Quantum Neural Networks

with a 2x3x2 simulated example in Qutip

Alexander Heilman

Northeastern University
Boston, Massachusetts

December 17, 2022

ABSTRACT

Classical neural networks have shown great efficacy in many tasks, suggesting an analogous quantum neural network may have some utility. Furthermore, in the Noisy Intermediate-Scale Quantum era (NISQ), algorithms that act on small states are desired due to hardware limitations. Quantum Neural Networks, as proposed in Beer et al. (2020), can be used to learn arbitrary, completely positive, linear maps (i.e. quantum operations) between quantum ensembles, given an appropriate data set to train on, and which require relatively small quantum states to be acted upon at a time.

Key words. Quantum Neural Networks, QuTip

Introduction & Motivation

Classical neural networks are seemingly ubiquitous in the modern research landscape due to an apparent efficacy in many predictive and generative roles.

Thus, with quantum computers on the bleeding edge of research, some have sought after a quantum analogy to the classical neural network architecture. One proposed framework is that in Kerstin Beer, et al's *Training Deep Quantum Neural Networks*, that which is discussed in the content below.

But first, a brief background passage will be included in an attempt to justify the following architecture, both in terms of generality and feasibility.

Quantum Operations

The most general evolution a quantum system may undergo is that of a completely positive linear map. This includes not only the unitary evolution of quantum pure states but also the evolution of a quantum subsystem or *principle system* interacting and evolving in accordance with the unitary evolution of some grander quantum system or *environment*, which may be considered closed itself (if large enough; since density matrices may always be purified in some larger space).

This evolution of the subsystem's state in accordance with some environment then is representable as a three step process: first a tensor product of the subsystem's state space and the environment's state space is taken; then some unitary operation acts on this tensor product space; and finally, the subsystem's state may be regained by restriction back to it's state space via a partial trace over the environment's state space.

$$\rho \to \rho \otimes \rho_{Env.} \to U(\rho \otimes \rho_{Env.})U^{\dagger} \to \mathrm{Tr}_{Env.}\left[U(\rho \otimes \rho_{Env.})U^{\dagger}\right] = \rho'$$

Thus, a most general framework for an arbitrary, unknown quantum operation must allow for the total action of tensoring with arbitrary sized environment's state, unitary evolution in this larger space, and restriction back down to the subsystem.

If one could construct an arbitrarily wide quantum system though (i.e. tensor with an arbitrarily large environment with high fidelity), this should only need be done once, in which state a large unitary would act and the restriction back down would be made. This should render a layered structure unnecessary. However, given modern hardware limitations, large product states are hard to hold onto, with very short coherence times, and also are hard to act on with large enough unitaries (with these generally broken down into swap gates and relatively local actions on 2 or 3 qubits). Hence, modern limitations justify a model that requires the handling of smaller numbers of elements at a time, which justifies a deeper circuit that can simulate restrictions from larger environments by iterative tensorings and restrictions.

1. General Mathematical Structure

With the above considerations in mind justifying a deep network with small width layers (where we consider only a few qubits at a time) and a requirement that we consider these larger evolution schemes at all (in hopes of generality), we now detail the implementation of a Quantum Neural Network (QNN) as proposed in Beer, et al.

In conjunction with the description of classical neural networks, we first introduce the relevant types of data; then consider what a forward pass of the network is; define some cost function by which we may quantitatively judge the network; and finally discuss how to train the network to improve the cost.

1.1. Data

Data will be provided for the training of the network via a set of arbitrary states (inputs), and the set of these same states after having some common unitary action act upon them (outputs). For pure states, the training data may take the following form:

$$\text{Training Data: } \{(|\psi_i\rangle, V|\psi_i\rangle) \mid 1 \leq i \leq N\}$$

But in general, data of some well defined operation may be given, described in the density matrix formalism, as below:

$$\text{Training Data: } \{(\rho_i, \varepsilon(\rho_i)) \mid 1 \leq i \leq N\}$$

where $\varepsilon(\rho)$ represents some general operation, on the density matrix ρ, satisfying the physical condition of being a linear completely positive map.

Note: Why the Partial Trace?

The partial trace returns a density matrix describing a subsystem of a larger quantum system. The returned state essentially represents the subsystem's quantum state if the observer 'forgets' about those subsystem's traced over, or, equivalently, the state relevant to an observer who can only interact with that subsystem left after the trace.

