Efficient Redundancy-Free Computation in Large-Scale Problems via Quantum Implementation of Machine Learning Approaches

SooJean Han[1]*

August 5, 2022

*[1]SooJean Han is with the Department of Computing and Mathematical Sciences, California Institute of Technology, Pasadena, CA 91125, USA. Email: soojean@caltech.edu

Chapter 1

Basics of Quantum Computing

An introductory discussion of quantum computing.

1.1 Fundamentals

1.1.1 The Schrödinger Equation

Before going thoroughly into a discussion of quantum computers, it is important to recognize the basic principles of quantum mechanics upon which they are founded on.

We begin from *Schrödinger's equation*, which governs the behavior of a quantum-mechanical system in the following way

$$i\hbar\frac{\partial}{\partial t}\Psi(t,x) = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\Psi(t,x) + V(t,x)\Psi(t,x) \tag{1.1}$$

Here, $i = \sqrt{-1}$, $\hbar = h/2\pi$ with $h = 6.62607015 * 10^{`34}$ J/Hz the Planck constant, and $V(t,x)$ is the *potential* that describes the environment in which the quantum system exists, often in terms of energy. The solution $\Psi(t,x)$ is a *wave function*, i.e., a mathematical function expressing the quantum state of the quantum system.

In cases where $\Psi(t,x) := \phi(t)\psi(x)$ is a separable function, the general (1.1) simplifies as follows:

$$i\hbar\frac{1}{\phi(t)}\frac{d}{dt}\phi(t) = -\frac{\hbar^2}{2m}\frac{1}{\psi(x)}\frac{d^2}{dx^2}\psi(x) + V(t,x) \tag{1.2}$$

The left side of (1.2) is an evolution entirely over time t, whereas the right side is an evolution entirely over space x. Since time and space are inherently independent axes, the only way for (1.2) to hold is if both sides were equal to a constant. We refer the constant to be E, the *total energy* of the quantum system.

We can explicitly solve the left side of (1.2) by separation of variables:

$$i\hbar\frac{1}{\phi(t)}\frac{d}{dt}\phi(t) = E \implies i\hbar\frac{d\phi(t)}{\phi(t)} = Edt \implies \phi(t) = Ce^{-\frac{E}{\hbar}t}$$

The solution to the right side of (1.2), often called the *time-independent Schrödinger equation*, is contingent upon simplifying assumptions about $V(t,x)$. In particular, one well-known form of $V(t,x)$ is the *infinite square well*, where $V(t,x) \equiv V(x)$ is 0 on some interval $[0,b]$ and ∞ outside of it. Physically, the infinite square well can be used to describe the energy of a particle which is bound within a box. In the interval $x \in [0,b]$, (1.2) simplifies as follows

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x) \implies \frac{d^2\psi(x)}{dx^2} = k^2\psi(x)$$

where $k = \sqrt{2mE}/\hbar$. The solution to the simple second-order ODE is $\psi(x) = A\sin(kx) + B\cos(kx)$. Now, the coefficients A and B can be determined via the boundary conditions prescribed by the infinite square well. First, $\psi(0) = 0$ implies that $B = 0$. Second, $\psi(b) = 0$ implies that $A\sin(kb) = 0$, which implies that $kb = n\pi$ for some $n \in \mathbb{N}$. Expanding this equality:

$$\frac{\sqrt{2mE}}{\hbar}b = n\pi \implies E_n = \frac{n^2\pi^2\hbar^2}{2mb^2}$$

where we assign a variable E_n to each level of the total energy E corresponding to $n \in \mathbb{N}$. This phenomenon is referred to as *energy quantization*. In particular, note that the lowest possible level of energy E_1 is nonzero; it is referred to as the *ground-state energy*.

Putting the two pieces together, the solution to the separable Schrödinger equation (1.2) in an infinite-square well potential on $[0,b]$ yields a discrete set of wave functions

$$\Psi_n(t,x) = Ke^{-\frac{E_n}{\hbar}t}\sin\left(\frac{n\pi x}{b}\right), n \in \mathbb{N}$$

for some constant $K \in \mathbb{R}$. This constant can be determined by normalizing the wave function, which is a condition that arises due to *Born's interpretation*: the probability of being able to find a particle located in some space $x \in [a,c]$ is

$$P = \int_a^c |\Psi(t,x)|^2 dx$$

This gives rise to the *normalization of the wave function* condition:

$$\int_{-\infty}^{\infty} |\Psi(t,x)|^2 dx = 1 \tag{1.3}$$

In particular, inside the infinite square well, (1.3) is satisfied with the limits $(-\infty,\infty)$ replaced with $[0,b]$. After some rote calculation:

$$\begin{aligned}
1 &= K^2\int_0^b e^{-\frac{E_n}{\hbar}t}\sin\left(\frac{n\pi x}{b}\right)\cdot e^{\frac{E_n}{\hbar}t}\sin\left(\frac{n\pi x}{b}\right)dx \\
&= K^2\int_0^b \sin^2\left(\frac{n\pi x}{b}\right)dx \\
&= \frac{K^2}{2}\int_0^b \left(1-\cos\left(\frac{2n\pi x}{b}\right)\right)dx \\
&= \frac{bK^2}{2} - \frac{K^2}{2}\int_0^b \cos\left(\frac{2n\pi x}{b}\right) \\
&= \frac{bK^2}{2} - \frac{bK^2}{4\pi n}\int_0^{2\pi n}\cos(u)du = \frac{bK^2}{2}
\end{aligned}$$

Thus, $K = \sqrt{2/b}$ and

$$\Psi_n(t,x) = \sqrt{\frac{2}{b}}e^{-\frac{E_n}{\hbar}t}\sin\left(\frac{n\pi x}{b}\right), \quad E_n = \frac{n^2\pi^2\hbar^2}{2mb^2}, \quad n \in \mathbb{N} \tag{1.4}$$

Definition 1 (Hamiltonian)**.** The time-independent Schrödinger equation is often written in terms of the *Hamiltonian operator*:

$$H\psi(x) = E\psi(x), \qquad H := -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x) \tag{1.5}$$

In this format, it becomes apparent that $\psi(x)$ is an eigenfunction of H, with eigenvalue E.

In the interpretation of Definition 1, where each sharply-defined energy level E_n associated with each quantum state is considered an eigenvalue of the Hamiltonian operator H, we henceforth use λ_n instead of E_n to denote the levels of energy, so that $\lambda_n = (n^2\pi^2\hbar^2)/(2mb^2)$. Precisely, if the quantum system is described by the wave function $\Psi_n(t,x) = \psi_n(x)e^{-i\lambda_n t/\hbar}$, then it is said to be in quantum state n.

Note that each quantum state wave function $\Psi_n(t,x)$ is a particular solution to the separable Schrödinger equation (1.2) for the potential $V(x)$. But since (1.2) is linear in the wave function, we expect that any linear combination of wave functions will also be a solution.

$$\Psi(t,x) := \sum_{n=1}^{\infty} c_n \Psi_n(t,x) = \sum_{n=1}^{\infty} c_n \psi_n(x) e^{-i\lambda_n t/\hbar} \tag{1.6}$$

where $c_n \in \mathbb{C}$ are coefficients.

The quantum state defined by the wave function (1.6) is said to be a linear *superposition* of quantum states. From the normalization condition (1.3), we can determine that

$$\begin{aligned}
1 &= \int_{-\infty}^{\infty} |\Psi(t,x)|^2 dx = \int_{-\infty}^{\infty} \left| \sum_{n=1}^{\infty} c_n \Psi_n(t,x) \right| dx \\
&= \int_{-\infty}^{\infty} \left(\sum_{n=1}^{\infty} \bar{c}_n \Psi_n^\dagger(t,x) \right) \left(\sum_{n=1}^{\infty} c_n \Psi_n(t,x) \right) dx \\
&= \int_{-\infty}^{\infty} \sum_{n=1}^{\infty} |c_n|^2 |\Psi_n(t,x)|^2 dx + \int_{-\infty}^{\infty} \sum_{n=1}^{\infty} \sum_{m>n} \left(\bar{c}_n c_m \Psi_n^\dagger(t,x)\Psi_m(t,x) + c_n \bar{c}_m \Psi_n(t,x)\Psi_m^\dagger(t,x) \right) dx
\end{aligned} \tag{1.7}$$

Note that $\int_{-\infty}^{\infty} |\Psi_n(t,x)|^2 dx = 1$ and for any $n, m \in \mathbb{N}$ such that $n \neq m$, using the solution to (1.2) yields:

$$\begin{aligned}
\int_{-\infty}^{\infty} \Psi_n^\dagger(t,x)\Psi_m(t,x) dx &= \frac{2}{b} e^{\frac{t}{\hbar}(\lambda_n - \lambda_m)} \int_0^b \sin\left(\frac{n\pi x}{b}\right) \sin\left(\frac{m\pi x}{b}\right) dx \\
&= \frac{1}{b} e^{\frac{t}{\hbar}(\lambda_n - \lambda_m)} \int_0^b \left[\cos\left(\frac{(n-m)\pi x}{b}\right) + \cos\left(\frac{(n+m)\pi x}{b}\right) \right] = 0
\end{aligned}$$

And so, different wave functions corresponding to different levels of energy are *orthogonal* to each other. Substituting back in:

$$1 = (1.7) = \sum_{n=1}^{\infty} |c_n|^2 \tag{1.8}$$

We thus interpret $|c_n|^2$ to be the probability that the superposition quantum state is measured to be n.

1.1.2 The Hamiltonian Operator

We take a brief detour to discuss the distinctions between three common approaches to analyzing physical systems. The most elementary formulation is *Newtonian mechanics*, which considers the total forces acting upon a physical system through the well-known relationship $F = ma$; we do not discuss the details of Newtonian mechanics here.

For the Lagrangian and Hamiltonian formulations, consider the general physical system described by the coordinate system $(\mathbf{q}, \dot{\mathbf{q}})$ for its positions and velocities, respectively, where $\mathbf{q}, \dot{\mathbf{q}} \in \mathbb{R}^n$. *Lagrangian mechanics* considers the difference between kinetic (K) and potential (V) energies by defining the *Lagrangian* $\mathcal{L}(t, \mathbf{q}, \dot{\mathbf{q}}) = K - V$, and substituting into the *Euler-Lagrange equation*

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\mathbf{q}}}\right) = \frac{\partial \mathcal{L}}{\partial \mathbf{q}} \tag{1.9}$$

In contrast, *Hamiltonian mechanics* considers the total energy $E = K + V$ in terms of the momenta of the system, defined as $p_i(t, \mathbf{q}, \dot{\mathbf{q}}) = \partial \mathcal{L}/\partial \dot{q}_i$. It is easy to show that this definition of momenta simplifies to the better-known expression $p_i = m\dot{q}_i$ common in Newtonian mechanics. The Lagrangian mechanics defines the *energy function*

$$E_L(t, \mathbf{q}, \dot{\mathbf{q}}) := \sum_{i=1}^{n} \dot{q}_i \frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \mathcal{L}(t, \mathbf{q}, \dot{\mathbf{q}}) \tag{1.10}$$

The Hamiltonian $H(t, \mathbf{p}, \mathbf{q})$ is precisely defined by substituting p_i into $\partial \mathcal{L}/\partial \dot{q}_i$ in (1.10):

$$H(t, \mathbf{p}, \mathbf{q}) = \langle \mathbf{p}, \dot{\mathbf{q}} \rangle - \mathcal{L}(t, \mathbf{q}, \dot{\mathbf{q}}) \tag{1.11}$$

Effectively, we've transformed the original coordinate system $(\mathbf{q}, \dot{\mathbf{q}})$ into *phase-space coordinates* $(\mathbf{p}, \dot{\mathbf{q}})$. In these new phase-space coordinates, the n-dimensional Euler-Lagrange equation becomes the $2n$-dimensional *Hamilton's equations*:

$$\dot{\mathbf{q}} = \frac{d\mathbf{q}}{dt} = \frac{\partial H}{\partial \mathbf{p}}, \qquad \frac{d\mathbf{p}}{dt} = -\frac{\partial H}{\partial \mathbf{q}} \tag{1.12}$$

Hence, a second-order differential equation has become split into two first-order equations. It does not necessarily mean that they are easier to solve, but the coordinate transformation to phase-space is often easier to work with when considering quantum systems. Plots of a system in phase-space manifest as multiple curves such that every point $(\mathbf{p}, \dot{\mathbf{q}})$ on the same curve has the same total energy.

We further emphasize that the total energy E is *not equivalent* to the energy described by the energy function (1.10), which defined the Hamiltonian (1.11). This was illustrated through the energy quantization phenomenon observed before. Even Definition 1 describes the different levels of energy as quantities which arise as eigenvalues of the Hamiltonian operator, but they are not the total energy itself.

In the quantum setting, functions on the phase space correspond to operators on the space of quantum states. The time-evolution of the quantum system prescribed by the Hamiltonian operator H is written as $|\Psi\rangle \to e^{-(i/h)Ht}|\Psi\rangle$, where $|\Psi\rangle$ is the quantum state the system is currently in.

Definition 2 (Expectation of Operator). The *expectation of an operator O with respect to quantum state* $|\Psi\rangle$ is defined as

$$\langle O \rangle_{|\Psi\rangle} := \frac{\langle \Psi | O | \Psi \rangle}{\langle \Psi | \Psi \rangle} := \int_{-\infty}^{\infty} \Psi^\dagger(t, x) O \Psi(t, x) dx \tag{1.13}$$

where the simplification on the rightmost side of (1.13) is made with the normalization assumption $\langle \Psi | \Psi \rangle = 1$.

Definition 3 (Variance of Operator)**.** The *variance of an operator O with respect to quantum state $|\Psi\rangle$* is defined as

$$\mathrm{Var}(O)_{|\Psi\rangle} := \langle O^2\rangle_{|\Psi\rangle} - \langle O\rangle^2_{|\Psi\rangle} = \frac{\langle\Psi|O^2|\Psi\rangle}{\langle\Psi|\Psi\rangle} - \frac{\langle\Psi|O|\Psi\rangle}{\langle\Psi|\Psi\rangle}^2 \tag{1.14}$$

where the simplification on the rightmost side of (1.13) is made with the normalization assumption $\langle\Psi|\Psi\rangle = 1$.

Typically, when the total energy E is measured while the quantum system is in some quantum state $|\Psi\rangle$, there is an uncertainty associated with the measurement with variance $\mathrm{Var}(H)_{|\Psi\rangle}$, where H is the Hamiltonian of the quantum system. However, if we measure the quantum state $|\Psi_n\rangle$, where Ψ_n is the wave function (1.4) representing one of the discrete energy levels in the quantum system, we get:

$$\langle H\rangle_{|\Psi_n\rangle} := \int \Psi_n^\dagger(t,x) H \Psi_n(t,x)dx = \lambda_n \int \Psi_n^\dagger(t,x)\Psi_n(t,x)dx = \lambda_n \text{ by normalization}$$

because the time-invariant part $\psi_n(x)$ of $\Psi_n(t,x)$ is an eigenfunction of H in the sense of Definition 1, with eigenvalue λ_n. Moreover, $H^2\psi_n(x) = \lambda_n^2\psi_n(x)$ holds, and so

$$\langle H^2\rangle_{|\Psi_n\rangle} := \int \Psi_n^\dagger(t,x) H^2 \Psi_n(t,x)dx = \lambda_n^2 \int \Psi_n^\dagger(t,x)\Psi(t,x)dx = \lambda_n^2 \text{ by normalization}$$

Hence, $\mathrm{Var}(H)_{|\Psi_n\rangle} = 0$ for all $n \in \mathbb{N}$, and the result of an energy measurement is certain. Any eigenfunction of the Hamiltonian always describes the state of definite energy.

1.2 Inside a Quantum Computer

1.2.1 Basic Components

The distinctions between a quantum computer and the classical, deterministic computer begin fundamentally with the re-definition of the *bit*. On classical computers, a bit is typically distinguished by two *basic states*: 0 or 1. On quantum computers, we have instead the qubit, defined below.

Definition 4 (Qubit)**.** A *quantum bit (qubit)* is the analogous version of the bit in the context of quantum computation. Like the classical bit, a qubit can take one of two basic states $|0\rangle$ or $|1\rangle$, but it also abides by the following basic principles from quantum mechanics:

- A qubit can also be in a state $|\Psi\rangle$ which is a superposition of $|0\rangle$ and $|1\rangle$:

  $$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

  where $\alpha, \beta \in \mathbb{C}$ are *amplitudes* such that $|\alpha|^2 + |\beta|^2 = 1$.

- *Measuring* a qubit with state $|\Psi\rangle$ yields 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Taking a measurement of a qubit also physically changes its state to $|0\rangle$ or $|1\rangle$ depending on the observation.

Remark 1. Each qubit in a quantum computer can be thought of as an entity whose energy can be described by the wave function $\Psi_n(t,x)$ from (1.4), the solution of the separable Schrödinger equation. In contrast to (1.4), however, each qubit is restricted to have only two levels of energy, one of which correspond to the $|0\rangle$ basic state, and the other to $|1\rangle$.

Example 1 (Common Qubits)**.** There are some qubits, aside from $|0\rangle$ and $|1\rangle$ which commonly arise in the quantum computing literature. Below, we two such qubits in terms of their superpositions.

$$|+\rangle \triangleq \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}^\top \quad |-\rangle \triangleq \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}^\top \tag{1.15}$$

Note that a qubit in the $|+\rangle$ state will be measured as $|0\rangle$ for 1/2 of the time, and $|1\rangle$ for the other half of the time. This is likewise true for a qubit in the $|-\rangle$ state.

From a physical hardware perspective, a qubit can be implemented in various different ways. For example:

- the spin of an electron has two states: "up", which corresponds to basic state $|0\rangle$, and "down", which correpsonds to $|1\rangle$.
- the polarization of a photon: horizontal corresponds to $|0\rangle$, while vertical corresponds to $|1\rangle$.

