Università degli Studi di Padova

Artificial neural network Solution for the 1D Schrödinger Equation

Assignment for Machine Learning for Chemistry & Coding in Chemistry

Nikola Donov

2025 / 2026

Contents

1. Introduction

In this research project, we propose a method using an artificial neural network (ANN) to approximate a solution to the time-independent Schrödinger equation, applied to the classical problem of a one-dimensional particle in a box.

The Schrödinger equation is fundamental to quantum mechanics and its applications to chemical problems. While analytical solutions exist for simple systems like the particle in a box, solving this equation for complex potentials typically requires more complex numerical methods such as finite element methods (FEM) or finite difference schemes.

Recently, advances in machine learning have provided an opportunity for alternative methods of solving partial differential equations (PDEs), including the Schrödinger equation in various dimensions and systems of interest to chemists.

The primary aim of this project is to demonstrate that ANNs can accurately approximate solutions to the time-independent Schrödinger equation based on a curated dataset built with a numerical solver, and its accuracy compared to other statistical models and the numerical solver itself.

The project is organized into several sections. Section 2 provides an overview of modern machine learning-based approaches to chemical problems. Section 3 formulates the problem of the 1D particle in a box, the decisions behind the dataset choice and the choice of the model. Section 4 shows the results of the implementation and their implications for the overall success of the model. The final Section 5 contains concluding remarks and the directions of further fine-tuning.

2. State of Art

Machine learning approaches to solving differential equations, including the Schrödinger equation, have gained significant attention in recent years as an alternative or complementary way to numerical methods.

Deep neural networks can be trained to approximate **solutions to partial differential equations**, including 2D Schrödinger equations. (Mills, Spanner and Tamblyn, 2017) Similar approaches have even been extended for the solutions of many-electron systems, including He, H_2, Be, B, LiH, and a chain of 10 hydrogen atoms. (Han, Zhang and E, 2019)

Physics-informed neural networks (PINNs) have also emerged as a special type of neural network, trained to solve supervised learning tasks while maintaining the laws of physics related to the problem. (Raissi, Perdikaris and Karniadakis, 2019) This is accomplished by directly encoding the physical constraints into the loss function.

Simpler statistical models have also been proven effective for chemical and physical problems. **Random Forest (RF)** and **Kernel Ridge Regression (KRR)** have been successfully applied to problems in chemistry, environmental science and materials science. (Mehtab *et al.*, 2023) When working with a limited dataset, these methods offer advantages in training efficiency and interpretability.

The particle in a box model is a classic example of a quantum problem, which has some advantages that can be used to explore the possibilities of integrating machine learning with quantum chemistry, namely:

- **Analytical Solutions:** the problem can be solved analytically for the case $V(x) = 0$ and this can be used to validate the model.
- **Low Computational Cost:** it is possible to generate large training datasets without too big of a computational cost

For the infinite square well where $V(x) = 0$, the analytical expression for the energy eigenvalues is:

$$E_n = \frac{n^2 \pi^2 \hbar^2}{2mL^2},\ n = 1, 2, 3, .. \qquad (1)$$

With this in mind, we can create a neural network to approximate the solutions of the problem, and then compare it to a numerical solver that has been validated against the analytical expression. We can also test how the neural network will compare to a simpler statistical model, such as the Random Forest.

3. Methods

3.1. Problem Formulation

The time-independent Schrödinger equation in one dimension describes the behavior of a particle under the influence of a potential energy function $V(x)$:

$$-\frac{\hbar}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x) \tag{2}$$

where $\hbar$ is the reduced Planck constant, m is the particle mass, $\psi(x)$ is the wavefunction, $V(x)$ is the potential energy and E is the energy eigenvalue. We can simplify the equation by using atomic units, setting $\hbar = m = 1$, to obtain:

$$-\frac{1}{2}\frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x) \tag{3}$$

The boundary conditions of the particle in a box problem impose that the particle is confined to a one-dimensional box centered at the origin, from $x = -L$ to $x = +L$. The box is surrounded by infinite potential walls at $\psi(-L) = \psi(+L) = 0$. Because of these conditions, the probability of finding the particle outside the box is zero.

