When Machine Learning Meets Quantum Computers: A Case Study

(Invited Paper)

Weiwen Jiang
wjiang2@nd.edu
University of Notre Dame
Notre Dame, IN, U.S.

Jinjun Xiong
jinjun@us.ibm.com
IBM Thomas J. Watson Research Center
Yorktown Heights, NY, U.S.

Yiyu Shi
yshi4@nd.edu
University of Notre Dame
Notre Dame, IN, U.S.

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

ACM Reference Format:
Weiwen Jiang, Jinjun Xiong, and Yiyu Shi. 2021. When Machine Learning Meets Quantum Computers: A Case Study: (Invited Paper). In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431629

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 leering 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.

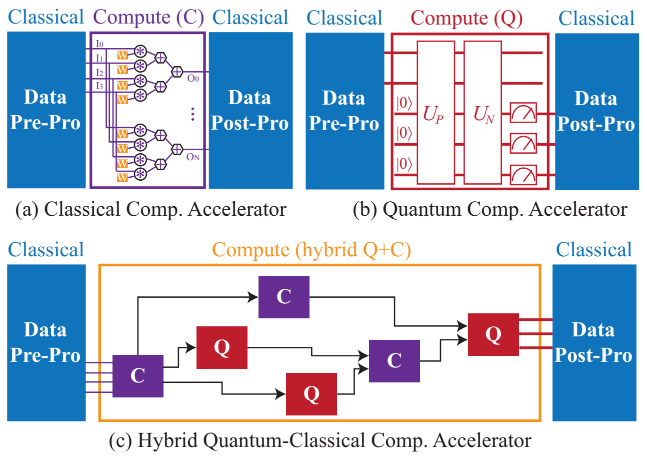

Figure 1: Illustration of three different types of computing schemes: (a) classical computing "C" based neural computation, where W is weights; (b) quantum computing "Q" based neural computation, where U_p is the quantum-state preparation and U_N is the neural computation; (c) hybrid quantum-classical computing "Q+C" based neural computation.

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 $|\psi\rangle = 1 \cdot |0\rangle^{\otimes N}$, we can obtain U_0 by conducting $U|\psi\rangle = U_0$, where $|0\rangle^{\otimes N}$ represents the zero state with N qubits.

```
1  import torch
2  import numpy as np
3  import torchvision.transforms as transforms
4  # Input: img_size=4 to represent the resolution of 4*4
5  class ToQuantumData(object):
6      def __call__(self, tensor):
7          device = torch.device("cuda" if torch.cuda.
       is_available() else "cpu")
8          data = tensor.to(device)
9          input_vec = data.view(-1)
10         vec_len = input_vec.size()[0]
11         input_matrix = torch.zeros(vec_len, vec_len)
12         input_matrix[0] = input_vec
13         input_matrix = np.float64(input_matrix.
       transpose(0,1))
14         u, s, v = np.linalg.svd(input_matrix)
15         output_matrix = torch.tensor(np.dot(u, v))
16         output_data = output_matrix[:, 0].view(1,
       img_size,img_size)
17         return output_data
18 # Similarly, we have "class ToQuantumMatrix(object)"
       which return the output_matrix
19 transform = transforms.Compose([transforms.Resize((
       img_size, img_size)), transforms.ToTensor(),
       ToQuantumData()])
20 # transform = transforms.Compose([transforms.Resize((
       img_size,img_size)),transforms.ToTensor(),
       transforms.Normalize((0.1307,), (0.3081,)),
       ToQuantumData()])
```

Listing 1: Converting classical data to qautnum 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_P: 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 $|0\rangle^{\otimes N}$ 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].

```
1  from qiskit import QuantumRegister, QuantumCircuit,
       ClassicalRegister
2  from qiskit.extensions import XGate, UnitaryGate
3  from qiskit import Aer, execute
4  import qiskit
5  # Input: a 4*4 matrix (data) holding 16 input data
6  inp = QuantumRegister(4,"in_qbit")
7  circ = QuantumCircuit(inp)
8  data_matrix = Q_InputMatrix = ToQuantumMatrix()(data.
       flatten())
9  circ.append(UnitaryGate(data_matrix, label="Input"),
       inp[0:4])
10 # Using StatevectorSimulator from the Aer provider
11 simulator = Aer.get_backend('statevector_simulator')
12 result = execute(circ, simulator).result()
13 statevector = result.get_statevector(circ)
14 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_N: 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 $|1111\rangle$, and the function of X gate is to swap one state to another state.

For example, if the weight for state $|0011\rangle$ is -1. We operate it on the input follows three steps. First, we swap the amplitude of state $|0011\rangle$ to state $|1111\rangle$ 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 $|1111\rangle$. Finally, in the third step, we swap the amplitude of state $|1111\rangle$ back to state $|0011\rangle$ 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.

