| Frontiers in Applied Mathematics and Statistics

TYPE Original Research
PUBLISHED 28 October 2022
DOI 10.3389/fams.2022.1036711

Check for updates

OPEN ACCESS

EDITED BY
Mark Parsons,
University of Edinburgh,
United Kingdom

REVIEWED BY
Mahmoud Abdel-Aty,
Sohag University, Egypt
Saravana Prakash
Thirumuruganandham,
Universidad Technologica de
Indoamerica, Ecuador

*CORRESPONDENCE
Stefano Markidis
markidis@kth.se

SPECIALTY SECTION
This article was submitted to
Statistical and Computational Physics,
a section of the journal
Frontiers in Applied Mathematics and
Statistics

RECEIVED 05 September 2022
ACCEPTED 05 October 2022
PUBLISHED 28 October 2022

CITATION
Markidis S (2022) On physics-informed
neural networks for quantum
computers.
Front. Appl. Math. Stat. 8:1036711.
doi: 10.3389/fams.2022.1036711

COPYRIGHT
© 2022 Markidis. This is an
open-access article distributed under
the terms of the Creative Commons
Attribution License (CC BY). The use,
distribution or reproduction in other
forums is permitted, provided the
original author(s) and the copyright
owner(s) are credited and that the
original publication in this journal is
cited, in accordance with accepted
academic practice. No use, distribution
or reproduction is permitted which
does not comply with these terms.

On physics-informed neural networks for quantum computers

Stefano Markidis*

Department of Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden

Physics-Informed Neural Networks (PINN) emerged as a powerful tool for solving scientific computing problems, ranging from the solution of Partial Differential Equations to data assimilation tasks. One of the advantages of using PINN is to leverage the usage of Machine Learning computational frameworks relying on the combined usage of CPUs and co-processors, such as accelerators, to achieve maximum performance. This work investigates the design, implementation, and performance of PINNs, using the Quantum Processing Unit (QPU) co-processor. We design a simple Quantum PINN to solve the one-dimensional Poisson problem using a Continuous Variable (CV) quantum computing framework. We discuss the impact of different optimizers, PINN residual formulation, and quantum neural network depth on the quantum PINN accuracy. We show that the optimizer exploration of the training landscape in the case of quantum PINN is not as effective as in classical PINN, and basic Stochastic Gradient Descent (SGD) optimizers outperform adaptive and high-order optimizers. Finally, we highlight the difference in methods and algorithms between quantum and classical PINNs and outline future research challenges for quantum PINN development.

KEYWORDS

quantum physics-informed neural network, Poisson equation, quantum neural networks, continuous variable quantum computing, heterogeneous QPU CPU computing

1. Introduction

One of the most exciting and lively current research topics in scientific computing is integrating classical scientific methods with Machine Learning (ML) and neural network approaches [1]. The usage of Physics-Informed Neural Networks (PINNs) is a major advancement in this promising research direction. PINNs have been applied in a wide range of traditional scientific computing applications, ranging from Computational Fluid Dynamics (CFD) [2] to solid mechanics [3], and electromagnetics [4], to mention a few PINN usages.

In essence, PINNs are neural networks that allow solving a Partial Differential Equation (PDE) of a specific domain area, such as Navier-Stokes equations for CFD or the Poisson equation in electrostatic problems. To achieve this, PINNs combine and connect two neural networks: a surrogate and a residual network. The first surrogate neural network takes as input the point we want to calculate the PDE at (this point is called *collocation point*) and provides the approximated solution at that point. Using a

training process, the surrogate neural network encodes the Partial Differential Equation (PDE) and associated boundary and initial conditions as weights of biases. The second residual network (not to be confused with the popular ResNet!) takes the input from the surrogate network. It provides the PDE *residual*, which is the error of the approximated solution from the surrogate network. The residual calculation requires the solution of differential operators on a neural network. This calculation is performed using automatic differentiation [5] that allows calculating differential operators at any given point without discretization and meshes (PINNs are gridless methods). The residual network has only a passive role during the training process. While it provides the surrogate network with the loss function, its weights and biases are not updated during the neural network training. A stochastic optimizer, such as the Stochastic Gradient Descent (SGD) [6] and Adam [7], updates the surrogate network weights and biases using the residual obtained from the residual network. Typically, in PINNs, we use a sequence of Adam and Limited-memory Broyden-Fletcher-Goldfarb-Shanno (BFGS) [8] optimizers: the L-BFGS optimizer is used to speed up the training or accelerate the convergence to the solution. PINNs do not require labeled datasets for the training, work in an unsupervised fashion, and necessitate only expressing the PDE in the residual network. For an in-depth description of the PINN, we refer the interested reader to the seminal work on PINNs by Raissi et al. [9] and the following enlightening articles [10–12].

From the computational point of view, one of the strategic advantages of using PINN methods for scientific computing is the possibility of exploiting high-performance frameworks, such as TensorFlow [13] or PyTorch [14], designed specifically for efficiently running ML workloads. Such frameworks rely heavily on Graphics Processor Units (GPUs) and accelerators for performance [15]. The use of GPUs for forward and back propagations led to large performance improvement and a renaissance of the research on neural networks, spurring the development of new deeper neural architectures. In addition to the performance, the domain scientist is not burdened with learning relatively low-level programming approaches, such as OpenCL or CUDA, but simply can pin computation to a device or rely on compiler technologies for accelerator automatic code generation [16]. However, with Dennard's scaling ending in 2005 [17] and Moore's law [18] possibly on its last days, many-core architectures (including GPUs) may not be enough to improve the performance scaling in a post-Moore era [19]. For this reason, researchers and companies are exploring alternative, disruptive computing directions, such as quantum computing, to further scale the performance beyond the limitation of silicon-based hardware. For instance, companies, such as Google and IBM, that invested heavily in the past in silicon-based accelerator technologies for AI workloads, now also investigate the development of quantum hardware and software for supporting ML workloads. Notable examples are the IBM-Q devices [20] and Qisikit [21] for IBM, the Sycamore quantum computer [22], and Quantum TensorFlow [23] for Google.

This work aims to investigate the potential of using an emerging co-processor, the Quantum Processing Unit (QPU), and associated software to deploy PINN on quantum computers. Quantum computing is an emerging technology and computational model for solving applications in various fields, ranging from cryptology [24] to database search [25] to quantum chemistry simulations [26]. Among these applications are traditional scientific computing and the solution of differential equations. These basic solvers are at the backbone of CFD, electromagnetics, and chemistry, among others. Algorithms and methodologies aimed at solving linear systems started with the work by Harrow et al. [27], the development of the Harrow-Hassidim-Lloyd (HHL). They continued with linear solvers based on variational quantum solvers [28], the seminal work on Differentiable Quantum Circuits (DQCs) for the solution of differential linear and non-linear equations [29–33], solvers with quantum kernel methods [34], and linear systems based on quantum walks [35]. On the road to fault-tolerant universal quantum computing systems, Noisy Intermediate-Scale Quantum (NISQ) systems [36], are currently major candidate systems for the design and development of near-term applications of quantum computing. Algorithms for NISQ systems are heterogeneous approaches as they combine code running on CPU (typically an optimizer or variational solver) and QPU (for a cost function evaluation). In this work, we focus on a hybrid variational solver approach [29, 37, 38] that can be deployed on NISQ systems. Quantum PINNs are essentially variational quantum circuits using quantum computers for the evaluation of the optimizer's cost function.

This work focuses on Continuous Variable (CV) quantum computing formulation, an alternative to the popular qubit-based universal quantum computing because CV quantum computing is a more convenient framework for PINN development than qubit-based approaches. CV quantum computing uses physical observables, such as the strength of an electromagnetic field, whose numerical values belong to continuous instead of discrete intervals, like qubit-based quantum computing. In some sense, CV quantum computation is analog in nature, while qubit-based quantum computation is digital. CV quantum computing dates back to 1999, first proposed by Lloyd and Braunstein [39]. Braunstein and Van Loock [40] and Weedbrook et al. [41] provide extensive in-depth reviews of CV quantum computing. While the most popular implementations of quantum computers, e.g., IBM, Rigetti, and Google quantum computers, use superconductor transmon qubits, CV quantum computing is implemented with mainly quantum optics [42] and also ion traps [43]. As the PINN method intends to approximate a continuous function, it is more natural to adopt CV quantum computing than a qubit approach [44]. In this work, we extend the work of Killoran et al. [45], Knudsen and Mendl [44], and Kyriienko

et al. [29] by investigating the performance of quantum PINN exploiting CV quantum neural networks. CV quantum neural networks are better suited than qubit-based quantum computing for performing regression calculations. Because the main PINN usage is for performing regression tasks (instead of classification), CV quantum neural networks are the ideal framework for PINN development.

The paper is organized as follows. We first briefly review CV quantum computing and neural network and present a design of the quantum PINN together with the experimental setup in Section 2. We discuss the impact of different optimizers, differentiation techniques, quantum neural network depth, and batch size on the PINN performance in Section 3. Finally, Section 4 summarizes the results, discusses the limitations of this work, and outlines future opportunities for quantum PINN development.

2. Quantum physics-informed neural networks

In this section, we introduce CV quantum computing, its basic gates, and a simple formulation of the quantum neural network unit, the basic building block for CV quantum neural networks. We then discuss the design of a quantum PINN and describe the experimental setup, comprising the programming setup and quantum computer simulator in use.

