Information Compression Of Molecular Representations Using Neural Network Auto-Encoders

By

Andrew Cameron

University of Prince Edward Island

Charlottetown, PE, Canada

A thesis submitted in partial fulfillment of

the requirements for the Honours Programme

in the Department of Physics

This Thesis is Approved

Signature (Supervisor)	Date
Signature (Second Reader)	Date

This Thesis is Accepted

Signature (Dean of Science)	Date

Abstract

As quantum chemistry continues to make stronger predictions about the physical observables of chemical systems, and larger datasets of quantum chemical data become public, more and more theorists are employing machine learning algorithms to make the predictions. Machine learning algorithms are limited by the quality of the data they are given. The current molecular representations used to train machine learning algorithms contain unnessecary information which make it difficult for data-driven methods to make accurate predictions. An attempt to overcome this issue was made by compressing molecular representations with neural network auto-encoders.

Three representations were evaluated (Cartesian coordinates, Coulomb Matrices, and Position Intracules) using auto-encoders in order to optimize reproducibility. Cartesian coordinates of 134,000 small organic molecules were compressed with an average in-sample reproducibility error of 0.4982 Å per coordinate and an information compression of 39.08%. Coulomb matrices of the same 134,000 molecules were compressed with an average in-sample reproducibility error of 0.2414Å^{-1} per matrix element and an information compression of 93.94%. Position intracules of 21,271 small organic molecules were compressed with an average in-sample reproducibility error of 0.8926 and an information compression of 29.30%.

Acknowledgements

It was an absolute honour to work with Trevor Profitt, Adam Proud, and Mat Larade over the past three years. This thesis is the largest academic venture I have completed at this point in my career, and you three have played a pivotal roll in pushing me to this point. For this, I thank you.

To my co-supervisors, Dr. Jason Pearson and Dr. Derek Lawther, thank you for your guidance both in the honours research project and in my developement as a scientist in my undergraduate schooling.

Finally, to my parents, Derrick and Sharon Cameron, thank you for the immeasurable support you have provided me in the 21 years leading up to graduation. You have always been my biggest supporters, fans, and source of inspiration.

Contents

List of Figures

List of Tables

1 Molecular Representations in Computational Chemistry

Understanding the differences in molecular structure between different compounds is pivotal in explaining scientific phenomena in many fields of science. It allows scientists to relate the properties of a molecule to its structure and serves as a starting point for exploring the behaviour of the molecule.

As computers present the opportunity to work with large amounts of data, the exploration of molecular properties and behaviour can be attempted on a larger scale than previously possible. This calls for the need to appropriately represent molecular structure in order to facilitate calculations, and the exact means of digital representation is crucial in order to succeed because data-driven algorithms (such as different forms of machine learning) depend heavily on the quality and representation of input data. Digital molecular representation varies in different areas of science to suit the specific requirements of each situation in theory by providing enough information to correctly model the problem of interest. It is not in the scope of this thesis to go into detail on every molecular representation in every field of science, but a few will be highlighted below.

For the *chemoinformatics* associated with drug discovery, fingerprint vectors are commonly used to characterize a compound. These fingerprint vectors are made up of binary, categorical, or numeric descriptors which give information about the chemical structure. They provide a way of storing and searching large numbers of complicated compounds digitally, thereby enabling scientists to keep track of potential drug structures.[1–6]

In the field of *protein structure prediction*, one important area of understanding is post-translational modifications of proteins. This is relevant for research in biology and medicine, and databases exist to collect information about the post-translational modifications. In this field, the molecular representation consists of a list of positions of interest, such as the positions of experimentally determined phosphorylation sites. Details regarding the specific kinases which produce the modifications are also included in the representation.[7,8]

This thesis will focus on molecular representations for use in *computational chemistry*. Sections 1.1 and 1.2 discuss Quantum Mechanics and Machine Learning as two approaches for predicting physical observables of molecules. Sections 1.3 and 1.4 discuss the basic requirements of molecular representations and compares different representations currently in use. Sections 1.5 and 1.6 introduce the concept of a machine-learned molecular representation and the strategy for evaluating its performance. These are the primary objectives of this thesis.

1.1 Prediction of Physical Observables using Quantum Mechanics

Of the many approaches to computational chemistry, *ab initio* methods based on the theory of quantum mechanics are the most rigorous. Classical mechanics is based on Newton's theories and formalization of physics.[9] In the late 1800s and early 1900s, it started to become apparent that this theory no longer predicted motion and energy correctly as the size of the system of interest decreased. When studying atoms and molecules, classical mechanics is lacking in theoretical rigor and cannot be used as a sufficient mathematical framework.

Quantum mechanics was the most widely accepted theory to replace classical mechanics in microscopic systems.[10] In this theory, the state of a system and all of its electronic properties are expressed by a mathematical function called the wavefunction. This function is denoted $\Psi(\mathbf{r}, \mathbf{R})$, where $\mathbf{r}$ and $\mathbf{R}$ are the spatial and spin coordinates of the electrons and nuclei, respectively, at a single moment in time.[11]

The Schrödinger equation encompasses the essence of quantum mechanics and is analogous to Newton's second law in classical mechanics. Both equations serve to predict outcomes of a system given initial values, and both are second order differential equations. One of the largest differences of the two equations illustrates a key difference in the theories: Newton's second law predicts an outcome with full certainty, whereas Schrödinger's equation predicts the probability of an outcome. This equation can be written in its time-independent form seen as

$$\hat{H}\Psi(\mathbf{r},\mathbf{R}) = E\Psi(\mathbf{r},\mathbf{R}) \tag{1.1}$$

where $\hat{H}$ is the Hamiltonian operator and E is the energy of the system. The Hamiltonian is an example of a quantum mechanical operator. If equation 1.1 is solved with the Hamiltonian acting on the wavefunction, then the eigenvalue will be the energy. Additionally, the probability of finding a value, or a range of values, of any physical observable is attainable. For each physical observable, there exists a Hermitian operator, which satisfies the time-independent Schrödinger equation.[12] Therefore, if the wavefunction of a system is known, quantum mechanics provides the tools to predict physical observables. In practice, this function is only exactly known for one natural system: the hydrogen atom. Scientists who work in computational chemistry often try to approximate the wavefunction of other atoms and molecules with a large variety of approaches; each with varying degrees of accuracy and computational cost.

Ab initio computational methods have a trade-off between accuracy and time required to complete the calculation. Accurate predictions can scale as poorly as k^7, where k is the number of basis functions (mathematical descriptions of the occupied electron orbitals). This scaling prohibits the use of these methods for most of chemical space.

1.2 Machine Learning as an Alternative to Quantum Mechanics

In recent years there has been significant progress in designing atomistic potentials with machine learning techniques.[13] These techniques are used across many areas of science where there are large amounts of data and a decision or prediction is to be made. This has become relevant in computational chemistry as databases containing quantum chemical information become greater in number and size.

Machine learning is a completely different strategy compared to conventional quantum mechanical methods as no work is done to approximate the wavefunction of the atoms. The overall goal remains the same however: map the structure of a chemical system to the physical observables of that system. Pictorally, this concept can be represented by Figure 1.1.

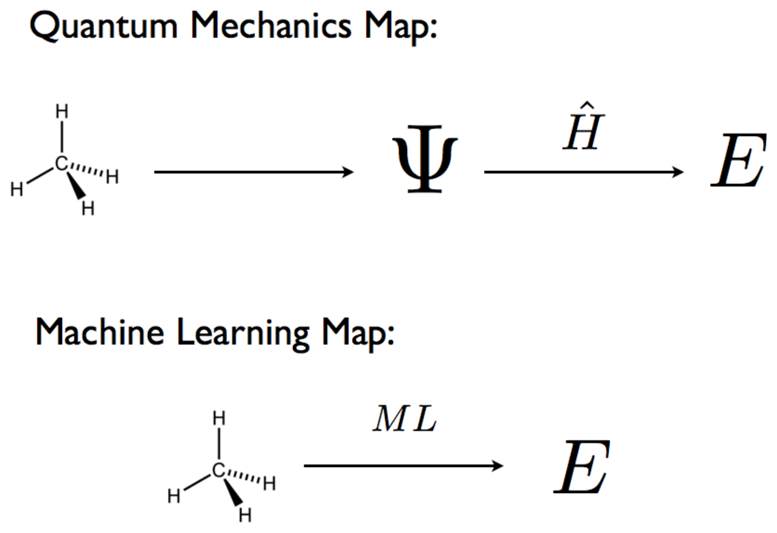

Figure 1.1: Conceptual mapping differences between quantum mechanics and machine learning. The important detail to highlight is that the beginning and end are conceptually identical.

Quantum mechanics is built on theoretical relationships and models which surround the structure of the atoms. Structure determines $\hat{H}$, which determines Ψ, which determines all observable properties. Therefore, the structure is all that is needed to predict properties. Knowing that some function must map structure to physical properties, machine learning algorithms - specifically neural networks - can certainly approximate the function, given enough data. Hornik et al. rigorously proved that any continuous function can be approximated by neural networks, and they classify these algorithms as "universal approximators".[14,15]

With large amounts of data to train an algorithm, the machine learning model can "learn" from the data to discover empirical relationships. The nature of the relationship between structure and physical observables is not specified *a priori* as these algorithms are purely data-driven.[16] Therefore, the algorithms are incredibly susceptible to a change in data representation. For example, if there is more than one way to represent the same molecule, this can be detrimental for machine learning algorithms as they attempt to learn the mapping from structure to physical observable.

1.3 Molecular Representation Requirements

1.3.1 Translational and Rotational Invariarance

In order for a digital molecular representation to be suitable for maximizing the efficiency and ability of machine learning algorithms, translational and rotational invariance are key requirements.[16]

If the molecular representation of a system remains unchanged when the molecule is translated in space, the representation is said to have translational invariance. A common way to accomplish this is to define the coordinates of an atom in a way that references other atoms - not an arbitrarily chosen origin. This way, when a translation occurs, the relative positions do not change and the representation is therefore translationally invariant.

Rotational invariance is a property of a molecular representation which remains unchanged after a system undergoes a rotation. It can be achieved in a similar manner to translational invariance since the relative distances between atoms do not change during a rotation.

Together, these two properties of a molecular representation are important if machine learning is to later be done on the data because rotations and translations do not change the physical observables of the system. Therefore, if the physical observables do not change and the representation does, it is difficult for a machine learning algorithm to discover the pattern of the relationship between the two.

1.3.2 Uniqueness

Computer input can be in the form of a single valued object, an array, or a matrix. If the digital representation of a molecule is given by a matrix, it is said to be permutation variant if there is only one way to order the columns and the rows.[17] A molecular representation should be unique. That is, if a representation is defined in a way where interchanging the columns or rows does not change the actual molecule, it is not said to be unique.

Much like translational and rotational invariance, the lack of permutation variance is detrimental to the ability of a machine learning algorithm to predict physical observables by learning from data. If a molecular representation is defined in a way that provides multiple possibilities to represent the same chemical system, machine learning will be impeded. For this reason, a unique representation should be defined for every molecule.

1.3.3 Information Density

For machine learning algorithms to effectively recognize the pattern which relates the structure of a molecule to certain physical properties, the data specifying the structural information must be presented clearly and concisely. This will make the pattern most evident and lead to effective learning. In this thesis, the term information density will be used to quantify how concisely the data is represented. To illustrate information density with an example, consider a function $f(x, y)$ that is the input vector for a calculation. If this function is integrated with respect to x, namely

$$g(y) = \int f(x,y)dx$$

and $g(y)$ is a suitable input vector as well, then $g(y)$ is said to have a higher information density.

For machine learning purposes, maximizing information density will lead to more accurate predictions, faster training, less critical failures, and increased generalizability.

1.4 Types of Molecular Representations

1.4.1 Cartesian Coordinates

Providing the Cartesian coordinates of a molecule is a common representation which lists the atoms and their positions relative to an arbitrarily chosen origin. This is a conceptually simple representation and provides a clear picture of the molecule when graphed in a three dimensional plot. Cartesian coordinates are the most prevalent molecular representations used in computational chemistry. The molecular input for most quantum chemistry packages is enclosed in an ".xyz" file which includes the number of atoms, the chemical symbol for each atom, and three spatial coordinates for each atom.

Cartesian coordinates are a poor candidate for input to machine learning algorithms, however, because the coordinates are defined relative to an arbitrarily chosen origin; the representation is not translationally or rotationally invariant. Moving the system relative to the origin changes all coordinates without changing the molecule itself.

This representation also has an ordering problem. There is no standard for which atoms should come first in the array. Therefore Cartesian coordinates are not permutation variant.

Additionally, there is no way to model excited or charged states of molecules using Cartesian coordinates. This is because information about the electrons is not explicitly given.

