QNN-Gen: A Framework for Quantum Neural Networks

Collin Farquhar[0000−0002−6935−1293] and Jithin Jagannath[0000−0002−4059−6481]

ANDRO Computational Solutions, LLC, Rome NY, 13440, USA
cfarquhar@androcs.com jjagannath@androcs.com

Abstract. Quantum Neural Networks (QNNs) have garnered interest for a myriad of reasons ranging from philosophical interests, to biological modeling, to their use in machine learning architectures. This work focuses on the use of QNNs as machine learning models which can classify classical data. The task of using a quantum computation as a model can be broken down into three main steps: encode input data into a quantum state, define the model architecture, and train the model based on measurement outputs. The methods used in these three steps define the QNN. QNNs are currently in a developmental stage with many recent proposals for different data encoding schemes and model architectures; therefore, it would be useful to compare the various proposals. To that end, we present QNN-Gen, a framework built on top of Qiskit for generating QNN circuits. With a small number of parameter specifications for data encoding and model architecture, users are able to create and train simulated QNN circuits using very few lines of code. QNN-Gen enables users to build QNN models that have been proposed in the literature as well as novel QNNs. We demonstrate the capabilities of QNN-Gen by using it to generate some example QNN models. The machine learning community has benefited from the ability to easily create models so that they can be compared and cross-evaluated. By providing a framework to easily generate QNNs, we hope that QNN-Gen provides a similar utility to the quantum machine learning community.

Keywords: Quantum neural networks · Quantum machine learning · Software framework.

1 Introduction

Quantum Computing (QC) and Machine Learning (ML) are both areas with active research which aims, among other things, to push the boundaries of the set of problems that can be solved with computation in various fields [16, 20, 26].

Quantum computation uses fundamentally different units of information, qubits, and exploits the uniquely quantum properties of superposition and entanglement [24]. An original proposal for quantum computation was to harness these properties for the simulation of quantum mechanical systems [11]. While that remains an interesting area for quantum computing, since then, quantum information has been further studied, and QC shows the promise of being useful

for a broader range of applications such as in simulations of chemical processes, cryptography, optimization, and machine learning. The questions of how to best use a quantum computation and how to characterize any potential quantum advantage, are largely still open.

Machine learning, particularly supervised learning, has been an incredibly useful tool for a variety of applications ranging from pattern recognition, classification, regression, among others. ML algorithms are able to improve at a task, based on some performance measure, without the programmer having to explicitly encode a solution [23]. A ML model improves by tuning its learnable parameters. Neural Networks constitute a particular class of models for supervised learning that have achieved state-of-the-art performance on many problems by leveraging a structure of multiple layers of computation and through learning a large number of parameters [18].

Connections made between the fields of QC and ML have led to the emerging field of Quantum Machine Learning (QML) [8,30,34,37]. The central question is: what advantages may arise in the bidirectional interplay of QC and ML? While prior famous results in QC algorithms [13, 14, 33] give reasons to be optimistic about the possibility of a computational speed-up for QML, we are reminded that the reality of pursuing practical speed-ups will be a nuanced endeavour [2, 38].

In this work, we consider particular case of Quantum Neural Networks (QNNs) on classical data and how a unified open source framework can accelerate the innovation, implementation, and testing in this area of research. Therefore, we summarize the major contribution of this paper in the following,

- Debut and showcasing of QNN-Gen, a software framework with a modular design which prioritizes ease-of-use to help researchers interested in QNNs.
- For this purpose, we provide unified definitions and descriptions of key components of QNNs.
- We provide open access to the framework via Git, along with examples, to accelerate QNN research endeavours in the community.

The rest of this paper is organized as follows; in Sections 2, we describe our motivation and design philosophy for creating QNN-Gen and discuss some related works. Next, the building blocks and definitions that form the basis of our QNN-Gen framework is provided in Section 3. To assist the reader further, we provide example use cases in Section 4. Finally, in Section 5, we provide conclusions and directions for future work.

2 Motivation and Related Work

2.1 Motivation

Previous work in converting classical ML algorithms to a formalism of quantum computing and analogizing between classical and quantum algorithms has laid the groundwork for a common structure of QML models [6,10,12,15,17,22,28,35].

Common amongst QML models for quantum circuits is the need to encode data, define a model with learnable parameters, measure the circuit, and possibly perform some post-processing of measurement results. New proposals for QNNs following this structure are actively being developed, and it would useful to cross-evaluate the performance of various QNN models. To do this, a framework that provides a common repository of configurable QNN models would be useful so that researchers can benchmark new ideas against existing models and establish de facto state-of-the-art models for certain problems.

