LONDON'S GLOBAL UNIVERSITY

Representing quantum states with generative neural networks

Justin Jude[1]

MEng Mathematical Computation

Supervisor: Simone Severini

Submission date: 30th April 2018

[1]**Disclaimer:** This report is submitted as part requirement for the MEng Degree in Mathematical Computation at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This project aims to test the difference in effectiveness in various neural network architectures when utilised in the learning of large and complex quantum states. Namely the project explores the use of variational autoencoders of contrasting depths and various restricted Boltzmann machines.

Quantum systems are notoriously difficult to store and manipulate due to the fact that the number of parameters required to describe them increases exponentially with the number of qubits. Expressing quantum states with classical methods becomes intractable when the number of qubits is large.

The contribution of this project would be to show concretely that learning approximate representations of large quantum states using neural networks (especially variational autoencoders) can be effective and accurate.

Contents

1 Quantum Systems **5**
1.1 Background . . . 5
1.1.1 Why this is important to us . . . 5
1.2 State Distributions . . . 6
1.2.1 Forming Samples . . . 8
1.3 Quantum Gates and Change of Basis . . . 9

2 Neural Networks **11**
2.1 Background . . . 11
2.2 Autoencoders . . . 12
2.3 Variational Autoencoders . . . 13
2.4 Restricted Boltzmann Machines . . . 15
2.4.1 Deep Restricted Boltzmann Machines . . . 17
2.4.2 Complex valued Restricted Boltzmann Machines . . . 18
2.5 TensorFlow . . . 19
2.6 Summary . . . 20

3 Objectives and previous work **21**
3.1 Takeaway from Rochetto et al. . . . 21
3.2 Other Work . . . 21
3.3 Aims of the project . . . 22

4 RBMs vs VAEs **24**
4.1 System Set up . . . 24
4.2 Experiments and results . . . 26
4.3 Analysis . . . 27

5 Multiple Bases **31**
5.1 System Set up . . . 33
5.2 Experiments and results . . . 34

5.3 Analysis . . . 35
5.3.1 Further experimentation . . . 35

6 Conclusions **37**
6.1 Achievements . . . 37
6.2 Evaluation . . . 38
6.3 Future Work . . . 38

A Project Plan **42**

B Interim Report **44**

C Code Listing **47**

List of Figures

1.1 Product State Distribution . . . 6
1.2 Hard State Distribution . . . 7
1.3 Random State Distribution . . . 7
1.4 Measurement distributions of X and Z bases of a product state . . . 10

2.1 Multilayer Perceptron . . . 12
2.2 Autoencoder . . . 13
2.3 Variational Autoencoder . . . 15
2.4 Restricted Boltzmann Machine . . . 17
2.5 Deep Restricted Boltzmann Machine . . . 18
2.6 Complex valued Restricted Boltzmann Machine . . . 19

4.1 Product state benchmark . . . 26
4.2 Hard state reconstruction overlap as a function of VAE Depth . . . 28
4.3 Final reconstruction of VAE learned states after training. . . . 29
4.4 Final reconstruction of (standard) RBM learned states after training. . . . 29
4.5 Final reconstruction of Deep RBM learned states after training. . . . 29

5.1 Adapted Variational Autoencoder (encoder switch) . . . 32
5.2 Adapted Variational Autoencoder (latent space switch) . . . 32
5.3 X and Z basis baselines . . . 34
5.4 X and Z basis reconstruction . . . 34

List of Tables

4.1 VAE Training configurations 25
4.2 VAE and RBM reconstruction overlaps 27

5.1 VAE Training configurations 33
5.2 VAE X and Z basis reconstruction overlaps 35

Chapter 1

Quantum Systems

Here we outline what we mean by a quantum state and why physicists and computer scientists alike may need to manipulate them in the present and in the near future. We also emphasise the rationale behind the difficulty in storing them.

1.1 Background

Quantum states contain all the information about a given quantum system. A Quantum state distribution is essentially a probability distribution, gained from the quantum state, representing likelihoods for all possible outcomes when the system is measured. We produce the state distribution by taking the amplitude squared of the state and normalizing. This state distribution will be the subject of our learning task. These distributions are represented as measurements of the state on a particular basis. When we measure a specific parameter of the state, this gives us the probability that parameter. This can be more accurately called a measurement distribution on a specific basis, and we can generate distinct measurement distributions from the same quantum state by measuring the probability of the parameters on a different basis (as we do in chapter 5). In order to measure on another basis, we just need to apply a linear transform to the state. The issue is that the number of parameters required to represent these states (and thus the distribution) can be considerably large. Quantum state tomography (QST) is the method by which we aim to reconstruct the full state from our compressed representation.

1.1.1 Why this is important to us

Quantum probability distributions require an exponential number of parameters to learn as a function of the number of qubits in the system, although various methods do exist for storing certain classes of quantum distribution which reduce this overhead somewhat. It is for this reason that we aim to compress the state distributions using neural network.

We use the fact that entanglement causes the distributions of some quantum systems to have a more learnable pattern than others. While entanglement is not by itself a sufficient condition to make a quantum state difficult to represent (even approximately), hard to represent states are usually hard due to being entangled. We task a neural network such as a Variational Autoencoder (VAE) with learning the parameters required to reconstruct such distributions once trained.

This would be intractable using conventional algorithms in computer science and physics. An example of such an algorithm is Matrix-Product-States (MPS) [9] which is the current state of the art for states which are characterized by being lowly-entangled.

1.2 State Distributions

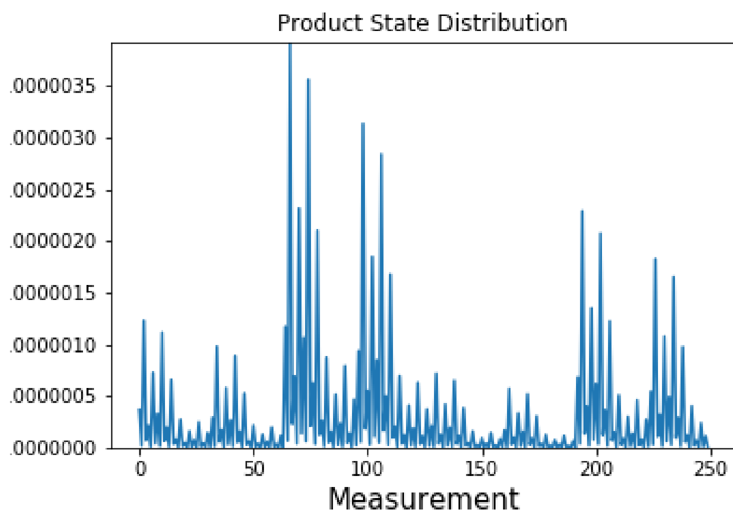


Figure 1.1: An example of a Product State Distribution. The x-axis here (and henceforth) represents the measurement of the corresponding parameter of the system (here we have restricted the plot to showing just the measurements of the first 250 parameters) and the y-axis gives their respective probability. Product states are relatively easy for a neural network to learn as product states are maximally factorisable and so we only need twice the number of qubits of complex numbers to represent them.

In this project, we aim to test the learnability of 3 quantum probability distributions. Using QuTiP [4] which is a Quantum toolbox written in the Python programming language, we generate a simulated quantum distribution using the command *qutip.rand_ket()*. To produce a simulated product state, we recursively produce a random ket vector for each qubit in the system and apply the tensor product between the current state and the next ket vector, and this tensor product then becomes the state for the next recursive call.

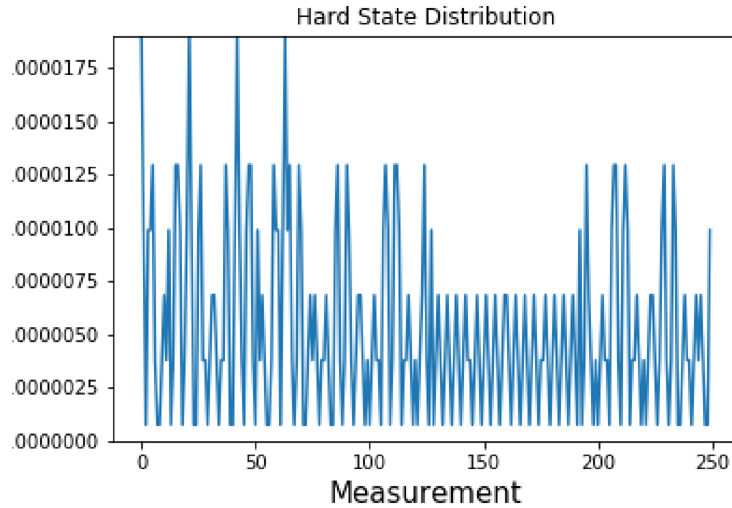


Figure 1.2: An example of a Hard State Distribution. Hard states are relatively difficult to learn due to the fact that they are highly entangled. Therefore as previously stated, we have less structure that a model can use as a basis for learning the distribution of the state when given samples from this distribution. However, Hard states are not as highly entangled as Random States, and so they do have some structure that can be learned by our model(s).

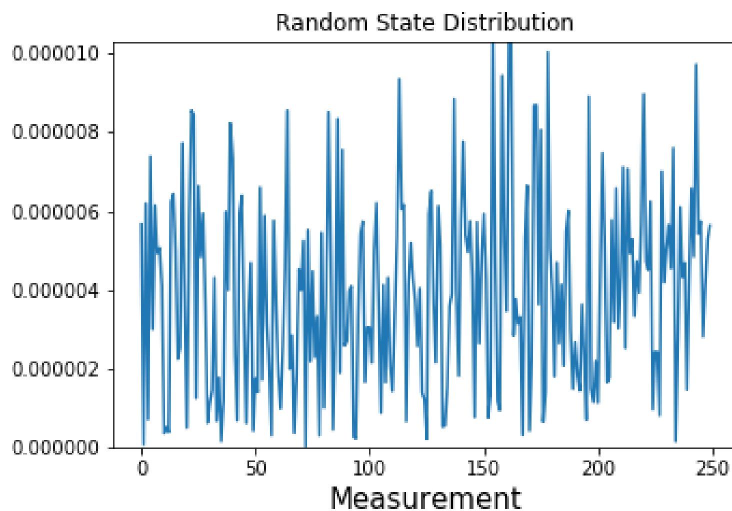


Figure 1.3: An example of a Random State Distribution. Random states are difficult to learn due to the fact that they are highly entangled and normally can't be factorised. To write down the wave-function we need 2^{N} complex numbers (amplitudes) where N is the number of qubits.

We then use this distribution to sample from repeatedly in order to feed these independent and identically distributed examples into our models.

In this project we attempt to reconstruct 3 different types of quantum states using generative models. Learning a state distribution with high fidelity using one of the models we outline in section 2 requires some repeated or inherent pattern to be embodied within the distribution to be learned.

In Quantum terms, we can generally say that the more entangled a state is, the less structure there is contained within the distribution, and so this is harder for a model to learn with fewer parameters than originally required to express the state. However there exist quantum states such as the W state which has a high degree of structure despite being highly entangled.

However, quantum states that can be factorised can be represented by fewer parameters and so when we train a model to learn a state that can be factorised, we expect these models to be able to reconstruct these states with high accuracy as opposed to states that cannot be factorised while keeping the number of parameters (compression rate) in the model constant.

In order of learnability from the easiest to learn to the hardest, we have: Product states, Hard states and Random states.

Although a high level of entanglement in a quantum state distribution generally reduces their learnability, some highly entangled states can be represented efficiently using neural networks. This is demonstrated in this project and in [12] in the hard state; it represents a state with high entanglement but some structure that can be learned.

1.2.1 Forming Samples

In order to use the distribution as an input to our models, we take many batches of samples from the distribution and use the sample batches as our input.

This is achieved by first forming a list of all of the binary permutations of the given number of qubits in the system for the current experiment. For example, if we have 8 qubits, then there are 2^8 different binary permutations, i.e (00000001,00000010,...,00000011,...,11111111). This is coincidentally the same number ($2^8 = 256$) of parameters required to express an 8 qubit quantum system. We then sample from this permutation list according to the distribution (itself a list of ($2^8 = 256$) probabilities all summing to 1 formed using the QuTiP library), thus creating a large batch of samples. These batches, which are sampled at every iteration in the described way, is how we train our models.

Then to sample from the Variational Autoencoder (described in the next chapter) we form batches of samples from the standard normal distribution which we feed into its decoder.

1.3 Quantum Gates and Change of Basis

Changes to a quantum state are brought about by the actions of *quantum gates*. Instead of the use of wires and logic gates as in classical computer systems, a quantum system utilizes a quantum circuit and quantum gates in order to perform computations on quantum information.

Quantum gates on one qubit can be specified by a 2x2 matrix. Primarily in this project we use the recursive tensor product of the *Hadamard* gate:

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

in order to construct a second basis of a given quantum state. This is also known as the Hadamard transform.

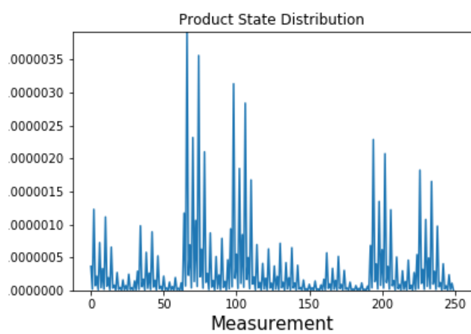
The recursive definition for the Hadamard transform we have as:

$$H_m \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix}$$

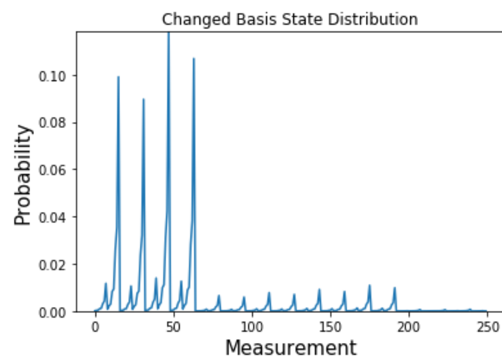
which is defined for multiple qubit systems as:

$$H_m = H_1 \otimes H_{m-1}$$

We then apply this large matrix to the original product state distribution. This gives us a different basis that we can measure the state distribution on in order to form a different measurement distribution form the one we had previously. Learning both of the measurement distributions generated from measuring on both bases of the state distribution (measurement basis prior to the transform and the measurement basis we form after the transform) is the objective in chapter 5.