```
1  def cccz(circ, q1, q2, q3, q4, aux1, aux2):
2      # Apply Z-gate to a state controlled by 4 qubits
3      circ.ccx(q1, q2, aux1)
4      circ.ccx(q3, aux1, aux2)
5      circ.cz(aux2, q4)
6      # cleaning the aux bits
7      circ.ccx(q3, aux1, aux2)
8      circ.ccx(q1, q2, aux1)
9      return circ
10 def neg_weight_gate(circ,qubits,aux,state):
11     for idx in range(len(state)):
12         if state[idx]=='0':
13             circ.x(qubits[idx])
14     cccz(circ,qubits[0],qubits[1],qubits[2],qubits[3],
        aux[0],aux[1])
15     for idx in range(len(state)):
16         if state[idx]=='0':
17             circ.x(qubits[idx])
18 # input: weight vector, weight_1_1
19 aux = QuantumRegister(2,"aux_qbit")
20 circ.add_register(aux)
21 if weight_1_1.sum()<0:
22     weight_1_1 = weight_1_1*-1
23 for idx in range(weight_1_1.flatten().size()[0]):
24     if weight_1_1[idx]==-1:
25         state = "{0:b}".format(idx).zfill(4)
26         neg_weight_gate(circ,inp,aux,state)
27         circ.barrier()
28 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 cccz utilizing 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 $|1\rangle^{\otimes 4}$. 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 accumulates all states to the zero states. Then, the second step swap the amplitude of zero state $|0\rangle^{\otimes N}$ and the one-state $|1\rangle^{\otimes N}$. Finally, the last step applies the N-control-X gate to extract the amplitude to one output qubit O, 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 applied to accumulate the amplitude of states, because the first row of $H^{\otimes 4}$ is $\frac{1}{4} \times [1, 1, 1, 1]$ and the $H^{\otimes 4}|\psi'\rangle$ performs the multiplication between the 4×4 matrix and the state vector U_0'. As a result, the amplitude of $|0\rangle^{\otimes N}$ will be the weighted sum with the coefficient of $\frac{1}{\sqrt{N}}$.

```
1  def ccccx(circ, q1, q2, q3, q4, q5, aux1, aux2):
2      circ.ccx(q1, q2, aux1)
3      circ.ccx(q3, q4, aux2)
4      circ.ccx(aux2, aux1, q5)
5      # cleaning the aux bits
6      circ.ccx(q3, q4, aux2)
7      circ.ccx(q1, q2, aux1)
8      return circ
9  # input: circ after stage 1
10 hidden_neuron = QuantumRegister(1,"out_qbit")
11 circ.add_register(hidden_neuron)
12 circ.h(inp)
13 circ.x(inp)
14 ccccx(circ,inp[0],inp[1],inp[2],inp[3],hidden_neuron,
       aux[0],aux[1])
```