1.4.2 Coulomb Matrices

Attempts to create a numerical representation that is more suited for machine learning input than Cartesian coordinates have already been successful to a certain extent. Rupp et al. have recently designed a matrix representation for chemical systems which they refer to as the Coulomb Matrix.[18] The elements of a Coulomb matrix, $\mathbf{M}$, are given as

$$M_{i,j} = \begin{cases} 0.5Z_i^{2.4} & \text{if } i = j \\ \frac{Z_i Z_j}{|\boldsymbol{R_i} - \boldsymbol{R_j}|} & \text{if } i \neq j \end{cases} \tag{1.2}$$

where $\boldsymbol{R_i}$ are the atomic coordinates of atom i, and Z_i is the nuclear charge of atom i. This definition states that the off-diagonal elements correspond to Coulomb repulsion between atoms i and j, and diagonal elements correspond to atomic energies given the charges of the nuclei.

The Coulomb matrix is a translationally and rotationally invariant representation because it is defined in terms of relative atomic coordinates. This gives the distance between atoms regardless of the choice of origin. When a system is translated or rotated in space, the relative distances between the atoms do not change and the representation remains the same.

One issue with this representation is that it is not permutation variant. Each row and column gives information for a single atom, therefore switching two columns or two rows gives an equally valid Coulomb matrix for the same molecule. By permuting rows and columns, one can obtain up to $d!$ different Coulomb matrices where d is the number of atoms in the system. The following section will highlight some variations of the Coulomb matrix which attempt to make a permutation invariant representation.

Another problem with Coulomb matrices is that the size of the matrix depends on d. In any vector space model, a constant dimensionality for the input is desired.[16] In the literature this problem is overcome on a case by case basis by having all Coulomb matrices have the same dimensionality. To avoid excluding any molecules, the largest molecule in the database would have to be chosen as the model for input dimensionality. All other Coulomb matrices would be filled with zeros in the extra rows and columns.

Similar to Cartesian coordinates, Coulomb matrices also fail to represent charged molecules or excited states. Both of these representations are built with nuclei as the focus, not the electrons.

1.4.3 Representations Derived from the Coulomb Matrix

Rupp et al. and Hansen et al. have proposed three candidate representations, which are all derived from the Coulomb matrix, with aims of fixing the ambiguity problem of the atomic ordering.[16,19]

The eigenspectrum representation considers the eigenvalue problem $\boldsymbol{M}\boldsymbol{v} = \lambda\boldsymbol{v}$. When this problem is solved, d eigenvalues will be obtained from a $(d \times d)$-dimensional Coulomb matrix $\boldsymbol{M}$. The eigenvalues are entered into a vector in decreasing order ($\lambda_i \geq \lambda_{i+1}$), and the vector makes up the eigenspectrum representation.

This representation is derived from the Coulomb matrix, but there is a loss of dimensionality when the eigenspectrum representation is computed. A Coulomb matrix has $3d$-6 degrees of freedom whereas the eigenspectrum representation vector has dimensionality d as there are d eigenvalues. It is possible for a loss in dimensionality to result in a loss of information. Moussa explores this problem in more detail, and suggests that the loss of dimensionality can also introduce noise.[20]

Sorted Coulomb matrices provide another solution to the arbitrary ordering of atoms. With this method, the rows and columns are permuted in the order of the value of their norms, i.e. $||M_i|| \geq ||M_{i+1}||$. Contrary to the eigenspectrum representation, this increases the dimensionality of the system to d^2.

With this method, if the molecular structure is changed even slightly, then the ordering of the Coulomb matrix rows and columns may change significantly. This may indicate that this way of ordering the columns may not be the best way to present the pattern to a machine learning algorithm.

A third possible alteration to the Coulomb matrix is randomly sorted Coulomb matrices. In this representation, multiple Coulomb matrices will be used to model a single molecule.[21] The first Coulomb matrix is ordered randomly. From this matrix, the norms of the rows are computed and entered into a vector, $||\boldsymbol{M}||$, and random noise ϵ is added. This vector is ordered in decreasing order just like the sorted Coulomb matrices, and the given permutation of atoms is used to create the random Coulomb matrix. The idea behind this method is that the random matrix is a sample from the distribution of all possible sorted Coulomb matrices. This way, when the atomic coordinates of a system change a little, it will not necessarily correspond to a large change in the molecular representation.

This method does not increase the dimensionality of the system. The main problem is that the machine learning computational cost is increased simply because there are multiple representations for each molecule.

1.4.4 Position Intracules

Up until this point, no representations have been discussed which include information about the electronic structure of a molecule. As a result of this, it would be impossible for machine learning algorithms to work with molecules in their excited state. One possible molecular representation that can handle excited states is the intracule.[22–25]

Intracules were first calculated by Coulson and Neilson in 1961.[24] The calculation of a position intracule is an integral which gives the probability distribution of the separation vector of the electrons. It has been widely studied and provides information about the electronic structure of a two-electron system. The position intracule is defined as

$$I(\boldsymbol{u}) = \int\int \rho(\boldsymbol{r}_1, \boldsymbol{r}_2)\delta(\boldsymbol{r}_{12} - \boldsymbol{u})d\boldsymbol{r}_1 d\boldsymbol{r}_2 \tag{1.3}$$

where $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ are the coordinates of electrons 1 and 2, $\boldsymbol{r}_{12}$ is the vector between the two electrons, $\rho(\boldsymbol{r}_1, \boldsymbol{r}_2)$ is the pair density of the two electrons, and $\boldsymbol{u} = \boldsymbol{r}_2 - \boldsymbol{r}_1$ is the inter-electronic vector.

When analyzing a system with more than two electrons, the probability distribution of every pair of electrons is computed. Therefore, if there are N electrons in a system, the position intracule describes the distance between all $\frac{N(N-1)}{2}$ electron pairs.

One disadvantage of using the position intracule as a molecular representaiton is that it must be computed. The integrals may require significant computational resources for many-electron molecules, and machine learning requires many molecules to effectively learn the relationship between structure and physical properties.

1.5 Machine-Learned Latent Molecular Representation

The objective of this thesis is to explore a new way to numerically represent molecules. From the criteria previously mentioned, the goal is to develop a representation that is translationally invariant, rotationally invariant, and unique. Additionally, high information density is an aim to present the relationship connecting the structure of molecules and their associated physical observables clearly with a relatively small amount of information. Finally, any computation involved to acquire a new representation must be relatively short in duration in order to have a representation that can be used in practice.

A neural network approach will be used to create a molecular representation that is information rich (high information density). This new representation will be called the Latent Representation in this work. Unlike Cartesian coordinates or a Coulomb matrix, this will likely lead to a representation that is conceptually very difficult, but expressed concisely. In essence, the project will result in a neural network which will take the input in the current forms of molecular representation, and output a machine learned molecular representation - the Latent Representation.

There are multiple machine learning algorithms that could be used to attempt a task such as this. Neural networks were chosen specifically for this project because out of all the methods of machine learning, neural networks are the most efficient at condensing a system down without losing information. In a sense, an attempt will be made to compress current molecular representations down to a more concise representation.

Another advantage of neural networks is the scaling of the computational cost. Compared to rival algorithms (kernel ridge regression), neural networks scale very well as the volume of data to be processed increases. This becomes increasingly important for the scale of this project, as hundreds of thousands of molecules will be entered to train the neural network.

1.6 Evaluation of Machine Learned Representations

1.6.1 QM9 Database

Like all machine learning algorithms, the neural network will require the molecular representation of a large number of molecules to successfully compress the data without losing important information. Therefore, a database with the Cartesian coordinates of many molecules will be needed to complete this work. As long as the Cartesian coordinates are known, all of the other representations with the exception of intracules can be easily calculated.

The QM9 database contains the Cartesian coordinates of approximately 134,000 stable, small organic molecules including all possible compounds with up to 9 heavy atoms (carbon, oxygen, nitrogen, and fluorine).[27,28] Other than having a large amount of information, this database is also useful because other researchers have used it to train their machine learning algorithms. This makes it particularily useful because we will be looking at the work of other researchers to validate our molecular representation as stated above.[16,29–31]

Once molecular representations are created with neural networks, a large portion of this project will involve determining how useful the representations are. To evaluate the results in this way, it is important to define "useful" and what it may mean in different situations.

There are two properties that the proposed molecular representation may have: retrieval accuracy and future machine learning performance. If we are successful in either category, our method may be useful to big-data scientists and chemists alike.

1.6.2 Retrieval Accuracy

Once the data contained in atomic coordinates, Coulomb matrices, or intracules is compressed by a neural network, there is no guarantee that it is possible to get back the original data. If reproduction is successful, then the new representation might work as an intermediate step for other machine learning algorithms to extract information such as physical observables. However, if there is a degredation in reproducibility, the method could produce a poor molecular representation for general purpose use or storage of data.

For this reason, a metric will be determined for measuring the retrieval accuracy of the machine learned molecular representation. If a Coulomb matrix, for example, could be compressed by a neural network and then be fully extracted, the compressed representation would receive a perfect retrieval accuracy score. Two metrics will be used to calculate retrieval accuracy: in-sample and out-of-sample reproducibility error. These will be discussed in detail in section 3.3.

1.6.3 Comparison of Representation Performance

It may turn out that for some molecules or input representations it is not possible to accurately retrieve the original representation once the compression has occured. This may not be optimal for a general purpose solution to the molecular representation problem, but a positive result could still come from a neural network compression.

Currently, there is much work being done in training machine learning algorithms to predict physical properties such as atomization energy or formation energies.[16,26] The accuracy of their results greatly depends on the molecular representation they used as input. If the compressed neural network representations proposed in this thesis replaced the input used in the cited studies, the accuracy of the results would be affected. If the resultant accuracy increased, then the compressed molecular representation would serve as a useful intermediate phase for machine learning in quantum chemistry.

Therefore, comparing the results of machine learning performance with current forms of molecular representation as input and the compressed representation as input will serve as further evaluation for our method.

2 Neural Network Theory

In this chapter, the mathematics behind neural networks will be explored. First, sections 2.1 and 2.2 introduce two different types of neurons; perceptrons and sigmoid neurons. Sections 2.3 and 2.4 discuss how the neurons communicate with each other to accomplish set tasks. In sections 2.5 through 2.7, the optimization mathematics are discussed, and the chapter concludes with a neural network summary in section 2.8.

2.1 Perceptrons

Neural networks are a form of machine learning originally modeled after a simplified analogy of the communication system used by the neurons in a human brain. The original neuron in these artificial networks is called the perceptron, and although they are not commonly used today, they still provide an excellent base for understanding neural networks.

A perceptron can be given any number of real number inputs, x_1, x_2, ... , x_n. The inputs are then added up in a weighted sum, and if the weighted sum is less than or equal to a threshold value, the perceptron outputs 0. If the weighted sum is more than the threshold value, the perceptron outputs 1. This gives the following piecewise function for the output, a, of a perceptron

$$a = \begin{cases} 0, & \sum_{j=1}^{n} w_j x_j \leq \text{threshold} \\ 1, & \sum_{j=1}^{n} w_j x_j > \text{threshold} \end{cases}$$

The threshold and the weights can be chosen to perform specific operations. For example, if both of the weights for a perceptron with two inputs are chosen to be -2, and the

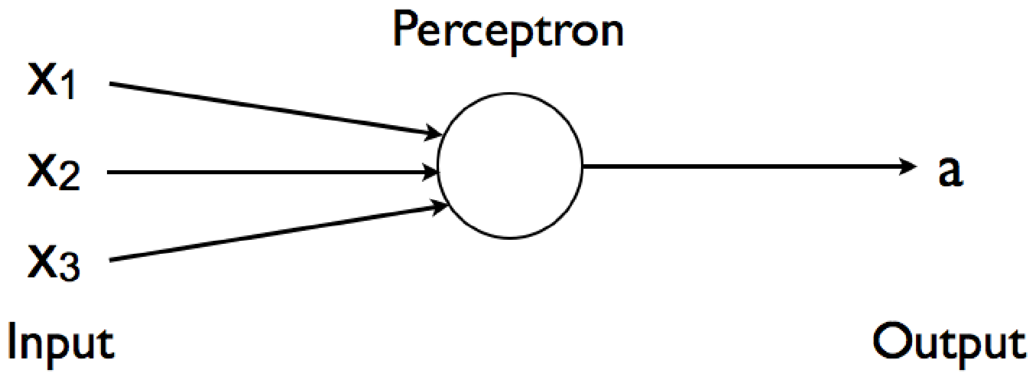

Figure 2.1: Diagram of a perceptron being given multiple inputs, denoted by x_i, and returning a single output, a.

threshold is chosen to be -3, the perceptron would act as a functional NAND gate (Short for Not AND). NAND gates are a fundamental basis of computing, so it follows that perceptrons can be used for all computer operations with intelligent connections and choices of weights and threshold values.

