ABSTRACT
Along with the development of AI democratization, the machine learning approach, in particular neural networks, has been applied to wide-range applications. In different application scenarios, the neural network will be accelerated on the tailored computing platform. The acceleration of neural networks on classical computing platforms, such as CPU, GPU, FPGA, ASIC, has been widely studied; however, when the scale of the application consistently grows up, the memory bottleneck becomes obvious, widely known as memory-wall. In response to such a challenge, advanced quantum computing, which can represent $2^N$ states with $N$ quantum bits (qubits), is regarded as a promising solution. It is imminent to know how to design the quantum circuit for accelerating neural networks. Most recently, there are initial works studying how to map neural networks to actual quantum processors. To better understand the state-of-the-art design and inspire new design methodology, this paper carries out a case study to demonstrate an end-to-end implementation. On the neural network side, we employ the multilayer perceptron to complete image classification tasks using the standard and widely used MNIST dataset. On the quantum computing side, we target IBM Quantum processors, which can be programmed and simulated by using IBM Qiskit. This work targets the acceleration of the inference phase of a trained neural network on the quantum processor. Along with the case study, we will demonstrate the typical procedure for mapping neural networks to quantum circuits.

KEYWORDS
neural networks, MNIST dataset, quantum computing, IBM Quantum, IBM Qiskit

1 INTRODUCTION
In the past few years, we have witnessed many breakthroughs in both machine learning and quantum computing research fields. On machine learning, the automated machine learning (AutoML) [47, 48] significantly reduces the cost of designing neural networks to achieve AI democratization. On quantum computing, the scale of the actual quantum computers has been rapidly evolving (e.g., IBM [13] recently announced to debut quantum computer with 1,121 quantum bits (qubits) in 2023). Such two research fields, however, have met the bottlenecks when applying the theoretical knowledge in practice. With the large-size inputs, the size of machine learning models (i.e., neural networks) significantly exceed the resource provided by the classical computing platform (e.g., GPU and FPGA); on the other hand, the development of quantum applications is far behind the development of quantum hardware, that is, it lacks killer applications to take full advantage of high-parallelism provided by a quantum computer. As a result, it is natural to see the emerging of a new research field, quantum machine learning.

Like applying machine learning to the classical hardware accelerators, when machine learning meets quantum computers, there will be tons of opportunities along with the challenges. The development of machine learning on the classical hardware accelerator experienced two phases: (1) the design of neural network tailored hardware [15, 16, 25, 26, 40, 45], and (2) the co-design of neural network and hardware accelerator [3, 5, 7, 11, 12, 14, 17, 21, 34–37, 39]. To best exploit the power of the quantum computer, it would be essential to conduct the co-design of neural network and quantum circuits design; however, with the different basic logic gates between quantum circuit and classical circuit designs, it is still unclear how to design a quantum accelerator for the neural network.

In this work, we aim to fix such a missing link by providing an open-source design framework. In general, the full acceleration system will be divided into three parts, the data pre-processing and data post-processing on a classical computer, and the neural network accelerator on the quantum circuit. In the quantum circuit, it will further include the quantum state preparation and the quantum computing-based neural computation. In the following of this paper, we will introduce all the above components in detail and demonstrate the implementation using IBM Qiskit for quantum circuit design and Pytorch for the machine learning model process.

The remainder of the paper is organized as follows. Section 2 presents an overview of the full system, Section 3 presents the case study on the MNIST dataset. Insights are discussed in Section 4. Finally, concluding remarks are given in Section 5.
2 OVERVIEW

Figure 1 demonstrates three types of neural network design: (1) the classical hardware accelerator; (2) the pure quantum computing based accelerator; (3) the hybrid quantum and classical accelerator. All of these accelerators follow the same flow that the data will be first pre-processed, then the neural computation is accelerated, and finally, the output data will go through the post-processing to obtain the final results.

