Improved Representation of Protein Structures Using Neural Implicit Representations with Periodic Activation Functions

by

Jay Jaewon Yoo

Supervisor: David Fleet
April 2021

B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

Improved Representation of Protein Structures Using Neural Implicit Representations with Periodic Activation Functions

by

Jay Jaewon Yoo

Supervisor: David Fleet

April 2021

Abstract

Cryogenic electron microscopy is a revolutionary method of reconstructing 3D protein structures. This thesis investigates the use of neural implicit representations with periodic activation functions for improved protein representations in cryogenic electron microscopy. Neural implicit representations theoretically enable representations that are significantly more memory efficient and enable atomic resolution representation compared to occupancy arrays, which are the common means of representing protein structures. Atomic resolution representations allow for protein representations to be upscaled to finer detail with minimal loss in information, creating representations that have ”atomic resolution.” The thesis demonstrates the potential for neural implicit representations with periodic activation functions to act as memory efficient protein representations with atomic resolution but also presents challenges that must be overcome before such representations can be effectively used in cryogenic electron microscopy. The code used during this thesis is available at `https://github.com/JayJaewonYoo/SIREN-Protein-Representations.git`.

Acknowledgements

I would like to thank my supervisor, Professor David Fleet, for taking the time to supervise my thesis despite having existing obligations at Google. Professor Fleet provided invaluable guidance throughout the thesis and acted as an excellent mentor who furthered my passion for research and pushed me to truly excel with my work. I am whole-heartedly grateful to have worked on this thesis with him. I would also like to thank Ali Punjani, CEO of Structura Biotechnology, for introducing me to Professor Fleet and the world of cryogenic electron microscopy. Without him, I would not have been able to explore this thesis nor would I have had the knowledge and skills to effectively pursue this area of research. For the mentorship he provided and the opportunities he created, I am truly grateful.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

1.1.1 Cryo-electron Microscopy

Cryo-electron microscopy (cryo-EM) is a revolutionary method of reconstructing the 3-dimensional (3D) shape of particles and proteins using electron microscope images of frozen protein solutions [1]. The resultant 3D reconstructions can then serve to understand protein function and inform drug development.

2-dimensional (2D) representations of slices from 3D structures are also useful for particle picking, the process of extracting particles from noisy electron microscope images, as the 2D representations provide a prior on the visual appearance of the particles of interest.

3D representations can be extended 4 dimensions by including the time dimension, enabling simulations of protein structure behavior over time. One such example of 4-dimensional (4D) representations is 3D Variability Analysis, which enables the resolution and visualization of ”detailed molecular motions of both large and small proteins by modeling ”the conformational landscape of a molecule as a linear subspace.” [2].

1.1.2 Resolution

The quality of protein structures is determined by its resolution measured in Angstroms (Å), where lower resolution values indicate finer detail. The resolution can be determined using the Fourier Shell Correlation (FSC) [3]. The FSC is defined as the normalized cross correlation between two

3D structures, expressed as a function of spatial frequency as defined in equation (1.1) [4].

$$FSC(k) = \frac{\sum_{k,\Delta k} F_{3D,1}(k) \cdot F_{3D,2}(k)^*}{\sqrt{\sum_{k,\Delta k} |F_{3D,1}(k)|^2 \sum_{k,\Delta k} |F_{3D,2}(k)|^2}} \tag{1.1}$$

$F_{3D,i}(k), i = [1, 2]$ are the Fourier transforms of the 3D structures, $*$ indicates the complex conjugate of a Fourier transform, k is a variable spatial frequency, and $\sum_{k,\Delta k}$ indicates a summation of $F(k)$ over a small range Δk around k [4]. Like traditional correlation, FSC outputs a value ranging from -1 to $+1$, where $+1$ indicates perfect correlation, 0 indicates no correlation, and -1 indicates perfect inverse correlation [5].

FSC is used in cryo-EM because when creating protein structures, there is often no ground truth to compare to. To circumvent this problem, the FSC between two halves of the reconstruction is calculated to measure the amount of noise in the reconstruction. As a normalized cross correlation, the FSC of the two halves can be interpreted as a measure of the signal to noise ratio in the reconstruction, providing a quantitative measure of quality.

FSC is commonly presented using FSC curves, graphs with FSC values in y-axis and spatial frequencies in the x-axis. Expected behavior in a FSC curve is for FSC values to be near $+1$ at lower frequencies and decrease toward 0 as the frequency increases. The resolution of a 3D representation can be determined from a FSC curve as the reciprocal of the frequency at which the FSC value falls below a predetermined threshold [4]. This predetermined threshold is often when the signal to noise ratio is estimated to be 1:1. In the context of this thesis, all FSC calculations will be calculated with one of the structures acting as a noiseless ground truth. Thus, a FSC of 0.5 corresponds to a signal to noise ratio of 1:1 and the resolution at which the FSC falls below 0.5 will be used as the resolution.

The resolution of a reconstruction is limited by the Nyquist rate. The Nyquist rate is one half of the sampling rate of a signal and represents the maximum frequency a signal can be sampled at without losing any information. Resolutions corresponding to frequencies higher than the Nyquist rate are not achievable because reconstructing proteins at these frequencies results with artifacts in the reconstruction [6]. Thus the resolution corresponding to the Nyquist rate is the highest theoretical resolution for a reconstruction at a given sampling rate. Overcoming this limit requires changing the Nyquist rate, which requires altering the sampling rate of a reconstruction.

In the context of this thesis, the sampling rate of the input is 1 due to the input signals being sampled at 1 sample per pixel and the corresponding Nyquist rate is 0.5. Thus, the frequencies in the FSC curves used in this thesis are presented up to 0.5. However, most of the FSC plots shown in this document present the FSC as a function of the resolution, directly converting the spatial frequency to resolution. An example FSC curve produced for this thesis can be seen in Figure 1.1.

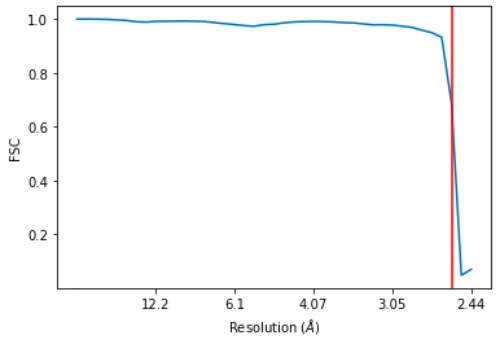


Figure 1.1: Example of an excellent FSC Curve.

The Fourier Ring Correlation (FRC) is the normalized cross correlation between two 2D signals. The definition of the FRC is effectively identical to that of FSC except the equation uses Fourier transforms of 2D images rather than 3D structures. As such, FRC is defined in equation (1.2) [7].

$$FRC(k) = \frac{\sum_{k,\Delta k} F_{2D,1}(k) \cdot F_{2D,2}(k)^*}{\sqrt{\sum_{k,\Delta k} |F_{2D,1}(k)|^2 \sum_{k,\Delta k} |F_{2D,2}(k)|^2}} \tag{1.2}$$

Similar to FSC, $F_{2D,i}(k), i = [1, 2]$ are the Fourier transforms of the 2D signals, and other mathematical definitions are identical to those defined with equation (1.1) [7]. FRC curves can be used to identify the resolution of a 2D representation the same way FSC curves can.

FSC and FRC cannot effectively capture the correlation at frequencies with negligible signal amplitudes due to numerical issues that arise in the computation of FSC and FRC when the signal amplitude is negligible. Thus, whether a representation has a low FSC or FRC at certain frequencies due to low correlation or low signal amplitude can be ambiguous. To determine if low FSC or FRC values in the corresponding curves are caused by low correlation, a graph measuring the signal amplitude over the spatial frequency must be used in conjuction with the FSC and FRC curves.

The code implementations of both FSC and FRC plots can be found in Appendix B.

1.2 Gap

While current methods produce high resolution reconstructions, the reconstructions are limited by how they are represented; occupancy arrays. Firstly, occupancy arrays are uniform over the shape. This uniformity results with a poor distribution of parameters. For example, occupancy arrays will use 1000 parameters to represent 1000 voxels of the same value when 1 parameter can be used to represent the 1000 voxels. Occupancy arrays also have significant memory costs due to the high number of elements in the array. Furthermore, the occupancy arrays are discrete representations and interpolating between voxels to increase the resolution requires interpolation techniques that

are potentially erroneous. Most notably, occupancy arrays are limited by their predefined dimensions which effectively represent the distance between the voxels of the reconstruction. As the dimensions of the occupancy arrays determine the quality of the representation, they directly limit the potential resolution of any particle representation. Occupancy arrays ultimately pose a bottleneck to the potential resolution of particle structures in cryo-EM, limiting the information that can be inferred from these structures and their ability to inform protein study and drug development.

1.3 Thesis Purpose

To address the issue of occupancy arrays, the thesis will investigate the hypothesis that neural networks with periodic activation functions can learn implicit neural representations of proteins and thus be used to produce high fidelity protein structures with any desired resolution. Neural networks can learn implicit neural representations by parameterizing signals as continuous functions by mapping coordinates to values. Neural networks with periodic activation functions are referred to as sinusoidal representation networks or SIREN by the original paper by Sitzmann et al. which introduced this architecture. This report will henceforth refer to neural representations with periodic activation functions as SIREN [9].

SIREN is explored because previous research demonstrated using SIREN is ”ideally suited for representing complex natural signals” [9]. Protein structures derived from cryo-EM methods are ultimately complex natural signals and thus should also be effectively represented by SIREN. Furthermore, the SIREN architecture is one that can be trained with inputs of one resolution and then be used to output any desired resolution. This is due to the inputs of the architecture being coordinate points. Once the network learns the latent features of the structure of interest, the coordinate points passed into the model can be sampled at any sampling rate.

The ability to sample protein representations at any sampling rate is crucial because, as detailed in Section 1.1.2, the resolution of a protein representation is limited by the Nyquist rate and to achieve any desired resolution, the Nyquist rate must be altered by changing the sampling rate. Thus SIREN’s capacity to output protein representations at any sampling rate enable SIREN to produce density maps at any desired resolution once the network learns the structure. The thesis first attempts to train SIREN to output 2D slices of protein structures. Then the output dimensionality will be increased, training the network to output 3D structures.

Overcoming the limitation of occupancy arrays allows for higher fidelity structures, enabling advances in the understanding of protein structures and drug development. Specifically, increased potential resolution of 2D and 3D protein representations can enable more complex study of particle picking and protein structure.

Chapter 2

Literature Review

2.1 Literature 1: Implicit Neural Representations with Periodic Activation Functions

2.1.1 Neural Implicit Functions

The paper Implicit Neural Representations with Periodic Activation Functions by V. Sitzmann et al. presents a class of functions Φ that aims to satisfy equations of the form:

$$F(x, \Phi, \nabla_x \Phi, \nabla_x^2 \Phi, ...) = 0, \Phi : x -> \Phi(x) \tag{2.1}$$

where $x \in \mathbb{R}^m$ are spatial or spatio-temporal coordinates. $\Phi(x)$ is a function that maps these coordinates to any desired quantity, while satisfying Equation 2.1. Φ is implicitly defined by the relation between x and $\Phi(x)$ defined by F. Thus Φ is an implicitly defined function. The paper names neural networks that parameterize such implicitly defined functions as implicit neural representations [9].

The pertinent aspect of the implicit neural representations is that they have the capacity to model representations using continuous, differentiable representations [9]. This is crucial in addressing the gap of producing density maps of any resolution as non-continuous functions that map coordinates to discrete density values directly will be unable to effectively output densities for coordinates that were not learnt during training. Producing density maps of any resolution requires neural networks that can effectively represent coordinates that are sampled at a different rate than the coordinate sampling used for training. Implicit neural representations enable this functionality by being capable of modeling fine detail without being limited by the grid resolution.

The continuous aspect of the implicit neural representations also offers the advantage of being much more memory efficient compared to a standard discrete representation. This memory

efficiency is due to the implicit neural representation's ability to represent high detail structures without being limited by the grid resolution. As discrete grid resolution requires significantly more memory than the memory required to store a neural network with a sufficient number of parameters to learn the implicit function, having the memory be bottle-necked by the network size rather than the grid resolution is an improvement in memory efficiency.

