MASTER'S THESIS

Artificial Neural Networks to Solve Inverse Problems in Quantum Physics

VICTOR LANTZ
TFY14VLA@STUDENT.LU.SE

SUPERVISORS:
MATS-ERIK PISTOL
MATS-ERIK.PISTOL@FTF.LTH.SE
NAJMEH ABIRI
NAJMEH.ABIRI@THEP.LU.SE
GILLIS CARLSSON
GILLIS.CARLSSON@MATFYS.LTH.SE

Abstract

Inverse problems are important in quantum physics as their solutions are essential in order to describe a number of systems using measurable information, e.g. excitation energies or material properties. The problem with inverse problems is that they are usually very hard to solve. One method that could be useful in solving these problems is artificial neural networks. Artificial neural networks have received a lot of attention lately as they have shown great results in solving many difficult problems e.g. super-resolution in imaging [1], [2]. However, their use in the field of physics have been limited so far.

In this master's thesis, neural networks have been applied to a few inverse quantum mechanical problems to see if it is possible for them to solve these problems. The inverse problems that are in focus in this project are: solving the external potential of quantum mechanical systems using either the eigenvalue spectrum or the density function. Lastly, the task of going from potential to density function is also treated. In the project the problems were limited to their 1D form with computationally generated data that made use of a finite difference method.

All of the problems investigated in the thesis have been successfully solved using dense networks. The inverse problem where the potential was computed using the eigenvalue spectrum was solved using a custom error function. This error function accounted for the 1D potentials having the same eigenvalue spectrum when reversed along spatial axis. It was shown that the network predictions were very close the the actual potentials. Both the density to potential problem and its reverse problem were solved as well. These results showed not only good predictions, but also that the networks were able to generalise well to particle numbers they had not trained on.

Based on the results, it has been shown that it is possible to solve these problems using artificial neural networks. Now that this has been shown, the next step would be to apply this method to real data in order to create solution tools that could have practical use for real problems. Additionally, it is likely that there are other problems in the field where neural networks could prove useful.

Acknowledgements

I would like to thank my supervisors for all their time and dedication to this project. Your comments and and input have helped me create something that I am very proud of, and without you, this project would not have been as fun and exciting as it has been.

Finally, I would like to thank my girlfriend Sara for her continuous support, encouragement and valuable ideas during this project. You are truly the best!

Nomenclature

ANN	-	Artifical Neural Network
BVP	-	Boundary Value Problem
DFT	-	Density Functional Theory
MAE	-	Mean Absolute Error
MSE	-	Mean Squared Error
PDE	-	Partial Differential Equation

Contents

1 Introduction

In all fields of science, there are problems described by complex systems of partial differential equations (PDEs) that relates to something observable [3]. Usually, these problems are solved by inserting known system parameters into the PDEs. This generates the value of the observable parameter. On the contrary, inverse problems are problems where the system parameters are recreated from the external parameters [4]. The thing that makes the inverse problems interesting is that many of the system parameters that are of interest are usually not accessible by measurements [3]. This means that they must be acquired from external parameters that are possible to measure [4].

However, the inverse problems are notorious for being difficult to solve. Because the measurements are subjected to measurement noise, the inverse mapping is usually ill-posed [4], and the might not even have a unique solution [3]. The reason for the latter is that the set of measured parameters could be the result of several combinations of the systems intrinsic parameters. Because of this there is no a general way to solve inverse problems and solutions require the usage of some prior knowledge of the system [3].

The inverse problem that is one of the topics in this project is the inverse spectral problem, where the goal is to calculate the atomic potential from its energy spectrum. The energy spectrum is usually described by the PDE commonly known as the Schrödinger equation. There is a theorem concerning the inverse of the Schrödinger equation called the Borg-Marchenko theorem. The theorem states that a potential and two boundary conditions can be uniquely determined by a discrete set of eigenvalues [5]. However, the theorem states that there is a solution to this problem, but it does not say anything about how to solve it.

In many quantum mechanical problems it is possible to get the energy spectrum by absorption or emission measurements, but it is not possible to directly measure the potential. Therefore, the inverse Schrödinger equation is an interesting problem to solve. Solving the inverse of the Schrödinger equation could also be seen as the first step in order find the mean-field potentials of systems using measured spectra.

Density functional theory (DFT) is a modelling method that is widely used to compute a variety of system properties using the particle distribution. In DFT, the particle distribution in a system is seen as the fundamental variable in describing a system [6]. However, the issue with DFT is either that the methods are computationally intense, or that there are no method to compute some system properties. DFT also has issues with the results needing corrections. If there was a way to efficiently improve these calculations it would mean a big difference for our ability to model nature. This could be done either by doing the full computation, or by calculating the corrections.

If the potential is known then everything about the system is known, meaning that all of its properties are known. By having a method that can easily compute the potential from something measurable for example, energy spectrum or electron distribution, then it would be much easier to predict the properties of new materials or pharmaceutical compounds.

Both of these problems could potentially be solved with the use of Artificial neural networks (ANNs). ANNs are a tool that have been rigorously used in research recently, and has shown great success in several fields. Most prominent has been the recent advances in the field of image analysis where the neural networks have been a central part. Here, deep neural networks have been used for, e.g., super-resolution, which is a difficult ill-posed problem where the goal is to get a high-resolution image by up-scaling one with low resolution [1], [2]. ANNs has also been shown to be able to compute eigenvalues and eigenvectors of symmetric matrices. The benefit of using a network for this problem is the ability to parallelise the process, which speeds up the computing [7]. They have also been shown to be able to solve ill-posed inverse problems, which appear in several different scientific fields [4].

Currently, there is not much research about the application of neural networks in the field of physics. This fact, in combination with the results of other studies, is the reason for it being interesting to investigate how well ANNs work on the previously stated problems.

1.1 Problem formulation

The focus of this project is to test if ANNs have uses in physics. To investigate this, there are two main problems to which ANNs are applied in an attempt to solve them.

The first main problem to solve in this master's thesis is to create a neural network that can solve the inverse spectral problem, which essentially is to solve the inverse of the Schrödinger equation for a potential well. This means that the network will use a subset of eigenvalues as inputs to compute the potential that gives the eigenvalue spectrum. If this is successful, one of the follow-up questions would be how the accuracy of the solution is dependent on the number of eigenvalues used as input.

The second main problem treated in this project is to compute the potential and number of particles using ANNs, which would show that ANNs can solve DFT problems. This network would use density function describing the particle distribution of a ground state as the inputs to solve the problem. Running this process in reverse would also be interesting to see if it is possible for a network to compute the ground state density function given a potential and a number of particles.

1.2 Problem scope

The scope of this project is to see if the previously mentioned problems could be solved using ANNs. However, it does not include any data gathering from real systems as the models make use of simpler 1D cases of the problems where the data is generated as a part of the project. This makes the problems simpler to model, and makes it easier to see if it is even possible to solve these problems for such systems, since if this does not work for simple systems then it most likely will not work for complex ones.

2 Background

This section will treat the underlying theory used in the project. It will start with explaining the physics of the problems, then continue on with the numerical methods used in the solutions. Finally, the section will close with a part about the artificial neural networks and the libraries used to implement them.

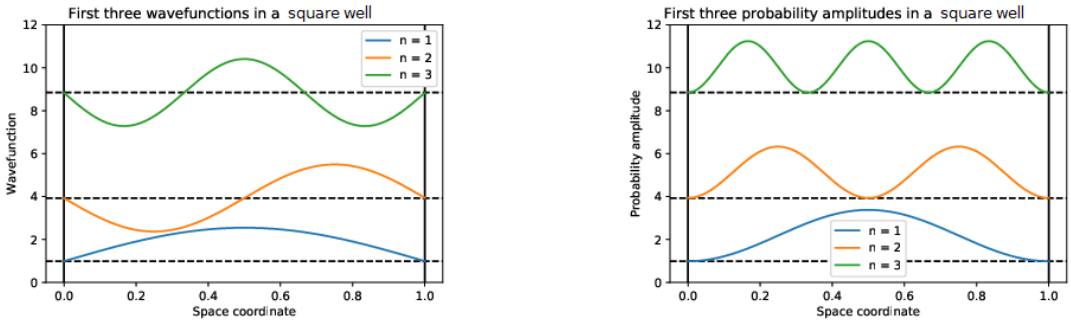
2.1 The spectral problem

A quantum mechanical state or wave function is described by the eigenvalue equation called the Schrödinger equation. There is a special case of the Schrödinger equation that is time independent, which is aptly named the time-independent Schrödinger equation. It is this variant of the Schrödinger equation that is of interest when solving the spectral problem, since these states are the stationary states of a particle stuck in a fix potential. This means that they do not change with time. The time-independent Schrödinger equation is shown in equation (1), where m is the particle mass, $V(\bar{r})$ is the potential, E is the energy, $\hbar$ is Planck's constant and ψ is called the wave function [8]. From now on in this rapport when referring to the Schrödinger equation, it refers to the time-independent variant.

$$\frac{-\hbar^2}{2m}\nabla^2\psi + V(\bar{r})\psi = E\psi \tag{1}$$

A particle trapped in a fix potential can only possess some discrete energy values that depends on the shape of the potential. This set of discrete energies are the energies that are the solution to the Schrödinger equation. The distance between the energy levels of a potential well is described by the energy level's dependence on an integer called the quantum number that is usually denoted n. For example, the energies of a square well potential with infinitely high walls have square dependence on n, while the energy levels in a harmonic oscillator potential is equidistant, meaning that the dependence is linear instead [8].

The wave functions ψ are the eigenfunctions to the Shrödinger equation and describes a particle in a quantum mechanical state, e.g. a bound state in a potential well. However, for a more physical interpretation one must look at the absolute square of the wave function. The absolute squared wave function is usually called the probability amplitude and gives the probability of finding a particle at a certain location [8]. An illustration of the wave function and the probability density for a square well is shown in Figure 1. This image also shows the square n dependency of the state energies previously discussed (dashed lines).



(a) *(b)*

Figure 1: 1a illustrates the wavefunction of the three first states in a square well potential. 1b illustrates the probability density of the first three states for the same potential. In both cases the functions are centred around their respective state energy, which are represented by the dashed line.

An electron in the potential well can occupy one of the energy levels and it can then jump to another level by either absorbing or emitting energy that correspond to the energy difference between the initial and final energy level [8]. It is these energy level differences that can be measured by measuring the absorption or emission spectra.

2.2 Density function

The density function $\rho(\bar{r})$ describes the particle density of a system. For a discrete position coordinate, the density function is described in a point x_j by the sum of the probability amplitude of every occupied state i. The probability amplitude is, as previously mentioned, the value in point x_j of the eigenvector ψ corresponding to the occupied state, multiplied with its complex conjugate, resulting in equation (2).

$$\rho(x_j) = \sum_{i=1}^{N} \psi_i^*(x_j) \cdot \psi_i(x_j) \tag{2}$$

Because the density function describes how the particles distribute over an area in space, the integral of the density function over that area should result in the particle number N. Due to space being discretised in this case, the integral is replaced by a sum over all K points. This gives the relation between the density function and particle count represented in equation (3).

$$N = \sum_{j=0}^{K} \rho(x_j) \tag{3}$$

The density function is central in DFT where it is seen as a fundamental variable from which the state of a system in an external potential can be described [6]. There is also a lemma saying that the density function of the ground state determines the potential uniquely (up to an additive constant) [9]. From this, the eigenvalues and eigenvectors and all other properties of the system can then be acquired [9]. This lemma is connected to the Hohenberg-Kohn theorem, which states that for a many particle system in its ground state, the total energy is a functional of the density function [6]. However, this theorem does not say anything about the functional other than its existence.

2.3 Finite Difference Method

The Schrödinger equation is a boundary value problem (BVP) with a linear second order partial differential equation. A BVP is a problem that depends on a number of constraints (most commonly on the boundary), where the number depend on the dimensionality of the problem [10]. In this case, it is a second order partial differential equation (PDE) with constraints on the boundary. The constraints on the boundaries can be of different kinds, the ones most relevant to this work are the Dirichlet boundary conditions. Dirichlet conditions mean that the values on the edges are set to constant values [10].

When looking at the Schrödinger equation, one question that comes to mind is how to treat the second order differential operator when solving it numerically. One approach is the Finite differences method where the seconder order differential operator is approximated using the function value $\phi(x_k)$ of a point x_k and values of nearby points [10]. This can be done in different ways, but the one used in this project is the centroid difference which uses the two adjacent points $x_{k\pm1}$ as shown in equation (4) [10].

$$\frac{\partial^2 \phi(x_k)}{\partial x^2} = \frac{\phi(x_{k-1}) - 2\phi(x_k) + \phi(x_{k+1})}{\Delta x^2} \tag{4}$$

This equation only describes the relation for one point x_k, but can be expanded as an equation system for every point on a discrete grid. By doing this, the differential operator can be expressed as a matrix as in equation (5) [10]. The matrix could then operate on a vector representation of ϕ by multiplying the matrix in from the left.

$$\frac{\partial^2}{\partial x^2} = \frac{1}{\Delta x^2} \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & & \vdots \\ 0 & 1 & -2 & 1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 1 & -2 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & -2 \end{pmatrix} \tag{5}$$

The boundary conditions that was used here was homogeneous Dirichlet conditions, which means that the value of the wave function outside the grid is set to 0. This means that the term ”outside” the first and last row is set to zero and does not need any special treatment. [10]

2.4 Artificial neural networks

ANNs are computational structures inspired by the brain. Just like its namesake, the ANNs consist of several connected cells that send and receive signals, and together do computations. But in order to solve problems it must first learn how to, by training on problems with known solutions. What the network does is that it tries to approximate the function $F : X \rightarrow Y$ that maps the input X to the target output Y [11].

The ANNs can be used to solve different kinds of problems like classification problems and regression problems. The classification problems are essentially labelling the inputs with the correct class label, which could be likened to the task of putting apples in one basket and bananas in another. In regression type problems, the task of the network is instead to predict numerical values.

2.4.1 Single-layer perceptron

The most basic ANN structure is the perceptron. What it does is that it takes several inputs, multiplies them with their corresponding weights, and compute an output. An illustration of a perceptron is shown in Figure 2 [11].

