# Cloud Opacity Calculation using Machine Learning

Olaf Willocx, Robin Van den Bliek

April 2021

**Abstract**

The potential of ML (machine learning) in accelerating Mie scattering codes is researched, for the particular application of calculating radiative feedback in exoplanet atmospherers. Mie scattering theory describes the interaction between electromagnetic plane waves and particles of microscopic size. This model plays a role in a larger scheme to understand atmospheric exoplanet data measured during transit.

Het potentiaal van ML (machine learning) om Mie-verstrooiingscodes te versnellen voor de specifieke toepassing van het berekenen van stralingsfeedback in exoplaneetatmosferen, wordt onderzocht. Mie verstrooiingstheorie beschrijft de interactie tussen elektromagnetische vlakkegolven en deeltjes van microscopische grootte. Dit model speelt een rol in een groter schema om atmosferische exoplaneet data gemeten tijdens transit te begrijpen.

## 1 Introduction

The viability of numerics in astronomy has been apparent for thousands of years. See for example the careful measurement, interpolation, and extrapolation work of Meton of Athens on lunar cycles, leading to the $\sim 19$ year lunar Metonic cycle. Over the last few decades, astronomy has seen a massive growth in both popularity and success, correlated with equally impressive progress in computer science. Today computers allow billions of calculations to be made in a matter of seconds, inconceivable even 70 years ago. However, even in modern times, the amount of data involved in many applications remains enormous, resulting in computers being unable to process it in any realistic time span.[19] This report explores ML methods, and particularly neural networks, as a solution to a problem that would otherwise be hard to solve efficiently with more classical numerical techniques. The underlying theory of ML is discussed, and the results are applied to the problem of exoplanet cloud opacity.

We use tools provided by modern computer science to accelerate numerical work in astronomy. Specifically we are using ML to significantly speed up

Mie scattering codes. These codes implement numerical techniques to solve Maxwell's equations for the scattering of electromagnetic plane waves on spherical particles. When calculating radiative feedback, fast approximate results are required. The models producing these approximate results are to be used in a grander numerical simulation seeking to understand light observed from exoplanet atmospheres during transits.

The results of this project can serve as an indirect way to verify cloud composition. In certain situations, they can also tell us something about the composition of the planet itself.

Exoplanet atmospheres vary in chemical composition. Many different molecules may be present in clouds, but usually, a small number dominate. A set of not more than 20 molecules can accurately describe the majority of exoplanet clouds. These are molecules that tend to cluster together; they form dust particles of various size and distribution. This ultimately leads to different wavelength-dependent cloud opacities; some clouds may be visible the ultraviolet range but invisible in the infrared range, and vice versa.

# 2 Theory

## 2.1 Mie Scattering Codes & Data

### 2.1.1 What is Mie Scattering?

Mie scattering is a solution to Maxwell's equation applied to scattering in situations where simplifications are not trivial, i.e. comparable particle size and EM radiation wavelength.

### 2.1.2 Data from Simulation

The data used to train models is numerical data from a Mie scattering simulation. This simulation (by Sven Kiefer, KU Leuven) utilizes the 'aa2' and 'shexqnn2' subroutines from MieX (2004), and also depends on DMiLay.[1] MieX is based on N.V. Voshchinnikov's work (2002), which itself uses under more Loskutov's work (1971).[2] DMiLay is a double precision floating point version of MiLay (1981), which is based on Dave, J.V.'s work at IBM (1968, 1969).[3, 4, 5]

### 2.1.3 Input and Output Data

The input data ultimately consists of the EM radiation wavelength $\lambda$, the overall size of the dust particles being scattered $d$, the standard deviation of the

distribution of the dust sizes $d$[1], and sixteen molecule[2] mixture ratios[3].

The output data consists of relative, unitless measurements of scattering cross sections and absorption coefficients.

These form the input and output data of all models.

## 2.2 Performance Requirements

The goal of this model is to quickly compute approximate scattering and absorption data (it can in principle be used for the full 3D solution) used to compute radiative feedback. This requires accuracy on the order of 10% relative error. If the model performs very well, it may also be used for calculating transmission spectra, which requires error significantly less than 1%, and small errors across the resonsance structures. This is not the goal, but acts as an upper limit on what is useful; error significantly smaller than 1% is not worth optimizing. The Mie scattering codes produce high quality data with errors certainly no larger than the requirements above. The requirement to compute quickly means the complexity of the model must be minimized and kept reasonable.

## 2.3 Mixture Models

The distribution of the random molecular mixture ratios is of major concern to the viability of the trained model. The random mixture ratios should be realistic if they are to be used in an unbiased way on real exoplanet data. Realistic clouds are often dominated by at most a few molecules. The article 'Mixture Models in Astronomy' by Kuhn and Feigelson gives a good overview of the general theory and application in astronomy.[6] Chronologically, mixture models were the last point considered to optimize the model's performance, and the problem was solved before finding out about the more general theory, so our solution may be unorthodox.

The simplest (and original) way of producing sixteen random mixture ratios is to generate 16 uniform random $[0, 1]$ numbers, and to normalize the sum. We noticed this results in unrealistic mixture distributions. See Figure 1. On average we expect a ratio of $1/16 \approx 0.06$, which can be seen on the figure as well. However, the ratio varies roughly between 0 and 0.1 and then drops off rapidly. In reality we expect ratios higher than 0.1 to occur, as most clouds are dominated by only a few molecules, not 16. Clearly a uniform distribution does not yield realistic results; molecules never dominate and the distributions are sharp.

[1] Realistic dust comes in all shapes and sizes, which the simulation accounts for. Although Mie scattering is concerned with scattering on spheres, multiple layers can be used to simulate more complex geometries ('stratified spheres'), which DMiLay is concerned with.

[2] The particular molecules are TiO2, Mg2SiO4, SiO, SiO2, Fe, Al2O3, CaTiO3, FeO, FeS, Fe2O3, MgO, MgSiO3, CaSiO3, Fe2SiO4, C, and KCl.

[3] This set is fifteen-dimensional; the sum is always one, signifying that clouds contain only these sixteen molecules.

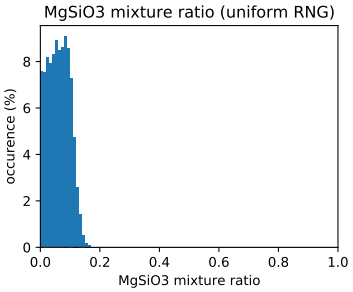


Figure 1: Distribution of enstatite[a] mixture ratio over $2^{14}$ samples with uniform RNG.

[a]Enstatite is often used as an example molecule throughout the whole study because its behavior is quite average.

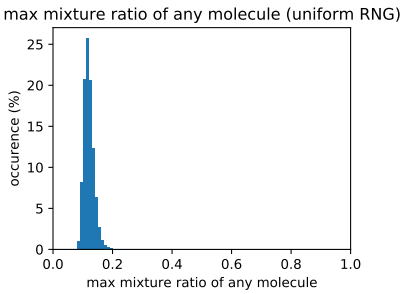


Figure 2: Distribution of the maximum mixture ratio of any molecule over $2^{14}$ samples with uniform RNG.

Also interesting is the over-all distribution of the maximum of mixture ratios across all molecules (Figure 2). Often a small number of molecules dominates, so high occurence of large maximum mixture ratios is desired, and an over-all wide range of various maximum mixture ratios.

The solution we came up with is to introduce an additional step between uniform random number generation and normalization, where some non-linear function is applied. The function that performs particularly well is $\tan(\pi x/2)$, as it maps $[0, 1] \to [0, \infty)$.[4]

The figures below show the results achievable (and used) with said solution.