2.1. Continuous variable quantum computing and neural networks

The CV quantum computing approach is based on the concept of *qumode*, the basic unit carrying information in CV quantum computing. We express the qumode $|\psi\rangle$, in the basis expansion of quantum states, as

$$|\psi\rangle = \int \psi(x)\,|x\rangle\,dx, \tag{1}$$

where the states are the eigenstates of the $\hat{x}$ quadrature, $\hat{x}\,|x\rangle = x\,|x\rangle$ with x being a real-valued eigenvalue. This is an alternative formulation to the popular qubit-based approach. In this latter case, the qubit $|\phi\rangle$ is expressed as the combination of the states $|0\rangle$ and $|1\rangle$ as

$$|\phi\rangle = \phi_0\,|0\rangle + \phi_1\,|1\rangle\,. \tag{2}$$

While for qubit-based quantum computing, we use a set of discrete coefficients, such as ϕ_0 and ϕ_1, in the case of CV based we have a continuous of coefficients (a continuous eigenvalue spectrum), giving the name of this approach. All the quantum computing settings with continuous quantum operators perfectly match CV quantum computing. The position ($\hat{x}$) and momentum ($\hat{p}$) operators, constituting the so-called phase space, are good examples of continuous quantum operators we use in this work. The position operator is defined as follows:

$$\hat{x} = \int_{-\infty}^{\infty} x\,|x\rangle\,\langle x|\,dx, \tag{3}$$

where the vectors $|x\rangle$ are orthogonal. Similarly, the momentum operator is defined as:

$$\hat{p} = \int_{-\infty}^{\infty} p\,|p\rangle\,\langle p|\,dp, \tag{4}$$

with $|p\rangle$ being orthogonal vectors. A qumode i is associated with a pair of position and momentum operators $(\hat{x}_i, \hat{p}_i)$. These operators do not commute, leading to the Heisenberg uncertainty principle for the simultaneous measurements of $\hat{x}$ and $\hat{p}$.

As in the well-established qubit-based formulation, CV quantum computation can be expressed using low-level gates that can be implemented, for instance, as optical devices. A CV quantum program can be seen as a sequence of gates acting on one or more qumodes. Four basic *Gaussian* gates operating on qumodes are necessary to develop CV quantum neural networks and PINNs. These four gates of linear character are:

- **Displacement** Gate - $D(\alpha)$: $\begin{bmatrix} x \\ p \end{bmatrix} \rightarrow \begin{bmatrix} x + \Re(\alpha) \\ p + \Im(\alpha) \end{bmatrix}$.
  This operator corresponds to a phase space shift by displacing a complex number $\alpha \in \mathbb{C}$.
- **Rotation** Gate - $R(\phi)$: $\begin{bmatrix} x \\ p \end{bmatrix} \rightarrow \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix}$.
  This operator corresponds to a rotation of the phase space by an angle $\phi \in [0, 2\pi]$.
- **Squeezing** Gate - $S(r)$: $\begin{bmatrix} x \\ p \end{bmatrix} \rightarrow \begin{bmatrix} e^{-r} & 0 \\ 0 & e^{r} \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix}$.
  This operation corresponds to a scaling operation in the phase space with a scaling factor $r \in \mathbb{C}$.
- **Beam-splitter** Gate - $BS(\theta)$:
  $\begin{bmatrix} x_1 \\ x_2 \\ p_1 \\ p_2 \end{bmatrix} \rightarrow \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & \cos(\theta) & -\sin(\theta) \\ 0 & 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ p_1 \\ p_2 \end{bmatrix}$.
  This operation is similar to a rotation between two qumodes by an angle $\theta \in [0, 2\pi]$.

An important derived gate is the **interferometer** that can be formulated as a combination of beam-splitter and rotation gates. In the limit of one qumode, the interferometer reduces to a rotation gate. By combining these Gaussian gates, we can define an affine transformation [45] that is instrumental for the expression of neural network computation.

In addition, to these four basic Gaussian gates defined above, non-Gaussian gates, such as the **cubic** and **Kerr** gates,

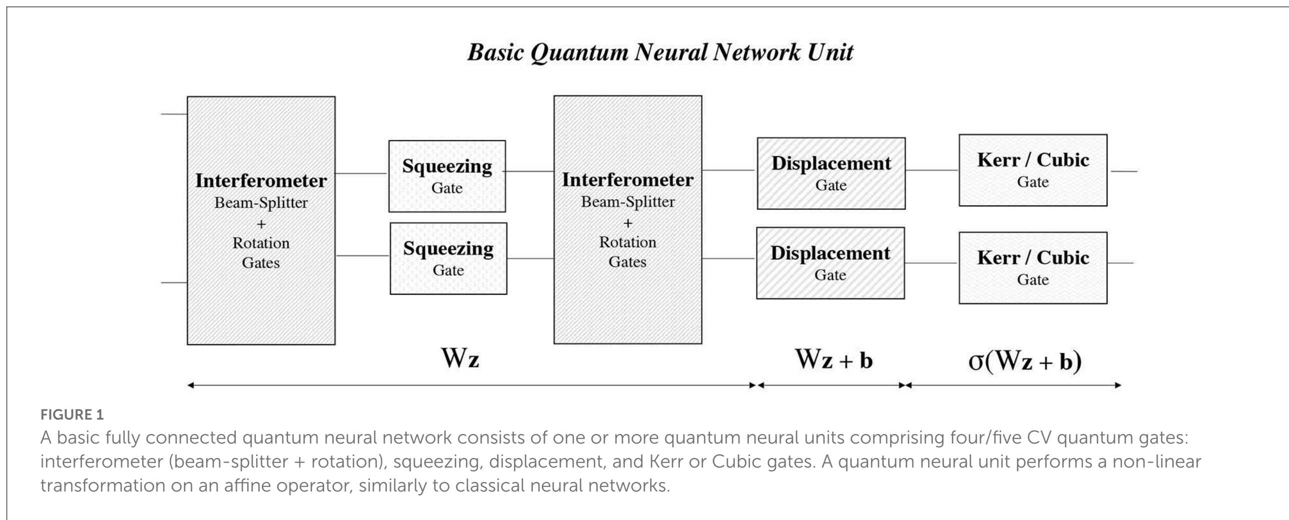

FIGURE 1
A basic fully connected quantum neural network consists of one or more quantum neural units comprising four/five CV quantum gates: interferometer (beam-splitter + rotation), squeezing, displacement, and Kerr or Cubic gates. A quantum neural unit performs a non-linear transformation on an affine operator, similarly to classical neural networks.

provide a non-linearity similar to the non-linearity performed by the activation functions in the classical neural network. Most importantly, the non-Gaussian gates, when added to the Gaussian gates to form a sequence of quantum computing units, provide the universality of the CV quantum circuit: we can guarantee that we can produce any CV state with at most polynomial overhead. In this work, we use the Kerr gate because the CV quantum simulators provide a Kerr gate model that is more accurate than the cubic model. The Kerr gate is often expressed as $K(\kappa)$ and has $\kappa \in \mathbb{R}$ as the quantum gate parameter.

Finally, an operation's result in quantum computing is a measurement operation. In this work, as a result of the measurements, we evaluate the expected value for the quadrature operator $\hat{x}$:

$$\langle \psi_x | \hat{x} | \psi_x \rangle . \tag{5}$$

As mentioned in Section 1, the most promising hardware implementation of CV quantum gates uses photonic technologies. An example of a quantum photonic computer is Xanadu's Borealis quantum computer[1] [46], designed for solving Gaussian Boson Sampling (GBS) problems. In this system, a laser source, generated by an Optical Parametric Oscillator (OPO), creates a train of identical light pulses, each constituting a qumode. These qumodes are then injected into a sequence of dynamically programmable loop-based interferometers: the beam splitters and rotation can be programmed to selectively route qumodes into optical delays lines so they can interfere with later qumodes (this technique is called *time-multiplexing*). Finally, the state of the system is measured on the photon number basis, or *Fock state*, employing an array of Photon-Number Resolving (PNR) detectors based on superconducting transition edge sensors. The PNR detectors require cryogenic cooling. The development of CV quantum computing systems is a very active current research area [47].