To summarize the advantages of implicit neural representations of density maps compared to traditional occupancy arrays:

- Can represent high detail structures without the limitation of discrete grid resolution.
- Requires less memory than networks representing discrete grids.
- Greater expressive power compared to networks representing discrete grids due to the reduced memory enabling a greater number of parameters.

According to the paper, implicit neural representations have been used to research continuous representations of occupancy arrays [9]. Such papers include "Occupancy networks: Learning 3d reconstruction in function space" by L. Mescheder et al., and "Learning implicit fields for generative shape modeling" by Z. Chen and H. Zhang.

2.1.2 Periodic Activation Functions

Significant research effort has been put into the implicit neural representations described in Section 2.1.1. However, a flaw of most of this research, as indicated by the paper, is that many of the suggested architectures fail to have the expressive power required to represent fine details in signals. This is due to these architectures using Rectified Linear Unit (ReLU) based multilayer perceptrons (MLPs).

MLPs are the most common base architecture for neural networks. MLPs consist of an input layer, an output layer, and a variable number hidden layers in between these two layers. Nodes are directed from earlier layers to later layers and nodes within the same layer do not directly connect with one another [10]. All nodes in the hidden layers and the output layers represent a linear transform of the nodes that are directed into them. This linear transform can be expressed as the following equation:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.2}$$

where $\mathbf{x}$ is the inputs, $\mathbf{W}$ is the weight matrix, $\mathbf{b}$ is the bias vectorm, and $\mathbf{z}$ is the output of the linear transform. However, a network solely consisting of linear transforms are limited to linear decision boundaries. To overcome this issue, the outputs of the linear transforms are passed into nonlinear functions referred to as activation functions. This serves to introduce nonlinearity to neural

networks, enabling nonlinear decision boundaries. Passing the linear transform into an activation function ϕ results with the following equation:

$$\mathbf{y} = \phi\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \tag{2.3}$$

where $\mathbf{y}$ is the output of a node in a MLP and the other variables are as defined for equation 2.2.

ReLU, defined as $f(z) = \max(0, z)$ for an input logit z [11], is one of the most common activation functions. The flaw with using ReLU based MLPs is not the MLPs, rather the flaw lies in the use of ReLU activation functions. ReLU functions are piecewise linear, which results with second derivatives of zero at all points [9]. Thus, ReLU has difficulty learning information contained in higher-order derivatives. High detail natural signals such as protein structures contain a significant amount of information in the second derivatives. The paper briefly explores other activation functions that are capable of capturing information in the second derivative such as hyperbolic tangent and softplus. However, the paper found that these activation functions also fail to represent a sufficient amount of detail and are not well-behaved [9].

To address this flaw in activation functions, the paper introduces periodic activation functions. These periodic activation functions are found to robustly fit complicated signals such as natural images and 3D shapes. The paper specifically applies the periodic activation function using the sine function [9]. Thus the transform performed by a node on inputs $\mathbf{x}$ in a SIREN is defined as follows:

$$\mathbf{y} = \sin(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.4}$$

This transform first uses weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ to apply an affine transform that geometrically manipulates the input features $\mathbf{x}$ before applying the periodic activation functions which convert the features to continuous sinusoidal form.

A notable benefit of the sinusoidal activation functions is the training speed. Unlike other architectures that represent continuous signals in high detail, SIREN is significantly faster as the computational cost of a sine function is minimal compared to other means of representing features in continuous sinusoidal space. To further enhance training speed, hyperparameters ω_0 and $\omega_{0,h}$ are introduced. ω_0 applies to the input layer, and $\omega_{0,h}$ applies to the hidden layers and output layer of the architecture. These two hyperparameters are factors that increase the spatial frequency of layers to better match the frequency spectrum of the desired implicit function. This increases gradients to the weight matrix by the factors ω_0 and $\omega_{0,h}$ while keeping gradients with respect to inputs to the sine neurons unchanged, ultimately improving the speed of convergence of the weight matrix during gradient descent [9].

2.2 Literature 2: Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains

2.2.1 Fourier Features

The paper Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains by M. Tancik et al. introduces the concept of mapping inputs to MLPs to Fourier features and then using the Fourier features as inputs into the MLP. These networks are referred to as Fourier feature networks. This approach overcomes the inability of standard MLPs to learn information in high frequencies [12]. By enabling MLPs to learn high frequencies, the ability of the model to express complex 3D structures is greatly enhanced. The Fourier mapping is shown in equation 2.5.

$$\gamma(\mathbf{v}) = \left[a_1 \cos\left(2\pi \mathbf{b}_1^T \mathbf{v}\right), a_1 \sin\left(2\pi \mathbf{b}_1^T \mathbf{v}\right), ..., a_m \cos\left(2\pi \mathbf{b}_m^T \mathbf{v}\right), a_m \sin\left(2\pi \mathbf{b}_m^T \mathbf{v}\right)\right]^T \tag{2.5}$$

This equation maps input coordinates $\mathbf{v}$ to Fourier space by passing the inputs into the sinusoidal functions sin and cos. The vectors $\mathbf{b}_i$ are frequency vectors which determine the range of frequencies that can be learned, and the scalars a_i are the amplitudes of each Fourier feature.

An important aspect of Fourier features is that by mapping the coordinates to Fourier features, the Fourier features can be sampled at different rates. As discussed in Section 1.1.2 and Section 1.3, the sampling rate determines the theoretical maximum resolution of the output representation. Thus sampling Fourier features at different sampling rates enable density maps of any desired resolution, accomplishing the thesis purpose.

The paper demonstrates that by properly setting the parameters a_i and $\mathbf{b}_i$, the rate of convergence and the ability for the network to generalize improves dramatically. The optimal settings of these parameters is dependent on the particular inputs being mapped to Fourier space, and the desired output representations [12]. As such, optimal performance with Fourier feature networks requires tuning the hyperparameters based on the protein and desired representation. This may pose a problem for representing 3D density maps as the optimal settings may be difficult to tune and vary drastically depending on the protein being represented. However, this also creates the opportunity to provide a prior to the model that enables excellent performance. The impact of these parameters in the context of cryo-EM and protein structure representation is difficult to ascertain without proper experimentation in the context of cryo-EM, which the paper does not provide.

The paper explores various regression tasks that can be performed using Fourier feature networks. One task that is mentioned is 3D shape regression to represent occupancy networks [12]. This demonstrates that Fourier feature networks have the potential for powerful representation of density maps.

2.3 Literature Synthesis

The literature reviewed each introduced a variation of MLPs: SIREN and Fourier feature networks. As both papers were preprinted in 2020, SIREN and Fourier feature networks are the latest innovations in neural networks that learn latent features in Fourier space, thereby enabling representations of protein structures using sampling rates that differ from the sampling rate of the structures used for training. Both architectures by accomplish this by learning the latent features in Fourier space but differ in that SIREN converts the latent features to Fourier space using sinusoidal activation functions while Fourier feature networks map the inputs directly to Fourier space. These introduce different benefits and drawbacks that are summarized in Table 2.1.

	SIREN	Fourier Feature Networks
Benefits		
	• Greater expressiveness and robustness in the representation of fine details of the implicit function such as information stored in derivatives of the signal.	• Able to effectively learn information at high frequencies, enhancing the model's ability to output high fidelity 3D structures.
	• Improved training speed due to low computation costs and the ability to utilize hyperparameters to better match spatial frequencies to the desired frequency spectrum.	• Can use hyperparameters a_i and $\mathbf{b}_i$ to set provide a prior on the protein representation, greatly improving the rate of training convergence.
	• Reduced memory costs allowing for increased number of parameters.	
Drawbacks		
	• Sinusoidal activation functions are possibly not as powerful as Fourier feature mapping in expressing higher frequencies.	•a_i and $\mathbf{b}_i$ may be difficult to tune to specific protein structures.
		• No indication of memory cost.

Table 2.1: Benefits and drawbacks of Fourier feature networks and SIREN for atomic resolution protein representations.

Comparing the two architectures, it seems that SIRENs have reduced memory costs and faster training speeds costs. However, Fourier feature networks demonstrate potential for higher fidelity representations than SIREN when its hyperparameters are tuned.

Both architectures meet the requirement of outputting any desired resolution by allowing outputs of any sampling rate. Moving forward, the first step in this thesis is to use experimentation with 2D protein slices to determine which architecture is best for the representing protein structures. Once an architecture is chosen, experimentation can be performed to verify results. Once sufficient results are acquired, the architecture can be adapted for 3D and 4D representations.

Chapter 3

Experiments: Methods, Results, and Discussions

3.1 Overview

The experiments performed for this thesis accomplished the following objectives:

1. Compare neural network architectures for neural implicit representations (Section 3.2).
2. Investigate the 2D case by using neural implicit representations to represent 2D slices of protein structures (Section 3.3).
3. Investigate the 3D case by using neural implicit representations to represent 3D protein structures (Section 3.4).

This chapter explores the experiments involved in each objective, in chronological order. Each experiment is split into a methods and results section. Once all the experiments pertaining to an objective are explored, a discussion section then uses the results of the objective's experiments to motivate the next set of experiments or reach conclusions on the thesis as a whole.

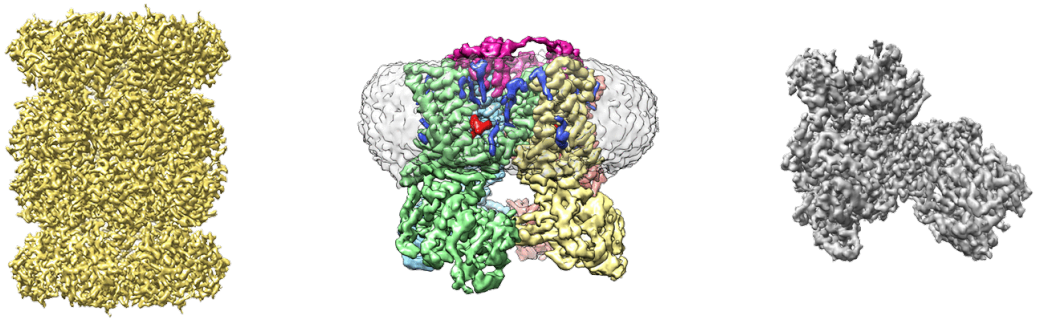
All experiments involved SIREN models, the code implementation for which can be found in Appendix C.

3.1.1 Experimental Protein Structures

All experiments performed for this thesis used one or more of the following protein structures:

1. T20S Proteasome
2. TRPV1 in Complex with DkTx and RTX

3. rNLRP1-rDPP9 Complex



(a) T20S Proteasome 3D Structure

(b) TRPV1 in Complex with DkTx and RTX 3D Structure

(c) rNLRP1-rDPP9 Complex 3D Structure

Figure 3.1: 3D protein structures for T20S proteasome (a), TRPV1 (b), and rNLRP1-rDPP9 (c).

For brevity, the T20S proteasome will be referred to as T20S, the TRPV1 in complex with DkTx and RTX will be referred to as TRPV1, and the rNLRP1-rDPP9 complex will be referred to as rNLRP1.

The structures for these proteins are stored as half maps in files with the extension ".map". Half map files for a plethora of proteins are available on the Electron Microscopy Data Bank (EMDB) hosted by the Protein Data Bank in Europe.[1][2][3] The EMDB is a "public repository for electron microscopy density maps" and is frequently used by scientists and researchers in cryo-EM for acquiring baseline proteins for comparison and for submitting newly found structures [13]. For example, in a peer review by Sjors Scheres to the publication by M. Campbell et al. which described the reconstruction of the T20S proteasome, the authors of the paper were encouraged to upload their data to EMDB for scientific use [14], verifying EMDB as a scientifically accepted source of protein structures in cryo-EM.

The half map files can be loaded into Python as Numpy arrays using the "gemmi" Python library.[4] Storing the half maps as Numpy arrays allow for efficient, multidimensional matrix operations. The code used to load in each protein as a Numpy array can be found in Appendix A.

The Numpy arrays are then converted to a set of coordinates and corresponding density values. The coordinates are used as the inputs to the models and the densities are used as the ground truths. The code for converting a Numpy array representation to this dataset format can be found in Appendix D.1. When methods refer to training models using structures of various shapes, the methods involve using the code in D.1.

