Quantum Code Generation with Conditional Recurrent Neural Networks

YOAV RABINOVICH

MARCH 2020

Abstract

A Conditional Stacked Recurrent Neural Network with GRU cells is used to generate quantum circuits based on desired target quantum states. The conditional network is trained on randomly sampled circuits and their simulated output states as conditions that are introduced into the internal memory state of the initial GRU cell. The network fails to achieve the desired target states, possibly due to the synthesized dataset which lacks correlations in gate placement within each circuit. An analysis of the method and the results is provided, as well as a discussion about possible avenues for refinement, and an overview of the subjects of conditional RNNs and quantum circuits.

CONTENTS

1 INTRODUCTION

1.1 *Quantum Computing*

The Extended Church-Turing Thesis hypothesized that simulation of physical processes using digital computers can be done efficiently, in up to polynomial time and space with respect to the problem size ([Kaye et al., 2008]). However, since the 1980s many have conjectured that the simulation of quantum mechanics is an example of problems that don't conform to the thesis, or that computers that employ quantum information for computation can provide exponential speedups compared to digital computers on some problems ([Feynman, 1982]). The complexity class bounded-error quantum polynomial time (BQP), is therefore conjectured to strictly contain the classical bounded-error probabilistic polynomial time (BPP), violating the Extended Church-Turing Thesis, and extending the realm of efficiently solvable decision problems through the use of quantum information processing, which is referred to as "Quantum Advantage" and sometimes as "Quantum Supremacy". One promising example motivating research in the field was Peter Shor's quantum algorithm for integer factorization in polynomial time, employing a quantum Fourier transform to provide an almost exponential speedup over the most efficient known classical method ([Shor, 2002]). Propelled by the theoretical possibility of fault-tolerant implementations of quantum computing using quantum error correction ([Gottesman, 2009]), the field of quantum computing is surging with activity in the academic and private sectors. Recently, a milestone in quantum computing was achieved by Google, sampling the distribution of outputs from random quantum circuits with an exponential speedup relative to classical computers, demonstrating "Quantum Advantage", although disputed by IBM and others ([Arute et al., 2019])[1][2][3].

[1] *#QuantumAlgorithms*[1]. *Throughout this paper, LO and HC tags will be mentioned in the margin notes and expanded upon in appendix B. Note that CP194 LOs are combined into other HC tags for clarity.*

[2] *#Purpose*[7]

[3] *#Context*[8]

1.1.1 *Quantum Mechanical Underpinnings*

The most important correction quantum information theory makes to the classical picture is the introduction of probability amplitudes as a generalization of probability. According to quantum mechanics, the state of the system is described by a normalized superposition of possible states, weighted by complex amplitudes:

$$|\Psi\rangle = \sum_{i}^{m} \alpha_i |i\rangle \, , \; \sum_{i}^{m} |\alpha_i|^2 = 1, \; \vec{\alpha} \in \mathbb{C}^m$$

. The systems then obey one of two mechanisms of time-evolution: When the system is isolated, the Schrodinger equation describes a smooth time-evolution dictated by the system's Hamiltonian: $\hat{\mathcal{H}} |\Psi\rangle = i\hbar \frac{\partial}{\partial t} |\Psi\rangle$. When the system is observed, the Borne rule describes a non-linear abrupt change in state, which is probabilistically determined by the (real) square magnitudes of the original superposition amplitudes. Therefore, while unobserved, the quantum system can be carefully manipulated with the imposition of appropriate Hamiltonians into a complex superposition of states that corresponds to some element of a decision problem, which then can be repeatedly prepared and measured to reveal the information to an observer ([Townsend and Brown, 1993]). For example, in the Deutsch-Jozsa algorithm, one of the first quantum algorithms to show a theoretical speedup, a quantum state can be arranged to represent the answer to a certain binary inquiry about a function in such a way that the amplitudes that correspond to

the wrong answer cancel out, analogously to destructive interference in other quantum contexts, while the amplitudes that correspond to the correct answer constructively interfere ([Deutsch and Jozsa, 1992]). This key aspect of quantum algorithms is only possible due to the amplitude generalization of probability: Since amplitudes, unlike probabilities, are complex, they can cancel each other out[4].

[4] #MeasurementTheory²

1.1.2 *Qubits, Entanglement and Space of States*

The basic computational unit in quantum computing is the qubit (quantum bit). It represents a system that can take two possible states, which make up the basis state of our representation, and its state is therefore a superposition of the two:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}$$

. While a classical bit embodies a phase-space of two states, the qubit embodies much more. The superposition of the $|0\rangle$ and $|1\rangle$ is constrained by the normalization restriction, and by global phase invariance, which states that multiplication of the entire state by scalars preserves the original state. Therefore, with two degrees of freedom, the phase-space of a qubit can be mapped onto a spherical shell in a representation known as the Bloch sphere (Fig. 1). The latitude θ is then proportional to the probability of measurement of each basis state, and the longitude ϕ represents the relative phase of the amplitudes. States perpendicular to each other, defined as states which are maximally distinguishable from one another such as the basis states, then point in opposite directions, while equal superpositions of perpendicular states lie on the equator between them. For example, the states

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$$
$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}$$

are perpendicular to each other, and represent equal superpositions of the basis states.

Quantum computers then operate on multi-qubit systems to process information. Unlike the state of a bit-string, which can be described by definition using only n bits, an n-qubit system is described by a set of complex amplitudes that correspond to the probabilities to measure the system in any configuration of n bits, leading to a state-space of 2^n complex numbers. [5]. The combinatorics behave this way because the system is not reducible to single-qubit superpositions: qubits can interact between themselves and their time-evolution can, therefore, be dependent on other qubits, in entangled states that can only be represented as superpositions of many-qubit systems. The simplest examples, Bell states, are maximally entangled states which encode only knowledge about the pair of qubits, but none about each individual one. For example, the state

[5] *The amplitudes in turn are represented by twice as many numbers, to capture the real and imaginary dimensions:* 2^{n+1} *in total*

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \bigotimes |0\rangle_B) + \frac{1}{\sqrt{2}}(|1\rangle_A \bigotimes |1\rangle_B) = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

will produce only measurements of aligned states, but the measurement outcome for each qubit is completely unknown before the other one is measured. Unentangled states can trivially be stored efficiently in a classical computer

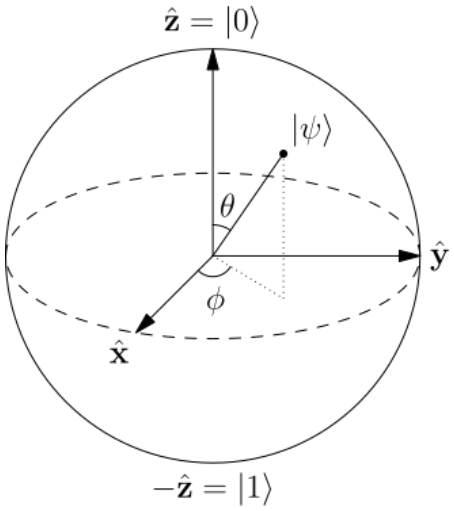

Figure 1: The Bloch sphere represents the state of a qubit as a unit-vector on a shell. Normally, the $|0\rangle$ state is represented as pointing upwards, the $|1\rangle$ state as pointing downwards, and equal superpositions of them populating the equator, such as the $|+\rangle, |-\rangle$.

with only $2n$ amplitudes, which leads to the necessity of entanglement for the achievement of any significant speedup[6]: [6] #Qubits[3]

$$|\Psi\rangle = (\alpha_1 |0\rangle + \beta_1 |1\rangle) \bigotimes (\alpha_2 |0\rangle + \beta_2 |1\rangle) ... \bigotimes (\alpha_n |0\rangle + \beta_n |1\rangle).$$

1.1.3 *Quantum Circuits and Universal Sets of Gates*

Different realizations of quantum computers employ different models of computation, but this paper will only discuss the gate model of quantum computing. Quantum circuits are represented as sequences of logical gates operating on a quantum mechanical equivalent of a n-bit register. Each gate represents a unitary transformation that the qubits are forced to undergo. Unitary operators, which satisfy $\hat{U}\hat{U}^\dagger = \hat{U}^\dagger\hat{U} = I$, are transformations that preserve the magnitude of the affected state, which guarantees that mapping of valid quantum states to valid quantum states. This entails that contrary to most classical gates, quantum gates are reversible, reflecting the conservation of information that the time-evolution of quantum states obeys.

For example, the Hadamard gate (H) shifts the basis of a qubit from the computational basis ($|0\rangle, |1\rangle$) to the polar basis ($|+\rangle, |-\rangle$) and vice versa, which corresponds to a 90-degree rotation around the y-axis on the Bloch sphere. The Hadamard gate can be represented as a unitary matrix, and its operation on the basis states can be verified:

$$\begin{aligned} H &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ H|0\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = |+\rangle \\ H|-\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \end{aligned}$$

Arbitrary unitary operations can transition any state to any target state precisely and with ease but are unfeasible for real implementations of gate

model quantum computers. Instead, a small set of fundamental gates which can be implemented must be chosen, and it must be able to approximate any output state. Moreover, the more applications are required to complete such an approximation, the longer and more error-prone the process becomes. Thankfully, arbitrary approximation of any target state is possible in a reasonably small amount of steps using manageable universal sets of gates.

Due to locality of interaction, any unitary transformation can be decomposed into a set of two-qubit gates. Additionally, any two-qubit unitary can be decomposed into a single-qubit unitary paired with a standard two-qubit operation such as the controlled-NOT ($CNOT$, or C_X) gate, which flips the state of one qubit depending on the state of the other[7]:

[7] *Since two-qubit gates operate on two-qubit states, they have dimensions* $2^n = 4$.

$$C_X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$C_X |01\rangle = \frac{1}{2} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = |01\rangle$$

$$C_X |11\rangle = \frac{1}{2} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |10\rangle .$$

Finally, the Solovay–Kitaev theorem implies that any single-qubit gate can be approximated to an accuracy ϵ using approximately $O(\log^2(1/\epsilon))$ gates from a small discrete set ([Dawson and Nielsen, 2006]). Therefore, the number of gates from the chosen universal set needed to approximate an arbitrary quantum circuit scales polylogarithmically with the size of the original circuit.

There are many universal sets of gates available to choose, but they're all guaranteed to include the necessary tools to explore the entirety of the phase-space of the n-qubit system:

1. The ability to create superpositions in single-qubit states (for example, using H).
2. The ability to create entanglement between two or more qubits (for example, using C_X).
3. The ability to introduce imaginary components to the amplitude[8].