Recall that the number of energy levels represented by the Schrödinger equation solutions can generally be more than two; in fact, in the general superposition state (1.6), the sum went up to ∞. Similarly, a qubit can also be extended to take more than two basic states, at which point it is referred to as a *qudit* instead. For some $d \in \mathbb{N}$, we can represent the state of a qudit as $|\Psi\rangle = \alpha_1|0\rangle + \alpha_2|1\rangle + \cdots + \alpha_d|d-1\rangle$, where $\sum_{i=1}^{d}|\alpha_i|^2 = 1$.

Remark 2 (Vector Representation of Qubits)**.** It is often convenient to represent the state of a qubit as a 2-dimensional vector of complex coefficients in $\mathbb{C}^2$, e.g., $|0\rangle = [1, 0]^\top$, $|1\rangle = [0, 1]^\top$, and the superposition state in Definition 4 is represented $|\Psi\rangle = [\alpha, \beta]^\top$. A qudit with $d \in \mathbb{N}$ basic states can similarly be represented with the vector $[\alpha_1, \alpha_2, \cdots, \alpha_d]^\top \in \mathbb{C}^d$. However, instead of extending the dimension of a single qubit to a qudit, it is more common to represent multiple states using multiple qubits, leading to the construction of *multi-qubit systems.* This construction is easy to see with the vector representation of quantum states, since *tensor products* can be leveraged, as shown in the following example with 2-qubits.

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \equiv |00\rangle, \quad |0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \equiv |01\rangle$$

$$|1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \equiv |10\rangle, \quad |1\rangle \otimes |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \equiv |11\rangle$$

Note that the $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ form the basic quantum states for a qudit with dimension $d = 4$. Such a qudit can also take on superposition states which are tensor product combinations of two qubit superposition states. For example:

$$|0\rangle \otimes |+\rangle = \begin{bmatrix}1 & 0\end{bmatrix}^\top \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}^\top = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \end{bmatrix}^\top \equiv \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle$$

One very common state which is also found in the literature is the *Bell state*, defined as $(1/\sqrt{2})|00\rangle + (1/\sqrt{2})|11\rangle$.

Definition 5 (Quantum Gates)**.** A *quantum gate* is a gate which is applied to a qubit in order to change its state. Similar to the gates of a classical computer (e.g., `AND`, `OR`, `NOT`, etc.), the most basic operations

Name	Function	Num. Inputs	U
`NOT`	Standard NOT gate, $0 \to 1, 1 \to 0$	1	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
`CNOT`	Controlled NOT (can have multiple controls)	≥ 2	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
`CSWAP`	Controlled SWAP (can have multiple controls)	≥ 3	X
`HAD` (`H`)	Hadamard transform	1	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
`Pauli-X`	Bit flip. Same as NOT for basic inputs.	1	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
`Pauli-Y`	Phase flip	1	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
`Pauli-Z`	Bit and phase flip	1	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
$\texttt{ROT}_X(\theta)$	Rotation by θ in x	1	$\begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$
$\texttt{ROT}_Y(\theta)$	Rotation by θ in y	1	$\begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$
$\texttt{ROT}_Z(\theta)$	Rotation by θ in z	1	$\begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$

Table 1.1: A table of some basic quantum gates.

performed on a quantum computer are those which can be described by a single quantum gate. In a two-state quantum computer, a quantum gate is mathematically described by a unitary matrix $U \in \mathbb{C}^2$, i.e., $U^\dagger U = UU^\dagger = I$, where $U^\dagger$ denotes the conjugate transpose of U. In the vector representation of quantum states given in Remark 2, application of the gate simply amounts to matrix multiplication by U. The unitary matrices of some common quantum gates and operations are summarized in Table 1.1; we will investigate some of them individually in more detail later throughout the notes.

Remark 3 (Reversibility)**.** Note that the classical `AND` and `OR` gates are not part of Table 1.1. This is because neither operation is *reversible*: 1) the number of inputs and outputs do not match (e.g., both take two inputs but yield only one output), and 2) given the output bit of the gate, it is impossible to determine precisely what the input bits were (e.g., both $\texttt{AND}(0,0)$ and $\texttt{AND}(0,1)$ return 0). The necessary condition that a quantum gate must be reversible is further strengthed by the representation of each gate as a unitary matrix U, which implies the existence of an inverse $U^{-1} = U^\dagger$. Non-reversibility is indicative of a *loss/erasure of information*; in the interpretation of physicists, this would correspond to the "dissipation of heat" or the "loss of energy", owing to the interpretation of *Landower*. In the following Section 1.2.2, we explain the analogous quantum circuits to the classical `AND` and `OR` gates in more detail.

Generally speaking, a *quantum circuit* can be divided into three distinct components which use a combination of quantum gates to affect quantum states in different ways:

1. *superposition*: when two or more states exist at the same time.
2. *entanglement*: when distant quantum states are able to influence each other at a speed faster than light (i.e., "dependence") regardless of their distance. Electron spin is a notable example of the entanglement phenomemon.
3. *interference*: when two quantum states amplify and offset each other. Light waves and oscillations on the surface of water are notable examples of interference.

A *quantum computer* is implemented using multiple quantum circuits. A *quantum system* can refer either to a quantum circuit or a quantum computer. We will continue a detailed discussion of quantum circuits in more detail in the following Section 1.2.2, but first, we finish the basic overview of a quantum computer.

Errors in classical compuation are usually referring to the efficiency of the circuit, quality of the system's gates, or speed of some software implementation, i.e., the operations which make up the actual computation. However, in quantum computers, the material of the qubit (e.g., barium, ytterbium) has a significant bearing on the performance of the quantum computer. As a result, there are two other sources of error which typically do not arise in the classical setting: *state preparation* and *measurement*.

Briefly, we consider the problem of state preparation. Suppose we are given some quantum system S, which performs gate operations on N qubits and is represented by the Hamiltonian operator H. Suppose the eigenfunctions of H are given by $|\Psi_n\rangle$, with corresponding eigenvalues λ_n, and let the indices be assigned such that the eigenvalues are ordered $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Because the eigenvalues represent levels of energy (see Definition 1), quantum states corresponding to smaller eigenvalues play a more dominant role in how the quantum system S behaves in a modest temperature environment (e.g., laboratory room).

Theorem 1 (Ritz Variational Principle)**.** Let $\theta \in \mathbb{R}^p$ be a vector of parameters and let $|\Psi(\theta)\rangle$ be a quantum state of quantum system S which depends on θ. Then

$$\langle H\rangle_{|\Psi(\theta)\rangle} := \frac{\langle\Psi(\theta)|H|\Psi(\theta)\rangle}{\langle\Psi(\theta)|\Psi(\theta)\rangle} \geq \lambda_1 \tag{1.16}$$

where λ_1 is the ground-state energy of S. Again, the normalization condition says that $\langle\Psi(\theta)|\Psi(\theta)\rangle = 1$.

The Variational Quantum Eigensolver (VQE) is a popular method of preparing the initial qubit states by computing λ_1 via Theorem 1. Essentially, it iteratively minimizes the right side of (1.16), $\langle\Psi(\theta)|H|\Psi(\theta)\rangle$.

Define $\rho(\theta)$ to be the density matrix representing the qubit material upon which the initial quantum states are prepared. Ideally, in an environment where state preparation is error-free, we have $\rho(\theta) = |\Psi(\theta)\rangle \otimes \langle\Psi(\theta)|$. Moreover, the expectation of an operator on a system S whose qubits were prepared using $\rho(\theta)$ is rewritten from Definition 2 as:

$$\langle O\rangle_\rho := \mathrm{tr}[\rho(\theta)O]$$

Moreover, Theorem 1 still holds regardless of θ and $\rho(\theta)$. So (1.16) is also rewritten:

$$\mathrm{tr}[\rho(\theta)H] \geq \lambda_1$$

The equivalent *variational principle for variances* is precisely the equation $\mathrm{Var}(H)_{|\Psi_n\rangle} = 0$, derived in the previous Section 1.1.2, where

$$\mathrm{Var}(O)_\rho := \langle O^2\rangle_\rho - \langle O\rangle_\rho^2 \tag{1.17}$$

for some operator O. The result from Section 1.1.2 also holds regardless of parameter choice θ and density matrix $\rho(\theta)$.

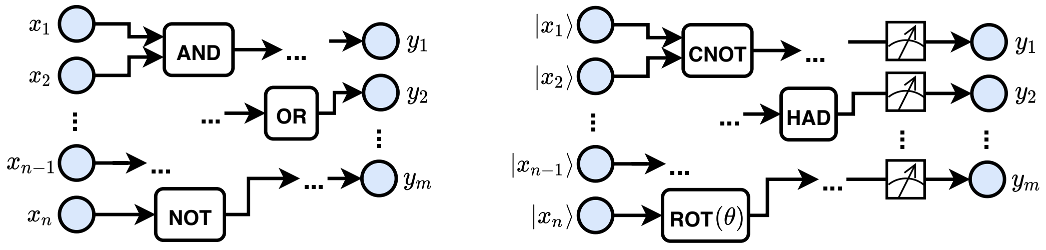


Figure 1.1: Circuit implementations for two different computers. [Left] The classical circuit, composed of well-known gates. [Right] The quantum circuit, composed of quantum gates (see Definition 5) and measurement devices.

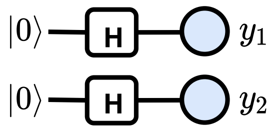


Figure 1.2: .

1.2.2 Basic Circuits

Classical circuits are constructed to compute some function $f : \{0,1\}^n \to \{0,1\}^m$ which maps an n-bit input to an m-bit output, for some $n, m \in \mathbb{N}$ not necessarily equal to each other. Quantum circuits are probabilistic circuits which behave similarly to classical circuits, but employ a different set of gates described in the previous sections (see Definition 5 and Table 1.1). A visual illustration emphasizing some key differences between classical and quantum circuits is roughly depicted in Fig. 1.1.

Example 2 (Motivating Quantum Circuit)**.** Consider a simple quantum circuit which takes two inputs and feeds them in parallel into two `HAD` gates (see Fig. 1.2). Then the output yields

$$(H \otimes H)|0\rangle \otimes |0\rangle =$$

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$$

Theorem 2. *Any classical circuit C computing Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ can be "efficiently" converted into a reversible (hence, a quantum) circuit Q, which implements $f_q : \{0,1\}^{n+1} \to \{0,1\}^{m+g}$ for $a \in \mathbb{N}$ the number of ancilla bits and $g \in \mathbb{N}$ the number of garbage bits such that $n + a = m + g$.*

Problem 1 (Conceptual Problems about Quantum Circuits)**.** Given the above setup, there are a handful of immediate questions one may ask about quantum circuits.

1. Are there some functions f for which quantum circuits can compute much more efficiently than classical ones?

2. Conversely, are there any functions f for which classical circuits can compute more efficiently than quantum circuits?

We begin by demonstrating the second question in Problem 1 with the following motivational use case. Recall from Remark 3, the reversibility property of quantum gates required an alternative implementation of the classical `AND` and `OR` gates. To turn the `AND`/`OR` gates into reversible gates, we invoke the `CSWAP` gate

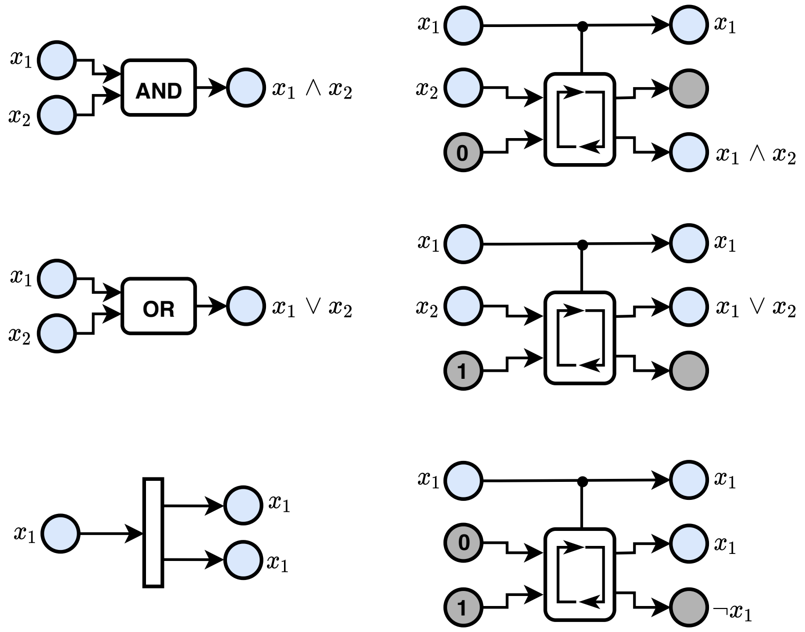


Figure 1.3: Quantum circuit implementations of classical gates and operations using the `CSWAP` gate; from top to bottom, `AND`, `OR`, and fanout.

x_1	x_2	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.2: Truth table for two-bit addition from Example 3.

(see Table 1.1) and introduce *ancilla* bits, i.e. additional bits which are separate from the input used to facilitate computation of the desired output. The resulting circuits are shown in Fig. 1.3, along with an analogous quantum implementation of the classical *fanout* gate, which returns multiple copies of the same output in order to feed each output to other distinct circuits.

Example 3 (Addition)**.** Consider the 2-bit addition operation, summarized by the truth table in Table 1.2. A classical gate which implements 2-bit addition can be built using one `XOR` gate and one `AND` gate, where the Sum bit is determined by `XOR`(x_1, x_2) and the Carry bit is determined by `NOT`(x_1, x_2). In comparison, a quantum circuit implementation would invoke a series of `CNOT` gates with a total of 4 inputs and 4 outputs; see

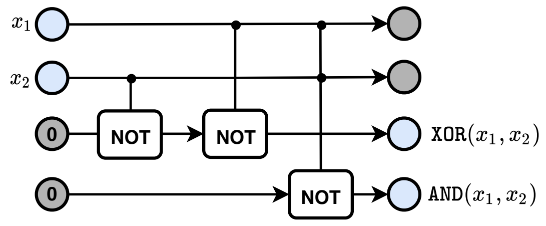


Figure 1.4: Quantum circuit implementations of 2-bit addition from Example 3.

x_1	x_2	$f(x)$
0	0	0
0	1	1
1	0	1
1	1	0

x_1	x_2	Output
0	0	$+\|00\rangle$
0	1	$-\|01\rangle$
1	0	$-\|10\rangle$
1	1	$+\|11\rangle$

Table 1.3: [Left] The truth table for the XOR operation. [Right] The input-output pairs of the quantum circuit $Q^{\pm}_{\texttt{XOR}}$.

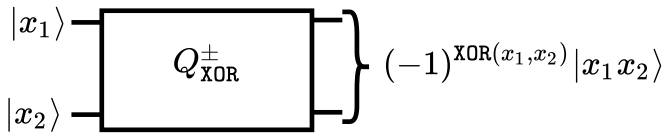


Figure 1.5: Implementation of Fig. 1.6 specifically for the XOR operation.

Remark 4. In Theorem 2, the term "efficiency" roughly refers to two aspects. First, the amount of computational parts (gates, wires, ancilla bits, etc.) to construct the quantum circuit is proportional to the amount of computation parts in the original classical circuit; essentially, it doesn't take a ridiculous number of extra parts to build the quantum analog of any classical circuit. Second, the resulting quantum circuit is easily interpretable by human standards; since many of the standard classical irreversible gates can be transformed using a simple CSWAP gate, it doesn't take too much extra knowledge to understand what a quantum circuit is doing.

Remark 5 (Sign-Implementation Trick)**.** Suppose Q_f is a quantum circuit which implements the operation/basic function $f : \{0,1\}^n \to \{0,1\}$, i.e. it has output dimension $m = 1$. It is common to invoke the *sign-implementation trick* to determine the "sign" of $f(x)$. The resulting circuit is implemented by $Q^{\pm}_f$ (see Fig. 1.6).

$$|x\rangle \otimes |0\rangle \xrightarrow{Q^{\pm}_f} (-1)^{f(x)}|x\rangle \otimes |0\rangle$$

A notable example of the sign-implementation trick is in the construction of the classical XOR.

Example 4 (Implementing XOR)**.** Consider the (XOR) function, which we define as $f : \{0,1\}^2 \to \{0,1\}$, with $f(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_1)$. The truth table is written in the left of Table 1.3.

Using the sign implementation trick, we can implement XOR on the quantum circuit; see Fig. 1.5 for the result. Note that in order to maintain reversability, the circuit still outputs 2 qubits. Thus, the true result is encoded into the sign; see the right table of Table 1.3.

Note that all four answers to the NEQ circuit are present as summation terms in the output to (1.18). However, we can't just measure the state to obtain the answers because we will only get one of the four *inputs* with uniform probability. If we used Q_f, we could encode both the inputs and outputs as pairs, and return $(1/2)(|00,0\rangle + |01,1\rangle + |10,1\rangle + |11,1\rangle)$; measuring it would yield one of the four *input/output pairs* with uniform probability. Neither approach is entirely appealing.

In general, quantum circuits can

- *implement* Boolean functions f on classical inputs, i.e., $Q_f[|x\rangle|b\rangle|0\cdots0\rangle] = |x\rangle|b \oplus f(x)\rangle|0\cdots0\rangle$, for inputs $x \in \{0,1\}^n$, outputs $b \in \{0,1\}^m$, ancilla bits $0\cdots0$, and $\oplus$ being the bitwise XOR. See Fig. 1.6 for a visualization of the circuit which implements f, accepting inputs x and b, and returning $b \oplus f(x)$.