Inside the box, the potential energy is represented as a polynomial function with the form:

$$V(x) = c_0 + c_1 x + c_2 x^2 + \dots + c_n x^n \tag{4}$$

By varying the coefficients of the polynomial, we can explore different potential energy functions within the box. The special case $V(x) = 0$, with all the coefficients at 0 corresponds to the classic particle in a box problem, for which analytical solutions exist which we can use to validate the numerical solver.

In the numerical solution, we discretize the domain $[-L, +L]$ into a uniform grid of $N = 512$ uniformly spaced points, with the spacing given as:

$$\Delta x = \frac{2L}{N-1} \tag{5}$$

The second derivative at each grid point is approximated using a second-order central finite-difference formula:

$$\left.\frac{d^2\psi}{dx^2}\right|_i \approx \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{(\Delta x)^2} \tag{6}$$

We can substitute this approximation back into the simplified time-independent Schrödinger equation to get:

$$-\frac{1}{2(\Delta x)^2}(\psi_{i+1} - 2\psi_i + \psi_{i-1}) + V_i\psi_i = E\psi_i \tag{7}$$

where $V_i = V(x)_i$ is the potential evaluated at grid point x_i. By collecting all the grid points i, the equation can be transformed from a differential equation into a matrix eigenvalue problem, written as:

$$H\psi = E\psi \tag{8}$$

where H is the Hamiltonian matrix and ψ is the vector of wavefunction values on the grid.

To proceed with an implementation of an algorithm to solve this problem, we must define a set of instructions to read the potential input, and solve the eigenvalue problem using the equations.

3.2. Fortran Numerical Solver

The dataset used to train the model is generated using a numerical solver built in Fortran 95. All procedures in the Fortran code are contained within a module that contains the subroutines that handle the input, construction of potential and Hamiltonian matrix, and finally the computation of the eigenvalues.

First, the **`read_input`** subroutine reads the **box length L and polynomial coefficients** from the `coefficients.txt` text file. The number of coefficients is automatically determined by counting the number of lines in the file, and the coefficient array is allocated accordingly. The potential energy function is represented as the polynomial given by:

$$V(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \tag{9}$$

The evaluation of the polynomial is done by using **Horner's method** as part of the **`build_poly`** function, which expresses it as a **nested form** to minimize the number of operations. This nested form is given by the equation:

$$V(x) = c_0 + x(c_1 + x(c_2 + \ldots + x c_n)) \tag{10}$$

Once the polynomial has been evaluated, the potential $V(x)$ must be built as an array of potentials at points $V(x_i)$ using the **`build_potential`** subroutine.

The domain $[-L, +L]$ is discretized into N uniformly spaced grid points with spacing calculated as $dx = \frac{2L}{N-1}$. To enforce the boundary conditions of $\psi(-L) = \psi(+L) = 0$, the Hamiltonian matrix takes only $N - 2$ grid points.

The Hamiltonian matrix itself is generated with the **`build_hamiltonian`** subroutine, with the size $(N - 2) \times (N - 2)$. The diagonal and symmetric off-diagonal elements of the matrix are given as, respectively:

$$H(i,i) = \frac{1}{(\Delta x)^2} + V(i) \tag{11}$$

$$H(i, i+1) = H(i+1, i) = -\frac{1}{2(\Delta x)^2} \tag{12}$$

Finally, the eigenvalue problem itself is solved using the DYSEV routine from the LAPACK library, which can compute both eigenvalues and eigenvectors for real symmetric matrices. A subroutine, **`solve_eigenvalue`** allocates the workspace and calls DYSEV with parameters to compute the eigenvalues corresponding to the ground state and excited states of the system.

