IOP Publishing Mach. Learn.: Sci. Technol. 3 (2022) 015034 https://doi.org/10.1088/2632-2153/ac5997

MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
20 December 2021

REVISED
17 February 2022

ACCEPTED FOR PUBLICATION
1 March 2022

PUBLISHED
25 March 2022

PAPER

Qsun: an open-source platform towards practical quantum machine learning applications

Quoc Chuong Nguyen[1,*], **Le Bin Ho**[2,3,*], **Lan Nguyen Tran**[2] **and Hung Q Nguyen**[4,*]

[1] Vietnamese-German University, Ho Chi Minh City 70000, Vietnam
[2] Ho Chi Minh City Institute of Physics, National Institute of Applied Mechanics and Informatics, Vietnam Academy of Science and Technology, Ho Chi Minh City 70000, Vietnam
[3] Research Institute of Electrical Communication, Tohoku University, Sendai 980-8577, Japan
[4] Nano and Energy Center, VNU University of Science, Vietnam National University, Hanoi 120401, Vietnam
[*] Authors to whom any correspondence should be addressed.

E-mail: quoc.chuong1413017@gmail.com, binho@riec.tohoku.ac.jp **and** hungngq@hus.edu.vn

Keywords: quantum virtual machine, quantum machine learning, quantum differentiable programming, quantum linear regression, quantum neural network

Abstract

Currently, quantum hardware is restrained by noises and qubit numbers. Thus, a quantum virtual machine (QVM) that simulates operations of a quantum computer on classical computers is a vital tool for developing and testing quantum algorithms before deploying them on real quantum computers. Various variational quantum algorithms (VQAs) have been proposed and tested on QVMs to surpass the limitations of quantum hardware. Our goal is to exploit further the VQAs towards practical applications of quantum machine learning (QML) using state-of-the-art quantum computers. In this paper, we first introduce a QVM named Qsun, whose operation is underlined by quantum state wavefunctions. The platform provides native tools supporting VQAs. Especially using the parameter-shift rule, we implement quantum differentiable programming essential for gradient-based optimization. We then report two tests representative of QML: quantum linear regression and quantum neural network.

1. Introduction

The advent of quantum computers has opened a significant turning point for exponentially speeding up computing tasks that classical computers need thousand years to execute [1, 2]. Although mankind has witnessed tremendous development in this field theoretically and experimentally in the last few years, most state-of-the-art quantum computers still rely on noisy intermediate-scale quantum (NISQ) computers. Noises and qubit-number constraints prevent to build high-fidelity quantum computers capable of substantially implementing quantum algorithms [3–6]. To bypass these constraints, various hybrid quantum–classical algorithms that use classical computers to optimize quantum circuits have been proposed [7–9]. Among these, variational quantum algorithms (VQAs) may be the most promising ones in the NISQ era.

VQAs generally consist of three essential steps: (a) initializing quantum states for a given wavefunction ansatz, (b) measuring a cost function suitable for problems being considered, and (c) minimizing the cost function and updating new parameters. The self-consistency is performed until convergence. VQAs have been extensively employed to tackle numerous tasks, including the variational quantum eigensolvers (VQEs) [10–16], quantum dynamics simulation [17–22], mathematical applications [23–31], quantum machine learning (QML) [32–40], and new frontiers in quantum foundations [8, 41–47].

Typically, VQAs employ VQCs to measure the cost function on a quantum computer and outsource its optimization to a classical computer. While one can manipulate gradient-free optimizers, such as Nelder-Mead simplex [48], to minimize the cost function, using gradient-based methods like gradient descent can help us speed up and guarantee the convergence of the optimization. Several quantum algorithms have been proposed to evaluate the cost function gradient measured on quantum computers [49–54]. Among those methods, quantum differentiable programming (QDP) has been introduced and utilized extensively [36, 51, 54–59]. It relies on a technique called the parameter-shift rule that evaluates the derivative of any differentiable function using quantum circuits [36, 55–59]. Therefore, this method is beneficial for developing 'on-circuit' gradient-based optimization techniques, especially for QML applications where various methods like quantum neural networks (QNNs) demand the derivative information of the cost function.

While quantum algorithms should be performed on quantum computers, the current limitation of NISQ computers cause challenges in developing and testing new quantum algorithms, demanding the use of virtual alternatives called quantum virtual machines (QVMs). Besides, QVMs are necessary for modeling various noisy channels to characterize the noises and the efficiency of quantum error correction. One can classify QVMs into two types according to the way to build them: (a) the matrix multiplication approach [4, 56, 60–65], and (b) the wavefunction approach [66–72]. While the former performs matrix multiplication for all qubits in quantum circuits, the latter represents quantum circuits by corresponding wavefunctions. On the one hand, the former can significantly reduce the memory capacity using tensor network contraction [73, 74], and the cost to pay is exponentially increasing the computational time. On the other hand, the latter, in principle, can require less computational time in a limit of qubits number and a sufficiently large memory size that can store the total quantum wavefunction. Therefore, both approaches exist side by side and a possible hybrid approach [1, 75] for convenient purposes.

There are several QVM's libraries developed for QML orientation, such as TensorFlow Quantum library [76] implemented in Cirq [61], Pennylane [56] designed for photonics devices, and TensorNetwork [62, 77]. These libraries are all constructed in the matrix-multiplication type of QVMs, designed to submit quantum tasks to the developed hardware conveniently.

In this work, we develop a QVM platform named Qsun using the wavefunction approach towards the QML applications. In Qsun, a quantum register is represented by a wavefunction, and quantum gates are manipulated directly by updating the amplitude of the wavefunction. Measurement results rely on probabilities of the wavefunction. Our simple approach yields faster computation speed for a small number of qubit when compared to other QVMs such as Qiskit, ProjectQ, or Pennylane. Basing on this generic QVM, we aim to exploit the advantages of QDP with the parameter-shift rule as the core engine towards practical applications in QML. Two representative examples of QML are demonstrated: quantum linear regression (QLR) and QNN. These algorithms are compared to standard progams: Qiskit, ProjectQ, and Pennylane, and classical algorithms when applicable. In these comparisons, Qsun performs slightly better for QDP, QLR, and QNN. All in all, Qsun is an efficient combination of QVM with QDP features that is oriented toward machine learning problems. In the following, we introduce the QVM platform Qsun and its performance compared to others in section 2, followed by an introduction to the QDP implementation within Qsun in section 3. We then discuss some QML applications of the Qsun package in section 4.

2. Quantum Virtual Machine implementation in Qsun

In practice, quantum computers work on quantum algorithms by composing a quantum register, or qubits, operated by a sequence of quantum gates. Results are then traced out from quantum measurements. We now introduce our QVM named Qsun, an open-source platform simulating the operation of a generic quantum computer [78]. We aim the platform to the development of QML and related problems. We develop it in Python and employ the Numpy library for fast matrix operations and numerical computations.

2.1. Simulating quantum computers using wavefunction basis

Unlike widely-used approaches based on matrix multiplication [4, 56, 60–65], our platform is developed using the class of 'wavefunction' approach [66–72], in which a quantum register is represented by its wavefunction. The operation of quantum gates is simulated by updating the wavefunction's amplitude, and output results are obtained by measuring wavefunction's probabilities. We expect that working directly on wavefunction is beneficial for QML applications, especially for building and training VQCs in QNNs. As depicted in figure 1, Qsun consists of three main modules `Qwave`, `Qgates`, and `Qmeas` for quantum register, quantum gates, and quantum measurement, respectively.

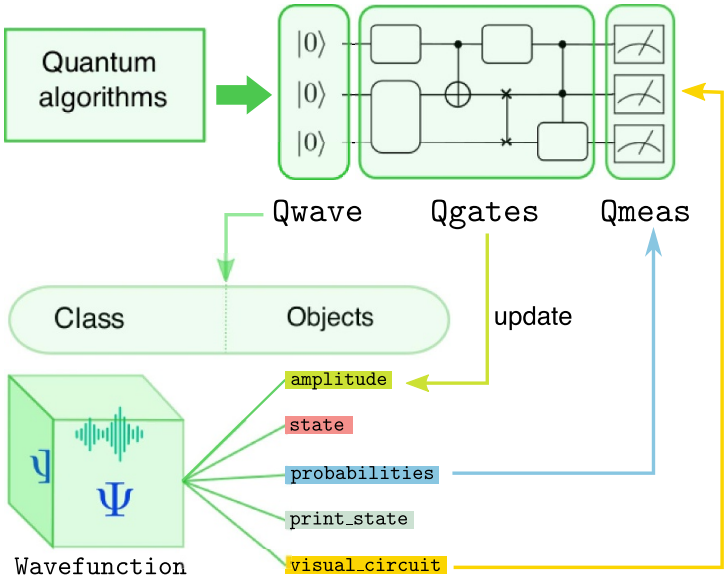


Figure 1. Qsun quantum virtual machine (QVM) consists of three main modules `Qwave`, `Qgates`, and `Qmeas`, to simulate quantum circuits with a quantum register (qubits), quantum gates, and quantum measurements, respectively. In Qsun, a quantum register is constructed by its wavefunction through the `Wavefunction` class and its containing objects, as illustrated in the figure.

Table 1. List of methods used in the class `Wavefunction`.

Methods	Description
`amplitude`	a NumPy array of complex numbers that stores the amplitudes of quantum states.
`state`	a NumPy array of strings that labels the quantum states's basis.
`probabilities`	return a list of corresponding probabilities for each basis vector in the superposition.
`print_state`	return a string representing a quantum state of the system in bra-ket notations.
`visual_circuit`	print a visualization of a quantum circuit.

2.1.1. Qwave

In general, for a quantum register with N qubits, its quantum states are represented in the 2^N-dimension Hilbert space as

$$|\psi\rangle = \sum_{j=0}^{2^N-1} \alpha_j |j\rangle, \tag{1}$$

where α_j are complex amplitudes obeying a completeness relation $\sum_{j=0}^{2^N-1} |\alpha_j|^2 = 1$, and vectors $|j\rangle$ are elements of the computational basis. We integrate quantum state's information into the class `Wavefunction` as described in figure 1 and table 1. The class allows us to access and update amplitudes directly according to the evolution of the quantum state under the action of the unitary quantum gates. It also measures probabilities that contain output information.

