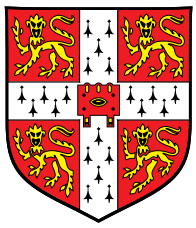


University of Cambridge

Machine Learning Landscapes for Neural Networks with a Single Hidden Layer

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the MPhil in Scientific Computing.

Sathya R. Chitturi

August 2019

Downing College

Supervisor: David J. Wales

Declaration

This dissertation is substantially my own work and conforms to the University of Cambridge's guidelines on plagiarism. Where reference has been made to other research this is acknowledged in the text and bibliography. This work contains fewer than 15,000 words including footnotes, tables and equations (verified via TEXcount).

Sathya Chitturi

August 2019

Acknowledgements

I thank Professor David Wales for his constant supervision and support. I am also grateful to Wei Kang, Samuel Coward and Dr. Songul Guryel Karasulu for helpful discussions regarding the group code. Sponsorship from the Maxwell Center for Scientific Computing and the Department of Chemistry is also acknowledged.

Abbreviations

AUC-ROC	Area Under Curve - Receiver Operating Characteristics
CIFAR-10	Canadian Institute For Advanced Research - 10 Dataset
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DNEB	Doubly-nudged Elastic Band
DPS	Discrete Path Sampling
EL	Energy Landscape
EM	Expectation Maximization
FC	Fully-connected
GPU	Graphics Processing Unit
H-EF	Hybrid Eigenvector-following
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno
MNIST	Modified National Institute of Standards and Technology Database
NM	Newton's Method
NN	Nearest-Neighbour
PES	Potential Energy Surface
ResNet-32	Residual Network - 32
RMS	Root Mean Square
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
XOR	Exclusive Or

Abstract

We study the effects of dataset mislabelling, training set diversity, and reduced node-connectivity on neural network loss landscapes, using various techniques developed for energy landscapes exploration. To complement these methods, we introduce a GPU neural network implementation which accelerates global optimization by approximately an order of magnitude.

The relevant training data includes custom generated geometry optimization datasets (D1.2-D3.0), as well as the MNIST digits dataset. For the D1.2-D3.0 datasets, we find that the number of stationary points increases with the size of the atomic configuration space, suggesting a correspondence between the number of local minima and statistical uncertainty.

In addition, we find that neural networks can effectively filter uniform random noise and show, numerically, that this result holds for both the average (sampled) local minima and the training global minimum. In addition, the variance of the testing AUC, computed over a sample of low-lying minima, grows significantly with the training error. We visualize these landscapes using disconnectivity graphs, topological maps of the surface, which provide interesting insights into the role of the bias-variance trade-off when training under noise.

Finally, we investigate the effects of reduced-connectivity on neural network landscapes. We find that the presence of locality in these networks can yield complex, frustrated landscapes which contain many high capacity minima. Further, our analysis suggests that, on a landscape level, neural networks may balance sparsity and expressiveness to perform well on unseen testing datasets.

This work helps shed further light on neural network loss landscapes and will likely be relevant for future work on neural network training and optimization.

Contents

1

Introduction

1.1 Background

Artificial neural networks are an important class of machine learning techniques [1,2] which have widely implemented in fields such as computer vision [3–5], natural language processing [6], finance [7], and robotics [8]. Neural networks can have very different architectures, ranging from simple feed-forward fully connected networks (FCs) featured in this dissertation, to more sophisticated models including Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) [1].

For simple FC architectures, a neural network is a mapping from input data ($\mathbf{x}$) to output predictions (output space) and consists of sequential linear layers with non-linear activation nodes (σ_i) (Eqn. 1.1). The parameters of a neural network are the weights ($\mathbf{W}_i$) that connect these layers, which are determined during a process known as training [9].

$$f(\mathbf{x}) = \sigma_{N+1}(\mathbf{W}_N\sigma_N(\mathbf{W}_{N-1}\sigma_{N-1}(...\sigma_1(\mathbf{W}_1\mathbf{x})))) \tag{1.1}$$

Training a neural network, like other optimization methods in statistics, involves minimizing a loss function with respect to the parameters of the model [9]. This loss function measures the deviation between the predicted value of the function and the provided labelled data. Many different loss functions are possible; modified versions of the mean-squared-error or cross-entropy losses are typically

used for most machine learning applications [9].

Current state-of-the-art neural networks can contain thousands of layers with millions of hidden nodes, defining the loss function in very high dimensional parameter space. Importantly, unlike other popular techniques, such as Linear Regression and Support Vector Machines, the neural network cost function is non-convex, meaning that it is extremely hard (NP hard) to find the global minimum [9, 10]. Currently, neural network optimization involves variations on the steepest-descent method, such as in Adam or RMSprop [11]. However, these algorithms typically do not usually converge to (true) local minima, let alone the global minimum. Generally, the fitted parameters are obtained at the point in the minimization sequence where the deviation between subsequent loss values is smaller than a fixed threshold [11]. Furthermore, in order to optimize for computational speed, these methods calculate the gradient stochastically, rather than using the entire training set.

The primary goal of training a neural network is to obtain a solution (minimum) that generalizes well to an unseen testing set [9]. For simplicity, consider a scalar model where $f(x)$ is the true target distribution and $\hat{f}(x)$ is the approximating function to be learned during training. Then y, as defined in Eqn. 1.2, is composed of the sum of the true distribution, $f(x)$, and an irreducible error term, ϵ.

$$y = f(x) + \epsilon \tag{1.2}$$

The generalization gap between $f(x)$ and $\hat{f}(x)$ is often quantified by $\mathrm{E}\left[\left(y - \hat{f}(x)\right)^2\right]$, which is composed of three terms – the bias, variance and irreducible error (Eqn. 1.3) [9].

$$\mathrm{E}\left[\left(y - \hat{f}(x)\right)^2\right] = \left(\mathrm{E}\left[\hat{f}(x)\right] - f(x)\right)^2 + \left(\mathrm{E}[\hat{f}(x)^2] - \mathrm{E}[\hat{f}(x)]^2\right) + \sigma^2 \tag{1.3}$$

The irreducible error, captures the inherent dataset noise, and is often an unknown component of training (which can be difficult to estimate). The bias

term measures the extent to which the model capacity is sufficient to capture the complexity of the true function. High bias models limit the capacity to completely model the training data (underfitting); however, this property can make them robust to unseen testing sets. On the other hand, the variance term measures the amount of noise (ϵ) the approximating function models when trying to estimate the true function. Models with high variance, therefore, perform well in training, but generalize poorly (overfitting). Although it seems as though minimizing either the bias or the variance would reduce the generalization gap (Eqn. 1.3), in practice these two components are inversely related (bias-variance trade-off) [9]. Thus, neural networks that do generalize well often effectively balance bias and variance.

Network network training, in particular, often occurs in the overparameterized limit, in which there are significantly more parameters than training data. Based on this fact, according to established statistical learning theory, one would expect neural networks to have high variance, overfit to the training data and generalize poorly. Surprisingly, however, it has been observed that neural networks can generalize well, despite the huge degree of overparameterization [12–14]. The motivation for this dissertation is to see whether studying the underlying neural network loss landscape, in terms of connectivity between local minima and saddle points, can help explain why neural networks are able to perform so well. Furthermore, we aim to understand how systematic perturbations to neural network architecture and data affect the resulting landscape and, more broadly, statistical generalizability.

1.2 Literature Review

This section provides a brief review of a number of areas addressed in this dissertation. Specifically, Section 1.2.1 introduces the idea of a machine learning energy landscape and summarize previous work studying its properties. Sections 1.2.2 and 1.2.3 summarize previous experiments and results involving dataset mislabelling and reduced-connectivity that we build on from an energy landscapes perspective. Finally, Section 1.2.4 presents past work on porting optimization

methods and neural network potentials to the GPU. These references serve as background for our own GPU implementation.

1.2.1 Neural Network Loss Landscapes

The cost function of a neural network can be visualized as a loss landscape in high-dimensional parameter space. This surface is highly dependent on the choice of non-linear activation function, architecture and loss metric (squared loss, cross-entropy etc.). Recently, there has been much interest in the structure of the loss landscape and its relation to notions of generalizability, sensitivity to initialization schemes, and choice of optimizer [12–14]. In theory, specific knowledge about the neural network loss landscape could improve initialization schemes, training speed, and quality of solutions which may aid in improving the prediction power of neural networks.

Unfortunately, direct calculation of the loss landscape is impossible due to issues of computational complexity. It is worth noting that even more modest tasks such as just finding one local minimum of an arbitrary neural network, is an NP hard problem [15]. Furthermore, based on much previous work, it is known that the number of stationary points grows exponentially with the dimensionality of the problem [10]. This can prove tricky for first- and second-order methods, such as stochastic gradient descent (SGD) and Newton's method (NM), as these algorithms can get trapped in local areas of high training loss [10]. For this reason, some authors have chosen to perform theoretical studies, backed by numerical simulations, to study the realtionship between the loss landscape and generalizability. Choromanska et al. consider the performance of various local minima for neural network optimization [14]. A good performance, by their metric, corresponds to obtaining very high accuracies on both an independent training and test set. Note that in the overparameterized limit, a neural network overfits the data and in this case the training accuracy should be very high. Thus, a poor performance would typically be characterized by having a local minimum with a low training error, but a high testing error. The authors show that theoretically, subject to a number of assumptions of independence, a neural network

optimization reduces to minimizing the energy of the spin-glass hamiltonian from statistical physics [14]. Based on the spin-glass model, bounds can be derived showing that there exists a tight band of local minima, bounded above the global minimum, which is characterized by having minima with low training and testing errors. Furthermore, it is exponentially less likely to find a minimum with relatively high testing error as the dimensionality of the neural network grows [14]; in other words, almost any local minimum that is found via standard optimization techniques should perform comparably to any other local minimum on an unseen test set. In fact, the authors claim that finding a local minimum solution may be even better than finding the global minimum, as it might introduce a degree of regularization [14].

Wu et al. agree with the conclusion that the majority of local minima solutions of the loss landscape tend to have properties similar to that of the global minimum [13]. This work asserts that neural networks are able to generalize well because they yield simple solutions (minima) with small Hessian norm. A theoretical analysis of two-layer networks suggests that these simple solutions occur because the volumes of the basins of attraction for minima with high test error are exponentially dominated by the volumes of the basins of attraction for minima with low test errors. In other words, good solutions lie in large, flat regions of parameter space and bad solutions lie in small, sharp regions [13]. These claims were supported via numerical simulations using the ResNet-32 architecture on the CIFAR-10 dataset [3] with a ratio of parameters to training examples of more than 100. The volume of attraction was approximated using the sum of the largest k eigenvalues of the Hessian matrix ($\mathbf{H}$). Li et al. propose a filter-wise normalization scheme to preserve scale invariant properties of neural networks, which allows for comparison between different architectures and landscapes [16]. Low-dimensional 2D contour plots are created capturing the value of the loss function along random directions (sampled from a multivarite Gaussian) near respective minima; these visualizations are supplemented by curvature information using 2D heatmaps of the lowest and highest Hessian eigenvalues. By studying a variety of different architectures on the CIFAR-10 dataset, Li et al. suggest that

flat minima (large basins) tend to generalize better than sharp minima (small basins). Furthermore, shallow, wide neural networks have contour surfaces with a convex appearance, which might make them more generalizable. For instance, a ResNet with 20 layers has a very convex landscape and is easy to train [16]. Conversely, very large deep networks can have chaotic appearances around local minima, which can make them untrainable [16]. Nguyen et al. agree and show that if a network has a pyramid-like structure following a very wide layer, then local minima are very close the global minimum and the surface is much easier to optimize [17].

However, the above investigations of the loss landscape suffer from many problems. The assumptions made in the theoretical modelling [13, 14] are strong and unlikely to hold in practice. Furthermore, low-dimensional representations of the landscape, as in [16], can miss the underlying non-convexity present in higher dimensions. For instance, it is possible to manipulate the dataset and optimization problem to create solutions with very high training accuracy but very low testing accurary; this manipulation can be done by adding a tuneable attacking term to the cost function and deliberately missassigning labels during training [13]. In addition, it is possible to arbitrarily create datasets in which specific initialization schemes will either not converge or converge to high-lying loss solutions [18]. This construction is achieved by carefully choosing inputs, such that most of the neural network nodes are zero during training [18].

In order to avoid the problems of projecting into low dimensions or making restrictive theoretical assumptions, the present work utilizes the energy landscapes approach to non-convex optimization. The energy landscape formalism has previously been used to study the non-convex structure of neural network landscapes [19–22]. Minima of the loss function are found by using basin-hopping global optimization with local minimization via L-BFGS [23–30] (Section 2.2.1). Transition states are found using the using the doubly-nudged [31, 32] elastic band [33, 34] approach with hybrid-eigenvector following [35–37] (Section 2.2.2). Using discrete path sampling, previous studies have created topological maps [38–40] corresponding to pathways between minima via transition states for vari-

ous neural network datasets, including molecular geometry optimization [19, 20], hand-written digit recognition [19], the logical XOR operation [22] and patient outcomes [21].

For geometry optimization and hand-written digit recognition, Ballard et al. show that the majority of high testing AUC minima are concentrated in a region of low training loss [19, 20], which is in line with other work [13, 14]. Furthermore, for all single-layered architectures, the corresponding landscapes are relatively convex, exhibiting a single-funnelled structure. In contrast, recent work on neural networks with multiple hidden layers, shows that, for the same number of optimizable parameters, it is possible to obtain much more complex multi-funnelled landscapes [41]; interestingly, these landscapes are greatly simplified with large amounts of (independent) training data [41]. Another report analyses a simple XOR model and shows that the number of minima depends strongly on the amount of **L2** regularization [22]; as the regularization is increased, the loss landscapes become more convex and easy to optimize. Furthermore, these networks have high capacity and are over-specified, and many of the minima learn sparse representations (i.e. many of the weights are zero for the optimized model) [22] .

1.2.2 Mislabelling

Many real datasets of interest have significant label noise. The source of this noise can arise from difficulties in the data cleaning and aquisition processes, or simply from ambiguous class differentiation criteria [42–44]. Additionally, to reduce acquisition costs, many practitioners prefer to obtain large amounts of slightly noisy data, rather than small amounts of perfectly clean data, for example, in massive crowd sourcing type procedures. While this scenario allows for the creation of much larger labelled training sets, it also has the potential to greatly deteriorate label quality [42, 43, 45]. In light of the positive advantages of acquiring cheap data, much effort has been dedicated to improving the training of neural networks under noise. These methods typically involve either pre-processing filtration schemes or models which explicitly consider noise during

training.

Considering the latter schemes, Bekker et al. use a modified expectation-maximization (EM) algorithm to estimate the correct labels of a dataset using the current iteration parameters [45]. Using this estimate, the authors build a model of the noise distribution $\theta(i, j) = p(z = j|y = i)$, where z is the incorrect label and y is the correct label. The corresponding likelihood function incorporates both the noise model and the parameters of the neural network, at no extra computational expense. The article shows that for uniform random label noise on the MNIST dataset, in which a fixed proportion of the training data labels are permuted to another class with equal probability, the testing accuracy is only slightly reduced. Specifically, Bekker et al. observe testing accuracies of 0.88 for datasets with more than 60 % noise [45]. For unknown stochastic noise, however, the baseline model performs much more poorly (testing accuracy of 0.2 at 60 % noise). The noise minimization model, on the other hand, performs significantly better, obtaining a testing accuracy of 0.4 at the 60 % noise threshold [45].

Similarly, Rolnick et al. show that neural networks can actually generalize surprisingly well under uniform random noise [42]. With enough correct training data, and a ratio of 100:1 incorrect to correct examples, neural networks can still obtain high testing accuracies on commonly used datasets, such as MNIST. The authors suggest that this phenomenon occurs because of a filtering effect due to favorable gradient cancellation [42]. This conjecture is supported by the observation that increasing the batch size of their stochastic optimizer decreases the testing error, since a gradient averaged over more training points will exhibit more cancellation. The authors posit that this trait might be present in the underlying structure of the loss landscape.

1.2.3 Reduced-Connectivity

Reducing the connectivity between neural network nodes is commonly used as a method to improve the generalizability of standard architectures [46]. In particular, the neural network DropOut procedure [46] involves sampling many less expressive networks by stochastically setting weights to zero during training. By

using many architectures, individual neural network nodes learn to perform well, independent of their global connectivity. Furthermore, since these models have less bias, they have less capacity and do not tend to overfit to noise. In effect, DropOut schemes can be conceptualized as a form of stochastic regularization. The original dropout model is still highly effective and, combined with a CNN, made the winning entry for the 2012 ImageNet challenge [47]. A similar model, with equally successful results is DropConnect [48]. In this formulation, weights (as opposed to nodes) are randomly dropped to prevent co-adaptation [48]; the metric we introduce in Section 2.5.1 follows the frozen weights regime. Since the initial development of these models, many other variants have been proposed [49].

In addition to regularization, there is also interest in reduced-connectivity models from the compressed sensing point of view. It has previously been demonstrated that highly overparameterized models can be replaced by equally good models which have much smaller (orders of magnitude) number of optimizable parameters [50]. Since large modern neural networks often have millions of parameters, methods which have sparse connectivity, but yet generalize well, are particularly attractive. LeCun et al. propose a reduced-connectivity metric based on saliency [51]. In this formulation, a fully-connected network is trained, less-important parameters are frozen, and then the network is re-optimized. The less explanatory parameters are identified using second-derivation information calculated using fast back-propagation. Using this formalism, the authors were able to reduce the number of optimizable parameters by factor of two and also improve generalizability. Recently, Changpinyo et al. developed a compression method, based on using sparse connectivity matrices, to reduce the computational burden of training large CNNs [52]. This sparse model is able to perform as well as the corresponding dense network on standard benchmark training sets such as MNIST, CIFAR-10 and ImageNet. Furthermore, Changpinyo et al. suggest that sparsely-connected CNNs can even promote expressibility, relative to the dense network, by perturbing the large degree of permutation symmetry [52].

