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

Chuong Nguyen Quoc,1 Le Bin Ho,2,3,† Lan Nguyen Tran,2 and Hung Q. Nguyen4,‡

1 Vietnamese-German University, Ho Chi Minh City, Vietnam, 70000
2 Ho Chi Minh City Institute of Physics, National Institute of Mechanics and Informatics, Vietnam Academy of Science and Technology, Ho Chi Minh City, 700000, Vietnam
3 Research Institute of Electrical Communication, Tohoku University, Sendai, Japan, 980-8577
4 Nano and Energy Center, VNU University of Science, Vietnam National University, Hanoi, Vietnam, 120401

(Dated: March 2, 2022)

Currently, quantum hardware is restrained by noises and qubit numbers. Thus, a quantum virtual machine 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 have been proposed and tested on quantum virtual machines to surpass the limitations of quantum hardware. Our goal is to exploit further the variational quantum algorithms towards practical applications of quantum machine learning using state-of-the-art quantum computers. In this paper, we first introduce a quantum virtual machine named Qsun, whose operation is underlined by quantum state wavefunctions. The platform provides native tools supporting variational quantum algorithms. Especially using the parameter-shift rule, we implement quantum differentiable programming essential for gradient-based optimization. We then report two tests representative of quantum machine learning: quantum linear regression and quantum neural network.

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

I. 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 computers NISQ. 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: (i) initializing quantum states for a given wavefunction ansatz, (ii) measuring a cost function suitable for problems being considered, and (iii) 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 [7, 41–47]. Typically, VQAs employ variational quantum circuits 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 [32, 49, 50, 55–59]. It relies on a technique called the parameter-shift rule that evaluates the derivative of any differentiable function using quantum circuits [49, 51]. Therefore, this method is beneficial for developing “on-circuit” gradient-based optimization techniques, especially for quantum machine learning (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: (i) the

* Electronic address: quoc.chuong1413017@gmail.com
† Electronic address: binho@riec.tohoku.ac.jp
‡ Electronic address: hungngq@hus.edu.vn
matrix multiplication approach \cite{4, 56, 60, 65}, and (ii) the wavefunction approach \cite{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 \cite{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 \cite{11} for convenient purposes.

There are several QVM’s libraries developed for QML orientation, such as TensorFlow Quantum library \cite{70} implemented in Cirq \cite{61}, Pennylane \cite{56} designed for photons devices, and TensorNetwork \cite{65, 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 quantum machine learning. Two representative examples of QML are demonstrated: quantum linear regression and quantum neural networks. As depicted in Fig. 1, Qsun consists of three main modules Qwave, Qgates, and Qmeas for quantum register, quantum gates, and quantum measurement, respectively.

\textbf{Qwave}

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

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

where $\alpha_j$ 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 Fig. 1 and Table 1. The class allows us to access and update amplitudes directly corresponding to the evolution of the quantum state under the action of the unitary quantum gates. It also measures probabilities that contain output information.

\textbf{Qgates}

To manipulate a single-qubit gate $U = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$ acting on the $n$th qubit, the nonzero elements in amplitude array are updated as \cite{70}

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

where $s_i = \text{floor}(i/2^n)2^{n+1} + (i \mod 2^n)$, for all $i \in [0, 2^{N-1} - 1]$. Here, \text{floor}(x) is the standard floor function taking the greatest integer less than or equal to the

\section{II. 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 quantum virtual machine (QVM) named Qsun, an open-source platform simulating the operation of a generic quantum computer \cite{75}. We aim the platform to the development of quantum machine learning (QML) and related problems. We develop it in Python and employ the Numpy library for fast matrix operations and numerical computations.

\subsection{A. Simulating quantum computers using wavefunction basis}

Unlike widely-used approaches based on matrix multiplication \cite{4, 56, 60, 65}, our platform is developed using the class of “wavefunction” approach \cite{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 variational quantum circuits in quantum neural networks (QNNs). As depicted in Fig. 1 Qsun consists of three main modules Qwave, Qgates, and Qmeas for quantum register, quantum gates, and quantum measurement, respectively.
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 gates 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 qubits. This generic algorithm allows us to 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 $\rho$ will transform it to

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

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

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

**Algorithm 1: Operation of a single-qubit gate:**

Result: Wavefunction with new probability amplitudes

$w \leftarrow$ Wavefunction;

states $\leftarrow$ w.state; ampl $\leftarrow$ w.amplitude;

