Investigation of Bond Strain Effects on XANES Spectra by Supervised Machine Learning

by
Jeremy K. Thaller

Professor Anatoly Frenkel, Advisor
Brookhaven National Laboratory
Chemistry Division
New York, USA

Professor Wolfgang Schmahl, Advisor
Ludwig-Maximilians-Universität
Fakultät Geowissenschaft
München, Germany

A Master thesis submitted to the Faculty of Earth- and Environmental Sciences
of Ludwig-Maximilians-Universität München in the framework of MaMaSELF

July 29, 2021

Abstract

A recently published method [1] enables the decoding of X-ray absorption near edge structure (XANES) spectra of nanoparticles to obtain important structural descriptors: coordination numbers and bond distances. Utilizing supervised machine learning (ML), the method trains an artificial neural network (ANN) to recognize a relationship between the nanoparticle structure and the XANES spectrum. Once trained, the ANN is used to "invert" an unknown spectrum to obtain the corresponding descriptors of the catalyst structure. Bond strain is known to be an important catalytic descriptor, yet, its accurate determination in reaction conditions is hampered by high temperature and low weight loading of real catalysts. ML–assisted XANES analysis offers a promising new direction for extracting the bond strain information from XANES—and not from extended x-ray absorption fine structure (EXAFS) analysis. Using simulated XANES spectra of Au nanoparticles, we have developed an ANN capable of "inverting" an unseen XANES spectrum and predicting structural disorder in the form of mean-squared displacement. The utility of the method was demonstrated on both the computer-simulated nanoparticles of different sizes and degrees of disorder, as well as on experimental data of disordered nanoparticles.

Executive Summary

Your executive summary will give a detailed summary of your thesis, hitting the high points and perhaps including a figure or two. This should have all of the important take-home messages; though details will of course be left for the thesis itself, here you should give enough detail for a reader to have a good idea of the content of the full document. Importantly, this summary should be able to stand alone, separate from the rest of the document, so although you will be emphasizing the key results of your work, you will probably also want to include a sentence or two of introduction and context for the work you have done.

Acknowledgments

The acknowledgment section is optional, but most theses will include one. Feel free to thank anyone who contributed to your effort if the mood strikes you. Inside jokes and small pieces of humor are fairly common here ...

Contents

List of Figures

Chapter 1

Introduction

Synchrotron radiation was first observed by General Electric in Schenectady, New York [2]. Initially just a side effect of particle accelerator experiments, it has since grown to be an important and powerful source of high-energy electromagnetic radiation for structural determination. Compared to lab-scale x-ray generation for diffraction experiments, arguably the most important benefit of synchrotron radiation is its high brilliance. Synchrotron radiation creates a highly collimated beam of photons characterized by small divergence and spatial coherence. Additionally, synchrotron radiation is tunable across a wide spectrum (microwaves to hard X-rays) and capable of high flux, useful for short time-scale-dependent experiments or weak scatterers. Synchrotron radiation can be produced in a pulsed structure. Importantly, the incoming photons are highly polarized, either linearly or circularly, depending on where the measurement system lies with respect to the plane of the synchrotron. The advent of a technique to manufacture such a high-quality source of x-rays allowed for new, advanced methods of x-ray absorption spectroscopy, and the two fields were developed in parallel.

1.1 X-ray Absorption Spectroscopy

X-ray absorption spectroscopy measures the absorption of high-energy photons by a sample as a function of energy [3]. The attenuation, or change in transmitted light intensity as a result of inelastic processes, is characterized by the Beer-Lambert Law (1.1). For an incident beam of intensity I_0, the transmitted intensity after interacting with an attenuation coefficient of μ and a sample of thickness x is:

$$I_t = I_0 e^{-\mu x} \tag{1.1}$$

Above the absorption edge, the condensed state has characteristic absorption jumps where the incident photon's energy matches the binding energy of a core electron. At this energy, nearly all the photon's energy is absorbed by the core electron, resulting in the characteristic absorption-edges first observed in 1920 [4, 5].

In experimental setups, particular energy photons are selected from the broad spectrum of synchrotron radiation via a pair of monochromators. The primary monochromator is a

crystal with interplanar spacing (d) chosen specify to satisfy the Bragg equation 1.2, reflecting photons of wavelength, λ at angle θ.

$$n\lambda = 2d\sin(\theta) \tag{1.2}$$

The secondary monochromator removes higher-order harmonics that satisfy the Bragg equation ($2\lambda, 3\lambda etc.$). Different wavelengths of light can be selected by changing the angle. In the XAFS setup, two ionization chambers are used to measure the incident and transmitted light intensity. For any study, the absorption spectrum for a reference sample is also measured to calibrate the energy scale.

1.1.1 XAFS

X-ray Absorption Fine-Structure (XAFS) spectroscopy refers to the study of absorption spectra created from high-intensity x-ray interactions. As the energy of the incident radiation increases, the photon's energy will eventually match the binding energy of a core-level electron. As a result, an "edge" in the spectrum will be observed. The location of these edges depends on the chemical and physical structure, as well as the electronic and vibrational states of the material. Absorption edges are like fingerprints used to identify elements, distinguish oxidation states, and even probe short-range order from the characteristic peaks and oscillations in the spectrum. XAFS spectroscopy can be performed on virtually any stable element since all atoms contain core-level electrons. Although a high-quality source of x-rays such as synchrotron radiation is required for the analysis, the ubiquity and utility of XAFS spectroscopy has made it an indispensable technique in fields such as materials biology, chemistry, and materials science [6] [7].

The XAFS equation is

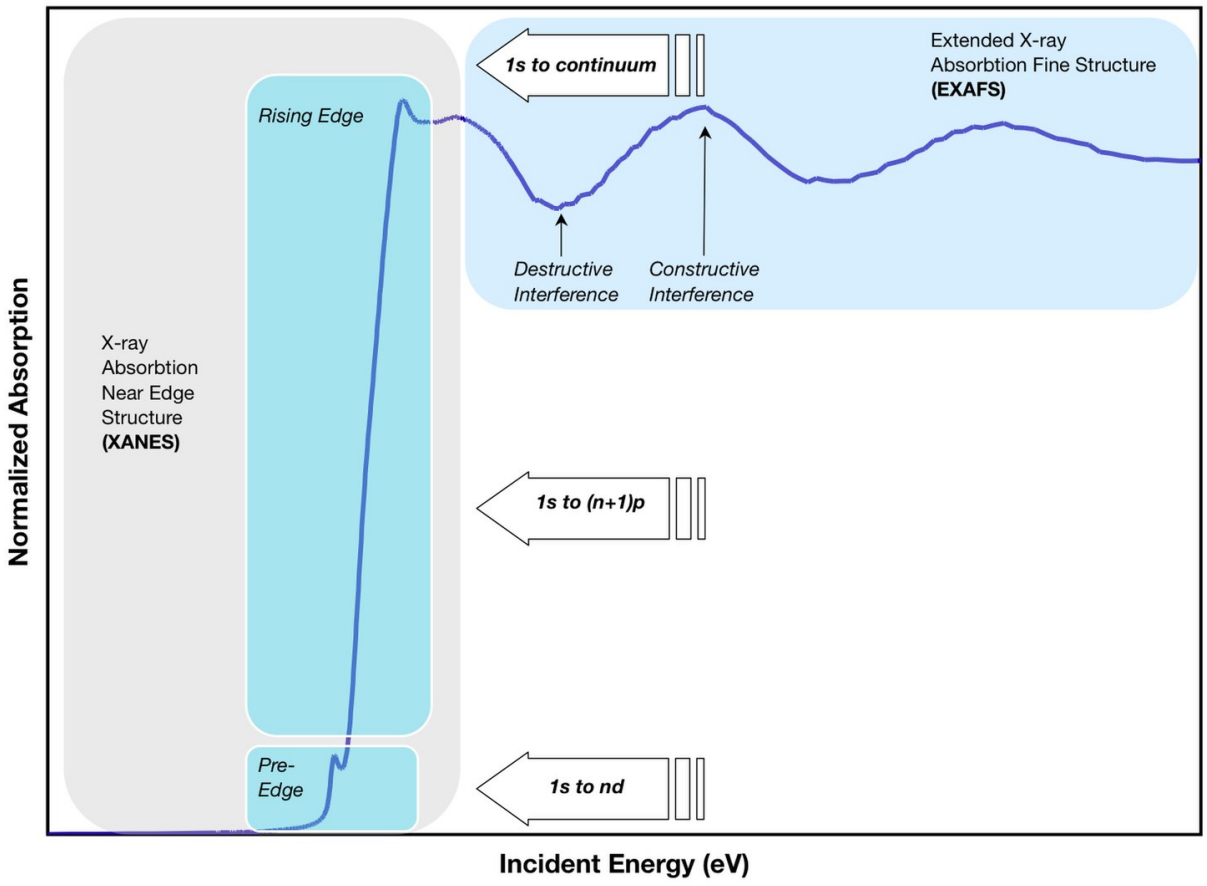
$$\chi = \frac{\mu(E) - \mu_0(E)}{\mu_0(E) - \mu_b(E)} \tag{1.3}$$

where μ is the measured absorption, μ_0 is the "atomic" absorption due to .... specific electrons, and μ_b is the absorption of other processes [8], typically approximated with the Victoreen polynomial (1.4).

$$\mu_b(E) = aE^{-3} + bE^{-4} \tag{1.4}$$

The coefficients α and β can be found via a simple regression on a sprectrum measured at pre-edge energies [8].

The XAFS spectrum is typically divided into two regions of study: the area near the first absorption peak (XANES) and the area after (EXAFS). XANES has a strong sensitivity to the oxidation state and coordination chemistry of the absorbing atom, while the EXAFS can be used to determine the bond lengths, coordination numbers, and atomic species of the absorbing atom's neighbors.


Normalized Absorption
Incident Energy (eV)
X-ray
Absorbtion
Near Edge
Structure
(XANES)
Rising Edge
Pre-
Edge
1s to continuum
1s to (n+1)p
1s to nd
Extended X-ray
Absorbtion Fine Structure
(EXAFS)
Destructive
Interference
Constructive
Interference

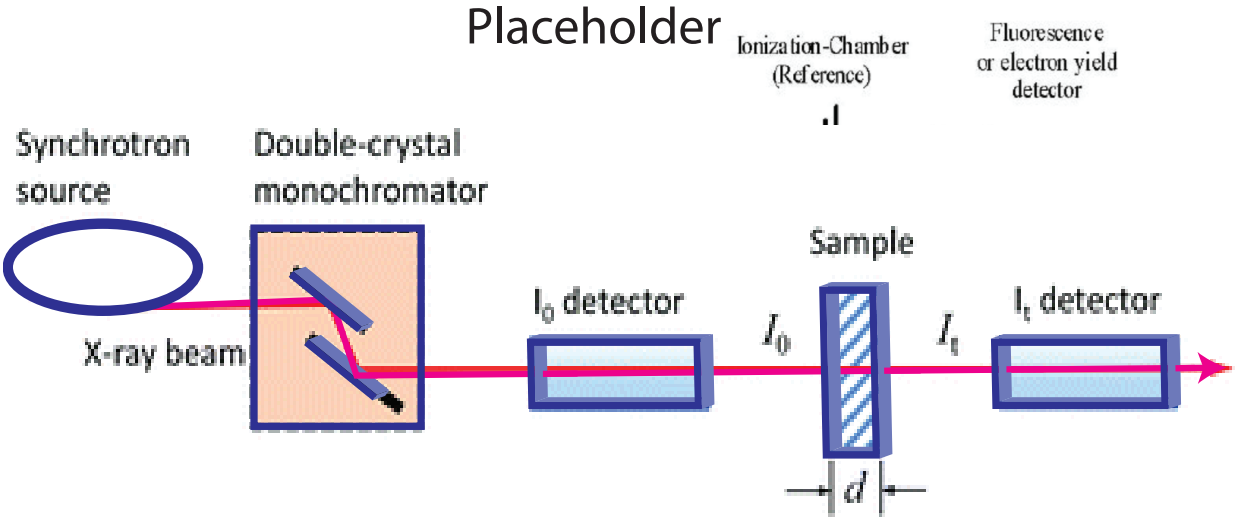


Figure 1.1: A simplified diagram for an x-ray absorption spectroscopy experiment.

1.1.2 EXAFS

Beyond the edge of the absorption spectrum lies the Extended X-ray Absorption Fine Structure (EXAFS) region. The spectral shape of this domain is determined by the multiple scattering of the photoelectron, interference of the incoming and outgoing waves of the photon, and electronic energy level splitting of the local structure. The oscillations in the EXAFS region are extremely sensitive to local bond lengths, coordination numbers, and atomic species of the surrounding elements.

1.1.3 XAS Experimental Setup

A diagram depicting a simplified experimental setup for an XAF experiment is presented in Figure 1.1. The high brilliance x-rays are produced from the synchrotron and then collimated through a monochromator. The beam is then split, with one half sent to an ionization chamber to measure the initial x-ray intensity, I_0 . The other half is sent towards the sample, where a second ionization chamber awaits the beam on the other side to measure the intensity of the transmitted beam, I_t.

EXAFS Data Reduction

Preparing the absorption data for fitting requires series of steps in preparation. These preparation steps are known as data reduction. First, the pre-edge background must be removed using the Victoreen formula (1.4) or an alternative polynomial. Next, the atomic

background, $\mu_0(E)$ is removed, and the absorption measured absorption is normalized accordingly. This is a non-trivial process, as the absorption coefficient is not that of a single, isolated atomic absorber; instead, it represents that of an atom and its surrounding neighbors. High-order spline fittings must be used to remove the low-frequency, immeasurable oscillations caused by photoelectron scattering with nearby valence electrons. For EXAFS Fitting, the EXAFS function is often transformed into k-space, and Fourier transformed into a radial distribution-like function via (1.5) [9].

$$\widetilde{\chi}(r) = \frac{1}{2\pi}\int_0^\infty k^n \chi(k) e^{2ikr} dk \tag{1.5}$$

Alternatively, one can convert the wave to R-space, which is what Artemis and Athena, two of the most common EXAFS fitting software, do.

The EXAFS Equation

With the data reduction complete, a multi-parameter fitting can be performed via the EXAFS equation. This equation is an approximation based principally on the many-body Fermi golden rule (1.6).

$$\mu(E) \propto \sum_f^{E_f > E_F} |\langle f|H_{int}|i\rangle|^2 \, \delta(E - E_F - Ef) \tag{1.6}$$

Fermi's golden rule describes the probability of a transition in state occurring. Here (1.6), $\mu(E)$ is the absorption coefficient, f and i are the final and initial states of the photoelectron, respectively. H_{int} is the matter-light interaction operator, and the delta function ensures conservation of energy. The summation over all energy values approximates the many-bodied problem as a single particle theory.

At energies above the core electron binding energy, excess energy is transferred to the photoelectron, which may then undergo multiple scattering with the surrounding atoms. The final wavefunction of the photoelectron is the superposition of the outgoing photoelectron and the scattered wave. EXAFS only takes into account the local region around the absorber at a distance r_j from the backscattering atom.

$$\chi(k) \approx F_j(k) \frac{\sin[2kr_j + \phi_{ij}(k)]}{2r_j^2} \tag{1.7}$$

The above equation (1.7) describes the EXAFS signal for only one backscattering atom. For a system with many atoms, one must sum over all the backscattered waves. To account for the inevitable variation in bond lengths, the Debye-Waller factor σ_j is introduced; this describes the standard deviation in bond-length of the sample.[1]. The lifetime of the photoelectron's excited state is taken into account by introducing an exponential term to account

[1]Note, this differs from the Debye-Waller factor used for x-ray absorption, which describes the broadening of a diffraction peak due to variations in inter-planar spacing [10]

for the mean-free-path, $e^{-2r_j/\lambda}$. Combining all this information with an amplitude correction factor of S_0, we arrive at the EXAFS equation (1.8.

$$\chi(k) = \sum_j \frac{N_j}{kr_j^2} F_j(k) e^{-2\sigma_j^2 k^2} e^{-\frac{2r_j}{\lambda(k)}} \sin[2kr_j + \phi_{ij}(k)] \tag{1.8}$$

Although popular, the EXAFS equation is still a first-order approximation with assumptions and limitations. For example, the fitted disorder parameter, the Debye-Waller factor, explicitly assumes a Gaussian distribution of nearest-neighbor bond length distances. Other more accurate but computationally intensive alternatives are increasing in popularity. Such alternatives include molecular dynamics, reverse Monte Carlo simulations and neural networks [11] [12].