In addition to benchmarking model performance there are many other open questions which make the study of QNNs interesting. Determining the generalization properties of QNNs and investigating whether quantum data encodings give rise to unique and useful kernel methods [29], to name a few. Furthermore, from a fundamental AI perspective, exploring the question of what AI tasks are easy/hard for quantum computation vs. classical computation is interesting as it relates a low-level concept theoretical model of computation and information to a high-level concept of achievable AI tasks. A framework allows users to quickly implement and get hands-on experience with QNN models provides one avenue to start addressing questions such as the ones raised here.

We present QNN-Gen as the first step towards this ideal. QNN-Gen is a novel, high-level Python framework that serves as a wrapper over Qiskit [3] to easily generate circuit-based QNN models. By facilitating the quick generation of QNN models, researchers can then use these models to determine a model's off-the-shelf performance for new problems, compare and characterize models, and evaluate new models against the current state-of-art. The modular design of QNN-Gen also allows users to swap various sub-components of QNNs, thereby giving the possibility to create new models. It is our hope that QNN-Gen will also function as a shared framework for popular models, much in the same way that classical ML frameworks do.

We also hope that the abstraction and interface QNN-Gen provides may help to consolidate and unify the emerging field with a set of common terms and names for models, as it has been our experience that there is a degree of variability in QNN definitions in the literature.

2.2 Recent works

While QML has been an active field for a number of years, software frameworks for QML are relatively new. Over the course of this work, we have become aware of other software packages for QML, such as TensorFlow Quantum [4] and PennyLane [5]. In our opinion, these are both excellent packages for integrating simulated quantum circuits with existing classical ML and optimization frameworks for quantum-classical hybrid computation. However, creating the quantum circuits that implement a given QNN model often requires implementation from scratch, which demands an in-depth understanding of the particular model and can be time-consuming, although, PennyLane's *Templates* go some way in addressing this. We believe that researchers and practitioners will find QNN-Gen

is a faster and easier option for implementing and tweaking quantum circuits for the QNN models in its repository.

2.3 QNN-Gen Design Philosophy

The previous sections motivate our introduction of the beta release of QNN-Gen. It serves as a wrapper over Qiskit for the easy creation of QNN circuits. Instructions to download and use QNN-Gen can be found at [1].

QNN-Gen makes it easy for users to swap, tweak, and combine encoding methods, models, and measurement transformations for the creation of novel QNNs.

QNN-Gen was designed for ease-of-use and allows users to generate QNNs in very few lines of code when compared to alternative methods. Furthermore, it is a deliberate part of the design that expert users are able to make any reasonable changes that they might like to.

To promote ease-of-use, QNN-Gen often specifies fields with default values, however, it is straightforward to overwrite these values by specifying them explicitly in function calls or class attributes. For example, when instantiating the *dense angle encoding* class a user may overwrite the default *rotation gate* attribute, $RY(\theta)$, with any other single-qubit (parameterized) gate.

The core structure of QNN-Gen is designed so that using the framework is as close as possible to specifying one's modeling choices with minimum writing of "boiler-plate" code. To achieve this, the class structure matches the subcomponents of a QNN listed in 3. Through the use of abstract classes, QNN-Gen mirrors this structure and is inherently modular. The modular design makes adding new encoding methods, models, and measurement transformations to QNN-Gen relatively simple. In the next section we consider what it means for a quantum circuit to be a QNN and break down a QNN into constituent parts. We then give examples and define specific subcomponents that are available in QNN-Gen.

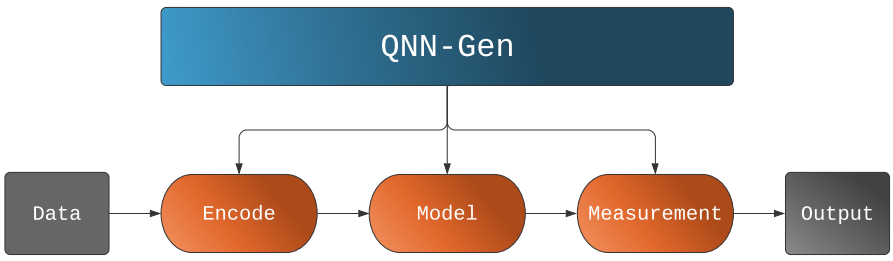

Fig. 1. The class structure of QNN-Gen

3 QNN-Gen Building Blocks & Definitions

QNN models can be broken into two major categories: models based on perceptron algorithms [27], and parameterized quantum circuits, which are related to the family of variational algorithms [21]. Quantum circuits consisting of multiple layers of parameterized unitary transformations, are analogized to classical neural networks in [22, 28]. The connection to classical NNs is made stronger if one incorporates classical non-linear post-processing of measurement results.