In machine learning notation, it is common to define the bias, b, as

$$b \equiv -\text{threshold} \tag{2.1}$$

Using dot product notation for the summation, this gives the output, a, of a perceptron as

$$a = \begin{cases} 0, & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

where **w** and **x** are vectors which include the weights and inputs respectively for each input.

A series of perceptrons that feed into each other made up the original neural networks. The weights and biases of the perceptrons can be chosen arbitrarily at first, and then modified to optimize a system. The major shortcoming of the perceptron is that its output is binary. The objective of machine learning is to always make small changes in system parameters (weights and biases in the case of neural networks) to optimize a system. With binary outputs, however, small changes in the parameters can cause large changes in the output. This problem is solved in part by a similar type of neuron called a sigmoid neuron.

2.2 Sigmoid Neurons

Artificial sigmoid neurons are very similar to perceptrons as they can be given any number of inputs and return a single output. The inputs do not have to be binary as in the case of the perceptron; they can be any real value. The output still depends on the weights, biases, and inputs, but is instead determined by the sigmoid function, $\sigma(z)$, defined as

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \tag{2.2}$$

This function avoids sharp changes in the output when small changes in the input are made, which is critical for machine learning. This property of the function is most easily understood graphically, and can be seen in Figure 2.2.

When $z >> 0$, and when $z << 0$, the sigmoid neuron behaves nearly identical to the step-wise function behaviour exhibited by the perceptron. The main advantage lies in the neighbourhood of $z = 0$, where there is a smooth transition instead of a sharp cutpoint. The reason this allows for efficient learning will be explained in greater detail when the gradient descent algorithm is explained, but the essence of the argument is that derivatives of the sigmoid function will need to be calculated to learn the optimized weights and biases. Derivatives of smooth curves are attainable, whereas derivatives of sudden steps are not.

The sigmoid neuron calculates z, called the weighted input, with the equation

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{2.3}$$

which is the same as the weighted input to the perceptron. The output, a, of the sigmoid

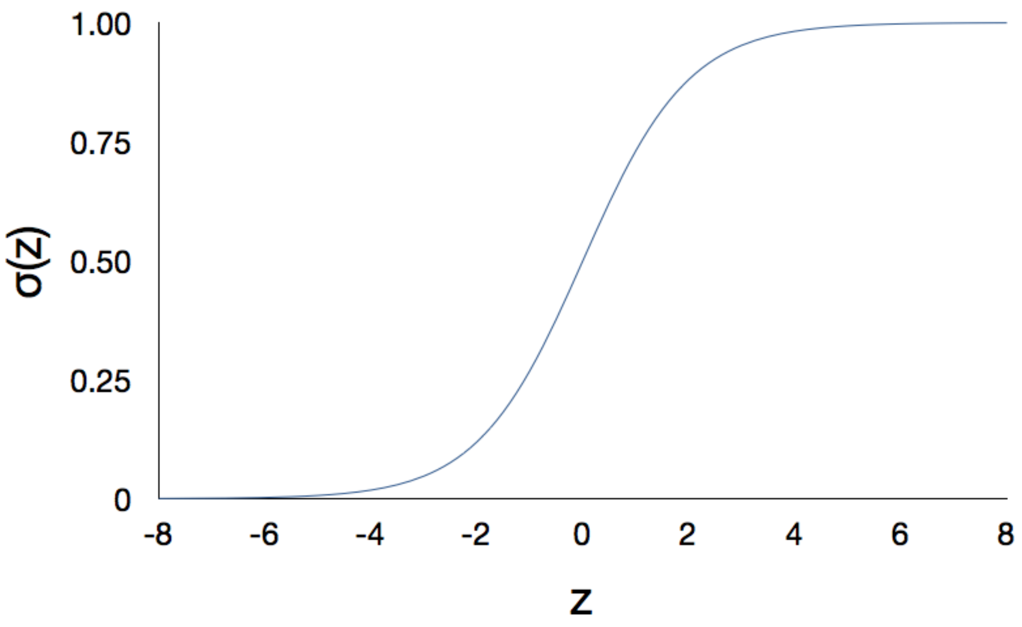

Figure 2.2: Graph of the sigmoid function defined by equation 2.2

neuron is therefore given by

$$a = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp[-\sum_{j=1}^{n} w_j x_j - b]} \tag{2.4}$$

Like perceptrons, the output of multiple sigmoid neurons can be fed into other sigmoid neurons to design complicated systems that can carry out operations and approximate functions. Now that the inputs are not restricted to being binary, neural networks composed of sigmoid neurons are capable of approximating any continuous function.[14,15] The accuracy to which a function is approximated is dependent on the algorithms used to optimize the weights and biases, the hyperparameters chosen to run the network, and the number of inputs given to the network. Hyperparameters will be explained in following sections.

2.3 Connecting Neurons

Neural networks are generally organized into layers of neurons. The input and output of the entire network are both enclosed in a single layer of neurons. In multilayer networks, such as those used in this thesis, there are hidden layers of neurons in between the input and output layers. When information is passed from one layer to the next, that specific type of machine learning approach is referred to as a feedforward neural network. This method of layer to layer communication is illustrated in Figure 2.3.

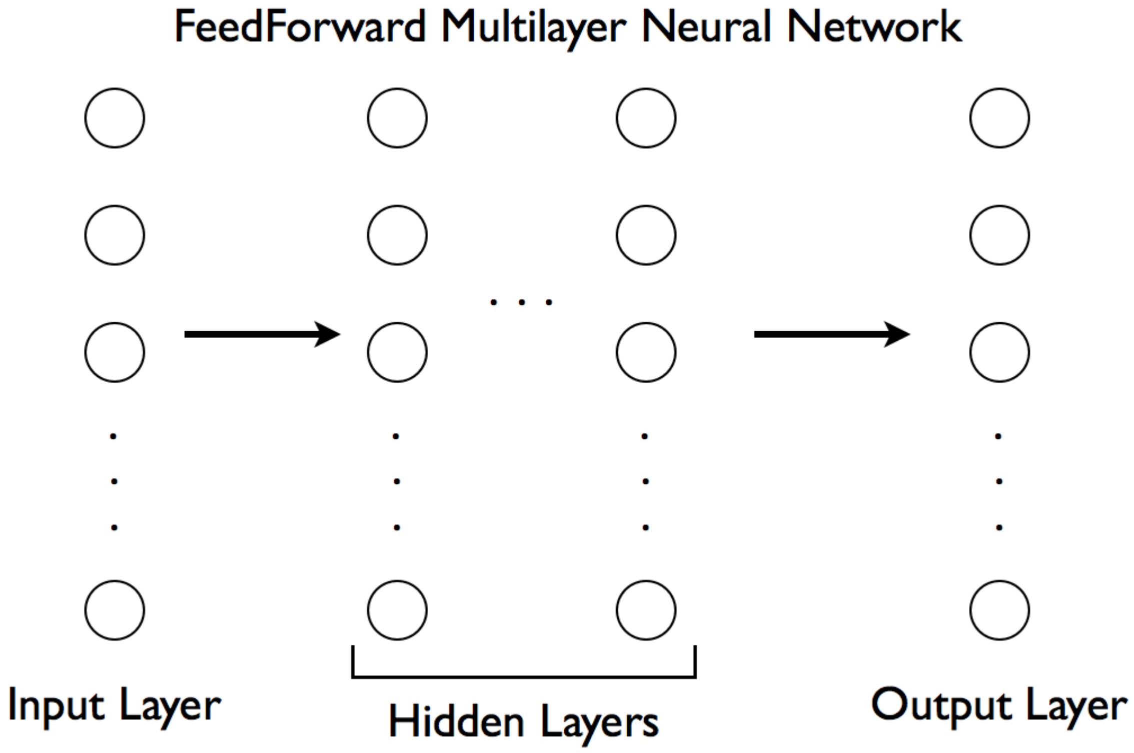

Figure 2.3: Concept diagram of a feedforward multilayer neural network. Information is past left to right from one layer of artificial neurons to the next.

In feedforward neural networks, neurons in one layer send information to neurons in the next layer. The neurons in the input layer are originally fed a value from the data that the network will learn from called the training data. Each neuron in this layer will send its output to every neuron in the first hidden layer. Using the same strategy, every neuron in the first hidden layer will send its output to every neuron in the second hidden layer. This pattern continues until every neuron in the final hidden layer sends an output to every neuron in the output layer. This interconnectedness is demonstrated by Figure 2.4. When there is a path from each neuron in one layer to each neuron in adjacent layers, the neural network is said to be dense. Every combination of neurons in adjacent layers has a unique weight, and every neuron has a unique bias.

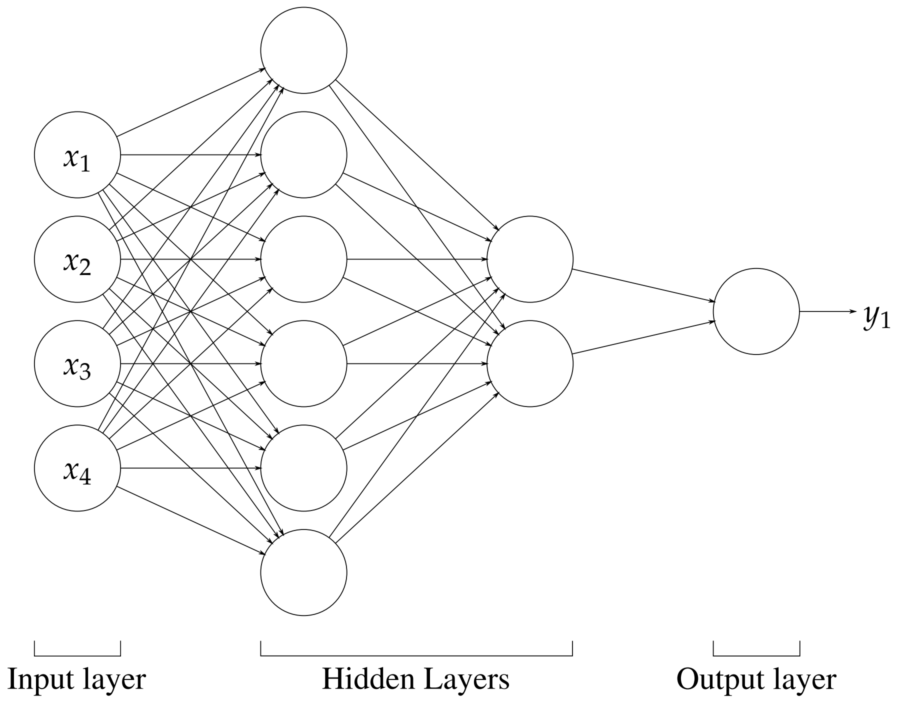

Figure 2.4: Concept diagram of a Dense multilayer neural network.

2.4 Cost Functions

A cost function, or loss function, is a measure of how well the output of a neural network approximates the desired output. An effective cost function depends on the weights and biases of the entire neural network. One commonly used cost function is the quadratic cost function, also often called the mean squared error:

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a||^2 \tag{2.5}$$

where w is a matrix of all weights in the neural network, b is a vector of all biases in the neural network, n is the number of training inputs, x is a vector of all the training inputs, $y(x)$ is the desired output of the neural network called the target function, and a is a vector of all outputs. At this point in the thesis, bold letters will no longer be used to indicate vectors and it can be assumed variables are vectors. To refer to individual components in a vector, indices will be used explicitly. The $\frac{1}{2}$ in the cost function exists to simplify the math later on when partial derivatives of the cost function will be taken (the 2 in the exponent will cancel with the $\frac{1}{2}$).

Optimization of parameters in a neural network is accomplished by minimizing the cost function. It is important to note that the quadratic cost function is a smooth function. Partial derivatives can therefore be taken for minimization. The algorithm that minimizes the cost function is called gradient descent, and it will be discussed in detail in the following section.

2.5 Gradient Descent

The following derivation of gradient descent is explained with two variables, v_1 and v_2, for simplicity. However, the math extends to many variables.

Let the cost function, C, be a smooth multivariable function dependent on v_1 and v_2. The change in $C(v_1, v_2)$ is given approximately by

$$\Delta C \approx \tfrac{\partial C}{\partial v_1}\Delta v_1 + \tfrac{\partial C}{\partial v_2}\Delta v_2$$

This expression is a better approximation for smaller changes in ΔC. Now, define the vector change of v to be

$$\Delta v \equiv (\Delta v_1, \Delta v_2)^T \tag{2.6}$$

where ^{T} denotes the transpose of the vector, making it a column vector.

Defining the gradient of the cost function, ∇C , as

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right) \tag{2.7}$$

the change in the cost function, ΔC, can then be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v \tag{2.8}$$

To make ΔC negative and consequently lower C, choose

$$\Delta v = -\eta \nabla C^T \tag{2.9}$$

where η is some small positive parameter, called the *learning rate*.