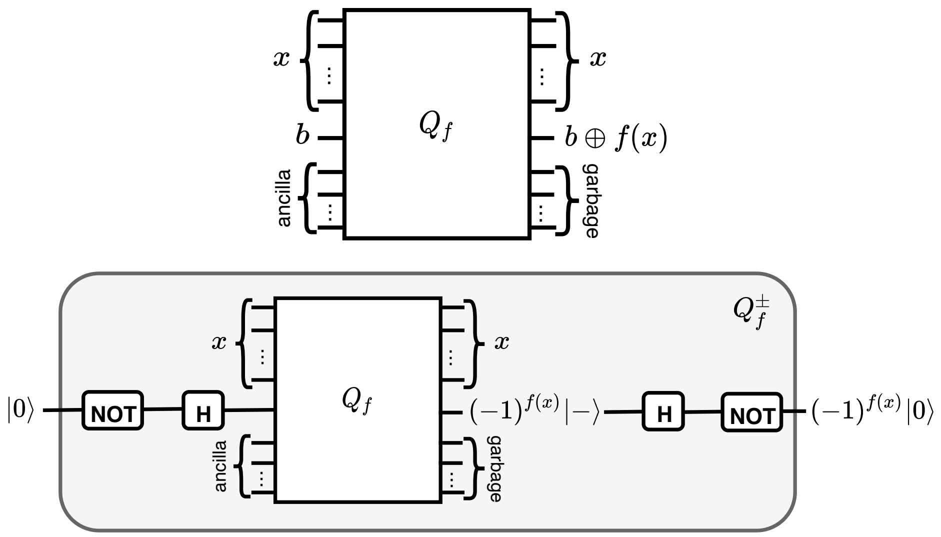


Figure 1.6: [Top] A quantum circuit implementing the function f. [Bottom] A quantum circuit corresponding to the sign implementation trick applied to f.

- accept superposition states as inputs. For example,

$$Q_f^{\pm}\left[\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)\right] = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) \tag{1.18}$$

  Note that similar to how tensor products of qubits can be used to create qudits, superpositions of qudit states can be built from superpositions of basic states:

$$|+\rangle|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$$

  Here, the tensor products $\otimes$ are omitted between adjacent $|\cdot\rangle$ for notation simplicity, but their presence is understood.

1.2.3 How Quantum Circuits Compute Functions

For more complex functions, we consider a further alternative circuit, inspired by Simon's famous "rotate-compute-rotate" mantra commonly emloyed in quantum computing literature. First, each input qubit is fed through the Hadamard gate `HAD`. Then the function is computed, possibly with sign-implementation. Finally, the result is fed through another Hadamard gate `HAD`. To examine what exactly this does, consider the output of Example 4 when fed through Fig. 1.7:

$$\frac{1}{2}(|{+}{+}\rangle - |{+}{-}\rangle - |{-}{+}\rangle + |{+}{+}\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Expanding the term corresponding to $|11\rangle$ yields

$$\frac{1}{2}\left(\frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}\left(-\frac{1}{\sqrt{2}}\right) - \left(-\frac{1}{\sqrt{2}}\right)\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\right)|11\rangle = 1 \cdot |11\rangle$$

However, by the normalization of amplitudes, this means that the amplitudes of the other terms, $|00\rangle, |01\rangle, |10\rangle$ must be zero! Hence, by a nice interference phenomenon, the probability of measuring $|11\rangle$ is precisely 1.

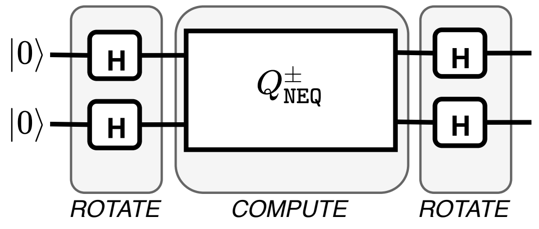


Figure 1.7: The ROTATE-COMPUTE-ROTATE paradigm implemented on `NEQ` (see Fig. 1.5).

We'll see that the reason we obtained an amplitude of 1 on $|11\rangle$ is due to an *XOR-pattern* in the truth table of `NEQ`.

What if there were garbage bits? It would prevent the nice interference from occurring. Suppose g_i represented garbage bits and we considered the ket notation $|y, g_i\rangle$, where g_i was the garbage bit output along with y. All the potential garbage bits output from Fig. 1.7 would be:

$$\frac{1}{2}\left(|{+}{+}, g_1\rangle - |{+}{-}, g_2\rangle - |{-}{+}, g_3\rangle + |{-}{-}, g_4\rangle\right)$$

Then part of the amplitude of $|11\rangle$ becomes

$$\frac{1}{4}\left(|11, g_1\rangle + |11, g_3\rangle + \cdots\right)$$

No cancellations occur, and the total amplitude from the above terms alone do not sum to 1. This illustrates why it is important to carefully process even the garbage outputs.

In general, for a Boolean function $f : \{0,1\}^n \to \{0,1\}$ on $n \in \mathbb{N}$ bits:

- *ROTATE I*: initialize n $|0\rangle$'s and put it through n `HAD` gates $H^{\otimes n}$. Returns

$$|+\rangle|+\rangle\cdots|+\rangle = \left(\frac{1}{\sqrt{2}}\right)^n (\text{summation terms}) = \frac{1}{\sqrt{N}} \sum_{x\in\{0,1\}^n} |x\rangle \tag{1.19}$$

  where $N := 2^n$. Here, (1.19) is defined as the *uniform superposition state.*

- *COMPUTE*: Plug the above result into $Q^{\pm}_f$. Returns

$$\frac{1}{\sqrt{N}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)}|x\rangle \tag{1.20}$$

  Essentially, the truth table of all possible 2^n combinations and answers is probabilistically encoded into the entangled state (1.20). At this stage, measuring the state (1.20) yields the correct result with uniform $\frac{1}{N}$ probability.

- *ROTATE II*: Apply n `HAD` gates $H^{\otimes n}$ to (1.20) to transform the amplitudes from uniformly-random to a single certainty.

$$H^{\otimes n}\left(\frac{1}{\sqrt{N}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)}|x\rangle\right) = \frac{1}{\sqrt{N}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)} H^{\otimes n}|x\rangle = \frac{1}{\sqrt{N}} \sum_{s\in\{0,1\}^n} \hat{f}(s)|s\rangle \tag{1.21}$$

  where $\hat{f}(s)$ is the "Boolean Fourier transform" coefficient correpsonding to $|s\rangle$. At this state, measuring the state (1.21) yields the correct result with probability 1.

One important distinction of quantum computers from classical computers are their ability to find (XOR) patterns in implicitly-represented data. In this section, we describe what this precisely means.

We now consider an alternative perspective which interchanges vectors of complex numbers and functions, so that we may simultaneously think of quantum states as vectors and functions. A vector $\mathbf{v} \in \mathbb{C}^n$ is treated equivalently to a truth-table of functions $f : \{0,1\}^n \to \mathbb{C}$ which map n-bit strings to complex numbers instead of bits. This truth-table of functions is also treated equivalently to an n-qubit state, where each entry of the complex vector $\mathbf{v}$ corresponds to a coefficient in one of the 2^n basis patterns. Note that this implies the vector $\mathbf{v}$ must have unit norm.

Definition 6. If $g : \{0,1\}^n \to \mathbb{C}$, then $|g\rangle$ (with slightly abusive notation) denotes the quantum state $\frac{1}{\sqrt{N}} \sum_{x \in \{0,1\}^n} g(x)|x\rangle$.

Remark 6. The $|g\rangle$ is a quantum state iff $\frac{1}{N} \sum_x |g(x)|^2 = 1$, e.g., if $g = (-1)^f$ for some $f : \{0,1\}^n \to \{0,1\}$

The basis $\{|\chi\rangle_0, \cdots, |\chi\rangle_{N-1}\}$ we focus on are XOR functions

$$\chi_s : \{0,1\}^n \to \{-1,+1\}, \;\; x \to (-1)^{\mathtt{XOR}_s(x)} \tag{1.22}$$

where

$$\mathtt{XOR}_s(x) := \sum_{\{i : s_i = 1\}} x_i \mod 2$$

A special property of the pattern vectors (1.22) is that they are Boolean-valued in $\{-1,+1\}$.

Fourier Sampling takes the above discussion from the perspective of Fourier transforms. To this end, we introduce a new notation $\hat{g}(s)$ which defines the "correlation" between the function g and the basis χ_s.

1.2.4 Hamiltonian Simulation

We are now interested in the simulation of quantum systems with respect to other quantum systems. Building from the representation of the quantum system at the end of Section 1.2.1, let S be some quantum system which performs gate operations on N qubits and is represented by the Hamiltonian operator H. Further define Q to be a different quantum system which acts on a subset of the N qubits from S, and is represented by the Hamiltonian operator H_Q. For example, if S is a quantum computer, then one can think of Q as a single quantum circuit inside. Or, if S is a quantum circuit, then Q may represent a quantum algorithm which uses S to evaluate some function result.

Assumption 1 (Polynomial Sum)**.** Operators O pertaining to Q belong to the class whose expectation can be measured efficiently on S, e.g., a decomposition of polynomial-time operators

$$O = \sum_{j=1}^{N_p} c_j O_j \tag{1.23}$$

where N_p is the number of operators Q_j which take polynomial-order computation time with respect to the size of the system S, and $c_j \in \mathbb{C}$ are coefficients.

Definition 7 (Hamiltonian)**.** Under Assumption 1, *Hamiltonian* $H(t)$ of the quantum system S can be expressed as a linear combination of simpler Hamiltonians $H_j(t)$ which replace the polynomial-order O_n sum terms in (1.23).

$$H(t) = \sum_{j=1}^{N_p} c_n H_j(t) \tag{1.24}$$

If each term abides by the following conditions: 1) bounded norm, i.e., $\|H_j(t)\| \leq h$ for some $h > 0$, and 2) acts on no more than some fixed number $k \in \mathbb{N}$ of qubits for a value of k independent of the system, i.e., $H_j(t) = 0$ for all $|X| > k$. Then $H(t)$ is said to be a *locally-bounded Hamiltonian.*

Example 5 (Decomposition as Pauli Operators)**.** Let S and Q be the quantum systems described above, with respective Hamiltonians H and H_Q. For concreteness, let $\{1, \cdots, N'\} \subseteq \{1, \cdots, N\}$, $N' \leq N$, be the subset of qubits from S upon which Q acts. Let $P_i^{(\alpha)}$ be the Hamiltonian for the Pauli-α gate (see Table 1.1) applied to the ith qubit, where $i \in \{1, \cdots, N\}$ and $\alpha \in \{X, Y, Z\}$. We henceforth refer to Pauli-gate Hamiltonian $P_i^{(\alpha)}$ as the *Pauli operator* in direction α for qubit i. Then note that H_Q is composed by enumerating over all possible strict subsets of qubits $\{1, \cdots, N'\}$ of Pauli operators:

$$\begin{aligned} H_Q &= \sum_{\mathcal{X} \subset \{1,\cdots,N'\}} c_{\mathcal{X}} H_{\mathcal{X}} \\ &:= \underbrace{\sum_{\substack{i=1 \\ \alpha \in \{X,Y,Z\}}}^{N'} c_i^{(\alpha)} P_i^{(\alpha)}}_{\text{singletons}} + \underbrace{\sum_{\substack{i,j=1 \\ \alpha,\beta \in \{X,Y,Z\}}}^{N'} c_{ij}^{(\alpha\beta)} P_i^{(\alpha)} \otimes P_j^{(\beta)}}_{\text{two elements}} + \cdots + \underbrace{\sum_{\substack{i=1 \\ \vec{\alpha} \in \{X,Y,Z\}^{N'-1}}}^{N'} c_{-i}^{(\vec{\alpha})} \bigotimes_{j \neq i} P_j^{(\alpha_j)}}_{N'-1 \text{ elements}} \end{aligned} \tag{1.25}$$

The reason Pauli operators are commonly considered to decompose any general Hamiltonian is because of their simplicity: many quantum devices can efficiently evaluate (i.e., in polynomial time) the expectation of a tensor product of an arbitrary number of Pauli operators.

Knowing the exact circuitry of a quantum system is not necessary in order to understand how its dynamics behave. In many cases, it is convenient to abstract away the details of the circuit and just consider the energy evolution over time.

Suppose we are given the following general form of the linear partial differential equation

$$\frac{\partial}{\partial t} f(t) = \mathcal{M} f(t) \tag{1.26}$$

where $\mathcal{M}$ is an operator or matrix. One example of (1.26) that we've already seen is the Schrödinger equation, where $f(t) \equiv \Psi(t, x)$ and $\mathcal{M} \equiv -iH$ for Hamiltonian H from Definition 1.

The solution to (1.26) is given by Green's function

$$f(t) = G(t, 0) f(0) \equiv e^{t\mathcal{M}} f(0) \tag{1.27}$$

However, a well-known problem with (1.27) is that the matrix exponential $e^{t\mathcal{M}}$ is difficult. A common way of obtaining $e^{t\mathcal{M}}$ is to diagonalize $\mathcal{M}$, but in quantum systems, this is also difficult, hence the development of algorithms such as the variational quantum eigenseolver.

The classical (time-invariant) Hamiltonian ((1.11) without the time argument t) can be expressed as a sum of kinetic and potential terms: $H(\mathbf{p}, \mathbf{q}) = K(\mathbf{p}) + V(\mathbf{q})$. Hamilton's equations (1.12) can also be expressed in matrix operator form:

$$\frac{d}{dt}\begin{bmatrix}\mathbf{p}(t)\\ \mathbf{q}(t)\end{bmatrix} = \begin{bmatrix}-\frac{\partial H}{\partial \mathbf{q}}\\ \frac{\partial H}{\partial \mathbf{p}}\end{bmatrix} = \begin{bmatrix}-\frac{d}{d\mathbf{q}}V(\mathbf{q})\\ \frac{d}{d\mathbf{p}}K(\mathbf{p})\end{bmatrix} = \begin{bmatrix}0 & -\tilde{V}\\ \tilde{K} & 0\end{bmatrix}\begin{bmatrix}\mathbf{p}(t)\\ \mathbf{q}(t)\end{bmatrix} \tag{1.28}$$

where $\tilde{K}$ and $\tilde{V}$ are operators defined such that

$$\tilde{K}\mathbf{p} = \frac{d}{d\mathbf{p}}K(\mathbf{p}), \quad -\tilde{V}\mathbf{q} = -\frac{d}{d\mathbf{q}}V(\mathbf{q}) \tag{1.29}$$

Thus, the Hamiltonian *operator* $\mathcal{H}$ is the matrix

$$\mathcal{H} = \mathcal{K} + \mathcal{V}, \quad \text{where } \mathcal{K} = \begin{bmatrix}0 & 0\\ \tilde{K} & 0\end{bmatrix}, \; \mathcal{V} = \begin{bmatrix}0 & -\tilde{V}\\ 0 & 0\end{bmatrix} \tag{1.30}$$

where $\mathcal{K}$ is the kinetic part, and $\mathcal{V}$ is the potential part.

Solving (1.28) like a linear ODE yields a matrix exponential-like solution

$$\begin{bmatrix}\mathbf{p}(t)\\ \mathbf{q}(t)\end{bmatrix} = e^{t\mathcal{H}}\begin{bmatrix}\mathbf{p}(0)\\ \mathbf{q}(0)\end{bmatrix} \tag{1.31}$$

Here, $e^{t\mathcal{H}}$ is often called the *time-evolution* of the Hamiltonian operator $\mathcal{H}$. The analogue for quantum systems is defined formally below.

Definition 8 (Time-Evolution Operator)**.** The *time-evolution operator* $U(0,t)$ correpsonding to the Hamiltonian $\mathcal{H}(t)$ of the form from Definition 7 over the interval of time $[0,t]$ satisfies Schrödinger's equation

$$i\frac{d}{dt}U(0,t) = \mathcal{H}(t)U(0,t)$$

The solution is written formally in terms of the following *time-ordered exponential*

$$U(0,t) = \mathcal{T}\left\{e^{-i\int_0^t \mathcal{H}(s)ds}\right\} := \sum_{n=0}^{\infty}\int_0^t\int_0^{s_n}\cdots\int_0^{s_2}\mathcal{H}(s_n)\cdots\mathcal{H}(s_1)ds_1\cdots ds_n \tag{1.32}$$

where $\mathcal{T}$ denotes the *time-ordered integral* operation.

In order to understand the time-ordered integral (1.32), it is necessary to first discuss the Trotter decomposition

Definition 9 (Trotter Decomposition)**.** Let H be an operator or matrix which can be decomposed into a sum of two simpler matrices or operators, $H = A + B$. Then, for some $t \in \mathbb{R}$, the *Trotter decomposition* approximates the time-evolution operator e^{tH} as

$$e^{tH} = \left(e^{\frac{t}{n}A}e^{\frac{t}{n}B}\right)^n \tag{1.33}$$

where $n \in \mathbb{N}$ is referred to as the *Trotter number.*

The Trotter decomposition can be motivated through a series of simple math calculations. First, by Taylor expansion:

$$e^{t(A+B)} = I + t(A+B) + \frac{1}{2}t^2\underbrace{(A+B)^2}_{=A^2+AB+BA+B^2} + O(t^3) \tag{1.34}$$

and

$$\begin{aligned} e^{tA}e^{tB} &= \left(I + tA + \frac{1}{2}t^2A^2 + O(t^3)\right)\left(I + tB + \frac{1}{2}t^2B^2 + O(t^3)\right) \\ &= I + t(A+B) + \underbrace{t^2AB + \frac{1}{2}t^2A^2 + \frac{1}{2}t^2B^2}_{=\frac{1}{2}(A^2+2AB+B^2)} + O(t^3) \end{aligned} \tag{1.35}$$

Note that the key difference in terms between (1.34) and (1.35) comes from the fact that matrices and operators are not necessarily commutative, hence $2AB \neq AB + BA$ in general. Setting them both equal to each other would require

$$\begin{aligned} e^{tA}e^{tB} &= e^{t(A+B)} - \frac{1}{2}t^2(A+B)^2 + O(t^3) + \frac{1}{2}(A^2 + 2AB + B^2) + O(t^3) \\ &= e^{t(A+B)} + O(t^3) + \frac{1}{2}t^2[A,B] \\ &= e^{t(A+B)+\frac{1}{2}t^2[A,B]+O(t^3)} \end{aligned} \tag{1.36}$$

where $[A,B] := AB - BA$. The last line can be verified by Taylor expansion. For the purposes of application, the parameter t is divided into $n \in \mathbb{N}$ slices.

$$\left(e^{\frac{t}{n}A}e^{\frac{t}{n}B}\right) = \left(e^{\frac{t}{n}(A+B)+\frac{1}{2}\left(\frac{t}{n}\right)^2[A,B]+O\left(\left(\frac{t}{n}\right)^3\right)}\right)^n = e^{\frac{t}{n}(A+B)+\frac{1}{2}\frac{t^2}{n}[A,B]+O\left(\frac{t^3}{n^2}\right)} \tag{1.37}$$

Note that the correction term $\frac{1}{2}\frac{t^2}{n}[A,B] + O\left(\frac{t^3}{n^2}\right)$ tends to 0 as the Trotter number $n \to \infty$.