The result is written into a .csv file in append mode with the **`write_result`** subroutine, writing to each row the total box length $2L$, the first five polynomial coefficients and the first ten eigenvalues.

Double precision with 13 significant figures and exponent range up to 10^{300} were used in all calculations. The solver has been validated against the analytical solutions for infinite square well potential $V(x) = 0$.

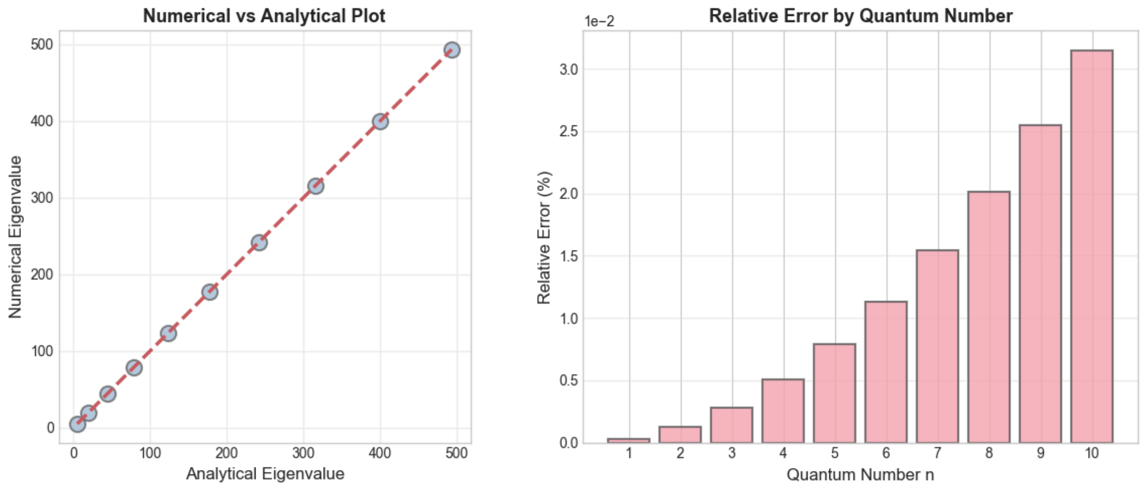

Figure 1: Validation of the numerical solver against the analytical solutions for $V(x) = 0$. The solver achieves an error below 0.001% for all eigenvalues.

3.3. Dataset Generation

Dataset generation is managed through a Python module that uses the **`subprocess` library** to run the Fortran executable automatically.

The user can input a set of parameters: box length L, coefficient range A, and number of samples N, and the function **`run_solver`** will generate N sets of polynomial potentials. The coefficients are sampled uniformly from the interval $[-A, +A]$ using NumPy's `linspace` function. A parameter t is generated as N evenly spaced values between 0 and L, and each coefficient gets assigned the value $a_i = -A + 2At$.

```
t = np.linspace(0.0, 1.0, N, endpoint = True)
a0 = -A + 2 * A * t
a1 = -A + 2 * A * t
a2 = -A + 2 * A * t
a3 = -A + 2 * A * t
a4 = -A + 2 * A * t
```

An obvious limitation of this approach is that for each sample, $c_0 = c_1 = c_2 = c_3 = c_4$, since t is the same parameter for all coefficients. This reduces the diversity of polynomial shapes and negatively affects the models' ability to generalize polynomial potentials. However, this approach gave better results than simply using random numbers in the $[-A, +A]$ range.

For each generated set of coefficients, the Python module writes the total box length L and coefficients to a text file and executes the Fortran solver:

```
for i in range(N):
    coeffs = [a0[i], a1[i], a2[i], a3[i], a4[i]]
    with open('coefficients.txt', 'w') as f:
        f.write(f"{L}\n")
        for coeff in coeffs:
            f.write(f"{coeff}\n")
    subprocess.run([str(exe_path)], cwd = str(dataset_dir), check = True)
```

The Fortran executable reads the coefficients, solves the eigenvalue problem and adds the results to the output .csv file. Each of the final N rows contains L, five polynomial coefficients and ten eigenvalues.