$N \leftarrow \text{size}(\text{state}[0])$; $n \leftarrow$ target qubit;

$n\_ampl \leftarrow [0, \ldots, 0]$, size($n\_ampl$) = size(ampl);

cut $\leftarrow 2^{N-n-1}$;

for $i \leftarrow 0$, size(ampl) do

if state[i][n] $= 0$ then

$$n\_ampl[i] \leftarrow n\_ampl[i] + a^*\text{ampl}[i];$$

$$n\_ampl[i + \text{cut}] \leftarrow n\_ampl[i + \text{cut}] + b^*\text{ampl}[i];$$

else

$$n\_ampl[i] \leftarrow n\_ampl[i] + d^*\text{ampl}[i];$$

$$n\_ampl[i - \text{cut}] \leftarrow n\_ampl[i - \text{cut}] + e^*\text{ampl}[i];$$

end

end

$w\_amplitude \leftarrow n\_ampl$

**Algorithm 2: Operation of Hadamard gate:**

Result: Wavefunction with new probability amplitudes corresponding to Hadamard state

$w \leftarrow$ Wavefunction;

states $\leftarrow$ w.state; ampl $\leftarrow$ w.amplitude;

$N \leftarrow \text{size}(\text{state}[0])$; $n \leftarrow$ target qubit;

$n\_ampl \leftarrow [0, \ldots, 0]$, size($n\_ampl$) = size(ampl);

cut $\leftarrow 2^{N-n-1}$;

for $i \leftarrow 0$, size(ampl) do

if state[i][n] $= 0$ then

$$n\_ampl[i] \leftarrow n\_ampl[i] + (1/\sqrt{2})^*\text{ampl}[i];$$

$$n\_ampl[i + \text{cut}] \leftarrow n\_ampl[i + \text{cut}] + (1/\sqrt{2})^*\text{ampl}[i];$$

else

$$n\_ampl[i] \leftarrow n\_ampl[i] + (-1/\sqrt{2})^*\text{ampl}[i];$$

$$n\_ampl[i - \text{cut}] \leftarrow n\_ampl[i - \text{cut}] + (1/\sqrt{2})^*\text{ampl}[i];$$

end

end

$w\_amplitude \leftarrow n\_ampl$
Table II. Comparison between Qsun and other simulators

| Algorithms                              | Qsun | ProjectQ | Qiskit | Pennylane |
|-----------------------------------------|------|----------|--------|-----------|
| Standard algorithms                     | ✔    | ✔        | ✔      | ✔         |
| Quantum differentiable programming      | ✔    | ✔        | ✔      | ✔         |
| Quantum Linear Regression               | ✔    | ✔        | ✔      | ✔         |
| Quantum Neural Network                  | ✔    | ✔        | ✔      | ✔         |

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

$$p(1) = 1 - p(0),$$

and the post-state is

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

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.

B. Assessing the QVM 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 Ref. [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 [A].

Fig. 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 Ref. [66]. 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 [66]. We further summarize a comparison between Qsun and other simulators in terms of practical quantum algorithms in Table III.

III. QUANTUM DIFFERENTIABLE PROGRAMMING IMPLEMENTATION IN QSUN

Given a quantum state $|\psi(\vec{\theta})\rangle$ with $\vec{\theta}$ as variational parameters and an observable $\hat{C}$, the task is to seek the global minimum of the expectation value $C(\vec{\theta}) = \langle \psi(\vec{\theta}) | \hat{C} | \psi(\vec{\theta}) \rangle$. 

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

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

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

$$|\psi'\rangle = \frac{\sum_{i=0}^{2^{N-1}-1} |j_{s_i}\rangle \langle j_{s_i}|\psi\rangle}{\sqrt{p(0)}}.$$
\[ \langle \psi (\bar{\theta}) | \hat{C} | \psi (\bar{\theta}) \rangle \] with respect to parameters \( \bar{\theta} \). For example, if \( \hat{C} \) is a Hamiltonian, its global minimum is the ground state energy. In general, \( C(\bar{\theta}) \) is called as the cost function, and minimizing the cost function requires its derivative with respect to parameters \( \bar{\theta}, \partial C(\bar{\theta})/\partial \theta \). In classical computing, if the analytical form of \( \partial C(\bar{\theta})/\partial \theta \) 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, quantum differentiable programming (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.