2.1 Classical acceleration

After the success of deep neural networks (e.g., Alexnet [23] and VGGNet [32]) in achieving high accuracy, designing hardware accelerator became the hot topic in accelerating the execution of deep neural networks. On the application-specific integrated circuit (ASIC), works [6, 8, 41–43] studied how to design neural network accelerator using different dataflows, including weight stationery, output stationery, etc. By selecting dataflow for a dedicated neural computation, it can maximize the data reuse to reduce the data movement and accelerate the process, which derived the co-design of neural network and ASICs [38].

On the FPGA, work [40] first proposed the tiling based design to accelerate the neural computation, and works [15, 16, 24, 45] gave different designs and extended the implementation to multiple FPGAs. Driven by the AutoML, work [21] proposed the first co-design framework to involve the FPGA implementation into the search loop, so that both software accuracy and hardware efficiency can be maximized. The co-design philosophy also applied in other designs [11, 12, 20, 44] and in this direction, there exist many research works in further integrating the model compression into consideration [19, 28], accelerating the search process [27, 46],

2.2 Pure quantum computing

Most recently, the emerging works in using the quantum circuit to accelerate neural computation. The typical work include [9, 18, 33], among which the work [18] first demonstrates the potential quantum advantage that can be achieved by using a co-design philosophy. These works encode data to either qubits [9] or qubit states [18] and use superconducting-based quantum computers to run neural networks. These methods have the following limitations: Due to the short decoherence times in the superconducting-based quantum computers, the condition logic is not supported in the computing process. This makes it hard to implement a function that is not differentiable at all points, like the commonly used Rectified Linear Unit (ReLU) in machine learning models. However, it also has advantages, such as the design can be directly evaluated on an actual quantum computer, and there is no communication between the quantum-classical interface during the computation.

In the quantum circuit design, it includes two components: $U_p$ for quantum states preparation and $U_N$ for neural computation, as shown in Figure 1(b). After the component $U_N$, it will measure the quantum qubits to extract the output data, which will be further sent to the data post-processing unit to obtain the final results.

2.3 Hybrid quantum-classical computing

To overcome the disadvantage of pure quantum computing and take full use of classical computing, the hybrid quantum-classical computing for machine learning tasks is proposed [4]. It establishes a computing paradigm where different neurons can be implemented on either quantum or classical computers, as demonstrated in Figure 1(c). This brings flexibility in implementing functions (e.g., ReLU). However, at the same time, it will lead to massive data transfer between quantum and classical computers.

2.4 Our Focus in The Case Study

This work focus on providing a full workflow, starting from the data pre-processing, going through quantum computing acceleration, and ending with the data post-processing. We will apply the MNIST data set as an example to carry out a case study.

Computing architecture and neural operation can affect the design. In this work, for the computing architecture, we focus on the pure quantum computing design, since it can be easily extended to the hybrid quantum-classical design by connecting the inputs and output of the quantum acceleration to the traditional classical accelerator; for the neural network, we focus on the multi-layer perceptron, which is the basic operation for a large number of neural computation, like the convolution.

3 CASE STUDY ON MNIST DATASET

In this section, we will demonstrate the detailed implementation of four components in the pure quantum computing based neural computation as shown in Figure 1(b): data pre-processing, quantum state preparation ($U_p$), neural computation ($U_N$), and data post-processing.

