CALCULATION OF OPTICAL ABSORPTION SPECTRA USING A SUPERVISED NEURAL NETWORK

by

Conrad W. Rosenbrock

Submitted to Brigham Young University in partial fulfillment
of graduation requirements for University Honors

Department of Physics and Astronomy
Brigham Young University
April 2013

Advisor: Dr. Bret Hess

Honors Representative: Dr. Sean Warnick

ABSTRACT

CALCULATION OF OPTICAL ABSORPTION SPECTRA USING A SUPERVISED NEURAL NETWORK

Conrad W. Rosenbrock

Department of Physics and Astronomy

Bachelor of Science

Artificial neural networks have been effective in reducing computation time while achieving remarkable accuracy for a variety of difficult physics and materials science problems. Neural networks are trained iteratively by adjusting the size and shape of sums of non-linear functions by varying the function parameters to fit results for complex non-linear systems. For smaller structures, ab initio simulation methods can be used to determine absorption spectra under field perturbations. However, these methods are impractical for larger structures. Designing and training an artificial neural network with simulated data from density functional theory may allow time-dependent perturbation effects to be calculated more efficiently. I investigate the design considerations of neural network implementations for calculating perturbation-coupled electron oscillations in small molecules. The neural network structure presented in this thesis is eventually shown to be flawed because it mishandled the complex-valued inputs and outputs that it was trained on. As a result, important complex behavior, required for an accurate approximation of the time-evolution for the system, was ignored. Despite this, valid theory and design considerations are discussed in connection with a new complex-valued network structure that may be adequate to solve the problem.

ACKNOWLEDGEMENTS

Special thanks go to Dr. Hess for his time, patience and otherwise good mentoring. I would additionally like to thank the donor(s) of the Robert K. Thomas scholarship, the Copley family for the C. Bryant Copley scholarship, and the Physics Department for their research assistant support at various times. Lastly, I want to thank my wife, Helen, for helping maintain my health and the household during those busy times when I neglected everything else, and for listening intently to problems that sometimes I didn't even understand.

Contents

Title and Signatures Page i

Abstract iii

Acknowledgements v

Table of Contents vii

List of Figures, Tables and Algorithms ix

I Introduction **1**

1 Artificial Neural Networks **5**

1.1 Single Neuron Architecture 6

1.2 Multi-neuron Layer Structure 7

1.3 Neural Network Architecture 8

1.4 Function Approximation with Neural Networks 10

1.5 Neural Network Training Algorithm: Backpropagation 13

1.6 Network Training Terminology 16

II Theory and Methodology **20**

2 Theoretical Basis in Density Functional Theory **20**

2.1 Schrödinger's Equation for an N-body Problem 20

2.2 Density Functional Theory 20

2.3 Time-Dependent Density Functional Theory 22

2.4 Derivation of the Neural Network Solution 25

3 Neural Network Implementation **27**

3.1 Network Design Overview 27

3.2 Network Design Considerations 28

3.3 Programmatic Implementation in Matlab 31

III Results and Conclusions 39

4 Optical Absorption Spectra for Methane **39**

4.1 Training, Fine-Tuning and Performance 39

4.2 Final Network Structure 42

4.3 Predictive Time Evolution 44

4.4 Discussion 49

5 Conclusion **53**

IV Appendices 58

List of Figures

1 Neural Network as a Function Approximator 6

2 A single neuron with multi-dimensional input 7

3 Dimensionality of a single neuron with multi-dimensional input 7

4 A multi-neuron single layer network. 8

5 Dimensionality of a multi-neuron single layer network. 8

6 An example of a three-layer neural network. 9

7 Network structure for a simple two-layer network with sigmoid and linear transfer functions and scalar input and output. 10

8 Two-layer network output a^2 as x is varied from -2 to 2. 11

9 Network functional behavior as weight and bias parameters are varied. 12

10 Example of overfitting a data by configuring more free parameters than necessary. 17

11 Schematic diagram showing the proposed structure of the neural network. 30

12 Enlarged output for $\Re(c_m^1)$ from a single network validation test that was performed for methane. 35

13 Sample output from validation of the dipole expectation value for methane (see equation 15). 36

14 Sample output from validation of the absorption spectra by performing a Fourier transform of the dipole expectation value for methane (shown in figure 13). 37

15 Agreement between the network's prediction of the expectation value for the electric dipole moment and DFT simulated data. 41

16 Fourier transform of the expectation value for the electric dipole moment shown in figure 15. 41

17 Graphical depiction of final network design. Compare with figure 11. 44

18 Visual validation of the $j = 1$ states for 100 time steps predicted outside of the networks' training set. 45
19 Visual validation by point for a prediction horizon of one time step. . 47
20 Expectation value of the electric dipole moment for the first 40 time steps predicted by the network. 48
21 Fourier transform of the dipole expectation values shown in figure 20. 48
22 Network approximation of $\mathbf{c}_m^1$. 81
23 Network approximation of $\mathbf{c}_m^2$. 82
24 Network approximation of $\mathbf{c}_m^3$. 83
25 Network approximation of $\mathbf{c}_m^4$. 84

List of Tables

1 Numerical training results from backpropagation training of the network. 39
2 Experimental determination of neuron numbers per layer. 43

List of Algorithms

1 Generic, gradient-descent, backpropagation algorithm for a neural network. The algorithm determines how to adjust the network's free parameters with respect to each input so that the network's error is reduced in the shortest possible time. 15
2 Implementation of a linear propagator to calculate $c_m^j(t+\Delta)$ from $c_m^j(t)$, using the unitary propagator $U(t + \Delta t, t)$, for a system perturbed by a step-function electric field. 24

Part I

Introduction

Neural networks have already been shown to solve a variety of differential equations in physics and materials science [11, 12, 20]. Recently, in the area of condensed matter physics, Kahliullin et al. used a neural network to model phase changes in carbon, specifically graphite and diamond [21]. One of the great advantages of neural networks is their ability to approximate data from multiple sources, even when their theoretical roots differ. For example, Density functional theory (DFT, described in section 2) alone lacks the ability to correctly predict long range van der Waal forces. Using other empirical models (e.g. Tersoff, Brenner) improves the accuracy of modeled structures but still ignores certain properties of interest. Behler et al. were able to use a high-dimensional neural network [22] trained on both empirically-based and ab initio data to perform a molecular dynamics study of graphite-diamond coexistence.

Just last year, Morawietz et al. constructed a neural network representation of the potential energy surface for water [23]. The unique properties of water still represent a significant challenge and many of the existing potentials for water rely on DFT calculations. Their network's predictions were in excellent agreement with the reference DFT calculations and enable the molecular dynamics simulations, relying on the potential, to run many orders of magnitude faster. Neural network implementations in physics and materials science are becoming more common and it is an active research area.

In condensed matter physics, an area of active research involves calculating the behavior of systems in a perturbed or excited state. Methods available for calculating ground state systems have been well developed and are, in many cases, accurate. Perturbation theory developed out of a desire to solve difficult problems using exist-

ing solutions that are much easier to calculate. An example of an application from perturbation theory is the use of accurate, ground-state results to approximate complicated behaviors in a system's excited states. Modeling perturbations instead of complete, excited systems has allowed these complex behaviors to be approximated in some cases.

For example, in optical materials science, predicting the optical properties and behaviors of molecules and systems of molecules is an active topic. The development of optical sensors, photovoltaic cells and LED technologies depends on a knowledge of the optical response and other optical properties of the molecules and materials being used. At the most basic level, electric fields accelerate charges. When light of a certain frequency intercepts an atom or molecule, a single electron may be moved by the field. However, the movement of this single electron affects each of the other electrons in the atom via coulombic repulsion. Determining how the original light wave affects subsequent electron motion contributes to our understanding of the molecule's optical properties.

Due to the complicated nature of solving Schrödinger's equation with the inclusion of all the inter-particle potential energy terms, several approximation schemes have been developed that offer approximate solutions to the wave function perturbations. One such method of approximation is DFT and its time-dependent counterpart, time-dependent density functional theory (TDDFT). TDDFT calculations are computationally expensive and become impractical for systems with hundreds of atoms. Although, optical properties have already been successfully predicted for small molecules using TDDFT, large nanostructures still present a significant challenge.

Artificial neural networks have been effective in reducing computation time for a variety of difficult physics and materials science problems with only slight loss in accuracy. Neural network solutions trained with data from ground-state DFT that are subsequently adapted to solve the TDDFT problem may be possible supplementary

tools to reduce computation time. The network methodology presented in this thesis is eventually shown to be flawed. Despite this, the theoretical basis for the discussed methodology remains valid, and the network structure's flaws provide a backdrop for the discussion of an alternative, complex-valued network structure that may solve the problem.

1 Artificial Neural Networks

[1]Artificial neural networks are programming constructs that try to mimic the structure and communication pathways of the brain to solve difficult problems. They are primarily used to solve two types of problems:

1. Classification problems where a set of data points correspond to discrete groups.

2. Function-fitting problems where the data points can be represented by a continuous unknown function.

Networks contain input, hidden, and output layers of neurons. Each neuron has input flowing into it from the neurons connected to its input side, multiplied by the weight of the connection (synapse) between them. All of the neuron's weighted inputs are summed and then evaluated by the neurons transfer function. The transfer function is usually a non-linear function that truncates large values (positive or negative). This function's output becomes the input for the next level of neurons [1]. Complicated neural networks may include multiple layers of hidden neurons.

Since the network implementation used in this thesis uses multiple multi-layer networks to fit non-linear functions, I will discuss function-fitting networks exclusively. A function-fitting network mirrors the functionality of a mathematical function, as shown in figure 1. The network accepts the same n-dimensional input as the function and predicts an approximation to the function (of same dimensionality as the function's output). The network's prediction is compared to the true output of the function and an error function (typically a mean-squared error function) determines how closely the predicted output matches the true output. The network uses the calculated error to adjust the internal parameters of its neurons and refine the prediction.

[1]The order and elements of discussion presented in this section were adapted from [9].

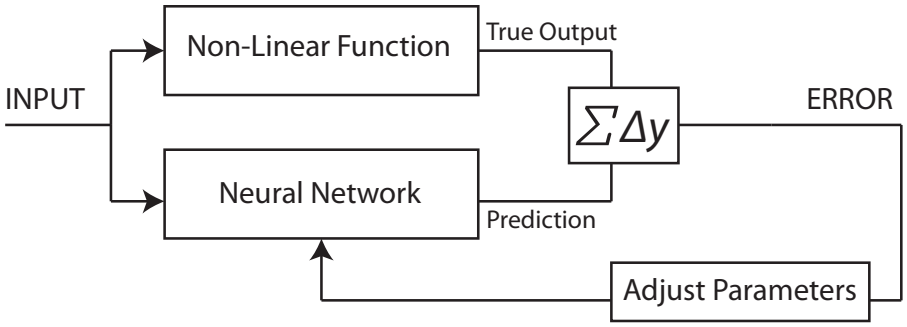

Figure 1: Neural Network as a Function Approximator

1.1 Single Neuron Architecture

An artificial neural network consists of layers of artificial neurons. A single neuron typically has the following components:

- **Inputs**: an R-dimensional vector of input values to operate on.
- **Adjustable Synapses (Weights)**: each input value is multiplied by a corresponding weight before being summed and evaluated by the transfer function.
- **Adjustable Bias:** a shifting value added separately to the weighted inputs before being evaluated by the transfer function.
- **Transfer Function:** usually a non-linear function that is evaluated for the sum of the weighted inputs.

These components are summarized in figure 2. The mathematical expression for the input to the transfer function and the result of the neuron's output are respectively:

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_R x_R + b \tag{1}$$

$$f(y) = f\left(\sum_{i=1}^{R} w_i x_i + b\right) = f(\vec{w} \cdot \vec{x} + b) \tag{2}$$

where the subscript indices $i = 1..R$ refer to the respective components of the input ($\vec{x}$) and weight ($\vec{w}$) vectors.

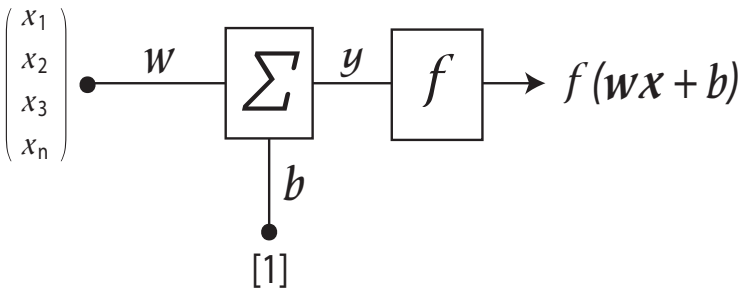

Figure 2: A single neuron with multi-dimensional input

The dimensionality of the inputs, bias and output for a multi-input neuron is summarized in figure 3.

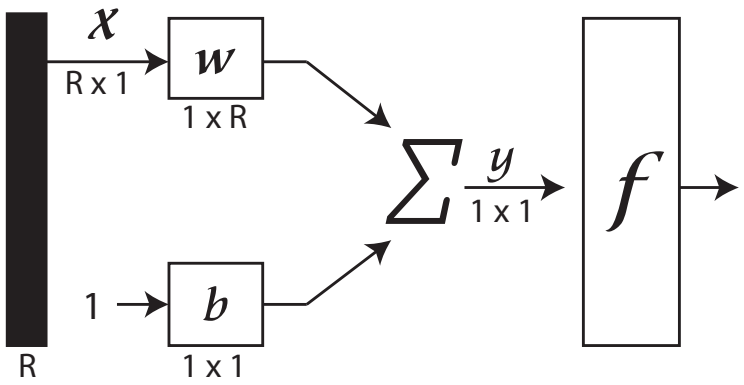

Figure 3: Dimensionality of a single neuron with multi-dimensional input

1.2 Multi-neuron Layer Structure

For complicated functions, a single neuron cannot approximate the function. In that case, multiple neurons can operate in parallel on the inputs; this introduces the possibility of approximating S-dimensional vector functions. When multiple neurons are used, each component of the input vector usually acts as an input to every neuron in the layer. Additionally, each input-neuron connection has an adjustable weight $w_{j,i}$ for the i^{th} input connecting to the j^{th} neuron in the layer. A graphical representation

of a typical multi-neuron, single layer network is shown in figure 4. The mathematical formula for such a network becomes: $\vec{a} = \mathbf{f}\left([\mathbf{W}].\vec{x} + \vec{b}\right)$ for the weight matrix $[\mathbf{W}]$ and vector function $\mathbf{f}$.

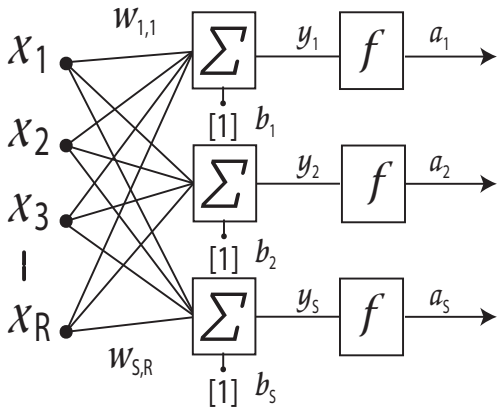

Figure 4: A multi-neuron single layer network.

The dimensionality of the inputs, bias and outputs for the one-layer network is summarized in figure 5.

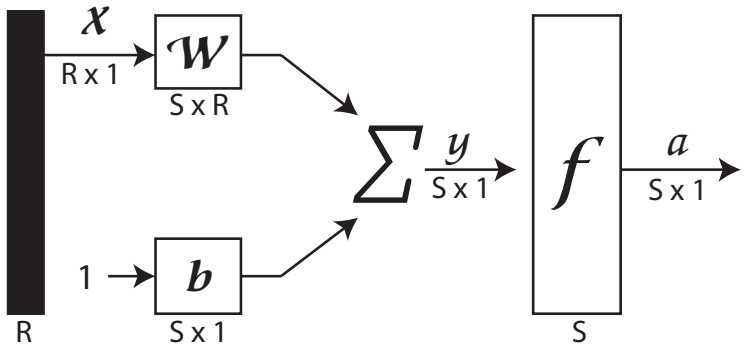

Figure 5: Dimensionality of a multi-neuron single layer network. Compare with the dimensionality of the single neuron network shown in figure 3.

1.3 Neural Network Architecture

Just as single neuron architecture may lack the free parameters to fit an arbitrary function, sometimes even a multi-neuron, single layer network lacks the complexity to produce an acceptable approximation. In this case, it may be necessary to have

multiple layers of neurons operating in parallel. Each layer has its own input $\vec{x}$, bias $\vec{b}$, weight matrix $[\mathbf{W}]$ and specific transfer function $\mathbf{f}$ that follow the general network formula $\vec{a} = \mathbf{f}\left([\mathbf{W}].\vec{x} + \vec{b}\right)$. The inputs to the layers do not necessarily need to be identical, and layers can have different numbers of neurons. I will use superscripts to differentiate between consecutive layers in the same network, e.g. $[\mathbf{W^1}]$ refers to the weight matrix in layer 1. As an example, for consecutive layers 1 and 2, the output vector $\vec{a}^1$ becomes the input vector $\vec{x}^2$. The final layer in the network, whose output vector $\vec{a}$ represents the output of the entire network, is called the output layer. All of the preceding layers in the network are referred to as hidden layers. Figure 6 shows the generic structure of a three-layer network.

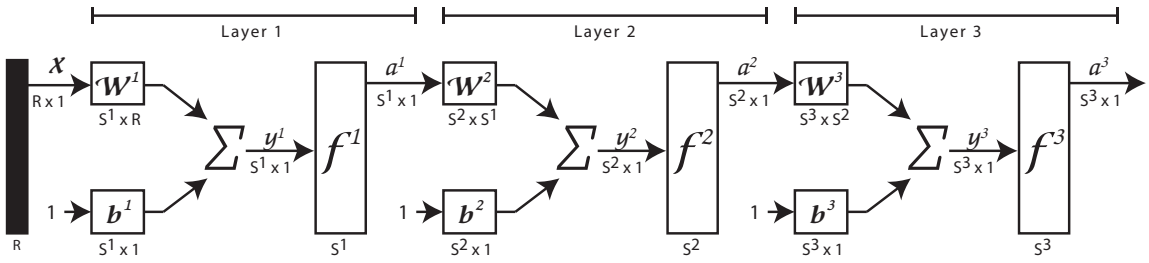

Figure 6: An example of a three-layer neural network.

In the case of the three-layer network displayed in figure 6, the mathematical formulas for each layer and the network as a whole are:

- Layer 1: $\vec{a}^1 = \mathbf{f}^1\left([\mathbf{W^1}].\vec{x} + \vec{b}^1\right)$
- Layer 2 (using $\vec{a}^1$ as input): $\vec{a}^2 = \mathbf{f}^2\left([\mathbf{W^2}].\vec{a}^1 + \vec{b}^2\right)$
- Layer 3 (using $\vec{a}^2$ as input): $\vec{a}^3 = \mathbf{f}^3\left([\mathbf{W^3}].\vec{a}^2 + \vec{b}^3\right)$

Combining these expressions into a single functional yields:

$$\vec{a} = \vec{a}^3 = \mathbf{f}^3\left([\mathbf{W^3}].\left[\mathbf{f}^2\left([\mathbf{W^2}].\left[\mathbf{f}^1\left([\mathbf{W^1}].\vec{x} + \vec{b}^1\right)\right] + \vec{b}^2\right)\right] + \vec{b}^3\right) \tag{3}$$

This illustrates how successive neural network layers are functions of the previous layers.

1.4 Function Approximation with Neural Networks

To illustrate the approximation features of neural networks we can consider a simple two layer network [9] with a structure as outlined in figure 7. The first layer utilizes the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ as its transfer function and has two neurons. The second layer has a linear transfer function $f(x) = x$ with a single neuron. We will present a single scalar value as input to the network.

