\;U(\mathbf{x},\,\theta)^{\dagger}\,\Big(-\,i\,P_k\Big)\,U(\mathbf{x},\,\theta)\;\Big|\;0^{\otimes m}\Big\rangle,$$

where P_k is the Pauli (or Pauli-string) generator on the kth parameter [16]. Equivalently, one may define two "half-circuit" states

$$|\Psi_k\rangle\;=\;\Big(e^{+\,i\left(\theta_k+\frac{\pi}{2}\right)P_k}\prod_{j<k}e^{-\,i\,\theta_j\,P_j}\Big)\;|0\rangle^{\otimes m},\qquad|\Phi_k\rangle\;=\;P_k\,\Big(\prod_{j>k}e^{-\,i\,\theta_j\,P_j}\Big)\;|\Psi_k\rangle\,,$$

so that $\partial L/\partial\theta_k$ is proportional to the overlap $\Im\langle\Phi_k\,|\,\Psi_k\rangle$. To access that overlap on hardware, one introduces a single *auxiliary qubit*[14] in the $|+\rangle_*$ state and coherently

[14]Throughout this book we reserve "auxiliary qubit" (rather than "**ancilla**") for any qubit used solely to mediate interference or perform temporary operations.

controls between $|\Psi_k\rangle$ (when the auxiliary is $|0\rangle_*$) and $|\Phi_k\rangle$ (when the auxiliary is $|1\rangle_*$). The joint state becomes

$$|\psi_k\rangle \;=\; \frac{1}{\sqrt{2}}\Big(|0\rangle_*\,|\Psi_k\rangle \;+\; |1\rangle_*\,|\Phi_k\rangle\Big).$$

Measuring the auxiliary qubit in the X basis is equivalent to applying a Hadamard on the auxiliary and then measuring in Z. One finds

$$\big\langle\psi_k\big|X_*\big|\psi_k\big\rangle \;=\; \frac{1}{2}\,\frac{\partial L}{\partial\theta_k},$$

so that extracting $\partial L/\partial\theta_k$ to accuracy ϵ by direct sampling costs $O(1/\epsilon^2)$ shots of $|\psi_k\rangle$. If we do this independently for each $k = 1, \ldots, d$, the total cost is $O(d/\epsilon^2)$, which offers no improvement over the parameter-shift rule.

The *shadow-tomography idea* is to **recycle** as many of those $|\psi_k\rangle$ preparations as possible by maintaining a *classical hypothesis* density matrix ω_k that *predicts* the overlap $\langle\psi_k|X_*|\psi_k\rangle$. Concretely, at step k:

First, the learner computes the **predicted value**

$$a_k \;=\; \mathrm{tr}\big[X_*\,\omega_k\big]$$

as an estimate of $\langle\psi_k|X_*|\psi_k\rangle$. Next, one takes a small batch of freshly prepared $|\psi_k\rangle$ copies and performs a **gentle threshold check**: one verifies whether

$$\big|a_k - \langle\psi_k|X_*|\psi_k\rangle\big| \;\le\; \epsilon.$$

If the check **succeeds**, we accept a_k as our gradient estimate and, crucially, do *not* collapse all remaining $|\psi_k\rangle$ copies, so we preserve quantum information for the next step. In that case, we set $\omega_{k+1} = \omega_k$ and move on.

If the threshold check **fails**, we know our hypothesis was off by more than ϵ. In that event we perform a fresh, higher-accuracy measurement of X_* on another batch of $|\psi_k\rangle$ to obtain $b_k \approx \langle\psi_k|X_*|\psi_k\rangle$ within ϵ. We then update ω_{k+1} (via an online-learning rule) to incorporate the datum $X_* = b_k$, discard the used copies of $|\psi_k\rangle$, and prepare a new batch of $|\psi_{k+1}\rangle$ for the next index.

By partitioning the total $O(\log^2 d/\epsilon^2)$ copies of $|\psi(\mathbf{x},\,\theta)\rangle$ into $O(m/\epsilon^2)$ sequential "batches," one can guarantee that only

$$d \;=\; O\Big(\frac{m\,\log^2 d}{\epsilon^4}\Big)$$

copies are consumed in all d steps combined. Moreover, each time we advance from $|\psi_k\rangle$ to $|\psi_{k+1}\rangle$, we need only apply $O(1)$ controlled-Pauli rotations (namely $U_{\Psi,k+1}\,U_{\Psi,k}^{\dagger}$ or $U_{\Phi,k+1}\,U_{\Phi,k}^{\dagger}$) to produce $|\psi_{k+1}\rangle$ from the remaining $|\psi_k\rangle$ copies. Hence the total number of controlled-unitary calls across all k grows **quasi-linearly** in d, rather than linearly in d. In other words, the *quantum-depth overhead per gradient component* is only $O(\log d)$, instead of $O(1)$ full forward runs. The trade-off is that each update to ω_k requires storing an $(m+1)$-qubit density matrix classically, at cost $O(2^{2n})$ per update. Unless ω_k admits an efficient compression (e.g. via tensor networks), that **classical overhead** is **prohibitive** for large m. Nevertheless, this shadow-tomography protocol shows how, *in principle*, one can achieve backpropagation-like scaling in *quantum resources* if one is willing to pay an exponential classical memory cost.

2. Quantum-memory approach. A second paradigm sidesteps the classical-memory burden by granting a stronger and (as of today) **hypothetical** quantum RAM: assume we can coherently store $r = O(\log d)$ *identical* copies of the full variational state

$$|\psi(\mathbf{x},\,\theta)\rangle \;=\; U(\mathbf{x},\,\theta)\;|0\rangle^{\otimes m}$$

simultaneously. Label those registers $\mathcal{R}_1, \ldots, \mathcal{R}_r$, each hosting $|\psi(\mathbf{x},\,\theta)\rangle$, plus one auxiliary qubit in $|+\rangle_*$. To estimate $\partial L/\partial\theta_k$, we perform:

First, in parallel on all r copies, apply

$$\text{(control-}|0\rangle_*\text{)} \;:\; e^{+\,i\left(\theta_k+\frac{\pi}{2}\right)P_k} \quad\text{and}\quad \text{(control-}|1\rangle_*\text{)} \;:\; e^{-\,i\left(\theta_k-\frac{\pi}{2}\right)P_k}$$

on each register. Because each controlled rotation acts on a distinct copy of $|\psi(\mathbf{x},\,\theta)\rangle$, the global circuit depth increases by only $O(\log d)$, rather than $O(d)$.

Finally, apply a Hadamard on the auxiliary and measure it (together with all data qubits) in the Z basis. One finds that

$$\langle Z_*\rangle \;=\; \langle\,+_*\,|\,X_*\,|\,+_*\rangle \;=\; \frac{1}{2}\,\frac{\partial L}{\partial\theta_k},$$

so a single batch of r coherent copies suffices to estimate $\partial L/\partial\theta_k$. Repeating the entire procedure $O(\log d/\epsilon^2)$ times yields all d gradients within ϵ. Hence the *quantum-depth overhead* for each gradient is $O(\log d)$, and the total cost to get all d gradients is

$$O\big(\mathrm{depth}[\,U(\mathbf{x},\,\theta)\,] + \log d\big)$$

up to **polylogarithmic** factors. In other words, one exactly matches classical backpropagation scaling in quantum resources if one can store $O(\log d)$ copies of an m-qubit state without collapse. Of course, today's hardware cannot maintain that many identical

coherent copies, so this remains a theoretical construction until fault-tolerant quantum RAM appears.

In **both approaches** the same tension emerges: you either incur an exponential **classical memory** cost (through a large classical hypothesis ω_k) to save quantum depth, or else demand large **quantum memory** (logarithmic in d) to avoid classical overhead. Neither variant is practical for large m and d on near-term devices; absent those resources we revert to the **parameter-shift rule** in Torch-Q, whose $O(d)$ circuit calls remain the simplest and most robust means to compute gradients.

11 Complete Model

Thus far we have introduced how to embed classical data into quantum states via *feature maps* and how to process those states with a trainable *variational form*. We also saw that, because every parametrized gate is of the form $R_P(\theta) = \exp(-\frac{i}{2}\,\theta\,P)$ with $P^2 = I$, one can compute exact gradients via the *parameter-shift rule* by running two shifted circuits per parameter. In practice, we combine the feature map and variational form into a single quantum circuit (a `cudaq.Kernel`), measure in the Z-basis, and hash each bit-string into class labels or collect all 2^n basis probabilities for a classical head. **Torch-Q** exposes all this functionality through four high-level classes (**QuantumFunction, QuantumLayer, QNN, and HybridQNN**) so that building and training a QNN feels almost identical to any other PyTorch model. In what follows, we describe each class in turn and give a small code snippet illustrating its usage.