[8] *#QuantumCircuits*[4]

One common set of universal gates, which this paper uses later to demonstrate the code generation technique, is indeed Hadamard (H), Controlled-Not (C_X) and the Phase gate (S), which corresponds to a 90-degree rotation around the computational ($|0\rangle, |1\rangle$) axis, and enables the exploration of the imaginary amplitude dimension of the phase-space:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

$$S |0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$

$$S |+\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ i \end{pmatrix} = |i\rangle .$$

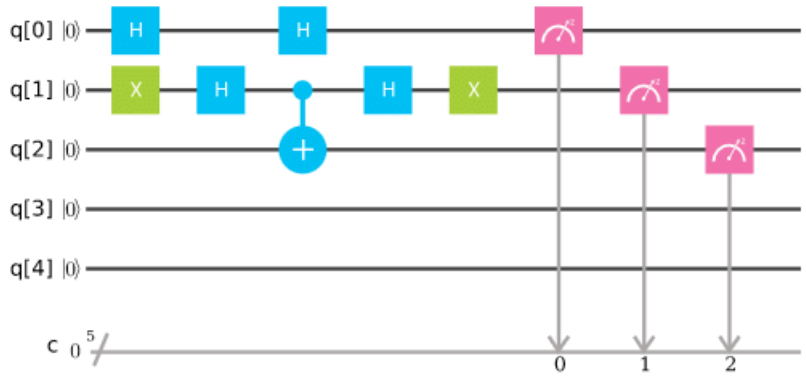

Figure 2: Example of a quantum circuit employing Hadamard gates (H), NOT gates (X), a Controlled-Not (where the state of qubit 1 determines whether or not qubit 2 is flipped) and measurements. Read from left to right, a sequence of gates are applied to a 4-qubit register, and measurements are recorded on a classical register.

1.2 *Deep Learning Theory and Techniques*

Deep learning uses multiple layers of machine learning algorithms to gradually extract high-level features from raw input ([Deng and Yu, 2014]). With artificial neural networks, supervised deep learning algorithms serve as universal function approximators that learn correlations between input features and output labels in data-sets of paired examples. This makes deep neural networks effective tools to automate processes that rely on complex relationships in high-dimensional data.

1.2.1 *Neural Networks and Backpropagation*

Artificial neural networks are constructed from layers of interconnected computational elements (neurons) which transform incoming signals into outgoing signals through activation functions, mimicking action potentials in biological brains (Fig. 3). The action of a neuron j in layer l on its incoming signals a_j^{l-1} given its weights w_j^l and its bias b_j^l and an activation function f_a is

$$z^l = (w_j^l)^T a^l + b_j^l$$
$$a^l = f_a(z^l).$$

Adjusting individual weights and biases on incoming signals for each neuron in a network can encode arbitrary transformations from input to output within the network ([Goodfellow et al., 2016]). To perform these adjustments, the neural network is exposed to paired examples of such inputs and outputs from a data-set pertaining to a specific problem, and the system parameters are corrected through gradient descent to produce the correct output given any input. The process of gradient descent is fueled by the backpropagation algorithm ([Nielsen, 2015]):

1. A given input x is fed through the network: going through the first layer of activation functions f_a to produce $a^1 = f_a(x)$.

2. Feeding forward through the network, each layer $l = 1, 2, .., L$ provides the transformation $z^l = w^l a^{l-1} + b^l$ and $a^l = f_a(z^l)$, given the current weights and biases w, a.

3. Given a cost function f_c, the final layer activation is compared to the expected output from our paired example: $C = f_c(y, a^L)$.

4. Then, since the cost and activation functions are all required to be differentiable, we can begin backpropagating the error to find the gradient of the cost function. The output error is found by element-wise multiplication of the gradient of the cost with respect to the output and the derivative of the activation function that was applied to the final layer inputs.

$$\delta^L = \frac{\partial C}{\partial z^L} = \nabla_a C \odot f'_a(z^L)$$

5. Backpropagation continues down the network, each layer $l = L-1, L-2...1$ being assigned a delta.

$$\delta^l = \frac{\partial C}{\partial z^l} = ((w^{l+1})^T \delta^{l+1}) \odot f'_a(z^l)$$

6. Finally, the gradient of the cost function with respect to each weight w^l_{jk} (of neuron j in layer l coming from neuron k in layer $l-1$) and bias b^l_j can be calculated by breaking down the term using the chain rule, since the derivative of the output of a neuron with respect to its weights is the incoming activation[9].

[9] #Optimization9

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k$$
$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial b^l_j} = \delta^l_j$$

Iterated through many training examples, a neural network learns to capture correlations of arbitrary complexity between input and output. The universal approximation theorem for feedforward neural networks states that given enough hidden layers (layers stacked between the input and output layers), a neural network can represent any function (within certain weak limitations) to arbitrary accuracy ([Goodfellow et al., 2016]). However, it does not state that the network has any way to learn this representation, since gradient descent might not be able to find a path there. Regardless, deep neural networks have demonstrated remarkable effectiveness in a wide arrange of tasks, despite the lack of guaranteed optimum[10] [11].

[10] #DeepLearning5

[11] *To synthesize the discussions on quantum computing and deep learning, a literature review of deep learning techniques in service of quantum computers is included in appendix A. This is not to be confused with quantum algorithms for deep learning, which is a subject beyond the scope of this review.*

1.2.2 *Recurrent Neural Networks*

Language models in machine learning aim to model the distribution of sequences of a certain vocabulary, and then to generate sequences by sampling from conditional probabilities for each word. Latent variable models avoid the exponential memory costs by instead approximating the distribution using a latent or hidden state which aims to capture the previous sequence elements in a single parameter ([Zhang et al., 2020]):

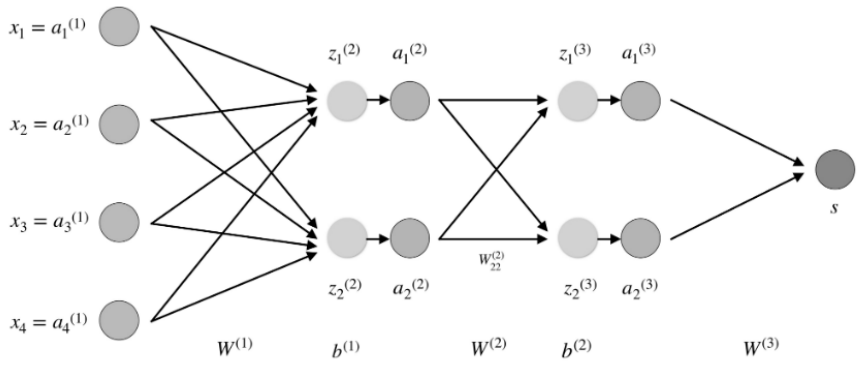

Figure 3: An example neural network, with four-dimensional input, two hidden layers, and one-dimensional output. The neurons between each layer are connected, and each layer applies an activation function to the weighted sum of inputs.

$$P(x^t|x^1, x^2..., x^{t-1}) \approx P(x^t|x^{t-1}, h^{t-1}).$$

Recurrent neural networks (RNNs) are latent variable models specialized for processing sequential data through parameter sharing. By sharing weights between layers that process different time-step values in the input allows the network to generalize its learning across the sequence. A dynamical system that's dependent on the previous state, as well as any other parameters, can be naturally unrolled into a nested representation:

$$s^3 = f(s^2, \theta) = f(f(s^1, \theta), \theta).$$

Similarly, we can represent a dynamic hidden state that depends on a subsequence of previous sequence elements as either a function of the elements, or a function that encompasses state transitions:

$$h^t = g^t(x^t.x^{t-1}, ..., x^1, \theta) = f(h^{t-1}, x^t, \theta).$$

We can, therefore, train a network to approximate a single transition function f to characterize the dynamic process, instead of some time-dependent set of functions g^t.

This hidden state, being a vector of fixed value that's dependent on previous elements serves as a "memory" for those projections of previous states that are imprinted on it. An optimized RNN would learn transition functions that maximize the utility of such memory for prediction of following elements in the sequence ([Goodfellow et al., 2016]).

A typical RNN, as illustrated in figure 4, includes three sets of weights for input-to-hidden-state connections (U), hidden-state-to-output connections (V) and hidden-state-to-hidden-state connections (W). A prediction $\hat{y}$ is then extracted from the output, and compared to label y to compute loss and run backpropagation.

In a stacked (or "deep") RNN model, RNN layers are fed into each other in a sequence, each layer containing several computational units. This added depth has been shown to be effective in allowing the representation of intricate mappings by [Pascanu et al., 2014] among others.

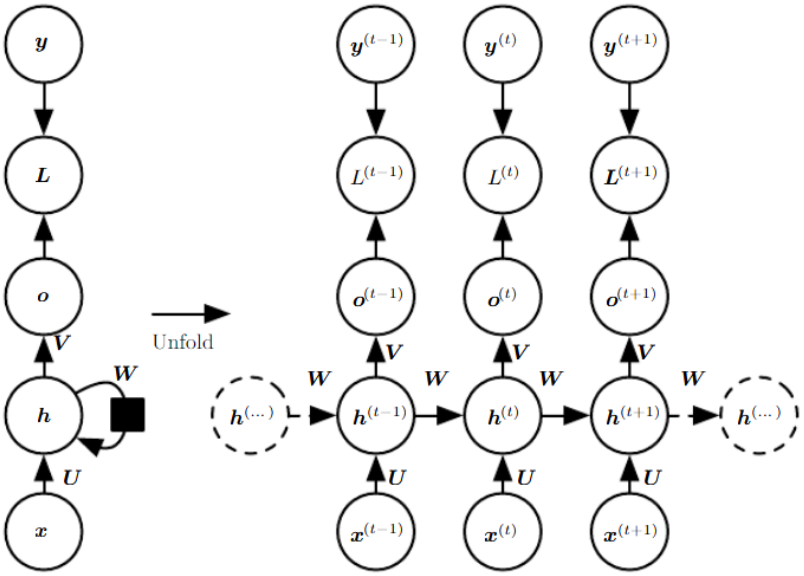

Figure 4: From [Goodfellow et al., 2016], an example of a typical RNN structure.

1.2.3 *Gated RNNs*

The vanishing gradient problem in the context of RNNs is used to describe the indiscriminate way hidden states forget information about previous states in exponential decay. If an important piece of information is found earlier in the sequence, it has little hope to survive prominently in the hidden state until the end. One way found to address this problem is by packaging the RNN layers in units that learn to monitor the change of the hidden state based on the inputs at time t.