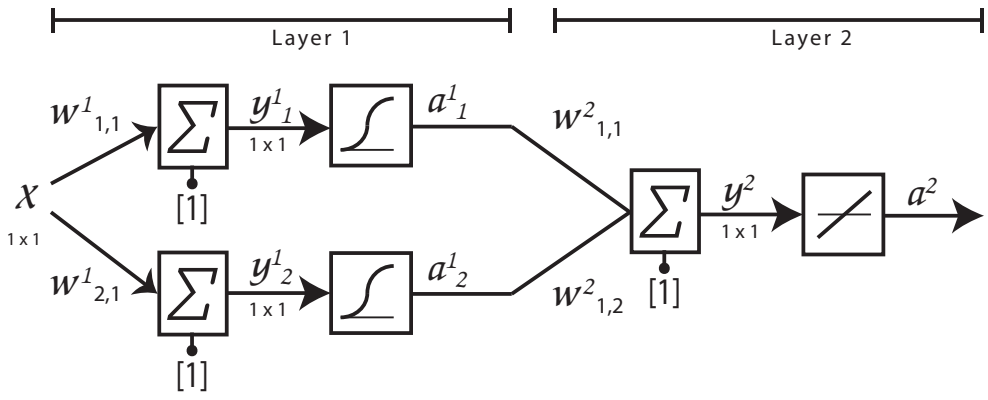

Figure 7: Network structure for a simple two-layer network with sigmoid and linear transfer functions and scalar input and output.

For someone unfamiliar with neural network theory and implementations, the abstractions depicted in figures 6 and 7 can seem daunting. The Matlab script below shows how the network in figure 7 would be implemented in code. It illustrates how simple neural network implementation can be and allows an investigation of the effects of parameter changes on functional behavior.

```
%Example Implementation of a two-layered neural network with sigmoid
%and linear transfer functions.

%Construct the transfer functions for each layer.
f1 = inline('1./(1+exp(-(w*x+b)))', 'w', 'x', 'b');
f2 = inline('w*x','w', 'x');

%Set initial values for the weights and biases for each layer.
```

```
w1 = [10 10];
b1 = [-10 10];
w2 = [1 1];
b2 = 0;

%Initialize some input values to plot
x = -2:0.01:2;

%Determine the outputs from the transfer functions of each neuron in
%the first layer.
a11=f1(w1(1),x,b1(1));
a12=f1(w1(2),x,b1(2));
%The output for the second layer combines the outputs from each
%neuron in the first layer before applying its transfer function.
a2=f2(w2(1),a11)+f2(w2(2),a12)+b2;
```

Plotting the functional representation of the network using the example bias and weight values specified in the script yields figure 8.

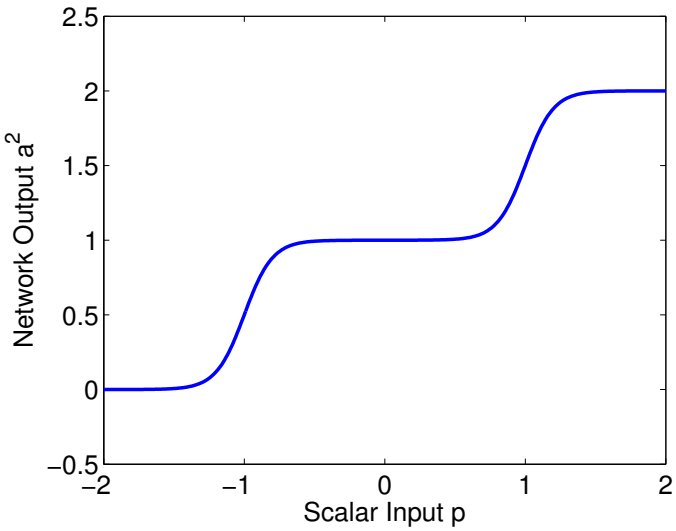

Figure 8: Two-layer network output a^2 as x is varied from -2 to 2.

The two steps in the function output are a result of summing the two sigmoid functions (neuron transfer functions) from layer 1. The position and steepness of

the curves can be adjusted by varying the weights and biases in the various layers. The effect of adjusting the $\vec{w}^1$ weights is quite intuitive since the weights are simply multiplicative factors in the argument of the sigmoid function and change the steepness of the step. Higher order layers in neural networks are far less intuitive because their transfer functions operate on functions and are therefore functionals. In figure 9 the weights and bias for the second layer are varied to show their effect on the step function generated in figure 8.

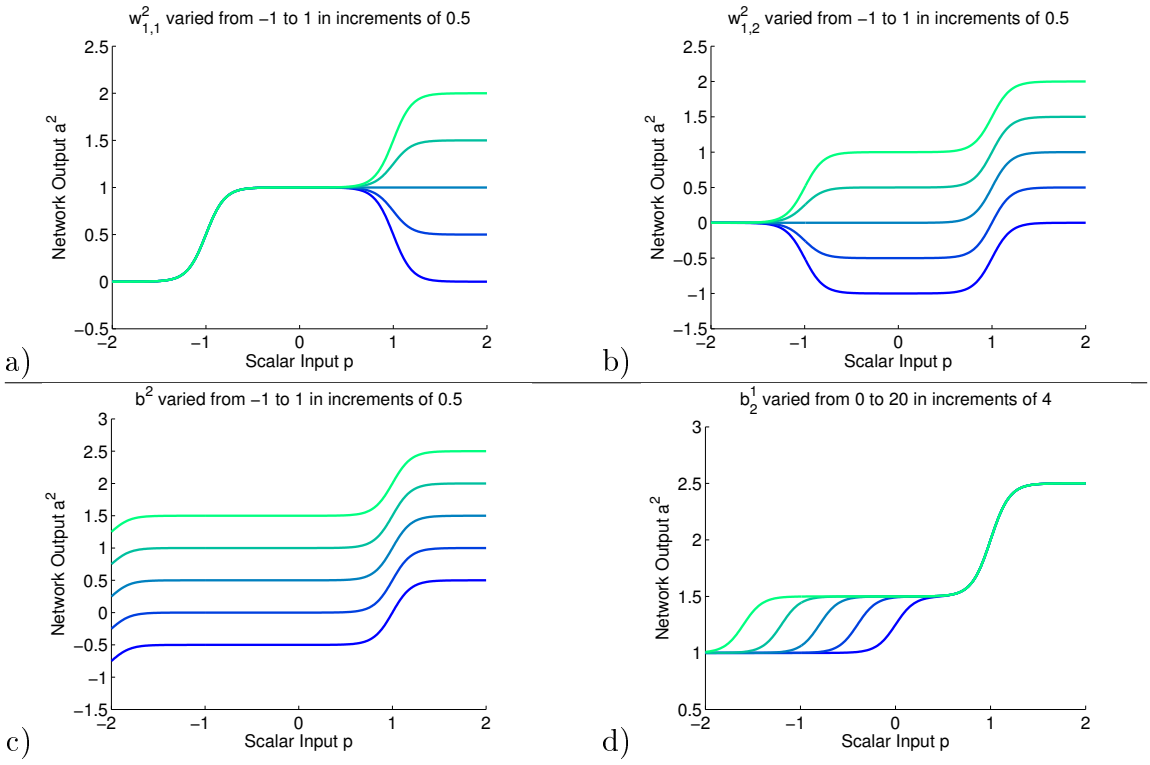

Figure 9: Network functional behavior as weight and bias parameters are varied: a) $-1 < w^2_{1,1} < 1$ in increments of 0.5; b) $-1 < w^2_{1,2} < 1$ in increments of 0.5; c) $-1 < b^2 < 1$ in increments of 0.5; d) $0 < b^1_2 < 20$ in increments of 4.

Qualitatively, we can describe the variations in the final function caused by the parameter changes as follows:

- $w^2_{1,1}$ and $w^2_{1,2}$ affect the maximum value and curvature of the sigmoid function that each multiplies. In the case of $w^2_{1,2}$ (figure 9b), careful examination shows that the sigmoid multiplied by $w^2_{1,1}$ is indeed unaffected by changes in $w^2_{1,2}$.

While this seems obvious in light of the mathematical construction of the network, the final output of the network is quite different than the case of $w^2_{1,1}$ (figure 9a), even though an identical transformation was applied to each.

- b^2 moves the entire network function up or down (figure 9c).
- b^1_1 (not shown) and b^1_2 (figure 9d) shift the sigmoid that each is added to either left or right. This is easily understood by considering a simple function $f(x)$ that is shifted by a value d, yielding $f(x \pm d)$. Compare this to the network output formula $\vec{a} = \mathbf{f}\left([\mathbf{W}].\vec{x} + \vec{b}\right)$.

It has been shown (see [2, 3]) that a neural network with a hidden layer of sigmoid functions and a linear output layer can approximate any function, making neural networks universal function approximators. For multivariable functions, the input $\vec{x}$ becomes a vector whose components are independently approximated. This makes the approximation of R-dimensional functions possible.

1.5 Neural Network Training Algorithm: Backpropagation

We have seen that neural networks can be universal function approximators. Networks require an algorithm to determine the correct values for the weights and biases on the neurons in the shortest amount of time. Typically, a training algorithm will involve iterative adjustment of neuron parameters by analyzing the error between the network's prediction and the true function output.

1.5.1 Generic Backpropagation Training Algorithm

The gradient operation from vector calculus forms the basis of the backpropagation training algorithm. Once we have determined a difference (delta) between the network approximation and the true value, we would like to change the parameters to reduce the error as quickly as possible. The gradient operator $\nabla =$

$(\frac{\partial}{\partial x_1}\hat{x}_1, \frac{\partial}{\partial x_2}\hat{x}_2, \frac{\partial}{\partial x_3}\hat{x}_3, \cdots, \frac{\partial}{\partial x_i}\hat{x}_i)$ finds the path of steepest descent for a given function and requires the function to be differentiable. While it can be shown (see for example [7]) that it is always possible to differentiate well-behaved functions for which neural networks are approximators, I will assume the functions we are approximating are differentiable. The algorithm then can be summarized as:

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k \nabla_k(\mathbf{f}) \tag{4}$$

Here the vector $\vec{x}_k$ represents the weights and biases for the current iteration step k. In order to calculate the new $\vec{x}_{k+1}$, we take the gradient of the network's functional representation with respect to each input to determine how to reduce the error. The variable α_k is a coefficient that affects the learning rate of the algorithm. Supervised algorithms will adjust α_k automatically during training depending on how quickly the error values are changing. For a multi-layer network the gradient descent algorithm is implemented in each layer in order:

Once each of the hidden layers has processed the error and adjusted the weights, the network will be ready to complete another training iteration. This involves computing the network's output using the newly adjusted parameters and then propagating the error back to complete the iteration. In practice, calculating the gradient becomes more complicated as additional layers are added to the network. An application of the chain rule allows the gradient to be calculated for the whole network with respect to each of the inputs.

$$\frac{\partial \mathbf{f}}{\partial w_{i,j}^m} = \frac{\partial \mathbf{f}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \tag{5}$$

$$\frac{\partial \mathbf{f}}{\partial b_i^m} = \frac{\partial \mathbf{f}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} \tag{6}$$

Algorithm 1 Generic, gradient-descent, backpropagation algorithm for a neural network. The algorithm determines how to adjust the network's free parameters with respect to each input so that the network's error is reduced in the shortest possible time.

1. At the Output Layer:
   (a) Determine the mean-squared error between the output layer's approximation and the true function output.
   (b) Compute the gradient with respect to each parameter in the layer that contributed to the output. This isolates neurons according to how much they each contribute to the total error.
   (c) At Each Neuron in the Output Layer:
      i. Determine which inputs to the neuron contributed most to the error at the neuron's output.
      ii. Adjust the weights for each of the inputs according to their calculated error from point $1.(c).i$.
      iii. Adjust the bias for the neuron.
   (d) Feed the error for each of the inputs back to a previous layer (if applicable).
2. At each Hidden Layer (repeated iteratively):
   (a) Using the mean-squared error passed back from the next layer,
   (b) Follow the steps as outlined in $1.(b-d)$.

Since we can write the input n_i^m to a transfer function as

$$n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m \tag{7}$$

the partial derivates for an arbitrary layer m can be computed as

$$\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1}, \frac{\partial n_i^m}{\partial b_i^m} = 1 \tag{8}$$

1.5.2 Algorithm Selection

Various network backpropagation training algorithms have been developed, including genetic and particle swarm optimization algorithms (see for example [4, 6, 5]). Each

algorithm comes with specific benefits and best-case applications. Typically there is a memory-performance tradeoff with algorithms. For example Levenberg-Marquardt uses standard gradient descent to determine parameter adjustments and is generally the fastest supervised algorithm. However, it also uses more memory than other, slower algorithms and is only good for networks with a few hundred adjustable parameters. For a benchmark comparison of training algorithms applied to real-world problems see [8].

1.6 Network Training Terminology

A typical function fitting problem using a supervised neural network requires a set of discrete points at which the function has been evaluated. For multi-variable, vector functions, the dimensionality of the input and output vectors need not be the same. The set of input and output vectors, sampled at various points in the function's range, form the dataset. Usually, the purpose of approximating the function with a network is to interpolate or extrapolate data points outside of the initial dataset. A neural network's ability to correctly predict function values for data points that it wasn't trained on is called generalizability. Ultimately, a good neural network achieves high generalizability in the shortest possible training time and using the smallest possible network size (layers and numbers of neurons). This interpolation/extrapolation introduces some additional considerations when designing and implementing the neural network:

- Overfitting: if too many neurons are specified to approximate a function, the possibility arises that the mean-squared error of the network's approximation might appear minimized while the actual approximation is severely deficient. Overfitting can usually be resolved by reducing the dimensionality of the hidden layers (see figure 10).

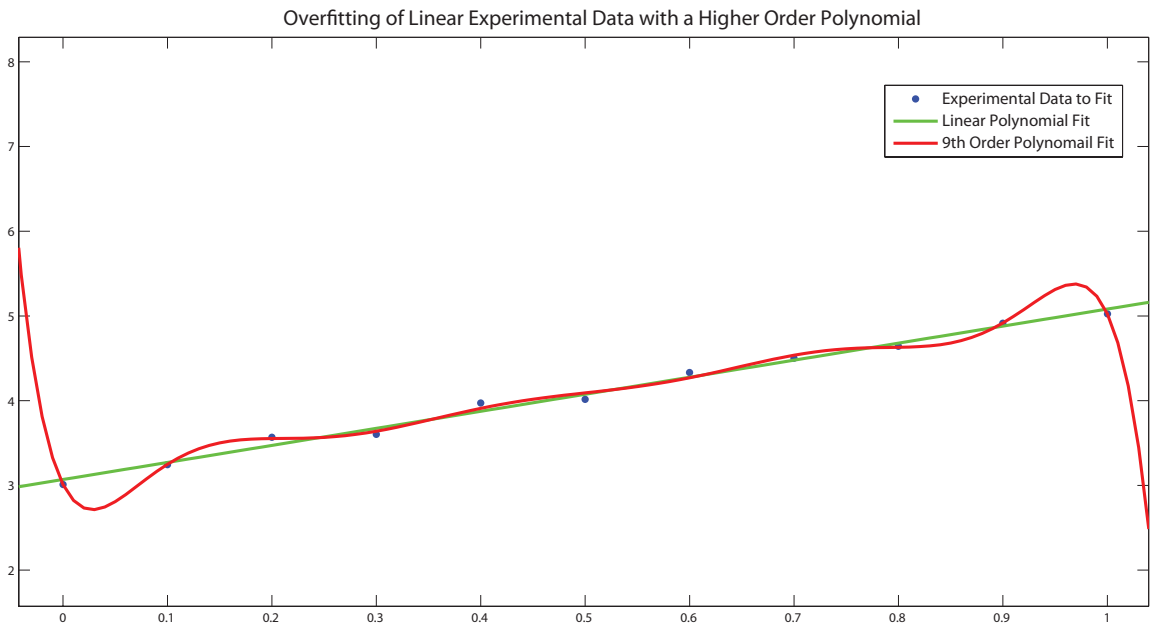

Figure 10: Example of overfitting a data by configuring more free parameters than necessary.

- Training Data: training data refers to the data points that are used during backpropagation training iterations to adjust the neuron parameters. Typically the training data points are selected from the initial dataset randomly. The training algorithm attempts to globally approximate these training points as closely as possible by adjusting parameters. Since the network is being adjusted to approximate the training data perfectly, the training data is not a good test candidate for determining the performance of the network. This introduces the necessity of a validation dataset.

- Validation Data: the initial dataset is split up randomly between training and validation data. Once the network has matched all the training points during a training cycle, the network is evaluated using the validation data. If the validation data is matched within specified error limits, the network is considered to be trained. If a pre-defined number of the validation points fail, the network returns to the training dataset to try and make further adjustments. If further

adjustments do not fix the problem, the network may need to be re-initialized.

- Network Initialization: before the network can begin training, an initial network approximation is required. If all the weights and biases have default values of zero, the network is likely going to have a large mean-squared error compared to the functional value, which may increase training time. Typically, the network's weights and biases are initialized to random values that are used to calculate the initial network prediction. This means that network training is not deterministic and could produce a different network each time a new training cycle is performed (new training includes a re-initialization of the network).

- Performance: network performance is typically measured using the summed, mean-square error across all the validation data points. Custom error functions can also be specified that may include specific knowledge about the problem being approximated. Such functions greatly improve the network's performance and training speed (for example see [11]).

- Validation Checks: when validation data is run through the network to determine the network's performance, validations checks may be enforced to try and prevent under/overtraining. A validation check sets a limit on how many validation data points match before training is stopped. For example, after initial training, perhaps only two validation points are within acceptable error limits, so the network returns to iterative adjustments of the parameters. A while later, perhaps 80% of the validation points are within the error limits; if the network is configured to accept 80% validation as acceptable, training could technically end. However, the network has not yet determined if another training iteration would match more of the validation points. The number of consecutive validation checks that need to pass before training will end is configurable. That way the network will continue optimizing to match more validation points. If the

value is set too high, the network might overtrain and lose generality.

- Minimum Gradient Limit: each time that parameters are adjusted to reduce the error, the network calculates its new prediction and a new error is calculated. The difference between last iteration's error and the current error is called the gradient value. If the gradient value doesn't change enough during an iteration, training will be terminated since it isn't making any progress.

Part II

Theory and Methodology

2 Theoretical Basis in Density Functional Theory

2.1 Schrödinger's Equation for an N-body Problem

For a problem involving N bodies, Schrödinger's equation takes the form of:

$$\hat{H}\Psi = \left[\hat{T} + \hat{V} + \hat{U}\right]\Psi = \left[\sum_{i}^{N} -\frac{\hbar^2}{2m_i}\nabla_i^2 + \sum_{i}^{N} V(\vec{r}_i) + \sum_{i<j}^{N} U(\vec{r}_i, \vec{r}_j)\right]\Psi = E\Psi \quad (9)$$

$V(\vec{r}_i)$ represents an external potential and $U(\vec{r}_i, \vec{r}_j)$ represents an interaction potential between each of the N bodies in the system. Analytically, this equation becomes impossible to solve exactly for $N > 2$. Various approximation schemes have bypassed this problem. One such approximation, density function theory (DFT) was developed by Hohenburg and Kohn and later enhanced by Kohn and Sham.

2.2 Density Functional Theory

[2]Density Function Theory reduces the dimensionality of the N-body problem by rephrasing the problem in terms of an average electronic density $n(\vec{r}) = \langle\rho(\vec{r})\rangle$. Kohn and Sham used the electronic density to recast Schrödinger's equation for non-interacting particles in terms of a new effective potential $v_{eff}(\vec{r}) = v(\vec{r}) - e\phi(\vec{r}) + v_{exc}(\vec{r})$ where ϕ is the electrostatic potential and $v_{exc}(\vec{r})$ is an "exchange-correlation" potential defined as $v_{exc}(\vec{r}) = \frac{\partial E_{exc}[n(\vec{r})]}{\partial n(\vec{r})}$. This new, effective, "Kohn-Sham potential" influences the now non-interacting electrons in a fictitious "Kohn-Sham system". The Kohn-

[2]For a thorough treatment of DFT, aimed at an undergraduate student level, consult [14].

Sham potential bears close resemblance to a mean-field theory [15]. Using this form of Schrödinger's equation and $n(\vec{r}) = \sum_{i=1}^{N} |\psi_i(\vec{r})|^2$, they formulated what are known as the Kohn-Sham equations [16]. The equations allow $n(\vec{r})$ to be calculated using a suitable estimate for the exchange correlation energy functional $E_{exc}[n(\vec{r})]$, a known potential $v(\vec{r})$, and the number of particles N. The Hamiltonian is also recast as a functional of the average density, $H[n(\vec{r})]$, which in turn allows the recalculation of a new set of $\{\psi_1(\vec{r}), \psi_2(\vec{r}), ..., \psi_i(\vec{r})\}$. A new averge electron density $n(\vec{r})$, and hence $H[n(\vec{r})]$, can then be calculated from the new set of ψ_i states. This iterative process facilitates a self-consistent calculation of the ground state energy for a system of N particles[3].

