Supervised Machine Learning in Quantum Feature Spaces with Reservoir-Inspired Applications

by

Ryan Hill

Submitted to the Graduate Faculty of

the School of Applied and Engineering Physics in partial fulfillment

of the requirements for the degree of

Master of Engineering

Cornell University

2020

Supervised Machine Learning in Quantum Feature Spaces with Reservoir-Inspired Applications

Ryan Hill, MEng

Cornell University, 2020

This paper builds to the introduction of a supervised quantum reservoir computing model that, in theory, provides an exponential speed-up over its classical analog. To get there, we survey the space of machine learning and discuss the merit of the classical reservoir computing paradigm in comparison to SVM-type models. We then outline a real-world quantum variational classifier that utilizes a quantum enhanced feature space to separate data. Finally, we apply aspects of classical reservoir-based algorithms towards high-dimensional quantum machine learning and propose a general purpose feed-forward quantum reservoir feature-learner designed for a fault-tolerant quantum computer.

Acknowledgments

Thank you to Peter McMahon, my faculty supervisor, for the terrific opportunity to conduct research in your lab. Thank you to Joel Brock, my program advisor, for your constant support and guidance. I would like to acknowledge and thank Nam Mannucci and Andrew Yates for pioneering the QRC project. Thank you to the members of the quantum machine learning sub-group Andrew Pareles, Dario Bernal, and Andrew Yates for our discussions surrounding the quantum variational algorithm. Thank you to Maxwell Anderson and Chengji Liu from the classical reservoir computing group, with whom my research in this lab began. And thank you to Logan Wright for our discussions and your insights.

Table of Contents

List of Tables

List of Figures

1.0 Introduction

Quantum computers are devices that harness quantum mechanics to perform computations in ways that classical computers cannot. They were originally proposed as an elegant, more accurate way to simulate the natural world. The key difference between a quantum computer and a classical computer is the basic unit of data (i.e. information) used. In classical computers, information is encoded in bits, where each bit can have the value 0 or 1. Quantum computers encode information into a two-level quantum system called a quantum bit, or qubit. Physical implementations of qubits include polarizations of a photon, two of the (multiple) discrete energy levels of an ion, a superconducting Transmon qubit, the nuclear spin states of an atom or the spin states of an electron. A qubit can be in state $|0\rangle$ or $|1\rangle$ or (unlike a classical computer) a linear combination (i.e. superposition) of states:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where α and β are complex numbers. Therefore, the state of a qubit is a vector in a two-dimensional complex vector space. The special cases of $|0\rangle$ and $|1\rangle$ form an orthonormal basis for this vector space. Classical bits have a completely well-define state at every point during computation. By contrast, when we measure a qubit we could get either the result 0, with probability $|\alpha|^2$, or the result 1, with probability $|\beta|^2$. These probabilities must sum to unity: $|\alpha|^2 + |\beta|^2 = 1$. This conservation of probability allows us to write (1) as [1]

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle \tag{2}$$

where ϕ and θ are real numbers and define a point on the unit-three dimensional sphere, often called the Bloch sphere.

[1] Formally, this expression contains a multiplicative factor $e^{i\theta}$. However this term has no *observable* effects, so is ignored here for simplicity.

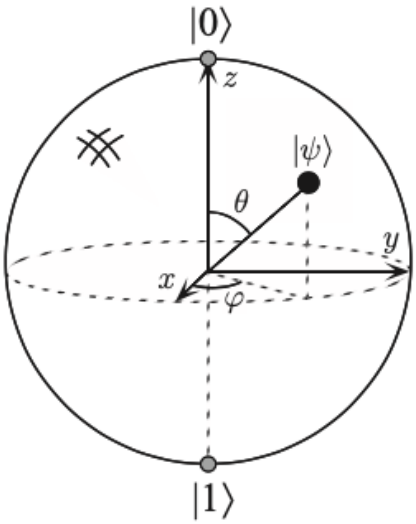


Figure 1: Bloch sphere representation of qubit [29]

Many interesting problems are impossible to solve on classical computers because of the astronomical resources required to solve realistic cases. Quantum computers allow us to construct new algorithms that render these problems feasible. Famously, Shor's (1994) algorithm for factoring integers provides an exponential speed-up over the best known classical algorithms, and Grover's search algorithm (1996) provides a quadratic speed-up over the best known classical algorithms. In this paper, we are interested in how we can use quantum algorithms to solve supervised machine learning problems with similar time complexity advantage.

2.0 Preliminaries

2.1 Machine Learning

2.1.1 Artificial Neurons

Artificial neural networks (ANNs) are computing systems that are inspired by, but not identical to, the biological neural networks that constitute animal (e.g. human) brains. A biological neural circuit is a population of neurons (or nerve cells) interconnected that carry out a specific function when activated. Neural circuits interconnect to form large-scale brain (or biological) neural networks. ANNs use connectionism [9], a cognitive theory based on simultaneously occurring, distributed signal activity via connections that can be represented numerically, where learning occurs by modifying connection strengths based on experience [33]. Connections between neurons in the brain are actually much more complex than those of the artificial neurons used in ANNs. Still, by using the "connectionist" model, ANNs attempt to exploit the architecture of the human brain. The brain has many desirable characteristics not present in von Neummann or modern parallel computers including massive parallelism, distributed representation and computation, learning ability, generalization ability, adaptivity, inherent contextual information processing, fault tolerance, and low energy consumption. By utilizing these characteristics, ANNs aim to perform tasks not well-solved by conventional algorithms.

All ANNs contain neurons, connections, and an activation function. *Neurons* (i.e. nodes) receive input, combine the input with their internal state using an activation function, and produce output using an output function. The initial inputs are external data, such as images. The ultimate outputs accomplish some task, such as recognizing an object in an image. *Connections* map the output of one neuron to the input of another neuron. Each connection is (typically) assigned a *weight* that represents its relative importance. A single neuron can have multiple input and output connections. In this way, the neurons of an ANN form a directed, weighted graph. The *activation function* defines the output of a neuron

from a given set of inputs. The characteristic feature of the activation function is that a small change in input produces a small change in output (i.e. it must be smooth). As a simple example, take one of the earliest models of an artificial neuron, the binary threshold unit, proposed by MucCulloch and Pitts in 1943 [25, 17].

This neuron computes a weighted sum of its n input signals, x_i, where $i = 1, 2, ..., n$, and generates an output of 1 if this sum is above a certain threshold T, or 0 otherwise. Mathematically,

$$y = \theta\Big(\sum_{i=1}^{n} W_i x_i\Big) \tag{3}$$

where θ is a linear step function at threshold T, W_j is the weight value associated with the jth input, and y is the binary output. The McCulloch-Pitts neuron has been generalized in many ways, the most obvious being the use of activation functions other than the threshold function (e.g. piecewise linear, sigmoid, Gaussian, etc.). A bias term can be added to the result of any activation function.

2.1.2 Network Architectures

Neurons are often organized into multiple layers. This is characteristic of deep learning. Typically, neurons of one layer connect only to neurons of the immediately preceding and immediately following layers. The input (first) layer receives external data. The output (final) layer produces the ultimate result. In between are zero or more hidden layers. Between two layers, many neuron-connection patterns are possible. For example, in fully connected layers every neuron in one layer connects to every neuron in the next layer. In pooling layers, a group of neurons in one layer connects to a single neuron in the next layer, thereby reducing the number of neurons in that layer. Based on the connection pattern (architecture), ANNs can be grouped into two categories (Figure 2): *feedforward* networks, in which neuron connections form a directed acyclic graph, and *recurrent* networks (i.e. feedback networks) that contain cyclic connections between neurons in the same or previous layers.

In feedforward neural networks (FFNNs), information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. Neuron connections do not form a cycle. Generally, feedforward networks are static and

memory-less: they produce only one set of output values for a given input (rather than a sequence of values), and the output is independent of the previous state(s) of the network.

The simplest type of FFNNs is the single-layer perceptron (SLP), which consists of a single layer of output nodes. In the SLP network, the inputs are fed directly to the outputs via a series of weights. The sum of the products of the weights and the inputs is calculated in each node (3). If the value is above some threshold, the neuron fires and takes the activated value. Otherwise, it takes the deactivated value. SLPs may also use a continuous function instead of a step function. Regardless, SLPs are only capable of learning linearly separable patterns.

The most common class of FFNNs is the multilayer perceptron (MLP). The neurons of the MLP are organized into multiple (at least 3) fully-connected layers. This means that each neuron in one layer has directed connections to each the neurons in the next layer. With the exception of the input neurons, each neuron in the MLP uses a *nonlinear* activation function. If the MLP instead used a linear activation function, any number of layers could be reduced to a two-layer input-output model using linear algebra. Two historically common activation functions are the hyperbolic tangent and the logistic functions:

$$y(x_i) = \tanh(x_i) \tag{4}$$

$$y(x_i) = \frac{1}{1 + e^{-x_i}} \tag{5}$$

The MLP's nonlinear activation allow it to distinguish data that is not linearly separable.

Unlike feedforward networks, recurrent neural networks (RNNs) can use their internal state (i.e. memory) to process sequences of inputs. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state. The connections between the nodes of a RNN therefore form a directed graph along a temporal sequence, allowing these networks to exhibit temporal, dynamic behavior.

It *is* possible to solve temporal problems using feedforward structures. Using Takens's delay embedding theorem [34], one can reconstruct a chaotic dynamical system from a sequence of observations of the state of that system. This can convert a temporal problem into

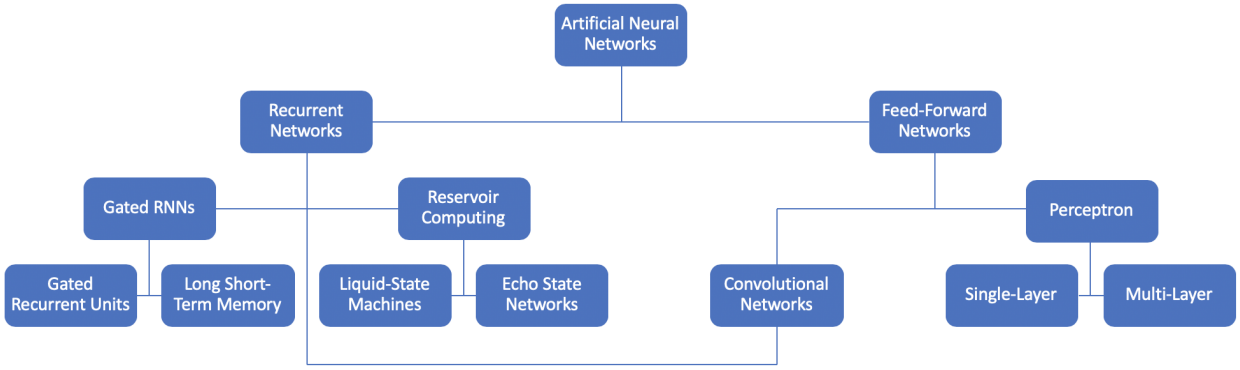


Figure 2: ANN Architectures Map

a spacial problem. However this is not a natural way to represent time, and can be costly. For example when a long time delay is introduced, many parameters are needed to model this large, "empty space."

To solve this problem, one can add recurrent connections, transforming the system into a (potentially) complex dynamical system. This recurrence opens the door to many temporal, real-world applications such as prediction, identification, adaptive filtering, noise reduction, speech recognition, and more.

2.1.3 Supervised Learning

In the context of an ANN, learning is the problem of updating the network architecture and connection weights so that the network can more effectively perform a specific task. In short, this is done by minimizing the network's observed error. Performance is improved over time by iteratively updating the weights in the network based on available training (input data) patterns. The ANN's ability to learn autonomously sets it apart from traditional systems. Instead of following a set of specified rules, ANNs (attempt to) discern underlying input-output relationships from a given collection of representative examples. Learning (i.e. training) is complete when providing the network additional data no longer reduces the error rate. "Error" is calculated by defining a loss-function that is periodically evaluated during

training. Even after training, the error rate rarely reaches zero. It is therefore up to the user to determine what error rate is acceptable, and whether the network should be redesigned. The information available to the network during learning is the learning paradigm. The procedure through which learning rules are used to update the network's weights is the learning algorithm.

There are three main learning paradigms: supervised, unsupervised, and hybrid. During supervised learning, the network is provided with a "correct answer" (output) for every input pattern. The connection weights are then modified so that the network produces answers as close as possible to the known correct answers. Unsupervised learning does not require a correct answer associated with each input pattern in the training data set. Instead, the network explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations. Hybrid learning combines supervised and unsupervised learning. Part of the weights are usually determined through supervised learning, while the others are obtained through unsupervised learning.

Formally, the supervised learning setup is, given an input domain $\mathcal{X}$[1] and an output domain $\mathcal{Y}$, and a training set $\mathcal{D} = \{(x_1, y_1), ..., (x_m, y_m)\}$ of training pairs $(x_m, y_m) \in \mathcal{X} \times \mathcal{Y}$ with $m = 1, ..., M$ of training inputs x_m and target outputs y_m as well as a new unclassified input $\tilde{x} \in \mathcal{X}$, predict the corresponding output $\tilde{y} \in \mathcal{Y}$ [31].

2.1.3.1 Linear Regression

Mathematically speaking, input-output pairs are generated by an unknown function $y = f(x)$. Therefore, another way to think about prediction is the supervised model seeking a function $\tilde{y} = h(x)$ that approximates the model function f. For this 'deterministic model' to work, it is assumed that similar inputs have similar outputs.

Consider classical linear regression based on a deterministic linear model function,

$$f(x; w) = w^T x + w_0 \tag{6}$$

[1]For our purposes $\mathcal{X}$ is chosen to be $\mathbb{R}^N$ of real N-dimensional input vectors (i.e. feature vectors), or for binary variables, $\{0, 1\}^N$ of N-bit binary strings. $\mathcal{X}$ can also be $\mathbb{C}^N$ for certain applications.

where $w \in \mathbb{R}^N$ contains the trainable parameters, $x \in \mathbb{R}^N$ is the input, and w_0 is an optional bias. We want to solve for w. Define design matrix X with rows that correspond to each of the x_i's. If $X^\dagger X$ is of full rank, there exists closed-form solution,

$$w = (X^\dagger X)^{-1} X^\dagger y = X^+ y \tag{7}$$

where X^+ is the Moore-Penrose pseudoinverse of X. However, if the data is not linearly separable (which is likely), we need to first perform a transformation (i.e. feature map) $\phi : \mathcal{X} \to \mathcal{F}$ into a Hilbert space $\mathcal{F}$ where the data *is* linearly separable.