A simple example of a gated RNN is the gated recurrent unit (GRU), which employs two extra sets of weights paired with their own activation functions: the Reset Gate (R) and the Update Gate (U), which together allow adjusting the proportion by which the old hidden state and the new input are represented in the new hidden state[12].

[12] #SequentialPrediction[6]

$$\begin{aligned}
\mathbf{R}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r),\\
\mathbf{Z}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z).\\
\tilde{\mathbf{H}}_t &= \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\,\mathbf{W}_{hh} + \mathbf{b}_h).\\
\mathbf{H}_t &= \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.
\end{aligned}$$

1.3 *Conditional RNNs*

RNNs lack a mechanism by which to consider non-sequential information in the correlations they learn. A conditional RNN attempts to force the network to output different predictions based on conditions external to the preceding sequence. Conditional RNNS aim to approximately model not only $P(x^t|x^1, x^2, ..., x^{t-1})$, but given some constant external condition $\boldsymbol{\theta}$, to model $P(x^t|x^1, x^2, ..., x^{t-1}, \boldsymbol{\theta})$[13].

[13] #Probability[10]

Figure 6 shows an example of a conditional RNN used to learn tasks such as image captioning, where a static external condition (the image) is correlated with the output sequence (the caption) throughout the captioning

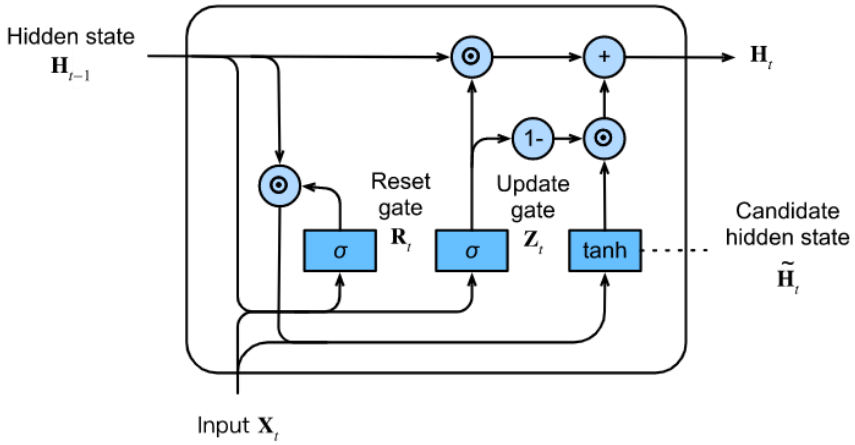

Figure 5: From [Zhang et al., 2020], an example of a GRU cell.

process. There are three main schemes to construct "conditional" RNNs in the literature:

1. Have the first time-step contain conditioning data, to set the internal state of the RNN. This is similar to what happens in encoder-decoder networks, which encode sequences into an input element which is then interpreted by a dedicated decoder, which feeds the prediction of a cell as input to the next until an end-of-sentence token is predicted. However, this restricts the conditioning data to the shape of an element in the sequence, and also still susceptible to the vanishing gradient problem, raising concern that the network doesn't the conditioning information as prediction continues.

2. Append the condition to the temporal input, such that each feature vector includes both the time series data and a constant condition vector. This approach pollutes the data and requires the network to learn the two different types of correlations at once, which is inefficient.

3. Directly set the internal state of the RNN at time zero according to the condition. For each training sample, we apply a trainable dense layer to reshape the condition data into the right shape as the internal state of the RNN, and for the first time-step, we insert the result to the internal state of the RNN. This approach has been demonstrated successfully by [Karpathy and Fei-Fei, 2017] and [Vinyals et al., 2015].

2 METHOD

This paper set out to generate quantum algorithms based on desired target n-qubit states using conditional RNNs.

To teach a neural network to generate circuits based on a target end-state, a data-set of circuits paired with known end-states was synthesized, based on a universal set of gates that served as a vocabulary for the network. A conditional recurrent neural network was then fed segments of circuits along with the target end-state as a condition, and the gate following the segment as a label for supervised training. The neural network then learns the distributions $P(G^t|\mathbf{G}^{<t}, \mathbf{c})$, where $G^t \in 1, .., N$ is the application of a certain

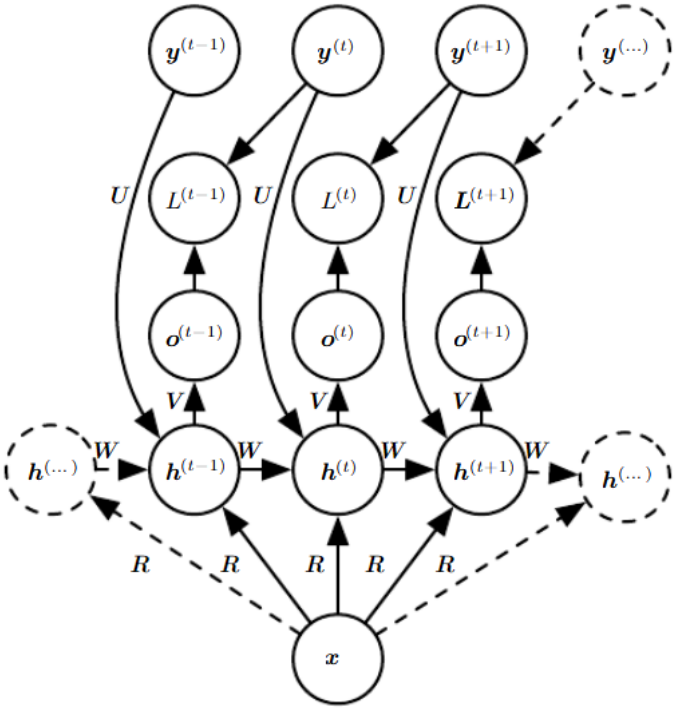

Figure 6: From [Goodfellow et al., 2016], an example of a conditional RNN. The condition x is static across time-steps.

gate in the vocabulary of size N[1], $\mathbf{G}^{<t} = \{G^1, G^2, ..., G^{t-1}\}$ is a specific sequence of previous gates, and $\mathbf{c}$ is the target end-state.

Since the training data is sampled completely randomly, the model was predicted to quickly learn to ignore the conditioning of gate probability on the previous gates $P(G^t|\mathbf{G}^{<t})$ as no information is contained there and instead learn the correlations between applied gates and the target state condition $P(G^t|\mathbf{c})$, and the interaction terms regarding previous gates and external condition[2].

Four variations of the model were examined. Firstly, two different circuit encoding schemes were built and tested, corresponding to two different loss functions for the model. Secondly, two different schemes of conditioning the network on the target states were built and tested, influencing the data-flow and making use of different RNN layers.

[1] *More accurately, a vocabulary of size $\alpha n + \beta n(n-1) + 3$ is required, where n is the size of the system in qubits, α and β are the amount of one- and two-qubit gates in the available set respectively, and start, end and padding tokens are included.*

[2] *#Sampling[11]*

2.1 *Data Synthesis*

Given a choice of system size in qubits n, a maximum circuit length k, and a selected set of allowable gates, a data-set of randomly sampled quantum algorithms is generated[3] ([Abraham Asfaw et al., 2020]). The circuits are then converted into an array of indices using a vocabulary scheme (described below) to encode each gate[4] ([QAS, 2020]). Special tokens are placed to denote the beginning and end of a circuit, as well as a token for padding each circuit to match with the maximum circuit length, to keep each example similar in dimensionality. Each circuit is then simulated to determine its output on an initialized state, which will be used as a condition when training the network on the corresponding circuit. Finally, the prediction labels for

[3] *This is done using the IBM's open-source library QisKit, which allows for construction, simulation and format conversion of quantum circuits, among other functionalities.*

[4] *Open Quantum Assembly Language (OpenQASM) provides a standard format for encoding quantum circuit which is supported by QisKit.*

each example are produced by matching each gate with the gate that follows it in the circuit.

The final data-set contains m examples of sampled circuits encoded as arrays of length[5] $k+2$, paired with target end-states encoded as arrays of length[6] 2^{n+1} and with a left-shifted copy of the encoded circuit as labels.

For this implementation, a choice of a universal set of quantum computational gates was made in advance, but in principle the model can be trained for any set of gates. The gates included in this implementation are the H gate, which puts qubits in and out of superpositions of computational states; the S gate which shifts superpositions but doesn't act on computational states; and the C_X gate which puts pairs of qubits into entangled states by flipping one target qubit depending on the state of another control qubit.

[5] *Fully padded, with beginning and end tokens.*

[6] *Floats required to represent the complex state-space.*

2.1.1 *Encoding Schemes*

Two different circuit encoding schemes were built and tested[7]:

[7] *#Algorithms*[12]

1. **Vocabulary Encoding Scheme**: Gates are encoded as integers indexing a vocabulary of possible gates, with single-qubit gates H and S encompassing the first $2n$ vocabulary words, and the two-qubit gate C_X encompassing the remaining n^2 words (ignoring tokens). This is a straightforward encoding scheme, but was predicted to produce bad results by comparison to one-hot encoding since the distance metric used as a loss function is mean-squared-error, which implies arbitrary "distances" between different gates in the vocabulary. This method casts the problem as a regression, while it's more fitting to treat it as a classification problem.

2. **One-Hot Encoding Scheme**: Gates are first encoded as integers based on the vocabulary scheme, but are then translated into one-hot-vector representation: each gate is represented by a vector with length equal to the vocabulary length, where one element is set to one. The paired loss function, logit softmax cross-entropy, more naturally fits the classification task.

2.2 *Model Structure*

The inputs are fed into a stacked conditional RNN with GRU units, the outputs of which are then passed through a stack of dense layers for classification and prediction. In the case of the integer encoding, the model's optimized according to mean-squared-error loss, and under the one-hot scheme I use logit softmax cross-entropy (Fig. 7). The conditional RNN was implemented in two different fashions:

1. **Hidden-State Model**: The condition is passed through a trainable dense layer and inserted directly into the hidden-state of the RNN at time-step one[8] (Corresponds to item 3 in section 1.3).

2. **Input-Concatenated Model** The condition is appended to the input of the RNN in each and every time-step. (Corresponds to item 2 in section 1.3):

[8] *As this functionality isn't present in* Keras, *we use Philippe Rémy's library* cond rnn, *available through* pip.

