Comparison of Pure and Hybrid Quantum Neural Networks

Saaketh Rayaprolu and Neema Badihian

Abstract: In this paper we discuss the similarities and differences between two types of quantum neural networks, which are pure quantum models and classical-quantum hybrid models. In this pursuit, we cover the history, theory, and mathematical intuition behind classical, quantum, and hybrid neural networks, while drawing analogies to the similarities in design between the models. We dive into our approach towards building pure and hybrid quantum neural networks using Python, Qiskit, and the IBM Aer simulator backend. After an in-depth review of the algorithmic complexity of each of these models, we present our conclusions on the benefits and drawbacks of both the pure and hybrid models, along with recommendations for when to best implement these models. © 2022 The Author(s)

1. Introduction

Quantum neural networks are a subset of the class of neural networks that seek to harness the properties of quantum information and apply them to machine learning models to accelerate data processing while improving on concepts such as randomness, which are classically hard to achieve with high accuracy. Two of the major ideologies behind quantum neural networks are those of the pure models and the hybrid models; while pure models seek to implement neural networks that rely only on the properties of quantum information, hybrid models aim to pursue models that integrate some properties of both quantum and classical models.

Towards this goal, this paper has the following sections. In Section II, we introduce the basic design, functions, and vision behind neural networks. In Section III, we cover the history of neural networks, from conception to modern use. Section IV explores in depth the theory behind all three kinds of networks. Finally, in section V, we examine the operational costs and algorithmic efficiency of both quantum models, as well as our approaches in building a pure and a hybrid quantum neural network.

2. Background

Neural networks are a type of deep learning models that are widely used in machine learning. With applications to predictive modeling, image recognition, financial analysis, pattern recognition, and more, neural networks are built on the premise of using machine learning to mimic the patterns of the human brain. The structure of these models generally follows a pattern of having data input into the primary layer, often referred to as the input layer, after which it is processed in a series of ”hidden” layers, before finally outputting a carefully calculated result. Each layer contains a series of nodes, called perceptrons, that mimic the human neurons and contain data from various attributes of the data being analyzed, and many series of ”weights” that connect the nodes between layers. A definitive feature of neural networks that sets it apart from other machine learning models is the usage of reinforcement learning by use of reward functions to assess the accuracy of the analysis. While the optimal quantum neural network is not yet realized, many schools of thought constantly attempt to refine their efforts to incorporate quantum information into the networks and bring this idea to fruition.

3. History

Before the neural network was ever programmatically conceived, the first philosophical step was taken by Alexander Bain [1] in 1873 and William James [2] in 1890, both of whom independently detailed their ideas of the relation of associative memory and neural activity. Years later, in 1943, neurologist Warren McCulloch and mathematician Walter Pitts created the first computational model of neural networks, which was called the threshold logic unit [3] and would eventually evolve into what is called the perceptron today. In 1949, psychologist Donald Hebb suggested the idea of learning via neural plasticity [4], and in 1958, Frank Rosenblatt created the first true perceptron, utilizing a two-layer neural network that allowed the nodes of the second layer to hold ”original” values [5]. The following year, Bernard Widrow and Marcian Hoff created the Multiple Adaptive Linear Elements (MADALINE) model, which was the first attempt at a multiple layer neural network [6]. In 1982, Jon Hopfield created a recurrent neural network [7], and three years later, D.B. Parker published his report on backpropogation [8]. As quantum computing entered the fray, one of the earliest mentions of the concept of a quantum neural system was proposed in Richard Feynman's *Foundations of Physics*, launching major efforts into quantum machine learning, including but not limited to quantum neural networks. While the area is still undergoing major formative research and development, no set model of quantum neural networks have yet distanced themselves from the pack.

4. Theory and Approaches

4.1. Classical Neural Networks

To understand the logic of quantum neural networks, an understanding of their classical counterparts is critical, as a lot of the quantum logic is derived from classical roots. Classical neural networks, like any other machine learning model, take in an array of data points as input data. In order to pass the data to the next layer in the network, however, the nodes with the input data have to be connected to the nodes in the second layer by weights, which are unique to each pair of nodes and adjust the value of the data being passed (Fig. 1). Eq. 1 refers to the information sent to each node in layer L, where a is the input information, w is the weight between the nth node in the $L-1$ layer and the mth node L layer, and b is the bias added to the nth node of the $L-1$ layer.

$$z_m^{(L)} = \sum_{n=1}^{N} a_n^{(L-1)} w_{n,m}^{(L-1)} + b_n^{(L-1)} \tag{1}$$

However, the information z that is obtained is not the final information sent to the mth node in layer L; it must be run through an activation $f(x)$ that maps the output information to a certain desired distribution. Some of the more commonly used equations, the Sigmoid function (2) and the Leaky ReLU function (3), are shown below [9].

$$f(x) = \frac{1}{1+e^{-x}} \tag{2}$$

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ -0.01x, & \text{otherwise} \end{cases} \tag{3}$$

Once the data resulting from Equation 1 is passed through its assigned activation function, the output will be assigned as the new data for its respective node, where it will be reentered into the algorithm for the next layer. Once the data has run through the neural network, a function referred to as a loss function computes how much information was "lost" during the process of running the neural network.