Example As a simple example, consider the two qubit density matrix:

$$\rho = \frac{1}{2}\left(|00\rangle\langle 00| + |10\rangle\langle 10|\right)$$

Now, for reasons that will be conducive to the example, we may rewrite the above state as the tensor product state:

$$\rho = \frac{1}{2}(|0\rangle\langle 0| + |1\rangle\langle 1|)_1 \otimes (|0\rangle\langle 0|)_2$$

with the tensor product structure explicitly labeled as subscripts. Now, we take the partial trace over the first qubit's state space, as follows:

$$\mathrm{Tr}_1(\rho) = \langle 0|_1\rho|0\rangle_1 + \langle 1|_1\rho|1\rangle_1 = |0\rangle\langle 0|$$

where we may now omit the subscript in the output since the result is a one-qubit state. Note that the state need not be separable to perform the partial trace.

1.2. Forward Pass

We now define the action of a QNN as defined in Beer et al. (2020). The overall action of the network is composed of layer-to-layer transition maps ϵ^ℓ for each layer ℓ s.t. $in \leq \ell \leq out$. Each layer may have a different number of nodes, or associated qubits m_ℓ, so at the end of each layer we have an m_ℓ qubit state represented as $\rho_{\ell+1}$. Explicitly, the ℓ-th layer's transition map takes the form:

$$\epsilon^\ell\left(\rho_{\ell-1}\right) = \mathrm{Tr}_{\ell-1}\left[\left(\prod_{m=1}^{m_\ell} U_m^\ell\right)\left(\left(|0\rangle^{\otimes m_\ell}\langle 0|^{\otimes m_\ell}\right)_\ell \otimes \rho_{\ell-1}\right)\left(\prod_{m=m_\ell}^{1} U_m^{\ell\dagger}\right)\right] = \rho_\ell$$

This layer-to-layer map allows us to define the total action of some QNN as a iterative composition of such maps. Hence, a total circuit of L layers returns ρ_{out}, defined below, for some given input state ρ_{in}.

$$\rho_{out} = \epsilon^{out}\left(\epsilon^L\left(\epsilon^{L-1}\left(...\epsilon^1\left(\rho_{in}\right)...\right)\right)\right)$$

Note that while the width of each intermediate layer is arbitrary, for data sets based on purely unitary evolution the state space of the input and output layers must be equal in size. That is, while the model's architecture is arbitrary in the intermediate layers, the input and output dimensions are dictated by the form of the data.

1.2.1. Forward Pass: Step-by-Step

To elucidate the somewhat dense form of the layer-to-layer transition map, we may further break each layer's action into three effective steps (reminiscent of the form of the general quantum operation given in the introduction).

For each layer ℓ, the corresponding transition map to the next layer $\varepsilon^\ell : \rho_{\ell-1} \to \rho_\ell$ may be broken into the following steps:

1. The next layer's m_ℓ qubits are prepared in the initial state $|0\rangle^{\otimes m}\langle 0|_\ell^{\otimes m}$ and tensor producted with the previous layer's output $\rho_{\ell-1}$.
$$\rho'_\ell = \left(|0\rangle^{\otimes m_\ell}\langle 0|^{\otimes m_\ell}\right)_\ell \otimes \rho_{\ell-1}$$
2. The ℓ-th layer's m_ℓ associated unitary matrices U_m^ℓ are applied to this tensor product state (from top to bottom).
$$\rho''_\ell = \left(\prod_{m=1}^{m_\ell} U_m^\ell\right)\left(\rho'_\ell\right)\left(\prod_{m=m_\ell}^{1} U_m^{\ell\dagger}\right)$$
3. The partial trace over the $(\ell-1)$th layer's Hilbert space is taken, resulting in the output state ρ_ℓ of the ℓ-th layer.
$$\rho_\ell = \mathrm{Tr}_{\ell-1}[\rho''_\ell]$$

Note that, as defined in Beer et al. (2020), the constituent unitary U_ℓ^m acts only on the state space of the m-th qubit in the ℓ-th layer as well as all the qubits of the $\ell-1$-th layer, and hence has a dimension of $2^{m_{\ell-1}+1} \times 2^{m_{\ell-1}+1}$ (where it acts as the identity on the otherwise excluded state space).

Furthermore, since these unitaries won't necessarily commute in general, the order must be specified, where the convention adopted by the authors is to apply the unitaries from 'top to bottom' in that first U_ℓ^1 is applied, then U_ℓ^2, etc.