1.1.4 XANES

The XANES region, or the near-edge region, encodes the chemical and electronic structural information of the sample within its shape. The XANES shape reflects the lower energy photons which scatter much more strongly than in the EXAFS region. XANES also has the advantage of a higher signal-to-noise ratio than the subtle interference-determined oscillations found in EXAFS. While there is an "EXAFS Equation," there is no "XANES Equation" equivalent; however, this does not mean that there is no structural information encoded in the XANES, only that the theory is underdeveloped. Mainly, the strong errors in potential and many-body effects limit the development of a "XANES equation."

With the recent explosion in the popularity of machine learning, the investigation of the XANES latent space has become a topic of modern research is. In other words, how much information is encoded in XANES?

Recent work at Brookhaven National Laboratory [1] has shown that a model can learn structural descriptors from the XANES spectrum. Specifically, their method enables the decoding of XANES spectra to obtain the coordination number of metallic nanoparticles. In this 2017 paper, the group trained an artificial neural network (ANN) to recognize a relationship between the nanoparticle structure and the XANES spectrum. Once trained, the ANN is used to "invert" an unknown spectrum to obtain the corresponding structural descriptors of the catalyst. These descriptors, the coordination numbers, are used to calculate the number of shells (nanoparticle size) and shape (Archimedian solid) of the sample. While this model can determine the structure of nanoparticles from XANES —a feat previously only possible with the full EXAFS spectrum —it still has one major limitation: the ANN does not predict the disorder of the structure. The bond-length disorder is known to be an important descriptor for catalyst [13] [14].

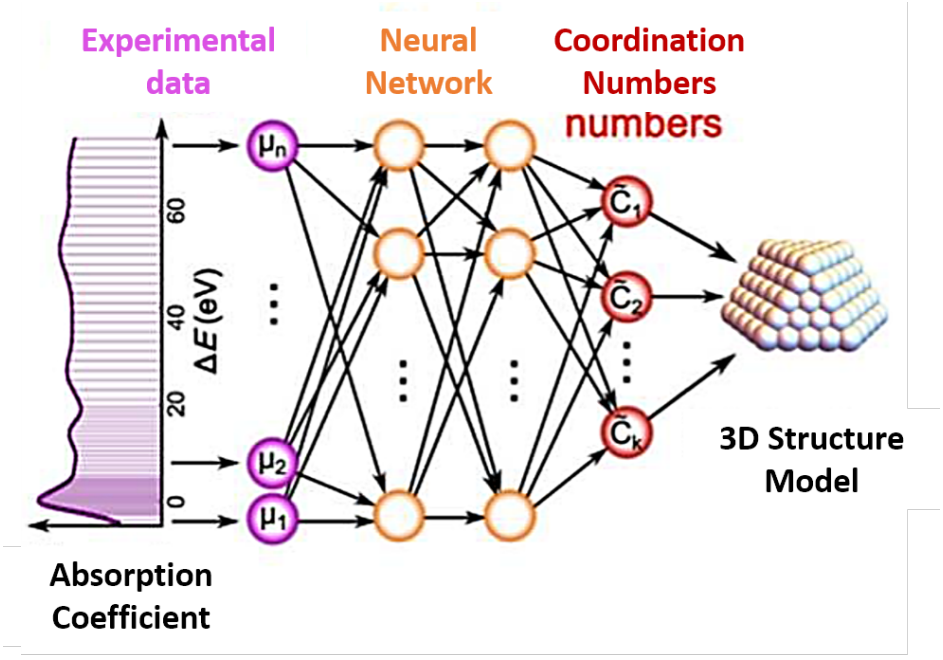


Figure 1.2: From [1], the neural netowrk is trained to take a XANES spectrum from a metallic nanoparticle, and predict the coordination number of the structure. This coordination number of nanoparticles is a known discriptor, which allows for easy calculation of the the nanoparticle's size and shape.

1.2 Goals of the Thesis and Approach

Building of the previous work [1], the goal of this thesis is comprised of two parts: first, determine whether bond-length information is encoded in the XANES spectrum; and second, utilize machine learning to predict the bond-length disorder of metallic nanoparticles from a XANES spectrum. As with the 2017 paper [1], the work is conducted with gold (Au) nanoparticles, in part to expand on the previous paper, and in part due to the access to the relevant experimental data. Machine learning requires a substantial amount of training data, far more than could be experimentally obtained. Consequently, we will rely on absorption simulation software to create the training data, a collection of XANES spectra for Au nanoparticles with known disorder. Finally, once the network is trained, the network must be modified to abstract to experimental data to compenstate for the systematic differences between the simulation and experimental data.

1.3 Outline of the Thesis

Often the most time-intensive part of any machine learning-based project is the process of collecting and preprocessing the data. Chapter 2 is dedicated entirely to the process of generating XANES spectra via simulations. Next, a solid foundational understanding of machine learning is important in understanding the approach. All machine learning terms present in later chapters are defined here. Chapter 4 describes the exact model architecture and results of the training process. Further discussions and future work are included in chapter 5. Appendix A includes a description of the main Python scripts written for this thesis, which are necessary for replicating the work. The file structure and some individual functions are included to clarify the calculation or creation of certain parameters and files. The files and scripts can be obtained upon request by contacting the author.

Chapter 2

Simulating Disorder

Before making any predictions, neural networks must first be trained on a large quantity of data. Specifically, to teach our neural network to predict the mean squared displacement (MSD), we must first generate a large quantity of training data comprised of XANES spectra, each labeled with a known MSD. Gathering such a large quantity of high-quality experimental data would be impractically time-intensitive and expensive. Rather, simulations provide a practical alternative, though even simulating each possible disordered structure individually, would be extremely time-intensive. This process is discussed in section 2.1. First, a discussion on the development of a new method for simulating disordered nanoparticles is presented in sections 2.3.1–2.3.3. The new process utilizes the statistical averaging of non-disordered structures. Instead of simulating hundreds of defined, disordered structures, we run many XANES simulations of simple, non-disordered structures and generate the disordered spectra via clever statistical averaging. In this chapter, we explain this statistical weighting process in-depth, beginning with the creation of simple, non-disordered spectra for the FEFF input files and culminating in the creation of many possible disordered spectra with known MSDs. The efficacy and limitations of this approach are discussed in section 2.4.

2.1 Traditional Particle-Averaged Simulations

The traditional method for running FEFF simulations is referred to as particle-averaged FEFF. The simulation software only simulates the absorption spectrum from one absorber at a time. In the skew-norm methodology, we only calculate a center absorber because of symmetry. For simulating disordered structures, however, it is necessary to calculate all the absorbers. For this project, we are interested, at first, in simulating bulk Au. Accordingly, we attempt to neglect the surface effects on the spectrum by simulating the absorption of only the first shell atoms and averaging (arithmetic mean) the results. This way, the first shell absorber atoms are surrounded by bulk structure and the potentials and multiple scattering in FEFF will take more bulk structure into account instead of more surface effects. The decision to simulate bulk was made to reduce the number of confounding variables; if the technique works for simulating disordered bulk structures, the next stage would be to expand

the process to nanoparticles.

Simulating absorption spectra via FEFF requires the creation of Feff input files, which include various user-defined parameters as well as the Euclidean coordinates of the structure. Determining how to create these structures is a project in and of itself. The approach taken for this experiment was to start with the perfectly ordered crystal of Au atoms. Then, for each structure, each individual atom is shifted by a random distance in a random direction. For each atom (in a given structure), the direction to be shifted is chosen from a uniform distribution, whereas the distance by which to shift the atom from its original location is chosen from a Gaussian. For each structure, the standard deviation of the gaussian from which the shifted distance is drawn differs. Thus, with a narrow width gaussian, the shifted distances on average tend to be smaller than for a gaussian with a large width. Thus, structures with small MSD's tend to be created when the width of the gaussian for shift distances is narrow, whereas high MSD structures tend to be created when the width of the shift gaussian is wide. The code for this script can be found in Appendix A.3.

FEFF requires tuning. There are various debates about choosing potentials blah blah blah

Each FEFF input file is run with the following parameters:

```
1  SCF 4.6 0 30 .5 1
2  EDGE    L3
3  EXCHANGE    5   0.2 0.5
4  S02 1.
5  XANES   3.7 0.05    0.1
6  FMS 7
7
8  POTENTIALS
9  0        79        Au        -1        -1        0.
10 1        79        Au        -1        -1        0.
```

While particle averaging is the traditional method for simulating XANES via FEFF, the method is computationally intensive for large structures with many absorbers. The goal of developing a skew-norm averaging method was to be able to create an infinite number of disordered XANES spectra from a limited number of simulations. For particle averaging, to create 1000 disordered structures requires 13,000 simulations, a process that necessitates several days of computation time on a distributed computing cluster.

2.2 FEFF Simulations here?

Not sure if this should go in this section. Maybe be relegated to the appendix? Basically just recap the intro of this article by them https://iopscience.iop.org/article/10.1088/1742-6596/430/1/012001/pdf and that lecture on FEFF [15] [16] [17].

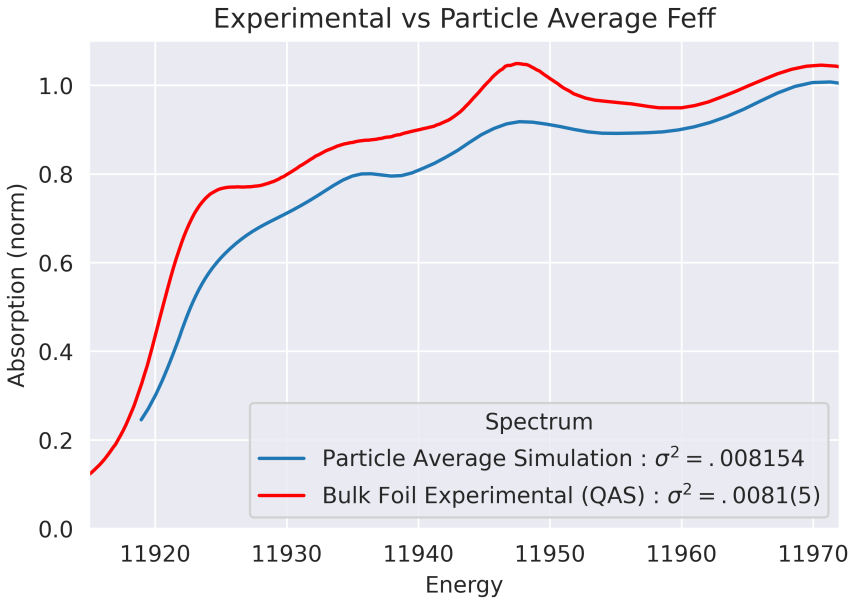


Figure 2.1: Comparing the bulk foil (red) measurement to a simulated large, bulk-like nanoparticle with the same disorder. The FEFF-

2.3 Statistical-Averaged Simulations

2.3.1 Generating Distortion Not Disorder

Instead of creating structures with a range of *disorder*, we instead begin by generating structures with a range of *distortion*. Wheres *disorder* refers to a statistical average of atomic displacement from their original position, characterized by MSD and the width σ^2 of a partial radial distribution function, *distortion* refers only to isotropic expansion or contraction of the subject. Equivalently, we define distortion as a radial shift in all atomic positions away from (or towards) the center atomic absorber.

A 2-dimensional projection of this isotropic distortion is presented in Figure 2.2. Though the figure only shows the xy-plane projection of the first 12 nearest neighbors, the actual structure used consists of either 55 atoms to simulate a nanoparticle, or the first four shells (561 atoms) to simulate bulk materials. Both structures were created with a lattice constant of 4.0782 Å to match that of bulk Au.*Citation? Wolfram Element Data?* In reality, the nearest-neighbor distances for Au nanoparticles are likely smaller *gold-lattice-const*; this can be accounted for later on in the averaging process since the original coordinates will only be one structure out of many. The important part is that the crystal structure is correct.

We generate a total of 91 FEFF input files with different levels of distortion. Each file contains the same center absorber located at $(0, 0, 0)$, all the atoms expanded or contracted radially. All the first shell atomic coordinates are shifted on the range of -0.45 Å to $+0.45$ Å in increments of 0.01 Å. For example, the FEFF input file with the greatest inward shift has

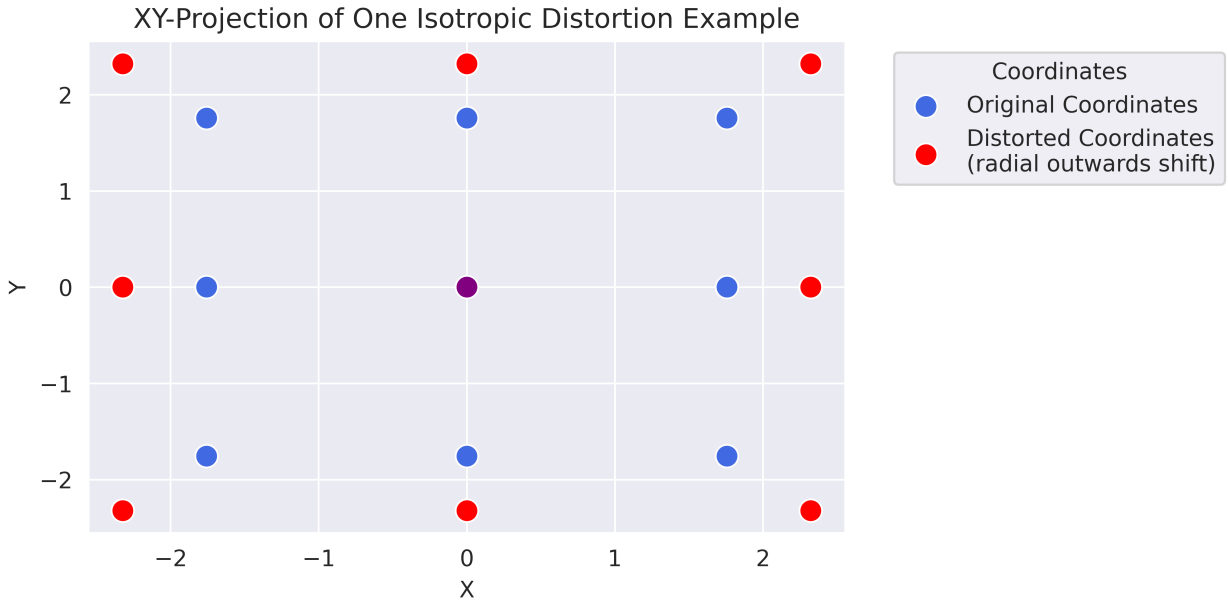


Figure 2.2: Each point represents an atom of first 12 nearest neighbors of a Au cluster projected onto the xy-plane. The four corner points actually represent two atoms because of the projection. The blue atoms represent the original coordinates, and the red atoms represent the radially shifted coordinates. The center absorber atom is purple since its original position is the same as its distorted position.

all first nearest neighbor atoms shifted 0.45 Å radially inwards towards the center absorber, and the FEFF input file with the largest outwards shift has the first nearest neighbor coordinates shifted 0.45 Å radially outwards away from the center absorber. The atoms in the outer shells are scaled accordingly to preserve the crystal structure according to equation 2.1

$$\boldsymbol{\rho}^{(i)}_{shifted} = \boldsymbol{\rho}^{(i)}_{\mathbf{0}} \left(\frac{a+\delta}{a} \right) \tag{2.1}$$

where $\boldsymbol{\rho}^{(i)}_{0}$ is the vector from the center absorber to the original (lattice) position of atom i, a is the lattice constant of Au, and $\delta \in [-0.45,\ 0.45]$ and is the distance the atoms in the first shell will be shifted.
Running the 91 simulations (one for each of the distorted structures) takes approximately 30 minutes. Were we to generate thousands more or employ RMC or MD, this process could take days or weeks of computation time. We plot the resulting XANES spectra from the FEFF simulations in figure 2.3.