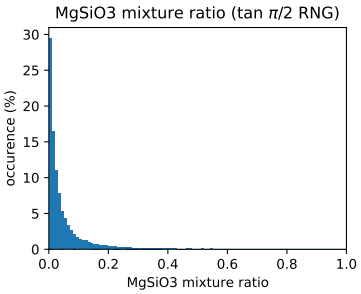


Figure 3: Distribution of enstatite mixture ratio over $2^{14}$ samples with $\tan(\pi x/2)$ RNG.

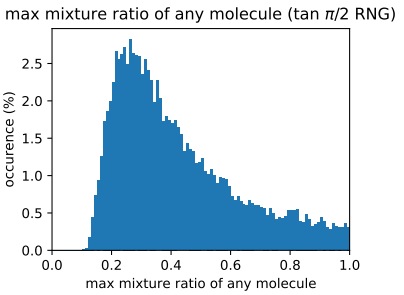


Figure 4: Distribution of the maximum mixture ratio of any molecule over $2^{14}$ samples with $\tan(\pi x/2)$ RNG.

[4]The raw function often produces results that are too extreme, causing the simulation to fail. As a result, an additional function before $\tan(\pi x/2)$ is used to transform $[0, 1] \to [a, b]$ with $a$ slightly more than 0 and $b$ slightly less than 1. Although a seemingly small point, this is a massive point of optimization. If the simulation would not fail frequently, much more realistic mixture ratios still can be generated.

## 2.4 Basics of Machine Learning

There are many different ML methods. Each ultimately results in a function (called the 'model') with some internal structure governed by parameters. Some of these parameters might be trainable. The structure itself might also be trainable, i.e. the structure may be parameterized by so-called hyperparameters. ML is the training of these parameters through experience, to obtain desired behavior from the model. This experience comes in the form of input and output data; abstractly, the machine learns to associate certain inputs with certain outputs. In supervised ML, data is tagged (for categorical data) or labeled (for numerical data). In unsupervised ML there is no tagging or labeling, e.g. because both input and output data are incomprehensible, or because there is no goal less vague than 'learn something'. In unsupervised ML, models learn to interpret things for themselves. In supervised ML, both the input and output data are usually understood by humans. We will be dealing with supervised ML.

There exist three distinct categories of data. These are categorical data, numerical data, and higher dimensional sparsely complex data[5]. Take for example a model seeking to predict house prices. It may be given as input a zip code and street name (categorical data), price and distance to the nearest major city (numerical data), and some images (higher dimensional sparsely complex data). Each category requires different treatment. We will be dealing with numerical data.

Training a supervised model comes down to defining the goal and making progress towards it. The goal is for the model to output the right data for the given input. To learn, the machine utilizes a loss function (e.g mean squared), and fits data to minimize the loss. A loss function is a (positive) function describing how well the model is performing. Its inputs are the expected output and the model's output. Zero loss occurs when the inputs match.

The loss function is passed on to an algorithm capable of using this information to modify the parameters of the model in such a way that it will probably perform better next time around, i.e. make progress. Such an algorithm is called an 'optimizer'. Almost all of these optimizers are based on some form of a gradient descent algorithm, so the loss function needs to be continuous and smooth. Abstractly this can be thought of as finding a minimum in the often highly-dimensional loss function; hopefully the global minimum, since the algorithm can converge to a local minimum (and hopefully the global minimum is near zero) or a saddle point. See Figure 5 for a 2D visualization.

[5]Take an image for example; highly compressible and sparsely complex.

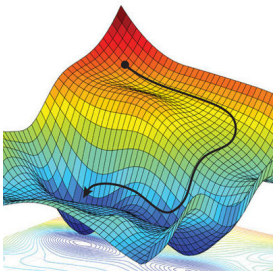

Figure 5: Ideal descent towards global minimum for function with 2D input.

Input and output data is split into training and testing sets. The model never gets to see the testing sets during training so that it can't cheat by memorizing the right answers. The testing data can thus be seen as unbiased, contrary to the training data.

### 2.4.1 Example & Overfitting

A common culprit in ML is overfitting the data, which leads to an overestimation of the influence of each data point in the training set, similar to how a polynomial of degree 9 would be overfitting 10 data points that are almost, but not exactly, on a straight line. Alternatively, there is the problem of underfitting, which is similar to choosing a linear fit when the data is distributed non-linearly.

Overfitting is commonly solved by calculating the accuracy (or other metric) for different values of the hyperparameters of a model, and choosing the one with the highest accuracy. The accuracy is simply the fraction of correct predictions, and applies only to classification problems (i.e. not regression problems like ours), but it is the simplest way to understand.

As an example, consider that we want a machine to be able to predict the breed of a dog based on certain features; these could be the height, weight, tail length, claw width, etc. The input set contains values for all the features, which practically is a $n$-dimensional array where $n$ is the number of features. The output variable is the breed of dog, a 1D array (where breeds are represented by numbers). To learn the pattern, the machine utilizes a loss function, and fits data to minimize the loss. In this example, the fit could be the $k$-nearest neighbors method. Here each point of $n$-space is assigned an output value based on the most frequent output value among the $k$ closest data points. See Figure 6. This is an example of the 1-nearest neighbor (1NN) algorithm, where each point of space gets classified according to whatever the nearest data point is classified as. The output variable is the color, and the input variables are spatial coordinates.

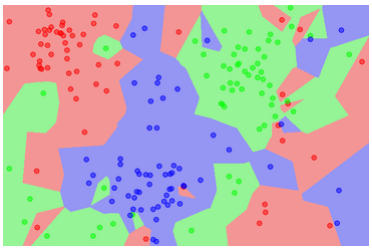

Figure 6: 1NN algorithm. Each point of space adopts the value (color here) of the nearest data point. (Wikipedia, '$k$-nearest neighbors algorithm')

The resulting color pattern is erratic and not smooth. This suggests overfitting. The left top corner is filled with red data points, with a single green point exception. We do not wish for this green point to color the surrounding area green, since it likely is nothing but a statistical fluctuation. We want the model to turn this area red, since almost all nearby points are red. A better fit for this problem is the 5NN algorithm, as showcased in Figure 7. Now, the area surrounding the green point is colored red, since most neighboring data points are red. The 5NN algorithm suffers less from overfitting compared to 1NN.

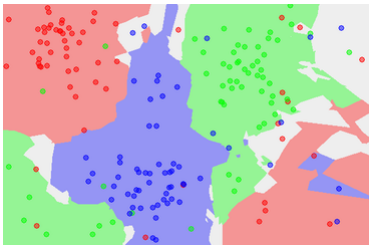

Figure 7: 5NN algorithm. Each point of space adopts the value (color here) of the average of the 5 nearest data point

An example of an underfit would be giving every point in space the same color, namely the color that most frequently occurs among all data points. This is definitely a fit, but better fits can be made given the with less over-all error. Not too little error tho, since that would become overfitting. There exists a balance between the two.

## 2.5 Preprocessing

Preprocessing is the processing of input and output data before teaching.

### 2.5.1 Feature Scaling

Feature scaling, i.e. data normalization, is used to treat each input and output variable equally. In the above 'dog breed' example, height and claw width were some of these features that must be scaled. Height is roughly 2 orders of magnitude greater than claw width. For the classification problem, this would

make height a factor with more weight than claw width; relative errors matter more for height. Taking the horizontal axis as claw width and the vertical axis as height, the data points fall on an almost vertical line. The problem is that the nearest neighbor for any new data point will be almost completely determined by the height. Letting the claw width vary from left to right, the nearest neighbour will remain unchanged. This is problematic since it could very well be that the claw width is more important than the height.