11.1 QuantumFunction

QuantumFunction is a custom `torch.autograd.Function` that wraps the entire quantum circuit evaluation and its gradients. In its constructor we specify

`in_features`, `out_features`, `feature_map`, `var_form`, `shots`, `reupload`, `shift`.

Upon **initialization**, it builds a single `cudaq.Kernel` which, depending on the `reupload` flag, either applies the feature map once or interleaves it between each variational layer. In the **forward pass**, `QuantumFunction` takes a tensor of parameters $\theta \in \mathbb{R}^d$ and a batch of inputs $x \in \mathbb{R}^{B\times m}$, runs `cudaq.sample` for each sample, hashes each bit-string via `int(bitstring,2)%C`, and returns an empirical $\mathbb{R}^{B\times C}$ probability tensor. In the **backward pass**, it loops over the d parameters and re-runs each circuit with $\theta_k \pm \frac{\pi}{2}$ to evaluate

$$\frac{\partial L}{\partial \theta_k} \;=\; \frac{L(\theta_k + \frac{\pi}{2}) - L(\theta_k - \frac{\pi}{2})}{2}\,,$$

accumulating `grad_output` $\times \big(L(\theta^{(+)}) - L(\theta^{(-)})\big)/2$ into ∇_{θ_k}. This way, **PyTorch** will automatically build the backward graph and compute exact gradients by parameter-shift when calling `QuantumFunction.apply`.

The following snippet demonstrates how to build a `QuantumFunction` for a 4-qubit circuit that classifies into 3 labels. We show only a forward-pass call; if you wrap this inside a training loop and call `loss.backward()`, PyTorch will invoke the parameter-shift logic automatically.

```python
import torch
import torch.nn as nn
import cudaq
import torchq.quantum.feature_maps as qfm
import torchq.quantum.variational_forms as qvf
from torchq.nn.functional import QuantumFunction

# 1) Define number of qubits and classes
in_features = 4  # 4 qubits
out_features = 3
shots = 512

# 2) Choose a feature map and an Ansatz
fm = qfm.ZFeatureMap(in_features=in_features, num_layers=1)
vf = qvf.EfficientSU2(num_qubits=in_features, num_layers=2)

# 3) Construct the QuantumFunction
quantum_fn = QuantumFunction(
    in_features=in_features,
    out_features=out_features,
    feature_map=fm,
    var_form=vf,
    shots=shots,
    reupload=False,
    shift=torch.pi/2
)

# 4) Prepare a small batch of inputs and parameters
x_batch = torch.rand(2, in_features)
theta = nn.Parameter(torch.rand(vf.get_parameter_count()))

# 5) Forward pass
probs = quantum_fn(theta, x_batch, quantum_fn)
print("Probs shape:", probs.shape)
```

11.2 QuantumLayer

QuantumLayer is an `nn.Module` that wraps `QuantumFunction` and declares its own trainable parameter tensor. Its constructor takes

`in_features`, `out_features`, `num_layers`, `shots`, `feature_map`, `var_form`, `reupload`.

If `feature_map` or `var_form` is given as a **string** (e.g. ‘"z"‘, ‘"zz"‘, ‘"realamplitudes"‘, ‘"efficientSU2"‘, or ‘"paulitwodesign"‘), `QuantumLayer` internally instantiates the corresponding Torch-Q objects. Alternatively, users can also supply their own **custom**

`FeatureMap` and `VariationalForm` instances directly. It then creates a `QuantumFunction` instance that holds the combined circuit, and declares a single $\mathrm{nn.Parameter}(\theta) \in \mathbb{R}^d$. Its `forward(x)` method simply does

$$\texttt{QuantumFunction.apply(self._theta, x, self.quantum\_circuit)},$$

which yields a batch of length-C probability vectors. In effect, all quantum logic is hidden inside `QuantumFunction`, and `QuantumLayer` behaves like any other `nn.Module`.

Below we show how to instantiate a `QuantumLayer` that maps 4 inputs to 3 classes, using two layers of Z-embeddings and an EfficientSU2 Ansatz with reuploading.

```python
import torch
from torchq.nn.layers import QuantumLayer

# Instantiate a QuantumLayer: 4 qubits, 3 outputs, 2
   variational layers, 1024 shots
qlayer = QuantumLayer(
    in_features=3,
    out_features=4,
    num_layers=2,
    shots=1024,
    feature_map="z",
    var_form="efficientSU2",
    reupload=True
)

# Make a small batch of 5 input vectors
x = torch.rand(5, 3)  # shape (5,3)

# Forward pass
probabilities = qlayer(x)
print("QuantumLayer output:", probabilities)
```

Because `QuantumLayer` declares its own θ as `nn.Parameter`, you can immediately plug it into a PyTorch optimization routine and call `probabilities.backward()` inside a loss loop; PyTorch will call into `QuantumFunction.backward` under the hood.

11.3 QNN

QNN is simply a thin wrapper around `QuantumLayer`. Its constructor is identical to `QuantumLayer`'s, and its entire `forward(x)` function just invokes the embedded quantum layer. Use `QNN` whenever you want a **purely quantum model** whose number of

output classes is at most 2^m. Training `QNN` is exactly like training any other PyTorch model, except that each backward step incurs the parameter-shift cost.

Here is a complete **training loop** for a `QNN` that classifies random 4-dimensional inputs into 3 classes. We use mini-batches of size 8 and the Adam optimizer.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.datasets import make_classification
from torchq.nn.layers import QNN

# 1) Generate a multiclass dataset with 4 features and 3 classes
X_np, Y_np = make_classification(
    n_samples=240,
    n_features=4,
    n_informative=4,
    n_redundant=0,
    n_classes=3,
    n_clusters_per_class=1,
    class_sep=1.0,
    random_state=0
)
X_data = torch.from_numpy(X_np).float()   # shape (240, 4)
Y_data = torch.from_numpy(Y_np).long()  # shape (240,)

# 2) Wrap in a TensorDataset and DataLoader
dataset = TensorDataset(X_data, Y_data)
batch_size = 20
loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# 3) Instantiate the QNN: 4 qubits, 3 classes, 2 layers, 1024 shots
model = QNN(
    in_features=4,
    out_features=3,
    num_layers=2,
    shots=1024,
    feature_map="z",
    var_form="efficientSU2",
    reupload=False
)

optimizer = optim.Adam(model.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()
```

```
40
41 # 4) Training loop using DataLoader
42 for epoch in range(10):
43     epoch_loss = 0.0
44     for x_batch, y_batch in loader:
45         # Forward pass
46         logits = model(x_batch)
47         loss = criterion(logits, y_batch)
48
49         # Backward pass
50         optimizer.zero_grad()
51         loss.backward()
52         optimizer.step()
53
54         epoch_loss += loss.item()
55
56     print(f"Epoch {epoch+1}, Loss = {epoch_loss:.4f}")
```

11.4 HybridQNN

When the number of classes exceeds 2^m, or when additional classical **flexibility** is desired, we use `HybridQNN`. It instantiates its own `QuantumLayer` with `out_features` set to 2^m, so the quantum layer returns the **probability distribution** over all 2^m quantum states. It then attaches a small classical `nn.Linear(`2^m`, out_features)` on top, allowing any desired number of final classes. During training, gradient flow goes through both the quantum parameters (via parameter-shift) and the classical linear head (via standard autodiff).

Below we train a `HybridQNN` on random 3-dimensional inputs, but classify into 5 classes.