Once the N-body problem has been rephrased in terms of the electronic density $n(\vec{r})$, calculating the dynamics of systems becomes a problem of finding suitable approximations for the exchange-correlation energy functional $E_{exc}[n(\vec{r})]$. Kohn and Sham [16] suggested a "Local Density Approximation" (LDA) for calculating $E_{exc}[n(\vec{r})]$ that has produced results within 1% of experimental values for many problems, and that produces better results than Hartree-Fock [13]. Additional refinements that have had some success include local spin density (LSD) [17] (which treats $n(\vec{r})$ for localized spins separately as $n \uparrow (\vec{r})$ and $n \downarrow (\vec{r})$) and generalized gradient approximations (GGA) [18] which take into account the rate of change of local densities in a way that is consistent with the Kohn-Sham equations (direct gradient operations conflict with a step in the derivation of the equations making an approximation necessary).

[3]An alternate derivation of DFT theory using principles from thermodynamics is presented by Argaman ([13]). In this case the problem involves a Hohenburg-Kohn free energy functional $F_{HK}[n(r)]$ that is used to create a self-consistent set of equations. Argaman's derivation is recommended as a short and elegant alternative.

2.3 Time-Dependent Density Functional Theory

The evolution of the electronic system in time necessarily involves time-dependent electron densities $n(\vec{r}, t)$. Time dependence in the average electron density, $n(\vec{r}, t)$, extends the effective Kohn-Sham potential to also be time-dependent.

2.3.1 Computational Solution for TDDFT using Propagators

Because the Schrödinger equation is linear, it is possible to define a linear "evolution" operator $U(T, t)$ that transforms the initial vector for the n^{th} state $\psi^n(0)$ at time $t = 0$ into its value $\psi^n(T)$ at time $t = T$. Tsolakidis et al. published a TDDFT solution to an electric field perturbation problem that uses a linear, unitary propagator. Their solution is of interest because the data used to train the neural networks is generated using such an algorithm [25]. Additionally, the neural network solutions to the optical excitation problem will use the results from Refs. [24] and [25] to determine whether the network propagation was successful. Tsolakidis solution has been reproduced and adapted below.

TDDFT problems are solved by finding the time-dependent solution to the Kohn-Sham equation,

$$i\frac{\partial \Psi}{\partial t} = H\Psi \tag{10}$$

which allows the wave function to be calculated using a time-dependent Hamiltonian:

$$H = \frac{1}{2}\nabla^2 + V_{ext}(\vec{r}, t) + \int \frac{\rho(\vec{r'}, t)}{|\vec{r} - \vec{r'}|}dr' + V_{exc}[\rho](\vec{r}, t) \tag{11}$$

Notice that the Hamiltonian is a function of the time-dependent density matrix$\rho(\vec{r}, t)$. Equation 10 is evaluated for each time step in the simulation to obtain Ψ. The wave function Ψ can be written as a linear combination of orthogonal basis functions Φ_m, such that $\Psi^j = \Sigma_{m=1}^{\infty} c_m^j \Phi_m$. The coefficients c_m^j of the occupied wave functions can

then be used to construct a new density matrix

$$\rho(t) = \Sigma_i c_i(t) \tag{12}$$

The formal solution of equation 10 is:

$$\psi^n(T) = U(t,0)\psi^n(0) = \left\{ \sum_{k=0}^{\infty} \frac{(-i)^k}{k!} \int_0^t d\tau_1 \cdots \int_0^t d\tau_k H(\tau_1) \cdots H(\tau_k) \right\} \psi^n(0) \tag{13}$$

After splitting the interval $[0, T]$ into discrete steps Δt for computation, the problem becomes one of solving for $\psi^n(t + \Delta t) = U(t + \Delta t, t)\psi^n(0)$, which reduces to $\psi^n(t+\Delta t) = \exp\{-iTH\}\psi^n(0)$ in the case of a time-dependent Hamiltonian. Using $Z = \frac{\tau H_0}{2i\hbar}$, a Crank-Nicholson approximation to the propagator is

$$U(t + \Delta t, t) = \left[1 - Z + \frac{Z^2}{2} - \frac{Z^3}{6}\right]^{-1} \left[1 + Z + \frac{Z^2}{2} + \frac{Z^3}{6}\right] \tag{14}$$

The computational implementation takes the place of the propagator U in the equations. With prior knowledge of the ground-state Hamiltonian $H_0 = H(0)$ (which is available from solving for E_0 using standard DFT) and the initial wave function $\psi^n(0)$, it is possible to calculate $\psi^n(t + \Delta t)$. For systems with many particles, a ψ^n will be propagted for each of the states, regardless of whether it is occupied.

2.3.2 TDDFT Calculation of of the Optical Response under a Step-Function Electric Field

Using the method from the previous section, it is possible to calculate the evolution of the electronic system in time for an electric field perturbation of a given frequency. Although the method could be repeated many times for each frequency required, it is easier to solve the system with a step function field, which can be written as an infinite sum of different frequencies. Once the system has been solved, the optical

response for each frequency can be obtained via a Fourier Transform of the electronic dipole moment's expectation value, which is defined as

$$\langle z \rangle = \langle \mathbf{C^j} \,|\, z \,|\, \mathbf{C^j} \rangle \tag{15}$$

for the matrix $\mathbf{C^j}$ containing the $c_m^j(t)$ for all m and t. This sample algorithm demonstrates a possible implementation of the linear propagator to solve for the optical response of a system:

Algorithm 2 Implementation of a linear propagator to calculate $c_m^j(t+\Delta)$ from $c_m^j(t)$, using the unitary propagator $U(t+\Delta t,\, t)$, for a system perturbed by a step-function electric field.

---

1. Solve the system with a small perturbation $H^{(1)} = -\vec{E} \cdot \vec{r}$ using standard DFT (for the applied electric field before $t = 0$). This provides $\psi(0)$ for the remaining steps in the propagation algorithm.
2. For each time step in the propagation:
   (a) Solve equation 10 to obtain $\psi^n(t)$.
   (b) Calculate the new $\rho(t)$ from the c_m^j values of the overlapping orbitals (equation 12).
   (c) Compute a new $H[\rho(t)]$ (equation 11)
3. Repeat step 2 until $t = T$.
4. Calculate the dipole moment of the electron system $\langle z \rangle$ (equation 15).
5. Calculate $FT\left(\langle z \rangle\right)$ to obtain the optical response for all frequencies present in the step function.

Using such a method, Tsolakidis et al. were able to correctly model the optical absorption spectrum of C60 [24], a large molecule. While the solution works, calculating $H[\rho(t)]$ is computationally expensive because it involves a matrix inverse (equation 14). If a neural network is able to find a functional representation for U using knowledge of changes in $\rho(t)$, we could avoid the matrix inverse.

2.4 Derivation of the Neural Network Solution

Returning to neural network theory, we know that the output of a network layer k can be written as $y_i^k = \sum_{j=1}^{S^{k-1}} w_{i,j}^k a_j^{k-1} + b_i^k$ (Equation 7). It is now my intent to show that the problem at hand can be written in this form using a linear propagator $U(T,t)$, as discussed in the previous sub-section, and by making appropriate approximations. At that point a simple comparison will lead to the theoretical structure for the neural network.

Since each state can have a maximum of two electrons with equal energy, we only need to propagate $\frac{1}{2}N_{\text{electrons}}$ states in the basis of $\{\psi_m^{(0)}\}$, the eigenstates of the ground state Hamiltonian H_0. Each perturbed state $\psi_j^{(1)} = \Sigma_m c_m^j \psi_m^{(0)}$ will be a vector of length N_{orbitals} (the number of eigenstates of the H_0 matrix). In that $\{\psi_m^{(0)}\}$ basis, the c_m^j coefficients for $\psi_j^{(0)}$ will be close to 1. The c_m^j for $m \neq j$ will have smaller values that are only the result of the static, perturbing field.

For a small timestep τ, the $\mathbf{c}^j$ vector evolves according to the linear propagator $U(t) \approx U^{(0)}U^{(1)}$ for the system (see equation 13). As shown in the previous sections, $U^{(0)} \approx exp\left[\frac{\tau}{i\hbar}H(t)\right]$ for $H(t) = H^0[\rho_0(t)] + H^1[\delta\rho(t)] + H^1[E_{ext}(t)]$. Assuming H_0 to be diagonal and working from that basis:

$$c^j(t+\tau) = U(t)\mathbf{c}^j(t) \approx U^{(1)} \begin{bmatrix} \cdots \\ e^{\beta E_{m-1,0}} c_{m-1}^j(t) \\ e^{\beta E_{m,0}} c_m^j(t) \\ e^{\beta E_{m+1,0}} c_{m+1}^j(t) \\ \cdots \end{bmatrix} \text{ and } \beta = \frac{\tau}{i\hbar}$$

If we model only the $\psi_j^{(1)}$ states, each one can be coupled under the perturbing field $H_1(t)$ to all the states $\psi_m^{(0)}$. As mentioned above, the c_m^j of $\psi_m^{(0)}$ for $j \neq m$ will have small values. Coupling between these states is second order in E and will be

ignored. The perturbation H^1 to the Hamitonian is linear in $\vec{E}$. For linear response, then, there will be two different behaviors:

1. $\psi_j^{(1)}$ couples to all $\psi_m^{(0)}$.

$$c_j^j(t+\tau) \approx \sum_m U_{jm}(t)\, e^{\beta E_{m,0}} c_m^j(t) \tag{16}$$

2. $\psi_m^{(0)}$ with values of c_m^j close to 1 couple only to $\psi_j^{(1)}$ but not to each other.

$$c_m^j(t+\tau) \approx e^{\beta E_{m,0}} c_m^j(t) + U_{mj}(t)\, e^{\beta E_{j,0}} c_j^j(t) \tag{17}$$

Using the notation of equation 17, the tiny, second order perturbation in E that is being ignored is $U_{mn} \to 0$ for $n \neq j$. In the limit of zero perturbation, U becomes the identity matrix.

Comparing equations (16) and (7) for the neural network's output shows that the problem we are solving can be written in neural network form. The operator U_{jm} is a functional of all c_m^j (represented by the matrix $\mathbf{C^j}$). If all the c_m^j are presented to the network as inputs, it is possible that the functional nature of the neural network may approximate the linear operator U_{jm}.

$$y_i^k = \sum_{j=1}^{S^{k-1}} w_{i,j}^k \mathbf{f}^2 \left([\mathbf{W^2}]. \left[\mathbf{f}^1 \left([\mathbf{W^1}].\vec{c} + \vec{b}^1 \right) \right] + \vec{b}^2 \right)_j^{k-1} + b_i^k \tag{18}$$

$$c_j^j(t+\tau) \approx \sum_m U_{jm}(\mathbf{C^j})\, e^{\beta E_{m,0}} c_m^j(t) \tag{19}$$

3 Neural Network Implementation

3.1 Network Design Overview

Because the $U_{jm} \leftrightarrow U^*_{mj}$ operator functionals feature in both cases, it should be possible to combine eqs. (16) and (17) into a single network that calculates diagonal and off-diagonal values of U at the same time. Taking the dimensionality of the network inputs into account, we have to operate under certain contraints:

- Neural networks can only accept 2D matrices as inputs/outputs. The columns are reserved for the $c^j_m(t)$ values at different time steps. The rows will be occupied by each of the $\{c^j_m\}$ for all j and m.

- Since the network must train with multiple time steps, the dimensionality of our input space is actually $N_{\text{prop}} \times N_{\text{basis}} \times N_{\text{timesteps}}$, where N_{prop} is the number of states in $\{\psi^{(1)}_j\}$ with c^j_j close to 1, N_{basis} is the number of eigenstates in $\{\psi^{(0)}_m\}$, and $N_{\text{timesteps}}$ is the number of time steps in the simulated DFT data that will be used to train the network.

- We have to reduce the dimensionality of the system by creating multiple neural networks, one for each propagated state j. Each network will approximate the evolution of a single $\mathbf{c}^j = \{c^j_m\}$ using all the $\mathbf{c}^j$ values as input (since the evolution of any single state depends on all of the states). This means that we will need N_{prop} neural networks each with input dimensionality $N_{\text{prop}} \times N_{\text{basis}}$ rows and $N_{\text{timesteps}}$ columns. Although we could theoretically approximate all of our $\mathbf{c}^j$ vectors with a single network that has output of dimensionality $N_{\text{prop}} \times N_{\text{basis}}$ rows and $N_{\text{timesteps}} - 1$ columns, in practice this results in an overdetermined network that is unable to converge well.

- Since the U_{jm} terms depend on all the c^j_m terms, we will need to transform the training data to form input/output targets for each neural network that take

the $\mathbf{c}^j$ terms of the other propagated states into account.

In light of these constraints, the planned vector evolution method using the neural network requires:

- Eigenvalues $E_{n,0}$ from the H_0 matrix.
- An initial matrix of coefficients $\mathbf{c}^j(t)$ of size $N_{\text{basis}} \times N_{\text{timesteps}}$ for the each of the N_{prop} states j, which will be used to train the network.

Once the network training is completed, the network should be able to predict new values for $\mathbf{c}(t+\tau)$ for a single time-step per calculation. An outside loop will feed the newly calculated $\mathbf{c}(t+\tau)$ values back into the network to continue propagation.

Examining equations (18) and (19), we draw the following conclusions (see figure 11):

- The network outputs (defined as Y) should be the difference between the next time step and the phase-shifted current time step. Phase shifting here refers to the multiplication of a single $c_m^j(t)$ by the natural complex phase factor $e^{\frac{\tau}{i\hbar}E_{m,0}}$. This is the difference between propagating with $H = H^0 + H^1$ vs. only propagating with $H = H^0$. If there were no perturbation, this phase shift would produce the natural, unperturbed oscillation in $c(t+\tau)$.
- The network weights and biases will be adjusted to determine the operator functions $U_{jm}(t)$.
- The network inputs (and therefore the inputs to the sigmoid operator functions) are the $e^{\beta E_{m,0}}$ phase-shifted $c_m^j(t)$ terms.

3.2 Network Design Considerations

In order to accomodate the constraints on network dimensionality, we will need to transform the initial matrix of $\mathbf{c^j}$ vectors to match the proposed network inputs. This

transformation involves two operations:

- Phase shift each of the off-diagonal $c_m^j(t)$ terms by its corresponding $e^{\beta E_{m,0}}$ energy term from the H_0 eigenvalues. Since this phase shift is logically equivalent to the phase shift for the $c_j^j(t)$ terms of the network outputs, this phase shift can be done to the entire $\mathbf{C^j}$ matrix.

- Subtract each $e^{\beta E_{j,0}}$ phase-shifted element of $\mathbf{c^j}(t)$ from its value at the next time step $\mathbf{c^j}(t+\tau)$ to form the network outputs. This will be done for each of the propagated states j. Programmatically, this is equivalent to $\mathbf{c^j}(2..N_{timesteps}) - \mathbf{c^j}(1..N_{timesteps}-1)\, e^{\beta \mathbf{E_{j,0}}}$.

- This transformation will have to be reversed after the network has been trained and produces actual output.

The number of neurons in a hidden layer is equal to the number of sigmoid functions used in the approximation. Using too many neurons slows down the training with no gain (and perhaps loss due to overfitting). The number of neurons will have to be adjusted by trial and error, after making a good estimate, by plotting the oscillations of some of the $\{\psi_j^{(1)}\}$ states in each propagated state. Taking this uncertainty into account, we will refer to N_{neurons} as the number of neurons that will be selected by experimentation.

Because the off-diagonal terms are so close to zero, if we use a single layer in the network, the mean-squared error calculation ignores the accuracy of the $\{\psi_m^{(0)}\}$ basis states with c_m^j values close to 0. This is overcome by introducing a separate layer for each of the $c_j^j(t+\tau)$ terms and having all the $m \neq j$ inputs connected to that layer. The N_{prop} layers are then connected by a final "super-layer" that approximates the target output of the neural network. The super layer combines the sigmoid function approximations of the U_{jm} operators using a linear transfer function. This structure is summarized in figure 11.

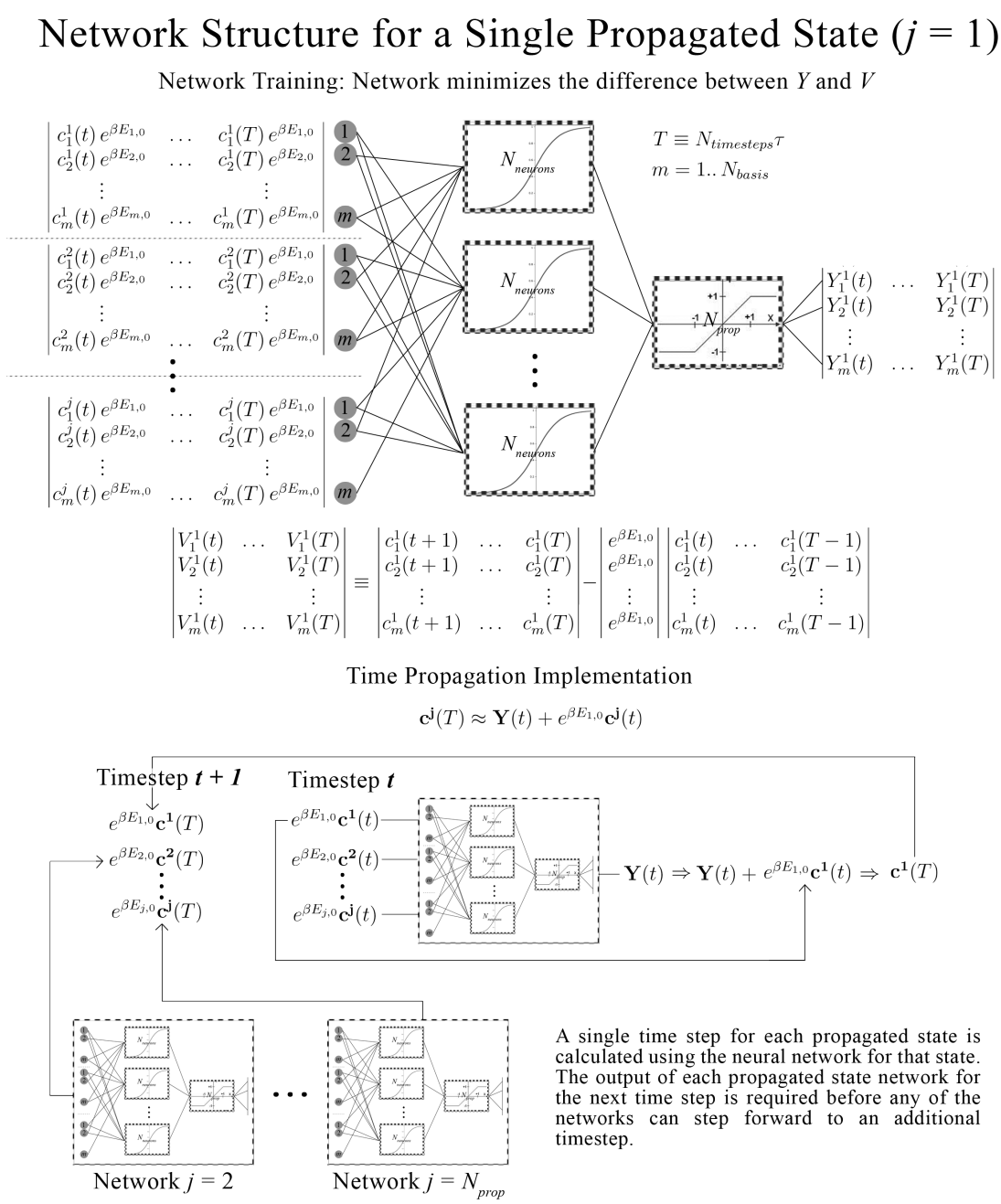

Figure 11: Schematic diagram showing the structure of the neural network and how the input and output matrices are calculated. One such network will be created for each of the propagated states.