2.1.2. Qgates

To manipulate a single-qubit gate $U = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$ acting on the nth qubit, the nonzero elements in `amplitude` array are updated as [69]

$$\begin{pmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^n} \end{pmatrix} \to U \begin{pmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^n} \end{pmatrix}, \tag{2}$$

where $s_i = \text{floor}(i/2^n)2^{n+1} + (i \text{mod} 2^n)$, for all $i \in [0, 2^{N-1} - 1]$. Here, floor(x) is the standard floor function taking the greatest integer less than or equal to the real of x. We unify the implementation of single- and multiple-qubit gates into a common framework. We outline the operation of single-qubit gates in algorithm 1 and an example of the Hadamard gate in algorithm 2. We emphasize that `Qgates` only update nonzero components of wavefunction amplitudes. This way, we can avoid demanding matrix multiplications that escalate exponentially $(2^N \times 2^N)$ with the number of N qubits. This generic algorithm allows us to

Algorithm 1. Operation of a single-qubit gate.

Result: Wavefunction with new probability amplitudes
w ← Wavefunction;
states ← w.state; ampl ← w.amplitude;
N ← size(state[i][0]); n ← target qubit;
n_ampl ← [0, …, 0], size(n_ampl) = size(ampl);
$cut \leftarrow 2^{N-n-1}$;
for i ← 0, size(ampl) **do**
  if state[i][n] == 0 **then**
    n_ampl[i] ← n_ampl[i] + a^*ampl[i];
    n_ampl[$i+cut$] ← n_ampl[$i+cut$] + b^*ampl[i];
  else
    n_ampl[i] ← n_ampl[i] + d^*ampl[i];
    n_ampl[$i-cut$] ← n_ampl[$i-cut$] + c^*ampl[i];
  end
end
w.amplitude ← n_ampl

Algorithm 2. Operation of Hadamard gate.

Result: Wavefunction with new probability amplitudes corresponding to Hadamard state
w ← Wavefunction;
states ← w.state; ampl ← w.amplitude;
N ← size(state[i][0]); n ← target qubit;
n_ampl ← [0, …, 0], size(n_ampl) = size(ampl);
$cut \leftarrow 2^{N-n-1}$;
for i ← 0, size(ampl) **do**
  if state[i][n] == 0 **then**
    n_ampl[i] ← n_ampl[i] + $(1/\sqrt{2})^*$ampl[i];
    n_ampl[$i+cut$] ← n_ampl[$i+cut$] + $(1/\sqrt{2})^*$ampl[i];
  else
    n_ampl[i] ← n_ampl[i] + $(-1/\sqrt{2})^*$ampl[i];
    n_ampl[$i-cut$] ← n_ampl[$i-cut$] + $(1/\sqrt{2})^*$ampl[i];
  end
    w.amplitude ← n_ampl
end

implement arbitrary unitary gates without decomposing them into universal ones, which may be advantageous to model a general class of neural networks using quantum circuits.

To mimic the actual operation of quantum computers, we introduce noises into the wavefunctions. In Qsun, the standard quantum depolarizing channel is implemented as a single-qubit gate $\mathcal{E}$ that is a part of quantum circuits acting on the wavefunctions. For a given noisy probability p, applying the gate $\mathcal{E}$ on a mixed quantum state ρ will transform it to [79]

$$\rho \xrightarrow{\mathcal{E}} (1-p)\rho + \frac{p}{2}I, \tag{3}$$

where I is an 2×2 identity matrix. In general, one can decompose the depolarizing channel into the bit-flip, phase-flip, and phase-bit-flip as

$$\rho \xrightarrow{\mathcal{E}} (1-p)\rho + p_x X\rho X + p_y Y\rho Y + p_z Z\rho Z, \tag{4}$$

where p_x, p_y, p_z are the probabilities of bit-flip, phase-bit-flip, and phase-flip, respectively [4, 79]. In Qsun, we use $p_x = p_y = p_z = p/3$ and apply to every qubits in the circuit each after the action of a quantum gate on the circuit.

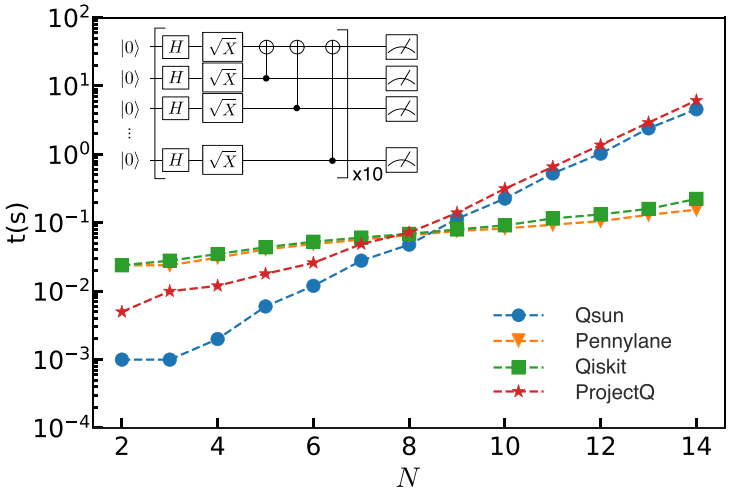


Figure 2. Comparing computational time t (in seconds) of different QVMs as increasing the number of qubits N. Inset: the testing circuit including an H, a $\sqrt{X}$, and a CNOT gates acts on each qubit. The depth is fixed at 10, and the number N of qubits is varied to assess the QVM performance.

2.1.3. Qmeas

The module `Qmeas` is designed to execute quantum measurements on a single qubit or all qubits in the quantum circuit. For a measurement on a single qubit n, the probability for that its outcome is $|0\rangle$ reads

$$p(0) = \sum_{i=0}^{2^{N-1}-1} \langle\psi|j_{s_i}\rangle\langle j_{s_i}|\psi\rangle = \sum_{i=0}^{2^{N-1}-1} |\alpha_{s_i}|^2, \tag{5}$$

with $|j_{s_i}\rangle$ as the basis element and the post-quantum state after the measurement is given as

$$|\psi'\rangle = \frac{\sum_{i=0}^{2^{N-1}-1} |j_{s_i}\rangle\langle j_{s_i}|\psi\rangle}{\sqrt{p(0)}}. \tag{6}$$

Similarly, the probability for getting the outcome $|1\rangle$ reads

$$p(1) = 1 - p(0), \tag{7}$$

and the post-state is

$$|\psi'\rangle = \frac{\sum_{i=0}^{2^{N-1}-1} |j_{s_i+2^n}\rangle\langle j_{s_i+2^n}|\psi\rangle}{\sqrt{p(1)}}. \tag{8}$$

For all-qubit measurement, the post-quantum state will collapse to one of $\{|j\rangle\}$ with the probability of $|\alpha_j|^2$. In Qsun, we build these two measurements onto `measure_one` and `measure_all`, respectively.

2.2. Assessing the Quantum Virtual Machine performance

Let us now assess the performance of Qsun and compare it with three existing ones in Pennylane, Qiskit, and ProjectQ. Note that Qsun and ProjectQ belong to the wavefunction class, while the others are in the matrix multiplication class. We have adopted the testing circuit from [80] composed of the Hadamard, $\sqrt{X}$, and CNOT gates acting on each qubit. We have fixed the depth of the circuit at 10 and varied the number of qubits N. The code for this test is shown in appendix A.

Figure 2 represents the change of computational time when the number N of qubits increases. In general, there are two magnitudes of slope corresponding to two ways of QVM implementation. While the wavefunction-based approach is faster than the matrix-multiplication one for small numbers of qubits ($N < 8$), the opposite behavior is observed for larger numbers of qubits. This observation reflects the basic properties of these two approaches as discussed above and see also [72]. However, there are some available techniques to improve the performance of the wavefunction approach for larger numbers of qubits, such as, SIMD (single-instruction, multiple data) optimization and multi-threading [72]. We further summarize a comparison between Qsun and other simulators in terms of practical quantum algorithms in table 2.

Table 2. Comparison between Qsun and other simulators.

Algorithms	Qsun	ProjectQ	Qiskit	Pennylane
Standard algorithms	✓	✓	✓	✓
Quantum differentiable programming	✓	✓	✓	✓
Quantum linear regression	✓	✓	✓	✓
Quantum neural network	✓		✓	✓

3. Quantum Differential Programming implementation in Qsun

Given a quantum state $|\psi(\vec{\theta})\rangle$ with $\vec{\theta}$ as variational parameters and an observable $\widehat{C}$, the task is to seek the global minimum of the expectation value $C(\vec{\theta}) = \langle\psi(\vec{\theta})|\widehat{C}|\psi(\vec{\theta})\rangle$ with respect to parameters $\vec{\theta}$. For example, if $\widehat{C}$ is a Hamiltonian, its global minimum is the ground state energy. In general, $C(\vec{\theta})$ is called as the cost function, and minimizing the cost function requires its derivative with respect to parameters $\vec{\theta}$, $\partial C(\vec{\theta})/\partial\theta$. In classical computing, if the analytical form of $\partial C(\vec{\theta})/\partial\theta$ is unknown, finite difference methods are often used to evaluate the derivative approximately. Although this approximation is fast and easy to implement, its accuracy depends on discretization steps. In contrast to the classical finite differentiation, QDP is an automatic and exact method to compute the derivative of a function. QDP is thus essential for accurate gradient computation in multiple VQAs, including QML models.

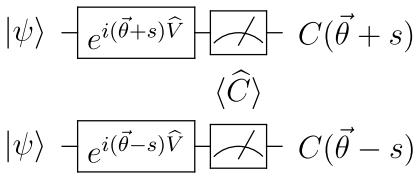
The heart of QDP is the parameter-shift rule that is analytically computed using quantum circuits. The algorithm is outlined in algorithm 3. Let us introduce a parameterized generator $\widehat{V}$ independent of $\vec{\theta}$ such that $|\psi(\vec{\theta})\rangle = e^{i\vec{\theta}\widehat{V}}|\psi\rangle$. The cost function is then rewritten as

$$C(\vec{\theta}) = \langle\psi|e^{-i\vec{\theta}\widehat{V}}\widehat{C}e^{i\vec{\theta}\widehat{V}}|\psi\rangle = \mathrm{Tr}\big(\widehat{C}e^{\mathcal{Z}}[\rho]\big), \tag{9}$$

with $\rho = |\psi\rangle\langle\psi|$; $\mathcal{Z} = i\vec{\theta}\widehat{V}$, and the superoperator $e^{\mathcal{Z}}[\rho] = e^{-\mathcal{Z}}\rho e^{\mathcal{Z}}$ [81]. The parameter-shift rule for each $\theta \in \vec{\theta}$ states that

$$\frac{\partial C(\vec{\theta})}{\partial\theta} = \mathrm{Tr}\big(\widehat{C}\partial_\theta e^{\mathcal{Z}}[\rho]\big) = c\big[C(\vec{\theta}+s) - C(\vec{\theta}-s)\big], \tag{10}$$

where $c = 1/(2\sin(s))$, and s is determined based on the superoperator and independent of $\vec{\theta}$. The values of the cost function C at $\vec{\theta} \pm s$ are measured on quantum computers by implementing two quantum circuits as follows



The derivative is finally obtained by subtracting measurement results from the two circuits. An advantage of the parameter-shift rule is that it can compute the derivative of the given function exactly while the shift s can be chosen arbitrarily large [82].