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 ccccx is based on the basic Toffoli gate (i.e., CCX) to implement a 4-control-X gate to swap the amplitude between the zero state $|0\rangle^{\otimes 4}$ and the one-state $|1\rangle^{\otimes 4}$. 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 set 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_neurons* be the outputs of 2 neurons in the hidden layer; let *weight*_2_1 be the weights for the 1^{st} output neuron in the 2^{nd} layer; let norm_flag_1 and norm_para_1 be the normalization related parameters for the 1^{st} output neuron. Then, we have the following implementation.

```
1  # Additional registers
2  inter_q_1 = QuantumRegister(1,"inter_q_1_qbits")
3  norm_q_1 = QuantumRegister(1,"norm_q_1_qbits")
4  out_q_1 = QuantumRegister(1,"out_q_1_qbits")
5  circ.add_register(inter_q_1,norm_q_1,out_q_1)
6  circ.barrier()
7  # Input and weight multiplication
8  if weight_2_1.sum()<0:
9      weight_2_1 = weight_2_1*-1
10 idx = 0
11 for idx in range(weight_2_1.flatten().size()[0]):
12     if weight_2_1[idx]==-1:
13         circ.x(hidden_neurons[idx])
14 circ.h(inter_q_1)
15 circ.cz(hidden_neurons[0],inter_q_1)
16 circ.x(inter_q_1)
17 circ.cz(hidden_neurons[1],inter_q_1)
18 circ.x(inter_q_1)
19 # quadratic function on weighted sum
20 circ.h(inter_q_1)
```

```
circ.x(inter_q_1)
circ.barrier()
# normalization for two cases
norm_init_rad = float(norm_para_1.sqrt().arcsin()*2)
circ.ry(norm_init_rad,norm_q_1)
if norm_flag_1:
    circ.cx(inter_q_1,out_q_1)
    circ.x(inter_q_1)
    circ.ccx(inter_q_1,norm_q_1,out_q_1)
else:
    circ.ccx(inter_q_1,norm_q_1,out_q_1)
# Recove the inputs for the next neuron computation
for idx in range(weight_2_1.flatten().size()[0]):
    if weight_2_1[idx]==-1:
        circ.x(hidden_neurons[idx])
```