1.2.4 Neural Networks on the GPU

Fast computation of the analytical neural network loss and gradient is an extremely important part of the optimization procedure. For this reason, most researchers use General Purpose Graphical Processing Units (GPUs) to train their models [53]. The reason for this trend is that typical neural network architectures involve multiple steps of high arithmetic intensity, large matrix multiplication operations, which are trivially parallelizable on a GPU [54]. These parallel algorithms, as well as much more complicated architectures and optimizers, are extensively utilized in open-source deep learning frameworks such as TensorFlow, Theano and Pytorch, and have allowed researchers to achieve one- to two-orders increase in computational speed [54–56]. In one study, Sierra et al. saw a 50-fold increase in speed to calculate the cost function and gradient of a 3-layered perceptron on the GPU [56]. Here, the NVIDIA CUDA language with the cuBLAS library [57], a parallel CUDA version of the popular BLAS library, was used to implement parallel matrix multiplication and the dot product operation.

In addition to calculating potential functions on the GPU, some work has been dedicated to implementing non-linear optimization methods, such as L-BFGS, on the GPU [30, 58, 59]. The rational for this setup is that porting the entire optimization algorithm to the GPU may avoid bottleneck memory transfer steps from the CPU [30,58,59]. A parallel L-BFGS algorithm was recently implemented and applied to the problem of centroidal Voronoi tessellation and achieved an order of magnitude speedup (vs. CPU) [60]. Another version of the method was implemented and applied to various protein energy landscapes [30] (Section 2.2) and also achieved order of magnitude speedups for large system sizes.

1.3 Thesis Overview

This work explores machine learning energy landscapes for perceptrons with a single hidden layer. Building on previous work [19–22], we use EL methods to further study neural network loss functions. While these previous studies have primarily explored the landscapes of ideal neural networks, in contrast, the present work

focuses on investigating the effect of introducing systematic changes in molecular configuration space diversity, label accuracy, and node-connectivity on the underlying neural network landscape. Specifically we vary the size (diversity) of the dataset configuration space, utilize a DropConnect-inspired method for reducing node-connectivity, and introduce uniform random label errors into each of our datasets. Finally, recognizing the computational difficulty of studying neural network energy landscapes, we also implement and benchmark a CUDA neural network potential for basin-hopping global optimization.

2

Methods

This work explores machine learning landscapes of single-layered neural networks[1] under systematic changes to dataset labels, molecular configuration space volume and reduced-connectivity. Section 2.1 describes the four datasets used in this dissertation (Section 2.1.1) and presents the original formulation [19, 20] of the neural network potential (Section 2.1.2). Section 2.2 describes the relevant energy landscape methods for global optimization (Section 2.2.1), transition state searching (Section 2.2.2), and visualization (Section 2.2.3). Section 2.3 describes the formulation (Section 2.3.1), implementation (Section 2.3.2), and testing (Section 2.3.3) of the GPU implementation introduced in this work. Finally, Sections 2.4 and 2.5 describe the numerical procedure used to study neural networks under mislabelling and reduced node-connectivity. Note, all serial experiments were performed on an Intel i7-8700 CPU machine (3.20GHz). All parallel global-optimization experiments used GeForce GTX TITAN Black GPUs running NVIDIA CUDA 8.0. In addition, please note, some of the material presented here has been adapted from my two projects for the course (MPhil Mini-Projects 1 and 2).

[1] In this work, single-layered neural networks refer to perceptrons with a single hidden layer

2.1 Machine Learning Model

2.1.1 Datasets

We explored the machine learning energy landscape of four datasets (D1.2-D3.0 and MNIST). Three of these datasets (D1.2-D3.0) correspond to a triatomic geometry optimization problem with varying volumes of molecular configuration space; these datasets were studied using the serial neural network implementation (Section 2.1.2). The fourth dataset was the commonly used digit-recognition database, MNIST. This problem was studied using the GPU implementation introduced in the present work (Section 2.3).

The D1.2, D2.0 and D3.0 datasets consist of bond lengths (r_{12}, r_{13} and r_{23}) for a triatomic cluster bound by a Leonard-Jones [61] plus Axilrod-Teller [62] potential. The datasets were generated by performing geometry optimization at 200,000 random points in molecular configuration space to find the resulting local minima (class labels). The machine learning classes correspond to four physical structures – namely three linear minima and one triangular minimum [19–21, 63]. The D1.2, D2.0 and D3.0 datasets differ by the volume of molecular configuration space the data is drawn from (D1.2 being the smallest). Here, it is worth mentioning that the distribution of outcomes varies with the size of the configuration space. For example, unsurprisingly, the most compact dataset (D1.2) contains a large number of equilateral triangle minima (class 0). Since we cannot uncouple the outcome distribution from the choice of configuration space in an unbiased manner, we studied all relevant properties using the size of the configuration space as an extra variable parameter; please see Appendix A for the outcome distributions. Furthermore, having three datasets generated in this way allowed us to systematically examine the effects of dataset diversity on the resulting landscapes.

The MNIST dataset is a standard benchmark dataset used for various machine learning applications [64]. The training data consists of 28×28 pixel images of hand-written digits between 0 and 9. Note, in order to use this dataset, the input training images were reshaped into a 784-dimensional vector.

2.1.2 Architecture

We used a single-layered neural network model with explicit weights for bias nodes. The outputs of the model (Eqn. 2.1) are discrete classes corresponding to each of the four minima for the D1.2-D3.0 datasets, and to each of the ten digits for MNIST. Here y_i is the i^{th} output of the neural network (Eqn. 2.1). In this formulation, a smooth hyperbolic tangent function (tanh) is used as the activation function. This function is chosen to ensure that the overall loss function has a smooth analytical gradient and Hessian with respect to the trainable weights. Note, we also choose tanh to satisfy the universal approximator theorem, which asserts that neural networks with a single hidden layer and enough hidden nodes can abitrarily approximate any function [65].

$$y_i = w_i^{bo} + \sum_{j=1}^{N_{hidden}} w_{ij}^{(1)} \tanh(w_j^{bh} + \sum_{k=1}^{N_{in}} w_{jk}^{(2)} x_k). \tag{2.1}$$

We apply a softmax activation function to y_i. This function takes the output of a neural network and converts it into a vector containing class identity probabilities (Eqn. 2.2):

$$p_c(\mathbf{w}; \mathbf{x}) = \frac{e^{y_c}}{\sum_a^{N_{out}} e^{y_a}}. \tag{2.2}$$

To deal with multi-class classification, we used a one-hot encoding scheme with a categorical cross-entropy loss [19, 20] (Eqn. 2.3). Here $\mathbf{w}$ contains the weights of the model ($w_{jk}^{(2)}$, $w_{ij}^{(1)}$, w_j^{bh} and w_i^{bo}), and $\mathbf{x}$ stores the input features (x_k). The model also includes an **L2** penalty term to reduce overfitting, with the hyperparameter λ, controlling the weight of the regularization [19,20] (Eqn. 2.3). In addition, the regularization term also facilitates transition states searches by shifting any zero eigenvalues of the Hessian matrix [22].

$$E(\mathbf{w}; \mathbf{x}) = -\frac{1}{N_{data}} \sum_{d=1}^{N_{data}} \log(p_{c(d)}(\mathbf{w}; \mathbf{x})) + \lambda \mathbf{w}^2. \tag{2.3}$$

In the present work, we use a short hand, [**A**,**B**,**C**,**D**,**E**], to refer to single-layered architectures with **A** inputs, **B** hidden nodes, **C** outputs, **D** training

data and a regularization constant of **E**. For the empirical work of this dissertation, we used three different single-layered neural network architectures. For the D1.2-D3.0 datasets, we used a [2,10,4,1000,0.0001] model (74 optimizable parameters). In addition, for the nearest-neighbour experiments, we additionally used a [2,5,4,1000,0.00001] architecture (39 optimizable parameters). Finally, for the MNIST dataset, we used a [784,10,10,1000,0.1] architecture (7960 optimizable parameters).

2.2 Energy Landscape Methods

2.2.1 Global Optimization

To find low-lying minima for the architectures described in Section 2.1, we use the basin-hopping global optimization algorithm GMIN, used in molecular science, to minimize high-dimensional potential energy functions in atomic configuration space [29, 66]. This method has been used succesfully to find global minima of atomic clusters, glasses and proteins [29, 66–68]. By making a correspondence between the potential energy surface (PES) and the neural network loss function, where the atomic configuration space becomes the neural network parameter space, we apply this method directly to study neural network energy landscapes [19, 20].

The GMIN algorithm operates via hypersurface deformation followed by local minimization and random structural perturbations. The hypersurface deformation step involves transforming the surface into a series of plateaux and basins of attraction by replacing the energy (E) of any point in atomic configuration space by the energy ($\widetilde{E}$) obtained via local minimization from that point ($\mathbf{X}$) [29]. This transformation is described mathematically in Eqn. 2.4 and depicted visually in Figure 1.

$$\widetilde{E} = \min(E(\mathbf{X})). \tag{2.4}$$

Importantly, the transformation in Eqn. 2.4 eliminates all saddle points but, by definition, preserves the energies of the local minima. This new surface makes

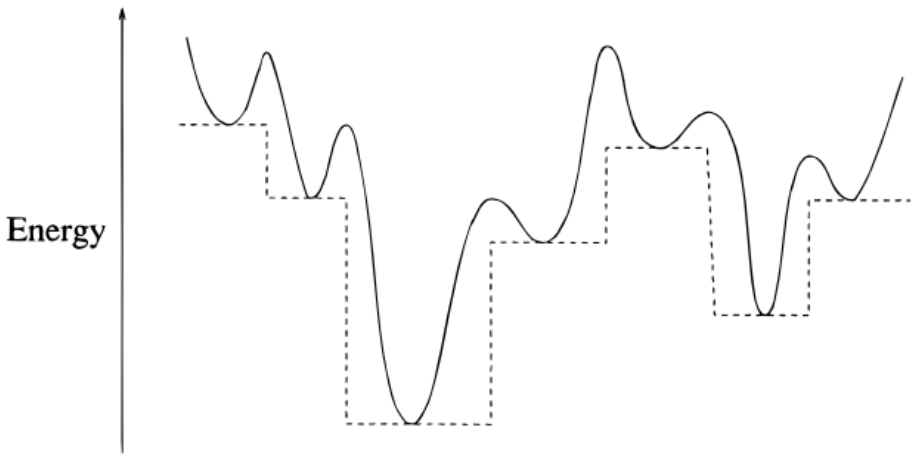


Figure 1: Representation of the basin-hopping energy landscape transformation; the dotted line is the transformed landscape. This figure was reproduced with permission from reference [66].

it easier to hop between minima in different basins, as the transition does not just have to occur at the transition state [29].

Local minimization (quenching) is performed using a custom limited memory [23, 24] version of the Broyden [25], Fletcher [26], Goldfarb [27], Shanno [28] (L-BFGS) algorithm, which is a rank-2 pseudo-Newton method that recursively approximates the Hessian matrix using information from the energy and gradient from a user-specified number of update steps [69]. In particular the Hessian is implicitly approximated using vectors, instead of by storing and using large matrices [30]. Recently, a CUDA version of L-BFGS was designed for global optimization and is described extensively in [30].

Having quenched to a local minimum, the GMIN method finds the next candidate minimum using stochastic parameter perturbations. Moves are generally determined using a Metropolis test for the potential energy [70]. If the new minimum lies lower in energy than the previous one, it is accepted; however, if it is higher in energy, it is accepted with an exponentially small probability which depends on a fictitious temperature (T):

1. **accept if** $E_{new} < E_{old}$ **or**

2. **accept if** $E_{new} > E_{old}$ and $\exp(\frac{E_{old}-E_{new}}{kT}) > (X \sim U[0,1])$

2.2.2 Saddle Point Searching

In addition to finding minima, we also use EL methods to find transition states, saddle points with Hessian matrices containing exactly one negative eigenvalue (all others non-negative), which connect pairs of minima [71]. Although many methods exist to find minima, transition state searching remains a challenging optimization problem [31, 37, 72, 73].

Transition state candidates are determined using the doubly-nudged [31, 32] elastic band [33, 34] (DNEB) approach, which involves approximating local surface curvature by minimizing a path containing a series of images connected by a harmonic potential. These candidates are refined using hybrid eigenvector-following [35–37] (H-EF), which modifies the Newton-Rhapson step [72], to perform systematic energy maximization along just one Hessian eigenvector. Having determined a candidate transition state, the connected minima are located by minimisation following small displacements along the eigenvector corresponding to the unique negative eigenvalue. This method can be employed to create databases of connected local minima [73], which are analogous to kinetic transition networks [74–76].

In this work, performance of neural networks is evaluated using standard Area Under Curve (AUC) metrics [19]. The AUC is calculated by determining the true positive and false positive statistics for the machine learning problem. The true positive rate (T_{pr}) is defined as the ratio of the number of examples correctly predicted to be in a given class to the total number of examples in that class. The false positive rate (F_{pr}) is similarly defined as the ratio of the number of false positives to the total number of examples in the other class. The T_{pr} is integrated as a function of F_{pr} from 0 to 1 to obtain the AUC (Eqn. 2.5).

$$\mathrm{AUC} = \int_0^1 \mathrm{T}_{pr} \mathrm{dF}_{pr}. \tag{2.5}$$

2.2.3 Discrete Path Sampling and Visualization

The program PATHSAMPLE, which acts as a driver for OPTIM, is used to create databases of local minima and transition states via discrete path sampling (DPS).

This approach allows us to quantify the number of stationary points (for fully sampled systems) on the energy landscape.

Visualization of the landscape was performed using disconnectivity graph analysis [38–40]. This approach bins the energy landscape into discrete energy basins containing minima that can interconvert below each energy threshold, via a sequence of transition states. Interconversion between minima is considered feasible if they are connected via a transition state. Using this topological method, an undirected tree is constructed [40]. This approach has previously been applied to visualize the energy landscape of atomic clusters, glasses and proteins [29].

2.3 CUDA Implementation

2.3.1 Formulation

To perform global optimization, it is necessary to be able to efficiently calculate the cost function and gradient of the neural network potential [30]. Importantly, the computational difficulty of training a neural network rises quickly with the number of parameters [2]. This makes calculations of very large networks unfeasible for even the best modern CPU architectures. Furthermore, the majority of time spent in the GMIN basin-hopping global optimization algorithm is in the computation of the analytical loss function and gradient [30]. Thus it is appropriate to devote maximal attention to efficient calculation of the neural network gradient and loss on the GPU; this is nicely conceptualized using Amdahl's law, which states that the expected speedup of a parallel algorithm is $\frac{1}{1-p}$, where p is the proportion of the time spent on the routine to be parallelized [77].

To facilitate a port to CUDA, the neural network implementation described in Section 2.1.2 was recast into matrix form. This reformulation decomposes the expressions for the loss into terms involving element-wise products and terms involving matrix products. In this formulation $\mathbf{X} \in \mathbb{R}^{N_{in} \times N_{data}}$ is the data matrix with rows as features and columns as data points. The matrices $\mathbf{W}_2 \in \mathbb{R}^{N_{hidden} \times N_{in}}$ and $\mathbf{W}_1 \in \mathbb{R}^{N_{out} \times N_{hidden}}$ contain the weights for the neural network nodes of the input and hidden layers, respectively. The vectors $\mathbf{w}^{bh} \in \mathbb{R}^{N_{hidden}}$ and $\mathbf{w}^{bo} \in \mathbb{R}^{N_{out}}$

contain the bias weights for the hidden and output layers, respectively. Finally, note that we use element-wise notation in this report. In other words, functions applied to matrices are applied to each matrix entry (Eqn. 2.6):

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \implies f(\mathbf{A}) = \begin{bmatrix} f(a_{11}) & f(a_{12}) \\ f(a_{21}) & f(a_{22}) \\ f(a_{31}) & f(a_{32}) \end{bmatrix}. \tag{2.6}$$

The circle operator is the element-wise product, defined in Eqn. 2.7.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \\ a_{31}b_{31} & a_{32}b_{32} \end{bmatrix}. \tag{2.7}$$

First we define the matrix $\mathbf{H}$ (Eqn. 2.8), which represents the non-linear activation function being applied to the first hidden layer.

$$\mathbf{H} = \tanh\left(\begin{bmatrix} \mathbf{w}^{bh} & \dots & \mathbf{w}^{bh} \\ & & \end{bmatrix} + \mathbf{W}^{(2)}\mathbf{X} \right). \tag{2.8}$$

From this expression, the matrices for the outputs and softmax probabilities $\mathbf{Y}, \mathbf{P} \in \mathbb{R}^{N_{out} \times N_{data}}$ are calculated as follows (Eqns. 2.9 and 2.10):

$$\mathbf{Y} = \begin{bmatrix} \mathbf{w}^{bo} & \dots & \mathbf{w}^{bo} \\ & & \end{bmatrix} + \mathbf{W}^{(1)}\mathbf{H}. \tag{2.9}$$

$$\mathbf{P} = \frac{\exp(\mathbf{Y})}{\sum \exp(\mathbf{Y})}. \tag{2.10}$$

Using these two expressions, E, the cost, can be be calculated using Eqn. 2.11, where $\mathbf{D} \in \mathbb{R}^{N_{out} \times N_{data}}$ is a matrix which contains a 1-of-K class representation for each data point. $\mathbf{1}_N$ is a column vector of length N with all elements equal to 1. Here, we add a **L2** penalty with regularization parameter, λ.

$$E = -\frac{1}{N_{data}} \mathbf{1}^T{}_{N_{out}} \log(\mathbf{P} \odot \mathbf{D}) \mathbf{1}_{N_{data}} + \lambda \mathbf{w}^T \mathbf{w}. \tag{2.11}$$

To calculate the gradient $\bigtriangledown E$ with respect to the weights, we need the following partial derivatives: $\frac{dE}{d\mathbf{W}^{(2)}}$, $\frac{dE}{d\mathbf{W}^{(1)}}$, $\frac{dE}{d\mathbf{w}^{bh}}$ and $\frac{dE}{d\mathbf{w}^{bo}}$. Since the first two expressions are derivatives of a scalar with respect to a matrix, we obtain matrix outputs. Hence we reshape these derivative matrices into column vectors, as indicated by the symbol (:). Thus we can write the gradient in vector form (Eqn. 2.12).

$$\bigtriangledown E = \begin{bmatrix} \frac{dE}{d\mathbf{W}^{(2)}}(:) \\ \frac{dE}{d\mathbf{W}^{(1)}}(:) \\ \frac{dE}{d\mathbf{w}^{bh}} \\ \frac{dE}{d\mathbf{w}^{bo}} \end{bmatrix} + 2\lambda \mathbf{w}. \tag{2.12}$$

The final equation for the gradient is shown in Eqn. 2.13. Note,

$$\mathbf{S} = \text{sech}^2\Big(\begin{bmatrix} \mathbf{w}^{bh} & \dots & \mathbf{w}^{bh} \end{bmatrix} + \mathbf{W}^{(2)}\mathbf{X} \Big).$$

$$\bigtriangledown E = -\frac{1}{N_{data}} \begin{bmatrix} \mathbf{S} \odot \mathbf{W}^{(1)^\mathbf{T}} (\mathbf{D} - \mathbf{P}) \mathbf{X}^\mathbf{T}(:) \\ (\mathbf{D} - \mathbf{P}) \mathbf{H}^\mathbf{T}(:) \\ \mathbf{S} \odot \mathbf{W}^{(1)^\mathbf{T}} (\mathbf{D} - \mathbf{P}) \mathbf{1}_{N_{data}} \\ (\mathbf{D} - \mathbf{P}) \mathbf{1}_{N_{data}} \end{bmatrix} + 2\lambda \mathbf{w}. \tag{2.13}$$

This form for the gradient is particularly convenient as it has many terms that occur repeatedly, facilitating ease of computation. More importantly, it is readily compatible with various cuBLAS functions (Section 2.3.2).