$$Loss = \frac{1}{N}\sum_{i=1}^{N}(Y_i' - Y_i)^2 \tag{4}$$

Based on this statistic, the output data is passed back into the neural network in a process called backpropogation, where a process called gradient descent is utilized to update and optimize the weights in the network in order to minimize the loss function. Gradient descent is a function that seeks to minimize the loss function by directing the gradient, or the slope of the direction of steepest change, to tend to zero. The algorithm for equation descent, with the thetas representing the prior and posterior loss functions, alpha representing the "step size," and J representing the direction with the highest change in slope, is as follows:

$$\Theta_{n+1} = \Theta_n - \alpha \nabla J(\Theta_n) \tag{5}$$

This cycle of forward propogation, backpropogation, and gradient descent is repeated until the cost function is minimized, which is when the neural network is able to retain maximum information in its output. As such, neural networks have been refined over time to become excellent forecasting and predictive tools.

4.2. Pure Quantum Neural Networks

The pure quantum neural network closely follows the logic of its classical counterpart, but aims to optimize its calculations through quantum methods. Like classical neural networks, it has an input layer, a number of hidden layers, and an output layer. The hidden layers each have their own set of perceptrons, which may vary in number. THe circuit must therefore be initializedd with the number of qubits equal to the total count of perceptrons. The first step in the circuit is to Hadamard the input perceptrons to place them in superposition. Since we cannot feed them a classical input, we use Y-Rotation where the value of theta is our input. To connect the input layer to the hidden layers, a random and distinct controlled unitary is acted upon the perceptrons of the first hidden layer. Between every two layers in this network, from the input to the output layer, every combination of prior and posterior perceptrons has a controlled unitary gate applied, with the control being the prior gate. This forward propagation that we see here can be described by the following equation, where L refers to the number of layers and n is the number of unitaries per layer:

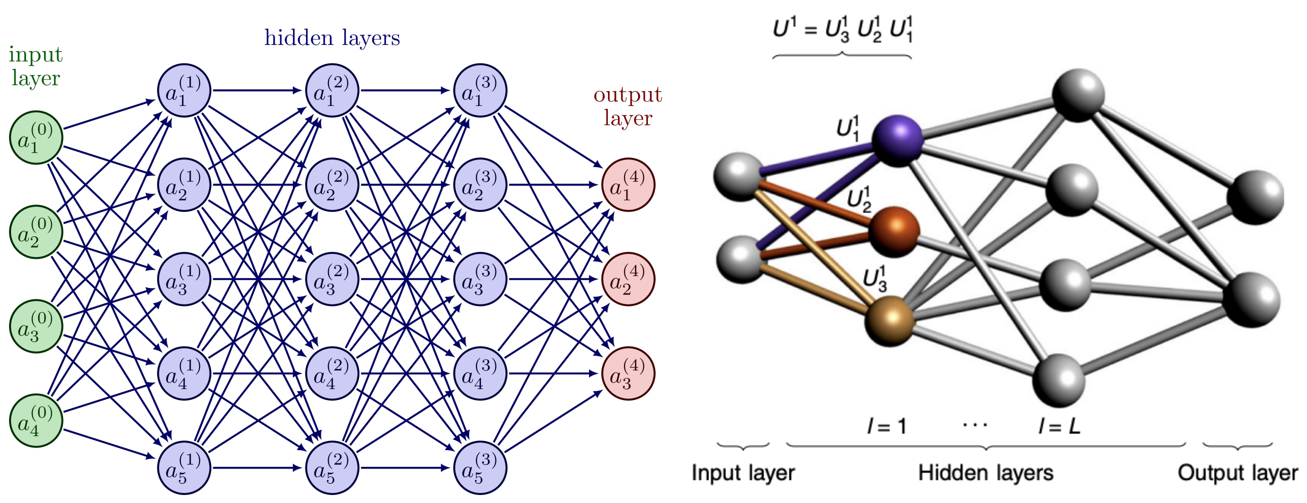

input layer
hidden layers
output layer
$a_1^{(0)}$ $a_2^{(0)}$ $a_3^{(0)}$ $a_4^{(0)}$
$a_1^{(1)}$ $a_2^{(1)}$ $a_3^{(1)}$ $a_4^{(1)}$ $a_5^{(1)}$
$a_1^{(2)}$ $a_2^{(2)}$ $a_3^{(2)}$ $a_4^{(2)}$ $a_5^{(2)}$
$a_1^{(3)}$ $a_2^{(3)}$ $a_3^{(3)}$ $a_4^{(3)}$ $a_5^{(3)}$
$a_1^{(4)}$ $a_2^{(4)}$ $a_3^{(4)}$
$U^1 = U_3^1\, U_2^1\, U_1^1$
U_1^1
U_2^1
U_3^1
$l = 1$ $\cdots$ $l = L$
Input layer
Hidden layers
Output layer

Fig. 1. Visualization of a classical neural network (left) and a pure quantum neural network (right). The green represents the input layer, where the data is entered into the model; the blue are the hidden layers, where the data processing and forward propogation occur, and the red is the output layer [10]. Like the classical neural network, the quantum model also is stratified into layers, and the colored nodes demonstrate how matrix multiplication is used as culmination of weights applied from each qubit to the qubits of the subsequent layer [11].