2.3.2 Generating Disorder via Probability Distribution Averaging

One way to characterize system disorder is with the Gaussian width, σ, of the partial radial distribution function. The idea of our statistical averaging method is to emulate this width

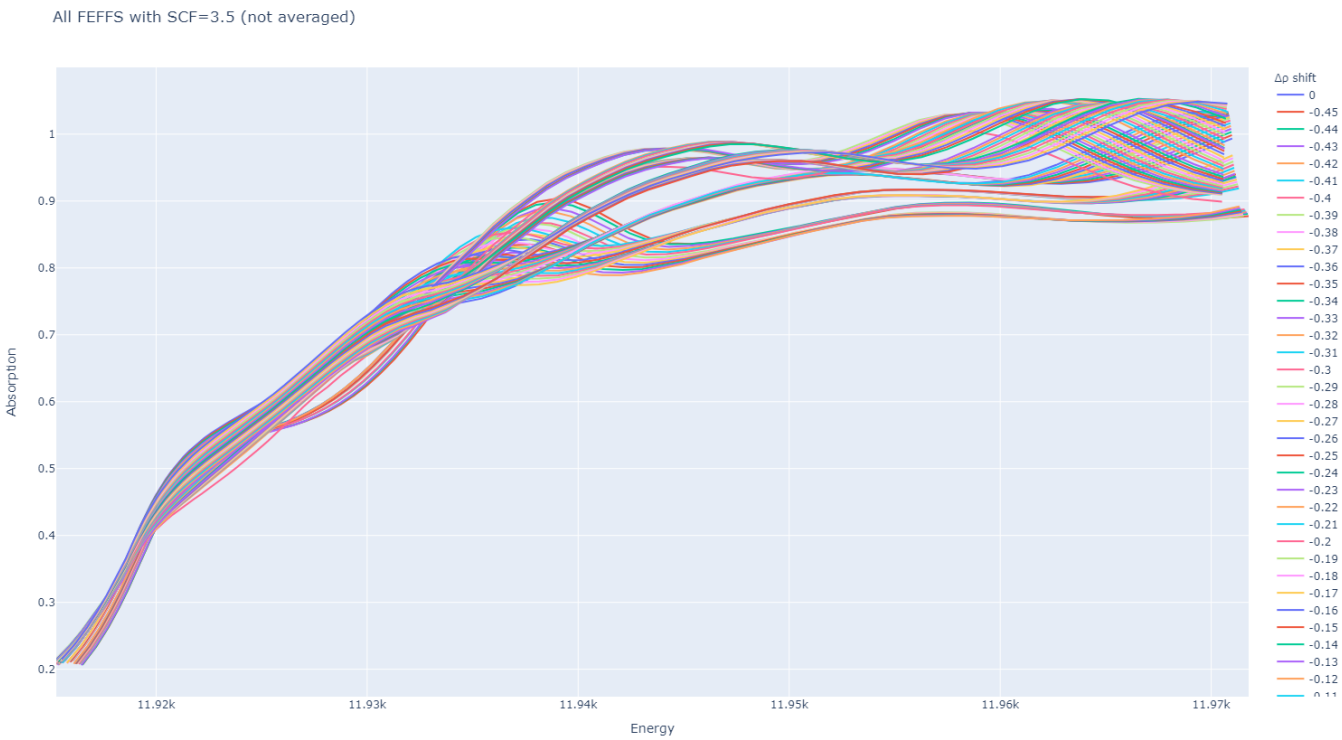


Figure 2.3: *TEMPORARY - way too much info. I'll select a few.* Each spectrum represents the FEFF simulation results for a different distorted structure. For each spectrum, the crystal structure and center absorber remain constant, the only parameter that varies is the euclidean distance from the center to the other coordinates.

by weighting the simulated XANES spectra according to this distribution. For example, Figure 2.4 depicts a histogram with $\sigma = 0.1$ Å. Each histogram bin represents a simulated XANES spectrum with a different isotropic displacement. For example, the bin at $\Delta\rho = 0.0$ Å represents the simulated XANES spectrum with no distortion, and the bin at $\Delta\rho = -0.2$ Å represents the simulated XANES spectrum with all the atomic coordinates shifted isotropically inwards towards the center absorber by 0.2 Å. The height of each bin, $f(\Delta\rho)$, represents the relative contribution of each simulated XANES spectrum towards the resulting weighted spectrum. For visual clarity, Figure 2.4 depicts only 40 bins; the actual weighting includes 91 bins ranging from -0.45 Å to $+0.45$ Å.

The disordered, gaussian-averaged XANES spectrum, $\langle\mu(E)\rangle$, using the histogram weighting of the gaussian in Figure 2.4 is calculate via Equation (2.2):

$$\langle\mu(E)\rangle = \frac{1}{S}\sum_{\Delta\rho=-.45}^{+.45} g\left(\Delta\rho \mid \mu = 0, \sigma = 0.1\right)\mu(E \mid \Delta\rho) \tag{2.2}$$

In the above equation, $\Delta\rho$ is the isotropic, radial displacement of each atom from its original

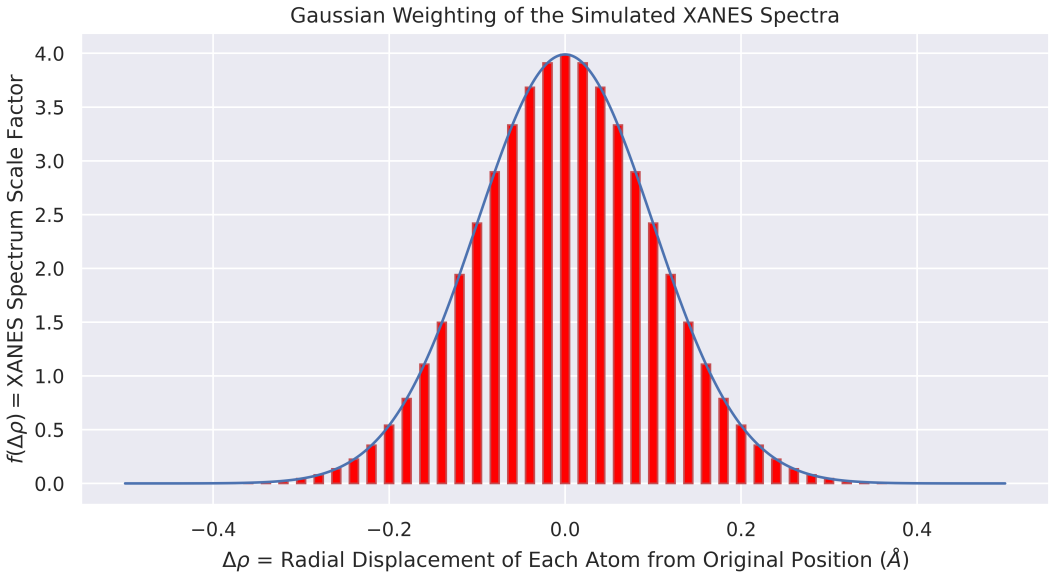


Figure 2.4: A Gaussian distribution probability density function can be used to calculate the relative weight of each FEFF generated XANES spectrum towards one simulated, disordered spectrum. Each bin (red bar) represents a FEFF generated spectrum; the x-axis is the isotropic shift of the atomic positions, and the y-axis is the relative weight factor.

position, and $\mu(E \mid \Delta\rho)$ is the simulated FEFF spectrum for the given $\Delta\rho$ configuration. Furthermore, in Equation (2.2), S represents a standardization factor needed to negate the effect of the changing Gaussian height as a function of the variance, σ^2. With the inclusion of S , only the relative heights of each bin matters for producing the averaged XANES spectrum. This standardization factor is defined in Eqation (2.3):

$$S = \sum_{\Delta\rho=-.45}^{+.45} g\left(\Delta\rho \mid \mu = 0, \sigma = 0.01\right) \tag{2.3}$$

In both equations (2.2) and (2.3), the function g is just the typical Gaussian distribution probability density function (Equation 2.4):

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{2.4}$$

The above example only generates one (simulated) disordered XANES spectrum and does so via weighting of a Gaussian distribution with mean and variance equal to 0 and 0.01, respectively. To simulate systems with different degrees of disorder, we can vary the shape of the probability density function. With a Gaussian distribution, we can only vary the mean and variance; to simulate even more conditions, however, we can instead use the multivariate skew-normal distribution (2.5) [18] [19], *f(x).*

$$f(x) = 2\phi(x)\Phi(\alpha x) \tag{2.5}$$

where $\phi(x)$ is the Gaussian PDF:

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \tag{2.6}$$

and $\Phi(x)$ is the Gaussian CDF:

$$\Phi(x) = \int_{-\infty}^{x} \phi(t)\ dt \tag{2.7}$$

Equation (2.5) includes the shape parameter, α, which has the nice property of producing a right-skewed distribution when positive and a left-skewed distribution when negative. When $\alpha = 0$, the distribution simply produces the typical Gaussian distribution (eq. 2.4). Utilizing equation (2.5), we can vary μ, σ, and α to alter the first four moments of the function: mean, standard deviation, skew, and kurtosis. Eighteen possible skew-norm weighting functions are plotted in Figure 2.5. 1000 unique combinations of weightings are used to produce the neural network's training data.

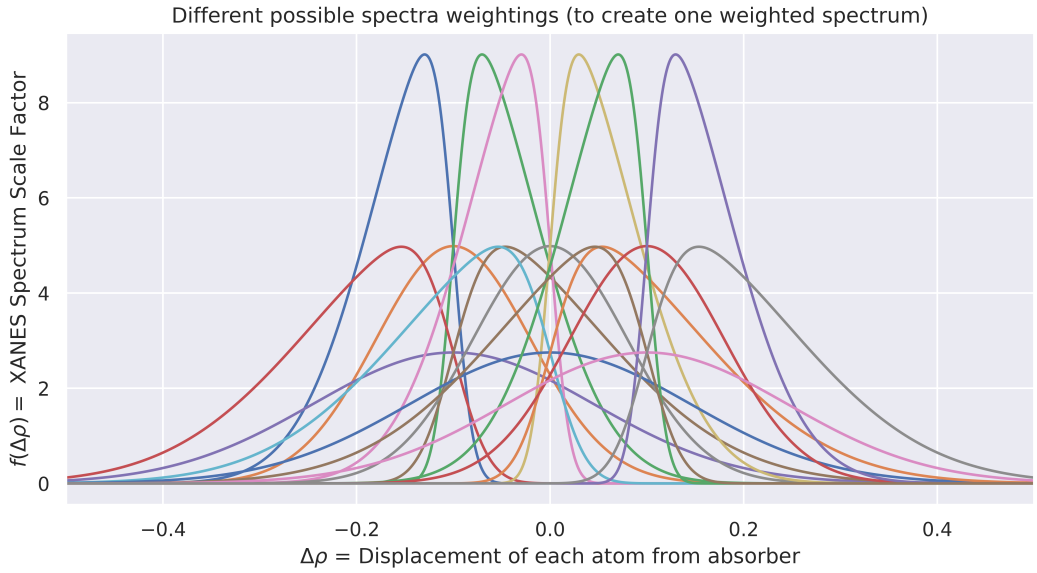


Figure 2.5: Eighteen skew-norm distributions plotted with all possible combinations of $\sigma \in \{.08, .145\}$, $\mu \in \{-.1, 0, .1\}$, and $\alpha \in \{-5, 0, 5\}$. Each represents a possible way to produce a simulated, disordered spectrum from many FEFF-simulated, distorted spectra

The disorder of the skew-norm generated, disordered spectrum is characterized by the mean squared displacement of each atom from its original position ($\Delta\rho$), weighted in the

same manner as the spectra. *Instead of characterizing the disordered spectrum by the standard deviation of the gaussian used to create it.* The weighted mean squared displacement, MSD, is calculated via equation (2.8):

$$S = \sum_{\Delta\rho=-.45}^{+.45} f\left(\Delta\rho \mid \mu, \sigma^2, \alpha\right) \tag{2.8}$$

$$\mu_{weighted} = \frac{1}{S} \sum_{\Delta\rho=-.45}^{+.45} \Delta\rho f\left(\Delta\rho \mid \mu, \sigma^2, \alpha\right) \tag{2.9}$$

$$MSD = \frac{1}{S} \sum_{\Delta\rho=-.45}^{+.45} \Delta\rho \left(f(\Delta\rho \mid \mu, \sigma^2, \alpha) - \mu_{weighted}\right)^2 \tag{2.10}$$

Here, $f(x)$ is the skew-norm function from equation (2.5). The code for this equation can be found in Appendix A.2, written in Python and optimized with NumPy [20].

2.3.3 Simulation vs. Experimental Data

To check our FEFF simulation parameters, as well as the validity of the gaussian-weighted disorder technique, we compare the simulation data to experimental data citeau-nanowires-silca-wang, [21] [22] [23]. In Figure 2.6, both experimental and simulation spectra for bulk-like and nanoparticle scenarios are plotted. EXAFS fitting was used to characterize the disorder in the experimental measurements. For the bulk foil, this parameter was found to be $\sigma^2 = 0.0081(5)$ Å^2, and for the 8 nm disordered particle, $\sigma^2 = 0.0102(8)$ Å^2. One simulated disordered spectrum was weighted according to the gaussian $N(0, 0.09)$ to represent the disordered nanoparticle, and the other was weighted according to the gaussian $N(0, 0.038)$ to represent the bulk. These weightings correspond to MSD values that match the measured σ^2 values for the experimental data.

In Figure 2.6, the bulk Au foil spectrum is above the 8 nm nanoparticle spectrum (more absorption) until the peak around 11937 eV, where the NP absorption becomes higher. The two criss-cross again over the next two peaks, changing which material has the higher absorbance in an energy range. This change is more easily seen in Figure 2.7, which plots the difference between the the bulk material and the nanoparticle absorption for both the experimental measurements and the simulations. The experimental and simulation difference-spectra follow the same trend with the exception of the peak around 11947 eV.

Figures 2.6 and 2.7 aren't meant to be perfect comparisons of simulations vs. experimental data. For one, the experimental data compares a bulk spectrum to a nanoparticle. By contrast, both the simulation spectra are of the same size 55 atom cluster. Still, much of the disorder trends are coded in the simulation approach.

To test if the size information is also coded in our simulations, we compare different size simulations to experimental data in Figure 2.8. As expected, including more atoms in the simulation produces more bulk-like spectrum characteristics, such as larger amplitude peaks.

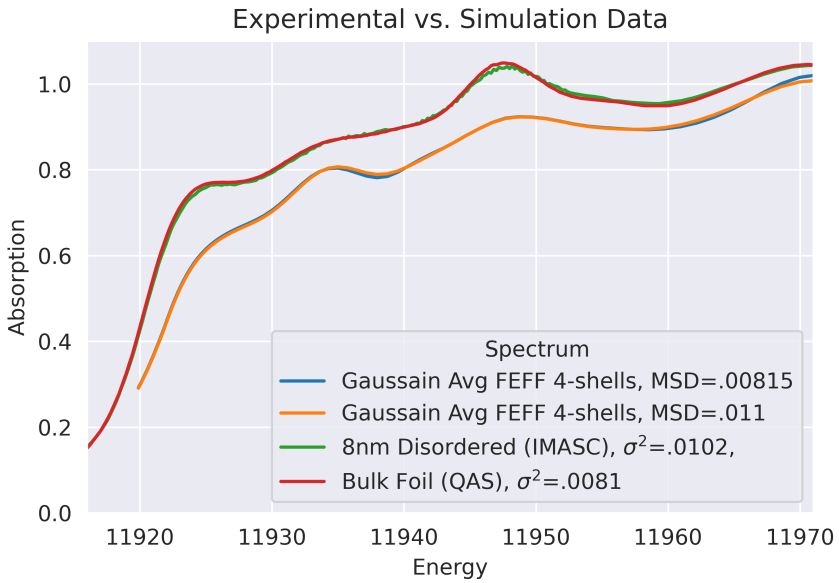


Figure 2.6: Comparing the bulk foil (red) measurement to the 8 nm disordered nanoparticle (green) measurement is an analog to comparing the simulated, non disordered FEFF spectrum (blue) to the simulated disordered spectrum (orange).

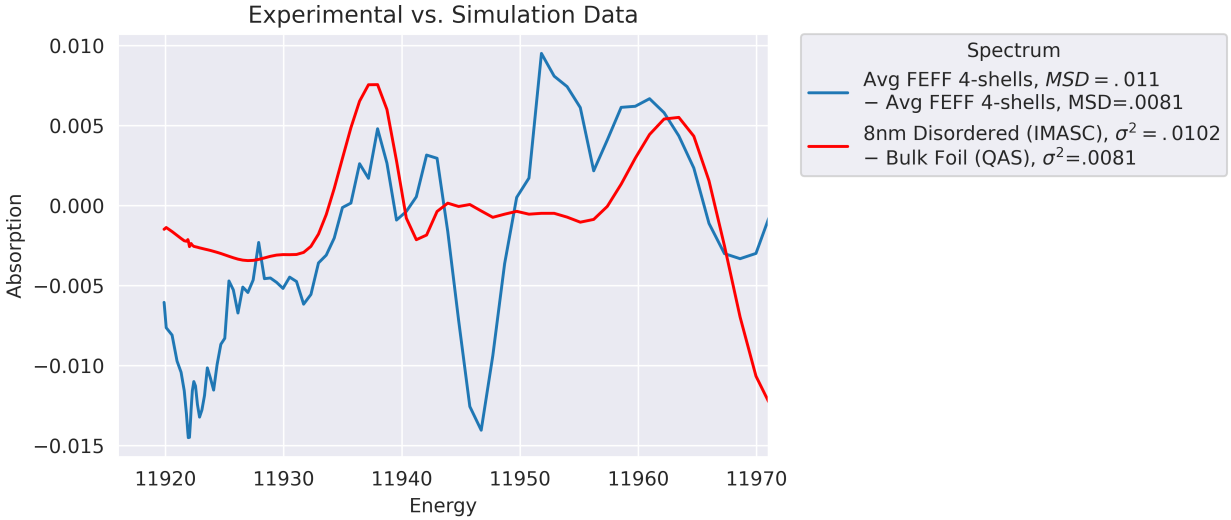


Figure 2.7: The difference between the nanoparticle spectrum and the bulk spectrum are plotted for the same data as in Figure 2.6. It is easier to see where the bulk and the nanoparticle absorption crisscross by plotting the difference.