The solution is to normalize the data in some way. This can be done by dividing each value of a variable by the average value for that variable, and this for all variables. In this way different input variables are treated on equal footing. Another method of normalization is minmax scaling, where the minimum and maximum of a dataset are used to force all values into the range $[0, 1]$. In this particular application, theoretical minmax scaling can even be used, where min and max signify the most extreme values possible from the random number generators.

Sometimes different variables deserve different weights, if we know that one input variable is less important for classification, we could scale it by a factor of $x$, where $0 < x < 1$ (e.g. if age were an input variable for the above example of dog breeds, this won't tell us much so we could scale it, or even leave it out altogether, reducing the dimensionality). Note that this does not result in full normalization since the importance of variables with high variance is overestimated, but it is often a good approximation.

### 2.5.2 Feature Standardization

Often (mistakenly) subclassed as feature scaling, feature standardization is a form of model normalization. Feature standardization is not well understood and disagreed upon, so we describe here only its effects, and a number of methods that lead to these effects (i.e. feature standardization implementations).

With feature standardization, models train at a much higher rate, and often with better asymptotic performance.[20] It prevents dead neurons (neurons becoming untrainable due to negligible slope) and neurons with astronomical parameters (which overpower nearby neurons, making the neighbors practically dead also). Having such neurons intuitively reduces the complexity and thereby capability of a network, hurting performance. Also, the training rate is affected, since training rate is slope-dependent for most optimizers.

In the context of data preprocessing, feature scaling refers to forcing data to be standard normally distributed by subtracting data by its mean and dividing it by the standard deviation, then returning it to a general normal distribution with a now trainable mean and standard deviation.

Another form of feature standardization is batch normalization. It normalizes locally on a certain number of samples called the batch size.

Feature standardization may also (confusingly) refer to layer normalization.[20] Here parameters in a layer are forced to be standard normally distributed, then the ouptut is multiplied by parameter $\gamma$ (standard deviation) and added to $\beta$ (mean), which are trainable parameters.

# 3 Artificial Neural Networks

The example from Section 2.4.1 concerned a classification problem. For continuous, non-classification problems, such as estimating the best value for opacities, all ideas considered still apply, but the particular ML model, loss function and evalulation metrics change. The $k$-nearest neighbors model may be replaced by a neural network or linear regression. The loss function can be mean square or mean absolute error. Now that the jargon and universal ML concerns have been introduced, it is time to delve into artificial neural networks (ANNs); the ML model of choice for this application.

## 3.1 Why Artificial Neural Networks?

The common ML algorithms for regression problems are linear regression, support-vector machines (SVMs), random forests, and ANNs.

Linear regression is out of the question since the opacity problem is non-linear.

SVMs typically perform well with small datasets, while neural networks do better with large datasets. Since our datasets are large, the primary benefit of SVMs disappears.

Random forests are based on decision trees, so they are more suited for classification problems. However, they can still perform well for regression problems. A more convincing reason to choose artificial neural networks instead of random forests, is that random forests have fewer parameters to train and tweak (hyperparameters in particular), meaning less control in optimalization. For relatively few input and output dimensions, hyperparameter training is not out of the question, an opportunity we'd miss out on when opting for random forests.

## 3.2 Artificial Neural Network Architecture

Neural networks consist of 'neurons' (also called 'cells', or 'nodes'), typically in layers. The number of neurons in the last layer is the output dimensionality. In this way, the value returned by this layer corresponds to the neural network's prediction of whatever the output variables are. There are different ways to connect layers, e.g. each cell of layer two could receive input from each cell of layer one, or only from two cells. Layer one could connect to just layer two, or it could connect to layer two and layer three, etc. The possibilities are endless, and so is the complexity. With each connection between any two cells (a nerve) is associated a weight, corresponding to the relative importance of that connection. Cells can also be given self-owned properties like a bias (number added to every result going through the cell from any connection), or threshold (cell outputs zero until it receives a sufficiently large value). To give ANNs more competence, layer results are often wrapped in non-linear functions called 'activation functions', allowing much more complicated patterns to be learned.[18] For the same reason solving non-linear equations is hard, neural networks benefit from them; they are simple structures that allow complex behavior. Many different choices

exist, see Figure 11. There is no single best activation function. What they all have in common is that they are continuous[6], fast to (approximately) compute, and can turn any linear equation into a nightmare (difficult to solve problems involving such an equation).

Mathematically a 'dense' layer's weights (from any neuron to any previous layer neuron) can be represented as a matrix $A$, the value of the previous layer as a column vector $x$, and the biases as another column vector $b$, all wrapped in a non-linear function $f$. Passing through a layer looks like $f(Ax + b)$. Passing through multiple sequentially (sequential ANN) looks like $f_2(A_2 f_1(A_1 x_1 + b_1) + b_2)$, where $A_1$ and $A_2$ do not have to be square, or even the same shape.

All these weights, biases, and thresholds, but also the number of neurons per layer, the number of layers, the choice of activation function, etc. are trainable parameters. Some by the optimizer (parameters), some manual (hyperparameters).

We consider first the case of perceptrons. Perceptrons can be used as logic gates, i.e. NAND gates, which can be used to add bits. NAND gates, and therefor also perceptrons, are universal for computations. This brings nothing new to the table; computers already operate by NAND gates. The reason they can be used for logic gates is because they have a strictly binary threshold, with adaptable weights. Typically perceptrons can tackle binary questions. E.g. if I want to know whether to go for a walk I can ask whether it rains, which is an input for perceptron one, and whether my leg hurts is another input for perceptron two. Different weights then correspond to the kind of importance each question has. Leg pain is a more important factor than weather, so give that connection a greater weight. Above some threshold, the perceptron returns one, corresponding to 'yes, let's go for a walk'. Below the threshold it returns zero, 'no, let's not'. Multiple layers result in more complexity, but each perceptron outputs a binary value.

Now for the general case. The threshold, weights, biases, etc. of the network are what the optimizer will optimize, so they are model parameters. The key difference with perceptrons is that here there is not a fixed threshold where the value is zero below and one above, instead the output varies across the entire floating point range, using the full capability of the machine.

## 3.3 Training Artificial Neural Networks

Say we have some network, and we want it to read handwritten digits. If the handwritten digit is nine, and the network gives eight, it is wrong. As punishment, we perturb the weights and biases slightly in a way that will help it return a nine instead for that sample. If it gave a nine in the first place, we punish the model for some otherm mistake instead. This idea of perturbing the system is not possible with perceptrons or discontinuous activation functions, because a slight perturbation either results in no change (output of perceptron remains the same) or in a huge change when the output of that perceptron flips.

[6]Section 3.3 explains why they need to be continuous.

This can make the entire network change radically with just a tiny perturbation, and that itself will mean that all the previous learning will be undone (e.g. it could be that the network can accurately predict numbers one to eight but not nine before the perturbation, but only predict nine accurately afterwards). This is why activation functions and layer output (high order and highly dimensional polynomials) are continuous; their outputs change only slightly, meaning the whole network changes only slightly.

The natural way to design the input layer is to have one input associated with one input neuron. E.g. if we have 28 by 28 pixels as data (image), then the input layer would consist of $28^2$ neurons, with outputs (intensity of each pixel) between say zero and one.