3.4. Dataset Curation

Several attempts were made to get a dataset that provides good performance. Initially, the Python wrapper varied both the box length L and the polynomial coefficients simultaneously, but this gave poor performance for both the neural network and the benchmark Random Forest.

The final dataset generator uses a fixed box length $L = 1.0$, which significantly improved performance for both models. This correction is justified since the energy calculated is related to the box length according to the relation $E \propto \frac{1}{L^2}$, which we know from the equation:

$$E_n = \frac{n^2 \pi^2 \hbar^2}{2mL^2}, \quad n = 1, 2, 3, .. \tag{13}$$

Varying L as a parameter introduced another source of variance in the data which complicated the learning task. By fixing it, the models only need to learn the relationship between the shape of the potential and the energy eigenvalues, rather than also learning the dependence on L which is already coded into the equations.

The default parameters for generating the dataset are $L = 1.0$, $A = 10.0$ and $N = 1024$. Lower values of A improve the R^2 scores, but lowers the number of potential shapes that the model explores. The number N is arbitrarily set to 1024 by default for faster testing, since higher values take longer to generate.

3.5. Model Choice

The neural network used is a **Multi-Layer Perceptron (MLP)**, which is a feed-forward artificial neural network where information goes from the input layer through hidden layers to an output layer. This particular MLP was implemented using the library **scikit-learn** and its `MLPRegressor` model.

For our problem, we have $n = 6$ input features: the box length L and 5 polynomial coefficients. The forward propagation from the input x to the first hidden layer $z_i^{[1]}$ is computed as:

$$z_i^{[1]} = f\left(b_i^{[0]} + \sum_{k=1}^{6} w_{k,i}^{[0]} x_k\right), \quad i = 1, 2, ..., N \tag{14}$$

where f is the activation function, $w_{k,i}^{[0]}$ are the weights connecting the input neuron k to the hidden neuron i, $b_i^{[0]}$ are the bias terms and N is the number of nodes in the hidden layer. This continues through the next layers until reaching the final output layer y:

$$y_j = b_j^{[L]} + \sum_k w_{k,j}^{[L]} z_k^{[L-1]}, \quad j = 1, 2, ..., 10 \tag{15}$$

where L is the total number of layers and the output provides predictions for the 10 energy eigenvalues.

The theoretical foundation for MLPs comes from the **universal approximation theorem**, which states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continous functions on compact subsets of $\mathbb{R}^n$ under mild assumptions. An appropriately configured MLP can learn the continous mapping between the potential functions we input and the energy eigenvalues.

The input layer contains 6 neurons, each corresponding to one of the input features. There are 3 hidden layers with the sizes (256, 128, 64), and the output layer contains 10 neurons corresponding to the first 10 eigenvalues. For a general neural network with scalar output and uniform hidden layer sizes, the number of parameters is given by $N^2L + N \times n$.

Our specific architecture with varying layer sizes contains 43,594 parameters. The decreasing width of the hidden layers creates an architecture that progressively extracts only the most important features from the input.

The choice of activation function $f(x)$ is also very important for the performance of the model. Two activation functions were tested:

- **Hyperbolic tangent**, where:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{16}$$

- **Rectified Linear Unit**, where:

$$\text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{17}$$

`ReLU` significantly outperformed `tanh` and the choice to continue with it was made very early. One assumption as to why is that when `tanh` handles very large values of $|x|$, the value goes to ± 1 and in these regions remains constant, providing little information on how the neural network should adjust the weights.

Training the MLP requires minimizing the mean square error loss function:

$$\mathcal{L} = \frac{1}{M} \sum_{j=1}^{M} \left(t_j - y_j\right)^2 \tag{18}$$

where t is the true eigenvalues and y is the network's predictions. The optimization is performed using the **Adam, or Adaptive Moment Estimation optimizer**, which implements the gradient descent:

$$\boldsymbol{w}^{(k+1)} \leftarrow \boldsymbol{w}^{(k)} - \alpha \nabla \mathcal{L} \tag{19}$$

where $\alpha = 0.001$ is the learning rate, and $\alpha \nabla \mathcal{L}$ is the gradient computed with backpropagation, which calculates the derivatives starting at the output layer and going backwards through the other layers.