2.3.2 Custom Kernels and cuBLAS

We used custom CUDA kernels and matrix operations from the cuBLAS library [57] to implement a parallel version of Eqn. 2.11 and Eqn. 2.13. Here, it is pertinent to note that using the cuBLAS library, as opposed to writing custom CUDA kernels, is highly favorable as code from the cuBLAS library automat-

ically determines the thread, grid and block structure, as well as the memory management for any GPU device architecture [57]. We implemented element-wise function operations, f($\mathbf{A}$), for exp($\mathbf{A}$), tanh($\mathbf{A}$) and sech^2($\mathbf{A}$) using custom CUDA kernels. These kernels all had the same form, the only difference being the function of interest (Algorithm 1).

Algorithm 1 Element-wise function kernel

ElementFunction($\mathbf{A}$, $\mathbf{B}$, size)
1. int i $\leftarrow$ blockDim.x$*$blockIdx.x + threadIdx.x
2. if (i $<$ size) $\implies$ $\mathbf{B}$[i] $\leftarrow$ f($\mathbf{A}$[i])

Note, we used specific formulations of $\tanh(x)$ and $\text{sech}^2(x)$ for numerical stability (Eqns. 2.14 and 2.15):

$$\tanh(x) = 1.0 - \frac{2.0}{1 + \exp(2x)}. \tag{2.14}$$

$$\text{sech}^2(x) = \frac{4.0}{(\exp(x) + \exp(-x))^2}. \tag{2.15}$$

In a similar manner, we also implemented CUDA kernels for element-wise operations involving two matrices (Algorithm 2). Here, f($\mathbf{A}$, $\mathbf{B}$) represents various operations, including elementary element-wise subtraction, multiplication, and division. The subtraction kernel was used for $\mathbf{D} - \mathbf{P}$, the multiplication kernel for $\mathbf{S} \odot \mathbf{W}^{(1)\mathbf{T}}$, and the division kernel for $\mathbf{P} = \frac{\exp(\mathbf{Y})}{\sum \exp(\mathbf{Y})}$.

Algorithm 2 Element-wise two matrix operations kernel

ElementTwoFunctions($\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, size)
1. int i $\leftarrow$ blockDim.x$*$blockIdx.x + threadIdx.x
2. if (i $<$ size) $\implies$ $\mathbf{C}$[i] $\leftarrow$ f($\mathbf{A}$[i], $\mathbf{B}$[i])

In addition to custom CUDA kernels, we also used the cuBLAS library for certain standard matrix operations for maximal efficiency and stability. In particular, we used the cublasDger function for outer products (Eqn. 2.16) to broadcast

the $\mathbf{w}^{bh}$ and $\mathbf{w}^{bo}$ vectors in the linear layers (Eqns. 2.8 and 2.9). In particular, we set $\mathbf{x}$ to either $\mathbf{w}^{bh}$ or $\mathbf{w}^{bo}$, $\mathbf{y} = \mathbf{1}_N$ and $\alpha = 1$.

$$\mathbf{A} = \alpha \mathbf{x}\mathbf{y}^{\mathbf{T}} + \mathbf{A}. \tag{2.16}$$

The cublasDgemm function was used for all matrix multiplication operations (Eqn. 2.17). For the linear layers (Eqns. 2.8 and 2.9), we set $\alpha = 1$ and $\beta = 1$ to calculate $\mathbf{W}^{(2)}\mathbf{X}$ and $\mathbf{W}^{(1)}\mathbf{H}$. To calculate the matrix-matrix multiplications in the gradient expression (Eqn. 2.13), we set $\alpha = 1$ and $\beta = 0$. This formulation allowed us to calculate the following matrix products:
$\mathbf{S} \odot \mathbf{W}^{(1)\mathbf{T}}(\mathbf{D}\text{-}\mathbf{P})$, $\mathbf{S} \odot \mathbf{W}^{(1)\mathbf{T}}(\mathbf{D}\text{-}\mathbf{P})\mathbf{X}^{\mathbf{T}}$ and $(\mathbf{D}\text{-}\mathbf{P})\mathbf{H}^{\mathbf{T}}$:

$$\mathbf{C} = \alpha op(\mathbf{A}) op(\mathbf{B}) + \beta \mathbf{C}. \tag{2.17}$$

Finally, we used the cublasDgemv function for matrix-vector operations (Eqn. 2.18). Specifically, we used this operation for summation of matrix entries via multiplication with $\mathbf{1}$ vectors by setting $\alpha = 1$ and $\beta = 0$:

$$\mathbf{y} = \alpha op(\mathbf{A})\mathbf{x} + \beta \mathbf{y}. \tag{2.18}$$

We also implemented the cross-entropy loss using a combination of Algorithm 2 with f($\mathbf{A}$, $\mathbf{B}$) = $\frac{-1}{N_{train}} \log(\mathbf{A})\mathbf{B}$ and cublasDgemm multiplication with appropriate $\mathbf{1}$ vectors for summation. Finally, we used Algorithm 2 to add the regularization components to the loss and gradient. We used $\mathbf{A} + \lambda\mathbf{B}$ and $\mathbf{A} + 2\lambda\mathbf{B}$ for f($\mathbf{A}$, $\mathbf{B}$), respectively.

2.3.3 Speed Testing

The parallel formulation of the potential introduced in this work was evaluated for speed against the serial version. We recorded the time taken to perform a specified number of potential calls on the D3.0 dataset using [2,$\mathbf{B}$,4,$\mathbf{D}$,0.0001] architectures. Here $\mathbf{B}$, the number of hidden nodes, takes values of 5, 10, 20, 50, 100, 500, 800 and 1000, and $\mathbf{D}$, the amount of training data, is either 1000 or 5000 training points. We recorded the average time taken to perform five

basin-hopping steps with 500 L-BFGS iterations for each **B**,**D** combination; the average was calculated using 25 random initial geometries. A low number of L-BFGS iterations were chosen intentionally to ensure that none of the quenches converged to a local minimum within the step limit. Furthermore, we kept the Hessian update size fixed at four for both the CPU and GPU to ensure that both implementations required the same number of potential calls (same number of steps).

In addition to evaluating the potential on a fixed call basis, we also tested the time taken to converge to a local minimum. Importantly, these two timing metrics are different. This is due to the trade-off between L-BFGS Hessian update size and the number of potential calls required to achieve convergence. The CPU implementation favours a large Hessian update size, since each potential call is more expensive relative to the GPU. However, for the GPU implementation, storing a number of L-BFGS updates is costly and thus it is preferable to instead call the potential many more times. For biomolecular energy landscapes, it has been shown that the optimal Hessian update size is four (relative to 100-500 on the CPU), which can be faster to convergence by approximately an order of magnitude [30].

We used the D3.0 dataset to determine the time taken to reach convergence for the CPU and GPU implementations for four different systems [2,**B**,4,**D**,0.0001], where **B** was either 50 or 150 hidden nodes and **D** was either 1000 or 5000 training points. These four architectures were chosen because they are systems that are currently too large to fully characterize using the CPU implementation; furthermore, GPU speedups are generally only expected for large problem sizes [30]. For each system, we recorded the time taken for 10 basin-hopping quenches with 10000 L-BFGS iterations to converge to 10 local minima. We defined convergence at an RMS value of 10^{-10} for the gradient. We used initial weight vectors corresponding to local minima and a step size of 0.1 to ensure that each quench would converge within 10000 L-BFGS iterations. We performed this procedure for Hessian update sizes of 4, 8, 50, 100, 150, 200, 250, 300, 400, 450 and 500 on both the CPU and GPU. The overall speedup was computed as a ratio of the

time corresponding to the optimal CPU update size and the time corresponding to the optimal GPU update size.

2.4 Mislabelling Experiments

2.4.1 Dataset Mislabelling Procedure

For each dataset, we uniformly permuted fixed percentages of 1000 training labels without replacement. Thus an outcome i would be mapped to any other outcome j ($j \neq i$) with probability $\frac{1}{N-1}$, where N is the number of output classes. Specifically for the D1.2-D3.0 datasets (four output model), class i could be mislabelled to that of any class $i \neq j$ with equal probabilities of $\frac{1}{3}$. Similarly, for the MNIST dataset, we used uniform mislabelling probabilities of $\frac{1}{9}$ for each of the ten output classes. Note, previous work by Rolnick et al. used a fixed amount of correct training data rather than a total error percentage. In this analysis, we opt for the error percentage formulation, as the number of stationary points decreases with the amount of training data [41] and would therefore interfere with our disconnectivity graph analysis.

2.4.2 Minima and Transition States in Noisy Datasets

To study neural networks subject to label noise, we first investigated the number of stationary points for the D1.2-D3.0 datasets for various error percentages. We mislabelled fixed proportions of four training datasets (0, 10, 50 and 100 % label error) according to the procedure delineated in Section 2.4.1 for the [2,10,4,1000,0.0001] neural network architecture (Section 2.1.2). We used the serial GMIN implementation (Section 2.2.1) to obtain a database of local minima; these experiments were converged to an RMS gradient of 10^{-10} for each minimization. We used OPTIM to tightly converge minima (Section 2.2.2). Finally, we used PATHSAMPLE to create a database of minima and transition states; note, the PATHSAMPLE stage used the DNEB and H-EF methods via OPTIM (Section 2.2.2). From this connected database, we extracted the loss of all minima as well as the total number of minima and transition states.

2.4.3 Generalization in Noisy Datasets

We also studied how minima obtained from noisy loss functions generalized to unseen testing sets. We used the same mislabelling procedures and architectures described in Section 2.4.2 with the [2,10,4,1000,0.0001] and [784,10,10,1000,0.1] architectures respectively for D1.2-D3.0 and MNIST. For the D1.2-D3.0 datasets, local minimization was performed via the serial neural network potential (Section 2.1.2) with an RMS gradient convergence criterion of 10^{-10}. Serial OPTIM and PATHSAMPLE were used to tightly converge minima and create a discrete path sampling database, respectively. The MNIST experiment, with 7960 optimizable parameters, required the GPU implementation introduced in this work (Section 2.3). We used the new GPU implementation (Section 2.3) to perform 2000 basin-hopping quenches with an RMS convergence criteria of 10^{-10}.

For both architectures, the serial OPTIM method was used to calculate training and testing AUC values for all minima at all error thresholds. For each dataset, the testing AUC values were calculated using 1000 correctly labelled unseen testing points drawn from the same distributions, using the parameters obtained from both the training global minimum as well as an average over all database minima on a corresponding testing dataset. Here it is pertinent to note, for both types of dataset, that the average quantities more correctly correspond to the average values calculated over *low-lying* sampled minima. Also, the global training minimum is more correctly described as the lowest training minimum we obtained in our analysis.

For further analysis, we computed the training AUC values for the clean and mislabelled entries of the training data for all the respective datasets. The motivation for this experiment was to see how well our architectures learn to filter uniform random noise. Note, a neural network that performs very well on the clean segment of the training data and very poorly on the mislabelled section filters noise effectively. To perform this experiment, we followed the same procedure delineated above for each subset of entries, with the appropriate modification for the number training points. More concretely, for a dataset with 10% error, we calculated the average training AUCs corresponding to 900 clean training examples

and 100 mislabelled training examples.

Finally, we used disconnectivity graph analysis using DISCONNECTION DPS [38–40] for the D1.2-D3.0 mislabelled datasets to visualize the neural network loss landscapes. Each landscape has its minima coloured by training and testing AUCs on separate plots.

2.5 Neural Network Nearest-Neighbours

2.5.1 Formulation

We explored how reduced-connectivity between neural network nodes affects the landscape of perceptrons with a single hidden layer. The motivation for these experiments was two-fold. First, it allows for a systematic study of the effects of locality on the network. For example, it may be possible to obtain glassy, multi-funnelled landscapes using this approach; this would be similar to molecular case, in which short-range forces can produce more complicated landscapes with many more stationary points [78–81]. Secondly, it allows us to study the relationship between neural network architecture and capacity.

In this work, the network connectivity is described by a straightforward linear mapping. First, we project all nodes onto an equally divided unit line at locations $0,\ 1/(N_\beta-1),\ 2/(N_\beta-1),\ \ldots,\ (N_\beta-2)/(N_\beta-1),\ 1$; here, N_β, refers to a general neural network node. The distance between hidden node h and input node i is defined as d_{hi} (Eqn. 2.19).

$$d_{hi} = \left| \frac{h-1}{N_{\text{hidden}}-1} - \frac{i-1}{N_{\text{in}}-1} \right|. \tag{2.19}$$

The distance between hidden node h and output node o is defined as d_{ho} (Eqn. 2.20).

$$d_{ho} = \left| \frac{h-1}{N_{\text{hidden}}-1} - \frac{o-1}{N_{\text{out}}-1} \right|. \tag{2.20}$$

We sort the distances by magnitude and keep weights corresponding to the specified number of nearest-neighbours as well as all bias weights; all other weights are set to zero. Equal distances are resolved by ascending order for node index.

Note, the potential described here was not developed for this dissertation, but was previously programmed in GMIN, OPTIM and PATHSAMPLE by Professor David Wales.

2.5.2 Experiments

The potential described in Section 2.5.1 was used to generate databases of minima and transition states for the [2,10,4,1000,0.0001] and the [2,5,4,1000,0.00001] architectures on the D3.0 dataset; this dataset was chosen specifically as it had a large number of minima for 1000 training points and 10 hidden nodes.

For the [2,10,4,1000,0.0001] architecture, we examined the effects of allowing 1, 2, 3 and 100 nearest-neighbours on the machine learning landscapes; these corresponded to 40, 20, 10 and 0 frozen weights, respectively. GMIN was used for local minimization, OPTIM was used for tight convergence and AUC calculation and PATHSAMPLE was used to build a DPS database. Note, the experiment for 100 nearest-neighbours was intended to test whether the modified potential could recover the same minima as the fully-connected potential in the limit of a large number of nearest-neighbours. We also created a database for the 3 nearest-neighbour potential, relaxed using the original single-layered potential. Specifically, we extracted the nearest-neighbour PATHSAMPLE minima and relaxed them with the fully-connected potential using OPTIM (RMS convergence 10^{-10}). All nearest-neighbour experiments for the [2,10,4,1000,0.0001] architecture were visualized using disconnectivity graphs, coloured by testing AUC.

For the [2,5,4,1000,0.00001] architecture, we compared the two and three nearest-neighbour models to the fully-connected model. Here we deliberately chose a smaller regularization constant to reduce the convexity of the landscape. The intention of this experiment was to determine whether, under low regularization, it is possible to obtain multi-funnelled landscapes using our reduced node-connectivity measure. Again, we used the GMIN, OPTIM, PATHSAMPLE and DISCONNECTION DPS workflow described earlier to create discrete path databases and disconnectivity graphs.

3

Results

3.1 Speed of The CUDA Implementation

We implemented a single-layered neural network potential on the GPU and interfaced it with the existing CUDAGMIN framework [30]. The speed for a fixed number of potential calls was evaluated as a function of the number of hidden nodes and the amount of training data. Tables 3.1 and 3.2 present the average GPU speedup for five basin-hopping steps and 500 L-BFGS iterations for various numbers of hidden nodes and for 1000 and 5000 training points. Large increases in speed were observed for very wide networks (500-1000 hidden nodes) for both 1000 and 5000 training points (Tables 3.1-3.2). However, the CUDA implementation was slower for smaller systems (5-20 hidden nodes, 1000 training points). For a fixed number of hidden nodes, the relative speedup was significantly faster for 5000 training points vs. 1000 training points. The maximum speedup was obtained for 1000 hidden nodes and 5000 training points (factor of nearly 30; Table 3.2).

Hidden nodes	Time CPU(s)	Time GPU(s)	Speedup
5	2.27	7.47	0.30x
10	3.86	10.32	0.37x
20	7.02	10.55	0.67x
50	15.15	12.92	1.17x
100	29.45	19.35	1.52x
500	145.28	21.62	6.72x
800	231.29	24.93	9.28x
1000	287.64	27.87	10.32x

Table 3.1: CPU vs. GPU implementations for 5 to 1000 hidden nodes and 1000 D3.0 training points. Average time was recorded for five basin-hopping steps with 500 L-BFGS iterations.

Hidden nodes	Time CPU(s)	Time GPU(s)	Speedup
5	11.09	9.78	1.13x
10	19.03	12.42	1.53x
20	32.65	18.27	1.79x
50	76.64	17.49	4.38x
100	148.16	21.02	7.05x
500	719.87	35.83	20.09x
800	1124.37	41.22	27.28x
1000	1404.59	48.86	28.75x

Table 3.2: CPU vs. GPU implementations for 5 to 1000 hidden nodes and 5000 D3.0 training points. Average time was recorded for five basin-hopping steps with 500 L-BFGS iterations.

In addition to the speed per fixed number of potential calls, we also recorded the time taken to perform ten basin-hopping quenches to *convergence* as a function of the L-BFGS Hessian update size. The four test systems were 1000 and 5000 training points with 50 and 150 hidden nodes (Tables 3.3-3.6). For 50 hidden nodes and 1000 training points, we observed the fastest convergence for an update size of 200 on the GPU; the optimal CPU update size was 350 (Table 3.3). For this system, the CPU implementation was faster than the GPU implementation by a factor of 1.3 (Table 3.3).