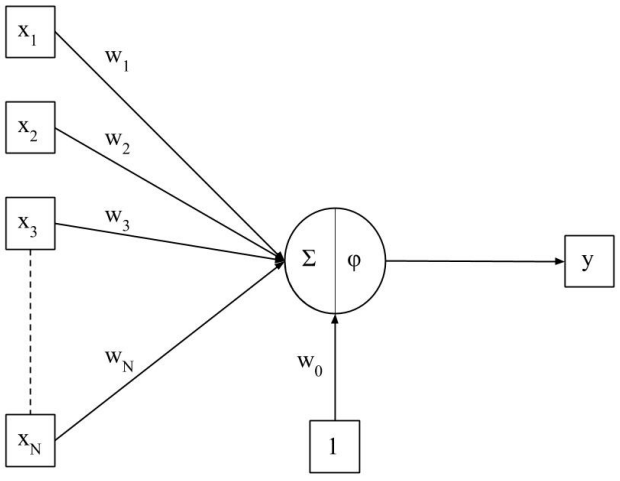


Figure 2: An illustration of the perceptron with the inputs $x_1 \ldots x_N$ and output y.

The output is computed by inserting the sum of all N inputs weighted by w_n and the bias term w_0 into the activation function ϕ as in equation (6) [11].

$$y(x) = \phi\left(\sum_{n=1}^{N} w_n x_n + w_0\right) \tag{6}$$

There are a number of commonly used activation functions that are useful in different scenarios. In this work, the activation functions used are the linear activation function, the rectified linear unit (ReLU) and the softmax function. The linear activation function, shown in equation (7), is commonly used as an activation function for the output layer in regression-type problems as it can assume all values, which is important in regression. The ReLU function in equation (8) is a non-linear function that is usually found as an activation function for the hidden layers in the network, which will be explained in the following section. Lastly, the softmax function in equation (9) is the activation function of the output layer in a multiclass classification problems, where it gives the probability of an instance belonging to the different classes [11].

$$\phi(x) = x \tag{7}$$

$$\phi(x) = \begin{cases} x \text{ for } x \geq 0 \\ 0 \text{ for } x < 0 \end{cases} \tag{8}$$

$$\phi(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{i'}^{K} e^{x_{i'}}}, \quad \mathbf{x} = [x_1, x_2 \dots x_K] \tag{9}$$

2.4.2 Multi-layer perceptron

It is possible to create more complex network structures by connecting layers of parallel perceptrons. This can be done in many different ways, but in this project simple feed-forward networks are used, where one layer feeds into the next as illustrated in Figure 3. The network shown in the Figure is a dense network since each node is connected to all nodes in the next layer. By having more layers and nodes, the network is able to estimate more complicated functions. The layers between the input and output layers are usually called hidden layers. Having non-linear activation functions in the hidden layers such as the ReLU also improves the networks ability to model complex relations between the input and output [11].

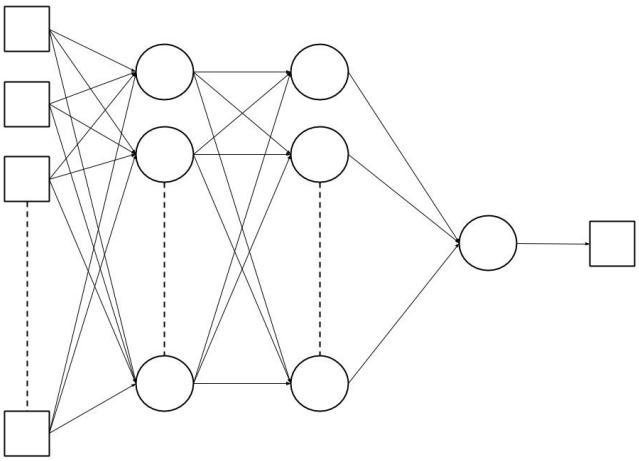


Figure 3: A schematic of a dense multilayer feed-forward network.

2.4.3 Training the network

As mentioned at the beginning of the section, the network needs to train on problems where it knows the solution. This is done by having data sets where an input vector has a corresponding output, and minimising the error between the network output and the target. The error is computed using an error function that in some way uses the difference between the network output and the target value. There are several commonly used error functions, two of the most common ones for regression type problems are the Mean Square Error (MSE) shown in equation (10) [11], and Mean Absolute Error (MAE) shown in equation (11) [12]. The index n notates every data instance in the update batch with N entries, $y(\mathbf{x_n})$ is the output acquired from input vector $\mathbf{x_n}$ that is compared to the target output d_n.

$$E(\boldsymbol{w}) = \frac{1}{2N}\sum_{n=1}^{N}\big(d_n - y(\boldsymbol{x}_n)\big)^2 \tag{10}$$

$$E(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}|d_n - y(\boldsymbol{x}_n)| \tag{11}$$

The goal is to find the weights that minimise the error function. This is done by iteratively updating the weights for a number of iterations, or epochs as they are usually called, then hopefully a minimum is found. Many of the methods used for weight update are derived from the gradient descent method. This method updates the weights by calculating the gradient of the error function with respect to a weight and taking a step in its negative direction. The mathematical representation of this is equation (12), where w_k is one weight and η is the learning rate, which is a factor that decides the size of the step [11].

$$\Delta w_k = -\eta\frac{\partial E(\boldsymbol{w})}{\partial w_k} \tag{12}$$

This can also be done in smaller batches where the weights are updated after calculating the error on parts of the set and normalising by the number of instances in the smaller batch. This is called stochastic gradient descent and is acquired from equation (12) by taking the average of P instances, resulting in equation (13) [11].

$$\Delta w_k = \frac{1}{P}\sum_{p=1}^{P}\Delta w_{pk}, \quad \Delta w_{pk} = -\eta\frac{\partial E_p(\boldsymbol{w})}{\partial w_k} \tag{13}$$

How well a network performs depends on many things, e.g. how the data is formatted, the structure of the network and the activation functions used in the different layers. All of these things are usually summarised as hyperparameters [11].

When training a network, the data set is usually split up into a training set, a validation set, and a test set. The training set is the largest set and it contains all the samples the network trains on. The validation set is also used during training, but not to update the weights. Instead, the validation set is used to see how the network performs on data it has not trained on. When selecting models, several models are trained and usually the ones with the best validation performances are chosen. The test set then works as a final validation after the model selection. This is done because the networks picked from the model selection may have a bias on the validation set, since they were selected from their performance on that set.

2.4.4 Generalisation & Regularisation

A good model is a model that generalises well, meaning that it will perform well on data that it has not encountered in its training. However, that is not always the case since the model sometimes performs really well on the training data and not on other data. This indicates that the model is overtrained, which

essentially means that the model is too specialised to solve the training problems and is unable to solve other problems. To prevent overtraining/overfitting, regularisation methods are usually introduced [11].

One method that is often used is called dropout. Dropout works by adding a layer that sets the outputs of randomly chosen nodes to zero during training. This essentially removes a specified percentage of the nodes and forces the network to adapt to be able to solve the problems even if some nodes are disabled. The result of this is that the network gets more general. [11]

L2 regularization, or weight decay, is a penalty enforced on the network error function that is directly related to the size of the weights. The penalty is introduced as the so called L2-term described by equation (14). This term is added to the error function and increases the value of the error function as the values of the weights increase. This helps in the generalisation of the model because the weights usually grow very large when the model is overly specialised. As seen in the equation, L2 regularisation introduces another hyperparameter called the L2 regularisation constant and is denoted α [11].

$$\frac{\alpha}{2}\sum_i w_i^2 \tag{14}$$

Another cause of overtraining is that the model trains for too many epochs. This effect can be avoided by implementing the early stop or model checkpoint callbacks. Early stopping ends the current model training session when the validation performance of the model starts to get worse. Model checkpoint instead lets the model train for all the epochs, but after each epoch, it will check if the validation performance is better than the previously saved model and if it is better, the current model will be saved as the new best model [11].

2.4.5 The autoencoder structure

A network structure that was used much in this project was the autoencoder structure. An autoencoder is a network that is trained to recreate the input it was given. This problem is trivial for networks where all layers have the same size as the input, but when having a smaller layer in the middle the problem gets harder. The smaller layer called bottleneck is trained to take out K features from which the input can be recreated, where K is the layer width. The part of the network leading to the bottleneck that transforms the input vector into a vector of features is called the encoder, while the part that does the opposite is called the decoder. A scheme of this structure is shown in Figure 4 where the input vector is called $\boldsymbol{X}$ and the feature vector is called $\boldsymbol{Z}$ [11].

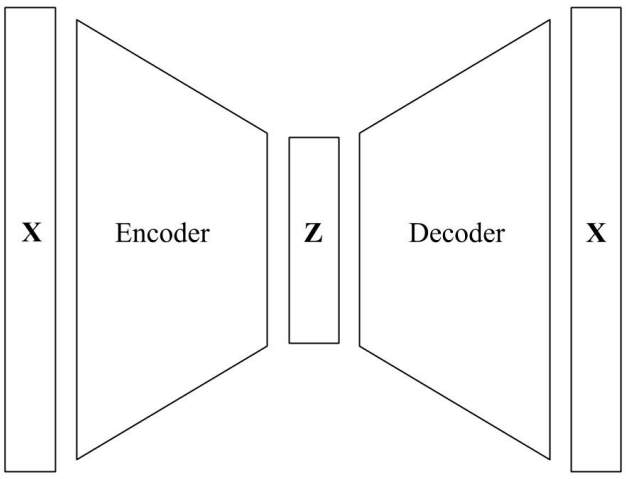


Figure 4: A scheme showing the structure and parts of an autoencoder.

One thing that makes the autoencoder interesting is that the structure can be deconstructed into an encoder and a decoder. Two separate networks are acquired by doing this where the former turns the input into the trained features while the latter turns a vector of features into the same type of data as the original input. This means that it is possible to transform one data-type into another by using the appropriate network.

The reason for the autoencoder structure being used was that it could possibly be easier solve the inverse spectral problem using it. The reasoning behind this was that it would be easier to train a network that computed the potential from the eigenvalues as the decoder part of an autoencoder that took the potential as input.

2.5 Keras

Keras is a Python library for neural network applications. It allows the user to more easily assemble the networks by building the networks layer by layer. This makes it easier to visualise the structure of the networks. Keras also simplifies the training, evaluation and prediction processes. It also allows one to save models in a single file so that it could be loaded and used at later times [13].

3 Modelling the inverse spectral problem

In this section the problem modelling and data generation of the inverse spectral problem will be treated.

3.1 Problem modelling

When investigating the inverse spectral problem a choice was made to model the problem in just one dimension. For the 1D case it is easier to model, generate data sets and to validate the computations as the general shape of the eigenvalue spectrum is easier to compare to known systems (e.g. one dimensional square well and harmonic oscillator).

The first steps were to model the problem and to create the data sets needed to train and validate the ANN. As mentioned in section 2.4, the supposed input and output of the was network needed to be defined. In the inverse spectral problem the goal was to use the energy spectrum to get a potential, meaning that the energy spectrum was the input and the potential was the output of the network.

The 1D problem was modelled was by discretising the potentials along the spatial axis. This means that the potential was represented by a vector, where the position corresponded to a point on the spatial grid, and the entry was the value of the potential in that point. Here, 1001 grid points were used.

This potential vector was then used to calculate the energies. This was done by solving the Schrödinger equation for its energies. Here, a finite differences method was used, since this method comes naturally when having a discretised spatial grid. The reason for this is that the second derivative of the wave function ϕ in every point k can be expressed as in equation (4). By using this, the Schrödinger equation could be rewritten as a matrix equation as in equation (15), which is an eigenvalue problem.

$$(T - V)\phi = E\phi \tag{15}$$

Here, T is the finite differences matrix and V is the potential matrix both shown in (16). V_k is the value of the potential in grid point k, Δx is the distance between grid points and the matrices are both of dimension $N \times N$ with N being the number of grid points. Here, homogeneous Dirichlet conditions were used.

$$T = \frac{-\hbar^2}{2m}\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & & \vdots \\ 0 & 1 & -2 & 1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & 1 & -2 & 1 \\ 0 & \cdots & \cdots & 0 & 1 & -2 \end{pmatrix}, \quad V = \begin{pmatrix} V_1 & 0 & 0 & \cdots & 0 \\ 0 & V_2 & 0 & \cdots & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & V_N \end{pmatrix} \tag{16}$$

The matrices are sparse since they only have entries on the diagonal and a few sub-/super-diagonals. Because of this, a sparse eigenvalue solver from the scipy.sparse library could be used to compute a chosen number of eigenvalues. However, a maximum of $N - 2$ eigenvalues could be computed for an $N \times N$ matrix using this method [14].

3.2 Generating potentials

The data sets used to train a neural network should be diverse in order to make the networks generalise better. The approach used to generate a data set of diverse potentials was to have a few different types of potentials from which every entry of the data set were randomly chosen. These different types of potentials

were polynomial potentials and square wells with either square- or sinusoidal perturbations. Each of the types had different kinds of parameters that had randomised values, creating differently shaped potentials.

The polynomial potentials v_{poly} were generated using the formula shown in equation (17) with the range of the random numbers r_z shown in Table 1. The polynomials here were engineered in a way so that they would have a pretty steep slope and a arbitrarily chosen height of around 6 near the edges in order for them to look like some sort of quantum mechanical potential. However, the height used was arbitrarily chosen, the important thing was that they had similar heights. This was all done by the first term on the right-hand side of equation (17), while the summation term is there to change the shape of the potential.

$$v_{poly}(x) = r_2(2 \cdot (x - 0.5))^{2r_1} + \sum_{n=1}^{2r_1-1} r_{1n}(x - r_{r_2 n})^n \tag{17}$$

Table 1: The random variables used to create the polynomial shaped potentials used in the data sets.

Variable	Range	Distribution
r_1	[1,4]	Random integer
r_2	[5,6]	Uniform
r_{1n}	[-1,1]	Uniform
r_{2n}	$\mu = 0.5, \sigma = 0.1$	Gaussian

For the square wells, the wall height of the potentials h_e was set to a value of 6 (same arbitrary choice as for the polynomials). Then it was a 50/50 random choice of putting a square- or sinusoidal perturbation in the well.