The subcomponents that define a QNN are:

1. A data encoding function
2. Choice of parameterized model
3. Measurement output and transformations in post-processing

From Figure 1 one can see that the information in this section is presented in a way that mirrors the class structure of QNN-Gen. The definitions we provide correspond with usable classes in QNN-Gen.

3.1 Data encoding

One of the most fundamental properties of any computational model is the space of inputs and outputs for the model. In general, QNNs can be defined for fully-classical inputs and outputs, fully-quantum [7,9], or a combination. In this paper, however, we consider QNNs that act on classical inputs and produce classical outputs. When considering theses networks, it is easier to draw connections and make comparisons to classical machine learning.

How one reads-in the data is an interesting challenge for quantum algorithms. For classical computing this is usually trivial, but for quantum computing to read-in classical data one must use a state preparation routine to somehow encode the data in the quantum state. A conventional data encoding function, E is one that maps a classical data vector, $\vec{x} \in \mathbb{R}^N$, to a quantum state $|\psi\rangle \in \mathcal{H}^M$.

$$E : \mathbb{R}^N \to \mathcal{H}^M \tag{1}$$

$$E(\vec{x}) = |\psi_x\rangle \tag{2}$$

(Encoding functions with a domain other than $\mathbb{R}$ are also possible.) Then, S_x is the unitary that implements the encoding function, E, on a starting register of qubits in the all-zero state.

$$S_x \left|0\right\rangle^{\otimes N} = |\psi_x\rangle \tag{3}$$

While not a requirement, it is a desirable property of encoding functions that they are injective.

The way that data is encoded is a consequential aspect of the full algorithm and may be considered to be part of the model itself. The encoding method determines the required dimensionality of the Hilbert Space, $\mathcal{H}$, of $|\psi\rangle$ needed to

encode a dataset. For example, $\vec{x} \in \mathbb{R}^N$ may map to $|\psi\rangle \in \mathcal{H}^{2^N}$, $|\psi\rangle \in \mathcal{H}^{2^{N-1}}$, or $|\psi\rangle \in \mathcal{H}^N$, depending on the choice of encoding function. We now define some of the encoding functions one can use in QNN-Gen.

Basis Encoding This method assumes a data vector with binary-valued features, $\vec{x_i} \in \mathbb{Z}_2^N$. Here, $\vec{x}$ can be viewed as a bit-string, and the basis encoding function maps it to the computational basis vector with the corresponding bit-string label. Note that the basis encoding algorithm associates the row-vector view of the data vector to the corresponding basis state label, rather than a state vector. For example, with $\vec{x} = [0, 1, 0]^T$ we have

$$E : \mathbb{Z}_2^N \to \mathcal{H}^{2^N} \tag{4}$$

$$E(\vec{x}) = |010\rangle \,. \tag{5}$$

Angle Encoding This method is an example of a *qubit encoding*. Angle encoding associates each feature of $\vec{x}$ to the state of a single qubit. We assume that the dataset is feature-normalized, so each $x_j \in [0, 1]$, and define angle encoding, referencing [17], as

$$E : \mathbb{R}^N \to \mathcal{H}^{2^N} \tag{6}$$

$$E(\vec{x}) = \bigotimes_{j=1}^{N} cos(cx_j)\,|0\rangle + sin(cx_j)\,|1\rangle \,. \tag{7}$$

Where c is a scaling factor. Angle encoding is presented in the literature with different scaling factors. Some authors [12, 34] use a scaling factor of $c = \pi/2$ which will not induce a relative phase between the qubit's $|0\rangle$ and $|1\rangle$ states. However, one could choose a scaling factor of $c = \pi$ and the function would remain injective for a single qubit. A function that is a sum (ignoring the tensor product in angle encoding) of $sin(\theta)$ and $cos(\theta)$ is 2π-periodic; however, θ going from $\pi \to 2\pi$ is equivalent, up to a global phase of -1, to the rotations as θ goes from $0 \to \pi$.