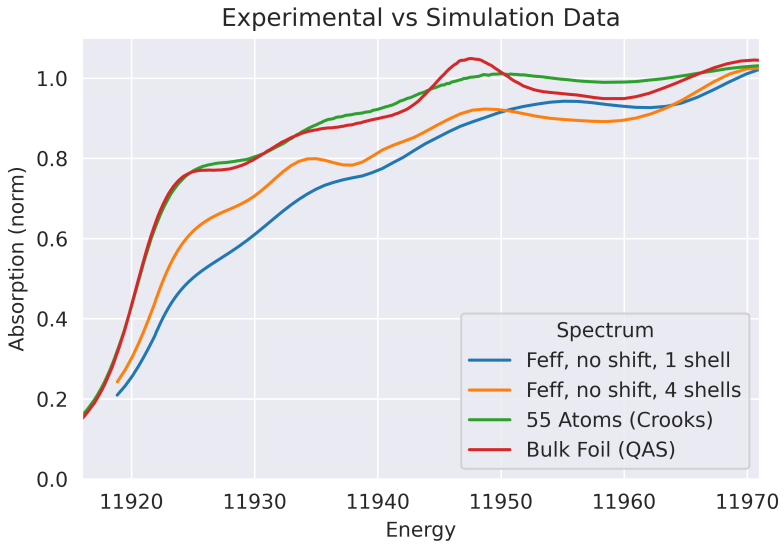


Figure 2.8: Comparing the bulk foil (red) measurement to the 55 atom nanoparticle (green) measurement is an analog to comparing the 13 atom simulated spectrum (blue) to the 55 atom simulated spectrum (orange).

2.4 Particle-Averaged FEFF vs. Skewnorm-Averaged Structures

While it is important to see that the skewnorm-averaging methodology maintains the systematic changes in XANS spectra as disorder increases (as seen in section 2.3.3), it is more important to compare the this methodology to the widely-used particle-avagerd methodology (seen in section 2.1)

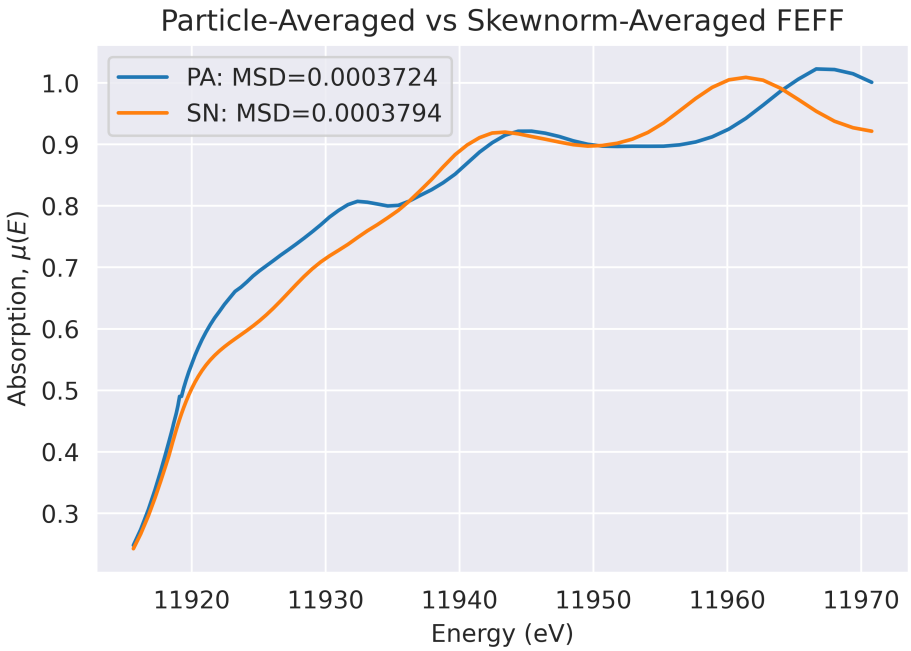


Figure 2.9: The particle Averaged FEFF vs. Skewnorm Averaged FEFF. The systematic difference reveal a flawed assumption in the skewnorm-Averaged Methodology

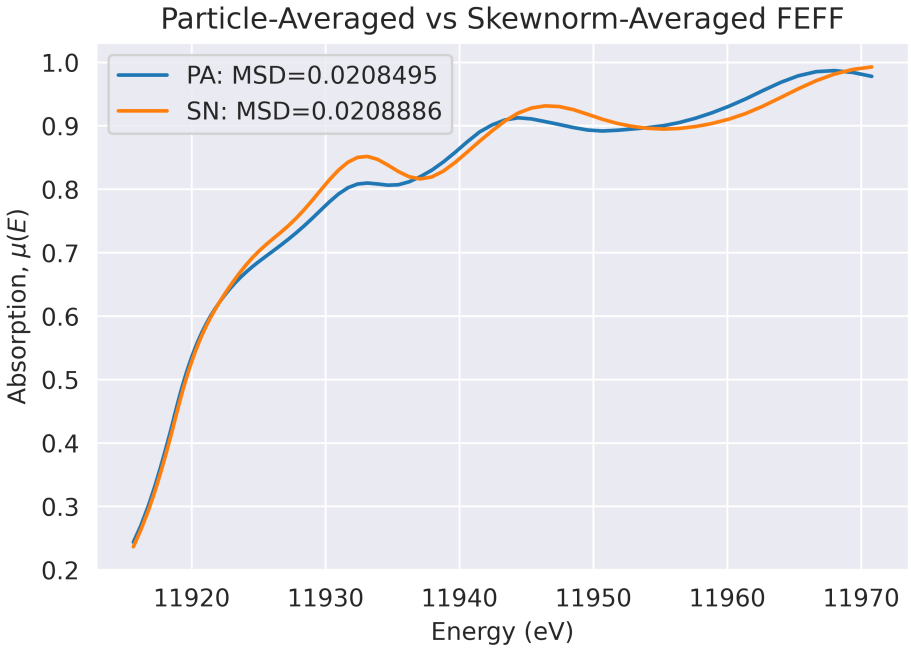


Figure 2.10: The particle Averaged FEFF vs. Skewnorm Averaged FEFF for particles of high disorder. The systematic differences between the particle averaged feff and the statistical averaged methods are less significant for particles of high-disorder. The persistent differences, however, suggest a false assumption in the statistical averaging mothodology.

Chapter 3

Machine Learning

Predicting disorder from XANES spectra requires a sophisticated model or algorithm capable of extracting non-linear features from the data. Increasing the disorder in the structure does not simply shift or scale the spectrum by a scalar; instead, disorder alters the spectrum in a complex and unknown way, for which we rely on machine learning (ML) to discern. The general goal of ML is to recognize patterns in data, iteratively learning to solve complex, non-linear problems to make powerful predictions on new, unseen data [24]. Due to the integral role ML plays in this thesis, the following chapter serves to establish how neural networks fundamentally work. All terms necessary for understanding the specific neural network implementation discussed in the chapter 4.2 will be defined here.

First, we clarify the distinction between the following commonly misused terms: machine learning (ML), deep learning, and artificial intelligence (AI). ML refers to the general computational technique of fitting a model to a collection of data utilizing an iterative training process. These models can either be regressors or classifiers, the former predicts a continuous range of values, and the latter being a discrete predictor. Neural networks are one example of a machine learning model that tends to be computationally intensive. They are capable of creating highly non-linear models able to solve complex tasks such as object detection [25] or speech recognition [26] [27]. Neural networks were inspired by biological nervous systems; consequently, graphical representations of ANNs include common terms such as "nodes" and "connections." The field of ML involving ANNs with many layers is referred to as deep learning [28]. AI is a subfield of deep learning where a neural network is trained to generate human-like responses. Common examples of AI are generative chatbots [29] and a virtual assistants [30] [31].

With these definitions, to predict disorder from XANES we utilize deep learning to train a regression-based neural network. The following sections will walk through the mathematical process of training a simple neural network. In practice, sophisticated APIs such as Google's TensorFlow [32] or Facebook's PyTorch [33] handle the mathematical backend and optimization; however, one must first build a fundamental understanding of the methods these frameworks are executing before attempting to implement them.

3.1 Feedforward and Backpropagation in ANNs

Understanding how a neural network makes a prediction requires a solid grasp of linear algebra. The process where a NN passes information from the input to the subsequent layers to make a prediction is called the feedforward process. The name comes from the operation where each layer of the NN passes information to the next layer until the reaching output layer. The process of updating the weights of the NN is called backpropagation and relies principally on vector calculus. Neither action is particularly mathematically complicated; however, there are so many parts that it is easy to get lost in the sea of similar-sounding partial derivatives. In this next section, we explicitly walk through the math for the feedforward process of a fully connected (affine) neural network.

3.1.1 Feedforward

Consider the neural network in figure 3.3 with an input layer of three nodes, a single hidden layer with five nodes, and an output layer with two nodes. The input layer (zeroth layer) has a cardinality of $\mu = 3$ and is represented in Einstein notation[1] as a row vector x_μ. Each edge in the graph represents a weight that is multiplied with the connected node on the left. Each node in the input layer is multiplied by the weight of the connected edge and added together. This operation for all input nodes and weights can thus be represented as a dot product between the input layer row vector and a weight matrix. The hidden layer (first layer) has a cardinality of $\nu = 5$. Thus, the resulting dot product is $x_\mu W_\nu^{\mu(1)}$ where $W_\nu^{\mu(1)}$ represents the matrix of weights to create the first layer. While this result has the correct dimensionality for the hidden layer, there are still two operations reuired to produce the actual values for the nodes $h_\nu^{(1)}$. First, a small trainable parameter, b_ν is added to every value from the previous calculation. The values in this row vector are called biases and are introduced to prevent overfitting—i.e. the phenomenon where a model predicts the training data well, but is unable to generalize and predict un-seen data. Biases are a regularization parameter. Regularization techniques are discussed below. An example is provided in Figure 3.2. The final operation applied to produce the first hidden layer's values is known as an activation function. Without an activation function, a neural network would be unable to learn non-linear features. There are several types of activation functions, the three most common being sigmoid, tanh, and ReLu.

Sigmoid and Tanh

The sigmoid activation function is defined as the following:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.1}$$

The sigmoid activation function maps the input between zero and one. Hence, sigmoid activation functions are often used in the final layer to output a probability. The sigmoid

[1]Recall that in Einstein notation repeated indices are implicitly summed over.

function approaches 1 and -1 around $x = 4$ and $x = -4$ respectively, meaning that the sigmoid activation function is only useful within that limited range. One issue with both these activation functions is the potential for creating a vanishing gradient. The gradient of either of these functions approaches zero for values above four. This asymptotic approach hurts the ability of the NN to meaningfully update its trainable parameters. The importance of calculating the gradients of these activation functions will be discussed in the next section in the context of backpropagation.

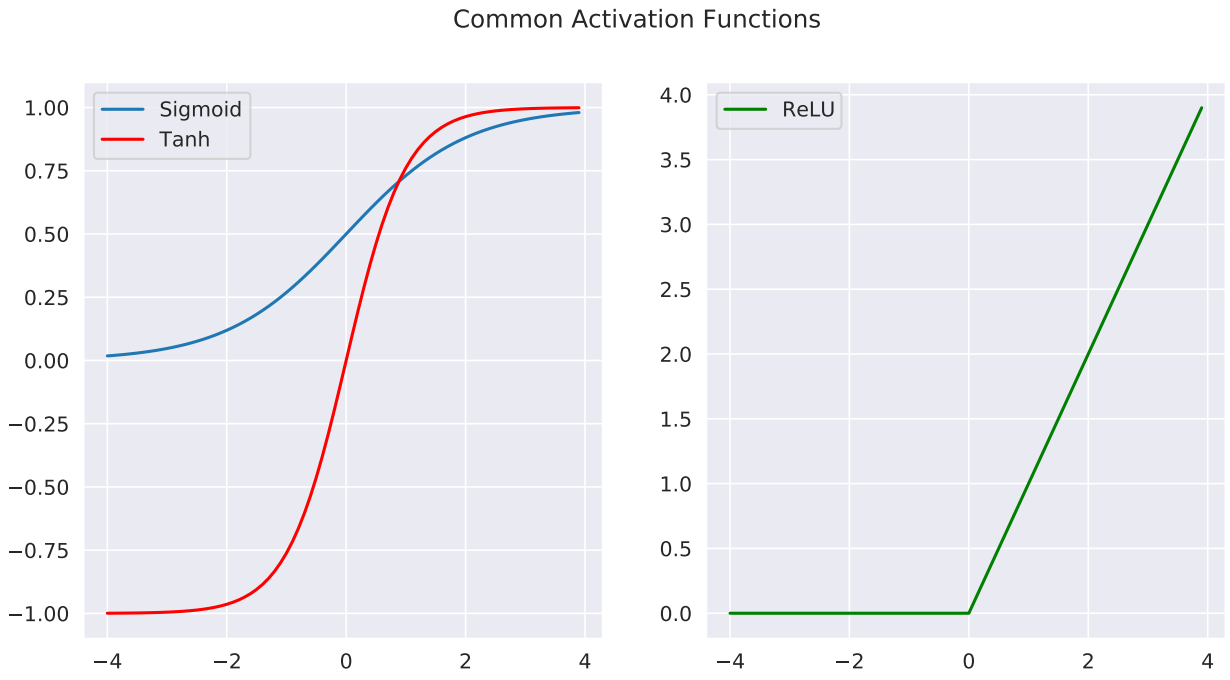


Figure 3.1: Plotted above are three common activation functions: sigmoid, tanh, and ReLU. Simgoid and tanh are particularly useful for scaling the output of a neural network layer to be within a given range. ReLU and its variations are useful for deep ANNs, where vanishing gradients are problematic.

ReLU

The **Re**ctified **L**inear **U**nit activation function, or ReLU has become an important staple of machine learning. The activation function, $f(x)$, is defined as the following:

$$f(x) = \begin{cases} 0 & \text{if } x <= 0 \\ x & \text{if } x > 0 \end{cases} \tag{3.2}$$

ReLU is important because it provides a much greater range in values. Whereas the sigmoid and tanh activation functions saturate around 4, ReLU never saturates for linear values.

Additionally, ReLU is simple to calculate and tends to help neural networks converge quickly. Arguably thier most major benefit is the reduced likelihood of creating a vanishing gradient. Further, because ReLU returns 0 for any negative value fed foward into the node, many ReLU activation functions in a given layer help lead to sparser layers, reducing the overall complexity of the model and helping prevent overfitting.