The **L2 regularization** term adds a penalty proportional to the sum of the squared weights:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_i w_i^2 \tag{20}$$

where $\lambda = 0.01$ is the strength of the regularization, which helps reduce overfitting. Early stopping was also tested, but it did not provide any significant improvement to the performance.

Parameter	**Value**	**Description**
`solver`	`adam`	Adaptive Moment Estimation optimizer
`max_iter`	1024	Maximum training iterations
`learning_rate_init`	0.001	Initial learning rate α
`alpha`	0.01	L2 regularization parameter λ
`activation`	`relu`	Rectified Linear Unit
`random_state`	42	Fixed seed for reproducibility

Figure 2: MLP training hyperparameters

An additional statistical model, scikit-learn's `RandomForestRegressor` was used as a benchmark to compare the results of the neural network. Random Forest is an ensemble learning method that constructs multiple decision trees during training and combines their predictions through averaging. Each tree is built using two sources of randomness:

- **Bagging:** Each tree is trained on a different random subset of training data. Some data points can appear multiple times while others are omitted.
- **Feature sub-sampling:** At each node split, only a random subset of features is considered for determining the best split.

The Random Forest Prediction for the regression task is the average across all trees:

$$\hat{y} = \frac{1}{B} \sum_{b=1}^{B} T_b(\boldsymbol{x}) \tag{21}$$

where $T_{b(x)}$ is the prediction from the $b-$th tree of the total B trees.

Parameter	**Value**	**Description**
`n_estimators`	300	Number of trees in ensemble
`max_depth`	None	Trees grown to full depth
`random_state`	42	Fixed seed for reproducibility
`n_jobs`	1	Single-threaded execution

Figure 3: Random Forest hyperparameters

The `StandardScaler` function of scikit-learn was also used to scale the data during early runs, which strangely worsened the performance of the neural network. In the end, both the neural network and the benchmark Random Forest were trained without scaling the data.

Model	**Scaling**	**R^2**	**MSE**
neural network	`StandardScaler`	0.9997996	0.007515
neural network	None	0.9998474	0.005777

Figure 4: Effect of feature scaling on model performance. Omitting StandardScaler improved neural network performance by 23% in MSE.

To evaluate the models, two performance metrics were used: **R^2 score and Mean Squared Error**. The R^2 score is calculated according to the formula:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \overline{y})^2} \tag{22}$$

The Mean Squared Error (MSE) is calculated according to the formula:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{23}$$

where y_i is the real value of the observation, while $\hat{y}_i$ is the predicted value and n is the number of observations in the test set.

4. Results

4.1. Model Performance

The neural network and the benchmark were trained on a dataset of 1024 samples, generated with a fixed box length $L = 1.0$ and polynomial coefficient range $A = 10.0$. The dataset was split $80 : 20$ into training and test sets, and the model performance was evaluated on the test set using the R^2 score and mean squared error MSE. Models were trained to predict the first five energy eigenvalues E_1 to E_5.

Model	$\mathbf{R^2}$	**MSE**
Random Forest	0.9999959	0.000152
neural network	0.9998474	0.005777

Figure 5: Performance comparison between the neural network and the benchmark Random Forest.

The neural network achieved high predictive performance, with an R^2 score exceeding 0.9998 which indicates that the model explains more than 99.98% of the variance in the data.

Despite that, the Random Forest has a clear lead in terms of the MSE. Although for this particular case this, the MSE is low enough to not present an issue, it could be a problem when higher precision is necessary.

The worse performance from the neural network is surprising, considering the much higher amount of trainable parameters in the neural network.