3.1 Data Pre-Processing

The first step of the whole procedure is to prepare the quantum data to be encoded to the quantum states. Kindly note in order
to utilize $N$ qubits to represent $2^N$ data, it has constraints on the numbers; more specifically, if a vector $U_0$ of $2^N$ data can be arranged in the first column of a unitary matrix $U$, then for the initial state of $|\psi\rangle = 1 \cdot |0\rangle^N$, we can obtain $U_0$ by conducting $U|\psi\rangle = U_0$, where $|0\rangle^N$ represents the zero state with $N$ qubits.

```python
import torch
import numpy as np
import torchvision.transforms as transforms

class ToQuantumData(object):
    def __call__(self, tensor):
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        data = tensor.to(device)
        input_vec = data.view(-1)
        input_matrix = np.float64(input_matrix.
        output_data = output_matrix[: , 0]. view (1 , 1)
    
input_vec = data . view ( -1)
input_matrix = torch . zeros ( vec_len , vec_len )

vec_len = input_vec . size () [0]
input_matrix = np. float64 ( input_matrix .

input_matrix = torch.zeros ( vec_len , vec_len )

input_matrix = np. float64 ( input_matrix .

input_vec = data . view ( -1)

input_matrix = torch.zeros ( vec_len , vec_len )

vec_len = input_vec . size () [0]

input_matrix = np. float64 ( input_matrix .

output_data = output_matrix[: , 0]. view (1 , 1)

Listing 1: Converting classical data to quantum data

Listing 1 demonstrates the data conversion from the classical data to quantum data. We utilize the transforms in torchvision to complete the data conversation. More specifically, we create the ToQuantumData class in Line 5. It will receive a tensor (the original data) as input (Line 6). We apply Singular Value Decomposition (svd) provided by np.linalg to obtain the unitary matrix output_matrix (Line 14), then we extract the first vector from output_matrix as the output_data (Line 16), where the output_matrix represents $U$ and the output_data represents $U_0$. After we build the ToQuantumData class, we will integrate it into one "transform" variable, which can further include the data pre-processing functions, such as image resize (Line 20) and data normalization (Line 21). In creating the data loader, we can apply the "transform" to the dataset (e.g., we can obtain train data by using "train_data=datasets.MNIST(root=datapath, train=True,download=True, transform=transform)").

3.2 $U_0$: Quantum State Preparation

Theoretically, with the $n \times n$ unitary matrix $U$, we can directly operate the oracle on the quantum circuit to change $2^N$ states from the zero state $|0\rangle^N$ to $U_0$. This process is widely known as quantum-state preparation. The efficiency of quantum-state preparation can significantly affect the complexity of the whole circuit, and therefore, it is quite important to improve the efficiency of such a process. In general, there are two typical ways to perform the quantum-state preparation: (1) quantum random access memory (qRAM) [29] based approach [1, 22] and (2) computing based approach [2, 10, 30]. Let’s first see the qRAM-based approach, where the vector in $U_0$ will be stored in a binary-tree based structure in qRAM, which can be queried in quantum superposition and can generate the states efficiently. In IBM Qiskit, it provides the initialization function to perform quantum-state preparation, which is based on the method in [31].

```python
from qiskit import QuantumRegister, QuantumCircuit, ClassicalRegister
from qiskit import Aer, execute

# Input: a 4*4 matrix (data) holding 16 input data
inp = QuantumRegister(4,"in_qbit")
circ = QuantumCircuit(inp)
data_matrix = Q_InputMatrix = ToQuantumMatrix()(data.
fluaten())
circ.append(UnitaryGate(data_matrix, labels="Input"),
imp[0:4])
# Using StatevectorSimulator from the Aer provider
simulator = Aer.get_backend("statevector_simulator")
result = execute(circ, simulator).result()(
statevector = result.get_statevector(circ)
print(statevector)

Listing 2: Quantum-State Preparation in IBM Qiskit

In Listing 2, we give the codes to initialize the quantum states, using the unitary matrix $U$ which is converted from the original data in Listing 1(see Line 18). In this code snippet, we first create a 4-qubit QuantumRegister “inp” (line 6) and the quantum circuit (line 7). Then, we convert the input data to data_matrix, which is then employed to initialize the circuit using function UnitaryGate from qiskit.extensions. Finally, from line 10 to line 14, we output the states of all qubits to verify the correctness.

3.3 $U$: Neural Computation

Now, we have encoded the image data (16 inputs) onto 4 qubits. The next step is to perform the neural computation, that is, the weighted sum with quadratic function using the given binary weights $W$. Neural computation is the key component in quantum machine learning implementation. To clearly introduce this component, we first consider the computation of the hidden layer, which can be further divided into two stages: (1) multiplying inputs and weights, and (2) applying the quadratic function on the weighted sum. Then, we will present the computation of the output layer to obtain the final results.

Computation of one neural in the hidden layer

Stage 1: multiplying inputs and weights. Since the weight $W$ is given, it is pre-determined. We use the quantum gate to operate the weights with the inputs. The quantum gates applied here include the $X$ gate and the 3-controlled-Z gate with 3 trigger qubits. The function of such a 3-controlled-Z is to flip the sign of state $|111\rangle$, and the function of $X$ gate is to swap one state to another state.