Hist.	**4**	**8**	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
GPU(s)	26.3	23.4	21.0	16.1	16.7	**15.8**	17.1	17.2	17.1	17.0	17.1	17.3
CPU(s)	56.7	55.1	22.4	14.5	12.0	13.3	13.5	12.3	**11.9**	12.1	12.0	12.1

Table 3.3: Time taken for ten basin-hopping quenches to reach convergence on the GPU and CPU for 50 hidden nodes and 1000 training points for various values of the L–BFGS Hessian update size. Optimal times for GPU and CPU are shown in bold.

For 50 hidden nodes and 5000 training points, we observed the fastest convergence for an update size of 100 on the GPU; the optimal CPU update size was 250 (Table 3.4). For this system, the GPU implementation was faster than the CPU implementation by a factor of 3.4 (Table 3.4).

Hist.	**4**	**8**	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
GPU(s)	34.0	31.7	25.9	**17.2**	19.2	17.7	17.6	17.7	17.7	17.7	17.7	17.7
CPU(s)	297	250	169	74.7	63.1	58.7	**57.9**	57.9	58.0	59.0	58.6	59.2

Table 3.4: Time taken for ten basin-hopping quenches to reach convergence on the GPU and CPU for 50 hidden nodes and 5000 training points for various values of the L–BFGS Hessian update size. Optimal times for GPU and CPU are shown in bold.

For 150 hidden nodes and 1000 training points, we observed the fastest convergence for an update size of 100 on the GPU; the optimal CPU update length was 350 (Table 3.5). For this system, the GPU implementation was faster than the CPU implementation by a factor of 1.6 (Table 3.5).

Hist.	**4**	**8**	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
GPU(s)	34.0	34.3	26.2	**24.3**	25.3	25.6	25.9	24.9	24.7	24.6	24.8	24.7
CPU(s)	173	159	70.1	42.9	40.7	40.3	39.5	39.2	**38.9**	39.1	39.6	38.9

Table 3.5: Time taken for ten basin-hopping quenches to reach convergence on the GPU and CPU for 150 hidden nodes and 1000 training points for various values of the L–BFGS Hessian update size. Optimal times for GPU and CPU are shown in bold.

Finally, for 150 hidden nodes and 5000 training points, we observed the fastest convergence for an update size of 100 on the GPU; the optimal CPU update size was 150 (Table 3.6). For this system, the GPU implementation was faster than the CPU implementation by a factor of 6.6 (Table 3.6).

Hist.	**4**	**8**	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
GPU(s)	45.7	42.6	30.3	**26.2**	27.9	31.1	27.2	28.1	28.4	28.2	28.4	28.5
CPU(s)	834	707	282	195	**173**	175	178	179	189	180	180	178

Table 3.6: Time taken for ten basin-hopping quenches to reach convergence on the GPU and CPU for 150 hidden nodes and 5000 training points for various values of the L–BFGS Hessian update size. Optimal times for GPU and CPU are shown in bold.

3.2 Mislabelling

We investigated the effect of introducing systematic errors into the training dataset on the resulting neural network loss function. These experiments were performed for three datasets that span distinct volumes of molecular configuration space, denoted D1.2-D3.0 (Section 3.2.1), as well as MNIST.

3.2.1 D1.2-D3.0

First, we determined the number of transition states and minima for the three geometry optimization datasets (D1.2-D3.0) and the loss associated with the training global minimum[1] (Tables 3.7-3.9).

Error (%)	**Min**	**Ts**	**GMIN Loss**
0	6	20	0.519
10	13	66	0.791
50	26	155	1.285
100	20	148	1.236

Table 3.7: Number of minima (Min) and transition states (Ts) for the D1.2 dataset as a function of the mislabelling error, along with the loss value of the training global minimum.

Error (%)	**Min**	**Ts**	**GMIN Loss**
0	167	779	0.803
10	203	817	1.002
50	292	1181	1.300
100	260	1372	1.312

Table 3.8: Number of minima (Min) and transition states (Ts) for the D2.0 dataset as a function of the mislabelling error, along with the loss value of the training global minimum.

Error (%)	**Min**	**Ts**	**GMIN Loss**
0	122	592	0.850
10	266	960	1.000
50	394	1474	1.291
100	490	1395	1.321

Table 3.9: Number of minima (Min) and transition states (Ts) for the D3.0 dataset as a function of the mislabelling error, along with the loss value of the training global minimum.

[1] More correctly, this refers to the the minimum with the lowest training loss in our database.

We observed that the number of local minima and transition states increased with the percentage of mislabelled data for all three datasets (Tables 3.7-3.9). The loss of the global minimum also increased with the percentage of mislabelled data (Tables 3.7-3.9). Furthermore, the larger molecular configuration spaces tended to have a greater number of stationary points (D3.0 > D2.0 > D1.2) for each error level.

To study generalization, we used the AUC value corresponding to the training global minimum as a metric to characterize the performance of the neural network on the D1.2-D3.0 datasets. For all three datasets, both the training and testing AUC decreased as the percentage of mislabelled data increased (Tables 3.10-3.12). For these datasets, at 0% error, the training and testing AUC values were very similar. Interestingly, we observed that for 10% and 50% mislabelled training sets, the testing AUCs corresponding to the global training minima exceeded the respective training AUCs. However, at an error rate of 100% the testing AUCs decreased significantly, falling to lower values than the corresponding training AUCs (Tables 3.10-3.12).

Error (%)	**Training AUC, Loss**	**Testing AUC, Loss**
0	0.810, 0.519	0.797, 0.552
10	0.730, 0.791	0.791, 0.622
50	0.604, 1.285	0.741, 0.994
100	0.772, 1.236	0.242, 2.771

Table 3.10: Training and testing loss and AUC values for the global minimum with the D1.2 dataset.

Error (%)	**Training AUC, Loss**	**Testing AUC, Loss**
0	0.808, 0.803	0.778, 0.867
10	0.724, 1.002	0.741, 1.050
50	0.647, 1.300	0.708, 1.232
100	0.578, 1.312	0.351, 1.609

Table 3.11: Training and testing loss and AUC values for the global minimum with the D2.0 dataset.

Error (%)	**Training AUC, Loss**	**Testing AUC, Loss**
0	0.749, 0.850	0.732, 0.891
10	0.727, 1.000	0.720, 0.927
50	0.639, 1.291	0.706, 1.131
100	0.589, 1.321	0.336, 1.918

Table 3.12: Training and testing loss and AUC values for the global minimum with the D3.0 dataset.

To study the properties of the cost function further, we also investigated the AUC value averaged over all the local minima in our database. These results, as well as the corresponding standard deviations, are reported in Tables 3.13-3.15. The same trends present for the training global minimum were observed for the averages. In addition, the standard deviation of both the training and testing AUC values increased as the percentage of mislabelled data increased. Furthermore, the standard deviation of the testing AUC values was significantly greater than for training (Tables 3.13-3.15). Please see Appendix B for violin plots visualizing the training and testing AUC distributions.

Error (%)	**Training <AUC>, σ(AUC)**	**Testing <AUC>, σ(AUC)**
0	0.810, 0.00033	0.796, 0.00031
10	0.728, 0.0021	0.791, 0.0020
50	0.602, 0.0018	0.739, 0.0030
100	0.768, 0.0047	0.245, 0.0043

Table 3.13: Average and standard deviation of AUC values for all the local minima in the D1.2 dataset for testing and training.

Error (%)	**Training <AUC>, σ(AUC)**	**Testing <AUC>, σ(AUC)**
0	0.806, 0.0010	0.778, 0.00091
10	0.723, 0.0047	0.754, 0.0071
50	0.638, 0.0031	0.691, 0.023
100	0.575, 0.0057	0.342, 0.022

Table 3.14: Average and standard deviation of AUC values for all the local minima in the D2.0 dataset for testing and training.

Error (%)	**Training <AUC>, σ(AUC)**	**Testing <AUC>, σ(AUC)**
0	0.746, 0.0035	0.733, 0.0025
10	0.724, 0.0036	0.726, 0.0043
50	0.638, 0.0029	0.699, 0.0083
100	0.591, 0.0061	0.340, 0.013

Table 3.15: Average and standard deviation of AUC values for all the local minima in the D3.0 dataset for testing and training.

We also studied the average training AUC for the mislabelled and correctly labelled components of each training dataset for each error level (Tables 3.16-3.18). Based on this analysis, we found that the training AUC was very low when calculated based only on the mislabelled portion of each training dataset. Conversely, the training AUC was very high when computed on the training dataset containing only the correctly labelled examples (Tables 3.16-3.18). For every dataset, the training AUC values for the correctly labelled components exceeded the corresponding testing AUC values (Tables 3.16-3.18 and 3.13-3.15).

Error (%)	**<AUC>, σ(AUC) INCORRECT**	**<AUC>, σ(AUC) CORRECT**	**<AUC>, σ(AUC) ALL**
0	-	0.810, 0.00033	0.810, 0.00033
10	0.190, 0.019	0.809, 0.0023	0.728, 0.0021
50	0.398, 0.010	0.779, 0.0054	0.638, 0.0031
100	0.768, 0.0047	-	0.768, 0.0047

Table 3.16: Average and standard deviation of training AUC values for the incorrect and correct components of the mislabelled D1.2 dataset. The ALL column is repeated from Table 3.13 for comparison.

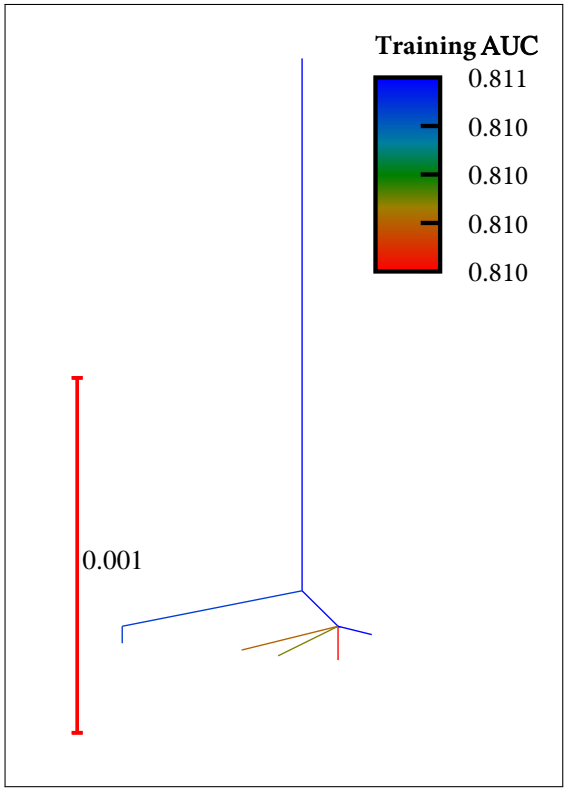
Error (%)	**<AUC>, σ(AUC) INCORRECT**	**<AUC>, σ(AUC) CORRECT**	**<AUC>, σ(AUC) ALL**
0	-	0.806, 0.0010	0.806, 0.0010
10	0.559, 0.013	0.745, 0.0045	0.723, 0.0047
50	0.522, 0.0065	0.772, 0.0040	0.638, 0.0029
100	0.591, 0.0061	-	0.591, 0.0061

Table 3.17: Average and standard deviation of training AUC values for the incorrect and correct components of the mislabelled D2.0 dataset. The ALL column is repeated from Table 3.14 for comparison.

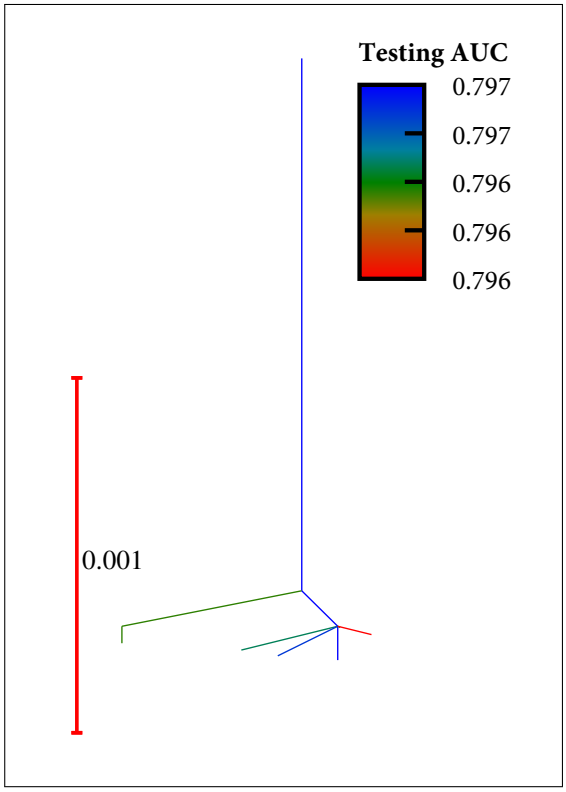
Error (%)	**<AUC>, σ(AUC) INCORRECT**	**<AUC>, σ(AUC) CORRECT**	**<AUC>, σ(AUC) ALL**
0	-	0.746, 0.0035	0.746, 0.0035
10	0.509, 0.015	0.747, 0.0034	0.724, 0.0036
50	0.539, 0.0079	0.760, 0.0072	0.638, 0.0029
100	0.591, 0.0061	-	0.591, 0.0061

Table 3.18: Average and standard deviation of training AUC values for the incorrect and correct components of the mislabelled D3.0 dataset. The ALL column is repeated from Table 3.15 for comparison.

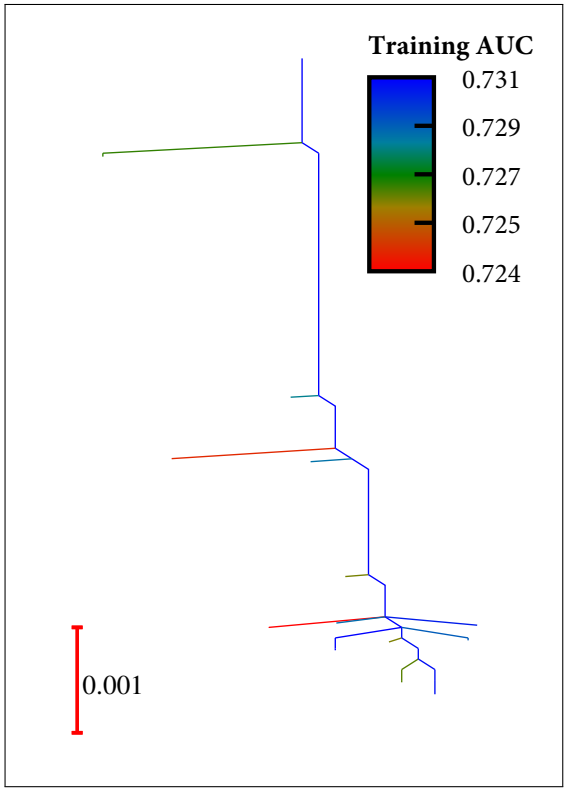
The databases of minima and transition states for D1.2-D3.0 were visualized using disconnectivity graphs (Figures 2-4). These graphs are presented for each error level (0-100%). To study generalizability, two graphs were created for each error threshold, with minima coloured by training and testing AUCs (Figures 2-4). We observed that the disconnectivity graphs coloured by training AUC tended to have a relatively high AUC values near the global minimum. In addition, for low error rates, the graphs coloured by testing AUC also had relatively high AUC values for low-lying minima. Conversely, for high error rates, many high performing (testing AUC) minima were found at high values of training loss (Figures 2-4).



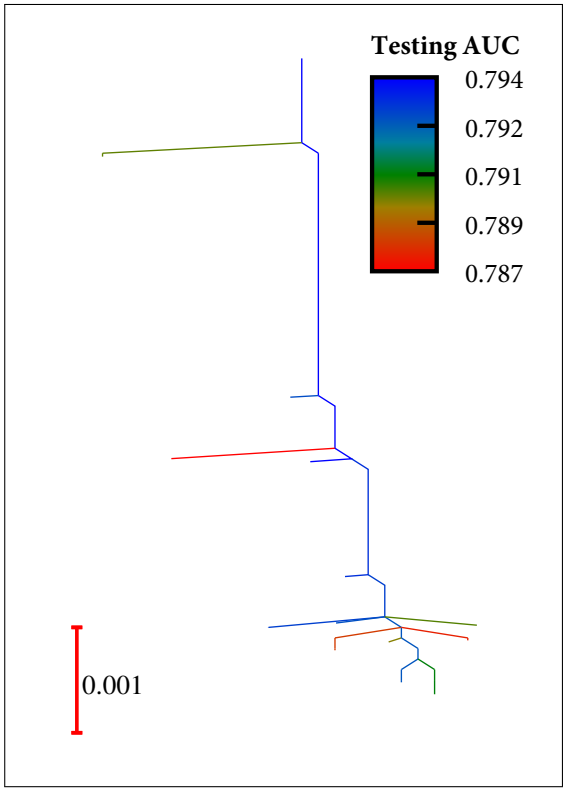
(a) 0 % Mislabelled (Training AUC)



(b) 0 % Mislabelled (Testing AUC)

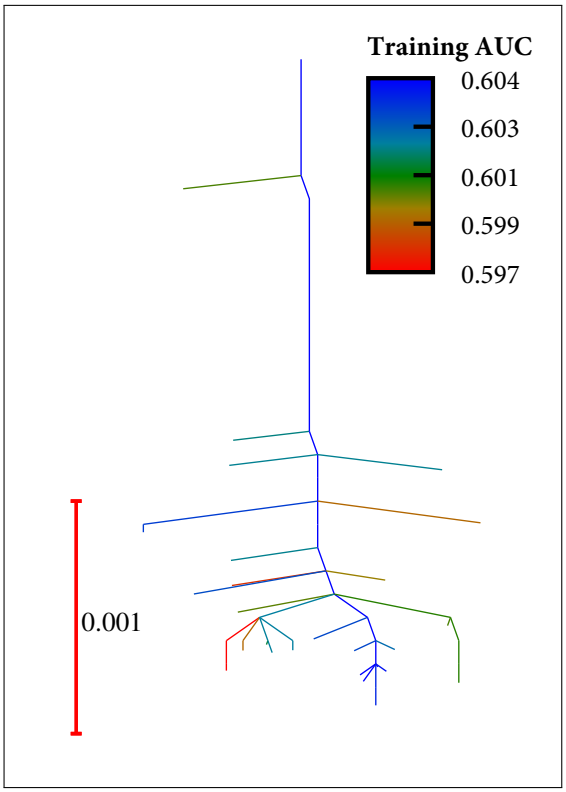


(c) 10 % Mislabelled (Training AUC)