Angle encoding can be implemented using a single layer of parameterized RY gates. As noted in [17] an encoding function with constant circuit-depth, (depth $= 1$), is particularly advantageous for NISQ-era hardware [25].

Dense Angle Encoding Also citing [17] we define dense angle encoding, which more efficiently uses the Hilbert space as it maps two features from $\vec{x}$ to a single qubit. However, dense angle encoding requires depth > 1, though it is still a constant depth method. This illustrates a common property of encoding functions, that oftentimes, efficiently utilizing the storage capacity of probability amplitudes comes at the cost of higher depth circuits.

$$E : \mathbb{R}^N \to \mathcal{H}^{2^{N-1}} \tag{8}$$

$$E(\vec{x}) = \bigotimes_{j=0}^{N/2} cos(cx_{2j}) |0\rangle + e^{2\pi i x_{2j+1}} sin(cx_{2j}) |1\rangle . \tag{9}$$

Where, again, c is a scaling factor. One can see for any pair of consecutive features one feature will be used as the argument of the trigonometric functions and the other will determine the relative phase. As a cautionary note: if cx_{2j-1} evaluates to an angle for which lead *sin* or *cos* to evaluate to 0, then the information will be lost for the other feature encoded in the relative phase.

Binary Phase Encoding This method exploits the storage capacity of qubits by mapping $m = 2^N$ features to the 2^N probability amplitudes of an N-qubit state. However, one has to come up with a way to encode these 2^N parameters using whatever set of gates is available. This can lead to exponential circuit-depth on hardware. We give the definition as it follows from the sign-flip algorithm in [35].

Binary phase encoding takes a binary vector $\vec{x} \in \{1, -1\}^m$ and maps it to a quantum state with uniform-magnitude probability amplitudes and corresponding signs. For example, with $\vec{x} = [1, -1, 1, -1]^T$ we have

$$E : \{1, -1\}^m \to \mathcal{H}^m \tag{10}$$

$$E(\vec{x}) = \frac{1}{2}[1, -1, 1, -1]^T = |\psi_x\rangle , \tag{11}$$

where the vector in (11) is written in the basis of the Hilbert space, and the $\frac{1}{2}$ factor is for normalization.

All of the data encoding methods have different advantages and disadvantages and are intimately tied with the functionality of a given model. It is shown in [17] that for certain quantum-circuit classifiers, the encoding function can determine some characteristics of the decision boundary.

3.2 Model selection

In this section we present a perceptron-based QNN model and models for QNNs consisting of several parameterized layers. For the latter category, selecting a model involves choosing a particular ansatz with parameterized gates. Choosing the right model or ansatz for a problem is not always a straightforward task, and often requires a combination of problem-specific knowledge and artistry gained from experience as a practitioner. This is one reason that researchers may find QNN-Gen to be useful, as being able to quickly implement different models makes it easier for one to compare them.

In the following sections, we describe three models relevant to the current state of QNNs and that are available in QNN-Gen.

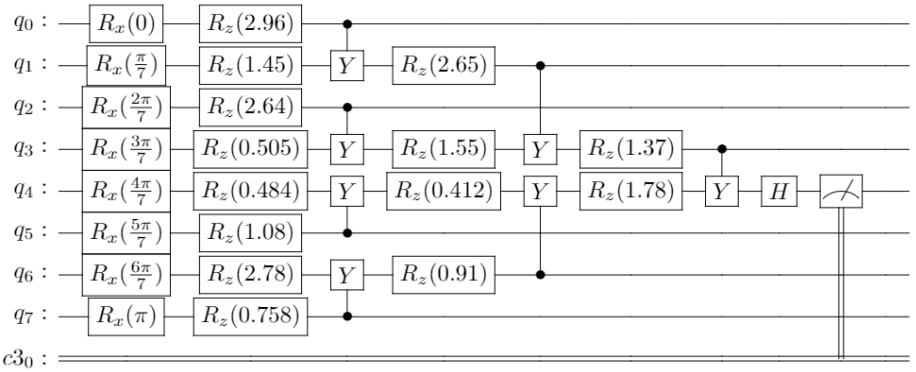

Fig. 2. A configurable tree tensor network created using QNN-Gen.

Tree Tensor Network In this section we describe the tree tensor network model as presented in [12]. Tensor networks are used to efficiently simulate certain states in quantum many-body physics [32, 36]. We highlight this as an interesting model inspired by an adjacent field.