Equipped with the concepts of CV quantum gates and their parameters and the expected value of the quadrature operator, we can now develop a quantum neural network by following the seminal work by Killoran et al. [45]. The basic building block of a quantum neural network is the quantum neural network unit (also called a quantum network layer in the literature) which is akin to the classical neural network unit. Figure 1 shows the basic components of a quantum neural unit. The first three components of the quantum neural unit are a succession of a first interferometer, a squeezing gate, and a second interferometer. It is shown in Killoran et al. [45] that the result of these operations is analogous to multiplying the phase space vector by the neural network weights W (parameters of the interferometers and squeezing gates). As in the classical neural networks, a displacement gate mimics the addition of bias $\mathbf{b}$. Finally, a Kerr gate (or a cubic gate) introduces a non-linearity similar to the activation function σ in classical neural networks.

$$|\mathbf{x}\rangle \rightarrow |\sigma(W\mathbf{x} + \mathbf{b})\rangle . \tag{6}$$

We can create a quantum neural network by stacking multiple quantum neural units in a sequence. It is important to note that for one qumode, each gate can be controlled by a total of seven gate parameters (α, ϕ, r, θ, and κ) that can be a real-value (ϕ, θ, and κ) or complex number (α, r). Two real numbers can express the complex-valued parameters in the Cartesian (with the real and imaginary part) or polar (with amplitude and phase) formulations. The quantum circuit parameters are further divided into *passive* and *active* parameters: the beamsplitter angles and all gate phases are passive parameters, while the displacement, squeezing, and Kerr magnitude are active parameters. The training of quantum neural networks aims at finding the parameter values (α, ϕ, r, θ, and κ) for different qumodes and quantum

1 https://www.xanadu.ai/products/borealis/

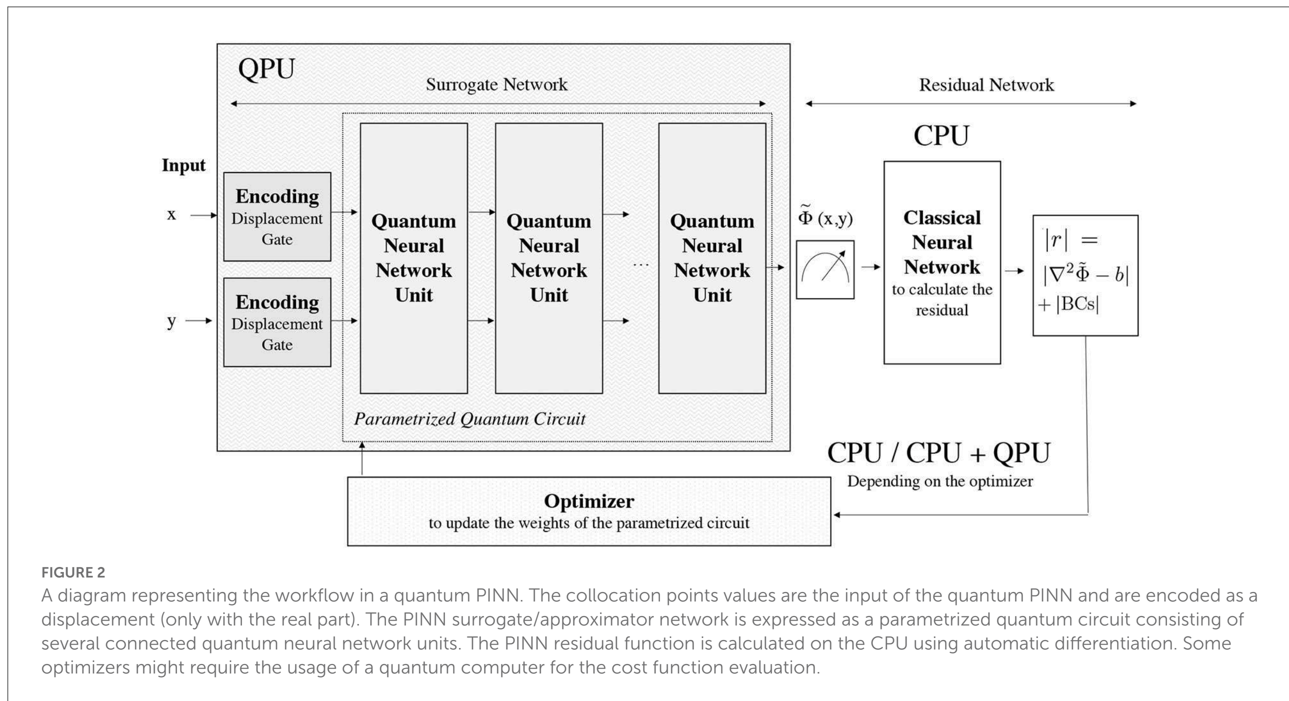

FIGURE 2
A diagram representing the workflow in a quantum PINN. The collocation points values are the input of the quantum PINN and are encoded as a displacement (only with the real part). The PINN surrogate/approximator network is expressed as a parametrized quantum circuit consisting of several connected quantum neural network units. The PINN residual function is calculated on the CPU using automatic differentiation. Some optimizers might require the usage of a quantum computer for the cost function evaluation.

neural units to minimize the PINN cost function, that is, the PDE residual.

2.2. Quantum physics-informed neural networks

Quantum PINNs extend the CV quantum neural networks [29, 44]. Figure 2 shows an overview of the workflow and resource usage of a quantum PINN for solving a 2D Poisson equation and associated boundary conditions, $\nabla^2 \tilde{\Phi}(x, y) = b(x, y)$. The Poisson equation is an omnipresent governing equation in scientific computing: electrostatic and gravitational forces are, for instance, governed by the Poisson equation.

As the first step of the quantum PINN, we encode the collocation point as a real-valued displacement of the vacuum state that is the lowest energy Gaussian state with no displacement or squeezing in phase space. One of the significant advantages of using CV quantum neural networks is the ease of encoding the input (the collocation point) into the quantum neural network without requiring normalization. After the initial displacement encodes the collocation point, the qumode feeds into the quantum PINN. A quantum PINN combines two neural networks working together in succession:

- **Quantum Surrogate Neural Network.** The quantum neural network surrogate is a CV quantum neural network as described in Killoran et al. [45]. The quantum surrogate neural network is a parametrized quantum circuit that takes as input the collocation point coordinates x and y, encoded as a displacement of the vacuum state, and gives the approximated solution $\tilde{\Phi}(x, y)$ as the expected value of the quadrature operator. The solution of the PDE is encoded in the parametrized quantum circuit and accessible by running the surrogate quantum surrogate neural network on the QPU. After the training process, we run the quantum circuit giving a displaced vacuum state to calculate the approximated solution.
- **Residual Neural Network on CPU.** In a matrix-free fashion, quantum PINNs do not require storing and encoding the problem matrix and the source vector. Instead, the residual network encodes the governing equations (in this case, the 2D Poisson equation). The residual network is not trained, e.g., its weights are not updated, and its only function is to provide the quantum surrogate neural network with a loss function that is the residual function for the domain inner points (ip):

$$|r|_{ip} = |\nabla^2 \tilde{\Phi}(x, y) - b(x, y)|. \quad (7)$$

In addition to satisfying the governing equation in the domain inner points, the collocation points on the boundaries must satisfy the problem boundary conditions. For instance, if a problem requires the solution to vanish at the boundaries then a specific residual function for the boundary collocation points (bp) is specified as $|r|_{bp} = |\tilde{\Phi}(x_B, y_B)|$. Traditionally, PINNs use automatic differentiation to calculate the differential operators, such

as the nabla operator in the Poisson equation. To perform the automatic differentiation, we rely on the ideal analytical model of the CV quantum gates, as presented in Section 2.1, and the chain rule. In addition to automatic differentiation, it is possible to perform numerical differentiation to calculate the differential operators. However, this requires introducing a computational grid (with the solutions calculated only on discrete points) and a discretization of the differential operators. The calculation of the residual function occurs only on the CPU and does not involve the QPU.

In the quantum PINN, the approximated solution of the quantum neural network is used to calculate the residual function. The absolute value of the residual function, including the boundary conditions, constitutes the loss function $\mathcal{L}$. During this work, we found that performing the operations in batches, e.g., running in parallel with the quantum PINN with different collocation points and averaging the loss function over the batch, improves the quality of the results in terms of smoothness of the solution and the computation performance of the quantum simulator. For this reason, as cost function, we take the average of cost functions evaluated at the different batch collocation points:

$$\mathcal{L} = \frac{\sum_i^{B_s} |r|_{ip}}{B_s} + \frac{\sum_i^{B_s} |r|_{bp}}{B_s} = \frac{\sum_i^{B_s} |\nabla^2 \tilde{\Phi}(x_i, y_i) - b(x_i, y_i)|}{B_s} + \frac{\sum_i^{B_s} |\tilde{\Phi}(x_{i,B}, y_{i,B})|}{B_s}, \quad (8)$$

where B_s is the batch size, (x_i, y_i) are random inner collocation points and $(x_{i,B}, y_{i,B})$ are boundary collocation points. To run the quantum neural in a batch is equivalent to running it using multiple qumodes. For instance, if the quantum neural network uses only one qumode, e.g., a one-dimensional Poisson equation problem, with a batch size of 32, we can run the full batch using 32 qumodes on the QPU.

The updates of the quantum surrogate neural network parameters are determined by running a stochastic optimizer that relies on the calculation of gradient (and Hessian for second-order optimizers) and often on a learning rate, e.g., a step size toward a minimum of a loss function. Stochastic optimizers are *adaptive* if the learning rate change during the optimizer iterations. Examples of adaptive optimizers are RMSprop, Adam, Adam with Nesterov momentum (Nadam) [48], and Adadelta [49]. If the calculation of the gradient uses automatic differentiation, then the optimizer only runs on the CPU; in the case of numerical differentiation, such as in the case of Simultaneous Perturbation Stochastic Approximation (SPSA) and L-BFGS-B, the optimizers use both CPU and QPU, called for the function evaluation with the quantum surrogate neural network.

2.3. Quantum neural network implementation

For the design and development of the Quantum PINN, we use Xanadu's `Strawberry Fields`[2] CV programming and simulation framework [50, 51]. Different backends to simulate the CV quantum computers are available in Strawberry Field, including the `fock`, `tf`, and `gaussian` backends. The `fock` and `tf` backends express the modes' quantum state with the Fock or particle basis. Arbitrary quantum states are expressed up to a photon cutoff when using these backends. However, this comes with an exponential increase in space complexity as a function of the number of modes with the base of the exponent scaling. This constrains the quantum simulator memory usage that critically depends on the number of qumodes and the cutoff number. On the other hand, the `gaussian` backend does not suffer the problem of exponential scaling in the memory required. However, only a subset of states can be represented.