The change in the cost function then becomes

$$\Delta C \approx -\eta \nabla C \cdot \nabla C^T = -\eta ||\nabla C||^2 \tag{2.10}$$

Note, $||\nabla C||^2 > 0$, so it has been ensured that $\Delta C \leq 0$ and minimization will occur. From equation 2.9, an update rule can be established. That is, in order to minimize C, each iteration of the algorithm must change v according to the rule

$$v \rightarrow v' = v - \eta \nabla C^T \tag{2.11}$$

The concept of an update rule can be applied to the weights and biases of the neural network because the cost function depends on these variables. The following update rules apply for any smooth cost function dependent on the weights and biases of a neural network

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{2.12}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l} \tag{2.13}$$

where k and l are indices indicating the specific parameter that the update rule is being applied to.

If these update rules are repeatedly applied to all weights and biases, the network will eventually learn the optimized set of parameters that minimize the cost function (though, there are multiple local minima which the cost function may have unintentionally fallen into). Learning occurs during the update, and the rate at which learning occurs, η, is a hyperparameter which is chosen before the algorithm begins. This is why η is called a learning rate. If learning occurs too quickly, the optimal parameters may be overshot. On the other hand, if learning does not occur fast enough, the algorithm will require an unnecessary amount of computational resources to optimize the parameters.

On the topic of computational resources, gradient descent is an incredibly resource demanding algorithm. It is more often than not impossible to compute that many partial derivatives for large datasets. For this reason, a different form of gradient descent, called stochastic gradient descent, has been designed to work with only a sample of the training inputs.

2.6 Stochastic Gradient Descent

With stochastic gradient descent, the goal is to estimate ∇C by computing ∇C_x for a small sample of m randomly chosen training inputs, referred to as a *minibatch*. For sufficient m, it is expected that

$$\frac{\sum_{j=1}^{m} \nabla C_{x_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

The update rules change to incorporate the sample of inputs, because ∇C cannot be acquired from a single sample. In the update rules previously given by equations 2.12 and

2.13, substitute $\frac{\partial C}{\partial w_k} = \frac{1}{m}\sum_{j=1}^{m}\frac{\partial C_{x_j}}{\partial w_k}$ and $\frac{\partial C}{\partial b_l} = \frac{1}{m}\sum_{j=1}^{m}\frac{\partial C_{x_j}}{\partial b_l}$, respectively, to account for the sample of training inputs.

This gives the following update rules for stochastic gradient descent

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m}\sum_{j=1}^{m}\frac{\partial C_{x_j}}{\partial w_k} \tag{2.14}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m}\sum_{j=1}^{m}\frac{\partial C_{x_j}}{\partial b_l} \tag{2.15}$$

Once training has occurred, another minibatch is chosen to train on. The algorithm continues this way until the training data has been exhausted. At that point, an entire *epoch* of training would have occurred, and everything starts over on a second epoch.

At this point in the chapter, the difference between hyperparameters and parameters will be explicitly restated to avoid confusion. A hyperparameter is chosen by the user before training occurs. These include number of epochs, minibatch size, target function $y(x)$, number of hidden layers, number of neurons in each layer, and learning rate η. Parameters are then learned by the neural network. These include weights (w), biases (b), weighted inputs (z), outputs (a) which are also called activations, and errors (δ) which will be introduced in the next subsection.

2.7 Back Propagation Algorithm

The update rules derived in subsections 2.5 and 2.6 involve partial derivatives of the cost function with respect to the weights and biases of every neuron in each layer of the network. The back propagation algorithm computes these partial derivatives.

Two assumptions need to be made for the back propagation algorithm to work.

1) The cost function, C, can be written as an average over the cost functions for indivdual training examples, C_x.

This assumption is needed because back propagation takes the partial derivatives $\frac{\partial C_x}{\partial w_k}$ and $\frac{\partial C_x}{\partial b_l}$ for a single training example and recovers $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ by averaging.

2) The cost can be written as a function of the outputs from a neural network. That is, $C = C(a^L)$, where L denotes the output layer of the neural network, and a represents the output of a sigmoid neuron as defined in section 2.2.

This is an important assumption because back propagation begins at the output layer and works backwards through the layers to compute partial derivatives, and this can only be done if this assumption is true.

Before explaining the back propagation algorithm, a few concepts must be introduced. First, a new index scheme will be needed because back propagation relates the variables of adjacent layers. Therefore, the layer and neuron a parameter is associated with is important to express with notation. Superscripts will represent the layer of the neuron to which a parameter belongs, and subscripts will represent which neuron in the layer the parameter belongs. Weights are a little more complicated because they require the position of the two different neurons in two adjacent layers, so a single superscript/double subscript scheme will be used.

To relate a^l_j, the activation of the j^{th} neuron in the l^{th} layer, to an activation in the $(l-1)^{\text{th}}$ layer, consider the following equation

$$a^l_j = \sigma\Big(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\Big) \tag{2.16}$$

where j is the neuron index in the l^{th} layer, k is the neuron index in the $(l-1)^{\text{th}}$ layer, and w^l_{jk} represents the weight of the connection of the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer. Only three indices are used for the weights instead of four because it can be assumed that weights only connect neurons of adjacent layers which makes it unnecessary to include two superscripts.

To understand the back propagation equations, the Hadamard product must be introduced. The Hadamard product, also known as the Schur product, of two vectors s and t is defined as

$$(s \odot t)_j = s_j t_j \tag{2.17}$$

The Hadamard product is taken with the $\odot$ symbol and is used for elementwise multiplication of two vectors.

The final item that must be introduced to understand back propagation is the error, δ. Specifically, the error of the j^{th} neuron in the l^{th} layer is given by δ_j^l. The error is an intermediate quantity computed in back propagation, and is defined as

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{2.18}$$

where z is the weighted input as defined in section 2.2.

The back propagation algorithm is given by four equations. The first is an equation for the error in the output layer.

$$\delta_j^l \equiv \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \tag{2.19}$$

where L is the layer index indicating the output layer, σ is the sigmoid function, and the prime indicates the first derivative of the sigmoid function with respect to z_j^L. If C does not depend heavily on a_j^L (the output of a particular neuron in the output layer), then the error, δ_j^L, will be small.

This quantity is simple to compute. If, for example, the quadratic cost function, $C = \frac{1}{2}\sum_j (y_j - a_j)^2$, is used in a neural network, the required derivative is simply $\frac{\partial C}{\partial a_j^L} = -(y_j - a_j)$ as all terms in the summation become zero except one.

The second back propagation equation is an equation for the error vector δ^l in terms of δ^{l+1}.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.20}$$

where $\odot$ denotes the Hadamard product. This equation is where back propagation got its name. The first equation calculated the error of each neuron in the output layer, and this equation can use these output layer errors to obtain the errors of the previous layer. It is said that the error propagates backwards through the neural network with this equation. Incrementally using this equation will provide the error of every neuron in the network.

The third equation used in back propagation is an equation for the rate of change of the cost with respect to any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.21}$$

and the fourth equation for back propagation is an equation for the rate of change of the cost with respect to any weight in the network.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{2.22}$$

These two equations use the error of a given neuron (which were all calculated using the first two back propagation equations) to obtain the rate of change of the cost function with respect to both network parameters. These partial derivatives are required for the update rule.

One final note about back propagation regards the speed at which particular weights or biases are learned. The third and fourth back propagation equations show that the partial derivatives of the cost function depend on the activation of a particular neuron in the previous layer and the error of a given neuron. When either this activation or the error are small, that particular partial derivative of the cost function is small. In this case, it is said that the parameter is learning slowly. The first two back propagation equations show that the error is proportional to the derivative of the sigmoid neuron. This quantity is small when the sigmoid function is not rapidly changing, which occurs at $z << 0$ and $z >> 0$ as can be visually understood from Figure 2.2 in subsection 2.2. This is interesting because in those two ranges of z, the sigmoid neuron behaves like a perceptron. Therefore, one might suggest that the sigmoid function has "collapsed" into a perceptron, and this has caused the neuron's parameters to learn slowly.

When the parameters of a neuron are learning slowly, the neuron is said to be saturated. This is another way of saying that the neuron has more or less decided in which extreme its output will lie.

2.8 Summary of Learning in Neural Networks

This section is meant to summarize the overall map of how learning occurs in a neural network based on the algorithms and concepts explored in this chapter.

Step 1) Input sample of training data

Step 2) For each sample of training data,
(a) Feedforward through the entire network. At each neuron, compute z and $a = \sigma(z)$.
(b) Output error δ^L for the neurons in the output layer using the first back propagation equation.
(c) Back propagate the error using the second back propagation equation.
(d) Use the third and fourth back propagation equations to compute the partial derivatives of the cost function with respect to every parameter in the neural network.

Step 3) Apply the update rules for each parameter in the neural network.

These three steps should be completed for each minibatch in an epoch.

3 Neural Network Auto-encoders

An auto-encoder is a framework that can map high dimensional representations of data, x, to low dimensional representations, y. This is done such that there is a high likelihood of replicating x given y. Auto-encoders are made of two parts: an encoder is used to map x to y, while a decoder is used to attempt to recover x given y.

In this work, encoders will be used to produce the latent representation. In theory, the latent representation will contain enough information to retrieve the original molecular representation, and the decoder will be used for this retrieval.

There will inevitably be a limit to the lower dimension which the original molecular representation can be successfully transformed into (as a limiting case, note that a molecule cannot be represented by a single number). Therefore, the goal of this work is to find a balance between retrieval accuracy and dimensionality reduction.

Machine learning has been previously used for autoencoding.[32,33] Recently, neural networks have been used as "quantum auto-encoders",[34] which were used to compress quantum states. The success of this work shows that auto-encoders made up of neural networks can be used to compress molecular data. The work in this thesis will extend this idea to molecular representations.

3.1 Learning to replicate

In the work on quantum auto-encoders,[34] the cost function was based off the difference between the values of neurons in the input layer and the values of the corresponding neurons in the output layer. This enables the neural network to update the weights and biases to maximize retrieval accuracy. Using the notation defined in section 2.4, $y(x) = x$ in an auto-encoder neural network.

This idea will be used in the auto-encoders in this work as well. The cost function, that was introduced in equation 2.5, will compare the value in an input neuron with the value of the corresponding output neuron. If perfect replication is achieved, the input and output layers will be identical.

Both mean squared error (equation 3.1) and mean absolute error (equation 3.2) will be used for the functional forms of the cost function:

$$C(w, b) = (1/2n) \sum_x ||x - a||^2 \tag{3.1}$$

$$C(w, b) = (1/2n) \sum_x ||x - a|| \tag{3.2}$$

where w is a matrix of all weights in the neural network, b is a vector of all biases in the neural network, n is the number of input neurons (which is equivalent to the number of output neurons), x is a vector of all the training inputs, and a is a vector of the outputs of each of the neurons in the output layer. Two cost function forms were used in this work because both are commonly used in machine learning literature.[35]

3.2 Auto-encoder Neural Network Structure

The cost function is summed over training inputs, or more simply, the neurons in the input layer. In order for this sum to return a value for each term in the sum, the number of input neurons must equal the number of output neurons ($\dim(x) = \dim(a)$). Therefore, although there is freedom to choose how many neurons will be in each hidden layer, the input and output layers must both be equal to the number of neurons required to represent a molecule.

An additional constraint on the auto-encoder structure is the location of the hidden layer which encloses the latent representation. For example, if you have a neural network with 8 hidden layers, there is no obvious way to choose which layer will contain the information that will be the result of the compression. This problem leads to the following structure mandate: all auto-encoders in this work will have an *odd* number of hidden layers, and the middle layer will be defined to be the layer containing the latent representation. This is illustrated in Figure 3.1.

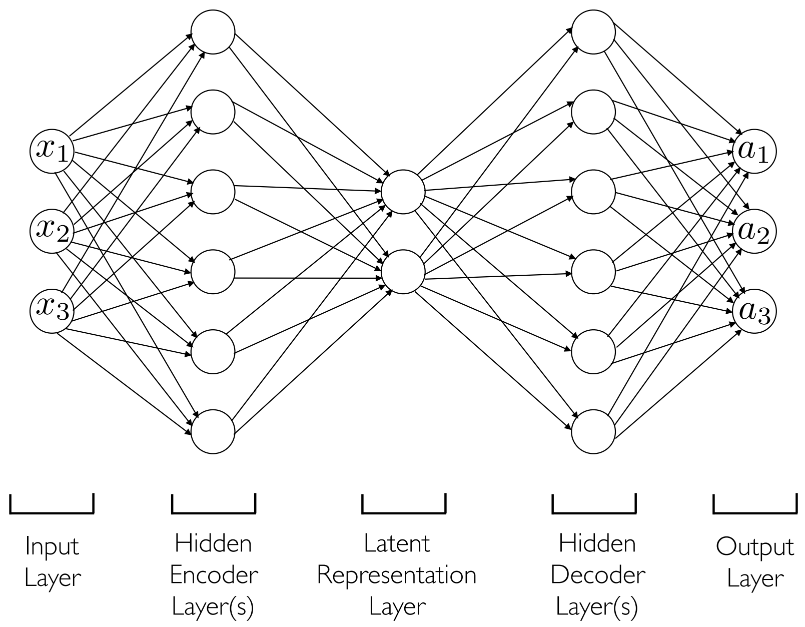

Figure 3.1: Ilustration of an auto-encoder neural network composed of three hidden layers.