[1]The half map for the T20S proteasome can be found here: https://www.ebi.ac.uk/pdbe/entry/emdb/EMD-6287
[2]The half map for the TRPV1 can be found here: https://www.ebi.ac.uk/pdbe/entry/emdb/EMD-8117
[3]The half map for the rNLRP1-rDPP9 can be found here: https://www.ebi.ac.uk/pdbe/entry/emdb/EMD-30458
[4]The gemmi library can be found https://gemmi.readthedocs.io/en/latest/

3.1.2 Generating Model Outputs

Each experiment requires generating protein representations from trained neural implicit representation models to evaluate the training results. To generate the representations, models are input coordinates and output densities which are then combined to form the generated representation. The calculation for the coordinates required to be passed into a SIREN trained on an image of size (y_{max}, x_{max}) to achieve an image of size $(y_{desired}, x_{desired})$ is presented in Equation (3.1).

$$\begin{aligned} coords_{required} = \{(y, x) : y \leq y \leq y_{max}, 0 \leq x \leq x_{max}, \\ y \bmod \frac{y_{max}}{y_{desired}} = 0, x \bmod \frac{x_{max}}{x_{desired}} = 0\} \end{aligned} \tag{3.1}$$

To maintain a consistent range of coordinates within which the models are trained and perform inference on, the coordinates passed into the model for training and inference are scaled between -1 to 1.

The descriptions of methods within this report will refer to training models on protein structures of certain shapes and generating outputs of certain shapes that are not necessarily equal to the shape of the training structure. Doing so involves the coordinate sampling and scaling described in this section. When methods refer to generating models of various shapes, the methods involve using the code in Appendix D.2 to do so. Appendix D contains the code implementation for calculating the coordinates and densities used for training SIREN models. Appendix D also includes code that calculates the coordinates for inference and combines the resulting density outputs to produce representations using SIREN models. The corresponding code for Fourier feature networks can be found in Appendix E.

3.2 Comparison of Neural Network Architectures

The first and most crucial design decision to be made was selecting which neural network architecture to use for the neural implicit representations. The two candidate architectures were the SIREN architecture discussed in Section 2.1, and the Fourier feature network architecture discussed in Section 2.2.

The experiments detailed in Section 3.2 will evaluate the representation quality and training times of each architecture to determine which architecture is best to use for this thesis. See Appendices C and E for the code implementations of both architectures.

3.2.1 Representation Quality Experiment

3.2.1.1 Method

This experiment involved training both models on the x-axis central slice of the T20S, presented in Figure 3.2. The goal of this experiment was to determine which architecture between SIREN and Fourier feature networks could best be used for neural implicit representations of protein structures. This experiment did not test how well the architectures interpolated when generating larger, more detailed structures. While this experiment was performed in the 2D case, the resultant comparisons done during this experiment were assumed to carry over to the 3D case.

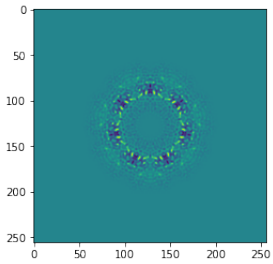


Figure 3.2: Input T20S proteasome x-axis central slice used in comparison experiments

The models were trained for 2500 epochs using each architecture's default hyperparameters. These default parameters were acquired from demos provided by the authors of each architecture's corresponding papers[5][6] and are presented below:

- SIREN hyperparameters:
  - Sinusoidal activation function hyperparameters
    - ω_0 (spatial frequency factor of initial layer) $= 30$
    - $\omega_{0,h}$ (spatial frequency factor of hidden layers and output layer) $= 30$
  - Network architecture hyperparameters
    - Number of input features $= 2$
    - Number of hidden features $= 128$
    - Number of output features $= 1$
    - Number of hidden layers $= 3$
    - Use a linear layer as the final layer $=$ True
  - Training hyperparameters

[5]The demo for the Fourier feature network paper can be found: `https://github.com/tancik/fourier-feature-networks/blob/master/Demo.ipynb`.

[6]The demo for the SIREN paper can be found: `https://colab.research.google.com/github/vsitzmann/siren/blob/master/explore_siren.ipynb`.

        * Learning rate $= 1\text{e}{-4}$

- Fourier feature network hyperparameters:
    - Fourier feature mapping hyperparameters:
        * α_i (Fourier feature amplitudes) $= 0$
        * $\mathbf{b}_i$ (Frequency vectors) $\in \mathbb{R}^{number of features} \sim \mathcal{N}(0, 1)$,
    - Network architecture hyperparameters
        * Number of layers $= 4$
        * Number of channels $= 256$
    - Training hyperparameters
        * Learning rate $= 1\text{e}{-4}$

An Adam optimizer was used to train both models but the loss functions used for each model differed slightly. The SIREN model training used a mean squared error (MSE) loss function while the Fourier feature network used a MSE loss function divided by 2. This selection of optimizer and loss function was replicated from each architecture's respective demos. The demo for the Fourier feature networks converted the half MSE loss to peak signal-to-noise ratio (PSNR) when presenting the training curves.

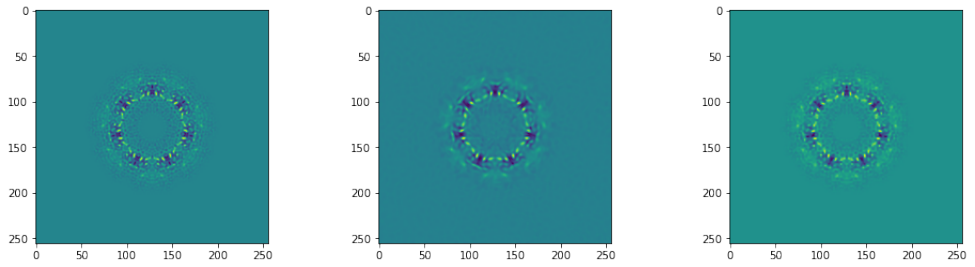
It is important to note that the batch size is not included in the hyperparameters listed earlier because the demos for each architecture used gradient descent instead of stochastic gradient descent. This means the gradients were calculated using the entire dataset instead of batches from the dataset. Gradient descent was switched out for stochastic gradient descent when performing experiments to investigate the 3D case. However, experiments prior to the 3D case investigation used gradient descent, including this experiment.

An excessive 2500 epochs was used because overfitting was not a concern in this experiment. Since this experiment is solely evaluating each architecture's ability to represent proteins without any regard for interpolation, overfitting to the training protein slice would theoretically be the model's best possible representation of the protein disregarding interpolation.

Once the training was completed, both trained models were used to output a reconstruction of the T20S protein slice used to train each model. The quality of each reconstruction was then compared both visually and using FRC plots. The protein slice was cropped from (300, 300) to (256, 256) because the demo for the Fourier feature networks used a training image of shape (256, 256). Thus the slice used for training was of shape (256, 256). The cropped out parts of the protein slice simply contained empty space so no information regarding the protein itself was lost with the cropping.

3.2.1.2 Results

After training both a SIREN model and a Fourier feature network using the x-axis central slice of the T20S structure, the SIREN model outputted the result shown in Figure 3.3b and the Fourier feature network outputted the result shown in Figure 3.3c. A side by side comparison of the original slice and the two reconstructions is shown in Figure 3.3.



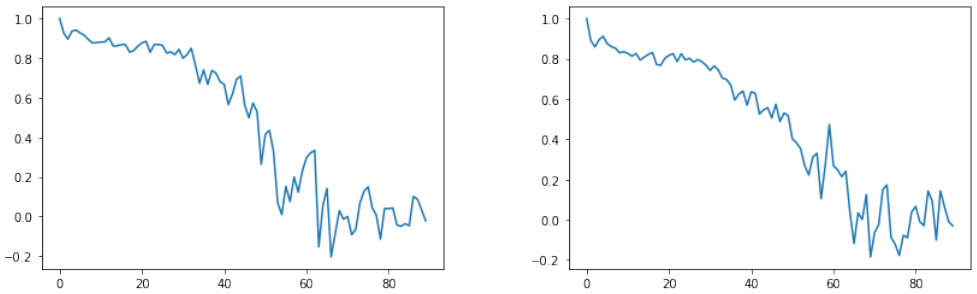
(a) Input T20S proteasome slice

(b) SIREN representation of T20S proteasome

(c) Fourier feature networks representation of T20S proteasome

Figure 3.3: Input protein slice (a), representation generated using SIREN (b), representation generated using a Fourier feature network (c).

These outputs yielded the FRC curves shown in Figure 3.4. These curves were produced using a preliminary method from [15]. This preliminary method presents the spatial frequencies in pixels, which in this case are measured from 0 to 128, as 128 is the Nyquist rate for a structure of shape (256, 256). Since the creation of these FRC curves, different visualization code from [7] has been used to generate the remaining FRC curves. Se Appendix B.2 for the code implementation of the new FRC plots.

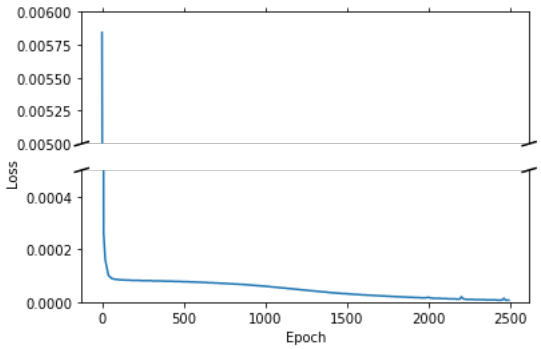


(a) FRC curve for SIREN representation of T20S proteasome

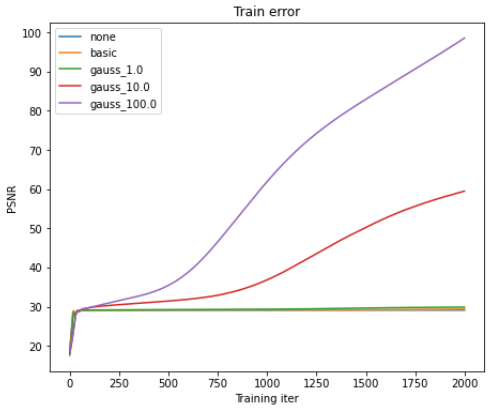
(b) FRC curve for Fourier feature networks representation of T20S proteasome

Figure 3.4: FRC curves for initial testing outputs of T20S proteasome generated using SIREN (b) and Fourier feature networks (b). The x-axis is the spatial frequency in pixels and the y-axis is the FRC value.

The training loss plot for training the SIREN model is shown in Figure 3.5a and the training PSNR plot for training the Fourier feature network is shown in 3.5b.



(a) Training loss plot for SIREN model over 2500 epochs



(b) Training PSNR plot for Fourier feature network over 2500 epochs

Figure 3.5: Training plots for SIREN and Fourier feature network over 2500 epochs.

Interpreting the PSNR plot in Figure 3.5b requires understanding each trend presented. These trends are summarized below:

- none: Training the Fourier feature network without using any Fourier mappings
- basic: Training the Fourier feature network using an identity Fourier feature mapping
- gauss_1.0: Training the Fourier feature network using Gaussian Fourier feature mappings with standard deviation of 1.0.
- gauss_10.0: Training the Fourier feature network using Gaussian Fourier feature mappings with standard deviation of 10.0.
- gauss_100.0: Training the Fourier feature network using Gaussian Fourier feature mappings with standard deviation of 100.0.

In general, training the Fourier feature network using Gaussian Fourier features with as large of a standard deviation results with the best representations. This is demonstrated when visually observing the outputs from training with each type of mapping, shown in Figure 3.6. GT in this figure indicates the ground truth.

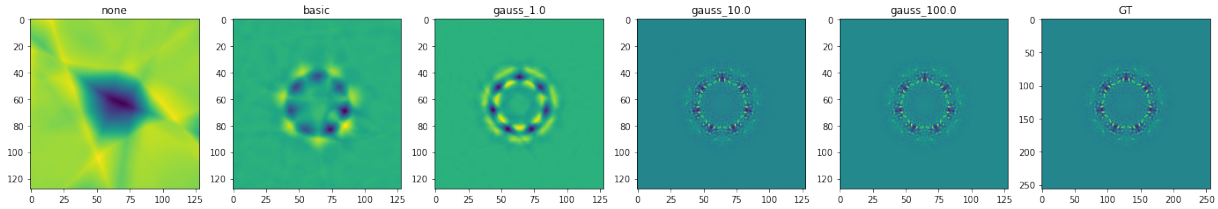


Figure 3.6: Outputs of learnt protein representation of T20S proteasome central slice of x-axis with varying Fourier feature mappings.