For example, if the weight for state $|001\rangle$ is $-1$. We operate it on the input follows three steps. First, we swap the amplitude of state $|001\rangle$ to state $|111\rangle$ using two $X$ gates on the first two qubits. Then, in the second step, we apply controlled-Z gate to flip the sign of the state $|111\rangle$. Finally, in the third step, we swap the amplitude of state $|111\rangle$ back to state $|001\rangle$ using two $X$ gates on the first two qubits. Therefore, we can transverse all weights and apply the above three steps to flip the sign of corresponding states. Kindly note that since the non-linear function is a quadratic function, if
the number of $-1$ is larger than $+1$, we can flip all signs of weights to minimize the number of gates to be put in the circuit.

```python
def cccz(circ, q1, q2, q3, q4, aux1, aux2):
    # Apply Z-gate to a state controlled by 4 qubits
    circ.ccz(q1, q2, aux1)
    circ.ccz(q3, aux1, aux2)
    circ.cz(aux2, q4)
    # cleaning the aux bits
    circ.ccz(q3, aux1, aux2)
    circ.ccz(q1, q2, aux1)
    return circ
def neg_weight_gate(circ, qubits, aux, state):
    for idx in range(len(state)):
        if state[idx] == 0:
            circ.x(qubits[idx])
            aux = QuantumRegister(2, "aux_qbit")
        if weight_1_1[idx] == -1:
            circ.add_register(aux)
            aux = QuantumRegister(2, "aux_qbit")
        circ.x(inter_q_1)
        for idx in range(len(state)):
            if state[idx] == 0:
                circ.x(qubits[idx])
    # input: weight vector, weight_1_1
    aux = QuantumRegister(2, "aux_qbit")
    norm_q_1 = QuantumRegister(1, "norm_qubit")
    out_q_1 = QuantumRegister(1, "out_qubit")
    # Additional registers
    inter_q_1 = QuantumRegister(1, "inter_q_1_qubits")
    norm_q_1 = QuantumRegister(1, "norm_q_1_qubits")
    out_q_1 = QuantumRegister(1, "out_q_1_qubits")
    circ.add_register(aux)
    circ.add_register(aux)
    circ.barrier()
    # Input and weight multiplication
    if weight_2_1.sum() < 0:
        weight_1_1 = weight_1_1 - 1
        if weight_1_1.flatten().size() == 0:
            return circ
        state = "(0,0)".format(idx).zfill(4)
        neg_weight_gate(circ, inp, aux, state)
        circ.barrier()
    print(circ)
```

Listing 3: Multiplying inputs and weights on quantum

Listing 3 demonstrates the procedure of multiplying inputs and weights. In the list, the function `cccx` utilizes the basic quantum logic gates to realize the 3-controlled-Z gate with 3 control qubits. The involved basic gates include Toffoli gate (i.e., CCX) and controlled-Z gate (i.e., CZ). Since such a function needs auxiliary (a.k.a., ancilla) qubits, we include 2 additional qubits (i.e., aux) in the quantum circuit (i.e., circ), as shown in Lines 19-20. The function `neg_weights_gate` flips the sign of the given state, applying the 3-step process. Lines 11-13 complete the first step to swap the amplitude of the given state to the state of $|0\rangle^N$. Then, the `cccz` gate is applied to complete the second step. Finally, from line 15 to line 17, the amplitude is swap back to the given state. With the above two functions, we traverse the weights to assign the sign to each state from Lines 21-27. Kindly note that, after this operation, the states vector changed from the initial state $|\Psi\rangle = U_0$ to $|\Psi'\rangle = U_0'$ where the states have the weights.

**Stage 2: applying a quadratic function on the weighted sum.** In this stage, it also follows 3 steps to complete the function. In the first step, we apply the Hadamard (H) gates on all qubits to accumulate all states to the zero states. Then, the second step swap the amplitude of zero state $|0\rangle^N$ and the one-state $|1\rangle^N$. Finally, the last step applies the N-control-X gate to extract the amplitude to one output qubit $|0\rangle^N$. In which the probability of $O = |1\rangle$ is equal to the square of the weighted sum.