We combine the QDP implemented in Qsun with a classical optimizer like adaptive moment (Adam) optimization and implement quantum gradient descent, making the optimization procedure fully quantum. Let us now examine the performance of the QDP in comparison with Pennylane, another QML-oriented platform. In this test, we measure the computational time spent for optimizing the ground state of an Ising model. The code for running this test is given in B, and a tutorial explaining how to use QDP with sample codes is written in appendix C.

Figure 3 presents the change of computational time when increasing the number of qubits N for three cases of Qsun, Pennylane, and ProjectQ. As shown in the inset of figure 3, for each qubit, we have two variational parameters $\vec{\theta}$, meaning that the number of variational parameters is twice of N. Qsun is faster than Pennylane for $N < 5$, whereas it is slower for larger numbers of qubits ($N > 5$). Among the three cases, ProjectQ consumes the most computational time for all N. Of course, the exponential growth with the number of qubits is not inevitable.

Algorithm 3. Quantum differentiable programming implementation in Qsun.

Result: Derivative of a function
$f \leftarrow$ Function; $c \leftarrow$ Quantum Circuit;
$p \leftarrow$ Params; $s \leftarrow$ Shift;
$diff \leftarrow [0, \ldots, 0]$, size($diff$) = size($p$);
for i $\leftarrow$ 0, size($diff$) **do**
  $p\_plus \leftarrow$ copy(p);
  $p\_subs \leftarrow$ copy(p);
  $p\_plus[i] \leftarrow p[i] + s$;
  $p\_subs[i] \leftarrow p[i] - s$;
  $diff[i] \leftarrow (f(c, p\_plus) - f(c, p\_subs))/(2^*\sin(s))$;
end

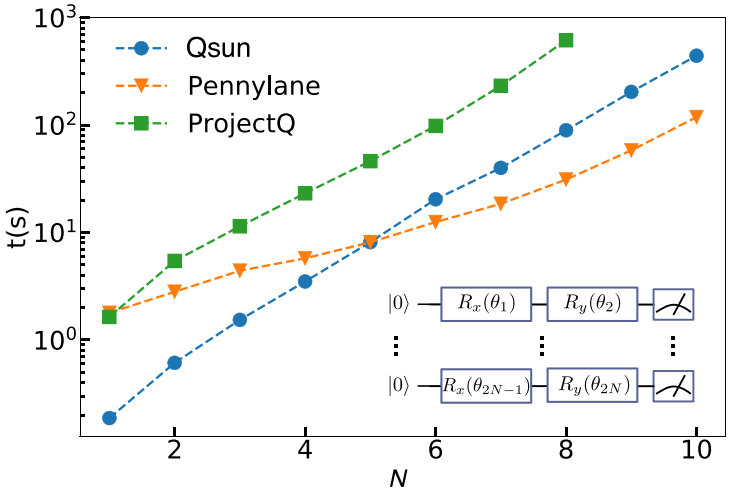


Figure 3. Computational time t in seconds of the QDP implementation in Qsun, ProjectQ using an engine Simulator() and Pennylane using a default.qubit device when the number of qubits increases. Inset: quantum circuits used for the QDP test.

4. Quantum Machine Learning applications using Qsun

Various QML models can be developed with QDP implementations to evaluate the gradient and employ gradient-based optimization. This section demonstrates QML applications using Qsun in two well-known models: QLR and QNN. Its performances are compared to other standard tools.

Before digging into detailed examples, let us derive the QDP implementation in derivative of a standard mean squared error cost function:

$$C(\vec{\theta}) = \frac{1}{M}\sum_{i=0}^{M-1}\left(y_i - \widehat{y}_i\right)^2, \tag{11}$$

where $\vec{\theta}$ is a tuple of variational parameters, y_i represents the true value and $\widehat{y}_i$ stands for prediction value, with $i \in [0, M-1]$, M the number of samples in the dataset to be trained. Concretely, we also consider the prediction value is a composition function of an activation function $f(\vec{\theta})$, i.e. $\widehat{y}_i(f(\vec{\theta}))$. Then, we derive at a chain rule:

$$\frac{\partial C(\vec{\theta})}{\partial \theta} = \frac{1}{M}\sum_{i=0}^{M-1}\frac{\partial C(\widehat{y}_i)}{\partial \widehat{y}_i}\frac{\partial \widehat{y}_i(f)}{\partial f}\frac{\partial f(\vec{\theta})}{\partial \theta}, \tag{12}$$

where $\theta \in \vec{\theta}$, with

$$\frac{\partial C(\widehat{y}_i)}{\partial \widehat{y}_i} = -2[y_i - \widehat{y}_i], \tag{13}$$

the derivative $\frac{\partial \widehat{y}_i(f)}{\partial f}$ depends on the particular form of the activation function, and

$$\begin{aligned}\frac{\partial f(\vec{\theta})}{\partial \theta} &= \sum_k \frac{\partial m_k(\vec{\theta})}{\partial \theta} \\ &= \sum_k c_k \Big[m_k(\vec{\theta} + s_k) - m_k(\vec{\theta} - s_k) \Big],\end{aligned} \tag{14}$$

where $m_k(\vec{\theta}) = \langle \psi(\vec{\theta}) | \widehat{\mathcal{M}}_k | \psi(\vec{\theta}) \rangle$ is a measurement outcome of an operator $\widehat{\mathcal{M}}_k$ for a measurement set $\{k\}$. To arrive at the second equality in equation (14), we have applied the parameter-shift rule (10).

4.1. Quantum Linear Regression

We implement QLR on a 'diabetes' dataset [83] available in the scikit-learn package [84] with 400 samples for the training set and 10 samples for the testing set. We write the linear regression model in the form

$$\widehat{y} = wx + b, \tag{15}$$

where w and b are the slope and intercept of the linear regression need to be obtained. To evaluate them on quantum computers, we store their values in two qubits. The values of w and b now become the expectation values of the Pauli matrix $\widehat{\sigma}_z$. The quantum version of linear regression model (15) states

$$\widehat{y}(\vec{\theta}) = k\big(\langle \widehat{w} \rangle x + \langle \widehat{b} \rangle\big), \tag{16}$$

where k is the scaling factor that transforms the output data from $[-1, 1]$ into $[-k, k]$. Its value should be chosen so that the R.H.S can cover all of the true values $\{y_i\}$. Here, w, b are functions of variational parameters $\vec{\theta}$, for example $w \equiv w(\vec{\theta}) = \langle \psi(\vec{\theta}) | \widehat{\sigma}_z | \psi(\vec{\theta}) \rangle$ with the initial state $|\psi\rangle = |0\rangle$, (see the inset figure 4). The cost function is the mean squared error as described in equation (11).

After having the model, we run the circuit in the inset figure 4 to find the optimized values for w and b via updating $\vec{\theta}$. We first calculate their derivatives using the chain rule as shown in equation (12) and the parameter-shift rule in the QDP as depicted in algorithm 3, and then update new parameters using gradient descent

$$\theta_{t+1} = \theta_t - \eta \frac{\partial C(\vec{\theta}_t)}{\partial \theta}, \tag{17}$$

where η the learning rate, $\vec{\theta}_t$ is the set of parameters at time step t. They are continuously updated until the optimized values are found. The quantum circuit that used to train the model is represented in the inset figure 4, and the code is given in appendix D.

For comparison, we next provide the analytical solution for minimizing the cost function (11) with (15). We vectorize:

$$\begin{pmatrix} w \\ b \end{pmatrix} = (X^T X)^{-1} X^T Y \tag{18}$$

where

$$X = \begin{pmatrix} x_0 & 1 \\ \vdots & \vdots \\ x_{M-1} & 1 \end{pmatrix}, Y = \begin{pmatrix} y_0 \\ \vdots \\ y_{M-1} \end{pmatrix}, \tag{19}$$

and T the transpose operator.

The performance of Qsun is compared to those of Pennylane, ProjectQ, and analytical solution using the same set of parameters. We use the maximum number of iterations 1000 and $k = 10$. For Pennylane, we use GradientDescentOptimizer() with its default configuration. For ProjectQ, we use the same optimize algorithm as we have used for Qsun, which is shown in appendix D. As we can see from figure 4, Qsun's result is closer to the analytical result than the Pennylane and ProjectQ one, which implies a high performance of Qsun. In the same case of the 'wavefunction' approach, Qsun has a better efficiency than ProjectQ in the speedup as shown in figures 2 and 3, and in the optimization process.

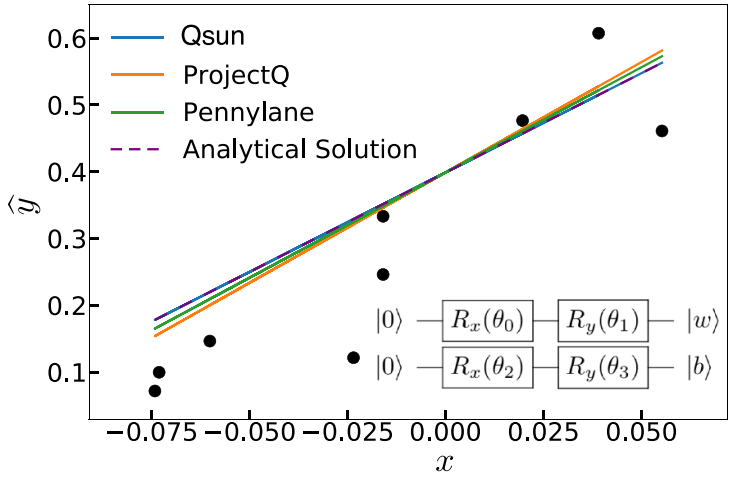


Figure 4. Linear regression models trained by quantum programming using Qsun (blue), Pennylane (green), ProjectQ (orange) and analytical solution (dashed purple). We use the diabetes dataset [83] with 400 samples for the training set and 10 samples for the testing set. The boundary $k = 10$ with 1000 iterations. Inset: the quantum circuit that trains the quantum linear regression models.