Thus when observing Figure 3.5b, the gauss_100.0 trend should be observed. In this case, it seems that despite training for 2500 epochs, the Fourier feature network had not yet converged. This indicates that the architecture converges extremely slowly or the default learning rate was too low for the training to be effective. For the SIREN model, it is evident from its training loss plot that the SIREN model's training converged.

Comparing the FRC curves from Figure 3.4, the two curves are extremely similar. Both demonstrate similar levels of fluctuation that increase at approximately the 55 pixels frequency. However, it does seem that the FRC curve corresponding to SIREN has slightly higher correlation throughout lower spatial frequencies. Overall, there is no significant difference between the two FRC curves and thus it can be concluded that the representations generated by each architecture are approximately equivalent in quality. However, there may be potential for the Fourier feature network to generate an improved representation as its training had yet to converge and plateau.

3.2.2 Training Time Experiment

3.2.2.1 Method

To compare the training times of the SIREN architecture to those of the Fourier feature network architecture, the time required to train a model for 50 epochs was recorded 5 times for each architecture. The 5 training times were then averaged resulting with a mean training time for each architecture. See Appendices C and E for the code implementations of both architectures as well as the training loop implementations for each model.

As was done in the method for the representation quality experiment described in Section 3.2.1.1, the models were trained on the x-axis central slice of the T20S structure using the default hyperparameters and an Adam optimizer. A MSE loss function was used for the SIREN model while a half MSE loss function was used for the Fourier feature network. Gradient descent was used again as opposed to stochastic gradient descent.

In this experiment, only the Fourier feature network with Gaussian Feature mappings with a standard deviation of 100.0 was timed.

3.2.2.2 Results

Table 3.1 presents the training times recorded from this experiment. The training of the Fourier feature network and SIREN were timed over 5 runs. The training times show that the Fourier feature networks approximately took 248.97 seconds to train while SIREN took approximately 0.67391 seconds to train. Based on these results, the SIREN network is approximately 369 times faster than the Fourier feature networks. It should be noted that the number of epochs chosen was only 50. As the number of epochs increases, the difference between the times should increase as well.

	SIREN	Fourier Feature Networks
Run 1	0.64345	248.96
Run 2	0.67908	251.30
Run 3	0.68249	249.84
Run 4	0.68603	248.40
Run 5	0.67852	246.35
Mean	0.67391	248.97

Table 3.1: Comparison of training times (s) between Fourier feature networks and SIREN over 5 runs with an input of shape $(256, 256)$ and 50 epochs.

3.2.3 Discussion of Comparison Experiments

From the results of the representation quality experiment detailed in Section 3.2.1, it is shown that the FRC curve corresponding to each architecture are similar enough such that there is no significant difference between the FRC of the two outputs. If anything, the SIREN representations have higher slightly higher correlation throughout lower spatial frequencies.

The more notable result is from the timing experiment, described in Section 3.2.2, where the SIREN model was demonstrated to train approximately 369 times faster than the Fourier feature networks. It is evident that SIREN is the superior architecture in terms of training speed by a significant margin compared to Fourier feature networks.

It should be noted that the Fourier feature network training had not yet converged in the quality comparison experiment so it is possible representations from the Fourier feature network to improve. However, this indicates that the Fourier feature network would require hyperparameter tuning and/or more training to begin possibly surpassing the quality of the SIREN representation.

Given that the SIREN training converged at approximately epoch 100, with another slight loss improvement at approximately epoch 1600 whereas the Fourier feature network failed to converge

by epoch 2500 while training approximately 369 times slower than the SIREN model, it can be concluded that SIREN is significantly faster than Fourier feature networks while outputting representations that are of equal or greater correlation to the original structure.

These conclusions led to the selection of SIREN over the Fourier feature network as the architecture to use for implicit neural representations.

3.3 Investigation of the 2D Case

After selecting SIREN as the architecture to use for implicit neural representations, the SIREN architecture was explored further in the 2D case. The experiments investigating the 2D case were used to assess the SIREN architecture's potential to represent 3D structures.

3.3.1 SIREN Representation Quality Experiment

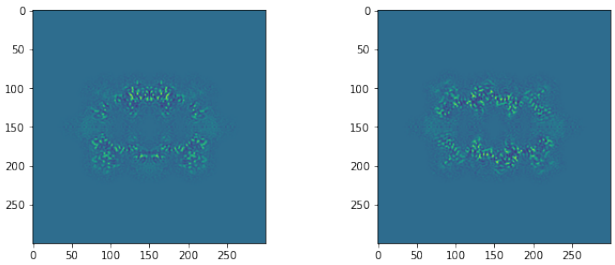
This experiment served multiple purposes. First, it tested the repeatability of SIREN training; whether the similar results can be produced multiple times. Second, it verified whether SIREN models can learn to represent different protein slices and structures or if the results from the quality comparison experiment only applied to the x-axis central slice.

3.3.1.1 Method

This experiment was a repeat of the quality comparison experiment described in Section 3.2.1 with two changes. The changes were that only the SIREN model was used and multiple models were trained using different protein slices. One SIREN model was trained on the x-axis central slice of the T20S proteasome and another SIREN model was trained on the y-axis central slice of the T20S. The y-axis central slice of the T20S is shown in Figure 3.7b. The z-axis central slice was not tested because it is similar to the y-axis central slice, as shown in Figure 3.7b. For this experiment, the protein slices were not cropped, so all the training slices were of shape (300, 300).

Aside from these changes, all other hyperparameters and design decisions of this experiment are identical to the design decisions of the experiment in Section 3.2.1 that pertain to SIREN.

The signal amplitude plot was also observed to determine whether the FRC curve from this experiment and the previous experiment are valid. The code used to generate the amplitude plot can be found in Appendix F.



(a) Input T20S proteasome y-axis central slice

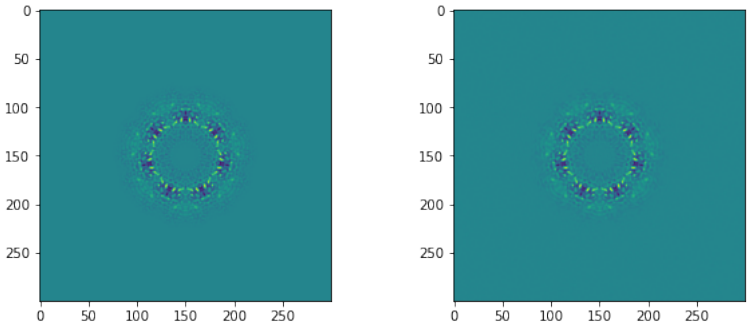
(b) Input T20S proteasome z-axis central slice

Figure 3.7: Y-axis and z-axis central slices of the T20S proteasome.

The code for training the SIREN models can be found in Appendix C.

3.3.1.2 Results

After training a SIREN model on an x-axis central slice for 2500 epochs again, the SIREN model outputted the result shown in Figure 3.11. The loss plot for this training is shown in Figure 3.9. The resultant FRC curve and the amplitude plot are shown in Figure 3.10.



(a) Input T20S proteasome x-axis central slice

(b) SIREN representation of T20S proteasome x-axis central slice

Figure 3.8: Input x-axis central slice of T20S proteasome (a), output fitted using SIREN (b).

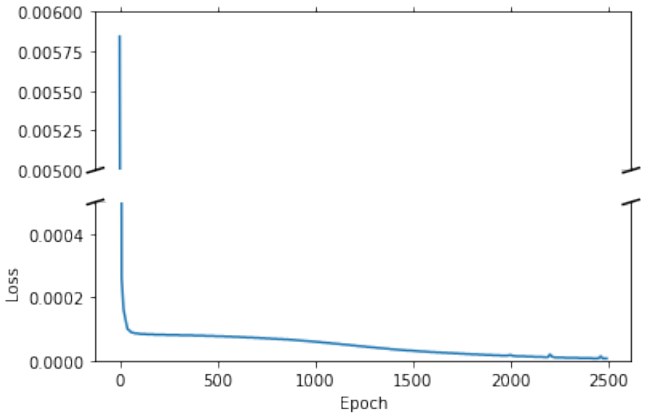
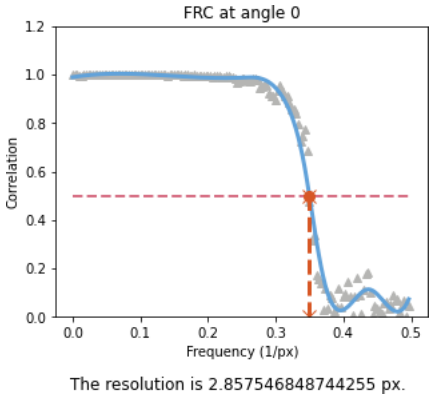
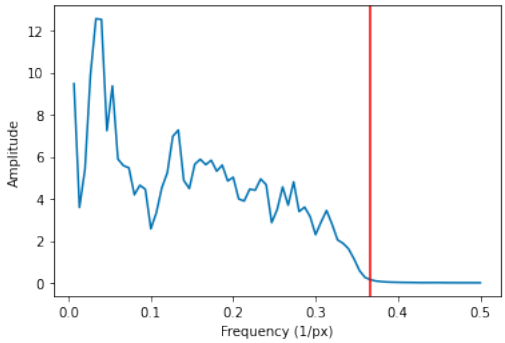


Figure 3.9: Training loss plot for SIREN model learning the T20S x-axis central slice over 2500 epochs



(a) FRC curve for the generated T20S proteasome x-axis central slice



(b) Amplitude for the generated T20S proteasome x-axis central slice

Figure 3.10: FRC curve for learnt reconstruction of the x-axis central slice of T20S proteasome (a), and the corresponding amplitude plot (b).

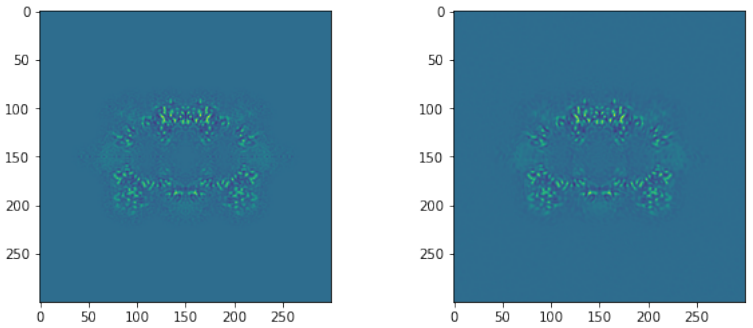
Visually, the reconstruction of the x-axis central slice looks extremely similar to the original. The loss plot also indicates that training has converged.

Observing the FRC curve, the correlation stays extremely close to 1 until approximately $0.3\frac{1}{px}$. The FRC then rapidly drops, reaching 0 at approximately $0.39\frac{1}{px}$. The resolution for this representation is approximately $2.86px$, which equates to approximately 2.80 Å. This is because for the T20S structure used, one pixel is equal to 0.98 Å. As the T20S structure that is used to train the models has a resolution 2.80 Å[13] this acquired resolution is almost ideal.

Using the amplitude plot to determine the validity of the FRC plot simply involves observing which frequencies have negligible amplitudes. As the FRC and FSC calculations fail at frequencies with negligible amplitudes, these are the frequencies at which high FRC and FSC values can no longer be trusted. It is observed in this case that the amplitudes falls below 0.25, an arbitrarily

chosen amplitude threshold for negligible amplitudes, at approximately 0.37 to 0.38 $\frac{1}{px}$ as indicated by the red line in the amplitude plot. This corresponds to the approximate frequency at which the FRC values falls below 0.2. This indicates that the FRC values up to and including the resolution are reliable. Thus, the resolution of the FRC curve can be trusted. Amplitude plots were used to verify the validity of all FSC and FRC curves presented in this thesis document but for brevity, the amplitude plots will henceforth only be mentioned and shown when they indicate that the FSC or FRC curves cannot be trusted.

The SIREN model trained on the y-axis central slice for 2500 epochs outputted the result shown in Figure 3.7b. The corresponding loss plot and FRC curve are shown in Figures 3.12 and 3.13 respectively.



(a) Input T20S proteasome y-axis central slice

(b) SIREN representation of T20S proteasome y-axis central slice

Figure 3.11: Input y-axis central slice of T20S proteasome (a), output fitted using SIREN (b).