### 3.4 Deep and Shallow Neural Networks

A shallow neural network (SNN) is a network with just one hidden layer (one layer that is not the input or output layer). A deep neural network (DNN) has multiple intermediary layers. A DNN can be feedforward or recurrent. These networks often perform far better than SNNs on difficult problems with complex relations. The reason is the ability of DNNs to build up a complex hierarchy of concepts, which is not possible with one hidden layer. The negative aspect is that each extra layer has a significant impact on runtime and makes optimization significantly more difficult and costly.

## 4 Neural Network Heuristics

In this section ANN hyperparameters are discussed, and how to make sense of the large amount of available information regarding each. Common for each hyperparameter is that there is a large amount of research for a variety of values of the parameters, but it is almost never the case that one value is better in all scenarios.

The most important design criteria are: the appropriate input (independent) variables, the learning method, the number of hidden layers, the number of nodes (to avoid both underfitting and overfitting).[7] Highly correlated variables should be removed from the input data. Analyze complexity of the problem to determine the number of hidden layers. More complexity typically requires more hidden layers.[7]

### 4.1 Hidden Layers and Nodes

For the input layer, the number of nodes is equal to number of features in the input data. The output layer contains a single node (for non-classifcation problems). One hidden layer is sufficient for the majority of problems. Or rather, it is hard to improve performance with more layers. There is an exception to this: hidden layers are required if the data must be separated non-linearly. The two figures below show examples of linear and non-linear data separation. $x$

and $y$ represent input variables, whereas the output in this case is a symbol (classification problem).

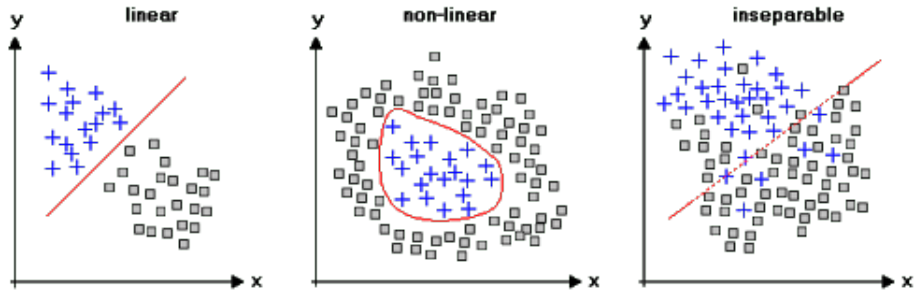


Figure 8: Linearly separated, non-linearly separated, and inseparable data.

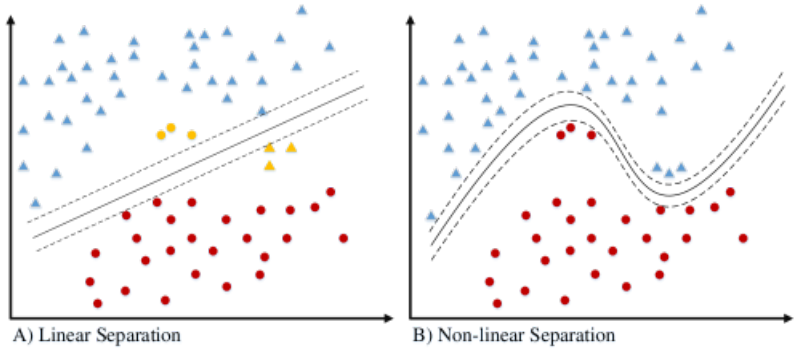


Figure 9: Linearly separated and non-linearly separated data.

For non-classification problems, the $z$-axis may contain the output variable (e.g. opacity, similar to our application).

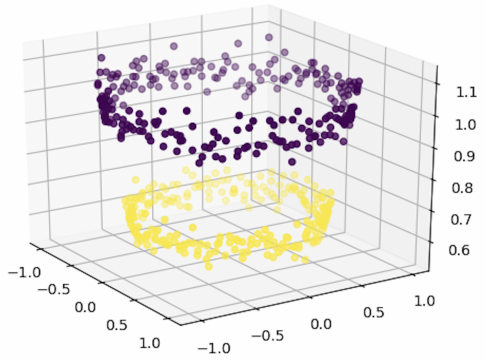


Figure 10: Data separation in a regression task.

Linear separation means a linear function can separate two classes of data. ANNs are most interesting for non-linear problems, as linear problems do not fully utilize the power of the ANNs non-linear activation functions.

The best approach is to start with a single hidden layer, study its performance, and verify that it captures the essential features of the theoretical model. Then repeat the process for two hidden layers, etc. Stop when the additional layer results in negligible accuracy improvement.

To ensure the network can generalize, the number of nodes should be kept as low as possible. With a large excess of nodes, the network can recall the training set accurately, but does not perform well on samples that were not part of the training set. This is overfitting, and has been previous discussed in Section 2.4.1.

## 4.2 Activation Functions

*This section is based on the detailed article by Casper Hansen (2019), see [9].*

An activation function (previously described in Section 3.2) determines the final output value of a neuron. Choice of activation function is a major hyperparameter, so they must be covered, understood, and studied. Some of the most powerful and common activation functions are: binaries, sigmoidals, hyperbolic tangents, rectifier linear units (ReLUs), softmaxes, exponential linear units (ELUs), Gaussian error linear units (GeLUs), and scaled exponential linear units (SeLUs). Each has its advantages, disadvantages, and special properties that must be understood.

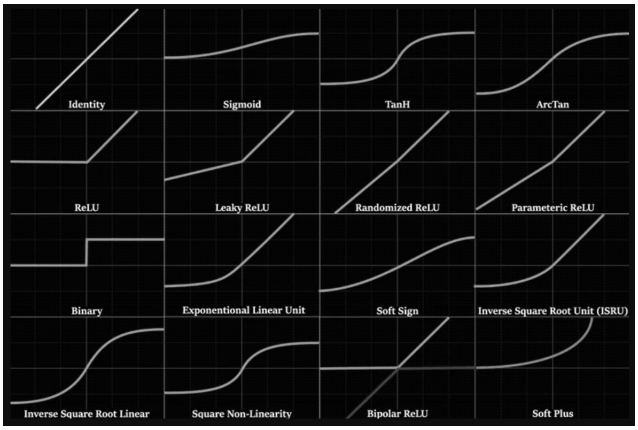


Figure 11: Examples of activation functions. The horizontal axis shows the input, the vertical axis the output, both unitless.

Let $x$ represent layer output.

### Binary

Usually used for simple binary classification tasks; zero corresponding to no signal, one to signal.

**Sigmoidal**

$$\frac{1}{1+e^{-x}}$$

One of the simplest activation functions. Usually used for classification. Used to be popular, but today critisized for slow training rates due to low gradients for extreme values. Output constrained to $[0, 1]$.

**Hyperbolic Tangent**

$$\tanh(x)$$

Here too the output is constrained to $[0, 1]$, and the mean is near 0. These are popular in hidden layers, since these properties help in centering data, making learning easier for the next layer.

**Rectified Linear Unit (ReLU)**

$$\begin{cases} 0 & x \leq 0 \\ x & x \geq 0 \end{cases} = \max(0, x)$$

Asymmetrical. Less computationally intensive than sigmoidal and hyperbolic tangent. Over-all great performance in a variety of tasks, but not very well understood why.

Due to being non-differentiable at zero, neurons tend to become inactive for all inputs, i.e they die out. This can be caused by high learning rates (a hyperparameter in the gradient descent method), which can reduce the model's learning capacity. Alternatives with non-zero gradient everywhere exist. Particularly noteworthy is leaky ReLU (designed to fix this shortcoming).