4.2. Predicted vs. Numerical Eigenvalues

If we take a look at the relationship between the predicted eigenvalues and the ones produced by the numerical solver, we can see that there is a near-perfect performance in terms of the neural network's R^2 score.

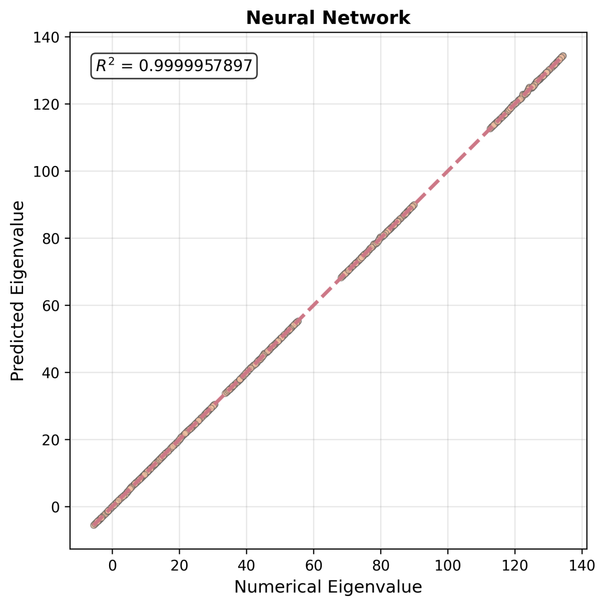

Figure 6: Predicted vs. Numerical Eigenvalues of the system

There also seems to be no significant difference between the prediction of the lower eigenvalues vs. the higher eigenvalues, suggesting that the model is fairly stable for energies up to the fourth excited state. We can further analyze this by taking a look at the error distribution by quantum number.

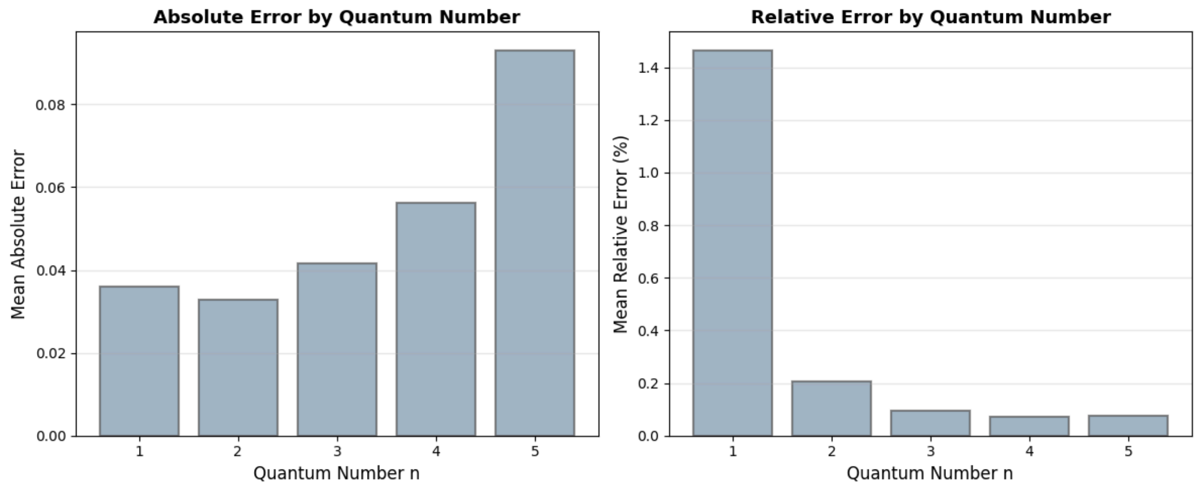

Figure 7: Error distribution per quantum number in the neural network Model.

The absolute error increases going from lower to higher energy states, which is easily explained since higher eigenvalues become much larger as the quantum number goes up. However, the decreasing relative error indicates that the model actually performs better on higher eigenvalues. The lower eigenvalues are smaller in magnitude, so the absolute errors have a larger impact on the relative error.