4.2. Quantum Neural Network

In this subsection, we show the ability of Qsun for deep learning by building and training a QNN. We model the QNN as a variational quantum circuit (VQC) parameterized with multiple variables that are optimized to train the model. Figure 5 represents the process for building and training our QNNs. It is a hybrid quantum–classical scheme with five steps summarized as follows:

- The quantum part:
  - Step 1: Quantizing and encoding dataset into quantum states using the amplitude encoding method [38, 85].
  - Step 2: Building a QNN circuit and measurement.
  - Step 3: Evaluating the derivative of measurement results using QDP.
- The classical part:
  - Step 4: Deriving the derivative of the defined cost (loss) function.
  - Step 5: Running a gradient-based optimization and updating parameters.

4.2.1. Data quantization and amplitude encoding

Set $x_j^{(i)}$ as the ith sample ($i \in [0, M-1]$) of the jth feature ($j \in [0, N-1]$) in the dataset of N features with M samples for each feature. Since we only analyze one sample (other samples are treated similarly), we can omit the indicator i and rewrite $x_j^{(i)}$ as x_j. We now map x_j into a qubit by normalizing its value in range $[x_j^{(\min)}, x_j^{(\max)}]$ into $[0, 1]$. Using the Min-Max normalization, we obtain

$$\widetilde{x}_j = \frac{x_j - x_j^{(\min)}}{x_j^{(\max)} - x_j^{(\min)}}. \tag{20}$$

Here, the normalized value $\widetilde{x}_j \in [0, 1]$. We map this value into the amplitudes of a qubit as $|\psi_j\rangle = \sqrt{\widetilde{x}_j}|0\rangle + \sqrt{1-\widetilde{x}_j}|1\rangle$. Then, the quantum state for N features reads

$$|\Psi\rangle = \bigotimes_{j=0}^{N-1} |\psi_j\rangle = \bigotimes_{j=0}^{N-1} \left(\sqrt{\widetilde{x}_j}|0\rangle + \sqrt{1-\widetilde{x}_j}|1\rangle\right). \tag{21}$$

The encoding implementation is given in algorithm 4.

4.2.2. Building the QNN circuits

The QNN model uses N qubits for N features and includes multiple layers. It is parameterized through a set of variables $\vec{\theta} = \{\theta_{kjj'}\}$ with k as the layer index, j as qubit (feature) indices, and j' indicates the number of rotation gates implementing on each qubit. Each layer has one R_x and R_y gates for each qubit followed by CNOT gates generating all possible entanglements between them if the number of qubits is more than one.

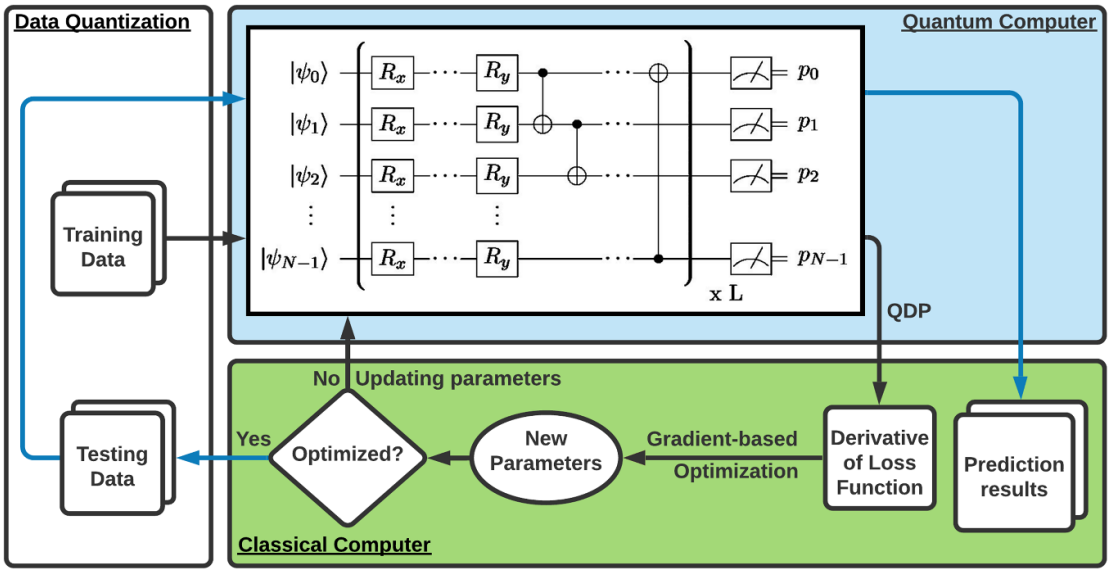


Figure 5. A quantum neural network (QNN) framework. The blue box is the quantum part of the QNN, and the green box is the classical part of the QNN. At first, quantify and encode the dataset (training and testing) into quantum states of qubits $|\psi_i\rangle$. Each feature in the dataset encodes into one qubit. The employed rotation gates will parameterize the quantum circuit, and the CNOT gates cause entangled in the circuit. These gates repeat L times for L layers in the QNN. After that, we measure the circuit and give the corresponding probabilities p_k. We employ a QDP scheme with the pentameter-shift rule to calculate the derivative of p_k and send the results to a classical computer to derive the derivative of the loss function. After that, we implement a gradient-based optimization to obtain new parameters. When the scheme is not optimal yet, we update the circuit with new parameters; when it is optimal, we turn it to the testing process.

Algorithm 4. Encoding data in Qsun.

Result: Probablity amplitudes of QNN's initial quantum states
sample ← sample_scaled;
N ← number_of_features;
ampl ← [0, …, 0], size(ampl) = N;
for i ← 0, N-1 **do**
  ampl[i] ← [sqrt(sample[i]), sqrt(1-sample[i])];
end
ampl_vec ← ampl[0] ⊗ ampl[1] ⊗ ⋯ ⊗ ampl[$N-1$];

4.2.3. Decoding probabilities into predictions

Here, we map measurement results from the previous step into classical predictions by using an activation function. We consider the expectation value of the projection operator $\widehat{\Pi} = |1\rangle\langle 1|$ as

$$p_j^{(i)} \equiv \langle \widehat{\Pi} \rangle = \langle \psi_j^{(i)} | 1 \rangle \langle 1 | \psi_j^{(i)} \rangle = |\langle 1 | \psi_j^{(i)} \rangle|^2. \tag{22}$$

Note that $|\psi_j^{(i)}\rangle$ is the final state of qubit j at sample i. We also emphasize that one can choose the projection operator $\widehat{\Pi}$ arbitrarily. We use the sigmoid function, $S(x) = 1/(1+e^{-x})$ to transform the measurement data into predictions. In our concrete example below, we have two features represented by two qubits, i.e, $j = 0, 1$. We use the prediction rule as

$$S_i(p) = S\big(\gamma(p_0^{(i)} - p_1^{(i)})\big), \tag{23}$$

where γ is a scaling shape for the sigmoid function such that for large γ then $S(x)$ becomes a Heaviside step function. To use a soft prediction, we introduce a label-conditional probability for a prediction value as

$$\widehat{y}_i(l) = \begin{cases} S_i(p), & \text{for} \quad l = 0 \\ 1 - S_i(p), & \text{for} \quad l = 1 \end{cases}. \tag{24}$$

4.2.4. Training the QNN model

In the current version of Qsun, we are using a quantum–classical hybrid scheme combining QDP and the classical Adam optimization to train our QNN model. The cost function is defined by

$$C(\vec{\theta}) = \frac{1}{M}\sum_{i=0}^{M-1}\left[1-\widehat{y}_i(y_i)\right]^2, \tag{25}$$

which is a function of $\vec{\theta}$. Here, $y_i \in \{0,1\}$ are true values, $\widehat{y}_i(y_i)$ are predicted probability conditioned on the label y_i. We then evaluate the derivative of the cost function using QDP, followed by a gradient-descent procedure to search for optimal parameters $\vec{\theta}$.

The derivative of the cost function concerning each $\theta_{kjj'} \in \vec{\theta}$ following the chain rule (12) gives (hereafter, we omit its indices)

$$\frac{\partial C(\vec{\theta})}{\partial\theta} = \frac{1}{M}\sum_{i=0}^{M-1}\frac{\partial C(y_i)}{\partial\widehat{y}_i(l)}\frac{\partial\widehat{y}_i(l)}{\partial p}\frac{\partial p(\vec{\theta})}{\partial\theta}, \tag{26}$$

with

$$\frac{\partial C(y_i)}{\partial\widehat{y}_i(l)} = -2[1-\widehat{y}_i(l)], \tag{27}$$

$$\frac{\partial\widehat{y}_i(l)}{\partial p} = (-1)^l\widehat{y}_i(l)[1-\widehat{y}_i(l)], \tag{28}$$

and

$$\begin{aligned}\frac{\partial p(\vec{\theta})}{\partial\theta} &= \gamma\frac{\partial}{\partial\theta}\left(p_0^{(i)}-p_1^{(i)}\right)\\ &= \gamma\Big\{c_0\left[p_0^{(i)}(\theta+s_0)-p_0^{(i)}(\theta-s_0)\right]\\ &\quad - c_1\left[p_1^{(i)}(\theta+s_1)-p_1^{(i)}(\theta-s_1)\right]\Big\},\end{aligned} \tag{29}$$

here, we have applied the parameter-shift rule (10). We finally update the model with new parameters using the Adaptive Moment (Adam) optimization algorithm [86] as follows. Let g_{t+1} be the gradient of the cost function w.r.t. θ at time step t+1, (t starts from 0):

$$g_{t+1} \equiv \frac{\partial C(\vec{\theta}_t)}{\partial\theta}, \tag{30}$$

where $\vec{\theta}_t$ is the set of parameters at time step t. Next, we estimate the first and second order moments of the gradient as

$$v_{t+1} = \beta_1 v_t + (1-\beta_1)g_{t+1} \tag{31}$$

$$w_{t+1} = \beta_2 w_t + (1-\beta_2)g_{t+1}^2,$$

and compute

$$\widehat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_1^{t+1}}, \text{ and } \widehat{w}_{t+1} = \frac{w_{t+1}}{1-\beta_2^{t+1}}. \tag{33}$$

Here, β_1, β_2 and ϵ are nonnegative weights, originally suggested as $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Also, $v_0 = w_0 = 0$. Finally, the parameter θ is updated to

$$\theta_{t+1} = \theta_t - \frac{\eta\widehat{v}_{t+1}}{\sqrt{\widehat{w}_{t+1}}+\epsilon}, \tag{34}$$

where η is the learning rate. The implementation of Adam algorthims in a combination with QDP is given in algorithm 5.