The square perturbation would have the width r_w, which was a random integer between 1 and $\frac{1}{10}$ of the number of grid points rounded down. The starting point of the perturbation r_s, was described as shown in Table 2, in order for the perturbation to not have a grid point adjacent to the walls. The height r_h was uniformly distributed between 1 and $\frac{1}{2}$ of the edge height. An equation that describes the square perturbation v_{sq} is shown in equation (18).

$$v_{sq}(x) = h_e \cdot (\delta(x) + \delta(x-1)) + r_h(\theta(x - r_s) - \theta(x - (r_w - r_s))) \tag{18}$$

The sinusoidal perturbations v_{sin} were generated using the formula in equation (19). Here the amplitude r_a was chosen from a uniform distribution between $\pm\frac{1}{4}$ of the edge height rounded down and the frequency r_f was a random integer between 1 and 10.

$$v_{sin} = r_a \sin(2\pi x \cdot r_f) + |r_a| \tag{19}$$

Table 2: The random variables used to generate the two types of square well potentials. h_e is the edge and N is the number of grid points.

Variable	Range	Distribution
r_h	$[1, \frac{h_e}{2}]$	Uniform
r_w	$[1, floor(\frac{N}{10})]$	Random integer
r_s	$[2, (N - r_w + 2)]$	Random integer
r_a	$[-\frac{h_e}{4}, \frac{h_e}{4}]$	Uniform
r_f	$[1, 10]$	Random integer

The data sets were then generated by randomly generating 10 000 entries where there was $\frac{1}{3}$ chance for every potential type to be chosen. Figure 5 shows an example of 10 random potentials generated by the method described.

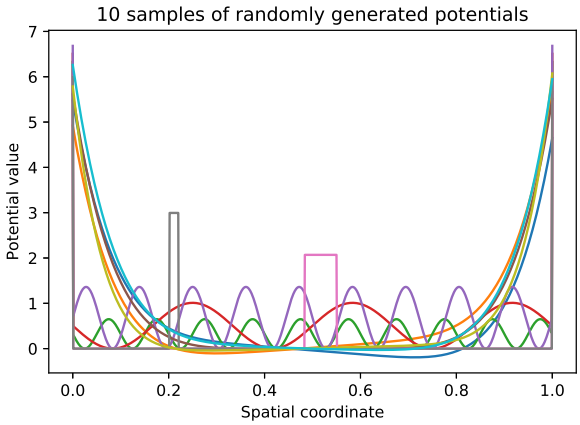


Figure 5: A plot of 10 randomly generated potentials randomly selected from the three different types.

The Schrödinger equation was then solved for every potential using the finite differences method to find the lowest N eigenvalues for every potential computed, where the number N depends on which data set was created. Figure 6 shows the lowest 100 eigenvalues computed using the sparse eigenvalue solver for the harmonic potential shown in the Figure. As can be seen in the Figure, the eigenvalues from the solver show the linear behaviour that is expected of a harmonic oscillator.

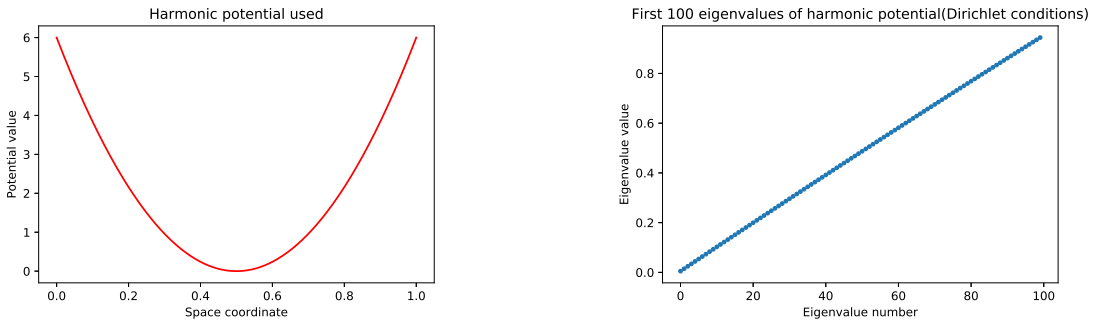


Figure 6: Harmonic oscillator potential (left panel) and the corresponding 100 lowest eigenvalues acquired by the eigenvalue solver (right panel).

The eigenvalues were then put in similar sets as the potentials. The positions in the sets correlated the potential to its respective eigenvalue set. For every data set, there are three sets of potentials and eigenvalues. These are the training set, validation set and test set. These were all printed out as .txt files so that they were stored in files that could be used in other scripts.

3.3 The data sets

Several batches of data sets have been created during the project. The reason for this was that different things were tested along the way, and some of the issues encountered were suspected to be linked to the data set used. This also meant that the data sets used in the different parts of the project were not the same. The different data sets are presented in Table 3.

Table 3: A list of the different data sets used during project for the inverse spectral problem part. All these sets used homogeneous Dirichlet conditions

data set	Eigs	Potential types	Scaled	Train-size	Val-size	Test-size
data1D	900	polynomial,sinus,square	no	8000	1000	1000
data1D_poly	900	polynomial	no	8000	1000	1000
data1D_scaled	900	polynomial, sinus,square	yes	8000	1000	1000
data1D_polyscaled	900	polynomial	yes	8000	1000	1000
data1D_scaledXL	999	polynomial	yes	16000	2000	2000
data1D_noncon	999	non-convex	yes	16000	2000	2000

The first batch of data sets (data1D and data1D_poly) was as previously mentioned a set with all three types of potentials that were not scaled with any physical constants. The number of eigenvalues solved for this set were 900. A second set with just the the polynomial potential type was also created.

After this, two sets with scaled potentials were created (data1D_scaled and data1D_polyscaled). The scaled potentials had all the physical constants (Planck's constant, electon mass etc.) in the calculations and the well width was set to be 1 nm. Here, one of the sets contained all three potential types while the other only contained polynomial potentials.

Later, a scaled set with the maximum 999 eigenvalues was created (data1D_scaledXL), but this set was twice as large, which meant that the training set had 16 000 instances and the validation and test sets had 2 000 each.

The last data set (data1D_noncon) created for the inverse spectral problem was a set where a negative parabolic term was added to the middle of the potential giving a non-convexity to most of the potentials in the set. This non-convex term was described using the expression in equation (20), where the height r_h had a uniform distribution between 2.5 and 4, and the sideways shift r_s had a Gaussian distribution with $\mu = 0.5$ and $\sigma = 0.1$. Ten of the resulting potentials are shown in Figure 7.

$$r_h \left(1 - 2(x - r_s)^2\right) \tag{20}$$

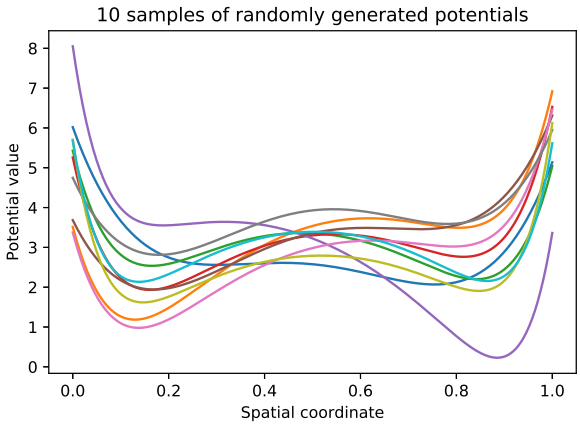


Figure 7: A plot of 10 randomly generated potentials with the added non-convexity term.

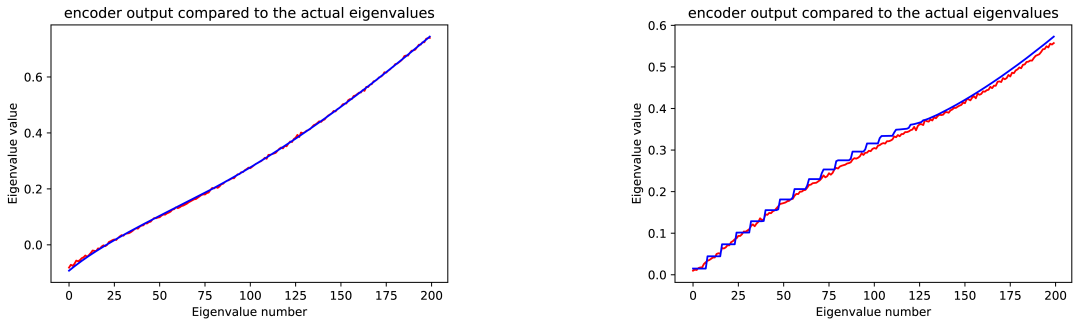
4 Networks for the inverse spectral problem

This section will treat the different approaches that were used when building networks to solve the inverse spectral problem. Then the results of the methods will be displayed. How the number of eigenvalues used affects the model performance will also be shown here. There will also be a summary and conclusion for the inverse spectral problem at the end of the section.

4.1 Autoencoder structure

An initial idea was to create a network that could be used for both the forward and inverse problems. A network structure suitable for this would be some kind of autoencoder that would first be trained to replicate potential. Then split the autoencoder into encoder and decoder that could solve the forward and the inverse problem respectively. However, the tricky part would be to train the autoencoder to have the eigenvalues as the features in the middle layer.

Before assembling this somewhat different autoencoder structure, a dense network were trained to solve the inverse spectral problem. It was shown that this network could not compute the potential from the eigenvalues, which meant that something else had to be done. The forward problem was also tested by computing a subset of the 200 lowest eigenvalues using the potentials. This problem was possible to solve with ANNs for some eigenvalue spectra as shown in Figure 8a, but for more difficult spectra as in Figure 8b it could only follow the trend of the eigenvalues. However, these results showed that the forward problem could be solved, which meant that there was a chance that the inverse problem could be solved using an autoencoder structure.



(a) *(b)*

Figure 8: Outputs (in red) from a simple decoder with 3 hidden layers containing 800, 400 and 200 hidden nodes respectively compared to the target eigenvalues (in blue).

As mentioned, the specifications for the autoencoder was that the features in the bottleneck would be the eigenvalues. To make this possible, the encoder part would first be trained using the potential as input and the eigenvalues as output. This network would then be set to be untrainable, meaning that its weights were fixed. Then a decoder part was made using the outputs of the encoder as inputs. This meant that a new autoencoder model consisting of the encoder followed by the decoder was created. However, when training the autoencoder, only the weights in the decoder would be updated, since the encoder part was untrainable.

4.1.1 Model selection

The first part of selecting the model was to train the encoder. Since going from potential to eigenvalues was an easier task, the method for choosing the encoder was just trying different combinations of hyper-

parameters until the trained model had an MSE that was sufficiently low. That was decided by comparing the MSE values of the encoders during the testings and stopping when one had a much lower MSE than the previously trained encoders. This was done for a different number of eigenvalues by taking the N lowest eigenvalues for the different encoders. The hyperparameters and performance of these encoders are presented with the decoders in Table 4.

Table 4: The different encoders used. The number of eigenvalues used in the training of the encoder is the first number in its name.

Model	Nodes	l_r	$L2$	Val MSE	Test MSE
200-encoder	[800, 600, 400]	0.01	0.001	$1.5094 \cdot 10^{-4}$	$1.4307 \cdot 10^{-4}$
200-encoder-polyset	[800, 600, 400]	0.01	0.001	$1.0054 \cdot 10^{-4}$	$6.1343 \cdot 10^{-5}$
200-encoder-scaledpoly	[600, 400, 200×4]	0.001	0.0001	$2.1611 \cdot 10^{-5}$	$2.2313 \cdot 10^{-5}$
100-encoder-scaledpoly	[700, 500, 300]	0.01	0.001	$1.387 \cdot 10^{-4}$	$1.4926 \cdot 10^{-4}$
900-encoder-scaledpoly	[960, 940, 920]	0.01	0.001	$6.4279 \cdot 10^{-4}$	$6.4454 \cdot 10^{-4}$

When the encoder was trained and the general structure of the network was cemented, the layers and hyperparameters of the decoder could be tweaked. This was done by randomly generating the decoder architecture and hyperparameters (within specifications). The number of layers in the decoder was randomly selected between 3 and 6 with the number of nodes in each layer being less than or equal to the number of nodes in the previous layer, while being less than or equal to the number of output nodes.

The other hyperparameters are presented in Table 5. The reason for the batch sizes to differ so much was to see if it made a difference to use batches, or if it was better to just load the whole training set at once. Loading the whole set makes the training less susceptible to outliers or grouping in the data. Batches smaller than 64 was not used since the variances between the batches could be too large as three different potential types were used.

Table 5: Table of the hyperparameters that where randomly selected in the model selection. All distributions are uniform.

Parameter	Range
Dropout probability	Random half-integer in range [0,0.45]
Learning rate	10^{-r}, where r is random integer in range [2,5]
l2 constant	10^{-r}, where r is random integer in range [1,5]
Batch size	Either 64 or 8000
Epochs	Random integer in range [30,70]

To find the best autoencoder models, a loop was implemented that generated networks with random parameters and then trained those models. From this loop, the networks with least validation MSE were extracted and then evaluated using the test set, which was up to this point unused. One of the best models had a validation MSE of 0.02922. It used the 200-scaled-poly encoder. The decoder of this model had four hidden layers with [517, 773, 829, 897] nodes respectively, dropout of 0.02 between layers, learning rate of 0.001, L2 value of 0.0001, batch size 64 and 60 epochs. As a comparison to the MSE, the size of the mean of the validation and test sets were 0.59508 and 0.5775 respectively, meaning that the validation MSE was below 5% of the validation data mean value. From the models that were extracted it was directly concluded that a batch size of 64 was superior since none of them had a batch size of 8 000.

When the autoencoders were giving good results, the decoder parts were extracted from them. The decoders were then tested using the eigenvalues as inputs and comparing the outputs to the actual potentials. The comparison is shown for three potentials in the images in the left panel of Figure 9, and the results are very poor. The question was then: why are the decoders so bad? One possible explanation was that the encoder did not perform as well as the error metrics might suggest and that this led to the decoders being trained on the wrong things. To check this, the encoder output was plotted along with the actual eigenvalues in the images to the right in Figure 9, which showed that the encoder was performing well. However, upon closer inspection of the autoencoder, it was shown that the encoder part had changed its weights, meaning that

even though the encoder was supposedly frozen it was changed during the training. The extracted encoder was then compared to the initial encoder for the eigenvalues corresponding to the previous three potentials as shown in the images to the right in Figure 9.