**Algorithm 3: Quantum differentiable programming implementation in Qsun**

**Result:** Derivative of a function

\[
\begin{align*}
& f \leftarrow \text{Function}; \ c \leftarrow \text{Quantum Circuit}; \\
& p \leftarrow \text{Params}; \ s \leftarrow \text{Shift}; \\
& \text{diff} \leftarrow [0, ..., 0], \ \text{size(diff)} = \text{size}(p); \\
& \text{for } i \leftarrow 0, \ \text{size(diff) do} \\
& \quad p\_plus \leftarrow \text{copy}(p); \\
& \quad p\_subs \leftarrow \text{copy}(p); \\
& \quad p\_plus[i] \leftarrow p[i] + s; \\
& \quad p\_subs[i] \leftarrow p[i] - s; \\
& \quad \text{diff[i]} \leftarrow (f(c, p\_plus) - f(c, p\_subs))/(2*\sin(s));
\end{align*}
\]

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 \( \hat{V} \) independent of \( \bar{\theta} \) such that \( |\psi (\bar{\theta})\rangle = e^{i \theta \hat{V}} |\psi \rangle \). The cost function is then rewritten as

\[
C(\bar{\theta}) = \langle \psi | e^{-i \hat{Z} \hat{V}} C e^{i \hat{Z} \hat{V}} | \psi \rangle = \text{Tr}(C e^{i \hat{Z}} | \rho \rangle), \tag{9}
\]

with \( \rho = |\psi\rangle \langle \psi |; \ Z = i \theta \hat{V} \), and the superoperator \( e^{i \hat{Z}} | \rho \rangle = e^{-i \rho e^{i \theta \hat{V}}} \). The parameter-shift rule for each \( \theta \in \bar{\theta} \) states that

\[
\frac{\partial C(\bar{\theta})}{\partial \theta} = \text{Tr}(\hat{C} \partial \theta e^{i \theta \hat{V}} | \rho \rangle) = c[C(\bar{\theta} + s) - C(\bar{\theta} - s)], \tag{10}
\]

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

\[
\begin{align*}
|\psi\rangle & \xrightarrow{\hat{C}} |\psi\rangle \\
|\psi\rangle & \xrightarrow{\hat{V}} |\psi\rangle
\end{align*}
\]

IV. QUANTUM MACHINE LEARNING APPLICATIONS USING QSUN

Various quantum machine learning models can be developed with quantum differentiable programming (QDP) implementations to evaluate the gradient and employ gradient-based optimization. This section demonstrates QML applications using Qsun in two well-known models: quantum linear regression (QLR) and quantum
neural network (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:

\[
C(\vec{\theta}) = \frac{1}{M} \sum_{i=0}^{M-1} (y_i - \hat{y}_i)^2 ,
\]

where \(\vec{\theta}\) is a tuple of variational parameters, \(y_i\) represents the true value and \(\hat{y}_i\) 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 \(f\), i.e., \(\hat{y}_i(f(\vec{\theta}))\). Then, we derive at a chain rule:

\[
\frac{\partial C(\vec{\theta})}{\partial \theta} = \frac{1}{M} \sum_{i=0}^{M-1} \frac{\partial C(\hat{y}_i)}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial f} \frac{\partial f}{\partial \theta} ,
\]

where \(\theta \in \vec{\theta}\), with

\[
\frac{\partial C(\hat{y}_i)}{\partial \hat{y}_i} = -2[y_i - \hat{y}_i],
\]

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

\[
\frac{\partial f(\vec{\theta})}{\partial \theta} = \sum_k \frac{\partial m_k(\vec{\theta})}{\partial \theta} = \sum_k c_k \left[ m_k(\vec{\theta} + s_k) - m_k(\vec{\theta} - s_k) \right],
\]

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

A. Quantum linear regression

We implement quantum linear regression on a “diabetes” dataset [33] available in the scikit-learn package [31] with 400 samples for the training set and 10 samples for the testing set. We write the linear regression model in the form

\[
\hat{y} = wx + b ,
\]

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 \(\hat{\sigma}_z\). The quantum version of linear regression model [14] states

\[
\hat{y}(\vec{\theta}) = k(\langle \hat{w} \rangle x + \langle \hat{b} \rangle) ,
\]

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 \(w \equiv w(\vec{\theta}) = \langle \psi(\vec{\theta}) | \hat{\sigma}_z | \psi(\vec{\theta}) \rangle\) with the initial state \(|\psi\rangle = |0\rangle\), (see the inset Fig. [4]). The cost function is the mean squared error as described in Eq. [11].