$$\rho_{out} \equiv Tr_{in,hid}\{(U^{out}\prod_{l=1}^{L}(\bigotimes_{i=1}^{n}U_i^l))(\rho^{in}\otimes|0...0\rangle\langle 0...0|)(U^{out}\prod_{l=1}^{L}(\bigotimes_{i=1}^{n}U_i^l))^{\dagger}\} \tag{6}$$

The output we get here is not necessarily optimal. To get a more accurate output, backpropagation is still required. This is done by first calculating the cost function, C, and calculate the change in the cost after each hidden layer and after the output. The cost function used takes advantage of the well established value known as the fidelity. The fidelity tells us the similarity between two states. We find the fidelity between the network output and the correct output and then average this over the training data.

$$C = \frac{1}{N}\sum_{x=1}^{N}\langle\phi_x^{out}|\rho_x^{out}|\phi_x^{out}\rangle \tag{7}$$

The change in the cost is calculated after each layer is calculated by the following formula:

$$\Delta C = \frac{\varepsilon}{N}\sum_{x=1}^{N}\sum_{l=1}^{L+1}Tr\{\sigma_x^1\Delta\mathscr{E}^1(\rho_x^{l-1})\} \tag{8}$$

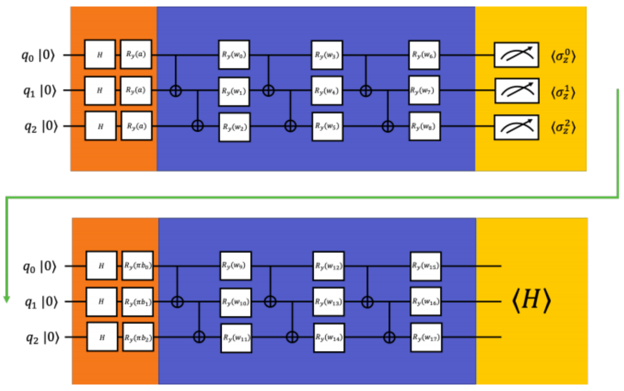
Here, the step size over the training data is multiplied by the the summations seen above. The summations are done over the training data and the layers up to the next layer. Inside the summations, it is evident that the trace of the adjoint channel is taken for the CP map at each layer acting on the correct output, which also acts on the product of the control unitaries up to any given layer, $\mathscr{E}$, and finally acts on the input. $\mathscr{E}^1$ is the product of all control unitaries acting on the first layer, but ρ_x^{l-1} is calculated by taking the product of the control unitaries up to layer $l-1$ and applying it to the input. The adjoint channel for the CP map is described by:

$$\sigma_x^l = \mathscr{F}^{l+1}(...\mathscr{F}^L(|\phi_x^{out}\rangle\phi_x^{out}|))...) \tag{9}$$

Now that the cost function and change in cost are defined, the network must be updated in order for it to improve. This is done by applying values calculated for each unitary. Each of these values is unique and depends on the unitary with which they are related. One part of these values are calculated by the following formula:

$$K_j^l = \eta\frac{2^{m_{l-1}}}{N}\sum_{x=1}^{N}Tr_{rest}\{\prod_{\alpha=j}^{1}U_\alpha^l(\rho_x^{l-1,l})\prod_{\alpha=1}^{j}U_\alpha^{l\,\dagger},\prod_{\alpha=j+1}^{m_l}U_\alpha^{l\,\dagger}(I_{l-1}\otimes\sigma_x^l)\prod_{\alpha=m_l}^{j+1}U_\alpha^l \tag{10}$$

In this formula, η is the learning rate, U_α^l is the controlled unitary, ρ is the tensor of input and the outer product of the perceptrons at any given layer, and σ holds the same value as defined in eq.(10). One we have the values of



q_0 $|0\rangle$
q_1 $|0\rangle$
q_2 $|0\rangle$
$\langle\sigma_z^0\rangle$
$\langle\sigma_z^1\rangle$
$\langle\sigma_z^2\rangle$
$\langle H\rangle$

Fig. 2. Visualization of the hybrid quantum neural network. The orange represents the encoding with Hadamard and R_y gates, the blue shows the propogation process, and the yellow shows the measurement and the classical cost function computation [12].

K, we apply $e^{i^l_j}$ to the corresponding unitaries, which will improve the unitaries and bring the result closer to the correct output. This process is repeated until the cost function reaches its maximum and we achieve the optimal network.

4.3. *Hybrid Quantum Neural Networks*

The hybrid quantum neural network seeks to draw from the computational speedup and increased accuracy of the quantum systems while incorporating the classical cost functions to improve algorithmic efficiency. To that end, the neural network itself is built on a Parametrized Quantum Circuit (PQC), meaning that it is initialized to a series of $|0\rangle$ states, all operations and logical gates are represented as unitary matrices on the quantum circuit, and it is intially parametrized by a free value θ [Fig. 3]. As such, the data is encoded by implementing Hadamard gates to each qubit, followed by Rotation gates over the Pauli-Y axis with a randomized input θ for each qubit, represented as follows:

$$|\psi_{enc}\rangle = \bigotimes_{i=0}^{n-1} R_y(\theta_i)H|0\rangle \tag{11}$$

Once the data is embedded as such, CNOT gates are used to entangle the qubits and more Rotation-Y gates as the weights; as such, they will be initialized as ansatze and updated over the course of the algorithm. The major benefit of the Rotation-Y gates that make its use ideal is that as it rotates on the Pauli-XZ plane, meaning it avoids all complex values and is consequently far simpler for calculation. Once the forward propogation is done, each qubit, in theory, is measured, and the resulting information is then used as the new θ for the Rotation-Y gates applied to next layer of qubits. However, since the measurement ranges from -1 to 1 (including phase) and θ can range from -2π to 2π, we scale the resulting measurement by a factor of π. However, measurement is an expensive process due to the fact that quantum tomography is inefficient. Therefore, instead of measuring the qubits, the expected value of each qubit can instead be used as the initialization θ in the next layer, which would allow the circuit to reuse the qubits without forcing measurement and collapse.