In this work, we use the TensorFlow `tf` backend [50] that provides the quantum computer simulator and an expressive domain-specific language to formulate CV quantum neural network calculations. The TensorFlow backend represents the quantum state of the modes using the Fock or particle basis, and it is subject to the memory constraints of representing quantum states with the Fock or particle basis. The advantage of using the TensorFlow backend is that the programmer can use all existing Keras and TensorFlow automatic differentiation, optimizers, and tensor operations.

While both the `Strawberry Fields` and `TensorFlow` interfaces will be subject to changes in the future, we show some snippet code to demonstrate the simplicity of developing a quantum PINN with these programming interfaces. The `Strawberry Fields` quantum computer simulator is easily initialized with:

```
eng = sf.Engine(backend="tf", backend_options={"
     cutoff_dim": 125, "batch_size":
     n_collocation_points})
prog = sf.Program(1)
```

Note that we select the `tf` backend, a cutoff for representing state equal to 125 and batch size equal to the number of collocation points we use in the PINN. Our code only uses one qumode set with `sf.Program(1)`.

To express our CV quantum circuit, we use `Blackbird`, which is the Assembly language built into `Strawberry Fields`. The `Strawberry Fields` framework allows us to express the quantum circuit, including the set of gates for the implementation of the quantum neural network, in a straightforward form by providing a sequence of gates acting on a qumode q. The quantum circuit for encoding the collocation point and executing a forward pass with a one-unit network

2 https://strawberryfields.ai/

using one qumode (in this case, the beam-splitter reduces to a rotation) is shown in the Listing of Python code below.

```
#    QPINN Surrogate Circuit: Input + One Quantum Unit
     / Layer
with prog.context as q:
    #  Initially the qumode q is in the vacuum state
    Dgate(x) | q    #  Encode the collocation point x
     with a displacement on q
    # One quantum neural unit to express the surrogate
      network
    Rgate(r1) | q  # Beam-splitter 1: beam-splitter
     reduces to rotation with one qumode
    Sgate(sq_r1, sq_phi1) | q # Squeezeer
    Rgate(r2) | q # Beam-splitter 2: beam-splitter
     reduces to rotation with one qumode
    Dgate(alpha1, phi1) | q   # Displacement (similar
     to adding bias in classical neural network)
    Kgate(kappa1) | q # Kerr gate: non-linear
     transformation (similar to activation function in
     NN)
```

Note that in a simple setup of one quantum neural unit and one qumode we have seven neural network weights (circuit parameters): `r1` for the first rotation, `sqr_r1` and `sqr_phi1` for the amplitude and phase of the squeezer, `alpha1` and `sqr1` for the real and the imaginary parts of the displacement, and finally `k1` for the Kerr gate. In the case of a network with one quantum neural unit and one qumode, the seven network weights (quantum circuit parameters) are optimized to minimize the PINN residual function. With one qumode, a network with one, two, three, and four quantum neural units will require to optimize seven, 14, 21, and 28 weights, respectively.

The snippet of Python code below shows the implementation of executing the quantum circuit, using the `run()` method, extracting the resulting state, and obtaining the expected value of the quadrature operator with `quad_expectation()` method. To calculate the second-order derivative of the Poisson equation for the residual function, we first identify the region of code defining the operations that are differentiated using the TensorFlow 2 *gradient taping* and then perform the differentiation with the `gradient()` method.

```
def QPINNmodel(...):
    if eng.run_progs:
       eng.reset()
    x_in = tf.Variable(np.random.uniform(low=0.0, high
     =Lx, size=n_collocation_points)) # Random points
    with tf.GradientTape() as tape2:
         with tf.GradientTape() as tape1:
              result = eng.run(prog, args={"x": x_in,
     "alpha1": alpha1_in, "phi1": phi1_in,
              "r1": r1_in, "sq_r1": sq_r1_in, "sq_phi1
     ": sq_phi1_in, "r2": r2_in, "kappa1": kappa1_in})
              state = result.state
              mean, var = state.quad_expectation(0)
     # calculate the second order derivative with
     automatic differentiation
         dudx = tape1.gradient(mean, x_in)
     du2dx2 = tape2.gradient(dudx, x_in)
     b =  ... # define the known term
     res = du2dx2 - b      # define the residual
     lossIP = tf.reduce_mean(tf.abs(res)) # take the
     mean over the batch (IP = Inner points)
     # now calculate the residual for the collocation
     points on the boundaries
     ...
     loss = lossIP + lossBC1 + lossBC2
     return loss
...
# calculate one optimization step
with tf.GradientTape() as tape:
  loss = QPINNmodel(...)
opt = tf.keras.optimizers.SGD(learning_rate=rate, name
     ="SGD")  # pick the optimizer
gradients = tape.gradient(loss, [alpha1_in, phi1_in,
     r1_in, sq_r1_in, sq_phi1_in,r2_in, kappa1_in])
opt.apply_gradients(zip(gradients, [alpha1_in,phi1_in,
     r1_in, sq_r1_in, sq_phi1_in, r2_in,kappa1_in ]))
...
```

We can calculate the loss function using the result from the differentiation and the value of the known term at the collocation point. Note that the operation is performed in parallel on all the collocation points, so we take an average of the absolute value of the residual with `tf.reduce_mean(tf.abs())`. To update the quantum network parameters, we calculate the first-order derivative of the loss function with respect to the quantum circuit parameters and use these values to update them with `apply_gradients()` operation. In this particular case, we use the SGD Keras optimizer.

2.4. Experimental setup and accuracy metrics

In this study, we rely on the Python Strawberry Fields CV quantum simulator and a series of Python modules to enable efficient vector calculations (on the CPU) and additional optimizers not included in the TensorFlow/Keras framework. We use Python 3.10.4, NumPy (1.22.4), and SciPy (1.8.1) modules. We perform the experiments using the quantum computer simulator provided by Strawberryfields framework, version 0.22.0. In all the simulations, we use one quantum mode and a cutoff dimension of 125 for the Fock basis. We check each measurement's state vector's norm to verify that quantum computer simulation is accurate. Neural networks and quantum computer simulators inherently comprise a level of stochasticity. For this, we set random number generator seeds for TensorFlow and NumPy.

As part of this study, we evaluate the usage of several optimizers: RMSprop, Adam, Adam with Nesterov momentum (Nadam) [48], Adadelta [49], Simultaneous Perturbation Stochastic Approximation (SPSA) [52], and the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS-B) [8] optimizers. We use the TensorFlow 2.9.0 SGD, RMSprop, Adam, Nadam, and Adadelta implementations and automatic differentiation capabilities. For the SPSA optimizer, we use its implementation, available at https://github.com/SimpleArt/spsa that provides an adaptive learning rate. Finally, we use

SciPy 1.8.1 L-BFGS-B optimizer in combination with the TensorFlow SGD.

For the sake of simplicity, to evaluate the quantum PINN, we use a simple one-dimensional Poisson's problem with Dirichlet boundary conditions fixed to zero: $d^2\Phi/dx^2 = b(x)$. For testing purposes, we choose kinds of sources (the b term in the Poisson equation) and two domain sizes:

1. **Quadratic**: $b(x) = x(x-1)$, $[\,0, 1\,]$, $\Phi(0) = 0, \Phi(1) = 0$. The solution, in this case, is a parabola with the first derivative equal to zero at the center of the domain, $x = 0.5$.
2. **Sinusoidal**: $b(x) = \sin(2x)$, $[\,0, 2\pi\,]$, $\Phi(0) = 0, \Phi(2\pi) = 0$. This is a more challenging test case as the solution has four points where the first derivative zeros.

An extension of quantum PINN to solve a two-dimensional would require to use of two qumodes to initially encode the x and y coordinates of the collocation points and having interferometers (instead of the simple rotation gate in the case of one qumode) in the quantum neural network to entangle the two qumodes. In the two-dimensional, the residual network encodes the Laplacian operator instead of the one-dimensional derivative.

The *baseline* quantum neural network hyper-parameters are the following: we set the learning rate equal to 0.01 and 0.0001 for the quadratic and sinusoidal source terms cases, respectively. Optimizers' performance highly depends on the initial learning rate. We completed a grid search for setting the learning rate for SGD as it used a fixed learning rate during the training and was more susceptible to the exploding and vanishing gradient problem. We perform 500 optimizer iterations. The collocation points are drawn randomly within the PDE domain at each iteration. The number of collocation points per is equal to the batch size. As baseline runs, we choose 32 for the batch size. For the boundary conditions, we also evaluate the cost functions at the boundaries with a number of collocation points equal to batch size, e.g., 32 for the baseline cases. The quantum circuit parameters are initialized with a normal distribution centered at zero and a standard deviation of 0.05. The simulation of a QPINN implemented with Strawberry Fields takes approximately 20 min on a modern laptop.