(a) Measurement distribution on X-basis.



(b) Measurement distribution on Z-basis.

Figure 1.4: Measurement distributions of (a) X and (b) Z bases of a product state. An example of the measurements from a basis change resulting from a Hadamard transform applied to a Product State Distribution.

Chapter 2

Neural Networks

In this chapter we discuss the two neural networks used in this project. The Variational Autoencoder (VAE) and the Restricted Boltzmann Machine (RBM) both lend themselves in different ways to the task at hand. Both architectures are examples of generative models and aid us in learning quantum probability distributions by learning trainable parameters. Ultimately our goal is to reconstruct these distributions from the learned parameters of the models thus giving us the original quantum probability distribution with varying fidelity.

Notably, the two models mentioned are generative models as opposed to most neural network architectures which tend to be discriminative models. This difference will be explored in depth in this chapter.

2.1 Background

An artificial neural network (ANN) can be essentially seen as a non-linear function approximator. They largely consist of layers of 'neurons' that are fully connected between layers but have no connections within the same layer with a non-linear activation function applied to the output of each hidden (inner) node.

Most ANN models described in literature ([7],[11]) take an input and transform the input through many layers to classify the usually much higher dimensional input into a much lower dimensional output . These are known as discriminative models as they essentially approximate $P(\text{output—input})$ once they have been trained. Generative models such as the VAE and the RBM on the other hand learn a model of the input data and can therefore generate previously unseen inputs which follow the same pattern as the ones it has trained on. The neural networks we use here have significantly fewer parameters than the amount of data that they are trained on, so the networks are encouraged to detect and efficiently incorporate the underlying pattern of the data in order to generate new examples. In our case, we aim to reconstruct the same quantum state distribution that we

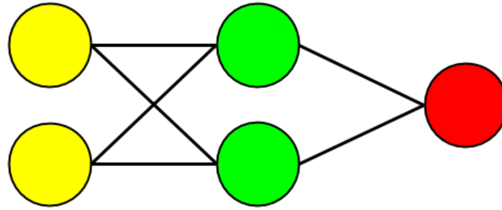


Figure 2.1: An example of a basic Multilayer Perceptron which forms the foundation of the other neural networks we outline. The two yellow nodes in the first layer serve as the inputs to the network, the green nodes serve as the 'hidden' layer and the red node is the output. The green nodes each have a matrix W_i of weights between both of the input neurons and itself and add a bias to the matrix multiplication $W_i * x$ where x is a vector (or matrix if x is more than 1 dimensional) of the inputs. In this case for each green node i we have $h_i = W_i x + b_i$ as the computation performed. Each W_i and each b_i are trained via backpropagation in order to minimize the difference between the ground truth label y_j for each input x_j and the output at the red neuron of the neural network which we denote y'_j.

inputted.

2.2 Autoencoders

The function of a standard autoencoder is to encode the input data using an encoder (a neural network in this case) producing a latent variable which is a vector of real values. This latent variable is then passed into the decoder (also a neural network) as an input and the output of the decoder is compared to the original data that was input to the encoder. The difference between them is used as a loss function which the whole network aims to minimize via gradient descent.

Multiple pieces of data (say images) can be used to train the autoencoder. By saving the latent vectors produced by the encoder, we can later reconstruct a particular image by inputting the latent vector into just the decoder portion.

Therefore we can say that the autoencoder is a form of compression as the total size

of the parameters in the autoencoder as a whole is far lower than the total size of all the pieces of data (images) that it has learnt.

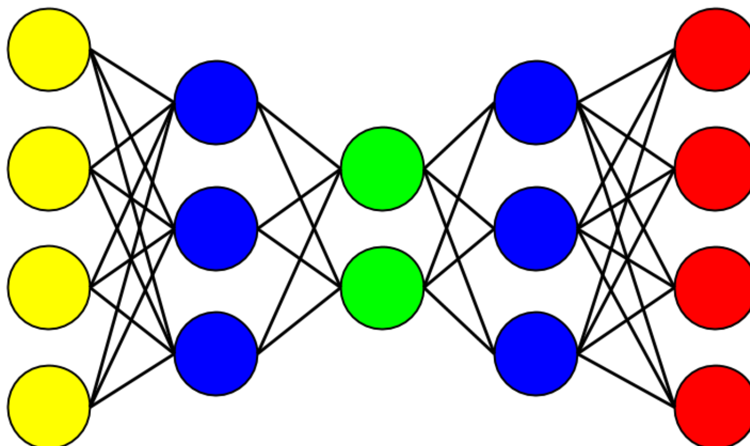


Figure 2.2: The graphical representation of an Autoencoder has an hourglass shape with a larger input layer dimension (in yellow) and output layer dimension (in red) along with a larger encoder and decoder layer dimension (in blue) than the latent space dimension (in green). This bottleneck aims to provide as compressed a representation as possible of the input data in the latent space layer. The decoder then aims to reconstruct the original data using just this compressed representation from the latent space.

2.3 Variational Autoencoders

With the Autoencoder just described, latent variables are encoded by the encoder layer, but there is no alternative to creating meaningful latent variables other than letting the encoder layer decide them with said architecture, and no way of generating different samples of the same data distribution as a generative model should be able to do.

The Variational Autoencoder (VAE) was introduced in 2013 by Welling and Kingma [5]. The VAE's architecture differs from the Autoencoder in one particular way. Instead of the latent variables being formed arbitrarily as with the Autoencoder, the VAE's encoder coerces the latent variables to follow a unit Gaussian distribution (standard normal distribution), that is $\mathcal{N}(\mu = 0, \sigma^2 = 1)$.

Now we can observe that the VAE is a true generative model; in order to generate new data following the same distribution the VAE was trained with, we just need to sample from the unit Gaussian distribution $X \sim \mathcal{N}(\mu = 0, \sigma^2 = 1)$ and pass this sample X into

the decoder of the trained VAE.

The loss function we optimized for previously with the autoencoder was just the reconstruction divergence from the original input. With the VAE our loss function to optimize is a combination of the aforementioned reconstruction divergence *and* the divergence between the the latent variables produced by the encoder of the VAE compared to the unit Gaussian distribution.

The divergence between the two distributions is measured using the Kullback-Leibler divergence, which is an indicator of the relative entropy between two distributions.

Thus our loss is now composed of:

$$\text{Reconstruction Loss} = \frac{\sum_i^{|Data|}(VAE(Data_i) - Data_i)^2}{|Data|}$$

$$\text{Latent Loss} = \text{Kullback-Leibler divergence(latent variables}||\mathcal{N}(\mu = 0, \sigma^2 = 1))$$

$$\text{where Kullback-Leibler divergence}(P||Q) = \int_{\infty}^{-\infty} log(\frac{dP}{dQ})dP$$

$$\text{And so, our combined loss} = \text{Reconstruction Loss} + \text{Latent Loss}$$

The VAE network would then aim to optimize this combined loss. However, calculating the Kullback-Leibler divergence analytically at every iteration would be computationally expensive, and so we reparameterize this measure. The resulting modification is that instead of the latent variables being of the form $V \in \mathbb{R}^d$ where d is the latent dimension, we instruct the encoder to generate a vector of means and a vector of standard deviations. Then in order to calculate the Kullback-Leibler divergence we now have:

$$\text{Latent Loss} = -\frac{1}{2}\sum_i(1 + log(\sigma_i^2) - \mu_i^2 - e^{log(\sigma_i^2)})$$

for each piece of data i, where μ_i and σ_i are the mean and standard deviation respectively.

Then when we want to test for how well the VAE has learnt a given set of data, we sample from the Gaussian distribution using the mean and standard deviation pairs. Particularly for each $\mu_i, \sigma_i \in$ latent space $= ((\mu_1, \sigma_1), ..., (\mu_n, \sigma_n))$ we sample from $\mathcal{N}(\mu_i, \sigma_i)$ to obtain a latent variable for each i which we use as an input for the decoder network. We then compare the difference between the output of the decoder and the original input that we passed to the encoder.

This process of sampling from the normal distribution to form the latent variable also

occurs when training the VAE. Consequently a particular input that caused the encoder to generate a certain mean and standard deviation, would therefore have a distinct latent variable value after the encoder portion and would now only have a high probability with some standard deviation of being the same latent variable value when input to the decoder.

This forces the VAE to be very efficient in the way it encodes and decodes inputs as it introduces inherent (Gaussian) noise to the architecture. This therefore means the model can generalize a great deal, and can generate examples from the same distribution as the input data with good fidelity.

In this project, we will test how well VAE's with differing numbers of layers can learn a given Quantum state distribution. Although the number of layers will differ, the number of parameters will remain the same in order to ensure that the depth of the network is the independent variable.

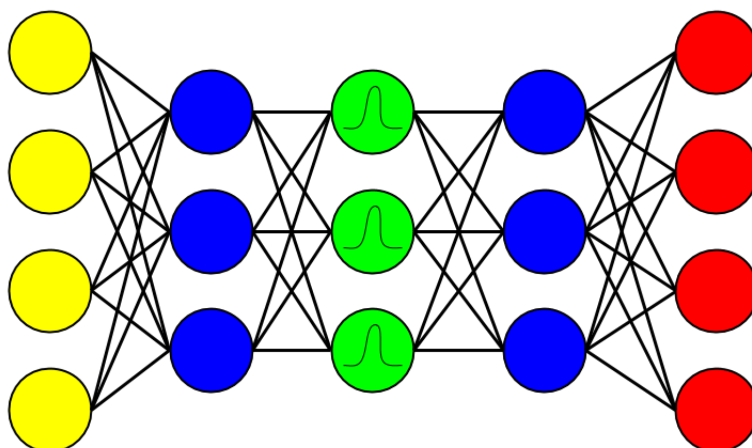


Figure 2.3: The graphical representation of the VAE is very similar to that of the Autoencoder. The major difference is that instead of trying to solely compress the input samples as the Autoencoder does, the VAE instead essentially aims to learn an approximation to the underlying probability distribution of the inputs. In this sense, they are a generative model similar in a sense to Restricted Boltzmann Machines which we will introduce in the next section.

2.4 Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBMs) are relatively shallow in comparison to the autoencoder models already described. Typically they have 2 layers with the first being the

visible layer and the second being the hidden layer. Also as previously, the two layers are fully connected with the restriction that there are no connections between any two nodes (neurons) in the same layer. This constraint is what gives the model the first part of its name.

The visible layer here acts as both the input and the output layer. As before we have a weight W_{ij} between each visible (input) node i and each hidden node j and a bias at each hidden node.

So for a given input $x \in \mathbb{R}^d$,where d is the dimension of the input (layer), we have as before $(W * x + b_{hidden})$ followed by an activation function (eg. sigmoid) being the computation performed at the hidden layer where $W \in \mathbb{R}^{d \times \text{hidden layer dim}}$ is a matrix with the same number of rows as the dimension of the input (d) and the same number of columns as nodes in the hidden layer and b_{hidden} is a vector with the same dimension as the number of nodes in the hidden layer.

However unlike previously, this would not form the output. As well as the bias at the hidden layer (b_{hidden}), we also have a bias applied at the visible layer $(b_{visible})$. The RBM now attempts to reconstruct the input from the output of the previous computation:

$$\text{hidden output} = sigmoid(W * x + b_{hidden})$$

. The previous output is now the input and it is multiplied by the weight matrix W and summed to the bias at each visible layer node.

The computation now performed at the visible layer is:

$$\text{visible output} = W * \text{hidden output} + b_{visible}$$

This is a (first) approximation of the original input to the RBM. The reconstruction error between this approximation and the original input is then used to update the weight matrix W and the $b_{visible}$ vector.

The b_{hidden} vector is updated by the difference between

$$\text{hidden output} = sigmoid(W * x + b_{hidden})$$

and

$$\text{hidden output} = sigmoid(W * \text{visible output} + b_{hidden})$$

. This process is then repeated continuously from the feed-forward computation until convergence (very low reconstruction error). This training method is known as Contrastive Divergence.

The RBM learns to reconstruct the original distribution from which the input is sampled in an unsupervised way by performing repeated forwards and backwards passes between

the visible layer and the hidden layer. The technique is unsupervised as the model is not shown a set of inputs and their corresponding labels (as with supervised learning); it is just shown a set of inputs and tasked with identifying the patterns that are present, which would then enable the model to internally replicate the distribution the inputs came from.

The ultimate goal would be to train a model that has the ability to generate examples from a distribution (almost) equivalent to the distribution of the original inputs. This makes the RBM a generative model.

Similarly to the VAE, the RBM is stochastic as we sample from the Bernoulli distribution after applying the sigmoid function at both the visible and hidden layer before we do another pass; thereby transforming our real value output into a binary one with some randomness. The Bernoulli is parameterised by the output of the sigmoid function in each case. This means that the model is inherently probabilistic in nature.