In the first step, the H gates can be accumulated to apply the amplitude of states, because the first row of $H^\otimes 4$ is $\frac{1}{2} [1, 1, 1, 1]$ and the $H^\otimes 4|\Psi\rangle$ performs the multiplication between the 4 x 4 matrix and the state vector $U_0'$. As a result, the amplitude of $|0\rangle^N$ will be the weighted sum with the coefficient of $\frac{1}{\sqrt{N}}$.

```python
def ccccz(circ, q1, q2, q3, q4, aux1, aux2):
    # Apply Z-gate to a state controlled by 4 qubits
    circ.ccz(q1, q2, aux1)
    circ.ccz(q3, aux1, aux2)
    circ.cz(aux2, q4)
    # cleaning the aux bits
    circ.ccz(q3, aux1, aux2)
    circ.ccz(q1, q2, aux1)
    return circ
```

Listing 4: Applying quadratic function on the weighted sum

Listing 4 demonstrates the implementation of the quadratic function on the weighted sum on Qiskit. In the list, function `cccz` is based on the Toffoli gate (i.e., CCX) to implement a 4-control-X gate to swap the amplitude between the zero state $|0\rangle^N$ and the one-state $|1\rangle^N$. In Line 14, `hidden_neuron` is an additional output qubit in the quantum circuit (i.e., circ) to hold the result for the neural computation, which is added in Lines 10-11.

For a neural network with $N$ neurons in the hidden layer, it has $N$ sets of weights. We can apply the above neural computation on $N$ sets of weights to obtain $N$ output qubits.

**Computation of one neuron in the output layer**

With these $N$ output qubits, we have two choices: (1) go to the classical computer and then encode the output of these $N$ outputs to log$_2 N$ qubits and then repeat these computations for the hidden layer to obtain the final results; (2) continuously use these qubits to directly compute the outputs, but the fundamental computation needs to be changed to the multiplication between random variables because the data associated with a qubit represents the probability of the qubit to be $|0\rangle$ state.

In the following, we demonstrate the implementation of the second choices (fundamental details please refer to [18, 33]). In this example, we follow the network structure with 2 neurons in the hidden layer. In addition, we consider there is only one parameter for the normalization function using one additional qubit for each output neuron. Let `hidden_neuron` be the outputs of 2 neurons in the hidden layer; let `weight_2_1` be the weights for the 1st output neuron in the 2nd layer; let `norm_flag_1` and `norm_para_1` be the normalized related parameters for the 1st output neuron. Then, we have the following implementation.
class_prob . append ( float ( mycount[b]) / qc_shots)
for b in range ( bits):
    class_prob = []
    (mycount, bits) = analyze ( counts)
    counts = fire_ibmq (circ, qc_shots, True)
    qc_shots = 8192
    return mycount, bits
else:
    mycount[i] += v
    if i in mycount.keys():
        if k[bits -1-i] == "1":
            mycount[i] = 0
        for i in range (2):
            def analyze (counts):
                mycount = ()
                for i in range (2):
                    mycount[i] = 0
                    for k, v in counts.items():
                        bits = len (k)
                        for i in range (bits):
                            if k[bits -1-i] == "1":
                                if i in mycount.keys():
                                    mycount[i] += v
                            else:
                                mycount[i] = v
                return mycount, bits
            qc_shots = 8192
            counts = fire_ibmq (circ, qc_shots, True)
            (mycount, bits) = analyze (counts)
            class_prob = []
        for b in range (bits):
            class_prob.append ( float (mycount[b]) / qc_shots)
        class_prob.index (max (class_prob))

Listing 5: Implementation of the second layer neural computation without measurement after the first layer

In the above list, it follows the 2-stage pattern for the computation in the hidden layer. If we modify all sub-index _1 to _2, then we can obtain the quantum circuit for the second output neuron.