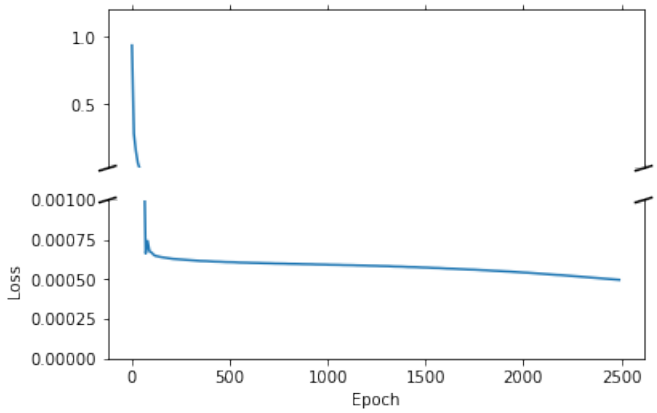


Figure 3.12: Loss plot for training a SIREN model to learn the y-axis central slice of T20S proteasome for 2500 epochs.

Visually, the reconstruction of the y-axis central slice looks extremely similar to the original.

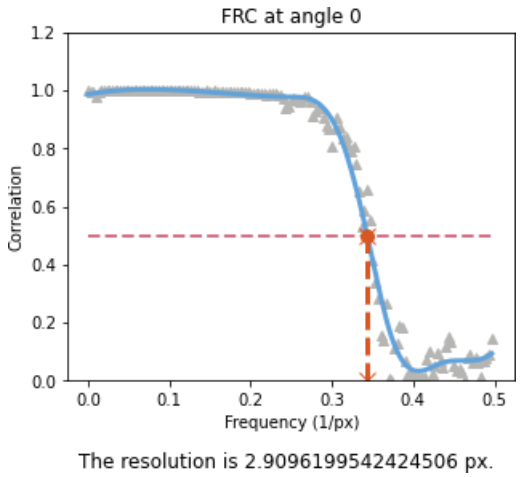


Figure 3.13: FRC curve for the generated T20S proteasome y-axis central slice.

The loss seems to have mostly converged, although the end of the loss plot shows the loss beginning to decrease even further. It should be noted that this training converged at a loss of approximately 0.0005, while the training on the x-axis central slice converged at a loss that is negligibly different from 0.

Observing the FRC curve, the correlation stays extremely close to 1 until approximately $0.27\frac{1}{px}$. The FRC then rapidly drops, reaching 0 at approximately $0.4\frac{1}{px}$. The resolution for this representation is approximately $2.9px$, which equates to approximately 2.85 Å using the same calculation from earlier in this section. This resolution approaches the optimal resolution of 2.8 Å.

3.3.2 Memory Experiment

Experiments to determine the memory cost of the SIREN architecture are crucial as representations of 3D structures often have high memory costs that limit the maximum number of parameters of the representation. This experiment observed the memory cost of scaling the number of hidden features per layer and the batch size from 1 to 256 when training using 2D representations. The results of this experiment can be used to infer the memory cost of 3D trained models as the GPU memory of a 3D SIREN is a cubic function of the GPU memory of a 2D SIREN.

It should be noted that in the results section of this experiment, Section 3.3.2.2, the term ”output side length” is used in the figures. The output side length in this case refers to the number of coordinates used to train the model and is functionally equivalent to the batch size in this experiment but the figures use the term output side length because at the time of this memory experiment, gradient descent was used and the model had yet to to be used to generate reconstructions of different shapes. As the output side length is equivalent to batch size in the context of this experiment, the explanations will refer to the ”output side length” as the ”batch size”.

3.3.2.1 Method

In this experiment, SIREN models with varying number of hidden features and batch sizes were instantiated. The resultant memory cost was then recorded at each number of hidden features and batch size. The other hyperparameters pertaining to the model's memory cost were set to their default parameters, as presented below:

- Number of input features $= 2$
- Number of output features $= 1$
- Number of hidden layers $= 3$
- Use a linear layer as the final layer $=$ True

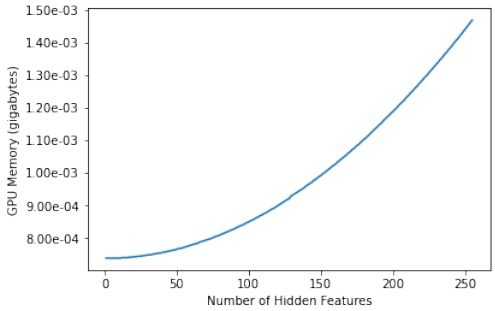
Three different memory tests were run. First, the number of hidden features was varied between 1 to 256 while the batch size was kept at 256. Then the batch size was varied between 1 to 256 while the number of hidden features was kept at 256. Finally, both the number of hidden features and the batch size were varied between 1 to 256. In this case, the memory was recorded at each combination of hyperparameters, resulting with 255^2 recorded memory values.

The number of hidden features per layer was varied because the memory cost of the number of parameters can be approximated by multiplying the memory cost of the number of hidden features per layer by the number of hidden layers. The batch size was also varied because it scales proportionately with the input sampling rate. Thus, the impact of batch size on GPU memory usage is indicative of the effect of sampling rate on GPU memory.

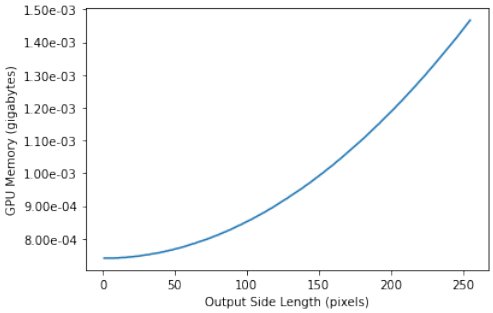
The code function `torch.cuda.memory_allocated()` was used to measure the memory cost of the instantianted SIREN models.

3.3.2.2 Results

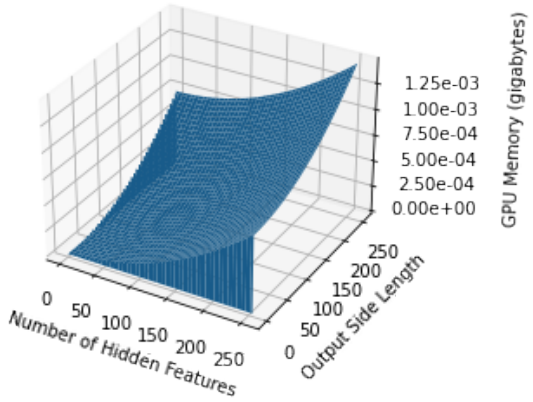
Figure 3.14a shows the relationship between the number of hidden features and the GPU memory when using a batch size of 256 coordinates. Figure 3.14b shows the relationship between the batch size and the GPU memory when using 256 hidden features per layer. Figure **??** shows the relationship between both the number of hidden features and batch size to the GPU memory.



(a) GPU memory (gigabytes) over the number of hidden features



(b) GPU memory (gigabytes) over the batch size (pixels)



(c) 3D plot of GPU memory over number of hidden features and batch size (pixels)

Figure 3.14: Plots demonstrating relationship between GPU memory and various hyperparameters (number of hidden features and batch size).

Observing Figure 3.14a and Figure 3.14b, the GPU memory scales with the number of hidden features and the batch size in nearly identical exponential fashion, indicating that both parameters affect the memory equally. Observing Figure 3.14c, the memory usage of SIREN is approximately $1.25\text{e}{-3}$ gigabytes when using 256 hidden features per layer and side length of 256.

These results indicate that while increasing either the number of hidden features or the batch size will both increase GPU memory costs exponentially, the GPU memory costs are ultimately small as the memory peaked at $1.25\text{e}{-3}$ gigabytes or 1.25 megabytes in this experiment.

3.3.3 Discussion of 2D Case Experiments

The results from the SIREN quality experiment demonstrate that the SIREN model is capable of representing different protein slices with excellent resolutions. While this experiment did not

demonstrate interpolation using the SIREN architecture, it did demonstrate its potential for representing protein structures as a whole.

The results of the memory experiment were also promising as they indicated that the using SIREN for 3D representations will have lower memory costs than expected. Thus, training SIREN models to learn 3D implicit neural representations can be trained with a much greater number of parameters than expected. Furthermore, it was realized after the memory experiment that the 3D models can be trained using stochastic gradient descent instead of gradient descent. If gradient descent were used when to learn a 3D structure of shape (300, 300, 300), then the memory cost would be that of using a batch size of 300^3. However, stochastic gradient descent enables a predefined batch size, which makes the GPU memory independent of the training structure shape. Training with stochastic gradient descent enables a model learning a structure of shape (300, 300, 300) to require the same amount of memory as a model learning a structure of shape (batch size, 1, 1) using gradient descent so long as the hyperparameters are the same. This substantially decreases the memory costs of learning 3D protein neural implicit representations. However, the impact of switching from gradient descent to stochastic gradient descent outside of memory improvements was unclear as stochastic gradient descent was untested at this point.

These results showed that SIREN can effectively represent 2D slices of proteins with sufficiently low memory costs. From these findings, it was then inferred that SIREN has the potential to effectively represent 3D protein structures while remaining memory efficient. Thus it was decided that experiments involving the 3D case would commence.

3.4 Investigation of the 3D Case

Once experiments in the 2D case provided sufficient evidence that the SIREN could potentially work in the 3D case, the 3D case was explored. These experiments not only tested how well trained SIREN models could represent 3D protein structures, but also the models' ability to interpolate to larger shapes.

For all experiments investigating the 3D case, the Adam optimizer and a MSE loss function was used.The experiments also switched from gradient descent to stochastic gradient descent for model training. The hyperparameter "Number of input features" was also switched from 2 to 3 for these experiments because this indicates to the SIREN model that it is learning 3D representations.

Most experiments exploring the 3D case used all three proteins presented in Section 3.1.1. The pixel to Angstrom ratios for each protein are listed below. These ratios were used to calculate the resolutions in these experiments.

- T20S: $1px = 0.98$ Å

- TRPV1: $1px = 1.22$ Å
- rNLRP1: $1px = 1.06$ Å

The proteins were downsampled by a factor of 2 when training the models so that the models can output an interpolated structure which can then be compared to the non-downsampled ground truth structure. This enables testing of the model's ability to interpolate without having to use unreliable interpolation techniques on the ground truth protein. The protein structures were also sometimes cropped across all three dimensions to increase the proportion of training coordinates that contained relevant protein densities. The T20S structure had 50 pixels cropped from each side along each dimension before downsampling. The cropping values were chosen manually based on visual interpretation of the structure's central slices. As the structure was originally of shape (300, 300, 300), the transformations resulted with a structure of shape (100, 100, 100). The TRPV1 nor the rNLRP1 were not cropped. This is because the approximate cropping number of pixels to crop at each side was determined to be 30 and 40 for the TRPV1 and rNLRP1 respectively. This would result shapes of (66, 66, 66) and (80, 80, 80) for the TRPV1 and rNLRP1 respectively. These shapes were thought to be too small for the SIREN model to effectively learn the neural implicit representation. Thus cropping was not used for these two proteins in most experiments. Experiments that do crop these proteins will mention so in their corresponding method section. The applied transformations are summarized in Table 3.2.

	T20S	TRPV1	rNLRP1
Initial Shape (px)	(300, 300, 300)	(192, 192, 192)	(240, 240, 240)
Cropping at Each Side (px)	50	0	0
Downsampling Factor	2	2	2
Final Shape (px)	(100, 100, 100)	(96, 96, 96)	(120, 120, 120)

Table 3.2: Summary of transformations applied to each protein structure prior to training.

For brevity, the loss plots will not be presented in these experiments as they all simply show that the training converged and plateaued for all models.

All models were trained using the code found in Appendix C.

3.4.1 Hyperparameter Optimization Experiment

This experiment aimed to perform hyperparameter optimizations to improve the performance of SIREN models on learning 3D neural implicit representations.

3.4.1.1 Method

This experiment involved training models on the cropped and downsampled T20S structure described in Section 3.4. While optimizing the hyperparameters, the model outputs using various hyperparameter configurations were only verified visually. To do so, the trained SIREN model was used to output a 3D representation of the same shape as the input structure. The x-axis central slice was then extracted from this reconstruction and compared to the x-axis central slice from the structure used for training.