To characterize the accuracy of the quantum PINN, we check the loss function values (the absolute value of the residual) and the final error norm after the training. To evaluate the final error, we compare the analytical solution and quantum PINN

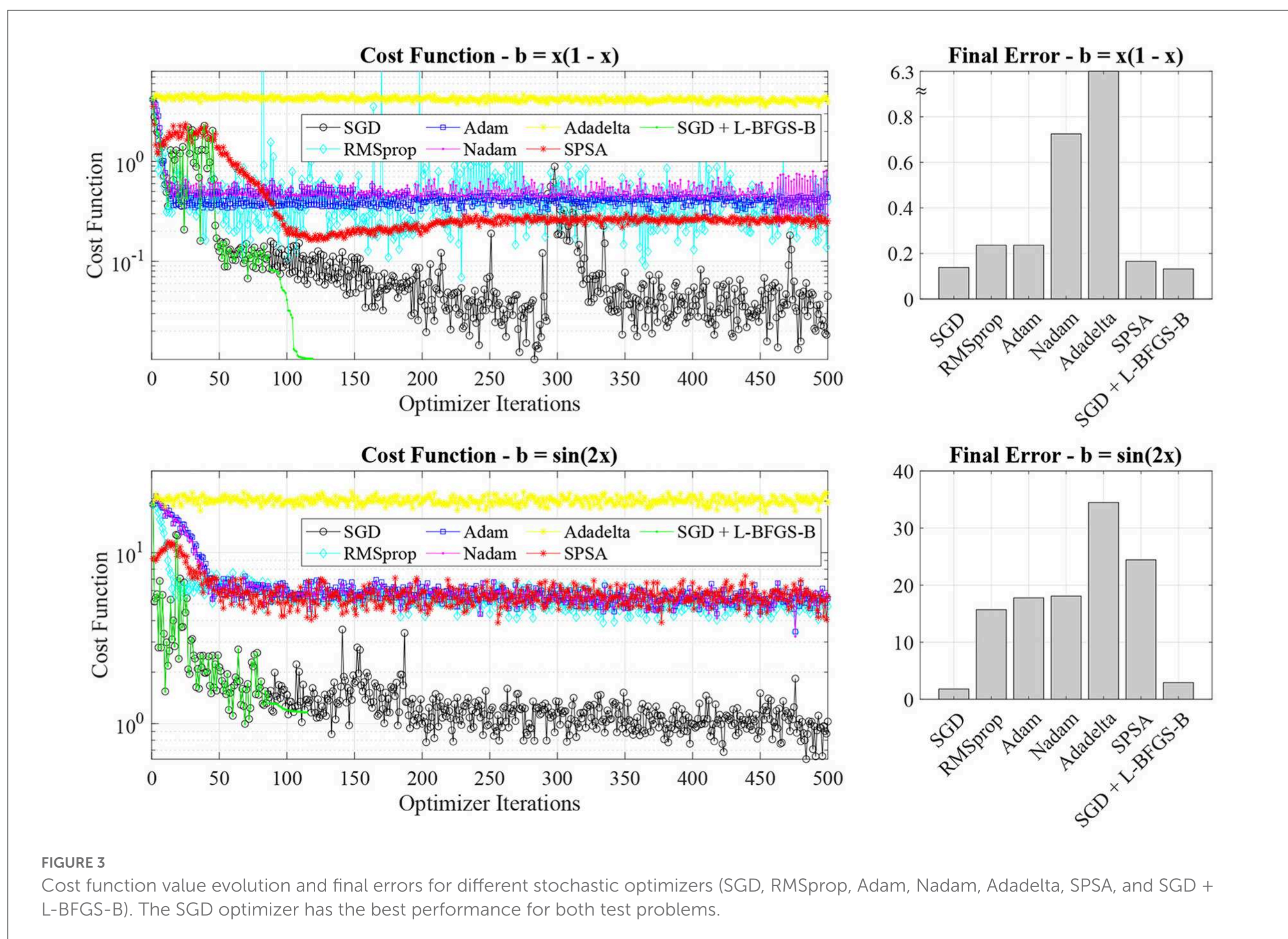

FIGURE 3
Cost function value evolution and final errors for different stochastic optimizers (SGD, RMSprop, Adam, Nadam, Adadelta, SPSA, and SGD + L-BFGS-B). The SGD optimizer has the best performance for both test problems.

results using a uniform grid of 32 points (including the boundary points) and take the Euclidean norm of the PINN approximated solution minus the analytical solution on the 32 points.

3. Results

As the first step in our experiments, we investigate the accuracy of different stochastic optimizers for the two one-dimensional Poisson problems with the quadratic and sinusoidal sources. Figure 3 shows the cost function value for the two test problems on the left panels, while the right panels show the final error after the training.

By analyzing the cost function value and final error, it is clear that the SGD optimizer outperforms the adaptive (RMSprop, Adam, Nadam, and Adadelta) and SPSA optimizers. In general, we find that adaptive optimizers, such as Adam, tend to converge to local minima in the training landscape without exiting. In the case of the Adadelta optimizer, we do not observe convergence to the solution. On the other hand, a noisier optimizer, such as SGD, can escape the local minima and better explore the optimization landscape. For instance, by analyzing the cost function value for the SGD optimizer in the parabolic source case (black line in the top left panel of Figure 3), we note that the optimizer escapes a local minimum approximately after 300 iterations. The SPSA optimizer also allows us to provide additional noise to hop between different convergence basins, possibly achieving a similar behavior of SGD. However, we find that SPSA does not perform better than SGD. While all the optimizers perform relatively well with the parabolic case (they all capture the parabolic nature of the solution), the adaptive optimizers fail to recover the sinusoidal nature of the solution in the second test case (not shown here) and leading to significant final errors (see the bottom right panel in Figure 3).

In classical PINN, a second-order optimizer, L-BFGS-B, is used after the Adam optimizer to speed up the PINN convergence, thus requiring considerably fewer iterations [1, 9]. In classical PINN, L-BFGS-B is not used from the start of the training, as it would quickly converge to a local minimum of the training landscape without escaping it. As in the classical case, we deploy an L-BFGS-B optimizer after 80 optimizer iterations. While L-BFGS-B can reduce the cost function evaluation in both cases, we note that the final error is approximately the same for plain SGD optimizers and SGD combined with L-BFGS-B.

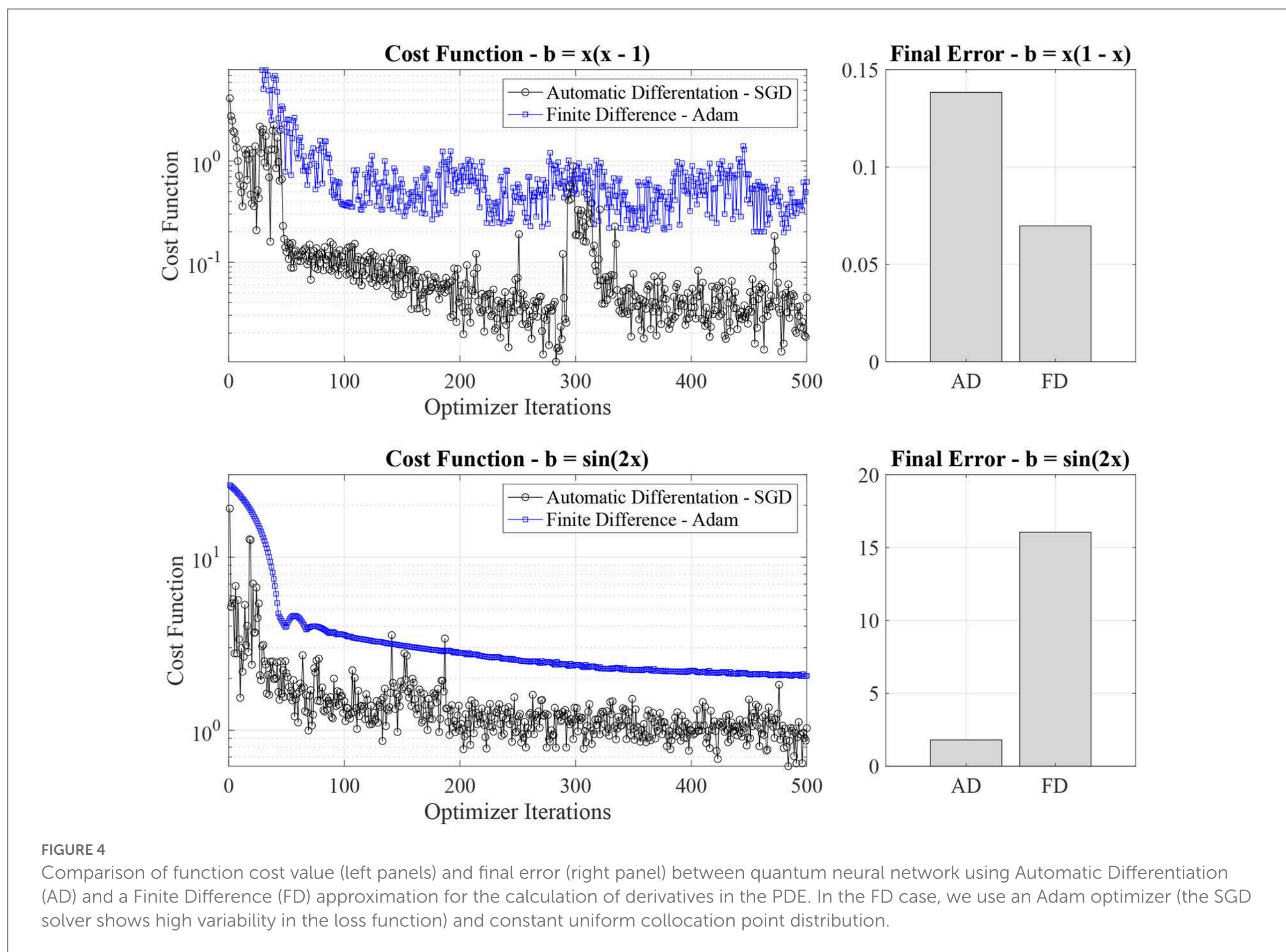

FIGURE 4
Comparison of function cost value (left panels) and final error (right panel) between quantum neural network using Automatic Differentiation (AD) and a Finite Difference (FD) approximation for the calculation of derivatives in the PDE. In the FD case, we use an Adam optimizer (the SGD solver shows high variability in the loss function) and constant uniform collocation point distribution.

