UNIVERSITY OF KWAZULU-NATAL

Determination of Quantum Entanglement Concurrence using Multilayer Perceptron Neural Networks

Author:
Lishen GOVENDER

Supervisors:
Prof. Francesco PETRUCCIONE
Dr. Ilya SINAYSKIY

Submitted in fulfilment of the requirements
for the degree of Master of Science

in the

School of Chemistry and Physics
University of KwaZulu-Natal

December 2017

Preface

The research contained in this thesis was completed by the candidate while based in the School of Chemistry and Physics of the College of Agriculture, Engineering and Science, University of KwaZulu-Natal, Westville Campus, South Africa. The research was financially supported by South African Research Chair Initiative (SARChI).

The contents of this work have not been submitted in any form to another university and, except where the work of others is acknowledged in the text, the results reported are due to investigations by the candidate. As the candidate's supervisors, we have approved this dissertation for submission.

Supervisor: Prof. Francesco Petruccione

Signed:

Date:

Co-Supervisor: Dr. Ilya Sinayskiy

Signed:

Date:

Declaration of Authorship

I, Lishen GOVENDER, declare that this thesis titled, "Determination of Quantum Entanglement Concurrence using Multilayer Perceptron Neural Networks" and the work presented in it is my own. I confirm that:

- The research reported in this thesis, except where otherwise indicated or acknowledged, is my original work.
- This thesis has not been submitted in full or in part for any degree or examination to any other university.
- This thesis does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons.
- I have acknowledged all main sources of help.
- This thesis does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in the report and in the References sections.

Signed: _____

Date: _____

Publications

Publication: L. Govender, I. Sinayskiy and F. Petruccione, "Determining concurrence using Artificial Neural Networks" (*In Preparation*)

"I think I can safely say that nobody understands quantum mechanics. "

– Richard P. Feynman

Abstract

Artificial Neural Networks, inspired by biological neural networks, have seen widespread implementations across all research areas in the past few years. This partly due to recent developments in the field and mostly due to the increased accessibility of hardware and cloud computing capable of realising artificial neural network models. As the implementation of neural networks and deep learning in general becomes more ubiquitous in everyday life, we seek to leverage this powerful tool to aid in furthering research in quantum information science.

Concurrence is a measure of entanglement that quantifies the "amount" of entanglement contained within both pure and mixed state entangled systems [1]. In this thesis, artificial neural networks are used to determine models that predict concurrence, particularly, models are trained on mixed state inputs and used for pure state prediction. Conversely additional models are trained on pure state inputs and used for mixed state prediction. An overview of the prediction performance is presented along with analysis of the predictions.

Acknowledgements

I would like to thank my project supervisors, Prof. Francesco Petruccione and Dr. Ilya Sinayskiy for their continuous support, invaluable advice and guidance, without which this work would not have been possible.

This research is supported by the South African Research Chair Initiative of the Department of Science and Technology and National Research Foundation.

Contents

Notations

$\mathcal{H}$	:	The Hilbert Space
$\langle .\vert$	:	A bra from Dirac's bra-ket notation
$\vert .\rangle$	:	A ket from Dirac's bra-ket notation
$\langle x\vert y\rangle$	:	Inner product of x and y
$\otimes$	:	Tensor Product
$\dagger$	:	Denotes complex conjugate transposition (Hermitian conjugation)
i	:	Imaginary unit equivalent to $\sqrt{-1}$
$\hbar$	:	Planck's constant
$\frac{\partial}{\partial t}$	:	Partial derivative with respect to time
μ	:	Reduced mass of the system
∇^2	:	The Laplacian
$\mathrm{Tr}(\varrho)$	:	Trace, sum of the diagonal of a square matrix
$\max\{x, y\}$	:	Returns the larger value between x and y
C	:	The actual concurrency of the entangled mixed/pure state.
$\hat{C}$	:	The predicted concurrency of the entangled mixed/pure state.
C_i^{error}	:	The unnormalised error differential.
$\bar{C}^{\mathrm{error}}$	:	The average of C_i^{error} across the entire dataset

Chapter 1

Quantum Information

Quantum Information is an intersection of quantum mechanics and information theory which seeks to use quantum mechanical properties to process or store information - or even a combination of both. It is a broad interdisciplinary field that investigates a quantum mechanical approach to a variety of information science fields; mainly computation and cryptography. The reason why quantum information research is of incredible importance is because there are properties inherent to quantum mechanics, that the theory suggests can provide significant improvements to classical computation amongst other things. The two key quantum properties that are central to this are superposition and entanglement.

This chapter serves as a Literature review to first build up the basic mathematical formalism of quantum mechanics that is used within quantum information, and then to also review quantum information theory with emphasis on the theory that is relevant to the investigation conducted for this thesis.

1.1 Mathematical Formalism of Quantum Mechanics

Quantum Mechanics is a fundamental field in physics which describes natural phenomena at the smallest scales of energy levels of atoms and subatomic particles. While the term "Quantum Mechanics" was coined in the early 1920's by a group of physicists at the University of Göttingen (this group included giants in the field such as Max Born, Werner Heisenberg and Wolfgang Pauli), the theory of Quantum Mechanics and the foundations underpinning the field, were laid as early as 1801 when Thomas Young conducted the double slit experiment and demonstrated the wave nature of light. We will briefly review the mathematical formalisms of quantum mechanics, with an emphasis on the theory that is central to understand quantum information.

The fundamental mathematic theory that will be covered in this section can be found in numerous linear algebra and mathematical physics textbooks [2]-[4].

1.1.1 Hilbert Space

Vectors in Hilbert Space

A Hilbert space $\mathcal{H}$ is a real or complex inner product space (a pre-Hilbert space as such) that is also considered to be a complete metric space with respect to the distance function that is induced by the inner product. When we state that $\mathcal{H}$ is a complex inner product space, this means that $\mathcal{H}$ is a complex vector space on which there is an inner product $\langle x, y\rangle$ associating a complex number to each pair of elements x, y of $\mathcal{H}$ that satisfies the following properties:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$ - The inner product of a pair of vectors is equal to the complex conjugate of the inner product of the swapped vectors.
- $\langle \alpha x_1 + \beta x_2, y\rangle = \alpha\langle x_1, y\rangle + \beta\langle x_2, y\rangle$ - It is linear in the first variable.
- $\langle x, \alpha y_1 + \beta y_2\rangle = \bar{\alpha}\langle x, y_1\rangle + \bar{\beta}\langle x, y_2\rangle$ - It is anti-linear in the second variable.
- $\langle x, x\rangle \geq 0$ - The inner product of an element with itself is positive definite.
- $||x|| = \sqrt{\langle x, x\rangle}$ - The norm is given by this real-valued function.
- $d(x, y) = ||x - y|| = \sqrt{\langle x - y, x - y\rangle}$ - The distance function, which implies that firstly it is symmetric with respect to x and y and secondly that the distance must be positive.
- $|\langle x, y\rangle| \leq ||x||\ ||y||$ - The equality holds if and only if x and y are linearly dependent. This inequality is the fundamental Cauchy-Schwarz inequality.
- If $x_1, y_1 \in \mathcal{H}_1$ and $x_2, y_2 \in \mathcal{H}_2$, then the inner product on the tensor product is defined as$\langle x_1 \otimes x_2, y_1 \otimes y_2\rangle = \langle x_1, y_1\rangle\langle x_2, y_2\rangle$.This formula then extends by sesquilinearity to an inner product on $\mathcal{H}_1 \otimes \mathcal{H}_2$.

Eigenvectors

David Hilbert coined the term spectral theory in his original formulation of Hilbert space theory, the theory encompasses the extension of eigenvalue and eigenvector theory from a single square matrix to a much broader theory of the structure of operators in a multitude of mathematical spaces.

In this section we briefly review the basis of spectral theory - eigenvalues and eigenvectors.

- In linear algebra, an eigenvector is a non-zero vector that only changes by a scalar factor when a linear transformation is applied to it. We formally identify it by the equation $T(\vec{v}) = \lambda\vec{v}$, where $T(\vec{v})$ is a linear transformation from a vector space V over a field F. λ is a scalar known as the eigenvalue associated with eigenvector $\vec{v}$.
- If vector space V is finite-dimensional, then the linear transformation T can be represented as a square matrix A and the vector $\vec{v}$ as a column vector which would render the mapping $T(\vec{v})$ as a matrix multiplication on one side and scaling a column vector on the other side, of which the equation is given by, $A\vec{v} = \lambda\vec{v}$

Hermitian Matrices

A Hermitian matrix is a complex square matrix with its defining property being that it is equal to its own complex conjugate transpose, ie every matrix element $a_{ij} = \overline{a_{ji}}$. As such, Hermitian matrices are considered to be a complex extension of real symmetric matrices.

Hermitian matrices are of significant interest to us because the matrix that describes a quantum system (density matrix) belongs to a subset of Hermitian matrices. It is therefore important to understand the properties of the Hermitian matrix. Some of the important properties of Hermitian matrices are as follows:

- All elements of the main diagonal (a_{ij} for $i = j$) are real, this necessary because they have to be equal to their complex conjugate.
- Due to complex conjugation, entries on the off-diagonal elements cannot be symmetric unless the matrix only has real entries.
- Every Hermitian matrix is a normal matrix ie, $AA^\dagger = A^\dagger A$.
- The sum of any two Hermitian matrices is Hermitian and the inverse of an invertible Hermitian matrix is Hermitian as well.
- Considering Hermitian matrix A with dimension n, all eigenvalues are real and matrix A will always have n linearly independent eigenvectors.

Unitary Matrices

A complex square matrix U is unitary if its conjugate transpose is equal to its inverse, i.e. $U^\dagger = U^{-1}$, consequentially it is then implied that $U^\dagger U = UU^\dagger = I$ where I is the identity matrix. The unitary matrix is the complex version of an orthogonal matrix.

Unitary matrices are of significant interest in quantum mechanics, since they retain the norms and thus the probability amplitudes. Some of the mathematical properties of unitary matrices that are of interest to us are:

- $U^\dagger U = UU^\dagger$ - Implies U is normal.
- $U = e^{iH}$ - Any unitary matrix U can be written in this form, where e is the matrix exponential and H is a Hermitian matrix.
- $U^\dagger HU = \Lambda = \mathrm{diag}[\lambda_1, ..., \lambda_n]$ - We can use a particular unitary matrix U to convert a Hermitian matrix H to a diagonal matrix Λ (all off diagonal elements of Λ are zero).
- The general expression of any 2×2 unitary matrix is:

$$U = \begin{bmatrix} a & b \\ -e^{i\varphi}b^* & e^{i\varphi}a^* \end{bmatrix},$$

  where a and b represents the phase and φ is the angle between a and b. It stands to reason then that $|a|^2 + |b|^2 = 1$. The determinant of a matrix in this form is given by $\det(U) = e^{i\varphi}$.

Schmidt Decomposition

The Schmidt decomposition is a way of expressing a vector as the tensor product of two inner product spaces. Its main application is in quantum information theory, in particular with determining whether a state is separable or not.

Theorem 1.1 Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be Hilbert spaces of dimensions n and m respectively. We assume that $n \geq m$. For any vector w in the tensor product space $\mathcal{H}_1 \otimes \mathcal{H}_2$, there exists orthonormal sets $\{u_1, ..., u_n\} \in \mathcal{H}_1$ and $\{v_1, ..., v_n\} \in \mathcal{H}_2$ such that $w = \sum_{i=1}^{m} \alpha_i u_i \otimes v_i$, where the scalars α_i are real, non-negative and as set, uniquely determined by w.

Consider that the vector w forms the rank 1 matrix $\rho = ww*$, then the partial trace of ρ (with respect to A or B) is a diagonal matrix whose non-zero elements are $|\alpha_i|^2$. Essentially the Schmidt decomposition shows that on either subsystem the reduced state of ρ, will have the same spectrum.

That brings us to the Schmidt rank, if we consider the values α_i (always positive) in the Schmidt decomposition of w, the values α_i are known as the Schmidt coefficients. The Schmidt rank is defined as the number of coefficients of w, counted with multiplicity.

If we can express w as a tensor product $u \otimes v$ then w is called a separable state, if we cannot express w like this then w is said to be an entangled state. Hence from the Schmidt decomposition, we can see that w is entangled if and only if w has Schmidt rank greater than 1 and consequentially two subsystems that partition a pure state are entangled if and only if their reduced states are mixed states.

The Von Neumann entropy (also known as the entropy of entanglement) is a direct consequence of this property of the Schmidt rank. (We give a brief overview of the Von Neumann entropy in section 1.3.2).

1.1.2 The Schrödinger equation

The Schrödinger equation is a fundamental equation in quantum mechanics that governs the time evolution of the wave function of a non-relativistic particle (system). We could say analogously that the Schrödinger equation is to a quantum mechanical particle (system) what Newton's second law is to a classical particle (system). We can solve the Schrödinger equation to determine how a quantum particle evolves over time, just as we can use Newton's second law to solve for a classical particles' future position and momentum.

In this section we briefly describe the Schrödinger equation mathematically and discuss quantum superposition, which mathematically refers to a property of solutions to the Schrödinger equation.

Time Dependent

The general form of the Schrödinger equation is the time-dependent Schrödinger equation which describes the system evolving with time,

$$i\hbar\frac{\partial}{\partial t}\Psi(\vec{r},t) = \hat{H}\Psi(\vec{r},t). \tag{1.1}$$

To solve or apply the Schrödinger equation, the Hamiltonian needs to be defined to account for the kinetic and potential energy of the particle/s. The resulting partial differential equation can be solved for the wave function which contains information about the system.

Time Independent

The time-dependent Schrödinger equation described in equation (1.1), predicts that wave functions can form "standing waves" called stationary states, that we describe using the time-independent Schrödinger equation. Understanding and solving for these

stationary states is of particular importance because it simplifies the task of solving for the time-dependent Schrödinger equation. The general time-independent Schrödinger equation is given by,

$$\hat{H}\Psi = E\Psi. \tag{1.2}$$

.
When we say that equation (1.2) is time-independent we mean that the Hamiltonian $\hat{H}$ is not dependent on time explicitly. However, even in this case the overall wave function still has a time dependency. Also from a linear algebra point of view, equation (1.2) is an eigenvalue equation and in this sense the wave function Ψ is an eigenfunction of the Hamiltonian operator with eigenvalues given by E.

Quantum Superposition

Quantum superposition is a key property of quantum mechanics that mathematically refers to a set of solutions of the Schrödinger equation. Essentially it means that any two or more quantum states can be superposed and the result will be another valid quantum state, basically since the Schrödinger equation is linear, any linear combination of solutions will also be a solution. Schrödinger famously described quantum superposition using the thought experiment we now call "Schrödinger's cat" [4].

Quantum superposition has many implications across quantum mechanics, but arguably the most interesting is that it led to the development of a new branch of quantum physics called quantum information. Quantum Superposition, along with quantum entanglement, form the basis of quantum information theory - to put it simply where classically we can reduce information and calculations to be described by a single bit, that's either 0 *or* 1 quantum superposition allows us to describe things in a superposition of 0 *and* 1,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle. \tag{1.3}$$

$|\psi\rangle$ is known as a qubit (quantum-bit) and where a classical bit is either 0 or 1 considering equation (1.3), our qubit $|\psi\rangle$ can be described by an infinite amount of superpositions of $|0\rangle$ and $|1\rangle$ provided the amplitudes α and β adhere to rule that the sum of the square of all amplitudes is equal to one. (in the case of equation (1.3) $|\alpha^2| + |\beta^2| = 1$) [4].

1.2 Basics of Quantum Information

Quantum Information is described as a field that merges quantum mechanics with computer science, on a fundamental level, the concept is rooted in the idea that quantum bits are analogous to classical bits while offering significant advantages (discussed in the previous section) and when combined with quantum entanglement theory, quantum information provides us with a more powerful way of processing and understanding information.

Historically the main research done in quantum information has been to develop the theoretical framework for a quantum computer as well as develop the framework for much more robust cryptographic methods (such as Quantum Key Distribution) that

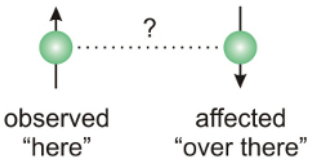

FIGURE 1.1: Central to the theory of entanglement is the idea that measuring the spin of one particle affects the spin of its entangled particle

uses the inherent properties of quantum entanglement (namely the no-cloning theorem). We've already discussed quantum superposition in the previous section so in this section we briefly discuss entanglement and then measures of entanglement with emphasis on the concurrence measurement which is being investigated in this thesis.

1.2.1 Entanglement

What is Entanglement?

Quantum entanglement is the physical phenomenon that occurs when particles interact in such a way that the quantum properties and quantum state of each particle cannot be described independently of the other particles, even when these particles are separated by a large distance the a quantum state must be described for the system as a whole [5].

History of Entanglement

The counter intuitive theory surrounding quantum entanglement was first explored in 1935 by Albert Einstein, Boris Podolsky and Nathan Rosen in a joint paper. In this paper they discuss the EPR (Einstein–Podolsky–Rosen) Paradox [6], which in essence is about the fact that particles that have quantum properties can interact in a way that makes it possible to measure both the position and the momentum of the particles more accurately than Heisenberg's uncertainty principle would allow.

The only way Heisenberg's uncertainty principle was not being invalidated was if measuring one particle would instantaneously affects the other. This idea posed a new problem, this implied superluminal communication between particles, something that should be impossible according to the theory of relativity. The research paper goes on further to discuss that the EPR paradox shows that quantum theory was incomplete and should be extended with "hidden variables" [6]. This "hidden variables" idea was that the state of the particles being measured carried some hidden variables, whose values would determine, right from the moment of separation, what the outcomes of the spin measurements were going to be.