After having the model, we run the circuit in the inset Fig. [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 Eq. [12] and the parameter-shift rule in the QDP as depicted in Algorithm [3] and then update new parameters using gradient descent

\[
\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \frac{\partial C(\vec{\theta})}{\partial \vec{\theta}} ,
\]

where \(\eta\) 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 Fig. [4] and the code is given in [D].

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

\[
\begin{bmatrix} w \\ b \end{bmatrix} = (X^T X)^{-1} X^T Y
\]

where

\[
X = \begin{pmatrix} x_0 & 1 \\ \vdots & \vdots \\ x_{M-1} & 1 \end{pmatrix}, \quad Y = \begin{pmatrix} y_0 \\ \vdots \\ y_{M-1} \end{pmatrix},
\]

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 [D]. As we can see from Fig. [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 Fig. [2] and Fig. [3] and in the optimization process.

B. Quantum neural network

In this subsection, we show the ability of Qsun for deep learning by building and training a quantum neural network (QNN). We model the QNN as a variational quantum circuit (VQC) parameterized with multiple variables that are optimized to train the model. Fig. [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 \[36, 85\].

  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.

**Data quantization and amplitude encoding**

Set \(x_j^{(i)}\) as the \(i\)th sample \((i \in [0, M - 1])\) of the \(j\)th 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 \([x_j^{(\text{min})}, x_j^{(\text{max})}]\) into \([0, 1]\). Using the Min-Max normalization, we obtain

\[
\bar{x}_j = \frac{x_j - x_j^{(\text{min})}}{x_j^{(\text{max})} - x_j^{(\text{min})}}.
\]

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

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

The encoding implementation is given in Algorithms 4.

**Algorithm 4: Encoding data in Qsun**

**Result:** Probability amplitudes of QNN’s initial quantum states

```plaintext```
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];
```

**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 \(\theta = \{\theta_{kj}\}\) 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.

**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 \(\Pi = |1\rangle\langle 1|\) as

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

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 \(\Pi\) 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

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

where \(\gamma\) is a scaling shape for the sigmoid function such that for large \(\gamma\) then \(S(x)\) becomes a Heaviside step func-

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.
Figure 5. A Quantum Neural Network 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 quantum neural network. 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.

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

$$C(\bar{\theta}) = \frac{1}{M} \sum_{i=0}^{M-1} [1 - \hat{y}_i(y_i)]^2,$$

(25)

which is a function of $\bar{\theta}$. Here, $y_i \in \{0, 1\}$ are true values, $\hat{y}_i(y_i)$ 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 $\bar{\theta}$.

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

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

(26)

with

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

(27)

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

(28)

and

$$\frac{\partial p(\bar{\theta})}{\partial \theta} = \gamma \frac{\partial}{\partial \theta} \left( p_0(i) - p_1(i) \right)$$

$$= \gamma \left\{ c_0 [p_0(i) (\theta + s_0) - p_0(i) (\theta - s_0)] - c_1 [p_1(i) (\theta + s_1) - p_1(i) (\theta - s_1)] \right\},$$

(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. $\bar{\theta}$ at time step $t+1$, ($t$ starts from 0):

$$g_{t+1} = \frac{\partial C(\bar{\theta}_t)}{\partial \theta},$$

(30)

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

$$v_{t+1} = \beta_1 v_{t} + (1 - \beta_1)g_{t+1},$$

(31)

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

(32)
and compute
\[
\hat{\theta}_{t+1} = \theta_t - \frac{\eta \hat{v}_{t+1}}{\sqrt{\hat{w}_{t+1}} + \epsilon},
\]
where \( \eta \) is the learning rate. The implementation of Adam algorithms in a combination with QDP is given in Algorithm 5.