It is important to note that this is not the only way to build an auto-encoder, it is simply a decision that was made for the auto-encoders that will be used in this work. It provides a clear definition of the latent representation while still providing the freedom to vary the neural network size.

The way this structure is built mandates that the encoder and decoder, which include the hidden layers other than the latent representation hidden layer, are made up of the same number of hidden layers. In a sense, there is a certain symmetry to the encoding and decoding work in this network. However, the encoder does not neccessarily have to have the same number of neurons as the decoder, it must only have the same number of hidden layers to meet the requirements of the symmetry mandate.

In building and testing these neural networks, one of the most interesting quantities to vary is the number of neurons in each layer. The limits on these optimization spaces will be discussed more in the results section. From a structure point of view, the only requirement at this point is that the middle hidden layer must be composed of a fewer number of neurons than the input/output layers. Otherwise, there would be no dimensionality loss.

For a given auto-encoder structure, only one type of optimization algorithm will be used, and all neurons will have the same activation functions. Optimization algorithms and activation functions may still be varied, but not in a single neural network. Rather, mutiple auto-encoders can be tested - each with different optimization algorithms and/or activation functions. Currently in neural network literature, there is no "best" optimization algorithm or activation function. Therefore, a few of the current leading candidates will be tested.[35]

3.3 Evaluation of Auto-Encoder Performance

Three metrics will be used to evaluate the performance of the auto-encoders. Reproducibility is a key interest, and there are two ways to measure the reproducibility. First, note that the neural networks are trained to maximize reproducibility. Looking at how much the cost function of a neural network was minimized is a way of measuring in-sample reproducibility. "Validation loss" is a measure of how well the neural network can reproduce molecules fed through the auto-encoder that were not included in the training data. This is a measure of out-of-sample reproducibility. Finally, information compression is a useful metric to define the magnitude of compression being applied by the auto-encoder. These will be discussed in more detail in the following subsections.

It is important to note that all three metrics may not be equally important, and certain metrics may be more or less useful when considering specific applications. For example, if there is a large (on the order of the size of the QM9 database) dataset of molecules and the auto-encoder is to be used only to compress the molecules in the dataset, then the out-of-sample reproducibility is unimportant because the auto-encoder can be trained on the molecules in the database. In this case, the only measure of reproducibility that is relevant is the in-sample reproducibility error because the training data is the target. Conversely, if there were only a small number of molecules to be compressed, then an auto-encoder would need to be trained on a large dataset because machine learning algorithms require large datasets to produce useful results. It could then be used to compress the smaller dataset, and in this case it is much more important that the auto-encoder has high out-of-sample reproducibility.

3.3.1 In-sample Reproducibility Error

In-sample reproducibility error will be defined to be equivalent to the minimized cost function in the case of a mean absolute error cost function. Some of the auto-encoders in this work were trained with mean squared error cost functions, and in this case the in-sample reproducibility error will be defined to be equivalent to the square root of the minimized cost function. This is done mainly to have an appropriate comparison of auto-encoders trained with different cost functions. This cost function will also be referred to as the training loss.

This metric is a measure of how well the auto-encoder can reproduce the molecules in the training set. Auto-encoders with low in-sample reproducibility errors will therefore perform well on the molecules they were trained on.

3.3.2 Out-of-sample Reproducibility Error

Out-of-sample reproducibility error is related to validation in machine learning. In most machine learning algorithms, the end goal is to have an algorithm trained on one dataset and then applied to other data. This is where the word "learning" comes from in machine learning - an algorithm can learn from one dataset and apply what it has learned to another dataset. Validation is a measure of how well the machine learning algorithm performed on data that was not included in the training set.

For the neural networks functioning as auto-encoders, the only way to test validation is to omit some of the molecules in the dataset. When a neural network is training on molecules represented by cartesian coodinates for example, it will not be able to "see" all of the QM9 database. Rather, the database will be broken up into five bins of equal size. Four of the bins will be used for training, and the fifth bin will be held for validation at the end of the training.

Once a neural network has completed training, validation is calculated with the same cost function as the network was trained on. Instead of using the first four bins however, the cost function is calculated by having the information in the fifth bin fed through the trained neural network. This cost function is referred to as the validation loss.

Out-of-sample reproducibility error is defined in the same way as in-sample reproducibility error, except with reference to the validation loss instead of the minimized cost function.

3.3.3 Information Compression

The neural networks being considered act as auto-encoders by having the middle hidden layer be made up of fewer neurons than the input layer. This way the latent representation will involve less data than the original representation. With this in mind, the metric that will be used for determining the magnitude of information compression will involve the ratio of the number of neurons in the middle layer to the number of neurons in the input layer. Specifically, information compression is defined as

$$\text{Information Compression} = \left(1 - \frac{N_{\text{latent}}}{N_{\text{input}}}\right) \times 100\% \tag{3.3}$$

where N_{latent} is the number of neurons in the middle layer which hold the information for the latent representation, and N_{input} is the number of neurons in the input layer.

3.4 Auto-encoder Hyperparameters

To build auto-encoders that encode with high reproducibility, it is necessary to intelligently choose hyperparameters for the neural network. There are many hyperparameters and each has a large window of acceptable values. As a starting point, the optimizer used is ADAM.[35] This is a different optimizer than stochastic gradient descent, but it still minimizes the cost function and has performed well in recent literature.[35] Stochastic gradient descent has the tendency to find a minimum quickly, which is a problem if a local minimum is found by accident. This is a problem that ADAM handles better than stochastic gradient descent. Using the paper that ADAM was released in as a guide, the learning rate will be set at 0.001.[35]

Much like stochastic gradient descent is not used much in practice anymore, sigmoid neurons are also out-performed by newer activations. Two activations that will be used in the following auto-encoders are rectified linear units, and exponential linear units.[36]

The software used for the neural networks is based off of *Keras*.[37] *Keras* is a neural networks library, and is accessed by the Keras Model Loader which was designed by Trevor Profitt at the University of Prince Edward Island. *Keras* communicates with tensor manipulation libraries, and the library choosen for this study was Theano.[38]

4 Molecular Representation Conversion

In this chapter, three molecular representations (Cartesian coordinates, Coulomb matrices, and position intracules) will be explored further, and the architecture of the neural networks required to compress each type of molecular representation will be discussed.

4.1 Cartesian Coordinates

The Cartesian coordinates auto-encoders used in this work have 87 input neurons and 87 output neurons. The largest molecule in the QM9 database has 29 atoms. Each atom requires 3 coordinates to represent it, and each of the $3 \times 29 = 87$ coordinates is given to a different input neuron when the network is initialized.

For the molecules with fewer than 29 atoms (most atoms in the database), the 87 input neurons must still be filled at the beginning because the neural network will attempt to minimize the cost function for all molecules in the database at once. The choice was made to fill these "extra" input neurons with zeros.

This size inconsistency presents an immediate issue. To the neural network, there is no real difference between an atom at position (0,0,0) and no atom at all. A hypothesis of this work is that molecules expressed in Cartesian coordinates will be difficult to compress with a neural network auto-encoder.

The coordinates of atoms will be given in units of Å. Typical bond lengths are between 1 Å and 2 Å. In order for an auto-encoder to effectively reproduce molecules after compression, an error of no more than 0.1 Å per atom is required.

4.2 Coulomb Matrices

A Coulomb matrix auto-encoder has 841 input neurons and 841 output neurons. The molecules with 29 atoms are accomodated this way because a matrix with 29 rows and 29 columns has 841 entries. Using a python NUMPY command, these matrices are flattened so that the first 29 input neurons correspond to the first row of the Coulomb matrix, the next 29 input neurons correspond to the second row of the Coulomb matrix, and so on. Molecules with less than 29 atoms will have the remaining input neurons filled with zeros, similar to the Cartesian coordinate auto-encoder.

The entries of a Coulomb matrix are given by equation 1.2. In a 29×29 matrix, there are many more off-diagonal elements than there are diagonal elements. Thus, the choice was made to report the results in units of $\mathring{A}^{-1}$ ($\text{charge}^2 \times \text{distance}^{-1}$ with the charge given in atomic units). Technically, the diagonal elements of the matrices have units of $\text{charge}^{2.4}$, but a neural network cannot differentiate between two different types of numbers. Therefore, the units of the more prevalent off-diagonal elements were chosen. This is not likely to be a source of error in reproducibility because the neural network's task is to simply reproduce numbers - regardless of units. However, it is important to consider when interpreting reproducibility errors.

The reproducibility errors will give a sense of how much each element in every Coulomb matrix in the dataset is off by, in units of $\mathring{A}^{-1}$. It is necessary to consider the typical values in a Coulomb matrix to understand whether the measures of reproducibility error are high or low. For diagonal elements, the entries will typically be between 0.5 $\mathring{A}^{-1}$ and 45 $\mathring{A}^{-1}$ (which was calculated with equation 1.2 considering the largest element in the QM9 database is Fluorine). For off diagonal elements, the entries will typically be between 0.001 $\mathring{A}^{-1}$ and 27 $\mathring{A}^{-1}$. There are many more entries closer to 0.001 $\mathring{A}^{-1}$ than 27 $\mathring{A}^{-1}$ because there are many hydrogen-hydrogen interactions in the molecules in the QM9 database. Hydrogen has an atomic number of one, and referring to equation 1.2 it can be seen that these entries in a Coulomb matrix will be less than one.

Interpreting errors will be difficult given the range of entries in Coulomb matrices is large and not normally distributed. A reproducibility error < 0.001 Å^{-1} is certainly low enough to guarantee reproducibility, but higher errors may be difficult to interpret.

4.3 Position Intracules

A position intracule auto-encoder has 256 input neurons and 256 output neurons. Unlike the case of loading an auto-encoder with cartesian coordinates, all molecules will be fully represented by 256 neurons. This is because the probability distribution given by calculating the position intracule is broken up into 256 pieces. Each value of the distribution is given to one input neuron. They are values of the probability density. To illustrate the range of these values, the intracule of methane can be seen in Figure 4.1. Note, the points are not evenly spaced in the $\mathbf{u}$ dimension. This is because the information of the probability density at low separation values is more informative of electronic structure.

Each molecule has a unique probability distribution given by the position intracule. Considering that smaller molecules do not need zeroes added to their input representation, there is a strong chance that a neural network will be able to encode many molecules simultaneously (relative to the two previous input representations).

Unlike the Cartesian coordinates and Coulomb matrix auto-encoders, the Intracule auto-encoder will not be trained on all molecules in the QM9 database. This is simply because the intracule data was not available for this project. There were 21,271 molecules used in the position intracule auto-encoder study.