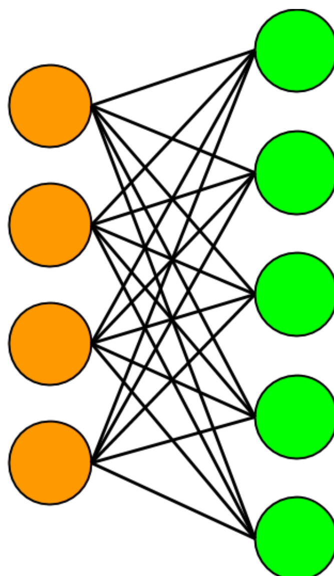


Figure 2.4: The graphical representation of the Restricted Boltzmann Machine is distinctly different to that of the Autoencoder. Here we have only 2 layers, no latent space and no encoder or decoder. Instead, we have a visible layer (in orange) and a hidden layer (in green) which are fully connected between the layers with no connections within the same layer. The visible layer acts as both the input and output layer. We make repeated forwards and backwards passes updating the weights between the two layers at each iteration until convergence as we attempt to reconstruct the input data accurately.

2.4.1 Deep Restricted Boltzmann Machines

Deep Restricted Boltzmann Machines are similar to standard RBMs in that they are also trained using contrastive divergence. The computations outlined in the previous section

also apply here, the forward pass computations just have an extra layer, as do the backwards passes. The extra layer of weights and biases allow the network to be more expressive and potentially learn more about the distribution it is training on. Depth in a neural network has shown to be an important factor in a network's ability to learn. Papers such as [7] used (convolutional) neural networks with vast numbers of layers to achieve their results. In this project we wish to observe whether depth in an RBM can be utilised to better learn our state distributions with fidelity superseding that of the standard RBM.

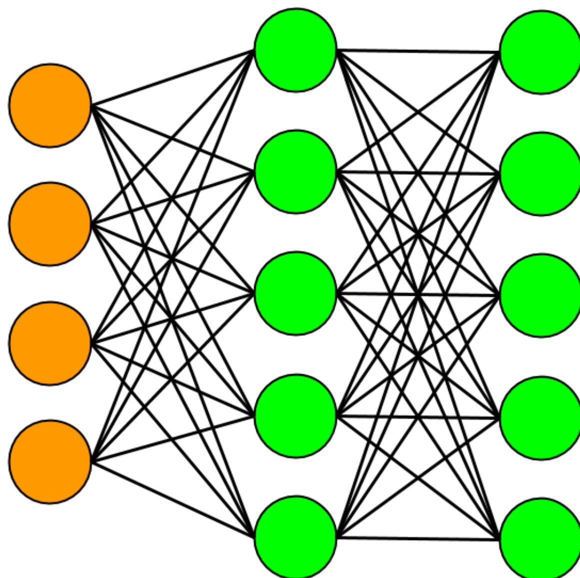


Figure 2.5: The graphical representation of a Deep Restricted Boltzmann Machine is similar to that of the standard RBM, but also features an extra hidden layer (hidden layers in green). The visible layer (in orange) is still both the input and output layer.

2.4.2 Complex valued Restricted Boltzmann Machines

As outlined here [8] and in [12], a complex valued RBM is specifically designed to have complex values as an input. In this project, we take the absolute value of the complex numbers in a quantum state distribution. This model gives us the opportunity to attempt to learn the raw distribution made up of complex numbers. As before, there are no connections in the same layer and training is still performed by contrastive divergence.

As shown in figure 2.6, the complex RBM has a complex valued visible layer, and therefore has a visible layer for the real part and a visible layer for the imaginary part of a complex number each with its own bias vector. There are connections between these layers to aid with learning. Between each of these layers and the (single) hidden layer we have a set of weights for each component and a single bias vector for the hidden layer.

This model will be tested along with the standard and deep RBMs to see if the complex nature of the input can be utilised to gain good results using the architecture class.

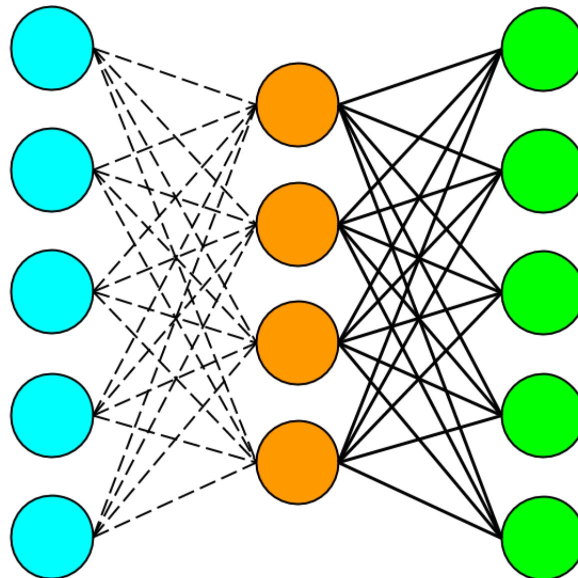


Figure 2.6: The graphical representation of the complex valued Restricted Boltzmann Machine. Here the visible layer (in orange) once again represents both the input and the output layer. The input here is complex valued, and so we pass the imaginary component to the hidden layer in blue, and the real component to the hidden layer in green. In this way, we have a separate set of weights for each component of the complex input. Sometimes the two hidden layers have connections between each other to help with learning.

2.5 TensorFlow

To utilize the above architectures, we need a robust way of running experiments on them computationally efficiently. This is especially important given their complex nature.

TensorFlow [1] is a deep learning framework which is primarily formed of a library for defining computational graphs and a runtime environment for executing those graphs once they have been specified on a variety of different hardware; namely CPU and GPU are the two main chipset architectures it is optimized to run on.

In this project we use the GPU implementation of TensorFlow as the computational resources required to run the experiments in this project (which each represent a huge number of large matrix multiplications and the storing of many thousands of floating point values throughout) can only be met by using a high memory and highly parallel set up afforded by an advanced GPU architecture. For this project an Nvidia GEFORCE GTX 1080Ti with 11 gigabytes of memory was used throughout.

Computational graphs are an abstract method of characterizing computations as a directed graph. Here the edges in the graph (as seen in Figures 2.1, 2.2, 2.3 and 2.4) are equivalent to multidimensional arrays which we denote as **Tensors**. The nodes in the graph construct these tensors and perform computations on them according to the procedure we prescribe. These are denoted as **Ops**.

In this project, our experiments run with TensorFlow are composed of firstly defining the computational graph we will be running on (e.g specifying the structure of the VAE). Then in a separate section we have the execution of the defined graph within a *tf.Session*.

TensorFlow also has a set of built in high level optimizers which automatically compute the gradients during backpropagation and apply the resulting updates which saves us having to do this manually. In this project we use the AdamOptimizer [6] which has been found in general to achieve good results with limited training when compared to the initially widespread GradientDescentOptimizer.

2.6 Summary

In this section we have outlined the neural networks which we will focus on for this project. The ultimate goal is to compress distributions of varying complexity into a fraction of the number of parameters required to represent them classically.

We test the above networks with differing numbers of layers and hyper-parameters to ascertain which architecture is best suited to this task. Thus we aim to establish which is the best for future work and in achieving the more ambitious goals in this project.

Chapter 3

Objectives and previous work

The work in this project builds upon the work in this paper [10] by Rochetto, Grant et al. In their paper they introduce the use of Variational Autoencoders in tackling the issue at hand: compressing (hard) quantum states efficiently and accurately. In this project, their work is built upon by comparing the use of VAEs of various depths against the use of other neural networks. The Restricted Boltzmann machine is largely seen as the basic unit in the field of deep learning, and is similar to the VAE in that its primary function is to learn a probability distribution and reproduce said distribution when sampled from.

3.1 Takeaway from Rochetto et al.

In [10], the authors show concretely that deep neural networks can be used to approximately represent hard quantum states. They go on to state that "neural networks are able to capture correlations in states that are hard to sample from for classical computers but not for quantum ones".

The major take away is that neural networks such as VAEs use far fewer parameters to approximately represent states that that are known to have an efficient representation. The compression rate that they achieve for hard states while maintaining 92.2% accuracy is a factor of 5. For easy states such as the product state distribution, the accuracy is very close to 100% while maintaining high compression.

3.2 Other Work

Quantum state tomography (QST) is discussed by Torlai et al. in their paper [12]. Their conclusion coincides with the previous paper in that neural networks can be efficiently used in Quantum state tomography, particularly when the states are highly entangled. They state that the current state of the art method (Matrix-Product-State) is effective for

tomography of low-entangled states, but that neural networks in particular when trained sufficiently can be a robust way of performing QST with highly-entangled quantum states. The authors here use a complex valued Restricted Boltzmann machine in order to reconstruct the state.

As stated, the current state of the art method for efficient quantum state tomography involves approximating quantum states using a matrix product state [9] in order to deduce the state of a quantum system efficiently instead of having to conventionally deduce the state purely from measured data, which becomes intractable with larger systems. We can see this approximation applied here [2] with computations only reaching a polynomial number of operations (as opposed to an exponential number of operations without the approximation).

3.3 Aims of the project

The aims of this project will be based on the work outlined in the first two papers above. The motivation is to compare the neural networks used in the two papers and compare them for fidelity and expressiveness.

Principal aims:

- Learn product, hard and random state distributions using Variational Autoencoders with varying numbers of layers.

  This enables us to infer how much of an effect the depth of the VAE encoder and decoder has on the fidelity of the network at set rates of compression.

- Learn product, hard and random state distributions using a Restricted Boltzmann Machine.

  We compare the RBM to the VAE results in order to confirm which generative architecture is best suited to the task of quantum state tomography.

- Adapt the Variational Autoencoder to learn a full state on a complete set of bases.

  Here we essentially attempt to coerce the VAE to learn more than one basis of the quantum state distribution simultaneously. This is essentially tasks the VAE with learning at least two probability distributions and reconstructing both from the same set of parameters (weights, biases and latent variables).

Advanced aims:

- Implement a deep RBM to learn product, hard and random state distributions

  Intuitively a deep RBM may be able to overcome the expressivity limitations of the standard (shallow) RBM due to having more layers. Thus by having a better suited architecture we may be able to learn and reconstruct the distributions with higher fidelity than previously.

- Implement a complex valued RBM to learn product, hard and random state distributions.

  The aim here is to implement the complex valued RBM model used in [12] and test against the VAE and standard RBM model.

- Reconstruct the full state from the learned VAE representation.

  If the VAE is successful in learning more than one basis of a state, we could theoretically reconstruct both bases and thus have the raw ingredients required to reconstruct the full state.

- Test and compare the fidelity of the distribution learned by the VAE encoder.

  The VAE is tasked with encoding the distribution of data in both the encoder and the decoder. Currently we use the decoder to reconstruct the distribution. Instead we attempt to use the encoder as it does not rely on sampling and thus we can directly compare the fidelity of the distribution learned by the VAE encoder.

Chapter 4

RBMs vs VAEs

In this chapter we describe the first set of experiments run in this project. We aim to solidify the claim that Variational autoencoders (VAEs) are the foremost approximator in learning large quantum states via Quantum state tomography.

We hypothesize that the Restricted Boltzmann machine will give a higher reconstruction error for the same number of parameters in a given model than the Variational autoencoder (especially deeper models with 4 and 5 layers).

If this is the case, then for the VAE models we expect this will give a higher overlap between the true distribution and the reconstructed one when learning the three quantum probability state distributions we outlined in chapter 1. The product, random and hard states will each be representing a 12 qubit system for this set of experiments. As previously stated, the number of parameters required to express a quantum state increases exponentially as a function of the number of qubits in the system. Therefore we would need 4096 (12^2) parameters in order to represent each of these state distributions.

The main task outlined in this chapter is to compress the state distributions using the generative models outlined in chapter 2 with a compression of 50% and 25%. Thus the models are limited to only 2048 and 1024 parameters in their respective networks in order to learn the state distributions. This translates to the number of nodes in each model being proportional to the number of parameters, thus dictating the number of nodes in each layer. We then compare the various networks and observe their associated fidelity when attempting to reconstruct the states.

4.1 System Set up

With regards to the VAE, our loss function is essentially:

$$\text{Reconstruction Error} + \text{Kullback-Leibler divergence}$$

as stated in 2.3. However, starting with this loss function in its entirety causes the VAE to prioritize generalisation ability over reconstruction fidelity as the two loss components are treated equally. Therefore, we start by weighting the Kullback-Leibler divergence (the divergence between the reconstructed distribution and the original state distribution) in the loss function as 0 and gradually increase this weighting at every iteration to a maximum weighting (which will be treated as a hyperparameter).

So the loss function we optimise is:

$$\text{Reconstruction Error} + (\text{Current KLD Weight} * \text{Kullback-Leibler divergence})$$

where

$$\text{Current KLD Weight} = \frac{\text{Maximum KL Divergence}}{\text{total iterations}} * \text{current iteration}$$

Parameter	Value
Total Iterations	1,000,000
Training Batch Size	1,000
Training Learning rate	0.001
Maximum KL Divergence Weighting in loss function	0.85
Sampling frequency	Every 10,000 iterations
Total samples at each sampling interval	245,760
Sampling Batch Size	1,000

Table 4.1: VAE Training configurations for experiments on 12 qubit product, random and hard state distributions. The batch size specifies how many samples from the state distribution we feed into the model at every iteration. The learning rate specifies the magnitude of the updates made to the weights of the network at every iteration by the optimizer; here we use the Adam optimizer [6]. Sampling frequency refers to how often we switch from training the model to sampling from it to calculate the overlap between the sampled distribution and the original state distribution. The number of samples at each batch size is large as we want to calculate an accurate measure of overlap; the more samples we have from the model, the better we can form the reconstruction distribution.

We also require a metric that indicates if the model is indeed learning the state distribution at all or has conversely not been able to learn anything useful. We actually expect that the latter is the case when attempting to learn using samples from the random state. In this capacity we check that the overlap (fidelity) between the reconstruction and the state for a given model is above the magnitude of the overlap between the uniform distribution $Uniform(0, 2^{12})$ and the state distribution. This can be visualized in figure 4.1.

We maintain that in order for a given model to have learned a given state distribution, the reconstruction overlap needs to be above this baseline value.