**Algorithm 5: Adam optimization implementation in Qsun**

**Result:** Updated parameters
\( \theta \leftarrow \text{Updated parameters} \)
\( p \leftarrow \text{Quantum Circuit; } s \leftarrow \text{Shift;} \)
\( \beta_1, \beta_2, \epsilon, \eta \leftarrow \text{Beta_1, Beta_2, Epsilon, Eta;} \)
\( v_\text{adam}, s_\text{adam} \leftarrow v_\text{adam}, s_\text{adam;} \)
\( t \leftarrow t^{th} \text{ iteration;} \)
\( \text{diff} \leftarrow \text{zero_matrix, size(diff) = size(p);} \)

for \( i \leftarrow 0, \text{size(param)} \) do
  for \( j \leftarrow 0, \text{size(param[j][j])} \) do
    \( p_\text{plus} \leftarrow \text{copy(p);} \)
    \( p_\text{subs} \leftarrow \text{copy(p);} \)
    \( p_\text{plus[i][j][k]} \leftarrow p[i][j][k] + s; \)
    \( p_\text{subs[i][j][k]} \leftarrow p[i][j][k] - s; \)
    \( \text{diff[i][j][k]} \leftarrow \text{QDP(p_\text{plus}, } p_\text{subs, c, s);} \)
  end
end
\( p, v_\text{adam}, s_\text{adam} \leftarrow \text{Adam(diff, p, v_\text{adam}, s_\text{adam, } \eta, \beta_1, \beta_2, \epsilon, t);} \)

**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, where y = Purchased. We evaluate the predicted value \( \hat{y} \) and minimize the loss function (35) to train the QNN model. The full code is given in [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 4 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 \( \epsilon = 10^{-6} \).

Fig. 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 function will reduce and get unchanged after sufficient layers. In our example, the loss function becomes saturated after 5 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, e) for a zoom in), the Qsun loss is converged faster than the Pennylane one.

We next focus on our QNN model with fixed 5 layers. In Fig. 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. Fig. 7(b, c) represent results for the training and testing sets. In the figure, both the true value \( y \) and the predicted value \( \hat{y} \) are labeled as ‘0’ and ‘1’ for no – purchase and purchase, respectively. \( \hat{y} \hat{y} = ‘00’ \) or ‘11’ indicates a correct prediction, whereas \( \hat{y} \hat{y} = ‘01’ \) or ‘10’ indicates a wrong prediction. It can be seen from Fig. 7 that our QNN model has a good performance, reflected by large values of main diagonal elements \( \hat{y} \hat{y} = ‘00’ \) 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.

**V. CONCLUSION**

We have developed Qsun, an open-source platform of quantum virtual machine, 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 quantum differentiable programming is implemented as a built-in function of Qsun, allowing us to execute quantum machine learning 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 quantum machine learning problems, as a generic quantum virtual machine, it can be used for mult-
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.

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.

multiple other purposes, such as developing and testing variational quantum algorithms 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].

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number 20F20021.

[1] F. Arute et al. Quantum supremacy using a programmable superconducting processor. Nature, 574(7779):505–510, Oct 2019.
[2] Zhong et al. Quantum computational advantage using photons. Science, 370(6523):1460–1463, 2020.
[3] John Preskill. Quantum Computing in the NISQ era and beyond. Quantum, 2:79, August 2018.
[4] N. Khammassi, I. Ashraf, X. Fu, C.G. Almudever, and K. Bertels. Qx: A high-performance quantum computer simulation platform. In Design, Automation Test in Europe Conference Exhibition (DATE), 2017, pages 464–469, 2017.
[5] Jianwei Wang, Stefano Paesani, Yunhong Ding, Raffaele Santagati, Paul Skrzypczyk, Alexia Salavrakos, Jordi Tura, Remigiusz Augusiak, Laura Mančinska, Davide Bacco, Damien Boneau, Joshua W. Silverstone, Qihuang Gong, Antonio Acín, Karsten Rottwitt, Leif K. Oxenløwe, Jeremy L. O’Brien, Anthony Laing, and
Mark G. Thompson. Multidimensional quantum entanglement with large-scale integrated optics. Science, 360(6386):285–291, 2018.

[6] Abhinav Kandala, Kristan Temme, Antonio D. Córcoles, Antonio Mezzacapo, Jerry M. Chow, and Jay M. Gambetta. Error mitigation extends the computational reach of a noisy quantum processor. Nature, 567(7749):491–495, Mar 2019.

[7] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. Variational quantum algorithms. Nature Reviews Physics, Aug 2021.

[8] Michael Lubasch, Jaewoo Joo, Pierre Moinier, Martin Kiffner, and Dieter Jaksch. Variational quantum algorithms for nonlinear problems. Phys. Rev. A, 101:010301, Jan 2020.