Algorithm 5. Adam optimization implementation in Qsun.

Result: Updated parameters
$c \leftarrow$ Quantum Circuit;
$p \leftarrow$ Params; $s \leftarrow$ Shift;
$\beta_1, \beta_2, \epsilon, \eta \leftarrow$ Beta$_1$, Beta$_2$, Epsilon, Eta;
v_adam, s_adam $\leftarrow$ v_adam, s_adam;
$t \leftarrow t^{th}$ iteration;
diff $\leftarrow$ zero_matrix, size(diff) $=$ size(p);
for i $\leftarrow$ 0, size(param) **do**
  for j $\leftarrow$ 0, size(param[i]) **do**
    for k $\leftarrow$ 0, size(param[i][j]) **do**
      p_plus $\leftarrow$ copy(p);
      p_subs $\leftarrow$ copy(p);
      p_plus[i][j][k] $\leftarrow p[i][j][k] + s$;
      p_subs[i][j][k] $\leftarrow p[i][j][k] - s$;
      diff[i][j][k] $\leftarrow$ QDP(p_plus, p_subs, c, s);
    end
  end
end
p, v_adam, s_adam $\leftarrow$ Adam(diff, p, v_adam, s_adam, η, β_1, β_2, ϵ, t) ;

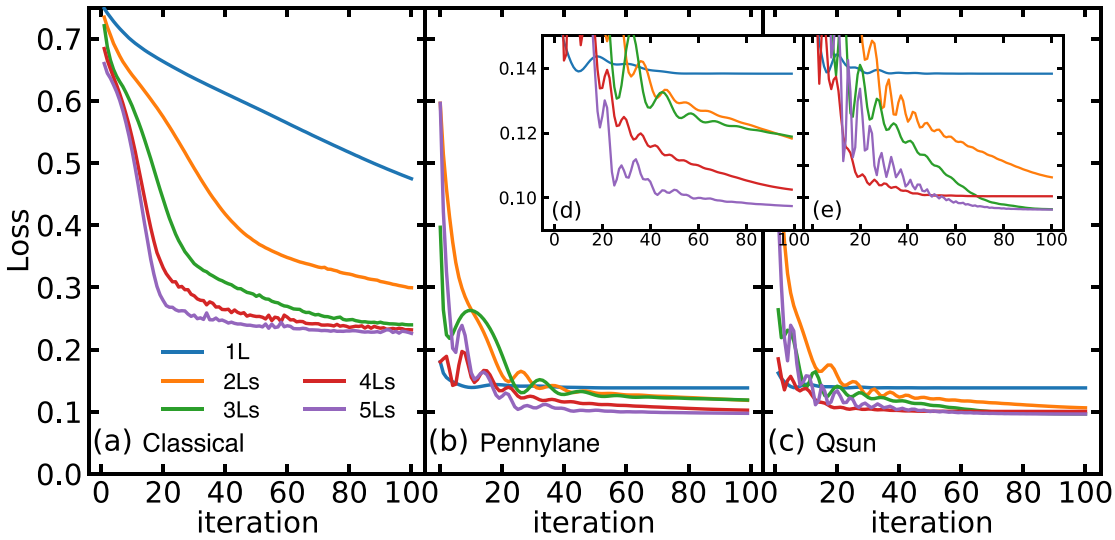


Figure 6. (a) Classical NN model; (b) and (c) QNN models using Pennylane and Qsun, respectively; (d) and (e) are the zoom-in of (b) and (c), respectively. Each plot shows from 1 layer (1L) to 5 layers (5Ls). See text for more details about parameters used for each model.

4.2.5. Preliminary QNN results

We apply the QNN model on the Social Network Ads dataset [87]. This is a categorical dataset to determine whether a user purchases a particular product or not. We consider two features 'Age' and 'EstimatedSalary' to train the model with the output 'Purchased'. We use 400 samples and split them into 323 samples for the training set and 80 samples for the testing set. Once the data is normalized, we encode 'Age' and 'EstimatedSalary' into two qubits $|\psi_j\rangle, j = 0, 1$, while $y =$ Purchased. We evaluate the predicted value $\widehat{y}$ and minimize the loss function (25) to train the QNN model. The full code is given in appendix E.

For the comparison of performance, we train the QNN model using Qsun and Pennylane. We note that while the QDP and Adam algorithm are implemented in Qsun and Pennylane separately, the QNN circuit, encoding, and decoding procedures are the same for both approaches. We also compare their results with those from a classical model to explore advantages of QNN. For the classical NN model, we use the MLPClassifier function from scikit-learn that has 100 nodes per layer with the ReLU activation function. For the QNN model, we have four nodes per layer with the sigmoid activation function. We fix $s_0 = s_1 = \pi/20$ in the parameter-shift rule and $\eta = 0.1$. For all optimization, we use the Adam algorithm with $\beta_1 = 0.9, \beta_2 = 0.99$, and $\epsilon = 10^{-6}$.

Figure 6 represents the loss function versus iteration for different layers as shown in the figure for three cases: a classical NN model (a) and a QNN model trained by Pennylane (b) and Qsun (c). In general, the loss

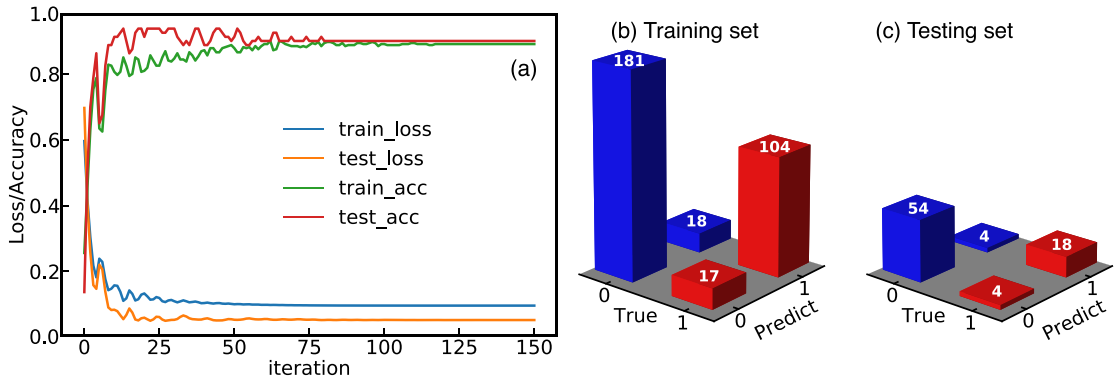


Figure 7. Results from our QNN model with five layers and two features on the Social Network Ads dataset. (a) Plot of loss and accuracy versus iteration for the training and testing process. (b), (c) The confusion matrices for the training and testing sets. Here we use 400 samples and split them into 320 samples for training and 80 samples for testing with 400 iterations.

function will reduce and get unchanged after sufficient layers. In our example, the loss function becomes saturated after five layers. As expected, the loss function of QNN model is much smaller than the the classical one. Interestingly, when comparing the two quantum approaches (see insets (d) and (e) for a zoom in), the Qsun loss is converged faster than the Pennylane one.

We next focus on our QNN model with fixed five layers. In figure 7(a), we show the reduction of cost functions and activation functions during the iteration for the training and testing processes. They both behave similarly and become saturated after 100 iterations. Figures 7(b) and (c) represent results for the training and testing sets. In the figure, both the true value y and the predicted value $\widehat{y}$ are labeled as '0' and '1' for 'no-purchase' and 'purchase', respectively. $y\widehat{y} =$ '00' or '11' indicates a correct prediction, whereas $y\widehat{y} =$ '01' or '10' indicates a wrong prediction. It can be seen from figure 7 that our QNN model has a good performance, reflected by large values of main diagonal elements $y\widehat{y} =$ '00' and '11'. Importantly, unlike the classical neural network [87], it does not require many nodes in each layer, which is one of the main advantages of QNN.

5. Conclusion

We have developed Qsun, an open-source platform of QVM, that simulates the operation of a real quantum computer using the wavefunction approach. For small qubit numbers, the current version of Qsun runs standard tasks significantly faster than other platforms do. The QDP is implemented as a built-in function of Qsun, allowing us to execute QML applications effectively. We have employed Qsun to implement two standard models in machine learning: linear regression and neural network. For the former, Qsun yields a quantum regression model nearly overlap with the classical reference, and somewhat better than the Pennylane one. For the latter, the QNN model trained using Qsun shows a good performance with a less number of nodes in each layer than the classical neural network. Although Qsun is aimed to QML problems, as a generic QVM, it can be used for multiple other purposes, such as developing and testing VQAs for electronic structures or quantum information. It is well-fit to personal use thanks to its light weight. Qsun is under active development to cover a wide range of contents in machine learning. Our code is open source and available on GitHub [78].

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://github.com/ChuongQuoc1413017/Quantum_Virtual_Machine.

Acknowledgment

This work was supported by JSPS KAKENHI Grant No. 20F20021.

Appendix A. Code for benchmarking

In this section, we present the code used to benchmark the performance of Qsun and compared it to other packages such as Qiskit, ProjectQ, and Pennylane. The criteria are the time to execute a certain quantum

circuit of constant depth. The circuit contain an H, a $\sqrt{X}$, and a CNOT with 'depth = 10' and measurements on all qubits. With the number of qubits as a variable, the time needed to execute the circuit is estimated, yielding the package's performance. The number of qubits varies from 2 to 15 qubits, which means we have to implement 14 circuits for each simulation. The implementation's time is stored in a Pandas's Dataframe and is shown in figure 2 in the main text.