1.3. Cost

The metric by which we will judge the performance of the network on the training data is the cost, here taken as the average fidelity between the networks output state and the corresponding state given in training and explicitly defined as:

$$C = \frac{1}{N}\sum_{i=1}^{N}\langle\psi_i^{out}|\rho_{out}|\psi_i^{out}\rangle$$

Note that this cost function is only applicable for training data based on pure states, for which the fidelity takes an especially nice form.

For data based on mixed states, we may replace the above with an averaged fidelity between output and target states of the form:

$$C = \frac{1}{N}\sum_{i=1}^{N}\left(\mathrm{Tr}\left[\sqrt{\sqrt{\rho_i}\rho_i^{out}\sqrt{\rho_i}}\right]\right)^2$$

which, again, is just the averaged fidelity between the given training output and their corresponding outputs from the network.

Fidelity is a natural choice for a cost function here since it is a metric already well known within quantum information science as an effective way to measure how similar two density matrices are. In fact, in the case of pure states, the fidelity reduces to the inner product between the states squared.

1.4. Training

We now wish to maximize the previously defined cost function (which has a maximum value of 1). This may be accomplished through training.

Training may be performed by evolving each unitary via the following map:

$$U_m^\ell \to e^{-\epsilon K_m^\ell} U_m^\ell$$

which is parameterized by the step size ϵ, and where K_ℓ^m is derived from the derivative of the cost function and takes the following form:

$$K_m^\ell = \eta \frac{2^{m_{\ell-1}}}{N} \sum_{i=1}^{N} \mathrm{Tr}_{\neg\ell,m}\left[\left(\prod_{n=1}^{m} U_n^\ell\right)\left(\left(|0\rangle^{\otimes m_\ell}\langle 0|^{\otimes m_\ell}\right)_\ell \otimes \rho_i^{\ell-1}\right)\left(\prod_{n=m}^{1} U_n^{\ell\dagger}\right), \left(\prod_{n=m_\ell}^{m+1} U_n^{\ell\dagger}\right)\left(\sigma_i^\ell \otimes \mathbb{I}_{\ell-1}\right)\left(\prod_{n=m+1}^{m_\ell} U_n^\ell\right)\right]$$

where the square brackets denote a commutator and $\sigma_i^\ell = \mathcal{F}^{\ell+1}(...\mathcal{F}^{out}(\rho_i^{out})...)$ is the adjoint channel to the layer-to-layer transition map ϵ^ℓ for layer ℓ. Note that the ρ_i^{out} used for the training matrix is that provided in the training data.

Note: Discrepancy with Original Paper

In the original paper, the update matrix acts on it's unitary, much as a generator of a Lie algebra acts on it's associated group elements, as below:

$$U \to e^{iK} U$$

a convention which we've neglected to adopted here. However, by this definition K must be Hermitian for U to remain unitary; and in the original paper, K, as defined, is skew-Hermitian. This requires us to either add a complex factor of i to K or to remove the factor of i from the map. We choose to do the latter here. In fact, upon analysis of their associated code, the authors neglect both factors of i and hence update according to the map $U \to e^{-K} U$ as well.

Proof It is well known that products of unitaries are unitary and that the generators of unitaries are Hermitian. However, by the construction of K (according to the same authors) K is the commutator of two density matrices, which are Hermitian. However the commutator of two Hermitians is skew-Hermitian.

$$[A, B]^\dagger = (AB - BA)^\dagger = B^\dagger A^\dagger - A^\dagger B^\dagger = -[A, B]$$

and hence, by the defined map, K must also be multiplied by an extra factor of the complex unit i to make it Hermitian.

The derivation of the above form of the training matrix is rather involved and beyond the scope of this overview. For more information, see the supplementary resources of Beer et al. (2020).

1.4.1. Adjoint Layers

Note that the training matrices require the computation of the network's associated adjoint layers, which are similar to the backward pass of a classical network.

The adjoint map for layer $\ell + 1$ is defined as the map taking a state in the $\ell + 1$-th layer to the corresponding state in the ℓ-th layer (effectively undoing the action of transition map ε). Explicitly, it takes the following form:

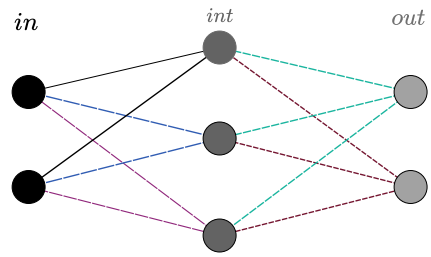
$$\mathcal{F}^{\ell+1}(\rho_{\ell+1}) = \mathrm{Tr}_{\ell+1}\left[\left((|0\rangle^{\otimes m_{\ell+1}}\langle 0|^{\otimes m_{\ell+1}})_{\ell+1} \otimes \mathbb{I}_\ell\right)\left(\prod_{m=1}^{m_{\ell+1}} U_m^{\ell+1\dagger}\right)(\rho_{\ell+1} \otimes \mathbb{I}_\ell)\left(\prod_{m=m_{\ell+1}}^{1} U_n^{\ell+1}\right)\right] = \rho_\ell$$

where $\mathbb{I}_\ell$ is the 2^{m_ℓ} dimensional identity, apparently unnormalized to trace one.