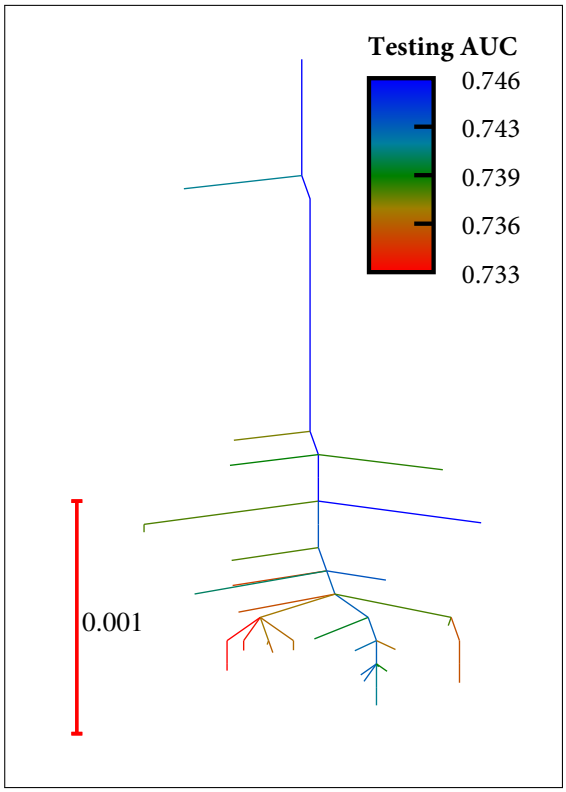


(d) 10 % Mislabelled (Testing AUC)

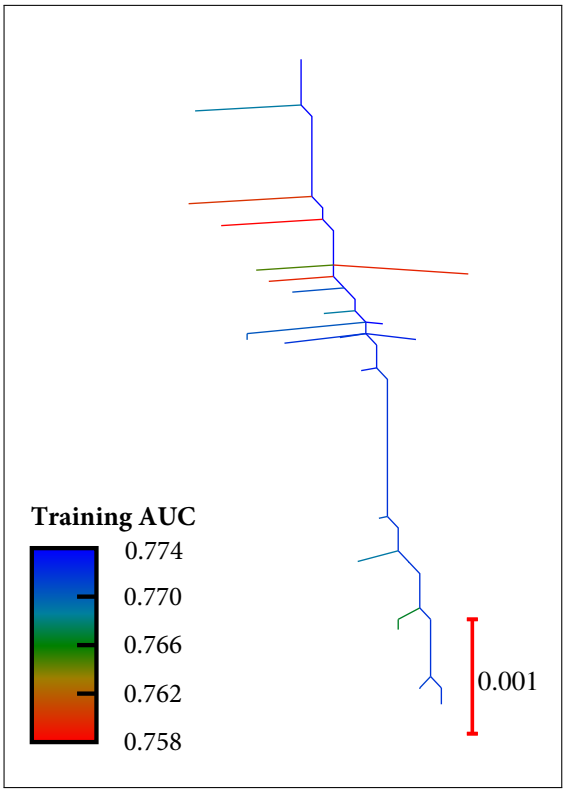
Figure 2: Disconnectivity graphs for dataset D1.2, 1000 training points, $\lambda = 0.0001$, coloured by training and testing AUC as a function of label errors.



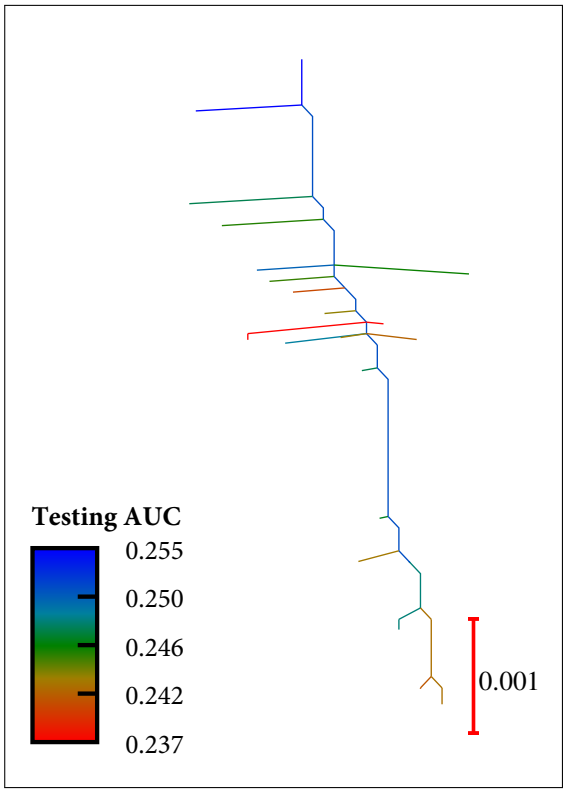
(e) 50 % Mislabelled (Training AUC)



(f) 50 % Mislabelled (Testing AUC)

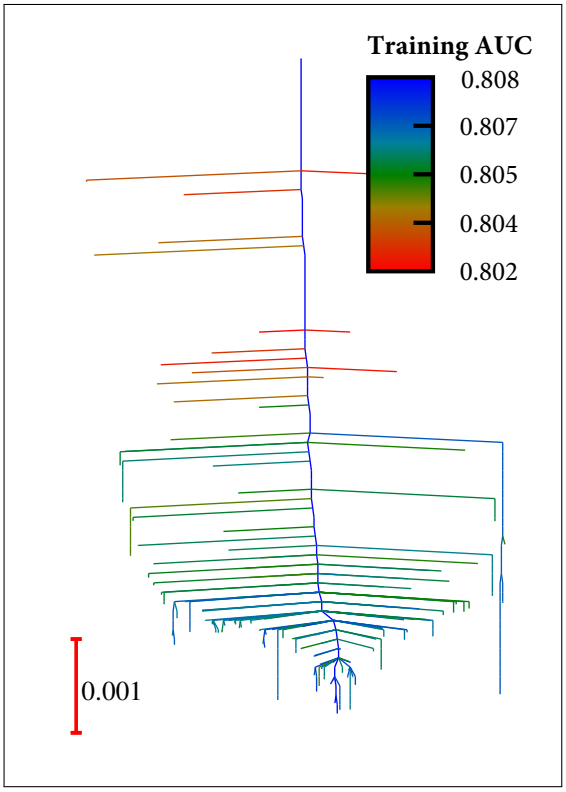


(g) 100 % Mislabelled (Training AUC)

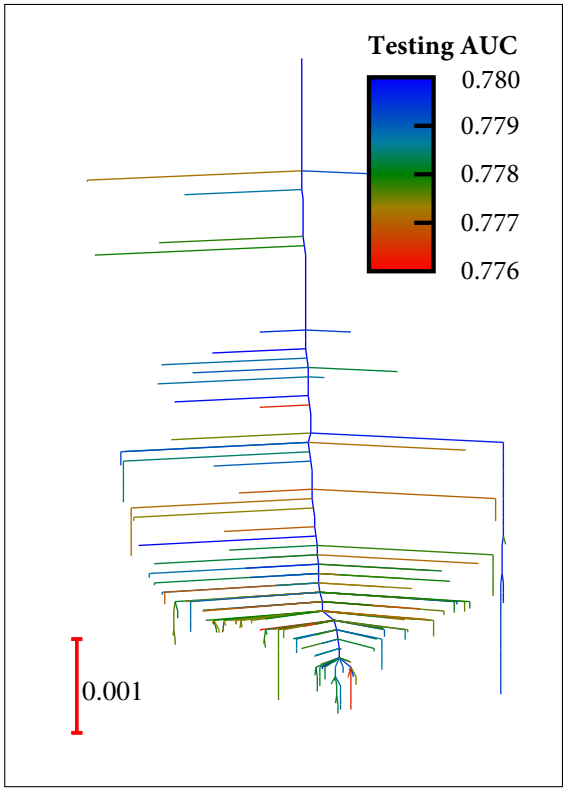


(h) 100 % Mislabelled (Testing AUC)

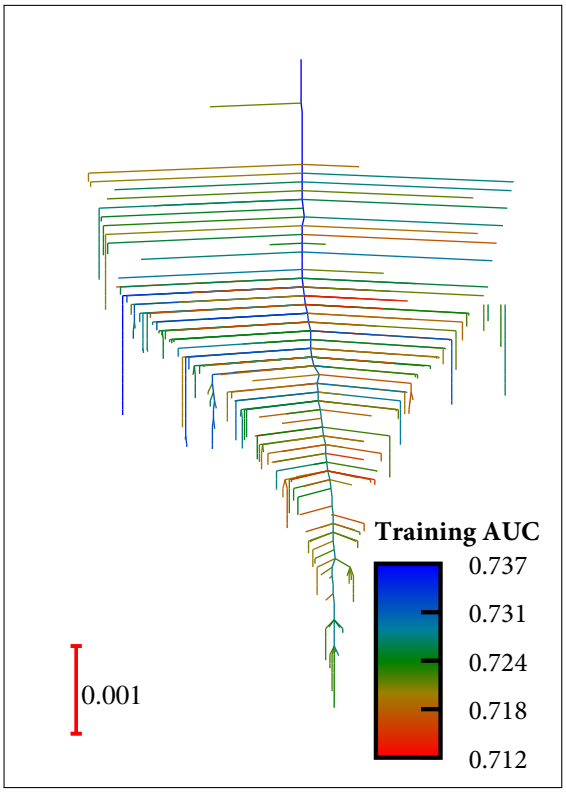
Figure 2: (continued) Disconnectivity graphs for dataset D1.2, 1000 training points, $\lambda = 0.0001$, coloured by training and testing AUC as a function of label errors.



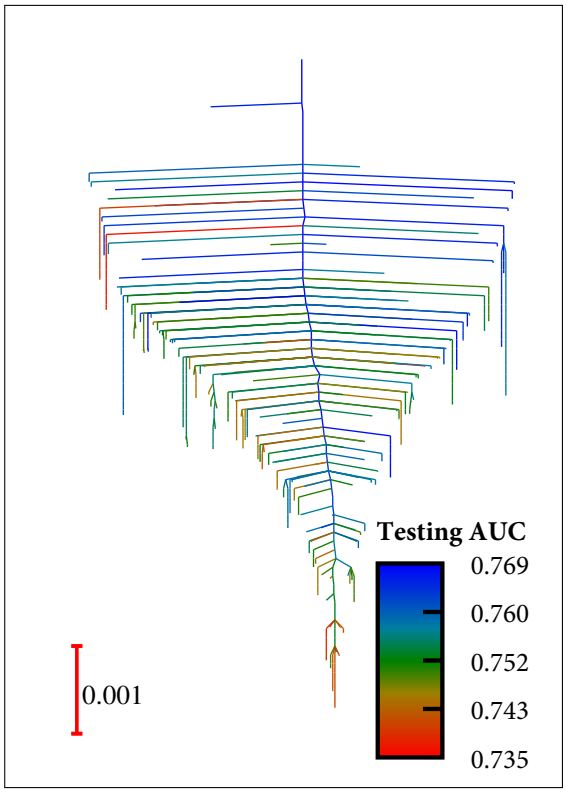
(a) 0 % Mislabelled (Training AUC)



(b) 0 % Mislabelled (Testing AUC)

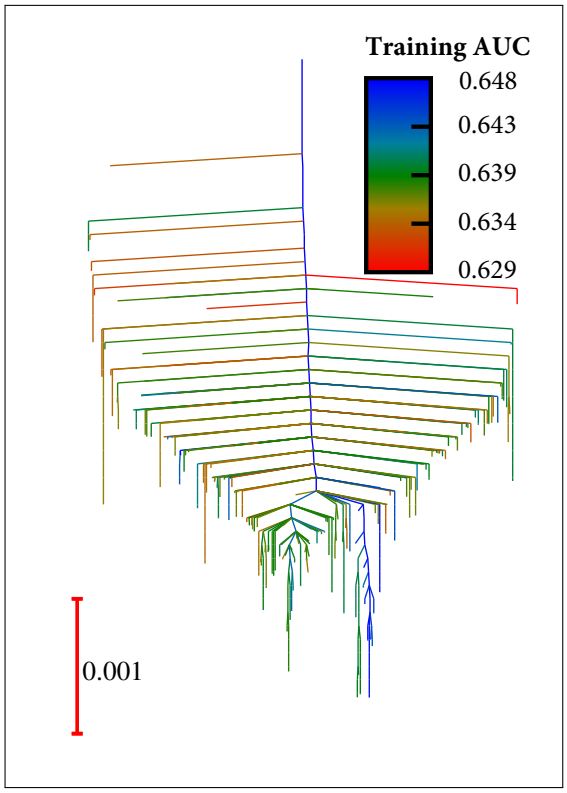


(c) 10 % Mislabelled (Training AUC)

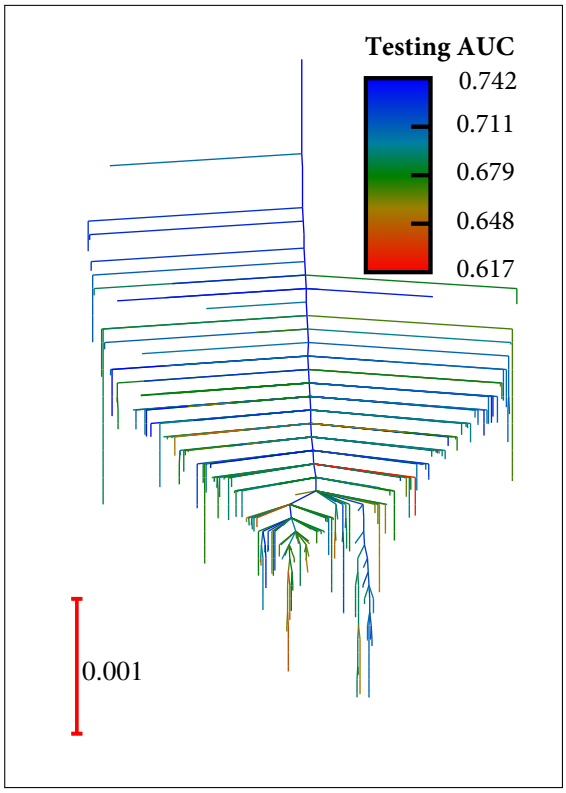


(d) 10 % Mislabelled (Testing AUC)

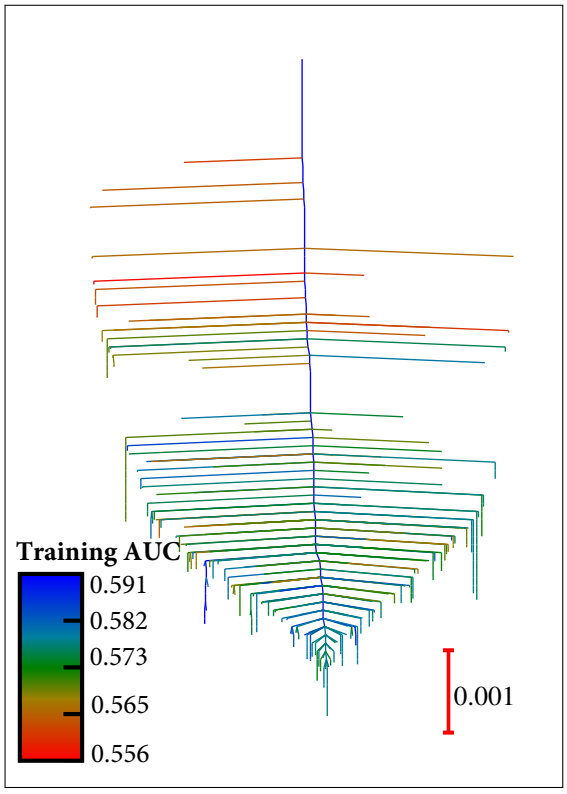
Figure 3: Disconnectivity graphs for dataset D2.0, 1000 training points, $\lambda = 0.0001$, coloured by training and testing AUC as a function of label errors.