```
1  import torch
2  import torch.nn as nn
3  import torch.optim as optim
4  from torch.utils.data import TensorDataset, DataLoader
5  from sklearn.datasets import make_classification
6  from torchq.nn.layers import HybridQNN
7
8  # 1) Generate a multiclass dataset with 3 features and 5
       classes
9  X_np, Y_np = make_classification(
10     n_samples=200,
11     n_features=3,
12     n_informative=3,
13     n_redundant=0,
```

```
    n_classes=5,
    n_clusters_per_class=1,
    class_sep=1.2,
    random_state=7
)
X_data = torch.from_numpy(X_np).float()   # shape (200, 3)
Y_data = torch.from_numpy(Y_np).long()  # shape (200,)

# 2) Wrap in a TensorDataset and DataLoader
dataset = TensorDataset(X_data, Y_data)
batch_size = 25
loader = DataLoader(dataset, batch_size=batch_size, shuffle=
    True)

# 3) Instantiate the HybridQNN: 3 qubits, 8 quantum outputs,
    5 classes
hybrid_model = HybridQNN(
    in_features=3,
    out_features=5,
    num_layers=2,
    shots=1024,
    feature_map="z",
    var_form="realamplitudes",
    reupload=False
)

optimizer = optim.Adam(hybrid_model.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()

# 4) Training loop using DataLoader
for epoch in range(12):
    total_loss = 0.0
    for x_batch, y_batch in loader:
        # Forward pass
        logits = hybrid_model(x_batch)
        loss = criterion(logits, y_batch)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}, Loss = {total_loss:.4f}")
```

11.5 Important Notes

Parameter-Shift Overhead. Whenever you call `loss.backward()`, if d is large, this can become expensive. In practice, choose d wisely and batch sizes modest to limit overall quantum calls.

Reuploading. Setting `reupload=True` causes the feature map to be applied before each variational layer. This can improve training by repeatedly re-encoding x at different depths [17], but it doubles the number of feature-map calls and may increase circuit depth.

Output Dimension Constraint. In `QuantumLayer` (and thus in `QNN`) you must ensure

$$1 < C \leq 2^m.$$

If you need more classes than 2^m, use `HybridQNN`, which allows a classical linear layer on top of the 2^m probability vector.

HybridQNN Parameter Overhead. Although `HybridQNN` removes the 2^m class-limit, its classical head carries $C \times 2^m$ trainable weights. This exponential parameter growth not only demands more memory and compute, but also amplifies finite-shot gradient noise and creates a more rugged, non-convex optimisation landscape; often making training slower and less stable than the simple hash-and-count variant.

Shot Noise. Finite `shots` implies statistical noise in your loss and gradient estimates. Increase `shots` or average over several forward passes to reduce variance.

Hardware Targets. By default, `QuantumFunction` uses whatever `cudaq.set_target` is active (e.g. `"qpp-cpu"` for CPU simulation or `"nvidia"` for GPU state-vector). To execute on actual hardware, simply call

`cudaq.set_target("ionq", qpu="qpu.aria-1")` or similar,

and all subsequent `cudaq.sample(...)` calls will run on that device.

In conclusion, **Torch-Q's abstractions** make it straightforward to **build, train and evaluate** both pure quantum and hybrid quantum–classical neural networks within the familiar PyTorch ecosystem. The only quantum-specific wrinkle is that each backward pass re-evaluates circuits via parameter-shift; beyond that, everything else behaves just like standard deep-learning code.

12 Information Geometry of Model Capacity

In the previous chapters we have introduced quantum neural networks (QNNs) and classical neural networks (NNs) as parametric families of functions. However, the **practical benefits** of quantum machine learning remain unclear. In particular, it is not sufficient merely to count the number of trainable parameters d to evaluate a model's *potential*, since many of those parameters may have only a negligible effect on the outputs when trained on finite data. Instead, we would like a **capacity** measure that reflects, for a given sample size n, how large a "volume" in function space the model can actually realize. Intuitively, a highly **expressive model** can fit a wide repertoire of complex functions (e.g., images or linguistic patterns). To understand why some models generalize well while others do not, we examine how sensitive the model's outputs are to perturbations in each **parameter direction**.

Parameter Directions: Informative versus Flat

Suppose our network has parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_d) \in \Theta$. A *parameter direction* is simply a **tangent vector** in the d-dimensional parameter space $\Theta \subset \mathbb{R}^d$. Moving along one of these directions corresponds to infinitesimally adjusting one (or a linear combination) of the network's weights, thereby perturbing its output function. For instance, one simple parameter direction is "increase θ_1 and θ_2 together," i.e. move along the vector $v = (1, 1, 0, \ldots, 0)$ in parameter space. In general, any choice of a vector $v \in \mathbb{R}^d \setminus \{0\}$ specifies a parameter direction.

If varying along a given direction causes a substantial change in the model's predictions for typical inputs, we call it an **informative** direction. Conversely, if that variation has almost no effect on the outputs (regardless of the data distribution), then the direction is effectively **flat** under the resolution permitted by n samples. When many parameter directions are essentially flat, the model behaves as though it has far fewer than d effective degrees of freedom, making it **effectively lower-dimensional** at that scale of data.

To calculate the *"real"* dimension of the entire **parameter manifold**[15], we turn to tools from information geometry. Information geometry is the study of the differential-geometric structure of families of statistical models (e.g. neural networks), where the

[15]The parameter manifold can be viewed as a **Riemannian manifold**. A Riemannian manifold is a smooth manifold M equipped with a **metric** g, which assigns to each tangent space T_pM a smoothly varying positive-definite inner product $\langle \cdot, \cdot \rangle_p$. This metric allows one to measure distances, angles, lengths, volumes and curvatures on the manifold.

Fisher information defines a metric on the parameter space [18]. In this section we will show how the **Fisher information** equips the parameter manifold with the Fisher–Rao metric, and how one can derive a scale-dependent notion of "**effective dimension**" from that function; that captures the number of parameters a model *truly leverages* on a dataset of size n.

Before delving into details, let us briefly recall why an information-geometric perspective is warranted. Traditional complexity measures (such as **VC dimension**) are asymptotic in n and often grow trivially with the raw parameter count d, making them ineffective for overparameterized models such as deep NNs or QNNs. In contrast, the *effective dimension* is **data-dependent** and **scale-dependent**: it measures how many **"distinguishable"** parameter directions remain at the statistical resolution set by n samples. In other words, when n is finite, flat directions (or, equivalently, small **Fisher eigenvalues**) cannot be resolved and effectively reduce the dimension of the hypothesis space. This is precisely the phenomenon we wish to quantify in order to compare QNNs and classical NNs on equal footing.

12.1 The Fisher Information

A (classical or quantum) neural network with d parameters defines a **conditional distribution**

$$p_\theta(y \mid x) \;=\; p(y \mid x;\theta), \quad x \in \mathcal{X} \subset \mathbb{R}^m,\; y \in \{1,\dots,C\},$$

by feeding x through the network and normalizing its outputs to a probability vector. Training on n i.i.d. samples $(x_i, y_i) \sim p_{\text{data}}(x)\, p_{\theta^*}(y \mid x)$ thus reduces to choosing θ to maximize the **conditional log-likelihood**

$$\ell_n(\theta) \;=\; \sum_{i=1}^{n} \ln p_\theta\big(y_i \mid x_i\big).$$

Each summand $\ln p_\theta(y_i \mid x_i)$ has a finite variance $\mathrm{Var}_{(x,y)\sim p_{\text{data}} p_{\theta^\star}}\big[\ln p_\theta(y \mid x)\big] = \sigma^2(\theta)$. By the central-limit theorem the **deviation** of $\ell_n(\theta)$ from its mean scales as $\sqrt{n}\,\sigma(\theta)$. Because of this noise, the maximizer of ℓ_n jitters by $O(\sqrt{n})$. We denote this intrinsic uncertainty by ε_n and call it the **resolution scale**. As $n \to \infty$ and under standard regularity conditions (smoothness of p_θ, identifiability, $\mathbb{E}_\theta[\nabla \ell_1] = 0$, dominated convergence), ε_n shrinks to zero and the estimator $\hat{\theta}_n$ converges to the true parameter $\theta^\star$.

Our **objective** remains to examine how sensitive the model's outputs are to perturbations in each parameter direction (i.e. "*how sharply the likelihood bends around* θ").

To quantify **local sensitivity** of the model to infinitesimal changes in θ, define the *score function* for a single data pair (x, y) by

$$s_i(x, y; \theta) \;=\; \frac{\partial}{\partial \theta_i} \ln p(x, y; \theta), \qquad i = 1, \dots, d.$$

The **Fisher information matrix** $F = (F_{ij})_{i,j=1}^{d} \in \mathbb{R}^{d \times d}$ is then defined by

$$F_{ij}(\theta) \;=\; \mathbb{E}_{(x,y)\sim p(x,y;\theta)}\Big[s_i(x, y; \theta)\, s_j(x, y; \theta) \Big] \;=\; -\mathbb{E}_{(x,y)\sim p(x,y;\theta)}\Big[\tfrac{\partial^2}{\partial \theta_i \partial \theta_j} \ln p(x, y; \theta) \Big],$$

where the **expectation** is taken over the joint distribution $p(x, y; \theta) = p_{\text{data}}(x)\, p_\theta(y \mid x)$. In words, $F_{ij}(\theta)$ measures the **average covariance** of the partial derivatives of the log-likelihood with respect to θ_i and θ_j.

Equivalently, one may view the parameter manifold Θ as a *Riemannian manifold* equipped with the **Fisher–Rao metric** [19]:

$$g_{ij}(\theta) \;=\; F_{ij}(\theta), \qquad 1 \le i, j \le d,$$

so that the squared infinitesimal **distance** between θ and $\theta + d\theta$ is

$$ds^2 \;=\; d\theta^{\top}\, F(\theta)\, d\theta.$$

Under this metric, the **volume** element at θ is $\sqrt{\det F(\theta)}\, d\theta$, and the total "**Jeffreys volume**" of the parameter manifold Θ is

$$\int_{\Theta} \sqrt{\det F(\theta)}\, d\theta.$$

This volume form quantifies how many effectively "**distinguishable**" parameter points lie in Θ given n samples.

To see how the curvature of the likelihood limits what the data can really distinguish, note first that the Fisher determinant factorizes through its **eigenvalues**,

$$\det F(\theta) = \prod_{i=1}^{d} \lambda_i(\theta), \qquad \sqrt{\det F(\theta)} = \prod_{i=1}^{d} \sqrt{\lambda_i(\theta)}.$$

A large eigenvalue $\lambda_k(\theta)$ means that a perturbation $\delta\theta = \varepsilon\, v_k$ along the corresponding **eigenvector (i.e. parameter direction)** v_k produces a sizable shift in the conditional distribution, so that direction is **informative**; when $\lambda_k(\theta) \approx 0$, the same perturbation hardly alters $p_\theta(y \mid x)$, making the direction **flat**. Quantitatively, the expected second-order log-likelihood change is

$$\langle \Delta \ell_n \rangle = \tfrac{n}{2} \lambda_k \varepsilon_n^2 = O(\lambda_k).$$

Because the noise floor of ℓ_n is $O(1)$, λ_k must itself be $O(1)$ to be detectable; a point stressed by Rissanen's stochastic-complexity argument [20]. Re-expressing the same idea through the **Cramér–Rao bound**[16], $\Delta\theta_k \simeq (n\lambda_k)^{-1/2}$, and comparing with ε_n gives the handy rule

$$\boxed{\begin{aligned} \lambda_k \gg n^{-1} \quad &\Longrightarrow \quad \Delta\theta_k \ll \varepsilon_n \quad \text{(informative)}, \\ \lambda_k \ll n^{-1} \quad &\Longrightarrow \quad \Delta\theta_k \gg \varepsilon_n \quad \text{(flat)}. \end{aligned}}$$

Hence only eigenvalues above $1/n$ contribute appreciably to $\sqrt{\det F(\theta)}$; if merely $\tilde{d} \ll d$ clear that threshold, the product $\prod_{k=1}^{d} \sqrt{\lambda_k(\theta)}$ is suppressed, shrinking the local Jeffreys volume and effectively lowering the model's dimension at finite n.

In the context of **deep learning**, $\{\lambda_i(\theta)\}$ depends on both the network architecture and the data distribution; computing $F(\theta)$ exactly is often **intractable**, but many approximations exist (e.g. block-diagonal, Kronecker-factored, or empirical Fisher approximations). The **empirical Fisher information matrix**, defined by replacing the true expectation over (x, y) with an average over a finite data set, is

$$\tilde{F}_n(\theta) \;=\; \frac{1}{n}\sum_{i=1}^{n} \big[\frac{\partial}{\partial\theta} \ln p(x, y; \theta)\big] \, \big[\frac{\partial}{\partial\theta} \ln p(x, y; \theta)\big]^{\top}.$$

Although this ignores variance due to resampling the labels, it is **efficient** to compute via back-propagation and captures useful curvature information in neural networks. In **Torch-Q**, precisely in `torchq.func`, the empirical Fisher is returned by `fisher`.