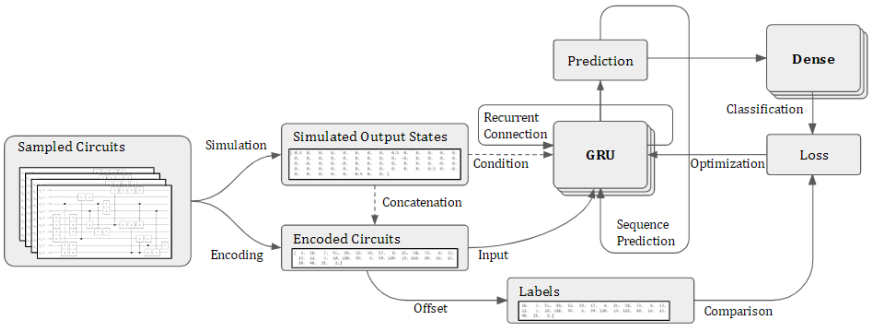

Figure 7: Diagram of the model structure. The sampled circuits are encoded based on one of the schemes to produce RNN inputs, offset to produce labels and simulated to create conditions. The conditions are then either fed into the hidden-state of the first time-step of the RNN or concatenated with the encoded inputs. Predictions are then classified using a stack of Dense layers, and loss is computed using the labels, with a loss function dependent on the encoding scheme.

2.3 *Metrics for Model Evaluation*

In these models, loss for gradient descent is based on the distance between a predicted gate and a label. While a good score in correct prediction will likely lead to a good result, the underlying goal of the model is to be able to reach the target state with its predicted algorithms. The random gate application of the synthesized data-set is likely to be an inefficient way to get there, and so this metric shouldn't be the only one upon which the model is evaluated. We will refer to this loss metric simply as "**RNN Loss**", regardless of whether it was computed on a vocabulary model using MSE or on a one-hot encoded model using softmax cross-entropy, and be cognizant of only comparing those losses that are comparable in this manner.

Another possible metric is the mean amount of correctly predicted gates for an entire generated algorithm when one is sampled in its entirety from the model using a target state condition from the test set. While this metric is again directed at reproduction rather than reaching close to the target, and although lacking accuracy in data-sets of short circuits, this metric has the benefit that the outputs of any of the models are directly comparable using it. We call this metric "**Sequence Loss**".

Finally, to judge our intended goal of approximating a target state, we simulate every single algorithm produced by the RNN based on test set conditions and compare it to the target state it was conditioned on, using MSE. While this metric accurately reflects our goal, it's important to note that while models using different encoding schemes are comparable using this metric, models trained on different size registers are not comparable: The larger the register, the larger the condition vector and the more elements in the end state remain zero after the application of our circuit, bringing large-register possible end-states closer together than smaller register end-states. We call this metric "**Target Loss**"[9].

[9] *#DescriptiveStats*[13]

3 RESULTS

In total, eight experiments were run, alternating conditional RNN structure and circuit encoding scheme, as well as running on two different data-sets:

Encoding	Dataset		Hidden-State Models Sequence Loss	Target Loss	RNN Loss
Vocabulary	Simple	Train	0.8	0.033	8.1
		Test	0.8	0.033	8.16
	Complex	Train	1.0	0.017	72.5
		Test	1.0	0.017	74
One-Hot	Simple	Train	0.5	0.033	1.48
		Test	0.51	0.035	1.53
	Complex	Train	0.81	0.019	2.88
		Test	0.85	0.019	3.07

Table 1: Computed metrics for the **hidden-state** models after 1000 epochs, for different experimental setups as specified in appendix C.

a **simple** data-set of 3-qubit circuits of length 3 and a **complex** data-set of 5-qubit circuits of length 10. Both data-sets were synthesized with 10,000 sampled circuits and all experiments were run for 1000 epochs. The full specification of the model parameters and computing set-up is available in appendix C.

All experiments resulted in a resounding failure in all metrics. Their summaries are available table 1 (hidden-state models) and table 2 (input-concatenated models). Their RNN loss was also recorded throughout training, and is presented in figures 8 and 9 respectively.

While all models learn the rules regarding "end-of-sequence" and padding tokens very quickly, no other learned insights can be clearly identified from sampling algorithms. As expected, the one-hot encoded model performs better than the integer scheme when comparing sequence loss, but the target loss is equal between them, amounting to random chance. Sampling the models, it's apparent that only single-qubit gates are ever predicted, omitting C_X gates which are essential for universal computation. The models ignore most of the entire vocabulary available to them and tend to simply predict the previous gate will repeat most of the time. Figure 10 shows a histogram of predicted gates in the test-set, showing mostly a single predicted value, and only a range of 12 values out of a vocabulary of 37 gates.

Examining the loss over training, it appears that one-hot encoding models are more susceptible to overfitting, as their test and train losses begin diverging earlier and do so more severely. In both conditional RNN structures, the one-hot encoding models showed erratic behavior training on the complex data-set, exploring more diverse areas of the phase-space.

4 DISCUSSION

RNNs are used in tasks like time-series prediction and language generation because they learn the conditional probability that an element in the vocabulary will follow a given subsequence. The random algorithms sampled in this work to produce the data-set are independent of each other and lack any correlations in the ordering of elements, and therefore external conditions in the form of target states were introduced to guide the learning, predicting that the model will learn that the uncorrelated sequence orders are devoid of useful information and thus the correlation between the order and the condition will be isolated. However, it seems that in lieu of order correlations,

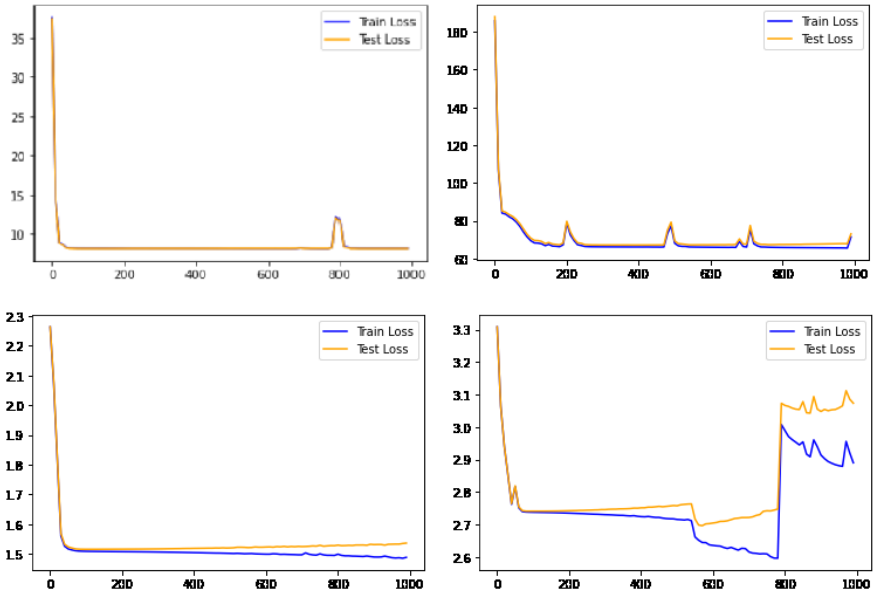

Figure 8: RNN loss over epochs of training for **hidden-state** models. Vocabulary encoding above, one-hot encoding below. Simple data-set left, complex data-set right.

Encoding	Dataset		Input-Concatenated Models Sequence Loss	Target Loss	RNN Loss
Vocabulary	Simple	Train	0.8	0.034	8.15
		Test	0.8	0.033	8.11
	Complex	Train	1.0	0.019	75.8
		Test	1.0	0.019	76.9
One-Hot	Simple	Train	0.5	0.035	1.53
		Test	0.51	0.035	1.57
	Complex	Train	0.85	0.019	2.97
		Test	0.85	0.019	2.97

Table 2: Computed metrics for the **input-concatenated** models after 1000 epochs, for different experimental setups as specified in appendix C.

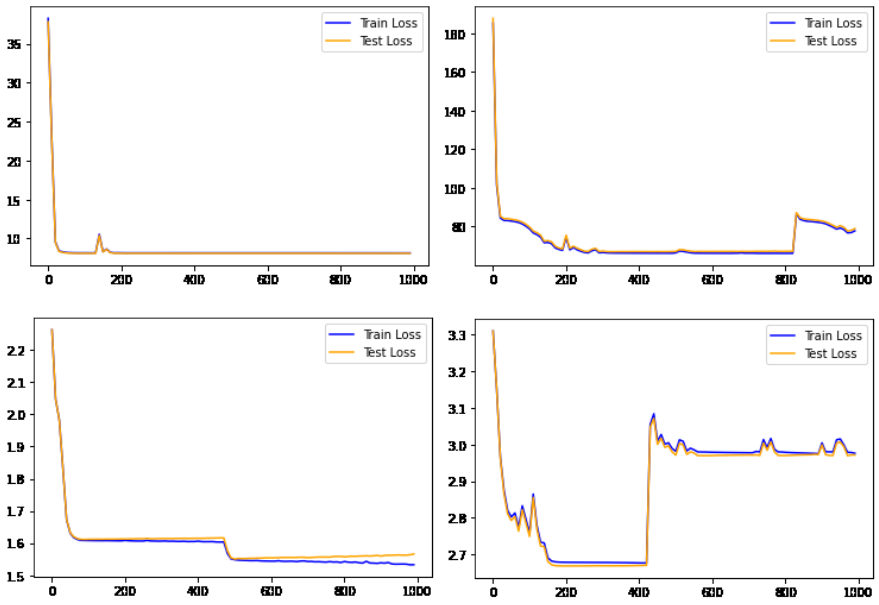

Figure 9: RNN loss over epochs of training for **input-concatenated** models. Vocabulary encoding above, one-hot encoding below. Simple data-set left, complex data-set right.

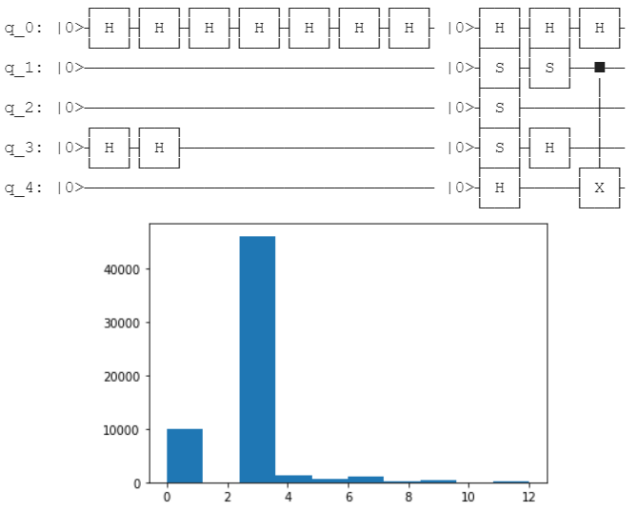

Figure 10: Example of predictions. Top left: predicted circuit. Top right: matching original circuit. Bottom: Histogram showing the distribution of predicted gates in the entire test-set. These examples are the results of the best performing experiment, hidden-state one-hot encoded model on the complex dataset.

the model is resorting to simple local minima such as repeating the most common gates or achieving the most common output.