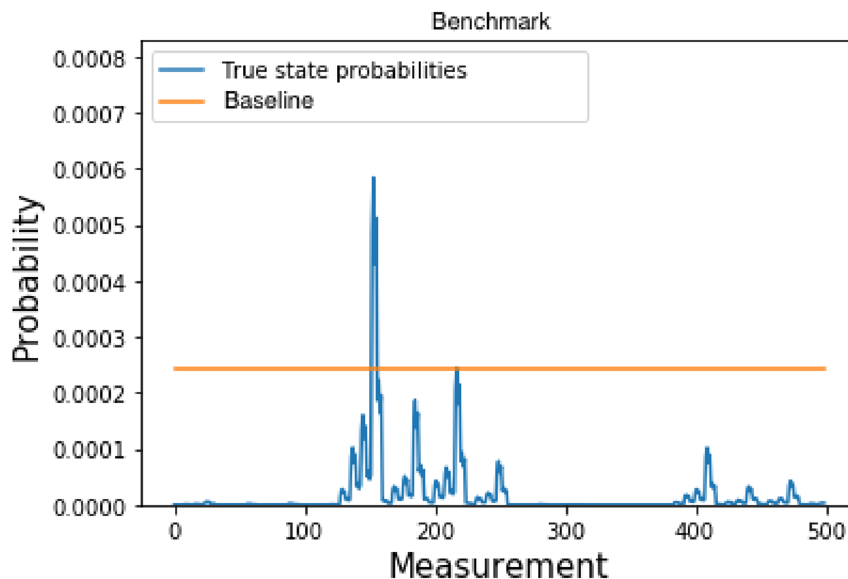


Figure 4.1: A plot of a uniform benchmark (in orange) on a 12 qubit product state distribution

For these experiments we test the fidelity of the different model architectures when learning the three variants of quantum distributions. The model architectures tested on were: Variational Autoencoders with 1,2,3,4 and 5 layers and with a parameter compression of 0.5 and 0.25 for each VAE of differing depth. We then compare these along with a Restricted Boltzmann machine architecture also with parameter compressions of 0.5 and 0.25.

Later experiments were also run on Deep and Complex valued RBMs to test if these variants of the standard RBM had a lower reconstruction error than that of the VAE models.

4.2 Experiments and results

As previously stated, these experiments were run using a high end graphics card with eleven gigabytes of memory; without access to such hardware, building and running neural networks using TensorFlow is prohibitive at this scale, both memory and time wise.

Training the VAE for just one epoch (even with the GPU) takes a few hours, and so adjusting hyperparameters between each set of iterations before rerunning was laborious. Training the RBM did not take nearly as long due to its shallow architecture. The caveat

of this of course is that the VAE models achieved better performance overall.

Model (Compression rate)	Product State	Random State	Hard State
Baseline for comparison	0.5864	0.9371	0.9031
VAE 1 layer (50%)	0.9868	0.9317	0.8977
VAE 1 layer (25%)	0.9929	0.9325	0.8978
VAE 2 layer (50%)	0.9883	0.9317	0.9162
VAE 2 layer (25%)	0.9908	0.9323	0.9144
VAE 3 layer (50%)	0.9919	0.9311	0.9188
VAE 3 layer (25%)	0.9916	0.9316	0.9124
VAE 4 layer (50%)	0.9906	0.9300	0.9189
VAE 4 layer (25%)	0.9908	0.9312	0.9124
VAE 5 layer (50%)	0.9900	0.9314	0.9212
VAE 5 layer (25%)	0.9941	0.9289	0.9106
Real valued RBM (50%)	0.6617	0.9348	0.9008
Real valued RBM (25%)	0.6331	0.9348	0.9009
Real valued Deep RBM (50%)	0.6540	0.9348	0.7961
Real valued Deep RBM (25%)	0.6534	0.9289	0.8014

Table 4.2: The overlap results of reconstructing the Product state, Random state and Hard state distributions using Variational Autoencoders with varying numbers of layers in the encoder and decoder and also using three variants of the Restricted Boltzmann machine. The baseline performance is also included to show whether or not the models are better than the benchmark outlined in the previous section. If performance is worse than the benchmark, then it is clear that the models are not learning the structure of the distribution they have been tasked with learning.

4.3 Analysis

From these results in table 4.2, it is clear that our original hypothesis was correct. All five VAE models outperform the RBM models in terms of overlap when reconstructing the product and hard states. The slight improvement in overlap of the RBM over some of the VAE models when reconstructing the random state can be largely ignored as the variance in this result is very low across all the models. Intuitively, random states are not learnable, and so any slight improvement here is not consequential.

A notable observation is that deeper VAE models (those with more layers) do not have concretely higher fidelity than shallower models when trained on product states, although

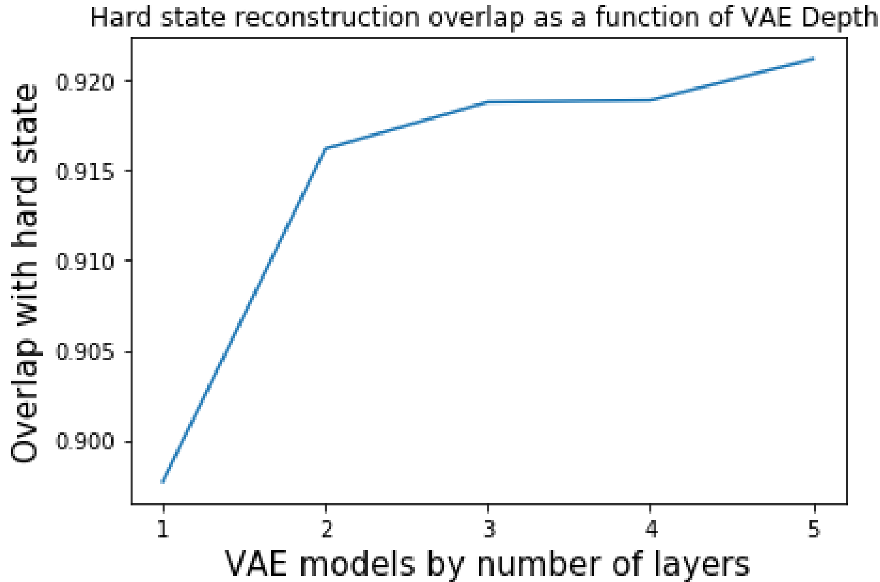


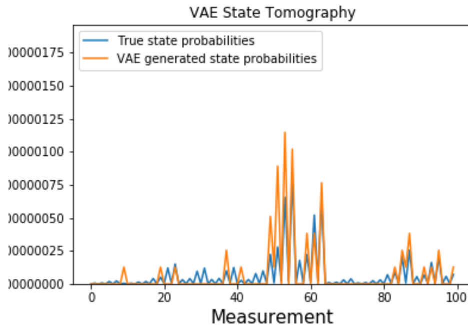
Figure 4.2: A plot of hard state reconstruction overlap as a function of VAE Depth. It is evident that increasing the number of layers in a VAE encoder and decoder network increases the learnability of a **hard state**. This is while keeping the number of parameters in the network the same. Thus the number of nodes overall are proportional, so the deeper the model, the fewer the number of nodes in a layer to maintain fairness.

there is a general upwards trend in the overlap of the hard state reconstruction with the original as can be seen in figure 4.2. This may be due to 12 qubit state distributions not being large enough to warrant explicit differences in the results when looking at the depth of the networks.

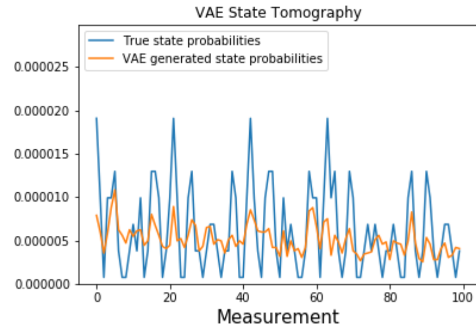
Similarly, the 50% and 25% compression rates do not show much difference when comparing product state overlap for a given depth of VAE; however when comparing hard state overlaps, we do find that a lower compression rate increases the overlap value. Thus we can derive that states that are hard to learn, require the expressivity granted by more network parameters to better learn the state distribution structure.

Examining the figures 5.3 and 4.4 which show the plots generated when we sample from the VAE decoder and the RBM visible layer respectively, we evidently note that the VAE reconstructed distribution very closely resembles the product state distribution whereas the corresponding RBM distribution does not nearly achieve the same likeness.

Then when looking at the hard state reconstructions, we can see that neither model (VAE nor RBM) manages to attain a close resemblance to the target hard state distribution. However, as the overlap value is above that of the baseline (uniform distribution), the models in these cases have nonetheless realized some of the structure of the hard state

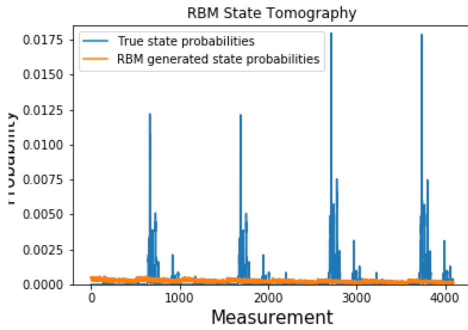


(a) 5 layer VAE reconstruction with 50% compression rate, trained on 12 qubit **product** state.

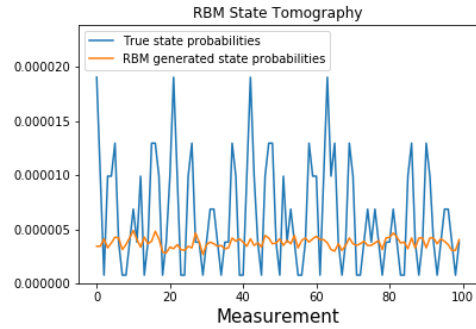


(b) 5 layer VAE reconstruction with 50% compression rate, trained on 12 qubit **hard** state.

Figure 4.3: Final reconstruction of VAE learned states after training.

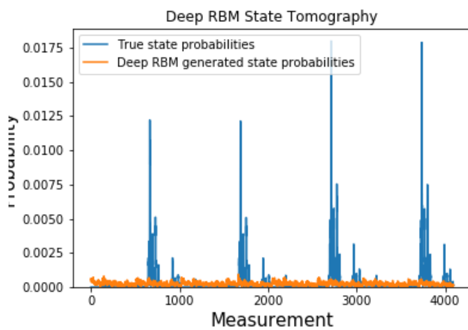


(a) RBM reconstruction with 50% compression rate, trained on 12 qubit **product** state. A greater range of the plot is included here to show the overall pattern of learning.

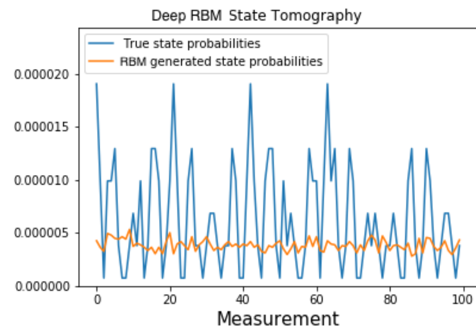


(b) RBM reconstruction with 50% compression rate, trained on 12 qubit **hard** state.

Figure 4.4: Final reconstruction of (standard) RBM learned states after training.



(a) Deep RBM reconstruction with 50% compression rate, trained on 12 qubit **product** state. A greater range of the plot is included here to show the overall pattern of learning.



(b) Deep RBM reconstruction with 50% compression rate, trained on 12 qubit **hard** state.

Figure 4.5: Final reconstruction of Deep RBM learned states after training.

distribution.

Proceeding to the RBM architectures, we observe that the deep RBM achieves noticeably better performance for both compression rates when compared with the standard RBM. This is likely due to the extra computations afforded by the extra layer in the deep model. From this we can deduce that the role of depth (to a point) is critical to reconstruction fidelity; this is clear when contrasting shallow models such as RBMs with deep models such as VAEs.

Chapter 5

Multiple Bases

In this experiment we attempt to use a single variational autoencoder to learn two bases of a state distribution. This is approximately equivalent to being tasked with learning two probability distributions using the same VAE network. The intuition here is that due to the two distributions being bases of the same state, there are structural similarities that are apparent when learning them, and thus this can be exploited by the VAE and so the model can indeed be effective in learning both bases.

As outlined in chapter 1, to form the two bases, we take an 8 qubit product state and take the modulus squared amplitude of its state vectors (which make up the distribution). This gives us the Z-basis. We apply the Hadamard transform to the product state and also take the modulus squared amplitude of the transformed state vectors to generate the X-basis.

By learning and reconstructing the two bases using the VAE decoder, we can consequentially reconstruct the original state. This is will be built upon in the future work section.

We hypothesize that we can learn both bases using a single VAE by slightly tweaking the architecture of the VAE. To achieve this, we use a switch which acts as a one-hot vector signalling the VAE as to which bases is being trained or sampled from. The switch takes the form of two binary valued neurons which are input to the encoder (during training) as well as the decoder portion (during sampling) of the VAE. We also try inputting the switch to the latent space instead of the encoder during training to see if this improves reconstruction accuracy.

We set the neurons to [1,0] when we want to learn or sample from the X-basis, and to [0,1] when we want to learn or sample from the Z-basis.