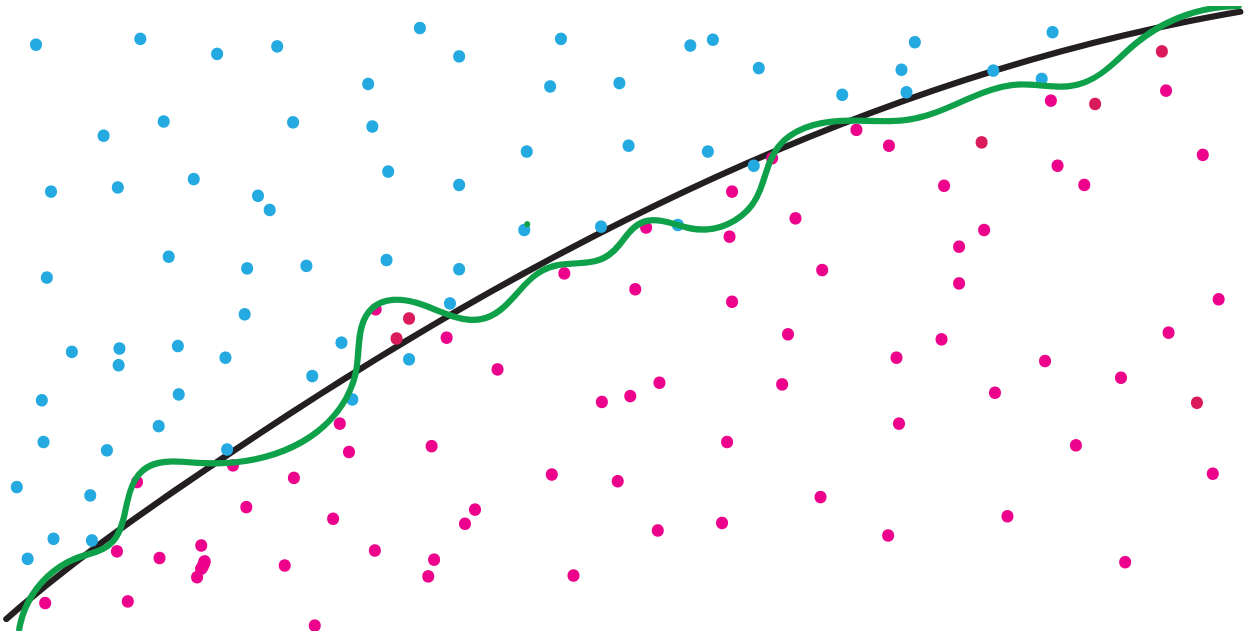


Figure 3.2: The green curve represents a model overfitting a binary classification problem. Although it makes near-perfect predictions for the training data in this figure, the model will not generalize as well as the simpler black curve when it tries to predict new, unseen data. Introducing biases and dropout layers in neural networks are strategies to prevent overfitting to reduce the model variance and fit the data like the black curve.

With the input nodes dotted with the weights, the baises added, and then the activation function calcuated for each node, we arrive at the final final for the first hidden layer. Mathematically, $h_\nu^{(1)} = \sigma\left(x_\mu W_\nu^\mu + b_\nu\right)$. This process is now repeated, only with the starting layer $h_\nu^{(1)}$ and the output $\hat{y} = \sigma\left(h_\nu W_\kappa^\nu + b_\kappa\right)$ is the final output of the neural network. The equations for each step as well as the dimensionality of each layer can be found in Figure 3.3.

3.1.2 Loss Metrics and Regularization

In order to update the netork parameters, it is necessary to evaluate the quality of every prediction the neural network makes. The measure of the error in one prediction is referred to as the *loss*, and the summed total of all the errors in predictions is called the *cost*. For regression problems, the two most common cost functions are the mean squared error and

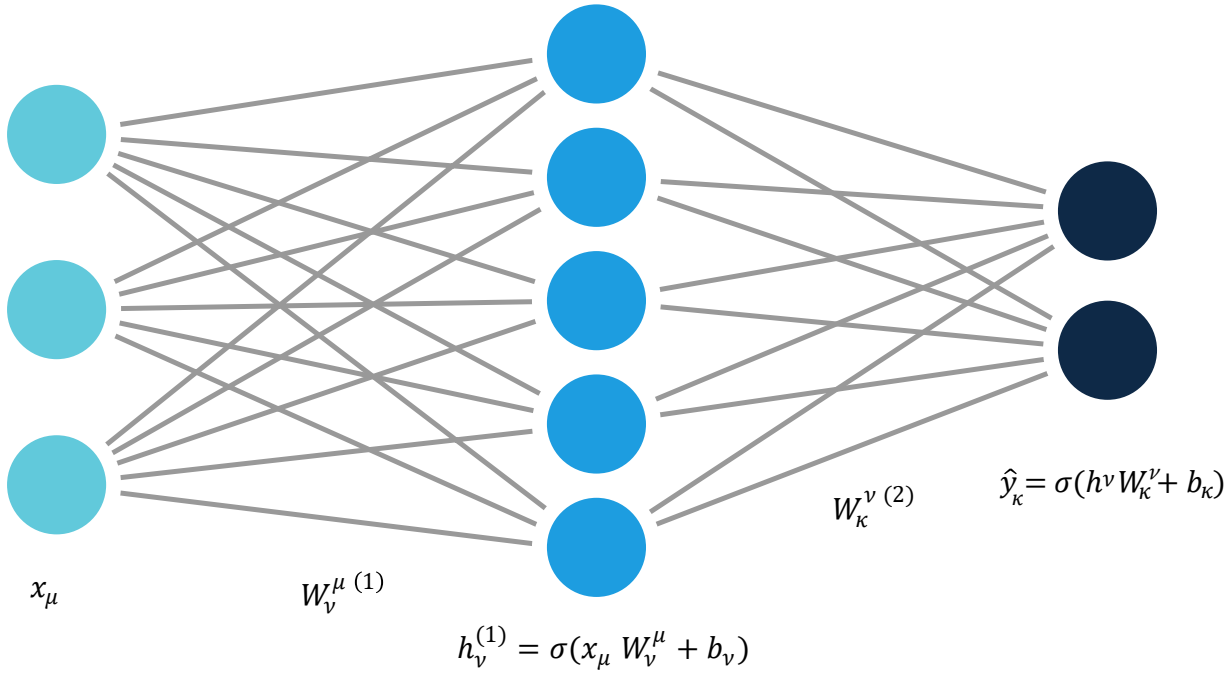


Figure 3.3: This diagram of a fully connected (affine) neural network has a single hidden layer and two output nodes. The tensors for each hidden layer are written in Einstein notation. The implicitely summed over greek letters and dimensionality are written below the tensors for clarification.

the mean absolute error [34]. These are also referred to as the L2 and L1 costs, respectively, and defined as:

$$\text{L1 Cost} = J = \sum(\hat{y}_i - y)^2 + \lambda \sum |W| \tag{3.3}$$

$$\text{L2 Cost} = J = \sum(\hat{y}_i - y)^2 + \lambda \sum (W)^2 \tag{3.4}$$

where n is the number of training sampels. The equation for updating the weights under L2 regularization is as follows:

$$W := (1 - \alpha\lambda)W - \frac{\partial J}{\partial W} \tag{3.5}$$

where α is the learning rate, λ is the regularization hyper-parameter, and J is the cost. L2 regularization is often referred to as weight decay. Every iteration the weights are pushed closer to zero due to the multiplicatiion of the weights by a value < 1. L1 is known as LASSO

(least absolute shrinkage and selection operator), because it shrinks the less important features' coefficients to zero. This is because for small values $|w|$ is a much stiffer penalty than w^2. Thus, L1 is thus a good choice when you have a dozens of features.

Dropout layers are another variation of model regularization used exclusively for neural networks [35]. The idea is to introduce a hidden layer with a probability "dropping out," i.e. ignored. Large weights in a neural network are indicative of a high-variance network, likely to be overfitting the data. By indroducing layers with a probability of dropping out, the inward connection to the next layer change stochastically from batch to batch. This has the effect the adding noise to the network, similar to the inclusion of biases [36] [37].

3.1.3 Backpropagation

Backpropagation is the process of updating all the trainable parameters of the machine learning model, including weights, biases, and any other trainable parameters. The partial derivative require repeated use of the chain rule. Representing the above neural network as a functional yields the equation:

$$\hat{y} = \sigma\left(h_\nu^{(1)}\left(x_\nu\right)\right) \tag{3.6}$$

where the the output layer $\hat{y}$ is a function of the hidden layer, $h_\nu^{(1)}$ which in turn is a function of the input layer x_μ. Consider the L2 loss (3.3). Note that the loss function is a function of the previous functional (3.6). To see how much to shift the weights, calcuate the gradients for each layer. The first partial derivative is trivial:

$$\frac{\partial J}{\partial J} = 1 \tag{3.7}$$

For the next layer (the output layer):

$$\frac{\partial \hat{y}}{\partial J} = \frac{\partial J}{\partial J}\frac{\partial \hat{y}}{\partial J} \tag{3.8}$$

For the next nested-function (the sigmoid):

$$\frac{\partial J}{\partial \sigma} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma} \tag{3.9}$$

The next layer is the hidden layer before the activation function. For simplicity it will be referred to as g where $g = x_\mu W_\nu^\mu + b_\nu$.

$$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma}\frac{\partial \sigma}{\partial g} \tag{3.10}$$

Next layer is the input layer, X_μ has no trainable parameters, so the process for this network architecture is complete. If there were more hidden layers, the chaining rule would continue by multiplying the gradient calculated in the previous step by the gradient of the next layer with respect to the previous layer (towards the input layer).

3.1.4 Concrete Example

Finally, we will repeat the backpropogation process of updating the weights for the previous example as explicitly as possible. Consider again the functional $\hat{y}$ in equation (3.6) and the L2 loss in equation (3.3), rewritten here as L . To determine how do shift the weights in the function, want to know $\frac{\partial J}{\partial W}$, where W is short for $W_\kappa^{\nu(2)}$, the weight matrix connected to the second layer (the output layer).

$$L = \frac{1}{m}\sum(\hat{y} - y)^2 \tag{3.11}$$

$$\tag{3.12}$$

As before, the first partial derivative is trivial

$$\frac{\partial J}{\partial J} = 1 \tag{3.13}$$

and the next partial derivative is also strait-foward:

$$\frac{\partial J}{\partial \hat{y}} = 2(\hat{y} - y) \tag{3.14}$$

Applying the chain rule yields:

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial J}{\partial J}\frac{\partial J}{\partial \hat{y}} = 1 \cdot 2(\hat{y} - y) \tag{3.15}$$

The next required term in the chain is the derivative of the loss with respect to the sigmoid:

$$\frac{\partial J}{\partial \sigma} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma} \tag{3.16}$$

We already found the first term (3.15), and the second term in trivial.

$$\frac{\partial \hat{y}}{\partial \sigma} = 1 \implies \frac{\partial J}{\partial \sigma} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma} = 2(\hat{y} - y) \cdot 1 \tag{3.17}$$

At this point, we have the derivative $\frac{\partial J}{\partial \sigma}$ for the σ in the final layer $\hat{y} = \sigma\left(h_\nu W_\kappa^\nu + b_\kappa\right)$. The next derivative in the chain will be $\frac{\partial J}{\partial g}$ where $g = h_\nu W_\kappa^\nu + b_\kappa$ as before. Continuing the chain,

$$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma}\frac{\partial \sigma}{\partial g} \tag{3.18}$$

where

$$\sigma(g) = \frac{1}{1 + e^{-g}} \tag{3.19}$$

$$\implies \frac{\partial \sigma}{\partial g} = \sigma(g)(1 - \sigma(g)) \tag{3.20}$$

Combining these previously calculated terms yields:

$$\frac{\partial J}{\partial f} = 2(\hat{y} - y) \cdot 1 \cdot \sigma(z)(1 - \sigma(z)) \tag{3.21}$$

Now comes the good part. Recall that the trainable parameters in the network are the weights and biases (W and b). The next step will is to calculate the gradients with respect to each, which in turn will be used to update the paramters.

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial g}\frac{\partial g}{\partial W} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma}\frac{\partial \sigma}{\partial g}\frac{\partial g}{\partial W} \tag{3.22}$$

The last partial in the chain is

$$\frac{\partial g}{\partial W} = W \tag{3.23}$$

So,

$$\frac{\partial J}{\partial W} = 2(\hat{y} - y) \cdot 1 \cdot \sigma(z)(1 - \sigma(g)) \cdot W \tag{3.24}$$

For the baises,

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial g} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \sigma}\frac{\partial \sigma}{\partial g}\frac{\partial g}{\partial b} \tag{3.25}$$

$$\frac{\partial g}{\partial b} = 1 \tag{3.26}$$

$$\implies \frac{\partial J}{\partial b} = 2(\hat{y} - y) \cdot 1 \cdot \sigma(g)(1 - \sigma(g)) \cdot 1 \tag{3.27}$$

Note that W is actually $W_\kappa^{\nu(2)}$, a matrix of weights, and b is actually b_κ, a row vector of biases. Thus, the above equation (3.25) is just the partial derivative for one term in the weight matrix or bias vector. Repeating the process for each term in the matrix W and vector b yields the gradients $\boldsymbol{\nabla}_w L$ and $\boldsymbol{\nabla}_B L$, which represent the gradient of the loss function with respect to the weights and biases, respectively. This is the origin of the term, "gradient descent," an optimization algorithm discussed in the next section. This was just the process to calculate the gradients need to update weights for the final layer, but one can see how continuing the process of chaining partial derivatives will yield the gradients for earlier layers in the network.

3.2 Optimizers

Having calculated all the gradients via backpropagation, the weights and biases of the network can now be adjusted. The general idea of gradient descent relies on the fact that the gradient of a function points in the direction of greatest increase. Thus, to optimize

the network—which is equivalent to finding the parameters that minimize the value of the loss function—the weights and biases are updated by shifting their values in the opposite direction of the gradient of the loss function with respect to the weights, $\boldsymbol{\nabla}_w L$. Gradient descent is the core principle of machine learning; this efficient algorithm for systematically updating model-parameters, made it possible to develop deep neural networks and train them with large datasets. Several improvements have been made to gradient descent since its inception [38], with the state of the art being AdamW [39]. In this section, several optimization algorithms are introduced in order to provide context for Adam, the optimizer used for training our model. In the previous section, the gradients for the weights and biases were written explicitly. For simplicity, the variable θ is introduced to refer to either parameter. As before, $J(\theta)$ is the cost function; it could be the L1 or L2 cost, or any other differentiable measurement of fit quality.

Gradient Descent

Vanilla gradient descent [40] updates the parameters in the following way:

$$\theta := \theta - \eta \cdot \boldsymbol{\nabla}_\theta J(\theta) \tag{3.28}$$

Here (3.28), η is a *hyperparameter* known as the learning rate. A hyperparameter is a user-defined parameter that must be chosen before the training process begins; it is not a trainable parameter. Hyperparameters can be "tuned" by repeating the entire training process with various hyperparameters set. Typically, one trains the model with a variety of hyperparameters over a short number of epochs. Once satisfied, the number of epochs is increased, and the training is repeated with the best-found hyperparameters. One limitation to gradient descent is the need for the entire cost J to be calculated. For large datasets, this can become impractical. Two common variants are batch gradient descent and stochastic gradient descent (SGD). In the former, the training set is divided into batches, and the gradient is updated after each batch. One iteration through all the batches is called an *epoch*. In the latter, the gradient is calculated using the loss function instead of the cost function—i.e. the gradient is calculated, and the parameters are updated after each training sample. Both methods greatly reduced training time with the help of optimized, parallel computing [41]. By updating the parameters after every training sample, SGD will move in the direction of the true gradient. The major limit to these methods, however, is the fixed learning rate [42]. If the learning rate, η is too large, the algorithm will be unstable and "bounce" around the global minimum of the cost function. If η is too small, the algorithm will, at best, take a long time to train, and at worst, end up stuck in a local minimum.

Stochastic Gradient Descent with Momentum

Compared to regular SGD, stochastic gradient descent with momentum [43] can greatly reduce the time to convergence. The general idea is to add a fraction of the previous parameter update to the current update. The exponential moving average (EMA) is an averaging of

points within a period that puts greater weight on more recent points[2]. Here, S_t is the t^{th} value in the sequence S, and V_t is the t^{th} value in the new exponential moving averaged sequence, V .

$$V_t = \beta V_{t-1} + (1-\beta)S_t \tag{3.29}$$

where $\beta\epsilon[0,1]$, a hyperparameter which partly defines how much weight the previous $1/(1-\beta)$ terms of S contribute[3]. EMA's are common in market forecasting, so often S_t is the price at time t. The continuous update for SGD with momentum is as follows:

$$V_t = \beta V_{t-1} + (1-\beta)\,\boldsymbol{\nabla}_\theta J\,(\theta) \tag{3.30}$$

$$w = W - \alpha V_t \tag{3.31}$$

Here, α is the learning rate, as always. To be clear, $\boldsymbol{\nabla}_w L$ is the gradient of the loss function with respect to the weights. Note that the cost function $J(\theta)$ may instead be the loss function $L(\theta)$ if updates are performed after each training sample (i.e. a batch size of one). SGD with momentum tends to perform better than SGD because it gives a closer estimate of the full gradient from the batch than SGD. Additionally, the momentum helps push the update through ravine-shaped local minima in the correct direction, whereas SGD tends to oscillate back and forth along the ravine's steeper dimension [44].