```python
from torchq.func import fisher

model_q = QNN(
    in_features=4,
    out_features=3,
    num_layers=9,
    shots=1024,
    feature_map="z",
    var_form="realamplitudes",
    reupload=False,
)

X_rand = torch.randn(100, 4)
f = fisher(model_q, X_rand)
```

[16] For any *unbiased* estimator $\hat{\theta}$ of the true parameter $\theta^{\star}$, the **Cramér–Rao inequality** states that its covariance obeys $\mathrm{Cov}(\hat{\theta}) \succeq \frac{1}{n} F(\theta^{\star})^{-1}$. Along an eigen-direction v_k of $F(\theta^{\star})$ this gives $\mathrm{Var}(v_k^{\top}\hat{\theta}) \geq (n\lambda_k)^{-1}$; the corresponding 1σ error bar is therefore $\Delta\theta_k \simeq (n\lambda_k)^{-1/2}$.

Note: By default, `fisher` averages over the model-predicted class probabilities (i.e. it sums over p_k) instead of using the observed labels. In our experiments this choice made no difference to the comparative results in Sec. 12.3.

Despite approximations, the conceptual link remains: the **geometry** induced by $F(\theta)$ characterizes which parameter directions the model can "usefully" distinguish given n samples.

12.2 The Effective Dimension

The Fisher analysis tells us *which* directions are informative when we probe an $\varepsilon_n = O(n^{-1/2})$ neighbourhood of $\theta^\star$, but not yet *how many* such directions are visible *at that resolution*. Because ε_n shrinks with n, the very notion of "how many" must itself be *scale dependent*. The construction below follows [21, 22].

First, fix the Fisher–Rao metric $F(\theta)$ built from the joint distribution $p(x, y; \theta)$. With n samples the smallest detectable displacement has squared length $\varepsilon_n^2 = 1/n$, so a natural way to measure model size is to count how many **Fisher cubes** of side ε_n are required to cover Θ. Instead of manipulating tiny cubes, one can **dilate** the metric by the factor

$$\kappa_n = \frac{n}{2\pi \log n}.$$

Multiplying $F(\theta)$ by κ_n magnifies all curvatures until a step of size ε_n in the original metric becomes a unit step in the dilated one. Directions whose eigenvalues satisfy $\lambda_k \gg 2\pi/n$ are blown up to order-one curvature; those with $\lambda_k \ll 2\pi/n$ remain almost flat and contribute negligibly. In this way the metric rescaling turns the qualitative visibility rule $\lambda_k \gtrsim n^{-1}$ into a sharp geometric cutoff.

The number of distinguishable parameter points is now approximated by the κ_n**-Jeffreys volume**

$$\mathrm{Vol}_{\kappa_n}(\Theta) = \int_\Theta \sqrt{\det\big(\kappa_n F(\theta)\big)}\, d\theta = \kappa_n^{d/2} \int_\Theta \sqrt{\det F(\theta)}\, d\theta.$$

Taking the **logarithm** of the volume $\log \mathrm{Vol}_{\kappa_n}$ breaks the product of eigenvalues inside the determinant into a sum, so that each direction contributes **additively** once it passes the visibility threshold.

A raw volume still depends on the overall scale of F. To remove that trivial freedom we **normalise** F by its average trace,

$$\hat{F}(\theta) = \frac{d\, F(\theta)}{\int_\Theta \mathrm{tr}\, F(\vartheta)\, d\vartheta}, \qquad \frac{1}{V_\Theta} \int_\Theta \mathrm{tr}\, \hat{F}(\theta)\, d\theta = d,$$

and define the **global effective dimension**

$$d_{\text{eff}}(n) = \frac{2}{\log \kappa_n} \log\Big(V_\Theta^{-1} \int_\Theta \sqrt{\det\big(\mathbf{I}_d + \kappa_n \hat{F}(\theta)\big)}\, d\theta \Big).$$

Because $\det(\mathbf{I}_d + \kappa_n \bar{F}) = \prod_{i=1}^{d} (1 + \kappa_n \lambda_i)$, each eigenvalue contributes $\log(1+\kappa_n\lambda_i)/\log\kappa_n$. This quantity is close to 1 when $\lambda_i \gg 2\pi/n$ and is strongly suppressed when $\lambda_i \ll 2\pi/n$; $d_{\text{eff}}(n)$ therefore counts, to excellent accuracy, *the number of Fisher directions whose curvature exceeds* $2\pi/n$ [21]. As $n \to \infty$ the threshold falls to zero and $d_{\text{eff}}(n) \to d$; for finite n it can be dramatically smaller whenever the Fisher spectrum contains many small eigenvalues.

During training the optimiser explores only a small region of parameter space. To probe the dimensionality *seen* by the trained model, restrict the integral to the ε-ball $B_\varepsilon(\theta^\star)$ around the learned parameter $\theta^\star$, with $\varepsilon = 1/\sqrt{n}$, and renormalise the trace inside that ball[22]. The resulting **local effective dimension**

$$d_{\text{eff}}^{\text{local}}(n; \theta^\star) = \frac{2}{\log \kappa_n} \log\Big(V_\varepsilon^{-1} \int_{B_\varepsilon(\theta^\star)} \sqrt{\det\big(\mathbf{I}_d + \kappa_n \bar{F}(\theta)\big)}\, d\theta \Big)$$

drops whenever optimisation collapses flat valleys and has been observed to track **generalisation error** across a wide range of classical and quantum architectures [4].

As n increases, Fisher eigenvalues cross the $2\pi/n$ visibility line one by one. Each crossing raises d_{eff} by nearly a unit, producing a staircase-like growth that records how additional data unveil new parameter directions. In **practice**, even though approximations are required, the ordering of d_{eff} is preserved and suffice for the capacity comparisons reported in Sec. 12.3.

Just as **Torch-Q** offers a function to calculate the empirical Fisher matrix (i.e. `fisher`), there is also the `eff_dim` function to calculate the global effective dimension given a grid of dataset sizes.