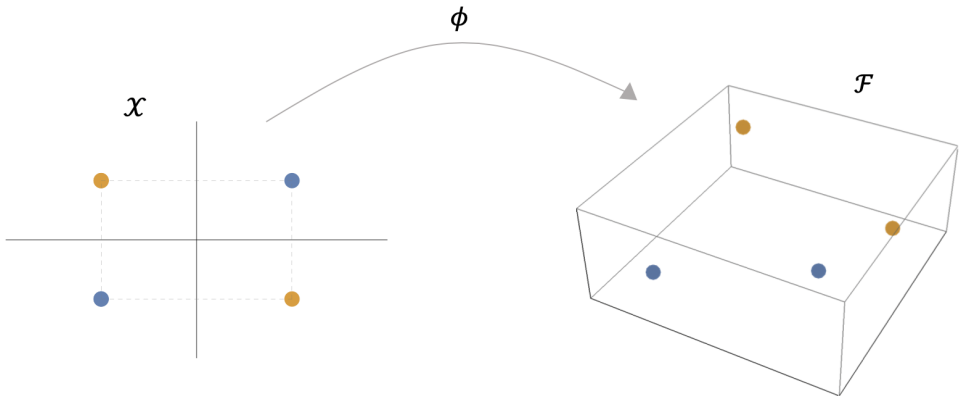


Figure 3: Data linearly non-separable in $\mathcal{X}$ feature mapped to a space $\mathcal{F}$ is now linearly separable

After applying the feature map onto the input data $\{\phi(x_i)\}$ we can perform linear regression within the output space.

2.1.3.2 Nonlinear Regression

To choose an effective feature map we need to know something about the data beforehand, which won't always be the case. This motivates the use of nonlinear regression, more descriptively named adaptive feature mapping, whereby a nonlinear function is applied to both the inputs *and* parameters,

$$f(x; w) = \varphi(w, x) = \varphi(w^T x) \tag{8}$$

Mathematically, neural networks can be viewed as a nonlinear regression model where model function φ is called the activation function. In (8) φ is chosen such that the input to the nonlinear functions is a linear model. In feed forward neural networks this expression is composed many times,

$$f(x; W_1; ...; W_L) = \varphi_L(W_L(\varphi_{L-1}...\varphi_1(W_1(x)))) \tag{9}$$

where W_i $(i = 1, 2, ..., L)$ are the parameters summarized as weight matrices for a network with L hidden-layers. Neural networks learn through backpropogation, a linear optimization algorithm that seeks to find the minimum of a loss function (e.g. a quadratic difference between the produced outputs and target outputs of the model) using gradient descent. Gradient descent updates the learnable parameters (weights) of the neural network at each training iteration, which, for a large number of training parameters, can be costly. Reservoir computing (RC) proposes a promising solution to this problem that makes it well suited for quantum computing.

2.1.3.3 Fully Tunable Networks

Fully tunable neural networks allow for the of tuning parameters inside of the feature map. This is most commonly done through a learning algorithm called gradient descent. Gradient descent is a first order linear optimization algorithm that seeks to find the minimum of a function. Using gradient descent, the learnable parameters (e.g. weights) of the neural network are updated at each training iteration in the direction opposite to the gradient of the loss function at the current point. The step size taken towards the local minimum at each iteration is determined by the learning rate, α. By following the direction of the slope "downhill" the network parameters are updated such that the error-rate is brought to a local minimum. For a loss function (i.e. error function) $E(p)$, parameter p is updated according to

$$p_{n+1} = p_n - \alpha \nabla_p E(p_n) \tag{10}$$

There are many gradient descent variants. (10) represents "vanilla," or batch gradient descent, which calculates the gradients for the whole training data set before performing an

update. This variant is slow and redundant. Stochastic gradient descent (SGD), in contrast, performs an update for each training input x_i and expected output y_i (assuming supervised learning):

$$p_{n+1} = p_n - \alpha \nabla pE(p_n; x_i; y_i) \tag{11}$$

While SGD is much faster than batch gradient descent, and eliminates redundancy, its frequent updates with high (fixed) variance often cause it to overshoot the minimum, sometimes never converging. Momentum is a method that helps accelerate SGD in the appropriate direction, while also dampening its oscillations once it approaches a minimum. It does so by retaining a fraction γ of the $t-1$ (previous) update:

$$v_t = \gamma v_{t-1} - \alpha \nabla pE(p_n) \tag{12a}$$

$$p_{n+1} = p_n - v_t \tag{12b}$$

Momentum can be viewed physically as a ball rolling down a slope. The momentum term increases for dimensions whose gradients point in the same directions, and reduces updates for dimensions whose gradients change directions. As a result, there is faster convergence and reduced oscillation.

The term backpropagation (i.e. backprop, or BP) refers only to the algorithm for computing the gradient. However, it is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as in SGD. BP is so named because to calculate the derivative in a multi-layer network, you must use the chain rule from the last layer (directly connected to the loss function, provides the prediction) to the first layer (takes the input data). In other words, you are "moving from back to front."

2.1.4 Reservoir Computing

The dynamic memory and high adaptability of RNNs postures them as a promising tool for many temporal, real-world applications. In addition, RNNs are the closest machine learning model to a biological brain, so understanding under what conditions they perform most optimally has massive implications for understanding natural, human computation

and learning. However, their complex, recurrent structure makes them inherently difficult to train. BP methods have been applied to RNNs [1] with varying success: Single neurons with both recurrent and forward connections can cause non-convergence during gradient descent. However, even with convergence, training RNNs using BP has shown to be slow and costly because of the large number of learning parameters. Reservoir computing (RC) is an extension of neural networks that proposes a promising solution to this problem in which optimal performance can be achieved without training over *all* of the network weights. Instead, only the final, linear readout layer is trained.

In RC, the input signal is connected to a fixed (non-trainable) and random dynamical system (the reservoir), thus creating a higher dimension representation (embedding) through complexity of a constant feature map. This embedding is then connected to the desired output via trainable units. The non-recurrent equivalent of reservoir computing, used in this paper, consists of a feed forward network in which only the readout layer is trainable.

This framework has the potential to use the computational power of naturally available systems, setting it apart from FFNNs. Liquid-state machines (LSMs) and echo state networks (ESNs) are the two main types of reservoir computing (Figure 2). Section (A.4) gives an in-depth tutorial on the implementation and application of ESNs.

2.2 Quantum Circuits

2.2.1 Quantum Fourier transform

One of the most useful ways of solving a problem in physics, mathematics, or computer science is to transform it into some other problem for which the solution is known. One such transformation is the discrete Fourier transform (DFT). The DFT occurs in many different versions throughout classical computing in areas ranging from signal processing to data compression to complexity theory. The quantum Fourier transform (QFT) is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction [3]. It can be carried out by an extremely efficient quantum circuit built entirely out of one-qubit

and two-qubit gates.

The QFT on an orthonormal basis $|0\rangle, ..., |N-1\rangle$ is defined to be a linear operator with the following action on the basis states:

$$|j\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} x_j e^{2\pi i jk/N} |k\rangle . \tag{13}$$

Equivalently, the action on an arbitrary state may be written

$$\sum_{j=0}^{N-1} x_j |j\rangle \longrightarrow \sum_{k=0}^{N-1} y_k |k\rangle , \tag{14}$$

where the amplitudes y_k are the DFT of the amplitudes x_j. As shown in (13) and (14), the QFT transforms computational basis states into equal superposition states modulated with a phase. This is a unitary transformation, and thus can be implemented as the dynamics for a quantum computer. In the context of qubits $N = 2^n$, where n is some integer, and the basis $|0\rangle, ..., |2^n - 1\rangle$ is the computational basis for an n qubit quantum computer. The QFT's circuit representation involves a combination of Hadamard and controlled rotation gates:

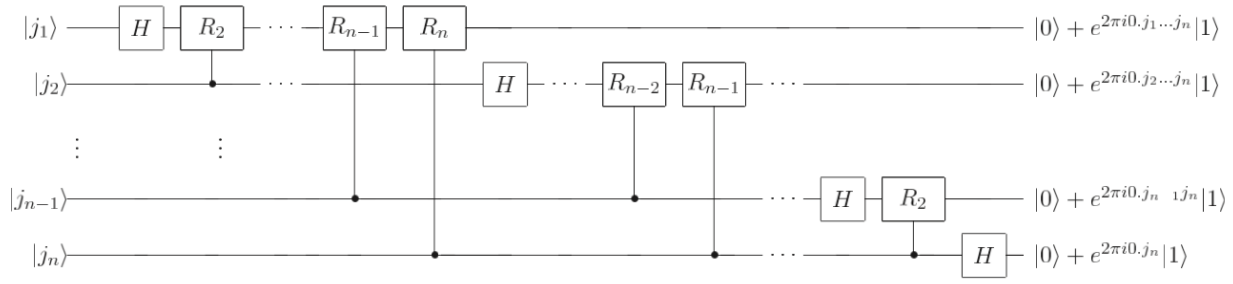


Figure 4: Quantum Fourier transform circuit [3]

The QFT has many fascinating applications. Shor's factorization algorithm uses the QFT to estimate a period. In a similar vein, we will use the QFT for phase estimation through eigenvalue solving.

2.2.2 Quantum Phase Estimation

The problem of QPE is as follows. Given a unitary operator U with eigenvalues and eigenvectors

$$U\left|u\right\rangle = e^{2\pi i\phi}\left|u\right\rangle, \tag{15}$$

estimate the phase, ϕ. The QPE procedure uses two registers. The first register contains an ancilla of t qubits. A greater t increases the accuracy of the estimate for ϕ, but decreases the probability that the procedure will be successful. The second register begins in the state $|u\rangle$, and contains as many qubits as is necessary to store $|u\rangle$.

Phase estimation can now be performed in two steps. First, we apply a Hadamard transformation to the first register, followed by application of controlled-U operations on the second register, with U raised to successive powers of two.

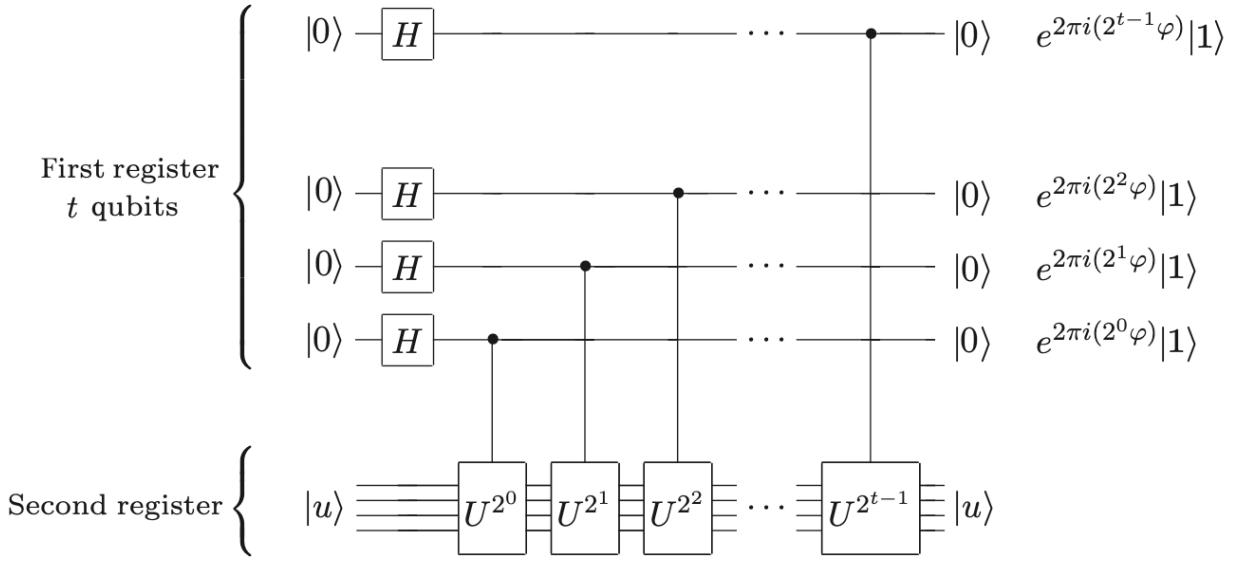


Figure 5: The first step of QPE [29]

The second step is to apply the inverse QFT on the first register. After these operations are complete, we read out the state of the first register by doing a measurement in the computational basis, thus obtaining an estimate for ϕ. An extended description of the QPE algorithm can be found in Section (A.2).

3.0 Variatonal Quantum Classifier

Variational quantum computing is a method that combines classical computing and quantum computing in order to use imperfect near-term quantum computers to solve problems "faster" than purely classical machines. Variational quantum algorithms provide a framework for trying to make use of the "along-the-way" quantum computers. For example, Peruzzo et al. [30] combined a highly reconfigurable photonic quantum processor with a conventional computer to create a variational eigenvalue solver with greatly reduced coherence time requirements.

A variational quantum algorithm consits of the preparation of a fixed initial state (e.g. the vacuum state), a quantum circuit $U(\theta)$ parameterized by a set of free parameters θ, and measurement of an observable $\hat{B}$ at the output. Typically, the expectation values $f(\theta) = \langle 0| U^{\dagger}(\theta)\hat{B}U(\theta) |0\rangle$ defines a scalar cost for a given task. The variational circuit is trained by a classical optimization algorithm that makes queries to the quantum device. The free parameters $\theta = (\theta_1, \theta_2, ...)$ of the circuit are tuned iteratively to optimize the cost function.

Havlíček et al. [14] explored two methods for applying noisy intermediate-scale quantum (NISQ) devices to machine learning. The first method, the quantum variational classifier, uses a variational quantum circuit to classify data. The second method, a quantum kernel estimator, estimates the kernel function on the quantum computer and optimizes a classical SVM. A key component in both methods is the use of the quantum state space as a feature space.

The following sections will detail the application of variational quantum computing to machine learning through the example of the Havlíček et al.'s variational quantum classifier. The proposed algorithm takes on the original supervised learning problem: we are given data from a training set T and a test set S of a subset $\Omega \subset \mathbb{R}^d$. Both are assumed to be labelled by a map m: $T \cup S \longrightarrow \{+1, -1\}$ unknown to the algorithm. The goal is to approximate the map $\tilde{m} : S \to \{+1, -1\}$ such that is agrees with high probability with the true map $m(\mathbf{s}) = \tilde{m}(\mathbf{s})$ on the test data $\mathbf{s} \in S$. A classical approach to this problem uses support vector machines (SVMs) [35]. A quantum version of this approach has already been

proposed by Farhi et al. [7], where an exponential improvement can be achieved if data is provided in a coherent superposition. However, when data is provided from a classical computer then Farhi et al.'s method does not yield such a speed-up.