Again, to interpret the reproducibility errors, it is useful to consider what a typical value of an intracule data point is. To get a sense for the range of possible values, the maximum value of the probability density of each molecule was considered. Of all maxima, the lowest was 19.0289 and the highest was 500.0753. Due to the factor of $\frac{1}{2}N(N-1)$, the distributions with smaller maxima correspond to smaller molecules. Therefore, reproducibility

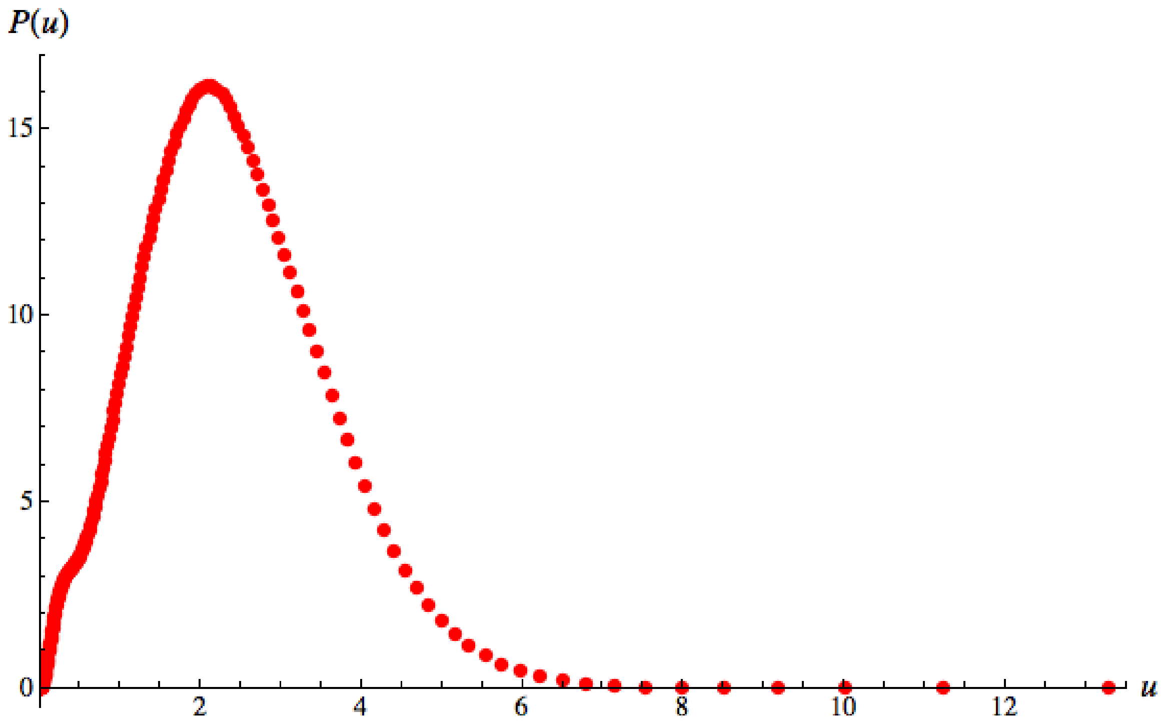

Figure 4.1: 256 data points of the intracule density for the ground state of methane.

errors will make it more difficult to reproduce smaller molecules after compression. For example, if an auto-encoder could reproduce all 256 probabilities of each molecule with a reproducibility error of 0.5, then the maxima of the largest molecule will have a relatively small percent error (500.1 ± 0.5 yields a percent error of 0.0999% per data point) while the maxima of the smallest molecule will be a relatively large percent error (19.0 ± 0.5 is yields a percent error of 2.6276% per data point). Note, this comparison is made on the maximum probability in each case, and the percent error would be even higher for other points in each distribution. The maxima were used in this comparison because they are easily calculable and still give an idea of the size of the values in the distributions.

5 Results and Discussion

The results of these thesis are presented in four sections. In section 5.1, strategies for exploring the hyperparameter space are compared. The following three sections discuss the results for Cartesian coordinate auto-encoders, Coulomb matrix auto-encoders, and position intracule auto-encoders, respectively.

5.1 Learning to Explore the Hyperparameter Space

There are two schools of thought in machine learning when exploring the hyperparameter space. One strategy is to iteratively exhaust many combinations of hyperparameters. This can be done more easily on smaller networks which can be trained quickly.

To try out this strategy, a three layer auto-encoder trained on position intracules was employed. A summary of the neural network settings and hyperparameters is given in Table 5.1, and a graph of the cost function as a function of number of neurons in the hidden layer can be seen in Figure 5.1.

As seen in Table 5.1, 20 epochs of training occured for each auto-encoder. Before running a complete trial of all 256 auto-encoders, a few single trials occured with a large number of epochs (>40). By comparing the cost function with the validation loss, it was found that the validation loss would start to increase as the cost function continued to decrease around epoch 25. This is an indication of overfitting, and is the reason 20 epochs were trained on each of the 256 auto-encoders during the optimization of hidden layer size.

Figure 5.1 shows the cost function of most of the one-hidden-layer auto-encoders. The

Table 5.1: Summary of fixed and varied hyperparameters, as well as additional neural network settings during initial hidden layer optimization.

Neural Network Settings	Value
Activation Function	Exponential Linear Unit
Cost Function	Mean Squared Error
Number of Input/Output Neurons	256

Fixed Hyperparameters	Value
Epochs	20
Minibatch Size	50
Number of Hidden Layers	1
Learning Rate, η	0.001

Varied Hyperparameter	Range
Number of Neurons in Hidden Layer	1-256

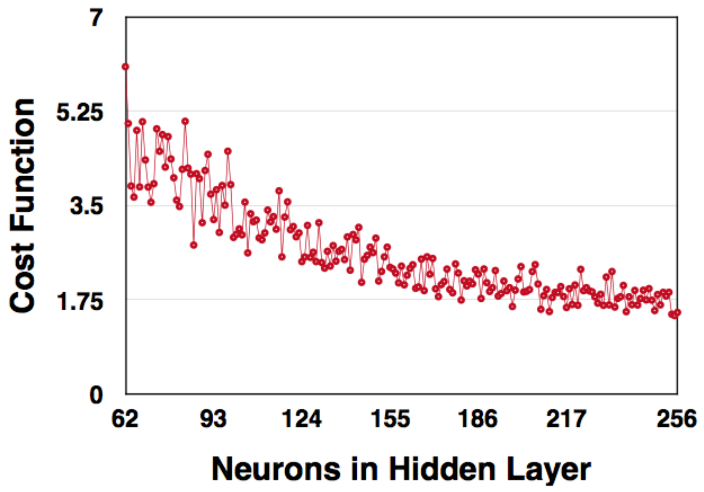

Figure 5.1: The mean squared error cost function in units of the Bohr radius as a function of neurons in the hidden layer of a three layer neural network (input layer, hidden layer, output layer) trainied on position intracules. Each data point represents a neural network auto-encoder with the settings outlined in Table 5.1.

cost function of auto-encoders with fewer than 62 neurons was very high, and diverged to errors in the thousands when only a few neurons were present. This is not surprising as it is unrealistic to expect reproducibility when each molecule in the training set is represented by only a few neurons. Therefore, these high cost function and low neuron number results are less interesting and the figure focuses on the more interesting area. It is clear in this figure that the cost function tends to smaller values as the number of neurons in the hidden layer increases. Additionally, it can be seen that even though the graph tends to lower cost functions globally, locally there are very large deviations. For example, when there were 117 neurons in the hidden layer the cost function was 2.54. Once the number of neurons was increased to 118, the cost function spontaneously rose to 3.28. This is both a significant change in magnitude, and unexpected in direction. One would expect that a larger number of hidden neurons (corresponding to a smaller compression) should result in a lower cost function (higher reproducibility).

This is an example of how the performance of neural networks does not neccessarily experience small gradual changes as corresponding changes are made in the hyperparameters. This suggests that this initial strategy of exhaustively going through every possible value of a given hyperparameter may be unnecessarily time consuming, especially considering the number of hidden layers has been fixed at one and adding hidden layers greatly increases computation time.

To interpret the cost function value, recall that intracules give the probability distribution of the separation distance of electron pairs. For a mean squared error cost function of 1.75, $\sqrt{1.75} = 1.3229$ is the absolute average in-sample reproducibility error of the electron pair probability distribution. It is averaged over the output neurons and over all minibatches, so of all 256 data points in the distribution, on average each point is off by 1.3229. This error should be representative of the entire dataset. Considering larger molecules have many probability density values on the order of ~ 100, an in-sample reproducibility error of 1.32 corresponds to a percent error of $\sim 1\%$ for large molecules. This is likely insufficient for reproducibility of all molecules in the dataset, but a more in-depth study of auto-encoding intracules can be found in section 5.4.

Looking at Figure 5.1, it may seem strange that the cost function does not go to zero as the number of neurons in the hidden layer goes to 256. Afterall, there is no compression happening. The reason for this is that the global minimum of the cost function was not found by the optimizer. Even with only one hidden layer and 256 neurons, the optimization problem is still very complicated. In theory, the global minimum for the cost function is 0 when there is no compression, but in practice it does not go to zero simply because the parameters of the neural network were optimized to give a cost function in a local minimum.

A competing strategy optimizes hyperparameters by randomly exploring the hyperparameter space. After going through many random combinations of hyperparameters, the top performing results can be studied. Furthermore, a more focused search can follow by trying many combinations of hyperparameters similar to those of the top performing auto-encoders. This second strategy will be used for building auto-encoders initialzed with Cartesian coordinates, Coulomb matrices, and position intracules. The results of each will be discussed over the next three sections.

5.2 Auto-encoding Cartesian Coordinates

For the first trial of randomly testing hyperparameters, the number of hidden layers was fixed at three. The main purpose of the random search was to understand how many neurons should be in each layer to maximize performance. Other than the middle layer, there is no reason to have the hidden layers smaller than the input layer (87 neurons long). The range of values for the number of neurons in the 1st and 3rd hidden layers was chosen to be 1 to 200. Also, instead of mandating that each auto-encoder undergo 18 epochs of training, the early-stopping function of the Keras Model Loader was implemented. This function takes an upper limit for epochs as an argument, and may end training early if the validation loss begins to rise. There are 134,000 molecules in the QM9 database, which offers more training data than the case of intracules. For this reason, the minibatch size was raised to 128 after running a few quick trials with different minibatch sizes. This minibatch size seemed to result in a relatively quick computation time which effectively leveraged the processors used to compute the network optimization. Other details of this trial are listed in Table 5.2.

Table 5.2: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the three-hidden-layer Cartesian coordinate auto-encoder trial.

Neural Network Settings	Value
Activation Function	Rectified Linear Unit
Cost Function	Mean Absolute Error
Auto-Encoders Trained in Trial	226
Number of Input/Output Neurons	87

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	128
Number of Hidden Layers	3
Learning Rate, η	0.001

Varied Hyperparameters	Range
Number of Neurons in 1st Hidden Layer	1-200
Number of Neurons in 2nd Hidden Layer	1-86
Number of Neurons in 3rd Hidden Layer	1-200

After running this trial, it was found that the early stopping command halted training roughly between 18 and 30 epochs. This is a good indicator that the upper limit of 50 was set high enough to ensure the validation loss was minimized thoroughly. The results of the top performing auto-encoder in the three-hidden-layer trial are presented in Table 5.3.

The in-sample reproducibility error is very similar to the out-of-sample reproducibility error. This indicates that no large amount of over-fitting has occured and the neural network has effectively (within the given errors, that is) learned how to compress molecules both inside and outside the training data.

An interesting result from the first trial is that there were more neurons in the 3rd hidden layer than there were in the 1st hidden layer in the top performing auto-encoder. In other

Table 5.3: Results of the top performing three-hidden-layer auto-encoder for Cartesian coordinates

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.4982 Å
Out-of-Sample Reproducibility Error	0.4994 Å
Information Compression	39.08%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	60
Number of Neurons in 2nd Hidden Layer	53
Number of Neurons in 3rd Hidden Layer	100

words, the decoder contains more neurons than the encoder. Without analyzing more auto-encoders it is difficult to say if this has a mathematical significance or if it is a coincedence. This is one of the drawbacks of randomly trying combinations of hyperparameters instead of trying all combinations - it is difficult to try to explain why certain hyperparameters produce better results than others.

To justify the use of rectified linear units instead of sigmoid functions, an identical test was carried out where only the activation functions differed. The results will not be presented in the same detail because the top performing auto-encoder had higher reproducibility errors. Specifically, the in-sample reproducibility error was 0.6173 Å, and the out-of-sample reproducibility error was 0.6191 Å. At least in this case, the auto-encoders built with rectified linear units as activation functions outperformed the sigmoid neuron auto-encoders. This is consistent with recent literature, and therefore sigmoid neurons will not be used in other auto-encoders in this work.[36]

A five-hidden-layer neural network was also tested. Similar to the three-hidden-layer trial, the range of number of neurons which was explored for the non-middle layers was set to 1-200. To compare to the three-hidden-layer trial, the minibatch size was also set to 128 and the early-stopping function was implemented. The details of this trial are listed in Table 5.4.

Table 5.4: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the five-hidden-layer Cartesian coordinate auto-encoder trial.

Neural Network Settings	Value
Activation Function	Rectified Linear Unit
Cost Function	Mean Absolute Error
Auto-Encoders Trained in Trial	107
Number of Input/Output Neurons	87
Fixed Hyperparameters	**Value**
Epochs Upper Bound	50
Minibatch Size	128
Number of Hidden Layers	5
Learning Rate, η	0.001
Varied Hyperparameters	**Range**
Number of Neurons in 1st Hidden Layer	1-200
Number of Neurons in 2nd Hidden Layer	1-200
Number of Neurons in 3rd Hidden Layer	1-86
Number of Neurons in 4th Hidden Layer	1-200
Number of Neurons in 5th Hidden Layer	1-200

After running this trial, it was found that the early stopping command halted training roughly between 25 and 45 epochs. This is a larger number of epochs than in the three-hidden-layer trial which is expected considering the neural network is larger. Larger neural networks with more connections are more difficult to train because there are more values to optimize. In addition to having more parameters to calculate, some of the calculations are longer when there are more layers. Consider the back propagation algorithm - the series of partial derivatives which are multiplied together become longer as there are more layers to back progagate through. In general, the optimization problem is more difficult and for this reason more epochs of training are required to train the neural network.