In 1964 John Bell proposed a mechanism to test for the existence of these hidden variables posited by the EPR paradox, and he developed his famous inequality as the basis for such a test. He showed that if the inequality were ever not satisfied, then it would be impossible to have a local hidden variable theory that accounted for the spin experiment. This led to John Bell's now famous theorem [7], which in its simplest form states:

Theorem 1.1: *No physical theory of local hidden variables can ever reproduce all of the predictions of quantum mechanics.*

Implicit to this theorem is the idea that the deterministic nature of classical physics is essentially not capable of describing quantum mechanics. Bell expanded on the theorem to supply what would become the conceptual groundwork for the Bell test experiments, which would eventually confirm the validity of his statement and quash Einstein's proposition of hidden variables. What had previously been a philosophical debate suddenly had testable consequences. Bell's experiment has since been performed in the laboratory, verifying the predictions of quantum mechanics and dramatically contradicting local hidden-variable theories. Mathematically we can clarify the statement in Theorem 1.1 by using the CHSH inequality, which is a generalisation of Bell's inequality. CHSH stands for John Clauser, Michael Horne, Abner Shimony, and Richard Holt, who described it in a much-cited paper published in 1969 [41]. The inequality is given as,

$$C_h(a,c) - C_h(b,a) - C_h(b,c) \leq 1,$$

where C_h denotes the correlation as predicted by any hidden variable theory and a, b and c refer to three arbitrary settings of the two analysers. Bell inequalities concern measurements made by observers on pairs of particles that have interacted and then separated. This inequality is explained in great detail in previously published literature (for example [41]) and is still of much interest to physicists that continue to research Bell inequalities to improve our understanding on quantum entanglement [42].
The simplest example of entanglement is represented by the Bell states. The Bell states are four specific maximally entangled quantum states of two-qubits states and are defined as:

$$\begin{aligned}
|\Phi^-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |0\rangle_B - |1\rangle_A \otimes |1\rangle_B), \\
|\Phi^+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |0\rangle_B + |1\rangle_A \otimes |1\rangle_B), \\
|\Psi^-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B), \\
|\Psi^+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B + |1\rangle_A \otimes |0\rangle_B).
\end{aligned} \tag{1.4}$$

Basic Formalism of Bipartite Entanglement

While entanglement is often considered to just be a property in quantum physics, it is of much interest to researchers due to the huge potential of quantum information and quantum computation. This section will only cover two qubit entanglement mainly for the sake of brevity but also because the entangled states that are investigated in this thesis are only two qubit entangled pure and mixed states. First, let us consider how to determine if a two qubit pure state is entangled or not.

Proposition 1.1: Any bipartite pure state $|\Psi_{AB}\rangle \in \mathcal{H}_{AB} = \mathcal{H}_A \otimes \mathcal{H}_B$ is called entangled if and only if it can not be written as a product of two vectors corresponding to Hilbert spaces of subsystems: $|\Psi_{AB}\rangle = |\psi_A\rangle \otimes |\phi_B\rangle$.

While proposition 1.1 deals with all two qubit pure states [8], unfortunately due to the decoherence phenomenon, in laboratories we unavoidably deal with mixed states rather than pure states, so we need to consider how to determine if a two qubit mixed state is entangled or not.

Proposition 1.2: Any bipartite state ϱ_{AB} defined on Hilbert space $\mathcal{H}_{AB} = \mathcal{H}_A \otimes \mathcal{H}_B$ is separable if and only if it can neither be represented nor approximated by the states of the form $\varrho_{AB} = \sum_{i=1}^{k} (p_i \varrho_A^i \otimes \varrho_B^i)$ where ϱ_A^i and ϱ_B^i are defined on local Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$.

In conclusion, to decide whether or not a two qubit mixed state is entangled or not, we apply Proposition 1.2 to the mixed state and we find that the mixed state is not separable then we deduce that it is entangled [8].

1.3 Measures of entanglement

When quantum theory was in its infancy, entanglement was mostly considered to be a qualitative feature of quantum theory and it was the quantum behavior that most noticeably distinguished quantum theory from classical physics theory and intuition. The eventual development of Bell's inequalities has made entanglement distinctly quantitative.

There are quiet a few properties that are governed by the non-separability of an entangled quantum system, properties such as momentum, position, spin and polarisation are all affected by the "entanglement" of the system, but none of these describe *how* entangled the system is or rather it's not possible to determine if we can quantify the level or "amount" of entanglement by comparing these values on its own in its current state, thus several different measures of entanglement were developed and are studied according to different goals. We will briefly discuss some of the more popular ones.

1.3.1 Positive Partial Transpose (PPT)

If we consider the density matrix ρ which is a density matrix that represents two quantum mechanical systems ρ_A^1 and ρ_A^2 (Note that the superscripts 1 and 2 denotes subsystem 1 and sub-system 2), the criterion by which we determine whether or not the systems ρ_A^1 and ρ_A^2 are separable or not is called the positive partial transpose (or Peres-Horodecki criterion) [9].

Particularly, it is used to determine the separability of mixed states where Schmidt decomposition does not apply. It is however important to note that the PPT is only conclusive in the 2×2 and 2×3 dimensional cases, in higher dimensions more advanced tests would need to done to determine separability. We define the PPT as follows:

The density matrix ρ has to be separable as a sum of direct products,

$$\rho = \sum_A w_A(\rho_A^1) \otimes (\rho_A^2), \tag{1.5}$$

where the positive weights w_A satisfy $\sum w_A = 1$. The partial transpose is then defined as,

$$\rho^{T_B} = I \otimes T(\rho), \tag{1.6}$$

only part of the state is transposed, to be more precise $I \otimes T(\rho)$ is the identity map applied to the system ρ_A^1 and the transposition map $T(\rho)$ applied to the system ρ_A^2. The PPT definition is better understood if we define ρ in matrix form.

$$\rho = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & & \\ \vdots & & \ddots & \\ A_{n1} & & & A_{nn} \end{pmatrix}, \tag{1.7}$$

then the partial transpose is given by;

$$\rho^{T_B} = \begin{pmatrix} A_{11}^T & A_{12}^T & \dots & A_{1n}^T \\ A_{21}^T & A_{22}^T & & \\ \vdots & & \ddots & \\ A_{n1}^T & & & A_{nn}^T \end{pmatrix}. \tag{1.8}$$

Where $n = \dim\mathcal{H}_A$, and each element (A_{ij}) is a square matrix of dimension $m = \dim\mathcal{H}_B$.

The PPT criterion essentially states that if ρ is separable, then the partial transpose matrix ρ^{T_B} is made up of density matrices with non-negative eigenvalues. Conversely we can say that if density matrix ρ^{T_B} has a negative eigenvalue then density matrix ρ will certainly be entangled.

1.3.2 Entanglement of Formation

In 1998 William K Wootters published an article in the Physical Review Letters (PRL) [1] that introduced us to the idea of entanglement of formation as well as concurrence in quantum entanglement. Entanglement of formation shares a lot of similarities with entanglement cost, namely that entanglement of formation quantifies how many Bell states (equation (1.4)) are needed to prepare many copies of $|\psi\rangle$ using the LOCC procedure. In the same article he showed that we can determine the concurrence C for both a pure state and a mixed state of two entangled qubits that could eventually be used to determine the entanglement of formation but is a measure of entanglement in its own right.

1.3.3 Von Neumann Entropy

A two qubit pure state system can be written as a superposition of basis states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. Each basis state has an associated amplitude α such that the two qubit entangled system can be described as follows,

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle, \tag{1.9}$$

where the probability of measuring a certain basis state $|ij\rangle$ is given by $|\alpha_{ij}|^2$.The bipartite Von Neumann entropy E is defined with respect to a bipartition of our state into

two partitions A and B. Then for a given pure state density matrix $\rho_{AB} = |\Psi\rangle\langle\Psi|_{AB}$ the bipartite Von Neumann entropy is given by

$$E = S(\rho_A) = -[\rho_A \ln \rho_A] = -\mathrm{Tr}[\rho_B \ln \rho_B] = S(\rho_B), \tag{1.10}$$

where $\rho_A = \mathrm{Tr}_B(\rho_{AB})$ and $\rho_B = \mathrm{Tr}_A(\rho_{AB})$ and E ranges from 0 to $\ln 2$ with 0 being a quantum state that is not entangled and $\ln 2$ being a quantum state that is maximally entangled [10].

1.3.4 Purity

Purity is a measure of quantum mixed states, giving information on how much a state is mixed. While purity is not a measure of entanglement, it is a useful measure that can assist in better understanding our entangled mixed states in comparison to the entangled pure states. Purity is a scalar defined as,

$$\gamma \equiv \mathrm{Tr}(\rho^2), \tag{1.11}$$

where ρ is the density matrix of a mixed state. The purity ranges from $\frac{1}{d}$ to 1 where d is the dimension of the Hilbert space upon which the mixed state is defined, so in the case of bipartite entanglement for the reduced state of each qubit would be $0.5 \leq \gamma \leq 1$, or $0.25 \leq \gamma \leq 1$ for two qubits [8]. For the case when $\gamma = 1$ this implies that our state is a pure state.

1.4 Concurrence

Concurrence C ranges between 0 to 1 with 0 meaning the qubits are not entangled and 1 meaning that the qubits are maximally entangled, the four Bell States (see equation (1.4)) for example are all maximally entangled ($C = 1$). For our research we will only be dealing with everything in-between, that being systems with concurrence values greater than zero and less than one. We will discuss the theory and mathematical formulation of how to determine the concurrence of a two qubit system (both pure and mixed states) as concurrence is the measure of entanglement that is central to our research.

1.4.1 Concurrence determination of an entangled two-qubit pure state

Consider density matrix ρ that represents a pair of quantum systems A and B, all possible pure-state decompositions of ρ with all ensembles of states $|\psi_i\rangle$ and probabilities p such that,

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|. \tag{1.12}$$

Wootters formula for entanglement makes use of the "spin flip" transformation. For a single qubit pure state the spin flip is defined by

$$|\tilde{\psi}\rangle = \sigma_y |\psi^*\rangle, \tag{1.13}$$

where $|\psi^*\rangle$ denotes the complex conjugate of $|\psi\rangle$ and σ_y denotes the Pauli matrix $\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$.
With the introduction of the spin flip we can now define the concurrence C for a two qubit pure state as

$$C(\psi) = |\langle\psi|\tilde{\psi}\rangle|. \tag{1.14}$$

Conveniently, for a two qubit pure state we can use only the probability amplitudes of the basis states to determine the concurrence. Consider the general pure state of a two qubit system $|\psi\rangle = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle$, using equation (1.13) along with equation (1.14), we get

$$C(\psi) = \langle\psi|\sigma_y \otimes \sigma_y|\psi^*\rangle = 2|ad - bc| \tag{1.15}$$

1.4.2 Concurrence determination of an entangled two-qubit mixed state

Consider density matrix ρ that represents a two qubit mixed state, the concurrence C of this system is given by

$$C(\rho) = \max\{0, \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4\}, \tag{1.16}$$

where the λ_i's are the eigenvalues in decreasing order ($\lambda_1 > \lambda_2 > \lambda_3 > \lambda_4 > 0$) of the Hermitian matrix

$$R \equiv \sqrt{\sqrt{\rho}\tilde{\rho}\sqrt{\rho}} \tag{1.17}$$

where $\tilde{\rho}$ denotes the spin-flipped state of ρ; $\tilde{\rho} = (\sigma_y \otimes \sigma_y)\rho^*(\sigma_y \otimes \sigma_y)$.

Equation (1.16) is also valid for determining the concurrence of pure state density matrices [1].

Chapter 2

Machine Learning

Machine learning is a sub-discipline of computer science that can be described as a way to give computers the ability to learn without being explicitly programmed [11]. Machine Learning has far reaching consequence in modern society and everyday life. What is interesting is that we rarely consider or even know that a machine learning process has taken place to enable an action on a piece of technology.

Examples range from a camera recognising a persons face and thus being able to focus on it, to a search engine like Google auto-filling your search bar depending on various analytics like your search history, popular trending searches at that very moment and the websites you have visited immediately before making the search. These and countless more implementations are a direct result of the progress that continues to be made in the field of machine learning.

2.1 Evolution of Machine Learning

Machine Learning in computer science can be traced as far back as the 1950s when researchers were using simple algorithms derived from statistical methods to process data in a far more autonomous way then we were used to.

While machine learning and artificial intelligence as a whole were both very interesting topics throughout the twentieth century, interest in the field died down for a very long time, in particular because computers were just not powerful enough to drive the research experimentally. That changed in the 1990s when personal computers became more accessible, the widespread availability of much more powerful computers reignited interest in machine learning and all things computer science. The result was that machine learning research experienced a renaissance. Research shifted from being knowledge-driven in the past to a more elegant data-driven approach [12].

In the twenty-first century machine learning began to gain momentum. Kernel methods such as support vector machines and artificial neural networks lead the way in solving a host of problems unique to machine learning - for instance handwriting recognition, and complex multi-variable classification problems such as protein classification in medical science [13].

It is 2017 and machine learning is at the peak of its powers and continues to get more influential and pervasive in life as we know it. Businesses and governments are collectively accumulating petabytes of data on its customers and citizens and they are using machine learning to process that data. Data has become a valuable resource; arguably

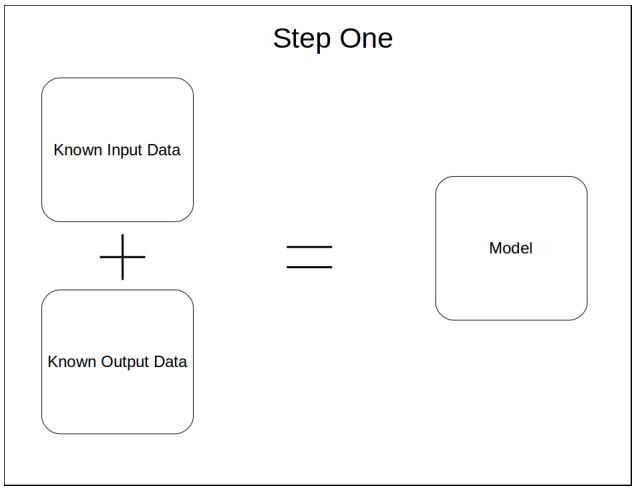

FIGURE 2.1: The first step of a supervised learning method is to take known inputs and known ouputs and use that to generate a model.

the most valuable resource given that the top 3 most valuable companies in the world are Apple, Alphabet (Google) and Amazon, three tech companies that are known for their data-centric business models [14].

Apart from data processing, machine learning has a few other important applications. The one most relevant is object recognition; from the computer reading handwriting to it identifying people and other objects in images and videos. Improvements in a computer's ability to recognise and correctly classify an object is of increasing importance and as we develop better ways to do this efficiently and accurately it will lead to many advancements in society. Self driving cars will benefit greatly from being able to better understand the world around them, a dual camera system with depth perception can replace a plethora of current authentication methods such as security cards or keys; or even be used in combination with current security measures to increase the security of a system.

2.2 Popular Machine Learning techniques

There are three major machine learning fields. Supervised learning, unsupervised learning and reinforcement learning. Within those fields there are many machine learning techniques that are used, we will briefly review some of the more popular ones.

2.2.1 Supervised Learning

Supervised learning, in the context of machine learning, is a system in which both the input data and desired output data are provided. Both the input and output data are labelled for classification to provide a learning basis. From this learning basis a model is developed which will be able to process new input data and find the correct output based on the data that the model was trained with. This can crudely be seen as a two-step process.

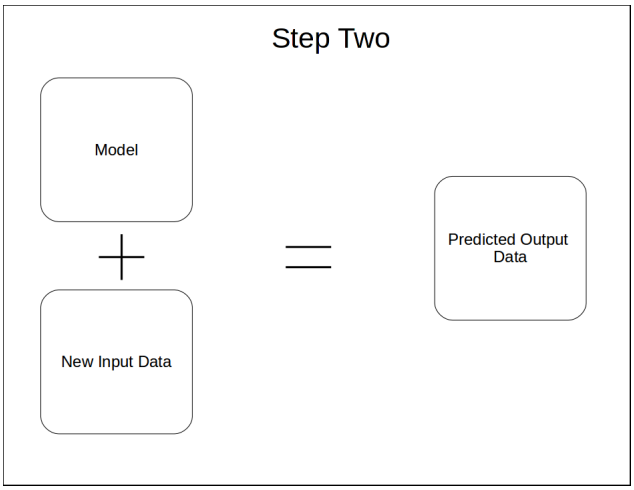

FIGURE 2.2: The second step of a supervised learning method involves using the model from Step One, along with new unseen input data, and then generating new *predicted* output data.

All supervised learning methods use the process shown in Fig. 2.1 to create a model, some methods explicitly determine a model in one go and some methods will iteratively update the model using Baysian inference methods or a combination of both.

Once a model has been trained and tested to ensure it does not over-fit the data, it can then be implemented. Implementation is where the magic happens so to speak. The model takes in completely new data and churns out predicted outputs as shown in Fig. 2.2. One of the strengths of machine learning is that model implementation (Fig. 2.2 - step two) does not take nearly as much processing power as it takes to train a model (Fig. 2.1 - step one), this key characteristic of machine learning in general is the driving force behind the growing popularity and reliance on machine learning. Models can be created and trained on powerful supercomputers and clusters and then be implemented almost anywhere from a personal laptop to a cellphone.

Regression Analysis

Regression analysis encompasses a wide range of machine learning techniques, with the two most ubiquitous ones being linear regression and logistical regression.

Linear regression is usually any person's first encounter with machine learning, it scales up and down very well and is fairly intuitive to understand. The most basic single variable linear regression takes the form of $y = ax + b$ where a and b are fixed values and x is the input variable. We can use this to predict an output y using the process shown in Fig. 2.1 and Fig. 2.2.

Logistical regression is somewhat similar to linear regression except that the outputs are binary, this is useful for "Yes/No" problems or classification problems with two classifiers. Logistical regression is often used in combination with various other machine learning methods and is the foundation of more sophisticated machine learning methods such as decision tree learning and random forest learning.