[9] Jacob Biamonte. Universal variational quantum computation. Phys. Rev. A, 103:L030401, Mar 2021.

[10] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O’Brien. A variational eigenvalue solver on a photonic quantum processor. Nature Communications, 5(1):4213, Jul 2014.

[11] Ken M. Nakamishi, Kosuke Mitarai, and Keisuke Fujii. Subspace-search variational quantum eigensolver for excited states. Phys. Rev. Research, 1:033062, Oct 2019.

[12] William M. Kirby, Andrew Tranter, and Peter J. Love. Contextual Subspace Variational Quantum Eigensolver. Quantum, 5:456, May 2021.

[13] Ilya G. Ryabinkin, Scott N. Genin, and Artur F. Izmaylov. Constrained variational quantum eigensolver: Quantum computer search engine in the fock space. Journal of Chemical Theory and Computation, 15(1):249–255, Jan 2019.

[14] Kazuhiro Seki, Tomonori Shirakawa, and Seiji Yunoki. Symmetry-adapted variational quantum eigensolver. Phys. Rev. A, 101:052340, May 2020.

[15] Bryan T. Gard, Linghua Zhu, George S. Barron, Nicholas J. Mayhall, Sophia E. Economou, and Edwin Barnes. Efficient symmetry-preserving state preparation circuits for the variational quantum eigensolver algorithm. npj Quantum Information, 6(1):10, Jan 2020.

[16] Nikolay V. Tkachenko, James Sud, Yu Zhang, Sergei Tikhonenko, Petr M. Anisimov, Andrew T. Arrasmith, Patrick J. Coles, Lukasz Cincio, and Pavel A. Dub. Correlation-informed permutation of qubits for reducing ansatz depth in the variational quantum eigensolver. PRX Quantum, 2:020337, Jun 2021.

[17] Ying Li and Simon C. Benjamin. Efficient variational quantum simulator incorporating active error minimization. Phys. Rev. X, 7:021050, Jun 2017.

[18] Sam McArdle, Tyson Jones, Suguru Endo, Ying Li, Simon C. Benjamin, and Xiao Yuan. Variational ansatz-based quantum simulation of imaginary time evolution. npj Quantum Information, 5(1):10, Jan 2020.

[19] Hiroyoshi Nishi, Taichi Kosugi, and Yu-chihiro Matsushita. Implementation of quantum imaginary-time evolution method on nisq devices by introducing nonlocal approximation. npj Quantum Information, 7(1):85, Jun 2021.

[20] Yong-Xin Yao, Niladri Gomes, Feng Zhang, Cai-Zhuang Wang, Kai-Ming Ho, Thomas Iadecola, and Peter P. Orth. Adaptive variational quantum dynamics simulations. PRX Quantum, 2:030307, Jul 2021.

[21] Zi-Jian Zhang, Jinhao Sun, Xiao Yuan, and Man-Hong Yung. Low-depth hamiltonian simulation by adaptive product formula, 2020.

[22] Chee Kong Lee, Pranay Patil, Shengyu Zhang, and Chang Yu Hsieh. Neural-network variational quantum algorithm for simulating many-body dynamics. Phys. Rev. Research, 3:023095, May 2021.

[23] Kenji Kubo, Yuya O. Nakagawa, Suguru Endo, and Shotaro Nagayama. Variational quantum simulations of stochastic differential equations. Phys. Rev. A, 103:052425, May 2021.

[24] Hsin-Yuan Huang, Kishor Bharti, and Patrick Rebentrost. Near-term quantum algorithms for linear systems of equations, 2019.

[25] Yi ğit Subaş ı, Rolando D. Somma, and Davide Orsucci. Quantum algorithms for systems of linear equations inspired by adiabatic quantum computing. Phys. Rev. Lett., 122:060504, Feb 2019.

[26] Oleksandr Kyriienko, Annie E. Paine, and Vincent E. Elfving. Solving nonlinear differential equations with differentiable quantum circuits. Phys. Rev. A, 103:052416, May 2021.

[27] Carlos Bravo-Prieto, Ryan LaRose, M. Cerezo, Yigit Subasi, Lukasz Cincio, and Patrick J. Coles. Variational quantum linear solver, 2020.

[28] Xiaosi Xu, Jinhao Sun, Suguru Endo, Ying Li, Simon C. Benjamin, and Xiao Yuan. Variational algorithms for linear algebra, 2019.