5. Algorithm Analyses

In order to compare the merits of the pure and hybrid quantum neural networks, we tested them both by utilizing them to estimate bond lengths of the hydrogen (H_2) molecule at various energy levels.

The quantum circuit was implemented with an input of two qubits, one hidden layer of three qubits, and an output layer of two qubits. First, both input qubits were initialized with Hadamard gates, after which Rotation-Y gates with the θ set to the input information were applied. After this, we apply controlled-unitary gates for our forward propogation process, with each unitary being randomly generated at first and updated via the backpropogation process over time. As this was a 2 qubit input algorithm, it was perfect for the estimation of the H_2 molecule. While the runtime was extremely high, the results were highly accurate.

As each matrix multiplication $M_{j,k} \times M_{k,l}$ has a complexity of O($jk+kl+jl$), and since we only use square matrices, every matrix multiplication is O(N^3) where N = 2^n. As such, with the maximum layer size denoted as p and the number of layers as h, the total number of operations between layers came to O(N^3p^2). Therefore, the forward propogation cumulates to O(hN^3p^2). The backpropogation process hinges on the evolution $U_n^h \longrightarrow e^{iK_n^h} U_n^h$, which is an O($N^3$) operation provided all variables. The matrix K is created via a sumation over traces of the matrices M_n^h, bringing it to O($2^{n^2}hp$). The generation of M is O($2N^3p^2$) multiplied by σ, which in turn is O(N^3h^2) times $\mathscr{F}$, where $\mathscr{F}_h$ is the adjoint of the unitary representative of the transform from layer $h-1$ to layer h. As the adjoint is calculated by Stinespring dilation and subsequent partial trace, coming to O(2^{3n^2}). As a result, the entire backpropogation process operates at O($2N^{11}p^3h^3$). Finally, the cost function operates at O(N^3), and the change in cost is measured at O(N^3p^2h). Therefore, by order of magnitude, the pure quantum neural network has an overall complexity of O($2^{n^{22}}$). As is evident, this model is likely best utilized for problems that strictly require quantum computation, given that it is computationally extremely expensive.

The hybrid circuit was initially implemented as shown above for three qubits; however, as the Hamiltonian representation of the LiH molecule was a 10 $\times$ 10 square matrix, it became necessary to implement this neural network on a 10-qubit PQC. After severe runtime issues, the H_2 molecule, which only needed a 2-qubit PQC, was implemented instead. Consequently, the runtime decreased; as we had to run a minimization on systems with dimensions of 2^n qubits, what might have taken hours for LiH instead took slightly under 45 minutes for H_2.

As stated above, every matrix multiplication in the algorithm is O(N^3) where N = 2^n. In the encoding step, we apply a Hadamard and a Rotation-Y to each qubit, after which we implement either $n-1$, n, or n^2-n CNOT gates and another Rotation-Y to each qubit for each segment prior to measurement. We repeat this process r times for r segments, after which the expected value is taken fro each qubit. To do so, each individual qubit has to be separated from the system via partial tracing, after which its density matrix is multiplied by the eigenvector either side to determine expected value. As a result, the measurement would operate with complexity O($n * \sum_{i=1}^{n-1}$ (2^{n+1-i}-1)), which roughly comes out to a complexity of O(2^{n^2}), which, when combined with our earlier calculation, amounts to O($2^{n^3+n^2}$). As we can see, while classical measurement options themselves are expensive, they are exponentially faster than their quantum equivalent, making it a far better fit for classical data at extremely large magnitudes.

6. Conclusion

As we can see from our results (Exhibit 4), the hybrid quantum neural network was slightly less accurate than the pure quantum neural network, while being significantly faster. While both networks use a similar forward propogation process, the cost function evaluation causes a divergence in methodology. While the quantum network evaluates a value and feeds matrices from that value through the cost function to update weights via backpropogation, the hybrid network simply uses expectation values as loss functions between layers and avoids backpropogation altogether, choosing to constantly just forward propogate with expected values. We can confidently assert that of the two models presented, the hybrid model is much more user-friendly and utilitarian given runtimes, but as it stands, the pure quantum neural network seems to have more trustworthy results. However, it would pose an interesting challenge in the future to potentially integrate backpropogation in the hybrid network without too much operational overhead or to reduce the complexity of the quantum network without cost of accuracy.

Contributions

Saaketh Rayaprolu—Researched and compared various classical, pure quantum, and hybrid quantum neural networks models, algorithm complexities, and history; designed slide deck; implemented pure quantum neural network and hybrid quantum neural network; participated in group discussions; drafted abstract, introduction, history, theory and approaches, algorithm analyses, conclusion, and references.