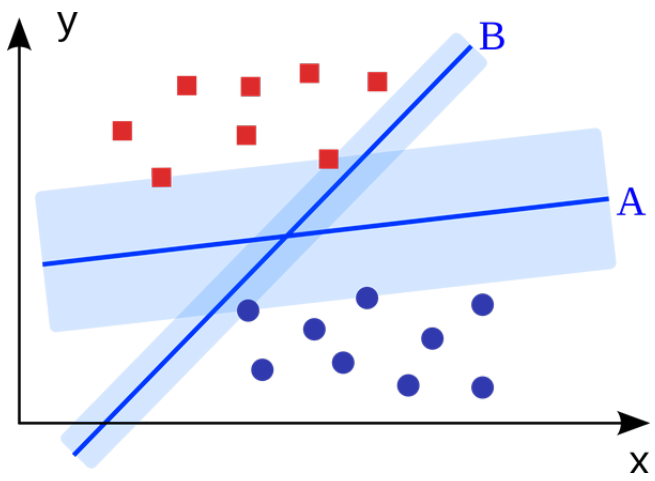

FIGURE 2.3: A support vector machine with two possible hyperplanes.

Support Vector Machines

A Support Vector Machine (SVM) is a supervised binary classification machine learning method. For example, given a set of points of two types in N dimensional place, a support vector machine generates a dimensional hyperplane to separate those points into two groups.

A support vector machine model is a representation of the input data as points in space, mapped in a way that the inputs of the separate categories are divided by a clear distance that is as wide as possible. If we consider Fig. 2.3 we see that both hyperplane A and hyperplane B are valid hyperplanes for separating the groups, but hyperplane B comes much closer to both the blue and red inputs, thus hyperplane A is a more optimal hyperplane for classifying the inputs. New inputs are then mapped into the same space and the result is that we use the hyperplane to make a prediction on the category of the new inputs.

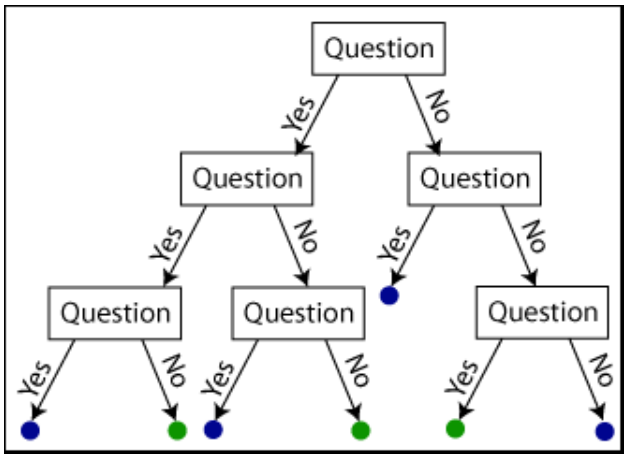

FIGURE 2.4: A simple example of the structure of a decision tree, there are only two possible outcomes, blue or green but there are multiple ways to get either outcome.

Decision Tree

Decision tree is a supervised learning algorithm that is mainly used in classification problems. In decision tree we split the dataset or sample into two or more homogeneous sets based on the most significant differentiator in input variables.

As we see from Fig. 2.4, the aim of the decision tree is to investigate only the relevant features during the outcome prediction process, this will not only increase the accuracy of the prediction, but significantly reduce the resources required to make that prediction.

Random Forest

The name random forest is given aptly because this machine learning method combines many decision tree algorithms. The combining of many decision trees not only allow for more complex models but also aids in reducing variance of a single decision tree [15].

2.2.2 Unsupervised Learning

Unsupervised machine learning is the branch of machine learning that infers a function or model to determine and describe hidden structures from "unlabelled" data. Since the data that we input are unlabelled, there is no way to validate the accuracy of the model. This learning of unlabelled data to give us an output that we cannot determine the accuracy of is arguably the main characteristic of unsupervised learning since it is what distinguishes it the most from supervised learning and reinforcement learning.

The learning process does not require labels and thus this unsupervised learning is completely data driven and hence better suited to finding the underlying structure of the data, structures which may sometimes be too abstract or unintuitive for a person to see especially when considering big data. While this is the main advantage of unsupervised learning it is also in a way its main disadvantage, since not having labels

and validation data means that the unsupervised learning algorithm could find nonsensical patterns that do not have significance and by the time this is realised a lot of computational resources have already been spent.

K-means clustering

K-means clustering is a technique for finding groups of data that are similar within a given dataset, these groups are called clusters. For example it attempts to group individuals in a population together by similarity, but it is not driven by a specific purpose nor does it have any idea on what kind of groups it will form (main feature of unsupervised learning).

K-means clustering has found an important application in feature learning [16] on big data, as mentioned in Chapter 2.2, we often do not know if we can find useful patterns in a "big-data" data set and that is mostly because we often do not know the features contained within the dataset. K-means clustering provides the means to learn the features or rather the labels of the dataset and once we have that we can use a supervised learning algorithm to try and find patterns within those labels.

A simple example of feature learning is as follows, imagine we need to process 1 billion jpeg photographs and we want to know about all of the different objects within those images. K-means provides a way for us to label all of those objects autonomously and then once we have the labels for all the objects we can use those labels with our images to train a supervised learning algorithm to find out where those objects appear in new photographs, without the feature learning power of K-means clustering it would take a lot of time to find all the labels within a dataset that contains one billion images.

Apriori algorithm

The Apriori algorithm derives its name from the Latin phrase "*a priori*" which means logic or knowledge derived from theoretical deductions rather than from experience or observation. The apriori algorithm shares many similaries with K-means clustering, however they do have a few differences that make each method better for different situations.

K-means clustering is computationally less sensitive to unnormalised data which makes it more robust with predictions involving datasets with large variance. The apriori algorithm thanks to its simplicity is easier to implement, and easier to parallelise across multiple central processing units (CPU's) or graphical processing units (GPU's) making training more efficient [17].

2.2.3 Reinforcement Learning

While reinforcement learning is not very commonly used in research or industrial applications, it is arguably the machine learning method that has the most potential. Reinforcement learning is a concept that is very familiar to us because it is how we all begin learning from infancy. An oversimplified example - a baby taking its first step, takes a large step and falls down (punishment), then takes a smaller step and keeps its balance (reward). The baby now knows smaller steps are better because the reward is

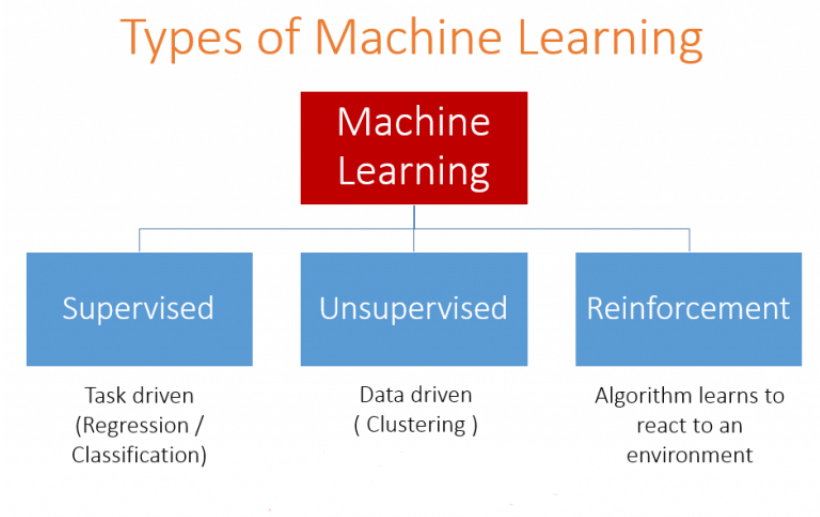

FIGURE 2.5: The three branches of machine learning [18].

not falling down - and that is basically what reinforcement learning is. It is a goal oriented reward vs punishment system. Since humans (it can be said that most animals) learn using reinforcement learning at a fundamental level, reinforcement learning is one of our best hopes for true artificial intelligence.

We have briefly discussed the three branches of machine learning, so to summarise and clearly differentiate between the three branches of machines learning, lets take a look at the differences.

As we can see from Fig. 2.5 supervised learning requires the most information with regards to the model development, it is task driven so it is very well defined and needs to have both the data and labels in order to learn. Unsupervised learning, while not needing labels, still needs data in order to learn, in order to recognise patterns and in most cases determine how to classify the data. Reinforcement learning on the other hand does not require either of these things and as such learns in a more organic way. While we are a long way off from having the necessary computational power to implement a reinforcement learning algorithm capable of taking in vast amounts information from the environment and learning in the way that a biological brain would, machine learning practitioners are actively developing the framework which would hopefully facilitate the creation of true artificial intelligence.

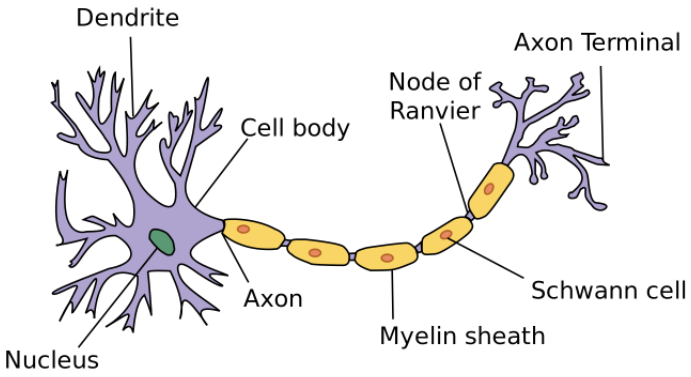

FIGURE 2.6: The structure of a typical neuron [19].

2.3 Artificial Neural Networks

To begin understanding an artificial neural network (ANN) we first need to look at what it is inspired by - a biological neural network. Humans, like nearly all other animals, have a brain. In a human brain there are billions of neurons. At a fundamental level these neurons are responsible for our ability to "think", it stores our memories, when our eyes see something or our ears hear something, those visual and aural signals are processed by the neurons in our brain, an example of the structure of typical neuron in a human brain is shown in Fig. 2.6.

While the billions of neurons in our brain serve thousands of different functions, the structure of these neurons are all the same, this is a remarkable thing to consider, that at the most basic level the neurons in our brains are all the same. Whether it is responsible for our ability to see, hear, touch, move, complete simple tasks like breathing or blinking and complex tasks like building a house or driving a car - the structure of the neurons that are responsible for all of those tasks remain the same.

The fact that all our tasks are completed by neurons that are the same is what led to the inspiration for artificial neural networks. If our brains can learn a multitude of tasks from these biological neurons, then maybe we can develop artificial neurons to learn a multitude of tasks as well.

2.3.1 Development of Neural Networks

In 1943 Warren McCulloch and Walter Pitts created a mathematical model for neural networks called threshold logics [20]. This work led to the application of neural networks to artificial intelligence. Through the 1950s and 1960s the underlying theory of artificial neural networks were developed, in 1958 Frank Rosenblatt introduced the idea of the perceptron, an algorithm for pattern recognition [21]. In 1969 machine learning research by Seymour Papert and Marvin Minsky showed a key issue with the computational machines that processed neural networks, that being, computers did not have enough processing power to adequately handle the vast amount of mathematical computations required by large neural networks, this discovery led to the stagnation of ANN research [22].

Six years on from Papert's and Minsky's critical research, interest in artificial neural networks were renewed thanks to work done by Paul Werbos on the back-propagation algorithm which accelerated the training of multi-layer networks. Beyond this interest in artificial neural networks steadily grew and while other machine learning methods such as support vector machines and regression models were more popular, ANN theory was still being tentatively developed as researchers realised computational devices were getting more powerful at a rate that would make ANN machine learning possible in the near future.

One of the most important events with regards to the development of artificial neural networks and possibly a key turning point in the field was Yann LeCun's application of neural networks to character recognition in 1998, his convolutional neural networks (CNN's) or ConvNets [23] as he called it sparked widespread interest in artificial neural networks across the machine learning and computer science field. We are into the 21^{st} century and the implementation of artificial neural networks have slowly become more prevalent in all facets of life, some would even say more pervasive when combined with the Internet of Things (IoT) and the exponentially growing amount of data that companies have accumulated (Big Data) artificial neural networks look set to change the world and how we live in it.

2.3.2 Advantages and Disadvantages of Neural Networks versus other machine learning techniques

Artificial neural networks have a host of advantages over other machine learning techniques, ranging from it is ability to approximate any function regardless of its linearity to significant improvements in model accuracy when compared to other machine learning techniques, but the greatest advantage that artificial neural networks have over other machine learning techniques is its scalability. Its ability to train huge amounts of data relatively quickly especially when making use of CPU and GPU parallelisation, has made it the model of choice of most big data implementation, especially data that is heteroscedastic due to the hidden layers in an ANN being able to learn "hidden" non-linear relationships in the data without them being explicitly defined or known to begin with.

Of course artificial neural networks are not without its disadvantages. If there were no disadvantages it would make other simpler models like regression and decision trees obsolete. One of the key disadvantage is that artificial neural networks are essentially a "black box" so once a model is trained, even if it works well and achieves excellent prediction accuracy, it is difficult to understand why and the machine learning practitioner can only speculate and guess why the model worked so well and what hidden non-linear relationships the ANN has "discovered" to aid with prediction. Apart from that artificial neural networks do not seem to work as well on small datasets as it does on large datasets, and often times increasing the dataset or number of perceptrons are the only ways to improve model accuracy, increasing both of these things can result in a significant increase in computational power requirements.

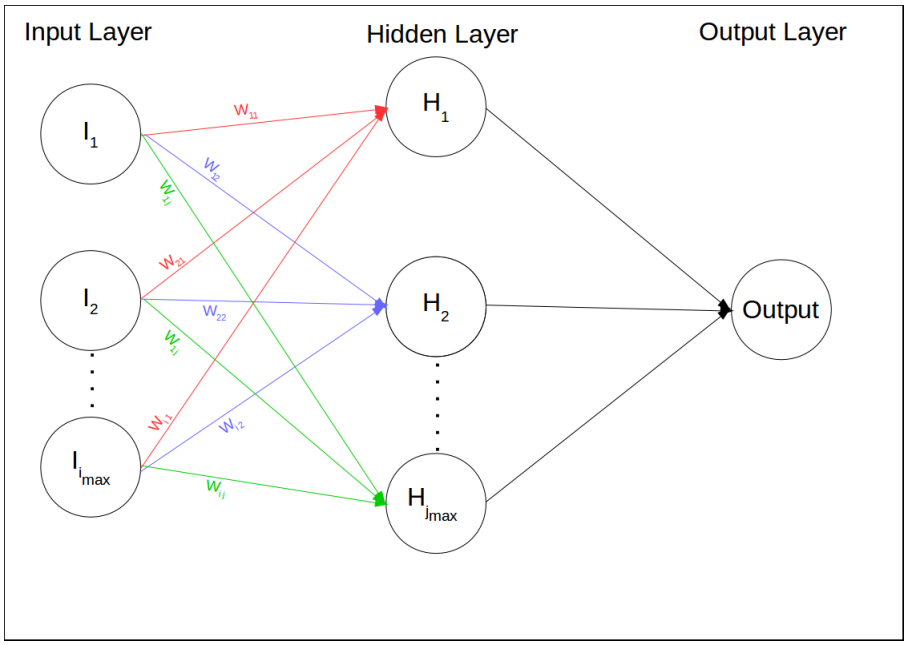

FIGURE 2.7: The basic structure of a simple artificial neural network.

2.3.3 Basic Topology of an Artificial Neural Network

We have already looked at the structure of a biological neural network and we have discussed how the theory surrounding artificial neural networks was developed. We now take a brief look at the mathematical structure of a basic ANN and then review some of the popular ANN structures and implementations.

An artificial neural network at the very least has 3 layers, an input layer, a hidden layer and an output layer. It is important to note that an ANN can have any number of hidden layers and varying the number of hidden layers can greatly assist with solving non-linear problems. The input layer is the layer that data is fed into. Generally we need one neuron per feature in our data and some neural network configurations add a neuron for the bias term which we will discuss later. The hidden layer(s) don't have a specific structure or a general rule to follow, given the nature of artificial neural networks the number of hidden layers and the number of neurons per layer are tweaked on a model basis and while there are "rules of thumb" to consider when determining hidden layer structure, there is currently no way to determine what the ideal hidden layer structure would be for a particular model.

Now that we have a better understanding of the neural network topology, let us discuss the mathematics behind how they work. Artificial neural networks are inspired by how biological neural networks work, ANNs however are not nearly as complex, in fact ANNs are essentially a whole lot of matrix multiplications - or to generalise especially when considering Convolutional Neural Networks - tensor products. If we consider the simple model in Fig. 2.7 there are four inputs (I_i) and thus the input layer has four neurons. We have five neurons in our hidden layer (H_j), these hidden neurons are each initialised to different random values between $0 - 1$. Each input makes a connection to each hidden layer. A product occurs between I_i and H_j, these are called

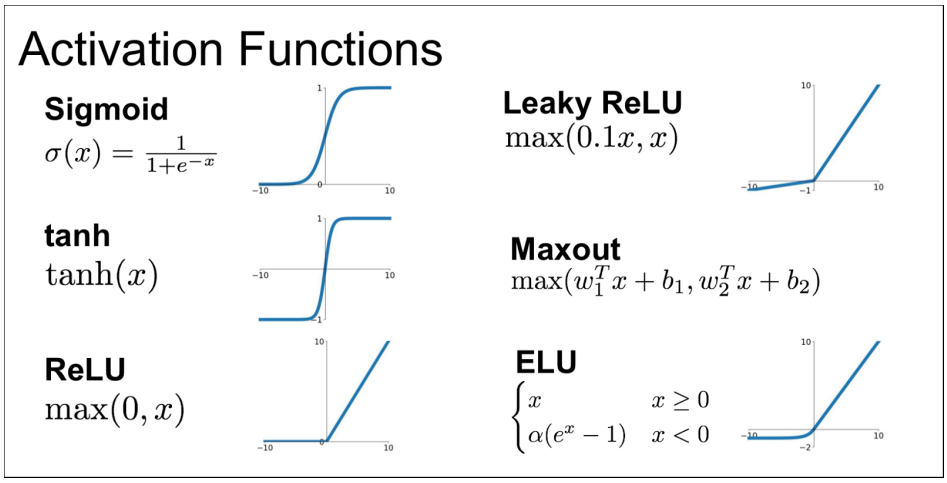

FIGURE 2.8: Some of the popular activation functions that are used[24].

weights (W_{ij}), since our hidden neurons are initialised to different values, the weights are all different values. Our model has a total of twenty weights between our input layer and hidden layer.