3.4 Data Post-Processing

After all outputs are computed and stored in the out_q_1 and out_q_2 qubits, we can then measure the output qubits, run a simulation or execute on the IBM Q processors, and finally obtain the classification as follows.

```python
from qiskit.tools.monitor import job_monitor
def fire_ibmq (circuit, shots, Simulation = False, backend_name = "ibmq_essex"):
    count_set = ()
    if not Simulation:
        provider = IBMQ.get_provider ("ibmq-q-academic")
        backend = provider.get_backend (backend_name)
    else:
        backend = Aer.get_backend ("qasm_simulator")
    job_ibm_q = execute (circuit, backend, shots=shots)
    job_monitor (job_ibm_q)
    result_ibm_q = job_ibm_q.result ()
    counts = result_ibm_q.get_counts ()
    return counts
def analyze (counts):
    mycount = ()
    for i in range (2):
        mycount[i] = 0
        for k, v in counts.items():
            bits = len (k)
            for i in range (bits):
                if k[bits -1-i] == "1":
                    if i in mycount.keys():
                        mycount[i] += v
                    else:
                        mycount[i] = v
                return mycount, bits

Listing 6: Extract the classification results

Listing 6 demonstrate the above three tasks. The fire_ibmq function can execute the constructed circuit in either simulation or a given IBM Q processor backend. The parameter "shots" defines the number of execution to be executed. Finally, the counts for each state will be returned. On the implementation, the probability of each qubit (instead of each state) gives the probability to choose the corresponding class. Therefore, we create the “analyze” function to get the probability for each qubits. Finally, we obtain the classification results by extracting the index of the max probability in the “class_prob” set.

Kindly note that the Listing 6 can also be applied for the hybrid quantum-classical computing.

4 INSIGHTS

From the study of implementing neural networks onto the quantum circuits, there are several insights in terms of achieving quantum advantages, listed as follows.

- **Data encoding**: this case study encodes $2^N$ data to $N$ quantum qubits, which provides the opportunity to achieve quantum advantage for conducting inference for each input. An alternative way is to encode $N$ data to $N$ qubits, however, with the consideration that each data needs to be operated in the neural computation, such an encoding approach can hardly achieve the quantum advantage.

- **Quantum-state preparation**: by encoding $2^N$ data to $N$ quantum qubits, we can achieve quantum advantage only if the quantum-state preparation can be efficiently conducted with complexity at $O(N)$.

- **Quantum computing-based neural computation**: Neural computation can also become the performance bottleneck, using the design in Listing 3 to flip one sign at each time, it requires $O(2^N)$ gates in the worst case. To overcome this, [18] proposed a co-design approach to reduce the number of gates to $O(N^2)$.

5 CONCLUSION

This work demonstrates the framework in implementing neural networks onto quantum circuits. It is composed of three main components, including data pre-processing, neural computation acceleration, and data post-processing. Based on such a working flow, the data will be first encoded to quantum states and then operated to complete the operations in a neural network. The source codes can be found in https://github.com/weiwenjiang/QML_tutorial

ACKNOWLEDGEMENTS

This work is partially supported by IBM and University of Notre Dame (IBM-ND) Quantum program, and in part by the IBM-ILLINOIS Center for Cognitive Computing Systems Research.

REFERENCES

[1] Jonathan Allocock, Chang-Yu Hsieh, Ioannis Kerenidis, and Shengyu Zhang. 2020. Quantum algorithms for feedforward neural networks. *ACM Transactions on Quantum Computing*, 1, 1 (2020), 1–24.
[2] Johannes Bausch. 2020. Fast Black-Box Quantum State Preparation. *arXiv preprint arXiv:2009.10709* (2020).
[3] Song Bian, Weiren Jiang, Qing Lu, Yiyu Shi, and Takashi Sato. 2020. Nasm: Optimizing secure inference via neural architecture search. *arXiv preprint arXiv:2001.11854* (2020).
[4] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J Martinez, Jae Hyeon Yoo, Sergey V Isakov, Philip Massey, Murphy Yuezheng Niu, Ramin Halavati, Evan Peters, et al. 2020. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989* (2020).