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.

```
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('ibm-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

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 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 Allcock, Chang-Yu Hsieh, Iordanis 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, Weiwen Jiang, Qing Lu, Yiyu Shi, and Takashi Sato. 2020. Nass: 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, Sergei V Isakov, Philip Massey, Murphy Yuezhen Niu, Ramin Halavati, Evan Peters, et al. 2020. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989* (2020).

[5] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[6] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[7] Yukun Ding, Weiwen Jiang, Qiuwen Lou, Jinglan Liu, Jinjun Xiong, Xiaobo Sharon Hu, Xiaowei Xu, and Yiyu Shi. 2020. Hardware design and the competency awareness of a neural network. *Nature Electronics* 3, 9 (2020), 514–523.

[8] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 92–104.

[9] Tacchino Francesco, Macchiavello Chiara, Gerace Dario, and Bajoni Daniele. 2019. An artificial neuron implemented on an actual quantum processor. *NPJ Quantum Information* 5, 1 (2019).

[10] Lov K Grover. 2000. Synthesis of quantum superpositions by quantum computation. *Physical review letters* 85, 6 (2000), 1334.

[11] Cong Hao, Yao Chen, Xinheng Liu, Atif Sarwari, Daryl Sew, Ashutosh Dhar, Bryan Wu, Dongdong Fu, Jinjun Xiong, Wen-mei Hwu, et al. 2019. NAIS: Neural architecture and implementation search and its applications in autonomous driving. *arXiv preprint arXiv:1911.07446* (2019).

[12] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-mei Hwu, and Deming Chen. 2019. FPGA/DNN Co-Design: An Efficient Design Methodology for 1oT Intelligence on the Edge. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[13] IBM. 2020. IBM's Roadmap For Scaling Quantum Technology. *https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/* (2020). Accessed: 2020-09-30.

[14] Weiwen Jiang, Qiuwen Lou, Zheyu Yan, Lei Yang, Jingtong Hu, X Sharon Hu, and Yiyu Shi. 2020. Device-circuit-architecture co-exploration for computing-in-memory neural accelerators. *IEEE Trans. Comput.* (2020).

[15] Weiwen Jiang, Edwin H-M Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–23.

[16] Weiwen Jiang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Lei Yang, Xianzhang Chen, and Jingtong Hu. 2018. Heterogeneous fpga-based cost-optimal design for timing-constrained cnns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2542–2554.

[17] Weiwen Jiang, Bike Xie, Chun-Chen Liu, and Yiyu Shi. 2019. Integrating memristors and CMOS for better AI. *Nature Electronics* 2, 9 (2019), 376–377.

[18] Weiwen Jiang, Jinjun Xiong, and Yiyu Shi. 2020. A Co-Design Framework of Neural Networks and Quantum Circuits Towards Quantum Advantage. *arXiv preprint arXiv:2006.14815* (2020).

[19] Weiwen Jiang, Lei Yang, Sakyasingha Dasgupta, Jingtong Hu, and Yiyu Shi. 2020. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4154–4165.

[20] Weiwen Jiang, Lei Yang, Edwin H-M Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. 2020. Hardware/Software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[21] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[22] Iordanis Kerenidis and Anupam Prakash. 2016. Quantum recommendation systems. *arXiv preprint arXiv:1603.08675* (2016).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[24] Bingzhe Li, M Hassan Najafi, and David J Lilja. 2015. An FPGA implementation of a restricted boltzmann machine classifier using stochastic bit streams. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 68–69.

[25] Bingzhe Li, M Hassan Najafi, and David J Lilja. 2016. Using stochastic computing to reduce the hardware requirements for a restricted Boltzmann machine classifier. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 36–41.

[26] Bingzhe Li, Yaobin Qin, Bo Yuan, and David J Lilja. 2017. Neural network classifiers using stochastic computing with a hardware-oriented approximate activation function. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 97–104.

[27] Yuhong Li, Cong Hao, Xiaofan Zhang, Xinheng Liu, Yao Chen, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. EDD: Efficient Differentiable DNN Architecture and Implementation Co-search for Embedded AI Solutions. *arXiv preprint arXiv:2005.02563* (2020).

[28] Qing Lu, Weiwen Jiang, Xiaowei Xu, Yiyu Shi, and Jingtong Hu. 2019. On neural architecture search for resource-constrained hardware platforms. *arXiv preprint arXiv:1911.00105* (2019).

[29] Alexander I Lvovsky, Barry C Sanders, and Wolfgang Tittel. 2009. Optical quantum memory. *Nature photonics* 3, 12 (2009), 706–714.

[30] Yuval R Sanders, Guang Hao Low, Artur Scherer, and Dominic W Berry. 2019. Black-box quantum state preparation without arithmetic. *Physical review letters* 122, 2 (2019), 020502.

[31] Vivek V Shende, Stephen S Bullock, and Igor L Markov. 2006. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 6 (2006), 1000–1010.

[32] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[33] Francesco Tacchino, Panagiotis Barkoutsos, Chiara Macchiavello, Ivano Tavernelli, Dario Gerace, and Daniele Bajoni. 2020. Quantum implementation of an artificial feed-forward neural network. *Quantum Science and Technology* (2020).

[34] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[35] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 10734–10742.

[36] Yawen Wu, Zhepeng Wang, Yiyu Shi, and Jingtong Hu. 2020. Enabling On-Device CNN Training by Self-Supervised Instance Filtering and Error Map Pruning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3445–3457.

[37] Lei Yang, Weiwen Jiang, Weichen Liu, HM Edwin, Yiyu Shi, and Jingtong Hu. 2020. Co-exploring neural architecture and network-on-chip design for real-time artificial intelligence. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 85–90.

[38] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. 2020. Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks. *arXiv preprint arXiv:2002.04116* (2020).

[39] Dewen Zeng, Weiwen Jiang, Tianchen Wang, Xiaowei Xu, Haiyun Yuan, Meiping Huang, Jian Zhuang, Jingtong Hu, and Yiyu Shi. 2020. Towards Cardiac Intervention Assistance: Hardware-aware Neural Architecture Exploration for Real-Time 3D Cardiac Cine MRI Segmentation. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–8.

[40] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 161–170.

[41] Jeff Zhang, Parul Raj, Shuayb Zarar, Amol Ambardekar, and Siddharth Garg. 2019. CompAct: On-chip Compression of Activations for Low Power Systolic Array Based CNN Acceleration. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–24.

[42] Jeff Zhang, Kartheek Rangineni, Zahra Ghodsi, and Siddharth Garg. 2018. Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[43] Jeff Jun Zhang and Siddharth Garg. 2018. FATE: fast and accurate timing error prediction framework for low power DNN accelerator design. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[44] Xinyi Zhang, Weiwen Jiang, Yiyu Shi, and Jingtong Hu. 2019. When neural architecture search meets hardware implementation: from hardware awareness to co-design. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 25–30.

[45] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[46] Yongan Zhang, Yonggan Fu, Weiwen Jiang, Chaojian Li, Haoran You, Meng Li, Vikas Chandra, and Yingyan Lin. 2020. DNA: Differentiable Network-Accelerator Co-Search. *arXiv preprint arXiv:2010.14778* (2020).

[47] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[48] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.