4.3. Parameter Sensitivity

Several different parameters were varied to see how the model accuracy changes in regards to the parameter changes, namely:

- **Number of data points (N)**. The model was tested for $N = 512; 1024; 2048; 4096$
- **Eigenvalue Predictions $\mathbf{E}_{\mathbf{max}}$**. The model was tested for $\mathrm{E}_{\mathrm{max}} = 5; 10$
- **Coefficient Range (A)**. The model was tested for $A = 5.0; 10.0; 20.0$

Examining the dependence on the number of data points N reveals the expected trend of the model improving when more data is available. Simply doubling the size of the dataset provides a small gain in R^2 metrics but a large 21.36% improvement in the MSE. In contrast, halving the number of data points also makes a minimal negative impact on the R^2, but creates a fairly negative impact on the MSE, increasing it by 35.84%. The largest improvement is observed when we quadruple the amount of data points in the dataset, which yields a 43.40% decrease in the MSE.

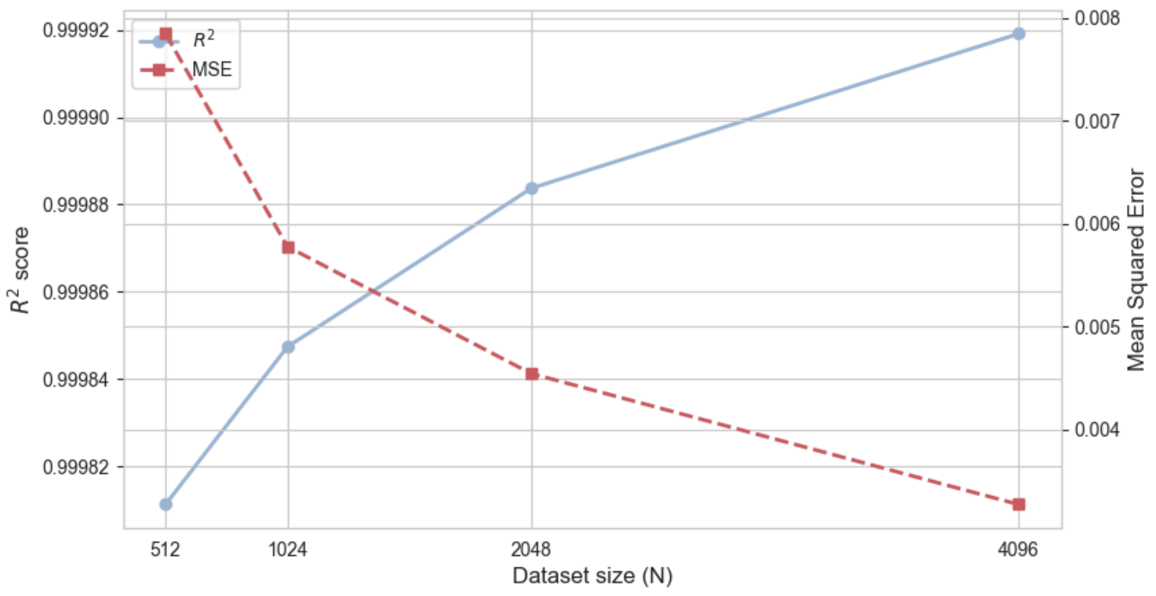

Figure 8: Dependence of the neural network's performance on the size of the dataset.

N	**R^2**	**MSE**
512	0.9998113	0.007847
1024	0.9998474	0.005777
2048	0.9998837	0.004543
4096	0.9999191	0.003270

Figure 9: Exact values of R^2 and MSE for each dataset size.

Limiting the number of eigenvalues predicted also improves the model, which is why only the first five eigenstates were considered in the final model. Extending the model to the prediction of the first ten eigenvalues worsens the R^2 by a factor of 2.5x, and increases the MSE by a huge margin of 1291.6%. It is clear that as we go to higher eigenvalues, the dependence is much less linear and harder for the model to learn.