Define the first and second-order Trotter approximants as follows

$$G_1(t, t+\Delta t) \triangleq e^{-i\Delta t A(t+\Delta t)}e^{-i\Delta t B(t+\Delta t)} \tag{1.38a}$$

$$G_2(t, t+\Delta t) \triangleq e^{-\frac{i}{2}\Delta t A\left(t+\frac{1}{2}\Delta t\right)}e^{-i\Delta t B\left(t+\frac{1}{2}\Delta t\right)}e^{-\frac{i}{2}\Delta t A\left(t+\frac{1}{2}\Delta t\right)} \tag{1.38b}$$

Note that these are merely *symmetrized* versions of the previous Trotter approximant (1.33) for $n = 1$ and $n = 2$. The case of $n = 2$ becomes clear when considering

$$S_2(t) \triangleq e^{\frac{t}{2}A}e^{tB}e^{\frac{t}{2}A} = e^{t(A+B)+t^3R_3+t^5R_5+\cdots} \tag{1.39}$$

where R_i is the ith-order remainder term from the Taylor expansion. Symmetrization is used primarily for its convenient property:

$$S_n(t)S_n(-t) = I \tag{1.40}$$

for any even $n \in \mathbb{N}$. This is easy to verify in the $n = 2$ case.

To construct higher-order approximants, *fractal decomposition* may be used. For example, in the case where $n = 4$, consider the matrix decomposition for an arbitrary $c \in \mathbb{R}$:

$$S(t) \triangleq S_2(ct)S_2((1-2c)t)S_2(ct) = e^{\frac{ct}{2}A}e^{ctB}\underbrace{e^{\frac{ct}{2}A}e^{\frac{(1-2c)t}{2}A}}e^{(1-2c)tB}\underbrace{e^{\frac{(1-2c)t}{2}A}e^{\frac{ct}{2}A}}e^{ctB}e^{\frac{ct}{2}A} \tag{1.41}$$

where the underbraced terms simplify as follows

$$e^{\frac{ct}{2}A}e^{\frac{(1-2c)t}{2}A} = \left(e^A\right)^{\frac{ct}{2}}\left(e^A\right)^{\frac{(1-2c)t}{2}} = \left(e^A\right)^{\frac{(1-c)t}{2}}$$

Using (1.39) on (1.41):

$$\begin{aligned} S_2(ct)S_2((1-2c)t)S_2(ct) &= \left(e^{ct(A+B)+c^3t^3R_3+O(t^5)}\right)\left(e^{(1-2c)t(A+B)+(1-2c)^3t^3R_3+O(t^5)}\right)\left(e^{ct(A+B)+c^3t^3R_3+O(t^5)}\right) \\ &= e^{t(A+B)+(2c^3+(1-2c)^3)t^3R_3+O(t^5)} \end{aligned} \tag{1.42}$$

Note that we are not using $e^{t(A+B)} = e^{tA}e^{tB}$ in order to simplify the computation, because the Trotter decomposition shows that this is not true in general. Instead, this simplification can be verified with the Taylor expansion

Applying (1.40), we get

$$S_2(ct)S_2((1-2c)t)S_2(ct) = S_2(-ct)S_2(-(1-2c)t)S_2(-ct) \implies 2c^3 + (1-2c)^3 = 0 \implies c = \frac{1}{2-\sqrt[3]{2}} \tag{1.43}$$

because all higher-order correction terms should vanish except for the linear term $t(A+B)$. Substituting c from (1.43) into (1.41) yields a symmetrized fourth-order approximant.

Now return back to the time-evolution operator from Definition 8. We introduce the following time-shift operator.

Definition 10 (Time-Shift Operator)**.** The *time-shift operator* $\mathscr{T} := i(\partial/\partial t)$ acts on a product of operators $F(t)$ and $G(t)$ in the following way

$$F(t)e^{-i\Delta t\mathscr{T}}G(t) = F(t+\Delta t)G(t) \tag{1.44}$$

This operator is also extendable to more than two operators:

$$F(t)e^{-i\Delta t\mathscr{T}}G(t)e^{-i\Delta t\mathscr{T}}H(t) = F(t+2\Delta t)G(t+\Delta t)H(t) \tag{1.45}$$

Using Definition 10, the time-ordered exponential (1.32) is transformed

$$\begin{aligned} e^{-i\Delta t(\mathcal{H}(t)+\mathscr{T})} &= \lim_{n\to\infty}\left(e^{-i\frac{\Delta t}{n}\mathcal{H}(t)}e^{-i\frac{\Delta t}{n}\mathscr{T}}\right)^n \quad \text{by Trotter decomposition} \qquad (1.46) \\ &= \lim_{n\to\infty}\underbrace{\left(e^{-i\frac{\Delta t}{n}\mathcal{H}(t)}e^{-i\frac{\Delta t}{n}\mathscr{T}}\right)\left(e^{-i\frac{\Delta t}{n}\mathcal{H}(t)}e^{-i\frac{\Delta t}{n}\mathscr{T}}\right)\cdots\left(e^{-i\frac{\Delta t}{n}\mathcal{H}(t)}e^{-i\frac{\Delta t}{n}\mathscr{T}}\right)}_{n \text{ times}} \\ &= \lim_{n\to\infty} e^{-i\frac{\Delta t}{n}\mathcal{H}(t+\Delta t)}e^{-i\frac{\Delta t}{n}\mathcal{H}(t+\frac{n-1}{n}\Delta t)}\cdots e^{-i\frac{\Delta t}{n}\mathcal{H}(t+\frac{1}{n}\Delta t)} \quad \text{by Definition 10} \\ &= \mathcal{T}\left\{e^{-i\int_t^{t+\Delta t}\mathcal{H}(s)ds}\right\} \qquad (1.47) \end{aligned}$$

Applying (1.46) to (1.30):

$$\mathcal{T}\left\{e^{-i\int_t^{t+\Delta t}\mathcal{H}(s)ds}\right\} = e^{-i\Delta t(\mathcal{K}(t)+\mathcal{V}(t)+\mathscr{T})} \tag{1.48}$$

For the purposes of Hamiltonian simulation of quantum systems, we consider only first and second-order approximants, given by (1.38). Applied to (1.48), we get:

$$G_1(t,t+\Delta t) \triangleq e^{-i\Delta t\mathcal{K}(t)}e^{-i\Delta t\mathcal{V}(t)}e^{-i\Delta t\mathscr{T}} = e^{-i\Delta t\mathcal{K}(t+\Delta t)}e^{-i\Delta t\mathcal{V}(t+\Delta t)} \tag{1.49a}$$

$$G_2(t,t+\Delta t) \triangleq e^{-\frac{i}{2}\Delta t\mathscr{T}}e^{-\frac{i}{2}\Delta t\mathcal{K}(t)}e^{-i\Delta t\mathcal{V}(t)}e^{-\frac{i}{2}\Delta t\mathcal{K}(t)}e^{-\frac{i}{2}\Delta t\mathscr{T}} = e^{-\frac{i}{2}\Delta t\mathcal{K}\left(t+\frac{1}{2}\Delta t\right)}e^{-i\Delta t\mathcal{V}\left(t+\frac{1}{2}\Delta t\right)}e^{-\frac{i}{2}\Delta t\mathcal{K}\left(t+\frac{1}{2}\Delta t\right)} \tag{1.49b}$$

In our quantum systems of consideration, the Hamiltonian is decomposed into more than two parts, as seen in Definition 7. Using the revised notation of (1.28) and (1.30), this sum is

$$\mathcal{H}(t) = \sum_{j=1}^{N_p} c_n \mathcal{H}_j(t) \tag{1.50}$$

Moreover, the time-evolution operator is broken apart into short segments of time. This can be thought of approximating the overall circuit into smaller quantum circuits, which is an approximation with "small-enough" error as long as $\mathcal{H}(t)$ varies over time "slowly enough".

$$G(0,t) = G(0,\Delta t)G(t_1, t_1 + \Delta t)\cdots G(t_m, t_m + \Delta t) \tag{1.51}$$

where $0 \leq t_0 < t_1 < t_2 < \cdots < t_m \leq t \leq t_m + \Delta t$ where $t_{k+1} - t_k = \Delta t > 0$ is the stepsize and $m \in \mathbb{N}$ is the number of partitions. Each piece $0 \leq k \leq m-1$ of the decomposed time-evolution operator can be written in the time-ordered integral form:

$$G(t_k, t_k + \Delta t) = \mathcal{T}\left\{ e^{-i\int_{t_k}^{t_k+\Delta t} \mathcal{H}(s)ds} \right\} \tag{1.52}$$

We've seen from the Trotter decomposition that if $\mathcal{H}(t) = \mathcal{H}_1(t) + \mathcal{H}_2(t)$, then (1.52) can be approximated as

$$\hat{G}(t_k, t_k + \Delta t) = \mathcal{T}\left\{ e^{-i\int_{t_k}^{t_k+\Delta t} \mathcal{H}_1(s)ds} \right\} \mathcal{T}\left\{ e^{-i\int_{t_k}^{t_k+\Delta t} \mathcal{H}_2(s)ds} \right\} \tag{1.53}$$

Note that $\mathcal{H}_1$ and $\mathcal{H}_2$ need not be the kinetic $\mathcal{K}$ and potential $\mathcal{V}$ parts of the Hamiltonian. For instance, the general Definition 7 makes no such condition, and in Example 5, we see the decomposition of a Hamiltonian in terms of simpler Pauli operators.

For a general Hamiltonian decomposition sum and partition of the time interval $[0, t]$, a numerical approximation of $G(0, t)$ can be made by using (1.51) and extending the Trotter decomposition (1.53) as follows:

$$G(0,t) \approx \prod_{k=1}^{m} \prod_{j=1}^{N_p} \mathcal{T}\left\{ e^{-i\int_{t_k}^{t_k+\Delta t} \mathcal{H}_j(s)ds} \right\} \tag{1.54}$$

As shown in (1.53), the product over the summation parts of the Hamiltonian do not need to be ordered.

In the special case where the Hamiltonian $\mathcal{H}(t)$ is approximated as a piecewise-constant function over the partitioned interval $0 \leq t_0 < t_1 < t_2 < \cdots < t_m \leq t \leq t_m + \Delta t$, (1.54) simplifies as

$$G(0,t) \approx \prod_{k=1}^{m} \prod_{j=1}^{N_p} e^{-i\mathcal{H}_j(t_k)\Delta t} \tag{1.55}$$

1.3 Algorithms in Quantum Implementation

1.3.1 Simon's Algorithm

Simon's algorithm was the first algorithm to demonstrate the computational benefits of using a quantum computer over a traditional computer.

x	$f(x)$
(000)	Red
(001)	Yellow
(010)	Blue
(011)	Green
(100)	Yellow
(101)	Red
(110)	Green
(111)	Blue

Table 1.4: An example of a color assignment to a function f with string length $n = 3$. Here, f is L-periodic with $L = (101)$.

Let f be a Boolean function implemented by quantum circuit Q_f. Compared to the Fourier Sampling algorithm discussed previously, there are two key differences:

- we use more than a single application of Q_f.
- f will output more than one bit.

Definition 11 (L-periodic)**.** A function f is L-periodic for some string $L \in \{0,1\}^n/(0\cdots 0)$ if for all $x \in \{0,1\}^n$, $f(x+L) = f(x)$ where the sum $+$ represents coordinatewise addition modulo 2. This implies that f assigns the same value to all x and $x + L$ pairs.

Suppose $f : \{0,1\}^n \to \{0,1\}^m$, where $m \geq n$ WLOG. We henceforth think of the outputs of f is terms of *colors*. An example of an L-periodic function f with $n = 3$ and $L = (101)$ is given by Table 1.4.

Problem 2 (Simon's Problem)**.** Given "black-box access" to a circuit Q_f which implements L-periodic function f, determine L. Here, "black-box access" refers to being able to observe the inputs and outputs of the function without knowing entirely what it does.

Classically, Simon's Problem is difficult to solve. Even when randomization is allowed, it requires at least $\sqrt{2^n} \approx 1.41^n$ applications of the circuit Q_f.

Theorem 3 (Simon's Theorem)**.** On a quantum computer, only $4n$ applications of Q_f are required to solve Simon's Problem.

How is the algorithm performed? First, we consider "loading the data", i.e., performing the first ROTATE and COMPUTE operations of the Fourier Sampling paradigm. This part of the circuit is shown in Fig. 1.8. By L-periodicity of the given functions, we note that the probability of observing each unique color assignment is $2/N$. Thus, at the end of Fig. 1.8, the measured output is some uniformly chosen color c^*; in the example of Table 1.4, if the measured output is Blue, the state collapses to

$$\frac{1}{\sqrt{2}}|010\rangle \otimes |\text{Blue}\rangle + \frac{1}{\sqrt{2}}|111\rangle \otimes |\text{Blue}\rangle$$

since $L = (101)$.

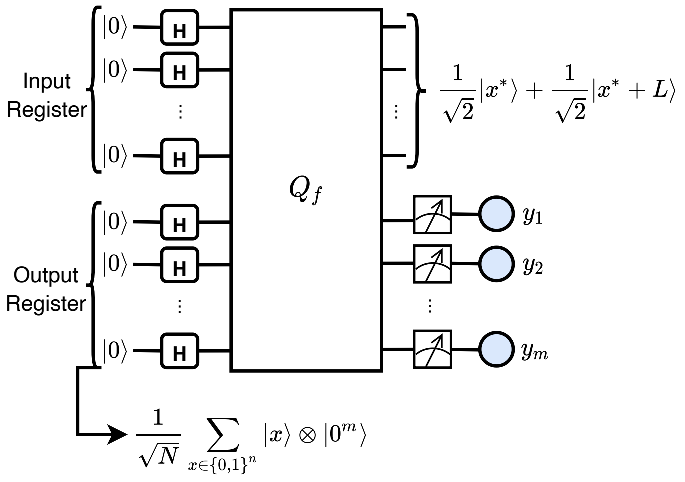

Input Register
$|0\rangle$
H
Q_f
$\frac{1}{\sqrt{2}}|x^*\rangle + \frac{1}{\sqrt{2}}|x^*+L\rangle$
Output Register
y_1
y_2
y_m
$\frac{1}{\sqrt{N}}\sum_{x\in\{0,1\}^n}|x\rangle\otimes|0^m\rangle$

Figure 1.8: The ROTATE I and COMPUTE parts of the circuit for Simon's Algorithm.

In general, for a given measured output color c^*, the state collapses to

$$\frac{1}{\sqrt{2}}|x^*\rangle\otimes|c^*\rangle+\frac{1}{\sqrt{2}}|x^*+L\rangle\otimes|c^*\rangle=\left(\frac{1}{\sqrt{2}}|x^*\rangle+\frac{1}{\sqrt{2}}|x^*+L\rangle\right)\otimes|c^*\rangle$$

where the original states x^* and x^*+L is the pair corresponding to color c^*.

Now, we feed the result of Fig. 1.8 into the ROTATE II phase, which comprises of applying $H^{\otimes n}$.

$$\begin{aligned}&H^{\otimes n}\left(\frac{1}{\sqrt{2}}|x^*\rangle+\frac{1}{\sqrt{2}}|x^*+L\rangle\right)\\&=\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{N}}\sum_{s\in\{0,1\}^n}(-1)^{x^*\cdot s}|s\rangle+\frac{1}{\sqrt{N}}\sum_{s\in\{0,1\}^n}(-1)^{(x^*+L)\cdot s}|s\rangle\right)\\&=\frac{1}{\sqrt{2N}}\sum_{s\in\{0,1\}^n}(-1)^{x^*\cdot s}|s\rangle=\end{aligned}\tag{1.56}$$

but note that

$$\left(1+(-1)^{L\cdot s}\right)=\begin{cases}2 & \text{if } L\cdot s=0 \bmod 2\\ 0 & \text{if } L\cdot s=1 \bmod 2\end{cases}$$

and so

$$(1.56)=\sqrt{\frac{2}{N}}\sum_{s:s\cdot L=0}(-1)^{x^*\cdot s}|s\rangle\tag{1.57}$$

The normalization condition of any quantum circuit outputs implies that (1.57) equals 1. Thus, there are exactly $N/2$ such output strings $s\in\{0,1\}$ which satisfy $s\cdot L=0$.

If the output of (1.57) is measured, we obtain an expression in terms of the bits of $L:=(L_1,\cdots,L_n)\in\{0,1\}^n$. For example, if $s=(0010\cdots0)$ such that $s\cdot L=0$, this implies $L_3=0$. In general, M repeated measurements of (1.57) yields M linear equations in terms of the bits of L.

Thus, we can repeat the entire procedure $n-1$ times and hope to obtain linearly-independent equations of the form $s\cdot L$, which we can then solve to obtain the L_i's. Each repetition costs $2n$ `HAD` gates, 1 application of Q_f, and n measurement gates. Note that there will always be at least two solutions to the system: $L = (0 \cdots 0)$ and the true underlying L of the function f. The best case scenario would occur if there are *exactly* two solutions.

To solve the system of equations, we need not use a quantum algorithm. A single application of classical Gaussian elimination costs approximately n^3 steps.

Proposition 1. The probability that the first $n-1$ equations collected through repeated measurements forms a linearly-independent system is at least $1/4$.