As additional tests, we also implemented a multi-step SGD and L-BFGS-B, e.g., a succession of SGD and L-BFGS-B, without achieving a final performance improvement. In classical fully-connected PINNs, Adam and L-BFGS-B optimizers are widely employed and successful for PINNs [1, 9], while in quantum PINN SGD provides better performance than Adam and L-BFGS-B. In quantum PINN, the optimization landscape is more diverse, and its optimizer exploration is more challenging.

One of the main advantages of using PINNs is to leverage Automatic Differentiation (AD) for approximating with high accuracy a derivative at an arbitrary point in the domain. This is a powerful and flexible mechanism for evaluating the residual function in PINN. However, we note that it is possible to calculate the residual function using a fixed number of collocation points, e.g., the nodes of a uniform grid, and approximate the derivative using a Finite Difference (FD) approximation. A similar approach is used in classical fractional PINN [53]. This method comes with the disadvantage of using a fixed grid point, hurting the generalization of the solution to different collocation points and expressing the calculations on finite difference stencils.

We compare the cost function value evolution and final error for PINN residual function, calculated with AD and FD in Figure 4. When using the SGD optimizer and an FD formulation for the residual function, we observed high variability of the cost function values and the occurrence of two-three characteristic cost function values, signaling that the quantum PINN is hopping quickly between a small number of local minima. Using an Adam optimizer, we have a smoother cost function value evolution behavior. For this reason, for the FD case, we switch to an Adam optimizer. From analyzing Figure 4, we see that AD provides a lower final error than the quantum PINN using AD for the residual function in the case of the quadratic source case, despite the higher loss function values. However, FD fails (significant final error) in capturing the sinusoidal nature of the solution in the second test case. Because of the use of fixed collocation points, the quantum PINN with FD cannot escape a local minimum leading to a large final error.

As an additional investigation, we study the impact of the quantum neural network depth (also the depth of the quantum circuit) on the calculations by varying the number of quantum neural network units in the surrogate network. The results of these experiments in terms of the cost function values and final error are presented in Figure 5.

Overall, we find that quantum PINNs with four and eight quantum units have a comparable final error. In general,

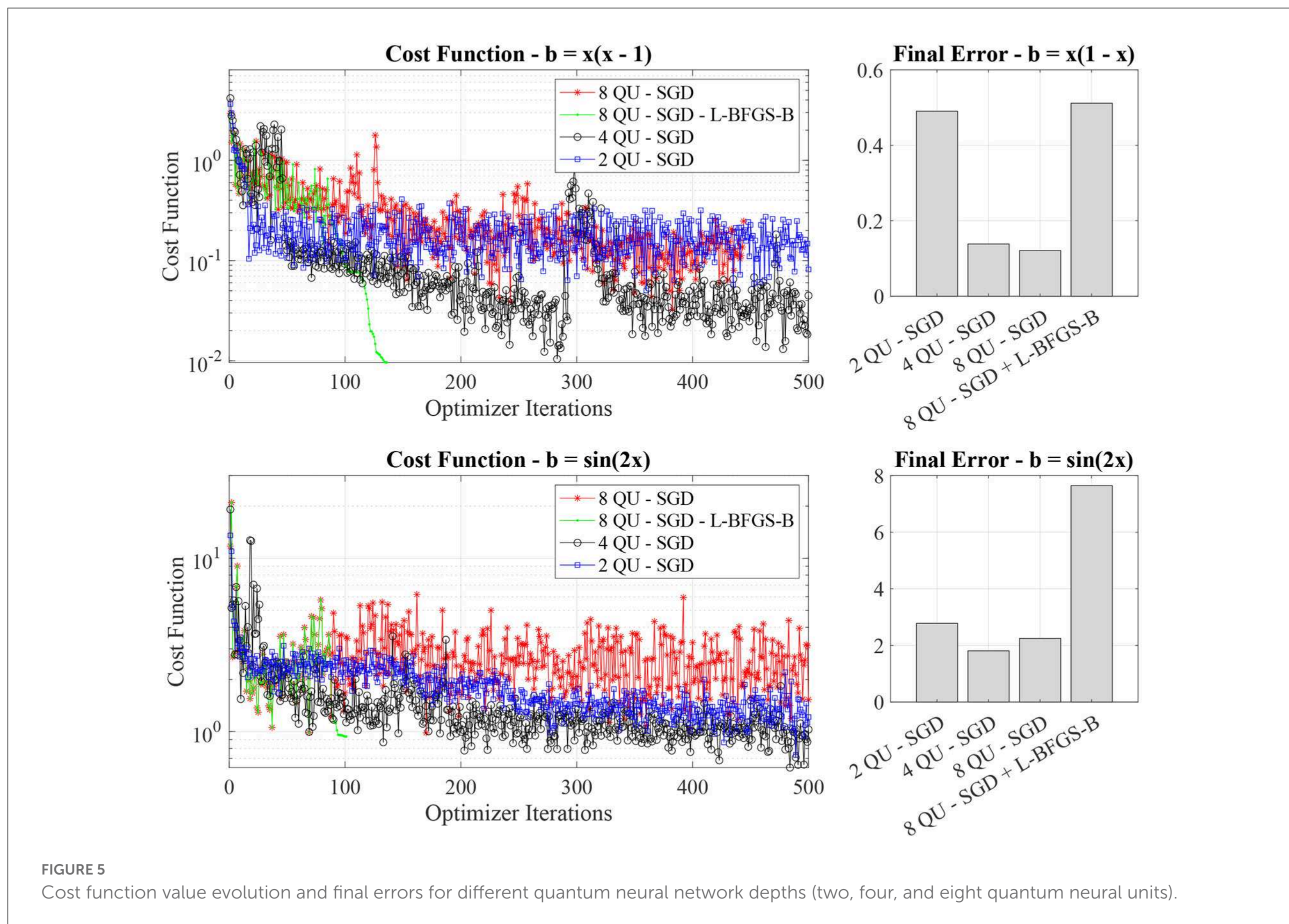

FIGURE 5
Cost function value evolution and final errors for different quantum neural network depths (two, four, and eight quantum neural units).

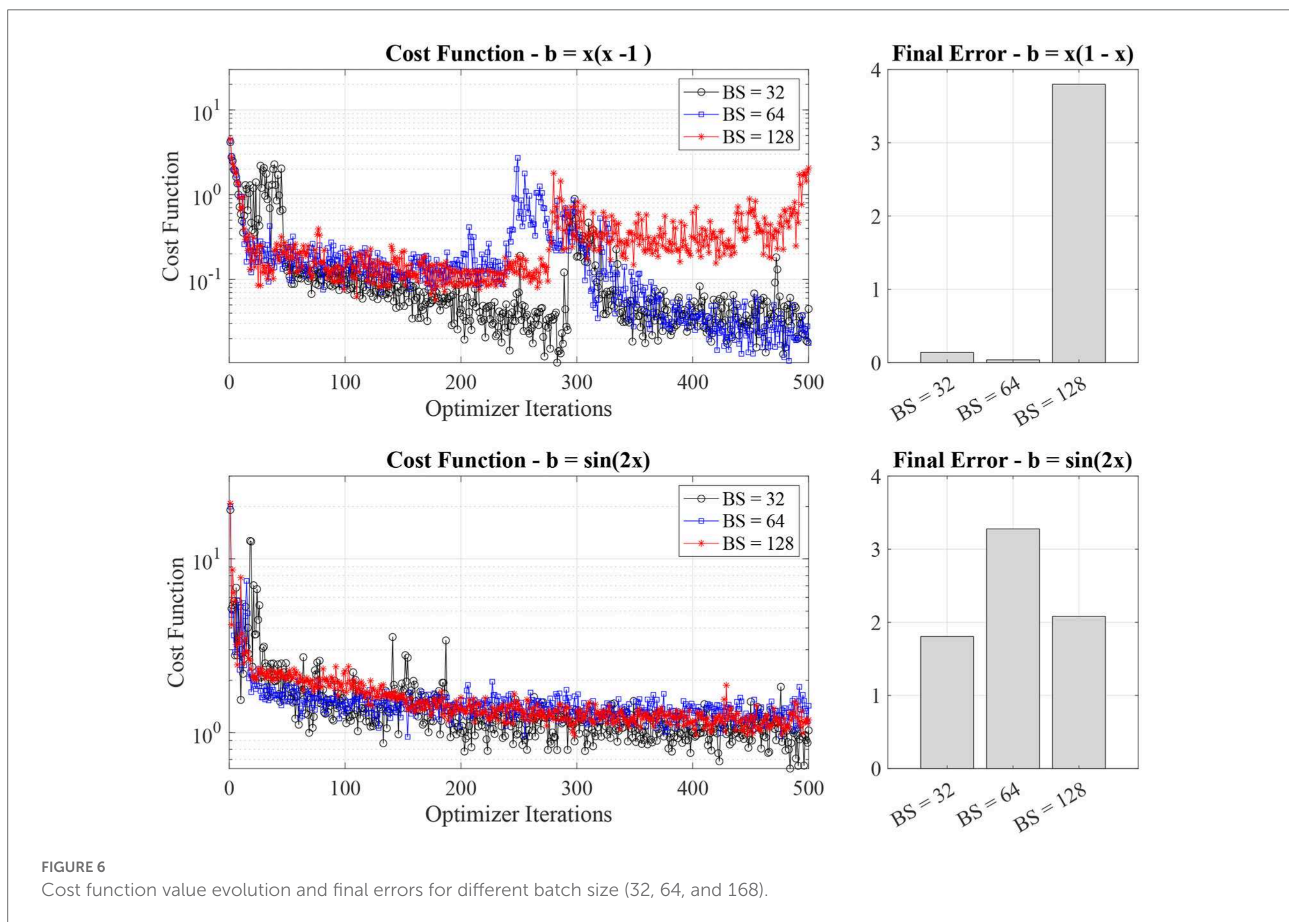

FIGURE 6
Cost function value evolution and final errors for different batch size (32, 64, and 168).