```
from torchq.func import fisher, fisher_norm, eff_dim

n_values = np.array(
    [1000, 2000, 8000, 10000, 40000, 60000, 100000, 150000,
   200000, 500000, 1000000]
)
X_batch = torch.randn(100, 4)

fishers = fisher(model, X_batch, num_thetas=100)
fhat, _ = fisher_norm(fishers)
effdims = eff_dim(fhat, n_values)
effdims_normalised = effdims / model_dim   # divide by d
```

The **call sequence** mirrors the theory: first estimate the empirical Fisher, then normalise its trace to d using `fisher_norm`, and finally feed the batch of normalised Fishers to `eff_dim`, which returns $d_{\text{eff}}(n)$ for every sample size on the grid. Dividing by the parameter count places the result in $(0, 1]$

12.3 Results and Capacity Analysis

Having established *how* to compute the empirical Fisher matrix and the scale-dependent effective dimension, we now compare two width-matched models ($d = 28$) on randomly generated data:

- **Classical network (FFN)** – a feed-forward ReLU neural network with (4 input, 4 hidden, 3 output) units.
- **Quantum network (QNN)** – a four-qubit variational circuit that encodes $x \in \mathbb{R}^4$ with a standard `ZFeatureMap` and applies **six** layers of single-qubit rotations interleaved with CNOTs (i.e. `RealAmplitudes` variational form).

For both architectures we draw $T = 100$ **random parameter initialisations**, evaluate at each of them the empirical Fisher on a batch of $B = 100$ i.i.d. inputs $x \sim \mathcal{N}(0, I_4)$, rescale the resulting matrix to trace d, diagonalise it, and pool the $T \times d$ eigenvalues into a single **spectrum**.

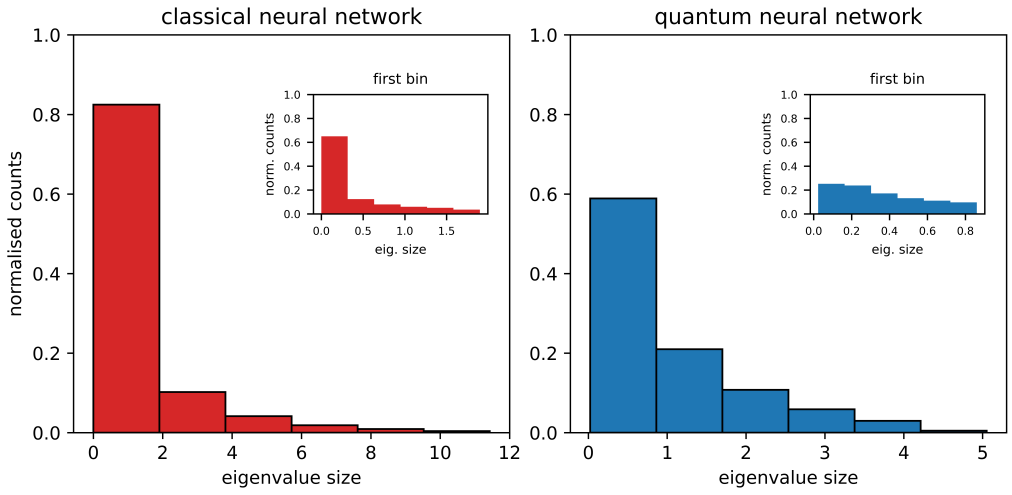


Figure 11: Normalised empirical Fisher spectra of the FFN (red) and QNN (blue), each binned into six equal-width intervals. Insets magnify the smallest-eigenvalue bin.

Figure 11 displays histograms of the normalised spectra. In accordance with the qualitative discussion of Sec. 3.1, the FFN spectrum (red) is **sharply peaked** near zero, implying many *flat* directions, whereas the QNN spectrum (blue) is far more **uniform**, indicating that a larger fraction of directions are already *informative* at random initialisation.

To translate these spectral differences into a single capacity number we evaluate the **global effective dimension**

$$d_{\mathrm{eff}}(n) = \frac{2}{\ln \kappa_n} \ln\!\Big(T^{-1} \sum_{t=1}^{T} \sqrt{\det\big(I + \kappa_n \, \widehat{F}_{\mathrm{norm}}(\theta^{(t)})\big)} \Big), \qquad \kappa_n = \frac{n}{2\pi \log n},$$

for sample sizes $n \in \{10^3, 2\times10^3, 8\times10^3, 10^4, 4\times10^4, 6\times10^4, 10^5, 1.5\times10^5, 2\times10^5, 5\times10^5, 10^6\}$, and normalise by d.

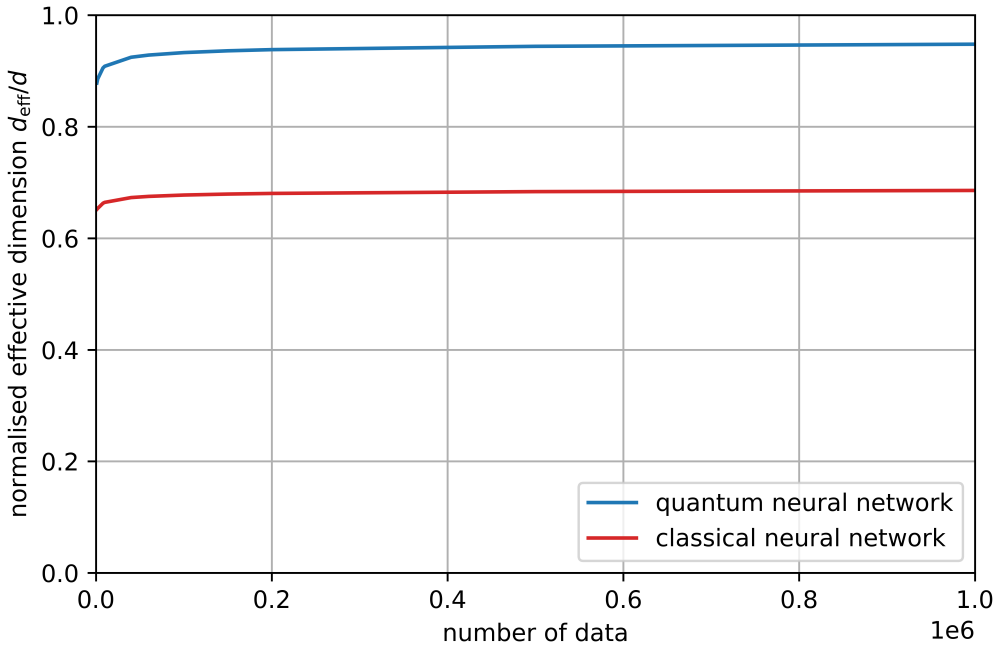


Figure 12: Normalised effective dimension d_{eff}/d as a function of sample size n (log scale) for the FFN (red) and the QNN (blue).

The curves in Fig. 12 show that the QNN exploits almost 90% of its parameters with only 10^3 data points and climbs to $\sim 95\%$ by 10^6, whereas the FFN starts around 65% and stagnates below 70%. Since d_{eff} increases whenever an eigenvalue crosses

the visibility line $2\pi/n$, this staircase behaviour directly visualises how additional data reveal new parameter directions in the QNN but leave most FFN directions unresolved.

At any fixed n we therefore find

$$d_{\text{eff}}^{\text{QNN}}(n) \gg d_{\text{eff}}^{\text{FFN}}(n),$$

so that, for the **same parameter budget**, the QNN can realise a strictly larger repertoire of distinguishable functions. Geometrically, the QNN spreads its Fisher curvature far more evenly, activating nearly the whole of its d-dimensional manifold, whereas the FFN leaves many directions trapped in data-invisible valleys. This yields a clear **capacity advantage**. Matching raw parameter counts hides substantial geometric differences, and the QNN makes almost the entire parameter space available to the data. Consequently, because generalisation bounds depending on $d_{\text{eff}}(n)$ tighten only when new directions become informative, the QNN should achieve a target error with markedly fewer training examples than the FFN (i.e. **faster learning**) [4].

13 Trainability

As shown in Section 12, QNNs hold an expressive power unmatched by size-matched classical networks. To test whether this advantage carries over to practice, we **benchmark** quantum neural networks directly against parameter-matched classical models on standard learning tasks. This head-to-head comparison reveals whether QNNs' theoretical gains yield tangible **performance improvements**.

13.1 Benchmarks

We begin with **Fisher's iris flowers dataset** [23]. Each bloom provides four real-valued measurements and must be assigned to one of three species. The **quantum model** encodes those four inputs with a single-layered `ZFeatureMap` and then applies nine variational layers of `EfficientSU2`. The companion **classical model** is a single-hidden-layer neural network whose width is chosen so that it carries *practically* the same number of trainable parameters. Apart from that everything is shared: mini-batches of eight, `Adam` with learning rate 0.1, twenty epochs over 150 samples, and weight decay 10^{-2}. The complete script lives in the `examples` folder of the official **Torch-Q repository**, so every experiment (including Sec. 12.3) can be reproduced.

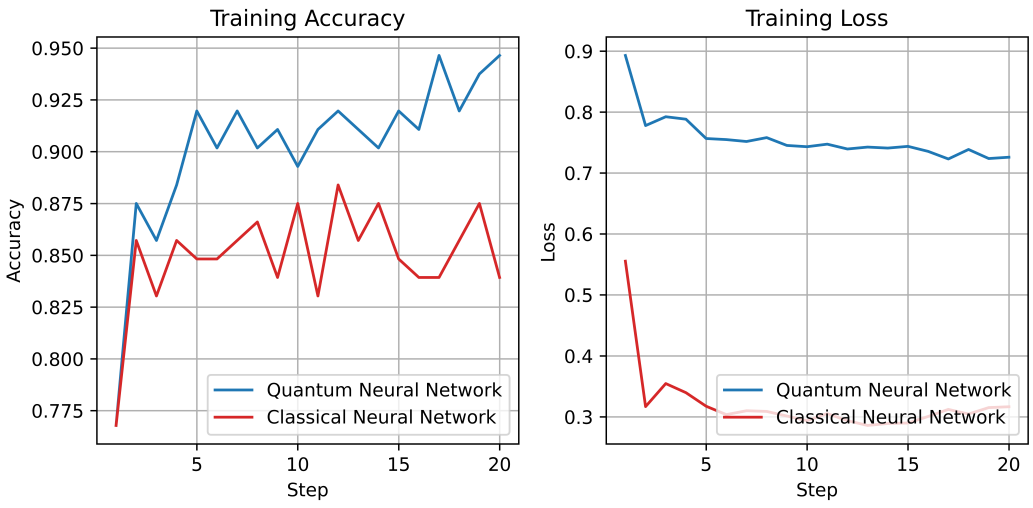


Figure 13: Iris training curves. Accuracy on the left, cross-entropy loss on the right. Quantum neural network in blue, classical neural network in red.

Figure 13 shows that the **quantum classifier** jumps above $90\,\%$ **accuracy** after only five optimization steps, sails through to a final **training** accuracy of 0.95, and, on

the held-out **test split**, predicts every single iris correctly. The classical network ends up with 0.84 training accuracy, yet still achieves a respectable 0.90 on the test set. The smoother blue trajectory hints at a gentler, *plateau*-like **loss landscape** for the quantum circuit; an observation we shall revisit.