Neema Badihian—Researched and compared various classical, pure quantum, and hybrid quantum neural networks models, algorithm complexities, and history; designed slide deck; implemented quantum neural network; participated in group discussions; drafted theory and approaches; assisted on drafting algorithm analysis, history, and references.

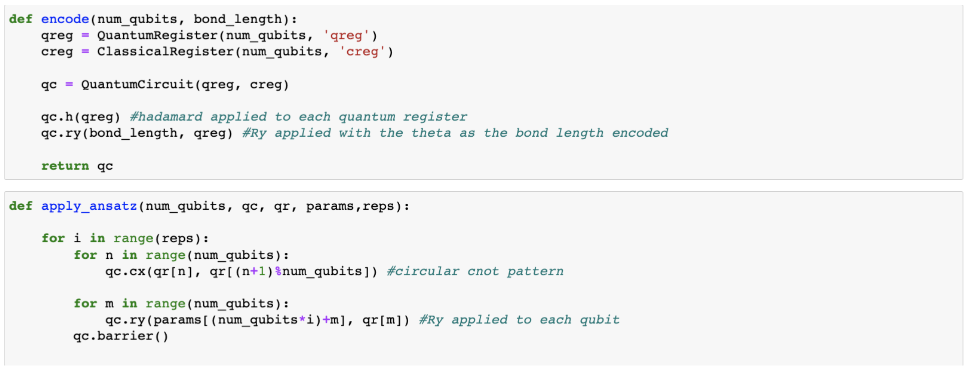
References

1. Bain (1873). *Mind and Body: The Theories of Their Relation*. New York: D. Appleton and Company.
2. James (1890). *The Principles of Psychology*. New York: H. Holt and Company.
3. McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". *Bulletin of Mathematical Biophysics*. 5 (4): 115–133. doi:10.1007/BF02478259.
4. Hebb, Donald (1949). *The Organization of Behavior*. New York: Wiley.
5. Rosenblatt, F. (1958). "The Perceptron: A Probalistic Model For Information Storage And Organization In The Brain". *Psychological Review*. 65 (6): 386–408. doi:10.1037/h0042519.
6. Widrow, Bernard, and Michael A. Lehr. "Artificial Neural Networks of the Perceptron, Madaline, and Backpropagation Family." *Neurobionics*, 1993, pp. 133–205., https://doi.org/10.1016/b978-0-444-89958-3.50013-9.

7. Hopfield, John J. (1982). Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences, 79 (8):2554–2558.
8. Parker, D. B. (1982). *Learning-logic* (Invention Report S81-64, File I). Stanford, CA: Office of Technology Licensing, Stanford University.
9. "Activation Functions: Fundamentals of Deep Learning." *Analytics Vidhya*, 19 July 2020, https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/.
10. Neutelings, Izaak. "Neural Networks." *TikZ.net*, 2 May 2022, https://tikz.net/neural_networks/.
11. Beer, Kerstin, et al. "Training Deep Quantum Neural Networks." *Nature Communications*, vol. 11, no. 1, 2020, https://doi.org/10.1038/s41467-020-14454-2.
12. Xia, Rongxin, and Sabre Kais. "Hybrid Quantum-Classical Neural Network for Calculating Ground State Energies of Molecules." *Entropy*, vol. 22, no. 8, 2020, p. 828., https://doi.org/10.3390/e22080828.

A. Appendix

Exhibit 1:



```python
def encode(num_qubits, bond_length):
    qreg = QuantumRegister(num_qubits, 'qreg')
    creg = ClassicalRegister(num_qubits, 'creg')

    qc = QuantumCircuit(qreg, creg)

    qc.h(qreg) #hadamard applied to each quantum register
    qc.ry(bond_length, qreg) #Ry applied with the theta as the bond length encoded

    return qc
```

```python
def apply_ansatz(num_qubits, qc, qr, params,reps):

    for i in range(reps):
        for n in range(num_qubits):
            qc.cx(qr[n], qr[(n+1)%num_qubits]) #circular cnot pattern

        for m in range(num_qubits):
            qc.ry(params[(num_qubits*i)+m], qr[m]) #Ry applied to each qubit
        qc.barrier()
```

Fig. 3. Code: Data encoding and ansatz application to PQC for hybrid neural network.

Exhibit 2:

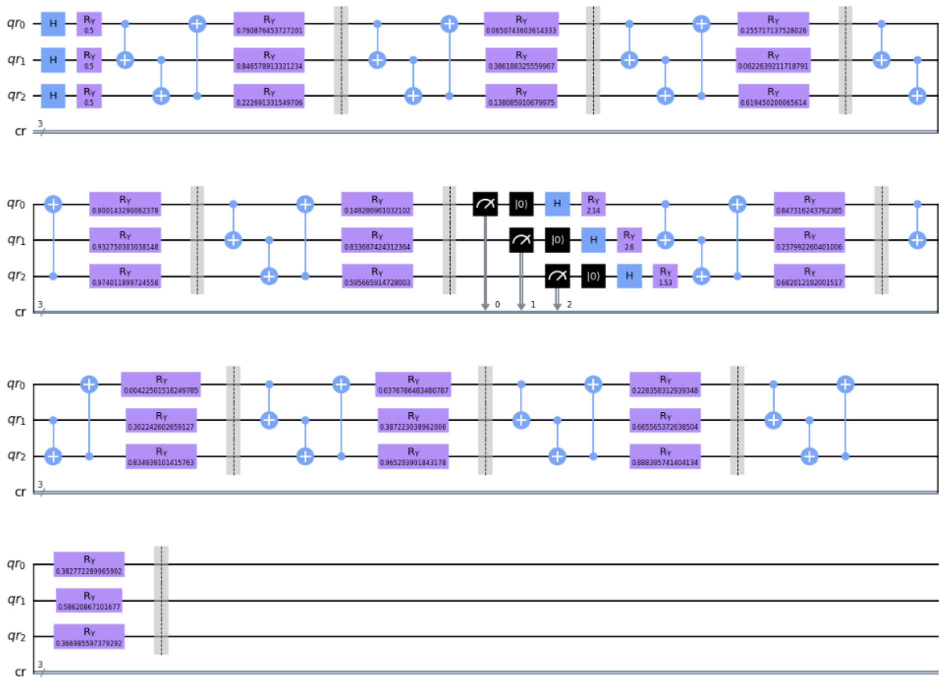
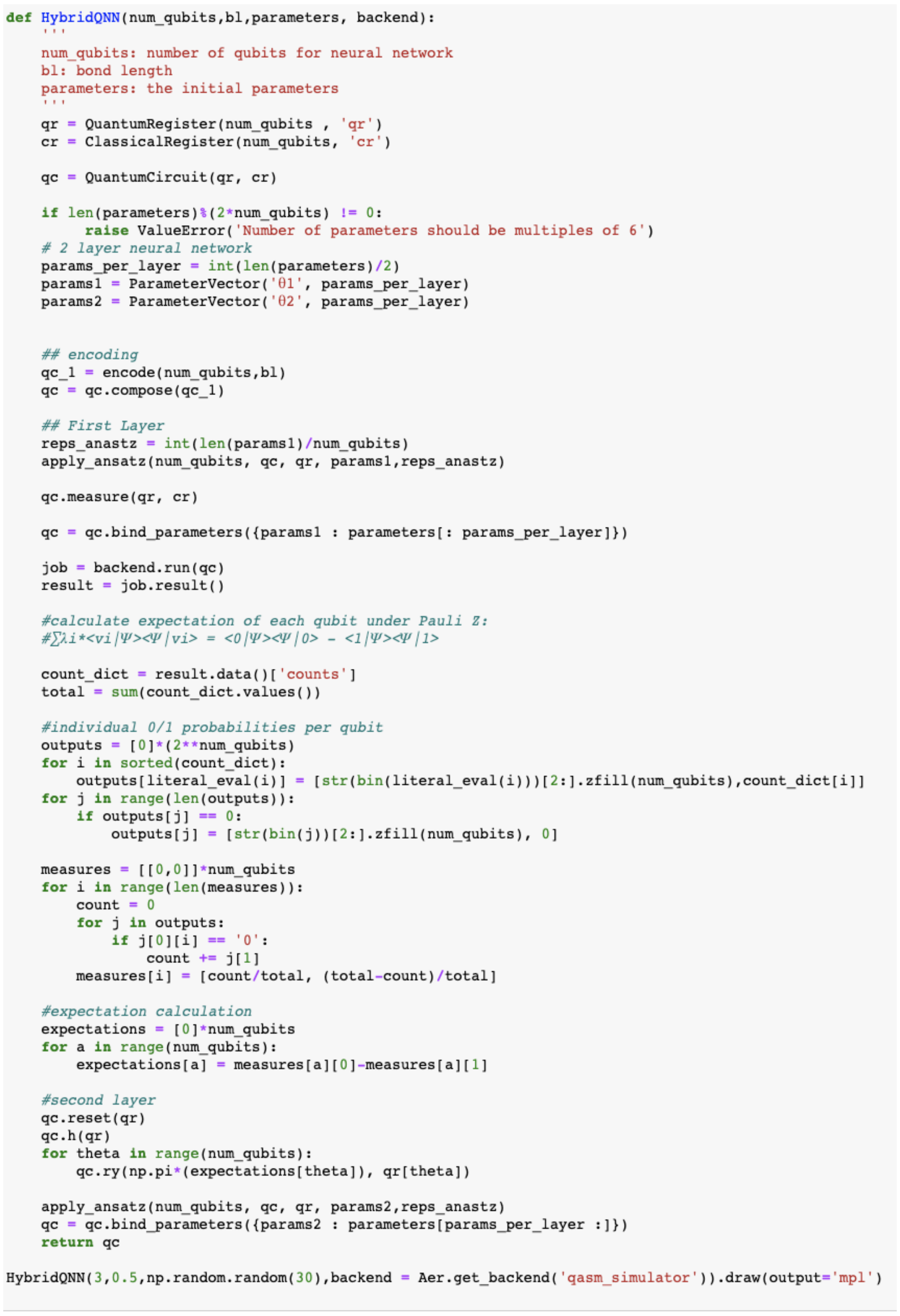


Fig. 4. Circuit: hybrid neural network for 3 qubits with 2 layers and 4 repetitions per layer.