[29] Rolando D. Somma and Yi ğit Subaş ı. Complexity of quantum state verification in the quantum linear systems problem. PRX Quantum, 2:010315, Jan 2021.

[30] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum principal component analysis. Nature Physics, 10(9):631–633, Sep 2014.

[31] Ryan LaRose, Arkin Tikku, Étude O’Neel-Judy, Lukasz Cincio, and Patrick J. Coles. Variational quantum state diagonalization. npj Quantum Information, 5(1):57, Jun 2019.

[32] K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii. Quantum circuit learning. Phys. Rev. A, 98:032309, Sep 2018.

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

[34] Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. Phys. Rev. Lett., 122:040504, Feb 2019.

[35] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. Nature, 567(7747):209–212, Mar 2019.

[36] Maria Schuld, Alex Bocharov, Krysta M. Svore, and Nathan Wiebe. Circuit-centric quantum classifiers. Phys. Rev. A, 101:032308, Mar 2020.

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

[38] Arthur Pesah, M. Cerezo, Sansson Wang, Tyler Volkoff, Andrew T. Sornborger, and Patrick J. Coles. Absence of barren plateaus in quantum convolutional neural networks, 2020.

[39] Kaining Zhang, Min-Hsiu Hsieh, Liu Liu, and Dacheng Tao. Toward trainability of quantum neural networks, 2020.
[40] Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J. Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. Training deep quantum neural networks. Nature Communications, 11(1):808, Feb 2020.

[41] Andrew Arrasmith, Lukasz Cincio, Andrew T. Sornborger, Wojciech H. Zurek, and Patrick J. Coles. Variational consistent histories as a hybrid algorithm for quantum foundations. Nature Communications, 10(1):3438, Jul 2019.

[42] Mark Wilde. Quantum information theory. Cambridge University Press, 2017.

[43] Carlos Bravo-Prieto, Diego García-Martín, and José I. Latorre. Quantum singular value decomposer. Phys. Rev. A, 101:062310, Jun 2020.

[44] Bálint Koczor, Suguru Endo, Tyson Jones, Yuichiro Matsuzaki, and Simon C Benjamin. Variational-state quantum metrology. New Journal of Physics, 22(8):083038, aug 2020.

[45] Raphael Kaubruegger, Pietro Silvi, Christian Kokail, Rick van Bijnen, Ana Maria Rey, Jun Ye, Adam M. Kaufman, and Peter Zoller. Variational spin-squeezing algorithms on programmable quantum sensors. Phys. Rev. Lett., 123:260505, Dec 2019.

[46] Jacob L. Beckey, M. Cerezo, Akira Sone, and Patrick J. Coles. Variational quantum algorithm for estimating the quantum fisher information, 2020.

[47] Johannes Jakob Meyer, Johannes Borregaard, and Jens Eisert. A variational toolbox for quantum multi-parameter estimation. npj Quantum Information, 7(1):89, Jun 2021.

[48] Nelder A. John and R. Mead. A simplex method for function minimization. Computer Journal, 7(4):308–313, 1965.

[49] Shaoqeng Zhu, Shih-Han Hung, Shouvanik Chakrabarti, and Xiaodi Wu. On the principles of differentiable quantum programming languages. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 272–285, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Frank Schäfer, Michal Kloc, Christoph Bruder, and Niels Lörch. A differentiable programming method for quantum control. Machine Learning: Science and Technology, 1(3):035009, aug 2020.

[51] James Stokes, Josh Izaac, Nathan Killoran, and Giuseppe Carleo. Quantum Natural Gradient. Quantum, 4:269, May 2020.

[52] Anna Lopatnikova and Minh-Ngoc Tran. Quantum natural gradient for variational bayes, 2021.

[53] Naoki Yamamoto. On the natural gradient for variational quantum eigensolver, 2019.

[54] Aram W. Harrow and John C. Napp. Low-depth gradient measurements can improve convergence in variational hybrid quantum-classical algorithms. Phys. Rev. Lett., 126:140502, Apr 2021.

[55] Gian Giacomo Guerreschi and Mikhail Smelyanskiy. Practical optimization for hybrid quantum-classical algorithms, 2017.

[56] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. PennyLane: Automatic differentiation of hybrid quantum-classical computations, 2020.