3.3 Programmatic Implementation in Matlab

[4]Although the end goal is to solve optical absorption spectra for large nanostructures, methane was selected for proof of concept. It is small and has only a few $\{\psi_j^{(1)}\}$ and $\{\psi_m^{(0)}\}$ states making it ideal for experimentation.

The neural network implementation needs to have the following functionality:

- Parsing of the data generated using density functional theory and transformation into the relevant inputs and outputs. Implemented by **DFT** class.
- Initialization of neural network structure as outlined in figure 11, configuration of inputs/outputs and the ability to train the network. Implemented by **PNet** class.
- Evolution of the trained network to predict future time steps, plotting of network performance, and validation of network results and expectation values. Implemented by **QVE** class.
- Analysis of network predictions to test conformity to known physics principles such as orthonormality and symmetrization requirements. Implemented in **Analyzer** class.

3.3.1 Matlab Neural Network Toolbox

Mathworks provides a neural network toolbox that is actively developed to keep up with the current state of neural network theory. The toolbox allows the creation and training of custom neural networks with many configurable parameters and custom layer and neuron structures. The **PNet** class constructs a custom neural network using the NN toolbox's **net** class and methods according to the structure in figure 11.

[4]Well commented source code for important functions in the network implementation in Matlab is provided in Appendix A.

3.3.2 Methods and Discussion of DFT Class

The **DFT** class handles all the data importing and transformation of the TDDFT data that the networks will be trained on. The class constructor accepts:

- A data file with the values of the $\mathbf{c}^j$ vectors for all the basis states.
- The energies of the ground-state Hamiltonian
- The value of the timestep τ (in seconds) that was used in the computational simulation that generated the data file.
- The number of time steps to use as the initial dataset for the neural networks (includes both training and validation data).
- The number of states $\{\psi_j^{(1)}\}$ with values close to 1 that will be propagated in the $\mathbf{c}^j$ evolution.

The class also has built-in values for methane, which can be automatically loaded by specifying the molecule chemical code as the only argument to the constructor.

During the initialization routines of the constructor, the data file is parsed and the network inputs and outputs are created using the transformations described in the previous section. Since the networks require cell-matrices, the network inputs and outputs are also boxed into the correct format for the **PNet** class. Many of the values required for the network training are also required for the preparation of the network inputs/outputs, the **DFT** class exposes these as public properties so that the other classes can use them without explicit re-initialization. Additionally, the **DFT** class also has methods that can reverse-transform network-predicted data. This facilitates comparison to the original DFT data that the networks were trained on.

3.3.3 Methods and Discussion of the PNet Class

The **PNet** class accepts an instance of the **DFT** class in its class constructor and uses its properties to construct the separate neural networks for the propagated states j. The class stores the initialized networks as publicly accessible properties so that other classes can interact with them and initiate evaluation or retraining. Once the networks have been initialized, the train() method can be called to begin a Levenberg-Marquardt training of the network parameters. Since fine-tuning neural networks can require many iterations, and since training takes longer for larger networks, the class also implements properties and methods to train a single network independently. This allows minor changes to network structure to be tested and validated more quickly.

For its trained networks, the **PNet** class has methods to evaluate the network for given inputs and to step all the networks forward by a single timestep. Since all the networks require *all* the $\mathbf{c}^j$ values in order to approximate the next time step, the networks are each stepped forward a single time step before combining their several outputs into a single input for the next time step. The new input is then fed into each of the networks and the process is repeated.

For molecules with orthonormal states, the class optionally implements Gram-Schmidt orthonormalization between the evolution of time steps. The orthonormalization removes small numerical errors in the approximation that would otherwise compound as further steps are propagated.

3.3.4 Methods and Discussion of the QVE Class

QVE is a wrapper class for **PNet** and **DFT** that contains an instance of each and handles all validation of the networks' predictions. **QVE** validates the networks' performance at two levels:

1) Visual comparison of the charge oscillations for specific basis states ($\psi_m^{(0)}$ subscript m) within a given propagated state ($\psi_j^{(1)}$ superscript j). This check compares

the network's predictions with known, valid, data obtained via density functional theory (the target comparison data is provided by the **DFT** class from its parsed data sets). The graphing methods plot a limited number of the c_m^j states in subplot form for a given propagated state j. They are most useful in comparing the performance of the various j-state neural networks relative to each other. A sample plot of such a test is presented in figure 12. Usually, separate plots are generated for the real, imaginary and complex magnitudes of the c_m^j values, Figure 12 shows only the real part for clarity.

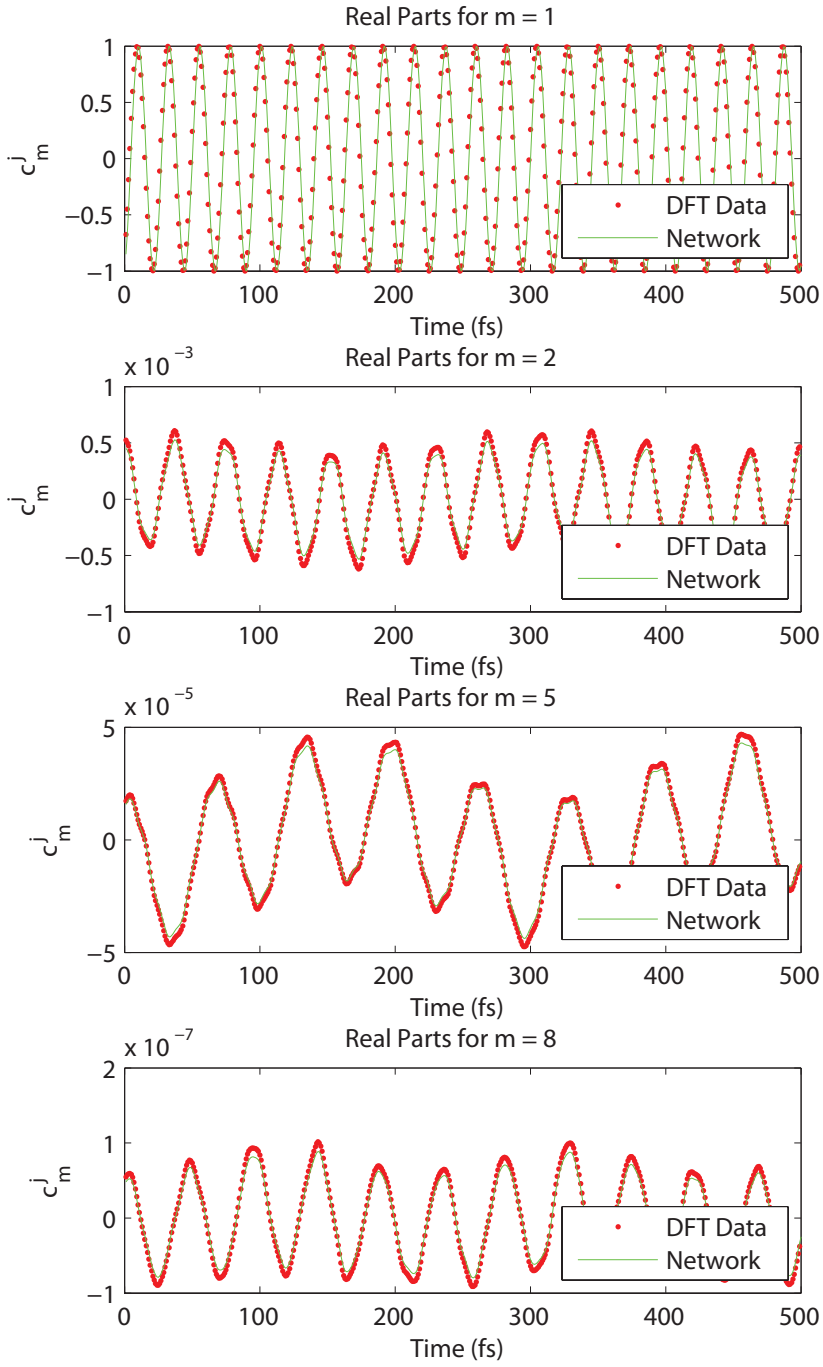

Figure 12: Enlarged output for $\Re(c_m^1)$ from a single network validation test that was performed for methane. The m in the title of each plot refers to the subscript of c_m^j, the "c-vector" being plotted. For $m = 1$, the values are extremely close to unity, confirming the assumption made in the derivation that the $\psi_j^{(1)}$ states for $m = j$ would have the largest occupation (i.e. values close to 1). In this case it is clear that the network approximated the data well. Once a visual check has been performed, the numerical performance values obtained during network training can be considered valid.

2) Calculation of the expectation value of the dipole matrix for the entire system, which is then compared with published data for the molecule. Errors in just one or two of the states c_m^j can have a powerful influence on the correctness of the dipole expectation. Although the visual checks in step 3.3.4 are helpful when fine-tuning the network structure, ultimately the correctness of the dipole expectation value determines the performance of the network. Small errors in the dipole expectation value may be acceptable if they do not affect the peak frequency drastically (obtained via Fourier transform). If the expectation value is severely wrong, individual neural networks can be examined using the tests in part 1 to determine which network is at fault. Sample output from the dipole expectation value and absorption spectra validation are presented in figures 13 and 14 respectively.

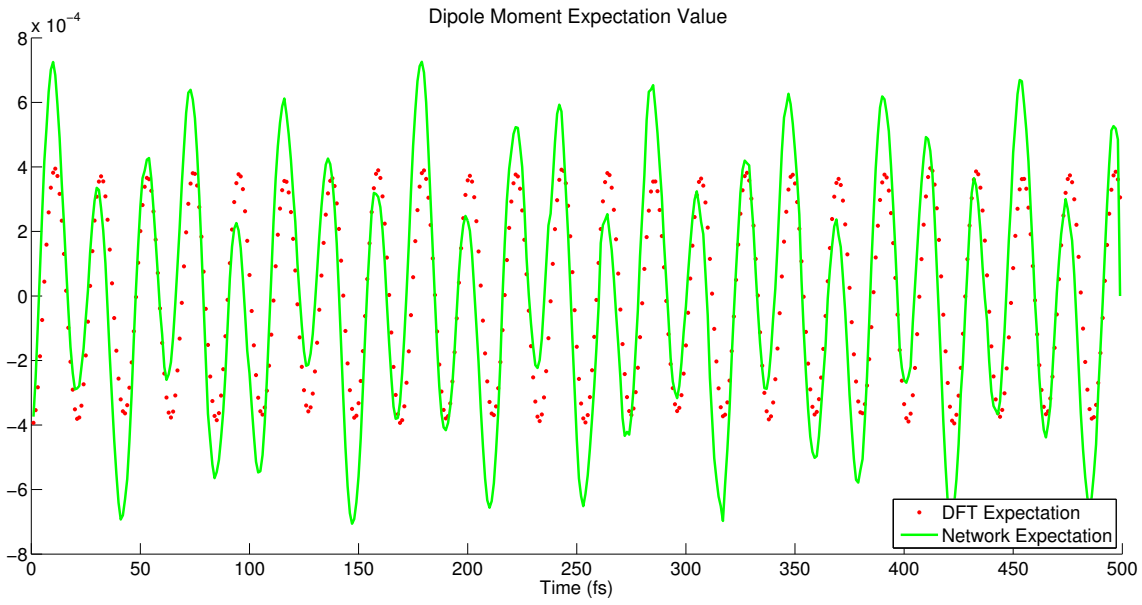

Figure 13: Sample output from validation of the dipole expectation value for methane (see equation 15).

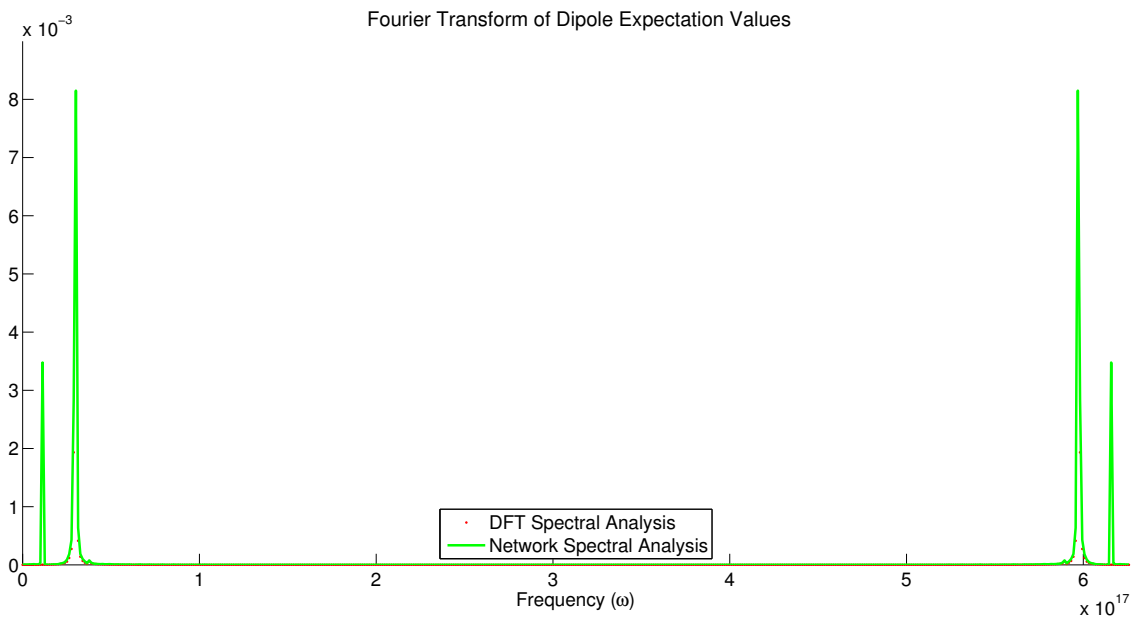

Figure 14: Sample output from validation of the absorption spectra by performing a Fourier transform of the dipole expectation value for methane (shown in figure 13). Even though the networks' prediction for the expectation value was slightly off, the prominent frequencies are sufficiently close that the network might be considered successful.

The first validation check performed on a set of networks ensures that they are able to faithfully reproduce the data they were trained on. If the networks are unable to predict the correct values within $\approx \frac{1}{1000}$ for the data they were trained on, the prediction of time steps outside of the training range is typically poor. This initial check also ensures that the Fourier transform of the expectation value is exact for the prominent frequencies.

If the initial validation check against the training data seems successful, the **QVE** class initiates a predictive time evolution, one step at a time using **PNet** methods. The data predicted by the networks for the extrapolated time steps is then checked for consistency at the network level (test 1) and then at the expectation value level (test 2). If the networks can correctly predict the prominent frequencies of the dipole expectation value, the predictive time evolution is considered successful.

3.3.5 Methods and Discussion of the Analyzer Class

The analyzer class validates both network predictions and the initial dataset from density functional theory to determine if basic physical principles are being followed. The methods of this class are not called automatically by any of the other classes. When network predictions have passed the **QVE** class' validation tests, the **Analyzer** class' methods can be applied to see if/how the networks' prediction deviates from physics. Possible tests that can be performed using this class are:

- Orthonormality: performs simple dot products between each of the $\mathbf{c}^j$ to make sure they are orthogonal. Checks whether $|\mathbf{c}^j| = 1$ for all time steps in the networks' prediction.

- Symmetry: for states known to be symmetrical, **Analyzer** can determine which of the two network approximations is better. Depending on the size of the molecule being approximated with the neural networks, it may be better to approximate only one of many symmetrical states. While there is an obvious gain in computational time, having two separate approximations for the same state may be beneficial if one of them is determined to be a better approximation and is used exclusively for the predictive time evolution.

3.3.6 Sample Network Approximation Script

The following Matlab script shows an example of approximating the absorption spectra of methane using the classes dicussed in this section.

```
%Create a QVE object for methane (ch4). The 'all' parameter ensures
%that all networks will be created and initialized.
q = QVE('ch4','all');
%Train all the neural networks.
q.nets.train;
%Perform a step-by-step evolution of the next 500 steps.
```

```
ev500 = q.nets.evolve(500);
%Visually validate the networks' prediction
q.evvalid(ev500)
```

Part III

Results and Conclusions

4 Optical Absorption Spectra for Methane

4.1 Training, Fine-Tuning and Performance

The network was trained on the first 500 fs of simulation data acquired using density functional theory.

4.1.1 Numerical Training Results

Numerical results of the backpropagation training for each of the networks is displayed in table 1. The average training time for a network was 231.36 seconds. The total training time for all networks was 1853 seconds. This training was carried out on a single core 1.65 GHz CPU.

Neural Network	Training Time (s)	Performance (MSE)
$\Re(\mathbf{c}_m^1)$	230.2	6.37×10^{-3}
$\Im(\mathbf{c}_m^1)$	230.8	5.79×10^{-3}
$\Re(\mathbf{c}_m^2)$	230.8	8.75×10^{-4}
$\Im(\mathbf{c}_m^2)$	231.8	8.27×10^{-4}
$\Re(\mathbf{c}_m^3)$	231.9	2.43×10^{-5}
$\Im(\mathbf{c}_m^3)$	231.0	2.53×10^{-5}
$\Re(\mathbf{c}_m^4)$	234.1	6.69×10^{-4}
$\Im(\mathbf{c}_m^4)$	232.4	7.56×10^{-4}

Table 1: Numerical training results from backpropagation training of the network.

4.1.2 Graphical Validation of Individual States

As explained in sections 1.5 and 1.6, training data is randomly sub-divided into three categories:

- **Training:** data that the network will use exclusively to alter weights and biases.
- **Validation:** data used by the backpropagation algorithm to determine when training is complete and to prevent overtraining.
- **Test:** data used to measure the performance of the network. These samples are compared to the network's prediction and then a mean squared error is calculated that represents how closely the network's prediction matches the true output being approximated.

Graphical validation involves evaluating the network's prediction for the entire training data set (includes training, validation and test data) and comparing that prediction to the original training data obtained using DFT. Graphical validation for each of the $\{\psi_j^{(1)}\}$ states being propagated can be viewed in Appendix B. Although the validation plots for the training data seem good, section 4.3 shows a fundamental flaw in this network structure that these plots failed to highlight.

4.1.3 Dipole Expectation Value and Optical Absorption Spectra

Ultimately, we are after the expectation value of the electric dipole moment, calculated as $\langle z \rangle = \langle \mathbf{C^j} \,|\, z \,|\, \mathbf{C^j} \rangle$ (see equation 15). The network's predicted values for the training data shown in figures 22 through 25 (in Appendix B) was evaluated for the electric dipole moment matrix for methane. The same operation was also performed on the corresponding DFT simulated data and plotted with the network's prediction in figure 15. The plot confirms that the network is able to approximate the expectation value of the electric dipole moment almost exactly for the data it was trained on. Performing

a Fourier transform on $\langle z \rangle$ confirms that the frequencies contained in the network's prediction matches DFT data well for the training set, see figure 16.

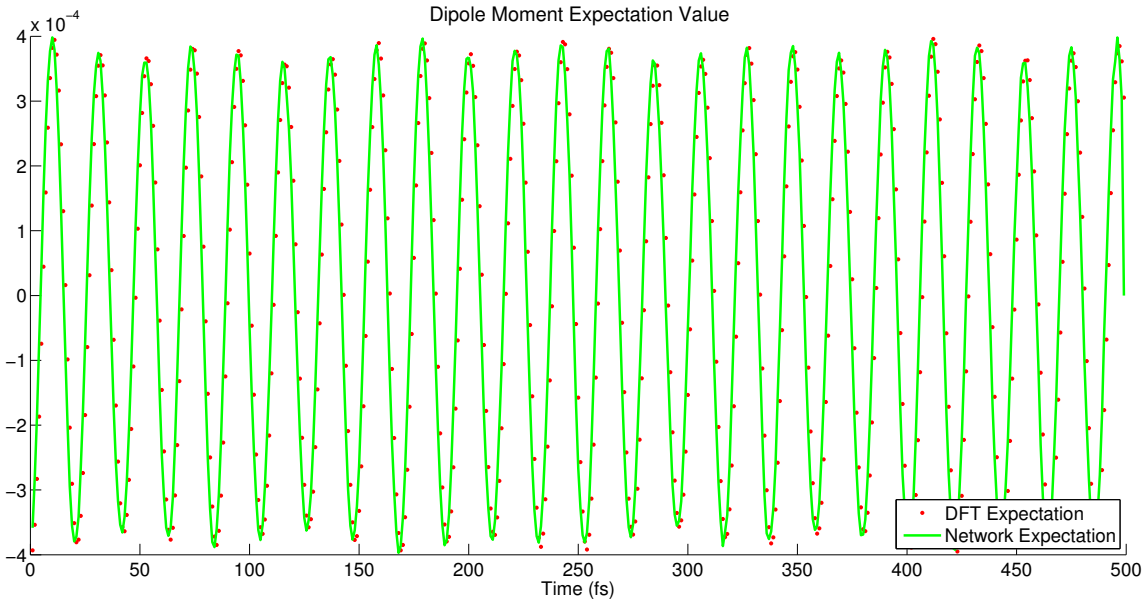

Figure 15: Agreement between the network's prediction of the expectation value for the electric dipole moment and DFT simulated data. Compare figure 13 which used a different network structure and only reached a MSE of 10^{-3} for the training data.