The tree tensor network ansatz available in QNN-Gen consists of layers of parameterized single-qubit gates and two-qubit entangling gates. For a concrete example, QNN-Gen specifies these gates to be $RY(\theta)$ and CX by default. Each layer acts on half as many qubits as the previous layer, thus leading to $log_2(N)$ layers in total for a model consisting of N qubits. See Figure 2 for an example circuit that implements a tree tensor network model.

Binary Perceptron This is the name by which the perceptron algorithm defined in [35] is referenced in QNN-Gen. This algorithm is formulated such that it encodes binary input and weight vectors, $\vec{i}, \vec{w} \in \{1, -1\}^{2^N}$ into a quantum state of N qubits. We highlight that this algorithm makes use of the information storage potential coming from the 2^N probability amplitudes that define a quantum state.

In QNN-Gen we implement this algorithm using the sign-flip method defined in [35], which manipulates the amplitudes of the state vector using layers of $C^{N-1}Z$ and X gates. The procedure to generate a circuit that implements this algorithm is described in the following steps:

– Apply a full layer of H gates on all N qubits, each in the starting state $|0\rangle$.

- Use the sign-flip method to encode the input and weight vectors $\vec{i}$ and $\vec{w}$.
- Apply a full layer of H gates.
- Apply a full layer of X gates.
- Apply a $C^N X$ operation with the target on an ancillary qubit.

The authors show that this amounts to the information of the inner-product computation of $\vec{i}$ and $\vec{w}$ being stored in the state of the ancillary qubit. Figure 3 shows an example implementation of a binary perceptron model using $3 + 1$ qubits.

Entangled Qubit We chose to highlight this well-cited paper [10] for a model that has recorded accuracy results on an applied problem, namely, the classification of down-sampled MNIST digits [19].

This model using layers of 2-bit gates of the form

$$U(\theta) = exp(i\theta\sigma_i^{(j)} \otimes \sigma_k^{(l)}) \tag{12}$$

where $\sigma_i^{(j)}$ is a Pauli operator designated by $i \in \{x, y, z\}$ acting on the qubit labeled by the superscript j. The identity operator implicitly acts on all other qubits. This model holds one subscript index constant, for example j, and lets the other, l, range across all other qubit indices. In doing so, $N - 1$ two-qubit gates act on qubit j so that the qubit may be entangled with all other qubits. In [10], the authors limit to $\sigma_x\sigma_x$ and $\sigma_z\sigma_x$ and define the output to be the expectation value of an observable with respect to the qubit with the constant index.

3.3 Measurement and output

The measurement can be thought of as the output layer of the QNN. The most natural measurement information for simulated and real circuits comes in the form of counts of particular measurement outcomes in the computational basis over many executions. However, it is often the case that the raw counts are not the most useful output for a QNN model. In the section, we introduce some of the transformations on circuit outputs that are available in QNN-Gen and can be used as an output layer for QNNs.

Measurement and other transformations implemented in classical post-processing provide an interesting opportunity to inject some non-linear processes into the model. Simulation of coherent, closed quantum mechanical systems is necessarily linear, but the measurement operation provides a natural non-linear transformation. Connections between the non-linearity of measurement and *activation functions* in classical NNs have been pointed out by [31, 35], and others.

For all of the methods discussed below, QNN-Gen also supports measurement with respect to the eigenbases of the Pauli operators, Hadamard operator,

or an arbitrary Hermitian operator by performing the appropriate unitary transformation to rotate the operator's eigenbasis to the computational basis before measurement.

Qubit Probability One option for the output of a QNN is to convert counts to a probability of a qubit being in a certain state, either $|0\rangle$ or $|1\rangle$ in the case of measurement in the computational basis. This is can also be extended to a vector of probabilities for each qubit. For example, if one is interested in the probability that qubit i is measured in state $|0\rangle$, then a qubit probability measurement would output the estimate of the scalar value

$$prob_j(0) = Tr[\rho_j \, |0\rangle\langle 0|] \tag{13}$$

where ρ_j is the reduced density operator describing qubit j and Tr is the trace function. It is an estimate because using counts of measurement outcomes entails sampling from the distribution defined by the quantum state. This estimate becomes more accurate, in expectation, as the number of samples increases.