Root Mean Squared Propagation

Root Mean Squared Propogation (RMSprop)[4] is another variant of SGD designed to improve convergence speed and remedy adagrad's [45] tendency to rapidly diminishing gradients [46][47]. The idea is to dampen oscillations in directions when the predictions are close to the cost function's minimum and accelerate movement when far away. In RMSprop, we keep a moving average of the squared gradients for each weight and use these to divide the learning rate by an exponentially decaying average. As before, $\boldsymbol{\nabla}_\theta J$ is the gradient of the cost with respect to weights.

$$S_{k+1} = \beta S_k + (1-\beta)(\boldsymbol{\nabla}_\theta J \cdot \boldsymbol{\nabla}_\theta J) \tag{3.32}$$

$$\theta_{k+1} = \theta_k - \alpha \frac{\boldsymbol{\nabla}_\theta J}{\sqrt{S_{k+1}} + \epsilon} \tag{3.33}$$

The hyperparameter ϵ is included in the denominator to prevent a possible division by zero as well as provide more stability.[5]

[2]In contrast a simple moving average treats each point as equally significant

[3]Typically 0.90 is a good starting point

[4]RMSprop has an interesting history. It is an unpublished algorithm, first introduced by Geoff Hinton in an online series of lectures. Nevertheless, it is an incredibly popular algorithm and included in most ML platforms.

[5]Typical values for α and β are 0.001 and 0.9, respectively.

Adaptive Moment Estimator

Adaptive Moment Estimator (Adam) is a combination of RMSprop and SGD with momentum. Adam uses the squared gradients to scale the learning rate for each parameter (similar to RMS prop), and it uses a moving average of the gradient (similar to SGD with momentum).

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)(\boldsymbol{\nabla}_\theta J) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\boldsymbol{\nabla}_\theta J \cdot \boldsymbol{\nabla}_\theta J) \end{aligned} \tag{3.34}$$

The new parameters in this algorithm, m_{t-1} and v_{t-1} are the first and second moments of the gradient (the mean and variance), respectively. Adam's 80,000 citations in the six years since its publication gives some indication of the importance and power of this algorithm [48]. This was the chosen algorithm for training our neural network for predicting disorder in XANES.

3.3 Normalization

Normalization is an important step for any machine learning algorithm. There are three types of normalization utilized in our training process: feature normalization, label normalization, and batch normalization. Without feature normalization, a model will put greater weight on features with larger values. For example, if a model is trained to predict surface stress of a silca bead in silicone gel given the diameter of the bead in microns and the adhesion energy in mNm^{-1}, the model would learn to ignore the bead's size in its predictions. This is because the values particle's size are $1000\times$ smaller than the adhesion energy [49]. To correct for this, each feature is "normalized" on the training data to be on the same scale. Often a z-score is used to center the features around zero with a standard deviation of one. This is known standardizing or applying a standard-scalar [50].

$$Z^{(i)}_{norm} = \frac{x_i - \mu}{\sqrt{\sigma^2 - \epsilon}} \tag{3.35}$$

In our neural network, the training features are normalized in this way.

For the same reasons feature normalization is important, the training labels must also be normalized. Instead of using the standard scalar, the labels are normalized using min-max normalization, which normalizes the values between zero and one. This is a useful when the labels are known to be evenly distributed over a range. To scale the i^{th} label (y) via min-max scaling:

$$Z^{(i)}_{norm} = \frac{x_i - \mathrm{Min}(y)}{\mathrm{Max}(y) - \mathrm{Min}(y)} \tag{3.36}$$

Scaling the training features means the neural network will predict the scaled values. To determine the true value, the prediction can simply be "un-scaled" via:

$$y^{(i)} = -\frac{Z^{(i)}_{norm}\left(\text{Max}(y) - \text{Min}(y)\right)}{\text{Min(y)}} \tag{3.37}$$

It is important that the scaling paramters μ, σ, Min(y) and Max(y) come from the training set—not the testing or validation set. Using the values from the entirety of data constitutes a form of *data leakage*, where the training process is given a hint of the validation or test data. It is important that the model never sees the testing or validation until testing or validation time; otherwise the model is unlikely to generalize as well to unseen data as one might expect given a loss curve.

Batch normalization is a technique designed to make NN's more robust to internal covariate shift [51]. Covariate shift referrs to a systematic difference between the training and validation data, or in the context of a training batch, a systematic shift in the distribution of data from batch to another [52]. For example, if a NN is trained for binary classification to predict whether or not an image includes a cat—and the network is trained on images of only black cats—the network is unlikely to make a correct prediction when it encounters an image of an orange cat.

The idea of batch normalization is to normalize each hidden layer similar to how training data or labels are normalized; however, whereas normalizing the training data centers the dataset or labels using fixed parameters such as the mean and variance, in batch normalization the mean and variance of the batch normalization are learnable parameters. In batch normalization, the values for a hidden layer are scaled via:

$$\widetilde{Z}_i = \gamma Z^{(i)}_{norm} - \beta \tag{3.38}$$

Notice that if $\sqrt{\sigma^2 + \epsilon}$ and $\gamma = \mu$, we get equation (3.35), i.e. the hidden layer is normalized in the same way as the input layer. This is generally not useful, however, because normalizing all parameters to be centered around zero causes the sigmoid-like activation functions to be mostly focused on the linear regime.

The output of the batch normalization is passed forward to the next hidden layer of the network, while the normalized input is retained in the current layer. Normalizing the hidden layers for each batch means that later hidden layers do not have to adapt as much to the earlier hidden layers. Consequently, this allows the deeper layers to do a better job tuning themselves a little more independently of the other layers, improving performance and speeding up the learning process. Note, because each batch is scaled ($z \to \widetilde{z}$), a small amount of noise is added, which acts as a slight form of regularization[6]. Recently, several papers [53] [54] [55] have been published disputing the reason batch normalization improves the model's performance; none, however, dispute its efficacy. Data augmentation may be vital in building the neural network trained on simulation data to predict experimental data, for which there is a sparsity of data for training.

[6]Note, while batch-norm adds regularization, this is not intended to be used as a form of regularization. L1, L2 regularization or dropout layers should be used instead.

3.4 Data Sparsity

Unfortunately, not all project goals include a plethora of diverse training data. This section introduces two techniques for dealing with training data: data augmentation and transfer learning.

3.4.1 Data Augmentation

Data augmentation is a technique for expanding the size and variance of the training data for machine learning purposes. It has been critically important for developing powerful deep neural networks, particularly in the domain of image processing. [56][57] [58] [59]. Consider the example in section 3.3 of a cat-vs-not-cat binary classifier. The idea of data augmentation is to take the dataset containing only images of the red cats and add new images created from the original dataset. Common methods of data augmentation are image cropping, image rotation, and introducing image filters that alter the color, sharpness, or contrast. For signal processing, including absorption spectroscopy, common methods for expanding the training data size include the introduction of Gaussian noise and shifting the spectra horizontally [60]. By artificially expanding the size and complexity of training data, data augmentation helps prevent models from overfitting.

3.4.2 Transfer Learning

Transfer learning was first introduced in 1976 to [61] [62] [63]. The idea is to alter a model trained on one task to be to solve a new, but similar task. One famous example involves a neural network originally trained to classify pastries, which, utilizing transfer learning, was re-purposed for detecting cancer cells [64]. Consider a model first trained on dataset A with the final goal of predicting dataset B. Note that A and B in this example are not a train-test split, but, instead, inherently different problems. By pre-training the model on A to solve a similar task with B , the model learns inductive biases which encourage the model's parameters θ_B to be similar to θ_A. This may have the effect of training the model to learn low-level features that may not have been learned from B alone [65].

In the context of this thesis, transfer learning will be an important tool for creating a neural network capable of predicting disorder from an experiemental XANES spectra. Because of the sparsity of experimental data, it is unfeasible to train the neural network on experimental data alone. Instead, we rely on simulated XANES spectra created with FEFF. The systematic differences between the simulated XANES spectra and the experimental counterparts, however, suggest a neural network trained purely off simulation spectra will not perform well when it encounters an experimental spectrum for the first time. Applying the principles of transfer learning, the neural network can first be trained with the simulated XANES spectra, for which there are ample examples. Then, using the limited number of experimental data—included extra examples created via data augmentation—the network can be trained again via transfer learning.

3.5 Covolutional Neural Networks

Convolution is a mathematical operation for combining two functions, the result of which is a third function revealing the effect of the second function on the first [66]. The convolution of two continuous functions, f and g, is a special type of integral transformation. The result is the integral of the product of f and the shifted inverse of g, where f can be thought of as the input function and g is often referred to as the kernel.

$$f \otimes g = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \tag{3.39}$$

The variable i (no relation to the imaginary number) is represents the weighted-shift in the function $g(\tau)$. Different values of i emphasize different parts of the other function, $f(\tau)$.

In computer science, convolutions are an important and powerful tool for signal and image processing [67] [68]. Because images and signals—which can be thought of as a 1D image—are comprised of a discrete number of points (e.g. pixels), a modified formula is required to perform the convolution. The convolution of the signal f with the kernel g can be written as [69]:

$$f \otimes g = \sum_{j=1}^{m} g(j) \cdot f(i-j+m/2) \tag{3.40}$$

Here (3.40), m is the length of the kernel g, and i and j are hyperparameters. To demonstrate visually, consider a simple, example absorption spectrum with only 10 data points (3.4).

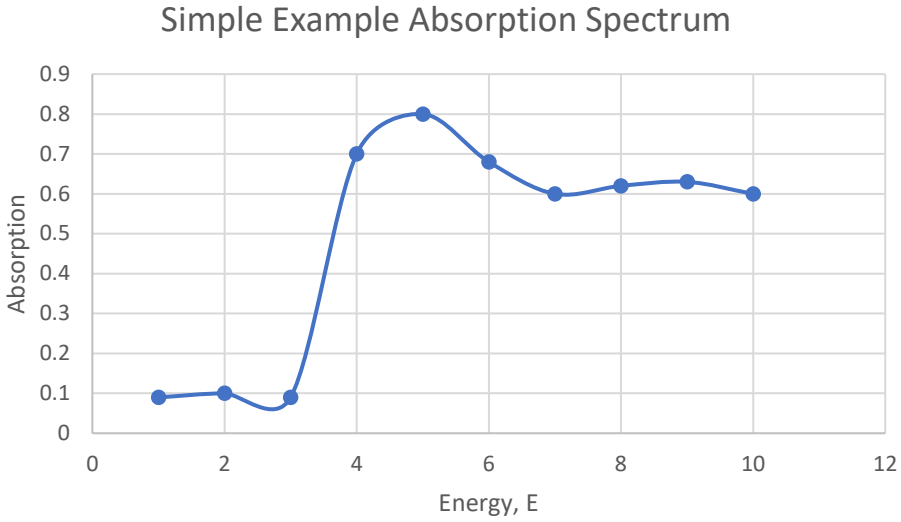


Figure 3.4: A simple abosrption spectrum for demonstration purposes.

Each data point, (E, μ) in the spectrum is described as a feature vector, where the feature is the the energy value for a given point (E, μ). The vector, f, is depicted below with boxes

to represent each element in the vector. Zeros are padded on both sides for reasons that will become clear soon.

$$f = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 0 & .09 & .10 & .09 & .70 & .80 & .68 & .60 & .62 & .63 & .60 & 0 \\ \hline \end{array}$$

Additionally, consider the kernel, g

$$g = \begin{array}{|c|c|c|}\hline .1 & .1 & .1 \\ \hline \end{array}$$

The convolution works by mutliplying each element in the input vector f by the corresponding element in the kernel g and summing the results. The kernel then moves to be centered around the next element in f. One way to think about this process is a kernel or filter sliding over an input signal. Applying the kernel g onto the first index of f yields:

0	.09	.10	.09	.70	.80	.68	.60	.62	.63	.60	0
.1	.1	.1									

$$h(1) = (0)(.1) + (.09)(.1) + (.10)(.1) = 0.019$$

where $h(1)$ is 1st index of the resulting vector. For the next point, the kernel shifts to be centered around it.

0	.09	.10	.09	.70	.80	.68	.60	.62	.63	.60	0
	.1	.1	.1								

$$h(2) = (.09)(.1) + (.10)(.1) + (.09)(.1) = 0.028$$

The final resulting vector is:

$$h = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline .019 & .028 & .089 & .159 & .218 & .208 & .190 & .185 & .185 & .123 \\ \hline \end{array}$$

The toy example was chosen to demonstrate the basics of a 1D convolution. In this example, the input spectrum was a vector of length ten and the kernel of length three. In the context of applying a 1D convolution to a neural network, the size of the input vector is the cardinality of the hidden layer directly preceding the convolutional layer. Often the hidden layer's output, represented as a vector, is reshaped into an n-dimensional tensor before applying the convolution. To apply a 1D-convolution to a Tensor of rank n, simply apply the convolution to each of the n-vectors separately, ensuring to pad the ends of each vector with zeros. Layers with dimensionality greater than one can be "un-raveled". Alternatively, the layers of each dimension can be truncated or "pooled" by averaging the layers that are stacked on one another (or taking the max of each layer) until the desired dimensionality is achieved. A common way to achieve this is with a max pooling layer or an average pooling layer. Max

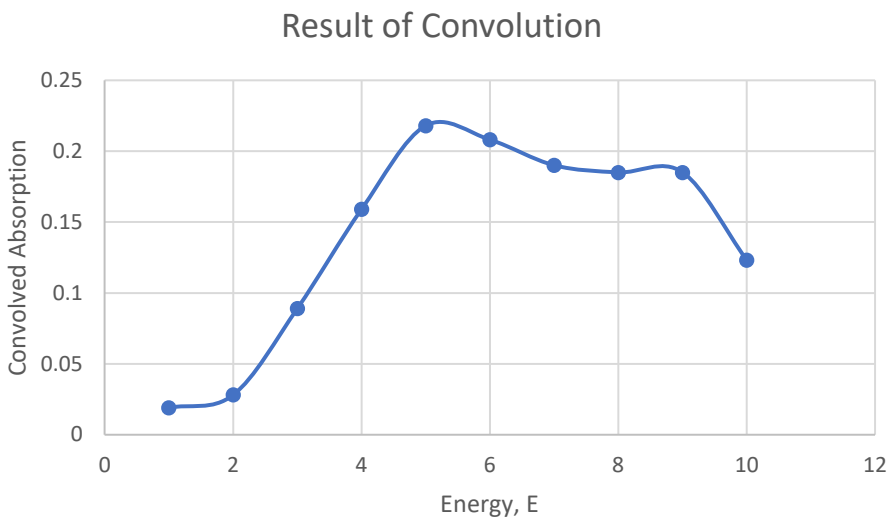


Figure 3.5: The result of the 1D convolution of kernel $g = (.1, .1, .1)$ on the spectrum in Figure 3.4.

pooling and average pooling layers also have the additional benefit of downsampleing the feature space, tending to make the network more robust to slight variations in the position of features in the input image or signal. This is referred to as "local translation invariance" [70].

Another important possible change in the above example is the stride length. In the above example, we "slid" the kernel across the input spectrum one point at a time. This is referred to as a stride size or stride length of one. The stride could be any integer less than the length of the input vector f, though in practice, typically stride lengths tend to be one or close to one. Lastly, in the above example, we only applied one convolutional kernel, or "filter." In practice, many filters are applied sequentially. In the above example, the filter $g = (.1, .1, .1)$ was simply decided upon *a priori*. In practice, the values for each filter in the convolutional layer are initialized randomly or according to the specified initialization function. The default in Keras is Glorot Uniform. The values of each filter are trainable parameters that evolve to produce the best final prediction given the subsequent layers in the neural network.