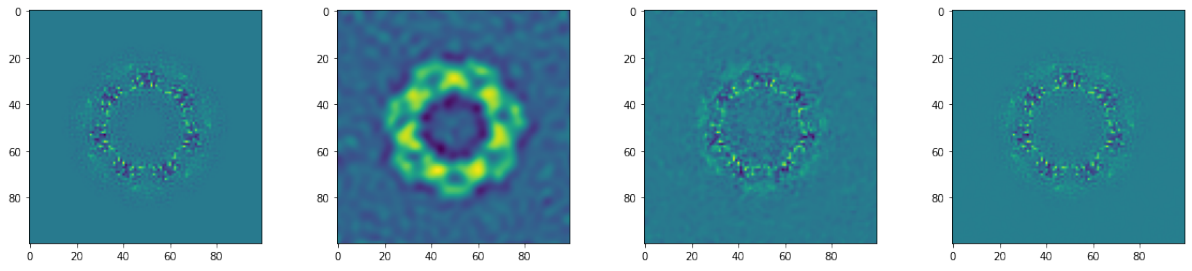
First, the SIREN model was trained on the T20S protein structure using a slight variation of the default parameters listed in Section 3.2.1.1. The parameters changed from the default values aside from the number of input features are listed below:

- Training hyperparameters
  - Number of epochs $= 5000$
  - Learning rate $= 1e{-}5$
  - Batch size $= 256$

When these hyperparameters proved to yield poor results, the number of hidden layers was increased to 5. The number of hidden layers was then further increased to 7. Afterwards, the impact of all other hyperparameters was evaluated visually in the same fashion.

3.4.1.2 Results

The visualization of the x-axis central slice reconstruction after training with 3, 5, and 7 layers are presented in Figure 3.15.



(a) Ground truth

(b) Reconstruction from training with 3 hidden layers

(c) Reconstruction from training with 5 hidden layers

(d) Reconstruction from training with 7 hidden layers

Figure 3.15: X-axis central slices of T20S for the ground truth (a) and reconstructions trained using 3 (b), 5 (c), and 7 (d) hidden layers.

From these visualizations, it was decided that 7 hidden layer will be used to conduct the in-depth 3D case experiments. Through similar tests scaling the batch size, SIREN omega values, and the number of hidden features, it was found that 512 hidden features per layer, a batch size of 256, and omega values of 50 performed best. Training using these hyperparameters were also found to converge at approximately epochs 120, 300, 100 for the T20S, TRPV1, and rNLRP1 respectively. Thus, the hyperparameters used for the remaining 3D case experiments are listed below. An Adam optimizer and a MSE loss function continued to be used for the remaining experiments.

- Sinusoidal activation function hyperparameters
  - ω_0 (spatial frequency factor of initial layer) $= 50$
  - $\omega_{0,h}$ (spatial frequency factor of hidden layers and output layer) $= 50$
- Network architecture hyperparameters
  - Number of input features $= 3$
  - Number of hidden features $= 512$
  - Number of output features $= 1$
  - Number of hidden layers $= 7$
  - Use a linear layer as the final layer $=$ True
- Training hyperparameters
  - Number of epochs $= 120$ for T20S, 300 for TRPV1, 100 for rNLRP1
  - Learning rate $= 1\text{e}{-5}$
  - Batch size $= 256$

3.4.2 3D Downsampled Representation Experiment

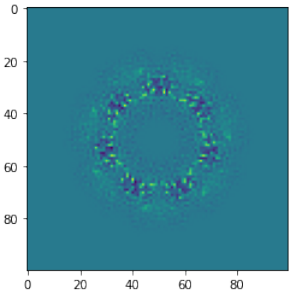
This experiment served to determine the SIREN model's capacity to learn downsampled 3D protein structures.

3.4.2.1 Method

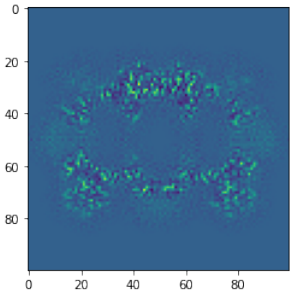
A SIREN model for each baseline protein was trained using the hyperparameters determined in Section 3.4.1.2. The trained models then generated structures of the same shape as the training proteins. The reconstructed structures were evaluated using the resultant FSC curves. The shapes of the training protein structures and the generated structures were as described in Table 3.4.

3.4.2.2 Results

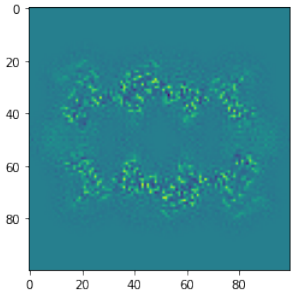
The central slices along each axis for the T20S protein generated by the corresponding trained SIREN model are compared to their ground truths in Figure 3.16. The FSC curve is presented in Figure 3.17. The slices comparison and FSC curve for the TRPV1 are presented in Figures 3.18 and 3.19. The slices comparison and FSC curve for the rNLRP1 are presented in Figures 3.20 and 3.21.



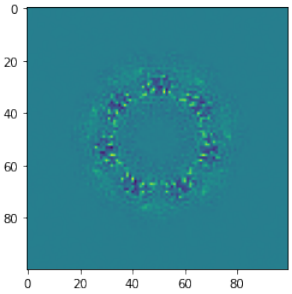
(a) X-axis central slice of ground truth T20S proteasome



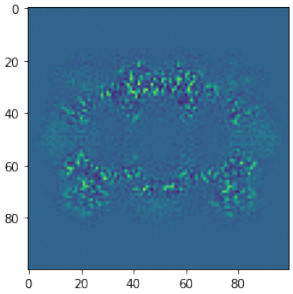
(b) Y-axis central slice of ground truth T20S proteasome



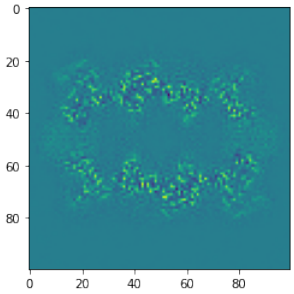
(c) Z-axis central slice of ground truth T20S proteasome



(d) X-axis central slice of generated T20S proteasome



(e) Y-axis central slice of generated T20S proteasome



(f) Z-axis central slice of generated T20S proteasome

Figure 3.16: Visualized central slices of ground truth and generated structure for the T20S proteasome from the downsampled representation experiment.

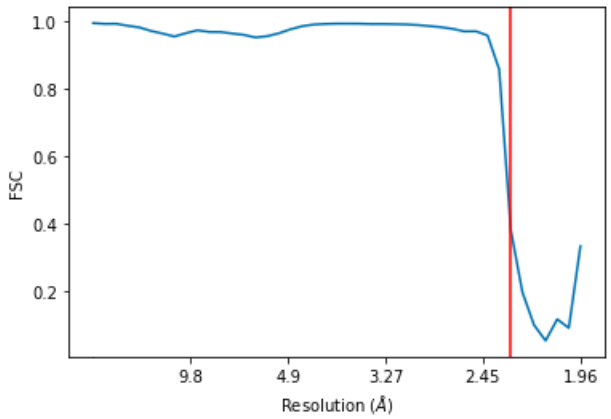
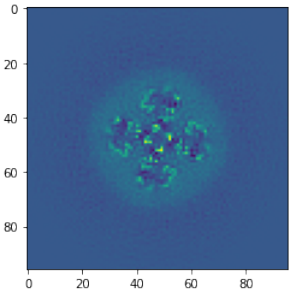
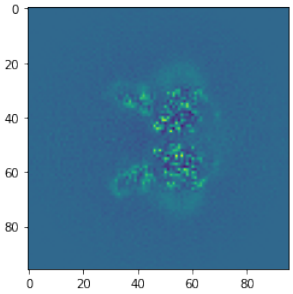


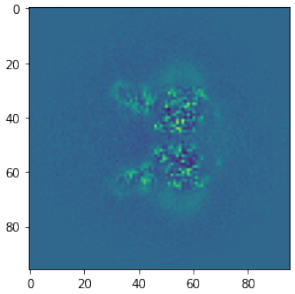
Figure 3.17: FSC curve for the generated 3D T20S proteasome structure of shape (100, 100, 100). The resolution is 2.2866666666666666 Å.



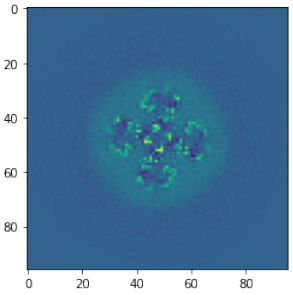
(a) X-axis central slice of ground truth TRPV1 proteasome



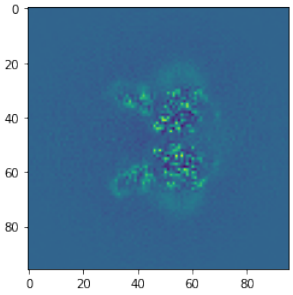
(b) Y-axis central slice of ground truth TRPV1 proteasome



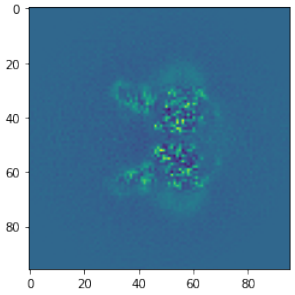
(c) Z-axis central slice of ground truth TRPV1 proteasome



(d) X-axis central slice of generated TRPV1 proteasome



(e) Y-axis central slice of generated TRPV1 proteasome



(f) Z-axis central slice of generated TRPV1 proteasome

Figure 3.18: Visualized central slices of ground truth and generated structure for the TRPV1 from the downsampled representation experiment.

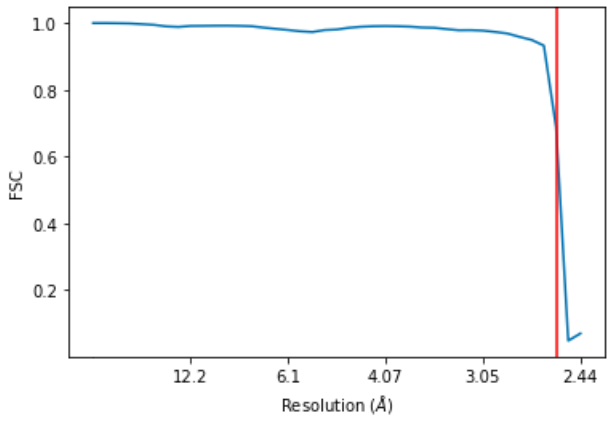
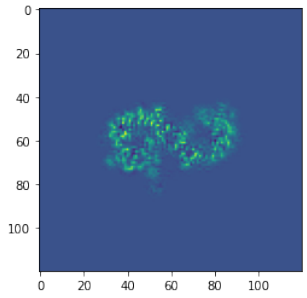
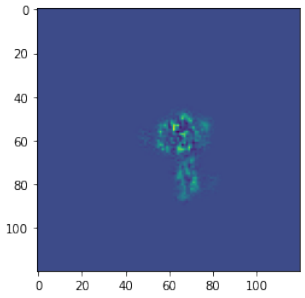


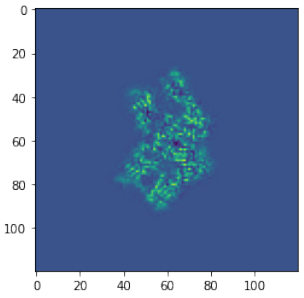
Figure 3.19: FSC curve for the generated 3D TRPV1 structure of shape (96, 96, 96). The resolution is 2.568421052631579 Å.



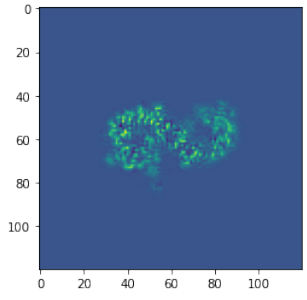
(a) X-axis central slice of ground truth rNLRP1 proteasome



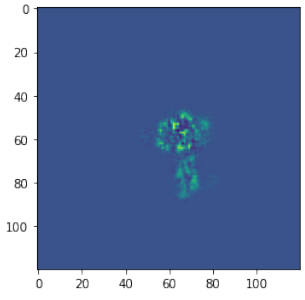
(b) Y-axis central slice of ground truth rNLRP1 proteasome



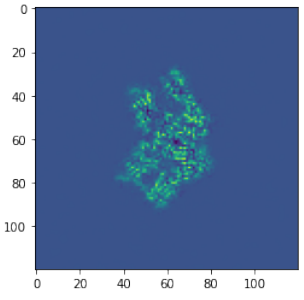
(c) Z-axis central slice of ground truth rNLRP1 proteasome



(d) X-axis central slice of generated rNLRP1 proteasome



(e) Y-axis central slice of generated rNLRP1 proteasome



(f) Z-axis central slice of generated rNLRP1 proteasome

Figure 3.20: Visualized central slices of ground truth and generated structure for the rNLRP1 from the downsampled representation experiment.