Similarly to the single-qubit case, if one was interested in the individual probability for each qubit in a set J of qubits to be measured in the $|1\rangle$ state, then the output would be the vector

$$[prob_{j \in J}(1), ...]. \tag{14}$$

Probability Threshold QNN-Gen's *probability threshold* function implements a step function that is dependent on a threshold value, t, and the probability of a qubit, j, being in a particular state, s, upon measurement. The output given by *probability threshold* is

$$\begin{cases} l_+ & prob_j(s) \geq t \\ l_- & prob_j(s) < t \end{cases} \tag{15}$$

where $\{l_+, l_-\}$ is the set of corresponding labels which may be specified by the user. Like *qubit probability*, the output of *probability threshold* can be a scalar for the case of measurement on one qubit or a vector for the case of multiple qubits.

Expectation of an Observable The expectation value of an observable, $\hat{O}$, given a particular quantum state, $|\psi\rangle$, is [24]

$$\mathbb{E}_\psi[\hat{O}] = \langle\psi| \, \hat{O} \, |\psi\rangle \, . \tag{16}$$

The expectation of an observable can be a useful output layer for a QNN as it outputs a scalar value and, as shown in (21), allows one to encapsulate all the subcomponents of a QNN in a convenient mathematical expression. One may

want the output of a QNN to be the expectation of an observable for a particular qubit, j, which may be entangled in a larger Hilbert space. In this case, we can use the reduced density operator for qubit j, ρ_j, to calculate the expectation of the observable by reformulating (16) for density matrices,

$$\langle\psi|\,\hat{O}\,|\psi\rangle = Tr[\langle\psi|\,\hat{O}\,|\psi\rangle] \tag{17}$$

$$= Tr[|\psi\rangle\,\langle\psi|\,\hat{O}] \tag{18}$$

$$= Tr[\rho\hat{O}] = \mathbb{E}_\rho[\hat{O}] \tag{19}$$

wherein the second line we use the cyclic property of the trace function, and the definition of the density matrix for a pure state is used in the third line. Then, for ρ_j we have

$$\mathbb{E}_{\rho_j}[\hat{O}] = Tr[\rho_j\hat{O}]. \tag{20}$$

3.4 Summary

To define any QNN, one must specify the encoding function, $E(x)$, the unitary that implements it, S_x, the model, $U(\vec{\theta})$, and the measurement output (and perhaps further classical post-processing). For example, we can write a QNN on N-qubits with an output measurement of the expectation of an operator $\hat{O}$, as

$$\langle 0|^{\otimes N} S_x^\dagger U(\vec{\theta})^\dagger \hat{O} U(\vec{\theta}) S_x\,|0\rangle^{\otimes N}\,. \tag{21}$$

The learning task is then to find the best values for $\vec{\theta}$. This is achieved by using the quantum circuit to get outputs and optimizing $\vec{\theta}$ with respect to some loss function on a classical computer. We note that this is an interesting area of research in its own right, but we do not devote much discussion to it in this paper as it is not the focus of QNN-Gen in its present form.

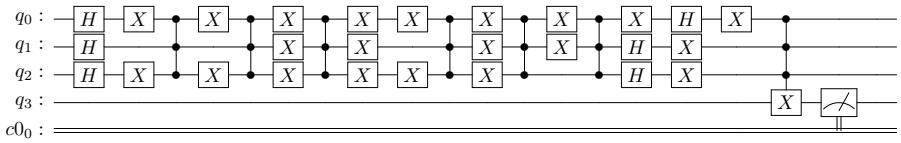

Fig. 3. A binary perceptron model created using QNN-Gen.

4 Example Use Cases

We present two examples to demonstrate both the ease-of-use and configurability of QNN-Gen.

```
encoder = BinaryPhaseEncoding()
model = BinaryPerceptron()
measurement = model.default_measurement()
perceptron = combine(data, encoder, model,
                     measurement)
```

Fig. 4. Code to create a binary perceptron in QNN-Gen.

The code example in Figure 4 illustrates how simple creating QNN models can be using QNN-Gen, as just four lines of code are sufficient to create the circuit. Moreover, one can see that no arguments are needed to pass in the initialization of the encoder, model, and measurement objects. It is a deliberate design that the functionality of QNN-Gen provides reasonable default values for arguments whenever possible. The intention behind this design is to decrease the chance of user error. The figure also shows that models have a *default measurement* function, as it is often the case that the structure of the model implies the sensible set of output qubits and measurement transformations. The starting weights for the *binary perceptron* model may specified explicitly as an argument, but if none are passed, as in the code example, QNN-Gen will initialize the model with random weights. Figure 3 shows the circuit generated by this code for input data $\vec{x} = [1, 1, -1, 1, 1, 1, 1, -1]^T$.

```
encoder = AngleEncoding(gate=Gate.RX)
model = TreeTensorNetwork(rotation_gate=Gate.RZ,
                          entangling_gate=Gate.CY)
X_obs = Observable.X()
measurement = Expectation(qubits=4, observable=X_obs)
ttn = combine(data, encoder, model, measurement)
```

Fig. 5. Code to create a tree tensor network in QNN-Gen.