**Softmax**

Used in classification tasks to scale from the full floating point range to a $[0, 1]$ certainty interval for final output. Not interesting for non-classification tasks.

**Exponential Linear Unit (ELU)**

$$\begin{cases} e^x - 1 & x \leq 0 \\ x & x \geq 0 \end{cases}$$

Smooth. Asymmetrical. Produces negative outputs unlike relu, which helps networks update weights and biases in the right directions.

Longer computation time. Does not avoid the 'exploding gradient problem'. Commonly sees extremely small gradients (effectively zero, so the same problem ReLU faces).

**Scaled Exponential Linear Unit (SeLU)**

$$\lambda \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x \geq 0 \end{cases}$$

Smooth. Asymmetrical. Internal normalization is faster than external normalization, so the network converges faster. Vanishing and exploding gradient problem is impossible.

Relatively new, so lacks research and practical success. Smooth.

**Gaussian Error Linear Units (GeLU)**

$$\frac{x}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

Smooth. Asymmetrical. Naturally avoids vanishing gradient problems. Contains multiple trainable hyperparameters (mean and standard deviation of Gaussian CDF it incorporates) (as does SeLU). Fast to compute approximately.

New, so lacks research, but state of the art in many applications with practical success (e.g. natural language processing).

## 4.3 Optimizer

*This section is based on 'Overview of various Optimizers in Neural Networks' (2020) by Satyam Kumar, see [10].*

Optimization refers to the process of changing weights after each iteration based on the results of the loss function, as to minimize the loss function. Some common optimizers are gradient descent (GD), stochastic gradient descent (SGD), mini-batch SGD (MB-SGD), momentum SGD, adaptive gradient (AdaGrad), AdaDelta, and Adam.

**Gradient Descent (GD)**

Very basic. Simple to implement and well understood.

High memory requirement. Minimum of loss takes long to find, and sometimes isn't found at all; algorithm can be stuck at local minima or saddle points. If the real output function is very non-linear, there's more chance to be stuck at such a local minimum, which is a common occurence[7].

**Stochastic Gradient Descent (SGD)**

Requires less memory than GD.

[7]This can be observed by repeated experiment. Different random initial parameters will lead to different asymptotic behavior. See Figure 5 as to why this might happen; different random initial parameters corresponds to different initial location in loss function input space.

Time required to complete an epoch is larger compared to GD. Long convergence time. Often gets stuck at local minima in the same way described for GD.

#### Mini-Batch SGD (MB-SGD)

Lower time complexity than SGD. Less memory required than GD.

Can still get stuck at local minima. Noisy, because the algorithm takes a batch of input data points instead of all, and as a result takes longer to converge than GD.

#### Momentum SGD

Accounts for a sense of momentum; if progress is being made fast, it probably won't stop in the near future. Converges faster than GD and has all advantages of SGD.

Requires one extra variable for each input variable.

#### Adaptive Gradient (AdaGrad)

No need to update learning rate manually.

Slow convergence.

#### AdaDelta and Adam

Faster convergence than AdaGrad. No need to update learning manually.

Adam is considered the best algorithm over-all, and is the standard.[10] It converges much faster than algorithms with GD.

### 4.4 Loss Function

*Much of the information in the section is found in the articles 'Loss and Loss Functions for Training Deep Learning Neural Networks' and 'Loss functions: Why, what, where or when?', see [11, 12].*

The optimizer minimizes the loss function. The loss function compares the difference between a measured output value and an expected value based on the fitted model. Thus, the numerical value of the loss function must in some way depend on this difference. The essential difference between all loss functions is the way in which they punish outliers. The most common ones for regression are mean absolute error (MAE), mean squared error (MSE), mean squared logarithmic error (MSLE), logarithmic hyperbolic cosine error, Huber loss, and quantile error (used for interval prediction; not interesting here). See Figure 12.

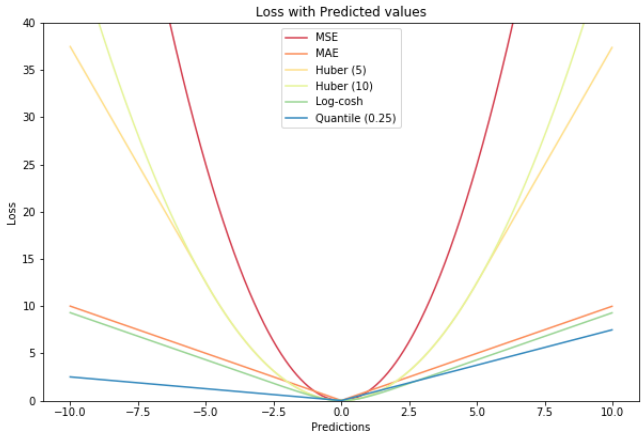


Figure 12: Various loss functions.

Let $x$ represent data and $\hat{x}$ model-predicted output.

**Mean Absolute Error (MAE)**

$$\sum_i |x_i - \hat{x_i}|$$

Neutral towards outliers. The most basic, least controversial choice.

**Mean Squared Error (MSE)**

$$\sum_i (x_i - \hat{x_i})^2$$

Compared to MAE, punishes large errors more harshly and is more lenient towards small errors.

**Mean Squared Logarithmic Error (MSLE)**

$$\sum_i (\log(x_i + 1) - \log(\hat{x_i} + 1))^2$$

Does not punish outliers so much, and does not mind small errors. So similar to MAE for outliers and similar to MSE for small errors.

**Logarithmic Hyperbolic Cosine**

A smooth version of MAE that like MSE, cares not about small errors.

#### Huber

The problem with MAE is a large gradient, which can lead to missing the minimum at the end of training (if gradiesnt descent is being used). Huber loss can be helpful in such cases. It is also more robust to outliers than MSE. Huber loss does however require extra hyperparameters ('delta's') that need to be iteratively trained.

### 4.4.1 Conclusion

Loss functions for fitting problems are relatively simple. Ultimately the loss function resulting in a model with the least systematic error for the application should be used.

## 4.5 Number of Epochs

An epoch is a training session in which all data is used to optimize the model. Too many epochs may cause overfitting.[8] The number of epochs is usually not that significant compared to other hyperparameters, and the right choice mostly depends on compute time and learning rate.[9] This is because it is practically impossible to choose too many epochs (any extra epoch causes runtime to increase, and overfitting only happens at much higher epoch values than is typical).[21]

## 4.6 Dropout & Noise Layers

*Much of the information in this section comes from the articles by Jason Brownlee and Robert Keim on the subject, see [13, 14, 15].*

Besides input, output, and standard hidden layers, a neural network may also contain bias, dropout and noise layers. Dropout deactivates the output of randomly chosen nodes of the previous layer. This selection of nodes changes after each iteration. The adds noise to the network during training, which avoids problems like overfitting. The dropout rate is the fraction of neurons in the dropout layer that is deactivated at any given moment. The dropout rate is another hyperparameter. Typical rates used are between 0.5 and 0.8. Problems with a large amount of training data may see less benefit from dropout.[13]

A (Gaussian) noise layer may be used on the input layer (most common), or any other layer. Noise is added with mean zero[10] while the standard deviation is a hyperparameter. The purpose is the same as for the dropout layer, i.e. preventing overfitting.

[8] If there is significantly more data than there are degrees of freedom in the model, overfitting is impossible.

[9] Feature standardization massively affects learning rate.

[10] A non-zero mean is systematic and would be learned. This is unacceptable since dropout layers are non-existent during inference (use of the model in production).