3.1 Data Encoding via Feature Map

In the quantum setting, the feature map is an injective encoding of classical information $\mathbf{x} \in \mathbb{R}^d$ into a quantum state $|\Phi\rangle\langle\Phi|$ on an n-qubit quantum register. There are many possible choices of feature maps. Though, to obtain a quantum advantage we want the map to give rise to a kernal $K(\mathbf{x}, \mathbf{y}) = |\langle\Phi(\mathbf{x})|\,|\Phi(\mathbf{y})\rangle|^2$ that is hard to estimate on a classical computer. Havlíček et al. [14] define such a unitary feature map on $n-$qubits, where H denotes the conventional Hadamard gate:

$$|\Phi(\mathbf{x})\rangle = \mathcal{U}_{\Phi(\mathbf{x})}\,|0\rangle^{\otimes n} = U_{\Phi(\mathbf{x})}H^{\otimes n}U_{\Phi(\mathbf{x})}H^{\otimes n}\,|0\rangle^{\otimes n} \tag{16}$$

where,

$$U_{\Phi(\mathbf{x})} = \exp\left(i\sum_{S\subseteq[n]}\phi_S(\mathbf{x})\prod_{i\in S}Z_i\right) \tag{17}$$

where the 2^n coefficients $\phi(\mathbf{x}) \in \mathbb{R}$ are now non-linear functions of the input data $\mathbf{x} \in \mathbb{R}^n$.

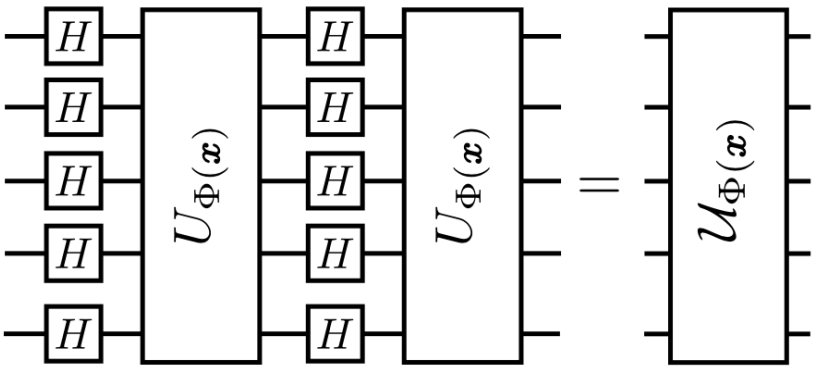


Figure 6: Circuit representation of feature map [14]

In general, maps with low degree expansions $|S| \leq d$ can be implemented efficiently. Havlíček et al. experimentally implemented their algorithm on a superconducting processor, so chose unitaries $U_{\Phi(\mathbf{x})}$ with interactions that are present in the actual connectivity graph[1] of the superconducting chip $G = (E, V)$. This ensured that the feature map could be generated from a short depth[2] circuit. For example, considering only Ising type interactions i.e. $d = 2$, the resulting unitary can be generated from one- and two-bit gates of the form

$$U_{\phi_{\{k\}}(\mathbf{x})} = \exp\left(i\phi_{\{k\}}(\mathbf{x})Z_k\right) \tag{18a}$$

$$U_{\phi_{\{l,m\}}(\mathbf{x})} = \exp\left(i\phi_{\{k,l\}}(\mathbf{x})Z_kZ_l\right) \tag{18b}$$

which leaves $|V| + |E|$ real parameters to encode the data. A feature map unitary satisfying $n = d = 2$ for an input $\mathbf{x} = [\pi/2, 3\pi/2]$ expressed in terms of its cartan decomposition (using a minimal number of CZ gates) is shown in Figure (7),

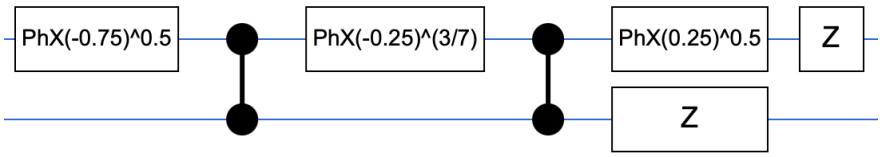


Figure 7: Two-qubit $U_\phi(\mathbf{x})$ implementation example

where PhX is the phased X gate.

3.2 Short-depth Discriminator Circuit

Following the feature map, we perform the classifier step of the variational algorithm. To define a seperating hyperplane we use a short-depth circuit comprised of single qubit and

[1] A graph is a pair $G = (E, V)$, consisting of a set of vertices V and a set of unordered pairs of vertices $E \subseteq V^{(2)}$ called edges.

[2] Near-term quantum machines can only run short gate sequences, since without fault tolerance every gate increases the error in the output.

multi-qubit gates:

$$W(\theta) = U_{loc}^{(l)}(\theta_l)U_{ent}...U_{loc}^{(2)}(\theta_2)U_{ent}U_{loc}^{(1)}(\theta_1) \tag{19}$$

We apply a circuit of l repeated entanglers as shown in Figure 8.b and interleave them with layers comprised of local single qubit rotations:

$$U_{loc}^{(t)}(\theta_t) = \otimes_{m=1}^{n} U(\theta_{m,t}) \tag{20}$$

$$U(\theta_{m,t}) = \exp\left(i\frac{1}{2}\theta_{m,t}^{z}Z_m\right)\exp\left(i\frac{1}{2}\theta_{m,t}^{y}Y_m\right) \tag{21}$$

parameterized by $\theta_t \in \mathbb{R}^{2n}$ and $\theta_{i,t} \in \mathbb{R}^2$. For this feature map that we use entangling unitaries comprised of products of controlled-Z phase gates $CZ(i,j)$ between qubits i and j. The entangling interations follow the interaction graph $G = (E, V)$ that we used to generate the feature map.

$$U_{ent} = \prod_{(i,j)\in E} CZ(i,j) \tag{22}$$

This short-depth circuit can generate any unitary if sufficiently many layers d are applied.

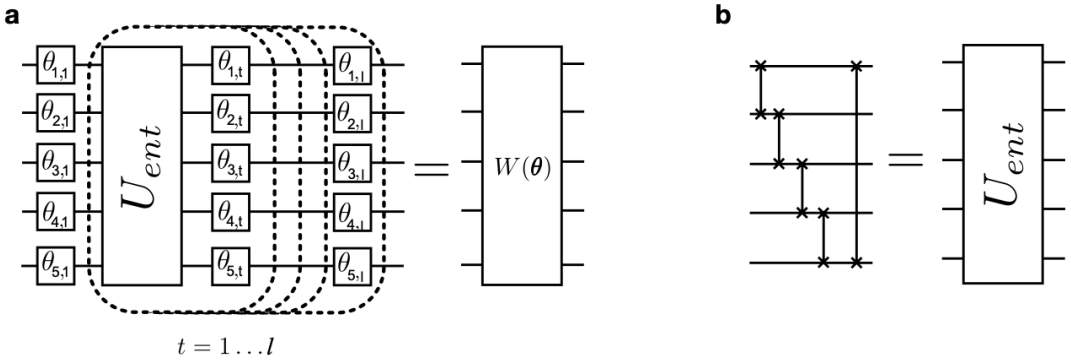


Figure 8: (a) Short depth variational circuit (b) Entangling gate [14]

A short-depth discriminator satisfying $n = d = 2$ for parameters

$$\theta = [\pi/4, \pi/2, 3\pi/4, \pi, 5\pi/4, 3\pi/2, 7\pi/4, 2\pi]$$

is shown in Figure (8):

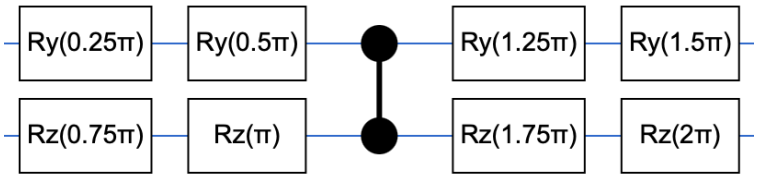


Figure 9: Two-qubit $W(\theta)$ implementation example

3.3 Training and Classification

The overall goal of the quantum variational classifier is to find an optimal classifying circuit $W(\theta)$ (19) that separates the input dataset into different labels. For the classification phase we will exploit the large dimensional Hilbert space of the quantum processor to find the optimal separating hyperplane (e.g. Figure 22) with a similar procedure to that used by Support Vector Machines (SVM). Here we will limit our discussion to binary classification tasks, although this algorithm can be extended to multi-label classification.

First, we need to define a cost function such that the optimization procedure minimizes the probability of assigning the wrong label after having constructed a distrubtion after R measurements. Thus, we define the empirical risk $R_{\text{emp}}(\theta)$ in terms of the error probability $\Pr(\tilde{m}(\mathbf{x} \neq m(\mathbf{x}))$ of assigning the incorrect label, based on the decision rule $\tilde{m}(\mathbf{s}) = \text{argmax}_y\{\hat{p}_y(\mathbf{s})\}$, averaged over the samples in the training set T.

$$R_{\text{emp}}(\theta) = \frac{1}{|T|} \sum_{\mathbf{x} \in T} \Pr(\tilde{m}(\mathbf{x}) \neq m(\mathbf{x})). \tag{23}$$

The probability that the label $m(\mathbf{x}) = y$ is assigned incorrectly is approximated by [14]

$$\Pr(\tilde{m}(\mathbf{x}) \neq m(\mathbf{x})) \approx \text{sig}\left(\frac{\sqrt{R}\left(\frac{1}{2} - \left(\hat{p}_y(\mathbf{x}) + \frac{yb}{2}\right)\right)}{\sqrt{2(1 - \hat{p}_y(\mathbf{x}))\hat{p}_y(\mathbf{x})}} \right) \tag{24}$$

where $\text{sig}(x) = (1 + e^{-x})^{-1}$ is the sigmoid function. The parameters θ can be trained using any optimization algorithm; the authors used simultaneous perturbation stochastic approximation (SPSA). Through the chosen optimization algorithm we iteratively update

the θ's until $R_{emp}(\theta)$ has converged, or a maximum number iterations is reached.

Once the training phase is complete we move to the classification phase, where the optimal parameters are used to label for new input data. The same circuit (Figure 10 is applied, but this time, the parameters are fixed. We take R measurements to create a distribution $\hat{p}_y(\mathbf{x})$, and return the label $y = \text{argmax}_y\{\hat{p}_y(\mathbf{x})\}$.

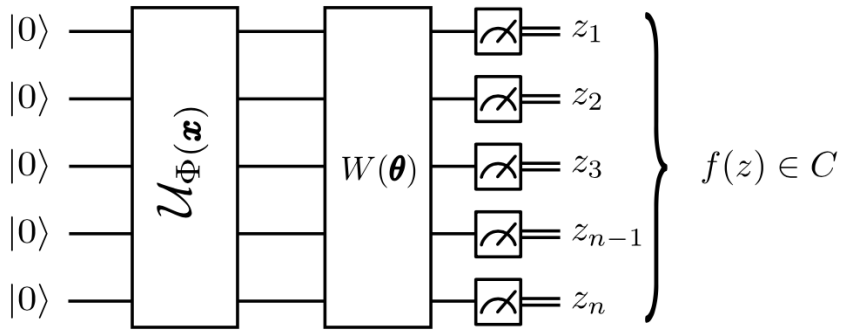


Figure 10: Quantum variational classification circuit [14]

We verified the effectiveness of Havlíček et al. supervised learning model for binary classification using a variational circuit satisfying $n = 2 = d$. We began with binary classification on the number-line. We chose a number $\alpha \in (0, 1)$, and created two labeled sets a, b consisting of data points $x_a > \alpha$ and $x_b < \alpha$. After training, the model was given an unseen data point $\tilde{x} \in \mathbb{R}^1$ and asked to sort it into set a or b. Our trained model did so correctly with a probability of 1. Next, we performed binary classification on the unit-circle. An angle $\theta \in [0, 2\pi)$ was chosen, and corresponding vectors $\theta_a = \cos\theta \cdot \hat{x} + \sin\theta \cdot \hat{y}$, $\theta_b = \cos\theta \cdot \hat{x} - \sin\theta \cdot \hat{y}$ were created, dividing the X-Y plane into two equal halves. Around these vectors, we generated two sets of data a, b as in Figure (11). After training, the model was given an unseen data point $\tilde{x} \in \mathbb{R}^2$ and asked to sort it into set a or b. Our trained model did so correctly with a probability of 1. Lastly, we performed binary classification in the Bloch sphere. Two random vectors θ_a, θ_b in the X-Z plane were chosen. Around these two vectors, we randomly sampled two sets a, b of quantum data points as in Figure (12). The binary task, analogous to the previous two tasks, was to learn to distinguish the two

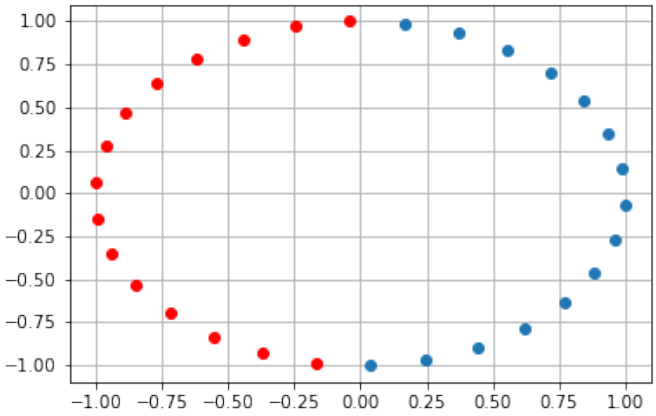


Figure 11: Quantum data represented on the unit-circle. States in category a are blue. States in category b are red. Parameters: $\theta = \pi/2$, $N = 30$.

sets. After training, the model was given an unseen data point $\tilde{x} \in \mathbb{R}^3$ and asked to sort it

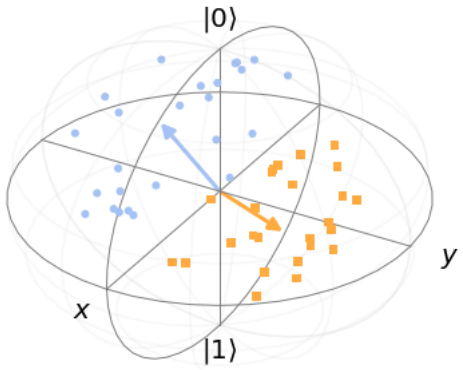


Figure 12: Quantum data represented on the Bloch sphere. States in category a are blue. States in category b are orange. The vectors are the states around which the samples were taken. Parameters: $\theta_a = 1$, $\theta_b = 4$, $N = 50$. Image generated using [16].

into set a or b. Our trained model did so correctly with a probability of 0.98. Havlíček et

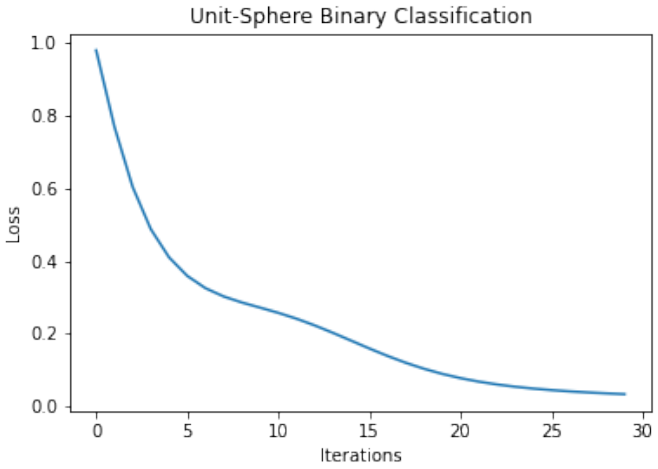


Figure 13: Loss function for Bloch sphere binary classification task. Metrics: N = 50, training iterations = 30, final loss = 6e−2, test accuracy = 98%.

al. report a success rate of 100% for the equivalent task. However, they used 250 training iterations while our model used only 30 training iterations. Figure (13) shows that for this number of iterations our training terminated with a non-zero loss. For a larger number of iterations, our model may have continued to converge and thus yielded a higher success rate. Regardless, this testing verified the effectiveness of the proposed variational circuit on binary classification for $n = d = 2$ and, for all intents and purposes, echoed the authors' results.