By looking at the output from the encoder extracted from the autoencoder it is obvious that it is not the eigenvalues that are plotted. This means that when the autoencoder gets to decide on it its own, it does not use the eigenvalues as features.

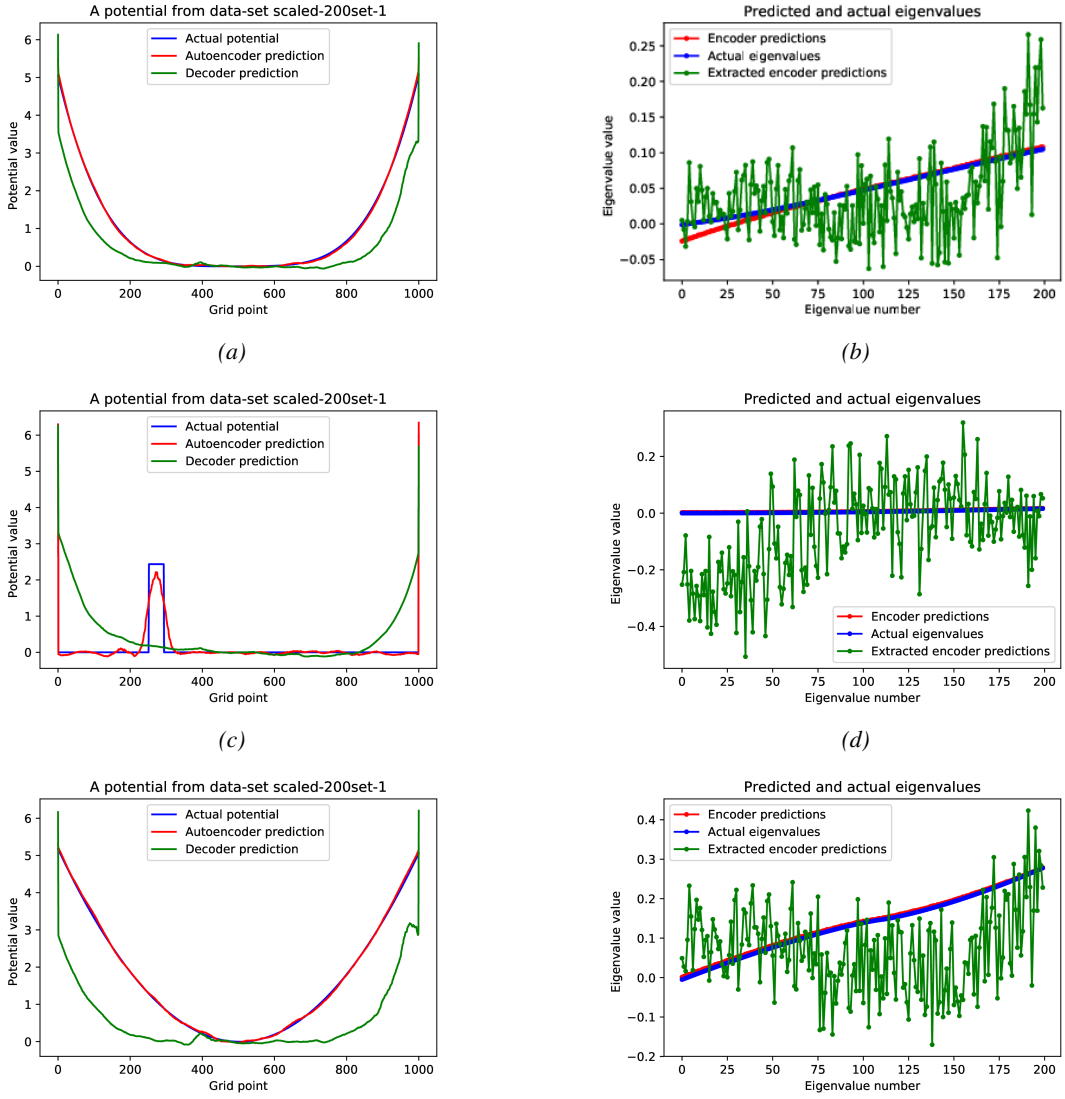


Figure 9: The plots to the left compares the autoencoder and decoder results to the actual potential for three different potentials. To the right, the output from the encoders and the encoders extracted from the autoencoders were compared to the actual eigenvalues corresponding to the potentials.

With the realisation that the encoders were not frozen, the code was altered so that the encoder really was frozen and a new model was trained. However, this autoencoder was not able to reproduce the potential when training, which showed that this approach was not the way to go.

4.2 Feature network

One thing that was noted during the testing with the trainable encoder was that the performance was good when the eigenvalues were not the features. This meant that the upscaling from the features that the autoencoder selected on its own was much better than the one from the eigenvalues. What if a network could get these features from the eigenvalues? Then the well performing upscaling from features to potential that was shown in the autoencoder test could be combined with the usage of eigenvalues as input. This approach also added a bit of flexibility as the number of features did not necessarily have to be the same as the eigenvalues.

The first thing that was done in test was to train a number of autoencoders from scratch with different dimensions of the feature vector. These autoencoders were symmetric, but that should not be a requirement as the symmetry is not used. The autoencoders used performed very well since they were more or less the same as the autoencoders used in the previous section when the encoder was not frozen.

In the next step, a well performing autoencoder was selected and deconstructed into an encoder and a decoder. The encoder was used to transform the potentials into sets containing the corresponding features and the decoder was saved to be used in the final network. Using the new feature data sets, a new network was trained to go from eigenvalues to features. By combining this ”eigenvalue to feature network” with the decoder, a new network was acquired, that would take the eigenvalues and make them into features and then potentials. A schematic of this is shown in Figure 10. A similar network that instead used an eigenvalue autoencoder and a trained feature to the potential network was also tested.

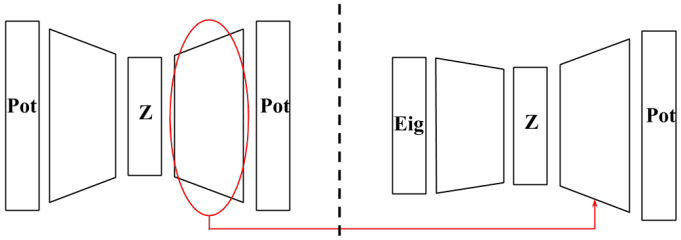


Figure 10: A schematic showing how the feature network was constructed using the decoder from the potential autoencoder combined with the input to feature network.

However, this approach did not work as well as thought since the feature to input networks all had trouble with poor performance which translated into poor performance when used as inputs to the decoder. This meant that this method could be discarded.

4.3 Tied weights

The tied weight approach makes use of the fact that the weights of a decoder can be the transpose of the weights of the encoder [11]. This approach could possibly show good results, since the previous results have shown that the performance of the networks that compute the eigenvalues from the potential has been much better than the networks doing the opposite. By taking a well-performing potential to eigenvalue network and transposing it, a well performing eigenvalue to potential network would hopefully be created.

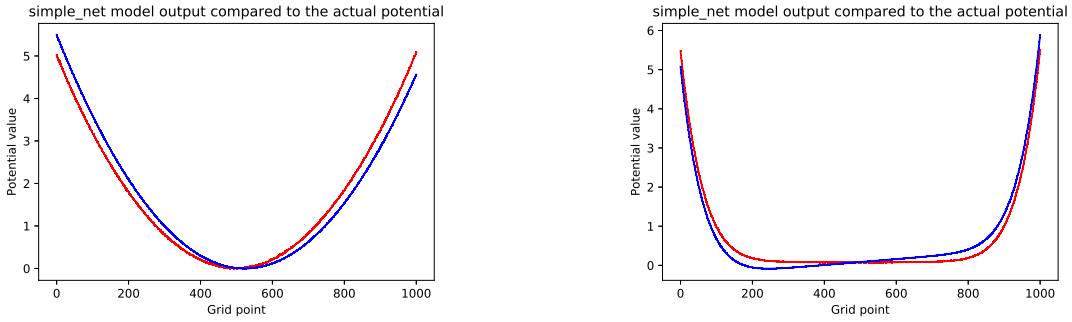
The first issue that needed solving was that there are no good tools in the Keras package for doing this task. This meant that a custom Densetied Keras layer had to be written, which was done by Najmeh Abiri.

The network was then constructed as an autoencoder by creating an encoder made of Dense layers and a decoder made of the custom layers that were tied to their respective encoder layers. The encoder was then trained, which also trained the decoder since it was built using the Densetied layers. The encoder and decoder were then put together into an autoencoder.

The trained encoder showed a good result with low MSE. However, both the decoder and autoencoder did not perform well at all. Their results were similar and looked mostly like noise and did not look like a potential. This meant that this method also did not work.

4.4 Back to basics

The previous attempts that in one way or another made use of the autoencoder structure did not perform well. This led to a direction change back to the simpler Dense networks that were tested right at the start just to check if the code could run. These nets were overlooked at first since the idea of an autoencoder which could be disassembled to two networks for both forward and inverse problems were more interesting. Also, these structures did not show good performance at the start, but when compared to the other failed attempts they did not seem that bad. Both deep and shallow variants were tested and it showed that the shallow models usually performed better. A network with a single 1 000 node hidden layer trained on the large-scaled data set gave the results shown in Figure 11. This showed that the network was close to being able to predict the potentials correctly, but there was something that made it unable to be on point. However, this result was paramount in the search for the next approach.



(a) *(b)*

Figure 11: Comparison between the target potential in blue and the output potential of the simple dense network in red.

4.5 Mirror_MSE error function

The results from all the simple dense networks tested had one thing in common: the resulting potential was always symmetric. This meant that the network did not train to match the potentials, but to find the best symmetric approximation to them. A consequence of this was that the network was not performing well on highly skewed potentials.

This makes sense when thinking about it, since two potentials that are each others' mirror images have the same eigenvalues and when the network tries to solve the problem, it can only guess which way the potential was oriented. If this guess then would happen to be wrong, it would be punished by having a large error. The fact that two potentials give the same eigenvalues is the reason why the networks were trained to give symmetric potentials, because they would learn that it was best to make a potential that minimises the error no matter the potential orientation. This fact was also why it was easy for the ANNs to compute the eigenvalues using the potential, while computing the potential using the eigenvalue spectrum was hard.

This observation led to the idea of writing a custom error function that did not punish the network for choosing the wrong orientation. This error function was a function that compared the normal MSE to the MSE when the target vector was reversed and took the smallest of the two. This error function was called mirror_MSE since it minimised the error compared to the target or its mirror image.

This was then applied to a network that used 999 eigenvalues as input to calculate the 1 001 point potential grid. The network had an input layer of 999 input nodes connected to a hidden layer with 1 000 nodes and the output layer had 1 001 nodes. The mirror_MSE error of this model was $2.3371 \cdot 10^{-4}$ for the data set data1D_scaledXL. In Figure 12 comparisons between network outputs and the corresponding actual potentials are shown. One effect of the mirror_MSE error function is that some of the outputs are mirrored compared to the target, but this could easily be solved by reversing the output as shown in the Figures. The output potentials were more or less on the target potential showing very good performance.

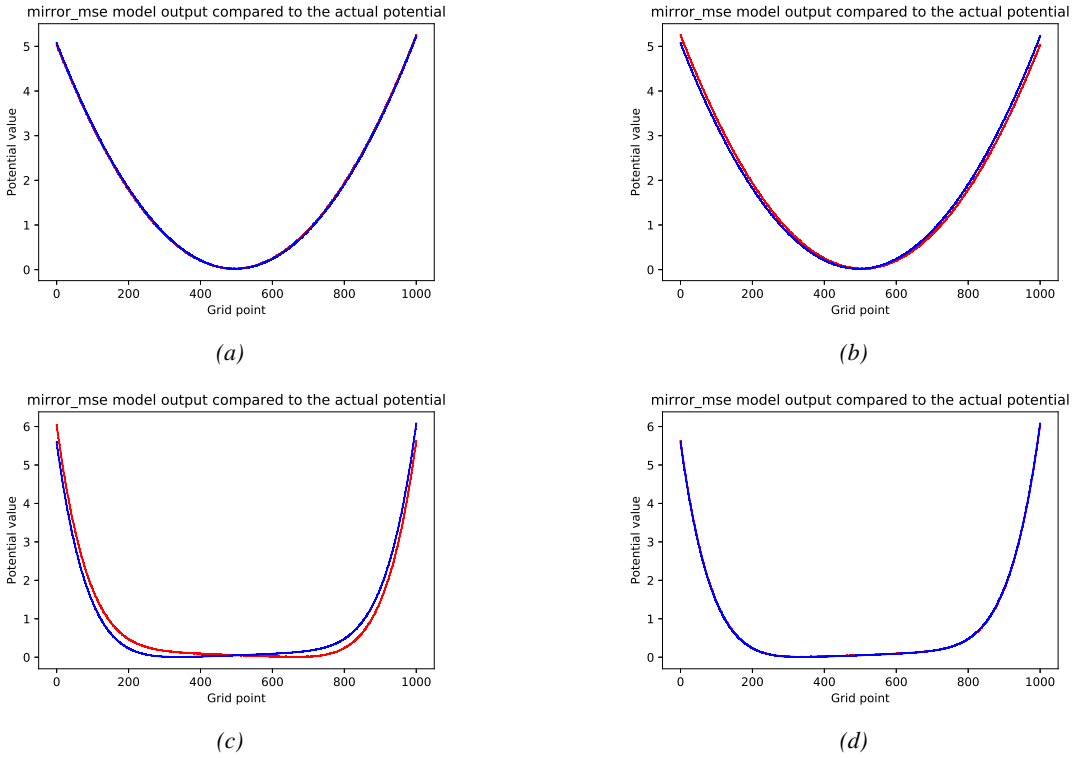


Figure 12: Plots comparing the target potentials (blue) with the outputs of a network using the mirror_MSE error function (red). The models were trained on the training set from the data1D_scaled set and the data generating the plots are from its validation set. The leftmost images are the actual output compared to the potential while the rightmost images are comparing the same output that has been mirrored.