## 4.7 Cross Validation

*The example in this section is from the article 'A Gentle Introduction to k-fold cross-validation' by Jason Brownlee, see [16].*

$k$-fold cross validation is a technique that splits the data in $k$ sets. $k-1$ of those are used for training, the other for testing. For example, 100 data points and 5-fold validation results in test set of 20 data points and training of 80 data points may be used. This procedure is repeated $k$ times for $k$ untrained models, every model utilizing a different set as testing set. Each model has the same values for all hyperparameters. They all produce some result for loss, or according to the rules of some other evaluation metric. The model is the same, but the $k$ values for loss will differ due to different testing data (and due to noise, etc.). The end result is the average of all the individual results. Now, cross validation increases the running time by a factor $k$, so it is often not worth using. The standard value for $k$ is 10, a value that has been found practically to generally produce good results (low bias and modest variance). Again, an optimal value could be found by iterating over $k$ (further hyperparameter optimization). See Figure 13.

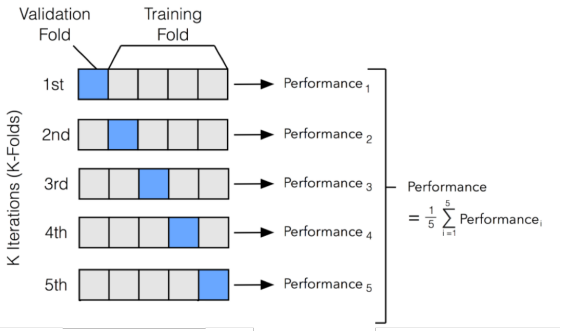


Figure 13: Cross validation for $k = 5$

## 4.8 Hyperparameter Optimization

There are practically an infinite number of combinations of hyperparameters resulting in different models. By doing research towards all parameters, we were able to apply a sieve which substantially cut down the number of combinations. Yet the number of possibilities is still very large. Proper hyperparameter optimization is impossible in all but the most simple cases and most optimized environments.

One way to make progress towards hyperparameter optimization is to have holes in the attempts, e.g. test only $k = 3, 6, 9, 12$ instead of all $k$ between 3 and 15.

Another approach is 'random search', where all hyperparameter values are randomly chosen, and this process is repeated $n$ times for a total of $n$ different

sets of hyperparameter values. It has been found that random search has a probability of roughly 95 percent of finding a combination of parameters within the 5 percent of optimal ones (i.e less than 5 percent of hyperparameter combinations perform better), with only 60 iterations ($n = 60$).[17]

Hybrid approaches also exist, where e.g. the activation function is iterated over fully but $k$-fold validation is randomized.

Another approach is to manually hyperparameter optimize a network by educated guesses, and then to study the influence of various hyperparameters on that network individually (uncorrelated). For example, vary the number of layers and the number of activation functions (not simultaneously), but instead of trying all activation functions for any layer, only try all activations functions on the decently optimized network. In this example the $\sim n^2$ optimization complexity becomes $\sim 2n$ complexity, at the cost of not testing for correlation. Correlation definitely exists (e.g. certain activation functions might like more or less layers because they need more or less help to represent the complex relationships), but these are less significant than the over-all effect of e.g. activation functions.

# 5 Results

In this section the choices made regarding training data and model design are covered and justified, along with the hyperparamter study made to come to many of these conclusions. Most of the justifications are separated in Section 5.3 on hyperparameter study. Also covered is the final over-all accuracy and performance of the model.

## 5.1 Training Data

The data used consists of a 50-50 mix between two sets of $2^{20}$ samples each. These sets have logarithmically distributed wavelength $\lambda$ and dust size $d$, which is standard due to the scale-dependent nature of scattering.

One set contains $\lambda$ in the $[0.01, 1.00]$ $\mu$m range and $d$ in the $[0.001, 1.000]$ $\mu$m range with $\sigma = 1.000$. The other set is identical, except it considers the broader $[0.01, 10.00]$ $\mu$m range for $\lambda$ (in less detail). Both use a $\tan(\pi x/2)$ mixture model, as described in Section 2.3, limited with the additional $[0, 1] \rightarrow [0.01, 0.99]$ transformation ($a = 0.01$, $b = 0.99$).

The model is capable of at least the entire $[0.01, 10.00]$ $\mu$m $\sigma$ range, with more attention paid near $[0.01, 1.00]$ (where all the complex behavior occurs). Scaling this up all the way to 1000 $\mu$m is trivial and does not affect performance, but is not done as to focus more on the difficult region.

Some absorption data is shown in Figures 14 and 15 for enstatite at a low $\sigma$, showcasing the typical waterfall structure of Mie scattering.

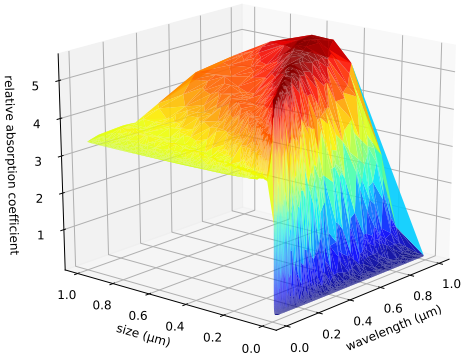


Figure 14: Small quantity of raw, unstructured absorption data for enstatite at low $\sigma$.

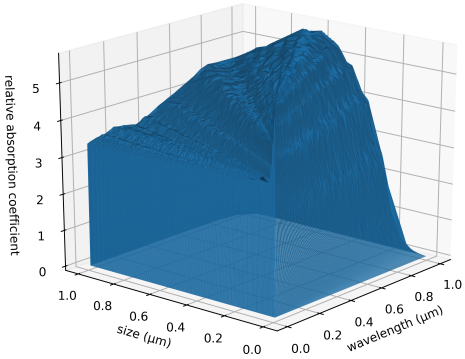


Figure 15: Large quantity of interpolated data for enstatite at low $\sigma$.

This narrow look at one molecule and one $\sigma$ is very simple. Really all nineteen dimensions are correlated with all eighteen others, and only two are visualized here across a tiny range.

## 5.2 Model Design

### 5.2.1 Data Preprocessing

To signify that Mie scattering is a scale-dependent problem (i.e. relative differences in wavelength $\lambda$ and dust size $d$ lead to different scattering behavior, not absolute differences), $\lambda$ and $d$ are logarithmically scaled (the model is trained on $\ln \lambda$ and $\ln d$ rather than $\lambda$ and $d$). This also helps scaling the problem later to greater $\lambda$ and $d$ ranges.

The mixture ratio input variables are by nature already minmax scaled. Since they are linearly dependent (the sum of all ratios must be one) and really are fifteen dimensional, one is simply removed. The other three variables ($\lambda$, $d$, and dust size standard deviation $\sigma$) are minmax scaled, where the min and max come from the random number generator rather than the data (the latter is more conventional). This way all input variables are on equal footing in the $[0, 1]$ range.

The two output variables (absorption and scattering coefficients) are not normalized since they are already unitless and the two are similar in size.

### 5.2.2 Model Geometry

The model is a basic feedforward ANN [11], but not a sequential network, i.e. there are branches. In particular, the model is set up so that the sixteen mixture ratios may be treated differently initally than the three physical variables ($\lambda$, $d$, $\sigma$), after which they are combined, and ultimately lead to the final output.

The model is relatively deep, with five deep dense layers with non-linear activation functions, followed by one hidden linear layer, and finally a linear output layer. Four layers gives identical asymptotic behavior, so the model can be made simpler still. Five layers however trains faster, so it is more scalable to more molecules and greater $\lambda$ and $d$ ranges.