4.0 Quantum Reservoir Computing

Quantum machines must be error corrected. This makes a QRC model appealing because the reservoir is stochastically generated and fixed for the entire computation, thus limiting the quantum circuit depth. By contrast, conventional neural networks train large numbers of parameters distributed over multiple hidden layers, requiring an equally large quantum circuit depth and therefore more error correction. QML models are differentiated from classical models by the addition of encoding (creating a qubit representation of the classical data and possibly applying some unitary evolution) and readout (some form of a measurement) steps. Our discussions in this section assume that both of these steps can be done perfectly, without error.

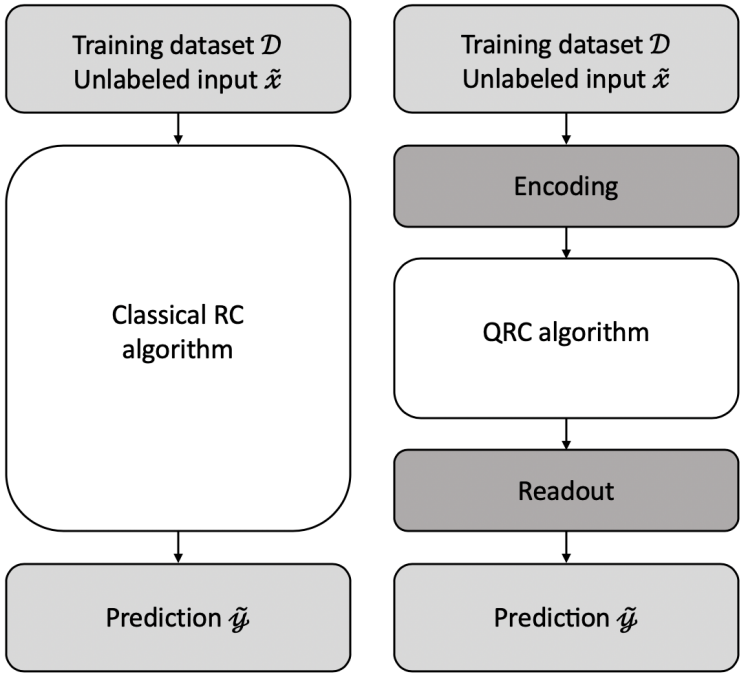


Figure 14: QRC using classical data requires additional (highly non-trivial) encoding and readout steps. Diagram inspired by [31].

The QRC algorithm as a whole can be split into three main components: input and state

preparation (i), quantum reservoir action (ii), and quantum linear regression and prediction (iii). In the following overview we assume that we start with a n-qubit system in the ground state $|0...0\rangle$ and that data is accessible from a classical memory. We will consider training data sets in the same format as those outlined in Section (2.1.3): M input vectors $\{x_i \in \mathbb{R}^N\}$ with M corresponding labels $\{y_i \in \mathbb{R}\}$ as well as an unlabeled input $\tilde{x} \in \mathbb{R}^N$.

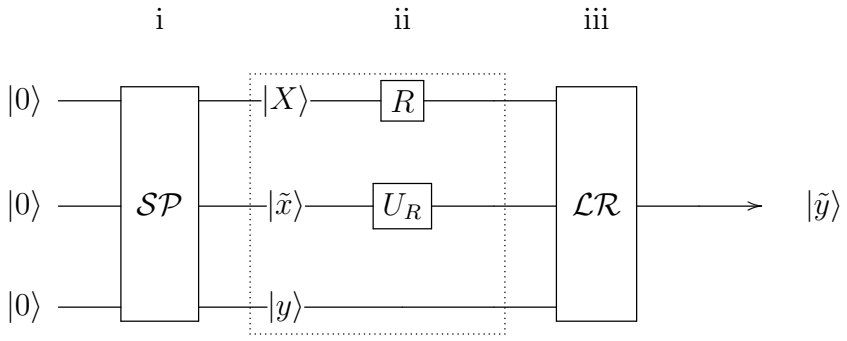


Figure 15: QRC schematic

4.1 State Preparation

During state preparation we encode each vector of the supervised learning model into the amplitudes of a (normalized) quantum state,

$$|\psi_X\rangle = \sum_{m=1}^{M} \sum_{i=1}^{N} x_i^{(m)} |i\rangle |m\rangle \tag{25}$$

$$|\psi_{\tilde{x}}\rangle = \sum_{i=1}^{N} \tilde{x}_i |i\rangle \tag{26}$$

$$|\psi_y\rangle = \sum_{m=1}^{M} y^{(m)} |m\rangle \tag{27}$$

Using this state preparation scheme (i.e. amplitude encoding) we only need $n = \log_2(MN)$ qubits to encode a training data-set of M inputs with N features each. The training outputs are encoded in the amplitudes of a separate quantum register, as are the unlabeled inputs

(Figure 15, i). State preparation using amplitude encoding (if done qubit-efficiently [2]) has asymptotic complexity $\propto \mathcal{O}(MN)/\mathcal{O}(log_2(MN))$. However, this does account for the original loading of features from memory hardware, which takes time linear in NM.

The routine for arbitrary state preparation with a qRAM succeeds only with probability $p_{\text{acc}} = \sum_i |x_i|^2/N$ depending on the state to encode [31], which means in the worst case we would have to repeat measurement $\mathcal{O}(N = 2^n)$ times to prepare the correct state. There are many other ways to state prepare beyond quantum circuits, which our group may investigate in the future. As mentioned in Sections (5.3) and (A.1), exponential speedups from qubit-efficient state preparation is still a topic of ongoing work in the world quantum computing.

4.2 Quantum Reservoir

In the classical reservoir computing paradigm the stochastically generated reservoir implements nonlinear activation. In the case of the quantum reservoir, design and function is entirely dependent upon choice of quantum nonlinearity. For instance, nonlinearity through classical preprocessing (Section (5.3)) would eliminate the need for explicit reservoir nonlinearity[1]. This would be convenient for simulation, as circuit-based reservoirs are naturally linear in the amplitudes of states. In this case one could implement a block diagonal unitary quantum reservoir, operating in constant time,

$$R = \begin{bmatrix} U_R & & 0 \\ & \ddots & \\ 0 & & U_R \end{bmatrix} \tag{28}$$

The reservoir acts on each of the N features of the training data individually to produce output of dimension D. For a one-qubit Hilbert space $\mathcal{F}$ [23],

$$R : \mathcal{F}^{\otimes \log_2(N)} \rightarrow \mathcal{F}^{\otimes \log_2(D)}$$

Each block U_R acts on $\log_2(D)$ qubits. Applying the reservoir to state prepared design matrix (25),

[1]Here, the term 'quantum reservoir' would encompass 'input + unitary reservoir', maintaining (for clarity) the nonlinearity and high dimensional mapping characteristic of the reservoir from the traditional RC model.

$$|\psi_X\rangle \text{ —}\boxed{R}\text{— } |\psi'_X\rangle$$

where

$$|\psi'_{X_i}\rangle = U_R\,|x_i\rangle \tag{29}$$

Encoded unlabeled data $|\tilde{x}\rangle$ is acted on by U_R in the same way, and encoded label vector $|y\rangle$ is not acted upon inside of the reservoir (Figure 15, ii). U_R should be sufficiently complicated to preserve linear separability achieved during high dimensional nonlinear feature mapping.

The unitary reservoir is appealing for simulation, but is not the only option. Choice of quantum nonlinearity will, in the end, dictate reservoir action. Apart from the options outlined in Section (5) time dynamical reservoir designs (e.g. Lindblad evolution [6]) and non-circuit based nonlinear quantum dynamics (e.g. photonic implementations) are appealing topics of future work.

4.3 Quantum Linear Regression

Quantum representations of the training data have now gone through some quantum or hybrid classical/quantum nonlinear transformation and can be used to set up an eigenvalue equation, $A\,|x\rangle = |b\rangle$. With this, we can use techniques from Harrow *et al.* (HHL) [13] (see Section (A.3)) and Lloyd *et al.* [20] to extract and invert singular values, and execute inner products to write a prediction.

To extract the eigenvalues λ_r of $X^\dagger X$ to eigenvectors $\vec{w}$ (these are the weights (7)) we invert the singular values of design matrix X, and use the HLL inversion procedure. To access the eigenvalues we make copies of (25), ignoring the $|i\rangle$ register, to obtain mixed state

$$\rho_{X^\dagger X} = \mathrm{tr}_m\{|\psi_X\rangle\,\langle\psi_X|\} \tag{30}$$

$$\rho_{X^\dagger X} = \sum_{m,m'=1}^{M}\sum_{i=1}^{N} x_m^{(i)} x_{m'}^{(i)*}\,|m\rangle\,\langle m'| \tag{31}$$

Next we apply $\rho_{X^\dagger X}$ to $|\psi_X\rangle$ using Hamiltonian simulation[2] [4], and then use quantum phase estimation (QPE) [20] to encode the eigenvalues λ_r in the qubits of a new register. The eigenvalues are inverted via controlled rotation[3] [13]. Upon success we uncompute and discard the eigenvalue register, leaving us with a quantum representation $|\psi_w\rangle$ of weight vector $\vec{w}$. Finally, we execute the inner product of the weight vector representation and unlabeled data vector representation using $|\psi_1\rangle \equiv |\psi_w\rangle$ and $|\psi_2\rangle \equiv |\psi_y\rangle\,|\psi_{\tilde{x}}\rangle$. To ensure we end with the correct sign we prepare the state

$$\frac{1}{\sqrt{2}}(|\psi_1\rangle\,|0\rangle + |\psi_2\rangle\,|1\rangle), \tag{32}$$

and trace out all of the registers except for the ancilla. The off-diagonal elements of the ancilla's reduced density matrix contain the desired prediction, $\tilde{y}$ [32].

Classical linear regression takes time $\propto \mathcal{O}(N)$, while quantum linear regression via this HHL-based algorithm performs training and prediction $\propto \mathcal{O}(\text{poly}(\log N))$, thus offering an exponential speedup. In order to prevent a readout of the entire density matrix, the final output layer needs to be executed 'quantumly' instead of classically, likely through measurement. Unfortunately, this HLL-based algorithm has a probability of failure i.e. a measurement of $|0\rangle$. So retrieving a nonzero measurement could require many repetitions of the entire routine, cutting into the quantum advantage.

4.3.1 Time Complexity Analysis

In quantum computing an efficient algorithm has polynomial runtime with respect to the number of qubits. State preparation using amplitude encoding (if done qubit efficiently) requires time $\propto \mathcal{O}(N)/\mathcal{O}(\log(N))$. A unitary reservoir operates in $\mathcal{O}(1)$ time. Quantum linear regression via HHL requires time $\propto \mathcal{O}(\text{poly}(\log N))$. However, loading in N features from the memory hardware and read-out of a N-dimensional computational result both require $\mathcal{O}(N)$ runtime. The read-in and read-out problems nullify any speed-up gained through quantum computation. With these factors considered, showing that, if data is

[2]It is not required that X be sparse.

[3]This operation is not unitary, so has some probability of failing.

loaded into the quantum computer with nonlinear features, one can execute a version of this algorithm in $\mathcal{O}(log(N))$ time, would constitute success.

5.0 Quantum Nonlinearity

A QRC model requires nonlinear activation (8). However, quantum mechanics is a linear theory; no matter how complicated the Hamiltonian state evolution is always first-order linear:

$$i\hbar\frac{\partial\rho}{\partial t} = [H, \rho] \tag{33}$$

Therefore, we need to design a circuit that, given an amplitude encoded state $|\psi\rangle$, outputs a state $|\psi'\rangle = |\varphi(\psi)\rangle$ amplitudes that are some nonlinear function of the input. Formally, for a quantum system with ensemble of pure states $\{p_i, |\psi_i\rangle\}$, the density operator for the system is defined by

$$\rho \equiv \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right| \tag{34}$$

We want some operator A such that $\rho \to A(\rho)$ is a nonlinear transformation [24],

$$A(\rho) = \sum_i \varphi(\rho_j)_i \left|\psi_i\right\rangle \left\langle\psi_i\right| \tag{35}$$

where φ is some nonlinear function. There are many ways to perform nonlinearity in quantum mechanics, but not all of them are practical in the context of a QRC algorithm.