A possible fault is that for this problem the space of sequences is much larger than the space of condition states, which might mean that gradient descent is not likely to take steps in directions that are strongly sloped in target loss without having access to this loss function during training, as there are likely to be many minima in the phase-space of conditional gate-sequence probabilities that are not necessarily correlated with target loss [1]. A data-set of more intelligently constructed algorithms might serve as a better proxy, also teaching the model some best-practices embedded in valuable sequential correlations within the circuits that will help it converge on a correct sequence[2].

[1] #SystemDynamics[14]

[2] #DesignThinking[15]

Models might be more successful could they be granted the ability to asses their progress towards their goal as they predict elements of the sequence: where the loss function is derived from the output of the sequence constructed. However, simulating quantum computers, while deterministic, requires the use of external packages employing numerical models, which are not differentiable and unusable for backpropagation. Even rewriting a differentiable Quantum simulator in TensorFlow, as section 1.1 discussed, simulating quantum algorithms on classical hardware cannot be done efficiently. A reinforcement learning approach could be viable, if simulating small enough quantum algorithms can be done fast enough to show results in a reasonable time-span[3].

[3] #Constraints[16]

It is possible that the vocabulary is too small. While a small universal set of gates (such as the minimal one chosen for this paper) is easier to learn, it might be too limiting, since there are relatively few possible paths to any target state. A larger arsenal might lead to better approximations when more diverse tools are at the model's disposal. Specifically, the inclusion of an "identity" operation could allow the model to halt the algorithm without having to deal with variable-length sequences. This might also open a path to circuit length optimization, where the model is rewarded for brevity. This is an important part of any application of quantum code generation since current quantum computers are noisy enough that any microsecond of computation time counts[4].[5]

[4] #MultipleCauses[17]

[5] Unanchored HC tags: #Professionalism[18], #Organization[19], #Composition[20], #SourceQuality[21], #EvidenceBased[22], #Responsibility[23].

REFERENCES

[QAS, 2020] (2020). Quantum Inspire Knowledge Base.

[Abraham Asfaw et al., 2020] Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Francis Harkins, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, David McKay, Zlatko Minev, Paul Nation, Anna Phan, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, Kristan Temme, Madeleine Tod, and James Wootton (2020). Qiskit Textbook.

[Arute et al., 2019] Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C., Barends, R., Biswas, R., Boixo, S., Brandao, F. G., Buell, D. A., Burkett, B., Chen, Y., Chen, Z., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Fowler, A., Gidney, C., Giustina, M., Graff, R., Guerin, K., Habegger, S., Harrigan, M. P., Hartmann, M. J., Ho, A., Hoffmann, M., Huang, T., Humble, T. S., Isakov, S. V., Jeffrey, E., Jiang, Z., Kafri, D., Kechedzhi, K., Kelly, J., Klimov, P. V., Knysh, S., Korotkov, A., Kostritsa, F.,

Landhuis, D., Lindmark, M., Lucero, E., Lyakh, D., Mandrà, S., McClean, J. R., McEwen, M., Megrant, A., Mi, X., Michielsen, K., Mohseni, M., Mutus, J., Naaman, O., Neeley, M., Neill, C., Niu, M. Y., Ostby, E., Petukhov, A., Platt, J. C., Quintana, C., Rieffel, E. G., Roushan, P., Rubin, N. C., Sank, D., Satzinger, K. J., Smelyanskiy, V., Sung, K. J., Trevithick, M. D., Vainsencher, A., Villalonga, B., White, T., Yao, Z. J., Yeh, P., Zalcman, A., Neven, H., and Martinis, J. M. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510.

[Dawson and Nielsen, 2006] Dawson, C. M. and Nielsen, M. A. (2006). The Solovay-Kitaev algorithm. *Quantum Information and Computation*, 6(1):081–095.

[Deng and Yu, 2014] Deng, L. and Yu, D. (2014). Deep Learning: Methods and Applications. Technical Report MSR-TR-2014-21, Microsoft.

[Deutsch and Jozsa, 1992] Deutsch, D. and Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558.

[Feynman, 1982] Feynman, R. P. (1982). Simulating Physics with Computers. Technical Report 6.

[Fösel et al., 2018] Fösel, T., Tighineanu, P., Weiss, T., and Marquardt, F. (2018). Reinforcement Learning with Neural Networks for Quantum Feedback. *Physical Review X*, 8(3).

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[Gottesman, 2009] Gottesman, D. (2009). An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation. pages 13–58.

[Karpathy and Fei-Fei, 2017] Karpathy, A. and Fei-Fei, L. (2017). Deep Visual-Semantic Alignments for Generating Image Descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):664–676.

[Kaye et al., 2008] Kaye, P., Laflamme, R., Mosca, M., Vedral, V., and Friedman, J. R. (2008). An introduction to quantum computing and introduction to quantum information science. *Optical Engineering*, 47(2):61–62.

[Liu et al., 2019] Liu, G., Chen, M., Liu, Y.-X., Layden, D., and Cappellaro, P. (2019). Repetitive Readout Enhanced by Machine Learning.

[Nielsen, 2015] Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

[Nielsen and Wang, 2012] Nielsen, M. A. and Wang, Y. (2012). Quantum computation and quantum information. *Statistical Science*, 27(3):373–394.

[Niu et al., 2019] Niu, M. Y., Boixo, S., Smelyanskiy, V. N., and Neven, H. (2019). Universal quantum control through deep reinforcement learning. *npj Quantum Information*, 5(1):33.

[Palmieri et al., 2019] Palmieri, A. M., Kovlakov, E., Bianchi, F., Yudin, D., Straupe, S., Biamonte, J., and Kulik, S. (2019). Experimental neural network enhanced quantum tomography.

[Pascanu et al., 2014] Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2014). How to construct deep recurrent neural networks. In *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR.

[Quek et al., 2018] Quek, Y., Fort, S., and Ng, H. K. (2018). Adaptive Quantum State Tomography with Neural Networks.

[Shor, 2002] Shor, P. (2002). Algorithms for quantum computation: discrete logarithms and factoring. pages 124–134. Institute of Electrical and Electronics Engineers (IEEE).

[Townsend and Brown, 1993] Townsend, J. S. and Brown, L. S. (1993). A Modern Approach to Quantum Mechanics. *American Journal of Physics*, 61(1):92–93.

[Vinyals et al., 2015] Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 07-12-June, pages 3156–3164. IEEE Computer Society.

[Zhang et al., 2020] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2020). *Dive into Deep Learning*.

A LITERATURE REVIEW: NEURAL NETWORKS FOR QUANTUM CONTROL

Quantum computing introduces a brand-new set of challenges for optimization: decoherence and noise play a non-negligible part, incentivizing fast performance, and the inaccessible nature of most information in quantum states requires complicated protocols to peer through. The universal function approximating properties of neural networks allow them to support speedup and fidelity improvements in existing protocols, as well as help design completely novel control procedures. This literature review covers some prominent uses of neural networks in quantum control, along with case studies.

A.1 *Neural Networks for Readout*

Measurement schemes in quantum computers vary with physical implementations. "Single-shot readout", in which a single measurement is sufficient for state determination is a key component in error correction, and therefore it's seen as necessary for scalable quantum computing. However, single-shot readout capabilities have so far not been achieved for several qubit implementations with otherwise favorable characteristics like high operating temperatures and long decoherence times, which also play a role in scalability. Repetitive readout protocols employ Quantum Non-Demolition measurements, which do not scramble the measured qubit's state after measurement and therefore can be applied repeatedly to infer the measurement outcome for such qubits where single-shot readout is not feasible. [Liu et al., 2019] have employed simple feed-forward neural networks to improve repetitive readout fidelity in room-temperature nitrogen-vacancy center (NV center) qubits, which provides a good case study to demonstrate the potential of neural networks for improving measurement procedures in quantum computing.

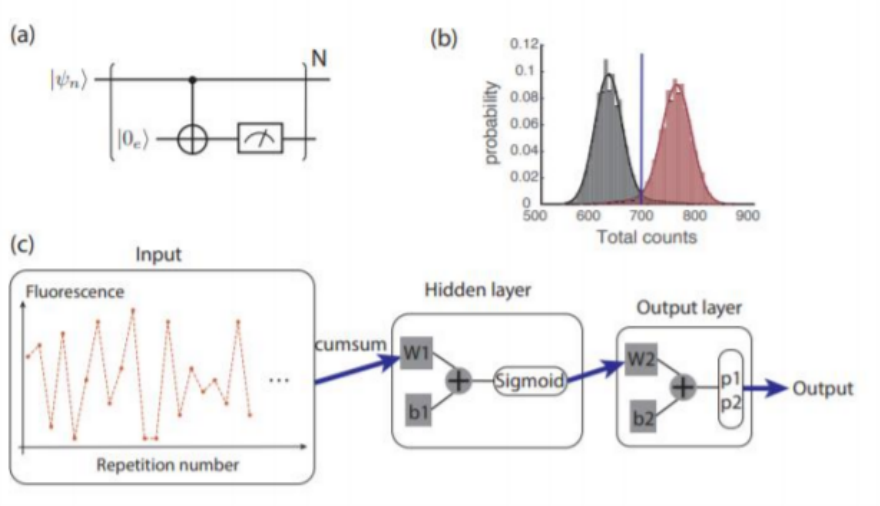

Figure 11: From [Liu et al., 2019]. (a) Quantum circuit for repetitive quantum-non-demolition readout of the nuclear spin state $|\psi_n\rangle$, using the ancilla electronic spin ($|0_e\rangle$). (b) A typical histogram of total photon numbers collected from repetitive readout, originating from bright (red) and dark (grey) states is generated using simulation and shown. A threshold at the cross point classifies future readout results in the threshold method. (c) Shallow neuron network architecture, with sigmoid as activation function and softmax output.

In NV center qubits, a nuclear spin state is read out by it correlating through hyperfine interactions with an ancilla electronic spin, which is then measured through detecting emitted photons (Fig. 11.a). After many measurements, the total photon number is traditionally compared to a threshold to infer the nuclear spin state: above a certain amount, the state is inferred to be in the bright state, and otherwise it's inferred to be in the dark state (Fig. 11.b). [Liu et al., 2019] replaced the threshold method by feeding fluorescence data from the photon counting mechanism directly to a simple, 1-hidden-layer neural network (Fig. 11.c), which provides a classification for the nuclear spin state.