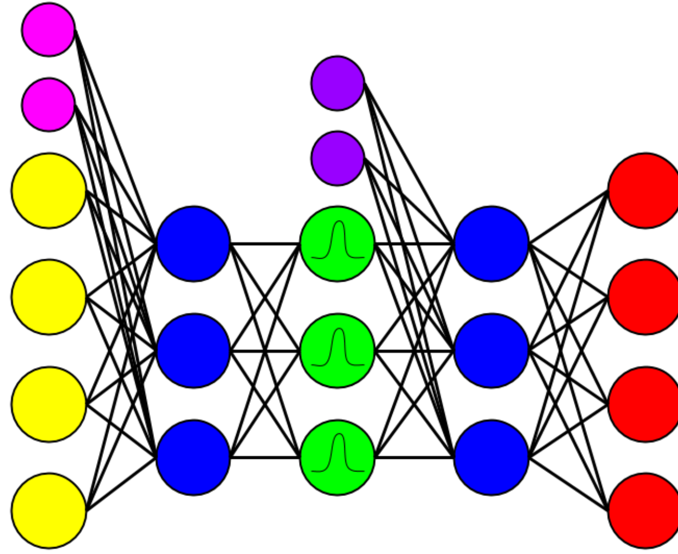


Figure 5.1: A graphical representation of the Variational Autoencoder adapted to learn two bases. The pink nodes represent the one hot vector being used as part of the input to the **encoder** when training the model while the purple nodes represent the one hot vector being used as part of the input to the decoder when sampling to form a reconstruction.

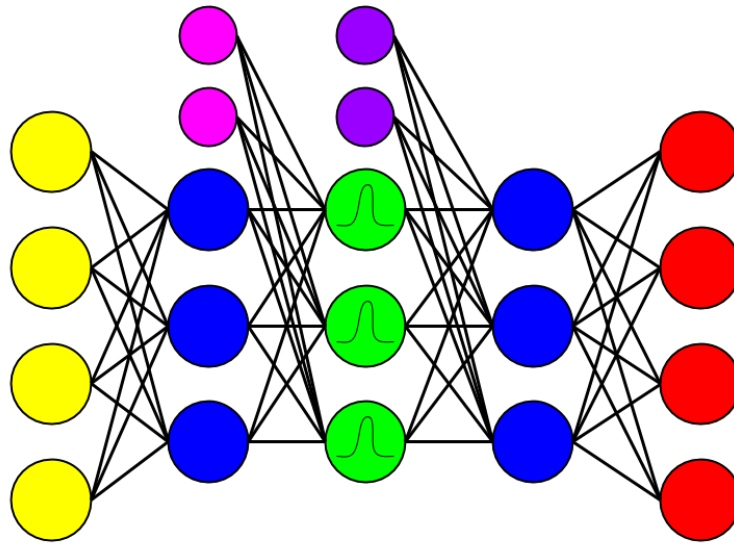


Figure 5.2: A graphical representation of the Variational Autoencoder adapted to learn two bases. In this alternative, the pink nodes represent the one hot vector being used as part of the input to the **latent space** when training the model while the purple nodes represent the one hot vector being used as part of the input to the decoder when sampling to form a reconstruction.

5.1 System Set up

We use the same loss function ass previously,

$$\text{Current KLD Weight} = \frac{\text{Maximum KL Divergence}}{\text{total iterations}} * \text{current iteration}$$

and also compare with the baseline uniform distribution to check that learning is occurring.

For this experiment we generate the two bases as described, then we train by randomly choosing at every iteration either the X-basis or the Z-basis to sample from to create a batch of samples which we then feed into the VAE. By randomly choosing the basis to train on at every iteration, we keep the network from favouring one distribution over the other.

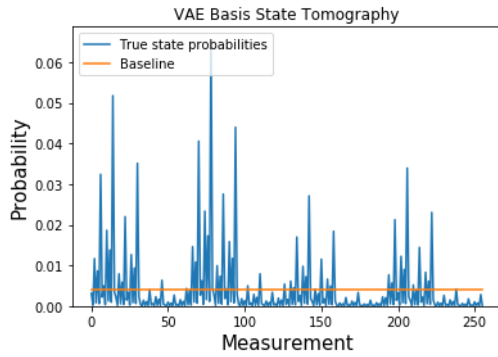
Once a basis to train on has been chosen, we set the one hot vector to the corresponding encoding and concatenate the vector to the batch of samples which is then used to train the VAE.

We can say that the technique outlined to simultaneously learn both bases has some merit if we have successfully reconstructed both bases with overlap values concretely above that of the baseline overlap value for each basis.

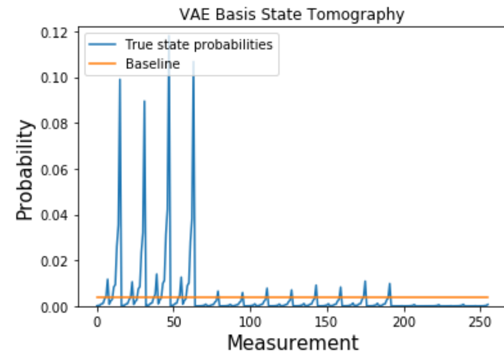
Other vector configurations were also tested to observe whether the one-hot vector form was optimal for signalling the current training basis to the VAE such as [0,0] and [1,1] for the X and Z basis respectively.

Parameter	Value
Total Iterations	1,000,000
Training Batch Size	500
Training Learning rate	0.001
Maximum KL Divergence Weighting in loss function	0.90
Sampling frequency	Every 10,000 iterations
Total samples at each sampling interval	51,200
Sampling Batch Size	100,000

Table 5.1: VAE Training configurations for experiments on X and Z basis formed from an 8 qubit product state distribution. Here we have smaller batches to avoid overfitting to one particular basis and larger sampling batch sizes as 8 qubits are far smaller than 12 qubit states (as before) and so we can sample far more from the decoder without running out of GPU memory.



(a) X-basis baseline



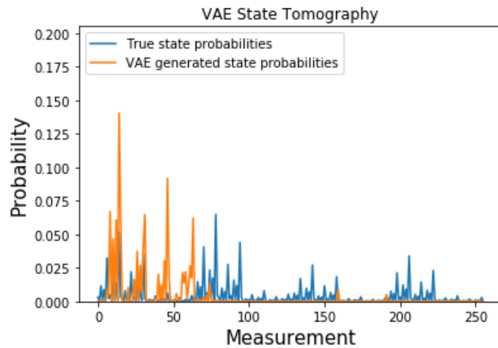
(b) Z-basis baseline.

Figure 5.3: Baseline and X and Z bases plots where bases are generated from an 8 qubit product state.

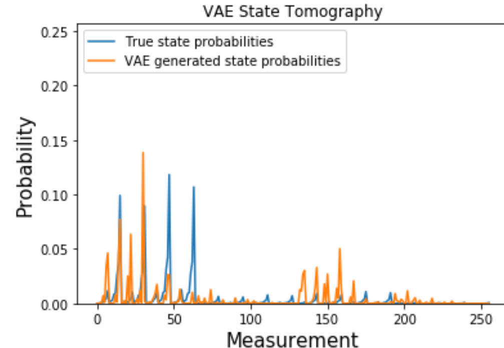
5.2 Experiments and results

Here we use the adapted VAE with the basis being used as a switch and being input to the VAE at the encoder during training and to the decoder during sampling as shown in figure 5.2. The VAE was trained over a million iterations randomly alternating between training on both bases. At the start of training, the overlap values for both fairly high (sufficiently above the baseline).

However as training went on, it appeared that the overlap value was decreasing and the loss for both the reconstruction loss and the KL divergence were both increasing as a result of consequent iterations.



(a) 4 layer VAE reconstruction trained on **X-basis** of 8 qubit product state.



(b) 4 layer VAE reconstruction trained on **Z-basis** of 8 qubit product state.

Figure 5.4: X and Z basis reconstruction.

Model (Compression rate)	X-basis overlap	Z-basis overlap
Baseline for comparison	0.6992	0.5505
VAE 4 layer	0.4552	0.6020

Table 5.2: The overlap results of reconstructing the X and Z bases formed from an 8 qubit product state using a 4 layer Variational Autoencoder. The baseline performance is also included to show whether or not the models are better than the uniform flat benchmark outlined in the previous section. If performance is worse than the benchmark, then it is clear that the models are not learning the structure of the distribution they have been tasked with learning.

5.3 Analysis

As noted, the fidelity of the two basis reconstructions began to decrease at about 250,000 iterations after being about constant up until this point. The results in table 5.2 show the results of the overlap accuracy after 1,000,000 iterations. The suspicion was that as the KL Divergence weighting was steadily increasing, reconstruction loss was decreasing in importance. This caused the VAE to slowly begin to prioritize the generalisation ability of the network over the reconstruction accuracy.

The VAE latent space was under increasing pressure to coerce the outputs of the encoder onto a standard Gaussian distribution and that doing so for both basis distributions increased the generalisation to a point where the network was almost seeing the two distributions as one.

This therefore caused the reconstruction overlap for both bases to lower. From the results in table 5.2 we can see that for the X-basis overlap the reconstruction fidelity decreased to below the baseline value by the final iteration.

From the plots in figure 5.4, it is evident by observing plot a) that the VAE has learnt some of the structure of the X-basis distribution from the shape of the VAE reconstruction plot. However upon closer inspection it can be seen that some of the structure of the Z-basis distribution is also apparent in the reconstruction.

Similarly in plot b), it is clear that on the left of the plot we see that the VAE has learnt some of the structure of the Z-basis distribution (especially the high peaks), however similarly if we look at the right hand side of the reconstruction plot we also see that some of the X-basis distribution has also been reconstructed here.

5.3.1 Further experimentation

The reasons for this are thought to be most likely due to the signal from the one hot vector not being strong enough to indicate to the VAE as to which basis distribution we

are learning from in a given iteration or sampling from when reconstructing.

In particular, due to the system being made up of 8 qubits (in our experiment), we have that the output of the encoder is concatenated to our one hot vector during training in our attempt to signal the distribution to learn. The dimension of the encoder output in this case is 20, whereas the one hot vector only has a dimension of 2. The ratio between the input to the latent space (output of the encoder) and the signal is fairly low. This could be the reason for low overlap despite considerable training on both basis distributions.

Other signal vectors were also tested in an attempt to increase the resulting overlap value for the respective basis reconstructions. Attempts such as having a vector with a dimension size of 4, i.e (0,0,0,0) and (1,1,1,1) for the X and Z basis respectively were tried.

In these instances it was found that the input was almost corrupted by the size of the one hot vector and this led to worse initial (as well as deteriorating) performance than previously. The original one hot vectors ([0,1] and [1,0]) were found to give the best results when generalizing across quantum systems of different sizes, e.g. for 8 and 12 qubit systems.

Chapter 6

Conclusions

6.1 Achievements

To conclude, in this project we have constructed Variational Autoencoders of varying depths and trained them to learn and sufficiently accurately reconstruct product, hard and random quantum state distributions.

We have also trained a Restricted Boltzmann machine to learn and reconstruct product, hard and random quantum state distributions. With our results we have shown that a VAE with 5 layers outperforms those with fewer layers, and that all of the VAE models outperform the RBM when it comes to reconstruction overlap accuracy when compared with the three state distributions.

We then went on to construct and train deep and complex RBM models based on [8] in an effort to improve on the original RBM results. This however did not prove very fruitful, perhaps a different implementation based more closely on [12] could better harness the complex valued nature of the states. Concretely we have shown that the depth of a generative network is key in reconstruction accuracy; this is evident in both the VAE and in the RBM. We still show that despite the adjustments to the original Restricted Boltzmann machine, the Variational Autoencoders should still be the generative artificial neural network model of choice when conducting quantum state tomography.

Our second experiment outlined and demonstrated a technique for learning a complete set of bases using a single Variational Autoencoder. After much work and testing, while the final results were not what we expected, with more work the desired outcome is certainly a possibility.

Once this has been achieved, our other advanced goal of reconstructing the full state from the learned VAE representation follows readily as all one must do to reconstruct the full state is sample from both bases (using a single VAE) with a sufficient level of fidelity to the original basis distributions.

6.2 Evaluation

The experiments in themselves require computing hardware with sufficient parallel processing power and memory to run and store the large computational graphs we require to run the models. Perhaps running the experiments with fewer qubit systems could have allowed more experimentation as the run time of said experiments would have been lower.

Then the time required to run these experiments once said GPUs were up and running was prohibitive in some cases when attempting to get an optimal result. This was especially apparent with the second experiment where the network had to train for a few hours before it was clear whether or not a proposed alteration to the signalling method would be fruitful.

Overall the implementations were robust, and as expected the RBM did not fare very well in learning the state distributions. This is expected, however the shape of the reconstruction by the RBM doesn't follow the shape of the state distribution very well; although the overlap is somewhat above the baseline value for all three states which is reassuring.

Although the plots produced by the VAE in the second experiment do roughly follow the basis distributions, the overlap value is fairly low; it is evident that the VAE is learning something, however there is no way to quantify this other than the overlap which is low. Perhaps another measure of distribution similarity could have been used to reflect this.

Due to the almost black box nature of the VAE in particular, it was difficult to diagnose the faults with the one hot vector solution in signalling the network as to which distribution was to be learned or reproduced. for this reason extensive experimentation had to be carried out to see if the performance of the VAE model with a one hot switch could be improved to a sufficient level. Unfortunately after countless hours of experimentation, the cause is still not very clear due to closed nature of neural networks in general. Experiments themselves were arduous to run due to the limited access to the GPUs.

The upside of all this is that neural networks represent a very powerful approximation paradigm and is very clearly the ideal means to perform quantum state tomography.

6.3 Future Work

As previously mentioned as one of the advanced aims of the project, reconstructing a full state would be the logical next step once we have successfully learned and reconstructed more than one basis of a state using a VAE. The involves using the one-hot vector to sample from both bases and reconstructing both distributions before combining them to reform the original full state.

Another advanced aim was to examine the distribution learned by the VAE encoder and potentially use this to reconstruct the state distribution. The advantage here is that instead of using the decoder and sampling from it to form a sampling distribution, we

can instead determine the distribution concretely from the encoder as we do not require sampling here. This saves on computation and improves our accuracy as we have access to the actual learned distribution instead of a sampled one.