3.6 How to Train a Neural Network

Building a successful neural network requires a combination of intuition and procedural know-how. The first key is to start with a simple model, perhaps a single hidden. If you start with a complex model with data augmentation and regularization, you'll never be able to tune the hyperparameters and find a good solution. Then, a good strategy is to pick a simple architecture and reasonable hyperparameters and train the model on a small subset

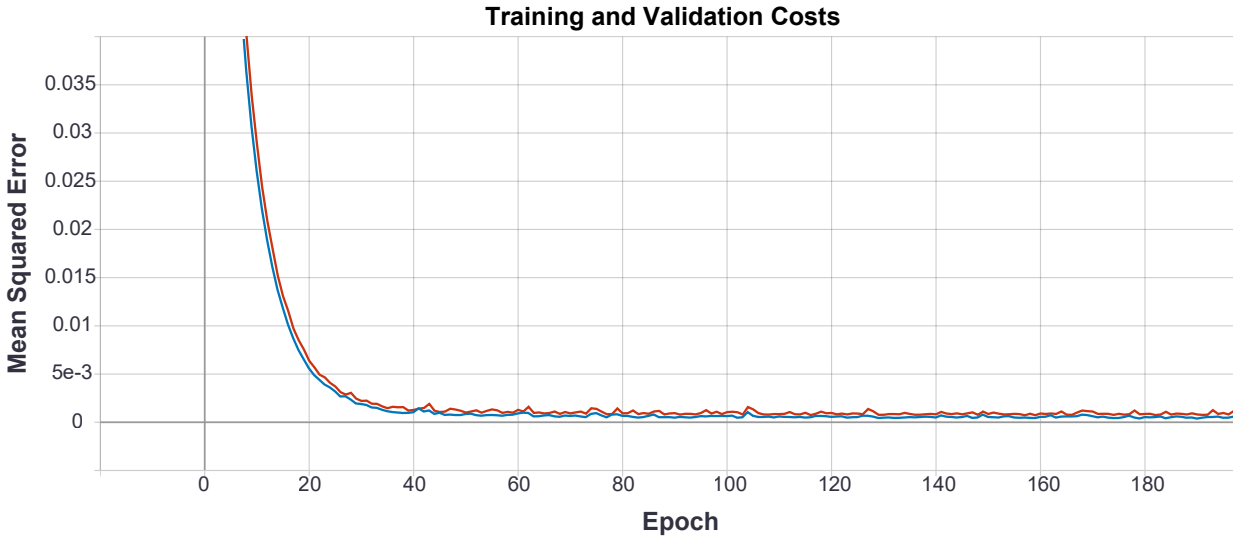


Figure 3.6: In this example loss curve, the x-axis is the epoch, and the y-axis is the mean-squared error of the prediction. The red curve is the training data, and the blue curve is the validation set. Notice how both curves decrease together, but the training curve has a smaller error than the validation curve. This is the expected behavior of a model that is not overfitting and training properly.

of training samples, say 1–10 samples. Train the model over 10 or so epochs and see if the training cost is decreasing and see if you can overfit. If you can overfit the small sample size, it means the code is working and the network architecture makes sense. Now you can increase the number of samples in the training data, either to the full train-test split or a subset of that if working with "big data."[7] Next, you can run a broad hyperparameter search. Afterward, run maybe 20 epochs and see how the training and validation loss is moving. If they both are going down, the architecture looks good. If not, start over. Ideally, the training and validation loss curves will be decreasing together like in figure 3.6. If the model still does not predict well after hyperparameter tuning, the model is underfitting, and a more complex architecture is required (add more layers).

Ideally the training and validation loss will decrease together over many epochs. It is likely, however, that after many epochs the training loss will continue to decrease while the validation loss plateaus. This is the point to start introducing regularization such as dropout layers as well and considering data augmentation. These techniques will allow the model to continue decreasing the validation loss and prevent overfitting. The main goal of training is to minimize the validation loss. The placement of dropout layers and the type of data augmentation are subject to trial and error. Generally, it is best to include dropout layers after a ReLU acticaiton and before an affine layer, and there has been some research

[7]Big data has become a nebulous term, but a reasonable example is when you there is so much data that the dataset cannot be loaded into the computer's memory (RAM) at the same time.

suggesting the inclusion of low-probability dropout layers after convolutional layers tends to improve model performance [36] [37], but these rules do not work for every network and it is still worth trying different options while training.

Chapter 4

Results

Here I expect to showcase lots of nice figures and data to show how well the neural network works

4.1 Training with Simulation Data

4.2 Experimental Data

Training the neural network entirely on simulation data and then making predictions on experimental data is unlikely to provide quality results.

4.2.1 Data Augmentation

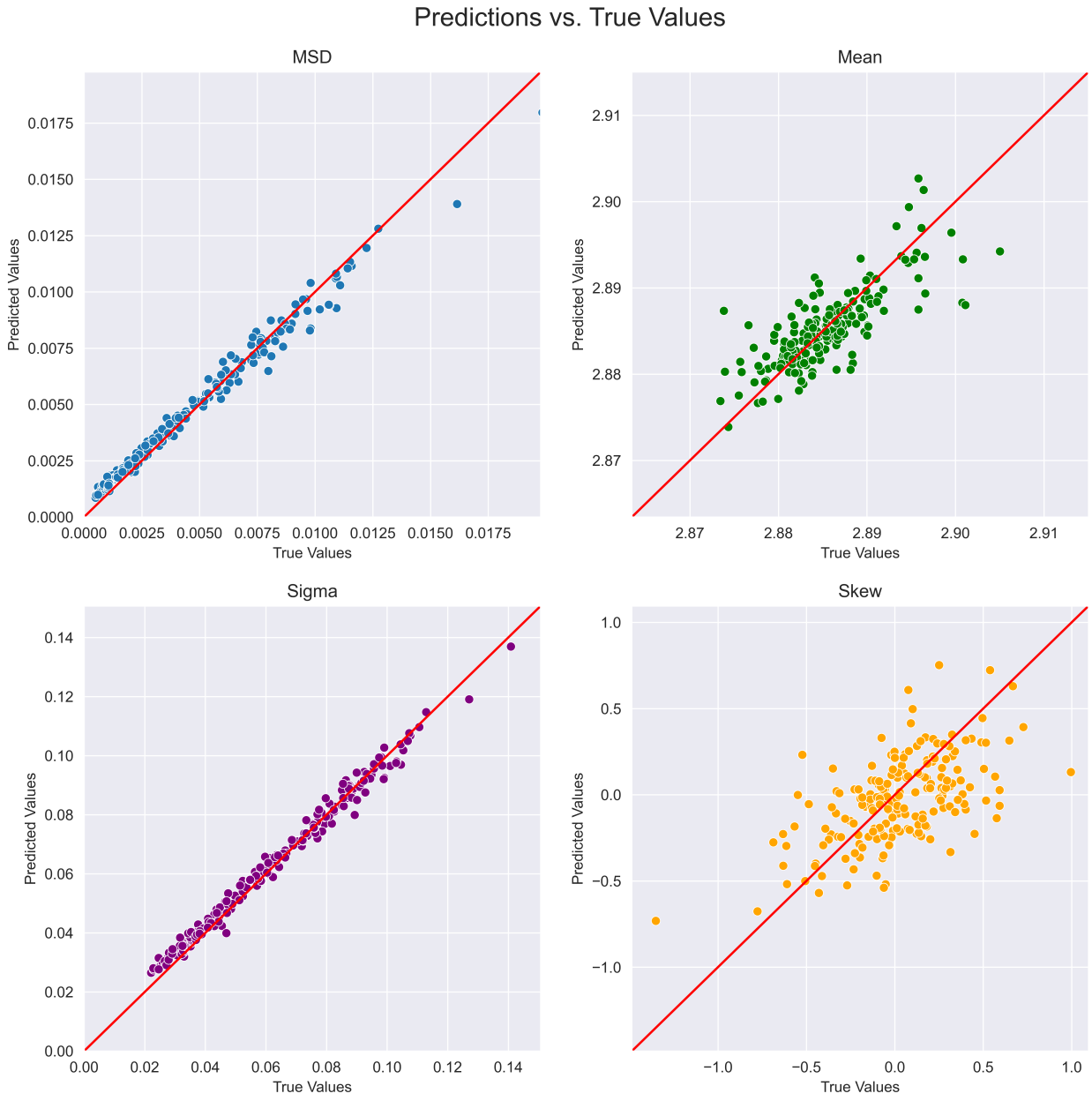


Figure 4.1: One version of the neural network has four output nodes: one for MSD, Sigma, Mean, and Kurtosis. Sigma is just the square root of the MSD and was included during training affirm the patterns recognized by the network.

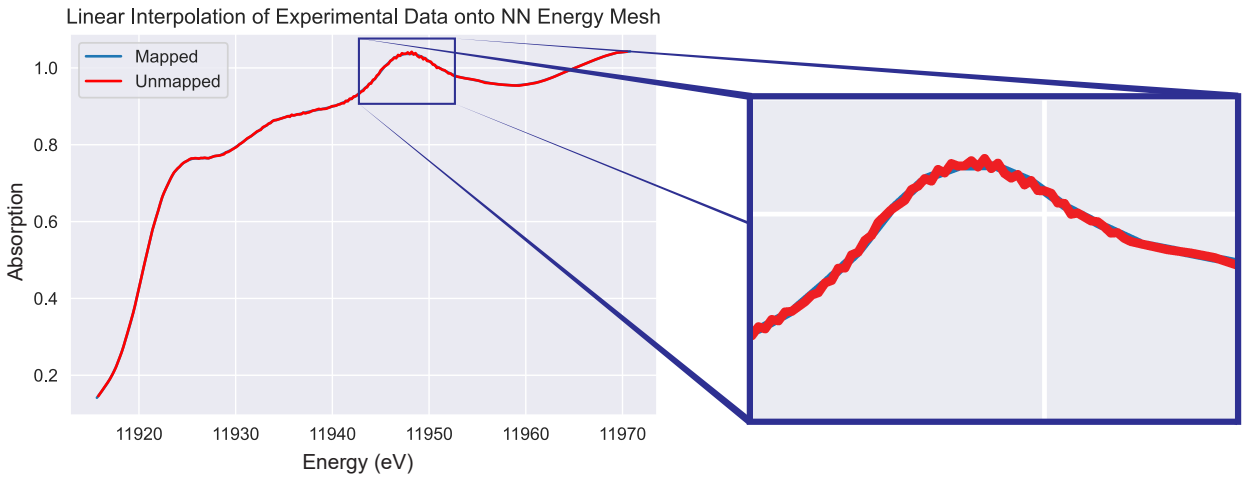


Figure 4.2: The experimental data is measured as a function of different energy values than the ones on which the neural network is trained. Consequently, the experimental spectrum must be mapped onto the proper energy mesh via linear interpolation.

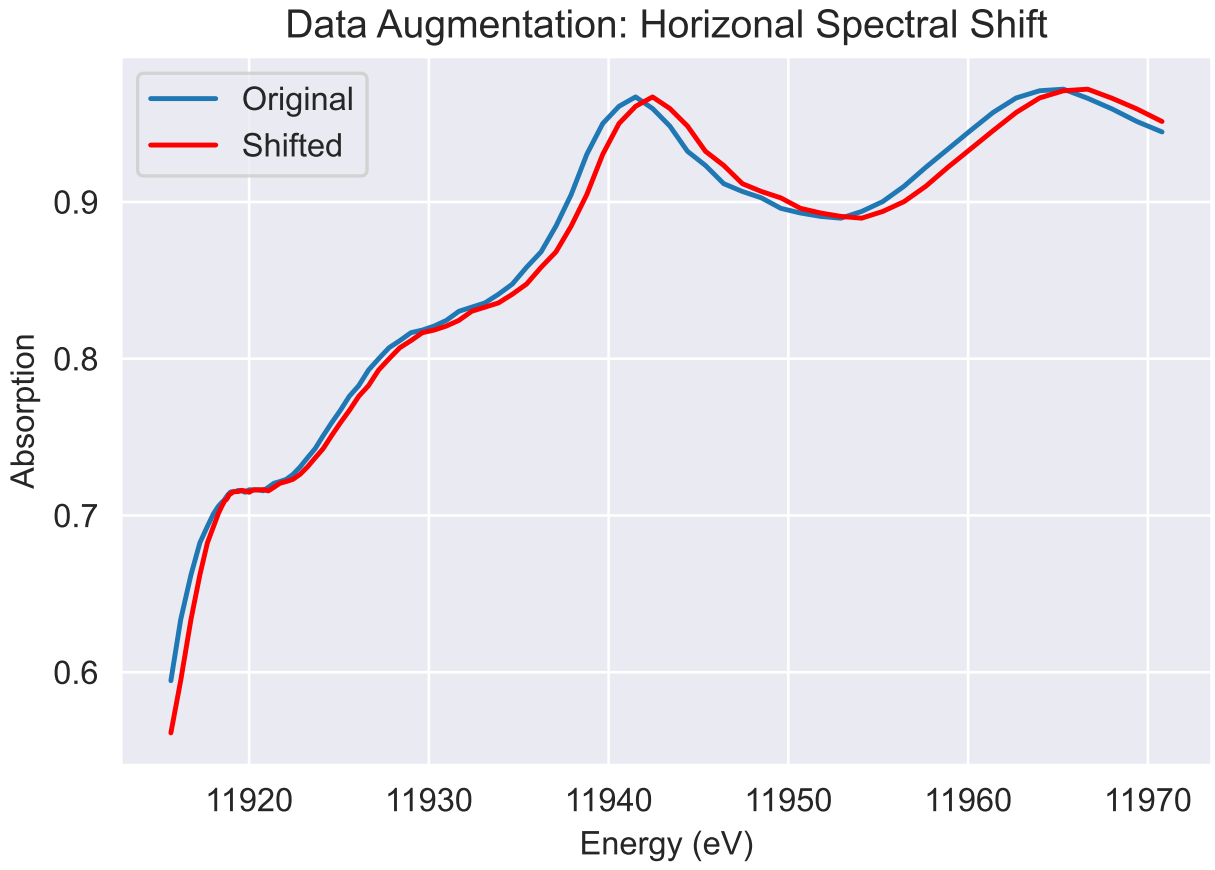

Data Augmentation: Horizonal Spectral Shift
Original
Shifted
Absorption
0.9
0.8
0.7
0.6
11920
11930
11940
11950
11960
11970
Energy (eV)

Figure 4.3: ...

Appendix A

Select Python Code

The code for this thesis is well-documented and stored in a GitHub^{TM} repository. This appendix includes short descriptions of the major scripts written for this thesis, as well as the inclusion of a selected few major functions. The mathematical formulas and descriptions for how these scripts work are included in the main chapters of this thesis; however, there are instances where looking at the code can be useful in understanding the approach. Also included are the file structures that the scripts generate or expect to find. This is important to know if actually running the scripts.

A.1 distortionator.py

Given `feff.inp` file, generate many `feff.inp` files — each with a structure slightly shifted radially outwards (or inwards) from the original structure. File structure is organized as the following:

```
BNL
|   distortionator.py
|   gaussianator.py
|
└──DATA
|   |
|   |
|   └──Original_Structure
|   |   |   feff.inp
|   |
|   └──minus_pt_0001
|   |   |   feff.inp
|   |
|   └──plus_pt_0001
|   |   |   feff.inp
|   |
|   ....
```

A.2 gaussianator.py

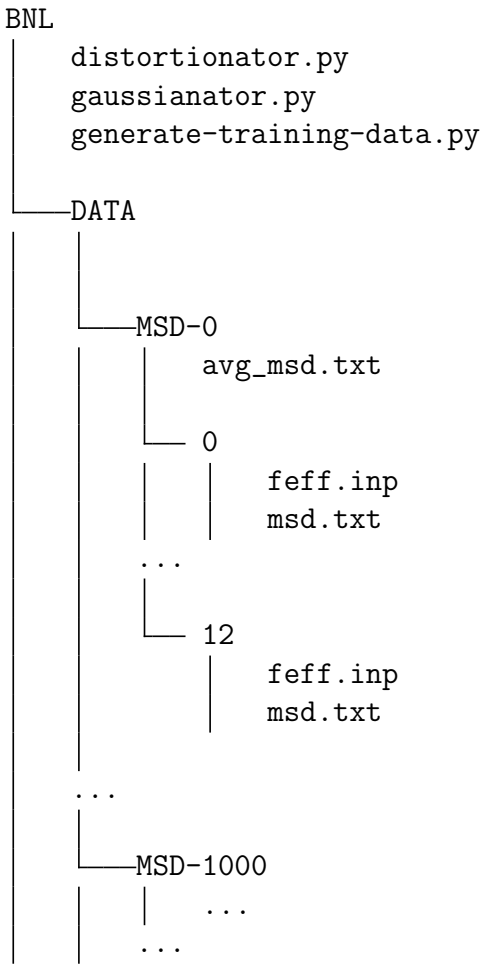
Take all the `xmu.dat` files (each one represents the sprectrum from the $\Delta\rho$ shifted crystrals) and generates many gaussian averaged XANES spectra. One file per different standard deviation of the gaussian. The File structure is organized as follows:

```
BNL
|   distortionator.py
|   gaussianator.py
|
└──DATA
|   |
|   └──Averaged_Spectra
|       |
|       |   sigma_0001.csv
|       |   sigma_0002.csv
|       |   ...
```

```
# Inputs: -----
# dataframe df = the already mapped concat dataframe of all the different
    delta_rho shifted feff xanes
# float64 mean = mean of gaussian.
# float64 std = standard deviation of gaussian
# float64 skewness = skew parameter of stats.skewnorm
# Outputs: ----
# dataframe df_weighted2 = one dataframe. It is one distribution-weighted
    spectra with cols=['omega','mu']
# float64 avg_MSD = the mean squared displacement of the skewnorm-averaged
    spectrum
# Note a skewness of 0, sigma 1, and mean 0 is a standardized normal
    distribution.
def weight_by_distribution2(df_concat, mean, std, skewness):
# calculate the MSD -------------------
# get the bin heights for all the bins
bin_heights = np.array([stats.skewnorm.pdf(nn_dist, loc=mean, scale=std,
    a=skewness) for nn_dist in BINS])
normalization_factor = np.sum(bin_heights)
# same as sum(bin_height_i * nn_bond_dist)/(sum(bin_heights))
weighted_nn_dist_mean = np.dot(bin_heights, BINS) / normalization_factor
# same as sum(bin_height_i * ( nn_bond_dist_i - mean_bond_dist )^2
    )/sum(bin_heights)
sq_dif = np.square(np.subtract(BINS, weighted_nn_dist_mean))
avg_msd = np.divide(np.dot(bin_heights, sq_dif), normalization_factor)
# now do the spectrum -----------------
df_weighted2 = pd.DataFrame(data={'omega': df_concat.loc['0'].omega, 'mu':
    np.zeros(df_concat.loc['0'].omega.shape[0])})
for shift, bin_height in zip(SHIFTS, bin_heights):
    df_weighted2.mu += df_concat.loc[shift]['mu'].multiply(bin_height)
df_weighted2.mu /= normalization_factor # correct for the sum, so the area
    under the PDF=1
return df_weighted2, avg_msd
```