5.1 Measurement Induced Nonlinearity

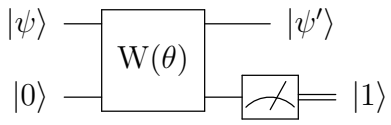
Projective measurement is nonlinear [24]. After performing a measurement described by measurement operators M_m the state is [29]

$$|\psi_i^m\rangle = \frac{M_m \left|\psi_i\right\rangle}{\sqrt{\left\langle\psi_i\right| M_m^\dagger M_m \left|\psi_i\right\rangle}} \tag{36}$$

We now have ensemble of states $|\psi_i^m\rangle$. This result is a nonlinear by $|\psi_i\rangle$, but is also probabilistic,

$$p(m|i) = \mathrm{tr}(M_m^\dagger M_m \left|\psi_i\right\rangle \left\langle\psi_i\right|) \tag{37}$$

so the output will vary upon repeated measurements. Consider a linear, unitary gate $\mathrm{W}(\theta)$ (for some classical parameters θ) applied to an input state $|\psi\rangle$ and ancilla qubit $|0\rangle$. One could use post-selection, only accepting the output state if, for instance, a $|1\rangle$ is measured,



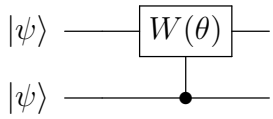
thus inducing nonlinearity on the first register [24]. The output of this circuit is,

$$|\psi'\rangle = \mathrm{W}(\theta)(|\psi\rangle \otimes |0\rangle) \tag{38}$$

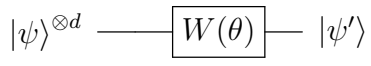
The drawback of this method is that a large quantity of measurement gates each requiring a specific result yields a low probability of success. The time complexity scales with the number of measurements required to ensure success, which could be costly. If the reservoir is somehow tailored such that the probability of a success is high, measurement induced nonlinearity could work.

5.2 Tensor Product Nonlinearity

Stringing together the same state via a tensor product $|\psi\rangle \to |\psi\rangle\otimes|\psi\rangle$ creates nonlinearity [24]. For example, we can use the tensor product trick to produce quadratic terms with the following circuit



For general d degree polynomial nonlinearity in the amplitudes of $|\psi\rangle$,



where

$$|\psi'\rangle = U(\theta) \bigotimes_{i=1}^{d} W_i \, |\psi\rangle \tag{39}$$

Unitary $W(\theta)$ mixes the basis vectors of the original state and $U(\theta)$ entangles the result. However, this presents a problem when applied to our amplitude encoded input states as it creates mixing of distinct training samples. If there exists a solution to this problem, tensor product nonlinearity could be another option.

5.3 Classical Preprocessing + QRAM

One could perform classical nonlinearity on the input vectors and then perform quantum encoding using qRAM [37] (Section (A.1)), essentially adding one step to the read-in process: $\vec{x} \rightarrow \text{qRAM} \rightarrow |\psi\rangle$ becomes

$$\vec{x} \rightarrow \varphi(\vec{x}) \rightarrow \text{qRAM} \rightarrow |\psi'\rangle \tag{40}$$

for any nonlinear function φ. The amplitudes of $|\psi'\rangle$ are some nonlinear transform of the amplitudes of the hypothetical state $|\psi\rangle$. This hybrid classical/quantum approach is not ideal, as classical nonlinearity takes linear time, removing our quantum speed-up. However, for simulation purposes this is a great option.

6.0 Discussion and Future Work

In this paper we have introduced the reservoir computing paradigm and discussed its merit versus SVM type models for training over large numbers of parameters. We then outlined a quantum variational classification model that utilizes a quantum enhanced feature space to separate data. Finally, we combined these two ideas and presented the framework for a general purpose feed-forward quantum reservoir computing model designed for a fault-tolerant quantum computer. This quantum reservoir feature learner, in theory, could temporally outperform its classical RC analog. However, this model is built on QFT, QPE, and HHL, which each require fault tolerance. Therefore, like many other proposed quantum algorithms, it is unfeasible for a near-term device. Low dimensional simulations are possible and produce promising results [38]. Adaptations that take this from a purely quantum model to a hybrid model could be used in the near term, but this would likely defeat the temporal advantage over existing variational and even classical algorithms. Quantum nonlinearity, state preparation, and readout are all topics of ongoing work in the world of quantum computing.

There are avenues for future work in quantum reservoir computing. For example, creating a reservoir model that exploits the quantum dynamics of naturally available systems similar to Fujii *et al.* [8]. Alternatively, one could target a specific problem well suited for the high memory capacity of a QRC, and pursue an optimization procedure or application that works well for that task, for example, Kutvonen *et al.* [19] and Nakajima *et al.* [27]. Another possibility, not as grandiose, is to take-on one aspect of the QRC model, for instance reservoir nonlinearity, and attempt various isolated, small-scale implementations.

Appendix

A.1 Quantum Random Access Memory

Classical data must be read-in before being processed on a quantum computer. While quantum algorithms can provide dramatic speedups for processing data they rarely provide advantages for reading data. The cost of reading-in data requires linear time at best, and thus dominates the cost of many quantum algorithms [2, 5]. Theoretically, this can be addressed using quantum RAM (qRAM). Quantum access to the training data allows the machine to be trained using quadratically fewer accesses to the training data than classical methods require. A quantum algorithm can train, for example, a deep neural network on a large training data set while reading only a small number of training vectors [36].

Classical RAM uses n bits to randomly address $N = 2^n$ distinct memory cells. A qRAM uses n qubits to address any quantum superposition of N memory cells [12]. qRAM allows access to classically stored information in superposition by querying an index register. Consider an address register a containing a superposition of addresses $\sum_j \psi_j |j\rangle_a$. The qRAM will return a superposition of data in a data register d,

$$\sum_j \psi_j |j\rangle_a \longrightarrow \sum_j \psi_j |j\rangle_a |D_j\rangle_d \tag{41}$$

where D_j is the content of the jth memory cell. However, the qRAM's exponential use of resources translates into a high decoherence. For this reason, there are not yet any physical realizations of QRAM nor clear solutions to the read-in problem. Our group is currently investigating alternate approaches including a reservoir-qRAM (reservoir quantum coupler) [37] which would sacrifice control over the format of the data read-in for a read-in speed-up, which seems an acceptable tradeoff. But this is still a topic of ongoing work. For now, our algorithm simply assumes the existence of a qRAM.

A.2 Quantum Phase Estimation cont.

To re-iterate, the problem of QPE is as follows. Given a unitary operator U with eigenvalues and eigenvectors

$$U\left|\psi\right\rangle = e^{2\pi i\theta}\left|\psi\right\rangle, \tag{42}$$

estimate the phase, θ. The general circuit is given in Figure (16).

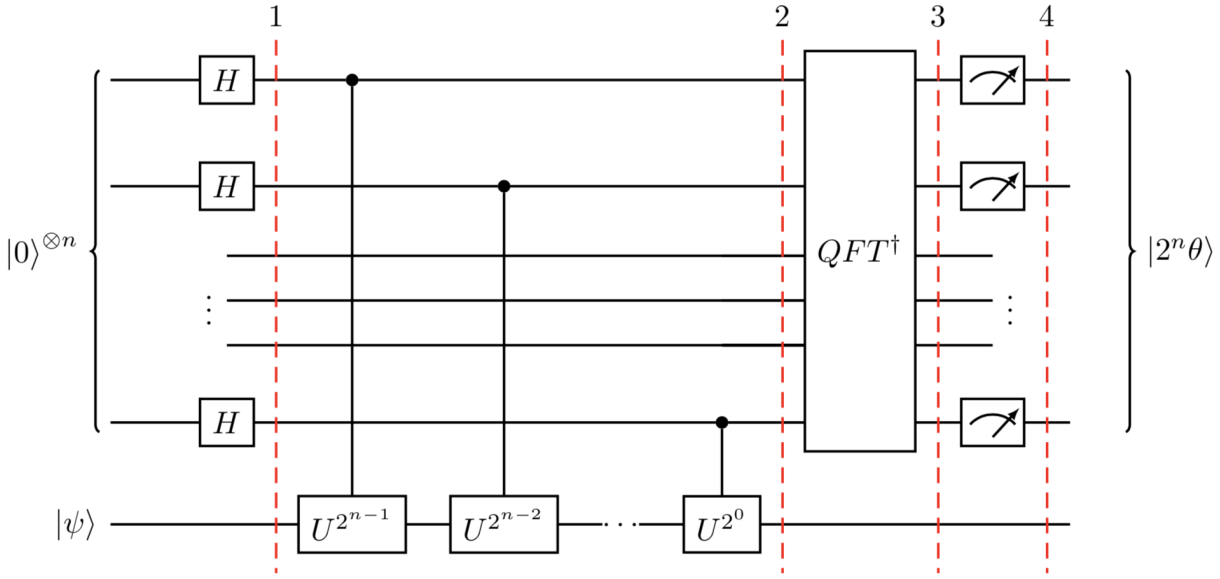


Figure 16: Quantum phase estimation circuit [3]

The QPE procedure can be broken into five steps as follows below. Each step will outline the general case and then specify a three-qubit ($n = 3$) implementation. In this example we will aim to estimate the phase of the T-gate:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}}, \end{bmatrix} \tag{43}$$

which adds a phase of $e^{\frac{i\pi}{4}}$ when applied to the state $|1\rangle$. From (42), we then have $2i\pi\theta = i\pi/4$ so expect to find

$$\theta = \frac{1}{8} \tag{44}$$

as an exact result.

A.2.1 Setup

The QPE procedure uses two registers. The first register contains n "counting" qubits. A greater n increases the accuracy of the estimate for θ, but decreases the probability that the procedure will be successful. The second register begins in the state $|\psi\rangle$, and contains as many qubits as is necessary to store $|\psi\rangle$.

$$|\psi_0\rangle = |0\rangle^{\otimes n} |\psi\rangle \tag{45}$$

A.2.2 Superposition

Next we create a superposition of states by applying a Hadamard operation $H^{\otimes n}$ on the counting register. The one-qubit Hadamard gate is given by

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{46}$$

so the n-qubit state is now given by

$$|\psi_1\rangle = \frac{1}{2^{n/2}} (|0\rangle + |1\rangle)^{\otimes n} |\psi\rangle \tag{47}$$

For the three-qubit example we will take $|\psi\rangle = X|0\rangle = |1\rangle$, an $|0\rangle$'s in the counting register, as required.

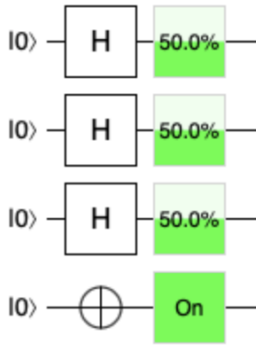


Figure 17: Initialization and superposition. Image generated using [11]

Above, the percentage contained in each box indicates the probability of that qubit being in the state $|1\rangle$. An "On" ("Off") qubit is in state $|1\rangle$ ($|0\rangle$) with probability 1. As intended, the counting qubits are in a superposition of states with equal probability of being measured $|1\rangle$ or $|0\rangle$.

A.2.3 Controlled Unitary Operations

Consider again our unitary operator U. The controlled unitary operation CU applies U on the target register only if its corresponding control bit is $|1\rangle$. If the control bit is $|0\rangle$, CU acts as the identity on its target bit. Using our eigenvalue equation (42):

$$U^{2^j}|\psi\rangle = U^{2^j-1}U|\psi\rangle = U^{2^j-1}e^{2\pi i\theta}|\psi\rangle = ... = e^{2\pi i 2^j\theta}|psi\rangle \tag{48}$$

We therefore need to apply n controlled operations CU^{2^j}:

$$|\psi_2\rangle = \frac{1}{2^{n/2}}\left(|0\rangle + e^{2\pi i\theta 2^{n-1}}|1\rangle\right) \otimes ... \otimes \left(|0\rangle + e^{2\pi i\theta 2^1}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i\theta 2^0}|1\rangle\right) \otimes |\psi\rangle$$

$$|\psi_2\rangle = \frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1} e^{2\pi i\theta k}|k\rangle \otimes |\psi\rangle \tag{49}$$

This sequence is clear when visualized. Building onto the three-qubit circuit from Figure (17) using $U = T$ (43):

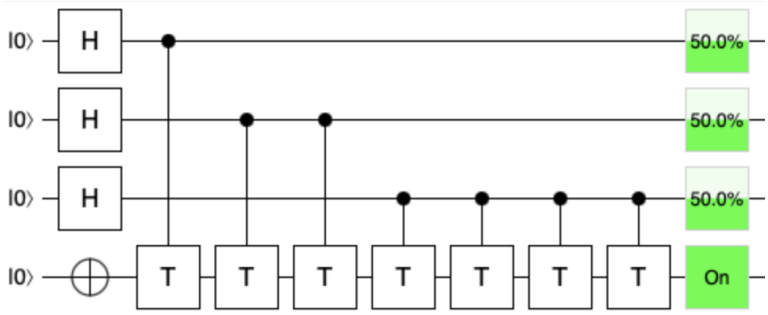


Figure 18: Addition of controlled T-gates. Image generated using [11]

Through these controlled operations we have used phase kick-back[1] to write the phase of U (in the Fourier basis) to the n qubits in the counting register.

A.2.4 Inverse Fourier Transform

Notice that (49) is exactly the result of applying the QFT (13) where $\theta = j/N \rightarrow \theta N = \theta 2^n = j$. Therefore, to recover the state $|\theta 2^n\rangle$ we must apply an inverse QFT on the ancilla[2] register:

$$|\psi_3\rangle = \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i \theta k} |k\rangle \otimes |\psi\rangle \xrightarrow{\mathcal{QFT}_n^{-1}} \frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{k=0}^{2^n-1} e^{-\frac{2\pi i k}{2^n}(x - 2^n\theta)} |x\rangle \otimes |\psi\rangle \tag{50}$$

The circuit that implements the QFT makes use of the one-qubit Hadamard gate (46) and the two-qubit controlled rotation gate,

$$CR_k = \begin{bmatrix} I & 0 \\ 0 & R_k \end{bmatrix} \tag{51}$$

[1] Kickback is where the eigenvalue added by a gate to a qubit is 'kicked back' into a different qubit via a controlled operation.