In Figure 5 the configurability of QNN-Gen is demonstrated by passing arguments to override default options. The code creates a *tree tensor network* using an *angle encoding* method, and measures the expectation value of the Pauli X observable on qubit 4. The gate argument for *angle encoding* is set to RX, overriding the default option, RY. Likewise, for the *tree tensor network* the *rotation gate* and *entangling gate* are set to RZ and CY, overriding the respective default values RY and CX. (We note that the argument choices are merely to demonstrate configurability and are not necessarily the best choices for performance.)

The code also uses a couple of utility classes included in QNN-Gen that have be written for the user's convenience: the *Gate* class provides easy access to Qiskit gate objects, and the *Observable* class can be used to implement measurements in different bases. Figure 5 exemplifies how possibly straight-forward

it is for a user to modify the default structures and create new QNN circuits. The circuit generated by this code for an 8-dimensional data vector with elements evenly spaced in the interval $[0, 0.5]$ is show in Figure 2.

As discussed earlier, we have designed QNN-Gen to ensure modularity in order to maintain ease-of-use and facilitate rapid development. Leveraging this modular design, users may write their own classes for an encoder, model, or measurement. This is made easier through the use of abstract classes. For example, *AngleEncoding* inherits from the abstract class *Encode*. Likewise *BinaryPerceptron* inherits from the abstract *Model* class, and so on. The abstract base classes make it clear how to write classes for new encoding methods, models, and measurements, that will allow new components to work with the existing functionality.

5 Conclusion and future work

In this work, we introduce a novel open-source framework for supporting rapid development and innovation in the growing field of QNN research. To this end, we have shown that QNN-Gen can be used to quickly generate quantum neural networks by specifying a small number of parameters. By prioritizing ease-of-use, we hope that the design of QNN-Gen helps researchers and practitioners by facilitating the creation of novel QNNs and in comparing models.

QNN-Gen is in a beta phase of release, and we will continue to add more methods for encoding methods, models, and measurement transformations. The design of QNN-Gen is inherently modular, making such additions relatively straightforward regarding the integration of new software into QNN-Gen. We welcome the community to take advantage of the modular design and contribute and share new methods and models that they create.

To train the quantum circuits generated with QNN-Gen, one option is for users to use the functionality provided by the PennyLane framework. PennyLane allows users to convert Qiskit circuits, such as those created with QNN-Gen, to Quantum Node objects in their framework, which then inherit all of the gradient and training functionality of PennyLane.

For future versions of QNN-Gen we plan on further developing functionality on the classical side of the quantum-classical hybrid scheme. This includes methods for taking gradients of quantum circuits, a library of default loss functions and optimization methods, and more. We plan on developing QNN-Gen into a comprehensive, end-to-end open source framework for quantum-classical machine learning algorithms.

Acknowledgment

Collin thanks Dr. Robert Parrish for a helpful discussion.