The next step was to push the networks and see how the performance was affected. This was done by using the non-convex data set data1D_noncon, which was thought to be a more difficult problem for the network since this data set included many potentials with two minima: one local and one global. In this test, only the data was different since the new network used the same parameters as the previous one. The mirror_MSE of this network was 0.0029 which was around 10 times larger than for the convex data set. However, by looking at the outputs from the validation set in Figure 13 they still matched the target really well. This shows that even a simple network with only one hidden layer could model a more complex potential shape almost perfectly. This also showed that the network generalises well to potentials it has not encountered since the examples shown are from the validation set.

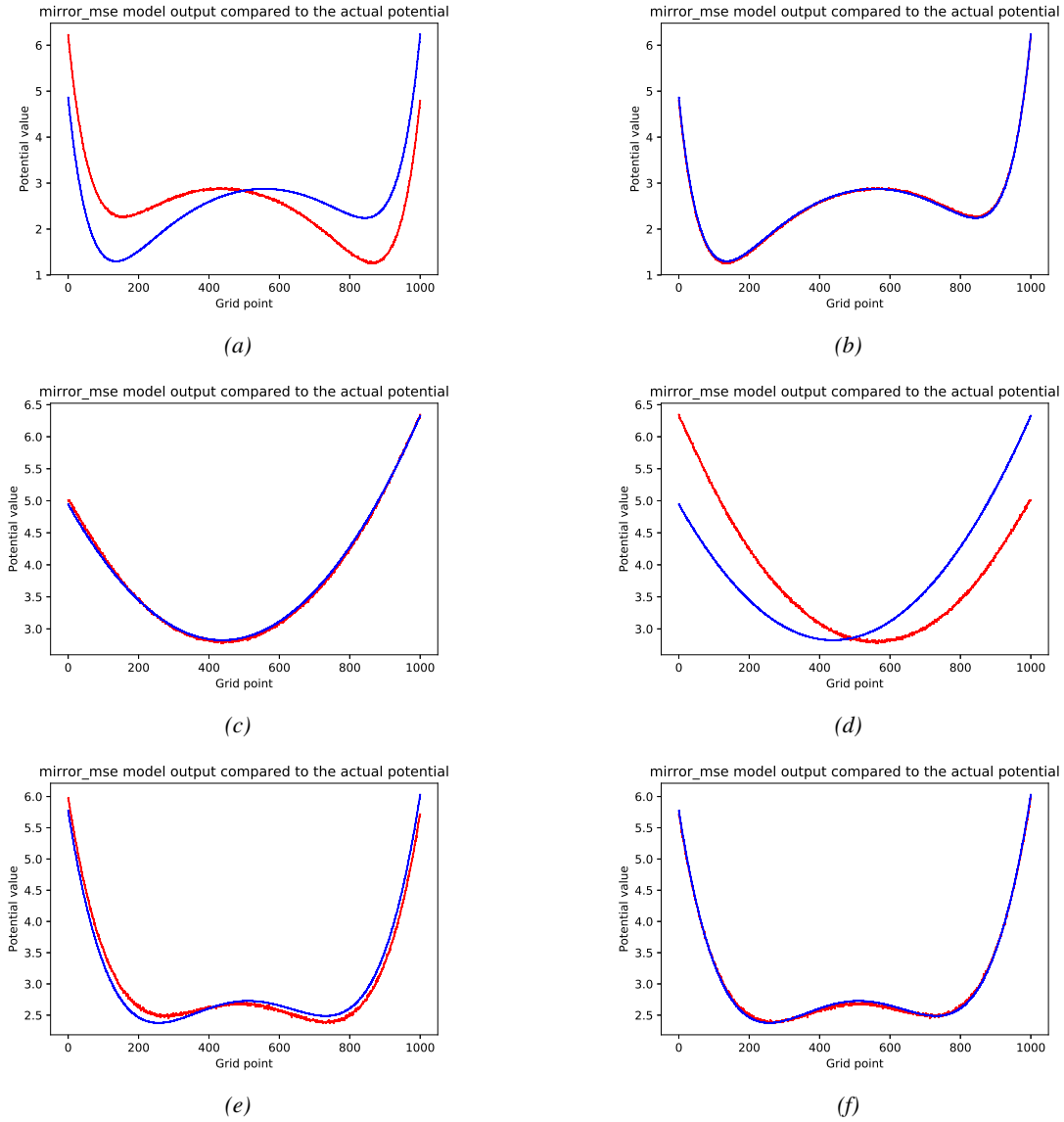


Figure 13: Plots comparing the target potentials (blue) with the output of a network using the mirror_MSE error function (red). The plots are generated from its validation set. The leftmost images are the actual output compared to the potential while the rightmost images are comparing the same output that has been mirrored.

A weighted variant of the mirrored error function was also created and tested. This function was the same ass the mirror_MSE function but every term squared comparison was divided by the exponential function with the target value as the argument. The terms in the error function were formulated as in equation (21).

$$\frac{(d_n - y_n)^2}{e^{y_n}} \tag{21}$$

The reason for using this kind of weight was to make the network to prioritise the lower parts of the potential, since these parts are more important. However, when using the weighted error function the results were considerably worse than when the unweighted variant was used.

4.6 Eigenvalue dependence

Another thing that was investigated was how the performance changed when the number of eigenvalues used was decreased. In order to do this comparison, several networks using the same parameters, but different number of input nodes, were trained using the subset as large as the input layer with the lowest eigenvalues. These networks had a hidden layer with 1 000 nodes. The number of eigenvalues used was 900, 700, 400, 200, 100 and 50. The results for three different potentials are shown in Figure 14. The images to the left compare the outputs of the 900, 700 and 400 input networks to the target potentials, while the images to the right instead show the comparisons with the 200, 100 and 50 input networks. Each row of images in Figure 14 has the same target potential. These results showed a direct correlation between performance and the number of eigenvalues, which could be seen as the networks using the smallest eigenvalue sub-sets performed worst for every potential. However, in some cases the performance was still pretty close even for 50 and 100 eigenvalues. This could mean that it is possible to create networks that have good performance for fewer eigenvalues, since the networks used here were pretty simple using only one hidden layer.

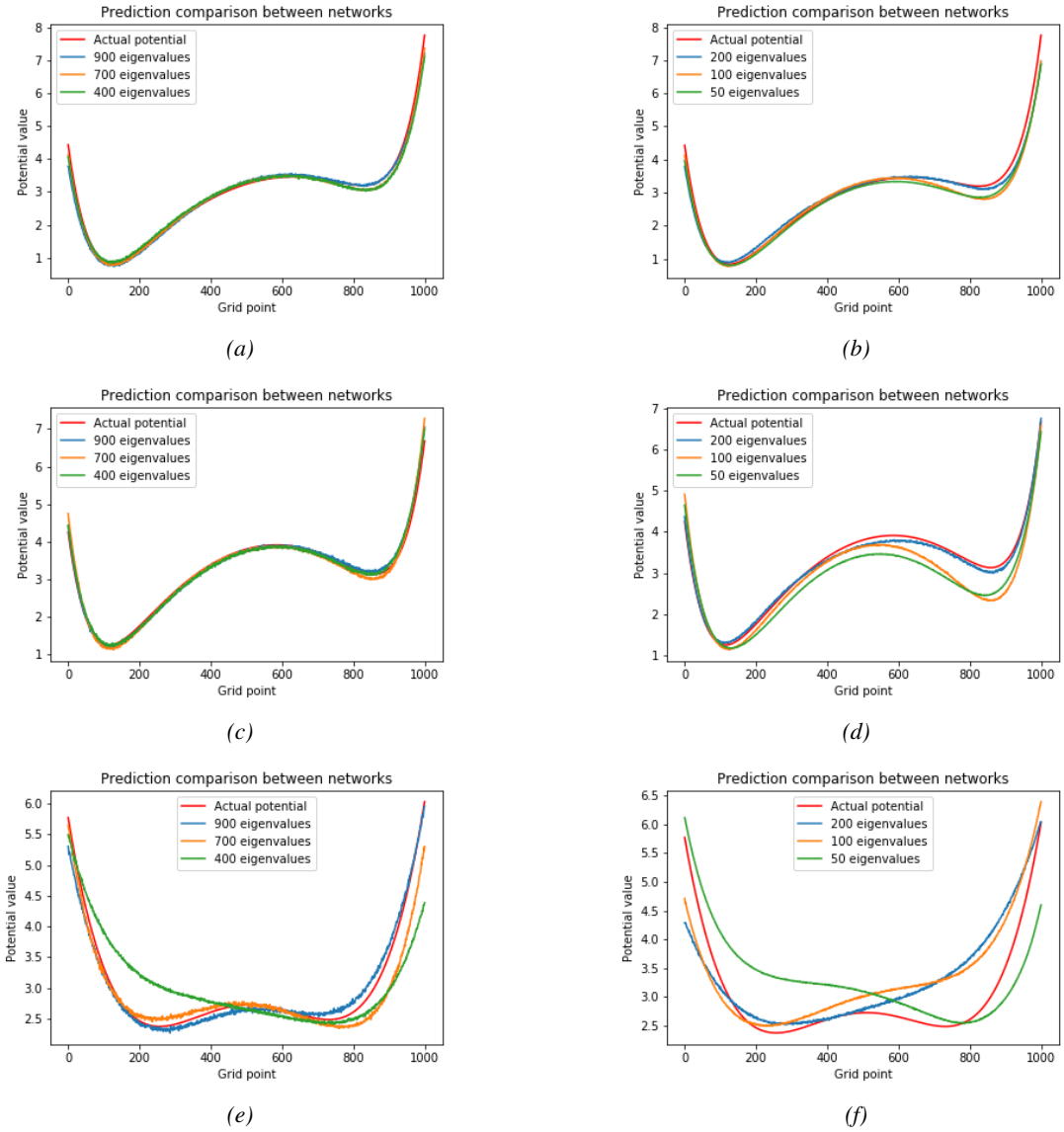
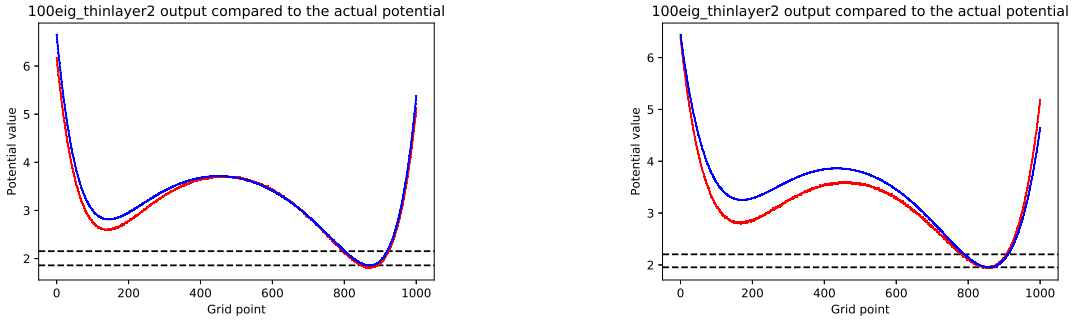


Figure 14: Images that compare the output of networks that use different numbers of eigenvalues as input. The images to the left show the output of the networks using 900, 700 and 400 eigenvalues, while the images to the right shows the output of the networks using 200, 100 and 50 eigenvalues. These networks were all trained using the nonconvex data set data1D_noncon

A natural follow-up on the result in Figure 14 was to see if the performance for a lower number of eigenvalues could be improved by using a deeper network. This was tested by training several models and the model that performed the best was a model called "100eig_thinlayer2" that used 9 hidden layers with node counts [1 000, 800, 400, 200, 100, 100, 100, 100, 100]. One thing that was noticeable for this network was that there was a few outputs that were not even close to the correct potential, while many managed to get the shape right in the bottom of the potential. Figure 15 shows the results for two such potentials.



(a) *(b)*

Figure 15: The output of model "100eig_thinlayer2" in red compared to the actual output in blue. The dashed lines in this plot corresponds to the largest and smallest values in the 100 long input vector.

4.7 Summary & Conclusion

To summarise the inverse spectral problem: the approaches first suggested, which made use of an autoencoder structure did not work. Instead, the problem was solved for both the convex and non-convex potentials by using a custom error function, which compared the output to both the actual potential and its mirror image. The networks used in these solutions were not deep as they only used 2 hidden layers.

The solution of using the custom error function shows that even when using ANN, prior information about the system is needed to solve the inverse problem. The reason to this is that the custom error function made use of the fact that a potential has the same eigenvalues as its mirror image. This also showcased that the ANNs perform poorly when there is a degree of freedom such as the orientation of the potential. The reason for this is that when one input maps to two potential outputs the network does not know which one of those two outputs that is the correct one. To solve, this one must either do was done in the solution with the custom error function that essentially told the network that either of the potential outputs are the "right answer", or by somehow differentiating the two inputs so that the inputs no longer map to several outputs.

One way to differentiate the mapping could in this case have been to separate the different potential orientations with an added term to the input representing the orientation, e.g. 0 or 1 for 1D case, or orthogonal vectors for higher dimensions. Then one would have to add each permutation of the potential with corresponding marker term on the eigenvalues to the training set. By doing this the normal MSE error function could be used instead of a custom error function. However, drawbacks of this would be to label the inputs depending on which way they are skewed and add all permutations. When using the network for unknown potentials an arbitrary orientation markers must be added to the inputs. Resulting in that the potential shape would be computed, but it would not be possible to get information about the orientation from the results.

The first methods used in this section did not show good results since they did not account for the 2 to 1 mapping of the problem. Although, these approaches could work well with the use of either the mirror_MSE error function or something like a differentiation method just discussed. This is something that could be investigated further, but due to time constraints it will not be done in this project.

It was shown that one layer was enough for this problem when using many eigenvalues, but as seen in the results, the performance dropped when using fewer eigenvalues. The performance could be raised again for a fewer eigenvalue count by using a deeper network, but even the predictions were usually not good outside of the first minima. An explanation for this behaviour could be that the lowest 100 eigenvalues

only gives information about the bottom of the potential. Because of this, the value of the first and one-hundredth eigenvalue were plotted as well, since these were the largest and smallest inputs. This shows that the eigenvalues were localised in the same energy range as the parts of the potential where the prediction was very close to the actual potential.