Since the amount of training data is so large, overfitting is impossible, so no noise is added during training. For the same reason cross-validation ultimately isn't used. Both were initially experimented with when less high quality data was available (before good mixture models) and when more complex models were studied.

The model features layer normalization (discussed in Section 2.5.2) after the initial non-linear dense layer (the highest neuron count layer). This improves the learning rate significantly, see Figure 16.

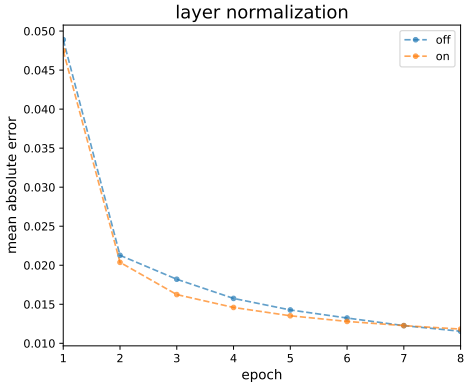


Figure 16: Effect of layer normalization on learning rate.

A high level overview of the model is given in Table 1. Each row represents one layer, starting from the top.

Ultimately this model has only 607 parameters, which is very few, by design (see Section 2.2).

### 5.2.3 Optimizer

For optimization, the Adam and Nadam algorithms were predominently used. These algorithms resulted in the both the best training rates and best asymp-

[11] A feedforward network is a network where there are no recurrent connections. Layer 1 connects to layer 2 and only layer 2. Layer 2 to layer 3 and so on. In a recurrent network, layer 2 could connect back to layer 1, adding a lot of complexity. Recurrent networks are poorly understood.

| |
|---|
| dense [minmax($\ln \lambda$), minmax($\ln d$), minmax($\sigma$)], no activation, *input 1*<br>dense [16 - 1 mixture ratios], *input 2* |
| concatenation |
| dense [12 dense neurons], GeLU activation, *deep* |
| layer normalization, *standardization* |
| dense [9 neurons], GeLU activation, *deep* |
| dense [8 neurons], GeLU activation, *deep* |
| dense [7 neurons], GeLU activation, *deep* |
| dense [6 neurons], GeLU activation, *deep* |
| dense [5 dense neurons], linear activation, *pre output* |
| dense [relative scattering, relative absorption], linear activation, *output* |

Table 1: High level overview of the final model.

totic behavior.

#### 5.2.4 Loss Function

We predominantly use MAE loss since we have no reason to punish errors non-linearly for the application of computing radiative feedback, and MAE gives acceptable residuals with no less structure than the alternatives. All tested loss functions give similar asymptotic behavior when it comes to structure in the residuals.

### 5.3 Hyperparameter Study

A basic, manual, uncorrelated hyperparameter study was performed; the last kind described in Chapter 4.8. Many of the choices described so far (number of layers, activation functions, optimizers, etc.) were decided upon in this way. Only the most important hyperparameters are studied in detail and covered here, as there are far too many. The hyperparameters resulting in the fastest learning rate and lowest MAE were used.

These studies were carried out with $2^{20}$ samples over eigth epochs[12], and are all based on the highly optimized product model from Table 1, which is why all errors are relatively low.[13]

[12] Eigth is the result of balancing computation time and learning progression. More epochs wouldn't result in much improvement for the model, as evident from the almost horizontal slope in the figure

[13] Relative absorption and scattering values are typically no greater than seven in these ranges for these molecules.

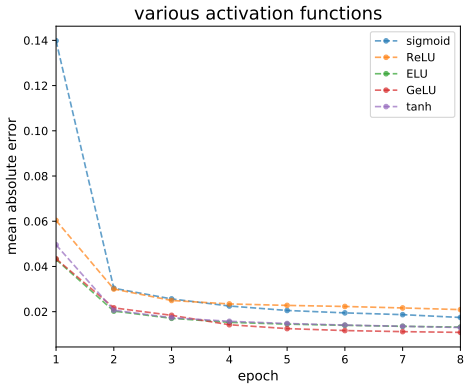


Figure 17: Activation function study.

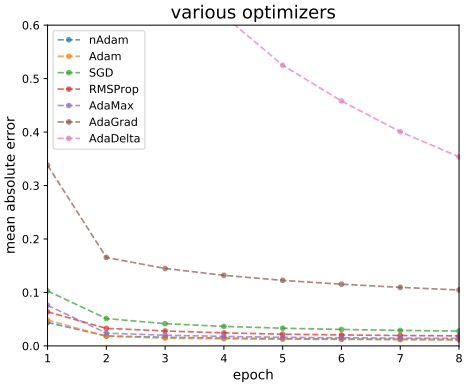


Figure 18: Optimizer study.

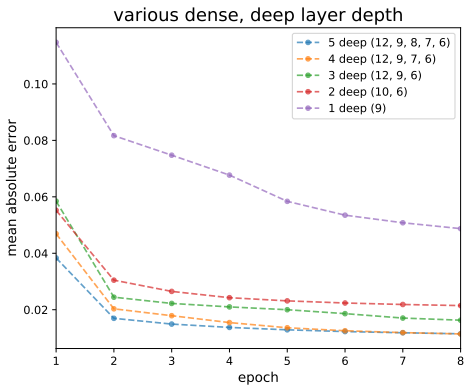


Figure 19: Dense, deep layer depth study.

Notice for example that four layers indeed gives identical asymptotic behavior to five, as previously described.

Studying loss functions requires a different approach since comparing hyperparameters thusfar has happened by comparing losses. Instead, the residuals are studied, i.e. the difference between data output and model output for some given data input. Positive values indicate the model over estimates, negative indicate it underestimates. Less structure, smaller over-all and peak residuals, and zero-based residuals are all preferred, as they point towards less systematic error.

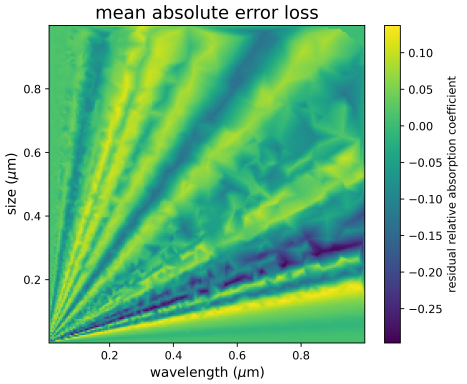


Figure 20: Enstatite residuals of models optimizing MAE.

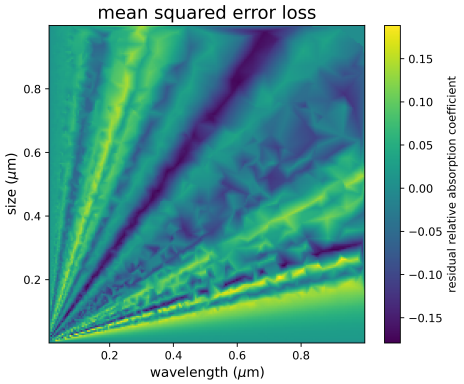


Figure 21: Enstatite residuals of models optimizing MSE.

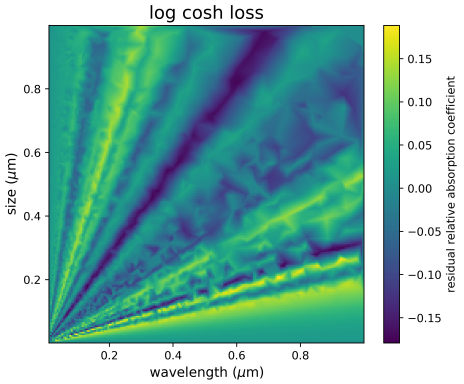


Figure 22: Enstatite residuals of models optimizing logarithmic hyperbolic tangent loss.