Exhibit 3:



```
def HybridQNN(num_qubits,bl,parameters, backend):
    '''
    num_qubits: number of qubits for neural network
    bl: bond length
    parameters: the initial parameters
    '''
    qr = QuantumRegister(num_qubits , 'qr')
    cr = ClassicalRegister(num_qubits, 'cr')

    qc = QuantumCircuit(qr, cr)

    if len(parameters)%(2*num_qubits) != 0:
        raise ValueError('Number of parameters should be multiples of 6')
    # 2 layer neural network
    params_per_layer = int(len(parameters)/2)
    params1 = ParameterVector('θ1', params_per_layer)
    params2 = ParameterVector('θ2', params_per_layer)


    ## encoding
    qc_1 = encode(num_qubits,bl)
    qc = qc.compose(qc_1)

    ## First Layer
    reps_anastz = int(len(params1)/num_qubits)
    apply_ansatz(num_qubits, qc, qr, params1,reps_anastz)

    qc.measure(qr, cr)

    qc = qc.bind_parameters({params1 : parameters[: params_per_layer]})

    job = backend.run(qc)
    result = job.result()

    #calculate expectation of each qubit under Pauli Z:
    #∑λi*<vi|Ψ><Ψ|vi> = <0|Ψ><Ψ|0> - <1|Ψ><Ψ|1>

    count_dict = result.data()['counts']
    total = sum(count_dict.values())

    #individual 0/1 probabilities per qubit
    outputs = [0]*(2**num_qubits)
    for i in sorted(count_dict):
        outputs[literal_eval(i)] = [str(bin(literal_eval(i)))[2:].zfill(num_qubits),count_dict[i]]
    for j in range(len(outputs)):
        if outputs[j] == 0:
            outputs[j] = [str(bin(j))[2:].zfill(num_qubits), 0]

    measures = [[0,0]]*num_qubits
    for i in range(len(measures)):
        count = 0
        for j in outputs:
            if j[0][i] == '0':
                count += j[1]
        measures[i] = [count/total, (total-count)/total]

    #expectation calculation
    expectations = [0]*num_qubits
    for a in range(num_qubits):
        expectations[a] = measures[a][0]-measures[a][1]

    #second layer
    qc.reset(qr)
    qc.h(qr)
    for theta in range(num_qubits):
        qc.ry(np.pi*(expectations[theta]), qr[theta])

    apply_ansatz(num_qubits, qc, qr, params2,reps_anastz)
    qc = qc.bind_parameters({params2 : parameters[params_per_layer :]})
    return qc

HybridQNN(3,0.5,np.random.random(30),backend = Aer.get_backend('qasm_simulator')).draw(output='mpl')
```

Fig. 5. Code: Data processing, forward propogation, and classical loss function (expectation value) for hybrid neural network.

Exhibit 4:

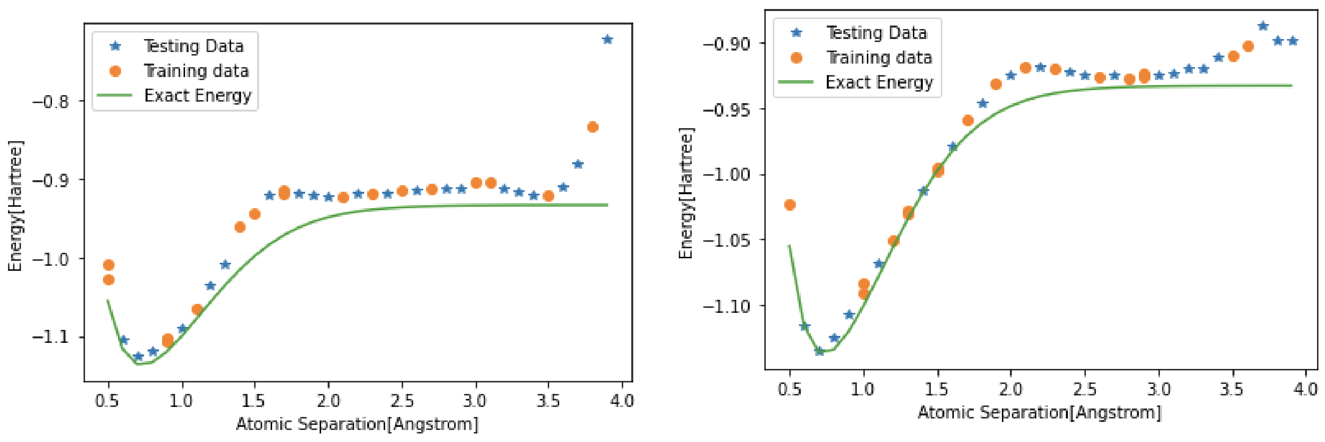


Fig. 6. Results: Side-by-side comparison of the hybrid neural network results (left) and pure QNN results (right) in estimating H_2 bond lengths.