Visually, the slices from the generated representations shown in Figures 3.16, 3.18, and 3.20 are indistinguishable from the ground truth slices.

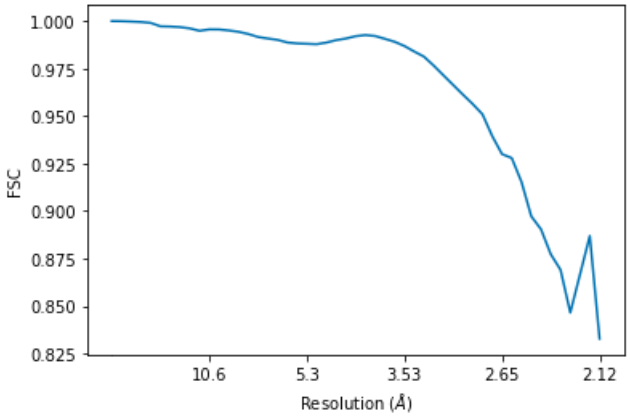


Figure 3.21: FSC curve for the generated 3D rNLRP1 structure of shape (120, 120, 120). The resolution could not be calculated from the FSC curve because the FSC never drops below 0.5 before the Nyquist rate. Thus the resolution is set equal to the resolution of the original structure, 3.18 Å.

The FSC curves for the T20S and TRPV1 are both quite excellent, as the FSC values remain near 1 until a low frequency at which the FSC drops off. The resultant resolutions are approximately 2.3 Åand 2.6 Åfor the T20S and TRPV1 respectively. These resolutions have lower values than the resolutions of the original structures, which are 2.8 Åand 2.95 Åfor the T20S and TRPV1 respectively. This is likely because the resolutions acquired in this experiment are from downsampled structures while the resolutions of the original structures are from non-transformed protein structures. Regardless, the resolutions for these two reconstructions are sufficiently low such that these reconstructions can be considered state of the art and excellent enough to be used in scientific practice.

The FSC curve for the rNLRP1, shown in 3.21, is different from the other FSC curves in that the reconstruction was so accurate that the FSC curve does not reach the 0.5 threshold before the Nyquist rate. Thus a resolution cannot be calculated from the FSC curve. Instead, the resolution is assumed to be that of the original structure which is 3.18 Å. Having a FSC curve that does not reach a signal to noise ratio of 1 prior to the Nyquist rate indicates a nearly ideal reconstruction.

Based on these results, it can be concluded that the SIREN models can represent the downsampled protein structures at state of the art resolutions.

3.4.3 Memory Experiment

Experimenting with the memory and number of parameters is important because these represent two of the bottlenecks that occupancy pose in protein representations as described in 1.2. Thus these experiments directly inform if SIREN and neural implicit representations can improve upon occupancy arrays in certain aspects.

3.4.3.1 Method

The RAM requirement and the number of parameters of the SIREN models from the 3D down-sampled representation experiment from Section 3.4.2 were measured. The RAM and number of parameters for these models were then compared to those of the corresponding occupancy array representations, stored as saved Numpy arrays. This served to evaluate the SIREN model's memory and parameter efficiency in the 3D case.

The RAM requirements of the SIREN model were determined using the following code where the mem variable contains the RAM requirements in bytes:

```
mem_params = sum([param.nelement()*param.element_size() for param in
↪  siren_model.parameters()])
mem_bufs = sum([buf.nelement()*buf.element_size() for buf in
↪  siren_model.buffers()])
mem = mem_params + mem_bufs # in bytes
```

The RAM requirements of the occupancy array representation in bytes were determined by using the code `structure.nbytes` where structure is a Numpy array containing the representation.

The number of parameters for the SIREN models was determined using the following code:

```
mem = sum(p.numel() for p in siren_model.parameters() if p.requires_grad)
```

The number of parameters within each Numpy occupancy arrays was calculated by counting the number of elements in the Numpy array. This can simply be done using the code `structure.size`.

It is important to note that the SIREN models and Numpy arrays used to represent the structures were of data type float32.

3.4.3.2 Results

The results of this experiment are summarized in Tables 3.3 and 3.4 below. It should be noted that as the same architecture was used for all three protein representations, the SIREN memory requirements and number of parameters will not vary as the represented protein changes.

	T20S	TRPV1	rNLRP1
SIREN Model	1847300	1847300	1847300
Occupancy Array	4000000	3538944	6912000

Table 3.3: Summary of RAM memory requirements of SIREN models and occupancy arrays in bytes.

	T20S	TRPV1	rNLRP1
SIREN Model	461825	461825	461825
Occupancy Array	1000000	884736	1728000

Table 3.4: Summary of number of parameters in SIREN models and occupancy arrays.

From these tables, it can be seen that the SIREN model outperforms the occupancy array in terms of both memory and number of parameters across all three represented proteins. The memory and number of parameters required by occupancy arrays increase with the representation size but the SIREN models do not. Thus if the SIREN models learned the non-transformed proteins, the memory discrepancy between the SIREN models and the occupancy arrays would be even greater than it already is in these results. To use a simple example, a common shape of protein representations is (300, 300, 300). An occupancy array of this size with data type float32 would have a memory cost of 108000000 bytes and 27000000 parameters. The SIREN models would have approximately 58 times less memory and number of parameters.

Thus it can be concluded that SIREN models are more efficient than occupancy arrays in terms of memory and number of parameters.

3.4.4 3D Interpolation Experiment

This experiment served to evaluate the SIREN models' ability to interpolate protein representations that are larger and more detailed than the structure used to train the model. Ideally, the SIREN models would be capable of producing reconstructions of the same shape as the non-downsampled ground truth structures with resolutions approximately equal to those found in Section 3.4.2.2.

3.4.4.1 Method

Similarly to the 3D downsampled representation experiment done in Section 3.4.2, a SIREN model for each baseline protein was trained using the hyperparameters determined in Section 3.4.1.2. Each model was trained using transformed proteins with the shapes presented in Table 3.2. However, the difference between this experiment and the 3D downsampled representation experiment is that the trained SIREN models generated models of shapes equal to those of the non-downsampled ground truth protein structures. It should be noted that the ground truth T20S structure used in this experiment was still cropped by 50 pixels at each side of each axis.

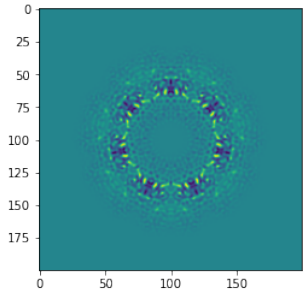
The shape of the structures used for training and the shape of the structures generated using the models in this experiment are summarized in Table 3.5.

	T20S	TRPV1	rNLRP1
Training Shape (px)	(100, 100, 100)	(96, 96, 96)	(120, 120, 120)
Generated Shape (px)	(200, 200, 200)	(192, 192, 192)	(240, 240, 240)

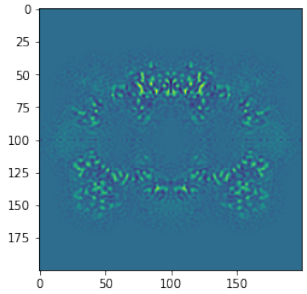
Table 3.5: Shape of structures used for training and shape of generated reconstructions for each protein in the interpolation experiment.

3.4.4.2 Results

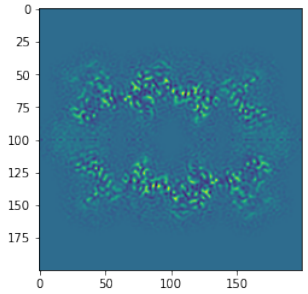
The central slices for the T20S structure generated by the corresponding trained SIREN model are compared to their ground truths in Figure 3.22. The FSC curve is presented in Figure 3.23. The visualizations of the generated structure's central slices for the TRPV1 and its FSC curve are presented in Figures 3.24 and 3.25 respectively. The visualizations of the generated structure's central slices for the rNLRP1 and its FSC curve are presented in Figures 3.26 and 3.27 respectively.



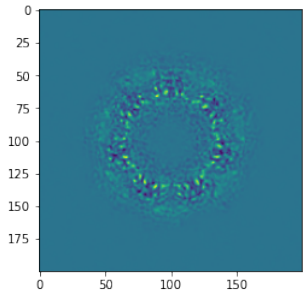
(a) X-axis central slice of ground truth T20S proteasome



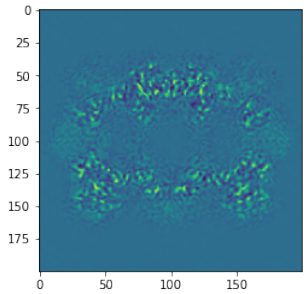
(b) Y-axis central slice of ground truth T20S proteasome



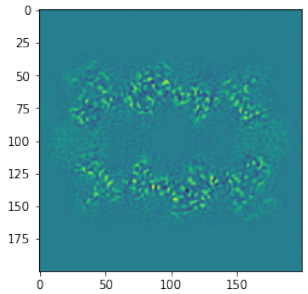
(c) Z-axis central slice of ground truth T20S proteasome



(d) X-axis central slice of generated T20S proteasome



(e) Y-axis central slice of generated T20S proteasome



(f) Z-axis central slice of generated T20S proteasome

Figure 3.22: Visualized central slices of ground truth and generated structure for the T20S proteasome from the interpolation experiment.

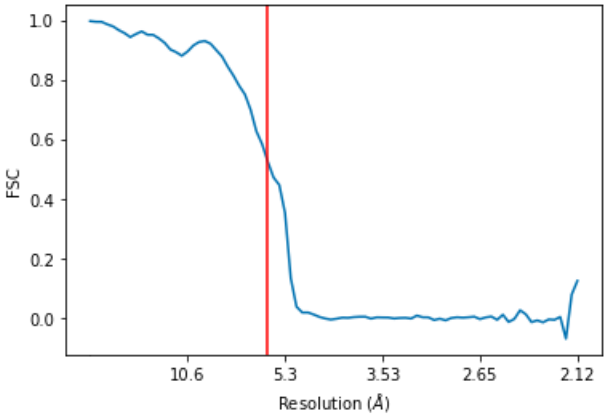
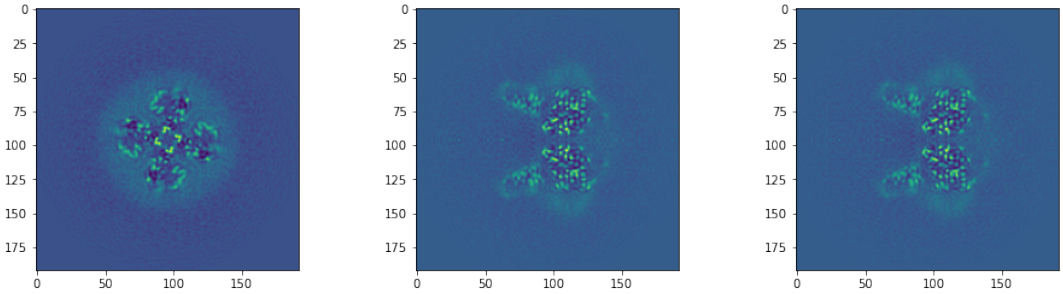


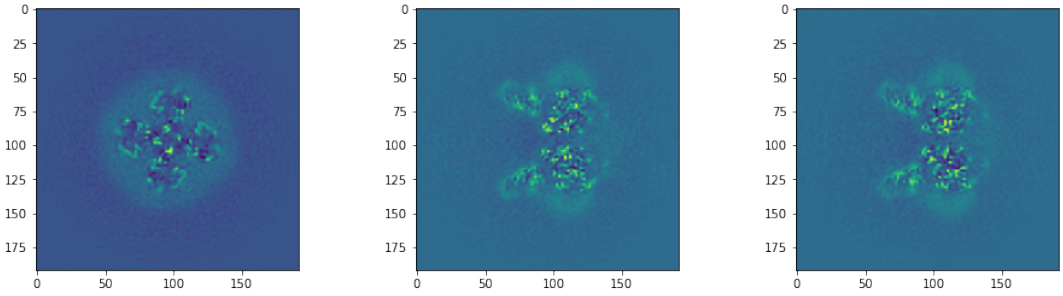
Figure 3.23: FSC curve for the generated 3D T20S proteasome structure of shape (200, 200, 200) when trained on a structure of shape (100, 100, 100). The resolution is 5.812903225806452 Å.