5 Density function to potential

This section will treat the modelling of the density function to potential problem and the networks used to solve the problem. The performance of the different networks trained will also be shown. The section will then be concluded with a conclusion and summary of the problem.

5.1 Model & Data generation

The focus on the second part of this project was to investigate if it was possible to train a network that could compute the potential and particle number from a ground state density function. The model is simplified as it is a 1D system where particle interaction is neglected.

To test this problem, new data was required, which was made by using the potential files from the previous problem to get the eigenvalues and the eigenvectors. This time a normal non-sparse solver from Numpy was used (which in hindsight should have been used previously). The eigenvalues were saved in the same manner as before. However, the eigenvectors were not saved as it took too much memory to save them for all potentials. Instead, the density functions for electron count between 1 and 50 were calculated using equation (2) for every potential and then saved. Even then, the density data was too large, so the files were split in two, giving two sets with 8 000/1 000/1 000 instances of training/validation/test data.

The actual training and validation set were then constructed by a method where a number of different particle numbers could be chosen. It then extracted all the density functions for those numbers. The particle numbers were then appended to the corresponding potentials, after which the new potential vectors were put into a data set so that their positions matched the right density functions. This meant that the same potentials appeared several times, but they had to in order for the network to learn how both the particle number and potential affected the density function. Another consequence of creating the sets in this way was that the original training set of 8 000 instances was expanded to 8 000 times the chosen amount of particle numbers.

5.2 Single output layer

In the first approach, the network was constructed so that that it took the density vectors as input and had the potential as output. Since the particle number was added to the potential vector it had an extra entry. This gave the network structure like shown in Figure 16.

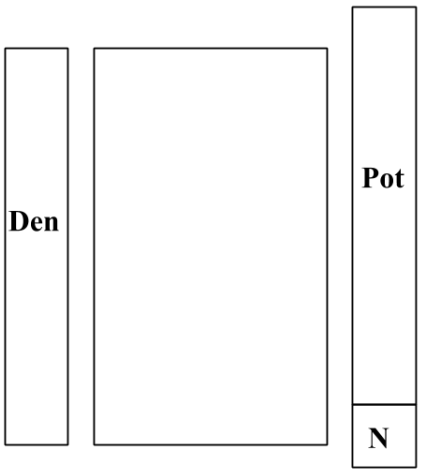
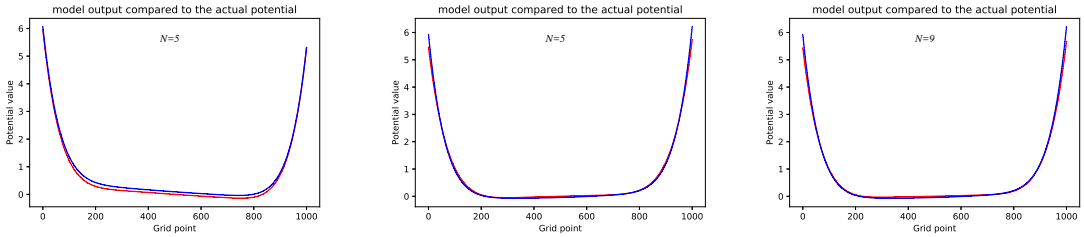


Figure 16: A schematic showing how the network structure when the particle number N was a part of the output potential vector.

The network itself used two hidden layers with 1 001 nodes and has the regular MSE as its loss function, since there is no symmetry in this problem. This network was trained on density functions corresponding to 2, 5, 7 and 14 particles. The outputs were compared to the target potentials, this is shown in Figure 17. The Figure shows that the network solved this problem well, since the output matched the actual potential almost completely.



(a) *(b)* *(c)*

Figure 17: The predicted potentials (red) compared to the target (blue) where the vector entry corresponding to the particle number has been omitted. N is the number of particles in the density function used as input.

Figure 17c shows the performance on a density function with a particle number it has not trained on. As seen in the Figure, when the network encountered particle numbers it had not encountered before, its ability to predict the potential was largely unchanged. The particle number predictions for these ”previously unseen” particle numbers were very close to the correct number, if it was between particle numbers that it has been trained on. Outside the known range of particle numbers, it delivered good guesses close to the range (1-2 numbers away), but was not performing well for values further away from the range. However, this still means that this network with only two hidden layers to some degree generalised to both particle number and potentials.

5.3 Two output layers

Just putting the particle number as an extra entry in the output vector could have a negative effect on the results, since it is not a part of the potential and that it could be on a different size scale. Having two output layers where one layer manages the potential and the other manages the particle count could lead to better performance. This network could be achieved by changing the structure to a structure similar to the

schematic shown in Figure 18. The same data matrices were used in these tests, but the particle number and potential parts of the matrices were separated when training.

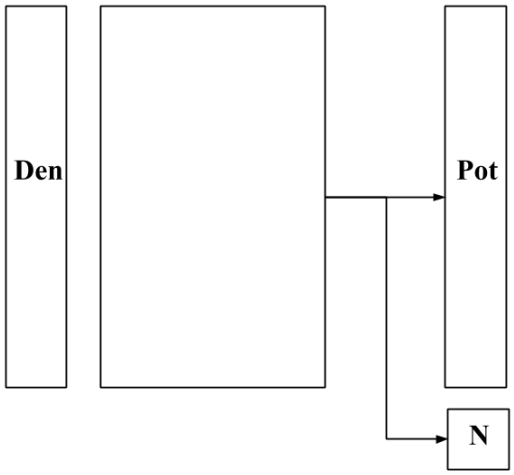


Figure 18: A schematic showing how the network structure with two output layers, where one layer puts out the potential and the other gives the particle number.

This was tested in two different ways: by having the particle number act as a kind of classification, or have single node output layer that computes the particle number with regression.

The particle number can be viewed as a classification where every particle number is a different class. Because of this, the network could instead be made to have two different output layers. The first being a layer that tries to compute the potential using linear regression. The second one being a classification layer using the softmax function to predict the particle number. In the classification case, the integer representing the particle number is converted to a vector of length equal to the number of classes (manually set). The vector only contains the number 1 on the position corresponding to the integer. This class to vector conversion is usually called ”one hot encoding” [11].

When treating the particle number as a class the network predicted the particle number correctly in every case where the particle number was one of the numbers it had seen in the training. However, it was no longer able to predict numbers it had not encountered during training. The network instead took a particle number it knew from the training data that was closest to the correct answer.

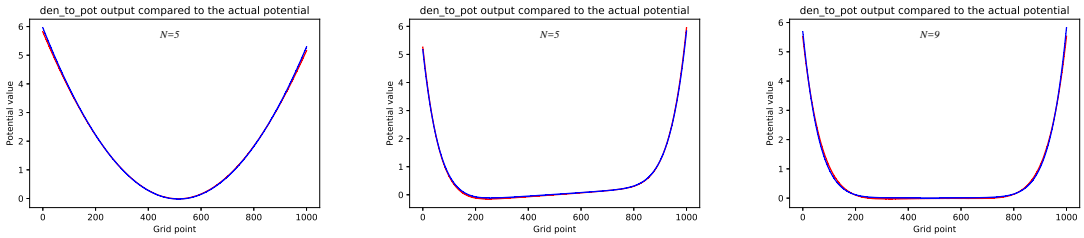
The issue with using the vector representation of the particle numbers is that all of the different vector representations are orthogonal to each other. This means that the distance between all classes are the same in the solution space, which is not the case in this problem as for example, 8 particles are closer to having 9 particles than having 3. For this reason, it is more suitable to use regression for the particle number.

Several different networks have been trained with this structure and these are displayed in Table 6. This shows the number of 1 001 node hidden layers, which data set was used and which particle numbers were used in the training. In some of the cases the data was modified before the networks were trained, which is notified by the shifted data.

Table 6: The different models used to solve the problem from density function to potential.

Model	Hidden-layers	Data-set	Particle numbers	Data-shift
den_to_pot	2	convex	2, 5, 7, 14	no
2layer	2	non-convex	2, 5, 7, 14	no
2layer_shift	2	non-convex	2, 5, 7, 14	yes
7layer_shift	7	non-convex	2, 5, 7, 14	yes
7layer_shift_low	7	non-convex	2, 4, 5, 7	yes

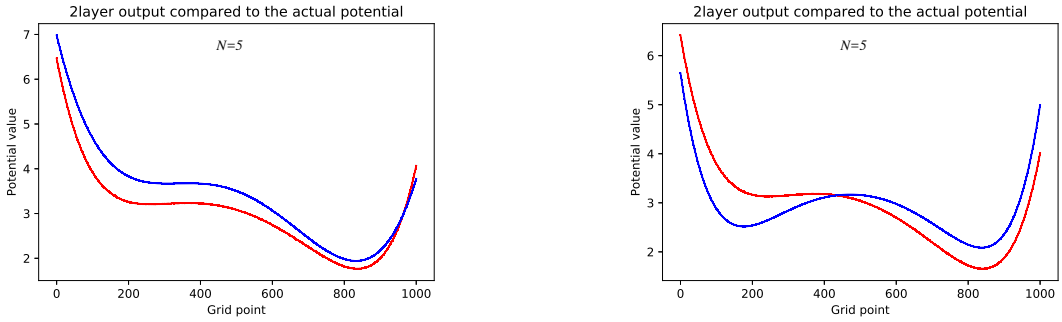
The first network that used regression to compute the particle number was called called den_to_pot. This network had 2 hidden layers with 1 001 nodes each, and was trained on densities with 2, 5, 7 and 14 particles of the convex data set. It had an MAE value on the particle number output that was around 0.02, which meant that it gave a number very close to the correct integer in every case. It even showed predictions very close to the correct integer in the case of 9 particles, which it was not trained on. The resulting output potentials are shown in Figure 19, which shows that the predictions were very close to the actual potential.



(a) *(b)* *(c)*

Figure 19: Plots comparing the target potentials (blue) with the output of the den_to_pot network, that took the density function and predicted the potential (red). N in the pictures are particle numbers of each input density.

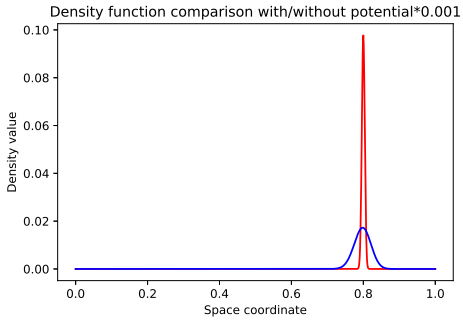
The non-convex set was then tested on the same network structure as before using the same particle numbers as well. However, this network, labelled 2layer, gave poor results as can be seen in Figure 20. In some cases, the network could not get the shape of the potential at all, while sometimes it had a hard time pinpointing the location and size of the part around the local minima.



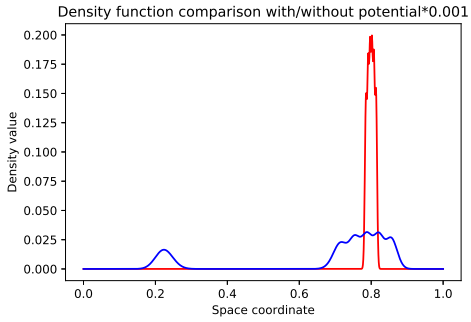
(a) *(b)*

Figure 20: The results of the model 2layer with 2 hidden layers that trained on the non-convex data set. The outputs are shown in red, while the blue lines are the actual potentials. These images are using 5 particle density functions as input.

To remedy this, a new data set was created by using the same method as before, but before putting the potentials into the solver they were multiplied with a factor 0.001. This changed the Hamiltonian making it a completely different problem to solve. The reason why was done was to broaden the wavefunctions and densities of the states, and to make it possible for lower states to appear close to the local minima instead of having the lowest 100 or so bunched at the global minima. This resultet in problems that would be easier for a neural network to solve. The broadening of the density function is illustrated in Figure 21, where the density function calculated with and without the factor 0.001 are compared for 1 and 6 particles.



(a)

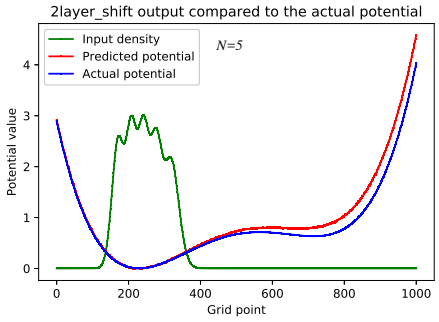


(b)

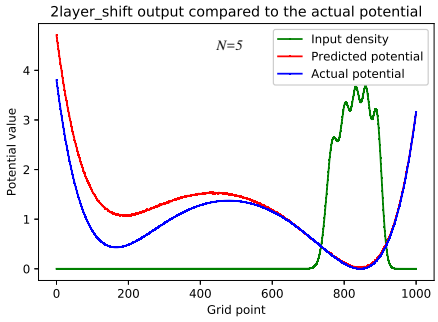
Figure 21: Comparison between the density function when the potential was scaled with a factor 0.001 (in blue) and without this factor (in red). This was done for 1 particle in the image to the left and for 6 particles in the image to the right.

One consequence was that factor 0.001 made the potential small when used as target data for the network, resulting in noisy output. To counter that, the factor was divided off before the network trained. However, there was one additional thing that became apparent when testing on the non-convex data, which was that the network sometimes shifted the whole potentials up or down. This meant that there was a freedom of choice where to put the potential, which led to worse performance. To solve this, all potentials in the data sets were shifted so that the bottom of the potentials were always equal to 0.

These changes improved the results considerably, which can be seen by comparing outputs of the new 2layer_shift network in Figure 22 with the previous results in Figure 20. This shows that the cause of the poor results was mostly due to the data and that the changes made led to the problem being solvable for the ANN. Still, the results are not perfect when using two hidden layers.



(a)



(b)

Figure 22: Plots comparing the actual potentials with the output of the 2layer_shift model that trained on density function of 2, 5, 7 and 14 particle systems. The inputs in the images are densities of 5 particles for two different potentials. The densities are rescaled and inserted in the images for illustrative purposes.