References

1. QNN-Gen Git Repository. https://github.com/Farquhar13/QNN-Gen, accessed: May. 22, 2020
2. Aaronson, S.: Read the fine print. Nature Physics **11**(4), 291–293 (2015)
3. et al., H.A.: Qiskit: An open-source framework for quantum computing (2019). https://doi.org/10.5281/zenodo.2562110
4. et al., M.B.: Tensorflow quantum: A software framework for quantum machine learning (2020)
5. et al., V.B.: Pennylane: Automatic differentiation of hybrid quantum-classical computations (2018)
6. Altaisky, M.: Quantum neural network. arXiv preprint quant-ph/0107012 (2001)
7. Beer, K., Bondarenko, D., Farrelly, T., Osborne, T.J., Salzmann, R., Scheiermann, D., Wolf, R.: Training deep quantum neural networks. Nature communications **11**(1), 1–6 (2020)
8. Biamonte, J., Wittek, P., Pancotti, N., Rebentrost, P., Wiebe, N., Lloyd, S.: Quantum machine learning. Nature **549**(7671), 195–202 (2017)
9. Dunjko, V., Briegel, H.J.: Machine learning & artificial intelligence in the quantum domain (2017)
10. Farhi, E., Neven, H.: Classification with quantum neural networks on near term processors (2018)
11. Feynman, R.P.: Simulating physics with computers. Int. J. Theor. Phys **21**(6/7) (1999)
12. Grant, E., Benedetti, M., Cao, S., Hallam, A., Lockhart, J., Stojevic, V., Green, A.G., Severini, S.: Hierarchical quantum classifiers. npj Quantum Information **4**(1) (Dec 2018)
13. Grover, L.K.: A fast quantum mechanical algorithm for database search (1996)
14. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Physical Review Letters **103**(15) (Oct 2009). https://doi.org/10.1103/physrevlett.103.150502, http://dx.doi.org/10.1103/PhysRevLett.103.150502
15. Havlíček, V., Córcoles, A.D., Temme, K., Harrow, A.W., Kandala, A., Chow, J.M., Gambetta, J.M.: Supervised learning with quantum-enhanced feature spaces. Nature **567**(7747), 209–212 (Mar 2019)
16. Jagannath, J., Polosky, N., Jagannath, A., Restuccia, F., Melodia, T.: Machine Learning for Wireless Communications in the Internet of Things: A Comprehensive Survey. Ad Hoc Networks (Elseier) **93**, 101913 (2019)
17. LaRose, R., Coyle, B.: Robust data encodings for quantum classifiers (2020)
18. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
19. LeCun, Y., Cortes, C.: MNIST handwritten digit database (2010), http://yann.lecun.com/exdb/mnist/
20. Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., Alsaadi, F.E.: A survey of deep neural network architectures and their applications. Neurocomputing **234**, 11 – 26 (2017)
21. McClean, J.R., Romero, J., Babbush, R., Aspuru-Guzik, A.: The theory of variational hybrid quantum-classical algorithms. New Journal of Physics **18**(2), 023023 (2016)
22. Mitarai, K., Negoro, M., Kitagawa, M., Fujii, K.: Quantum circuit learning. Physical Review A **98**(3) (Sep 2018)

23. Mitchell, T.: Machine learning. McGraw-hill New York (1997)
24. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information. Cambridge University Press, second edn. (2010)
25. Preskill, J.: Quantum computing in the nisq era and beyond. Quantum **2**, 79 (Aug 2018). https://doi.org/10.22331/q-2018-08-06-79, http://dx.doi.org/10.22331/q-2018-08-06-79
26. Rieffel, E., Polak, W.: An introduction to quantum computing for non-physicists. ACM Computing Surveys (CSUR) **32**(3), 300–335 (2000)
27. Rosenblatt, F.F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review **65 6**, 386–408 (1958)
28. Schuld, M., Bocharov, A., Svore, K.M., Wiebe, N.: Circuit-centric quantum classifiers. Physical Review A **101**(3) (Mar 2020)
29. Schuld, M., Killoran, N.: Quantum machine learning in feature hilbert spaces. Physical Review Letters **122**(4) (Feb 2019). https://doi.org/10.1103/physrevlett.122.040504, http://dx.doi.org/10.1103/PhysRevLett.122.040504
30. Schuld, M., Petruccione, F.: Supervised learning with quantum computers, vol. 17. Springer (2018)
31. Schuld, M., Sinayskiy, I., Petruccione, F.: The quest for a quantum neural network. Quantum Information Processing **13**(11), 2567–2586 (Aug 2014)
32. Shi, Y.Y., Duan, L.M., Vidal, G.: Classical simulation of quantum many-body systems with a tree tensor network. Physical Review A **74**(2) (Aug 2006)
33. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (Oct 1997). https://doi.org/10.1137/s0097539795293172, http://dx.doi.org/10.1137/S0097539795293172
34. Stoudenmire, E., Schwab, D.J.: Supervised learning with tensor networks. In: Lee, D.D., Sugiyama, M., Luxburg, U.V., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29, pp. 4799–4807. Curran Associates, Inc. (2016)
35. Tacchino, F., Macchiavello, C., Gerace, D., Bajoni, D.: An artificial neuron implemented on an actual quantum processor. npj Quantum Information **5**(1) (Mar 2019)
36. Vidal, G.: Class of quantum many-body states that can be efficiently simulated. Phys. Rev. Lett. **101**, 110501 (Sep 2008)
37. Wittek, P.: Quantum machine learning: what quantum computing means to data mining. Academic Press (2014)
38. Wright, L.G., McMahon, P.L.: The capacity of quantum neural networks (2019)