Moving onto the switch mechanism itself for signalling to the VAE which distribution to currently learn, we could instead use a neural network to act as this signal instead which is also input to the encoder during training and to the decoder when sampling. The advantage of using a neural network instead of a static switch (one hot encoder) is that the neural network is trainable, and so we can define the loss function of the network as the difference between the distribution it predicts is being input and the actual distribution being input on that iteration. We then add this loss to our overall loss function for the VAE and try different weightings of the 3 loss components in order to optimize final reconstruction accuracy.

Other future work includes possibly using other generative models to contrast with the performance of the VAE in Quantum state tomography; for example, Generative Adversarial networks (GANs) [3] could prove useful in this regard and have shown to display remarkable results in other sub-fields of Machine Learning. The structure of the generative network of a GAN is somewhat similar to that of a VAE so not too much work is needed to test this.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. Cramer, M. B. Plenio, S. T. Flammia, R. Somma, D. Gross, S. D. Bartlett, O. Landon-Cardinal, D. Poulin, and Y.-K. Liu. Efficient quantum state tomography. *Nature Communications*, 1:149, December 2010.

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[4] J. R. Johansson, P. D. Nation, and F. Nori. QuTiP 2: A Python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 184:1234–1240, April 2013.

[5] D. P Kingma and M. Welling. Auto-Encoding Variational Bayes. *ArXiv e-prints*, December 2013.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[8] T. Nakashika, S. Takaki, and J. Yamagishi. Complex-Valued Restricted Boltzmann Machine for Direct Speech Parameterization from Complex Spectra. *ArXiv e-prints*, March 2018.

[9] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac. Matrix Product State Representations. *eprint arXiv:quant-ph/0608197*, August 2006.

[10] A. Rocchetto, E. Grant, S. Strelchuk, G. Carleo, and S. Severini. Learning hard quantum distributions with variational autoencoders. *ArXiv e-prints*, October 2017.

[11] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[12] G. Torlai, G. Mazzola, J. Carrasquilla, M. Troyer, R. Melko, and G. Carleo. Many-body quantum state tomography with neural networks. *ArXiv e-prints*, March 2017.

Appendix A

Project Plan

Justin Jude

Project: Representing quantum states with variational autoencoders

Motivation: The description of a quantum state can be exponential in the number of qubits . Describing quantum states through classical means becomes intractable when the number of qubits is large. To overcome this approximation methods can be used effectively in some cases. Recently machine learning techniques have shown promise in learning approximate representations of large quantum states: 1) https://arxiv.org/abs/1703.05334 and their associated distributions 2) https://arxiv.org/abs/1710.00725. Representing large states is of great relevance to many-body physics and for characterising near term quantum devices.

2) May have advantages over existing methods because encodes states in a deep network. Depth is known to be efficient for many problems. However, so far 2) has only been used to encode quantum distributions and not actual state vectors.

Tasks:

a) Adapt the VAE to learn the distribution of a state on a complete set of bases. (Done)

b) Reconstruct the full state from the learned representation (Currently doing).

c) The VAE encodes the distribution of data in both the encoder and the decoder. Existing work uses the decoder to reconstruct the distribution. Using the encoder has some advantages because it does not rely on sampling. The probability of each data-point (measurement outcome) can be determined directly. In this part of the project you will test and compare the fidelity of the distribution learned by the VAE encoder.

d) TBD

Appendix B

Interim Report

- Name: Justin Jude

- Project Title:
    - Representing quantum states with variational autoencoders

- Current Project Title:
    - Representing quantum states with variational autoencoders and restricted Boltzmann machines

- Internal Supervisor Name: Simone Severini

- <u>Progress made to date</u>

    - Adapt the VAE to learn the distribution of a state on a complete set of bases.

  This has been somewhat achieved although the network architecture needs to be changed and improved in order to much more accurately learn more than one basis. Currently we have an accuracy of around 0.93 for both bases, ideally this would be much closer to 1.0. In order to achieve this I will work on adjusting the network architecture to include one hot vectors to represent the training of different bases in a fashion that more faithfully preserves the two distributions. This will require substantial experimentation.

    - Learn the distribution of a 12 qubit product state on one basis using a Restricted Boltzmann Machine.

  This has been achieved in order to contrast the fidelity of the distribution learnt by the RBM to that learnt by the VAE. Currently we have an accuracy of ~0.984 using the RBM to learn the distribution with 7 nodes in the hidden layer. With some experimentation this should be improved upon. We can vary the hidden layer dimension and vary the learning rate etc. When having equal compression ratios we get:

Performance (overlap) of Models (compression) for each state type

Model	Product State	Random State	Hard State
VAE 1 layer (50%)			
VAE 1 layer (25%)			
VAE 2 layer (50%)			
VAE 2 layer (25%)			
VAE 3 layer (50%)			

VAE 3 layer (25%)			
VAE 4 layer (50%) 22 nodes per layer	0.993		
VAE 4 layer (25%) 14 nodes per layer	0.977		
VAE 5 layer (50%)			
VAE 5 layer (25%)			
Real valued RBM (50%) 34 nodes per layer	0.673		
Real valued RBM (25%) 22 nodes per layer	0.807		

When we compress further, the expectation is that the VAE will continue to outperform the RBM as it can exploit depth efficiency.

- <u>Remaining work to be done before the final report deadline</u>
    - Use the RBM to learn harder states (such as random states and 'hard' states).
    - (If time) Implement a complex valued RBM for the same task
    - Test on a range of VAEs with differing depths.

    - Reconstruct the full state from the learned VAE representation.

  Sample from the complete set of bases to reconstruct the original state. Firstly I will need to sufficiently adapt the VAE to learn the distribution on a complete set of bases. Once a fidelity of ~1.0 has been reached I can move onto this task.

    - (If time) Test and compare the fidelity of the distribution learned by the VAE encoder.

  Using the encoder has some advantages because it does not rely on sampling

    - (If time) Adapt the RBM to Learn the distribution of a state on a complete set of bases

  Similar to the ongoing task with the VAE. This could require more experimentation as the network architecture of the RBM is more rigid.

Appendix C

Code Listing

Listings

C.1 Learning multiple bases 48
C.2 Standard RBM 53
C.3 Deep RBM 58
C.4 Complex valued RBM 62
C.5 RBM Functions 67

color

Listing C.1: Learning multiple bases

```
## VAE learning distributions
"""

 Full_state_4layer.py

 Used in Chapter 5 - Multple bases
 Created on Tue Feb  13 22:26:04 2018

 @author: justinjude

BASED ON ORIGINAL WORK BY EDWARD GRANT
"""
# Imports
from __future__ import division
from __future__ import print_function
import numpy as np
import random
import os
import matplotlib.pyplot as plt
import tensorflow as tf
import os.path
from scipy.stats import ortho_group
import scipy
import itertools
from state_gen_hard import *
from network_tools import *
from state_tools import *
```

```
from state_gen_time_evolution import *
import sys
from qutip import *
sys.dont_write_bytecode = True
cwd = os.getcwd()
from scipy.linalg import hadamard as hdm

# Seed
np.random.seed(0)
tf.set_random_seed(0)
eps = 1e-40

# Get hard distribution https://arxiv.org/abs/1507.05592
# Distribution parameters = L^n^2, qubit number = log2(parameters)
n = 3# Dimension of factoradic for strelchuk distirbution
L = 4 # Sample items for strelchuk distirbution
num_qubits = 6#Number of qubits

def get_product_state(num_qubits):
    state = qutip.rand_ket(2**1)
    state = state.full()
    for i in range(num_qubits-1):
        qubit = qutip.rand_ket(2**1)
        state = np.kron(state,qubit.full())
    return state

state = get_product_state(num_qubits)
hadamard = np.matrix([[1,1],[1,-1]]) * (1/np.sqrt(2))


kron_hadamard = tensor_product(num_qubits,hadamard,hadamard)

p_truth = state.reshape([-1,1])


xxx = np.abs(p_truth)**2
Xxx = xxx/np.sum(xxx)

hd = hdm(2**num_qubits)

zzz = np.matmul(hd, p_truth)

zzz = zzz.tolist()

zzz = np.abs(zzz)**2
zzz = zzz/np.sum(zzz)
```

```
xxx = [item for sublist in xxx for item in sublist]
zzz = [item for sublist in zzz for item in sublist]


# Training config
iterations = 1000000 # Number of training iterations 1000000
display_error_frequency= 1000
# Number of measurements to sample from VAE decoder
num_z_samples = 500000*2**num_qubits
sample_z_frequency = 10000
batch_size = 1000
# Batch size for sampling
batch_size_sampling = 100000
#learning rate
lr = 0.001
# Max weighting for KLD wwarmup https://arxiv.org/pdf/1602.02282.pdf
KLD_weight_max = 0.80


# Network structure
num_layers = 1#Number of layers
compression_rate_schedule = np.array([10.0])
num_params_schedule = compression_rate_schedule*(2**num_qubits)#Total params


# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)

# Flat distribution for benchmarking
p_flat= get_uniform_distribution(num_qubits)
# Benchmark target distributions
overlaps_benchmark_xxx= get_state_overlap(xxx,p_flat)
overlaps_benchmark_zzz= get_state_overlap(zzz,p_flat)
print("Benchmark XXX: " + str(overlaps_benchmark_xxx))
print("Benchmark ZZZ: " + str(overlaps_benchmark_zzz))
np.save(save_loc_benchmarks,overlaps_benchmark_xxx)

plotter(xxx[0:500],p_flat[0:500])
plotter(zzz[0:500],p_flat[0:500])



p_plot = zzz[0:250]
#plotter(p_truth[0:100],p_flat[0:100])
title = "Changed Basis State Distribution"
plt.plot(p_plot)
plt.title(title)
```

```
plt.xlabel('Measurement', fontsize = 15)
plt.ylabel('Probability', fontsize = 15)
#plt.legend(loc='upper left')
plt.ylim([0,np.max(p_plot)])
# Experiment - Overlap by number of params

#KLD warmup https://arxiv.org/pdf/1602.02282.pdf
KLD_weight_schedule = np.linspace(0,KLD_weight_max,iterations)
# Save number of parameters
num_params_keep=np.zeros(len(num_params_schedule))

for i in range(len(num_params_schedule)):
    # Index of overlap calculaution. Get overlap several times during
        training.
    overlap_ind = int(0)
    # Number of decoder parameters
    num_params = num_params_schedule[i]
    num_params_keep[i]=num_params
    np.save(save_loc_params,num_params_keep)
    # Number of nodes for fixed number of network params
    num_nodes = get_node_number(num_qubits,num_layers,num_params)

     # Network config
    input_dim = num_qubits #6
    encoder_dim = 20 #num_nodes
    print(encoder_dim,"encoder_dim")
    latent_dim =  20#num_nodes
    decoder_dim = 20 #num_nodes
    num_layers_encoder = num_layers #1
    num_layers_decoder = num_layers #1

    #TF placeholders
    #basis = tf.placeholder(tf.int32, shape=[len(xxx)*2,])
    x = tf.placeholder("float", shape=[None, input_dim])
    is_training = tf.placeholder(tf.bool)
    KLD_weight = tf.placeholder("float")

    basis = tf.placeholder("float", shape=[None,2])
    # Network
    encoder = make_encoder_network_basis(x,num_qubits,num_layers_encoder,
        encoder_dim,latent_dim,is_training,basis)

    z,mu_encoder,logvar_encoder = make_z_network(encoder,encoder_dim,
        latent_dim,eps)

    decoder = make_decoder_network_basis(z,num_qubits,num_layers_decoder,
        decoder_dim,latent_dim,is_training,basis)
```

```
x_hat_sigmoid =tf.nn.sigmoid(decoder)

# Objectives
BCE = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(logits=
    decoder, labels=x),reduction_indices=1) # Binary cross entropy
KLD = -0.5 * tf.reduce_sum(1.0 + logvar_encoder - tf.square(mu_encoder) -
     tf.exp(logvar_encoder), reduction_indices=1) # KLD regularizer
loss = tf.reduce_mean(BCE + KLD*KLD_weight)

# Update ops
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
    train_step =  tf.train.AdamOptimizer(lr).minimize(loss) # https://
        arxiv.org/abs/1412.6980

# Init TF session
sess = tf.InteractiveSession()

# Iinit TF variables
tf.global_variables_initializer().run()

# Train network
for iteration in range(iterations):
    dist_to_learn = np.random.choice([0, 1])

    if dist_to_learn == 0:
        to_sample = xxx;
        one_hot = [0,1]

    else:
        to_sample = zzz;
        one_hot = [1,0]

    one_hot_rep = np.tile(one_hot, (batch_size,1))
    batch = get_batch2(to_sample,binary_perms,batch_size)
    sess.run([train_step], feed_dict={basis:one_hot_rep, x: batch,
        is_training: True,KLD_weight:KLD_weight_schedule[iteration]})

    if (iteration+1) % display_error_frequency == 0: # Display errors
        BCE_tmp = BCE.eval(feed_dict={basis:one_hot_rep,x:batch,
            is_training:False,KLD_weight:KLD_weight_schedule[iteration]})
             # Get BCE error
        KLD_tmp = KLD.eval(feed_dict={basis:one_hot_rep,x:batch,
            is_training:False,KLD_weight:KLD_weight_schedule[iteration]})
             # Get KLD errror
        display_errors(BCE_tmp,KLD_tmp,iteration,iterations) # Display
            errors
```

```
        if (iteration+1) % sample_z_frequency == 0: # Samples from z
            one_hot_rep = np.tile(one_hot, (batch_size_sampling,1))
            z_sample = np.random.normal(0,1,(batch_size_sampling,latent_dim))
            VAE_out = x_hat_sigmoid.eval(feed_dict={basis:one_hot_rep, z:
                z_sample,is_training:False})
            for sampling_batch_i in range(int(num_z_samples/
                batch_size_sampling-1)):
                z_sample = np.random.normal(0,1,(batch_size_sampling,
                    latent_dim))

                print("Sampling batch " + str(sampling_batch_i)+' of '+str(
                    int(num_z_samples/batch_size_sampling-1)))
                out_tmp =  x_hat_sigmoid.eval(feed_dict={basis:one_hot_rep, z
                    :z_sample,is_training:False})
                VAE_out = np.append(VAE_out,out_tmp,axis=0)
            VAE_out_binary = VAE_out>0.5# Step function for VAE outputs that
                must be 0 or 1
            p_gen = get_dist_from_generated_measurements(VAE_out_binary,
                binary_perms)#Get distribution of measurements

            #overlaps[i,overlap_ind] = get_state_overlap(p_truth,p_gen)#
                Calculate the overlap
            if dist_to_learn == 0:
                overlaps_xxx = get_state_overlap(xxx,p_gen)# Calculate the
                    XXX overlap
                print('XXX Overlap is:' + str(overlaps_xxx))
                p_gen_xxx = p_gen
                plotter(xxx[0:500],p_gen_xxx[0:500])
            else:
                overlaps_zzz = get_state_overlap(zzz,p_gen)# Calculate the
                    ZZZ overlap
                print('ZZZ Overlap is:' + str(overlaps_zzz))
                p_gen_zzz = p_gen
                plotter(zzz[0:500],p_gen_zzz[0:500])

            overlap_ind = overlap_ind + 1 # Increment the index of overlap.
                Many are stored during each train.


    sess.close()
    tf.reset_default_graph()
```