[2] When translating a classical circuit into a quantum circuit, you often need to introduce extra qubits simply because quantum computers only implement reversible logic. Such extra qubits are ancilla (or ancillary qubits). In Figure (16) the ancillary register consists of one qubit initialized in the state $|\psi\rangle$.

where

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\frac{2\pi i}{2^k}\right) \end{bmatrix} \tag{52}$$

Here, I is the identity matrix, and the controlled action of CR on R works in the same way as the controlled action of CU on U from Section (A.2.3). The first half of the phase estimation circuit essentially performs the forward QFT, and by doing so, reverses the order of qubits. So the first action in the inverse QFT is to use the SWAP gate to return the original qubit ordering. We then continue to follow the procedure of Figure (4) in precisely reverse order. The addition of the inverse QFT onto the three-qubit circuit looks as follows:

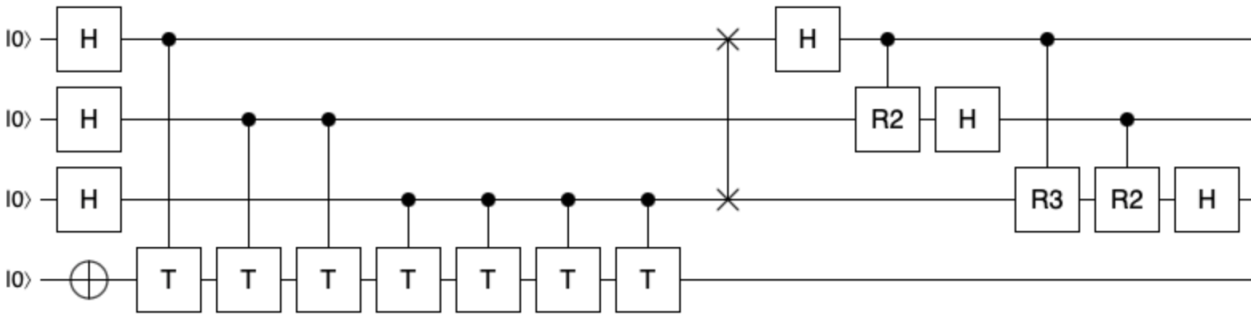


Figure 19: Addition of inverse QFT. Image generated using [11]

A.2.5 Measurement

After all operations are complete, we read out the state of the ancilla register by doing a measurement in the computational basis. 50 peaks near $x = 2^n\theta$, so for the case when $2^n\theta$ is an integer we measure the correct phase with high probability:

$$|\psi_4\rangle = |2^n\theta\rangle \otimes |\psi\rangle \tag{53}$$

When $2^n\theta$ is not an integer an accurate measurement can still be achieve with better than $4/\pi^2 \approx 40\%$ probability [29]. Applying measurement gates to the three-qubit circuit,

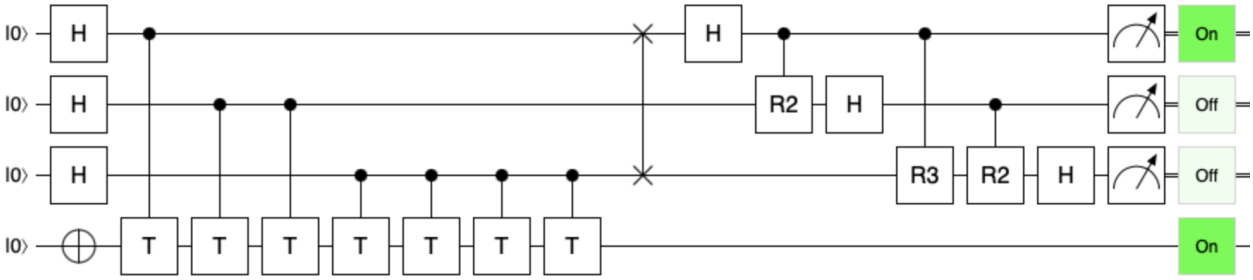


Figure 20: Addition of measurement gates. Image generated using [11]

we get the binary result (001) with probability 1. Converting to decimal form, and dividing by 2^n:

$$\theta = \frac{1}{2^3} = \frac{1}{8} \tag{54}$$

which agrees with the theoretical prediction of (44).

A.2.6 C++ Implementation

We simulated the QPE algorithm in C++ using the Quantum++ general-purpose multi-threaded quantum computing library [10]. The library is written in C++11 and composed solely of header files. It's simulation capabilities are only restricted by the amount of available physical memory. On a typical machine Quantum++ can successfully simulate the evolution of 25 qubits in a pure state or of 12 qubits in a mixed state reasonably fast. Estimating the phase of the T-gate following the procedure above our program produces the following:

```
>> QPE on n = 3 qubits. The sequence of applied gates is:
X3 H0 H1 H2
CU(0, 3)
CU(1, 3) CU(1, 3)
CU(2, 3) CU(2, 3) CU(2, 3) CU(2, 3)
SWAP(0, 2)
H0
R2(0, 1) H1
R3(0, 2) R2(1, 2) H2

>> Measurement result q0: 1
>> Probabilities: [0, 1]
>> Measurement result q1: 0
>> Probabilities: [1, 0]
>> Measurement result q2: 0
>> Probabilities: [1, 0]

>> Input theta = 0.125
>> Estimated theta = 0.125
>> Norm difference: 0
Program ended with exit code: 0
```

Figure 21: qpe.cpp program output

The sequence of applied gates agrees with the model, as does the estimated phase to a norm difference of 0. The source code for this example is available on the Quantum++ GitHub.

A.3 HHL Put Simply

The HHL algorithm [13], named after it creators Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd, is a quantum algorithm for solving linear systems of equations. Given the Hermitian $N \times N$ matrix A and a unit vector $\vec{b}$, HHL finds a vector $\vec{x}$ such that $A\vec{x} = \vec{b}$. Classical algorithms can find $\vec{x}$ in time $\propto \mathcal{O}(N)$. The HHL quantum algorithm can find $\vec{x}$ in time $\propto \mathcal{O}(\text{poly}(\log N))$, thus providing an exponential speed-up. The HHL procedure is as follows:

First, we represent $\vec{b}$ as a quantum state:

$$|b\rangle = \sum_{i=1}^{N} b_i |i\rangle . \tag{55}$$

Next, we use techniques of Hamiltonian simulation to apply to apply e^{iAt} to $|b\rangle$ at different times t. Through quantum phase estimation (QPE), they then decompose $|b\rangle$ into the eigenbasis of A and find the corresponding eigenvalues λ_j. At this point the system is in a state close to

$$\sum_{j=1}^{N} \beta_j |u_j\rangle |\lambda_j\rangle \tag{56}$$

where u_j is the eigenvector basis of A, and

$$|b\rangle = \sum_{i=1}^{N} \beta j |u_j\rangle . \tag{57}$$

They we take a linear map

$$|\lambda\rangle \longrightarrow C\lambda^{-1} |\lambda\rangle \tag{58}$$

where C is a normalizing constant. This operation is not unitary, so has some probability of failing. After it succeeds, we uncompute the $|\lambda_j\rangle$ register and are left with a state proportional to

$$|x\rangle = A^{-1} |b\rangle = \sum_{j} \lambda_j^{-1} \beta_j |u_j\rangle \langle u_j| . \tag{59}$$

A.4 ESN Tutorial

A.4.1 Formalism

To reiterate, the characteristic task of supervised learning (Section (2.1.3)) with RNNs, and with all ANNs for that matter, is to learn a functional relation between a given input $\mathbf{u}(n) \in \mathbb{R}^{N_u}$ and a desired output $\bar{\mathbf{y}}(n) \in \mathbb{R}^{N_y}$, where n = 1,...,T, and T is the number of data points in the training dataset $(u(n), \bar{\mathbf{y}}(n))$. For a non-temporal task (i.e. data points are independent of each other) the goal is to learn a function $\mathbf{y}(n) = y(\mathbf{u}(n))$ such that the

loss function $E(\mathbf{y}, \bar{\mathbf{y}})$ is minimized. In the context of RNNs, a common loss function is the normalized root-mean-square error (NRMSE):

$$E(\mathbf{y}, \bar{\mathbf{y}}) = \sqrt{\frac{\langle \left\| \mathbf{y}(n) - \bar{\mathbf{y}}(n) \right\|^2 \rangle}{\langle \left\| \mathbf{y}(n) - \langle \bar{\mathbf{y}}(n) \rangle \right\|^2 \rangle}} \tag{60}$$

A temporal task is where $\mathbf{u}$ and $\bar{\mathbf{y}}$ are signals in a discrete time domain (n = 1,...,T), and the goal is to learn a function $\mathbf{y}(n) = y(..., \mathbf{u}(n-1), \mathbf{u}(n))$ such that $E(\mathbf{y}, \bar{\mathbf{y}})$ is minimized. Thus, the difference between the temporal and non-temporal task is that the function $\mathbf{y}(n)$ we are trying to learn has memory in the first case and is memory-less in the second.

Many tasks cannot be accurately solved by a simple linear relation between $\mathbf{u}$ and $\bar{\mathbf{y}}$. In this case, we resort to a nonlinear model, which can be achieved by taking a nonlinear expansion of the input $\mathbf{u}$. In RC, the function of the reservoir is to act both as this nonlinear expansion and as a memory input. To perform a nonlinear high-dimensional expansion $\mathbf{x}(n) \in \mathbb{R}^{N_x}$ of the input signal $\mathbf{u}(n) \in \mathbb{R}^{N_u}$, we require $N_x >> N_u$. Input data which is not linearly separable in the original space $\mathbb{R}^{N_u}$ often becomes separable in the expanded space $\mathbb{R}^{N_x}$. We are therefore searching for solutions of the form

$$\mathbf{y}(n) = \mathbf{W}_{out}\mathbf{x}(n) = \mathbf{W}_{out}x(\mathbf{u}(n)) \tag{61}$$

where $\mathbf{W}_{out} \in \mathbb{R}^{N_y \times N_x}$ are the trained output weights and $\mathbf{y}(n)$ is the the learned function. Expansion $\mathbf{x}(n)$ is often referred to as the "state vector" of the system at time step n. The network is usually initialized in state

$$\mathbf{x}(0) = \mathbf{0} \tag{62}$$

The reservoir also serves as memory, providing temporal context. This is a crucial reason for using RNNs in the first place. In a temporal task the function to be learned depends also on the history of the input. Thus, the expansion function has memory: $\mathbf{x}(n) = x(..., \mathbf{u}(n-1), \mathbf{u}(n))$. This function has an unbounded number of parameters, so we can express it recursively:

$$\mathbf{x}(n) = x(\mathbf{x}(n-1), \mathbf{u}(n)) \tag{63}$$

With the recursive (temporal) definition of the state vector, the output $\mathbf{y}(n)$ is (typically) still calculated in the same way as for non-temporal methods (61). For non-temporal tasks, this recursive definition can act as a type of a spatial embedding of temporal information. This enables learning of high-dimensional dynamical tendencies (or "attractors" [26]) of the system from low-dimensional observations. This is shown possible by Takens's theorem [34], as mentioned in Section (2.1.2).

Combining the state vector non-linear expansion and memory components leads to the following general RNN state update equation,

$$\mathbf{x}(n) = f(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{64}$$

where f is the neuron activation function (Section (2.1.1)), usually symmetric tanh (4), applied element-wise, $\mathbf{W}_{in} \in \mathbb{R}^{N_x \times N_u}$ is the input weight matrix, and $\mathbf{W} \in \mathbb{R}^{N_x \times N_x}$ is the internal (hidden) weight matrix of network connections.

The ESN is a recurrent neural network with a sparsely connected hidden layer (the reservoir), driven by a (one- or multi-dimensional) time signal, and applied to supervised temporal ML tasks. The connectivity and weights of hidden neurons are fixed and randomly assigned. The weights connecting the hidden neurons to the output neurons are the only trainable parameters in the network. Although ESNs are dynamic, with non-linear behavior, their "single-layer training" attribute makes their loss function (E(p) in (10)) quadratic in "parameter-space," so it can be differentiated to a linear system. Quadratic loss functions are desirable because they are easily manipulated (property of variances), symmetric, and allow easy application of linear regression.

ESNs are an attractive RC implementation method because they are conceptually simple and computationally inexpensive. However, creating an effective ESN is not, in and of itself, straightforward. The following sections will overview ESN implementation, training, and application. We will approach ESN implementation in the practical context of a simple Python3 program using the PyTorch open-source ML library. To do so, we will work through each of the "To Dos" in example code outline `class Reservoir()` and `class ESN(.)`.

In the code outlines (Figure 22, Figure 23), a question mark, "`?`", indicates a numerical value selected by the user. "TODO", abbreviated TD# from here on, indicates a section of

code yet to be implemented, where # in $[1, 6]$ is the order of implementation followed in this paper.

```
import torch
import torch.nn as nn
from reservoir import Reservoir

class ESN(nn.Module):
  def __init__(self, input_size, output_size, hidden_size=?):
    super().__init__()
    self.reservoir = Reservoir(input_size, hidden_size)
    self.linear = nn.Linear(hidden_size,
      output_size, bias=False)
    self.softmax = nn.Softmax(dim=-1)

  def forward(self, u):
    # TODO 3
    x = self.reservoir(u)
    # TODO 6
    return y
```

Figure 22: esn.py

```
import torch
import random

class Reservoir:
  def __init__(self, input_size, res_size, sparsity=?,
  spect_rad=?, input_scale=?, leak_rate=?):
    self.input_size = input_size
    self.res_size = res_size
    self.sparsity = sparsity
    self.spect_rad = spect_rad
    self.input_scale = input_scale
    self.leak_rate  = leak_rate
    self.w_in = self.gen_w_in(self.res_size,
      self.input_size, self.input_scale)
    self.w = self.gen_w(self.res_size, self.spect_rad)
    self.state_x = self.init_state(self.res_size)

  def init_state(self, res_size):
    # TODO 4
    return init_state

  def gen_w_in(self, res_size, input_size, input_scale):
    # TODO 2
    return w_in

  def gen_w(self, res_size, spect_rad):
      # TODO 1
      return w

  def _forward(self, u):
    assert(u.size() == self.input_size)
    with torch.no_grad():
      # TODO 5
      self.state_x = updated_x
      return updated_x

  def __call__(self, u):
    return self._forward(u)
```

Figure 23: reservoir.py

A.4.2 Reservoir Generation

The reservoir is defined by tuple $(\mathbf{W}_{in}, \mathbf{W}, \alpha)$. $\mathbf{W}_{in}$ and $\mathbf{W}$ are generated randomly according to a number of hyperparameters (in analogy to other ANN approaches, "hyperparameters" refer to parameters governing the distribution of connection weights, as opposed to the connection weights themselves). α is the leaking rate, and will be defined formally in Section (A.4). Reservoir hyperparameters must be subtly chosen, as each significantly impacts the behavior of the network, and therefore its overall performance. Characteristic reservoir hyperparameters include the reservoir size, sparsity, and distribution of nonzero elements, the spectral radius of $\mathbf{W}$, the scaling of $\mathbf{W}_{in}$, and, of course, the leaking rate α. These values are initialized in the `Reservoir` class constructor. In this implementation, the input size and desired reservoir size are set inside of the `ESN` class upon instantiation of a `Reservoir` object.