Proof. Assume the first $s^{(1)}, \cdots, s^{(i)}$, for some $i < n-1$, are already linearly-independent, and let $\mathrm{sp}(s^{(1)}, \cdots, s^{(i)})$ define their span. We must have $s^{(i+1)} \notin \mathrm{sp}(s^{(1)}, \cdots, s^{(i)})$ in order for $s^{(1)}, \cdots, s^{(i+1)}$ to be linearly-independent as well. This occurs with probability $1 - 2^i/2^{n-1}$, so by recursion, the overall probability when $i = n-1$ becomes

$$\begin{aligned}\mathbb{P}(\text{ all } n-1 \text{ linearly-independent}) &= \left(1 - \frac{1}{2^{n-1}}\right)\left(1 - \frac{2}{2^{n-1}}\right)\left(1 - \frac{4}{2^{n-1}}\right)\cdots\left(1 - \frac{2^{n-2}}{2^{n-1}}\right) \\ &= \frac{1}{2}\frac{3}{4}\frac{7}{8}\cdots\left(1 - \frac{1}{2^{n-1}}\right) \qquad (1.58)\end{aligned}$$

By virtue of $(1-a)(1-b) \geq (1-ab)$ for any $0 < a, b < 1$, the above becomes

$$(1.58) \geq \frac{1}{2}\left(1 - \frac{1}{4} - \frac{1}{8} - \cdots\right) = \frac{1}{2}\frac{1}{2} = \frac{1}{4}$$

■

Note that rearranging the terms of Proposition 1 and counting the number of components needed per repetition proves the claim in Theorem 3.

Fourier Transform over $\mathbb{Z}_N$, i.e. the integers mod $N \in \mathbb{N}$. Discusses the decomposition of $g : \mathbb{Z}_N \to \mathbb{C}$, functions more general than the Boolean ones we've considered previously in Fourier Sampling and other algorithms.

New set of "orthonormal basis" vectors $\chi_0, \cdots, \chi_{N-1} : \mathbb{Z}_N \to \mathbb{C}$, defined by the roots of unity

$$\chi_s(x) = \omega_N^{s \cdot x}, \quad \omega_N := e^{i\frac{2\pi}{N}} \qquad (1.59)$$

One crucial note: when $N = 2^n$, the Fourier transform which maps $|g\rangle$ to $\sum_{s=0}^{N-1} \hat{g}(s)|s\rangle$ is computable by a quantum circuit with approximately n^2 1 or 2-qubit gates.

As before, we associate functions $g : \mathbb{Z}_N \to \mathbb{C}$ to vectors

$$\frac{1}{\sqrt{N}}\begin{bmatrix} g(0) \\ g(1) \\ \cdots g(N-1) \end{bmatrix} \in \mathbb{C}^N$$

and also to quantum states

$$\frac{1}{\sqrt{N}}\sum_{x=0}^{N-1} g(x)|x\rangle$$

iff $(1/N)\sum_{x\in[0,N)}|g(x)|^2 = 1$. More precisely, the "orthonormal basis" (1.59) actually refers to the vector notation $|\chi_0\rangle, |\chi_1\rangle, \cdots, |\chi_{N-1}\rangle$. A matrix where the columns are composed of the $|\chi_0\rangle, |\chi_1\rangle, \cdots, |\chi_{N-1}\rangle$ will be $N \times N$ unitary.

Recall a feature of the previous `XOR` functions on the binary $\{0,1\}$ space, $\chi_s(x) = (-1)^{s\cdot x}$: we have $\chi_s(x \oplus y) = \chi_s(x)\chi_s(y)$.

Now what happens when we consider Simon's Algorithm over $\mathbb{Z}_N$ instead of $\mathbb{F}_2^n$?

Problem 3 (Simon's Problem on $\mathbb{Z}_N$)**.** Given "black-box access" to a circuit Q_f which implements L-periodic function f, where L divides N, determine L. On $\mathbb{Z}_N$, L-periodic means that for all $x \in \mathbb{Z}_N$, $f(x) = f(x+L) + f(x+2L) = \cdots$, otherwise the assignments are distinct.

Two key points to remember:

- this version of Simon's algorithm works for any N not necessarily a power of 2.
- the algorithm still works (more or less) despite L not being a factor of N, so f is only "mostly periodic".

The architecture of the ROTATE-COMPUTE-ROTATE paradigm is essentially the same as the binary case.

State collapses to

$$\sqrt{\frac{L}{N}} \sum_{x:f(x)=c^*} |x\rangle \otimes |c^*\rangle =: |g_{c^*}\rangle$$

for some $g_{c^*} : \mathbb{Z}_N \to \mathbb{R}$ such that

$$g_{c^*}(x) = \begin{cases} \sqrt{L} & \text{if } f(x) = c^* \\ 0 \text{ else} \end{cases}$$

where $|g\rangle$ is a state iff $(1/N)\sum_{x=[0,N-1]}|g(x)|^2 = 1$.

The new state is thus represented as

$$\sum_{s=0}^{N-1} \hat{g}_{c^*}(s)|s\rangle$$

Measure output color/register where each color appears N/L times (each color appears as a measurement outcome with probability $1/L$ conditioned on measuring c^* as the color). This means each $s \in \mathbb{Z}_N$ reads out with probability $|\hat{g}_{c^*}(s)|^2$.

Recall a property of Simon's algorithm for binary inputs, which also holds for the more general $\mathbb{Z}_N$ case: $|g_{c^*}|^2$ does not depend on c^*.

1.3.2 Shor Factorization

Shor's factoring algorithm pertains to the problem of factoring integers. The classical interpretation manages to achieve this task at a time which is exponential in the number of bits. However, most of Shor's algorithm can be discussed in terms of classical computation and number theory.

Original	Remainder (mod N)
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	11
2^6	1

Table 1.5: A simple demonstration of solving Problem 4.

Let $B \in \mathbb{N}$ be a large integer which can be encoded into $m \in \mathbb{N}$ bits. We are interested in the prime factorization of B.

Suppose for simplicity, $B = PQ$ where $P, Q \in \mathbb{N}$ are prime numbers. The trick is to find a nontrivial square-root of 1 mod B, i.e., R such that $R^2 \equiv 1 \pmod{B}$. This implies $R \equiv \pm 1 \pmod{B}$. Thus, either P divides $R-1$ and Q divides $R+1$, or P divides $R+1$ and Q divides $R-1$. The problem turns into one of computing the GCD of $R+1$ and B, which yields either Q (or P, WLOG).

Problem 4. Find x such that $x \not\equiv \pm 1(\bmod x)$ but $x^2 \equiv 1(\bmod N)$. Then N divides $(x+1)(x-1)$, but N does not divide $x \pm 1$.

Example 6. Consider $N = 21$ and $x = 2$. From Table 1.5, we can see that $(2^3)^2 \equiv 1(\bmod 21)$, which implies that 8 is a nontrivial square which solves Problem 4.

In general, this example demonstrates the following procedure. Pick x at random and continue exponentiating it until $r \in \mathbb{N}$ such that $x^r \equiv 1(\bmod N)$. If we are lucky, r is an even number, and if we are even luckier, we find that $x^{r/2} \not\equiv \pm 1(\bmod N)$. The following lemma shows that the probability of being "luckier" is at least $1/2$.

Lemma 1. Let N be an odd composite, with at least two distinct prime $P, Q \in \mathbb{N}$ such that $N = PQ$, and let $x \in \{0, 1, \cdots, N-1\}$ be randomly chosen. If $\gcd(x, N) = 1$, then with probability at least $1/2$, the order r of $x(\bmod N)$ is even, and $x^{r/2} \not\equiv \pm 1(\bmod N)$.

If $\gcd(x, N) \neq 1$, then we have already successfully determined a factor of N!

Why is this procedure helpful from a quantum perspective? Note that if we had continued to exponentiate x in Table 1.5, we would have begun to observe repeating values. In other words, exponentiating and taking the mod is a *periodic function*. Moreover, the period is exactly r, the desired order. Thus, the problem of integer factorization reduces to the period-finding problem, which could be solved with Simon's algorithm and Fourier Sampling!

Overall, *Shor's algorithm* is simply an application of Simon's algorithm to the integer factorization problem. If we obtain a value of r which is "unlucky", then we can simply repeat the procedure again. By Lemma 1, the number of repetitions needed until we are "lucky" is, on average, at most 2 (thinking of it as a Geometric random variable).

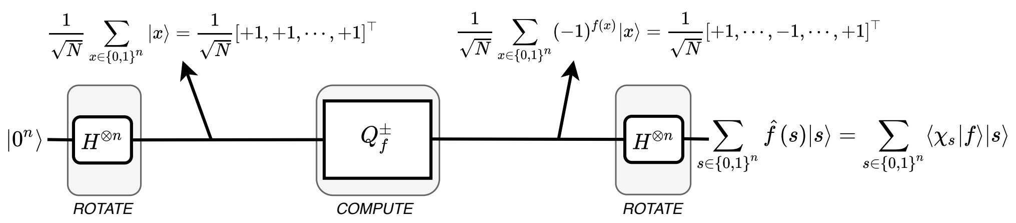

$\frac{1}{\sqrt{N}}\sum_{x\in\{0,1\}^n}|x\rangle = \frac{1}{\sqrt{N}}[+1,+1,\cdots,+1]^\top$
$\frac{1}{\sqrt{N}}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}|x\rangle = \frac{1}{\sqrt{N}}[+1,\cdots,-1,\cdots,+1]^\top$
$|0^n\rangle$
$H^{\otimes n}$
$Q_f^\pm$
$H^{\otimes n}$
$\sum_{s\in\{0,1\}^n}\hat{f}(s)|s\rangle = \sum_{s\in\{0,1\}^n}\langle\chi_s|f\rangle|s\rangle$
ROTATE
COMPUTE
ROTATE

Figure 1.9: A direct application of the ROTATE-COMPUTE-ROTATE paradigm, posed as a potential implementation of Grover's Algorithm.

1.3.3 Grover's Algorithm

Problem 5 (Grover's Problem)**.** Given implicitly represented data of N bits, find a '1'. Here, the implicit representation is in terms of a truth table of a Boolean function or circuit $C : \{0,1\}^n \to \{0,1\}$.

One application of Grover's Problem is in determining the correct binary string x^*, among all $N = 2^n$ combinations, which is the secret code to a lock.

Typically, for a "black-box query model":

- Deterministic algorithms need at most N applications of C.
- Randomized algorithms need at most N applications.

As we'll see with Grover's algorithm, quantum algorithms need at least $\sqrt{N}$ queries.

A more formal restatement of Grover's Problem. Assume we are given a quantum circuit Q_f implementing $f : \{0,1\}^n \to \{0,1\}$, and we want to find $x \in \{0,1\}^n$ such that $f(x) = 1$.

For now, suppose $f(x) = 1$ for exactly one string $x^* \in \{0,1\}^n$, and that we want to find x^*.

Consider the sign-implementing version $Q_f^\pm$, which maps $|x\rangle \to (-1)^{f(x)}|x\rangle$.

$$\begin{bmatrix}\alpha_{0\cdots0}\\ \alpha_{0\cdots1}\\ \vdots\\ \alpha_{x^*}\\ \vdots\alpha_{1\cdots1}\end{bmatrix} \to \begin{bmatrix}\alpha_{0\cdots0}\\ \alpha_{0\cdots1}\\ \vdots\\ -\alpha_{x^*}\\ \vdots\alpha_{1\cdots1}\end{bmatrix}$$

What happens when we apply the ROTATE-COMPUTE-ROTATE paradigm, as in Fig. 1.9? Define $\mu := (1/N)\sum_{x\in\{0,1\}^n} f(x) \equiv \hat{f}(0,\cdots,0)$. Note that the amplitude ("coefficient") of the sth term in the output sum yields

$$\langle\chi_s|f\rangle := \frac{1}{N}\sum_{x\in\{0,1\}^n} f(x)\texttt{XOR}_s(x)$$

However, when this amplitude is considered for $s = (0,\cdots,0)$, we obtain $\mu = 1-(2/2^n)$, which is extremely close to 1. Thus, the circuit Fig. 1.9 only returns the all-zero string $(0,\cdots,0)$ with near certainty. We obtain no information about x^*.

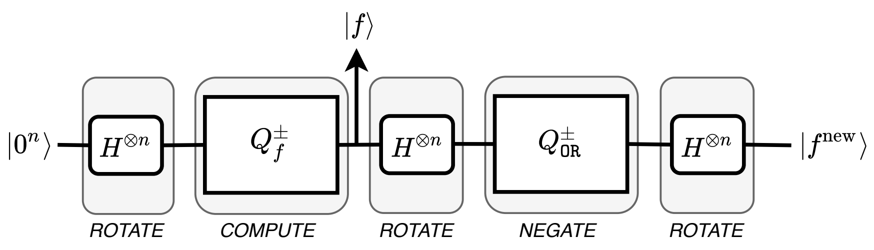


Figure 1.10: The corrected version of Fig. 1.9, which corresponds to the true Grover's Algorithm.

Grover's Trick: Consider the deviation of each function output f away from the mean μ, i.e. $|f\rangle = \mu|\chi_{0\cdots 0}\rangle + |f^{\text{dev}}\rangle$. From a vector viewpoint:

$$\frac{1}{\sqrt{N}}\begin{bmatrix} f(0,0,\cdots,0) \\ \vdots \\ f(1,1,\cdots,1) \end{bmatrix} = \frac{1}{\sqrt{N}}\begin{bmatrix} \mu \\ \mu \\ \vdots \\ \mu \end{bmatrix} + \frac{1}{\sqrt{N}}\begin{bmatrix} f^{\text{dev}}(0,0,\cdots,0) \\ \vdots \\ f^{\text{dev}}(1,1,\cdots,1) \end{bmatrix}$$

Define $f^{\text{new}} := \mu - f^{\text{dev}}$. Note that

$$\hat{f}^{\text{new}}(s) = \begin{cases} \hat{f}(0,0,\cdots,0) & \text{if } s = 0^n \\ -\hat{f}(s) & \text{if } s \neq 0^n \end{cases}$$

In order to use a quantum circuit for transforming $|f\rangle$ to $|f^{\text{new}}\rangle$, we require two additional steps after the usual ROTATE-COMPUTE-ROTATE scheme from Fig. 1.9:

- Negate all the amplitudes except the one on $|00\cdots 0\rangle$, i.e.

$$|s\rangle \to \begin{cases} |s\rangle & \text{if } s = (0,0,\cdots,0) \\ -|s\rangle & \text{else} \end{cases}$$

  We define this computation by the circuit $Q_{\texttt{OR}}^{\pm}$, since the $\texttt{OR}$: $\{0,1\}^n \to \{+1,-1\}$ operation negates signs: $(-1)^{\texttt{OR}(x)} = +1$ if $x = (0,0,\cdots,0)$ and -1 otherwise.

- Invert by applying $H^{\otimes n}$ once more.

Altogether, the circuit Fig. 1.9 becomes Fig. 1.10

A visualization of how the circuit Fig. 1.10 varies the amplitudes to extract x^* is shown in Fig. 1.11. In summary, ignoring the slight "fudging" of μ not being precisely equal to $1/\sqrt{N}$, each iteration of $Q_f^{\pm}$, $H^{\otimes n}$, $Q_{\texttt{OR}}^{\pm}H^{\otimes n}$, the amplitude increases in a sequence

$$\frac{1}{\sqrt{N}} \to \frac{3}{\sqrt{N}} \to \cdots \to \frac{2T+1}{\sqrt{N}}$$

After T iterations, the amplitude at $|x^*\rangle$ becomes decently high, and we can measure it with high probability. Roughly, $T \approx \sqrt{N}2$. The variation in amplitude for the other $x \neq x^*$ (i.e., "fudge" factor) is close to $2/2^n$, and negligible.

The same approach can be used in the case where there is more than one location $1 \leq i \leq n$ in x^* such that $x_i^* = 1$. Now, the single amplified string corresponds to the x^* string.

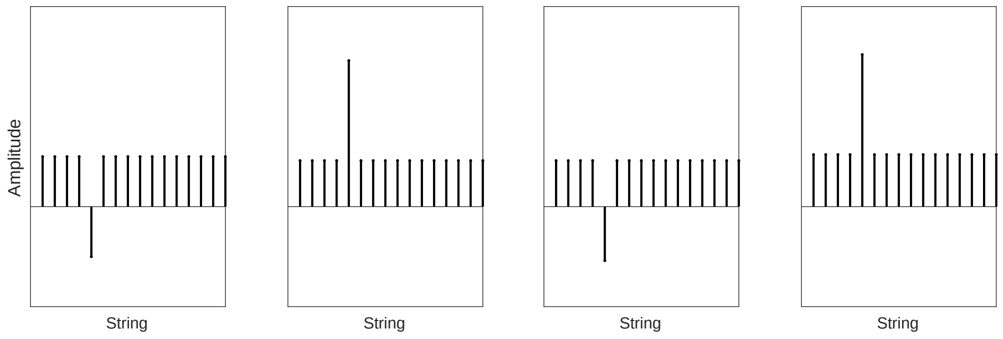


Figure 1.11: Sample visualization of amplitude variations through repeated applications of $Q_f^{\pm}$ and $Q_{\texttt{OR}}^{\pm}$ in Fig. 1.10. The x-axis consists of all the $N = 2^n$ possible combinations of n-bit strings which is the correct one.

Chapter 2

Neural Networks for Spatiotemporal Data

Classical and quantum implementations of a class of neural networks designed specifically to process spatiotemporal data, including convolutional neural networks, graph neural networks, and recurrent neural networks.

Hierarchy: Artificial Intelligence $\supset$ Machine Learning $\supset$ Neural Networks $\supset$ Deep Learning
Neural Networks make up the backbone of *Deep Learning* algorithms. In deep learning, neural networks must have *more than three* node layers (depth). Furthermore, deep learning algorithms typically automate much of the feature extraction process, whereas more beginner machine learning algorithms use structured data (which features to consider are labeled and structured in the data).

2.1 Classical vs. Quantum Neurons

We begin with a discussion of the most basic unit in a neural network: the neuron. First, we recall the classical implementation of the neuron, defined as follows.

Definition 12 (Neuron)**.** Given a collection of inputs $\{x_i\}_{i=1}^n$, a bias term $b \in \mathbb{R}^{\geq 0}$, and weights $\{w_i\}_{i=1}^n$, the *neuron* is a function which yields outputs of the following form:

$$z := g(y) := g\left(\sum_{i=1}^{n} w_i x_i - b\right) \tag{2.1}$$

where $g : \mathbb{R} \to \mathbb{R}$ denotes the nonlinear *activation function.* Activation functions are often applied in order to be able to handle nonlinearly-separable data.