2. Explicit Example: 2 × 3 × 2 QNN

When first encountering novel mathematical structures, it is generally instructive to consider a simple example in an attempt to build some intuition for such structures. Hence, we now restrict our discussion to a specific, small QNN (but not too small to be uninteresting); for this purpose we choose a three layer network with a two-qubit input and output layer, and one three-qubit intermediate layer.

This is a reasonable size network to consider as an example, in that it consists of wider intermediate layers than the input and output states, so it can be taken to consider an environment larger than the given state, but it is small enough to explicitly write out the previously described operations in terms of it's respective components, without being entirely unwieldy.



$$\mathrm{Tr}_{int}\left[U_2^{out}U_1^{out}\left(\mathrm{Tr}_{in}\left[U_3^{int}U_2^{int}U_1^{int}(\rho_{in} \otimes |000\rangle\langle 000|_{int})U_1^{int\dagger}U_2^{int\dagger}U_3^{int\dagger}\right] \otimes |00\rangle\langle 00|_{out}\right)U_1^{out\dagger}U_2^{out\dagger}\right]$$

Fig. 1. *Diagrammatic Representation of 2x3x2 QNN:* Under the diagram representing the network structure, the output state ρ_{out} for some given ρ_{in} is displayed after it's forward pass through the network. The constituent unitaries are color coded to match edges connected to the nodes they act upon.

Since there are $3_{(int)} + 2_{(out)} = 5$ qubits beyond the input layer, there are 5 constituent unitaries composing the QNN: with U_1^1, U_2^1, and U_3^1 for the intermediate layer; and with U_1^{out} and U_2^{out} for the final layer. Unitaries for the intermediate layer U_m^1 then act non-trivially on a state space of dimension $2^{2+1} \times 2^{2+1}$ but are tensored with identity in the rest, resulting in a matrix of dimension $2^{2+3} \times 2^{2+3}$. Similarly, unitaries for the output layer U_m^1 then act on a state space of dimension $2^{3+1} \times 2^{3+1}$ but are tensored with identity in the rest, resulting in a matrix of dimension $2^{3+2} \times 2^{3+2}$.

These 5 unitaries then require us to construct 5 training matrices K_m^ℓ, each corresponding uniquely to one of the above unitaries.

2.1. Forward Pass: $2 \times 3 \times 2$

The forward pass of the 2x3x2 QNN is now able to be printed explicitly as the following map :

$$\rho_{in} \to \rho_{out} =$$

$$\mathrm{Tr}_1\Big[U_2^2 U_1^2\Big(\mathrm{Tr}_{in}\left[U_3^1 U_2^1 U_1^1 \left(\rho_{in} \otimes |000\rangle\langle 000|_1\right) U_1^{1\dagger} U_2^{1\dagger} U_3^{1\dagger}\right] \otimes |00\rangle\langle 00|_2\Big) U_1^{2\dagger} U_2^{2\dagger}\Big]$$

(where U^{out} is used interchangeably with U^2). Again, note that the constituent unitaries U_m^ℓ act only on the entire last layer's state space as well as the m-th qubit of the current ℓ-th layer. So, for example, U_2^1 acts on the input space and the second qubit of the intermediate layer, and acts as identity on the other two qubits of the intermediate layer; while U_1^1 acts on the input space and the first qubit of the intermediate layer, and then acts as identity on the rest, etc.

2.2. Training: $2 \times 3 \times 2$

We now construct the training matrices for enough of the unitaries for the pattern to be apparent (in the sake of brevity). Note that the for the output layer's unitaries, the adjoint layer is effectively the identity map, and so ρ_{out} appear explicity as opposed to an element of the adjoint channel.