`gen_w(.)`

To begin, we wish to define a function `gen_w()` (TD1), which generates the random, internal weight matrix $\mathbf{W}$ of size $N_x \times N_x$ (64). The bigger the space of reservoir signals $\mathbf{x}(n)$, the easier it is to find a linear combination of the signals to approximate $\bar{\mathbf{y}}(n)$. Therefore, the bigger the reservoir, the better the obtainable performance. However, reservoir size *is* bounded by its memory capacity (number of values it must remember from the input to accomplish the task, lower bound) and by the over-fitting threshold (upper bound):

$$N_y \leq N_x \leq T - N_u \tag{65}$$

where T is the number of training data points.

Weight matrix $\mathbf{W}$ is typically sparse. Compared to dense representations, this enables faster state vector updates while also giving slightly better performance [28]. Nonzero elements can take any distribution, though common choices include uniform, discrete bi-valued, and normal [21]. There does not exist built-in Python functionality to construct, simultaneously, a matrix according to a given density (or sparsity) *and* a given nonzero element distribution. Therefore, these tasks must be performed in sequence. For example, using

`torch.randn`, we can easily generate a random matrix of size N_x with values (weights) normally distributed around zero. The initial width of the distribution does not matter, as it is eventually reset according to the reservoir's spectral radius. We can then iterate over this generated matrix, drawing a random number $p \in [0.0, 1.0)$ at each entry $\mathbf{W}_{i,j}$ using Numpy library function `random.random()`. For desired sparsity $s \in (0.5, 1.0)$, if $p_{i,j} \leq s$, we set $\mathbf{W}_{i,j} = 0$. Else, $\mathbf{W}_{i,j}$ is unchanged. This is merely one of many possible approaches.

Spectral radius is one of the ESN's most central hyperparameters; it specifies the largest absolute eigenvalue of weight matrix $\mathbf{W}$. After a random sparse $\mathbf{W}$ (now named $\mathbf{W}_0$) is generated, we can compute its spectral radius

$$\rho_0(\mathbf{W}_0) = \lambda_{max}(\mathbf{W}_0) \tag{66}$$

using PyTorch's `torch.eig` and `torch.max`, or equivalent functions. Next, we rescale $\mathbf{W}_0$ according to user-specified spectral radius, ρ (28):

$$\mathbf{W} = \frac{\rho}{|\rho_0|}\mathbf{W}_0 \tag{67}$$

Finally, `return` the rescaled weight matrix $\mathbf{W}$, now with largest absolute eigenvalue ρ, and `gen_w()` is complete.

For the ESN model to work, the reservoir must satisfy the echo-state property (ESP): the effect of previous state $\mathbf{x}(n)$ and previous input $\mathbf{u}(n)$ on a future state $\mathbf{x}(n+k)$ should vanish gradually as time passes ($k \to \infty$) [22]. In other words, the reservoir must asymptotically "wash out" any information from initial conditions. For reservoirs with tanh activation and input $\mathbf{u}(n) = \mathbf{0}$, the ESP is *violated* if $\mathbf{W}$ is scaled such that

$$\rho(\mathbf{W}) > 1 \tag{68}$$

Contrary to many simplistic descriptions, $\rho < 1$ does not guarantee the ESP. The ESP can be obtained for $\mathbf{u}(n) \neq \mathbf{0}$ even if $\rho > 1$, and can be violated even if $\rho < 1$ [22] (although the latter is unlikely). Through a process of trial and error, ρ should be selected to maximize performance. Generally, ρ should be close to 1 for tasks that require long term memory, and smaller for tasks where too much memory may be harmful.

We have now constructed internal weight matrix $\mathbf{W}$ according to user-specified hyperparameters reservoir size, sparsity, distribution of nonzero elements, and spectral radius.

`gen_w_in(.)`

Following a similar procedure, we will now define a function `gen_w_in()` (TD2), which generates the random, input weight matrix $\mathbf{W}_{in}$ of size $N_x \times N_u$ (64). N_x is the reservoir size, and was specified while generating $\mathbf{W}$. N_u is the dimensionality of the input data, so is also predetermined. $\mathbf{W}_{in}$ is usually generated according to the same distribution as $\mathbf{W}$. Therefore, a normal distribution of weights around 0 may, once again, be carried out using the `torch.randn` function. This time, however, the width of the distribution *does* does matter: for normal distributed input weights we can take the standard deviation as a scaling measure [21]. This scaling measure, known as the input scaling, a, is a vital ESN optimization parameter. It dictates how "nonlinear" reservoir responses are.

For small input scaling values, the units of $\mathbf{W}_{in}$ will collapse towards 0, where their tanh activation is essentially linear (Figure 23). So for linear tasks, a should be small. For large input scaling values, the units of $\mathbf{W}_{in}$ will quickly saturate towards 1 and -1, exhibiting nonlinear, binary "switching" behavior. So for nonlinear tasks, reservoir response *might* be optimized for larger a. The "amount" of nonlinearity a task requires is not always easy to assess. This hyperparameter is another best tuned through trail and error.

Referring back to the general state update equation (64), it is clear, for sigmoid activation, that the scaling a of $\mathbf{W}_{in}$, and the scaling $\rho(\mathbf{W})$ of $\mathbf{W}$, together dictate the amount of nonlinearity and memory of previous states present in the current state representation.

To implement `gen_w_in()`, sequentially construct $\mathbf{W}_{in}$ by first creating an $N_x \times N_u$ dimensional matrix with the same element distribution as $\mathbf{W}$. But this time, form the distribution according to the input scaling, a: For uniform distributions, sample values from the interval $[-a, a]$. For normal distributions, set $\sigma = a$. Then, iterate through the resulting matrix, drawing random number $p \in [0.0, 1.0)$ at each entry, and setting $\mathbf{W}_{in_{i,j}} = 0$ where $p_{i,j} \leq s$. $\mathbf{W}_{in}$ is typically dense, so $s \in [0.0, 0.5)$. `return` $\mathbf{W}_{in}$ and `gen_w_in()` is complete.

We have now constructed input weight matrix $\mathbf{W}_{in}$ according to the input data dimension, reservoir size, nonzero element distribution of $\mathbf{W}$, and user-specified input scaling

parameter. In the next section, we will further specify our reservoir model by revisiting neuron activation (Section (2.1.1)), now in the context of RNNs.

A.4.3 Activation States

This section will overview the leaky integrator neuron model, and then describe the computation and collection of reservoir activation states $\mathbf{x}(n)$ for a training input $\mathbf{u}(n)$.

`forward(.)` Pt. 1

In ANN training (Section (2.1.3)), the "forward pass" refers to the computation performed during each training iteration. In a single forward pass, the network is provided one training data sample whose information is propagated through the hidden units of the network to eventually produce an output $\mathbf{y}$, at which point a linear readout is performed. In the first component of the `class ESN` "forward" method (TD3) we manipulate the input, $\mathbf{u}$, to match the user-specified reservoir input dimension. You can read-in data any way you like, as along as all of the input information is inside the reservoir simultaneously. Take, for example, an input of dimension $1 \times N \times N$. A natural way to perform this read-in would be to generate a reservoir $\mathbf{W}$ of $N \times N$ neurons, and to flatten and transpose each input into a $N^2 \times 1$ column vector. This could be achieved through the `torch.flatten()` and `torch.t()` methods. However, you may alternatively rely on the fact that the reservoir has a memory, and read-in each single column of the input as a $N \times 1$ vector at each time point, ultimately taking N time points to get the entire input into the reservoir.

`init_state`(.)

Before we can use activation to perform a state update, we need an initial state (TD4). The reservoir state vector must be of dimension $N_x \times 1$ (64). If we have chosen a spectral radius such that our reservoir satisfies the ESP ((68) *not* true), then the final state of the network does not depend on its initial conditions. Therefore, the initial state can be any vector of the above dimension. A common choice is the zero vector (62), which can be created using the `torch.zeros` function.

`forward(.)` Pt. 2

As previously stated, standard ESNs use sigmoid neurons, i.e., reservoir states are computed using (64), where the nonlinear function f is a sigmoid, usually the tanh function:

$$\mathbf{x}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{69}$$

Leaky integrator neuron models represent another frequent option for ESNs. These type of neurons perform a "leaky" integration of their activation from previous time steps. Generally, a leaky integrator is analogous to a first-order filter with feedback.

For arbitrary input $\mathbf{u}(n)$ and output $\mathbf{y}(n)$, a leaky integrator can be described by non-homogeneous first-order linear ODE

$$\dot{\mathbf{y}}(n) + \alpha\mathbf{y}(n) = \mathbf{u}(n) \tag{70}$$

where α is the leaking rate. Using (68), let us redefine

$$\mathbf{u}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{71a}$$

$$\alpha\mathbf{y}(n) = \mathbf{x}(n) \tag{71b}$$

Plugging our substitutions (70) into (69) and rearranging,

$$\dot{\mathbf{x}}(n) + \mathbf{x}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{72}$$

$$\dot{\mathbf{x}} = -\mathbf{x} + \tanh(\mathbf{W}_{in}\mathbf{u} + \mathbf{W}\mathbf{x}) \tag{73}$$

where (72) describes the continuous time reservoir update dynamics. Performing an Euler's discretization in time on (72),

$$\dot{\mathbf{x}} \approx \frac{\Delta\mathbf{x}}{\Delta t} = \frac{\mathbf{x}(n+1) - \mathbf{x}(n)}{\Delta t} \tag{74}$$

For brevity, we will redefine the right hand side of (71):

$$\tilde{\mathbf{x}}(n+1) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{75}$$

Setting (20) equal to (19), and using simple algebra:

$$\frac{\mathbf{x}(n+1)-\mathbf{x}(n)}{\Delta t} = -\mathbf{x}(n) + \tilde{\mathbf{x}}(n+1) \tag{76}$$

$$\mathbf{x}(n+1)-\mathbf{x}(n) = \Delta t(-\mathbf{x}(n) + \tilde{\mathbf{x}}(n+1)) \tag{77}$$

$$\mathbf{x}(n+1) = (1-\Delta t)\mathbf{x}(n) + \Delta t\tilde{\mathbf{x}}(n+1)) \tag{78}$$

The leaking rate $\alpha \in (0, 1]$ of the reservoir nodes denotes the speed of the reservoir update dynamics *discretized in time*. Therefore, we can substitute α for the sampling interval Δt. And finally, by performing time-translation $n \to n-1$, we arrive at the leaky-integrated discrete-time continuous-value unit ESN state update equations:

$$\tilde{\mathbf{x}}(n) = \tanh(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1)) \tag{79a}$$

$$\mathbf{x}(n) = (1-\alpha)\mathbf{x}(n-1) + \alpha\tilde{\mathbf{x}}(n) \tag{79b}$$

where $\mathbf{x}(n)$ is the current state vector and $\tilde{\mathbf{x}}(n)$ is its update, each at time step n. To apply a bias b (as in $y = mx + b$), simply use

$$\mathbf{u}(n) \to [b; \mathbf{u}(n)] \tag{80}$$

where $[\cdot;\cdot]$ stands for a vertical vector (or matrix) concatenation [21]. Notice, to use the model without leaky integration, simply set $\alpha = 1$ and thus $\tilde{\mathbf{x}}(n) \equiv \mathbf{x}(n)$.

We are now equipped to implement the second component of the `class ESN(.)` "forward" method, i.e. the "forward" method of `class Reservoir()`, (TD5): `_forward()` updates and returns the state representation $\mathbf{x}(n)$ for given input $\mathbf{u}(n)$. The single leading underscore is a convention used to indicate that this method is meant only for internal use. Here, it also used to differentiate between `forward()` of `class ESN(.)`, which is where the readout, and remainder of the "forward pass" will be performed (Section (A.4.4)). Abstracting the code in this way, organizing methods acting in and on the reservoir into a

separate `Reservoir` class, will prove exceptionally useful in debugging, parameter selection, and extending to multi-reservoir networks (DeepESNs).

Notice the global variable `state_x` inside of the `Reservoir` constructor method. This variable stores the state update of the previous training iteration (or pass), $\mathbf{x}(n-1)$ so it can be applied to perform a new update during the current pass. To implement TD5, use the built-in `torch.tanh` and `torch.mm` (matrix multiply) methods to evaluate (79). Store the result in a temporary variable, and evaluate (80) to produce `updated_x`. Finally, we update global variable `state_x` for the next pass, and return the updated state.

We have now created methods to generate an initial state $\mathbf{x}(0) = \mathbf{0}$, and to perform a leaky-integrated state update for given input $\mathbf{u}(n)$ using our previously generated $\mathbf{W}$ and $\mathbf{W}_{in}$, and globally stored $\mathbf{x}(n-1)$.

A.4.4 Compute a Probability Distribution

Restating (61) for convenience, our main objective is to find an output weight matrix $\mathbf{W}_{out}$ in $\mathbb{R}^{N_x}$ such that

$$\mathbf{W}_{out}\mathbf{x}(n) = \mathbf{y}(n) \approx \bar{\mathbf{y}}(n) \tag{81}$$

For tasks where feedback or temporal sequential processing is important, it often helps significantly to include the original input $\mathbf{u}(n)$ and a bias, b. In this case, the linear readout becomes

$$\mathbf{W}_{out}[b; \mathbf{u}(n); \mathbf{x}(n)] = \mathbf{y}(n) \tag{82}$$

where $\mathbf{y}(n) \in \mathbb{R}^{N_y}$, $\mathbf{W}_{out}(n) \in \mathbb{R}^{N_y \times (1+N_u+N_x)}$, and $[\cdot;\cdot;\cdot]$ again stands for a vertical vector (or matrix) concatenation [21].

Since the readouts from an ESN are typically linear and feedforward (Section (**??**)) [21], learning the output weights (82) can be viewed as solving a system of linear equations

$$\mathbf{W}_{out}\mathbf{X} = \bar{\mathbf{Y}} \tag{83}$$

where $\mathbf{X} \in \mathbb{R}^{N \times T}$ are all $\mathbf{x}(n)$ produced by presenting the reservoir with $\mathbf{u}(n)$, and $\bar{\mathbf{Y}} \in \mathbb{R}^{N_y \times T}$ are all $\bar{\mathbf{y}}(n)$, both collected intro respective matrices over the training period $n = 1, ..., T$. Note: for linear readout using (82) and (83) respectively,

$$\mathbf{X} \in \mathbb{R}^{N \times T} \rightarrow \mathbf{X} \in \mathbb{R}^{N_x \times T} \tag{84a}$$

$$\mathbf{X} \in \mathbb{R}^{N \times T} \rightarrow [\mathbf{B};\ \mathbf{U};\ \mathbf{X}] \in \mathbb{R}^{(1+N_u+N_x) \times T} \tag{84b}$$

The goal is to minimize the (quadratic) error-rate $\mathbf{E}(\bar{\mathbf{Y}}, \mathbf{W}_{out}\mathbf{X})$ as in (60). This problem seems familiar from our previous discussion of supervised learning (Section (**??**)), so let's first approach it as we know how, using gradient descent (GD).