In an attempt to make better predictions, a network called 7layer_shift with 7 hidden layers each having 1 001 nodes were trained on particle numbers 2, 5, 7 and 14. The results for the same target potentials and particle numbers as in Figure 22 are shown in Figure 23. This shows that the increased depth of the network improved the results, which means that the added network complexity was needed to model the problem better. Here it is also shown that the networks generalised well for both particle numbers and potentials. This conclusion can be made from Figure 23, since all potentials in the images are from the validation set, and the inputs in the images to the right are from a density function with 9 particles. Some of the panels in Figure 23 show an output very close to the actual output, while for other panels the network was not able to get the whole potential shape right. It was usually the second minima that was not predicted well, but the predictions of the second minima improved when there were particles there.

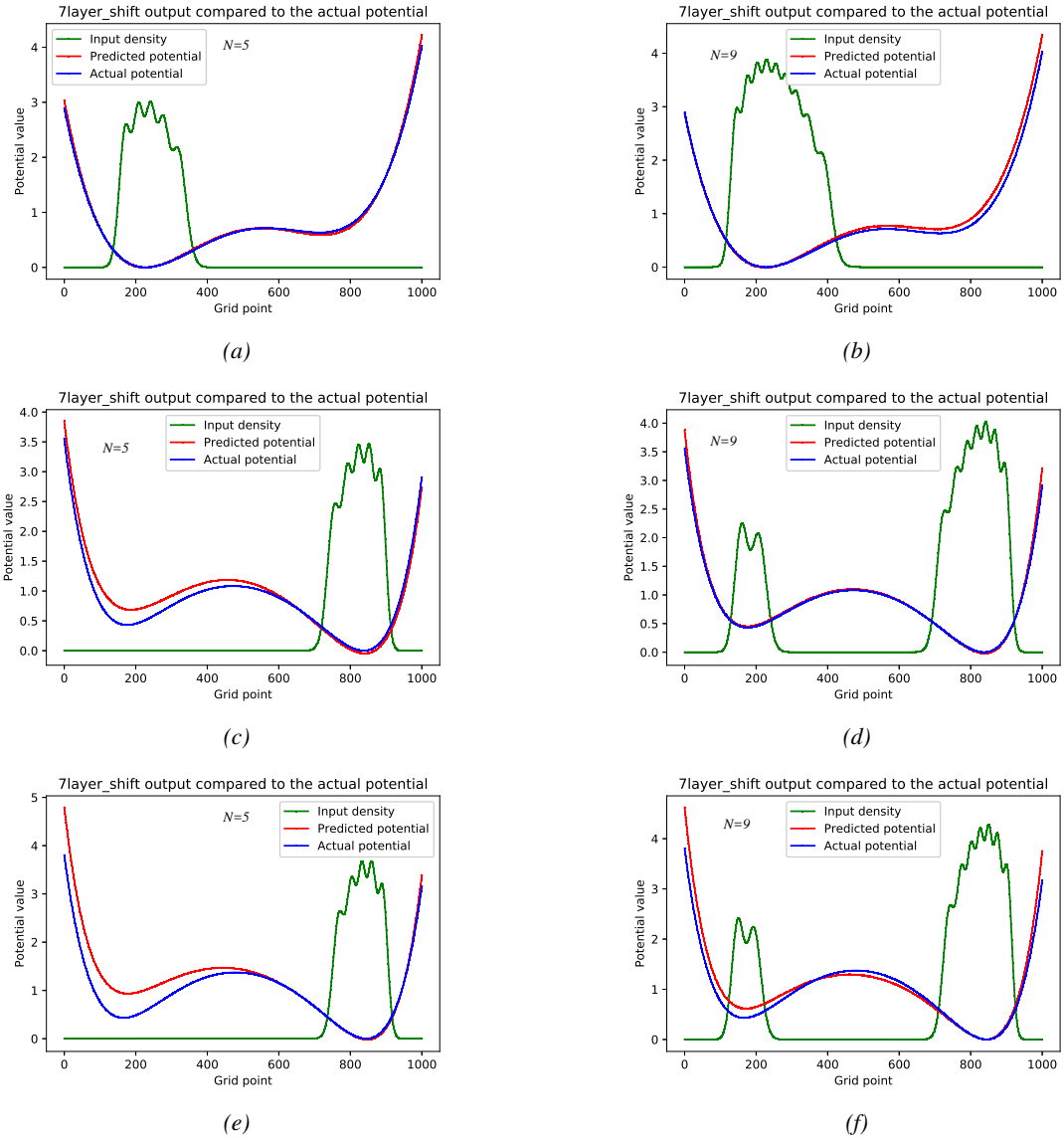


Figure 23: Plots comparing the actual potentials with output of the 7layer_shift. The particle numbers of the input densities are listed for each image as N. The input density is rescaled and shown in in the plot for illustrative purposes.

One interesting thing to see from the results were that the network handled the case with 9 particles well, but would a network that had not trained on numbers above 9 perform well on these density functions as well? This was tested by training a model called 7layer_shift_low that was exactly the same, except that it was trained on 2, 4, 5 and 7 particles instead. Figure 24 shows the results of this network for the same potentials as in Figure 23. The model that has not trained on the high particle number performed very similarly to the other network on the 5 particle densities, which was to be expected, since both of the networks have trained on 5 particles. However, the performance on 9 particles was slightly worse as this network is forced to extrapolate instead of interpolate.

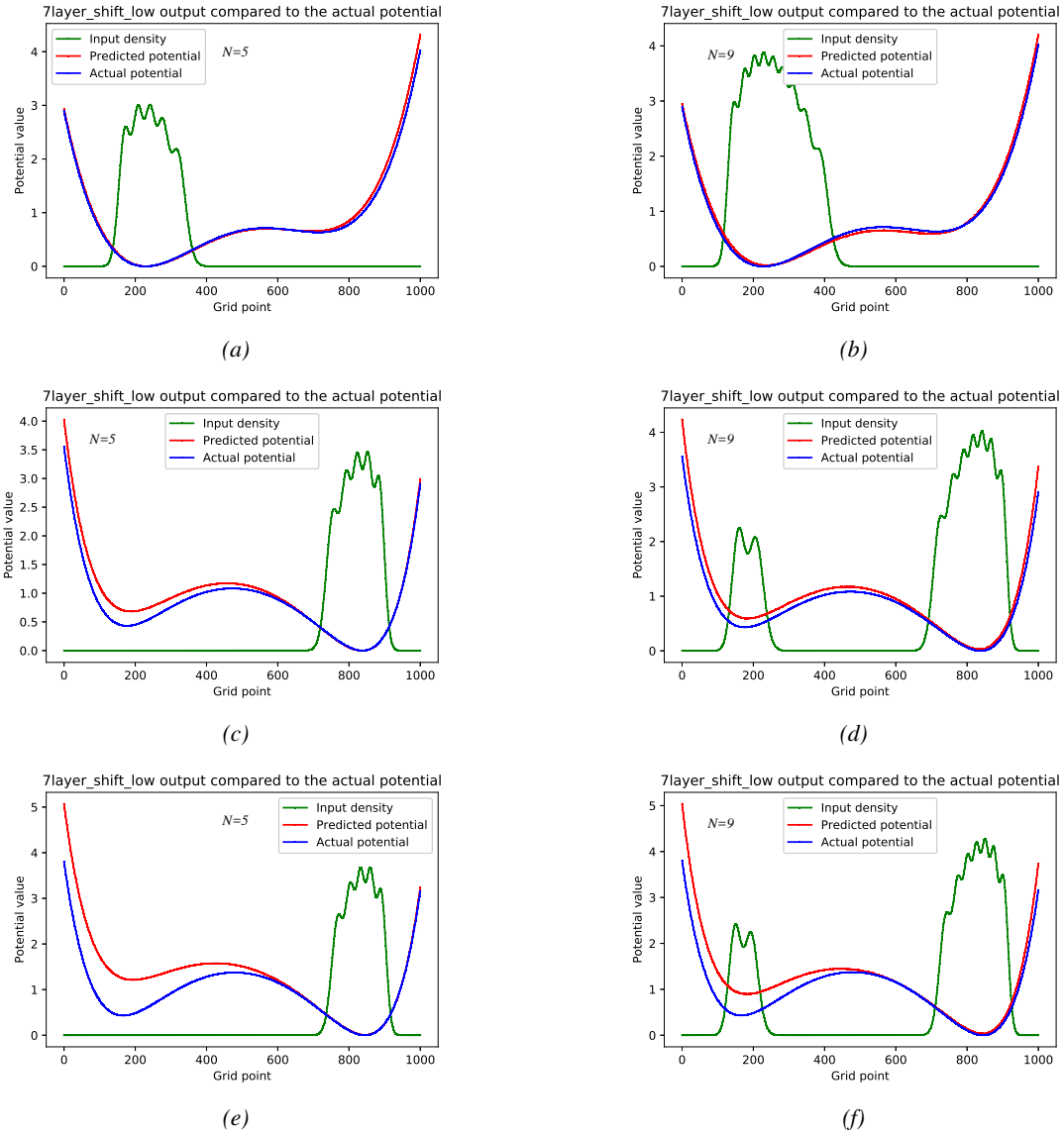
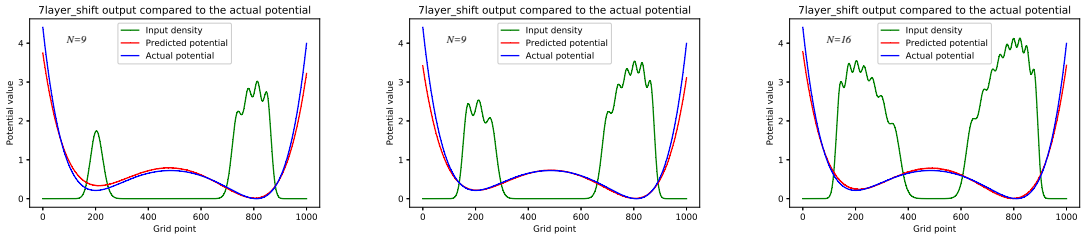


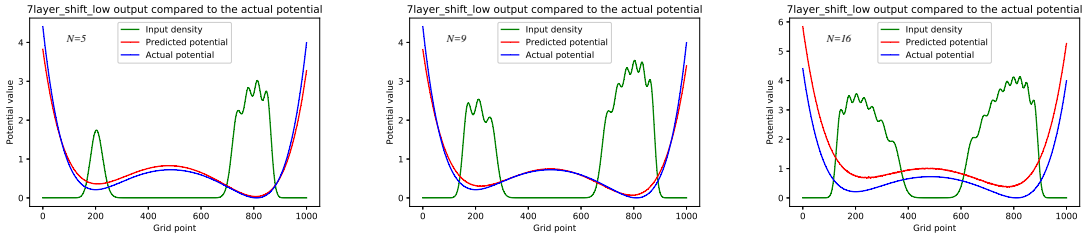
Figure 24: Plots comparing the actual potentials with output of the 7layer_shift_low. The particle numbers of the input densities are listed for each image as N. The input density is rescaled and shown in in the plot for illustrative purposes.

As a final test, both of the 7 layer networks were tested on a density function with 16 particles, which is higher than the particle numbers the networks were trained on (much higher for the 7layer_shift_low network). In Figure 25, the outputs from 7layer_shift are shown. Figure 25c shows that this network has no problem handling this case, while the other Figures show the performance for 5 and 9 particles in the same potentials as reference. The 7layer_shift_low network did not perform as well for 16 particles, which can be seen in Figure 26c. This was no surprise since the network has trained on at most 7 particles, and that is not close to 16. However, the shape of the output is pretty similar to the actual potential when looking past the fact that the output is shifted upwards. In this case, the other Figures are there as a reference to show the performance for 5 and 9 particles.



(a) *(b)* *(c)*

Figure 25: These images compares the network output of the 7_layer_shift network to a potential using the corresponding inputs densities with 5, 9, and 16 particles respectively.



(a) *(b)* *(c)*

Figure 26: These images compares the output of model 7layer_shift_low to the actual potential. The corresponding inputs densities with 5, 9, and 16 particles respectively are rescaled and plotted in the images.

5.4 Summary & Conclusion

The problem could be solved with very good precision for the convex data set. For the non-convex set, the potential predictions were not quite as good, but that was not as surprising since these problems are more difficult. In both cases, the predictions of the particle numbers worked very well, and networks with a separate output layer that used regression were good at predicting particle numbers they had not trained on. It was also shown that the networks were better at interpolating than extrapolating when it came to new particle numbers.

When looking at the results from the density to potential with the non-convex set, the results improved considerably when setting the bottom of the potential to be equal to 0. This is actually not that strange since the density function shows how the particles distribute in the potential; thus it holds information about the potential shape and not the energy where it is located. This is in agreement with the lemma in section 2.2 which states that the potential can be determined from the density function up to an additive constant.

When the potentials are not matching that well, it is usually the second minima that is the part where the fit is bad. It is also more likely to happen in the cases where there are no particles in the second minima. The reason for this is probably due to numerical rounding effects. All information about the potential should be in the density function, but most of the information about the second minima should be in the function tail when the density is very small in that area. This means that the potential in that area is decided by very small values and differences, which makes it susceptible to rounding errors of the numerical methods and noise in the networks.

This is most likely the reason for the poor performance before the potentials were rescaled, since the steeper and more localised potentials are affected more by these errors. For that reason, the wider density functions of the rescaled potentials are easier problems for the network to solve.

6 Potential to density & ground state energy

In this section the problem of computing the density function and ground state energy from the potential will be treated. Here, both the predictions of the density functions and the ground state energies will be shown. This section will then close with a conclusion and summary.

6.1 Problem model & Solution

The reverse process of computing the ground state density function from a potential and a particle was also investigated. This was done by more or less taking the problem in the previous section and switching the input and output. However, there was one difference: the ground state energy was also computed along the density function. Hence, the inputs were potential vectors and the particle numbers, while the outputs were density function vectors and ground state energy values. In this test, the non-convex data set were used with the scaling factor 0.001 to get the wider density function peaks, and the potentials were all shifted so that the global minima were equal to 0. The models were trained on the particle numbers 2, 5, 7 and 14.

When using two hidden layers the networks were able to predict the general shape of the density functions, but were unable to model the oscillating behaviour of the functions. When instead using 8 hidden layers with 1 001 nodes as model pot_den, the results were better in some cases, which can be seen in the Figure 27. The network manages the two peaks in Figure 27a and the smaller peaks in Figure 27b. However, as shown in Figures 27c and 27d it was not working that well in all cases, but even though the network did not capture the peaks, it managed to get the general shape of the density function.