Eigenvalues	**R^2**	**MSE**
5	0.9998474	0.005777
10	0.9978428	0.080409

Figure 10: neural network performance when predicting five versus ten eigenvalues.

Narrowing the coefficient range A to 5.0 dramatically reduces MSE by 42.4% while expanding it to 20.0 shows a small MSE increase of 9.93%. This trend is likely due to how the coefficient range affects the number of polynomial shapes that the model needs to learn from, so as the coefficients get larger, the variety of the shapes expands, but not too dramatically, since we train the model on uniform coefficient scaling.

Coefficient Range A	**R^2**	**MSE**
5	0.999646	0.003327
10	0.9998474	0.005777
20	0.9999576	0.006351

Figure 11: neural network performance as a function of polynomial coefficient range A.

4.4. Comparison with Random Forest Model

The finding that the neural network underperforms when compared to the Random Forest is strange. Neural networks traditionally outperform simpler statistical models on chemical prediction tasks with enough data and nonlinear mappings, which has been demonstrated on problems such as molecular property predictions. (Segler *et al.*, 2018)

Despite that, it does not necessarily mean that neural networks are always the best option. A study which benchmarked 179 classifiers across 121 datasets found that Random Forest models achieved top performance 94.1% of the time, outperforming other models specifically on tabular data with moderate sizes and nonlinear relationships. (Fernández-Delgado *et al.*, 2014)

We can argue that our dataset, with a size of ≤ 4096 points and tabular data where the relationships are not too complex falls within that category. Additionally, the fact that we only uniformly vary the data in the $[-A, +A]$ range rather than independently varying each coefficient significantly reduces the nonlinearity and dimensionality of the problem, which reduces the need for a neural network approach.

5. Conclusions

An artificial neural network was used to approximate solutions to the time-independent Schrödinger equation for the one-dimensional particle in a box problem. The project included a Python-based code to train the model on a set of potentials given in the form of polynomials, based on datasets generated from a Fortran numerical solver.

The neural network achieved high accuracy $\mathrm{R}^2 > 0.9998$, which shows that machine learning-based approaches can successfully be used to predict quantum mechanical eigenvalues with agreement to numerical solutions. The model showed consistent performance across all five predicted eigenvalues, with relative error actually decreasing for higher energy states.

However, the model was outperformed by the benchmark Random Forest model (MSE of 0.005777 vs. 0.000152), despite the assumption that the problem is best suited for a neural network approach. This suggests that the simple structure of the problem, using uniformly scaled coefficients, one dimension, a smaller dataset and strictly polynomial function potentials reduced the complexity to a level where simpler statistical models outperform complex ones. To fully make use of the potential of artificial neural networks for these types of problems, the dataset should be expanded to a larger size, with a much more varied set of polynomial functions, which would also make the models applicable to more real world problems.

References

Fernández-Delgado, M. *et al.* (2014) "Do we need hundreds of classifiers to solve real world classification problems?", 15(1), pp. 3133–3181.

Han, J., Zhang, L. and E, W. (2019) "Solving many-electron Schrödinger equation using deep neural networks," *Journal of Computational Physics*, 399, p. 108929.

Mehtab, V. *et al.* (2023) "Reduced Order Machine Learning Models for Accurate Prediction of CO2 Capture in Physical Solvents," *Environmental Science & Technology*, 57(46), pp. 18091–18103.

Mills, K., Spanner, M. and Tamblyn, I. (2017) "Deep learning and the Schrödinger equation," *Physical Review A*, 96, p. .

Raissi, M., Perdikaris, P. and Karniadakis, G. (2019) "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, 378, pp. 686–707.

Segler, M.H.S. *et al.* (2018) "Generating Focused Molecule Libraries for Drug Discovery with Recurrent Neural Networks," *ACS Central Science*, 4(1), pp. 120–131.