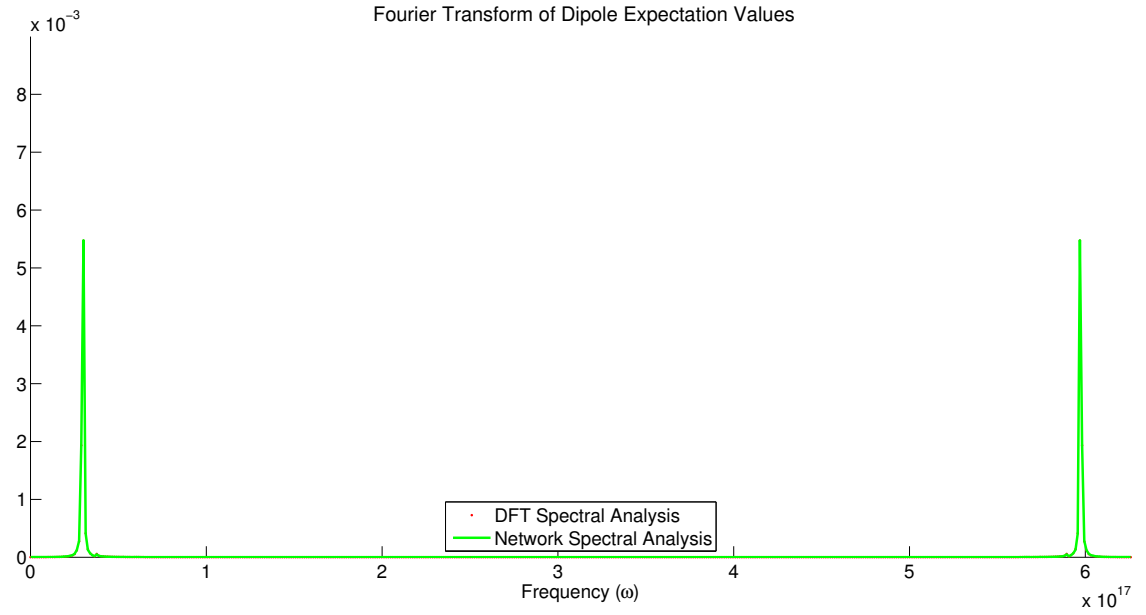

Figure 16: Fourier transform of the expectation value for the electric dipole moment shown in figure 15. See caption of figure 15.

Examining these validation figures seems to indicate that the network's approximation of the data it was trained on is faithful to the DFT data.

4.2 Final Network Structure

The following adjustments resulted from experimentation with the network structure:

1. Real and imaginary parts of the $\mathbf{c}_m^j$ values were approximated using separate networks. The pre-packaged network training algorithms provided with the Matlab neural network toolbox could not handle backpropagation training with complex values. Since the evolution of the $\mathbf{c}_m^j$ values physically depends on the phase information contained in the imaginary part, both real and imaginary parts were presented as inputs to all of the neural networks.

2. Experimentally, the networks were unable to approximate the $\mathbf{c}_m^j$ outputs well when the imaginary parts of all the $\mathbf{c}_m^j$ vectors were used. The final network design approximated the phase evolution by only using the imaginary parts of $\mathbf{c}_m^j$ for the j value being approximated. For example, if $j = 1$ is approximated for $\Re(\mathbf{c}_m^1)$, the network would use as input $\Re(\mathbf{c}_m^j)\,\forall\, j$ and $\Im(\mathbf{c}_m^1)$ (only for $j = 1$).

3. Since the complex phase physically shifts the waveform, the network approximation for the imaginary part $\Im(\mathbf{c}_m^j)$ was connected directly to the output layer. The summation of inputs to the final layer would include this approximation as a linear addition to the other network inputs, mathematically equivalent to a functional shift in the independent parameter.

After experimentally determining this network structure, the number of neurons in each layer was determined. Table 2 shows the experimentally-determined optimal number of neurons in each respective layer. The layer names are described below:

- **Primary:** the primary layers connect directly to the network inputs (excluding

the inputs containing complex phase). These layers are functional approximations of the $\mathbf{c}_m^j$ vector evolution for each respective j.

- **Parent:** the primary layers' outputs become inputs to the parent layers. Every primary layer's output is connected to the input of every parent layer.

- **Output:** the output layer connects to the target output that the network is trying to approximate. Its inputs are the outputs of the parent layers. The parent layers and output layer together represent the U_{jm} linear propagator. The number of neurons in the output layer is fixed by the dimensionality of the target outputs being approximated, i.e. N_{basis}.

- **Phase:** the phase layer connects directly to the input containing complex phase, $\Im(\mathbf{c}_m^j)$, bypasses the parent layer and connects directly to the output layer.

Network Layer	Optimal N_{neuron}
Primary	12
Parent	4
Phase	4

Table 2: Experimental determination of neuron numbers per layer.

The optimal number of neurons N_{neuron} is the result of a trade-off between a lower mean squared error (MSE) and additional time required to the train the network. For example, in the case of the primary layer of the network, having $N_{\text{neuron}} = 24$ reduced the MSE by approximately 3×10^{-3}, but took 2.3× longer to train. Since the final performance of the network was on the order of 10^{-3} in the case of $j = 1$ (see table 1), it was not worth the additional training time to reduce the MSE by so little. Similar ratios (of N_{neuron} to time increase) were calculated for each of the layers. In all cases, the MSE reduction was small compared to additional training time. This suggests that a network's structure (at least in this case) may be more important than the adjustment of the number of neurons.

A graphical representation of the final network structure is shown in figure 17. This network structure is shown to be fundamentally flawed in section 4.3.

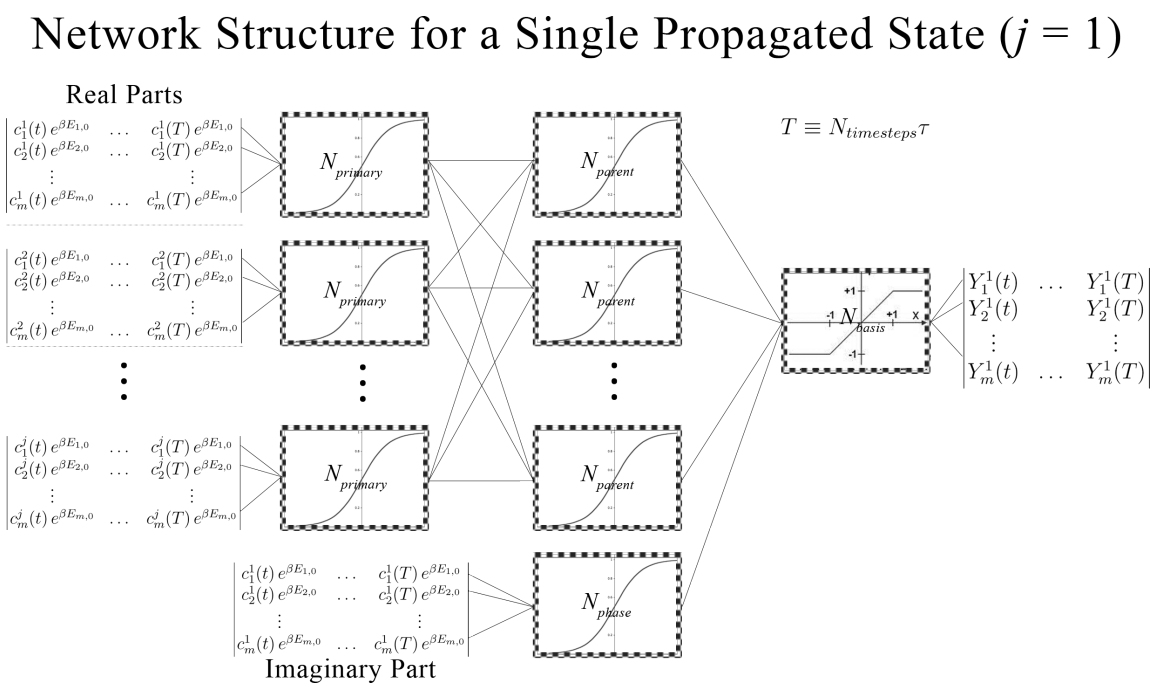

Figure 17: Graphical depiction of final network design. Compare with figure 11.

4.3 Predictive Time Evolution

The trained neural network presented in figure 17 was used to predict c_m^j values outside of the trained data set. Using a single step propagation, each of the 8 networks was advanced by one timestep to predict $c_m^j(t+1)$. The predicted values were then recombined and presented as input to each of the networks in order to produce $c_m^j(t+2)$. This single-step process was repeated for 100 timesteps. Visual validation of this prediction is shown in figure 18. Examination of the figure shows that the network was reasonably successful at predicting the first 40 time steps, after which the prediction became unstable.

During the time steps in which the approximation had the right magnitude, it still seems that the network has no knowledge of the oscillatory nature of the data. The training validation plots seemed to produce the correct absorption spectra, but

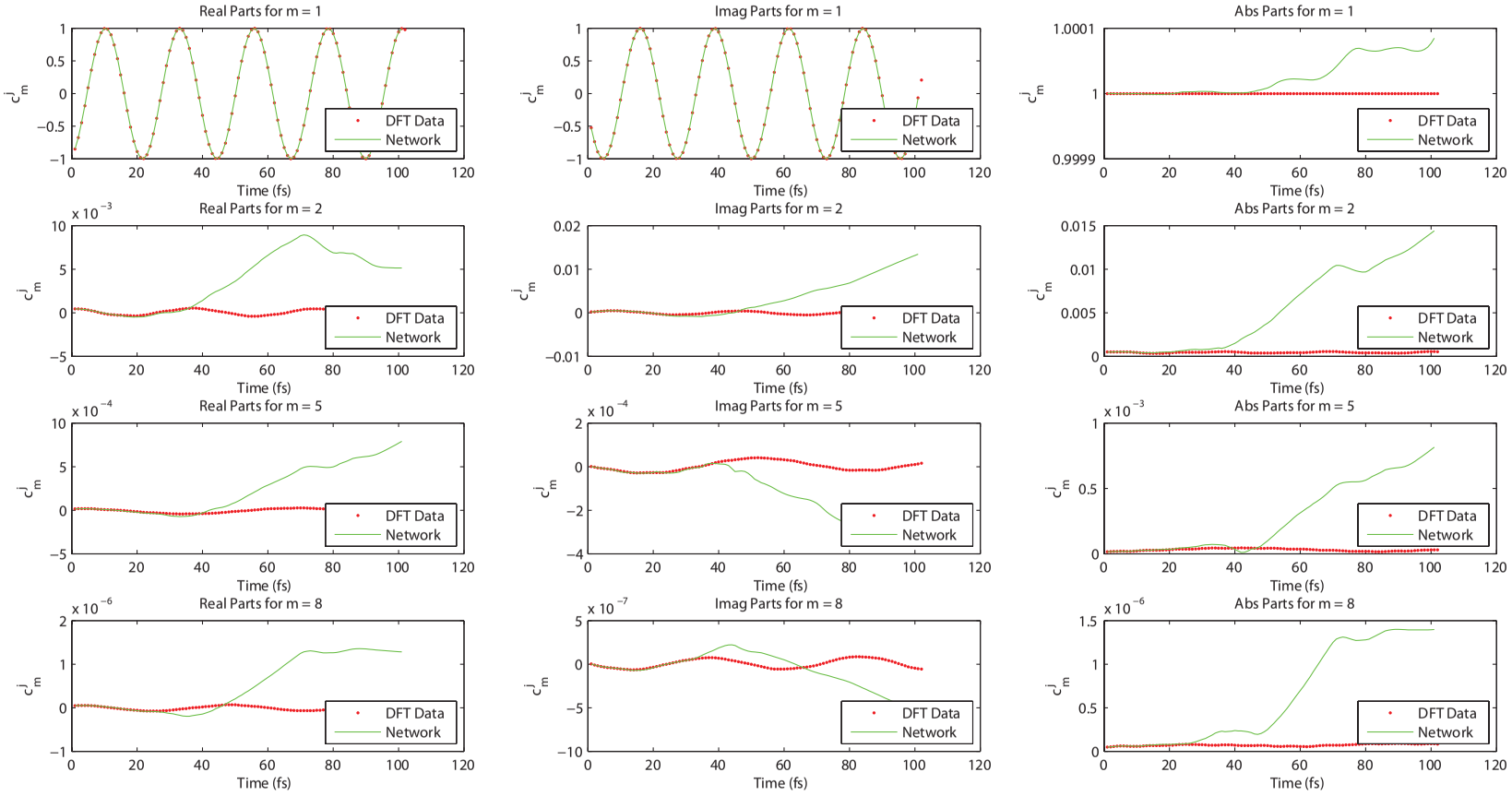

Figure 18: Visual validation of the $j = 1$ states for 100 time steps predicted outside of the networks' training set.

prediction outside the training range is dismal. Since the network is supposed to represent the U_{jm} linear propagator it is instructive to examine instead a validation plot (called propagator plot hereafter) that shows the network's prediction with a single time step prediction horizon. A prediction horizon of 1 means that for each known, true value from DFT, only 1 future time step is predicted. Thus, for the plot in figure 19, each DFT point is only used to predict one other point. A propagator plot for the first 50 time steps from figure 18 is displayed in figure 19. Examining the figure reveals a serious problem with the way that the network is propagating the c_m^j coefficients. Although as a whole the mean squared error is small, it appears that even though the network can globally predict the right wave form, for a single time step, the propagation is completely wrong. It is obvious that for a prediction horizon greater than 1, a correct prediction is not possible. The slopes of the network predictions for the propagator are wrong with few exceptions (for example $t \in (10, 20)$ for $\Re(c_2^1)$ and $\Im(c_5^1)$). In other words, the network does not represent the linear propagator U_{jm} even for the training data.

4.3.1 Optical Absorption Spectra Prediction

Despite the obvious shortcomings of the prediction, the predicted expectation value of the electric dipole moment is surprisingly close. In figures 20 and 21 the dipole expectation value and the Fourier transform of the expectation values is shown. Even though the dipole expectation deviates severely toward the end of the predicted period, the frequency components in the Fourier transform of the prediction still matches the DFT data. Considering the observations in the previous section, it is likely that the problem is not so severe because the network's prediction for the training data was only off by a phase shift. In any case, whatever closeness it resembles to the true system is a matter of chance since the network is not a true functional representation of U_{jm}.

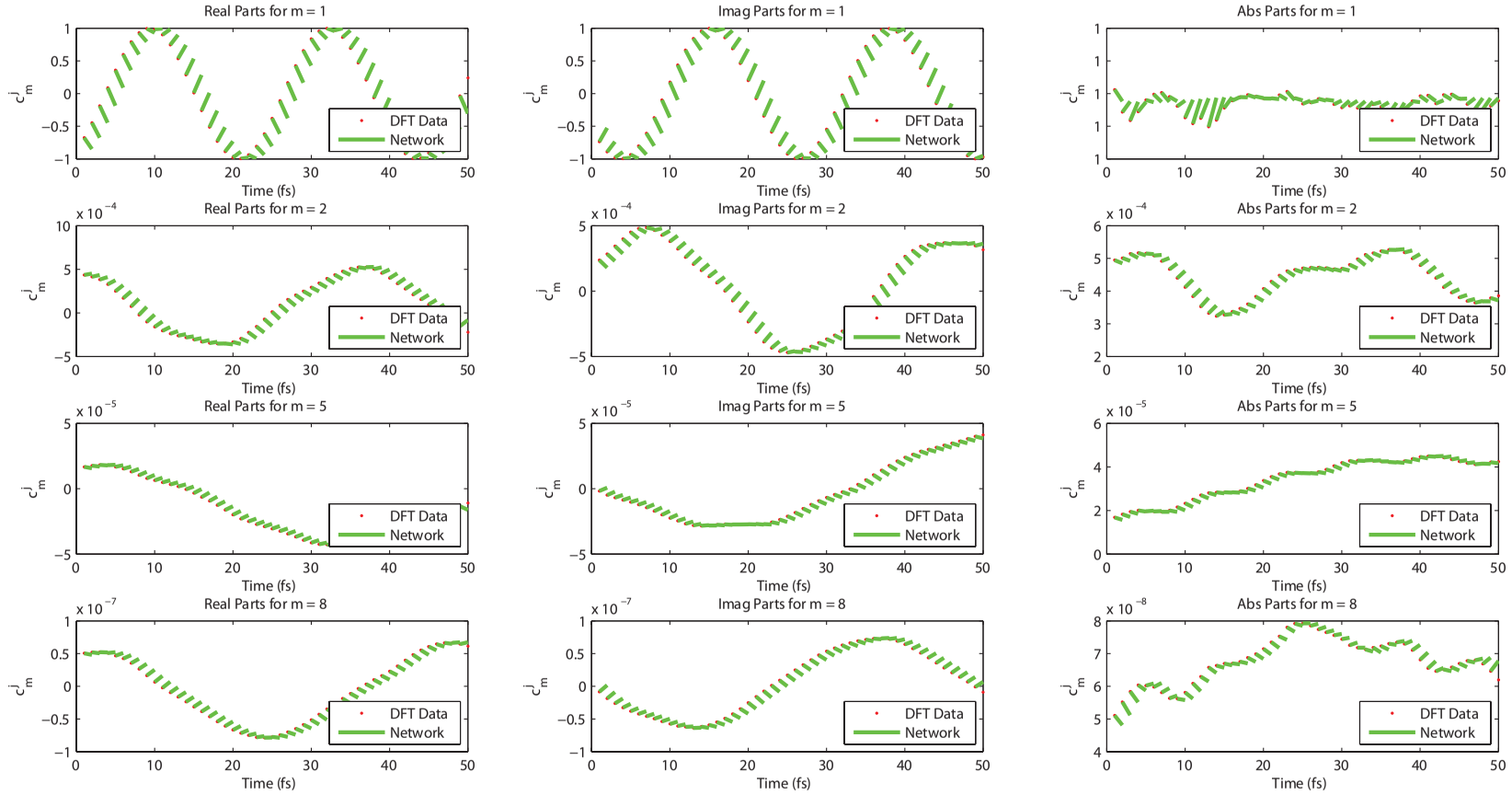

Figure 19: Visual validation by point for a prediction horizon of one time step. The plot uses the same data as figure 18, but only for the first 50 time steps. The lines show the slope of the network's prediction for $c_m^j(t+1)$ for each DFT point. The slope of the network's single time step predictions shows its flawed representation of the U_{jm} linear propagator.