Code example 1. Implementation of the benchmarking test on four typical languages: Qsun, Qiskit, ProjectQ, and Pennylane.

```
#import module for visualization
import pandas as pd
import seaborn as sns
import time
import matplotlib.pyplot as plt

#import Qiskit module
import qiskit import IBMQ, BasicAer
import qiskit.providers.ibmq import least_busy
import qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

#import ProjectQ module
from projectq import MainEngine
import projectq.ops as ops
from projectq.backends import Simulator

#import Qsun module
from Qsun.Qcircuit import *
from Qsun.Qgates import *

#import Pennylane module
import pennylane as qml
from pennylane import numpy as np

# Qsun circuit
def qvm_circuit(n_qubit, depth):
    circuit = Qubit(n_qubit)
    for m in range(depth):
        for i in range(n_qubit):
            H(circuit, i)
            Xsquare(circuit, i)
        for i in range(1, n_qubit):
            CNOT(circuit, i, 0)
    return circuit.probabilities()

# Pennylane circuit
dev = qml.device('default.qubit', wires = qubit_max)

@qml.qnode(dev)
def qml_circuit(n_qubit, depth):
      for m in range(depth):
          for i in range(n_qubit):
              qml.Hadamard(wires = i)
              qml.SX(wires = i)
          for i in range(1, n_qubit):
              qml.CNOT(wires = [i, 0])
      return qml.probs(wires = range(n_qubit))

# Qiskit circuit
def qiskit_circuit(n_qubit, depth):
      qr = QuantumRegister(n_qubit)
      cr = ClassicalRegister(n_qubit)
      circuit = QuantumCircuit(qr, cr)
      for m in range(depth):
          for i in range(n_qubit):
              circuit.h(qr[i])
```

```
            circuit.rx(math.pi/2, qr[i])
        for i in range(1, n_qubit):
            circuit.cx(qr[i], qr[0])
    circuit.measure_all()
    backend = BasicAer.get_backend('qasm_simulator')
    result = execute(circuit, backend = backend, shots = 1024).result()
    return result.get_counts(circuit)

# ProjectQ circuit
def projectq_circuit(n_qubit, depth):
    eng = MainEngine(backend = Simulator(gate_fusion = True),engine_list = [])
    qbits = eng.allocate_qureg(n_qubit)
    for level in range(depth):
        for q in qbits:
            ops.H|q
            ops.SqrtX|q
            if q ! = qbits[0]:
              ops.CNOT|(q,qbits[0])
    for q in qbits:
        ops.Measure|q
    eng.flush()

#run test for Qsun
data_qvm = []

for n_qubit in range(qubit_min, qubit_max):
    start = time.time()
    qvm_circuit(n_qubit, depth)
    end = time.time()
    data_qvm.append(end-start)

#define the test parametes
qubit_min = 2
qubit_max = 10
depth = 10

#run test for Pennylane
data_qml = []

for n_qubit in range(qubit_min, qubit_max):
    start = time.time()
    qml_circuit(n_qubit, depth)
    end = time.time()
    data_qml.append(end-start)

#run test for Qiskit
data_qiskit = []

for n_qubit in range(qubit_min, qubit_max):
    start = time.time()
    qiskit_circuit(n_qubit, depth)
    end = time.time()
    data_qiskit.append(end-start)

#run test for ProjectQ
data_projectq = []

for n_qubit in range(qubit_min, qubit_max):
    start = time.time()
    projectq_circuit(n_qubit, depth)
    end = time.time()
    data_projectq.append(end-start)

df = pd.DataFrame({'QVM': data_qvm, 'Pennylane': data_qml,
                   'Qiskit': data_qiskit, 'ProjectQ': data_projectq},
    index = range(qubit_min, qubit_max))
```

```
122
123 sns.lineplot(data = df)
124 plt.xlabel('Number of Qubits with depth = '+str(depth))
125 plt.ylabel('Time(s)')
126 plt.grid()
127 plt.savefig('compare.png')
```

Appendix B. Code for Quantum Differential Programming benchmarking

For this code, we run the QDP algorithm on Qsun and measure how long it will take to update parameters. We implement the circuit described in figure 3 and measure the expected values of output, then record the time. The number of qubits varies from 1 to 10, and the maximum number of iterations is 1000. Parameters for QDP, in this case, are $\eta = 0.1$ and $s = \pi/20$.

Code example 2. Implementation of the quantum differentiable programming test in Qsun.

```
from Qsun.Qcircuit import *
from Qsun.Qgates import *
import numpy as np
import time

def circuit(params, n):
    c = Qubit(n)
    for j in range(0, 2*n, 2):
        RX(c, j//2, params[j])
        RY(c, j//2, params[j+1])
    return c

def output(params):
    c = circuit(params, len(params)//2)
    prob = c.probabilities()
    return -np.sum([i*prob[i] for i in range(len(prob))])

def cost(params):
    expval = output(params)
    return expval

def grad(params, shift, eta):
    for i in range(len(params)):
        params_1 = params.copy()
        params_2 = params.copy()
        params_1[i] + = shift
        params_2[i] - = shift
        diff[i] = (cost(params_1)-cost(params_2))/(2*np.sin(shift))
    for i in range(len(params)):
        params[i] = params[i] - eta*diff[i]
    return params

time_qvm = []
n = 10
for i in range(1, n+1):

    start = time.time()
    params = np.random.normal(size = (2*i,))
    diff = np.random.normal(size = (2*i,))

    for i in range(1000):
        params = grad(params, np.pi/20, eta = 0.01)
    end = time.time()

    time_qvm.append(end-start)
    print(cost(params))
```

Appendix C. Quantum Differential Programming tutorial appendix

This appendix demonstrates how to run a gradient descent algorithm using QDP in Qsun. We use $\eta = 0.1$ and $s = \pi/20$ to find the derivative of a circuit. By doing that, we maximize the expectation values of one qubit. So the objective function we want to maximize is $|\langle 1|\psi\rangle|^2$, where $|\psi\rangle$ is a quantum state of that qubit.

Code example 3. Implementation of the Gradient Descent algorithm by using QDP in Qsun.

```
import numpy as np
from Qsun.Qcircuit import *
from Qsun.Qgates import *

def circuit(params):
    c = Qubit(1)
    RX(c, 0, params[0])
    RY(c, 0, params[1])
    return c

def output(params):
    c = circuit(params)
    prob = c.probabilities()
    return 0.*prob[0] + 1*prob[1]

def cost(params):
    c = circuit(params)
    prob = c.probabilities()
    expval = output(params)
    return np.abs(expval - 1)**2

def grad(params, shift, eta):
    for i in range(len(params)):
        params_1 = params.copy()
        params_2 = params.copy()
        params_1[i] + = shift
        params_2[i] - = shift
        diff[i] = (cost(params_1)-cost(params_2))/(2*np.sin(shift))
    for i in range(len(params)):
        params[i] = params[i] - eta*diff[i]
    return params

params = np.random.normal(size = (2,))
diff = np.random.normal(size = (2,))

for i in range(1000):
    params = grad(params, np.pi/20, eta = 0.01)

print('Circuit output:', output(params))
print('Final parameter:', params)

>>> Circuit output: -0.9381264201123851
>>> Final parameter: [ 0.29017649 -2.93657549]
```

Appendix D. Quantum Linear Regression appendix

Here, a QLR model is programmed in Qsun, with results shown in figure 4. Its parameters are $k = 10$, $\eta = 0.1$, and $s = \pi/20$. The optimization algorithm used in this code is Gradient Descent.

Code example 4. Implementation of the linear regression model in qSUN.

```
from Qsun.Qcircuit import *
from Qsun.Qgates import *
from sklearn import datasets

def circuit(params):
    c = Qubit(1)
    RX(c, 0, params[0])
    RY(c, 0, params[1])
    return c

def output(params):
    c = circuit(params)
    prob = c.probabilities()
    return 1*prob[0] - 1*prob[1]

def predict(x_true, coef_params, intercept_params, boundary = 10):
    coef = boundary*output(coef_params)
    intercept = boundary*output(intercept_params)
    return coef*x_true.flatten() + intercept

def errors(x_true, y_true, coef_params, intercept_params, boundary):
    return mean_squared_error(y_true, predict(x_true, coef_params, intercept_params,
    boundary))

def grad(x_true, y_true, coef_params, intercept_params, shift, eta, boundary):

    coef_diff = np.zeros((2,))
    intercept_diff = np.zeros((2,))

    for i in range(len(coef_params)):
        coef_params_1 = coef_params.copy()
        coef_params_2 = coef_params.copy()
        coef_params_1[i] += shift
        coef_params_2[i] -= shift
        for x, y in zip (x_true, y_true):
            coef_diff[i] -= x*(y-predict(x, coef_params, intercept_params,
    boundary))*(output(coef_params_1)-output(coef_params_2))/(2*np.sin(shift))


    for i in range(len(coef_params)):
        intercept_params_1 = intercept_params.copy()
        intercept_params_2 = intercept_params.copy()
        intercept_params_1[i] += shift
        intercept_params_2[i] -= shift
        for x, y in zip (x_true, y_true):
            intercept_diff[i] -= (y-predict(x, coef_params, intercept_params,
    boundary))*(output(intercept_params_1)-output(intercept_params_2))/
    (2*np.sin(shift))

    coef_diff = coef_diff*boundary*2/len(y_true)
    intercept_diff = intercept_diff*boundary*2/len(y_true)

    for i in range(len(coef_params)):
        coef_params[i] = coef_params[i] - eta*coef_diff[i]

    for i in range(len(intercept_params)):
        intercept_params[i] = intercept_params[i] - eta*intercept_diff[i]

    return coef_params, intercept_params

X, y = datasets.load_diabetes(return_X_y = True)
y = (y - np.min(y)) / (np.max(y) - np.min(y))
```

```
# Use only one feature
X = X[:, np.newaxis, 2]

# Split the data into training/testing sets
X_train = X[:400]
X_test = X[-10:]

# Split the targets into training/testing sets
y_train = y[:400]
y_test = y[-10:]

coef_params = np.random.normal(size = (2,))
intercept_params = np.random.normal(size = (2,))

start = time.time()
for i in range(1000):
    coef_params, intercept_params = grad(X_train, y_train,
    coef_params, intercept_params, np.pi/20, eta = 0.01, boundary = 10)
end = time.time()

y_pred = predict(X_test, coef_params, intercept_params, boundary = 10)
```

Appendix E. Quantum Neural Network appendix

The code showing in this appendix implements a process from training a QNN model with five layers and two qubits on the 'Social_Network_Ads' dataset. From this dataset, we use a QNN to predict whether a customer buys a car or not. The function train_test_split() splits the original dataset into the training and testing datasets. MinMaxScaler() scales the training dataset to [0, 1] before feeding it into the QNN circuit. The optimization algorithm in this model is the Adam optimization algorithm, as presented in the main text. The parameters used in the model are: test_size $= 0.2$; random_state $= 0$; s $= \pi/20$; $\eta = 0.1$; $\beta_1 = 0.9, \beta_2 = 0.99$; $\epsilon = 10^{-6}$; number_of_iterations $= 150$.