Various types of activation functions g yield different types of neurons. Below are some of the most common activation functions found in the literature:

- *perceptron*: the activation function g is defined as

$$z = \begin{cases} 1 & \text{if } \sum\limits_{i=1}^{n} w_i x_i \geq b \\ 0(\text{ or } -1) & \text{else} \end{cases} \tag{2.2}$$

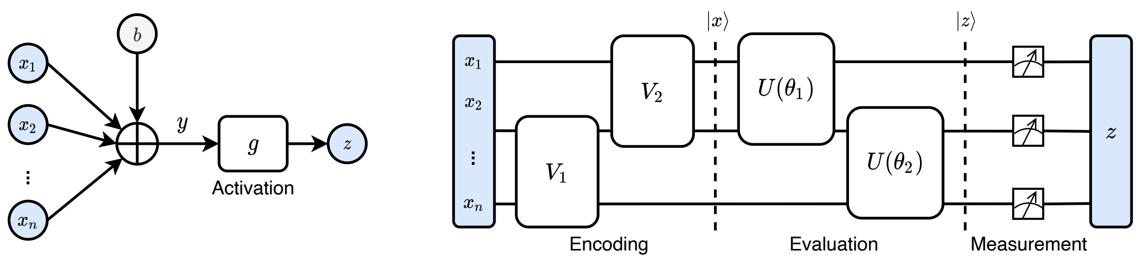


Figure 2.1: High-level implementation of a neuron. [Left] Classical. [Right] Quantum.

- *sigmoid*: the activation function g is defined as

$$z = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}+b}} \tag{2.3}$$

- *tanh*: the activation function g is defined as

$$z = \tanh(-\mathbf{w}^T\mathbf{x} + b) = \frac{e^{\mathbf{w}^T\mathbf{x}-b} - e^{-\mathbf{w}^T\mathbf{x}+b}}{e^{\mathbf{w}^T\mathbf{x}-b} + e^{-\mathbf{w}^T\mathbf{x}+b}} \tag{2.4}$$

  Essentially, the tanh function is just a stretched version of the sigmoid to a range larger than $(0, 1)$.

- *rectified linear unit (ReLU)*: a piecewise linear function which outputs the input $x \in \mathbb{R}$ directly if $x > 0$, otherwise zero if $x \leq 0$. Both the sigmoid and hyperbolic tangent activation functions cannot be used in neural networks with large depth because of the *vanishing gradient problem*. However, the ReLU function overcomes this problem, and is thus the default activation function used with developing multilayer perceptron models and CNN models.

- *softmax*: converts a vector of numbers (retrieved from the output layer of a neural network) into a vector of probabilities. The terminology of "softmax" is similar to "argmax", and this is not a coincidence. Argmax is a special case of softmax where the highest probability value is assigned a 1 and the remaining values are assigned to zero, hence extracting the most probable argument in a vector of values.

Roughly, the distinction between the classical neuron and its quantum analogue is illustrated in Fig. 2.1. In the quantum implementation (right of Fig. 2.1):

- V_1 is the operator which prepares the initial state of the ancilla bits
- V_2 is the operator which prepares the quantum dataset
- $U(\theta_1)$ is the operator which computes the linear $\mathbf{w}^T\mathbf{x}$
- $U(\theta_2)$ approximates the nonlinear activation function

and θ_1, θ_2 refer to parameters of the circuit. In the following, we make concrete what these operators exactly do.

The *quantum neuron (quron)* is a quantum implementation of Definition 12, which means that one clear distinction is the ability to process superposition inputs. There are various different ways to implement the quron in literature [1, 2]. However, the key challenge faced by all these implementations is the representation of the nonlinear, dissipative activation functions in the linear, reversible quantum setting.

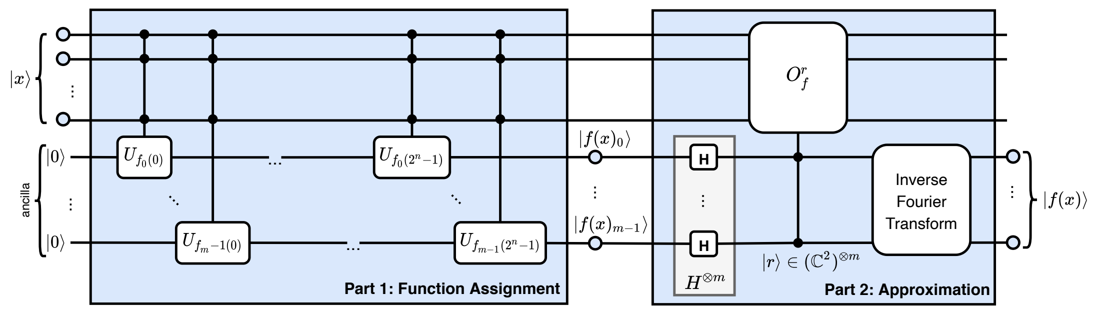


Figure 2.2: The quantum circuit corresponding to the approximate evaluation of a nonlinear Boolean function.

First, we discuss how to approximate nonlinear functions via quantum circuits. Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function mapping n-bit strings to a single bit, and suppose Q_f is the quantum circuit which implements f via the structure at the top of Fig. 1.6:

$$Q_f : |x\rangle|b\rangle \to |x\rangle|b \oplus f(x)\rangle$$

where $|x\rangle \in (\mathbb{C}^2)^{\otimes n}$ is the n-bit superposition input, and $|b\rangle \in \mathbb{C}^2$ is the single-bit superposition output.

Nonlinear function approximation is generally achieved using a circuit architecture composed of two parts: 1) function assignment, and 2) approximation. The circuit is visualized in Fig. 2.2, and can be roughly described as follows. Let U_f be the quantum circuit for a general Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ mapping n-bit inputs to m-bit outputs.

$$U_f : |x\rangle|0\rangle^{\otimes m} \to |x\rangle|f(x)\rangle \tag{2.5}$$

Note that the output registers are specifically initialized to be the all-zero string. This is so that we can easily reconstruct the approximation of $f(x)$

$$f(x) = |f(x)_{m-1}\rangle \cdot 2^{m-1} + \cdots + |f(x)_0\rangle \cdot 2^0 \tag{2.6}$$

where $|f(x)_i\rangle$ represents the value of the ith bit in the m-bit binary expansion of $f(x)$.

Now we discuss each of the two parts in further detail. The first part of the circuit involves *function assignment*. In Fig. 2.2, the operator U_f from (2.5) is split into multiple operators of the form $U_{f_i(j)}$, where $i \in \{0, \cdots, m-1\}$ refers to the output bit that the operator acts upon, and $j \in \{0, \cdots, 2^n - 1\}$ refers to the *digit* (note, not *bit*) of the input that the function is applied upon. Each operator $U_{f_i(j)}$ is one of two simple operators $\{\texttt{I}, \texttt{NOT}\}$, where $\texttt{I}$ is the identity operator, and $\texttt{NOT}$ is the bit-flip NOT gate.

The second part of the circuit involves *approximation*. We construct a minimum phase-change operator of the form

$$O_f = \sum_{x=0}^{2^n-1} e^{\frac{2\pi i}{2^m} f(x)} |x\rangle\langle x|$$

which essentially adds a phase factor of $|f(x)\rangle$ to all possible n-bit inputs $|x\rangle$. In Fig. 2.2, the operator O_f is applied as a series of controlled $O_f^{2^i}$ gates, for $i \in \{0, \cdots, m-1\}$ depending on the value of the m-bit string $|r\rangle$. Afterwards, the inverse Fourier transform is used to approximately recover $f(x)$ as (2.6).

2.2 Classical Neural Networks

2.2.1 Convolutional Neural Networks

Remark 7 (Common Training Techniques)**.**

- Gradient descent is used to find locally optimal weights in the neural network such that we can minimize the loss function.
- Overfitting to a particular training dataset is one of the most prevalent issues surrounding deep learning neural networks. One of the most computationally cheap ways of regularization used to overcome overfitting is *dropout*, which randomly "drops out" neurons in the network during training in order to simulate the effect of varying neural network architectures. In a Dense neural network, dropout drops the nodes, but for the leftover nodes, the connections are still full.
- Another way of regularization is to do *early stopping*, which is to stop training before we have a chance of overfit. A key difference between early stopping and cross-validation is that cross-validation divides the training data, while early stopping divides the training iterations. One may think of the two distinctions as a regularization over space versus regulariation over time.

A *neural network* takes the structure of

$$X_1 \xrightarrow[W_1]{X}{}_2 \xrightarrow[W_2]{\cdots} \xrightarrow[W_L]{Z}$$

Here, X_ℓ represents a *layer* of neurons, where $\ell \in \{1, \cdots, L\}$ for total number of layers $L \in \mathbb{N}$. Moreover, W_ℓ denotes the weights and Z denotes the output.

For the specific *convolutional neural network (CNN)*, $X_\ell \in \mathbb{R}^{H_\ell \times W_\ell \times D_\ell}$ is a tensor representing the ℓth layer, where $X_\ell(h, w, d)$ is the element in spatial location (h, w) along channel d. For a standard RGB image, $D_\ell = 3$ for all ℓ, H is the number of pixels along the image's height, and W is the number of pixels along its width.

The CNN is often composed of two types of layers: 1) the *convolution layer* and 2) the *max-pooling layer.*

Definition 13 (Convolution Layer)**.** A *convolution layer* in a CNN summarizes the presence of certain features in an input image. Each layer ℓ takes a *convolution kernel*, also called a *filter*, $F \in \mathbb{R}^{H' \times W' \times D}$, where $H' < H_\ell, W' < W_\ell$ and convolves it across the layer. When the filter is applied by sliding it over the layer one element at a time, the resulting image has dimension $(H_\ell - H' + 1) \times (W_\ell - W' + 1) \times D_\ell$.

Definition 14 (Pooling Layer)**.** A *pooling layer* downsamples the result of the convolution layer by further summarizing the presence of features. There are two types of pooling schemes which are typically used in practice: 1) *max-pooling*, and *average-pooling.*

$$\text{[Max]} \quad X_{\ell+1}(h_{\ell+1}, w_{\ell+1}, d) = \max_{\substack{0 \le h_\ell \le H_\ell \\ 0 \le w_\ell \le W_\ell}} X_\ell(h_{\ell+1}H' + h_\ell) \tag{2.7a}$$

$$\text{[Avg]} \quad X_{\ell+1}(h_{\ell+1}, w_{\ell+1}, d) = \frac{1}{HW} \sum_{\substack{0 \le h_\ell \le H_\ell \\ 0 \le w_\ell \le W_\ell}} X_\ell(h_{\ell+1}H' + h_\ell) \tag{2.7b}$$

Here, $H' < H_\ell$ and $W' < W_\ell$ refer to the *window size* of the pooling scheme for the ℓth layer.

Packages and Implementation Details: Typically use `tensorflow` and `keras` to implement in Python.

```
model = Sequential()
model.add(Dense(8, input_shape=(8,), activation = 'relu'))
model.add(Dense(4, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.summary()
```

Two types of APIs in `keras`: `Sequential` and `Functional`. In CS155, only the `Sequential` API was used for the code see hw4).

However, the `Sequential` model cannot be used to implement networks such as the *recurrent neural network* because **it does not allow you to backpropagate across more than one layer**. It also only allows you to map **one input to one output, not multiple inputs and outputs**. For example, in the above code, `Sequential` yields $D8 \to D4 \to D1$, but it cannot be rewired to anything else, e.g. $D4 \to D8$ or $D1 \to D8$.

```
layer1 = Input(shape = (8,))
layer2 = Dense(8, activation = 'relu')(layer1)
layer3 = Dense(4, activation = 'relu')(layer2)
output = Dense(1, activation = 'sigmoid')(layer3)
model = Model(inputs = layer1, outputs = output)
model.summary()
```

Case Study: MNIST Dataset. See `0_intro_expmt.py`. Train on 60000 photos of 28×28 grayscale images of digits from 0 to 9, then test on 10000 photos. Each photo is represented as a 28×28 matrix of values from 0 to 1 (0 to 255 normalized). Vectorize the images into 784×1 column vectors to feed as input to the neural net.

Case Study: CIFAR-10. See `1_convolutional_nn.py`. Train on 782 photos of 32×32 colored images (3 channels RGB) of random objects and animals, then test on 157 photos. Each photo is basically a tensor of three dimensions, $(32, 32, 3)$.

Definition 15 (One-Hot Encoding)**.** *One-hot encoding* enumerates individual elements in the set of possible labels. For example, in the MNIST dataset, there are 10 total labels from 0 to 9. With one-hot encoding, we introduce 3 as $[0, 0, 0, 1, 0, \cdots, 0]$.

2.2.2 Recurrent Neural Networks and LSTMs

Recurrent neural networks (RNNs) are designed such that outputs from previous timesteps are able to be used as inputs, and are a more suitable deep learning architecture in datasets with a notion of time (as a sequence). For example, instead of a single photo of a ball, we have a sequence of photos which depict the ball rolling to the side. A single neuron with *recurrence* emphasizes this input over time. A sequence of data over time is processed by this neuron according to the subfigure on the right of Fig. 2.3, which can be represented simply by introducing a *feedback loop* (see the subfigure on the left). This feedback loop can also be viewed as a recurrence relation $y_t = f(x_t, h_{t-1})$, where x_t is the input state at time t, h_{t-1} is an internal state which describes the relationship between $x_{t-1}$and x_t, and y_t is the output.

Definition 16 (Recurrent Neural Networks)**.** A *recurrent neural network* maintains an internal state (i.e., *cell state*) h_t which is updated at each timestep according to the reccurrence relation $h_t = f_W(x_t, h_{t-1})$ with W being the set of weights in the neural network. This set of weights W is constant over all time t.

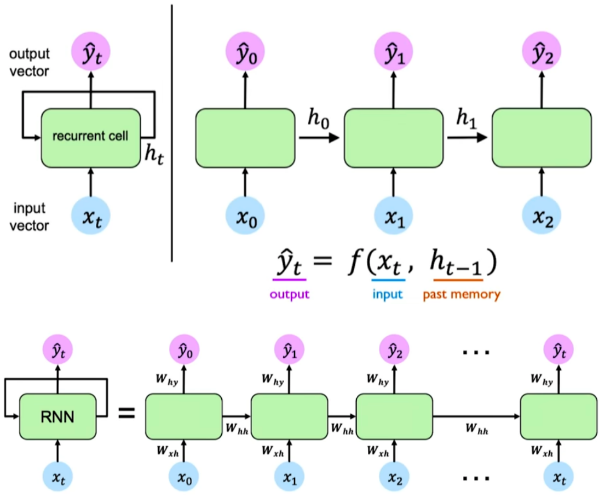


Figure 2.3: [Top] A single neuron with recurrence involved. [Bottom] A detailed model of a single neuron in the recurrent neural network. Credit to towardsdatascience.com. [to be replaced.]

How to update the recurrent neural network? The same loss function as in the standard sequential neural network, but the loss function is computed at every sequence in time. Then we accumulate the loss function values over time into a single large loss function, and optimize over that.

Backpropagation through Time (BPTT). Due to the ability of RNNs to be unrolled into standard feedforward networks (FFNs) (shown in Fig. 2.3), the backpropagation algorithm for RNNs is a simple extension of backpropagation for FFNs.

Define the input state to be $\mathbf{x}_t \in \mathbb{R}^n$, hidden state $\mathbf{h}_t \in \mathbb{R}^m$, output state $\mathbf{o}_t \in \mathbb{R}^d$, and (scalar) desired label $y_t \in \mathbb{R}$. Here, we assume a single hidden layer for simplicity, and it is unrolled over a horizon of time as illustrated in Fig. 2.3. Define the input-to-hidden weight matrix to be $W_{hx} \in \mathbb{R}^{m\times n}$, hidden-to-hidden weight matrix to be $W_{hh} \in \mathbb{R}^{m\times m}$, and hidden-to-output weight matrix to be $W_{oh} \in \mathbb{R}^{d\times n}$. Then the forward propagation equations are written as

$$\mathbf{h}_t = f(W_{hx}\mathbf{x}_t + W_{hh}\mathbf{h}_{t-1}) + \mathbf{b}_h, \quad \mathbf{o}_t = g(W_{oh}\mathbf{h}_t + \mathbf{b}_o)$$

where the $\mathbf{b}$ are bias terms of the appropriate dimensions, and f and g are activation functions. For RNNs, f is typically chosen to be the tanh function.

The weight matrices are trained via BPTT to optimize a loss function of the form

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T} \ell(\mathbf{o}_t, y_t) \tag{2.8}$$

Because there are so many gradients to account for in a recurrent neural network, the vanishing gradient problem is especially prevalent. One common trick to bypass this issue is to use a more complex recurrent unit called a *gated cell*, which uses *gates* to selectively add or remove information within each recurrent unit. Examples of gated cells are *long short-term memory (LSTM) units* and *gated recurrent units (GRUs)*.

Definition 17 (LSTM)**.** Compared to the standard recurrent neural network, the *LSTM network* 1) maintains the cell state h_t, 2) uses gates to control the flow of information, and 3) does backpropagation through time with partially uninterrupted gradient flow. In particular, to control the flow of information, the LSTM uses a *forget gate* which removes irrelevant information; the remaining relevant information is stored from the current input and used to selectively update the cell state. The output is then a filtered version of the original cell state whcih would have been obtained using a regular recurrent neural network.

Gating is simply multiplying elementwise with proportions in the range $[0, 1]$. This is why the tanh activation function is often more usefule in recurrent neural networks than any other neural network. Think of the inputs as varying volumes of water, and the gate weights as faucets which control the amount of water which is allowed to flow through.

Three main bottlenecks in recurrent neural networks: 1) encoding bottleneck, 2) no parallelization in computation, 3) longer memories require larger computation. Although one-hot encoding is used in recurrent neural networks, one can imagine the computational burden associated with encoding an entire dictionary of words.