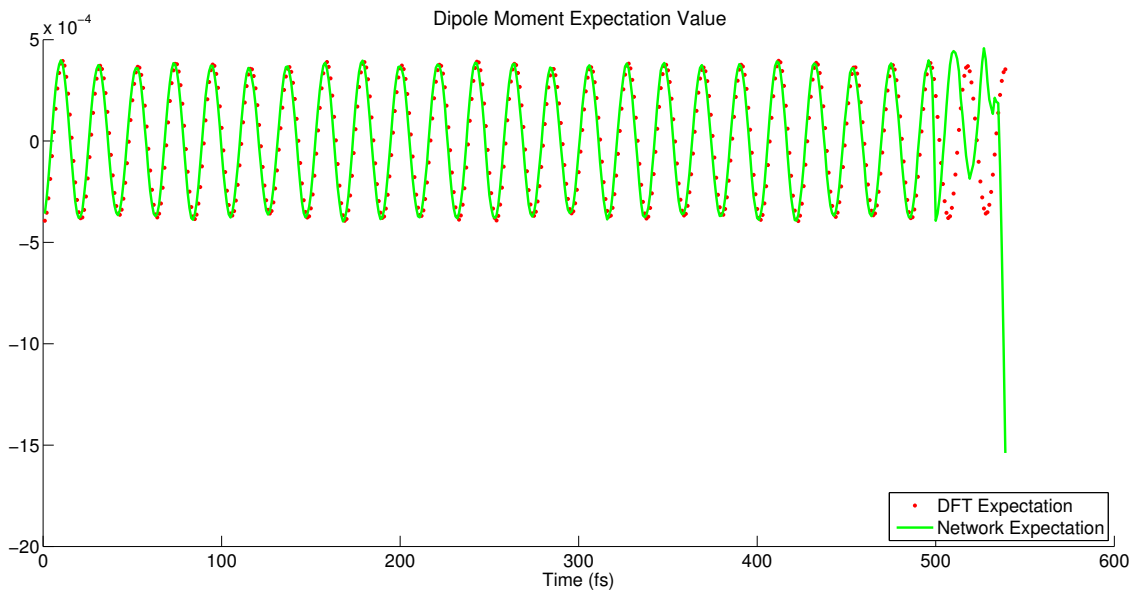

Figure 20: Expectation value of the electric dipole moment for the first 40 time steps predicted by the network.

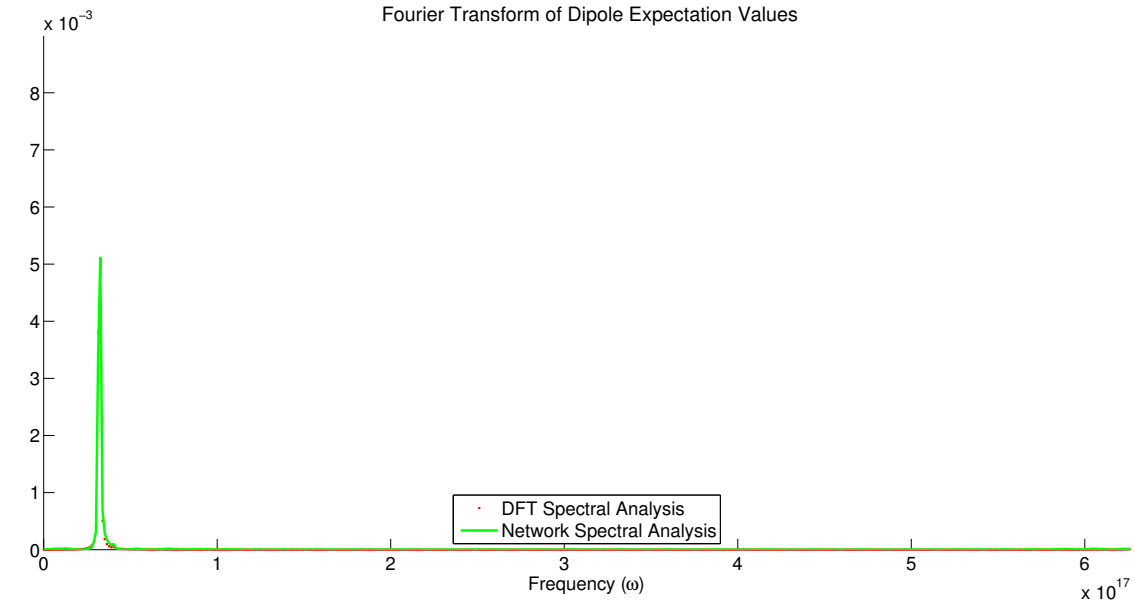

Figure 21: Fourier transform of the dipole expectation values shown in figure 20.

4.4 Discussion

The greatest difficulty with a single-step time propagation is that slight errors in a single time step carry easily and tend to make the entire prediction unstable. That is why it is essential that the network truly represent the linear propagator U_{jm}. In section 2, a sample implementation from Tsolakidis [24] used a linear propagator to solve the optical excitation problem. Their solution also only calculated a single step per iteration but was successful for a prediction horizon of infinity. Their success indicates that the theory of the linear propagator is a valid way to solve the problem. It is clear then, that the problem with the "final" network implementation presented in section 4.2 is that it was not a true representation of U_{jm}.

The current network implementation from section 4.2 is able to correctly predict a phase-shifted wave form for the training data c_m^j across all j and m (albeit only for a prediction horizon of 1). This does grant some confidence that a network structure for U_{jm} could be found. Because the network is off by a phase shift, the most likely culprit is the lack of complex information as network inputs. In section 4.2, the approximation was suggested that some of the complex data, for other j states not being propagated, could be ignored. This was clearly a false assumption. The network theory in section 2 requires a complex matrix U_{jm} as a linear propagator. Two possible options for taking this limitation into account are 1) to create, from scratch, a neural network implementation that can handle complex inputs and outputs [26] or 2) to simultaneously train the real and complex parts for each j state in a single neural network.

Option 1 would provide the most flexibility since equality comparers and algorithms could be changed to handle the complex values as required. Additionally, the network would more fully represent a functional of the complex linear propagator. Books with C++ code excerpts for various algorithms (example [4]) could provide an excellent starting point for ready-made network implementations and speed up the

network creation process. One great disadvantage is that existing Matlab classes and functionality would become obselete. Additionally, Method 2 could end up converging with minimal changes to the network structure and perhaps be a better option. It could also turn out that implementation of the gradient descent training algorithms for complex networks are more complicated, decreasing code maintainability. To that problem, however, Szilagyi does present a complex learning rule [26] that may be suitable.

Option 2 has the problem that the Matlab neural network toolbox cannot handle complex valued inputs, weights and biases. As such, the network treats the real and imaginary parts of the inputs, weights and biases as equivalent. Thus, the real and imaginary parts of the inputs are presented to the network as exclusively real, and important phase information contained in the theory of imaginary numbers becomes lost (and is perhaps irretrievable). If weights and biases were allowed to be complex (method 1), multiplication by weights during network training would constitute a phase shift and could immediately solve the phase problem present in the current design. While experimenting with the structure of the final network presented in section 4.2, various schemes and structures were attempted that approximated real and imaginary parts separately, in parallel, within a single network. None of those attempts were able to converge with errors smaller than 10^{-1}. When the original (flawed) validation plots (for example figures 22 through 25) seemed to indicate convergence for the training data despite the lack of all complex information, it seemed that perhaps the network had managed to approximate the phase changes through its complex functional nature. Valid plots for the expectation values of the dipole matrix (equation 15) added to the false sense of accomplishment. Reviewing the convergence difficulties of managing the complex numbers as groups of real numbers suggests that option 2 is not likely to succeed, even with additional adjustments and experimentation.

4.4.1 Possible Steps for Remediating the Flawed Propagator Network

At this stage, it would be best to return to an extremely simple (toy) model such as a two state system with H^0 equal to the identity matrix (though still time-dependent on the c_m^j), and a small electric field perturbation. A correct network approximation to U_{jm} will most likely require Method 1 discussed above. An existing neural network code base from [4] could be modified with the suggestions in [26] to handle complex inputs, weights and transfer functions as well as network training. Visual validation plots would use propagator plots (like figure 19) exclusively. This should help identify the actual progress of the network as a functional representation of U_{jm} instead of the global approximation in a single time step prediction horizon. A correct neural network solution should accept, as complex inputs, all of the c_m^j values and train toward complex targets.

Most of the decisions outlined in section 3 (relating to the neural network structure) remain valid since they were based in the theory of section 2. As mentioned above, the fatal approximation was ignoring some of the complex parts of c_m^j values and attempting to train the network's real and imaginary parts separately. Those changes were only implemented during the experimentation stage (deviating from the proposed network structure in figure 11). A good starting point for the complex network implementation would still be the network structure presented in section 3 and summarized in figure 11.

4.4.2 General Notes Regarding Network Experimentation

The final network structure (figure 17) was quite successful at approximating the global wave form over a single time step prediction horizon (off only by a phase factor). Thus, even though it didn't solve the linear propagator problem, it was optimized to minimize the error in plots like figs. 22 through 25 (in Appendix B). It is therefore still instructive to consider improvements to the network training process

that will likely need to be implemented in the new complex network.

The predicted results in figure 18, though wrong, are interesting in that the prediction remained stable across multiple maxima and minima for 40 time steps before deviating. Once the deviation started, it carried quickly. The deviations were most likely caused by the $j = 1$ networks that only achieved a perfomance of order 10^{-3}, an entire order of magnitude off from the other networks. Although a single network structure across all the j states and networks would be preferable for code maintenance, it may be necessary to maintain a separate structure for "problem" states such as $j = 1$. As discussed with table 2, it is possible to boost the numbers of neurons (and network performance) in the various layers at the cost of computation time in the training algorithm. Since the other networks $j \neq 1$ performed well, it seems unnecessary to introduce such changes into their network structures.

As the number of inputs and layers in the network increases, the combinations of possible hidden layer connections and neuron counts goes up dramatically, increasing the experimentation time required to have the network converge on a good approximation. Although some parts of the network structure can be predicted by comparison with the physics behind the system (as discussed in section 3.1), there is still a lot of room for experimentation and parameter adjustment. Because the network is approximating a complex, non-linear function as a linear functional of a sum of sigmoid functionals, there is not a unique network solution to the problem. This introduces the necessity of experimentation in network structure and the number of neurons in the various layers. As the network design becomes more complicated and training time increases, adjusting parameters and experimenting with their results takes longer. Ideally, experimentation with numbers of neurons and connections between layers could be scripted and run in parallel on a supercomputer. A program could be written to manage the combinations of parameters tried and analyze their performance without intervention.

The network design and training for the methane approximation was all performed on a single-core, 1.65 GHz CPU. One of the great advantages of neural networks and the methodology presented in this thesis is that they lend themselves well to parallelism on a supercomputer. Only the single-step propagation requires interaction between the various networks, allowing each network to be trained on separate processors. Since the results at this point are incomplete, it is not possible to make an accurate comparison with the CPU time required to produce the full-length DFT simulation that the networks are trying to approximate. The trained neural networks take approximately 1.1 MB of memory for the methane case. Taking both memory and CPU into account, the neural network methodology remains a possible avenue for improving the computation time involved in an optical absorption spectra calculation, if a complex-value network can be found to approximate U_{jm}.

4.4.3 Additional Considerations for the New Complex-valued Network

The linear propagator U_{jm} is supposed to be unitary. This preserves the orthogonality of the states during propagation. An additional validation check that should be performed should be the application of U_{jm} (represented by the neural network functional) onto a unit matrix to determine the network's matrix value. That matrix could then be compared to theoretical expectations for U_{jm} to determine if it is a close fit. The orthonormality functions in the **Analyze** class should also be applied routinely to network predictions as part of the validation.

5 Conclusion

The ability of neural networks to approximate arbitrary functions was applied to a complex, non-linear, electric-field perturbation problem. Although the network methodology used to approximate the perturbation seemed effective at approximat-

ing the data on which it was trained, closer examination revealed a fundamental flaw in the network's structure. The network structure that was determined by experimentation ignored important complex information because of limitations in the Matlab neural network toolbox. The networks trained real and imaginary parts separately and didn't use all the complex information available in the approximation. Additionally, the complex information that was used lost important phase information. These factors, relating to the complex c_m^j values, resulted in a network that was not an approximation to the linear propagator.

Creating a new, complex-valued neural network using the same theoretical basis and theoretically motivated network structure may be able to approximate the linear propagator. While most of the proposed network structure from this thesis would remain valid, the existing code base will be inadequate. New network classes and a complex training algorithm could be adapted from existing, third-party code and publications about complex networks. This would reduce the amount of time required to create a new code base from scratch.

References

[1] Swingler, K. “Applying Neural Networks: A Practical Guide”, London: Academic Press Limited, 1996. 5

[2] Hornik, K. M.; Stinchcombe, M. and White, H. “Multilayer feedforward networks are universal approximators,” Neural Networks, vol. 2, no. 5, 359–366 (1989) 13

[3] Hecht-Nielsen, Robert “Kolmogorov’s Mapping Neural Network Existence Theorem”, Proc. I987 IEEE International Conference on Neural Networks, IEEE Press, New York, III (11-13), 1987. 13

[4] Masters, Timothy “Advanced algorithms for neural networks: a C++ sourcebook”, Wiley, 1995. 15, 49, 51

[5] Gudise, V.G.; Venayagamoorthy, G.K. "Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks," Swarm Intelligence Symposium, 2003. SIS ’03. Proceedings of the 2003 IEEE , pp. 110- 117, 24-26 April 2003 15

[6] Van Rooij, A. J. F.; Johnson, R. P. and Jain, L. C. “Neural Network Training Using Genetic Algorithms” World Scientific Publishing Co., Inc., River Edge, NJ, USA. 1996. 15

[7] Hecht-Nielsen, "Theory of the backpropagation neural network," International Joint Conference on Neural Networks, 1989. IJCNN., pp.593-605 vol.1. 14

[8] Mathworks, Neural Network Toolbox Documentation, "Multilayer Training Speed and Memory", Accessed December 2012. <http://www.mathworks.com/help/nnet/ug/speed-and-memory-comparison-for-training-multilayer-networks.html> 16

[9] Hagan, Martin T.; Demuth, Howard B. and De Jesús, Orlando "An introduction to the use of neural networks in control systems", International Journal of Robust and Nonlinear Control, 2002. pp. 959-985. 5, 10

[10] bssaonline.org, "Adapting the Neural Network Approach to PGA Prediction", Accessed December 2012. <http://www.bssaonline.org/content/102/4/1446/F4.expansion.html>

[11] Quito, Marcelino Jr.; Monterola, Christopher; Saloma, Caesar "Solving N-Body Problems with Neural Networks", Phys. Rev. Lett. 86.4741, 2001. 1, 18

[12] Monterola, Christopher; Saloma, Caesar "Solving the nonlinear Schrodinger equation with an unsupervised neural network", Optics Express vol 9, pg 72-84, 2001. 1

[13] Argaman, N. and Makov, G. "Density Functional Theory - An Introduction," American Journal of Physics, pp. 69-79, 2000. 21

[14] Burke , K. et al. "The ABC of DFT", Irvine: University of California, 2007. 20

[15] Marques, M. et al. "Time-dependent density functional theory", Berlin: Springer, 2006. 21

[16] Kohn, W. and Sham, L.J. "Self–consistent equations including exchange and correlation effects," Phys. Rev. 140, A1133–1138 (1965). 21

[17] Vosko, S.H.; Wilk, L. and Nusair, M. "Accurate spin–dependent electron liquid correlation energies for local spin density calculations: a critical analysis," Can. J. Phys. 58, 1200–1211 (1980). 21

[18] Perdew, J.P. et al. "Atoms, molecules, solids, and surfaces: applications of the generalized gradient approximation for exchange and correlation," Phys. Rev. B 46, 6671–6687 (1992) 21

[19] Castro, Alberto; Rubio, Angel et al. "The Octopus Manual", v 4.0.1. Accessed December 2012. <http://www.tddft.org/programs/octopus/wiki/index.php>

[20] Van Milligen, B. Ph.; Tribaldos, V.; Jiménez, J. A. "Neural Network Differential Equation and Plasma Equilibrium Solver" (1995) 1

[21] Khaliullin, Rustam Z. et al. "Graphite-diamond phase coexistence study employing a neural-network mapping of the ab initio potential energy surface" (2010) 1

[22] Behler, J and Parrinello, M "Atom-centered symmetry functions for constructing high-dimensional neural network potentials" (2007) 1

[23] Morawietz, Tobias; Sharma, Vikas and Behler, Jörg "A neural network potential-energy surface for the water dimer based on environment-dependent atomic energies and charges" J. Chem. Phys. 136, 064103 (2012) 1

[24] Tsolakidis, Argyrios; Daniel Sa´nchez-Portal and Richard M. Martin "Calculation of the optical response of atomic clusters using time-dependent density functional theory and local orbitals" Phys. Rev. B 66, 235416 (2002) 22, 24, 49

[25] Hess, Bret C; Jensen, Daniel S and Okhrimenko, Ivan G "Spatial distribution of electron densities during optical excitation of C60" J. Phys.: Condens. Matter 22 (2010) 22

[26] Szilagyi, Miklos N . "Neural Networks with Complex Activations and Connection Weights" Complex Systems 8 (1994). 49, 50, 51

Part IV

Appendices

Appendix A: Selected Function Listing for Network Implementation

Note: in some of the larger files, code that is not relevant to the network implementation was removed. Examples of these removals are property definitions, getters and setters, numerical constants, file paths, and exception generators. Since class definition files are usually exclusively constructed of these items, they have been excluded. The reader may assume that references to the class object, its properties and methods exist, even though they are not included.

DFT Class

Network Input Creation Function cminput.m

```
function cminput(obj)
%CMINPUT Prepares the input matrices for network training
%PARAMETERS - retrieved from properties of referenced object
%   rdata: the real part of the c-vector values for all propagated
%   states and time steps in the simulation.
%   idata: the imaginary part of the c-vector values for all
%   propagated states and time steps in the simulation.
%   energies: the eigenvalues of the H0 basis {psi_m^0}
%   tau: the time step of the simulation
%DESCRIPTION
%   Creates cell arrays for the real and imaginary parts of the input
%   dataset for each of the propagated states. Returns the actual
%   input matrix that can be used to train the networks. The returned
%   matrix has dimensions nbasis*nprop rows x ntimesteps - 1 cols.
%OUTPUTS - Values changed in the referenced object
%   rins: the real part of the combined network inputs
%   iins: the imaginary part of combined network inputs
%METHOD
%   Because the data is supplied as alternating real and imaginary
%   columns, the import script returns separate real and imaginary
%   datasets. The script combines the datasets and then multiplies by
%   the exp(i*energy) phase factor. It is important to use the
%   complex numbers because the phase factor involves multiplication
%   with i.

    %Get the combined real and imaginary data.
    combdata = obj.cdata;
    %Extract the real and imaginary parts of the combined cell array
    obj.rins = cell(obj.nprop,obj.timesteps-1);
    obj.iins = cell(obj.nprop,obj.timesteps-1);

    for k=1:obj.nprop
        %Now phase shift each of the c-vectors from the imported
```

```
        %cdata matrix for pstate j and bstate m
        for m=1:obj.timesteps-1
            value = obj.shiftm(combdata{k}(:,m))';
            obj.rins{k,m} = real(value);
            obj.iins{k,m} = imag(value);
        end
    end

    %Combine the real and imaginary matrices into a single matrix
    %so that all complex data is presented as input to the network
    obj.ains = vertcat(obj.rins,obj.iins);
end
```