Code example 5. Implementation of the QNN with two layers and two features in Qsun.

```
# import libraries
from Qsun.Qcircuit import *
from Qsun.Qgates import *
from Qsun.Qmeas import *
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt
import time

# one layer with full entanglement
def layer(circuit, params):
    circuit_layer = circuit
    n_qubit = len(params)
    for i in range(n_qubit):
        RX(circuit_layer, i, params[i][0])
        RY(circuit_layer, i, params[i][1])
    for i in range(n_qubit-1):
        CNOT(circuit_layer, i, i+1)
    CNOT(circuit_layer, n_qubit-1, 0)
    return circuit_layer

# encoding the features
def initial_state(sample):
    circuit_initial = Qubit(len(sample))
```

```
    ampli_vec = np.array([np.sqrt(sample[0]), np.sqrt(1-sample[0])])
    for i in range(1, len(sample)):
        ampli_vec = np.kron(ampli_vec, np.array([np.sqrt(sample[i]),
    np.sqrt(1-sample[i])]))
    circuit_initial.amplitude = ampli_vec
    return circuit_initial

# QNN circuit
def qnn(circuit, params):
    n_layer = len(params)
    circuit_qnn = circuit
    for i in range(n_layer):
        circuit_qnn = layer(circuit_qnn, params[i])
    return circuit_qnn

# QNN model
def qnn_model(sample, params):
    circuit_model = initial_state(sample)
    circuit_model = qnn(circuit_model, params)
    return circuit_model

# activation function
def sigmoid(x):
    return 1 / (1 + math.exp(-10*x))

# make a prediction
def predict(circuit):
    prob_0 = measure_one(circuit, 0)
    prob_1 = measure_one(circuit, 1)
    expval_0 = prob_0[1]
    expval_1 = prob_1[1]
    pred = sigmoid(expval_0-expval_1)
    return [pred, 1-pred]

# make a prediction for exp
def predict_ex(circuit):
    prob_0 = measure_one(circuit, 0)
    prob_1 = measure_one(circuit, 1)
    expval_0 = prob_0[1]
    expval_1 = prob_1[1]
    return [expval_0, expval_1]

# loss function
def square_loss(labels, predictions):
    loss = 0
    for l, p in zip(labels, predictions):
        loss = loss + (1 - p[l]) ** 2
    loss = loss / len(labels)
    return loss

# loss function of QNN
def cost(params, features, labels):
    preds = [predict(qnn_model(x, params)) for x in features]
    return square_loss(labels, preds)

# https://d2l.ai/chapter_optimization/adam.html?highlight=adam
def adam(X_true, y_true, params, v, s, shift, eta, drop_rate, beta1, beta2, eps,
    iter_num):
    diff = np.zeros(params.shape)
    for i in range(len(params)):
        for j in range(len(params[i])):
            for k in range(len(params[i][j])):
                dropout = np.random.choice([0, 1], 1, p = [1 - drop_rate,
    drop_rate])[0]
```

```
            if dropout == 0:
                params_1 = params.copy()
                params_2 = params.copy()
                params_1[i][j][k] += shift
                params_2[i][j][k] -= shift
                for x, y in zip(X_true, y_true):
                    circuit = qnn_model(x, params)
                    circuit_1 = qnn_model(x, params_1)
                    circuit_2 = qnn_model(x, params_2)
                    ex_plus = predict_ex(circuit_1)
                    ex_subs = predict_ex(circuit_2)
                    pred = predict(circuit)
                    diff_loss = ((-1)**y)*(-2)*(1-pred[y])*pred[y]*(1-pred[y])
                    diff_ex = 10*((ex_plus[0] - ex_subs[0]) - (ex_plus[1] -
    ex_subs[1]))/(2*np.sin(shift))
                    diff[i][j][k] += diff_loss*diff_ex

    diff /= len(y_true)
    v = beta1 * v + (1 - beta1) * diff
    s = beta2 * s + (1 - beta2) * (diff**2)
    v_bias_corr = v / (1 - beta1 ** (iter_num+1))
    s_bias_corr = s / (1 - beta2 ** (iter_num+1))
    params -= eta * v_bias_corr / (np.sqrt(s_bias_corr) + eps)

    return params, v, s

# source: https://www.kaggle.com/rakeshrau/social-network-ads
dataset = pd.read_csv('Social_Network_Ads.csv')
X = dataset.iloc[:, 2:-1].values
y = dataset.iloc[:, -1].values

# splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
    random_state = 0)

# scaling feature
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# create parameters
n_layer = 5
params = np.random.normal(size = (n_layer, len(X_train[0]), 2,))
v = np.zeros(params.shape)
s = np.zeros(params.shape)

# training model
start = time.time()
for i in range(150):
    params, v, s = adam(X_train, y_train, params, v, s,
                        shift = np.pi/20, eta = 0.1, drop_rate = 0.0,
    beta1 = 0.9, beta2 = 0.999, eps = 1e-6, iter_num = i)

# confusion matrix
label = y_test
pred = [predict(qnn_model(x, params)) for x in X_test]
pred = np.argmax(pred, axis = 1)
con = confusion_matrix(label,pred)
sn.heatmap(con, annot = True, cmap = ''Blues'')
```

ORCID iD

Le Bin Ho https://orcid.org/0000-0002-8816-4450

References

[1] Arute F *et al* 2019 Quantum supremacy using a programmable superconducting processor *Nature* **574** 505–10
[2] Zhong H-S *et al* 2020 Quantum computational advantage using photons *Science* **370** 1460–3
[3] Kandala A, Temme K, Córcoles A D, Mezzacapo A, Chow J M and Gambetta J M 2019 Error mitigation extends the computational reach of a noisy quantum processor *Nature* **567** 491–5
[4] Khammassi N, Ashraf I, Fu X, Almudever C G and Bertels K 2017 Qx: a high-performance quantum computer simulation platform *Design, Automation Test in Conf. Exhibition (DATE), 2017* pp 464–9
[5] Preskill J 2018 Quantum computing in the NISQ era and beyond *Quantum* **2** 79
[6] Wang J *et al* 2018 Multidimensional quantum entanglement with large-scale integrated optics *Science* **360** 285–91
[7] Biamonte J 2021 Universal variational quantum computation *Phys. Rev.* A **103** L030401
[8] Cerezo M *et al* 2021 Variational quantum algorithms *Nat. Rev. Phys.* **3** 625
[9] Lubasch M, Joo J, Moinier P, Kiffner M and Jaksch D 2020 Variational quantum algorithms for nonlinear problems *Phys. Rev.* A **101** 010301
[10] Gard B T, Zhu L, Barron G S, Mayhall N J, Economou S E and Barnes E 2020 Efficient symmetry-preserving state preparation circuits for the variational quantum eigensolver algorithm *npj Quantum Inf.* **6** 10
[11] Kirby W M, Tranter A and Love P J 2021 Contextual subspace variational quantum eigensolver *Quantum* **5** 456
[12] Nakanishi K M, Mitarai K and Fujii K 2019 Subspace-search variational quantum eigensolver for excited states *Phys. Rev. Res.* **1** 033062
[13] Peruzzo A, McClean J, Shadbolt P, Yung M-H, Zhou X-Q, Love P J, Aspuru-Guzik Aan and O'Brien J L 2014 A variational eigenvalue solver on a photonic quantum processor *Nat. Commun.* **5** 4213
[14] Ryabinkin I G, Genin S N and Izmaylov A F 2019 Constrained variational quantum eigensolver: quantum computer search engine in the fock space *J. Chem. Theory Comput.* **15** 249–55
[15] Seki K, Shirakawa T and Yunoki S 2020 Symmetry-adapted variational quantum eigensolver *Phys. Rev.* A **101** 052340
[16] Tkachenko N V, Sud J, Zhang Y, Tretiak S, Anisimov P M, Arrasmith A T, Coles P J, Cincio L and Dub P A 2021 Correlation-informed permutation of qubits for reducing ansatz depth in the variational quantum eigensolver *PRX Quantum* **2** 020337
[17] Lee C K, Patil P, Zhang S and Hsieh C Y 2021 Neural-network variational quantum algorithm for simulating many-body dynamics *Phys. Rev. Res.* **3** 023095
[18] Li Y and Benjamin S C 2017 Efficient variational quantum simulator incorporating active error minimization *Phys. Rev.* X **7** 021050
[19] McArdle S, Jones T, Endo S, Li Y, Benjamin S C and Yuan X 2019 Variational ansatz-based quantum simulation of imaginary time evolution *npj Quantum Inf.* **5** 75
[20] Nishi H, Kosugi T and Matsushita Y-ichiro 2021 Implementation of quantum imaginary-time evolution method on NISQ devices by introducing nonlocal approximation *npj Quantum Inf.* **7** 85
[21] Yao Y-X, Gomes N, Zhang F, Wang C-Z, Ho K-M, Iadecola T and Orth P P 2021 Adaptive variational quantum dynamics simulations *PRX Quantum* **2** 030307
[22] Zhang Z-J, Sun J, Yuan X and Yung M-H 2020 Low-depth hamiltonian simulation by adaptive product formula (arXiv:2011.05283 [quant-ph])
[23] Bravo-Prieto C, LaRose R, Cerezo M, Subasi Y, Cincio L and Coles P J 2020 Variational quantum linear solver (arXiv:1909.05820)
[24] Huang H-Y, Bharti K and Rebentrost P 2019 Near-term quantum algorithms for linear systems of equations (arXiv:1909.07344 [quant-ph])
[25] Kubo K, Nakagawa Y O, Endo S and Nagayama S 2021 Variational quantum simulations of stochastic differential equations *Phys. Rev.* A **103** 052425
[26] Kyriienko O, Paine A E and Elfving V E 2021 Solving nonlinear differential equations with differentiable quantum circuits *Phys. Rev.* A **103** 052416
[27] LaRose R, Tikku A, O'Neel-Judy E, Cincio L and Coles P J 2019 Variational quantum state diagonalization *npj Quantum Inf.* **5** 57
[28] Lloyd S, Mohseni M and Rebentrost P 2014 Quantum principal component analysis *Nat. Phys.* **10** 631–3
[29] Somma R D and Subaşı Y 2021 Complexity of quantum state verification in the quantum linear systems problem *PRX Quantum* **2** 010315
[30] Subaşı Y, Somma R D and Orsucci D 2019 Quantum algorithms for systems of linear equations inspired by adiabatic quantum computing *Phys. Rev. Lett.* **122** 060504
[31] Xu X, Sun J, Endo S, Li Y, Benjamin S C and Yuan X 2021 Variational algorithms for linear algebra Science Bulletin **66** 2181–8
[32] Beer K, Bondarenko D, Farrelly T, Osborne T J, Salzmann R, Scheiermann D and Wolf R 2020 Training deep quantum neural networks *Nat. Commun.* **11** 808
[33] Cong I, Choi S and Lukin M D 2019 Quantum convolutional neural networks *Nat. Phys.* **15** 1273–8
[34] Farhi E and Neven H 2018 Classification with quantum neural networks on near term processors (arXiv:1802.06002 [quant-ph])
[35] Havlíček Věch, Córcoles A D, Temme K, Harrow A W, Kandala A, Chow J M and Gambetta J M 2019 Supervised learning with quantum-enhanced feature spaces *Nature* **567** 209–12
[36] Mitarai K, Negoro M, Kitagawa M and Fujii K 2018 Quantum circuit learning *Phys. Rev.* A **98** 032309
[37] Pesah A, Cerezo M, Wang S, Volkoff T, Sornborger A T and Coles P J 2021 Absence of barren plateaus in quantum convolutional neural networks Phys. Rev. X **11** 041011
[38] Schuld M, Bocharov A, Svore K M and Wiebe N 2020 Circuit-centric quantum classifiers *Phys. Rev.* A **101** 032308
[39] Schuld M and Killoran N 2019 Quantum machine learning in feature hilbert spaces *Phys. Rev. Lett.* **122** 040504
[40] Zhang K, Hsieh M-H, Liu L and Tao D 2020 Toward trainability of quantum neural networks (arXiv:2011.06258 [quant-ph])
[41] Arrasmith A, Cincio L, Sornborger A T, Zurek W H and Coles P J 2019 Variational consistent histories as a hybrid algorithm for quantum foundations *Nat. Commun.* **10** 3438
[42] Beckey J L, Cerezo M, Sone A and Coles P J 2020 Variational quantum algorithm for estimating the quantum fisher information *Quantum* **4** 248
[43] Bravo-Prieto C, García-Martín D and Latorre J I 2020 Quantum singular value decomposer *Phys. Rev.* A **101** 062310
[44] Kaubruegger R, Silvi P, Kokail C, van Bijnen R, Rey A M, Ye J, Kaufman A M and Zoller P 2019 Variational spin-squeezing algorithms on programmable quantum sensors *Phys. Rev. Lett.* **123** 260505
[45] Koczor Balint, Endo S, Jones T, Matsuzaki Y and Benjamin S C 2020 Variational-state quantum metrology *New J. Phys.* **22** 083038
[46] Meyer J J, Borregaard J and Eisert J 2021 A variational toolbox for quantum multi-parameter estimation *npj Quantum Inf.* **7** 89