For the intermediate layer's unitaries, we then have the corresponding training matrices:

$$K_1^1 = \eta \frac{2^2}{N} \sum_{i=1}^{N} \mathrm{Tr}_{2,3_{int}} \left[U_1^1 \left(\rho_i^{in} \otimes |000\rangle\langle 000|_1\right) U_1^{1\dagger}, U_2^{1\dagger} U_3^{1\dagger} \left(\mathbb{I}_{2^2} \otimes \sigma_i^1\right) U_3^1 U_2^1\right]$$

$$K_2^1 = \eta \frac{2^2}{N} \sum_{i=1}^{N} \mathrm{Tr}_{1,3_{int}} \left[U_2^1 U_1^1 \left(\rho_i^{in} \otimes |000\rangle\langle 000|_1\right) U_1^{1\dagger} U_2^{1\dagger}, U_3^{1\dagger} \left(\mathbb{I}_{2^2} \otimes \sigma_i^1\right) U_3^1\right]$$

$$K_3^1 = \eta \frac{2^2}{N} \sum_{i=1}^{N} \mathrm{Tr}_{1,2_{int}} \left[U_3^1 U_2^1 U_1^1 \left(\rho_i^{in} \otimes |000\rangle\langle 000|_1\right) U_1^{1\dagger} U_2^{1\dagger} U_3^{1\dagger}, \left(\mathbb{I}_{2^2} \otimes \sigma_i^1\right)\right]$$

And then for the output layer's unitaries, we have the corresponding training matrices:

$$K_1^{out} = \eta \frac{2^3}{N} \sum_{i=1}^{N} \mathrm{Tr}_{2_{out}} \left[U_1^{out} \left(\rho_i^{int} \otimes |00\rangle\langle 00|_{out}\right) U_1^{out\dagger}, U_2^{out\dagger} \left(\mathbb{I}_{2^3} \otimes \rho_i^{out}\right) U_2^{out}\right]$$

$$K_1^{out} = \eta \frac{2^3}{N} \sum_{i=1}^{N} \mathrm{Tr}_{1_{out}} \left[U_2^{out} U_1^{out} \left(\rho_i^{int} \otimes |00\rangle\langle 00|_{out}\right) U_1^{out\dagger} U_2^{out\dagger}, \left(\mathbb{I}_{2^3} \otimes \rho_i^{out}\right)\right]$$

where $\rho_i^{int} = \varepsilon^1(\rho_i^{in})$, $\rho_i^{out} = \varepsilon^{out}(\varepsilon^1(\rho_i^{in}))$, and $\sigma_i^1 = \mathcal{F}^{out}(\rho_i^{out})$, with ρ_i^{out} again being from the training data.

2.2.1. Adjoint Layer: $2 \times 3 \times 2$

For each intermediate layer, we must construct the associated adjoint layer to implement the training matrices, as defined above.

Since there is only one intermediate layer for the given structure, there is only one relevant adjoint map, analogous to a backward pass from the output layer to the intermediate layer. We now define this adjoint map as follows:

$$\mathcal{F}^{out}(\rho_{out}) =$$

$$\mathrm{Tr}_{out}\left[\left(|00\rangle\langle 00|_{out} \otimes \mathbb{I}_{2^3}\right) U_1^{out\dagger} U_2^{out\dagger} (\rho_{out} \otimes \mathbb{I}_{2^3}) U_2^{out} U_1^{out}\right]$$

$$= \rho_{int}$$

3. QuTip Simulation: $2 \times 3 \times 2$ QNN

With the explicit structures of the example $2 \times 3 \times 2$ QNN now given, we shall demonstrate the performance of this example model on a set of data by simulating it's learning process for some set of data. To accomplish this simulation, we choose QuTip, a python module that is designed for general quantum information science applications.

3.1. Implementation in QuTip

The implementation require a few sets of functions and routines to be defined: first, a small set of helper functions; then, the forward pass function; the cost function; a way to define some set of data; and the adjoint layer and training matrices.

3.1.1. Helper Function

To implement the example network in QuTip, we define a few helper functions. The first, and most important, is that which allows us to properly size the unitaries U_m^ℓ, such that the unitaries act on the appropriate subspaces of the total system.
This operation is performed by the swapper function, which generates a permutation index applicable to QuTip's permutation method for quantum objects.

```
def unit_sizer(U,n,n_last=2,n_next=3):
    assert U.shape[0] == 2**(n_last+1)
    assert n>0 and n<=n_next
    diml = [2 for i in range(n_last+1)]
    U.dims = [diml,diml]
    Id = qt.qeye(2)
    Idl = [Id for i in range(n_next-1)]
    un = qt.tensor(U,*Idl)
    if n>1:
        un = qt.tensor(U,*Idl)
        un = swapper(un, n_last, (n_last-1)+n)
    return un
```

Note: Tensor Structure of Unitaries

Recall that the unitary U_m^ℓ acts on the $(\ell-1)$-th layer's entire subspace but only the m-th qubit of the ℓ-th layer.