Encouraged by that outcome we next raise the stakes without touching any hyper-parameter except circuit depth. First comes the **Wine chemistry dataset** [24], whose 178 rows each contain *thirteen continuous* features which must be mapped to one of *three cultivars*. The circuit now has *twelve layers* of `EfficientSU2`, and, after the usual twenty epochs, settles on a test accuracy of 0.88. Finally we turn to *computer vision*. The original 8×8 **hand-written digit images** [25] are down-sampled and flattened to nine inputs. To keep the training duration reasonable, we subsample $N = 500$ instances and restrict the task to digits 0–4. Even with *twelve variational layers* the QNN now reaches a more modest 0.74 test accuracy.

Table 1: Quantum neural network – final accuracies (20 epochs).

Data set	Samples	Inputs	Classes	Layers	Train acc.	Test acc.
Iris	150	4	3	9	0.95	1.00
Wine	178	13	3	12	0.91	0.88
Digits†	500	9	5	12	0.80	0.74

†Centre-crop to 6×6, 2×2 average pool, digits 0–4 only.

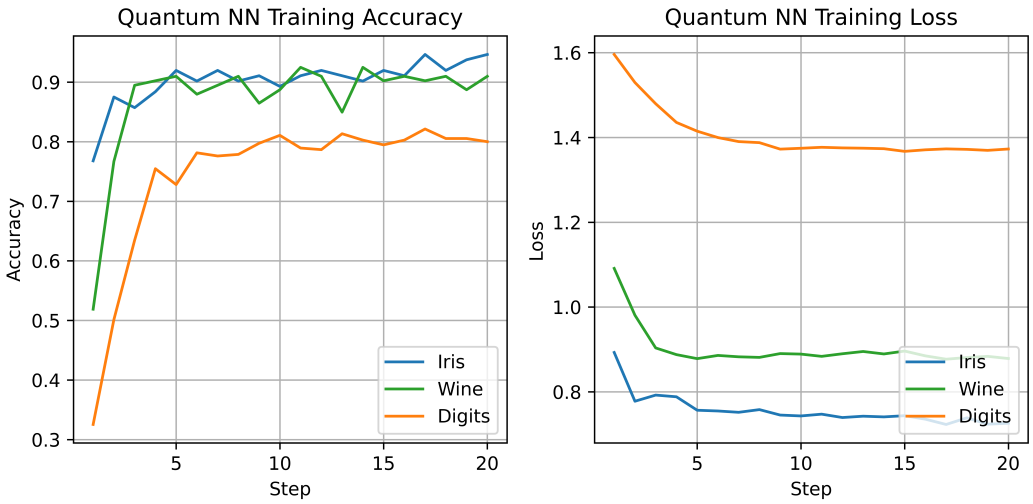


Figure 14: Quantum classifier trained on three data sets. Left: accuracy; right: loss. Nine layers for Iris, twelve for Wine and Digits.

Table 1 summarizes the journey. The quantum model sails smoothly through Iris, copes well with Wine, but begins to struggle once the Dimensionality-and-Class counter ticks up on Digits. Larger circuits could in principle claw back performance, yet deeper layers also amplify gradient decay, and in quantum networks that decay is exacerbated. In other words, training is not merely getting harder for the usual statistical reasons; it is running into genuinely **quantum obstacles**, the subject of the next subsection.

It is also important to note that, beyond the statistical and quantum-physical challenges, QNNs incur a substantial **computational overhead** in training. Recall that gradient evaluation via the *parameter–shift rule* requires two full circuit executions per trainable parameter for each optimization step. Instead, a classical network's backpropagation typically costs roughly one forward pass plus one backward pass (i.e. about twice a single inference). In practice this means that, even when matching parameter counts, the wall-clock time to train a QNN is currently *orders of magnitude* longer than its classical counterpart, although emerging quantum-compatible backward-pass methods (cf. Section 10) may in **future** help mitigate this overhead.

13.2 Barren Plateaus

Despite the remarkable expressivity highlighted in Sec. 12.3, numerical experiments in Sec. 13.1 consistently show that training quantum neural networks becomes substantially harder as the number of qubits and variational layers increases. Models that exhibit high effective dimension can still perform no better than random guessing, especially on problems that demand deeper or more entangled circuits. This performance decay, while superficially similar to classical overfitting or *vanishing gradients*, stems from a uniquely quantum pathology: the **barren plateau** (Fig. 15).

Barren plateaus are regions of the **loss landscape** where the gradient vanishes exponentially with qubit count. More precisely, for a parameterized quantum model $\theta \mapsto C(\theta)$, a probabilistic barren plateau arises when

$$\mathrm{Var}\left[\partial_\mu C(\theta)\right] = \mathcal{O}(b^{-n}) \quad \text{for all } \mu, \quad \text{with } b > 1,$$

so that the gradient becomes indistinguishable from **noise** beyond a modest number of qubits. When the gradient *itself* decays pointwise, i.e. $|\partial_\mu C(\theta)| = \mathcal{O}(b^{-n})$, one speaks of a deterministic barren plateau. Both effects render first-order optimization essentially inoperative. Crucially, this exponential decay reflects a fundamental **geometric suppression of signal**.

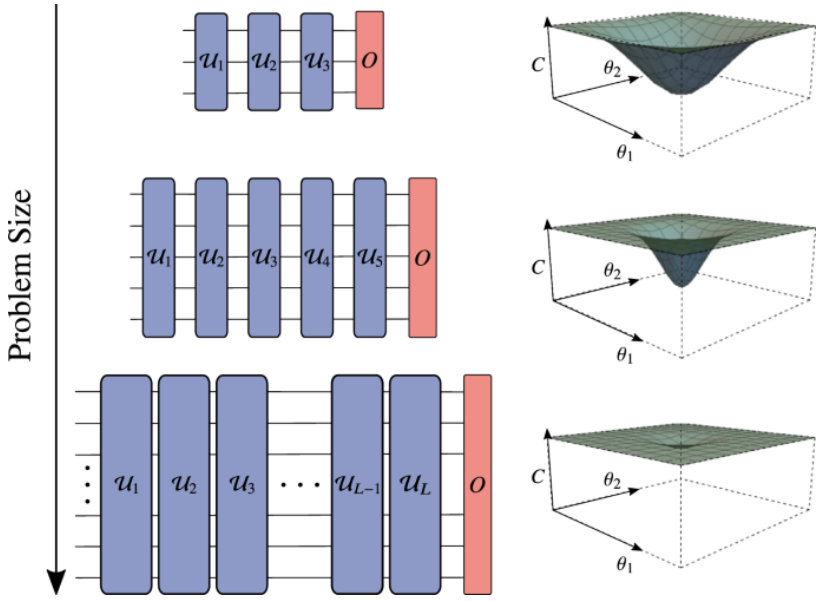


Figure 15: In the presence of local noise, gradients vanish exponentially in depth of the variational form, leading to **barren plateaus** [26]

.

The work in [27] identifies the following main mechanisms as contributing factors:

- In **high-depth circuits** with randomly chosen parameters, the distribution over unitaries approaches a 2-design. This leads to strong *concentration of measure*: expectation values and gradients converge to the ensemble mean, which for traceless observables vanishes. More precisely, the adjoint orbit of the observable O under such unitaries fills operator space so evenly that almost every parameter direction becomes equally (un)informative. In this case, $\mathrm{Var}[\partial_\mu C] \sim 1/2^n$, and gradients effectively disappear.

- **Entangling feature maps**, especially those involving data reuploading, also contribute. When the encoding circuit spreads classical inputs across highly entangled states, many marginal subsystems become maximally mixed. Subsequent variational layers cannot easily recover global distinguishability. In such regimes, even if the effective dimension remains high, the signal cannot flow from input to output in a learnable way.

- **Noise**, too, plays a critical role. Consider a variational circuit interleaved with unital channels such as depolarising noise. Since each such channel pushes the state toward the maximally mixed state, the composition rapidly flattens the cost landscape. In the extreme limit, all gradients vanish and the cost becomes

constant. This form of noise-induced barren plateau is particularly problematic for near-term devices, where circuit depth and coherence time are both constrained.

- Even in the absence of decoherence, the process of estimating gradients via finite sampling introduces its own scale: for a circuit with n qubits and N shots, the variance of gradient estimators behaves as $O(1/\sqrt{N})$, while the signal decays as $O(2^{-n})$. This sets a practical bound beyond which optimization becomes impossible due to **shot noise** dominating the signal.

Importantly, **barren plateaus and effective dimension** measure opposite facets of model geometry. While the Fisher spectrum in Sec. 12.1 quantifies *how many directions are informative* at a given scale, barren plateaus indicate that *no direction is learnable* under feasible resolutions. This is **not a contradiction**: *a model can have full capacity and still be untrainable* if all informative directions are exponentially hard to locate. This distinction is observable in **benchmark datasets** 13.1. On low-dimensional problems like Iris, models marvelously. But as the dimensionality and number of required qubits increases (e.g. the Wine or Digits datasets), QNN performance deteriorates faster than that of their classical counterparts.