[47] Wilde M 2017 *Quantum Information Theory* (Cambridge: Cambridge University Press)
[48] John N A and Mead R 1965 A simplex method for function minimization *Comput. J.* **7** 308–13
[49] Harrow A W and Napp J C 2021 Low-depth gradient measurements can improve convergence in variational hybrid quantum-classical algorithms *Phys. Rev. Lett.* **126** 140502
[50] Lopatnikova A and Tran M-N 2021 Quantum natural gradient for variational bayes (arXiv:2106.05807 [quant-ph])
[51] Schäfer F, Kloc M, Bruder C and Lörch N 2020 A differentiable programming method for quantum control *Mach. Learn.: Sci. Technol.* **1** 035009
[52] Stokes J, Izaac J, Killoran N and Carleo G 2020 Quantum natural gradient *Quantum* **4** 269
[53] Yamamoto N 2019 On the natural gradient for variational quantum eigensolver
[54] Zhu S, Hung S-H, Chakrabarti S and Wu. X 2020 On the principles of differentiable quantum programming languages *Proc. 41st ACM Conf. on Programming Language Design and Implementation (PLDI 2020)* (New York, NY: Association for Computing Machinery) pp 272–85
[55] Banchi L and Crooks G E 2021 Measuring analytic gradients of general quantum evolution with the stochastic parameter shift rule *Quantum* **5** 386
[56] Bergholm V *et al* 2020 Pennylane: automatic differentiation of hybrid quantum-classical computations (arXiv:1811.0496 [quant-ph])
[57] Guerreschi G G and Smelyanskiy M 2017 Practical optimization for hybrid quantum-classical algorithms (arXiv:1701.01450 [quant-ph])
[58] Li J, Yang X, Peng X and Sun C-P 2017 Hybrid quantum-classical approach to quantum optimal control *Phys. Rev. Lett.* **118** 150503
[59] Schuld M, Bergholm V, Gogolin C, Izaac J and Killoran N 2019 Evaluating analytic gradients on quantum hardware *Phys. Rev.* A **99** 032331
[60] Qiskit *Qiskit: Open-source quantum development* (available at: https://qiskit.org/)
[61] Cirq Developers 2021 Cirq (*See full list of authors on Github:* https://github.com/quantumlib/Cirq/graphs/contributors) (https://doi.org/10.5281/zenodo.5182845)
[62] Roberts C, Milsted A, Ganahl M, Zalcman A, Fontaine B, Zou Y, Hidary J, Vidal G and Leichenauer S 2019 Tensornetwork: a library for physics and machine learning (arXiv:1905.01330 [physics.comp-ph])
[63] Smith R S, Curtis M J and Zeng W J 2017 A practical quantum instruction set architecture (arXiv:1608.03355 [quant-ph])
[64] Svore K *et al* 2018 Q#: enabling scalable quantum computing and development with a high-level dsl *Proc. Real World Domain Specific Languages Workshop 2018 (RWDSL2018)* (New York: Association for Computing Machinery)
[65] Villalonga B, Boixo S, Nelson B, Henze C, Rieffel E, Biswas R and Mandrà S 2019 A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware *npj Quantum Inf.* **5** 86
[66] De Raedt H, Jin F, Willsch D, Willsch M, Yoshioka N, Ito N, Yuan S and Michielsen K 2019 Massively parallel quantum computer simulator, eleven years later *Comput. Phys. Commun.* **237** 47–61
[67] De Raedt K, Michielsen K, De Raedt H, Trieu B, Arnold G, Richter M, Lippert T, Watanabe H and Ito N 2007 Massively parallel quantum computer simulator *Comput. Phys. Commun.* **176** 121–36
[68] Guerreschi G G, Hogaboam J, Baruffa F and Sawaya N P D 2020 Intel quantum simulator: a cloud-ready high-performance simulator of quantum circuits *Quantum Sci. Technol.* **5** 034007
[69] Jones T, Brown A, Bush I and Benjamin S C 2019 Quest and high performance simulation of quantum computers *Sci. Rep.* **9** 10736
[70] Kelly A 2018 Simulating quantum computers using opencl (arXiv:1805.00988 [quant-ph])
[71] Steiger D S, Häner T and Troyer M 2018 ProjectQ: an open source software framework for quantum computing *Quantum* **2** 49
[72] Suzuki Y *et al* 2021 Qulacs: a fast and versatile quantum circuit simulator for research purpose Quantum **5** 559
[73] Markov I L and Shi Y 2008 Simulating quantum computation by contracting tensor networks *SIAM J. Comput.* **38** 963–81
[74] Pan F, Zhou P, Li S and Zhang P 2020 Contracting arbitrary tensor networks: general approximate algorithm and applications in graphical models and quantum circuit simulations *Phys. Rev. Lett.* **125** 060503
[75] Markov I L, Fatima A, Isakov S V and Boixo S 2018 Quantum supremacy is both closer and farther than it appears (arXiv:1807.10749 [quant-ph])
[76] Broughton M *et al* 2020 Tensorflow quantum: a software framework for quantum machine learning (arXiv:2003.02989 [quant-ph])
[77] Huggins W, Patil P, Mitchell B, Birgitta Whaley K and Stoudenmire E M 2019 Towards quantum machine learning with tensor networks *Quantum Sci. Technol.* **4** 024001
[78] Quoc C N, Ho L B, Tran L N and Nguyen H Q 2021 Qsun, a platform towards quantum machine learning (available at: https://github.com/ChuongQuoc1413017/Quantum_Virtual_Machine)
[79] Nielsen M A and Chuang I L 2019 *Quantum Computation and Quantum Information* (Cambridge: Cambridge University Press)
[80] LaRose R 2019 Overview and comparison of gate level quantum software platforms (arXiv:1807.02500 [quant-ph])
[81] Miller W 1990 *Symmetry Groups and Their Applications* (New York: Academic)
[82] Crooks G E 2019 Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition (arXiv:1905.13311 [quant-ph])
[83] Efron B, Hastie T, Johnstone I and Tibshirani R 2004 Least angle regression *Ann. Stat.* **32** 407–99
[84] Pedregosa F *et al* 2011 Scikit-learn: machine learning in Python *J. Mach. Learn. Res.* **12** 2825–30
[85] LaRose R and Coyle B 2020 Robust data encodings for quantum classifiers *Phys. Rev.* A **102** 032420
[86] Kingma D P and Ba J 2017 Adam: a method for stochastic optimization (arXiv:1412.6980 [cs.LG])
[87] Social network Ads KNN, LR, neural network (available at: www.kaggle.com/sunilsharanappa/social-network-ads-knn-lr-neural-network)