Exhibit 5:

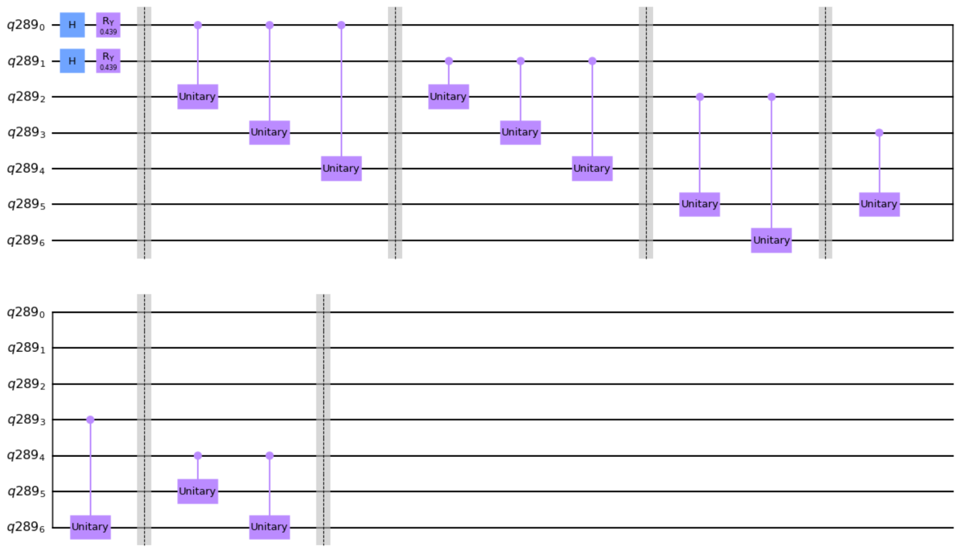
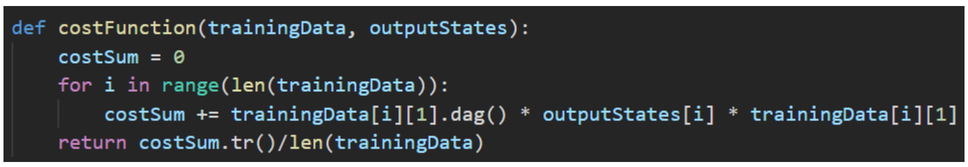


Fig. 7. Circuit: pure quantum neural network forward propogation on a [2,3,2] perceptron system.

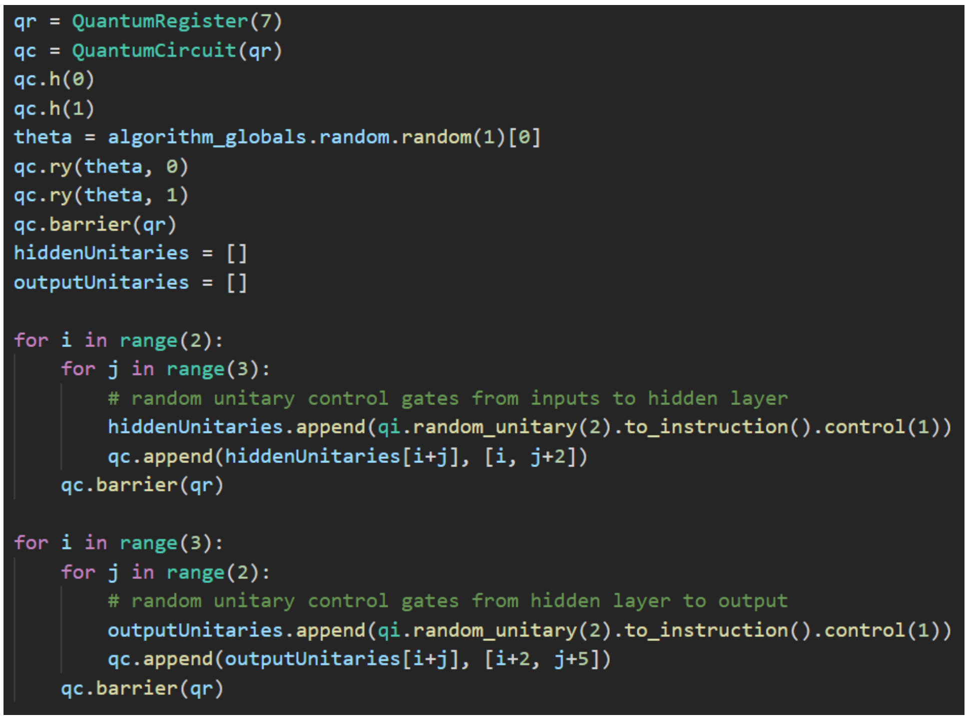
Exhibit 6:



```
def costFunction(trainingData, outputStates):
    costSum = 0
    for i in range(len(trainingData)):
        costSum += trainingData[i][1].dag() * outputStates[i] * trainingData[i][1]
    return costSum.tr()/len(trainingData)
```

Fig. 8. Code: Cost function calculation for pure quantum neural network, applied to backpropogation.

Exhibit 7:



```python
qr = QuantumRegister(7)
qc = QuantumCircuit(qr)
qc.h(0)
qc.h(1)
theta = algorithm_globals.random.random(1)[0]
qc.ry(theta, 0)
qc.ry(theta, 1)
qc.barrier(qr)
hiddenUnitaries = []
outputUnitaries = []

for i in range(2):
    for j in range(3):
        # random unitary control gates from inputs to hidden layer
        hiddenUnitaries.append(qi.random_unitary(2).to_instruction().control(1))
        qc.append(hiddenUnitaries[i+j], [i, j+2])
    qc.barrier(qr)

for i in range(3):
    for j in range(2):
        # random unitary control gates from hidden layer to output
        outputUnitaries.append(qi.random_unitary(2).to_instruction().control(1))
        qc.append(outputUnitaries[i+j], [i+2, j+5])
    qc.barrier(qr)
```

Fig. 9. Code: Initialization and unitary definition for quantum neural network forward propogation.