Understanding and mitigating barren plateaus is thus critical to unleashing the full potential of quantum neural networks. A wide array of architectural and algorithmic **solutions** have been proposed, which we review in the section.

13.3 Optimization Strategies

Understanding and mitigating barren plateaus is not merely a matter of picking a “better” optimizer; it calls for an *orchestrated* combination of architectural choices, cost-function design, and geometry-aware updates. We summarize below the most prominent ideas, progressing from circuit-level design to large-scale optimization heuristics; citations point to the detailed derivations in [27, 28].

Layer-wise training. Freezing deep parameters and optimizing successive "front layers" avoids solving the full high-dimensional problem at once; additional layers are thawed only after shallower blocks have converged.

Spectrally controlled initializations. Naïve i.i.d. random seeds often place the circuit near a maximally mixed region, collapsing gradients at step 0. Even lightweight fixes (*e.g.* the recent β-initialization [29]) already improve models’ accuracies. More

principled, spectrally controlled schemes such as identity-plus-noise or Gaussian kernels of width $\sigma \sim 1/\sqrt{n}$ keep the initial Fisher spectrum finite. By ensuring no eigenvalue falls below 2^{-n} at epoch 0, these prescriptions postpone the onset of plateaus and stabilize early training.

Natural-gradient philosophy (NGD and QNG). Section 12.1 showed that the classical **Fisher information** $F(\theta)$ provides a Riemannian metric on parameter space. Natural Gradient Descent rescales the raw gradient by $F^{-1}(\theta)$,

$$\Delta\theta = -\,\eta\, F(\theta)^{-1}\nabla_\theta \mathcal{L},$$

so that each step corresponds to a unit *Fisher distance*. Because F^{-1} amplifies directions whose curvature is exponentially small, NGD effectively flattens the landscape, turning many barren plateaus into gently sloping valleys that standard SGD (and its derivatives e.g. Adam) could not traverse.

When the loss depends explicitly on the quantum state, the relevant geometry is no longer Fisher–Rao but the **Fubini–Study metric**, whose matrix representation is the real part of the Quantum Geometric Tensor $G(\theta)$. The *Quantum Natural Gradient* simply upgrades NGD by replacing F with G:

$$\Delta\theta = -\,\eta\, G(\theta)^{+}\nabla_\theta \mathcal{L},$$

where G^{+} denotes the Moore–Penrose pseudoinverse. A block-diagonal approximation to G can be estimated with only $2d$ additional circuit evaluations and has been shown to outperform Adam and other state of the art optimizers across depths up to $L = 6$ and $n = 11$ qubits [30]. Note that G was *not* required in the capacity analysis of Chap. 12, where **output distinguishability** is a **classical statistical notion** already captured by F; for optimization, however, the search unfolds on the manifold of pure states, making G indispensable.

Barren-plateau research is fast-moving and far from settled. Many of the remedies above act more like *vitamins* (alleviating barren plateau symptoms for particular architectures or depths) than like a universal cure. A complete resolution will likely demand deeper insight into quantum loss landscapes, noise-resilient metrics, and adaptive protocols that blend classical and quantum resources. Until then, the strategies surveyed here offer practical relief and a roadmap for **future exploration**.

CONCLUSION

In this book we framed the entire enterprise as a response to a tension: on the one hand the astonishing **empirical success of deep learning**, on the other its unsustainable **appetite for energy and compute**. Over four parts we have watched how quantum mechanics (II) and quantum computing (III) transform that tension into an opportunity by offering physical resources (e.g. superposition, entanglement, interference) that classical processors cannot emulate in polynomial time. When those resources are organized into quantum neural networks (IV), they generate hypothesis spaces whose effective dimension already saturates the parameter count with only modest data, whereas a width-matched classical network leaves many directions buried in extremely low curvature. In small-sample regimes that added geometric volume translates into learning curves that bend sooner and reach higher accuracies; tangible evidence that quantum models can in practice outpace classical baselines in both **expressivity and sample efficiency**.

That said, the very quantum rules that unlock this capacity simultaneously impose **two sharp handicaps**. First, gradients are typically obtained through the parameter-shift rule (10.2); each optimization step therefore demands two fresh circuit evaluations per parameter and inherits shot noise that decays only as $1/\sqrt{\text{shots}}$. Second, even when we are willing to pay that price the landscape itself may flatten into a *barren plateau* (13.2), a region where every partial derivative becomes exponentially small in the number of qubits, so training stalls for lack of measurable signal. These twin obstacles, **expensive and vanishing gradients**, define the present boundary of quantum-enhanced deep learning. They neither negate the capacity results nor doom quantum models; they simply mark where engineering ingenuity and theoretical creativity must next concentrate.

Several lines of attack have already appeared. On the gradient-estimation front, *shadow-tomography* overlaps and the more speculative *quantum memory constructions* promise to amortize backward-pass cost without forfeiting unbiasedness. Deeper understanding of shadow protocols and advances toward practical quantum memories may push this route much further. For the vanishing-gradient problem, natural-gradient and, more specifically, *quantum natural gradient* updates pre-condition steps to respect the underlying state geometry. Furthermore, *layer-wise training* and *spectrally balanced initializations* tame barren plateaus by steering the circuit through regions whose curvature remains resolvable. These methods, though still young, sketch a coherent **interdisciplinary research programme** that links algorithmic design, geometric theory and hardware development.