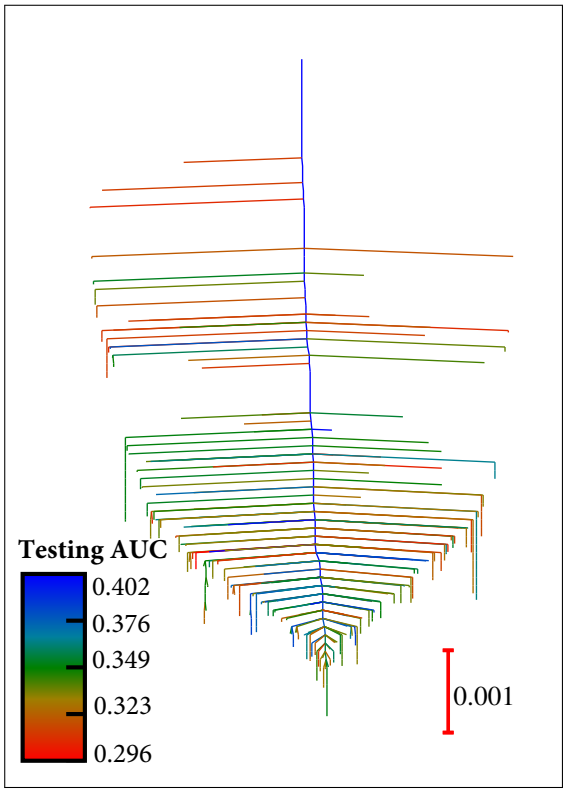
(e) 50 % Mislabelled (Training AUC)



(f) 50 % Mislabelled (Testing AUC)

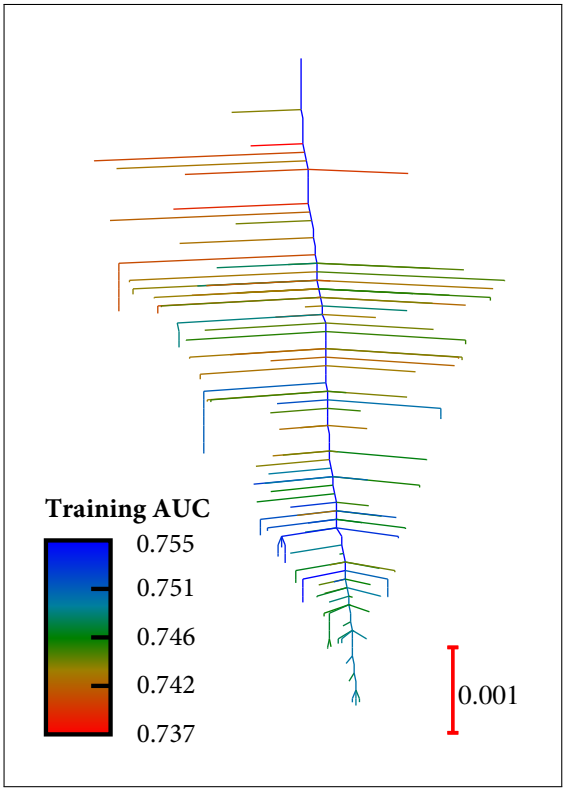


(g) 100 % Mislabelled (Training AUC)

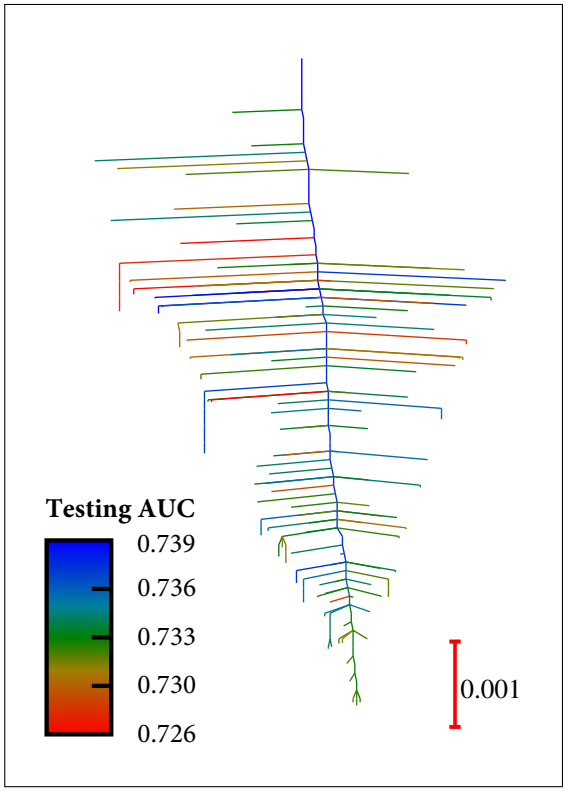


(h) 100 % Mislabelled (Testing AUC)

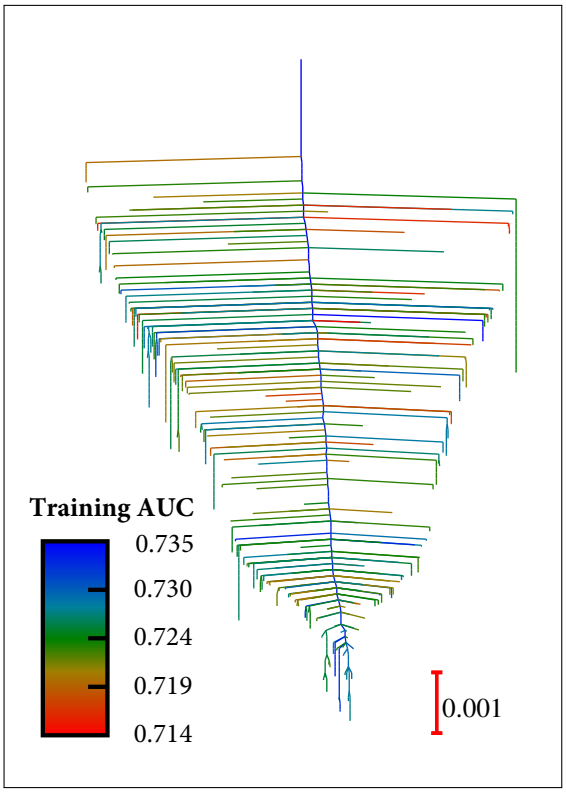
Figure 3: (continued) Disconnectivity graphs for dataset D2.0, 1000 training points, $\lambda = 0.0001$, coloured by training and testing AUC as a function of label errors.



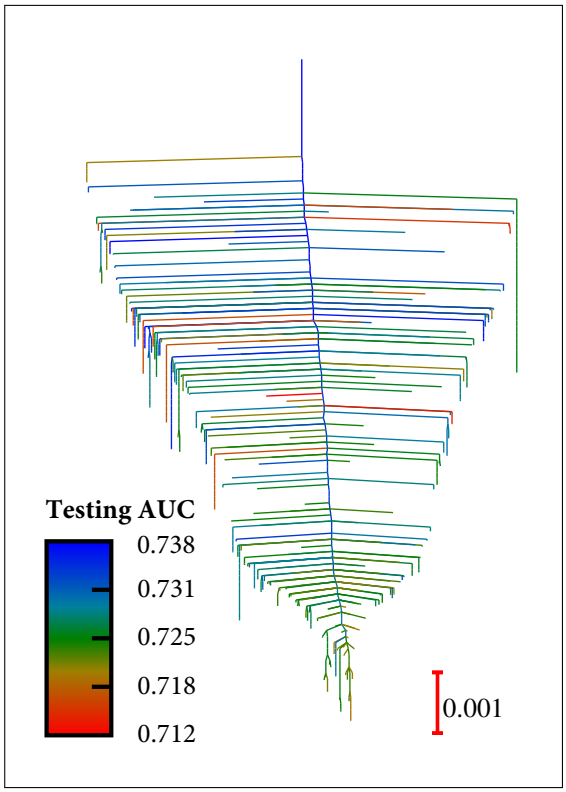
(a) 0 % Mislabelled (Training AUC)



(b) 0 % Mislabelled (Testing AUC)

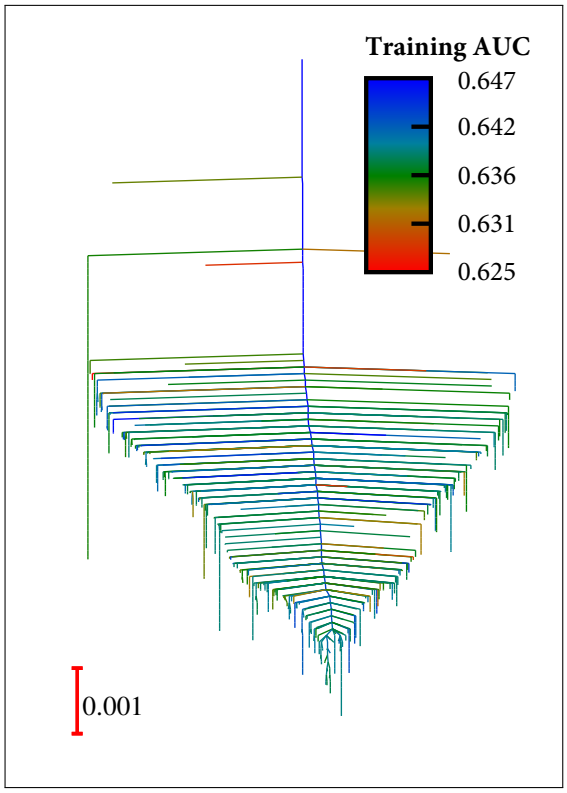


(c) 10 % Mislabelled (Training AUC)

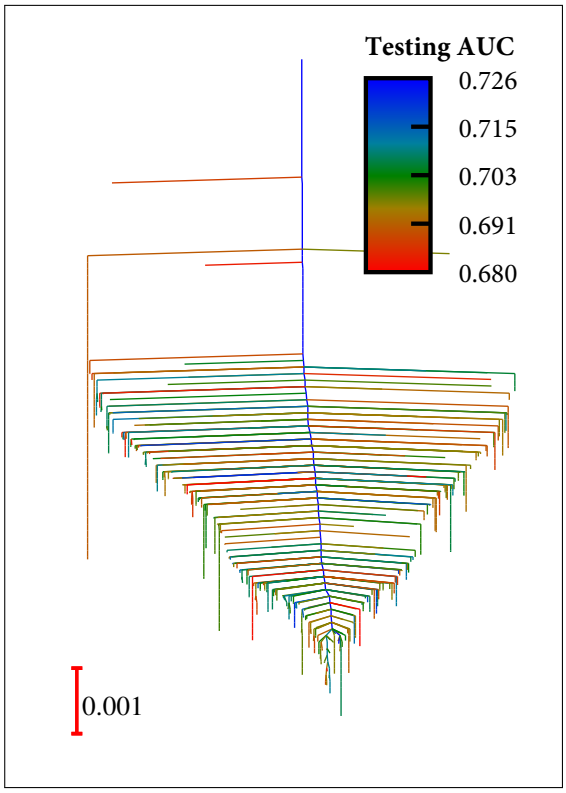


(d) 10 % Mislabelled (Testing AUC)

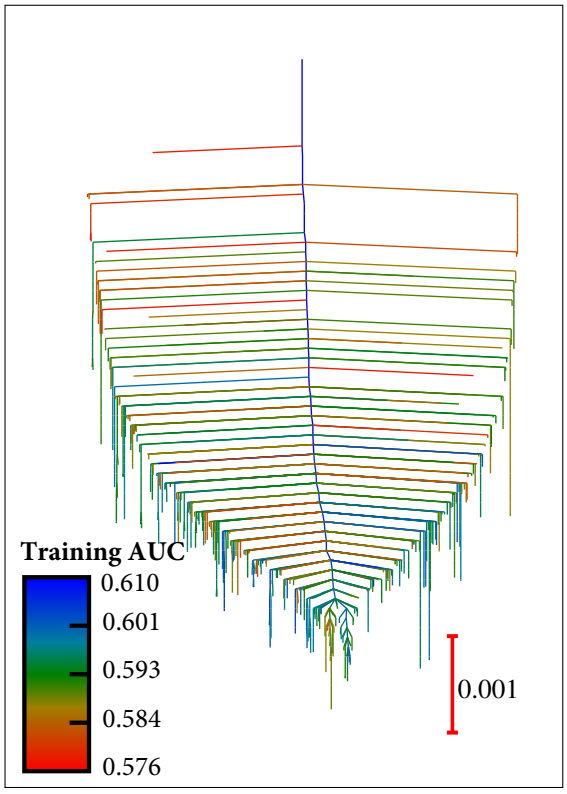
Figure 4: Disconnectivity graphs for dataset D3.0, 1000 training points, $\lambda = 0.0001$, coloured by training and testing AUC as a function of label errors.



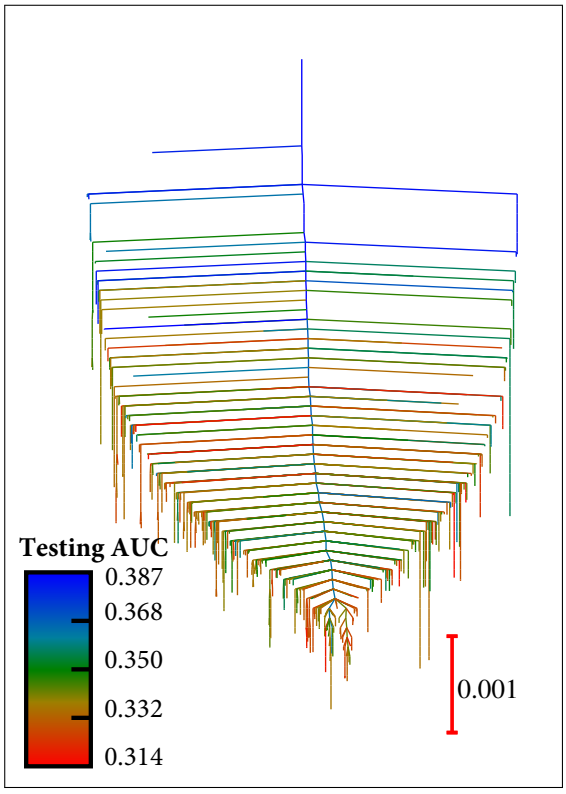
(e) 50 % Mislabelled (Training AUC)



(f) 50 % Mislabelled (Testing AUC)



(g) 100 % Mislabelled (Training AUC)



(h) 100 % Mislabelled (Testing AUC)

Figure 4: (continued) Disconnectivity graphs for dataset D3.0, 1000 training points, $\lambda = 0.0001$, coloured by training and testing AUC as a function of label errors.

3.2.2 MNIST

We used the GPU implementation to obtain a database of minima for the MNIST dataset for various percentages of mislabelled training data. Analogously to Section 3.2.1, we obtained the average and standard deviation of the training and testing AUCs at each error threshold (Table 3.19). We found that even at relatively high error levels (75%), we were able to obtain high testing AUCs. Furthermore, we found that the standard deviation over the database of minima increased with the amount of mislabelled data and was significantly higher for testing vs. training (Table 3.19 and Figure 5).

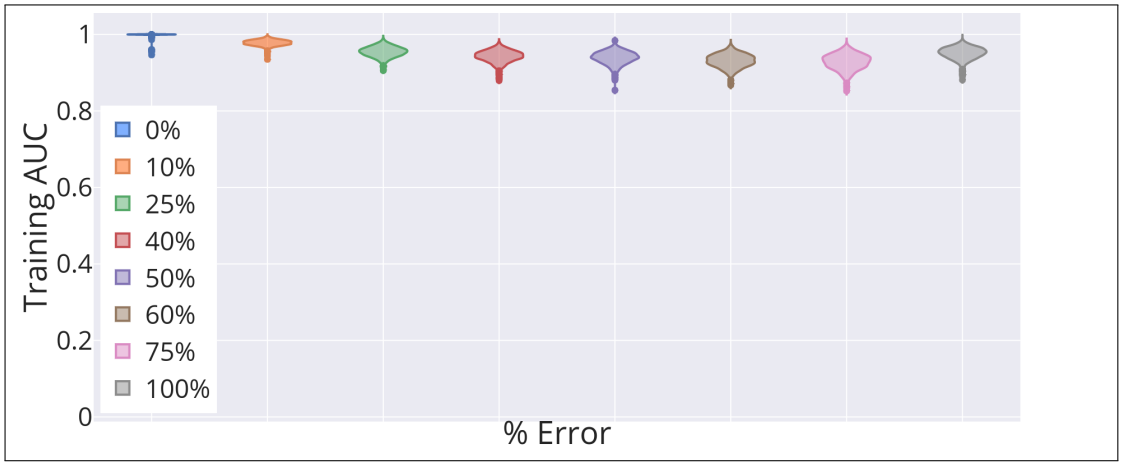
Error (%)	**Training <AUC>, σ(AUC)**	**Testing <AUC>, σ(AUC)**
0	0.9996, 0.0027	0.9687, 0.010
10	0.9783, 0.0070	0.9645, 0.012
25	0.9545, 0.013	0.9472, 0.018
40	0.9429, 0.015	0.9304, 0.022
50	0.9390, 0.016	0.9197, 0.024
60	0.9310, 0.017	0.8940, 0.031
75	0.9281, 0.020	0.7716, 0.061
100	0.9509, 0.015	0.2333, 0.072

Table 3.19: Average and standard deviation of AUC values on the MNIST dataset for testing and training.

Finally, we found that the average local minimum performed significantly better on the clean segments of the mislabelled training data (Table 3.20). In particular, even when more than half the training set was mislabelled, the neural networks preferentially performed better on the clean subsection (Table 3.20).

Error (%)	**<AUC>, σ(AUC) INCORRECT**	**<AUC>, σ(AUC) CORRECT**	**<AUC>, σ(AUC) ALL**
0	-	0.9996, 0.0027	0.9996, 0.0027
10	0.7747, 0.050	0.9997, 0.0014	0.9783, 0.0070
25	0.8011, 0.044	0.9991, 0.0025	0.9545, 0.013
40	0.8440, 0.032	0.9976, 0.0042	0.9429, 0.015
50	0.8707, 0.029	0.9950, 0.0062	0.9390, 0.016
60	0.8893, 0.026	0.9891, 0.010	0.9310, 0.017
75	0.9164, 0.024	0.9729, 0.019	0.9281, 0.020
100	0.9509, 0.015	-	0.9509, 0.015

Table 3.20: Average and standard deviation of training AUC values for the incorrect and correct components of the mislabelled MNIST dataset. The ALL column is repeated from Table 3.19 for comparison.



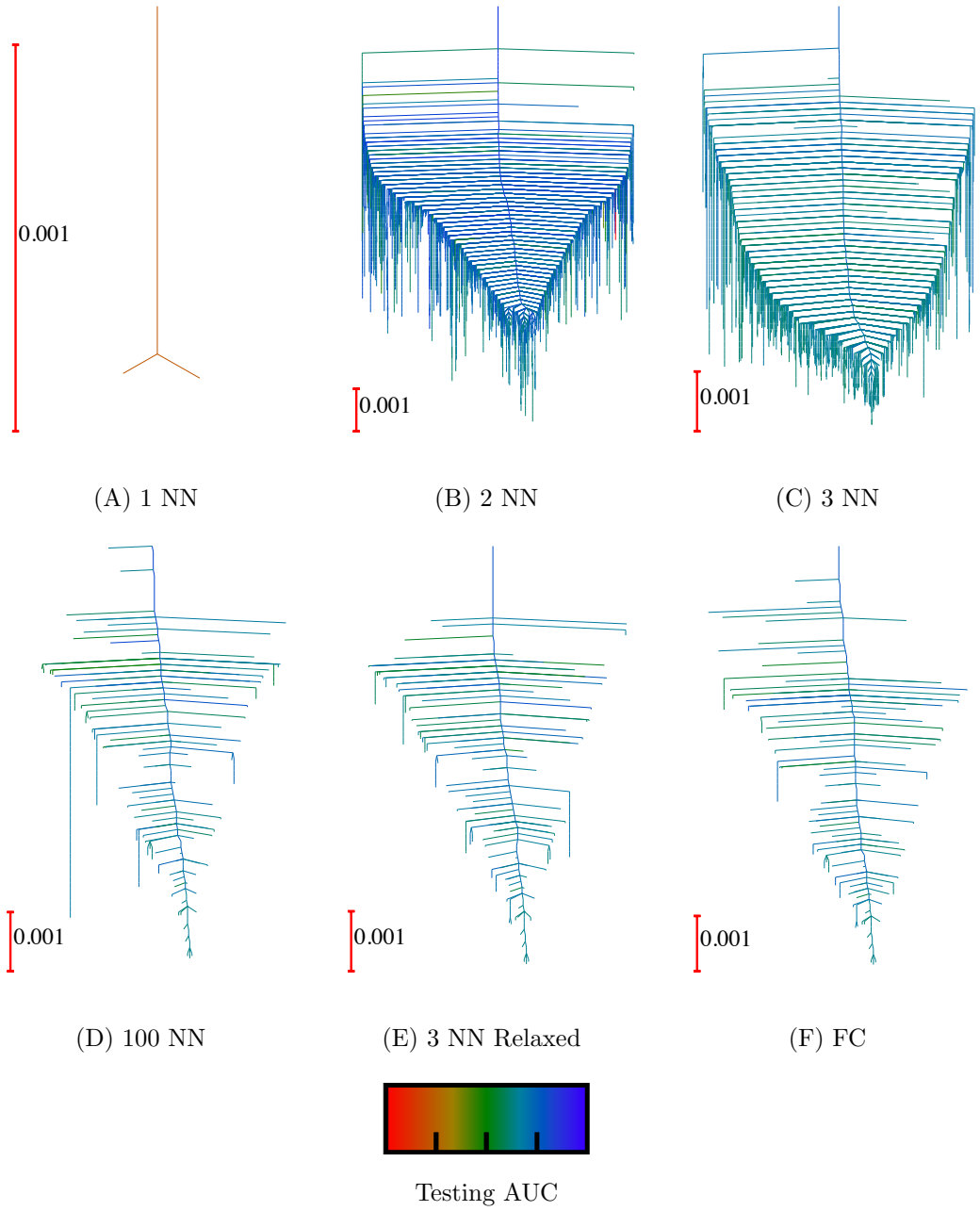
(a) Training AUC for various percentages of mislabelled data.