Each neuron in the hidden layer will consider each weight and send this weight through an activation function. We use different activation functions based on the type of problem. For example, a sigmoid function as shown in Fig. 2.8 may be preferred for a problem with a binary output (Yes or No) while a rectified linear unit (ReLU) function may be preferred for a problem with a probabilistic output. After passing through the activation functions these weights are summed for each neuron in the hidden layer and they are sent to the output neuron. Again the output neuron will pass all of the weights it received through another activation function and then determine an output. This predicted output is compared to the actual output and depending on how close the prediction was, the hidden layers will tweak the H_j value using an optimiser such as gradient descent and repeat the whole process to try and make a better prediction. If the new prediction improves, it keeps making small changes in the same direction. If the prediction gets worse the it will make larger changes in the opposite direction.

The process of "looking over" the data and passing it through the layers is known as the training phase. This is when our network learns. After the network has looked at all the data we then go into the validation phase. We input new data that's unseen by the network and compare the predicted outputs to the known outputs. This phase is to ensure that the errors in the training phase are similar in magnitude to errors of unseen data. This is done to minimise over-fitting.

After we have trained our network and validated the prediction to make sure there is no over-fitting we then go on to the testing phase. The testing phase is done after we've completely trained our model and we want to see how well the model actually performs. So to summarise:

Training Phase: Neurons learn and generate a prediction, weights are created between the input layer and hidden layer and these weights change as our neural network

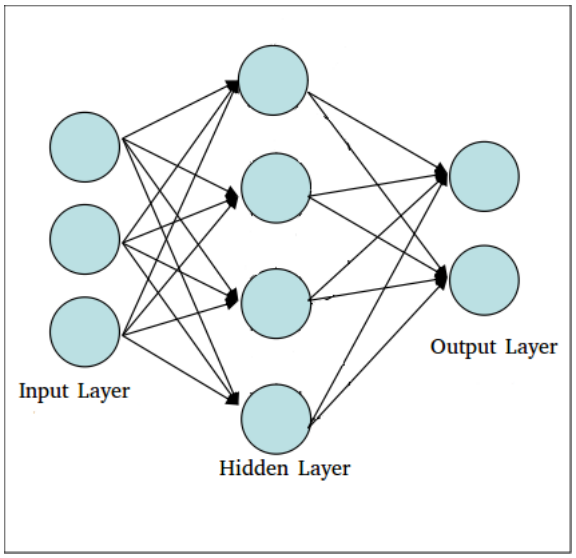

FIGURE 2.9: The basic structure of a feed-forward neural network.

learns.

Validation Phase: The network takes in new inputs and generates predictions. If these predictions show the same error margins as the training data then all is well and the training phase begins again to reduce the error. If the error margin is different, this means the model is over-fitted, training resumes but this time with reinitialising some or all of the weights, depending on the optimiser used.

Testing Phase: The model has been trained and the validation phase shows that there is minimal over-fitting We then give our neural network new unseen input data and see if it can make predictions as accurately as it was during the training and validation phase. If it can then the model is ready to be implemented.

2.3.4 Popular types of Artificial Neural Networks

In section 2.3.3 we gave a generalised and simplified explanation for how artificial neural networks work. However, not all artificial neural networks work this way, in fact there are many different types of artificial neural networks and some of them work very differently from one another. We will briefly review some of the more popular types of artificial neural networks.

Multi-Layer Perceptron Neural Network

A multi-layer perceptron (MLP) contains one or more hidden layers (apart from one input and one output layer). While the single-layer perceptron neural network contains no hidden layers (it has just an input layer and an output layer) and can only learn linear functions, multi layer perceptron can also learn non-linear functions.

Multi-layer perceptron neural networks belong to the class of neural networks called feed-forward neural networks (see Fig. 2.9); this simply means that connections are formed in the forward direction between layers. For example if the network has two

FIGURE 2.10: On the left hand side is a normal image of a dog playing outside. On the right hand side is the same image, but convolved with itself.

hidden layers, the input will pass through the first layer, then from the first hidden layer it will pass go through the second hidden layer and then the output.

Convolutional Neural Network (ConvNet)

Convolutional neural networks or ConvNets also belong to the feed-forward neural network class. ConvNets share many similarities with multi layer perceptron networks. The difference, however, is that the ConvNet can have one or many hidden convolutional layers; these convolutional layers are the core building blocks of a ConvNet.

To differentiate between a ConvNet and multi-layer perceptron without getting into the mathematical and statistical theory, we can say that the key differences are that:

1. A lot of the weights are forced to be the same - think of this to be analogous to what happens when an image is being convolved (see Fig. 2.10). This significantly improves training time.

2. Many of the weights are forced to be zero, again to reduce training time.

3. A sub-sampling happens so that the layers get progressively smaller. We can think of this as simple image resolution down scaling. Like the previous two points, the aim of this is to reduce training time.

Considering the differences between multilayer perceptrons and ConvNets then, we can see that the key idea is to reduce training time, this is particularly useful for image recognition, where for example sub-sampling our data (pixels) does not destroy the integrity of it or make it significantly difficult to recognise a pattern.

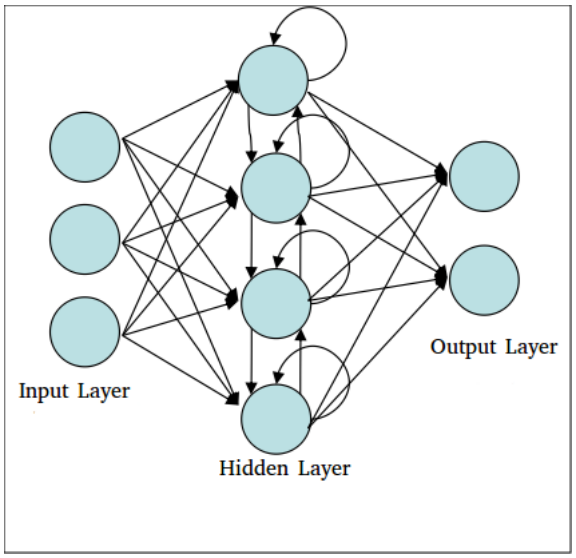

FIGURE 2.11: The basic structure of a recurrent nueral network.

Simple Recurrent Neural Network

A recurrent neural network is structurally very different from a feed forward neural network, while weights only move forward in a feed-forward network, in a recurrent network they move sideways (between neurons in the same layer) as well as backwards, for example from the second hidden layer to the first hidden layer (see Fig. 2.11).

When comparing the multi layer perceptron neural network with the recurrent neural network, if we take the same number of layers for each network and the same number of neurons in each layer, we see that a recurrent neural network contains significantly more weights. More weights means more information can be extracted from the network per a single pass through from our dataset, when compared to multi layer perceptron neural networks. Since these extra weights are "moving sideways" they are *recurring* within the network, hence the name recurrent neural network. The primary advantage of this recurrent property is that the network can now detect changes over time [39].

2.4 Machine Learning and its relationship with Physics

Machine Learning is an important tool for data and function analysis, not just in physics research, but across all science fields. For physics in particular, there are numerous fields of research which are heavily dependent on machine learning techniques. Nearly all experimental physics needs some kind of data processing or function approximation to be done, whether it is linear regression or artificial neural networks machine learning plays an important role in the analysis and understanding of data driven physics.

In this section we take a look at a few interesting intersections of physics and machine learning.

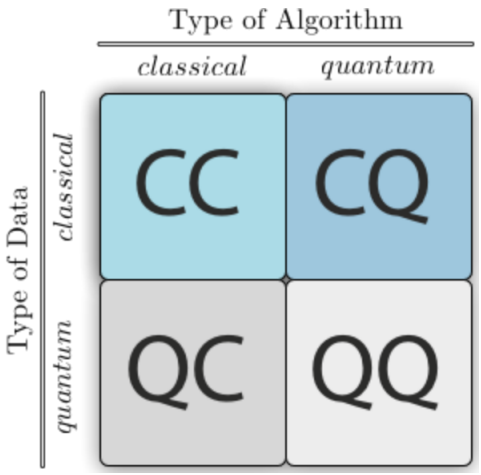

FIGURE 2.12: When considering the interdisciplinary field of "Quantum Machine Learning we can break it down further into four subcategories" [25].

2.4.1 Artificial Neural Networks in Physics

While artificial neural networks have only recently begun to gain popularity, physicists have been implementing artificial neural network models for a very long time. In the early 1990's feed-forward neural networks were used by particle physicists to solve a handful of problems. Feed-forward neural networks were mainly used to solve classification problems, such as for particle identification or separating light quark jets from heavy quark jets [26].

Physicists are not stopping at implementing neural networks to solve physics problems, but are also taking it one step further and are improving neural network implementation. Solid state physicists are using memristors (memory resistor) to develop application-specific integrated circuits called neuromorphic chips [27] with the sole purpose of making neural network training and testing more efficient, both in terms of power consumption and computational time.

2.4.2 Machine Learning and Quantum Physics

In Chapter 1 and up until now in Chapter 2, we have been building up Quantum Information theory and Machine Learning theory independently. In this section we seek to build links between the two disciplines and different ways in which each field can contribute to each other.

Fig. 2.12 presents a well defined overview to the relationship between Machine learning and Quantum physics. Training classical data on a classical system is currently what machine learning is about. However quantum physics can still play a role in this process with quantum inspired machine learning methods [28]. Training quantum data on a classical machine could still somewhat be seen as classical machine learning, but it is still very useful and often times can produce very interesting results.

There is a lot of excitement around the next subcategory which is training classical data on quantum based computational devices. The reason is because if we are able

to successfully implement this, it would revolutionise the way big data is processed and could profoundly change the field of machine learning. The current popular technique used in machine learning is deep learning or more specifically artificial neural networks, which uses matrix or more accurately, tensor multiplication. Calculating the product of extremely large matrices with millions of rows and columns is something that would receive significant speed ups on a quantum computer [29].

Lastly is training quantum data on a quantum based computational device. This is of great interest to physicists and it makes intuitive sense that quantum algorithms would be better suited to understanding a lot of the quantum properties and features of a dataset containing quantum information that we still don't understand completely.

2.4.3 Quantum Neural Network

Quantum neural networks merge the theory of quantum mechanics with neural networks. There are two branches of quantum neural networks. The first branch has to do with trying to find a link between quantum mechanics and a biological neural network. The second branch is to use quantum information theory to improve current artificial neural network models, as well as to develop the quantum information theory to implement artificial neural networks on quantum computers in such a way that the artificial neural network makes use of the advantages allowed by quantum computing. One of the candidates for such an implementation is on a dissipative quantum computer which itself is based on the theory of open quantum systems [30].

One notable recent realisation of a Quantum Neural Network is the Ising model implemented by the National Institute of Informatics and the University of Tokyo. The goal of their implementation is to solve various NP-hard mathematical problems such as the Max-Cut problems. One possible real world application could include helping to ease traffic congestion by finding optimal routes [31].

If a quantum neural network can be realised on a quantum computer it would be of tremendous value to various research fields such as quantum chemistry, molecular biology or astrophysics.

Chapter 3

Data generation, preprocessing and Neural Network Structure

3.1 Data Generation

For this investigation we generated over 3 million entangled two qubit pure state density matrices as well as over 3 million entangled two qubit mixed state density matrices. In this section I will briefly review the mathematics behind the pure state generation and mixed state generation, as well as the random number generation used for constructing mixed states. It is important to note that the generation of the density matrices and calculation of the corresponding concurrence values which were used for this investigation was done by my supervisor Dr. Ilya Sinayskiy. The Python implementation of the following mathematical procedures can be found in appendix A1.

3.1.1 Generating random pure states

The mathematical procedure for numerically generating a two qubit entangled pure state is fairly straightforward. The first step is to generate random 4×4 unitary matrices [36]. To do this we first generate a 4×4 matrix M that contains complex elements from a standard normal distribution ($\mathcal{N}(0, 1)$). We start with a matrix given as

$$M_1 = \begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{pmatrix},$$

and each element x_{ij} is determined randomly from a normal distribution.
Then we define,

$$M_2 = \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix},$$

where again each element is determined randomly from a normal distibution. M is then defined as,

$$M = M_1 + iM_2,$$

so that matrix M is essentially a random complex matrix containing random complex numbers with both real and complex components. After that we perform QR decomposition on matrix M, which gives us

$$M = QR,$$

where M is decomposed into a product of two matrices Q and R with matrix Q being an orthogonal matrix, then matrix R is an upper triangular matrix. We then take the diagonal of matrix R (r_{ij}, where $i = j$) and normalise it,

$$R_D = \begin{pmatrix} \frac{r_{00}}{|r_{00}|} & & & \\ & \frac{r_{11}}{|r_{11}|} & & \\ & & \frac{r_{22}}{|r_{22}|} & \\ & & & \frac{r_{33}}{|r_{33}|} \end{pmatrix}$$

Once we have R_D we then define our random unitary matrix,

$$U = QR_D.$$

To generate the random pure state density operators, we take a column of a random 4×4 unitary matrix U so that,

$$|\psi\rangle = \frac{U}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle).$$

This essentially lets us generate 4 independent complex numbers c_0, c_1, c_2 and c_3 according to the normal distribution. We can then write it as,

$$|\psi\rangle = c_0|00\rangle + c_1|01\rangle + c_2|10\rangle + c_3|11\rangle.$$

Where $|c_0|^2 + |c_1|^2 + |c_2|^2 + |c_3|^2 = 1$. This procedure generates random two qubit pure state composite systems. We then determine the pure state density matrix,

$$\rho_{pure} = |\psi\rangle\langle\psi|$$

The next step is to use equation (1.15) to determine the concurrence of our composite density matrix ρ_{pure}. We discard the density matrix if the concurrence is zero and keep it if the concurrence is greater than zero [32].

3.1.2 Generating random mixed states

The theory and concepts provided in the journal article titled "*On the volume of the set of mixed entangled states*" by K Życzkowski, P Horodecki, A Sanpera, and M Lewenstein (namely Appendix A of the article) was used as a basis for generating the random numbers and thus the random state density matrices [33]. As such we will review that procedure briefly, then show how the two qubit mixed state density matrices were generated.

We determine our two qubit mixed state density matrix by applying the random unitary matrix U on the diagonal matrix D, such that

$$\rho_{mixed} = UDU^{\dagger}.$$

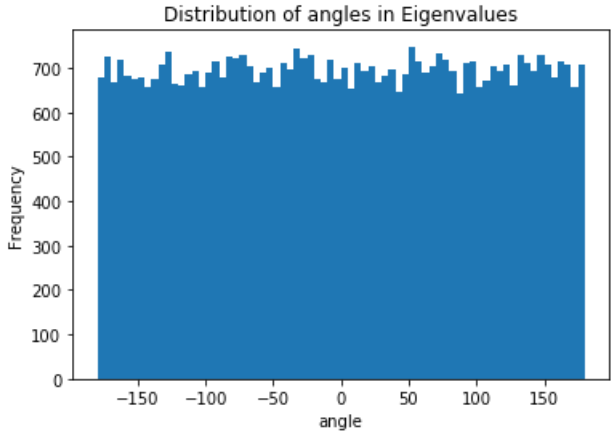

FIGURE 3.1: The distribution of the angle α of the eigenvalues of the unitary matrices. ($\lambda = e^{i\alpha}$)

The random unitary matrix U will be generated in the same procedure described in section 3.1.1, all that needs to be done is to construct the valid diagonal matrix D so that we may begin generating random mixed density matrices ρ_{mixed}. Fig 3.1 shows that the distribution of the angles (α) of the eigenvalues of randomly generated unitary operators with described above algorithm (see section 1.1.1, theory on unitary matrices) [4]. It is clear from the picture that as expected the distribution is uniform.

Random number generation

The aim is to construct a uniform distribution of 4 real, non-zero numbers that satisfy,

$$\Lambda_1 + \Lambda_2 + \Lambda_3 + \Lambda_4 = 1. \tag{3.1}$$

Since we are subject to the constraint given by equation (3.1), we just need to determine Λ_1, Λ_2 and Λ_3 then we can get Λ_4 by manipulating equation (3.1). First we generate ξ_1, ξ_2, ξ_3 and ξ_4 which are 4 random numbers on the normal distribution in the interval (0, 1). We can then define our Λ's as,

$$\Lambda_1 = 1 - \xi_1^{\frac{1}{3}},$$

$$\Lambda_2 = [1 - \xi_2^{\frac{1}{2}}](1 - \Lambda_1),$$

$$\Lambda_3 = [1 - \xi_3](1 - \Lambda_1 - \Lambda_2) \quad \text{and}$$

$$\Lambda_4 = 1 - \Lambda_1 - \Lambda_2 - \Lambda_3.$$

If we sum Λ_1-Λ_4 we see that the sum equals to 1, so our method adheres to the constraint in equation (3.1). We generate the string of numbers in this fashion so that the spectrum of the random density matrices are uniform.

Using the random numbers to generate our mixed states

We now use our 4 random numbers (Λ_1-Λ_4) to create a diagonalised matrix,

$$D = \begin{pmatrix} \Lambda_1 & 0 & 0 & 0 \\ 0 & \Lambda_2 & 0 & 0 \\ 0 & 0 & \Lambda_3 & 0 \\ 0 & 0 & 0 & \Lambda_4 \end{pmatrix}.$$

Finally just like with the pure state density matrices, we apply equation (1.16) to ρ_{mixed} to determine the concurrence of our composite mixed state density matrix, we discard all density matrices where the concurrence is equal to zero and keep them if they are greater than zero.