The results of the top performing auto-encoder in the five-hidden-layer trial are presented in Table 5.5. The performance metrics are very similar to the results for the three-hidden-layer neural network. Again, the decoder component of the auto-encoder (com-

Table 5.5: Results of the top performing five-hidden-layer auto-encoder for Cartesian coordinates

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.4993 Å
Out-of-Sample Reproducibility Error	0.4966 Å
Information Compression	32.18%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	88
Number of Neurons in 2nd Hidden Layer	69
Number of Neurons in 3rd Hidden Layer	59
Number of Neurons in 4th Hidden Layer	108
Number of Neurons in 5th Hidden Layer	85

posed of the 4th and 5th hidden layers) contains more neurons than the encoder component (composed of the 1st and 2nd hidden layers). If top performing results continue to include larger decoders than encoders, then the range of values allowed while optimizing hyperparameters could be decreased to focus in on more high performing auto-encoders.

It may seem counterintuitive that the out-of-sample reproducibility error is lower than the in-sample reproducibility error, and this warrants more discussion. One hypothesis for this seemingly unlikely result involves considering how these two quantities are calculated. The cost function for the final epoch of training is an average over the entire epoch. As the neural network makes updates to the weights and biases from one minibatch to the next, the cost function will be making small decreases. These decreases will not be as significant as the changes made in earlier epochs, but they become relevant when comparing to the out-of-sample reproducibility error. The validation loss, which gives the final measure of out-of-sample reproducibility error, is not an average over the entire final epoch. Rather, it is a calculation made at the very end of the final epoch. It is likely that this explanation holds as the primary reason why the out-of-sample reproducibility error is lower. Simply, the training loss is an average over quantities which should be higher at the beginning of the final epoch, and the validation loss is not.

In the case of auto-encoders built to encode cartesian coordinates, there does not seem to be much of a benefit to moving from a three-hidden-layer network to a five-hidden-layer network based on reproducibility errors. Also, the method of auto-encoding does not seem to be overly promising when used on molecules represented by cartesian coordinates. The best input reproducibility error was found to be 0.4982 Å. This means on average, each coordinate was 0.4982 Å off of the corresponding coordinate of the original molecule. This is likely not a sufficient amount of information to recover the original molecule because the average errors per coordinate are comparable to the average bond length. The poor performance of the auto-encoders in this section is likely related to the hypotheses made in previous sections. That is, that the arbitrarily chosen origin in Cartesian coordinates might make it difficult for a machine learning algorithm to apply a single compression transformation to all molecules in the dataset. For this reason, auto-encoders working with Coulomb matrices and position intracules as inputs will be explored in the following two sections.

5.3 Auto-encoding Coulomb Matrices

The results for Coulomb matrix auto-encoders are split up into four sections. In the first section, the number of hidden layers was fixed at one. In the second section, the number of hidden layers was fixed at three, and likewise the 3rd and 4th sections have five and seven hidden layer results, respectively.

The range of neurons for the latent representation was chosen to be 1-200. There are 841 input neurons, so the compression in these trials will result in an information compression of at least 76.22%. This range was chosen to give a relatively small window for the possible number of neurons in the hidden layer. The larger this range is chosen to be, the less the hyperparameter space can be explored in the same amount of time. This way, the results are focused on large compressions and the less interesting results of small compressions are not considered. This dataset is the same size as the Cartesian coordinate auto-encoders, so the minibatch size remains at 128.

5.3.1 One-Hidden-Layer Auto-Encoders

Details of the auto-encoders trained are provided in Table 5.6, and the results can be seen in Table 5.7. The main result to focus on is the reproducibility error. The ~ 0.18 $\mathring{\mathrm{A}}^{-1}$ error is promising when compared to the range of possible values in the Coulomb matrices (0.001 $\mathring{\mathrm{A}}^{-1}$ to 45 $\mathring{\mathrm{A}}^{-1}$). The smaller numbers in Coulomb matrices will certainly be more difficult to reproduce with these auto-encoders, but the larger numbers can be reproduced with a percent error on the order of $\sim$ 0.4% Training only 52 auto-encoders to acquire this result suggests that Coulomb matrices are a better starting point for molecular representation auto-encoders than Cartesian coordinates.

Additionally, the information compression (76.69%) is much higher than that acquired with the Cartesian coordinate auto-encoders (granted, this was done by design in this case). This also provides insight into the information density of Coulomb matrices. If the data can be compressed this much and still be replicated within the given errors, then Coulomb matrices are inherently less information dense than they may need to be to convey the required amount of information for molecular representation.

Table 5.6: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the one-hidden-layer Coulomb matrix auto-encoder trial.

Neural Network Settings	Value
Activation Function	Rectified Linear Unit
Cost Function	Mean Absolute Error
Auto-Encoders Trained in Trial	52
Number of Input/Output Neurons	841

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	128
Number of Hidden Layers	1
Learning Rate, η	0.001

Varied Hyperparameters	Range
Number of Neurons in 1st Hidden Layer	1-200

Table 5.7: Results of the top performing one-hidden-layer auto-encoder for Coulomb Matrices

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.1832 Å^{-1}
Out-of-Sample Reproducibility Error	0.1803 Å^{-1}
Information Compression	76.69 %

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	196

5.3.2 Three-Hidden-Layer Auto-Encoders

Over the next three trials, the number of hidden layers was progressively increased. The one-hidden-layer trial showed the best results of the four trials. This may be because a smaller neural network is the best choice for Coulomb matrix auto-encoders, but it may also be because a smaller percentage of the total possible hyperparameters were explored in the larger auto-encoder trials.

Note that the non-middle hidden layers were given a range of 1-500 neurons. As done

Table 5.8: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the three-hidden-layer Coulomb matrix auto-encoder trial.

Neural Network Settings	Value
Activation Function	Rectified Linear Unit
Cost Function	Mean Absolute Error
Auto-Encoders Trained in Trial	56
Number of Input/Output Neurons	841

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	128
Number of Hidden Layers	3
Learning Rate, η	0.001

Varied Hyperparameters	Range
Number of Neurons in 1st Hidden Layer	1-500
Number of Neurons in 2nd Hidden Layer	1-200
Number of Neurons in 3rd Hidden Layer	1-500

in previous trials, this is a greater range than the middle-layer. In theory this range could be greater, but the more pathways in a neural network, the longer the network takes to train. A balance must be found where the hyperparameters have enough freedom to randomly search a large parameter space, and where the network can be trained within a reasonable amount of time.

Table 5.9: Results of the top performing three-hidden-layer auto-encoder for Coulomb Matrices

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.2162 Å^{-1}
Out-of-Sample Reproducibility Error	0.2128 Å^{-1}
Information Compression	83.12%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	191
Number of Neurons in 2nd Hidden Layer	142
Number of Neurons in 3rd Hidden Layer	450

Although this auto-encoder has a slightly higher set of reproducibility errors, the information compression is higher. Depending on the application, a larger compression may be preferred to a low reproducibility error. In those cases, a three-hidden-layer auto-encoder may be preferred based on these results.

Similar to the Cartesian coordinate auto-encoders, the decoder has more neurons than the encoder (450 vs 191). An exception to this trend for top performing auto-encoders has yet to be observed in the data.

5.3.3 Five-Hidden-Layer Auto-Encoders

The reproducibility errors experience a slight increase again for the five-hidden-layer auto-encoder. However, the highest information compression of any results in this work is reported from this trial at 93.94%. Even as it becomes more difficult to train the larger networks, high compressions seem to be made possible as the number of hidden layers is increased. Again in this trial, the decoder (made up of the 4th and 5th hidden layers) contains more neurons than the encoder.

Table 5.10: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the five-hidden-layer Coulomb matrix auto-encoder trial.

Neural Network Settings	Value
Activation Function	Rectified Linear Unit
Cost Function	Mean Absolute Error
Auto-Encoders Trained in Trial	55
Number of Input/Output Neurons	841

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	128
Number of Hidden Layers	5
Learning Rate, η	0.001

Varied Hyperparameters	Range
Number of Neurons in 1st Hidden Layer	1-500
Number of Neurons in 2nd Hidden Layer	1-500
Number of Neurons in 3rd Hidden Layer	1-200
Number of Neurons in 4th Hidden Layer	1-500
Number of Neurons in 5th Hidden Layer	1-500

Table 5.11: Results of the top performing five-hidden-layer auto-encoder for Coulomb matrices.

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.2414 Å^{-1}
Out-of-Sample Reproducibility Error	0.2401 Å^{-1}
Information Compression	93.94%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	447
Number of Neurons in 2nd Hidden Layer	56
Number of Neurons in 3rd Hidden Layer	51
Number of Neurons in 4th Hidden Layer	88
Number of Neurons in 5th Hidden Layer	448

5.3.4 Seven-Hidden-Layer Auto-Encoders

The seven-hidden-layer auto-encoder has the highest reproducibility errors (both in-sample and out-of-sample) of all of the four trials conducted on Coulomb matrices. One possible

Table 5.12: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the seven-hidden-layer Coulomb matrix auto-encoder trial.

Neural Network Settings	Value
Activation Function	Rectified Linear Unit
Cost Function	Mean Absolute Error
Auto-Encoders Trained in Trial	54
Number of Input/Output Neurons	841

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	128
Number of Hidden Layers	7
Learning Rate, η	0.001

Varied Hyperparameters	Range
Number of Neurons in 1st Hidden Layer	1-500
Number of Neurons in 2nd Hidden Layer	1-500
Number of Neurons in 3rd Hidden Layer	1-500
Number of Neurons in 4th Hidden Layer	1-200
Number of Neurons in 5th Hidden Layer	1-500
Number of Neurons in 6th Hidden Layer	1-500
Number of Neurons in 7th Hidden Layer	1-500

reason for this is that an insufficient amount of hyperparameter combinations were tested to discover the optimal auto-encoder. Another explanation is that the optimization problem is much more difficult when there are more hidden layers, so the global minimum of the cost function is more difficult to find.

The one-hidden-layer auto-encoder showed the lowest reproducibility errors, while the five-hidden-layer auto-encoder showed the highest information compression. This suggests that when auto-encoding Coulomb matrices, a range of one to five hidden layers is optimal. Furthermore, adding hidden layers may be beneficial for increased compression. Due to the incomplete scanning of the parameter space, these trends cannot be confirmed; merely hypothesized.

In all Coulomb matrix auto-encoders, it was found that every top performing auto-

Table 5.13: Results of the top performing seven-hidden-layer auto-encoder for Coulomb matrices.

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.2506 Å^{-1}
Out-of-Sample Reproducibility Error	0.2441 Å^{-1}
Information Compression	77.29%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	83
Number of Neurons in 2nd Hidden Layer	125
Number of Neurons in 3rd Hidden Layer	371
Number of Neurons in 4th Hidden Layer	191
Number of Neurons in 5th Hidden Layer	91
Number of Neurons in 6th Hidden Layer	398
Number of Neurons in 7th Hidden Layer	474

encoder had a larger decoder than encoder. If future work is to be done exploring the parameter space, this work shows that smaller windows of neuron numbers per layer should be explored. For example, a range of 80-300 neurons for the hidden layers making up the encoder and a range of 300-500 neurons making up the decoder may produce better results for Coulomb matrix auto-encoders.

5.4 Auto-encoding Position Intracules

The results for position intracule auto-encoders are split up into two sections. In the first section, the number of hidden layers was fixed at three, and in the second section the number of hidden layers was fixed at five. For a discussion of one-hidden-layer auto-encoders trained on position intracules, see section 5.1. The results of three and five hidden layer auto-encoders show significantly lower errors than the one-hidden-layer auto-encoder, so these results will be discussed in greater detail.

5.4.1 Three-Hidden-Layer Auto-Encoders

The range of neurons for the latent representation was chosen to be 50-255 as the input layer is 256. This is the first trial in which a lower bound (greater than one) was given for the latent representation neuron count. In previous trials, no top performing auto-encoders had hidden layers with fewer than 50 neurons. Using this knowledge from previous trials with Coulomb Matrix and Cartesian auto-encoders as a guide, the choice was made to have 50 as a lower bound for the number of neurons in all hidden layers. Similar to the trial conducted in section 5.1, the minibatch size was chosen to be 50. More details on the auto-encoder in this trial can be seen in Table 5.14.