Not only has this simple approach achieved higher maximum fidelity of readout compared to the threshold method, but it has also proved robust against the threshold method's most significant flaws, nuclear spin flips. In repetitive readout, despite the non-demolition nature of the measurements, the unobserved nuclear spin state might randomly flip during the measurement due to "back-action" effects. This probability grows as we increase the number of measurements, as more time passes, and noise builds up. Early flipping might not lead to the threshold method identifying the later state rather than the original one. However, as the neural network examines the time-series fluorescence data, it picks up on time-based correlations that allow it to detect the flips, and therefore accuracy increases monotonously as measurement count increases (Fig. 12.a), rather than reaching an optimum point limiting measurement fidelity. This ability of neural network learning methods to make use of non-trivial correlations gives them the power to improve current protocols in many components of quantum computing, a field in which small improvements to effectiveness and efficiency are crucial to future scalability in the pursuit of feasible large-scale quantum computers.

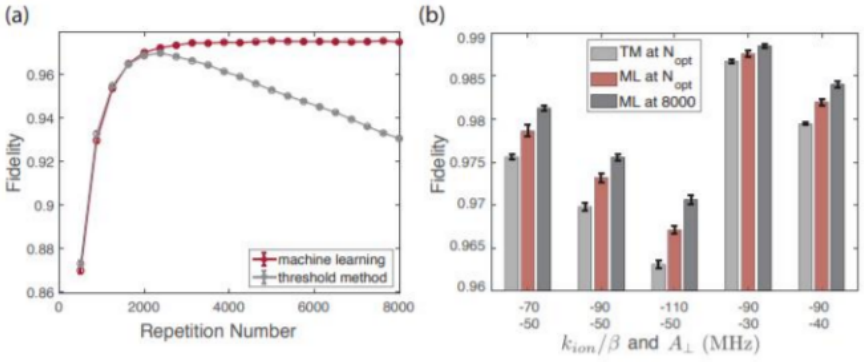

Figure 12: From Liu et al. (2019). (a) Readout fidelity as a function of repetition number N in the repetitive readout. The fidelity from the threshold method (grey) declines after an optimum $N_{\text{opt}} = 2375$ due to increasing probability of nuclear spin flips. The fidelity from the neural network (red) keeps improving, although the increase rate slows down. (b) Fidelity comparison of threshold method at its optimal repetition number N_{opt}, the neural network at Nopt, and the neural network at $N = 8000$ under different NV parameters. The neural network consistently outperforms the threshold at N_{opt}, and also improves with larger N.

A.2 *Deep Learning for Quantum State Tomography*

Calibration of qubit sources and gates requires a process of reconstructing the density matrix for an arbitrary mixed state, using many measurements of the output states of the objects of interest. A d-dimensional quantum system is described using d^2 parameters, each of which needs to be estimated with several measurements, which is a costly procedure. Maximum likelihood estimation and Bayesian mean estimation are popular classical techniques used to statistically infer the density matrix from observations. State preparation-and-measurement (SPAM) errors arise from imperfect knowledge of the trial states and measurement operators. Approaches like randomized benchmarking, where the average error over a set of random gates cancels out, and gate-set tomography, where measurement protocols are used an experimental apparatus as if it represents a quantum computational "black box" to determine its characteristics have been reasonably successful, but are costly: They require both long sequences of operations and measurement, as well as long processing times for the estimation of the states, rendering them too slow for use in applications such as quantum communication. [Palmieri et al., 2019] have used over-complete autoencoders, a type of deep neural network, which converges on a function reducing the Kullback-Leibler divergence between the noisy, SPAM error afflicted measurement distribution and an ideal distribution according to the system specifications. The trained network can then effectively de-noise measurement data, allowing for higher fidelity reconstruction of the measured states (Fig 13). This autoencoder can be applied quickly to raw data to provide faster, more accurate reconstructions.

Adaptive Bayesian Quantum Tomography is a technique in which results of previous measurements are used to optimize the next measurement taken, to extract as much useful information as possible with each measurement. [Quek et al., 2018] used neural networks to shorten the processing time by substituting components of Bayesian adaptive methods with a neural-network-learned heuristic. The team used a recurrent neural network, which takes previous inputs into account when processing new inputs, which en-

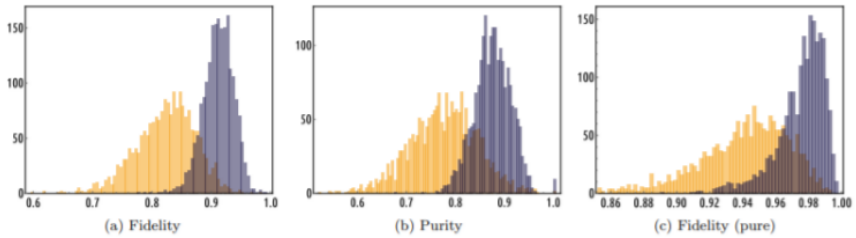

Figure 13: From [Palmieri et al., 2019]. Results of experimental state reconstruction. (a) Fidelity of the experimentally reconstructed states for 2000 test states reconstructed from raw data (orange bars) and reconstructed after neural network processing of the data (blue bars). (b) A similar diagram for purity of the reconstructed states. (c) Fidelity histogram for the case, when the state is reconstructed to be pure.

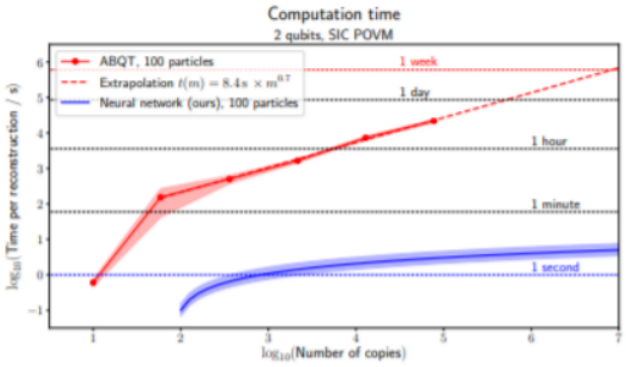

Figure 14: From [Quek et al., 2018]. Scaling of reconstruction run-time with number of copies measured. The neural algorithm run-time empirically scales logarithmically with the number of copies measured, while ABQT empirically scales polynomially (approximately as $m^0.7$).

codes sensitivity to correlations over time across samples. The model, trained within hours, reaches comparable fidelity to previous Bayesian methods but is to 6 orders of magnitude faster to execute (Fig. 14).

A.3 *Deep Learning for Quantum Error Correction*

Quantum error-correcting codes are a key development in quantum computation, allowing for the construction of composite "effective" qubits from several qubits, able to maintain their represented quantum state in the face of noise and decoherence by using parity measurements to detect state changes due to noise. However, quantum computers with different hardware resources and under different types of noise require different strategies that are hard to pinpoint in a search space of arbitrary sequences of gate applications. Despite this, [Fösel et al., 2018] developed a deep learning model that was able to find error detection and mitigation strategies for arbitrary few-qubit computer architectures and under arbitrary noise models. The network was trained threw reinforcement learning, only having access to gate application and readout without any other knowledge of the qubit states or any encoded knowledge of the underlying quantum mechanics. The team first developed a suitable metric for error correction to enable training, using "recoverable quantum information" defined as the minimum probability to distinguish

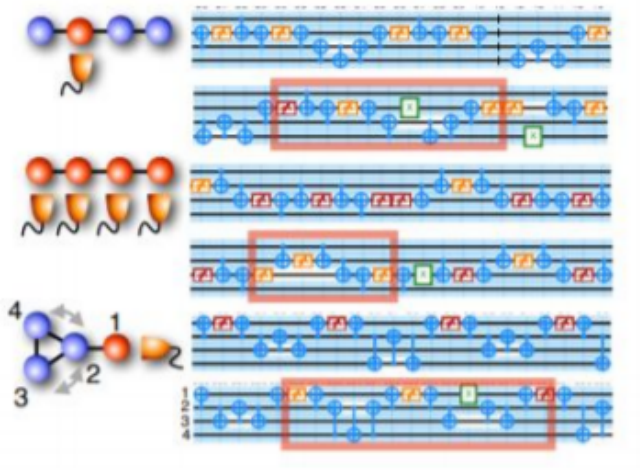

Figure 15: From [Fösel et al., 2018]. Error correction codes converged upon by the network for different computer architectures, featuring different connectivity schemes and different possible measurement points. The red rectangle highlights an interval during which the quantum state is not dominated by a single component, indicating that the precise location of the error hasn't been pinpointed. These nontrivial detection/recovery sequences are considerably more complex than the periodic detection cycle.

between any two orthogonal states on the Bloch sphere following the application of noise, and the network's attempt at detecting and correcting its effects. Using this metric as a cost-function, they trained a network using a teacher/student scheme, where a teacher network was trained on an easier problem (having access to the state information of the system at all times) and is then used to help train a student network on the harder problem (having only access to measurement on certain qubits, per the system architecture). The resulting strategies are not only suited to the architecture and noise model (Fig. 15) but are also adaptive: they react to measurement results non-trivially. This complexity leads in some cases to better performance than standard periodically applied algorithms for error correction (Fig. 16).

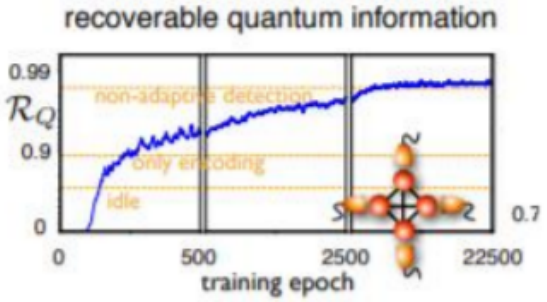

Figure 16: From [Fösel et al., 2018]. Training progress in terms of the average recoverable quantum information R_Q, evaluated at the final step of a 200-step gate sequence. Eventually, the network performs even better than a combination of encoding and periodic parity checks, due to the adaptive recovery sequences that are triggered by unexpected measurements (i.e. upon error detection). In this example, that leads to ca. 15% increase of the decoherence time over a non-adaptive scheme.

B HCS AND LOS

NOTES

[7]#*Purpose*: I analyze the goals and motivations behind research in quantum computing in the past four decades. I explain the theoretical predictions that motivate the activity in the academic and private sectors. As justification, I reference key papers and developments.

[1]#*IL181.007QuantumAlgorithms*: I effectively summarize the justification for the pursuit of quantum computing technology in terms of complexity theory, and offer a succinct informative rundown of important results in the field motivating research.