MSE and logarithmic hyperbolic tangent look indistinguishable because these functions are approximately identical for small errors. MAE only looks different in color because it treats peaks diffently, but otherwise shows the same structure.

## 5.4 Accuracy

The final model has a conservative over-all systematic error across the testing datasets of $\sim 1\%$ and was trained over 60 epochs of $2^{21}$ samples. The exact accuracy is difficult to calculate since the output variables aren't normalized, so this is assuming an average absorption and scattering coefficient across the datasets near unity. Peak errors can hit significantly more than 1%, but 99.9% of samples have an error under 5%. True peak error is of course unknown due to the extreme sparsity of a dataset of a few million points in nineteen dimensional input. Programs are available to compare an existing model to existing unstructured data in an unbiased way (by residual plot), which can be used to study peak error of any mixture ratio at any $\lambda$, $d$, and $\sigma$.

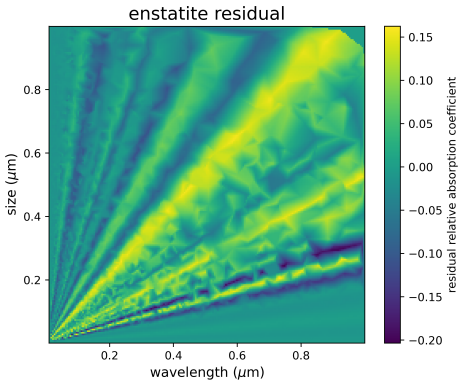


Figure 23: Final model enstatite residual.

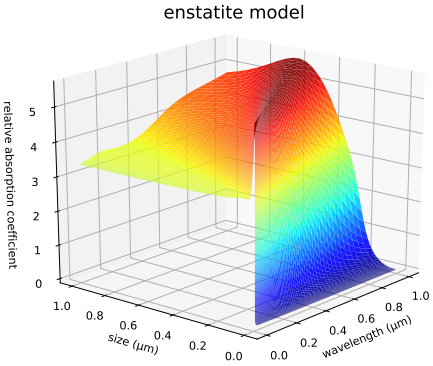


Figure 24: Final model showcasing enstatite waterfall. Compare to Figures 14 and 15 showing the data.

# 6 Discussion

Over-all progress was made with relative ease. The three major breakthroughs (chronologically) were to take time to hyperparameter optimize manually, to use proper preprocessing with logarithmic scaling, and to create data with a realistic mixture model.

A subtle but important find is that adding more deep layers to the network does not result in better asymptotic behavior. This implies that at the high accuracy we are dealing with, training is bottlenecked by the geometry of the model itself. Judging by Figure 24, it is apparent the model is missing some of the finer resonance structure (as seen in Figure 15). Developing specialized layers of a shape that are able to more naturally approximate these patterns (trigonometric, etc.) is a natural next step towards being able to calculate transmission spectra.

A major point also is the mixture models. More research should be done here as to what exactly is realistic. Preventing the simulation from failing for highly dominated mixture ratios (allowing $a$ to decrease and $b$ to increase slightly) would have a large positive effect on the mixture ratios in the data (widening the distribution of the maximum of any molecule's mixture ratio).

The model has very few parameters, and is thus much faster than the classical Mie scattering codes. 607 parameters in practice, but if the five deep layers are scaled down to four at the cost of some training rate, under 500 is doable. A two layer deep model with less than 300 parameters still performs borderline sufficiently for computing radiative feedback.

Scaling to greater $\lambda$ and $d$ range is important practically and can be done with ease, although it was not ultimately used in the final model as to simplify residuals. This is easy because little additional complexity is introduced, and because of the design choice of having $\lambda$ and $d$ logarithmically scaled in

preprocessing.

Order of magnitude improvements were achieved with relative ease by logarithmic scaling and using a proper mixture model. One more order of magnitude improvement in accuracy would make these models comfortably capable of calculating transmission spectra.

# 7 Conclusion

After careful preprocessing and mixture ratio modeling, a small, highly optimized model was trained on realistic data with performance sufficient for computing radiative feedback. This model can easily be scaled to more molecules and greater wavelength and dust size ranges. With some custom layers, more advanced hyperparameter optimization, and better mixture ratio models, we believe this model is capable of calculating transmission spectra too, and for more realistic molecular distributions than studied here.

# References


[1] Wolf & Voshchinnikov, 'MieX', 2004

[2] N.V. Voshchinnikov, 'Optics of Cosmic Dust', Astrophysics and Space Physics Review 12, 1, (2002)

[3] Toon O., T. Ackerman, 'Algorithms for the calculation of scattering by stratified spheres', Applied Optics 20, 3657, 1981

[4] Dave, J.V., 'Subroutines for Computing the Parameters of the Electromagnetic Radiation Scattered by a Sphere', IBM Scientific Center, Palo Alto, California, Report No. 320 - 3236, 1968

[5] Dave, J.V., 'Scattering of Visible Light by Large Water Spheres', Applied Optics 8, 155, 1969

[6] Michael A. Kuhn, Eric Feigelson, 'Mixture Models in Astronomy', 2017

[7] Steven Walczak, 'Heuristic principles for the design of artificial neural networks', Information and Software Technology, volume 41, 1999
doi.org/10.1016/S0950-5849(98)00116-5

[8] Jeff Heaton, 'Introduction to neural networks for Jav, second edition', 2008

[9] Casper Hansen, 'Activation Functions Explained - GELU, SELU, ELU, ReLU and more', 2019
mlfromscratch.com/activation-functions-explained/

[10] Satyam Kumar, 'Overview of various Optimizers in Neural Networks', 2020
towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5

[11] Jason Brownlee, 'Loss and Loss Functions for Training Deep Learning Neural Networks', 2019
machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/

[12] Phuc Truong, 'Loss functions: Why, what, where or when?', 2019
phuctrt.medium.com/loss-functions-why-what-where-or-when-189815343d3f

[13] Jason Brownlee, 'A Gentle Introduction to Dropout for Regularizing Deep Neural Networks', 2018
machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

[14] Jason Brownlee, 'How to Improve Deep Learning Model Robustness by Adding Noise', 2018
machinelearningmastery.com/how-to-improve-deep-learning-model-robustness-by-adding-noise/

[15] Robert Keim, 'Incorporating Bias Nodes Into Your Neural Network', 2020
www.allaboutcircuits.com/technical-articles/incorporating-bias-nodes-into-your-neural-network/

[16] Jason Brownlee, 'A Gentle Introduction to k-fold Cross-Validation', 2020
machinelearningmastery.com/k-fold-cross-validation/

[17] Bergstra and Bengio, 'Random Search for Hyper-Parameter Optimization', Journal of Machine Learning Research 13 281-305, 2012

[18] Hinkelmann, Knut., 'Neural Networks, p.7', University of Applied Sciences Northwestern Switzerland

[19] Yanxia Zhang  Yongheng Zhao, 'Astronomy in the big data era.', Data science journal, 2015

[20] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton, 'Layer Normalization.', 2016
arxiv.org/abs/1607.06450

[21] Saahil Afaq, Dr. Smitha Rao, 'Significance Of Epochs On Training A Neural Network', International Journal of Scientific & Technology Research, Volume 9, Issue 6, 2020