(b) Testing AUC for various percentages of mislabelled data.

Figure 5: Violin plots for the training and testing AUC values for various error percentages on the MNIST dataset. This plot was created using Plotly [82].

3.3 Neural Network Nearest-Neighbours

We used the nearest-neighbour scheme described in Section 2.5 to study landscapes with reduced-connectivity.



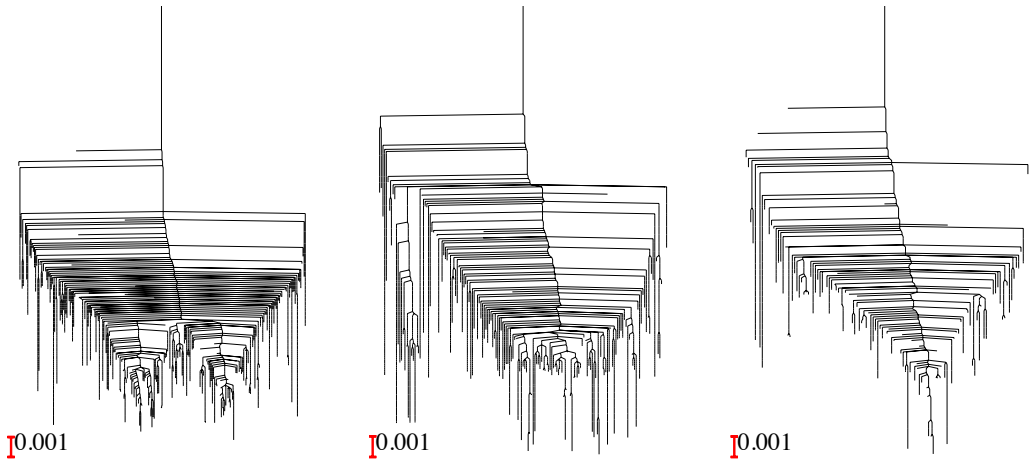
(A) 1 NN (B) 2 NN (C) 3 NN

(D) 100 NN (E) 3 NN Relaxed (F) FC

Figure 6: Disconnectivity graphs for 1, 2, 3 and 100 nearest-neighbours (NN) for the D3.0 dataset. FC refers to the fully-connected original architecture. The colouring runs from red (low testing AUC) to blue (high testing AUC).

The results for the [2,10,4,1000,0.0001] architecture for 1, 2, 3 and 100 (i.e. fully connected) nearest-neighbours were visualized using disconnectivity graphs (Figure 6). Here, all the graphs are coloured by testing AUC (same scale). The graphs for 3 NN relaxed and 100 NN were very similar to the fully-connected (FC) network. The graphs for two and three nearest-neighbours had many more stationary points and exhibited frustrated, single-funnelled landscapes. Interestingly, for the majority of minima, the two nearest-neighbour model outperformed the three nearest-neighbour model and the fully-connected model. Finally, we only found two minima, with very low testing AUC values, for one nearest-neighbour.

For the reduced regularization [2,5,4,1000,0.00001] architecture, the disconnectivity graphs for two and three nearest-neighbours were significantly more frustrated than the fully-connected model; in addition, these models exhibited multi-funnelled landscapes (Figure 7).



(A) 2 NN (B) 3 NN (C) FC

Figure 7: Disconnectivity graphs for 2 and 3 nearest-neighbours (NN) for the D3.0 dataset, compared to the fully-connected [2,5,4,1000,0.00001] architecture (FC).

Note, all the results presented here are analysed in detail in the next section.

4

Discussion

4.1 Speed of The CUDA Implementation

In this dissertation, we successfully implemented a single-layered neural network potential in CUDA. We interfaced this potential with GMIN by adapting a Fortran90-CUDA wrapper. We studied the speed, for a fixed number of potential calls, of this new formulation as a function of the number of hidden nodes and the amount of training data. We observed large speedups for wide networks for both 1000 and 5000 training points (Tables 3.1-3.2). However, the potential was slower for smaller systems (5-10 hidden nodes and 1000 training points). These results are in line with expected results from GPU programming, namely that speedups generally scale with the problem size [30, 53]. This is because GPUs are programmed for high arithmetic intensity, low throughput operations (such as matrix multiplication) [53]. Thus, any given GPU core will perform more slowly than a corresponding CPU core but speedups will arise due to the sheer number of processing units on a typical GPU. For larger systems, the number of parallel computations increases, and therefore so does the speed relative to the CPU. Unsurprisingly, we observed the largest speedup (29-fold) for the largest system: 1000 hidden nodes and 5000 training points (Table 3.2). Satisfyingly, we also observed that the time for a fixed number of potential calls grew much more slowly for the GPU implementation than for the CPU implementation. For example, a 10 fold increase in the number of nodes from 100 to 1000 hidden nodes

(5000 training points) increased the CPU computation time by a factor of 9.5, while the same increase in the number of nodes on the GPU only increased the computation time by a factor of 2.3 (Table 3.2).

We also found that the GPU implementation converges faster than the CPU implementation. As explained in [30], there is a difference between the time per potential call and the time to convergence. On the CPU, it is more efficient to use a large number of L-BFGS updates because the time taken per call is relatively high. Since the Hessian is approximated quite accurately, fewer L-BFGS steps are needed to minimize the objective function. However, on the GPU, it is more efficient to use a cruder approximation to the Hessian, which requires many more L-BFGS steps to reach convergence. Thus, to enable fair comparison, we measured the time taken to reach convergence for CPU and GPU optimized values of the Hessian update size for larger systems of 50 and 150 hidden nodes (Tables 3.3-3.6). Interestingly, unlike the biomolecular case [30], we found that the optimal update size for GPU machine learning landscapes can be large (generally around 100 steps). This relatively high Hessian update size allows for an accurate computation of the Hessian and thus allows the minimizer to converge in fewer steps. Since the optimal Hessian update sizes for the GPU and CPU implementations are within an order of magnitude, we observed only small differences between the speedup per fixed number of potential calls and the speedup to convergence (Tables 3.1-3.2 and 3.3-3.6). This result was consistent with our expectations since the speedup on a fixed call basis should be upper-bounded by the speedup on a convergence basis.

4.2 Mislabelling

4.2.1 Minima and Transition States in Noisy Datasets

For each error level and all three geometry optimization datasets (D1.2-D3.0), we catalogued the number of minima and transition states, as well as the loss corresponding to the training global minimum (Tables 3.7-3.9). We found that, on average, the number of local minima and transition states increased with

the percentage of mislabelled data for all three datasets (Tables 3.7-3.9). This observation suggests that a large number of local minima reflect many competing values for the parameters of the model, and thus produce higher uncertainty in the statistical fit. Based on this reasoning, it is unsurprising that noisier datasets lead to greater uncertainty in fitting the training data. The loss of the global minimum also increased as the percentage of mislabelled data increased (Tables 3.7-3.9). This result suggests that a clean cost function / dataset combination produces a simpler structure (relative to fitting noise). Furthermore, we also observed that the larger the molecular configuration space, the greater the number of minima and transition states (Tables 3.7-3.9). This result is expected as there should be greater uncertainty in predicting final outcomes from more distant molecular configurations. In other words, the diversity of the dataset depends on the size of the configuration space. This interpretation is further supported by the observation that the loss of the global minimum increases as a function of configuration space size (Tables 3.7-3.9).

4.2.2 Fitting Accuracies in Noisy Datasets

For the geometry optimization datasets, we also studied how the training global minimum as well as the average local training minimum performed on an unseen testing set; we performed these experiments as a function of a fixed percentage of mislabelling error. First, for the case of 0 % error, we observed a tight band of low-lying local minima with high AUCs, which agrees closely with previous work [13, 14].

We also observed that, in all three datasets, as the percentage of mislabelled data increased, the training and testing AUC for both the global and average training minimum decreased (Tables 3.10-3.12 and 3.13-3.15). This trend is consistent with expectations, as it should be relatively more difficult to obtain robust fits in noiser datasets (especially in the underparameterized regime). Interestingly, however, we observed that for 10% and 50% error, the testing AUC *outperformed* the training AUC for both the global and average local training minimum (Tables 3.10-3.12 and 3.13-3.15). This result implies that the neural

networks learn the structure of the correct data, as suggested in [42].

Thus, since these training AUCs are calculated on the mislabelled dataset, the networks perform poorly (since they have actually learned the correct structure). However, since the testing AUCs are calculated on a correctly labelled dataset, the corresponding networks perform significantly better. Note that when the error rate is increased to 100%, the training error does not decrease very much. In this regime, the neural networks exclusively fit noise. However, here, the testing AUCs decrease precipitously. This result was expected as the neural network was fitted to noise, and thus cannot possibly generalize to an unseen dataset (Tables 3.10-3.12 and 3.13-3.15).

Due to the increased speeds observed in Section 4.1 for large neural network architectures, we used the GPU implementation to perform a similar analysis for the MNIST dataset and examined the average training and testing AUC values (for sampled low-lying minima) for various error thresholds. We observed relatively high testing AUC values for all error thresholds between 0 and 75%. Note that even under 75% uniform random error, we were able to obtain average testing AUC values in excess of 0.75 (Table 3.19). These results are in line with previous work on the MNIST dataset [42, 45], which also show that neural networks can achieve high testing accuracy under uniform random label noise. Unlike the D1.2-D3.0 datasets, we do not obtain higher testing accuracies relative to training accuracies as the error percentage is increased. However, after studying average neural network performance on the correct and incorrect portions of the mislabelled dataset, it is still true that the networks perform significantly better on the clean segment, even with large amounts of noise (Table 3.20).

We also examined the variance of the training and testing AUC values for the D1.2-D3.0 datasets and the MNIST dataset. In both cases, we find a systematic trend towards increased testing AUC variance with the increase in dataset error, which indicates a change in the structure of the underlying landscape. Thus, while we do find good minima, which agree with [42, 45], we also find many bad minima in agreement with previous attacking type experiments [13, 18]. In other words, based on our empirical results, we find that the relatively tight band of

local minima above the global minimum no longer exists for the mislabelled case. Furthermore, in almost every example, the variance of the average testing AUC is greater than the variance of the average training AUC. This result indicates that the uncertainty in the training dataset is directly manifested in the uncertainty of the landscape. Thus, while [42,45] do show that it is possible to obtain high testing accuracies under uniform random error, the landscapes perspective indicates that the probability of finding such solutions diminishes as the error percentage increases. This suggests that for practical neural network applications, if computationally feasible, when training using noisy datasets, it might be beneficial to optimize a given architecture a greater number of times to determine the best model. It also indicates that it might be possible to design better optimizers to preferentially find these good solutions, even with significant training noise.

4.2.3 Disconnectivity Graphs for Noisy Datasets

To study generalization under noise, for each dataset and error threshold, we produced disconnectivity graphs coloured by both training and testing AUC values for the D1.2-D3.0 datasets (Figures 2-4). Interestingly, for all error thresholds, we observed single-funnelled energy landscapes. Since even the graphs at 100% error had a funnelled appearance, we conclude that this structure arises due to the single-layered feed-forward architecture, and not the input data. These results are consistent with previous work on single-layered neural networks [19, 20]. As expected, for all error thresholds, we observed that low-lying minima have high training AUCs. Furthermore, the training and testing AUC values are reasonably correlated for 0% error (Figures 2-4). This result was also unsurprising, as the hope of neural network training is that low-lying minima generalize well to unseen testing sets.

Interestingly, as we mislabel the training dataset, the better testing AUC minima (in the graphs, green-blue) can be found at higher loss values, and the low-lying minima can have relatively low testing AUCs. This result highlights that for a fixed irreducible error (training set error), each minima has its own bias-variance type trade-off (Eqn. 1.3). Some lower minima tend to overfit to noise,

leading to high training AUCs and low testing AUCs. However, some higher loss training minima, pay a training AUC/loss cost, but are able to filter noise more effectively and thus generalize well. These results are consistent with the hypothesis that it can sometimes be better to converge to local minima, rather than the global minimum, to prevent overfitting [14]. Together, these results help explain why the testing variance of AUC values increases as the percentage of mislabelled training data increases.

4.3 Neural Network Nearest-Neighbours

We investigated the effects of reduced-connectivity on single-layered networks using a nearest-neighbour formulation described in Section 2.5.1. For the [2,10,4,1000,0.0001] architecture, we observed that the potential correctly reproduced the fully-connected landscape (Figure 6) in the limit of full-connections (100 nearest-neighbours) and via relaxation with the original potential (three nearest-neighbours). These control experiments validate the nearest-neighbour potential and allow for systematic comparison. The small visual differences between these landscapes are likely due to incomplete sampling. Furthermore, the presence of some large barriers away from the global minimum is likely due to artificial frustration (Figure 6).

For two and three nearest-neighbours, the number of stationary points increased significantly. This situation is consistent with previous results for interatomic potentials with short-range forces in molecular systems [78–81]. The present analysis also suggests that strong locality can induce a more complex machine learning landscapes. This conjecture is supported by recent results for two- and three-layered neural networks, which can have more locality than single-layered neural networks, and exhibit more local minima for a similar number of edge weight variables [41].

Local minima for the two and three nearest-neighbour networks performed well on an unseen testing set, with the two nearest-neighbour model even outperforming the fully-connected model (Figure 6). Here it is worth highlighting that this result did not occur in the overparametrized limit, as the number of training data outnumbered the number of optimizable parameters by more than a factor

of 10. One possible reason for this phenomenon is the DropOut argument; i.e. the reduced neural network minimizes the problem of local regions of network coadaption, and instead produces a small number of connections which are independently good at predicting the correct class [46,51]. Another possible reason could be that the new network no longer has highly degenerate solutions arising from parameter permutation, and thus may instead be able to express more complex fitting functions [52]. This perspective is at least partially substantiated by the observation of much more complicated landscapes for reduced-connectivity (Figure 6). Interestingly, however, only two poorly performing minima were found for the one nearest-neighbour model. This observation likely reflects the fact that the architecture has significantly reduced capacity, since more than half the trainable weights are zero. Taken together, our results suggest that in terms of the landscape, optimal architectures may balance sparsity and expressiveness to perform well on unseen testing sets.

Although the reduced-connectivity landscapes obtained for the [2,10,4,1000,0.0001] architecture were significantly more frustrated than the fully-connected model, they were still relatively single-funnelled (Figure 6). To determine whether we could obtain multi-funnelled or glassy landscapes, we used the [2,5,4,1000,0.00001] architecture, which had a much smaller regularization constant (factor of 10). Since the regularization term is a convex **L2** penalty, it is possible that part of the single-funnelled appearance of the reduced-connectivity networks is due purely to regularization; i.e higher **L2** regularization convexifies the landscape [22]. Again, for the fully-connected case, we observed a single-funnelled appearance, substantiating our previous claim that this landscape is architecture dependent (Section 3.2). However, for the two and three nearest-neighbour models, we observe somewhat multi-funnelled landscapes (Figure 7). This result further reinforces the strong effect locality has on single-layered architectures.

5

Conclusions and Future Work

In this dissertation we studied the effect of perturbations on the behaviour of relatively small neural networks, where the underlying solution landscape could be properly characterised, using optimization techniques developed for energy landscapes research.

First, in order to study larger systems, we implemented and interfaced a CUDA machine learning potential with GMIN, allowing us to systematically explore much larger machine learning landscapes. We found that for large system sizes, the GPU implementation significantly outperformed the CPU implementation (approximately an order of magnitude increase in speed), on both a potential call basis and a convergence basis. This new potential will make it possible, in future work, to produce disconnectivity graphs for very wide systems and enable a detailed study of neural network capacity as a function of the number of hidden nodes. Exploration in this manner might be helpful in understanding how to systematically choose better architectures.

Using custom generated high-quality geometry optimization training data, in the limit of full landscape sampling, we showed that increasing training diversity (in this case, configuration space volume for an atomic cluster) leads to landscapes with many more stationary points and higher loss values. These results suggest a correspondence between the number of local minima and the statistical uncertainty of the landscape. In future work we would like to explore the effects of other systematic additions to training set diversity. One possible experiment

may include using some D3.0 data within the D1.2 or D2.0 datasets.

In our mislabelling analysis, we found that neural networks are able to correctly filter uniform noise for very high levels of dataset poisioning and that these findings remain (empirically) true for averages over the database local minima. We also find that for mislabelling, a tight band of minima around the global minimum does not occur. Instead, the variance of the testing AUC increases significantly with the training error. Furthermore, we observe that many high loss training minima perform well on unseen testing input, as they do not overfit to noise, highlighting a bias-variance type trade-off. The results presented here may, therefore, help guide the creation of specific optimizers to preferentially search for these better solutions. Pertinent future work involves studying the properties of minima that perform well under training set mislabelling; in particular, we plan to examine the Hessian curvature of these solutions. In addition, we would also like to consider other types of noise. Much of the realistic (and difficult) noise in machine learning datasets is not uniform, but instead highly feature dependent or adversarial [44]. As a first step, we plan to see whether a landscape analysis might illustrate why it is more difficult to train under stochastic permutation noise than uniform random noise [45]. We would also like to extend our noise analysis to neural networks with more than one hidden layer, which may be more resilient to labelling noise [42].