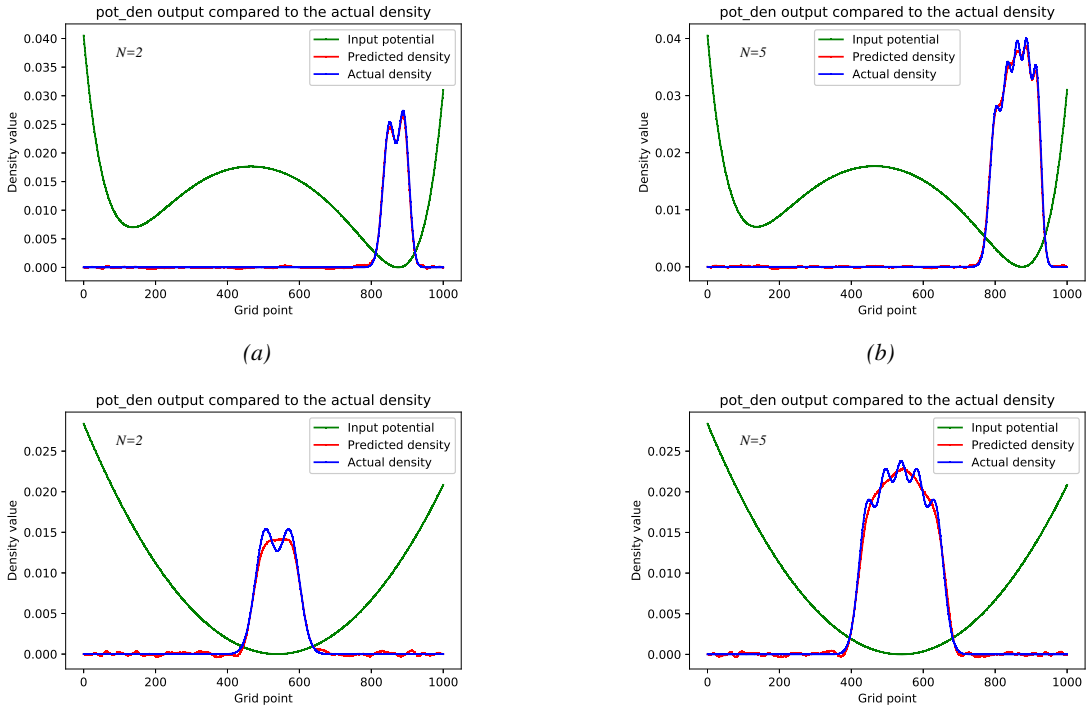
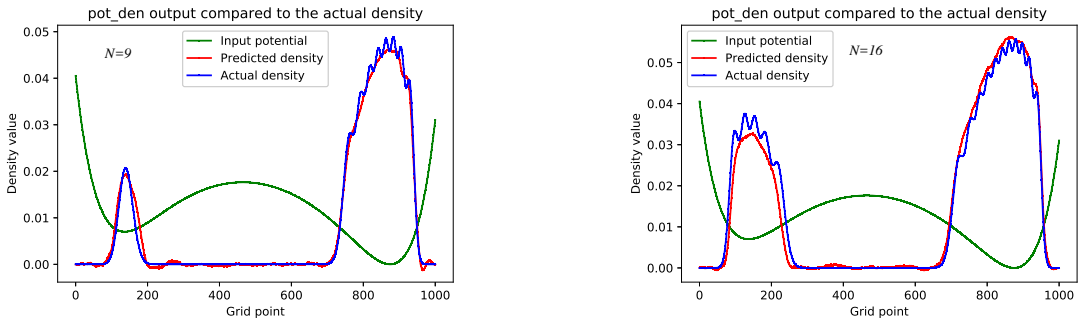


Figure 27: Comparisons between network output of pot_den and the actual density functions. This network has been trained on 2, 5, 7 and 14 particles. N is the particle number used as input. The input potentials are rescaled and included in the images for illustrative purposes.

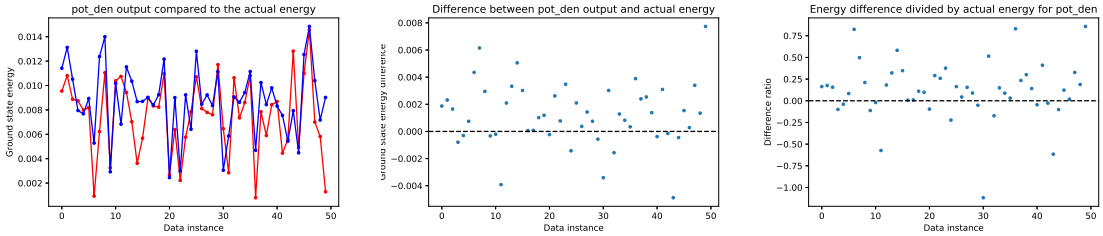
Just like in section 5, the network was tested on particle numbers it had not encountered. The results from that test is shown in Figure 28 that used the same potential as in Figures 27a and 27b, and particle numbers 9 and 16. These results show that the network generalised to particle numbers it has not trained on, since it captured the width and general shape of the density functions. However, it had problems capturing the small peaks.



(a) *(b)*

Figure 28: Comparisons between the pot_den output and the actual density functions when using the particle numbers 9 and 16 as inputs. The input potentials are rescaled and included in the images for illustrative purposes.

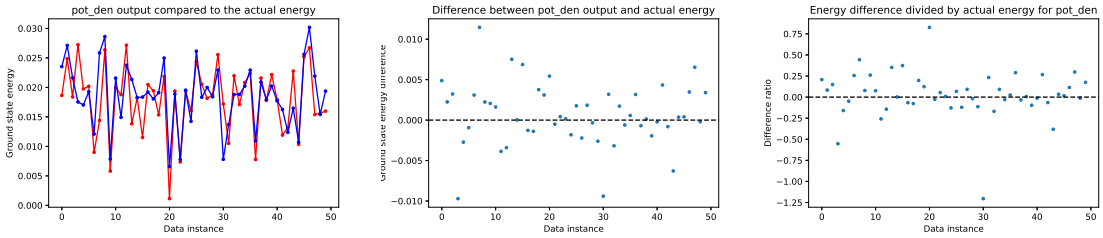
Figure 29 consists of three plots where different comparisons between the output and the actual ground state are shown. The comparisons were made using 50 different validation potentials when the particle number was 4. Figure 29a shows the ground state energy of both the output and the real data set value. By taking the difference between the values, the plot in Figure 29b is acquired. However, to get a better perspective of whether the differences are large or not they were divided by the actual values. This led to the plot in Figure 29c, which shows that a few predictions were very bad, but most of them were within $\pm 25\%$ of the target value. When first testing this problem by training models with fewer hidden layers, the validation MAE of the energy prediction was lower than what was observed for the 8 layer model. This means that increased accuracy for the density function prediction in this case came at the expense of the energy prediction, and hints that these results would most likely improve by changing the network structure.



(a) *(b)* *(c)*

Figure 29: Images to evaluate the ground state energy predictions of the network when using 5 particles as input. Every instance is a potential from the validation set.

This test was done in the same way, but with 9 particles as input instead, resulting in Figure 30. The plots in this case look similar to those in Figure 29. It is especially apparent in subfigure 30c, which shows that in this case the predictions fall within $\pm 25\%$ as well. This shows that once again that the network handles particle numbers it has not trained on.



(a) *(b)* *(c)*

Figure 30: Images to evaluate the ground state energy predictions of the network when using 9 particles as input. Every instance is a potential from the validation set.

6.2 Summary & Conclusions

The results from this section were similar to that in section 5 in that the networks worked well for the problems, and that they generalised well for particle numbers they had not trained on. However, the networks sometimes had a problem in capturing the oscillating behaviour of the density function. Although, this might not be important since the a completely accurate network would not be much help when a model neglecting particle interactions is inaccurate from the start.

The accuracy of the ground state energies were around $\pm 25\%$ for the 8 layer networks, which is not that good. This number could most likely be improved since the more shallow networks made when the code was tested showed a better performance for the energy prediction, but worse performance for the potential.

One simple thing that could improve the performance of the network would be to split the network earlier in the structure so that the output layers did not have the last few layers in common. By doing this, the different branches could be made so that they are not equally deep either.

This also shows that the network structure is even more important in problems where several different kinds of outputs are desired, which means that more emphasis on the network optimisation will be required.

7 General summary & Discussion

This section will be a summary and discussion of the project as whole.

7.1 General result analysis

It has been shown that the ANNs could solve all the problems investigated in this project quite well. This means that ANNs can solve inverse problems that currently does not have any real solution method, which means that this is a tool with lot of potential. It could mean that there are areas within quantum physics where ANNs could be used with high efficiency as well.

The results of the first two problems have both shown that degrees of freedom are detrimental to the performance of the ANNs. In the inverse spectral problem, the orientation led to many failed networks and it was not until this was accounted for, with the mirror_MSE error function, that it was possible to solve the problem. In the density function to potential problem, it was instead the additive constant that led to problems. This was solved by setting the global minima of all potentials to be at 0, which meant that the networks were taught that the bottom of the potential should always be at 0.

The fact that the mirror image of a potential has the same eigenvalues could be seen as prior information of the problem structure. This prior information was necessary to solve the inverse spectral problem, which could mean that the same applies to other inverse problems. This implies that there are specific things for every problem that must be taken care of in the model in order to get a good result. A consequence of this is that it makes every solution specific to the problem, which is similar to the conventional inverse solutions. However, when working with ANNs this is not a big issue, as the hyperparameters that solves a problem well is already specific to the task. Although, in some cases the problem-specific approach from one problem could be used on other similar problems with some modifications.

One downside with using ANNs is that a lot of data is required in order to get good performance from them. In many other applications that is not a problem, but depending on the problem there could be a shortage of examples when using it on real systems as some only exist in a finite number of variations. If that would prove to be a limitation then methods for solving that problem would have to be developed.

7.2 Flaws in the method

7.2.1 Simple models

The model is quite simple and there are flaws. One of these is the use of homogenous Dirichlet conditions even though the walls in the supposed potentials are not infinite. However, this should not affect the fact that the problems were solved. Also, previously when checking the eigenvalue spectrum of the harmonic oscillator it had the expected linear behaviour as shown in Figure 6 even though the harmonic oscillator is usually not solved with Dirichlet conditions. This indicates that the choice of boundary conditions does not have that big impact on the solutions.

One other thing that could be detrimental to the results were the potential values, since the potential heights were arbitrarily chosen. The reason for suspecting this is because the states were very localised in the non-convex potentials. No problems appeared during the inverse spectral problem, except that it could have an impact on the results when using a smaller subset of eigenvalues. The issue showed itself in full when using the density functions for the non-convex networks. This was shown because of the poor performance on the first set used and that the problem could be solved when using the set where the potential was scaled down by a factor 0.001.

There is also the issue with the density functions calculated in this work that the particle interactions have been neglected: by using data of real systems with particle interactions it might be possible for a network

to learn this as well. Since it was possible for it to learn the functional that relates the density function to the potential, the particle interactions should not be out of its reach.

7.2.2 Non-optimized networks

The random search was only used during some of the network constructions in section 4 to find a good autoencoder. After this, the random search was not used since it took a long time and it was better to try a few different networks in order to find one that was sufficiently good, just to show if a method worked. There was no need to find the most suitable network for each problem since the purpose of this study was not to optimise a network to solve the problems, but instead, show that there are networks able to solve them.

8 Conclusion & Future prospects

In this thesis project all the problems treated has been solved using ANNs, which could be seen as an indicator that there are more possibilities for this tool to find use in quantum physics. Even though the systems used have been quite simple, they have been solved by quite simple network architectures. The networks have shown good performance, and there are still room for improvement, which is another indicator that ANNs could be a powerful tool for several applications within physics in the future. However, if one input maps to several outputs this must be accounted for in the method, otherwise the networks will not be able to solve the problems well.

The next step for ANNs with applications in physics would be to train networks to solve these kinds of problems using real data. For some systems, it would be possible to model them in a similar way as done here, but for other problems one would have to do changes because of e.g. the problem being in 3D. It is a step that must be done in order to create new methods with actual practical value.

There are also several other types of problems to explore and see if ANNs could help. These could be problems where there currently are no methods, where current methods are slow, or to produce a first guess for an iterative method. The network does not even have to do the whole process, but instead be a part of methods where current well working methods are used as far as possible, with the networks doing the step currently not possible.

9 References

[1] Wenming Yang, Xuechen Zhang, Yapeng Tian, Wei Wang, Jing-Hao Xue, and Qingmin Liao. Deep learning for single image super-resolution: A brief review. *arXiv e-prints*, 2018.

[2] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *arXiv e-prints*, 2016.

[3] Gottfried Anger. *Inverse Problems in Differential Equations*. Akademie-Verlag, Berlin, 1990.

[4] Jonas Adler and Ozan Öktem. Solving ill-posed inverse problems using iterative deep neural networks. *Inverse Problems*, 2017.

[5] Tuncay Aktosun and Ricardo Weder. The borg-marchenko theorem with a continuous spectrum. *arXiv e-prints*, 2005.

[6] Jaime Keller and José Luis Gázquez. *Density Functioal Theory*. Springer-Verlag, Berlin, 1983.

[7] Zhang Yi, Nan Fu, and HuaJin Tang. Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix. *Computers & Mathematics with Applications*, 2004.

[8] Gunnar Ohlén. *Kvantvärldens Fenomen - Teori och Begrepp*. Studentlitteratur AB, Lund, 2005.

[9] Eberhard K. U. Gross and Reiner M. Dreizler. *Density Functional Theory*. Plenus Press, New York, 1995.

[10] Lennart Edsberg. *Introduction to Computation and Modeling for Differential Equations*. John Wiley & Sons, Hoboken, New Jersey, 2008.

[11] Simon Haykin. *Neural networks and learning machines*. Pearson Education, Upper Saddle River, New Jersey, 2009.

[12] Cort J. Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate Research*, 2005.

[13] Keras: The python deep learning library. https://keras.io/. Accessed 2019-04-24.

[14] scipy.sparse.linalg.eigs. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigs.html. Accessed 2019-04-24.