`forward(.)` Pt. 3

The third and final component of the `class ESN(.)` "forward" method (TD6) aims to use the state update(s) (79) computed in `_forward(.)` (TD5) to generate output predictions, $\mathbf{y}(n)$. In accordance with GD, we can then compare these predictions against true values $\bar{\mathbf{y}}(n)$ to iteratively update the weights of $\mathbf{W}_{out}$ (10) in direction opposite to the gradient of the error function (60). Due to the auto-feedback nature of ESNs (i.e. their "echo" property), the reservoir state $\mathbf{x}(n)$ holds traces of past activations $\mathbf{x}(n-1)$, $\mathbf{x}(n-2)$, ..., etc. [3] It is therefore often sufficient to take $\mathbf{x}(n)$ alone as a representative encoding of the input history, and compute $\mathbf{y}(n)$ according to (81). We can do so using the PyTorch `nn.Linear` class, which applies a linear transformation to incoming data

$$\mathbf{y} = \mathbf{x}\mathbf{A}^\intercal + \mathbf{b} \tag{85}$$

Note: for some tasks (e.g. classification) performance is improved using time-averaged activations, $\sum \mathbf{x}$ [21]. But for now, we will proceed with just $\mathbf{x}(n)$ for clarity. In the `ESN` class definition (Figure 22) we instantiated `nn.Linear` such that (85) takes $\mathbf{x} \in \mathbb{R}^{N_x} \rightarrow \mathbf{y} \in \mathbb{R}^{N_y}$, as required. Setting `bias=False` ensures $\mathbf{b} = \mathbf{0}$ so that the only learnable parameters are the weights of $\mathbf{W}_{out}$ (i.e. elements of $\mathbf{A}^\intercal$). In TD6, transpose just-computed state update $\mathbf{x}(n)$ as necessary and pass it through the linear layer. Finally, apply the `nn.Softmax` function,

[3] The extent of this dynamical short-term memory can be calculated explicitly, and is called the ESN's memory capacity, MC. It has been shown [15] that $MC \leq N$ for networks with linear output units and independent and identically distributed (iid) input, where N is the reservoir size.

normalizing the linear layer resultant vector into a probability distribution, i.e. rescaling its elements such that they fall in the range $[0, 1]$ and sum to 1:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{86}$$

Many prefer Softmax over other normalization functions (e.g. argmax, standard normalization) because it is differentiable (useful for backprop, but not applicable in our case) and is exponential, emphasizing the extremes of the distribution. After returning the result of Softmax normalization `forward(.)` is fully implemented.

Our PyTorch ESN model (Figure 23, 9) is now complete. We started by implementing methods to generate a reservoir $(\mathbf{W}_{in}, \mathbf{W}, \alpha)$ according to user-specified hyperparameters reservoir size, sparsity, distribution of nonzero elements, spectral radius, input scaling, and leaking rate. We then initialized the reservoir state vector, and finally, implemented a complete forward pass: we specified the read-in of input data, used that data to perform a tanh leaky-integrated state update, linearly transformed the resultant state according to weight matrix $\mathbf{W}_{out}$, and ultimately, produced a prediction in the form of a normalized probability distribution.

Remember, this is only one of *many* possible implementations. Each component of the network can (and should be) adapted to fit the demands of each unique learning task. However, the class structure, method specification, and general abstraction used in this paper should be intuitive and naturally extensible for anyone just starting out.

A.4.5 Iterative Optimization

In Section (A.4.4) we formulated a goal to solve the linear system of equations given by (83) or, equivalently, minimize the error-rate $\mathbf{E}(\bar{\mathbf{Y}}, \mathbf{W}_{out}\mathbf{X})$. First, we took the gradient descent approach. Now, for a given input $\mathbf{u}(n)$, our ESN generates a prediction $\mathbf{y}(n)$ in the form of an $N_y \times T$ dimensional probability distribution.

Next, we need to define an error (or loss) function. Two common, predefined choices in PyTorch are `nn.MSELoss()` and `nn.CrossEntropyLoss()`. Both loss functions have explicit probabilistic interpretations: MSE corresponds to estimating the mean of a distribution;

cross-entropy corresponds to minimizing the negative log likelihood of a distribution (i.e. maximizing the log likelihood). MSE is thus better suited for regression (narrower, smoother decision boundary where the goal is to be close), while cross-entropy is preferred for classification (larger, more abrupt decision boundary where the penalty for being wrong increases exponentially as you get closer to predicting the wrong output.)

For a PyTorch optimizer, a good choice is Adam (`optim.Adam(.)`). First published in 2014, Adam is an algorithm for first-order gradient-based optimization of stochastic objective functions (randomness present), based on adaptive estimates of lower-order moments (measures of the "shape" of a function) [18]. Adam can be viewed as a combination of RMSprop and Stochastic Gradient Descent (SGD) with momentum (Figure **??**). Like RMSprop, it uses the squared gradients to scale the learning rate, and like SGD with momentum, it uses the moving average of the gradient instead of gradient itself.

Before training, we can quickly verify that the network is generating the correct number of learnable parameters. Assuming all the hyperparameters of `class Reservoir()` are set, specify a hidden size $\texttt{R} \in \mathbb{Z}$ for `class ESN(.)`. Now we create an instance

```
esn = ESN(N, M)
```
(87)

where $\texttt{N}, \texttt{M} \in \mathbb{Z}$ are the input and output size, respectively. Finally, executing

```
params = list(esn.parameters())
print(len(params))
print(params[0].size())
```

should result in output

```
1
torch.Size([M, R])
```

As some back-of-the-envelope linear algebra, or a glance at (61) will prove, $\texttt{M} \times \texttt{R}$ is the correct dimension for trained weight matrix $\mathbf{W}_{out}$.

With our ESN model complete, error and optimizer functions selected and defined, and learned parameters correctly reflecting the dimension of $\mathbf{W}_{out}$, we are ready to perform

gradient descent. This process is relatively straight forward and well documented on the PyTorch website, so will be omitted for the purposes of this paper.

A.4.6 Closed-Form Solutions

Using an alternative approach, we can evaluate (83) for $\mathbf{W}_{out}$ algebraically to find an exact, or closed-form solution. Typically $T \gg N$, so (83) presents an overdetermined system of linear equations:

$$\begin{bmatrix} & & \\ & & \\ & & \\ \vdots & & \\ & & \\ & & \\ & & \end{bmatrix} \begin{bmatrix} & & \\ \mathbf{x}_1(n) & & \\ \vdots & & \\ \mathbf{x}_N(n) & & \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1(n) & & \\ & & \\ \vdots & & \\ & & \\ \mathbf{y}_{N_y}(n) & & \end{bmatrix} \tag{88}$$

To be clear, the dimensionality of this equation follows $(N_y \times N) \cdot (N \times T) = (N_y \times T)$ where N is given by choice of linear readout using state vectors with or without vertical concatenation of input and bias (84) (or just one or the other; not given but easily deduced). The method for finding least square solutions of overdetermined systems of linear equations is also known as linear regression. Solutions following this method are outside the scope of this paper, but could be a subject of future work.

A.4.7 Application

The first application of our ESN was towards traditional MNIST classification. Each MNIST image is delivered as a 3-dimensional tensor (2 spacial dimensions, 1 RGB dimension). By transposing each image into a $28^2 \times 1$ column vector, we could deliver all of the information contained in an image to the reservoir at once (TD3). Figure 24 is a depiction of the performance of our network over 15 training iterations, or epochs.

Table 1: ESN & Training Parameters

Parameter	Value
Reservoir size (N_x)	1000
Spectral radius (ρ)	0.99
Input scaling (a)	0.6
Reservoir density (d_w)	0.1
Input density ($d_{w_{in}}$)	1.0
Leak rate (α)	1.0
Bias (b)	1
Training length (T)	8000

The data shows us that for careful choice of parameters, we can apply an echo-state system (typically suited for temporal tasks) to a classification problem with very respectable results.

The reservoir did not respond well to leaky integration, because the MNIST images bear no temporal connection, so we set α to 1. The most impactful parameter on performance was the reservoir size. Over all of the training, there seemed no upper to limit to where a larger reservoir would not improve performance, besides what the computer could computationally handle. Though, the marginal benefit of continuing to increase reservoir size slowed to a near plateau for sizes roughly equivalent to that of the training data set. There may have been an over-fitting threshold (65), though the reservoir was still improving at larger sizes, just quadratically more slowly. A complete list of ESN and training parameters used during this readout are given in Table 1.

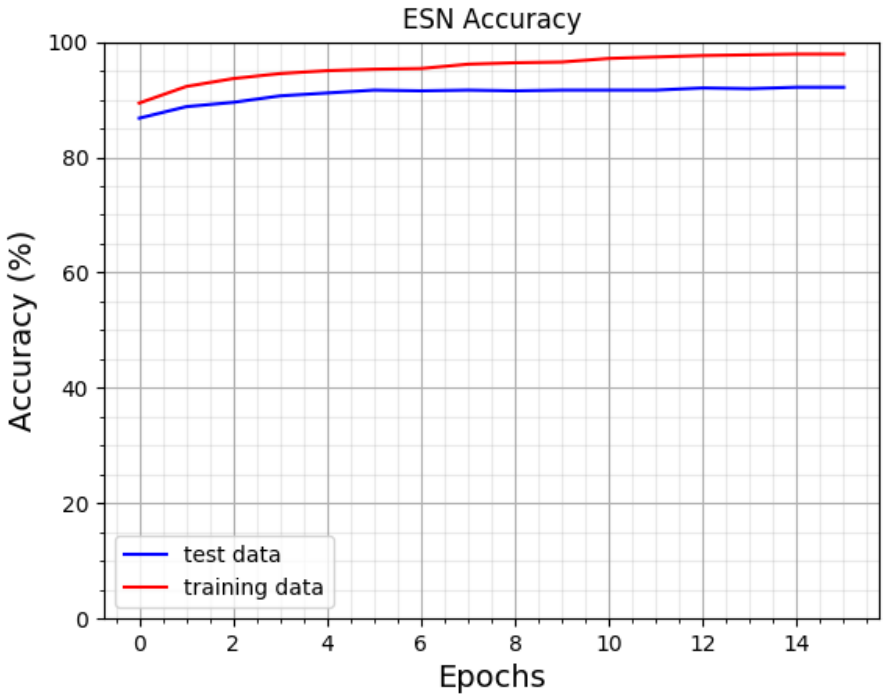


Figure 24: Accuracy of ESN of dimension $N_x = 1000$ on MNIST

Bibliography

[1] *Backpropagation through time: what it does and how to do it*, volume 78 of *Proceedings of the IEEE*, 1990.

[2] S. Aaronson. Read the fine print. *Nature Phys*, 11:291–293, 2015.

[3] Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Nicholas Bronn, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, David McKay, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicni, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, John Stenger, Kristan Temme, Madeleine Tod, Stephen Wood, and James Wootton. Learn quantum computation using qiskit, 2020.

[4] D.W. Berry, G. Ahokas, R. Cleve, et al. Efficient quantum algorithms for simulating sparse hamiltonians. *Commun. Math. Phys.*, 270:359–371, 2007.

[5] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, Sep 2017.

[6] Richard Cleve and Chunhao Wang. Efficient quantum algorithms for simulating lindblad evolution, 2016.

[7] E. Farhi, J. Goldstone, S. Gutmann, and H. Neven. Quantum algorithms for fixed qubit architectures, 2017.

[8] Keisuke Fujii and Kohei Nakajima. Harnessing disordered-ensemble quantum dynamics for machine learning. *Physical Review Applied*, 8(2), Aug 2017.

[9] James Garson. *Reference Reviews*, 29, 8:14–16, 2015.

[10] Vlad Gheorghiu. Quantum++: A modern c++ quantum computing library. *PLOS ONE*, 13(12):e0208073, Dec 2018.

[11] Craig Gidney. Quirk: Quantum circuit simulator, 2020.

[12] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Phys. Rev. Lett.*, 100:160501, Apr 2008.

[13] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009.

[14] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, Mar 2019.

[15] H. Jaeger. Short term memory in echo state networks. Technical report, German National Research Center for Information Technology, 2002.

[16] J.R. Johansson, P.D. Nation, and Franco Nori. Qutip 2: A python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 184(4):1234 – 1240, 2013.

[17] Kiyoshi Kawaguchi. The mcculloch-pitts model of neuron, 2000.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[19] A. Kutvonen, K. Fujii, and T. Sagawa. Optimizing a quantum reservoir computer for time series prediction. *Scientific Reports*, 10(2), September 2020.

[20] S. Lloyd, M. Mohseni, and P. Rebentrost. Quantum principal component analysis. *Nature Phys*, 10:631–633, 2014.

[21] M. Lukoševičius. A Practical Guide to Applying Echo State Networks. *Lecture Notes in Computer Science Neural Networks: Tricks of the Trade*, pages 659–686, 2012.

[22] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3:127–149, 2009.

[23] Mannucci, Nam. Quantum reservoir computing theory. 2019.

[24] Mannucci, Nam. The cost of nonlinearity. 2020.

[25] W.S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5.4:115–133, 1943.

[26] Miller, P. Dynamical systems, attractors, and neural circuits. *F1000Research*, 5:992, 2016.

[27] Kohei Nakajima, Keisuke Fujii, Makoto Negoro, Kosuke Mitarai, and Masahiro Kitagawa. Boosting computational power through spatial multiplexing in quantum reservoir computing. *Physical Review Applied*, 11(3), Mar 2019.

[28] Erich Narang, Sharan an Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks, 2017.

[29] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

[30] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1), Jul 2014.

[31] M. Schuld and F. Petruccione. *Supervised Learning with Quantum Computers*. Springer, 2019.

[32] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. Prediction by linear regression on a quantum computer. *Phys. Rev. A*, 94:022342, Aug 2016.

[33] Paul Smolensky. Grammar-based connectionist approaches to language. *Cognitive Science*, 23:589–613, 1999.

[34] Floris Takens. Comparative analysis of recurrent and finite impulse response neural networks in time series prediction. *Lecture Notes in Mathematics Dynamical Systems and Turbulence, Warwick 1980*, pages 366–381, 1981.

[35] Vladimir Vapknik. *The Nature of Statistical Learning Theory*. Springer Science & Business Media, 2013.

[36] Nathan Wiebe, Ashish Kapoor, and Krysta M. Svore. Quantum deep learning, 2014.

[37] Wright, Logan. Reservoir quantum coupler. 2019.

[38] Yates, Andrew. Quantum amplitude feed-forward reservoir computers. 2020.