3.2 Data Preprocessing:

The density matrices for our entangled mixed states and entangled pure states that were generated in python are in the form of,

$$\varrho = \begin{pmatrix} d_1 & x_1 + iy_1 & x_2 + iy_2 & x_3 + iy_3 \\ x_1 - iy_1 & d_2 & x_4 + iy_4 & x_5 + iy_5 \\ x_2 - iy_2 & x_4 - iy_4 & d_3 & x_6 + iy_6 \\ x_3 - iy_3 & x_5 - iy_5 & x_6 - iy_6 & d_4 \end{pmatrix}. \tag{3.2}$$

Considering this form we can see that as expected they are Hermitian matrices. Looking at the data contained within the density matrix, we can see that there are a total of 16 parameters to consider - that's four real numbers on the diagonal (d_1-d_4) and six imaginary numbers on the off-diagonal made up of six real parts (x_1-x_6) and six imaginary parts (y_1-y_6). While we have 16 parameters that we can use as inputs to train our neural network, we can consider the following constraint:

$$Tr(\varrho) = d_1 + d_2 + d_3 + d_4 = 1. \tag{3.3}$$

This means that if we know any three of the diagonal numbers in our density matrix, we can determine the fourth unknown diagonal number by using the constraint shown in equation (3.2), ie one diagonal term can be seen as a linear combination of the other three, which makes it redundant. As such we can omit it from training; having one less input parameter will reduce the time it takes to train our network while not affecting the accuracy our model. Considering our dataset, we will omit d_4 and we will train our neural network on 15 inputs.

3.2.1 Feature Scaling

Feature scaling is a process used to standardize the range of independent variables or features of data. In the case of our dataset inputs range between -1 to 1 and as such feature scaling should not be necessary. However, when investigating the minimum and maximum for each input we found that a lot of features have different ranges, some are between -1 to 1 while others are between 0 to 1. Some even have very small

ranges such as between 0 to 0.4, so we decided to rescale our data. We can rescale our data to any range using the following equation,

$$f' = \frac{f - \min(f)(b - a)}{\max(f) - \min(f)}, \tag{3.4}$$

where f is the value being rescaled, a is the lower bound of our new scale, b is the upper bound and f' is the rescaled value. While our neural network is designed to implement batch normalisation to overcome internal covariate shift, normalising all of our inputs using the process defined in equation (3.3) will help reduce internal covariate shift, thus reduce training time and improve the accuracy of the neural network [34].

In particular, since all of our hidden layers use an activation function with a range from 0 to 1, our inputs that range between -1 to 1 will have to immediately undergo batch normalisation before being processed by the neural network as such; normalising our inputs prior to training will save significant computational time when compared to batch normalisation which will have to normalise inputs each time after data is processed.

3.3 Artificial Neural Network Implementation

In Chapter 2 we discussed general artificial neural network topologies. Now we take a closer look at how we have implemented one of those topologies in our research. For our investigation we will be implementing a multi layer perceptron neural network. This implementation will be done on python using the Keras library which is backended by the TensorFlow library, both of which are powerful tools for developing and implementing artificial neural networks. We will briefly review these two neural network libraries then discuss in detail the structure and key characteristics of the neural network models that we implemented.

3.3.1 TensorFlow and Keras

TensorFlow is an open-source software library used for data driven tasks. It was developed by the Google Brain team for internal use but was made available to the public in November 2015. It is a math library primarily used for machine learning applications, namely neural networks. It is named TensorFlow for the similarities between mathematical tensors and the multi dimensional arrays often used for image recognition models in machine learning. Initially one of the main strengths that TensorFlow had over other neural network libraries was its ability to parallelise neural network training across multiple GPUs and whilst there are now some libraries that can now also do this, none of them do it as efficiently as TensorFlow.

Keras is an open source library developed by François Chollet, a Google engineer. Keras only deals with neural network development and implementation. While TensorFlow is a math library Keras serves as a "container" as such for many deep learning libraries such as TensorFlow, Theano and MXNet, amongst others. The goal of Keras is to be an interface rather than an end-to-end machine-learning framework. It presents a higher-level, simpler, more intuitive set of abstractions, that make it easy to configure neural networks even if the user does not have a strong programming background.

3.3.2 Construction of the implemented artificial neural networks

When constructing a neural network model within the Keras framework, there were various model parameters that we needed to decide on. The choices that we make on these parameters would ultimately affect model accuracy and after analysing our results. An investigation into which neural network parameters were used could provide insight on why our model performed the way it did.

We will briefly discuss the choice in parameters and then take a look at the three different structures that will be used for training. We will also discuss why we chose those structures.

- **Activation Function** - In every single hidden layer neuron, the weight that each of those neurons outputs is "governed" or decided by the activation function that we choose. For our model we chose **ReLU (Rectified Linear Unit)** primarily because ReLU is known to be the best activation function when the output being considered is a regression output [40] as opposed to a classification output. We did try tanh and sigmoid activation functions as well in early models and they both did not perform as well as ReLU.

- **Initialiser** - When inputs are first sent to the hidden layer, the weights of the hidden layer need to be a value to start of with. We have the option of random values, zeroes, ones or we can even initialise each weight to an exact value that we want by giving it an array to initialise from. Ultimately, the the choice of the initialiser does not have a big impact on the model accuracy or the training time, but rather how long it takes for our training to converge to a good model. We chose to initialise our weights to a **standard normal distribution (**$\mathcal{N}(0, 1)$**)**.

- **Loss function** - The loss function (sometimes called error function or cost function) is simply a mathematical function that is dependent on the model's internal learn-able parameters which are used in computing the target values(Y) from the set of inputs(X) used in the model. We chose the **mean squared error** estimator as the loss function for our model. Our neural network is trained with the goal of lowering our loss function to as close to zero as possible, which brings us to our optimiser.

- **Optimiser** - Our optimisation function is integral to the performance of our neural network model. It will affect both the time it takes to train the model and the accuracy of the model. The optimisers job is to minimize the loss function, for our implementation we used the **Adam** optimiser which is a variation of stochastic gradient descent. The reason we chose Adam optimisation is because a lot of recent research has shown that it generally performs the best out of all the optimisers [35].

- **Epochs and Patience** - In machine learning, an epoch is a single pass through the entire training set, generally having a high number of epochs is not advised since "looking at" the same data many times has a tendency of causing over-fitting We set the epochs at an arbitrarily high value (**Epochs = 1000**), since training a single epoch took between 3-10 minutes, this meant that our training could run for as long as 7 days if left unchecked. This brings us to our next parameter, the patience function. Within Keras there is a patience function that allows us to stop our training early if our validation loss stops improving for a number of epochs

(we used **patience = 10 epochs**). So if our model didn't improve after ten epochs in a row, our training would stop and Keras returned the model that had the best accuracy.

- **Training, Validation and Test Split** - Our two datasets had 3.2 million density matrices each. We needed to split each dataset into three parts. A sub-dataset used for training, a sub-dataset used for validation (to prevent over-fitting) and the third sub-dataset was used for testing our model. We went with **Training set = 3 000 000**, **Validation set = 100 000** and **Test set = 100 000**. The process of training yielded predicted concurrence outputs, to determine the quality of our model, the predicted concurrence is compared to the actual concurrence, the difference between actual and predicted is represented by model loss graphs, which demonstrates how the model accuracy changes over time.

3.3.3 One Hidden Layer Neural Network

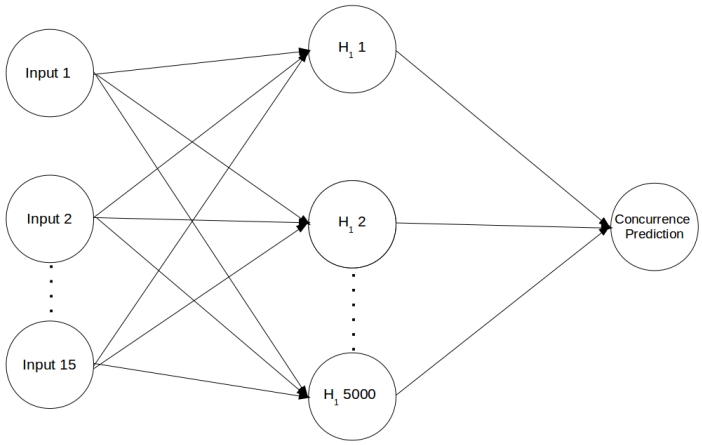

FIGURE 3.2: The one hidden layer network structure.

Our one hidden layer neural network was designed with 5000 neurons in the hidden layer (see Fig. 3.2), as a result the neural network contained 80000 trainable parameters (weights). According to a lot of artificial neural network resources, a one hidden layer network is capable of efficiently learning features and hidden features of non linear data.

Training on one hidden layer neural network

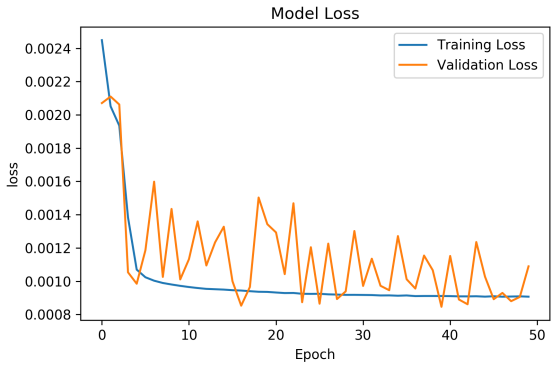

FIGURE 3.3: Loss data for the one hidden layer neural network that was trained on mixed state inputs.

Our one hidden layer neural network trained on mixed state inputs (Fig. 3.3) performed curiously during training when compared to all the other models that were trained. It was the only model where the validation loss was generally much worse than the training loss over 50 epochs. This suggests that the model could be over-fitted, something we will know for sure once we attempt to make predictions on the model. The best model was saved on the 39^{th} epoch with a validation loss of 0.0008466684.

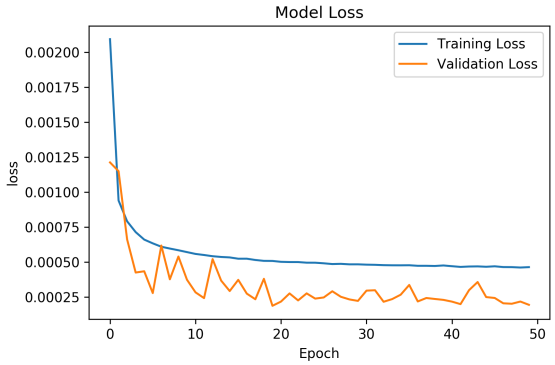

FIGURE 3.4: Loss data for the one hidden layer neural network that was trained on pure state inputs.

The one hidden layer neural network that was trained on pure state data (Fig. 3.4) also behaved curiously, but unlike the model in Fig. 3.3 which shows signs of over-fitting, this model seems to be returning validation loss values lower than training loss values, this is something that technically should not happen so we tried to investigate what could be causing this before proceeding with further training.

It turned out that this was due to the setting we used for our batch size, we used a batch size of 4000 throughout training. Our training set was 3 000 000 density matrices and our neural network processed 4000 at a time. Doing the processing in pieces was

necessary since training was done on a machine with 8GB of RAM. Keras would iteratively update the loss after every batch of 4000 density matrices.

To explain by example, after the first batch in an epoch, the training loss would be 0.002. If the training loss for the next batch is 0.001 then Keras would output our training loss as 0.0015. Basically the average loss is taken over all the batches per a single epoch. When training a single epoch we had 750 batches, the result of which meant that the model always appeared like the validation test set was performing better than the training test set, which is not possible. At the very least we are certain that over-fitting is not an issue given the validation performance. The best model was saved on the 19^{th} epoch with a validation loss of 0.0001862786.

3.3.4 Two Hidden Layer Neural Network

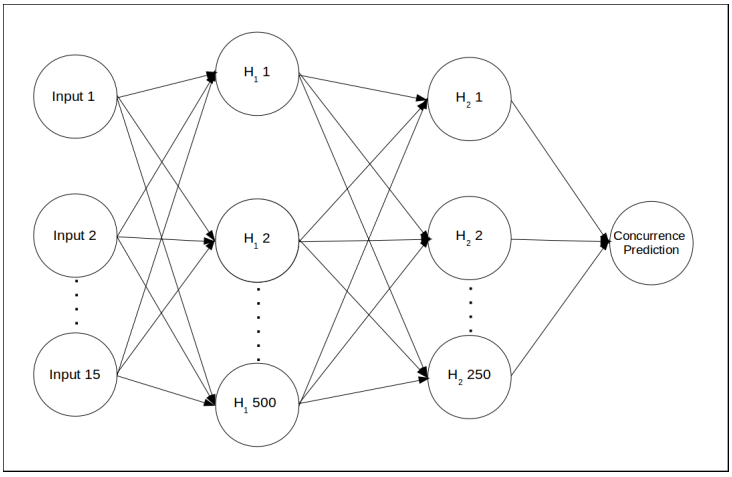

FIGURE 3.5: The two hidden layer network structure.

Our two hidden layers neural network contained 500 neurons in the first hidden layer and 250 neurons in our second hidden layer (see Fig. 3.5). At a glance one may think that this neural network implementation contains less parameters (weights) than our 5 000 neuron one hidden layer neural network, but in actuality this neural network contains 132 750 trainable parameters. This is due to the multiplicative nature of weights between layers. In our first neural network we had a structure of **15-5000-1**, so that is $(15 \times 5000) + (5000 \times 1) = 80000$. For our two hidden layer neural network which has a structure of **15-500-250-1**, this calculation becomes $(15 \times 500) + (500 \times 250) + (250 \times 1) = 132750$. So whilst our two hidden layer network uses 85% less neurons than the one hidden layer neural network, the two hidden layer neural network still contained 66% more parameters to train.

Training on two hidden layer neural network

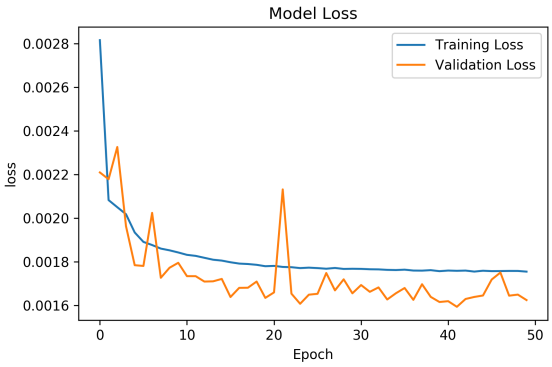

FIGURE 3.6: Loss data for the two hidden layer neural network that was trained on mixed state inputs.

Training mixed state density matrices on our two hidden layer neural network demonstrated our first instance of how the neural network model self corrects over-fitting. If we consider Fig. 3.6, we see that at the 21^{st} epoch, the validation loss spiked. This means that our model was over-fitted. When when the neural network detected the over-fitting, it rolled back the weights to the weights used for epoch 20 and proceeded to apply the cost function again, albeit with a different learning rate. The best model was saved on the 41^{st} epoch with a validation loss of 0.0015933322.

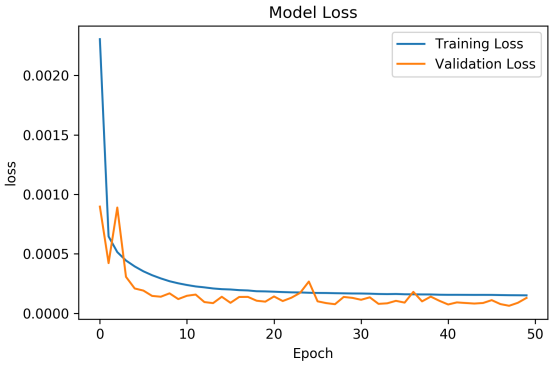

FIGURE 3.7: Loss data for the two hidden layer neural network that was trained on pure state inputs.

The pure state model training for the two hidden layer neural network (Fig. 3.7) followed the same trend as its one hidden layer counterpart shown in Fig. 3.4 The best model was saved on the 47^{th} epoch with a validation loss of 0.0000635506787148.

3.3.5 Three Hidden Layer Neural Network

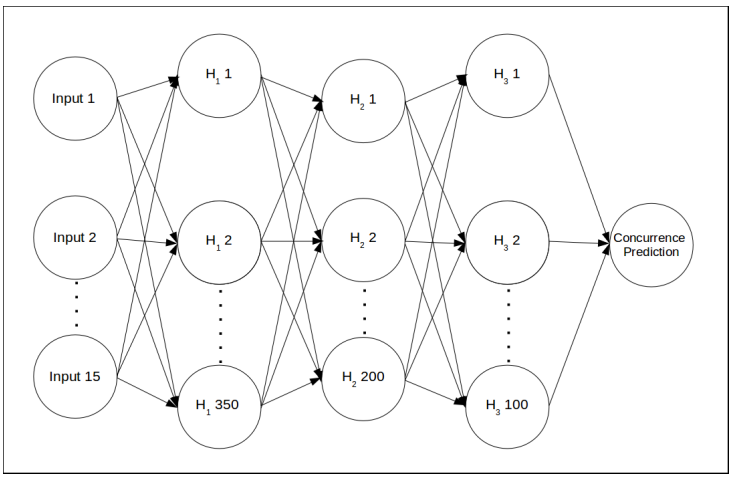

FIGURE 3.8: The three hidden layer network structure.

Finally, our three hidden layer neural network contained 350 neurons in the first hidden layer, 200 neurons in our second hidden layer and 100 neurons in our third hidden layer (see Fig. 3.8). Again due to the multiplicative nature we discussed, our three hidden layer neural network contained 93 850 trainable parameters (weights).

Training on three hidden layer neural network

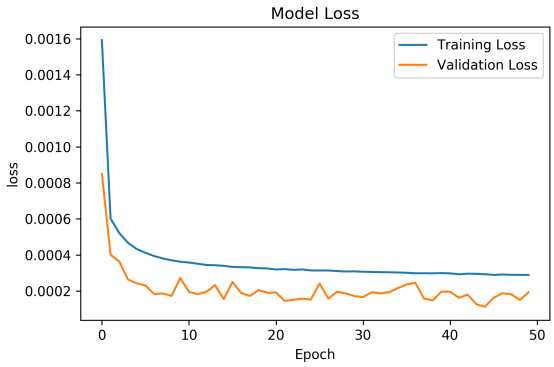

FIGURE 3.9: Loss data for the three hidden layer neural network that was trained on mixed state inputs.

The best model we got when training mixed state inputs on our three hidden layer neural network (see Fig. 3.9) was saved on the 44^{th} epoch with a validation loss of 0.000112616956433. This also happened to be the best validation loss across all mixed state models.