Definition 18 (Attention)**.** One way to try and remove the three primary bottlenecks of recurrent neural networks. Focus attention on the most important parts of an input. For example, in a photo of a person standing in a park, our brains can immediately pick out the important subject of the photo (the person) from the less important background (the park). As another example, searching for a keyword in a large database such as Google or Youtube; compute similarity between query and key via *attention weighting*.

2.2.3 Graph Neural Networks

Graph neural networks (GNNs) seek to achieve combinatorial generalization by learning structured representations of data and manipulating these structures through relational reasoning. An example of this task in action is in the way humans are able to construct a limitless combination of new sentences out of finite number of words ("combinatorial generalization"), while respecting the rules of grammar ("relational reasoning"). We refer the interested reader to [3] for a gentle tutorial on GNNs. Here, we present append of the basics described in [3] with some mathematical derivations, and an example application (inspired by [Link]) on predicting the topic of a scientific paper given its citation network and the most frequent words it uses.

The graph representation $\mathcal{G} := (\mathcal{V}, \mathcal{E}, \mathbf{u})$ typically used for GNNs are as follows:

- nodes $\mathbf{v} \in \mathcal{V}$ correspond to *entities*, or elements with attributes (e.g., road sensors, or social media users)
- edges $\mathbf{e} \in \mathcal{E}$ correspond to *relations*, or properties between two entities (e.g., there is a nonempty flow of traffic that passes between two road sensors, or two social nedia users are Friends)
- parts of the graph are passed through *rules*, or functions which map entities and relations to other entities and relations (e.g., binary logical questions such as "are two sensors within X distance of each other?" or "do two users share the same Friend?")

The $\mathbf{u}$ represents a vector of global attributes which are shared across the entire graph $\mathcal{G}$. We further distinguish the edges to be directed, and for each $\mathbf{e}_k \in \mathcal{E}$, $k \in \{1 \cdots, |\mathcal{E}|\}$, we designate the ID of the sender and receiver nodes s_k and r_k, respectively, where $s_k, r_k \in \{1, \cdots, |\mathcal{V}|\}$.

A single GNN block contains three "update" functions and three "aggregation" functions, which are performed in an interleaving fashion.

1. for $k \in \{1, \cdots, |\mathcal{E}|\}$, update edge attributes: $\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$, where $\mathcal{E}' = \{\mathbf{e}'_k\}$ is the set of updated edge attributes.
2. for $i \in \{1, \cdots, |\mathcal{V}|\}$, aggregate edge attributes per node: $\overline{\mathbf{e}}'_i = \rho^{e \to v}(\mathcal{E}'_i)$, where $\mathcal{E}'_i \triangleq \{\mathbf{e}_k \in \mathcal{E}' : r_k = i\}$.
3. for $i \in \{1, \cdots, |\mathcal{V}|\}$, update note attributes: $\mathbf{v}'_i = \phi^v(\overline{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$.
4. Aggregate global edge attribute: $\overline{\mathbf{e}}' = \rho^{e \to u}(\mathcal{E}')$.
5. Aggregate global node attribute: $\overline{\mathbf{v}}' = \rho^{v \to u}(\mathcal{V}')$, where $\mathcal{V}' = \{\mathbf{v}'_i\}$ is the set of updated node attributes.
6. Update the global attribute: $\mathbf{u}' = \phi^u(\overline{\mathbf{e}}', \overline{\mathbf{v}}', \mathbf{u})$.

The result which is returned is $(\mathcal{V}', \mathcal{E}', \mathbf{u}')$. A key requisite of the aggregation functions ρ is that they must be invariant to permutations of their inputs, e.g., summation, maximization, average.

We may frame the procedure above using the popular *message-passing* schematic, introduced by [4]. Let $G \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ be the adjacency matrix of the graph $\mathcal{G}$ with zeros on the diagonal entries, and let $X \in \mathbb{R}^{|\mathcal{V}| \times d}$ be the feature matrix with $d \in \mathbb{N}$ features. Then the *node representations* computed at layer $k \in \mathbb{N}$ is given by

$$H^{(k)} = \varphi(G, H^{(k-1)}, W^{(k)}), \qquad H^{(0)} = X \tag{2.9}$$

where φ is the message propagation function, and $W^{(k)}$ are the weights to be learned for the kth layer. As a specific example, one popular choice of the function φ is as follows:

$$H^{(k)} = \text{ReLU}(\tilde{D}^{-1/2}\tilde{G}\tilde{D}^{-1/2}H^{(k-1)}W^{(k-1)}) \tag{2.10}$$

where $\tilde{G} := I_{|\mathcal{V}|} + G$, and $\tilde{D}$ is the diagonal degree matrix with entries $\tilde{D}_{ii} := \sum_j \tilde{G}_{ij}$ for each $i \in \mathcal{V}$. he message-passing implementation prescribed by (2.10) is often used in *graph convolutional networks.*

As a concrete example, we now describe how the above steps and functions translate to the specific example of paper topic classification. The specific form of GNN implemented here is the *graph convolutional neural network (GCNN)*, which is a popular variant of GNNS first introduced by [5].

Data The Cora dataset consists of $N = 2708$ machine learning papers which fall under one of $M = 7$ different topics ('Genetic Algorithms', 'Reinforcement Learning', etc.). The citation graph among these N papers are such that every paper cites or is cited by at least one other paper in the dataset. The vocabulary of common words used in these papers has size $V = 1433$. The dataset is organized into two separate files:

- `.content`, which maps each paper ID $i \in \{1, \cdots, N\}$ to two objects: 1) a word attribute vector $\mathbf{w}_i \in \{0,1\}^V$ where $w_{ij} = 1$ if the jth word is present in paper i, 0 otherwise, and 2) the topic label vector $\mathbf{c}_i \in \{0,1\}^M$.
- `.cites`, which encodes the citation graph of the N papers, in the format of nonzero entry pairs (i, j) for two papers $i, j \in \{1, \cdots, N\}$ such that paper j cites paper i.

A citation graph of a subset of 1500 papers is generated and visualized in Fig. 2.4.

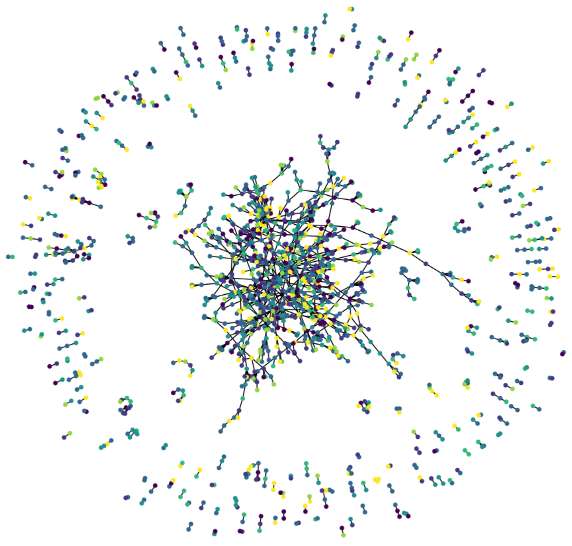


Figure 2.4: The citation graph for a smple of 1500 papers from the original dataset. Nodes are colored according to one of the M topics.

Preprocessing We split `.content` into *training* and *test* datasets at approximately a half-half partition of the original dataset. We store the values within matrices $Z_{\text{train}} \in \{0,1\}^{N_{\text{train}} \times V+M}$ and $Z_{\text{test}} \in \{0,1\}^{N_{\text{test}} \times V+M}$, where $N_{\text{test}} = N - N_{\text{train}}$. The columns of the training and test datasets are further divided into submatrices for the observed features $X_\chi \in \{0,1\}^{N_\chi \times V}$ (i.e., the occurrence of certain words in the paper) and the classification labels $Y_\chi \in \{0,1\}^{N_{\text{chi}} \times M}$ (i.e., the topic of the paper), for placeholders $\chi \in \{\text{train}, \text{test}\}$.

Graph Construction The graph is constructed straightforwardly from `.cites`. The adjacency matrix $G \in \{0,1\}^{N \times N}$ is such that $G_{ij} = 1$ if paper i is cited by paper j. A `graph_info` tuple is created containing the node features (i.e., the word attribute vectors $\{\mathbf{w}_i\}$) and the directed edges of the graph. Note that this graph is created for the entire dataset, irrespective of training or testing data.

The Neural Network Architecture Tensorflow allows for the construction of custom models via *model subclassing*, as well as the construction of custom layers. For this example, we build a network using two *graph convolution layers* with *skip connections* interleaved; a diagram of the architecture is shown in Fig. 2.5 and the corresponding pseudocode for the forward pass is below:

```
    def call(self, input_node_indices):
        x = self.preprocess(self.node_features)

        x1 = self.conv1((x, self.edges, self.edge_weights))
        x = x1 + x # skip connection

        x2 = self.conv2((x, self.edges, self.edge_weights))
        x = x2 + x # skip connection

        x = self.postprocess(x)

        node_embeddings = tf.gather(x, input_node_indices)

```

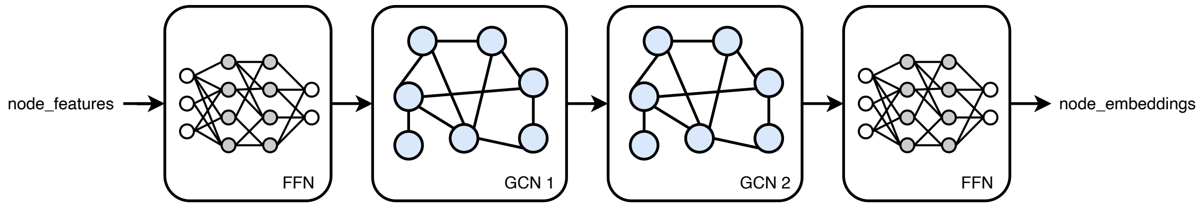


Figure 2.5: The graph neural network architecture used for the paper topic classification example.

```
        return self.compute_logits(node_embeddings)
```

The syntax `tf.gather(x, indices)` simply retrieves the values of the `Tensor` x at the indices `indices`, and stores them in an array. `compute_logits(node_embeddings)` is similar to a Softmax operator: it is used to transform the output of the neural network into a vector $\mathbf{c}_i$ which classifies paper i as one of the M topics.

Juxtaposing N graph convolution layers means that each node in the network are able to obtain information about nodes which are within a N hops away from itself; an animation of this propagation is demonstrated in the first figure of [Link]. Skip connections are an "engineering hack" designed to address the vanishing gradient problem by skipping some of the layers in the originally-intended design of the neural network. In this architecture, the initial input layer is placed after each graph convolution layer.

A single graph convolution layer is implemented in the following way.

```
class GraphConvLayer(layers.Layer):
    def __init__(self, hidden_units, dropout_rate=0.2, aggregation_type="mean",
        combination_type="concat", normalize=False, *args, **kwargs):
        super(GraphConvLayer, self).__init__(*args, **kwargs)
        ...

    def call(self, inputs):
        ...
```

The class `GraphConvLayer` inherits methods and properties from the existing `layers.Layer` class from Keras. By inheriting in this way, typical neural network training operations such as backpropagation, stochastic gradient descent, etc. are taken care of without explicit user implementation. The `super().init` line runs the constructor of the parent class.

The `call(self, inputs)` function of the layer implements how the forward pass is done. For GCNNs, we implement the forward pass as described in the above steps. Note that the update and aggregation functions are defined as separate methods of the `GraphConvLayer` class.

```
    def call(self, inputs):
        node_repesentations, edges, edge_weights = inputs
        node_indices, neighbour_indices = edges[0], edges[1]

        neighbour_repesentations = tf.gather(node_repesentations, neighbour_indices)

        # 1. Prepare the messages of the neighbours.
        neighbour_messages = self.prepare(neighbour_repesentations, edge_weights)

        # 2. Aggregate the neighbour messages.
        aggregated_messages = self.aggregate(node_indices, neighbour_messages)

        # 3. Update the node embedding with the neighbour messages.
        return self.update(node_repesentations, aggregated_messages)
```

First, an optional prerequisite step that is invoked prior to aggregating and updating is *preparing*: *input node representations* are transformed into separate more appropriate `messages` for the neural network to handle as input. One can think of an information theory analogy in which a transmitter uses an encoding scheme to pass the message through a channel.

```
    def prepare(self, node_repesentations, weights=None):
        messages = self.ffn_prepare(node_repesentations)
        if weights is not None:
            messages = messages * tf.expand_dims(weights, -1)
        return messages
```

In this example, nodes correspond to papers $i \in \{1, \cdots, N\}$, the node feature `self.node_feature` for paper i is simply the word attribute vector $\mathbf{w}_i \in \{0,1\}^V$. The node representation $\mathbf{w}_i^* = \texttt{self.ffn\_prepare}(\mathbf{w}_i) \in \{0,1\}^{h_2}$ of the original node feature $\mathbf{w}_i$ is created by feeding $\mathbf{w}_i$ through the following *feedforward neural network (FFN)*. Here, the feature and representation variables span across all nodes: `node_features` is $N \times V$ and `node_representations` is $N \times h_2$.

```
# x = self.preprocess(self.node_features)
# self.preprocess = create_ffn(hidden_units, dropout_rate, name="preprocess")

# self.ffn_prepare = create_ffn(hidden_units, dropout_rate)

def create_ffn(hidden_units, dropout_rate, name=None):
    fnn_layers = []

    for units in hidden_units:
        fnn_layers.append(layers.BatchNormalization())
        fnn_layers.append(layers.Dropout(dropout_rate))
        fnn_layers.append(layers.Dense(units, activation=tf.keras.activations.relu))

    return keras.Sequential(fnn_layers, name=name)
```

We concretely choose `hidden_units` $\triangleq [h_1, h_2] = [32, 32]$ and `dropout_rate` $\triangleq d = 0.2$. From the for loop above, it is easy to see that the FFN is constructed in the form of BatchNormalization $\rightarrow$ Dropout(d) $\rightarrow$ Dense(h_1) $\rightarrow$ ReLU $\rightarrow$ BatchNormalization $\rightarrow$ Dropout(d) $\rightarrow$ Dense(h_2) $\rightarrow$ ReLU. The model design choices for this FFN are arbitrary, and can be replaced with an alternative architecture.

This prerequisite preparation step is applied to each node's neighbors $\mathcal{N}_i := \{j \in \{1, \cdots, M\} | G_{ij} = 1\}$. Moreover, the `neighbour_repesentations` $\{\mathbf{w}_j^* : j \in \mathcal{N}_i\}$ are converted to `neighbour_messages` $\{\mathbf{w}_j^{**} : j \in \mathcal{N}_i\}$ further by applying the FFN once more. In this particular example, `neighbour_repesentations` and `neighbour_messages` have the same dimension, and because `neighbour_repesentations` is fed into the FFN to construct `neighbour_messages`, it is intuitive to choose hidden layer dimensions such that $h_1 = h_2$.

Second, the neighbor messages $\{\mathbf{w}_j^{**} : j \in \mathcal{N}_i\}$ are *aggregated* by each node using a permutation-invariant operation to create the *aggregated message* $\mathbf{v}_i = \texttt{aggregate}(\{\mathbf{w}_j^{**} : j \in \mathcal{N}_i\}) \in \mathbb{R}^{h_2}$. In the below, three common choices, summation, averaging, and maximization, are implemented. The full `aggregated_message` variable again spans across all nodes: `aggregated_message` is $N \times h_2$.

```
    def aggregate(self, node_indices, neighbour_messages):
        num_nodes = tf.math.reduce_max(node_indices) + 1

        if self.aggregation_type == "sum":
            aggregated_message = tf.math.unsorted_segment_sum(neighbour_messages,
                node_indices, num_segments=num_nodes)
        elif self.aggregation_type == "mean":
            aggregated_message = tf.math.unsorted_segment_mean(neighbour_messages,
                node_indices, num_segments=num_nodes)
        elif self.aggregation_type == "max":
            aggregated_message = tf.math.unsorted_segment_max(neighbour_messages,
                node_indices, num_segments=num_nodes)
        else:
            raise ValueError(f"Invalid aggregation type: {self.aggregation_type}.")

        return aggregated_message
```

Finally, the aggregated message of the node's neighbors is combined with the node's own representation. The combined result is then used to update each node's attribute value. Three types of combinations are implemented below: stacking, concatenating, and summing. We note that `node_repesentations` and `aggregated_messages` have the same dimension $N \times h_2$.

```
    def update(self, node_repesentations, aggregated_messages):
        if self.combination_type == "gru":
            # Create a sequence of two elements for the GRU layer.
            h = tf.stack([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "concat":
            # Concatenate the node_repesentations and aggregated_messages.
            h = tf.concat([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "add":
            # Add node_repesentations and aggregated_messages.
            h = node_repesentations + aggregated_messages
        else:
            raise ValueError(f"Invalid combination type: {self.combination_type}.")

        node_embeddings = self.update_fn(h)
        ...
        return node_embeddings
```

The `self.update_fn` is simply the same FFN we used in the preprocessing and preparation stage of the layer. For the special case of *gated reccurrent units (GRUs)* being used in the neural network, we stack (independently store) the two messages as the combination operation, and invoke a GRU as `self.update_fn`. Essentially, this is because GRUs choose between the two representations: either the neighbors' aggregated message is returned or the node's own original representation, but not both.

Postprocessing the final `node_embeddings` via `self.postprocess` amounts to feeding the result through the above-described FFN. The post-processed result is then renamed `node_embeddings`, which is of size $N \times h_2$, and classified using the Softmax layer (`compute_logits`).

2.2.4 Application: Vehicle Traffic Forecasting

The traffic forecasting problem is described as follows: given a previous history of time series about the characteristics of traffic flow (e.g., speed, volume) across a road network, we are interested in predicting the future characteristics of traffic flow.

Common machine learning architectures used to solve this problem invoke some combination of the above-mentioned three neural network types.

RNNs were introduced in the previous Section 2.2.2. In this section, we provide an example application where RNNs are used along with GCNNs to forecast traffic, inspired by [Link]. Given a time series of traffic volume speeds over time, collected over a graph network of traffic sensor stations, what is the predicted future horizon of traffic volume speeds?