Network Output Creation Function cmoutput.m

```
function cmoutput(obj)
%CMOUTPUT Prepares the output matrices for network training
%PARAMETERS - retrieved from properties of referenced object
%   rdata: the real part of the c-vector values for all propagated
%   states and time steps in the simulation.
%   idata: the imaginary part of the c-vector values for all
%   propagated states and time steps in the simulation.
%   energies: the eigenvalues of the H0 basis {psi_m^0}
%   tau: the time step of the simulation
%DESCRIPTION
%   Creates cell arrays for the real and imaginary parts of the input
%   dataset for each of the propagated states. Returns the actual
%   output matrix that can be used to train the networks. The
%   returned matrix has dimensions nbasis x ntimesteps - 1.
%OUTPUTS - Values changed in the referenced object
%   routs: the real part of the combined network outputs
%   iouts: the imaginary part of combined network outputs
%METHOD
%   Because the data is supplied as alternating real and imaginary
```

```
20 %   columns, the import script returns separate real and imaginary
21 %   datasets. The script combines the datasets and then multiplies
22 %   steps 2:ntimesteps by the exp(i*energy) phase factor and
23 %   subtracts them from the 1:ntimesteps matrix.
24
25     %Prepare a temporary vector of the exp(i*energy) terms that need
26     %to be multiplied by timesteps 2:ntimesteps
27     expenergy = exp(obj.tau/(1i*obj.hbar)*obj.energies);
28
29     %Get the combined real and imaginary data.
30     combdata = obj.cdata;
31     comblhs = cell(obj.nprop,1);
32     combrhs = cell(obj.nprop,1);
33
34     %Get the LHS and RHS of matrix V constructed (see figure 11)
35     for k=1:obj.nprop
36         comblhs{k} = combdata{k}(:,2:obj.timesteps);
37         combrhs{k} = combdata{k}(:,1:obj.timesteps-1);
38     end
39
40     %Now phase shift the diagonal elements of the combrhs matrix.
41     %Since all c-vectors for basis state 1 are all in the same row,
42     %we only need to multiply that row by the phase factor.
43     occind = obj.occupied;
44     %The indices of the mostly occupied states in each cell matrix
45     for k=1:obj.nprop
46         index = occind(k);
47         combrhs{k}(index,:) = expenergy(k)*combrhs{k}(index,:);
48     end
49
50     %Finally, we need to subtract the mostly occupied states at
51     %1:ntimesteps-1 from the corresponding ones in 2:ntimesteps
52     couts = cell(obj.nprop,1);
```

```
53      for k=1:obj.nprop
54          couts{k} = comblhs{k};
55          couts{k}(:,:) = couts{k}(:,:)-combrhs{k}(:,:);
56      end
57
58      %Transform the complex outputs to the nn required format
59      couts = obj.box(couts);
60      %Now we just need to separate the real and imaginary parts,
61      %initialize the cell arrays in the DFT object.
62      obj.routs = cell(obj.nprop,1);
63      obj.iouts = cell(obj.nprop,1);
64
65      for k=1:obj.nprop
66          %initialize a temporary cell array to hold the outputs.
67          temprouts = cell(1,obj.timesteps-1);
68          tempiouts = cell(1,obj.timesteps-1);
69
70          %Get the real and imaginary parts of the calculated nn outputs.
71          for m=1:obj.timesteps-1
72              temprouts{1,m} = real(couts{k,m});
73              tempiouts{1,m} = imag(couts{k,m});
74          end
75
76          %Set the outputs of the actual DFT object.
77          obj.routs{k} = temprouts;
78          obj.iouts{k} = tempiouts;
79      end
80  end
```

Network Output Reverse Transformation Function rtransform.m

```
1  function rapprox = rtransform(obj,netapprox,index)
2  %RTRANSFORM Reverse transforms network output data to a c-matrix
3  %PARAMETERS
```

```
4  %   netapprox: the cell matrix of network predicted outputs for
5  %   c_m^j of j = index.
6  %   index: the zero-based index of the jth {psi_j^(1)} state.
7  %DESCRIPTION
8  %   In order to train the network, the DFT data was transformed to
9  %   match the network structure from the theory (see figure 11).
10 %   Before we can compare this data to DFT predicted values, it
11 %   needs to be reverse transformed. According to theory,
12 %   c(t+tau) = Y(t) + exp(energy)*c(t)
13
14     %Prepare a temporary vector of the exp(i*energy) terms that
15     %need to be multiplied by timesteps 2:ntimesteps
16     expenergy = exp(obj.tau/(1i*obj.hbar)*obj.energies);
17
18     %Get the combined real and imaginary data.
19     combdata = obj.cdata;
20     combrhs = combdata{index}(:,1:obj.timesteps-1);
21
22     %Now phase shift the diagonal elements of the combrhs matrix.
23     %Since all c-vectors for basis state 1 are all in the same row,
24     %we only need to multiply that row by the phase factor.
25     %Find the indices of the states in each cell matrix with c_m^j
26     %values that are close to 1.
27     occindex = obj.occupied(index);
28     combrhs(occindex,:) = expenergy(index)* ...
29         combdata{index}(occindex,1:obj.timesteps-1);
30
31     %Finally, we need to add the {psi_j^(1)} states at
32     %1:ntimesteps-1 to the corresponding ones in 2:ntimesteps
33     %to undo the initial subtraction that was done.
34     rapprox = netapprox+combrhs;
35 end
```

Propagated State j Phase Shiftor shiftj.m

```
function cjout = shiftj(obj,cjin)
%SHIFTJ Phase shifts a c-matrix for a {\psi_j^(1)} state.
%PARAMETERS
%   cjin: the matrix of c-vectors(rows) by time-steps(columns)
%   for state j.
%DESCRIPTION
%   Calls shiftm() on each c-vector in the specified input matrix
%   to produce the LHS input for the network (see figure 11).
    dsize = size(cjin);
    cjout = zeros(dsize(1),dsize(2));

    for k=1:dsize(2)
        cjout(:,k)=obj.shiftm(cjin(:,k));
    end
end
```

Basis State m Phase Shiftor shiftm.m

```
function cshifted = shiftm(obj, cvector)
%SHIFTM Phase shifts a c_m^j vector by exp(energy) term.
%PARAMETERS
%   cvector: the c_m^j vector associated with a specific state
%   and time-step that is being propagated.
%DESCRIPTION
%   Prepares an input c_m^j vector that includes the occupation of
%   all the {psi_m^(0)} states for the propagated {psi_j^(1) state
%   by including the natural (unperturbed) oscillating term
%   exp(tau/i*hbar*energy(m))
    cshifted = exp(obj.tau/(1i*obj.hbar)*obj.energies).*cvector';
end
```

PNet Class

Network Approximation Function approx.m

```
function ntransform = approx(obj,index,ains)
%APPROX Evaluates the network for the specified input.
%PARAMETERS
%   index: the zero-based index of the {psi_j^(1)} being propagated
%   ains: (optional) the cell matrix of all complex inputs to
%   evaluate. The matrix dimensionality should be nprop*nbasis rows
%   with as many columns as timesteps that need to be evaluated. If
%   unspecified, the training data set is used.
%DESCRIPTION
%   This function simulates the network for the specified j index
%   and returns the reverse-transformed complex approximation. The
%   specified inputs to evaluate are normalized to be of order 1 in
%   all rows using mapminmax(). After the network has predicted an
%   output, it is reverse-normalized using mapminmax() again before
%   being passed to the DFT class' rtransform function.
    if (nargin > 2)
        rapprox = sim(obj.trnets{index},mapminmax(ains));
        iapprox = sim(obj.tinets{index},mapminmax(ains));
        uiapprox = mapminmax('reverse',iapprox{1}, ...
            obj.inormalizations{index});
        urapprox = mapminmax('reverse',rapprox{1}, ...
            obj.rnormalizations{index});
        napprox = urapprox + 1i*uiapprox;
        ntransform = obj.DFT.rtransform(napprox, index);
    else
        rapprox = sim(obj.trnets{index},mapminmax(obj.DFT.ains));
        iapprox = sim(obj.tinets{index},mapminmax(obj.DFT.ains));
        uiapprox = mapminmax('reverse',iapprox, ...
            obj.inormalizations{index});
```

```
30          urapprox = mapminmax('reverse',rapprox, ...
31              obj.rnormalizations{index});
32          napprox = cell2mat(urapprox) + 1i*cell2mat(uiapprox);
33          ntransform = obj.DFT.rtransform(napprox, index);
34      end
35  end
```

Network Creation Function createnets.m

```
1  function createnets(obj)
2  %CREATENETS Creates all neural networks for c_m^j vector data
3  %DESCRIPTION
4  %   Creates a two neural networks (one real, one imaginary) for
5  %   each of the propagated {psi_j^(1)} states in the object's
6  %   DFT training data.
7  %INPUTS - retrieved from object reference
8  %   ains: the combined real and complex inputs to train with.
9  %   routs: the real targets that the networks will approximate.
10 %   iouts: the imag targets that the networks will approximate.
11 %OUTPUTS - altered in the referenced object.
12 %   rnets/inets: cell arrays for the real and imaginary networks
13 %   being created.
14
15     %Initialize the cell arrays for the real and imaginary networks
16     %and training data.
17     obj.rnets = cell(obj.DFT.nprop,1);
18     obj.rtrain = cell(obj.DFT.nprop,1);
19     obj.inets = cell(obj.DFT.nprop,1);
20     obj.itrain = cell(obj.DFT.nprop,1);
21
22     %Iterate through each of the states and create a network for
23     %the occupied states. The netinit function takes care of all
24     %the initialization etc.
25     for k=1:obj.DFT.nprop
```

```
26          obj.rnets{k} = obj.initnet(obj.DFT.ains, ...
27              obj.DFT.routs{k},k,1);
28          obj.inets{k} = obj.initnet(obj.DFT.ains, ...
29              obj.DFT.iouts{k},k,0);
30      end
31
32      %Also initialize the cell arrays that will hold the
33      %trained networks.
34      obj.trnets = cell(obj.DFT.nprop,1);
35      obj.tinets = cell(obj.DFT.nprop,1);
36      %Set the initialization flag so that training can continue.
37      obj.initialized = 1;
38  end
```

Single-step Time Evolution Function evolve.m

```
1   function approx = evolve(obj,timesteps,firststep)
2   %EVOLVE Steps the simulation forward in time predictively
3   %PARAMETERS
4   %   timesteps: the number of timesteps to predict outside of the
5   %   training data.
6   %   firststep: the initial c_m^j vector that will be presented to
7   %   the network.
8   %DESCRIPTION
9   %   Uses the initial (i.e. t = 0) c—vector for the real and
10  %   imaginary network training data and then steps the networks
11  %   forward in time (one step at a time) to predict timesteps.
12
13      %Initialize the approximation that is being returned and get
14      %the first timestep from the training data
15      approx = cell(obj.DFT.nprop,1);
16      for k=1:obj.DFT.nprop
17          approx{k} = zeros(obj.DFT.nbasis,1);
18      end
```

```
19
20      %Initialize the variable that will hold the network
21      %approximation from the previous step.
22      papprox = cell(obj.DFT.nprop,1);
23
24      %See if we are starting from step 1 automatically, or if we
25      %have a training start variable.
26      if (nargin < 3)
27          for k=1:obj.DFT.nprop
28              papprox{k} = obj.DFT.cdata{k}(:,1);
29              approx{k}(:,1) = papprox{k};
30          end
31      else
32          for k=1:obj.DFT.nprop
33              papprox{k} = firststep{k};
34              approx{k}(:,1) = papprox{k};
35          end
36      end
37
38      %Now just iterate for the specified number of timesteps
39      expenergy = exp(obj.DFT.tau/(1i*obj.DFT.hbar)* ...
40          obj.DFT.energies);
41      occupied = obj.DFT.occupied;
42      timetotal = 0;
43
44      for k=1:timesteps
45          tstart = tic;
46          %Calculate the network's approximation
47          napprox = obj.pstep(papprox,expenergy,occupied, k);
48
49          %The network approximation is a cell variable. We need to
50          %set the value of the final approximation for this timestep
51          %one cell at a time because of how the matrices are setup.
```

```
52          for m=1:obj.DFT.nprop
53              approx{m}(:,k+1) = napprox{m};
54          end
55
56          %Finally, overwrite the value of papprox to be the value%
57          %that we just calculated
58          papprox = napprox;
59          telapsed = toc(tstart);
60          timetotal = timetotal + telapsed;
61          fprintf('Simulated t = %g\t%g seconds\tTotal Time: %g\n', ...
62              k, telapsed, timetotal);
63      end
64  end
```

Generic Network Initialization Function initnet.m

```
1  function net = initnet(obj,input,output,index,real)
2  %INITNET Initializes a new neural network
3  %PARAMETERS
4  %   input: the combined complex input to the network.
5  %   output: the target output being approximated.
6  %   index: the zero-based index of the {psi_j^(1)} to approximate
7  %   real: boolean, specifies whether the network is being setup
8  %   for the real or imaginary parts of the complex inputs.
9  %DESCRIPTION
10 %   Prepares a custom network as shown in section 4.2, figure 17.
11     net = network;
12     nstates = obj.DFT.nprop;
13
14     %Initialize the number of inputs and layers
15     net.numInputs = 2*nstates;
16     net.numLayers = nstates*2 + 2;
17
18     %Connect the inputs to the layers. Each of the bound states
```

```
19      %will be connected to its own layer within the network. These
20      %layers will be summed into the super layer
21      for k=1:nstates
22          if (real)
23              net.inputConnect(k,k)=1;
24          else
25              net.inputConnect(k,k+nstates)=1;
26          end
27          net.layerConnect(nstates+1+k,1:nstates)=1;
28      end
29
30      %Connect up the additional real/imag layer for the state
31      %being calculated.
32      if (real)
33          net.inputConnect(nstates+1,index+nstates) = 1;
34      else
35          net.inputConnect(nstates+1,index) = 1;
36      end
37
38      %Connect each of the state layers to its parent and the parents
39      %to the super layer.
40      net.layerConnect(net.numLayers,nstates+1:nstates*2+1)=1;
41      %Set a bias on the parent and super-layers to shift the result
42      net.biasConnect(nstates+1:net.numLayers) = 1;
43      %Now connect up the final super-layer to the output so that
44      %we can train against it
45      net.outputConnect(net.numLayers)=1;
46
47      %Now we need to choose the number of neurons to train with.
48      %These values are altered over repeated trainings to find the
49      %optimal number for performance and training time.
50      for k=1:nstates
51          net.layers{k}.size = 12;
```

```
52          net.layers{k}.transferFcn = 'tansig';
53          net.layers{k}.initFcn = 'initnw';
54
55          net.layers{k+nstates}.size = 4;
56          net.layers{k+nstates}.transferFcn = 'tansig';
57          net.layers{k+nstates}.initFcn = 'initnw';
58      end
59
60      %Set additional imag/real layer settings
61      net.layers{nstates+1}.size = 4;
62      net.layers{nstates+1}.transferFcn = 'tansig';
63      net.layers{nstates+1}.initFcn = 'initnw';
64
65      %Now before we can train we need to specify the initialization,
66      %training, peformance and dividing functions for the networks
67      net.adaptFcn='adaptwb';
68      net.initFcn = 'initlay';
69      net.performFcn = 'mse';
70      net.trainFcn = 'trainlm'; %
71      net.divideFcn = 'dividerand';  % Divide data randomly
72      net.divideMode = 'sample';  % Divide up every sample
73
74      %Divide the data so that there is some test and validation
75      %data from the set (see section 4.1.2)
76      net.divideParam.trainRatio = 50/100;
77      net.divideParam.valRatio = 25/100;
78      net.divideParam.testRatio = 25/100;
79
80      %Prepare the network with the example input dimensions and
81      %types. Next, set the example output of the network
82      %(i.e. the c(t+tau) vectors that will be trained against to
83      %initialize it.
84      fprintf('Configuring network for specific inputs and output\n');
```

```
85      net = configure(net,mapminmax(input),mapminmax(output));
86
87      %Now we just need to initialize the network
88      fprintf('Performing network initialization for training\n\n');
89      net = init(net);
90
91      %Set the training parameters so that a window isn't opened
92      net.trainParam.showWindow = false;
93      net.trainParam.showCommandLine = true;
94      net.trainParam.show= 1;
95      net.trainParam.max_fail = 10;
96      %Train for 50 time steps.
97      net.trainParam.epochs = 50;
98  end
```

Single Step Time Propagator pstep.m

```
1   function next = pstep(obj,previous,expenergy,occupied)
2   %PSTEP Simulates the next timestep using all the neural networks.
3   %PARAMETERS
4   %   previous: the previous step's combined complex input.
5   %   expenergy: the exp(i*energy) phase shifting terms.
6   %   occupied: a vector indicating which indices in the inputs
7   %   correspond to {psi_j^(1)} values close to 1.
8   %DESCRIPTION
9   %   The neural network for the j = 1 propagated state can only
10  %   predict the c—vector for the next timestep for the j = 1
11  %   state. Same goes for the other j = 2..nprop states. However,
12  %   in order to predict the next step, each neural network requires
13  %   the previous timesteps of ALL the propagated states (i.e. a
14  %   vector of nprop*nbasis length). This method simulates the next
15  %   timestep for each of the networks and then joins the results
16  %   into a new vector of length nprop*nbasis that represents the
17  %   input for the next timestep.
```

```
18
19      %Initialize the resulting vector to be the right length
20      approx = zeros(obj.DFT.nbasis,obj.DFT.nprop);
21
22      %Initialize variables for the previous values so we can
23      %separate the real and imaginary parts
24      aprevious = cell(2*obj.DFT.nprop,1);
25      for k=1:obj.DFT.nprop
26          %Perform the phase shift according to the equations in
27          %figure 11.
28          transf = obj.DFT.shiftm(previous{k})';
29          aprevious{k} = real(transf);
30          aprevious{k+obj.DFT.nprop} = imag(transf);
31      end
32
33      mprevious = mapminmax('apply',aprevious,obj.anormins);
34      %Step each of the networks forward in time.
35      for k=1:obj.DFT.nprop %1,2,3,etc.
36          %We need to perform a mapminmax on the input vectors
37          %to normalize their values to be of order 1 for all rows.
38          ntr = obj.trnets{k}(mprevious);
39          netrapprox = mapminmax('reverse',ntr, ...
40              obj.rnormalizations{k});
41          nti = obj.tinets{k}(mprevious);
42          netiapprox = mapminmax('reverse',nti, ...
43              obj.inormalizations{k});
44
45          %Get the complex network approximation
46          netapprox = netrapprox{1} + 1i*netiapprox{1};
47
48          %Finally, we need to add the mostly occupied states from
49          %the previous timestep to the corresponding ones in new
50          %timestep to undo the initial subtraction that was done.
```

```
51          occindex = occupied(k);
52          combrhs = previous{k};
53          combrhs(occindex) = expenergy(k)*combrhs(occindex);
54          approx(:,k) = netapprox + combrhs;
55      end
56
57      %This orthonormalization seemed to blow up on the smaller size
58      %scales (starting with 1e-4)
59      %orthon = obj.gramschmidt(approx);
60      orthon = approx;
61      next = cell(obj.DFT.nprop,1);
62
63      for k=1:obj.DFT.nprop
64          next{k} = orthon(:,k);
65      end
66  end
```

QVE Class

Expectation Value Calculator expectation.m

This is the programmatic implementation of equation 15. The dipole matrix function in the DFT class (referenced in this script) extracted the dipole matrix for methane from a text file.

```
1  function expect = expectation(obj, cdata)
2  %EXPECTATION Calculates the expectation value for the c-matrix.
3  %DESCRIPTION
4  %   The expectation value for a system is psi* z psi where z is
5  %   the dipole matrix for the molecule obtained from
6  %   DFT.dipolematrix()
7  %INPUTS
8  %   cdata: the complex cell array representing the psi
9  %   probability matrices for each of the {psi_j^(1)} states.
```

```
10 %OUTPUTS
11 %   expect: the expectation value <z>(t) as a function of time.
12
13     %Get the dipole matrix
14     dipole = obj.DFT.dipolematrix();
15     dsize = size(cdata{1});
16     expect = zeros(dsize(2),1);
17
18     for k=1:dsize(2)-1
19         zexpsum = 0;
20         for m=1:obj.DFT.nprop
21             zexpsum = zexpsum + cdata{m}(:,k)'*dipole*cdata{m}(:,k);
22         end
23         %Set the value for this timestep to the sum of the expectation
24         %values of the propagates states.
25         expect(k) = zexpsum;
26     end
27     expect = real(expect);
28 end
```