the network with eight quantum units exhibits a higher variability in the cost function evolution (see the red lines on the left panels of Figure 5). The neural network with two quantum units is too shallow to express the solution at higher accuracy. The interesting point is that in deep PINNs, e.g., with eight quantum neural units, the L-BFGS-B leads to significant error as it converges to local minima of the training landscape with an incorrect solution. This is probably due to switching too early to L-BFGS-B (after 85 SGD iterations) without letting the optimizers increase the exploration of the landscape.

Finally, we investigate the impact of the batch size per iteration and show it in Figure 6. The increase in batch size leads to a rise in the number of collocation points and associated cost function evaluations per optimizer iterations.

While intuitively, we would expect a noticeable improvement of the quantum PINN network when increasing the batch size, by analyzing Figure 6, in practice, we do not note a significant impact of the batch size on the performance of the quantum PINN. This is except for the quadratic source case and batch size equal to 128 (red line in the top left panel of Figure 6, where the performance drops due to convergence to a local minimum of the training landscape.

4. Discussion and conclusion

This work investigated the development of PINN for solving differential equations on heterogeneous CPU-QPU systems using a CV quantum computing approach. Quantum PINN solvers are fundamentally variational quantum circuit solvers [54] with an additional classical residual network to provide a cost function for the optimization step. The evaluation of the approximated solution is always carried out on the quantum computer (in this work, a simulated one). Currently, the calculation of the loss function *via* the residual neural network is carried out on the CPU. While this study uses a residual network on the CPU, the automatic differentiation might likely be implemented on quantum hardware. Given the gate formulation in Section 2.1, differential operators might be automatically synthesized as additional gates to express differentiation.

Overall, we found the programming interfaces conveniently abstract the QPU and quantum hardware: the programmer does not explicitly take care of the data movement or offloading of the quantum circuit execution. As in the case of GPU, the Strawberry Fields and TensorFlow frameworks provide a convenient approach for the transparent usage of the QPU

co-processors. In particular, the option of having a batch operation abstracts the parallel operation of multiple parallel cost function evaluations and measurements similarly to GPU. The programmer needs a basic understanding of expressing quantum circuits using Quantum Assembly languages, such as Blackbird.

We showed that a CV quantum neural network could provide a surrogate neural network for approximating a PDE solution to a certain degree in two test cases, solving the one-dimensional Poisson equation with quadratic and sinusoidal source terms. Our results showed that the optimizer's choice is the most impactful on the PINN solver accuracy: SGD solvers lead to a more accurate and stable solution than adaptive optimizers. The depth of the quantum neural network affects the PINN performance. In the two test cases, we found four layers provided higher accuracy than a two-layer quantum network and comparable performance to the eight-layer network. In the two test cases, we did not find a dependency on batch size, e.g., the number of collocation points we need for the training.

The main priority for developing further quantum PINN is to address their current low accuracy results. In all our tests, in practice, reducing the error below a certain value or increasing the convergence in a finite number of iterations has been challenging. This is likely related to the *barren plateau* problem [55], affecting all the hybrid quantum-classical algorithms involving an optimization step: the classical optimizer is stuck on a barren plateau of the training landscape with an exponentially small probability of exiting it. The *barren plateau* problem is fundamentally due to the geometry of parametrized quantum circuits (our quantum surrogate neural network) and training landscapes related to hybrid classical-quantum algorithms [55, 56]. When comparing quantum to classical PINN in our implementation, we found that the optimizer exploration of the training landscape in the case of quantum PINN is not as effective as in classical PINNs, and adaptive optimizers are less performant than basic SGD optimizers. Potential *classical* strategies to mitigate this problem are the usage of a quantum ResNet or convolutional surrogate quantum networks [45, 55], skip connections [57], dropout techniques (in multi-qumodes neural networks) [58], a structured initial guess, as used in quantum simulations, and pre-training segment by segment [59].

This exploratory work has several limitations. First, this work does not prove any quantum advantage but shows challenges and opportunities for implementing PINNs on quantum computing systems. The current problem (one-dimensional Poisson equation) does not exploit quantum entanglement. A PINN for solving higher-dimensional problems or systems of equations would require the usage of entangled qumodes. Second, for CV quantum computing, the QPINN has only been tested on a quantum computer simulator, the Strawberry Fields TensorFlow backend. To execute on future CV quantum computers, additional work must be done to consider the connectivity of the qumodes and hardware constraints, e.g., availability and the performance of gates on the given quantum systems [60]. Despite all these limitations, this work further investigates quantum PINN, showing the difference between classical and quantum PINNs and pinpointing current challenges to be addressed for PINN solvers on hybrid QPU-CPU systems.

Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found below: https://github.com/smarkidis/Quantum-Physics-Informed-Neural-Network.

Author contributions

SM developed the code, performed the simulations, analyzed the results, and wrote the manuscript.

Funding

Financial support was provided by the Swedish e-Science Research Centre Exascale Simulation Software Initiative (SESSI).

Acknowledgments

The author would like to thank Felix Liu and Pawel Herman for the insightful suggestions about stochastic optimizers and Vincent Elfving for discussing previous fundamental work on DQCs and quantum PINNs.

Conflict of interest

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

1. Markidis S. The old and the new: can physics-informed deep-learning replace traditional linear solvers? *Front Big Data*. (2021) 4:669097. doi: 10.3389/fdata.2021.669097

2. Cai S, Mao Z, Wang Z, Yin M, Karniadakis GE. Physics-informed neural networks (PINNs) for fluid mechanics: a review. *Acta Mechanica Sinica*. (2022) 37:1727–38. doi: 10.1007/s10409-021-01148-1

3. Haghighat E, Juanes R. Sciann: a keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *Comput Methods Appl Mech Eng*. (2021) 373:113552. doi: 10.1016/j.cma.2020.113552

4. Chen Y, Lu L, Karniadakis GE, Dal Negro L. Physics-informed neural networks for inverse problems in nano-optics and metamaterials. *Opt Express*. (2020) 28:11618–33. doi: 10.1364/OE.384875

5. Baydin AG, Pearlmutter BA, Radul AA, Siskind JM. Automatic differentiation in machine learning: a survey. *J March Learn Res*. (2018) 18:1–43. doi: 10.48550/arXiv.1502.05767

6. Amari SI. Backpropagation and stochastic gradient descent method. *Neurocomputing*. (1993) 5:185–96. doi: 10.1016/0925-2312(93)90006-O

7. Kingma DP, Ba J. Adam: a method for stochastic optimization. *arXiv[Preprint].arXiv:14126980*. (2014). doi: 10.48550/arXiv.1412.6980

8. Liu DC, Nocedal J. On the limited memory BFGS method for large scale optimization. *Math Program*. (1989) 45:503–28. doi: 10.1007/BF01589116

9. Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J Comput Phys*. (2019) 378:686–707. doi: 10.1016/j.jcp.2018.10.045

10. Lu L, Meng X, Mao Z, Karniadakis GE. DeepXDE: a deep learning library for solving differential equations. *SIAM Rev*. (2021) 63:208–28. doi: 10.1137/19M1274067

11. Shin Y, Darbon J, Karniadakis GE. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs. *arXiv[Preprint].arXiv:200401806*. (2020) doi: 10.4208/cicp.OA-2020-0193

12. Mishra S, Molinaro R. Estimates on the generalization error of physics-informed neural networks for approximating PDEs. *IMA J Numer Anal*. (2022) 42, 981–1022. doi: 10.1093/imanum/drab093

13. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. TensorFlow: a system for Large-Scale machine learning. In: *12th USENIX Symposium on Operating Systems DESIGN and implementation (OSDI 16)*. Savannah, GA (2016). p. 265–83.

14. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: an imperative style, high-performance deep learning library. In: *Advances in Neural Information Processing Systems Vol. 32*. Vancouver, BC (2019).

15. Chien SW, Markidis S, Olshevsky V, Bulatov Y, Laure E, Vetter J. TensorFlow doing HPC. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Rio de Janeiro: IEEE (2019). p. 509–18.

16. Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, et al. MLIR: a compiler infrastructure for the end of Moore's law. *arXiv[Preprint].arXiv:200211054*. (2020). doi: 10.48550/arXiv.2002.11054

17. Horowitz M, Alon E, Patil D, Naffziger S, Kumar R, Bernstein K. Scaling, power, and the future of CMOS. In: *IEEE InternationalElectron Devices Meeting, 2005 IEDM Technical Digest*. Washington, DC: IEEE (2005).