[8]#*Context*: This paragraph also serves for putting my project in the appropriate context for the reader. I explain the promise of quantum computing and the reason for its current popularity, as an introduction for the rest of the paper (#*feedback*: This need for context was a constant point of improvement brought up by my friend Dasha reviewing this piece, and I kept iterating until she was satisfied, while I offered her constructive criticism regarding the clarity and flow of her technical introduction sections. I'd like to use this opportunity to thank her and my friend Mano, as well as our advisor, for the constant support and feedback throughout this process).

[2]#*NS142MeasurementTheory*: I describe quantum measurements in the Copenhagen interpretation as a sudden non-linear localization of the wave function, as opposed to smooth time evolution of the quantum state. I describe probability amplitudes as an extension of probabilities and explain their ability to interfere applied in the context of quantum algorithms.

[3]#*IL181.007Qubits*: I effectively describe the Hilbert space of single- and multi-qubit systems, in terms of perpendicular basis states, the Bloch sphere, and entanglement, and tie that description to their significance in quantum computing.

[4]#*IL181.007QuantumCircuits*: I describe the implementation of unitary transformation in terms of universal sets of gates, bringing examples, including outlining the proof of the existence of such sets and the Solovay-Kitaev theorem regarding their ability to efficiently approximate any n-qubit unitary. I also outline basic intuition behind universal sets of gates through the justification of exploration of the complex and entangled Hilbert space of the quantum system.

[9]#*Optimization*: My discussion of backpropagation as a means of achieving gradient descent in the high-dimensional optimization space of neural network shows an understanding of optimization techniques as well as relevant mathematical background.

[5]#*IL181.016DeepLearning*: I describe deep neural networks as universal function approximators referencing the theoretical limits of function representation versus representation learning. I describe the basics of neural networks including the neuron structure and backpropagation, as well as the high-level description of a progressive coarse-graining correlation model.

[6]*#IL181.016SequentialPrediction*: I describe the probability modeling and mathematical background on language models, latent variable models, RNN unrolling, gated units and gradient vanishing in the context of hidden-state memory in the context of sequential prediction.

[10]*#Probability*: My discussion of modeling probability in language models, latent variable models, RNNs, conditional RNNs and later when describing my hypothesis regarding how the network might learn to ignore sequence-order correlations shows a grasp of the mathematical background of probability, conditional probability, and probability factoring.

[11]*#Sampling*: My hypothesis regarding how the network might learn to ignore sequence-order correlations relies on an assumption of uniform sampling of gates to prove independence. I then choose to synthesize my data for that reason among others, showing a grasp of the sampling dynamics with relation to the probability model.

[12]*#Algorithms*: While I refer the reader to my code as a whole in grading this HC, which I believe is clean efficient, and well-annotated, and uses simple and effective design patterns, I'd like to point out the encoding code as well as the input concatenation code as an example of efficient `numpy` array computation.

[13]*#DescriptiveStats*: In this description and in the respective code I exhibit clear and appropriate uses of simple descriptive stats and custom problem-specific metrics while paying attention to their strengths and weaknesses as well as the logic behind their possible comparability (*#metrics*).

[14]*#SystemDynamics*: I effectively discuss the problem in terms of spaces-of-states and their exploration by the system, describe the effect of the dimensionality and shape of the phase-space on the dynamics of gradient descent, and propose ways by which to influence the system's path through the phase-space.

[15]*#DesignThinking*: In the discussions section I immediately purpose my insights about the results into a list of ideas for incremental steps for improvement. My advisor will also readily admit that this list used to be longer: I already effectively used the results of previous attempts to improve my end product. In fact, the entire idea of building and testing a second conditional RNN structure was ideated as part of this iterative design process (*#planningarchitecture*: Even since my very recent latest pivot near the end of Fall semester, I was able to iterate quickly and improve on my projecto.)

[16]*#Constraints*: I effectively considered the constraints of my chosen model and my chosen problem when designing the method, as well as when discussing possible solutions: differentiability of loss functions and efficiency of quantum circuit simulation being two key constrains around which I designed my solutions.

[17]*#MultipleCauses*: I effectively identify and contrast several sources from which effects could spring that combine to make my models fail. For example, I identify lack of gate variety and lack of variable-length options as possible contributors to the model's failure.

[18]#*Professionalism*: My paper accomplishes a high degree of presentation and professionalism, uses proper citations, margin notes, endnotes, figures and tables (#*qualitydeliverables*).

[19]#*Organization*: My paper is organized clearly, in the clear format of other academic papers in the fields, and with neatly flowing logical connections.

[20]#*Composition*: My writing is professional, concise and accurate.

[21]#*SourceQuality*: I chose all of my article citations from reputable publications, all of my book citations from standard textbooks in the field, and made sure to correctly cite them all for ease of fact-checking and quality verification as well as to credit the authors responsibly.

[22]#*EvidenceBased*: Throughout this paper, I support my facts and my methods with reputable sources from academic papers and renowned textbooks (#*research*: This careful justification is a result of rigorous research I've completed over the past year, regarding mostly graduate-level material in both the field of computational sciences and that of physics).

[23]#*Responsibility*: Thanks to the generosity of the academic team regarding one deadline this semester, I was able to overcome the struggle of two unplanned pivots and submit this project on time (#*accountability*)(#*connect*: Those pivots, as my advisor knows, arose from my success in independently contacting professionals in the field and pitching my project for their sponsorship, initially successfully. I am proud of this achievement, and I'm on track to continue working with them this summer).

C MODEL CODE AND EXPERIMENTAL SETUP

This project as well as all code used to build, train and analyze the model can be accessed through GitHub using the following URL: https://git.io/JvHG0. The code is additionally attached at the end of this appendix.

The experiments in this paper were run using Google's Colab platform, hosted on a GPU. The model parameters that were used in all experiments are:

```
# Model Parameters
N = 5 # size of qubit register
MAX_LENGTH = 10 # maximum length of gate sequence
VOCAB_DIM = 2+2*N+N**2 # amount of possible gates
NUM_SAMPLES = 10000 # number of circuits in the database
TRAINTEST = 0.75 # proportion of training data
CUTOFF = int(NUM_SAMPLES//2) # train-test cutoff
TIME_STEPS = MAX_LENGTH+2 # length of sequence including tokens
INPUT_DIM = 1 # size of input element
LABEL_DIM = 1 # size of label element
COND_DIM = 2*(2**N) # length of condition state
NUM_CELLS = 256 # number of cells in each RNN layer
STACK_DEPTH = 5 # number of stacked RNN layers
NUM_DENSE = 64 # size of dense classification layers
CLASS_DEPTH = 5 # number of stacked classification layers
LOSS_DELAY = 10 # number of epochs between each loss recording
PRINT_DELAY = 10 # number of recordings between each loss printing
SAVE_DELAY = 10000 # number of epochs between each model checkpoint
```

```
# Training Parameters
BATCH_SIZE = 64
EPOCHS = 1000 # hard cutoff for training
```

Code follows:

capstone

March 27, 2020

Capstone Project: Deep Learning and Quantum Information Theory

0.0.1 Quantum Code Generation with Conditional RNNs

Yoav Rabinovich, March 2020

Installs and imports

```
In [0]: !pip install qiskit
        !pip install tensorflow-gpu --upgrade
        !pip install -Iv cond-rnn==1.4
```

```
In [0]: import numpy as np
        import matplotlib.pyplot as plt
        import qiskit as qk
        import tensorflow.compat.v1 as tf
        import cond_rnn as crnn
        import re
        import os
        from google.colab import output
        tf.disable_v2_behavior()
        qs_backend = qk.Aer.get_backend('statevector_simulator')
```

Circuit Functions

```
In [0]: def sample_circuits(n,size,amount):
            """Sample an amount of random n-qubit circuits with a certain size in
            number of operations from the allowed set"""

            circuits =[]
            for _ in range(amount):
                # Create circuit object of n qubits
                circ = qk.QuantumCircuit(n)
                # Generate random gates on random qubits from the universal set {H,S,CX}
                for _ in range(size):
                    gate = np.random.randint(0,3)
```

```
            target = np.random.randint(0,n)
            if gate==0: # Hadamard
                circ.h(target)
            if gate==1: # S-gate
                circ.s(target)
            if gate==2: # CNOT
                control = np.random.randint(0,n)
                if control == target:
                    circ.h(target)
                else:
                    circ.cx(control,target)
        circuits.append(circ)
    return circuits

def encode_circuits(circuits,n,max_size,label=True):
    """Takes an array of n-qubit QuantumCircuit objects, and encodes them based on a
    vocabulary of possible gates to apply, including tokens to signify the start
    and end of sequences. Elements after EoS are padded to match maximum circuit
    size using a special token.
    Labels can be also be generated for the circuits."

    Vocabulary scheme:
    Padding = 0,
    SoS = 1,
    EoS = 2,
    h[0]=3, h[1]=3+1...
    s[0]=3+n, s[1]=3+n+1...
    cx[0,0]=3+2n, cx[0,1]=3+2n+1...
    cx[1,0]=3+(2+1)n, cx[1,1]=3+(2+1)n+1... etc. """

    encoded = []
    for circ in circuits:
        # Use the QASM format to convert the circuit to a string
        lines = circ.qasm().splitlines()[3:]
        size = len(lines)
        # Initialize to padding tokens
        encoded_circ = np.zeros(max_size+2)
        # Add SoS and EoS tokens
        encoded_circ[0] = 1
        encoded_circ[size+1]=2
        for i,line in enumerate(lines):
            # Detect gate name and qubits involved
            gate_str = line[:2]
            integers = [int(s) for s in re.findall(r'-?\d+\.?\d*',line)]
            # Encode gates based on scheme above
            if gate_str=="h ":
                encoded_circ[i+1]=int(3+integers[0])
            if gate_str=="s ":
```

```
                    encoded_circ[i+1]=int(3+n+integers[0])
                if gate_str=="cx":
                    encoded_circ[i+1]=int(3+(2+integers[0])*n+integers[1])
            encoded.append(encoded_circ)
        encoded = np.array(encoded)
        if label:
            # Simulate labels for each circuit and attach to dataset
            labels = generate_labels(circuits)
            return np.concatenate((encoded,labels),axis=1)
        else:
            return np.array(encoded)

    def decode_circuit(encoded,n,debug=False):
        """Takes an encoded output from the network and generates the corresponding
        circuit as described above."""

        # Start with opening syntax
        decoded = "OPENQASM 2.0;\ninclude \"qelib1.inc\";\nqreg q["+str(n)+"];\n"
        for line in encoded:
            # decode each non-token element into its QASM string
            line = int(np.around(line))
            if debug: print(line)
            if line > 2:
                gate_num = int(np.ceil((line-2)/n))
                if debug: print(gate_num)
                if gate_num==1:
                    decoded += "h q["+str(line-3)+"];\n"
                elif gate_num==2:
                    decoded += "s q["+str(line-n-3)+"];\n"
                elif (str(gate_num-3)!=str(line-(gate_num-1)*n-3)):
                    decoded += "cx q["+str(gate_num-3)+"],q["+str(line-(gate_num-1)*n-3)+"];
            if line == 2:
                decoded = decoded[:-1]
                break
            if debug: print(decoded)
        if debug:
            print(decoded)
        # Build circuit object from QASM string
        return qk.QuantumCircuit.from_qasm_str(decoded)
```