Listing C.2: Standard RBM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
Created on Tue Dec 12 01:07:51 2017

@author: justinjude

STANDARD RBM IMPLEMENTATION
"""

#https://arxiv.org/pdf/1703.05334.pdf
from __future__ import division
from __future__ import print_function
import numpy as np
import random
import os
import matplotlib.pyplot as plt
import tensorflow as tf
import os.path
from scipy.stats import ortho_group
import scipy
import itertools
from state_gen_hard import *
from network_tools import *
from state_tools import *
from state_gen_time_evolution import *
from rbm_tools import *
import sys
from qutip import *
import numpy as np
import pandas as pd
#import msgpack
import glob
import tensorflow as tf
from tensorflow.python.ops import control_flow_ops
from tqdm import tqdm
sys.dont_write_bytecode = True
cwd = os.getcwd()

# Seed
np.random.seed(0)
tf.set_random_seed(0)
eps = 1e-40
dists=[]
dist_names=[]
# location of overlap file
save_loc_overlaps = cwd+'/results_layers_random/overlaps_layers_random'

#load random kets
```

```
num_qubits = 18 #Number of qubits
state = qutip.rand_ket(2**num_qubits)
p_truth = np.abs(state.full())**2
dists.append(p_truth)
dist_names.append('Random')



n = 3# Dimension of factoradic for strelchuk distirbution
L = 4 # Sample items for strelchuk distirbution
num_qubits = int(np.log2(L**n**2))  #Number of qubits 18
p_truth = get_hard_distribution(n,L,mode = 'full')
dists.append(p_truth)
dist_names.append('Hard')

p_truth2=get_product_distribution(num_qubits)
p_truth2=p_truth2/np.sum(p_truth2)
dists.append(p_truth2)
dist_names.append('Product')
# Network structure
num_layers_schedule = np.array([1]) #Number of layers
compression_rate_schedule = np.array([0.5,0.25])
num_params_schedule = compression_rate_schedule*(2**num_qubits)#Total params

# Training config
iterations = 100000# Number of training iterations
display_error_frequency= 1000
num_z_samples = 50*2**num_qubits # NUmber of measurements to samplse from VAE
     decoder
sample_z_frequency = 10000
batch_size = 50000 # Size of training batches
batch_size_sampling = 50000 # Batch size for sampling
lr = 0.001 #learning rate
KLD_weight_max = 0.85 # Max weighting for KLD warmup https://arxiv.org/pdf
    /1602.02282.pdf

# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)

# Store overlaps
overlaps = np.zeros((num_params_schedule.shape[0],num_layers_schedule.shape
    [0],int(iterations/sample_z_frequency))) # Store the best overlaps
    obtained during training

# Flat distribution for benchmarking
p_flat= get_uniform_distribution(num_qubits)
```

```
epochs = 1
visible_dim = num_qubits

# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)
p_gen=[]
#Build Network
VAE_out=[]

for dist in range(0,p_truth.shape[0]):
        p_truth = dists[dist][0:100]
        #plotter(p_truth[0:100],p_flat[0:100])
        title = dist_names[dist] + " State Distribution"
        plt.plot(p_truth)
        plt.title(title)
        plt.xlabel('Measurement', fontsize = 15)
        plt.ylabel('Probability', fontsize = 15)
        #plt.legend(loc='upper left')
        plt.ylim([0,np.max(p_truth)])

        plt.savefig(title)
        plt.show()
with tf.Session() as sess:
    for dist in range(0,p_truth.shape[0]):
        p_truth = dists[dist]
        print('Overlap is:' + str(get_state_overlap(p_truth,p_flat)))
        plotter(p_truth[0:100],p_flat[0:100])
        for ii in range(num_params_schedule.shape[0]):
            visible_dim = num_qubits
            num_params  = num_params_schedule[ii]
            num_nodes = get_node_number(num_qubits,1,num_params) # Number of
                nodes for fixed number of network params
            hidden_dim = num_nodes
            print("Hidden Dim:", hidden_dim)
            x = tf.placeholder(tf.float32, [None,visible_dim], name = "x") #
                placeholder for input
           W = tf.Variable(tf.random_normal([visible_dim,hidden_dim], (2/
                visible_dim)), name = "W") #weight matrix between visible and
                 hidden layer
            bh = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "bh
                ")) #bias vector for hidden layer
            bv = tf.Variable(tf.zeros([1, visible_dim], tf.float32, name = "
                bv"))#bias vector for visible layer

            x_sample = gibbs_sample(1,x,W,bh,bv)
            h = sample_bin(tf.sigmoid(tf.matmul(x, W) + bh))
            h_sample = sample_bin(tf.sigmoid(tf.matmul(x_sample, W) + bh))
```

```
#Contrastive Divergence
size_bt = tf.cast(tf.shape(x)[0], tf.float32)
W_adder  = tf.multiply(lr/size_bt, tf.subtract(tf.matmul(tf.
    transpose(x), h), tf.matmul(tf.transpose(x_sample), h_sample)
    ))
bv_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(x,
    x_sample), 0, True))
bh_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(h,
    h_sample), 0, True))

updt = [W.assign_add(W_adder), bv.assign_add(bv_adder), bh.
    assign_add(bh_adder)]


#First, we train the model
#initialize the variables of the model
init = tf.global_variables_initializer()
sess.run(init)
for epoch in range(epochs):
    for iteration in tqdm(range(iterations)):
        if dist == 0:
            batch = get_batch_full_state(p_truth,binary_perms,
                batch_size)
        else:
            batch = get_batch2(p_truth,binary_perms,batch_size)
        #tr_x = song[i:i+batch_size]
        sess.run([updt], feed_dict={x: batch})
        if (iteration+1)  == iterations: # Samples from z
           VAE_out = gibbs_sample(1,x,W,bh,bv).eval(session=sess,
                feed_dict={x: np.zeros((batch_size_sampling,
               visible_dim))})
           for sampling_batch_i in range(int(num_z_samples/
               batch_size_sampling -1)):
               print("Sampling batch " + str(sampling_batch_i)+'
                   of '+str(int(num_z_samples/batch_size_sampling
                   -1)))
               sample = gibbs_sample(1,x,W,bh,bv).eval(session=
                   sess, feed_dict={x: np.zeros((
                   batch_size_sampling, visible_dim))})
               VAE_out = np.append(VAE_out,sample,axis=0)
           VAE_out_binary = VAE_out>0.5# Step function for VAE
               outputs that must be 0 or 1
           p_gen = get_dist_from_generated_measurements(
               VAE_out_binary,binary_perms)#Get distribution of
               measurements
```

```
            print('Overlap_is:' + str(get_state_overlap(p_truth,p_gen)))
            plotter2(p_truth[0:100],p_gen[0:100],1,compression_rate_schedule[
                ii],dist_names[dist],get_state_overlap(p_truth,p_gen))
```

Listing C.3: Deep RBM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 12 01:07:51 2017

@author: justinjude

DEEP RBM IMPLEMENTATION
"""

#https://arxiv.org/pdf/1703.05334.pdf
from __future__ import division
from __future__ import print_function
import numpy as np
import random
import os
import matplotlib.pyplot as plt
import tensorflow as tf
import os.path
from scipy.stats import ortho_group
import scipy
import itertools
from state_gen_hard import *
from network_tools import *
from state_tools import *
from state_gen_time_evolution import *
from rbm_tools import *
import sys
from qutip import *
import numpy as np
import pandas as pd
#import msgpack
import glob
import tensorflow as tf
from tensorflow.python.ops import control_flow_ops
from tqdm import tqdm
sys.dont_write_bytecode = True
cwd = os.getcwd()

# Seed
np.random.seed(0)
```

```
tf.set_random_seed(0)
eps = 1e-40
dists=[]
dist_names=[]
# location of overlap file
save_loc_overlaps = cwd+'/results_layers_random/overlaps_layers_random'

#load random kets
num_qubits = 9 #Number of qubits
state = qutip.rand_ket(2**num_qubits)
p_truth = np.abs(state.full())**2
dists.append(p_truth)
dist_names.append('Random')



n = 3# Dimension of factoradic for strelchuk distirbution
L = 2 # Sample items for strelchuk distirbution
num_qubits = int(np.log2(L**n**2))  #Number of qubits 18
p_truth = get_hard_distribution(n,L,mode = 'full')
dists.append(p_truth)
dist_names.append('Hard')

p_truth2=get_product_distribution(num_qubits)
p_truth2=p_truth2/np.sum(p_truth2)
dists.append(p_truth2)
dist_names.append('Product')
# Network structure
num_layers_schedule = np.array([1]) #Number of layers
compression_rate_schedule = np.array([0.5,0.25])
num_params_schedule = compression_rate_schedule*(2**num_qubits)#Total params

# Training config
iterations = 1000# Number of training iterations
display_error_frequency= 1000
num_z_samples = 5000*2**num_qubits # NUmber of measurements to samplse from
    VAE decoder
sample_z_frequency = 10000
batch_size = 5000 # Size of training batches
batch_size_sampling = 50000 # Batch size for sampling
lr = 0.001 #learning rate
KLD_weight_max = 0.85 # Max weighting for KLD warmup https://arxiv.org/pdf
    /1602.02282.pdf

# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)
```

```
# Store overlaps
overlaps = np.zeros((num_params_schedule.shape[0],num_layers_schedule.shape
    [0],int(iterations/sample_z_frequency))) # Store the best overlaps
    obtained during training
#overlaps_benchmark = np.zeros(num_params_schedule.shape[0]) # Overlap
    between target state and maximally mixed (flat) state

# Flat distribution for benchmarking
p_flat= get_uniform_distribution(num_qubits)

# Training config
#iterations = 100000 # Number of training iterations 1000000
#batch_size = 100 # Size of training batches 500
#batch_size_sampling = 1000000 # Batch size for sampling
#lr = 0.005 #learning rate
epochs = 1
visible_dim = num_qubits
#hidden_dim = 22 #7 in the hidden layer seems to be optimal regardless of
    num_qubits
# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)
p_gen=[]
#Build Network
VAE_out=[]
with tf.Session() as sess:
    for dist in range(1,p_truth.shape[0]):
        p_truth = dists[dist]
        print('Overlap is:' + str(get_state_overlap(p_truth,p_flat)))
        plotter(p_truth[0:500],p_flat[0:500])
        for ii in range(num_params_schedule.shape[0]):
            visible_dim = num_qubits
            num_params  = num_params_schedule[ii]
            num_nodes = get_node_number(num_qubits,1,num_params) # Number of
                nodes for fixed number of network params
            hidden_dim = num_nodes
            print("Hidden Dim:", hidden_dim)
            x = tf.placeholder(tf.float32, [None,visible_dim], name = "x") #
                placeholder for input
            W = tf.Variable(tf.random_normal([visible_dim,hidden_dim], (2/
                visible_dim)), name = "W") #weight matrix between visible and
                 hidden layer
            W2 = tf.Variable(tf.random_normal([hidden_dim,hidden_dim], (2/
                hidden_dim)), name = "W2") #weight matrix between visible and
                 hidden layer

            bh = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "bh
                ")) #bias vector for hidden layer
```

```
bh2 = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "
    bh2")) #bias vector for hidden layer
bv = tf.Variable(tf.zeros([1, visible_dim], tf.float32, name = "
    bv"))#bias vector for visible layer

x_sample = gibbs_sample_deep(10,x,W,W2,bh,bh2,bv)

l1 = sample_bin(tf.sigmoid(tf.matmul(x, W) + bh))
l1_sample = sample_bin(tf.sigmoid(tf.matmul(x_sample, W) + bh))

h = sample_bin(tf.sigmoid(tf.matmul(l1, W2) + bh2))
h_sample = sample_bin(tf.sigmoid(tf.matmul(l1_sample, W2) + bh2))

#Next, we update the values of W, bh, and bv, based on the
    difference between the samples that we drew and the original
    values
size_bt = tf.cast(tf.shape(x)[0], tf.float32)
W_adder  = tf.multiply(lr/size_bt, tf.subtract(tf.matmul(tf.
    transpose(x), l1), tf.matmul(tf.transpose(x_sample),
    l1_sample)))
W2_adder  = tf.multiply(lr/size_bt, tf.subtract(tf.matmul(tf.
    transpose(l1), h), tf.matmul(tf.transpose(l1_sample),
    h_sample)))
bv_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(x,
    x_sample), 0, True))
bh_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(l1,
    l1_sample), 0, True))
bh2_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(h,
    h_sample), 0, True))
#When we do sess.run(updt), TensorFlow will run all 3 update
    steps
updt = [W.assign_add(W_adder),W2.assign_add(W2_adder), bv.
    assign_add(bv_adder), bh.assign_add(bh_adder), bh2.assign_add
    (bh2_adder)]


#First, we train the model
#initialize the variables of the model
init = tf.global_variables_initializer()
sess.run(init)
for epoch in range(epochs):
    for iteration in tqdm(range(iterations)):
        if dist == 0:
            batch = get_batch_full_state(p_truth,binary_perms,
                batch_size)
        else:
            batch = get_batch2(p_truth,binary_perms,batch_size)
```

```
                    #tr_x = song[i:i+batch_size]
                    sess.run([updt], feed_dict={x: batch})
                    if (iteration+1) == iterations: # Samples from z
                        VAE_out = gibbs_sample_deep(10,x,W,W2,bh,bh2,bv).eval(
                            session=sess, feed_dict={x: np.zeros((
                            batch_size_sampling, visible_dim))})
                        for sampling_batch_i in range(int(num_z_samples/
                            batch_size_sampling-1)):
                            print("Sampling batch " + str(sampling_batch_i)+' 
                                of '+str(int(num_z_samples/batch_size_sampling
                                -1)))
                            sample = gibbs_sample_deep(10,x,W,W2,bh,bh2,bv).
                                eval(session=sess, feed_dict={x: np.zeros((
                                batch_size_sampling, visible_dim))})
                            VAE_out = np.append(VAE_out,sample,axis=0)
                        VAE_out_binary = VAE_out>0.5# Step function for VAE
                            outputs that must be 0 or 1
                        p_gen = get_dist_from_generated_measurements(
                            VAE_out_binary,binary_perms)#Get distribution of
                            measurements

            print('Overlap is:' + str(get_state_overlap(p_truth,p_gen)))
            plotter2(p_truth[0:100],p_gen[0:100],1,compression_rate_schedule[
                ii],dist_names[dist],get_state_overlap(p_truth,p_gen))
```