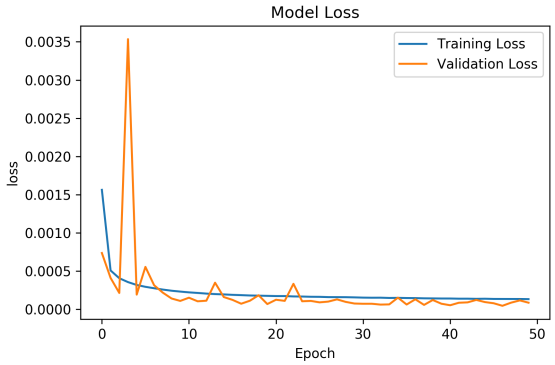

FIGURE 3.10: Loss data for the three hidden layer neural network that was trained on pure state inputs.

The best pure state input trained model on our three hidden layer neural network (see Fig. 3.10) was saved on the 46^{th} epoch with a validation loss of 0.0000451485884696, like the mixed state three hidden layer model, this was our best validation loss, suggesting that a deeper neural network performs better for concurrence prediction using density matrices.

Chapter 4

Concurrence Prediction using Neural Networks

After constructing our neural networks and training them, we're now ready to test our networks and see how they perform on completely unseen data. We have three different network structures that were used and each structure yielded two models, one model trained on mixed states and another model trained on pure states. In total, we have six models that were trained and we will be investigating how our two test datasets perform on each model.

In order to make sense of the predictions that our models will be making, we need to use a few error analysis metrics. The metrics that were used will be briefly described along with the acceptable error range that defines whether the results are good or bad. While the error ranges were arbitrarily chosen at the start, they represent a reasonable accuracy that would allow us to implement the model for use with other applications.

4.1 Error Metrics

Each time after we trained the neural network, we tested each neural network model on two unseen datasets. One dataset contained features from 100 000 entangled mixed state density matrices and the other dataset contained features from 100 000 entangled pure state density matrices. The results of our tests provided us with concurrence predictions. Since we already know the actual concurrencies of the density matrices, we can analyse the predicted error. Various error metrics were used to aid us with analysis. We will briefly discuss how each error metric was implemented.

Root Mean Squared Error and Mean Absolute Error

$$\text{Root Mean Squared Error (RMSE)} = \sqrt{\frac{\sum_{i=1}^{n}(C_i - \hat{C}_i)^2}{n}}$$

Where C_i is the actual concurrence for each entangled density matrix, $\hat{C}_i$ is the predicted concurrence for each entangled density matrix and n is the total number of density matrices in the dataset.

$$\text{Mean Absolute Error (MAE)} = \frac{\sum_{i=1}^{n}|C_i - \hat{C}_i|}{n}$$

Acceptable error range for RMSE: ± 0.02

Acceptable error range for MAE: ± 0.02

Root mean squared error and mean absolute error share a lot of similarities and will often return similar values (With root mean squared error being slightly larger by design) except that since the errors are squared before being summed it gives a larger weighting to large errors. Since large errors are undesirable for our prediction, we used root mean squared error as the loss function for training our neural network, as such to maintain consistency it's the most useful metric to consider when analysing the implementation of our model.

Mean and Standard Deviation of Error distribution

$$C_i^{error} = C_i - \hat{C}_i$$

$$\bar{C}^{error} = \frac{\sum_{i=1}^{n} C_i^{error}}{n}$$

$$\text{Standard Deviation} = \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(C_i^{error} - \bar{C}^{error})^2}$$

Acceptable error range for Mean: ± 0.03

Acceptable error range for Standard Deviation: ± 0.03

The manner in which we calculated this metric means that it is not useful to analyse the numbers on their own, since we do not apply any method of normalising the negative errors. The negative errors will mostly cancel the positive errors and the mean of the error distribution will end up very small and very inaccurate and as a result of that, the standard deviation will be of little use outside of analysing the error distribution. We did however need to determine the values C_i^{error} to plot a histogram distribution of the errors to better investigate the magnitude of the errors across the entire dataset. Since we found that the error distributions often followed a Gaussian distribution, we made note of the standard deviation and mean of the distribution which should come in handy to plot a probability density function of the error distribution.

Average Percentage Error

$$\text{Average Percentage Error} = \frac{\text{Root Mean Squared Error}}{C_{Max} - C_{Min}} \times 100$$

Acceptable error range for Average Percentage Error: $\pm 3\%$

The average percentage error is tied to and is simply an extension of the root mean squared error. We divided the root mean squared error by the range of our values to normalise it and took that as a percentage value which is much more palatable to a wider audience compared to unnormalised error metrics.

4.2 Prediction on a one hidden layer neural network:

4.2.1 Trained on mixed state inputs

ANN Structure: One Hidden Layer (5000)	
Trained on: Mixed State Data	
Metrics for Mixed State prediction	
Root Mean Squared Error:	0.015151927483
Mean Absolute Error:	0.011018506973
Standard Deviation:	0.015052486370
Mean:	0.002036756067
Average Percentage Error:	1.61%

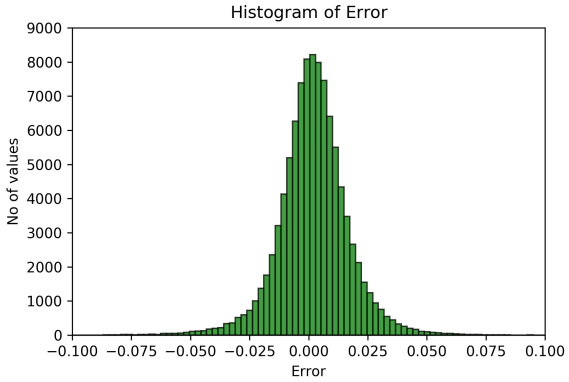

FIGURE 4.1: On the left is a table containing error metrics for the one hidden layer neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled mixed state concurrence on the neural network.

After training the one hidden layer neural network on mixed state data, we used it to predict both mixed state concurrence and pure state concurrence. As per Fig. 4.1 our model predicted mixed state inputs accurately and the error distribution was clearly Gaussian.

ANN Structure: One Hidden Layer (5000)	
Trained on: Mixed State Data	
Metrics for Pure State prediction	
Root Mean Squared Error:	0.027589287702
Mean Absolute Error:	0.021439037538
Standard Deviation:	0.024614865545
Mean:	-0.012461026849
Average Percentage Error:	2.77%

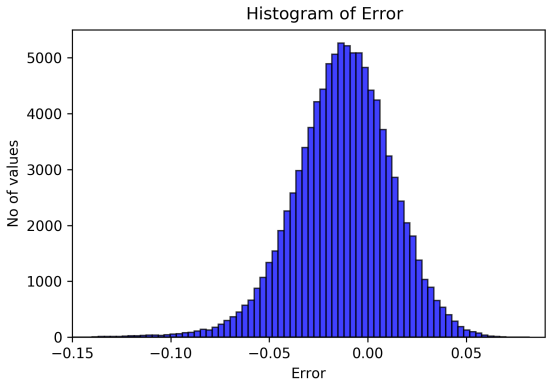

FIGURE 4.2: On the left is a table containing error metrics for the one hidden layer neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled pure state concurrence on the neural network.

Next we used the model to predict pure state inputs, considering our model had been trained on mixed state data, pure states were completely "unknown" to our model and yet it predicted the concurrence of the pure state inputs fairly accurately when compared to the mixed state prediction. When considering Fig. 4.2 however, there are two interesting things to note, one is that the errors showed a bias towards being less than zero, considering how we determined these errors (see section 4.1). This means that more often than not the neural network was predicting a concurrence that was larger

than the actual concurrence. The second notable thing is that the distribution was not as symmetric as the mixed state.

4.2.2 Trained on pure state inputs

ANN Structure: One Hidden Layer (5000)	
Trained on: Pure State Data	
Metrics for Mixed State prediction	
Root Mean Squared Error:	0.202989851031
Mean Absolute Error:	0.189854350886
Standard Deviation:	0.073430073450
Mean:	-0.188853226390
Average Percentage Error:	21.57%

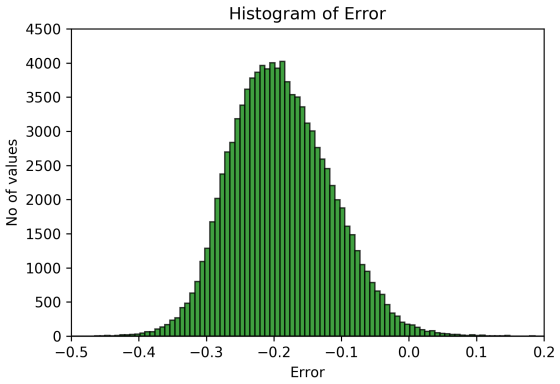

FIGURE 4.3: On the left is a table containing error metrics for the one hidden layer neural network trained on pure state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled mixed state concurrence on the neural network.

We then trained our one hidden layer network using pure state data and the results were even more intriguing. When we used this model to predict mixed state inputs - which is completely unseen to the neural network - it failed miserably. Looking at Fig. 4.3 we see that the errors range between -0.4 to 0.1 while at-least 90% of the errors were less than zero, which means our neural network model was almost always predicting a larger concurrence than the actual concurrence.

ANN Structure: One Hidden Layer (5000)	
Trained on: Pure State Data	
Metrics for Pure State prediction	
Root Mean Squared Error:	0.018207545054
Mean Absolute Error:	0.014497047871
Standard Deviation:	0.018184723737
Mean:	0.000911328426
Average Percentage Error:	1.83%

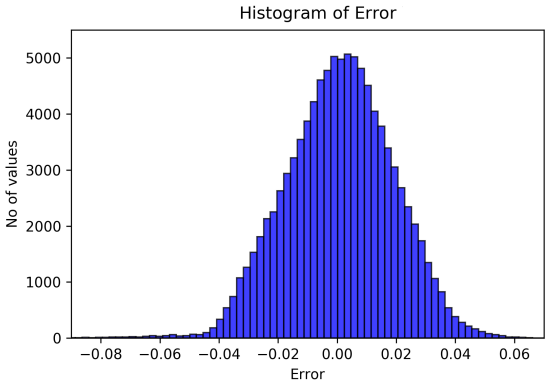

FIGURE 4.4: On the left is a table containing error metrics for the one hidden layer neural network trained on pure state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled pure state concurrence on the neural network.

We then tried to predict pure states on this model and it predicted the pure states accurately. The fact that our model predicted pure states accurately tells us that our model was trained correctly (something that was in doubt after failing to predict mixed states). However, the error distribution shown in Fig. 4.4 was interesting because just like the distribution in Fig. 4.3 and Fig. 4.2 it was not a proper Gaussian distribution.

4.3 Prediction on a two hidden layer neural network:

Moving on to our two hidden layer network, we were expecting marginally better predictions across the board, since in theory an additional hidden layer should help our model learn nuanced hidden features in our data.

4.3.1 Trained on mixed state inputs

ANN Structure: Two Hidden Layers (500-250)	
Trained on: Mixed State Data	
Metrics for Mixed State prediction	
Root Mean Squared Error:	0.016750756649
Mean Absolute Error:	0.012817206394
Standard Deviation:	0.016731170264
Mean:	-0.001328364688
Average Percentage Error:	1.78%

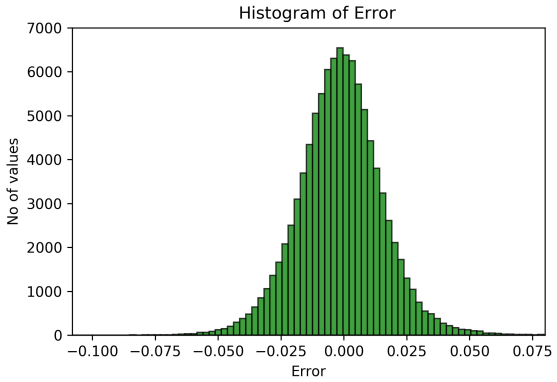

FIGURE 4.5: On the left is a table containing error metrics for the two hidden layers neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled mixed state concurrence on the neural network.

Fig. 4.5 shows exactly what we expect by now, that when predicting mixed state concurrence on our two hidden layer neural network that was trained on mixed state inputs, both the root mean squared and mean absolute error were better when compared to the corresponding result in our one hidden layer neural network and our distribution was clearly Gaussian.

ANN Structure: Two Hidden Layers (500-250)	
Trained on: Mixed State Data	
Metrics for Pure State prediction	
Root Mean Squared Error:	0.029120505657
Mean Absolute Error:	0.022893510259
Standard Deviation:	0.023916125659
Mean:	-0.016613933406
Average Percentage Error:	2.92%

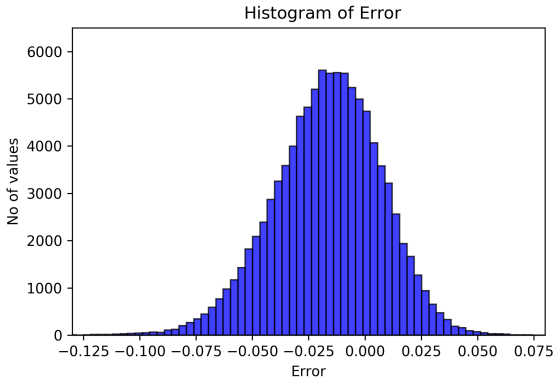

FIGURE 4.6: On the left is a table containing error metrics for the two hidden layers neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled pure state concurrence on the neural network.

Interestingly enough the accuracy on pure state concurrence prediction got slightly worse, furthermore the error distribution in Fig. 4.6 looks less Gaussian to the corresponding one layer error distribution (Fig. 4.2) and just like our one hidden layer distribution, it is not centred around zero but rather has a strong bias for negative errors. So both the one and two layer network trained on mixed state data are mostly predicting concurrence values that are larger than the actual concurrence.

4.3.2 Trained on pure state inputs

ANN Structure: Two Hidden Layers (500-250)	
Trained on: Pure State Data	
Metrics for Mixed State prediction	
Root Mean Squared Error:	0.296349276008
Mean Absolute Error:	0.280717995342
Standard Deviation:	0.095876591403
Mean:	-0.279922490809
Average Percentage Error:	31.49%

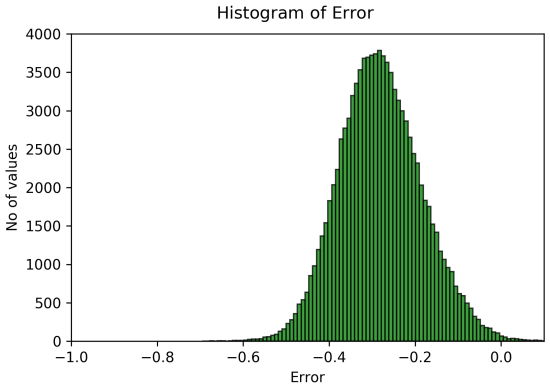

FIGURE 4.7: On the left is a table containing error metrics for the two hidden layers neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled mixed state concurrence on the neural network.

Somewhat surprisingly the two hidden layer neural network trained on pure states still cannot predict mixed state inputs (as per Fig. 4.7), with the errors actually getting worse.

ANN Structure: Two Hidden Layers (500-250)	
Trained on: Pure State Data	
Metrics for Pure State prediction	
Root Mean Squared Error:	0.010910691573
Mean Absolute Error:	0.008677588784
Standard Deviation:	0.009324994427
Mean:	0.005664597915
Average Percentage Error:	1.09%

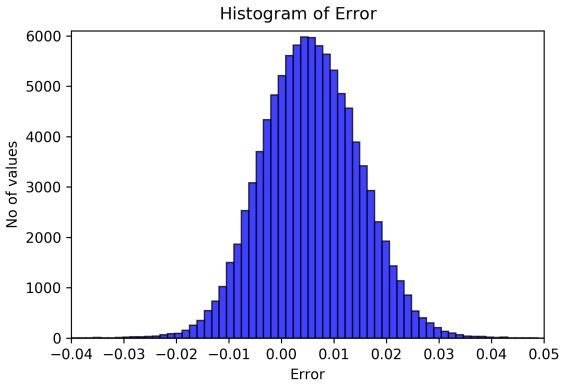

FIGURE 4.8: On the left is a table containing error metrics for the two hidden layers neural network trained on pure state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled pure state concurrence on the neural network.

Predicting pure state concurrence however is not a problem, with our two hidden layer neural network, with predictions being more accurate than its one hidden layer counterpart.

4.4 Prediction on a three hidden layer neural network:

On to our three hidden layer models, we followed the same procedure as with the one layer and two layer models.

4.4.1 Trained on mixed state inputs

ANN Structure: Three Hidden Layers (350-200-100)	
Trained on: Mixed State Data	
Metrics for Mixed State prediction	
Root Mean Squared Error:	0.008995575817
Mean Absolute Error:	0.006822669203
Standard Deviation:	0.014113349681
Mean:	-0.011321898117
Average Percentage Error:	0.96%

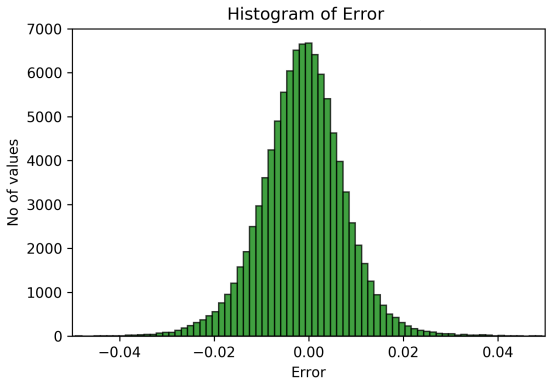

FIGURE 4.9: On the left is a table containing error metrics for the three hidden layers neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled mixed state concurrence on the neural network.

At this point we see a result that we expected before training any models. we observe that when we train our network on a certain type of data (either mixed or pure) and then try to predict the same type of data, it predicts it accurately. This is something that was expected and for mixed states at-least and is clearly confirmed by the similar error distributions seen in Fig. 4.1, Fig. 4.5 and Fig. 4.9.