Preparation

Parameters

In [0]:
```
# Model Parameters
N = 5 # size of qubit register
MAX_LENGTH = 10 # maximum length of gate sequence
VOCAB_DIM = 2+2*N+N**2 # amount of possible gates
```

```
NUM_SAMPLES = 10000 # number of circuits in the database
TRAINTEST = 0.75 # proportion of training data
CUTOFF = int(NUM_SAMPLES//2)#(1/TRAINTEST))
TIME_STEPS = MAX_LENGTH+2 # length of sequence including tokens
INPUT_DIM = 1 # size of input element
LABEL_DIM = 1 # size of label element
COND_DIM = 2*(2**N) # length of condition state
NUM_CELLS = 256 # number of cells in each RNN layer
STACK_DEPTH = 5 # number of stacked RNN layers
NUM_DENSE = 64 # size of dense classification layers
CLASS_DEPTH = 5 # number of stacked classification layers
LOSS_DELAY = 10 # number of epochs between each loss recording
PRINT_DELAY = 10 # number of recordings between each loss printing
SAVE_DELAY = 10000 # number of epochs between each model checkpoint

# Training Parameters
BATCH_SIZE = 64
EPOCHS = 1000 # hard cutoff for training

# Model Structure
ONE_HOT = True
INPUT_CONCATENATED = False

# Correct input dimensions
if ONE_HOT:
    INPUT_DIM = VOCAB_DIM
    LABEL_DIM = VOCAB_DIM

if INPUT_CONCATENATED:
    INPUT_DIM += COND_DIM
```

Generate Data (Option)

In [0]:
```
# Generate Data
# Data = encode_circuits(sample_circuits(N,MAX_LENGTH,NUM_SAMPLES),N,MAX_LENGTH,label=Tr
# Save Data
# np.savetxt(f"Encoded_Circuits_{str(N)}_{str(MAX_LENGTH)}_{str(NUM_SAMPLES)}.csv", Data
```

Load Data (Option)

In [0]:
```
# Load Data
Data = np.loadtxt(f"Encoded_Circuits_{str(N)}_{str(MAX_LENGTH)}_{str(NUM_SAMPLES)}.csv",
```

Data Preprocessing

In [0]:
```
# Shuffle dataset
np.random.shuffle(Data)
```

```
# Inputs
X = Data[:,:TIME_STEPS]
X = X.reshape((X.shape[0],X.shape[1],1))
if ONE_HOT:
    # Convert to One-Hot Encoding
    eye = np.eye(VOCAB_DIM)
    X = np.array([[eye[int(gate)] for gate in X[int(row),:]] for row in range(X.shape[0]

# Labels
offset = X[:,1:]
y = np.concatenate((offset,np.zeros((X.shape[0],1,LABEL_DIM))),axis=1)

# Conditions
c = Data[:,TIME_STEPS:]

# Split
X_train = X[:CUTOFF]
X_test = X[CUTOFF:]
c_test = c[:CUTOFF]
c_train = c[CUTOFF:]
y_train = y[:CUTOFF]
y_test = y[CUTOFF:]

if INPUT_CONCATENATED:
    # Append conditions to inputs
    c_train_dup = np.repeat(c_train[:, :, np.newaxis],TIME_STEPS,axis=2)
    c_train_rot = np.rot90(c_train_dup,1,(1,2))
    X_train = np.concatenate((X_train,c_train_rot),axis=2)

    c_test_dup = np.repeat(c_test[:, :, np.newaxis],TIME_STEPS,axis=2)
    c_test_rot = np.rot90(c_test_dup,1,(1,2))
    X_test = np.concatenate((X_test,c_test_rot),axis=2)
```

Build Model

In [0]:
```
# Define Session
sess = tf.Session()

# Placeholders for variables
inputs = tf.placeholder(name='inputs', dtype=tf.float32, shape=(None, TIME_STEPS, INPUT_
targets = tf.placeholder(name='targets', dtype=tf.float32, shape=(None, TIME_STEPS, LABE
if not INPUT_CONCATENATED:
    # Condition variables for hidden-state model
    cond = tf.placeholder(name='conditions', dtype=tf.float32, shape=(None, COND_DIM))

# Conditional RNN
if INPUT_CONCATENATED:
    # Regular GRU
```

```
        outputs = tf.keras.layers.GRU(NUM_CELLS, return_sequences=True)(inputs)
        for _ in range(STACK_DEPTH-1):
            outputs = tf.keras.layers.GRU(NUM_CELLS, return_sequences=True)(outputs)
    else:
        # Conditional GRU
        outputs = crnn.ConditionalRNN(NUM_CELLS, cell='GRU', cond=cond, dtype=tf.float32, re
        for _ in range(STACK_DEPTH-1):
            outputs = crnn.ConditionalRNN(NUM_CELLS, cell='GRU', cond=cond, dtype=tf.float32

    # Classification
    for _ in range(CLASS_DEPTH):
        outputs = tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(units=NUM_DENSE, act
    outputs = tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(units=LABEL_DIM, activat

    # Loss + Optimizer
    if ONE_HOT:
        cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=outputs, lab
    else:
        cost = tf.reduce_sum(tf.reduce_mean(tf.squared_difference(outputs, targets)))
    optimizer = tf.train.AdamOptimizer().minimize(cost)

    # Initialize variables (tensorflow)
    sess.run(tf.global_variables_initializer())

    # Define the binding between placeholders and real data.
    if INPUT_CONCATENATED:
        train_feed_dict = {inputs: X_train, targets: y_train}
        test_feed_dict = {inputs: X_test, targets: y_test}
    else:
        train_feed_dict = {inputs: X_train, targets: y_train, cond: c_train}
        test_feed_dict = {inputs: X_test, targets: y_test, cond: c_test}
```

Train

In [0]:
```
    # Main loop. Optimize then evaluate.
    saver = tf.train.Saver()
    train_losses = []
    test_losses = []
    for epoch in range(EPOCHS):
        sess.run(optimizer, train_feed_dict)
        if epoch % LOSS_DELAY == 0:
            train_outputs, train_loss = sess.run([outputs, cost], train_feed_dict)
            test_outputs, test_loss = sess.run([outputs, cost], test_feed_dict)
            train_losses.append(train_loss)
            test_losses.append(test_loss)
        if epoch % (PRINT_DELAY*LOSS_DELAY) == 0:
            print(f'[{str(epoch).zfill(4)}] train cost = {train_loss:.4f}, test cost = {test
        if epoch % SAVE_DELAY == 0:
```

```
            saver.save(sess, f'Checkpoints/QCG-{str(NUM_CELLS)}', global_step=int(epoch/SAVE
```

Analyze

```
In [0]: # Plot Losses
        epochs = np.arange(0,epoch,PRINT_DELAY)
        plt.title("Loss Over Epochs")
        plt.plot(epochs,train_losses,c="blue",label="Train Loss")
        plt.plot(epochs,test_losses,c="orange",label="Test Loss")
        plt.legend()
        plt.show()
```

```
In [0]: # Forward pass examples
        test_examples, test_example_loss = sess.run([outputs, cost], test_feed_dict)
        train_examples, train_example_loss = sess.run([outputs, cost], train_feed_dict)

        if ONE_HOT:
            # OHE to Vocab
            ones_test = np.full((test_examples.shape[0],1),1)
            ones_train = np.full((train_examples.shape[0],1),1)
            ones_y_test = np.full((y_test.shape[0],1),1)
            ones_y_train = np.full((y_train.shape[0],1),1)
            vocab_test = np.argmax(np.around(test_examples),2)[:,:-1]
            vocab_train = np.argmax(np.around(train_examples),2)[:,:-1]
            vocab_y_test = np.argmax(np.around(y_test),2)[:,:-1]
            vocab_y_train = np.argmax(np.around(y_train),2)[:,:-1]
            test_examples = np.concatenate((ones_test,vocab_test),1)
            train_examples = np.concatenate((ones_train,vocab_train),1)
            test_y_examples = np.concatenate((ones_y_test,vocab_y_test),1)
            train_y_examples = np.concatenate((ones_y_train,vocab_y_train),1)
        else:
            test_y_examples =  np.copy(y_test)
            train_y_examples = np.copy(y_train)

        # Decode examples and simulate
        decoded_test = []
        decoded_train = []
        for i in range(train_examples.shape[0]):
            if i<=test_examples.shape[0]:
                encoded_test = np.concatenate((np.array([1]),test_examples[i].flatten()[:-2],np.
                decoded_test.append(decode_circuit(encoded_test,N))
            encoded_train = np.concatenate((np.array([1]),train_examples[i].flatten()[:-2],np.ar
            decoded_train.append(decode_circuit(encoded_train,N))
        test_labels = generate_labels(decoded_test)
        train_labels = generate_labels(decoded_train)

        # Compute Metrics
        test_sequence_loss=np.mean(np.mean(test_examples!=test_y_examples,axis=1))
```

```
        test_target_loss=np.mean(np.mean((c_test-test_labels)**2,axis=1))
        train_sequence_loss=np.mean(np.mean( train_examples!=train_y_examples,axis=1))
        train_target_loss=np.mean(np.mean((c_train-train_labels)**2,axis=1))

        # Print
        print("Sequence Loss (test/train):",test_sequence_loss,"/",train_sequence_loss)
        print("Target Loss   (test/train):",test_target_loss,"/",train_target_loss)
        print("RNN Loss      (test/train):",test_example_loss,"/",train_example_loss)
```

```
In [0]: # Demonstrate example
        i=np.random.randint(test_examples.shape[0])
        print(np.around(test_examples[i]).T)
        print(test_y_examples[i].T)
        print(test_labels[i])
        print(c_test[i])
        decoded_test[i].draw()

        # Draw histogram of predicted gates
        # plt.hist(np.around(test_examples.flatten()))
```

Audio Notification of Completion (Option)

```
In [0]: output.eval_js('new Audio("https://upload.wikimedia.org/wikipedia/commons/0/05/Beep-09.o
```

Yoav Rabinovich, March 2020