[57] Jun Li, Xiaodong Yang, Xinhua Peng, and Chang-Pu Sun. Hybrid quantum-classical approach to quantum optimal control. Phys. Rev. Lett., 118:150503, Apr 2017.

[58] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. Phys. Rev. A, 99:032331, Mar 2019.

[59] Leonardo Banchi and Gavin E. Crooks. Measuring Analytic Gradients of General Quantum Evolution with the Stochastic Parameter Shift Rule. Quantum, 5:386, Jan 2021.

[60] Qiskit: Open-source quantum development.

[61] Cirq Developers. Cirq. August 2021. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

[62] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A practical quantum instruction set architecture, 2017.

[63] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In Proceedings of the Real World Domain Specific Languages Workshop 2018, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Riefel, Rupak Biswas, and Salvatore Mandrà. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. npj Quantum Information, 5(1):86, Oct 2019.

[65] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning, 2019.

[66] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, Yuria Hiraga, Masahiro Nakadai, Jiabao Chen, Ken M. Nakanishi, Kosuke Mitarai, Ryosuke Imai, Shiro Tamiya, Takahiro Yamamoto, Tennen Yan, Toru Kawakubo, Yuya O. Nakagawa, Yohei Ibe, Youyuan Zhang, Hirotsugu Yamashita, Hikaru Yoshimura, Akihiro Hayashi, and Keisuke Fujii. Qulacs: a fast and versatile quantum circuit simulator for research purpose, 2020.

[67] Gian Giacomo Guerreschi, Justin Hogaboam, Fabio Baruffa, and Nicolas P D Sawaya. Intel quantum simulator: a cloud-ready high-performance simulator of quantum circuits. Quantum Science and Technology, 5(3):034007, may 2020.

[68] Damian S. Steiger, Thomas Hänner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. Quantum, 2:49, January 2018.

[69] Adam Kelly. Simulating quantum computers using opencl, 2018.

[70] Tyson Jones, Anna Brown, Ian Bush, and Simon C. Benjamin. Quest and high performance simulation of quantum computers. Scientific Reports, 9(1):10736, Jul 2019.

[71] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. Computer Physics Communications, 176(2):121–136, 2007.

[72] Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Willsch, Naoki Yoshioka, NobuyaSu Ito, Shengjun Yuan,
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 contains an H, a √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 Fig. 2 in the main text.

```python
# Appendix A: Code for benchmarking
import module for visualization
import pandas as pd
import seaborn as sns
import time
import matplotlib.pyplot as plt

# import Qiskit module
from qiskit.providers.ibmq import least_busy
# import Pennylane module
from projectq.backends import Simulator
import projectq.ops as ops
import projectq as p
from projectq import MainEngine
# import Pennylane module
import pennylane as qml

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

# PennyLane circuit
@qml.qnode(dev)
def qml_circuit(n_qubit, depth):
circuit = qml.qnodes['default.qubit'](wires=qubit_max)
for m in range(depth):
    for i in range(n_qubit):
        if i in range(depth):
            qml.Hadamard(wires=i)
        qml.SX(wires=i)
    for i in range(n_qubit):
        qml.CNOT(wires=[i, 0])
return qml.probs(wires=range(n_qubit))
```

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 contains an H, a √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 Fig. 2 in the main text.
# 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=[])  # ProjectQ engine
    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 parameters
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))
sns.lineplot(data=df)
plt.xlabel('Number of Qubits with depth = ' + str(depth))
plt.ylabel('Time (s)')
plt.grid()
plt.savefig('compare.png')

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

Appendix B: Code for QDP 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 Fig. 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. Implementing the QDP algorithm on Qsun and measuring the time taken to update parameters.
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))

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

Appendix C: QDP 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.

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

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

print('Final parameter:', params)

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

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

Appendix D: Quantum Linear Regression Appendix

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

import numpy as np
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
diff[i] = (cost(coef_params_1) - cost(coef_params_2))/(2*np.sin(shift))
    for i in range(len(params)):
        params[i] = params[i] - eta*diff[i]
    return params

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

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
diff[i] = x*(y-predict(x, coef_params_1, intercept_params, boundary))
    for i in range(len(coef_params)):
        coef_params_1 = coef_params.copy()
intercept_params_2 = intercept_params.copy()
intercept_params_1[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:]
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)
                    diff[i][j][k] += 1
            else:
                diff[i][j][k] += np.zeros_like(params[i][j][k])
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/rakeshrao/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")

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