Data The dataset was sampled over $M = 228$ stations (e.g., traffic sensors) and a time horizon $[0, T_{\text{sim}} - 1]$, where the numerical value $T_{\text{sim}} = 12672$ was obtained by sampling 30-second intervals of time between May and June 2012. It consists of two main components:

- `route_distances`: a matrix $R \in \mathbb{R}^{M \times M}$ such that R_{ij} is the distance between stations i and j.
- `speeds_array`: a matrix $S \in \mathbb{R}^{T_{\text{sim}} \times M}$ such that the jth column $S_{\cdot,j}$ contains time series data about the observed speed of vehicle flow over station j.

We create an initial visualization of the data to get a rough sense of what we are working with. The full timeseries of speeds accumulated by two chosen stations, Station 0 and Station 227 is shown at the top of Fig. 2.6. The bottom of Fig. 2.6 shows the color-coded Pearson product-moment correlation coefficient computed between each pair of stations $(i, j) \in \{1, \cdots, M'\}$ using the timeseries data from S. Here, $M' = 26 \leq M$ is a subset of stations chosen for simpler visualization.

Preprocessing We split the speeds timeseries matrix S into *training* S_{train}, *validation* S_{val}, and *test* S_{test} datasets. Essentially, the full time interval $[0, T_{\text{sim}}]$ is divided into three disjoint segments $[0, T_{\text{train}} - 1]$, $[T_{\text{train}}, T_{\text{train}} + T_{\text{val}} - 1]$, and $[T_{\text{train}} + T_{\text{val}}, T_{\text{sim}} - 1]$ corresponding to the training, validation, and test sets respectively. Further define $T_{\text{test}} = T_{\text{sim}} - T_{\text{train}} + T_{\text{val}}$ to be the length of the test set. Thus, the dimensions of the datasets are $S_\chi \in \mathbb{R}^{T_\chi \times M}$, for placeholder variable $\chi \in \{\text{train}, \text{val}, \text{test}\}$. The specific concrete values used are $T_{\text{train}} = 6336$, $T_{\text{val}} = 2534$, and $T_{\text{test}} = 3802$, which roughly corresponds to 50% of the dataset used for training, 20% for validation, and 30% for testing. For numerical stability, the columns of all three datasets are normalized by the z-score method.

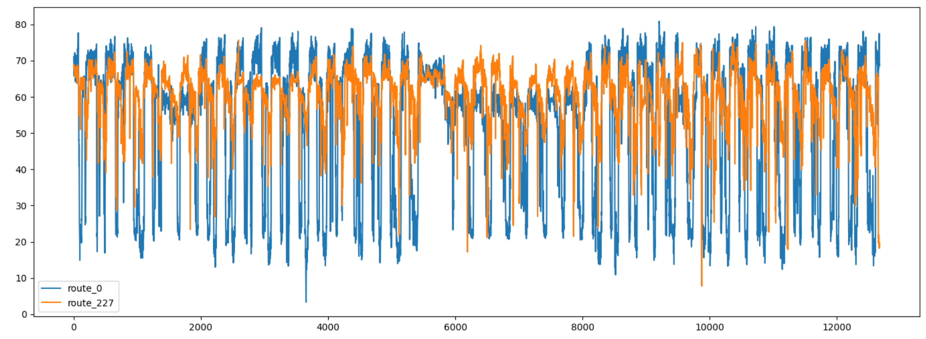
The training set is further divided into overlapping batches of size $b = 64$; this means that there are a total of $B = T_{\text{train}}/b = 99$ batches for the model to train on.

Define $H_p, H_f \in \mathbb{N}$ to be the past and future horizons of time. Training examples in each batch is comprised of two parts:

1. a length-H_p time series of vehicle flow speeds with intervals of the form $[t, t + H_p - 1]$ for each time $t \in [0, T_{\text{train}} - H_p]$
2. a length-H_f times series of vehicle flow speeds with intervals of the the form $[t + H_p, t + H_p + H_f - 1]$ for each time $t \in [H_p, T_{\text{train}} - H_f]$

Note that the number of examples in 1. and 2. are the same. Essentially, the model is trained on a finite history of speeds (each length-H_p time series) so that it can accurately predict the behavior on the corresponding finite future horizon of speeds (each length-H_f time series). This data preprocessing behavior is visualized in Fig. 2.7.

Implementation detail: the batched datasets are transformed from `Numpy` matrices into `tf.data.Dataset` structures. The corresponding variables are stored into the variables `train_dataset`, `val_dataset`, and `test_dataset`.



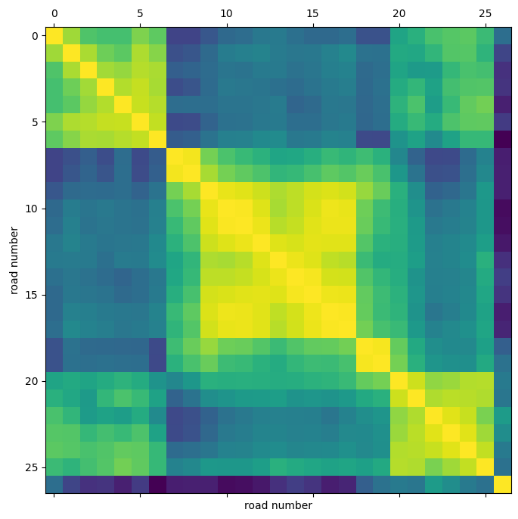


Figure 2.6: [Top] Speed of vehicle flow versus time for Station 0 and Station 26. [Bottom] The color-coded correlation coefficient matrix for the first 26 stations, based on their speed samples from S. Lighter, yellower colors correspond to higher correlation between the row and column entries.

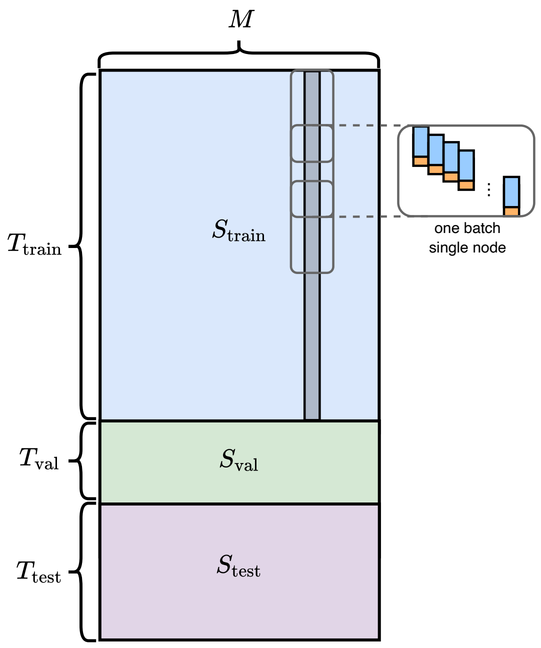


Figure 2.7: Preprocessing the speeds array into batches for training and predicting.

Graph Construction It is fruitful to invoke a GNN architecture (Section 2.2.3) to encapsulate the graph structure of the road network. The entities of our GNN correspond to traffic sensor stations. The relations are derived by considering the distance R_{ij} between two stations $i, j \in \{1, \cdots, M\}$. For the concreteness of this particular example, an edge is connected between i and j if the normalized route distance is less than a certain threshold. Specifically, the weighted adjacency matrix $W \in \mathbb{R}^{M \times M}$ is derived from the route-distance matrix R via

$$W_{ij} = \begin{cases} \exp\left(-R_{ij}^2/\sigma^2\right) & \text{if } i \neq j, \exp\left(-R_{ij}^2/\sigma^2\right) \geq \epsilon \\ 0 & \text{else} \end{cases}$$

for threshold parameters $\sigma, \epsilon > 0$ chosen by the user.

Implementation detail: the graph is encoded into a user-defined `GraphInfo` data structure comprised of a `tuple` of edges (`node_indices`, `neighbor_indices`) and the number of nodes M.

The Neural Network Architecture There are two types of custom layers that we use in our model: 1) the graph convolution layer detailed in the previous Section 2.2.3, and 2) an *LSTM layer*. The overall neural network architecture is visualized in Fig. 2.8. Because the graph convolution layer and the LSTM layer are implemented sequentially in this architecture, traffic over space and traffic speed over time are updated sequentially each epoch. The corresponding pseudocode for the forward pass is below:

```
    def call(self, inputs):
        ...
        gcn_out = self.graph_conv(inputs)
        ...
        lstm_out = self.lstm(gcn_out)
        dense_output = self.dense(lstm_out)
        ...
        output = tf.reshape(...)

        return output
```

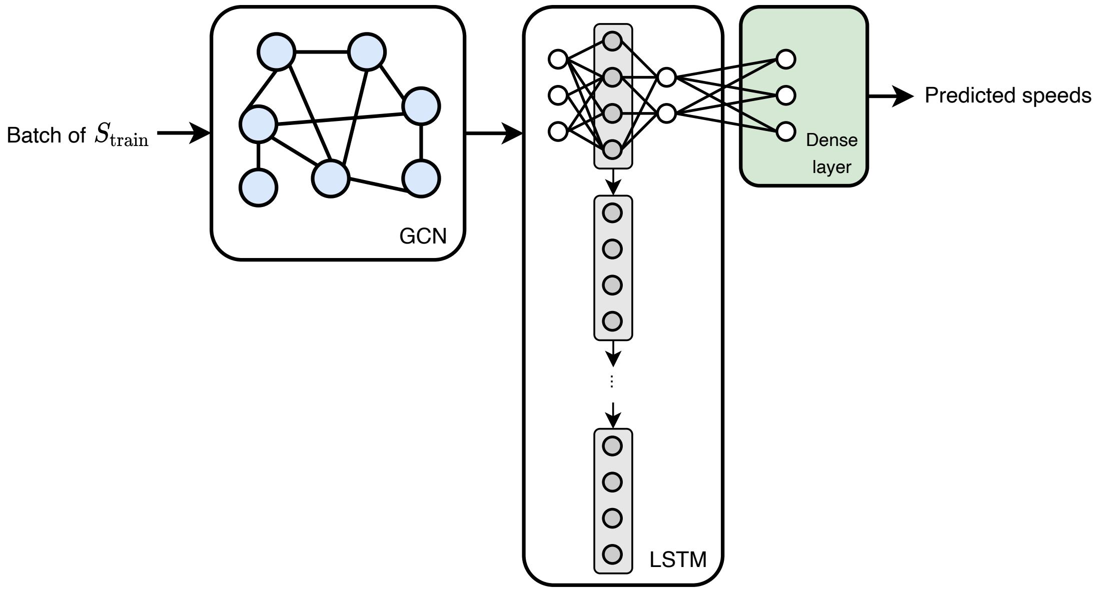


Figure 2.8: The graph convolutional neural network + LSTM architecture used for the vehicle traffic forecasting example.

Implementation details: the `...` in the code above represents ommitted details about reshaping tensors appropriately before passing them into each layer. For example, Keras' pre-defined LSTM layer only takes in 3D tensors as input.

Training the Model The constructed model can be trained using standard Tensorflow API.

```
model = keras.models.Model(inputs, outputs)
model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=0.0002),
    loss=keras.losses.MeanSquaredError(),
)

model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=epochs,
    callbacks=[keras.callbacks.EarlyStopping(patience=10)],
)
```

Making Predictions We train the model on 10 epochs (which takes approximately 11 minutes to run) and visualize the resulting prediction for a single node over the entire horizon of length T_{test} is shown in Fig. 2.9. The mean absolute error of a naive forecasting method (provided as the last value of the speed for each traffic station in the dataset) evaluates to 0.19905 while the mean absolute error metric for this neural network architecture evaulates to 0.19024.

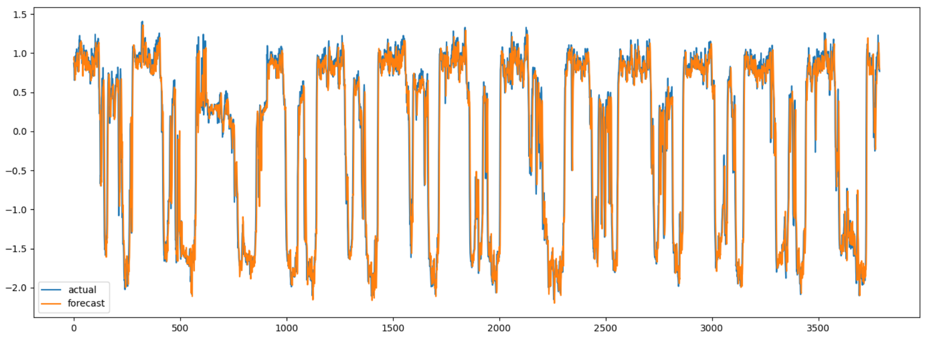


Figure 2.9: The predicted and true speeds versus time for a single node in the network.

2.3 Quantum Neural Networks

2.3.1 Graph Neural Networks

Quantum implementations of graph neural networks and some of its popular extensions were introduced in [6]. Given the usual graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$ (we will consider the vector of global attributes $\mathbf{u}$ separately), we embed a *quantum subsystem* to each vertex in the graph. A quantum subsystem may refer to any one of the usual quantum components, i.e., a qubit, qudit, or an entire quantum circuit. Define $\mathcal{H}_v$ to be the Hilbert space associated with the quantum subsystem embedded within vertex $v \in \mathcal{V}$, and define $\mathcal{H}_\mathcal{V} := \bigotimes_{v\in\mathcal{V}} \mathcal{H}_v$ to be the global Hilbert space for the entire graph. Similarly, we may define $\mathcal{H}_e$ to be the Hilbert space associated with each edge $e \in \mathcal{E}$, and likewise $\mathcal{H}_\mathcal{E}$.

It is easiest to abstract away the details of the quantum circuit used to implement the general graph neural network on $\mathcal{G}$, and instead use the Hamiltonian simulation framework described in Section 1.2.4. Let $N_p \in \mathbb{N}$ be the number of parts the Hamiltonian H is decomposed into, and let m be the number of subintervals the simulation duration T_{sim} is partitioned in as increments of $\Delta t > 0$. Let η and θ denote the parameters of the quantum implementation of the network, where $\eta \in \mathbb{R}^{m\times N_p}$ is a matrix containing the weight of each part of the Hamiltonian towards the total sum.

Then as shown in (1.54), the correpsonding unitary time-evolution operator is written as the following time-ordered product:

$$U^{\text{gen}}_{(\eta,\theta)}(0, T_{\text{sim}}) = \prod_{k=1}^{m} \prod_{n=1}^{N_p} \mathcal{T}\left\{ e^{-i\int_{t_k}^{t_k+\Delta t} \eta_{kn} H_n(s,\theta) ds} \right\}$$

To emphasize that the operator is unitary, we henceforth change the notation of (1.54) from G to U. Furthermore, to distinguish the Hamiltonian from the Hilbert space on which it operates, we revert the notation $\mathcal{H}$ (from Section 1.2.4) to H.

For simplicity, we approximate the Hamiltonian of the system as a piecewise-constant function over the simulation duration $[0, T_{\text{sim}}]$ (see (1.55)).

$$U_{(\eta,\theta)}(0, T_{\text{sim}}) = \prod_{k=1}^{m} \prod_{n=1}^{N_p} e^{-i\eta_{kn} H_n(\theta)} \tag{2.11}$$

These Hamiltonian sum parts $H_n(\theta)$ are parametrized by θ and also encode the topology of $\mathcal{G}$. Mathematically:

$$H_n(\theta) = \sum_{(i,j)\in\mathcal{E}} \sum_{r\in\mathcal{I}_{ij}} W^{(e)}_{nrij} O^{(nr)}_i \otimes P^{(nr)}_j + \sum_{v\in\mathcal{V}} \sum_{r\in\mathcal{I}_v} W^{(v)}_{nrv} R^{(nr)}_v \tag{2.12}$$

where $\theta := \cup_n(\cup_{i,j,r} W^{(e)}_{nrij} \cup_{v,r} W^{(v)}_{nrv})$ specifically pertains to the weights of the network, $\mathcal{I}_{ij}, \mathcal{I}_v$ are index sets for the edges and nodes respectively, and $O^{(nr)}_i, P^{(nr)}_j, R^{(nr)}_v$ are Hermitian operators which act on the nth term of the Hamiltonian decomposition. For the simplest possible decomposition, the operators $O^{(nr)}_i, P^{(nr)}_j, R^{(nr)}_v$ can be further decomposed into sums of simple Pauli operators, as in Example 5. However, we abstract away the specific Hamiltonain operators in terms of $O^{(nr)}_i, P^{(nr)}_j, R^{(nr)}_v$ because 1) they are application-specific, and 2) the form of (2.12) demonstrates the topology of the graph most clearly. We refer to the above quantum circuit implemented by (2.11) with Hamiltonian sum parts (2.12) as the general *quantum graph neural network (QGNN).*

There are two specialized versions derived from the above general architecture. First, the *quantum graph recurrent neural network (QGRNN)* has its parameters based specifically on temporal expansion. Similarly, the *quantum graph convolutional neural network (QGCNN)* has its parameters tied specifically to spatial expansion. In the following discussion, we provide concrete example implementations of each.

Bibliography

[1] S. Yan, H. Qi, and W. Cui, "Nonlinear quantum neuron: A fundamental building block for quantum neural networks," *Phys. Rev. A*, vol. 102, p. 052421, Nov 2020.

[2] Y. Cao, G. G. Guerreschi, and A. Aspuru-Guzik, "Quantum neuron: an elementary building block for machine learning on quantum computers," *ArXiv preprint, arXiv:1711.11240*, Nov 2017.

[3] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," *ArXiv preprint, arXiv:1806.01261*, Apr 2018.

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17, 2017, p. 1263–1272.

[5] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *Proceedings of the 5th International Conference on Learning Representations*, ser. ICLR '17, 2017.

[6] G. Verdon, T. McCourt, E. Luzhnica, V. Singh, S. Leichenauer, and J. Hidary, "Quantum graph neural networks," *ArXiv preprint, arxiv:1909.12264*, Sep 2019.