18. Moore GE. Cramming more components onto integrated circuits. *Proc. IEEE*. (1998) 86:82–5. doi: 10.1109/JPROC.1998.658762

19. Theis TN, Wong HSP. The end of Moore's law: a new beginning for information technology. *Comput. Sci. Eng*. (2017) 19:41–50. doi: 10.1109/MCSE.2017.29

20. Chow J, Dial O, Gambetta J. *IBM Quantum Breaks the 100-Qubit Processor Barrier*. New York, NY: IBM Research Blog (2021).

21. McKay DC, Alexander T, Bello L, Biercuk MJ, Bishop L, Chen J, et al. Qiskit backend specifications for openqasm and openpulse experiments. *arXiv[Preprint].arXiv:180903452*. (2018). doi: 10.48550/arXiv.1809.03452

22. Arute F, Arya K, Babbush R, Bacon D, Bardin JC, Barends R, et al. Quantum supremacy using a programmable superconducting processor. *Nature*. (2019) 574:505–10. doi: 10.1038/s41586-019-1666-5

23. Broughton M, Verdon G, McCourt T, Martinez AJ, Yoo JH, Isakov SV, et al. Tensorflow quantum: a software framework for quantum machine learning. *arXiv[Preprint].arXiv:200302989*. (2020). doi: 10.48550/arXiv.2003.02989

24. Gidney C, EkeråM. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*. (2021) 5:433. doi: 10.22331/q-2021-04-15-433

25. Grover LK. A fast quantum mechanical algorithm for database search. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. New York, NY (1996). p. 212–9.

26. O'Malley PJ, Babbush R, Kivlichan ID, Romero J, McClean JR, Barends R, et al. Scalable quantum simulation of molecular energies. *Phys Rev X*. (2016) 6:031007. doi: 10.1103/PhysRevX.6.031007

27. Harrow AW, Hassidim A, Lloyd S. Quantum algorithm for linear systems of equations. *Phys Rev Lett*. (2009) 103:150502. doi: 10.1103/PhysRevLett.103.150502

28. Bravo-Prieto C, LaRose R, Cerezo M, Subasi Y, Cincio L, Coles PJ. Variational quantum linear solver. *arXiv[Preprint].arXiv:190905820*. (2019). doi: 10.48550/arXiv.1909.05820

29. Kyriienko O, Paine AE, Elfving VE. Solving nonlinear differential equations with differentiable quantum circuits. *Phys Rev A*. (2021) 103:052416. doi: 10.48550/arXiv.2011.10395

30. Paine AE, Elfving VE, Kyriienko O. Quantum quantile mechanics: solving stochastic differential equations for generating time-series. *arXiv[Preprint].arXiv:210803190*. (2021). doi: 10.48550/arXiv.2108.03190

31. Heim N, Ghosh A, Kyriienko O, Elfving VE. Quantum Model-Discovery. *arXiv[Preprint].arXiv:211106376*. (2021). doi: 10.48550/arXiv.2111.06376

32. Kyriienko O, Paine AE, Elfving VE. Protocols for trainable and differentiable quantum generative modelling. *arXiv[Preprint].arXiv:220208253*. (2022). doi: 10.48550/arXiv.2202.08253

33. Kumar N, Philip E, Elfving VE. Integral transforms in a physics-informed (Quantum) neural network setting: applications & use-cases. *arXiv[Preprint].arXiv:220614184*. (2022). doi: 10.48550/arXiv.2206.14184

34. Paine AE, Elfving VE, Kyriienko O. Quantum kernel methods for solving differential equations. *arXiv[Preprint].arXiv:220308884*. (2022). doi: 10.48550/arXiv.2203.08884

35. Chen CC, Shiau SY, Wu MF, Wu YR. Hybrid classical-quantum linear solver using Noisy Intermediate-Scale Quantum machines. *Sci Rep*. (2019) 9:1–12. doi: 10.1038/s41598-019-52275-6

36. Preskill J. Quantum computing in the NISQ era and beyond. *Quantum*. (2018) 2:79. doi: 10.22331/q-2018-08-06-79

37. Peruzzo A, McClean J, Shadbolt P, Yung MH, Zhou XQ, Love PJ, et al. A variational eigenvalue solver on a photonic quantum processor. *Nat Commun*. (2014) 5:1–7. doi: 10.1038/ncomms5213

38. McClean JR, Romero J, Babbush R, Aspuru-Guzik A. The theory of variational hybrid quantum-classical algorithms. *New J Phys*. (2016) 18:023023. doi: 10.1088/1367-2630/18/2/023023

39. Lloyd S, Braunstein SL. Quantum computation over continuous variables. In: *Quantum Information With Continuous Variables*. New York, NY: Springer (1999). p. 9–17.

40. Braunstein SL, Van Loock P. Quantum information with continuous variables. *Rev Mod Phys*. (2005) 77:513. doi: 10.1103/RevModPhys.77.513

41. Weedbrook C, Pirandola S, García-Patrón R, Cerf NJ, Ralph TC, Shapiro JH, et al. Gaussian quantum information. *Rev Mod Phys*. (2012) 84:621. doi: 10.1103/RevModPhys.84.621

42. Slussarenko S, Pryde GJ. Photonic quantum information processing: a concise review. *Appl Phys Rev*. (2019) 6:041303. doi: 10.1063/1.5115814

43. Ortiz-Gutiérrez L, Gabrielly B, Mu noz LF, Pereira KT, Filgueiras JG, Villar AS. Continuous variables quantum computation over the vibrational modes of a single trapped ion. *Opt Commun*. (2017) 397:166–174. doi: 10.1016/j.optcom.2017.04.011

44. Knudsen M, Mendl CB. Solving differential equations *via* continuous-variable quantum computers. *arXiv[Preprint].arXiv:201212220*. (2020). doi: 10.48550/arXiv.2012.12220

45. Killoran N, Bromley TR, Arrazola JM, Schuld M, Quesada N, Lloyd S. Continuous-variable quantum neural networks. *Phys Rev Res*. (2019) 1:033063. doi: 10.1103/PhysRevResearch.1.033063

46. Madsen LS, Laudenbach F, Askarani MF, Rortais F, Vincent T, Bulmer JF, et al. Quantum computational advantage with a programmable photonic processor. *Nature.* (2022) 606:75–81. doi: 10.1038/s41586-022-04725-x

47. Fukui K, Takeda S. Building a large-scale quantum computer with continuous-variable optical technologies. *J Phys B.* (2022) 55:012001. doi: 10.1088/1361-6455/ac489c

48. Ruder S. An overview of gradient descent optimization algorithms. *arXiv[Preprint].arXiv:160904747.* (2016). doi: 10.48550/arXiv.1609.04747

49. Zeiler MD. Adadelta: an adaptive learning rate method. *arXiv[Preprint].arXiv:12125701.* (2012). doi: 10.48550/arXiv.1212.5701

50. Killoran N, Izaac J, Quesada N, Bergholm V, Amy M, Weedbrook C. Strawberry fields: a software platform for photonic quantum computing. *Quantum.* (2019) 3:129. doi: 10.22331/q-2019-03-11-129

51. Bromley TR, Arrazola JM, Jahangiri S, Izaac J, Quesada N, Gran AD, et al. Applications of near-term photonic quantum computers: software and algorithms. *Quant Sci Technol.* (2020) 5:034010. doi: 10.1088/2058-9565/ab8504

52. Spall JC. An overview of the simultaneous perturbation method for efficient optimization. *Johns Hopkins APL Tech Dig.* (1998) 19:482–92.

53. Pang G, Lu L, Karniadakis GE. fPINNs: Fractional physics-informed neural networks. *SIAM J Sci Comput.* (2019) 41:A2603–26. doi: 10.1137/18M1229845

54. Cerezo M, Arrasmith A, Babbush R, Benjamin SC, Endo S, Fujii K, et al. Variational quantum algorithms. *Nat Rev Phys.* (2021) 3:625–44. doi: 10.1038/s42254-021-00348-9

55. McClean JR, Boixo S, Smelyanskiy VN, Babbush R, Neven H. Barren plateaus in quantum neural network training landscapes. *Nat Commun.* (2018) 9:1–6. doi: 10.1038/s41467-018-07090-4

56. Arrasmith A, Cerezo M, Czarnik P, Cincio L, Coles PJ. Effect of barren plateaus on gradient-free optimization. *Quantum.* (2021) 5:558. doi: 10.22331/q-2021-10-05-558

57. Li H, Xu Z, Taylor G, Studer C, Goldstein T. Visualizing the loss landscape of neural nets. In: *Advances in Neural Information Processing Systems, Vol. 31.* Montréal, QC (2018).

58. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. *J Mach Learn Res.* (2014) 15:1929–58. Available online at: https://dl.acm.org/doi/10.5555/2627435.2670313

59. Bengio Y, Lamblin P, Popovici D, Larochelle H. Greedy layer-wise training of deep networks. In: *Advances in Neural Information Processing Systems, Vol. 19.* Boston, MA (2006).

60. Watabe M, Shiba K, Chen CC, Sogabe M, Sakamoto K, Sogabe T. Quantum circuit learning with error backpropagation algorithm and experimental implementation. *Quant Rep.* (2021) 3:333–49. doi: 10.3390/quantum3020021