Table 5.14: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the three-hidden-layer auto-encoder trial

Neural Network Settings	Value
Activation Function	Exponential Linear Unit
Cost Function	Mean Squared Error
Auto-Encoders Trained in Trial	150
Number of Input/Output Neurons	256

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	50
Number of Hidden Layers	3
Learning Rate, η	0.001

Varied Hyperparameter	Range
Number of Neurons in 1st Hidden Layer	50-500
Number of Neurons in 2nd Hidden Layer	50-255
Number of Neurons in 3rd Hidden Layer	50-500

The top performing auto-encoder resulted in a small (relative to the Coulomb matrix auto-encoders) information compression of 12.50%. This may be an indication that intracules are more information dense than Coulomb matrices. Although it is tempting to rush to

Table 5.15: Results of the top performing three-hidden-layer auto-encoder for position intracules

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.6518
Out-of-Sample Reproducibility Error	0.4911
Information Compression	12.50%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	108
Number of Neurons in 2nd Hidden Layer	224
Number of Neurons in 3rd Hidden Layer	251

that conclusion, note that it may be dangerous to compare the two cases directly. Coulomb matrices have 841 neurons in the input layer, while position intracules have 256 input neurons. The way the neural networks have been set up may bias this conclusion. One of the advantages of intracules is that every molecule can be represented with the same amount of data. Coulomb matrices are smaller for smaller molecules, and so input neurons for the smaller molecules were partially filled with zeros. Therefore, it may not be a general result that Coulomb matrices are less information dense (or more easily compressible) as a result of this data. For example, for a database including only small molecules, Coulomb matrices may have a higher information density than intracules.

The lowest in-sample reproducibility error obtained while working with position intracules (0.6518) is a result of this trial. For larger molecules with probability density values on the order of $\sim$ 100, an in-sample reproducibility error of 0.6518 corresponds to a percent error of $\sim$ 0.65%. For smaller molecules, this percent error would be higher. It is difficult to say whether or not position intracules are reproducible within a tolerable error. Probability density values close to the maxima are likely reproducible, but information near the tail of the distribution (represented by much smaller probability density values) may be completely lost in this compression. For this reason, auto-encoders acting on position intracules may be more useful for studying specific molecular properties that are well represented by the maxima of the probability distribution, and less useful for studying molecular properties that are better represented by the tail of the distribution. Furthermore, larger molecules may be easier to study with this technique than smaller molecules.

An unexpected result in Table 5.15 is that there are far fewer neurons in the encoder than there are in the latent representation layer (108 compared to 224). This suggests that there may be a better way to define information compression in future work. Clearly, at one point in the neural network the molecular representation was compressed to 108 neurons, yet the latent representation was defined to be the middle layer, and therefore the information compression was defined to be solely based on the middle layer compared to the input layer. Two adjustments to the auto-encoder model will be proposed below; each providing a possible solution.

One way of ensuring the latent representation is defined as the most compressed layer is to impose a restriction for the hidden layers which mandates that they be at least as big as the input/output layers. Note that neural networks which obeyed this restriction were welcomed by this model, yet the smaller encoders were shown to be a property of the optimal auto-encoders. Another solution to this problem is to define the latent representation layer to be the smallest layer in the neural network once the size of each layer is optimized. This removes the symmetry mandate imposed earlier, and may be the best solution because it would provide the auto-encoder hyperparameters more flexibility. In hindsight, the symmetry requirement was not a neccessary restriction.

5.4.2 Five-Hidden-Layer Auto-Encoders

From the three-hidden-layer trial to the five-hidden layer trial, an increase in both reproducibility errors is observed, as can be seen in Table 5.17. Simultaneously, the information compression is increased from 12.50% to 29.30%. Both of these changes, as two hidden layers are added, are consistent with the trend observed in the Coulomb matrix auto-encoder trials. Again, this suggests that larger networks are better for compressing the data,

Table 5.16: Summary of fixed and varied hyperparameters, as well as additional neural network settings during the five-hidden-layer auto-encoder trial

Neural Network Settings	Value
Activation Function	Exponential Linear Unit
Cost Function	Mean Squared Error
Auto-Encoders Trained in Trial	206
Number of Input/Output Neurons	256

Fixed Hyperparameters	Value
Epochs Upper Bound	50
Minibatch Size	50
Number of Hidden Layers	5
Learning Rate, η	0.001

Varied Hyperparameter	Range
Number of Neurons in 1st Hidden Layer	50-500
Number of Neurons in 2nd Hidden Layer	50-500
Number of Neurons in 3rd Hidden Layer	50-255
Number of Neurons in 4th Hidden Layer	50-500
Number of Neurons in 5th Hidden Layer	50-500

Table 5.17: Results of the top performing five-hidden-layer auto-encoder for Position Intracules

Auto-Encoder Performance Metric	Value
In-Sample Reproducibility Error	0.8926
Out-of-Sample Reproducibility Error	0.6450
Information Compression	29.30%

Optimized Hyperparameters	Value
Number of Neurons in 1st Hidden Layer	240
Number of Neurons in 2nd Hidden Layer	329
Number of Neurons in 3rd Hidden Layer	181
Number of Neurons in 4th Hidden Layer	288
Number of Neurons in 5th Hidden Layer	288

while smaller networks result in lower reproducibility errors.

In this trial, the decoder was once again composed of more neurons than the encoder. This is the final trial completed in this work, so it can be said that all top performing auto-encoders showed this same trend.

6 Conclusions and Future Work

Theorists in quantum chemistry have shown that machine learning provides a powerful way of exploring the properties of chemical systems in cases where there are large amounts of data. One of the big challenges to this strategy lies in the molecular representation of the chemical systems. In this work, current leading candidates for molecular representations, such as atomic coordinates, Coulomb matrics, and position intracules were compressed using auto-encoders built from neural networks. This was done with the aim of producing a latent representation that would be maximally compact, translationally invariant, rotationally invariant, and unique.

The Coulomb matrix and position intracule auto-encoders proved to have lower in-sample and out-of-sample reproducibility errors than the auto-encoders trained on Cartesian coordinates. The latent representations produced by these two auto-encoders may help overcome the challenges of molecular representation for subsequent machine learning purposes.

For auto-encoders trained on Coulomb matrices, the highest information compression achieved was 93.94%. The in-sample reproducibility error for this compression was 0.2414 Å^{-1}. For auto-encoders trained on position intracules, the highest information compression achieved was 29.30%. The in-sample reproducibility error for this compression was 0.8926.

The auto-encoders developed in this work could be implemented on other forms of molecular representations as well. Any current molecular representation that can be converted into a single array has the potential to be compressed with the auto-encoders presented in this thesis.

6.1 Future Work

The latent representations in this work were created to be used by other machine learning algorithms which map chemical systems to their properties. Therefore, it is an immediate extension of this project to attempt to create a molecule-to-physical-observable mapping with other machine learning algorithms that use the latent representations created in this thesis as a starting block.

There is also more work that could be done in developing the auto-encoders. For the majority of the results in this thesis, hyperparameters were optimized with a random search technique by specifying a relatively broad range of values to be searched. The trends observed in this thesis could be used to make smaller windows for the optimization of hyperparameters. It is likely that higher information compression values and lower reproducibility errors could be obtained by further exploring the parameter space for quantities such as number of neurons in a hidden layer.

Bibliography

[1] R. Nasr, P. B., D. Hirschberg *Journal of Chemical Information and Modeling* **2010**, *32*, 1358 – 1368.

[2] J. Chen, Y. D.-J. B.-P. B., S. J. Swamidass *Bioinformatics* **2005**, *21*, 4133–4139.

[3] J. J. Irwin, B. K. S. *Journal of Chemical Information and Computer Sciences* **2005**, *45*, 177–182.

[4] J. Chen, S. J. S.-D. W.-P. B., E. Linstead *Bioinformatics* **2007**, *23*, 2348–2351.

[5] Y. Wang, T. S. J. Z. J. W. S. B., J. Xiao *Nucleic Acids Research* **2009**, *37*, W623–W633.

[6] E. Sayers, D. B. E. B. S. B. K. C. V. C. D. C. M. D. S. F., T. Barrett *Nucleic Acids Research* **2010**, *38*, D5–D16.

[7] Dinkel, H. e. a. *Nucleic Acids Research* **2011**, *39*, D261–D267.

[8] K. Nagata, P. B., A. Randall *Bioinformatics* **2014**, *30*, 1681–1689.

[9] Newton, I. *Isaac NewtonâĂŹs Philosophiae Naturalis Principia Mathematica, Harvard University Press, Cambridge, MA* **1972**,

[10] Griffiths, D. J. *Introduction to Quantum Mechanics*, 2nd ed.; Pearson Education, Inc: Upper Saddle River, NJ, 2005.

[11] Levine, I. *Quantum Chemistry*, 6th ed.; Pearson Education, Inc: New Jersey, 2009.

[12] Cramer, C. J. *Essentials of Computational Chemistry: Theories and Models*, 2nd ed.; Wiley: Hoboken, New Jersey, 2004.

[13] Kenny Lipkowitz, O. I., Thomas R. Cundari *Reviews in computational chemistry*; Wiley-VCH: Hoboken, New Jersey, 2007; Vol. 23; p 1413.

[14] Hornik, K.; Stinchcombe, M.; White, H. *Neural Networks* **1989**, *2*, 359 – 366.

[15] Cybenko, G. *Mathematics of Control, Signals, and Systems* **1989**, *2*, 303–314.

[16] Hansen, K.; Montavon, G.; Biegler, F.; Fazli, S.; Rupp, M.; Scheffler, M.; von Lilienfeld, O. A.; Tkatchenko, A.; MÃijller, K.-R. *Journal of Chemical Theory and Computation* **2013**, *9*, 3404–3419, PMID: 26584096.

[17] Shivaswamy, P. K.; Jebara, T. **2006**, 817–824.

[18] Rupp, M.; Tkatchenko, A.; Müller, K.-R.; von Lilienfeld, O. A. *Phys. Rev. Lett.* **2012**, *108*, 058301.

[19] Rupp, M. *International Journal of Quantum Chemistry* **2015**, *115*, 1058–1073.

[20] Moussa, J. E. *Phys. Rev. Lett.* **2012**, *109*, 059801.

[21] Montavon, G.; Hansen, K.; Fazli, S.; Rupp, M.; Biegler, F.; Ziehe, A.; Tkatchenko, A.; Lilienfeld, A. V.; Müller, K.-R. In *Advances in Neural Information Processing Systems 25*; Pereira, F., Burges, C. J. C., Bottou, L., Weinberger, K. Q., Eds.; Curran Associates, Inc., 2012; pp 440–448.

[22] Gill, P. M. W.; Crittenden, D. L.; O'Neill, D. P.; Besley, N. A. *Phys. Chem. Chem. Phys.* **2006**, *8*, 15–25.

[23] Thakkar, A. J. In *Density Matrices and Density Functionals*; Erdahl, R. M., Smith, V. H., Jr., Eds.; Reidel: Dordrecht, Holland, 1987; pp 553–581.

[24] Coulson, C. A.; Neilson, A. H.

[25] Hennessey, D. C.; Sheppard, B. J. H.; Mackenzie, D. E. C. K.; Pearson, J. K. *Phys. Chem. Chem. Phys.* **2014**, *16*, 25548.

[26] Felix Faber, O. A. v. L. R. A., Alexander Lindmaa *International Journal of Quantum Chemistry* **2015**, *115*.

[27] Ramakrishnan, R.; Dral, P. O.; Rupp, M.; von Lilienfeld, O. A. *Scientific Data* **2014**, *1*.

[28] L. Ruddigkeit, L. C. B. J.-L. R., R. van Deursen *Journal of Chemical Information and Modeling* **2012**, *52*, 2864–2875.

[29] Montavon, G.; Rupp, M.; Gobre, V.; Vazquez-Mayagoitia, A.; Hansen, K.; Tkatchenko, A.; Müller, K.-R.; von Lilienfeld, O. A. *New Journal of Physics* **2013**, *15*, 095003.

[30] Lopez-Bezanilla, A.; von Lilienfeld, O. A. *Phys. Rev. B* **2014**, *89*, 235411.

[31] Schütt, K. T.; Glawe, H.; Brockherde, F.; Sanna, A.; Müller, K. R.; Gross, E. K. U. *Phys. Rev. B* **2014**, *89*, 205118.

[32] C.-Y. Liou, J.-C. H.; Yang, W.-C. *Neurocomputing* **2008**, *71*, 3150–3157.

[33] C.-Y. Liou, J.-W. L., W.-C. Cheng; Liou, D.-R. *Neurocomputing* **2014**, *139*, 84.

[34] Jonathan Romero, A. A.-G., Jonathan P. Olson *arXiv* **2017**,

[35] Diederik P. Kingma, J. B. *arXiv* **2017**,

[36] Djork-ArnÃl' Clevert, S. H., Thomas Unterthiner *arXiv* **2016**,

[37] Chollet, F. **2015**,

[38] Rami Al-Rfou, A. A.-C. A.-D. B. N. B. F. B. J. B. A. B. A. B. Y. B. A. B. J. B. V. B. J. B. S. N. B. N. B.-L. X. B. A. d. B. O. B. P.-L. C. K. C. J. C. P. C. T. C. M.-A. C. M. C. A. C. Y. N. D. O. D. J. D. G. D. S. D. L. D. M. D. V. D. S. E. K. D. E. Z. F. O. F. M. G. X. G., Guillaume Alain *arXiv* **2016**,