Since QuTip only allows us to take the tensor product of the composite objects first, which results in the first structure below (before swap). Then, we must swap the unitary's last subspace of operation to the relevant qubit's subspace, i.e. the $(m_{\ell-1} + m)$-th subspace.

Essentially, we need to be able to swap the the tensor structure of the unitary after tensoring it with identity, as follows:

$$U^{(1)} \otimes U^{(2)} \otimes \ldots \otimes U^{(m_{\ell-1})} \otimes U^{(m_{ell-1}+1)} \otimes \mathbb{I} \otimes \ldots \otimes \mathbb{I}$$

$$\downarrow \; Swap_m$$

$$U^{(1)} \otimes U^{(2)} \otimes \ldots \otimes U^{(m_{\ell-1})} \otimes \mathbb{I} \otimes \ldots \otimes \mathbb{I} \otimes U^{(m_{\ell-1}+m)} \otimes \mathbb{I} \otimes \ldots \otimes \mathbb{I}$$

Dims in QuTip The composite tensor structure of an object in QuTip is encoded in it's dims method, which returns a list of lists describing the dimensionality, and implicit order, of its subsystems.

This is relevant since these dims must reflect an n-qubit operation as opposed to one operation on a 2^n dimensional space in order for us to transform the matrix representation accordingly.

This then allows us to define unitaries with the appropriate tensor structure such that they may be applied appropriately to the intermediate states.

3.1.2. Data

For the simulation of the network, a set of relevant training and test data needed to be generated. This was accomplished by forming a list of two lists: a list of randomly generated density matrices, using QuTip's native random density matrix function; and a list of a randomly generated unitary applied to the corresponding density matrix.

$$\text{Data Structure: } \{(\rho_i, V\rho_i V^\dagger \mid 1 \leq i \leq N\}$$

Note that the same unitary was used for all the training data, except for in the noisy section, for which a new random unitary acting on an additional set of random density matrices was mixed in with the training data at some set dilution.

In general, a larger set of training data than test data was used since both scaled similarly in simulation time and a small set of test data was still relevant given it's independence.

3.1.3. Forward Pass

The forward pass function was a straight-forward function taking as input the set of training data inputs, before the unkown unitary operation, and 5 unitaries: 3 of dimension $2^3 \times 2^3$, and 2 of dimension $2^4 \times 2^4$. These unitaries were first converted into the appropriate tensor structure via the helper function discussed above.

Then, for each layer: the appropriate intermediate states were formed (*pin1, pint1* below); the corresponding unitaries were applied from top to bottom (*pin2, pint2*); and the relevant partial trace was taken to leave the layer's output state (*pint, pout*).

```
def forward(U11,U12,U13,U21,U22,pin):
    U11 = unit_sizer(U11,1,n_last=2,n_next=3)
    U12 = unit_sizer(U12,2,n_last=2,n_next=3)
    U13 = unit_sizer(U13,3,n_last=2,n_next=3)
    U21 = unit_sizer(U21,1, n_last = 3, n_next=2)
    U22 = unit_sizer(U22,2, n_last = 3, n_next=2)
    zro = qt.fock_dm(2, 0)
    pin1 = qt.tensor(pin,zro,zro,zro)
    pin2 = U13*U12*U11*pin1*U11.dag()*U12.dag()*U13.dag()
    pint = pin2.ptrace([2,3,4])
    pint1 = qt.tensor(pint,zro,zro)
    pint2 = U22*U21*pint1*U21.dag()*U22.dag()
    pout = pint2.ptrace([3,4])
    return pin,pint,pout
```

Note that QuTip's partial trace method for quantum objects takes as argument the list of indices of subspaces *to keep* after the partial trace is performed.

3.1.4. Cost

Cost was simply defined using an averaged fidelity, summing over QuTip's natively defined fidelity applied to training and output pairs, then dividing the sum by the length of the data list. A cost was calculated for both a set of training data used in the formulation of the update matrices, as well as an independent set of test data which the update process was blind to.

3.1.5. Adjoint Layer & Training Routine

The functions implemented for the adjoint layer and training matrices had a similar structure to the forward pass, with arguments representing, the input states to each layer, the adjoint states to each layer, and the current constituent unitaries.