V CONCLUSION

Our discussion has touched *only* a subset of the architectures now under exploration: kernel-inspired circuits, tensor-network ansätze, observable-based outputs, error-mitigated noise models and hybrid classical heads all beckon. The **CUDA-Q and Torch-Q toolkit** introduced in III and IV makes those variants **accessible**: swap a feature map, alter an entangling graph, review and modify the source code, try a different gradient estimator and then benchmark under the metrics that matter (e.g. accuracy, wall-clock time, joules consumed). Doing so will require the concerted effort of physicists probing coherence budgets, computer scientists refining compilers, mathematicians extending capacity theory and domain experts posing tasks that stretch today's prototypes.

The **purpose** set out at the start was *to educate so that collaboration could flourish*. The reader who now commands the algebra of qubits, the calculus of effective dimension and the pragmatics of running kernels on simulators or early QPUs is well placed to contribute. **Reproduce** our experiments on data that matter to you; **publish** both successes and failures; **open-source** your kernels so others can extend them. The tools are free, the problems open-ended and the environmental and intellectual stakes high. May the concepts, code and experiments collected here serve not as an endpoint but as a springboard toward the next generation of **quantum enhanced deep learning architectures**.

References

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

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: https://books.google.es/books?id=STDBswEACAAJ.

[3] El País. *AI will consume as much energy in the U.S. as all of Spain*. Language: Spanish. Accessed: 2025-06-01. Jan. 2025. URL: https://elpais.com/tecnologia/2025-01-23/la-ia-consumira-en-ee-uu-tanta-energia-como-toda-espana.html.

[4] Amira Abbas et al. "The power of quantum neural networks". In: *arXiv preprint arXiv:2011.00027* (Oct. 2020). DOI: 10.1038/s43588-021-00084-1.

[5] PyTorch Contributors. *PyTorch documentation — PyTorch 2.7 documentation*. Accessed: 2025-06-01. 2025. URL: https://docs.pytorch.org/docs/stable/index.html.

[6] David J. Griffiths and Darrell F. Schroeter. *Introduction to Quantum Mechanics*. 3rd. Cambridge University Press, 2018. DOI: 10.1017/9781316995433.

[7] R. Shankar. *Principles of Quantum Mechanics*. 2nd ed. Springer New York, NY, 1994. DOI: 10.1007/978-1-4757-0576-8.

[8] Maria Schuld and Francesco Petruccione. *Supervised Learning with Quantum Computers*. Quantum Science and Technology. Springer Cham, 2018. DOI: 10.1007/978-3-319-96424-9.

[9] NVIDIA Corporation & Affiliates. *CUDA Quantum (CUDA-Q) Documentation*. Accessed: 2025-06-01. 2025. URL: https://nvidia.github.io/cuda-quantum/latest/index.html.

[10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Anniversary. Cambridge University Press, Jan. 2011. DOI: 10.1017/CBO9780511976667.

[11] Sergio Rodríguez Vidal. *torch-quantum: Quantum-ready layers for PyTorch*. 2025. URL: https://github.com/SeroviICAI/torch-quantum.

[12] Xanadu Quantum Technologies. *Quantum Machine Learning*. Accessed: 2025-06-01. 2025. URL: https://pennylane.ai/qml/quantum-machine-learning.

[13] Vojtěch Havlíček et al. "Supervised learning with quantum enhanced feature spaces". In: *arXiv preprint arXiv:1804.11326* (Apr. 2018). DOI: 10.1038/s41586-019-0980-2.

[14] IBM Corporation. *IBM Quantum Documentation: Guides*. Accessed: 2025-06-01. 2025. URL: https://quantum.cloud.ibm.com/docs/es/guides.

[15] Yoshifumi Nakata et al. "Unitary t-designs from random X- and Z-diagonal unitaries". In: *Journal of Mathematical Physics* (2017). arXiv:1502.07514 [quant-ph]. DOI: 10.1063/1.4983266.

[16] Amira Abbas et al. "On quantum backpropagation, information reuse, and cheating measurement collapse". In: *arXiv preprint arXiv:2305.13362* (May 2023). DOI: 10.48550/arXiv.2305.13362.

[17] Adrian Perez-Salinas et al. "Data re-uploading for a universal quantum classifier". In: *Quantum* 4 (2020). arXiv:1907.02085 [quant-ph]. DOI: 10.22331/q-2020-02-06-226.

[18] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. 2nd. Wiley/Interscience, Apr. 2005. DOI: 10.1002/047174882X.

[19] Tengyuan Liang et al. "Fisher-Rao Metric, Geometry, and Complexity of Neural Networks". In: *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*. arXiv:1711.01530 [cs.LG]. 2019. DOI: 10.48550/arXiv.1711.01530.

[20] J. J. Rissanen. "Fisher information and stochastic complexity". In: *IEEE Transactions on Information Theory* 42.1 (1996). DOI: 10.1109/18.481776.

[21] Oksana Berezniuk et al. "A scale-dependent notion of effective dimension". In: *arXiv preprint* (2020). arXiv:2001.10872 [stat.ML]. DOI: 10.48550/arXiv.2001.10872.

[22] Amira Abbas et al. "Effective dimension of machine learning models". In: *arXiv preprint* (2021). arXiv:2112.04807 [cs.LG]. DOI: 10.48550/arXiv.2112.04807.

[23] R. Fisher. *Iris [Dataset]*. 1936. DOI: 10.24432/C56C76. URL: https://archive.ics.uci.edu/dataset/53/iris.

[24] S. Aeberhard and M. Forina. *Wine [Dataset]*. 1992. DOI: 10.24432/C5PC7J. URL: https://archive.ics.uci.edu/dataset/109/wine.

[25] E. Alpaydin and C. Kaynak. *Optical Recognition of Handwritten Digits [Dataset]*. 1998. DOI: 10.24432/C50P49. URL: https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digits.

[26] Samson Wang et al. "Noise-induced barren plateaus in variational quantum algorithms". In: *Nature Communications* 12 (2021), p. 6961. DOI: 10.1038/s41467-021-27045-6.

[27] Martin Larocca et al. “Barren plateaus in variational quantum computing”. In: *Nature Reviews Physics* 7 (2025). arXiv:2405.00781 [quant-ph], pp. 174–189. DOI: 10.1038/s42254-025-00813-9.

[28] Amira Abbas et al. “Challenges and opportunities in quantum optimization”. In: *Nature Reviews Physics* 6.12 (2024). DOI: 10.1038/s42254-024-00770-9.

[29] Sabrina Herbst, Vincenzo De Maio, and Ivona Brandic. “On Optimizing Hyperparameters for Quantum Neural Networks”. In: *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2024. DOI: 10.1109/QCE60285.2024.00174.

[30] James Stokes et al. “Quantum Natural Gradient”. In: *Quantum* 4 (2020). arXiv:1909.02108 [quant-ph]. DOI: 10.22331/q-2020-05-25-269.