Finally, we explored the landscapes of neural networks with reduced-connectivity. We found that for two and three nearest-neighbour models, the corresponding networks retained sufficient expressive capacity. In particular, the network for two nearest-neighbours systematically outperformed the fully-connected case on unseen testing data. Furthermore, the landscapes corresponding to these networks were complex and highly-frustrated due to the effects of stronger locality. For very limited connectivity (one nearest-neighbour), we found only a few minima with poor predictive capability, reflecting the reduced capacity of the network. These results indicate that, for the networks studied in this work, good solutions are able to effectively balance sparsity and capacity. Future work in this area will likely include a generalized scheme for reduced-connectivity of deep neural net-

works. In addition, it may be interesting to visualize energy landscapes generated using a saliency-based reduced-connectivity scheme [51].

The hope from this work is that the insights gleaned from the study of small networks will carry over to very large networks, where there may be too many parameters to locate a single local minimum. By exploring the relationship between the loss landscape and generalizability, we hope that some of this work will be helpful in guiding the development of better neural network training procedures, loss functions and optimizers.

References

[1] Y. LeCun, Y. Bengio and G. Hinton, *Deep learning*, Nature **521**, 436 (2015).

[2] J. Schmidhuber, *Deep learning in neural networks: An overview*, Neural Netw. **61**, 85 (2015).

[3] A. Krizhevsky, *Learning multiple layers of features from tiny images*, Tech. rep. (2009).

[4] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, arXiv preprint arXiv:1409.1556 (2014).

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, in *Advances in Neural Information Processing Systems 27*, edited by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence and K. Q. Weinberger, pp. 2672–2680, Curran Associates, Inc. (2014).

[6] R. Collobert and J. Weston, in *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pp. 160–167, New York, NY, USA (2008), ACM.

[7] R. R. Trippi and E. Turban, *Neural networks in finance and investing: Using artificial intelligence to improve real world performance*, McGraw-Hill, Inc. (1992).

[8] W. T. Miller, P. J. Werbos and R. S. Sutton, *Neural networks for control*, MIT press (1995).

[9] T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning*, Springer, New York (2009).

[10] Y. Dauphin, R. Pascanu, Ç. Gülçehre, K. Cho, S. Ganguli and Y. Bengio, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, CoRR **abs/1406.2572** (2014).

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980 (2014).

[12] C. Zhang, S. Bengio, M. Hardt, B. Recht and O. Vinyals, *Understanding deep learning requires rethinking generalization*, arXiv preprint arXiv:1611.03530 (2016).

[13] L. Wu, Z. Zhu and W. E, *Towards understanding generalization of deep learning: Perspective of loss landscapes* (2017).

[14] A. Choromanska, M. Henaff, M. Mathieu, G. Arous and Y. LeCun, in *Artificial Intelligence and Statistics*, pp. 192–204 (2015).

[15] A. Anandkumar and R. Ge, in *Conference on learning theory*, pp. 81–102 (2016).

[16] H. Li, Z. Xu, G. Taylor, C. Studer and T. Goldstein, in *Adv. Neural Inf. Process. Syst.*, pp. 6389–6399 (2018).

[17] Q. Nguyen and M. Hein, in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2603–2612. JMLR. org (2017).

[18] G. Swirszcz, W. Czarnecki and R. Pascanu, *Local minima in training of deep networks* (2016).

[19] A. J. Ballard, R. Das, S. Martiniani, D. Mehta, L. Sagun, J. D. Stevenson and D. J. Wales, *Energy landscapes for machine learning*, Phys. Chem. Chem. Phys. **19**, 12585 (2017).

[20] A. J. Ballard, J. D. Stevenson, R. Das and D. J. Wales, *Energy landscapes for a machine learning application to series data*, J. Chem. Phys. **144**, 124119 (2016).

[21] R. Das and D. J. Wales, *Machine learning prediction for classification of outcomes in local minimisation*, Chem. Phys. Lett. **667**, 158 (2017).

[22] D. Mehta, X. Zhao, E. A. Bernal and D. J. Wales, *Loss surface of xor artificial neural networks*, Physical Review E **97**, 052307 (2018).

[23] J. Nocedal, *Updating quasi-newton matrices with limited storage*, Mathematics of Computation **35**, 773 (1980).

[24] D. C. Liu and J. Nocedal, *On limited memory bfgs method for large scale optimization*, Math. Prog. **45**, 503 (1989).

[25] C. G. Broyden, *The convergence of a class of double-rank minimization algorithms 1. general considerations*, IMA J. Appl. Math. **6**, 76 (1970).

[26] R. Fletcher, *A new approach to variable metric algorithms*, Comput. J. **13**, 317 (1970).

[27] D. Goldfarb, *A family of variable-metric methods derived by variational means*, Math. Comp. **24**, 23 (1970).

[28] D. F. Shanno, *Conditioning of quasi-newton methods for function minimization*, Math. Comp. **24**, 647 (1970).

[29] D. Wales, *Energy Landscapes: Applications to Clusters, Biomolecules and Glasses*, Cambridge Molecular Science, Cambridge University Press (2004).

[30] R. G. Mantell, C. E. Pitt and D. J. Wales, *Gpu-accelerated exploration of biomolecular energy landscapes*, Journal of Chemical Theory and Computation **12**, 6182 (2016).

[31] S. A. Trygubenko and D. J. Wales, *A doubly nudged elastic band method for finding transition states*, J. Chem. Phys. **120**, 2082 (2004).

[32] S. A. Trygubenko and D. J. Wales, *Analysis of cooperativity and localization for atomic rearrangements*, J. Chem. Phys. **121**, 6689 (2004).

[33] G. Henkelman, B. P. Uberuaga and H. Jónsson, *A climbing image nudged elastic band method for finding saddle points and minimum energy paths*, J. Chem. Phys. **113**, 9901 (2000).

[34] G. Henkelman and H. Jónsson, *Improved tangent estimate in the nudged elastic band method for finding minimum energy paths and saddle points*, J. Chem. Phys. **113**, 9978 (2000).

[35] L. J. Munro and D. J. Wales, *Defect migration in crystalline silicon*, Phys. Rev. B **59**, 3969 (1999).

[36] Y. Zeng, P. Xiao and G. Henkelman, *Unification of algorithms for minimum mode optimization*, J. Chem. Phys. **140**, 044115 (2014).

[37] A. Banerjee, N. Adams, J. Simons and R. Shepard, *Search for stationary points on surfaces*, J. Phys. Chem. **89**, 52 (1985).

[38] O. M. Becker and M. Karplus, *The topology of multidimensional potential energy surfaces: Theory and application to peptide structure and kinetics*, J. Chem. Phys. **106**, 1495 (1997).

[39] D. J. Wales, M. A. Miller and T. R. Walsh, *Archetypal energy landscapes*, Nature **394**, 758 (1998).

[40] F. Despa, D. J. Wales and R. S. Berry, *Archetypal energy landscapes: Dynamical diagnosis*, J. Chem. Phys. **122**, 024103 (2005).

[41] P. C. Verpoort, A. A. Lee and D. J. Wales, *Machine learning landscapes for artificial neural networks (in preparation)* (2019).

[42] D. Rolnick, A. Veit, S. Belongie and N. Shavit, *Deep learning is robust to massive label noise*, arXiv preprint arXiv:1705.10694 (2017).

[43] G. Patrini, A. Rozza, A. K. Menon, R. Nock and L. Qu, in *Proc. IEEE. Comput. Soc. Conf. Comput. Vis. Pattern. Recognit.*, pp. 1944 – 1952 (2017).

[44] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow and R. Fergus, *Intriguing properties of neural networks*, arXiv preprint arXiv:1312.6199 (2013).

[45] A. J. Bekker and J. Goldberger, in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2682 – 2686. IEEE (2016).

[46] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting*, J. Machine Learning Res. **15**, 1929 (2014).

[47] A. Krizhevsky, I. Sutskever and G. E. Hinton, in *Adv. Neural Inf. Process. Syst.*, pp. 1097–1105 (2012).

[48] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun and R. Fergus, in *International conference on machine learning*, pp. 1058–1066 (2013).

[49] A. Labach, H. Salehinejad and S. Valaee, *Survey of dropout methods for deep neural networks*, arXiv preprint arXiv:1904.13310 (2019).

[50] M. Denil, B. Shakibi, L. Dinh and N. D. Freitas, in *Advances in neural information processing systems*, pp. 2148–2156 (2013).

[51] Y. LeCun, J. S. Denker and S. A. Solla, in *Advances in neural information processing systems*, pp. 598–605 (1990).

[52] S. Changpinyo, M. Sandler and A. Zhmoginov, *The power of sparsity in convolutional neural networks*, arXiv preprint arXiv:1702.06257 (2017).

[53] D. B. Kirk and H. W. Wen-Mei, *Programming massively parallel processors: a hands-on approach*, Morgan kaufmann (2016).

[54] S. Shi, Q. Wang, P. Xu and X. Chu, in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pp. 99–104. IEEE (2016).

[55] K. Oh and K. Jung, *Gpu implementation of neural networks*, Pattern Recognition **37**, 1311 (2004).

[56] X. Sierra-Canto, F. Madera-Ramirez and V. Uc-Cetina, in *2010 Ninth International Conference on Machine Learning and Applications*, pp. 307–312. IEEE (2010).

[57] nVidia, *CUBLAS Library User Guide*, nVidia, v5.0 edn. (Oct. 2012).

[58] Y. Fei, G. Rong, B. Wang and W. Wang, *Parallel l-bfgs-b algorithm on gpu*, Computers & Graphics **40**, 1 (2014).

[59] S. Yatawatta, S. Kazemi and S. Zaroubi, in *2012 Innovative Parallel Computing (InPar)*, pp. 1–6. IEEE (2012).

[60] G. Rong, Y. Liu, W. Wang, X. Yin, D. Gu and X. Guo, *Gpu-assisted computation of centroidal voronoi tessellation*, IEEE transactions on visualization and computer graphics **17**, 345 (2011).

[61] J. E. Jones and A. E. Ingham, *On the calculation of certain crystal potential constants, and on the cubic crystal of least potential energy*, Proc. R. Soc. A **107**, 636 (1925).

[62] B. M. Axilrod and E. Teller, *Interaction of the van der waals type between three atoms*, J. Chem. Phys. **11**, 299 (1943).

[63] D. J. Wales, *Exploring energy landscapes*, Ann. Rev. Phys. Chem. **69**, 401 (2018).

[64] L. Deng, *The MNIST database of handwritten digit images for machine learning research*, Signal Process. Mag. **29**, 141 (2012).

[65] B. Irie and S. Miyake, in *IEEE International Conference on Neural Networks*. IEEE (1988).

[66] D. J. Wales and J. P. K. Doye, *Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms*, J. Phys. Chem. A **101**, 5111 (1997).

[67] Z. Li and H. A. Scheraga, *Monte carlo-minimization approach to the multiple-minima problem in protein folding*, Proc. Natl. Acad. Sci. USA **84**, 6611 (1987).

[68] Z. Li and H. A. Scheraga, *Structure and free energy of complex thermodynamic systems*, J. Mol. Struct. **179**, 333 (1988).

[69] D. Liu and J. Nocedal, *On the limited memory bfgs method for large scale optimization*, Math. Prog. **45**, 503 (1989).

[70] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *Equation of state calculations by fast computing machines*, J. Chem. Phys. **21**, 1087 (1953).

[71] J. N. Murrell and K. J. Laidler, *Symmetries of activated complexes*, Trans. Faraday. Soc. **64**, 371 (1968).

[72] J. Baker, *An algorithm for the location of transition states*, J. Comp. Chem. **7**, 385 (1986).

[73] J. P. K. Doye and D. J. Wales, *Surveying a potential energy surface by eigenvector- following - applications to global optimisation and the structural transformations of clusters*, Z. Phys. D **40**, 194 (1997).

[74] F. Rao and A. Caflisch, *The protein folding network*, J. Mol. Biol. **342**, 299 (2004).

[75] F. Noé and S. Fischer, *Transition networks for modeling the kinetics of conformational change in macromolecules*, Curr. Opin. Struct. Biol. **18**, 154 (2008).

[76] D. Prada-Gracia, J. Gómez-Gardenes, P. Echenique and F. Fernando, *Exploring the free energy landscape: From dynamics to networks and back*, PLoS Comput. Biol. **5**, e1000415 (2009).

[77] M. D. Hill and M. R. Marty, *Amdahl's law in the multicore era*, Computer **41**, 33 (2008).

[78] P. A. Braier, R. S. Berry and D. J. Wales, *How the range of pair interactions governs features of multidimensional potentials*, J. Chem. Phys. **93**, 8745 (1990).

[79] J. P. K. Doye and D. J. Wales, *The structure and stability of atomic liquids - from clusters to bulk*, Science **271**, 484 (1996).

[80] D. J. Wales, *A microscopic basis for the global appearance of energy landscapes*, Science **293**, 2067 (2001).

[81] D. J. Wales, *Highlights: Energy landscapes of clusters bound by short-ranged potentials*, ChemPhysChem **11**, 2491 (2010).

[82] P. T. Inc., *Collaborative data science* (2015).

Appendices

A Molecular Configuration Space Distribution

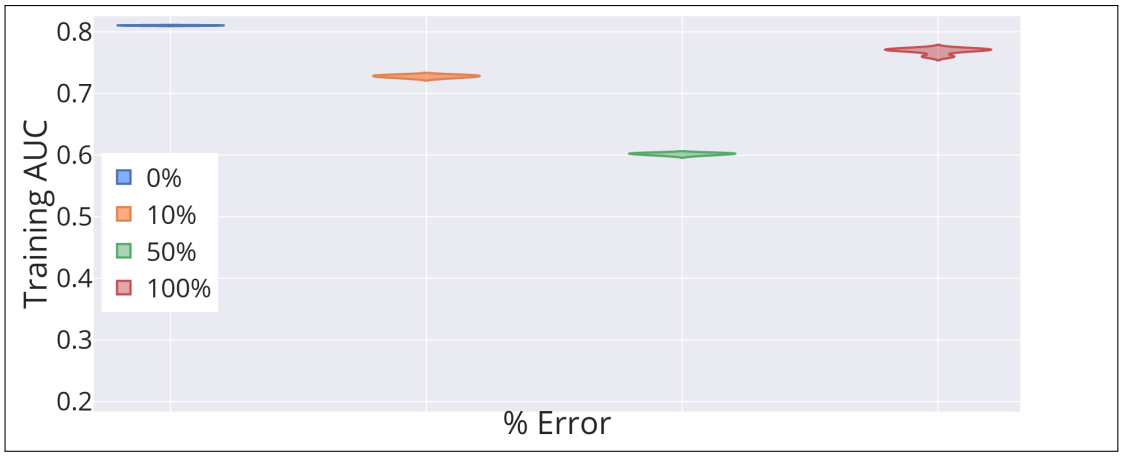
This section contains the molecular configuration space distributions for the D1.2, D2.0 and D3.0 datasets (Table 1). Here, class 0 corresponds to the equilateral triangle structure, and classes 1-3 correspond to linear permutations of the three atoms. Note that for D1.2, the majority of the bond-length data correspond to the equilateral triangle class.

Dataset	**Class 0**	**Class 1**	**Class 2**	**Class 3**
D1.2	759	79	71	91
D2.0	360	204	213	223
D3.0	197	252	274	277

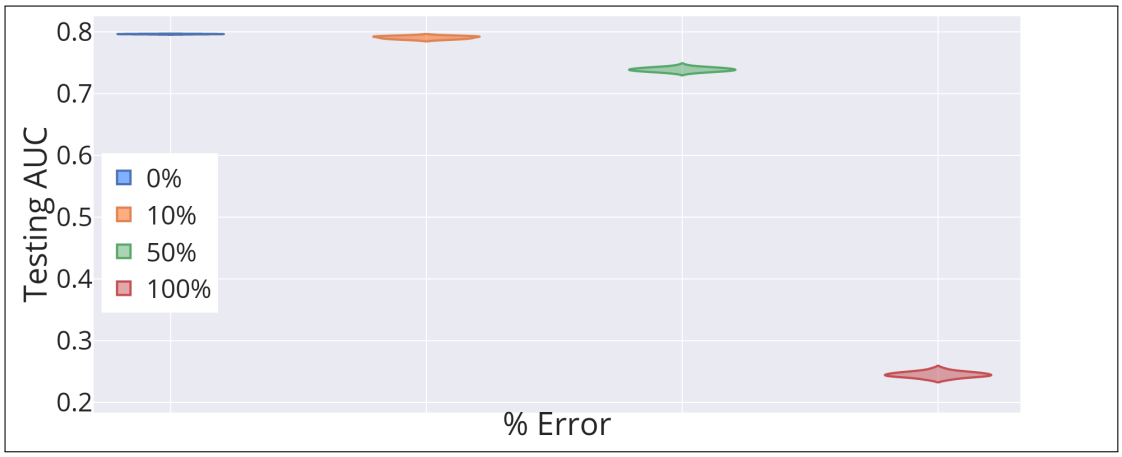
Table 1: Number of entries in each class (0-3) for 1000 training points on the D1.2, D2.0 and D3.0 datasets.

B Training and Testing AUC Distribution

This section contains violin plots for training and testing AUCs for the D1.2-D3.0 datasets as a function of the mislabelling error (Figures 8-10).



(a) Training AUC for various percentages of mislabelled data.



(b) Testing AUC for various percentages of mislabelled data.

Figure 8: Violin plots for the training and testing AUC values for various error percentages on the D1.2 dataset. This plot was created using Plotly [82].

(a) Training AUC for various percentages of mislabelled data.

(b) Testing AUC for various percentages of mislabelled data.

Figure 9: Violin plots for the training and testing AUC values for various error percentages on the D2.0 dataset. This plot was created using Plotly [82].

(a) Training AUC for various percentages of mislabelled data.

(b) Testing AUC for various percentages of mislabelled data.

Figure 10: Violin plots for the training and testing AUC values for various error percentages on the D3.0 dataset. This plot was created using Plotly [82].