A.3 generate-training-data.py

This script generates the FEFF input files (`feff.inp`) for the disordered structures—i.e. the true-disordered structures, NOT the distorted structures used for the skew-norm averaging.



```
# Inputs: -----
# pandas dataframe df = the unshifted dataframe with spherical coordinates
# float shift_sigma = the width of the np.random.normal distribution from which
    shift distances are chosen
# Outpts: ----
# np arrays x, y, z = the shifted coorinates
# Notes: -----
# shift_val = radius of sphere project new point onto = distance of new
    disordered atom from original location
def gen_random_delta_rho_shift(df, shift_sigma):
    df_temp = df.copy()
    # SHIFT
    df_temp['shift_val'] = np.random.normal(loc=0, scale=shift_sigma,
        size=df_temp.shape[0])
    df_temp['theta'] = 6.28 * np.random.random_sample(df_temp.shape[0])
    df_temp['phi'] = 6.28 * np.random.random_sample(df_temp.shape[0])
    # Calculate the new coordintes
    df_temp['x'] +=
        round(df_temp.shift_val*np.sin(df_temp.phi)*np.cos(df_temp.theta), 5)
    df_temp['y'] +=
        round(df_temp.shift_val*np.sin(df_temp.phi)*np.sin(df_temp.theta), 5)
    df_temp['z'] += round(df_temp.shift_val*np.cos(df_temp.phi), 5)
    # turn to numpy array
    x1 = df_temp.loc[:, 'x'].values
    y1 = df_temp.loc[:, 'y'].values
    z1 = df_temp.loc[:, 'z'].values
    return x1, y1, z1
```

```
# Inputs: -----
# str folder path of one structure (contains 13 subfolders, one for each absober)
# Outputs: ----
# Returns float64 MSD, the mean-squared-displacement of the structure.
def do_one_structure(folder):
    bonds = set()
    rhos = []
    duplicates = 0
    for i in range(13):
        subfolder_path = os.path.join(folder, str(i))
        file = os.path.join(subfolder_path, 'feff.inp')
        df_absorbers = (load_initial_file(file)
                        .pipe(to_spherical)
                        .query('rho < 3.5')
                        )
        for index, row in df_absorbers.iterrows():
            option1 = (df_absorbers[df_absorbers.absorber==0].index[0], index)
            option2 = (index, df_absorbers[df_absorbers.absorber==0].index[0])
            if df_absorbers[df_absorbers.absorber==0].index[0] == index:
                pass
            elif option1 in bonds or option2 in bonds: # duplicate bond found
                duplicates += 1
            elif option1 not in bonds or option2 not in bonds: # new bond found
                bonds.add(option1)
                rhos.append(row.rho)
    if len(rhos) != 120 or len(bonds) != 120:
        raise Not120BondsException(len(rhos), len(bonds))
    dif = np.array(rhos) - np.mean(arr)
    squared = np.square(dif)
    summed = np.sum(squared)
    msd = summed/len(rhos)
    with open(os.path.join(folder, 'fixed_avg_msd.txt'), "w") as f:
        f.write(str(msd))
    return msd
```

A.4 create-g(r).ipynb

This iPython notebook loops through all the disordered structures and creates a histogram of nearest neighbor distances for each structure. Because there are 13 absorbers, each of which has 13 nearest neighbors, there are a total of 169 bond lengths. Many of these bonds are shared with absorbers and would be counted twice if one were not careful. There are only 120 unique bonds for the nearest neighbors of each atom in the first shell. This script

keeps track of all the unique bonds to ensure no bond-length is counted twice.

A.5 nn.ipynb

The neural network, a Jupyter notebook.

A.6 nn-buddy.py

The sole purpose of this python script is to be imported by `nn.ipynb`. The script contains many useful helper functions that take care of data-loading, plotting, and linear interpolation of experimental data on the same energy mesh used for the training sample. This way, the

References

[1] J. Timoshenko, D. Lu, Y. Lin, and A. I. Frenkel. *Supervised machine-learning-based determination of three-dimensional structure of metallic nanoparticles.* The Journal of Physical Chemistry Letters, 8(20):5091–5098 (2017).

[2] F. R. Elder, A. M. Gurewitsch, R. V. Langmuir, and H. C. Pollock. *Radiation from electrons in a synchrotron.* Phys. Rev., 71:829–830 (1947).

[3] D. J. Gardenghi et al. *Synchrotron radiation-based spectroscopic investigation of the electronic and geometric structures of iron-sulfur clusters, particles, and minerals.* Ph.D. thesis, Montana State University-Bozeman, College of Letters & Science (2012).

[4] H. Fricke. *The k-characteristic absorption frequencies for the chemical elements magnesium to chromium.* Physical Review, 16(3):202 (1920).

[5] G. Hertz. *ueber die absorptionsgrenzen in derl-serie.* Zeitschrift fuer Physik, 3(1):19–25 (1920).

[6] J. J. Rehr and R. C. Albers. *Theoretical approaches to x-ray absorption fine structure.* Reviews of modern physics, 72(3):621 (2000).

[7] M. Newville. *Fundamentals of xafs.* Reviews in Mineralogy and Geochemistry, 78(1):33–74 (2014).

[8] K. Klementev. *Xafs spectroscopy. i. extracting the fine structure from the absorption spectra.* arXiv preprint physics/0003086 (2000).

[9] B. K. Teo. *EXAFS: Basic Principles and Data Analysis*, volume 9. Springer Science & Business Media (2012).

[10] M. Rühle and M. Wilkens. *Transmission electron microscopy.* In R. W. Cahn and P. Haasenae (editors), *Physical Metallurgy (Fourth Edition)*, chapter 11, pp. 1033–1113. North-Holland, Oxford, fourth edition edition (1996).

[11] Y. Lin, M. Topsakal, J. Timoshenko, D. Lu, S. Yoo, and A. I. Frenkel. *Machine-learning assisted structure determination of metallic nanoparticles: A benchmark.* In *Handbook on big data and machine learning in the physical sciences: Volume 2. Advanced Analysis Solutions for Leading Experimental Techniques*, pp. 127–140. World Scientific (2020).

[12] J. Timoshenko, A. Anspoks, A. Cintins, A. Kuzmin, J. Purans, and A. I. Frenkel. *Neural network approach for characterizing structural transformations by x-ray absorption fine structure spectroscopy.* Physical review letters, 120(22):225502 (2018).

[13] D. Uzio and G. Berhault. *Factors governing the catalytic reactivity of metallic nanoparticles.* Catalysis Reviews, 52(1):106–131 (2010).

[14] M. Jørgensen and H. Grönbeck. *Strain affects co oxidation on metallic nanoparticles non-linearly.* Topics in Catalysis, 62(7):660–668 (2019).

[15] M. Newville. *EXAFS Analysis Using feff and feffit.* Journal of synchrotron radiation, 8(2):96–100 (2001).

[16] K. Jorissen and J. J. Rehr. *New Developments in FEFF: FEFF9 and JFEFF.* In *Journal of Physics: Conference Series*, volume 430, p. 012001. IOP Publishing (2013).

[17] J. J. Rehr, J. J. Kas, F. D. Vila, M. P. Prange, and K. Jorissen. *Parameter-free Calculations of X-ray Spectra with FEFF9.* Physical Chemistry Chemical Physics, 12(21):5503–5513 (2010).

[18] A. Azzalini and A. Capitanio. *Statistical applications of the multivariate skew normal distribution.* Journal of the Royal Statistical Society: Series B (Statistical Methodology), 61(3):579–602 (1999).

[19] P. Virtanen et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python.* Nature Methods, 17:261–272 (2020).

[20] C. R. Harris et al. *Array programming with NumPy.* Nature, 585(7825):357–362 (2020).

[21] Y.-G. Kim, S.-K. Oh, and R. M. Crooks. *Preparation and characterization of 1- 2 nm dendrimer-encapsulated gold nanoparticles having very narrow size distributions.* Chemistry of Materials, 16(1):167–172 (2004).

[22] Y. Guo et al. *Uniform 2 nm gold nanoparticles supported on iron oxides as active catalysts for co oxidation reaction: structure–activity relationship.* Nanoscale, 7(11):4920–4928 (2015).

[23] C.-W. Pao et al. *Photoconduction and the electronic structure of silica nanowires embedded with gold nanoparticles.* Physical Review B, 84(16):165412 (2011).

[24] G. Carleo et al. *Machine learning and the physical sciences.* Reviews of Modern Physics, 91(4):045002 (2019).

[25] C. Szegedy, A. Toshev, and D. Erhan. *Deep Neural Networks for Object Detection* (2013).

[26] G. Hinton et al. *Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups.* IEEE Signal Processing Magazine, 29(6):82–97 (2012).

[27] S. Ruan, J. O. Wobbrock, K. Liou, A. Ng, and J. Landay. *Speech is 3x faster than typing for english and mandarin text entry on mobile devices.* arXiv preprint arXiv:1608.07323 (2016).

[28] J. Schmidhuber. *Deep learning in neural networks: An overview.* Neural networks, 61:85–117 (2015).

[29] A. S. Lokman and M. A. Ameedeen. *Modern chatbot systems: A technical review.* In *Proceedings of the future technologies conference*, pp. 1012–1023. Springer (2018).

[30] G. Lopez, L. Quesada, and L. A. Guerrero. *Alexa vs. Siri vs. Cortana vs. Google Assistant: a comparison of speech-based natural user interfaces.* In *International Conference on Applied Human Factors and Ergonomics*, pp. 241–250. Springer (2017).

[31] R. Sarikaya. *The technology behind personal digital assistants: An overview of the system architecture and key components.* IEEE Signal Processing Magazine, 34(1):67–81 (2017).

[32] M. Abadi et al. *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015). Software available from tensorflow.org.

[33] A. Paszke et al. *Pytorch: An imperative style, high-performance deep learning library.* In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alche-Buc, E. Fox, and R. Garnett (editors), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc. (2019).

[34] J. Kukavcka, V. Golkov, and D. Cremers. *Regularization for deep learning: A taxonomy.* arXiv preprint arXiv:1710.10686 (2017).

[35] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. *Dropout: a simple way to prevent neural networks from overfitting.* The journal of machine learning research, 15(1):1929–1958 (2014).

[36] S. Cai, Y. Shu, G. Chen, B. C. Ooi, W. Wang, and M. Zhang. *Effective and efficient dropout for deep convolutional neural networks.* arXiv preprint arXiv:1904.03392 (2019).

[37] H. Wu and X. Gu. *Towards dropout training for convolutional neural networks.* Neural Networks, 71:1–10 (2015).

[38] A. Cauchy et al. *Méthode générale pour la résolution des systemes d'équations simultanées.* Comp. Rend. Sci. Paris, 25(1847):536–538 (1847).

[39] I. Loshchilov and F. Hutter. *Decoupled weight decay regularization.* Conference Paper at ICLR 2019 (2019).

[40] S. Ruder. *An overview of gradient descent optimization algorithms.* CoRR, abs/1609.04747 (2016).

[41] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. *Parallelized stochastic gradient descent.* In *NIPS*, volume 4, p. 4. Citeseer (2010).

[42] D. R. Wilson and T. R. Martinez. *The general inefficiency of batch training for gradient descent learning.* Neural networks, 16(10):1429–1451 (2003).

[43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning representations by back-propagating errors.* nature, 323(6088):533–536 (1986).

[44] N. Qian. *On the momentum term in gradient descent learning algorithms.* Neural networks, 12(1):145–151 (1999).

[45] J. Duchi, E. Hazan, and Y. Singer. *Adaptive subgradient methods for online learning and stochastic optimization.* Journal of machine learning research, 12(7) (2011).

[46] C. Igel and M. Hüsken. *Improving the rprop learning algorithm.* In *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*, volume 2000, pp. 115–121. Citeseer (2000).

[47] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. *The marginal value of adaptive gradient methods in machine learning.* arXiv preprint arXiv:1705.08292 (2017).

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

[49] J. Thaller. *Toward a direct measurement of strain-dependent surface stress in soft solids* (2019).

[50] R. Larson and B. Farber. *Elementary statistics.* Pearson Education Canada (2019).

[51] S. Ioffe and C. Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* In *International conference on machine learning*, pp. 448–456. PMLR (2015).

[52] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. *How does batch normalization help optimization?* In *Proceedings of the 32nd international conference on neural information processing systems*, pp. 2488–2498 (2018).

[53] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. *How does batch normalization help optimization?* In *Proceedings of the 32nd international conference on neural information processing systems*, pp. 2488–2498 (2018).

[54] J. Kohler, H. Daneshmand, A. Lucchi, T. Hofmann, M. Zhou, and K. Neymeyr. *Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization*. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 806–815. PMLR (2019).

[55] G. Yang, J. Pennington, V. Rao, J. Sohl-Dickstein, and Schoenholz. *A mean field theory of batch normalization*. arXiv preprint arXiv:1902.08129 (2019).

[56] D. A. Van Dyk and X.-L. Meng. *The art of data augmentation*. Journal of Computational and Graphical Statistics, 10(1):1–50 (2001).

[57] S. Frühwirth-Schnatter. *Data augmentation and dynamic linear models*. Journal of time series analysis, 15(2):183–202 (1994).

[58] L. Perez and J. Wang. *The effectiveness of data augmentation in image classification using deep learning*. arXiv preprint arXiv:1712.04621 (2017).

[59] M. A. Tanner and W. H. Wong. *The calculation of posterior distributions by data augmentation*. Journal of the American statistical Association, 82(398):528–540 (1987).

[60] C. Shorten and T. M. Khoshgoftaar. *A survey on image data augmentation for deep learning*. Journal of Big Data, 6(1):1–48 (2019).

[61] S. Bozinovski. *Reminder of the first paper on transfer learning in neural networks, 1976*. Informatica, 44(3) (2020).

[62] S. J. Pan and Q. Yang. *A survey on transfer learning*. IEEE Transactions on knowledge and data engineering, 22(10):1345–1359 (2009).

[63] L. Torrey and J. Shavlik. *Transfer learning*. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pp. 242–264. IGI global (2010).

[64] J. Somers. *The Pastry A.I. That Learned to Fight Cancer*. The New Yorker (2021).

[65] M. T. Rosenstein, Z. Marx, L. P. Kaelbling, and T. G. Dietterich. *To transfer or not to transfer*. In *NIPS 2005 workshop on transfer learning*, volume 898, pp. 1–4 (2005).

[66] M. L. Boas. *Mathematical methods in the physical sciences; 3rd ed.* Wiley, Hoboken, NJ (2006).

[67] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman. *1d convolutional neural networks and applications: A survey*. Mechanical Systems and Signal Processing, 151:107398 (2021).

[68] W. Rawat and Z. Wang. *Deep convolutional neural networks for image classification: A comprehensive review*. Neural computation, 29(9):2352–2449 (2017).

[69] R. Zabih. *Cs1114 section 6: Convolution*. Cornell University (2013). Cornell University.

[70] O. S. Kayhan and J. C. v. Gemert. *On translation invariance in cnns: Convolutional layers can exploit absolute spatial location*. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14274–14285 (2020).

I declare that I prepared and wrote this thesis work independently and with no other means than those referenced in the text.

Jeremy K. Thaller	Date