Listing C.4: Complex valued RBM

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Mar  6 22:26:04 2018

@author: justinjude

COMPLEX RBM IMPLEMENTATION

"""
#https://arxiv.org/pdf/1703.05334.pdf
from __future__ import division
from __future__ import print_function
import numpy as np
import random
import os
import matplotlib.pyplot as plt
import tensorflow as tf
import os.path
from scipy.stats import ortho_group
```

```
import scipy
import itertools
from state_gen_hard import *
from network_tools import *
from state_tools import *
from state_gen_time_evolution import *
from rbm_tools import *
import sys
from qutip import *
import numpy as np
import pandas as pd
#import msgpack
import glob
import tensorflow as tf
from tensorflow.python.ops import control_flow_ops
from tqdm import tqdm
sys.dont_write_bytecode = True
cwd = os.getcwd()

# Seed
np.random.seed(0)
tf.set_random_seed(0)
eps = 1e-40
dists=[]
dist_names=[]
# location of overlap file
save_loc_overlaps = cwd+'/results_layers_random/overlaps_layers_random'

#load random kets
num_qubits = 8 #Number of qubits
rnd_state = qutip.rand_ket(2**num_qubits)
rnd_state = rnd_state.full()
p_truth = np.abs(rnd_state)**2
dists.append(p_truth)
dist_names.append('Random')



n = 2 # Dimension of factoradic for strelchuk distirbution
L = 4 # Sample items for strelchuk distirbution
param = L**n**2
num_qubits = 8  #Number of qubits 12
p_truth = get_hard_distribution(n,L,mode = 'full')
dists.append(p_truth)
dist_names.append('Hard')

p_truth2=get_product_distribution(num_qubits)
```

```
print(p_truth2)
p_truth2=p_truth2/np.sum(p_truth2)
dists.append(p_truth2)
dist_names.append('Product')
# Network structure
num_layers_schedule = np.array([1]) #Number of layers
compression_rate_schedule = np.array([0.5,0.25])
num_params_schedule = compression_rate_schedule*(2**num_qubits)#Total params

# Training config
iterations = 10000 # Number of training iterations
display_error_frequency= 1000
num_z_samples = 5000*2**num_qubits # NUmber of measurements to samplse from
    VAE decoder
sample_z_frequency = 10000
batch_size = 500 # Size of training batches
batch_size_sampling = 50000 # Batch size for sampling
lr = 0.001 #learning rate
KLD_weight_max = 0.85 # Max weighting for KLD warmup https://arxiv.org/pdf
    /1602.02282.pdf

# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)

# Store overlaps
overlaps = np.zeros((num_params_schedule.shape[0],num_layers_schedule.shape
    [0],int(iterations/sample_z_frequency))) # Store the best overlaps
    obtained during training
#overlaps_benchmark = np.zeros(num_params_schedule.shape[0]) # Overlap
    between target state and maximally mixed (flat) state

# Flat distribution for benchmarking
p_flat= get_uniform_distribution(num_qubits)

epochs = 1
visible_dim = num_qubits
# Get possible measurement outcomes
binary_perms = get_binary_permutations(num_qubits)
#Build Network
VAE_out=[]
with tf.Session() as sess:
    for dist in range(0,p_truth.shape[0]):

        p_truth = rnd_state
        p_real = p_truth.real ** 2
        p_real = p_real/np.sum(p_real)
        p_imag = p_truth.imag ** 2
```

```
p_imag = p_imag/np.sum(p_imag)
p_truth = dists[dist]
print('Overlap is:' + str(get_state_overlap(p_truth,p_flat)))
plotter(p_truth[0:100],p_flat[0:100])
for ii in range(num_params_schedule.shape[0]):
    visible_dim = num_qubits
    num_params  = num_params_schedule[ii]
    num_nodes = get_node_number(num_qubits,1,num_params) # Number of
        nodes for fixed number of network params
    hidden_dim = num_nodes
    print("Hidden Dim:", hidden_dim)
    #REAL PART
    xr = tf.placeholder(tf.float32, [None,visible_dim], name = "xr")
        #placeholder for input
    Wr = tf.Variable(tf.random_normal([visible_dim,hidden_dim], 0.01)
        , name = "Wr") #weight matrix between visible and hidden
        layer
    bh = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "bh
        ")) #bias vector for hidden layer
    bvr = tf.Variable(tf.zeros([1, visible_dim], tf.float32, name = "
        bvr"))#bias vector for visible layer

    #IMAGINARY PART
    Wi = tf.Variable(tf.random_normal([visible_dim,hidden_dim], 0.01)
        , name = "Wi") #weight matrix between visible and hidden
        layer
    bvi = tf.Variable(tf.zeros([1, visible_dim], tf.float32, name = "
        bvi"))#bias vector for visible layer
    xi = tf.placeholder(tf.float32, [None,visible_dim], name = "xi")
        #placeholder for input


    xr_sample = gibbs_sample(1,xr,Wr,bh,bvr)
    xi_sample = gibbs_sample(1,xi,Wi,bh,bvi) #sample from RBM using
        unit Gaussian

    hr = sample_bin(tf.sigmoid(tf.matmul(xr, Wr) + bh))
    hi = sample_bin(tf.sigmoid(tf.matmul(xi, Wi) + bh))
    #comp_to_real = tf.rt(tf.sq(xr) + tf.sq(xi))


    hr_sample = sample_bin(tf.sigmoid(tf.matmul(xr_sample, Wr) + bh))
    hi_sample = sample_bin(tf.sigmoid(tf.matmul(xi_sample, Wi) + bh))
    #comp_to_real = tf.rt(tf.sq(xr_sample) + tf.sq(xi_sample))


    #Contrastive Divergence
```

```
size_bt = tf.cast(tf.shape(xr)[0], tf.float32)
Wr_adder  = tf.multiply(lr/size_bt, tf.subtract(tf.matmul(tf.
    transpose(tf.conj(xr)), hr), tf.matmul(tf.transpose(tf.conj(
    xr_sample)), hr_sample)))
Wi_adder  = tf.multiply(lr/size_bt, tf.subtract(tf.matmul(tf.
    transpose(tf.conj(xi)), hi), tf.matmul(tf.transpose(tf.conj(
    xi_sample)), hi_sample)))

bvr_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(xr,
     xr_sample), 0, True))
bvi_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(xi,
     xi_sample), 0, True))
bh_adder = tf.multiply(lr/size_bt, tf.reduce_sum(tf.subtract(hr +
     hi, hr_sample + hi_sample), 0, True))
#When we do sess.run(updt), TensorFlow will run all 3 update
    steps
updt = [Wr.assign_add(Wr_adder),Wi.assign_add(Wi_adder), bvr.
    assign_add(bvr_adder), bh.assign_add(bh_adder),bvi.assign_add
    (bvi_adder)]


#First, we train the model
#initialize the variables of the model
init = tf.global_variables_initializer()
sess.run(init)
for epoch in range(epochs):
    for iteration in tqdm(range(iterations)):
        #if dist == 0:
        batch_real = get_batch_full_state(p_real,binary_perms,
            batch_size)
        batch_imag = get_batch_full_state(p_imag,binary_perms,
            batch_size)
        #else:
            #batch = get_batch2(p_truth,binary_perms,batch_size)
        #tr_x = song[i:i+batch_size]
        sess.run([updt], feed_dict={xr: batch_real, xi:
            batch_imag})
        if (iteration+1) == iterations: # Samples from z
            s_im = gibbs_sample(1,xi,Wi,bh,bvi).eval(session=sess,
                 feed_dict={xi: np.zeros((batch_size_sampling,
                visible_dim))})
            s_re = gibbs_sample(1,xr,Wr,bh,bvr).eval(session=sess,
                 feed_dict={xr: np.zeros((batch_size_sampling,
                visible_dim))})
            VAE_out = np.sqrt(s_im**2 + s_re**2)
            for sampling_batch_i in range(int(num_z_samples/
                batch_size_sampling -1)):
```

```
                        print("Sampling batch " + str(sampling_batch_i)+' of '+str(int(num_z_samples/batch_size_sampling-1)))
                        s_im = gibbs_sample(1,xi,Wi,bh,bvi).eval(session=sess, feed_dict={xi: np.zeros((batch_size_sampling, visible_dim))})
                        s_re = gibbs_sample(1,xr,Wr,bh,bvr).eval(session=sess, feed_dict={xr: np.zeros((batch_size_sampling, visible_dim))})
                        out = np.sqrt(np.square(s_im) + np.square(s_re))

                        VAE_out = np.append(VAE_out,out,axis=0)
                    VAE_out_binary = VAE_out>0.5# Step function for VAE outputs that must be 0 or 1
                    p_gen = get_dist_from_generated_measurements(VAE_out_binary,binary_perms)#Get distribution of measurements

            print('Overlap is:' + str(get_state_overlap(p_truth,p_gen)))
            plotter2(p_truth[0:100],p_gen[0:100],1,compression_rate_schedule[ii],dist_names[dist],get_state_overlap(p_truth,p_gen))
```

Listing C.5: RBM Functions

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 25 12:57:15 2018

@author: justinjude

RBM FUNCTIONS
"""
import tensorflow as tf

def sample_rbm(probs):
    #Takes in a vector of probabilities, and returns a random vector of 0s and 1s sampled from the input vector
    return tf.floor(probs + tf.random_normal(tf.shape(probs), 0, 1))
def sample_bin(probs):

    return tf.floor(probs + tf.random_uniform(tf.shape(probs), 0, 1))




def gibbs_sample(k,x,W,bh,bv):
    #Repeatedly sample (k times) from the visible and hidden layer of the RBM
```

```
        defined by W, bh, bv
    def gibbs_step(count, k, xk):
        #Runs a sampling step. The visible values start as xk
        hk = sample_bin(tf.sigmoid(tf.matmul(xk, W) + bh)) #Forward pass
            using the visible values to sample the hidden values
        xk = sample_rbm(tf.sigmoid(tf.matmul(hk, tf.transpose(W)) + bv)) #
            Backwards pass using the hidden values to sample the visible
            values
        return count+1, k, xk
    #Run sampling step for k iterations
    ct = tf.constant(0) #counter

    def c(ct,k,x):
        return tf.less(ct, k)

    def body(ct,k,x):


        return gibbs_step(ct,k,x)

    [_, _, x_sample] = tf.while_loop(c, body, [ct,tf.constant(k), x])
    x_sample = tf.stop_gradient(x_sample)
    return x_sample

def gibbs_sample_deep(k,x,W,W2,bh,bh2,bv):
    #Repeatedly sample (k times) from the visible, hidden layer1  and 2 of
        the RBM defined by W,W2,bh,bh2,bv
    def gibbs_step(count, k, xk):
        #Runs a sampling step. The visible values start as xk
        l1 = sample_bin(tf.sigmoid(tf.matmul(xk, W) + bh))
        hk = sample_bin(tf.sigmoid(tf.matmul(l1, W2) + bh2)) #Forward pass
            using the visible values to sample the hidden values

        back_l1 = sample_rbm(tf.sigmoid( tf.matmul(hk, tf.transpose(W2)) + bh
            ))
        xk = sample_rbm(tf.sigmoid(tf.matmul(back_l1, tf.transpose(W)) + bv))
             #Backwards pass using the hidden values to sample the visible
            values
        return count+1, k, xk

    #Run gibbs steps for k iterations
    ct = tf.constant(0) #counter

    def c(ct,k,x):
        return tf.less(ct, k)

    def body(ct,k,x):
```

```
        return gibbs_step(ct,k,x)

    [_, _, x_sample] = tf.while_loop(c, body, [ct,tf.constant(k), x])

    x_sample = tf.stop_gradient(x_sample)
    return x_sample
```