Training was done through an iterative update of the constituent unitaries via the map discussed previously. For each epoch, the training matrices are derived from the entire training data set for the current unitaries. These are then updated at the beginning of the next epoch, as below.

```
K11=K11(pins, sigmas, U11,U12,U13)
K12=K12(pins, sigmas, U11,U12,U13)
K13=K13(pins, sigmas, U11,U12,U13)
K21=K21(pints,train_data[outs],U21,U22)
K22=K22(pints,train_data[outs],U21,U22)

U11 = (-learn_rate*K11).expm()*U11
U12 = (-learn_rate*K12).expm()*U12
U13 = (-learn_rate*K13).expm()*U13
U21 = (-learn_rate*K21).expm()*U21
U22 = (-learn_rate*K22).expm()*U22
```

Note that QuTip has a native method for matrix exponentiation of appropriate quantum objects (callable by *.expm()*) .

3.2. Results

First and foremost, the cost vs. epoch data showed a clear trend overall in a direction suggesting the network is somehow 'learning' to simulate the action of the unknown unitary evolution.

A few different experiments with the model network were run: first consisting of clean data with varied parameters, and then the effects of noisy data investigated.

3.2.1. Clean Data

Clean data was used to experiment with parameters and compare performance on unseen data.

First the performance between the training and test cost was compared, as in **Fig. 3.2.1**, which showed a close correlation between the two, as should be expected. The test data will then be used in other experiments to compare performance.

Note: Discrepancies in Results

Through experimentation, three large discrepancies were noted in the performance of the comparative models: **1.** A much larger learning rate was required for somewhat comparable convergence in my implementation of the network; **2.** The cost ceiling in training seemed to be lower, in the range of .96 as opposed to 1; **3.** The initial cost of the randomly chosen training data was much higher than their data's.

This may depend on several factors. One suspect feature, given point **3.** is that their method of generating initial, random unitaries and training data allowed for better convergence. In fact, their training data was only based on pure states, whereas mine was chosen from random density matrices. The impact of this factor is supported by a preliminary test of their model on a similarly generated set of data.

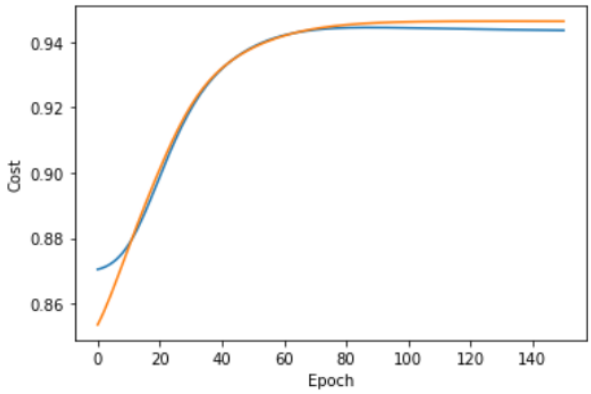


Fig. 2. *Simulated Results of 2x3x2 QNN on Clean Data:* Above is plotted the cost vs. training epoch for the simulated network for 250 training pairs over 150 epochs with a set learning rate of $\epsilon = 0.5$ and with $\eta = 1$ for all K. Note that the orange line represents the cost for the training data, while the blue line represents the cost for an independent set of 10 test pairs.

The impact of limited training data on this test data performance was then explored for several different sized training sets. The results can be found in **Fig. 3.2.1**. The differing performance based on varied learning rate is shown in **Fig. 3.2.1**.

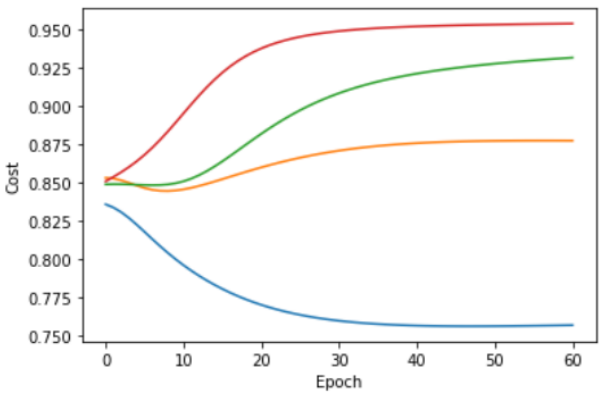


Fig. 3. *Simulated Results of 2x3x2 QNN on Clean Data with Few Training Pairs:* Above is plotted the cost vs. training epoch for the simulated network for a different number of a few training pairs over 60 epochs with a set learning rate of $\epsilon = 0.5$ and with $\eta = 1$ for all K. Note that the orange represent the cost for 5 training pairs, blue 10 training pairs, green 25 training pairs, and red 50 training pairs; all the costs are for an independent set of 50 test pairs.