ANN Structure: Three Hidden Layers (350-200-100)	
Trained on: Mixed State Data	
Metrics for Pure State prediction	
Root Mean Squared Error:	0.018093424667
Mean Absolute Error:	0.014481626894
Standard Deviation:	0.008919356298
Mean:	-0.001205095049
Average Percentage Error:	1.81%

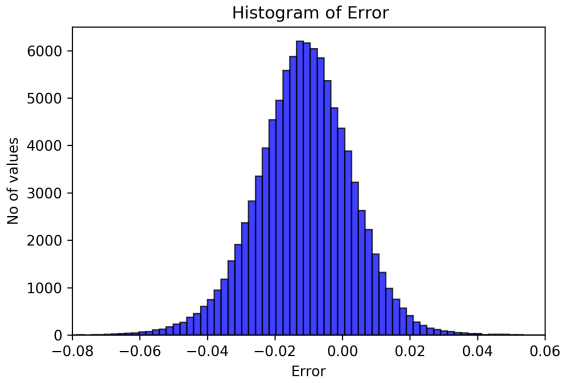

FIGURE 4.10: On the left is a table containing error metrics for the three hidden layers neural network trained on mixed state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled pure state concurrence on the neural network.

Like our first three hidden layer test, this follows the trend we've seen with the corresponding error analysis tests done in Fig. 4.2 and Fig. 4.6. Our neural network that was trained on mixed states (Fig. 4.10), once again accurately predicts pure states and once again has a slight tendency to make concurrence predictions that are larger than

the actual predictions. This is an interesting result that we will discuss further in the next chapter.

4.4.2 Trained on pure state inputs

ANN Structure: Three Hidden Layers (350-200-100)	
Trained on: Pure State Data	
Metrics for Mixed State prediction	
Root Mean Squared Error:	0.540317921589
Mean Absolute Error:	0.512645231367
Standard Deviation:	0.170640226627
Mean:	0.511859017690
Average Percentage Error:	57.41%

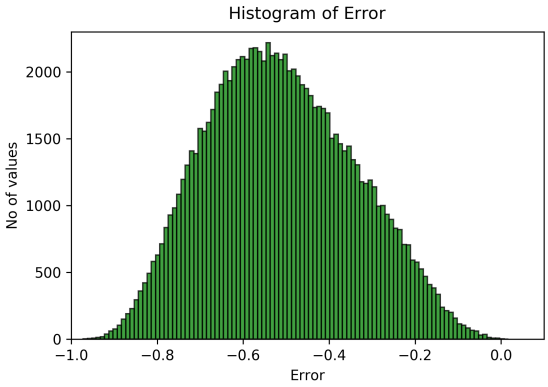

FIGURE 4.11: On the left is a table containing error metrics for the three hidden layers neural network trained on pure state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled mixed state concurrence on the neural network.

Confusingly our three hidden layer neural network trained on pure states predicts mixed state data significantly much worse (see Fig. 4.12) than the corresponding one and two layer neural networks (Fig. 4.4 and Fig. 4.8). In fact if we look at Fig. 4.11 we see that most of our errors are over -0.5. This means that out model was excessively over-predicting the concurrence value. If the three hidden layer neural network made predictions that had an error distribution that was similar or slightly less, that would be a result that's expected, but for the three hidden layer neural network to perform so much worse than the one and two hidden layer network is remarkable and something that we will discuss further.

ANN Structure: Three Hidden Layers (350-200-100)	
Trained on: Pure State Data	
Metrics for Pure State prediction	
Root Mean Squared Error:	0.008395458302
Mean Absolute Error:	0.006545392621
Standard Deviation:	0.007988924549
Mean:	0.002580853471
Average Percentage Error:	0.84%

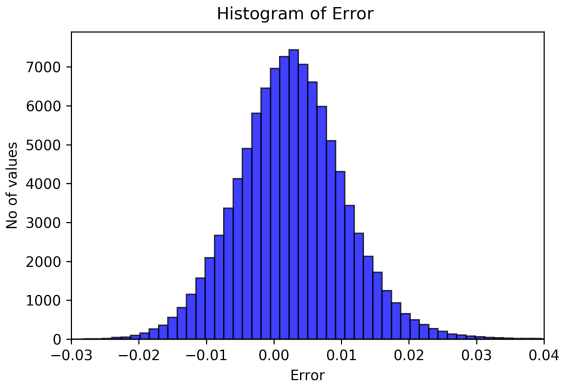

FIGURE 4.12: On the left is a table containing error metrics for the three hidden layers neural network trained on pure state density matrices and on the right the corresponding histogram showing the error of the test dataset when predicting entangled pure state concurrence on the neural network.

Like we have mentioned on Fig. 4.9, our prediction on the neural network that has been trained on the same type of data, is excellent and the results shown in Fig. 4.12

are no different.

Figures 4.1, 4.4, 4.5, 4.8, 4.9 and 4.12 are all error distributions of errors where the data being predicted was the same type as the data used to train the neural network (either pure or mixed) as such these predictions don't tell us anything significant but rather serve as a baseline and measure of accuracy when interpreting the other case - where our models were used to predict a data type that it has not seen before.

Chapter 5

Analysis of Results and Recommendations for further research

5.1 Results Analysis

Some of the results presented in Chapter 4 are very interesting, particularly because there were results that were unexpected. Firstly lets look at the results that were positive. That is when we trained our neural networks using mixed state data and tried to predict pure state data. For all three predictions (see Fig. 4.2, Fig. 4.6 and Fig. 4.10) our model successfully predicted the pure state concurrence values. There was however one interesting caveat - that our error distributions for these cases showed bias towards a negative error, which in our case meant that our model was more often than not predicting concurrence values that were larger than the actual concurrence values of the pure state matrices. This is interesting when we consider the distribution of concurrence values within our datasets.

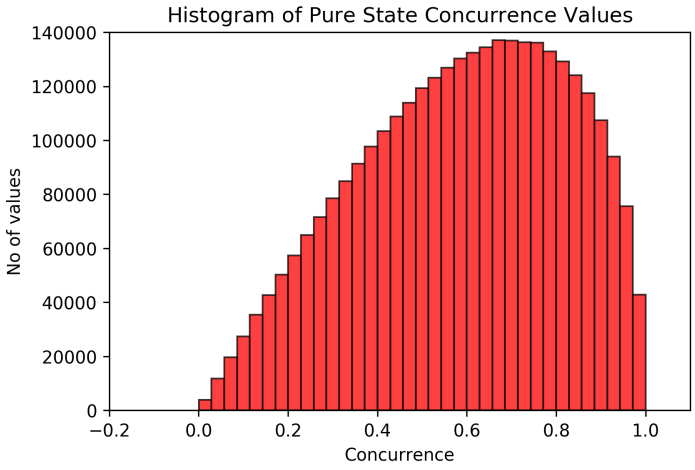

FIGURE 5.1: The distribution of concurrence values across our 3.2 million entangled pure state density matrix dataset.

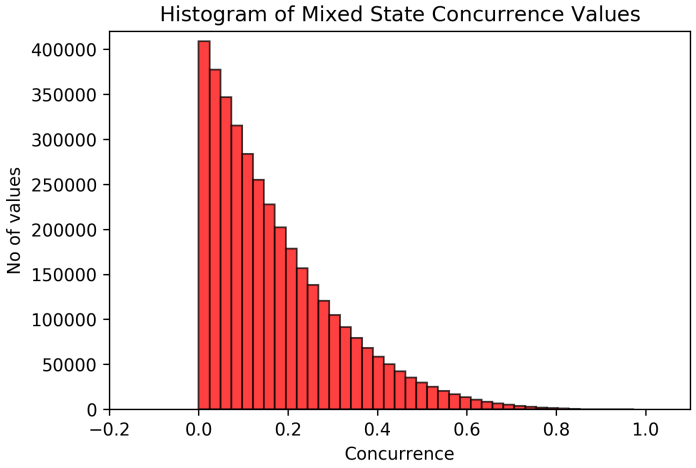

FIGURE 5.2: The distribution of concurrence values across our 3.2 million entangled mixed state density matrix dataset.

When we consider Fig. 5.1 we see that the distribution of concurrence values in our entangled pure state dataset is spread out very well when compared to our entangled mixed state concurrence distribution. In fact looking at Fig. 5.2, it turns out that more than 90% of our entries have a concurrence value of less than 0.4. This is what's interesting. Recall what we discussed earlier; our neural networks trained on mixed state data, was predicting pure state concurrence values that were *larger* than the actual pure state concurrence. If we consider that our mixed state neural network has mostly been trained on concurrence values that are very low, it is counterintuitive that the model is predicting values that are larger. Predicting slightly smaller values would be expected but not slightly larger.

To take this interesting observation further, we would like to analyse the behavior of our best mixed state model. When considering the error metrics, the neural network trained on mixed state inputs that predicted pure state concurrence, the best was our three hidden layer neural network. Considering our model testing dataset, lets split that data set into two subsets; subset A being values with the actual concurrence $\leq .4$ and subset B being values with the actual concurrence > 0.4.

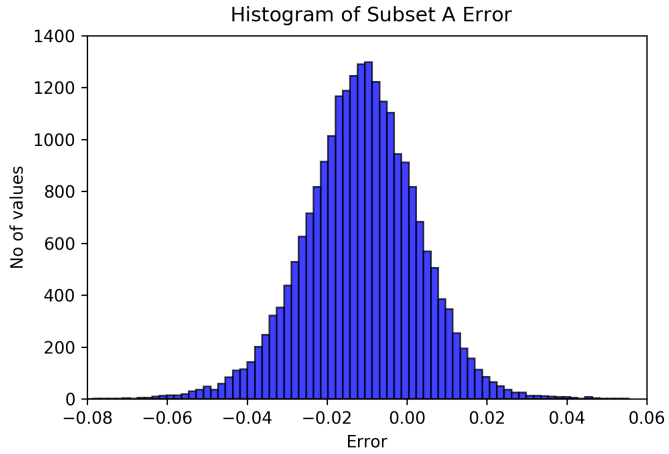

FIGURE 5.3: Subset A contained concurrence values $\leq .4$.

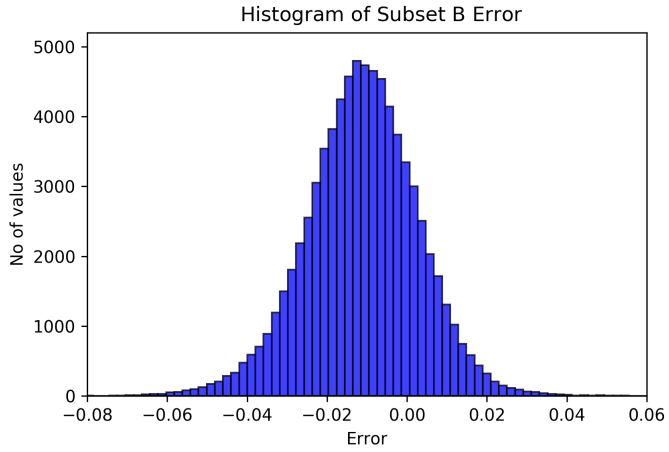

FIGURE 5.4: Subset B contained concurrence values > 0.4.

Once we split our dataset, we then determined the errors for each subset and plotted histograms for each subset (see Fig. 5.3 and Fig. 5.4). Surprisingly both error histograms show the same bias towards predicting a concurrence value that is larger than the actual concurrence. This means that the bias is not a result of our mixed state trained neural networks "learning" concurrence values that were mostly less than 0.4, but rather the bias is an inherent feature of predicting pure state concurrence on a neural network trained on mixed states. We can confirm this by training another mixed state neural network on a larger mixed state data set on a neural network that contains more neurons and then investigate the error variance. If predicting a slightly larger concurrence is an inherent feature of this type of prediction, then that neural network while being slightly more accurate, should show the same bias we have seen across all three mixed state trained neural networks in our investigation.

After everything said about the bias of the pure state predictions, our mixed state

trained neural networks are nonetheless predicting pure state concurrence values exceptionally well and since the bias we are seeing in the error is uniform across all concurrence values, we could introduce a function after prediction to offset or shift that bias and centre our error distribution around zero. It would be interesting to see what function offsets the bias accurately and if the same function works for all our pure state predictions on the different mixed state trained neural networks, then try to understand why that function works.

That brings us to our next interesting result, predicting mixed state concurrence on pure state trained neural networks. Considering Fig. 4.3, Fig. 4.7 and Fig. 4.11. It's safe to say that our pure state trained neural networks do not predict mixed states well at all. Lets apply the same school of thought we did for pure state prediction on mixed state trained neural networks. We know our concurrence distribution for mixed states heavily favours low concurrence density matrices. Whilst the concurrence distribution for pure state density matrices is quiet even, if we look at Fig. 5.1 we see that our distribution does contain more density matrices with higher concurrence values ($>$ 0.4) than lower $\leq$ $.4$. Again lets follow the same procedure as before. The pure state model that predicted mixed states the best was surprisingly the one hidden layer neural network, so we will split the mixed state prediction dataset into two subsets, Subset C ($\leq$ $.4$) and Subset D ($>$ 0.4).

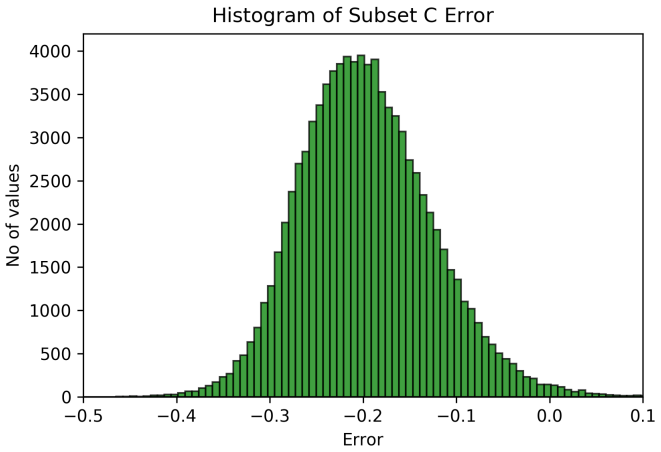

FIGURE 5.5: Subset C contained concurrence values $\leq$ $.4$.

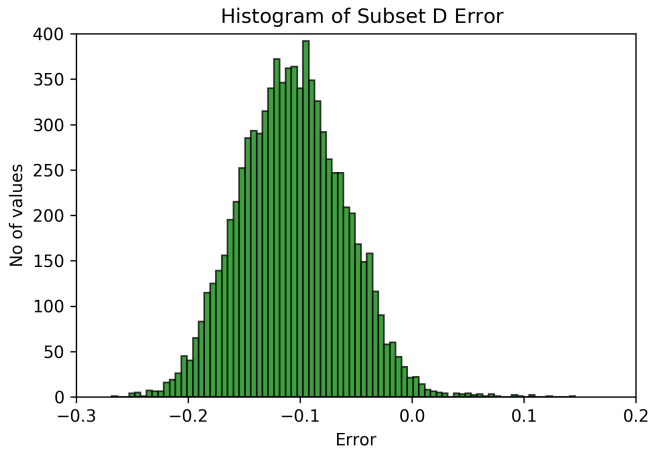

FIGURE 5.6: Subset D contained concurrence values > 0.4.

Again considering Fig. 5.5 and Fig. 5.6, we see the same trend that we saw in Fig. 5.3 and Fig. 5.4 (that the magnitude of the concurrence being predicted has no effect on the accuracy). Both Fig. 5.5 and Fig. 5.6 along with the all the mixed state predictions on pure state trained neural networks (Fig. 4.3, Fig. 4.7 and Fig. 4.11) all exhibit the same traits. The mixed state prediction on pure state trained networks is very bad and the prediction is almost always higher than the actual value. In some cases as seen in Fig. 4.11 the neural network was predicting concurrence values of 0.8 and higher for entangled mixed state density matrices that had an actual concurrence of 0 to 0.2, essentially getting it horribly wrong. In fact when we consider our three hidden layer neural network, the predictions were so bad that we would have got more accurate predictions had we just chosen random values between 0 and 1.

So what could be the reason for mixed state prediction on pure state trained neural networks failing so badly? Well, there are three possible explanations:

The first explanation regards the functions being approximated by our neural networks. For our model that works (predicting pure states on a mixed state trained neural network), we could say that those neural networks are "learning" how to approximate equation (1.16). Equation (1.16) is valid for both mixed states and pure states, and as a result of learning this generalised function, it can approximate the concurrence of pure states. Now, when we consider training neural networks on pure state input, it could be that our neural network is "learning" how to approximate equation (1.15) which is a function that can only calculate the concurrence of pure state inputs. Unfortunately, due to the black box nature of neural networks, there is no easy way to verify whether this is actually what is happening and it is important to note that our pure state concurrence values were determined using the generalised equation (1.16) and not equation (1.15). So in essence our pure state neural network learnt to approximate a function that played no part in how the data was formed.

The second explanation relates to the geometry of the space of density matrices. Mathematically, density matrices belongs to the space of convex bounded hermitian trace class operators acting on the vectors of the Hilbert space ($\rho \in \mathcal{S}(\mathcal{H})$) [37]. In this space

pure states are represented by projection operators. It is known that the set of projectors in $\mathcal{S}(\mathcal{H})$ is of the measure zero. This means that projectors "occupy" an unmeasurably small corner of $\mathcal{S}(\mathcal{H})$ and mixed states are a very typical state in $\mathcal{S}(\mathcal{H})$. In any large enough random section of mixed states ($\mathrm{Tr}(\rho^2) \neq 1$) there will be enough mixed states which are close to all pure states, such that an artificial neural network trained using mixed state inputs could accurately approximate the concurrence of the pure states, however, the converse statement is not true [38]. If this hypothesis is correct then a consequence of this is that mixed states with a low purity may be difficult to predict. We can test this by determining the purity (see section 1.3.4) of all our entangled mixed states and then investigate the prediction performance in relation to the purity. If this explanation is correct then mixed states of a higher purity - in particular mixed states with purity close to 1 should be predicted correctly, while mixed states with a low purity should be predicted incorrectly.