Multiple State Plotter plotstates.m

This code produced the flawed validation plots (included in Appendix B). Although these plots did show how closely the network could predict values for training data, the prediction was only for a single time step prediction horizon and masked the failure of the network to approximate U_{jm}.

```
1 function plotstates(obj,states,targets,maxsubplots, ...
2     dolegend,arrowplot,ts)
3 %PLOTSTATES Shows real, imag and abs validation plots
4 %PARAMETER
5 %   states: the network-predicted c_m^j values.
6 %   targets: the DFT calculated data that the network was trying
```

```
7  %   to approximate
8  %   maxsubplots: the max number of vertical plots in the figure.
9  %   dolegend: specifies whether the legend will be generated.
10 %   arrowplot: specified whether to produce a propagator plot.
11 %   ts: specifies the number of time steps in the data to plot.
12 %DESCRIPTION
13 %   The plotstate function visually compares the states and their
14 %   targets for each of the states that have values greater than
15 %   1e-15. This function performs those same plots repeatedly for
16 %   the real, imaginary and magnitude curves on the same figure.
17
18     oscil = cell(3,1);
19     plots = cell(3,1);
20     %oscil holds a flag indicating whether the basis state will
21     %be plotted or not. We need to determine how many of the
22     %states actually have values and only plot those.
23     for r=1:3
24         oscil{r} = zeros(obj.DFT.nbasis,1);
25
26         switch r
27             case 1
28                 lstates = real(states);
29             case 2
30                 lstates = imag(states);
31             case 3
32                 lstates = abs(states);
33         end
34
35         for k=1:obj.DFT.nbasis
36             if max(lstates(k,:)) > 1e-12
37                 oscil{r}(k) = 1;
38             end
39         end
```

```
40
41          %Determine how many plots and figures will be made.
42          %Plots has the indices of the basis states with values
43          %large enough to be plotted.
44          plots{r} = find(oscil{r});
45      end
46
47      if (nargin < 4)
48          maxsubplots = obj.maxsubplots;
49      end
50
51      plotcount = 0;
52      for r=1:3
53          rplot = length(plots{r});
54          if (rplot > plotcount && rplot <= maxsubplots)
55              plotcount = rplot;
56          else
57              plotcount = maxsubplots;
58          end
59      end
60
61      figure
62      for r=1:3
63          %Determine which part of the states is being graphed
64          switch r
65              case 1
66                  lstates = real(states);
67                  ltargets = real(targets);
68              case 2
69                  lstates = imag(states);
70                  ltargets = imag(targets);
71              case 3
72                  lstates = abs(states);
```

```
73                  ltargets = abs(targets);
74          end
75
76          %For this figure, create the relevant number of subplots
77          %and include the target data on the same graph if it is
78          %specified
79          for k=1:plotcount
80              %We can only plot the values if we have indices for
81              %that plot num.
82              if (length(plots{r}) >= k)
83                  subplot(plotcount,3,2*(k-1)+k+(r-1))
84                  dft = lstates(plots{r}(k),:);
85                  if nargin == 7
86                      tc = ts;
87                  else
88                      [tr, tc] = size(dft);
89                  end
90                  plot(1:tc,dft(1:tc),'r.')
91
92                  switch r
93                      case 1
94                          title(sprintf('Real Parts for m = %g', ....
95                              plots{r}(k)))
96                      case 2
97                          title(sprintf('Imag Parts for m = %g', ...
98                              plots{r}(k)))
99                      case 3
100                         title(sprintf('Abs Parts for m = %g', ...
101                             plots{r}(k)))
102                 end
103
104                 %Only plot the targets if they were provided
105                 if (nargin > 2 && ~arrowplot)
```

```
106                     hold on
107                     plot(ltargets(plots{r}(k),:), 'g-')
108                     hold off
109                 elseif (arrowplot)
110                     pts = ltargets(plots{r}(k),:);
111                     hold on
112                     for l=1:tc-1
113                         y1 = dft(l);
114                         y2 = pts(l);
115                         line([l, l+1],[y1,y2],'Color','Green', ...
116                             'LineWidth',2);
117                     end
118                     hold off
119                 end
120                 if (dolegend)
121                     legend('DFT Data', 'Network', ...
122                         'Location', 'SouthEast')
123                 end
124 
125                 xlabel('Time (fs)')
126                 ylabel('\bf{c^{j}_{m}}')
127             end
128         end
129     end
130 end
```

Appendix B: Visual Validation of $\{\psi_j^{(1)}\}$ States

The following series of plots allows visual validation of the network's prediction over training data. The following figures (22, 23, 24, and 25) were generated by evaluating the network's prediction for the entire training dataset. As discussed in section 3.3.4,

they provide a visual confirmation that the mean-squared errors reported numerically during the training correspond to an actual approximation of the target function. Figures have been produced for each of the j states in the system since typically each of them would have to be checked before a decision can be made about the network's performance. Section 4.3 highlighted a flaw in the network design that was not apparent by examining these plots. As such, in future network interations, they will be replaced by a propagator plot (for example, figure 19).

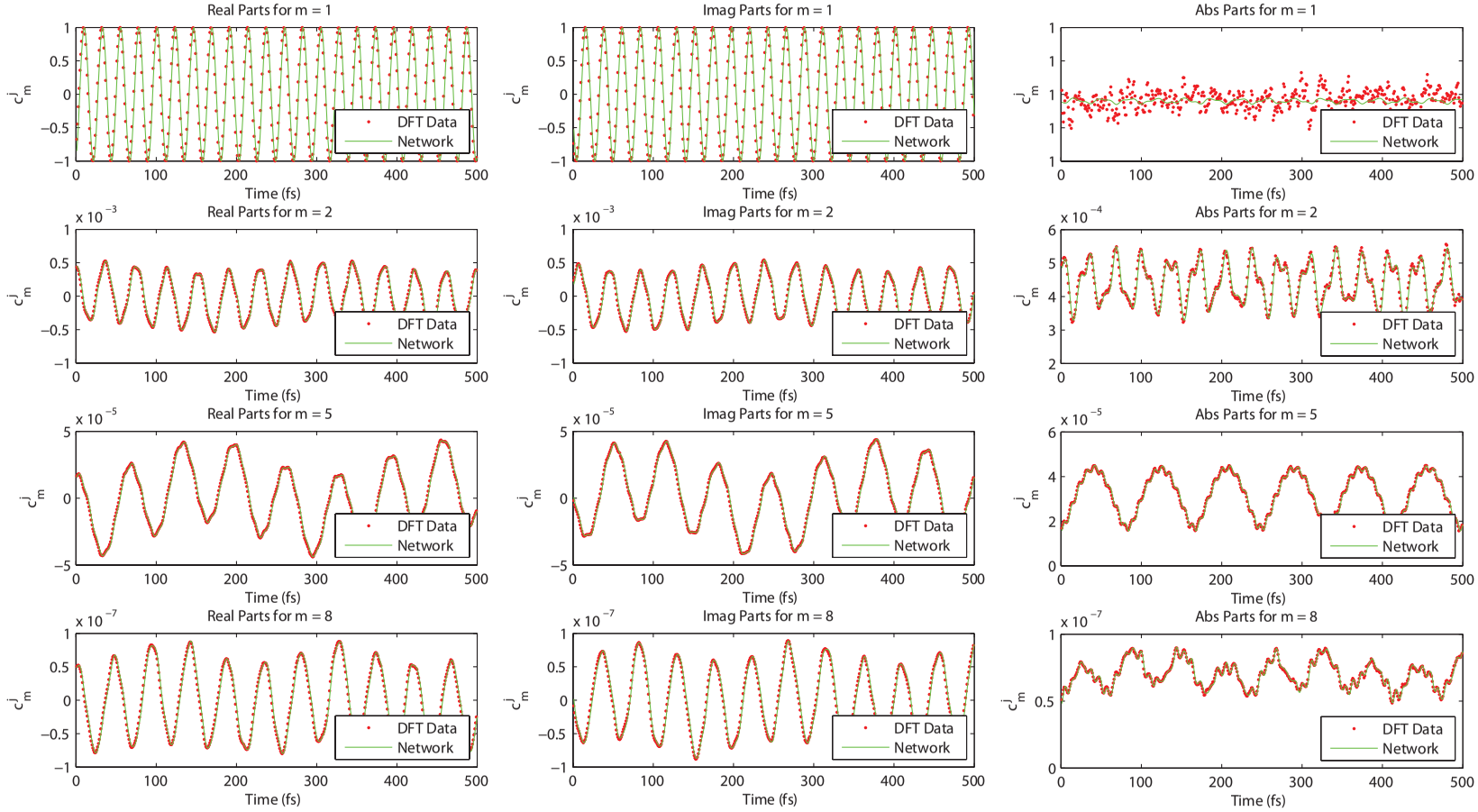

Figure 22: Network approximation of $\mathbf{c}_m^1$.

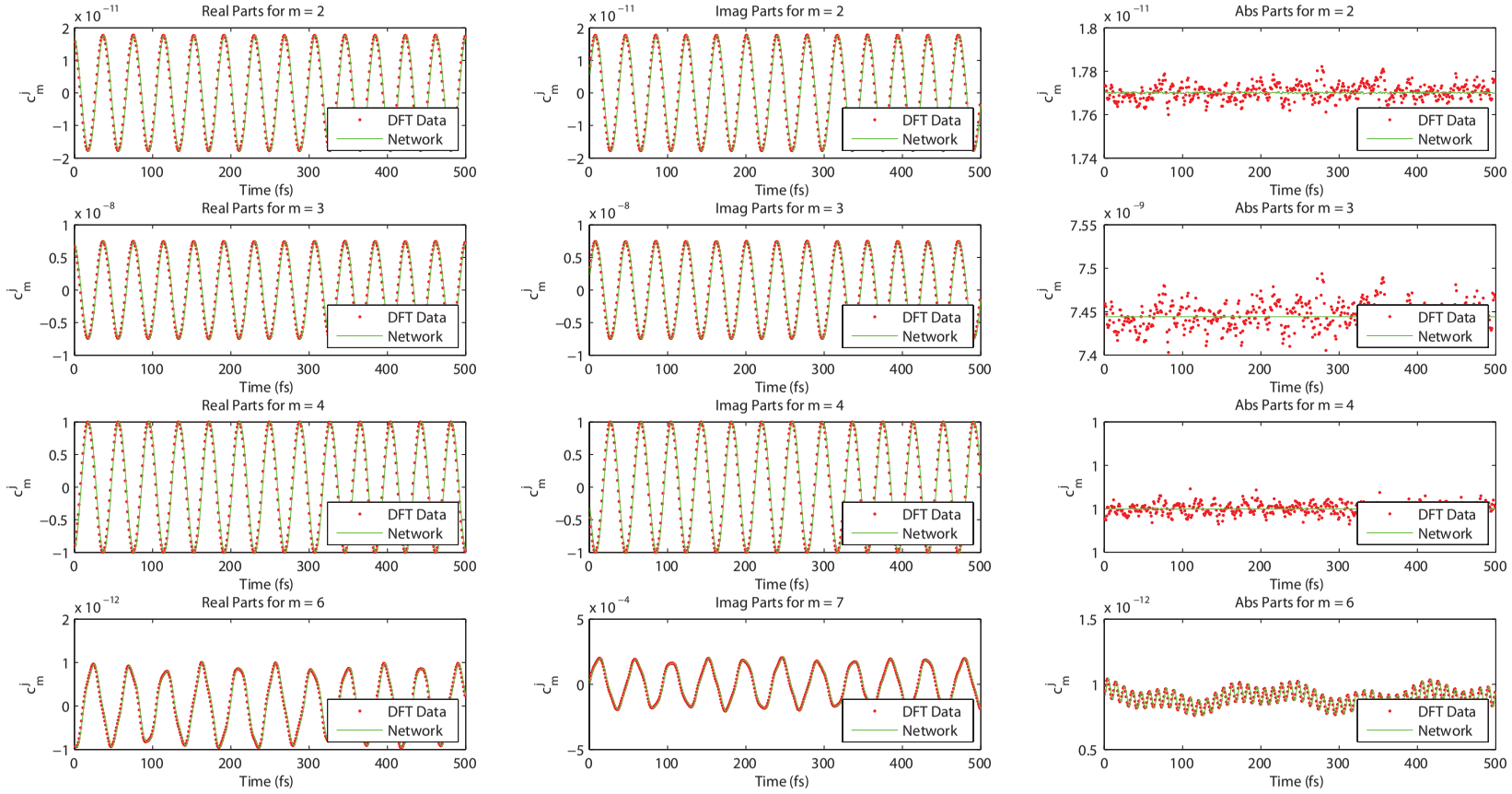

Figure 23: Network approximation of $\mathbf{c}_m^2$.

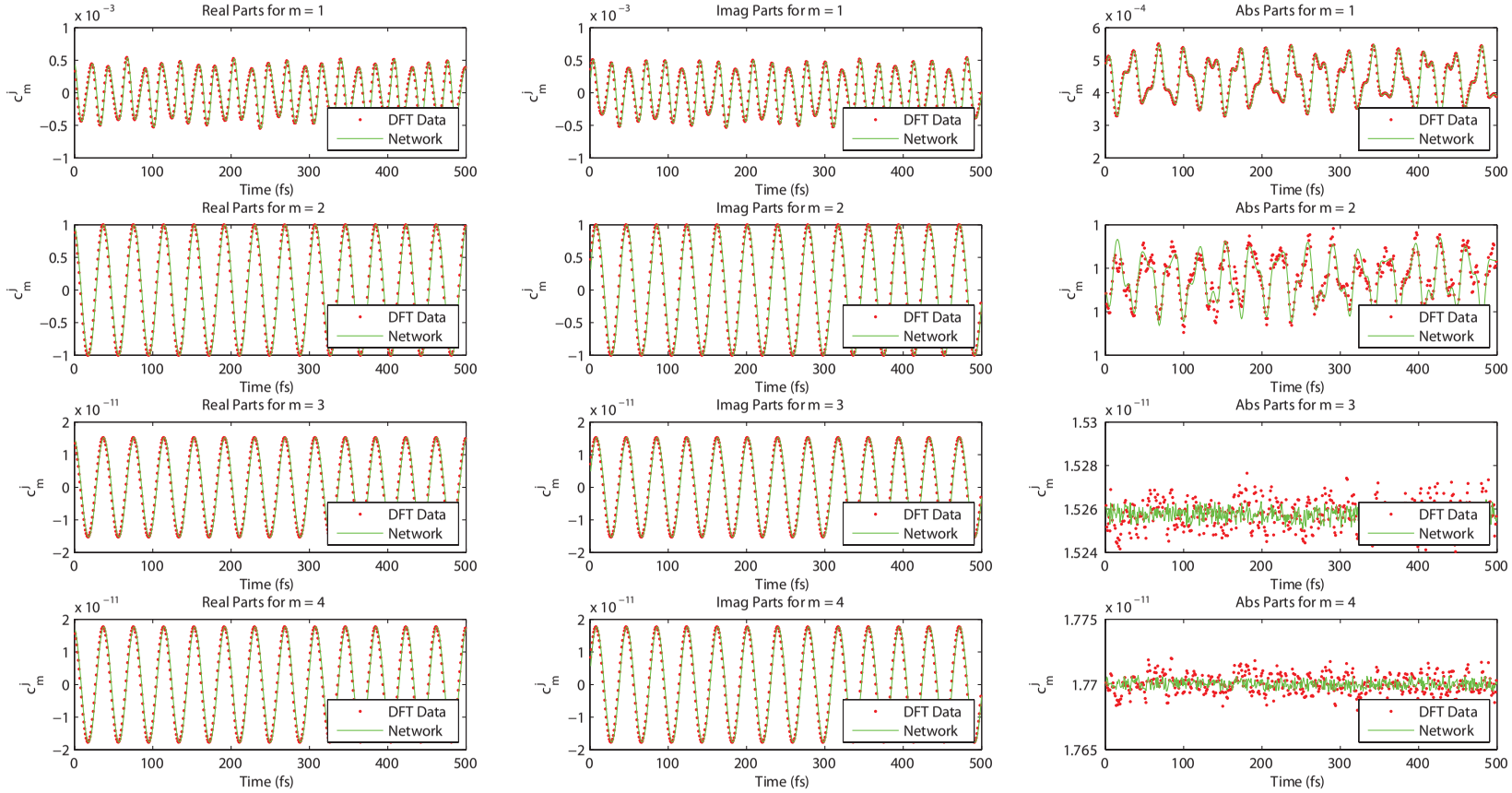

Figure 24: Network approximation of $\mathbf{c}_m^3$.

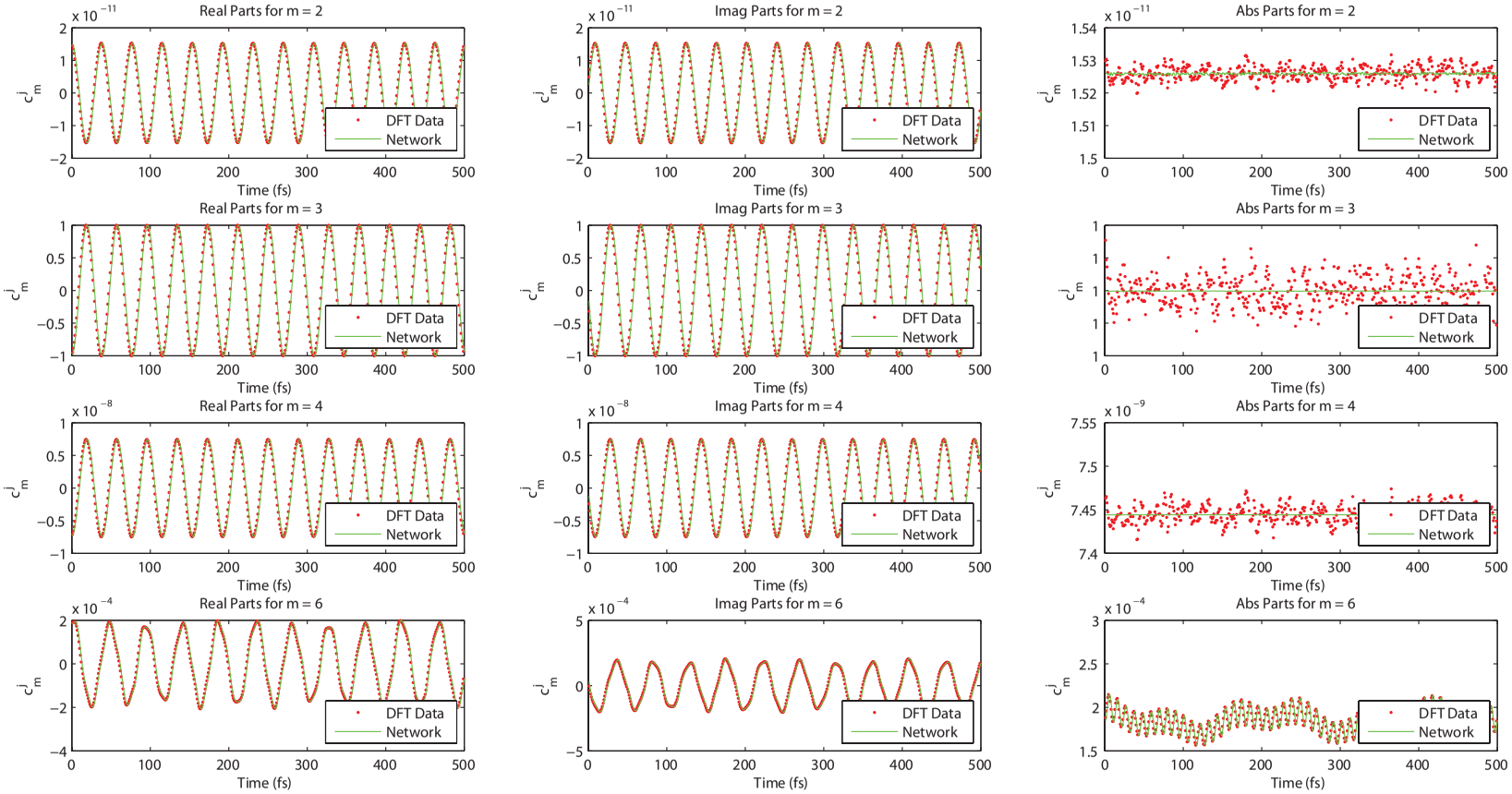

Figure 25: Network approximation of $\mathbf{c}_m^4$.

Index

Complex input approximation, 42

Complex value networks, 50

Dipole moment expectation, 24

Electric field perturbation, 22, 25, 28

Electron density functional, 20

Exchange correlation energy, 21

Kohn-Sham equation, 20, 22

Linear propagator
- approximations, 26
- defined, 23
- implementation, 24
- network representation of, 28, 30, 43, 46

Network implementation
- layers defined, 42

Network Training
- data, 17, 32, 40
- optimal neuron counts, 43

Neural Networks
- biases, 13
- convergence, 27, 50
- error function, 5, 15, 41
- functional representation, 9, 30
- general uses of, 5
- hidden layers, 9, 15, 16, 52
- initialization, 18
- input constraints, 27, 28
- notation, 9
- parameter adjustment, 12, 17
- performance, 18, 35, 39, 43, 52
- transfer function, 5, 15
- universal approximators, 13
- validation, 18, 34, 38, 45, 51
- weights, 5, 12, 15

Propagator plot, 46

Transformation of inputs/outputs, 28, 58, 60, 62