3.2.2. Noisy Data

Finally, the networks performance on independent, clean test data after training on otherwise clean training data diluted with with varied quantities of random noise is shown in **Fig. 3.2.2**. This noise was generated from completely random unitaries acting on completely random states, and then appended to the clean training data. The dilution clearly effects the model's performance on the clean test data, although with suprisingly good results even up to a dilution of 50%.

Conclusion

The example network can clearly be taken as a proof of concept for the proposed network. For varied cases the network showed incremental improvement in the simulation of an arbitrary unitary evolution given some set of representative states.

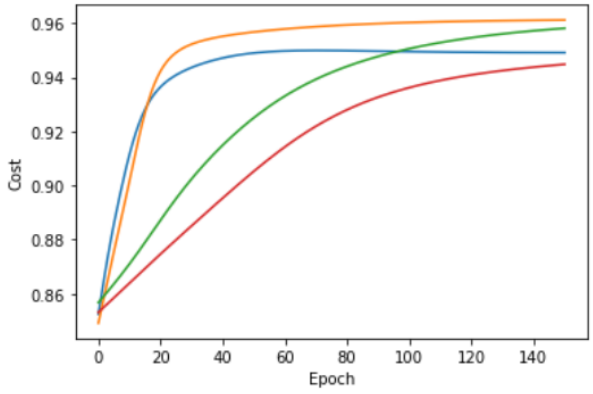


Fig. 4. *Simulated Results of 2x3x2 QNN on Clean Data with Differing Learning Rate:* Above is plotted the cost vs. training epoch for the simulated network for 50 training pairs over 150 epochs with $\eta = 1$ for all K. Note that the orange represent the cost for a learning rate of $\epsilon = 1$, blue $\epsilon = 0.5$, green $\epsilon = 0.25$, and red $\epsilon = 0.1$; all the costs are for the same training data.

Further experiments could include deeper networks, wider networks, and performance on data representative of channels that require partial traces over larger environments.

This general framework for a circuit may prove useful over time, even soon in the NISQ era given it's relatively small qubit requirements for deeper networks (the largest state being 2^{n+m} dimensional for the largest pair of adjacent layer qubits $n + m$).

For more information about quantum neural networks, see `https://alexheilman.com/res/qis/qml`. To download the jupyter notebook used to generate the shown results and including all the code used, use `https://alexheilman.com/products/projects/qnn/qnn_232.ipynb`. To compare to the original code's notebook, see `https://github.com/qigitphannover/DeepQuantumNeuralNetworks/blob/master/DQNN_basic.ipynb`

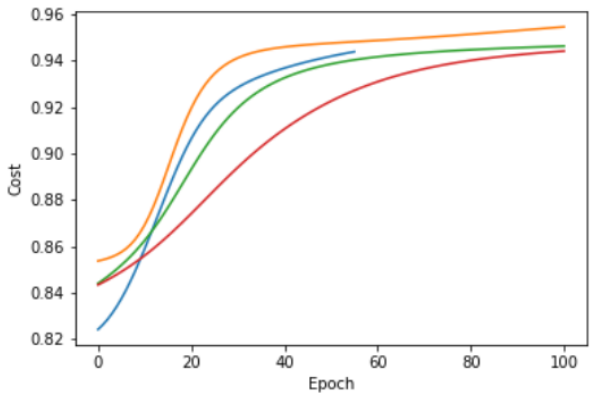


Fig. 5. *Simulated Results of 2x3x2 QNN on Noisy Data:* Above is plotted the cost vs. training epoch for the simulated network for 200 (with varied noisy) clean training pairs over 100 epochs with a set learning rate of $\epsilon = 0.5$ and with $\eta = 1$ for all K. Orange represents 0 noisy pairs in the training, blue 50 noise, green 150 noise, and red 200 noise. The cost displayed is for an independent set of 30 test pairs corresponding to the clean data's operation. Note further that the training was stopped at a training data cost of .95 (since a perfect match wouldn't correspond exactly to the unknown unitary, given noise).

References

Beer, K., Bondarenko, D., Farrelly, T., Osborne, T. J., Salzmann, R., Scheiermann, D., and Wolf, R. (2020). Training deep quantum neural networks. *Nature communications*, 11(1):1–6.

Nielsen, M. A. and Chuang, I. (2002). Quantum computation and quantum information.