The third explanation which is less likely, is actually a simpler idea. It could just be that we did not have enough data. In early model testing which we did not cover in this investigation, we noticed that as we began to train our network with more and more data - we started on ten thousand density matrices, then one hundred thousand, then one million and then finally three million - as we increased the size of the dataset we noticed a trend, that the accuracy of the predictions would improve significantly as we gave it more training data. This makes intuitive sense considering how neural networks work. So could training a neural network on fifteen million pure state density matrices or perhaps one hundred million pure state density matrices allow us to reduce the error on mixed state prediction enough that it's viable? If it does reduce the error significantly then that would invalidate our first two explanations regarding the approximation of the simpler function.

5.2 Recommendations for further research

Moving forward there are a lot of possibilities for future research both directly related to the work within this paper and more broadly looking at neural network implementation for many other applications within the field of quantum physics.

Firstly considering our research, we have already discussed in this chapter an expansion on predicting pure state concurrence using mixed state trained neural networks, by both increasing the size of the dataset being trained and by introducing a function to offset the bias in prediction. Then for mixed state prediction on pure state trained neural networks, there are a few things we can examine further. Firstly, how much of an impact would training on a larger dataset have, then secondly a nice idea by my co-supervisor Dr. Ilya Sinayskiy was to try and find a way to "force" our pure state trained neural network to learn the more complex function (equation (1.16)) that is capable of generalising mixed states well - note this assumes that indeed our pure state trained neural network does learn how to predict concurrence by approximating the simpler equation (1.15), something we need to prove first.

Beyond that there are different investigations we can do with the same dataset for example an idea posed by Dr. Maria Schuld was to omit inputs from our training data (look at equation (3.2) and (3.3)) and investigate if our training data still contained enough information for concurrence prediction. We could omit some of the inputs that

are not on the diagonal (x_1-x_6) and then also maybe omit some or all of our diagonal elements (d_1-d_4) or a combination of those and then see how well we are able to predict concurrence.

Then, not considering density matrices and entanglement measures, there is still an endless amount of novel implementations of neural networks that we can investigate in quantum information science.

Appendix A - Python Code

A1 - Data Generation Code

The Python code below was used to generate our two qubit entangled mixed states and two qubit entangled pure states. After that it determined the concurrence of the corresponding entangled states, then save the density matrix elements and the concurrence in a csv file.

```
import numpy as np
import pandas as pd
from numpy import linalg as LA

#define a set functions to generate random state vectors and density matrices

def H(matr):
    return matr.conj().T

def randU(n):
    X = (complex(1,0)*np.random.randn(n,n)+complex(0,1)*np.random.randn(n,n))
    /np.sqrt(2)
    q, r = LA.qr(X)
    r = r.diagonal()
    r = np.diag(r/np.absolute(r))
    res = np.dot(q,r)
    return res

def vecnorm(x):
    return np.sqrt(x.dot(x.conj()))

def randP(n):
    p = np.ones(n)
    p = np.dot(randU(n),p)
    p = p/vecnorm(p)
    return p

def toDM(x):
    return np.outer(x,x.conj())

#function to generate random string of 4 numbers
def rand4st():
    ks = np.random.rand(3)
    res = np.zeros(4)
    res[0] =  1-(ks[0])**(1.0/3.0)
    res[1] = (1-(ks[1])**(1.0/2.0))*(1-res[0])
    res[2] = (1- ks[2])*(1-res[0]-res[1])
    res[3] = 1 - res[0] - res[1] - res[2]
    return res

# function to create random 4*4 density matrix
def randDMn():
    p = np.diag(rand4st())
    U = randU(4)
    p = np.dot(U,p)
```

```
    p = np.dot(p,H(U))
    return p

#function to determine concurrence
def Conc1(x):
    r = x.conj()
    sy = np.array([[0,complex(0,-1)],[complex(0,1),0]])
    sy2 = np.kron(sy,sy)
    r = np.dot(sy2,r)
    r = np.dot(r,sy2)
    r = np.dot(x,r)
    EV = LA.eigvals(r).real
    EV[::-1].sort()
    EV = np.sqrt(np.abs(EV))
    return np.maximum(0,EV[0]-EV[1]-EV[2]-EV[3])

# function to create random 4*4 density matrix
def randDMnBELL():
    bell1 = np.array([[0.5,0,0,0.5],[0,0,0,0],[0,0,0,0],[0.5,0,0,0.5]])
    bell2 = np.array([[0.5,0,0,-0.5],[0,0,0,0],[0,0,0,0],[-0.5,0,0,0.5]])
    bell3 = np.array([[0,0,0,0],[0,0.5,0.5,0],[0,0.5,0.5,0],[0,0,0,0]])
    bell4 = np.array([[0,0,0,0],[0,0.5,-0.5,0],[0,-0.5,0.5,0],[0,0,0,0]])
    L = rand4st()
    p = L[0]*bell1+L[1]*bell2+L[2]*bell3+L[3]*bell4
    U = randU(4)
    p = np.dot(U,p)
    p = np.dot(p,H(U))
    return p

# function to transform DM 4*4 to list 15 parameters
def fromDMtoARpCC(x):
    res = []
    res.append(x[0,0].real)
    res.append(x[1,1].real)
    res.append(x[2,2].real)
    res.append(x[0,1].real)
    res.append(x[0,1].imag)
    res.append(x[0,2].real)
    res.append(x[0,2].imag)
    res.append(x[0,3].real)
    res.append(x[0,3].imag)
    res.append(x[1,2].real)
    res.append(x[1,2].imag)
    res.append(x[1,3].real)
    res.append(x[1,3].imag)
    res.append(x[2,3].real)
    res.append(x[2,3].imag)
    res.append(Conc1(x))
    return np.array(res)

def ClassC(x, bins=None):
    if bins is None:
        bins = 20
    #-------
    p = Conc1(x)
    if p==0:
        res = 0
    elif int(p*bins)<p*bins:
        res = int(p*bins)+1
    else:
        res = int(p*bins)
```

```
    return res

# generate random projector (|a><a|)
def randSW(n):
    return toDM(randP(n))

collist = ['f01', 'f02', 'f03','f04', 'f05', 'f06','f07', 'f08', 'f09','f10',
     'f11', 'f12','f13', 'f14', 'f15','conc']
train = pd.DataFrame(columns=collist,dtype='float')
NTries = 100000


print('Data generation started ... ')

DBcounter = 0
for i in range(NTries):

    #for pure state
    p = randSW(4)

    #for mixed state
    #p = randDMn()


    c = ClassC(p)
    if c!=0:
        train.loc[DBcounter] = fromDMtoARpCC(p)
        DBcounter=DBcounter+1
    if i % 5000 == 0:
        print(' -> processed ',i,' out of ', NTries)

print('total # of records added',DBcounter)
print(train.info())
train.to_csv("train_concp3m.csv")
```

A2 - Model Training Code

The Python code below was used to import our datasets containing the pure and mixed state density matrices, then create and train the model to predict concurrence.

```
"""
@author: lishen
"""

import pandas as pd
import numpy as np

# Read training dataset into X and Y
dataframe = pd.read_csv("train_conc_p3m.csv")
dataset = dataframe.values
X = dataset[:,1:16].astype(float)
Y = dataset[:,17].astype(float)

#read validation dataset into X_valid and Y_valid
valid = pd.read_csv("validation_mixed.csv")
valid1 = valid.values
X_valid = valid1[:,1:16].astype(float)
Y_valid = valid1[:,17].astype(float)


# Define the neural network
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization
from keras.optimizers import SGD, adam
from keras.callbacks import EarlyStopping, ModelCheckpoint, History, Callback
from keras.models import model_from_json


def build_nn():
    model = Sequential()
    model.add(Dense(15, input_dim=15, init='normal', activation='relu'))

    # 3 layer model
    model.add(BatchNormalization())
    model.add(Dense(350, init='normal', activation='relu'))
    model.add(BatchNormalization())
    model.add(Dense(200, init='normal', activation='relu'))
    model.add(BatchNormalization())
    model.add(Dense(100, init='normal', activation='relu'))

     # 2 layer model
#    model.add(BatchNormalization())
#    model.add(Dense(500, init='normal', activation='relu'))
#    model.add(BatchNormalization())
#    model.add(Dense(250, init='normal', activation='relu'))

    # 1 layer model
#    model.add(BatchNormalization())
#    model.add(Dense(5000, init='normal', activation='relu'))

# No activation needed in output layer (because regression)
    model.add(Dense(1, init='normal'))

# Additional parameters for tweaking if using stochastic gradient descent (
    sgd)
#    epochs = 100
```

```
#     learning_rate = 0.1
#     decay_rate = learning_rate / epochs
#     momentum = 0.8
#     sgd = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate,
    nesterov=False)
    # Compile Model
    model.compile(loss='mean_squared_error', optimizer='adam')
    print(model.summary())

    #save model structure to json file
    model_json = model.to_json()
    with open("model_structure350-200-100.json", "w") as json_file:
        json_file.write(model_json)
    # save model weights to HDF5 file
    print("Saved model to disk")
    return model


model_path = 'model_weights350-200-100_mixed.h5'

# prepare callbacks
callbacks = [
    EarlyStopping(
        monitor='val_loss',
        patience=10,
        verbose=1),

    ModelCheckpoint(
        model_path,
#         monitor='val_loss',
        monitor='loss',
        save_best_only=True,
        verbose=0)]

# Evaluate model
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.metrics import  mean_squared_error


#define evaluation parameters

estimator = KerasRegressor(
    build_fn=build_nn,
    epochs=500,
    batch_size=4000,
    verbose=0
)

#seed starting point to ensure reproduciblilty of results
seed = 7
np.random.seed(seed)


history = estimator.fit(
    X,
    Y,
    epochs = 500, # increase it to 20-100 to get better results
    validation_data=(X_valid, Y_valid),
    verbose=1,
    callbacks=callbacks,
    shuffle=True
```

```
)


#save loss and validation loss data of each epoch
pd.DataFrame(history.history).to_csv("history350-200-100Nmixed.csv")


print(estimator.predict(X_valid))
print()
print('MSE train: {}'.format(mean_squared_error(Y, estimator.predict(X)))) #
    mse train

# check performance on validation set
print('MSE val: {}'.format(mean_squared_error(Y_valid, estimator.predict(
    X_valid))))
```

A3 - Model Testing and Error Analysis

The Python code below was used to import the trained models, test the models, determine error metrics and save the results of predictions.

```
"""
@author: lishen
"""
import pandas as pd
import numpy as np
from keras.models import model_from_json
from sklearn.metrics import mean_squared_error, mean_absolute_error


# Read pure state test dataset into X and Y
dataframe = pd.read_csv("train_conc_p.csv")
dataset = dataframe.values
X = dataset[:,1:16].astype(float)
Y = dataset[:,17].astype(float)


# Read mixed state test dataset into X1 and Y1
dataframe1 = pd.read_csv("train_conc_un.csv")
dataset1 = dataframe1.values
X1 = dataset1[:,1:16].astype(float)
Y1 = dataset1[:,17].astype(float)

#define min and max variables for calculating average percentage error
maxp = np.amax(Y)
maxm = np.amax(Y1)
minp = np.amin(Y)
minm = np.amin(Y1)

json_file = open( 'model_structure350-200-100.json' , 'r' )

loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)


# load weights into loaded model
loaded_model.load_weights("model_weights350-200-100_mixed.h5")


print("Loaded model from disk")


# evaluate loaded model on test data
loaded_model.compile(loss = 'mean_squared_error', optimizer= 'adam')
score = loaded_model.evaluate(X, Y, verbose=1)
score1 = loaded_model.evaluate(X1, Y1, verbose=1)

#create prediction arrays for computing prediction error metrics
predictp =  loaded_model.predict(X)
predictm =  loaded_model.predict(X1)

#print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score1[1]*100))
print
print('================')
print('Pure State Stats')
```

```
print
print('MSE test: {}'.format(mean_squared_error(Y, predictp)))
print('RMSE test: {}'.format(mean_squared_error(Y, predictp)**0.5))
print('MAE test: {}'.format(mean_absolute_error(Y, predictp)))
print
errorp = (mean_squared_error(Y, predictp)**0.5/(maxp - minp))*100
errorp1 = np.around(errorp, decimals = 2)
print"Percentage error: %s %%\n" % errorp1
print
print('=================')

print('Mixed State Stats')
print
print('MSE test: {}'.format(mean_squared_error(Y1, predictm)))
print('RMSE test: {}'.format(mean_squared_error(Y1, predictm)**0.5))
print('MAE test: {}'.format(mean_absolute_error(Y1, predictm)))


print
errorm = (mean_squared_error(Y1, predictm)**0.5/(maxm - minm))*100
errorm1 = np.around(errorm, decimals = 2)
print"Percentage error: %s %%\n" % errorm1
print('=================')

#save actual and predicted concurrence data to csv file for possible further
    analysis (such as graphs)
np.savetxt("y_comp_350-200-100_mixed.csv", zip(Y, predictp,Y1,predictm),
    delimiter=",")
```

References

[1] W Wootters. “Entanglement of Formation of an Arbitrary State of Two Qubits”. In: Physical Review Letters 80.10, (Mar. 1998), pp. 2245–2248.

[2] G Strang. "Introduction to Linear Algebra, 5th edition". Wellesley-Cambridge Press, (2016).

[3] M Nielsen, I Chuang. "Quantum Computation and Quantum Information 10th anniversary edition". Cambridge University Press, (2016).

[4] S Hassani. "Foundations of mathematical physics". Allyn and Bacon, (1991).

[5] J Bub. “Quantum Entanglement and Information”. In: The Stanford Encyclopedia of Philosophy, (2017).

[6] A Einstein , B Podolsky, N Rosen. "Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?". In: Phys. Rev. 47, 777, (1935).

[7] J Bell "On the Einstein-Poldolsky-Rosen paradox" In: Physics Vol 1, (1964).

[8] R Horodecki, P Horodecki, M Horodecki, K Horodecki, "Quantum entanglement". Rev.Mod.Phys.81, (2009), pp. 865-942.

[9] A Peres. “Separability Criterion for Density Matrices”. In: Phys. Rev. Lett. 77, (Aug. 1996), pp. 1413–1419.

[10] D Janzing. "Entropy of Entanglement". In: Greenberger, (2009), pp. 205–209.

[11] A Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: IBM Journal of Research and Development 3 (3), (1959).

[12] B Marr. "A Short History of Machine Learning - Every Manager Should Read". Forbes. Retrieved 28 Sep 2016.

[13] R Cuingnet. "Spatial regularization of SVM for the detection of diffusion alterations associated with stroke outcome". In: Medical Image Analysis 15 (5), (2011), pp. 729–737.

[14] http://dogsofthedow.com/largest-companies-by-market-cap.htm.

[15] T Hastie, R Tibshirani, J Friedman, "The Elements of Statistical Learning (2nd ed.)" Springer (2008), pp. 587–588.

[16] A Coates, H Lee, A Ng. "An analysis of single-layer networks in unsupervised feature learning". In: Int'l Conf. on AI and Statistics, (2011).

[17] C Charliepaul, G Gnanadurai. "Comparison of k-mean algorithm and apriori algorithm–An analysis". In International Journal On Engineering Technology and Sciences, (July 2014), pp. 2349-3968.

[18] https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/.

[19] https://commons.wikimedia.org/wiki/File:Neuron.svg

[20] W McCulloch, W Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: Bulletin of Mathematical Biophysics. 5 (4), (1943), pp. 115–133.

[21] F Rosenblatt. "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain". In: Psychological Review 65 (6), (1958), pp. 386–408.

[22] M Minsky, S Papert. "Perceptrons: An Introduction to Computational Geometry" (The MIT Press), (1969)

[23] Y LeCun, L Bottou, Y Bengio, P Haffner. "Gradient Based Learning Applied to Document Recognition" In: Proceedings of IEEE 86(11), (1998), pp. 2278–2324.

[24] https://medium.com/machine-learning-world/how-to-debug-neural-networks-manual-dc2a200f10f2

[25] M Schuld, https://commons.wikimedia.org/wiki/File:Qml_approaches.tif.

[26] H Kolanoski. "Applications of artificial neural networks in particle physics". In: Nuclear Instruments and Methods in Physics Research A 367, (1995), pp. 14-20.

[27] D Monroe."Neuromorphic computing gets ready for the big time". In: Communications of the ACM. 57, (2014), pp. 13–15.

[28] E Stoudenmire, D Schwab. "Supervised Learning with Quantum-Inspired Tensor Networks". In: Advances in Neural Information Processing Systems 29, 4799, (2016).

[29] X Zhang, XM Zhang, ZY Xue. "Quantum hyperparallel algorithm for matrix multiplication". In: Nature Publishing Group, (April 2016).

[30] M Schuld, I Sinayskiy, F Petruccione. "The quest for a Quantum Neural Network". In: Quantum Information Processing 13, (2014).

[31] K Zyczkowski. "On the volume of the set of mixed entangled states". In: Phys.Rev. A58, (1998).

[32] S Ioffe, C Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: arXiv:1502.03167, (Feb 2015).

[33] S Ruder. "An overview of gradient descent optimization algorithms". In: arXiv:1609.04747, (Sep 2016).

[34] K Zyczkowski. “Generating random density matrices”. In: J. Math. Phys. 52, (2011).

[35] https://qnncloud.com/document/WhitePaper-En.

[36] E Prugovečki. "Quantum mechanics in Hilbert space, 2nd edition". Academic Press, (1981).

[37] F Mezzadri. “Generating random density matrices”. In: Notices of the AMS, Vol. 54 (2007), pp. 592-604.

[38] J von Neumann. "Mathematical Foundations of Quantum Mechanics, 1st edition". Princeton University Press, (1955).

[39] M Karim, S Rivera. "Comparison of feed-forward and recurrent neural networks for bioprocess state estimation". Computers and Chemical Engineering, Volume 16, Supplement 1, (1992), pp. 369-377.

[40] I Goodfellow. "Deep Learning (Adaptive Computation and Machine Learning series)". The MIT Press, (2016)

[41] J Clauser, M Horne, A. Shimony, R.A. Holt. "Proposed experiment to test local hidden-variable theories", Phys. Rev. Lett. (1969)

[42] B Hensen, H Bernien. "Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres", Nature volume 526, (2015), pp. 682–686