(a) X-axis central slice of ground truth TRPV1 proteasome

(b) Y-axis central slice of ground truth TRPV1 proteasome

(c) Z-axis central slice of ground truth TRPV1 proteasome



(d) X-axis central slice of generated TRPV1 proteasome

(e) Y-axis central slice of generated TRPV1 proteasome

(f) Z-axis central slice of generated TRPV1 proteasome

Figure 3.24: Visualized central slices of ground truth and generated structure for the TRPV1 from the interpolation experiment.

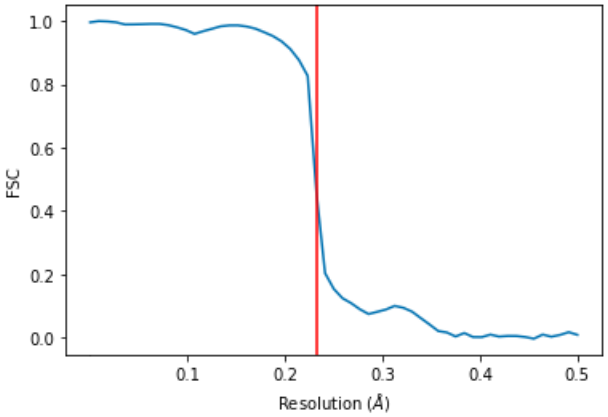
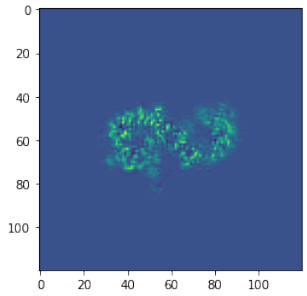
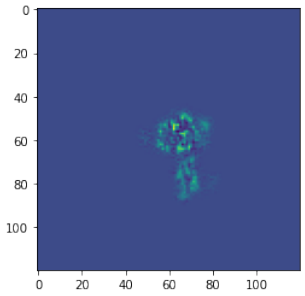


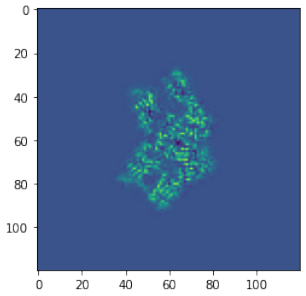
Figure 3.25: FSC curve for the generated 3D TRPV1 structure of shape (192, 192, 192) when trained on a structure of shape (96, 96, 96). The resolution is 5.255384615384616.



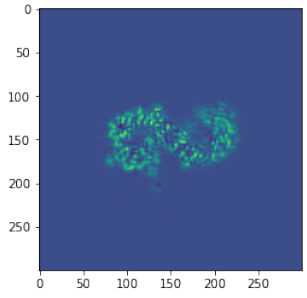
(a) X-axis central slice of ground truth rNLRP1 proteasome



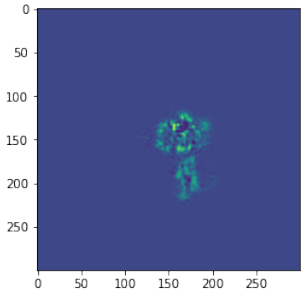
(b) Y-axis central slice of ground truth rNLRP1 proteasome



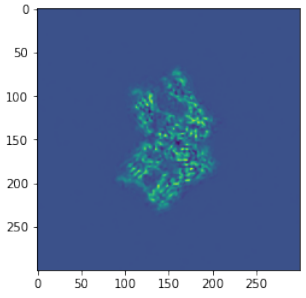
(c) Z-axis central slice of ground truth rNLRP1 proteasome



(d) X-axis central slice of generated rNLRP1 proteasome



(e) Y-axis central slice of generated rNLRP1 proteasome



(f) Z-axis central slice of generated rNLRP1 proteasome

Figure 3.26: Visualized central slices of ground truth and generated structure for the rNLRP1 from the interpolation experiment.

Visually, the slices from the generated representations shown in Figures 3.22, 3.24, and 3.26 seem to have the same pattern as the ground truths. However, the colors of many of the generated

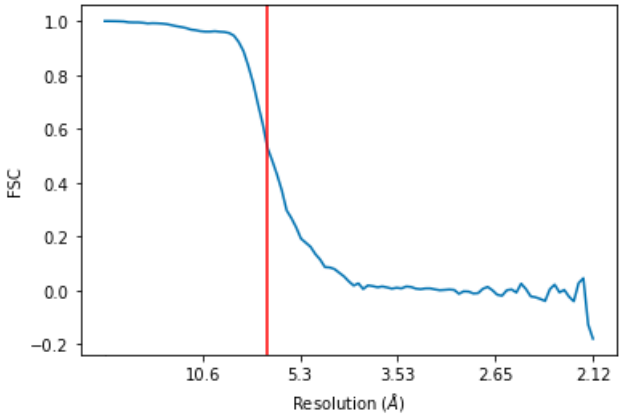


Figure 3.27: FSC curve for the generated 3D rNLRP1 structure of shape (240, 240, 240) when trained on a structure of shape (120, 120, 120). The resolution is 6.36 Å.

slices differ from their corresponding ground truths. This indicates that the models were capable of interpolating the relative density patterns of the protein structures but were unable to interpolate the correct density values themselves.

The FSC curves for the T20S, TRPV1, and rNLRP1 are all significantly worse compared to the FSC curves from Section 3.4.2.2. The FSC values drop at a much higher resolution value than when generating representations that were of the same shape as the training structure, resulting with a worse final resolution. A comparison of resolutions when generating non-interpolated and interpolated structures are presented in Table 3.6 below.

	T20S	TRPV1	rNLRP1
Non-interpolated Downsampled Output Resolution (Å)	2.29	2.57	3.18
Interpolated Output Resolution (Å)	5.81	5.26	6.36

Table 3.6: Comparison of approximate resolutions for generated non-interpolated and interpolated structures.

Based on these results, it can be concluded that the SIREN models trained on downsampled proteins cannot easily interpolate to larger protein structures.

This leads to the question of why the interpolation structures are poor. There are three possible causes for this issue. First, the hyperparameters may still be tuned poorly. Second, the model may be too small to represent proteins of such size. Finally, the model may be incapable of interpolating well.

3.4.5 Hyperparameter Investigation Experiment

This experiment investigated if poorly tuned hyperparameters were a possible cause of the mediocre interpolated representations generated in the interpolation experiment presented in Section 3.4.4. This experiment also provides insight in how the investigated hyperparameters affect the resolution of interpolated 3D structures.

3.4.5.1 Method

For this experiment, the number of hidden features was varied between the values of 256, 512, and 768 while the omega value of the SIREN model was varied between the values of 30, 50, and 70. For each combination of number of hidden features and omega, a model was trained on each downsampled protein structure. Each trained model then generated an interpolation structure whose resolution was then calculated. The code for acquiring the resolution can be found in Appendix B.1.

3.4.5.2 Results

The resolutions acquired from this experiment are detailed in Tables 3.7, 3.8, and 3.9, which present the resolutions for T20S, TRPV1, and rNLRP1 respectively. Table entries of -1 indicate that the experiment at those hyperparameters failed to acquire a realistic resolution, likely due to a failure in the training. Many resolutions are repeated in the tables because the FSC calculation rounds to small resolution intervals. Thus, table entries with the same resolution value have extremely similar resolutions.

		Omega		
		30	50	70
Number of Hidden Features	256	5.37419355	5.20626	5.55333333
	512	-1	4.9	5.04848485
	768	5.04848485	5.20625	5.37419355

Table 3.7: Resolutions of interpolated structures of T20S with changing number of hidden channels and changing omega.

		Omega		
		30	50	70
Number of Hidden Features	256	7.19157895	6.832	6.832
	512	6.832	6.50666667	6.50666667
	768	6.50666667	6.50666667	6.50666667

Table 3.8: Resolutions of interpolated structures of TRPV1 with changing number of hidden channels and changing omega.

		Omega		
		30	50	70
Number of Hidden Features	256	6.00666667	6.00666667	6.26782609
	512	6.00666667	6.26782609	6.26782609
	768	6.26782609	6.26782609	6.26782609

Table 3.9: Resolutions of interpolated structures of rNLRP1 with changing number of hidden channels and changing omega.

As shown in these tables, the resolutions of the T20S and TRPV1 interpolated structures seem to improve as the number of hidden features and omega increase but the rNLRP1 demonstrates an opposite trend. This suggests that there exists an optimal hyperparameter configuration unique to each protein. However, the difference in resolutions is ultimately quite small. This highly suggests a weak correlation between the number of hidden channels and omega to the final resolution.

3.4.6 Model Capacity Investigation Experiment

This experiment investigate if the mediocre interpolated representations generated in the interpolation experiment from Section 3.4.4, are caused by the used SIREN models being too small to represent proteins of the size that they were interpolated to.

3.4.6.1 Method

Like in the 3D downsampled representation experiment done in Section 3.4.2, a SIREN model for each baseline protein was trained using the hyperparameters determined in Section 3.4.1.2. This experiment differs in that each model was trained using non-downsampled protein structures. The trained models were then used to generate structures of the same shape. The FSC curves and resolutions of the structures were then observed.

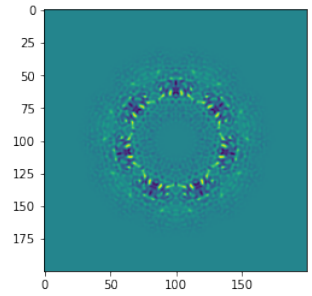
For this experiment, the TRPV1 and rNLRP1 proteins were cropped by 30 and 40 pixels respectively at each side. This cropping removed empty space in the structure, which was hypothesized

to help the models better learn the structures. The shape of the structures used for training and the shape of the structures generated using the models in this experiment are summarized in Table 3.10.

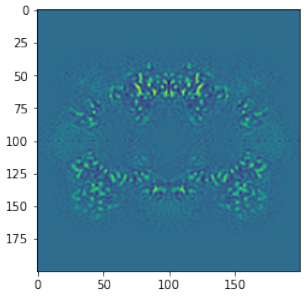
	T20S	TRPV1	rNLRP1
Initial Shape (px)	(300, 300, 300)	(192, 192, 192)	(240, 240, 240)
Cropping at Each Side (px)	50	30	40
Downsampling Factor	1	1	1
Final Shape (px)	(200, 200, 200)	(132, 132, 132)	(160, 160, 160)

Table 3.10: Summary of transformations applied to each protein structure prior to training.

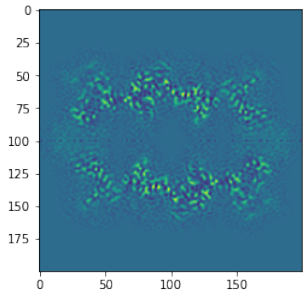
3.4.6.2 Results



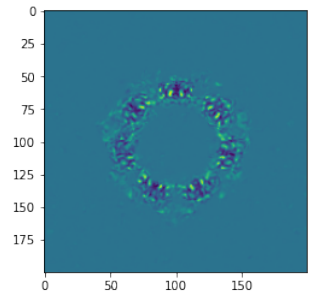
(a) X-axis central slice of ground truth T20S proteasome



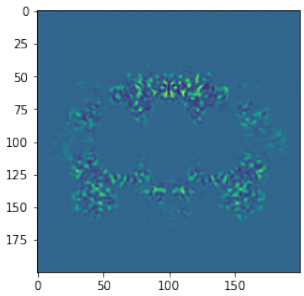
(b) Y-axis central slice of ground truth T20S proteasome



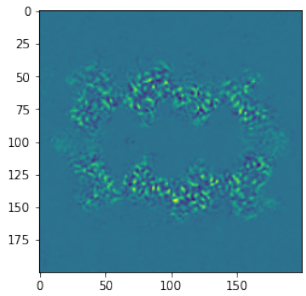
(c) Z-axis central slice of ground truth T20S proteasome



(d) X-axis central slice of generated T20S proteasome



(e) Y-axis central slice of generated T20S proteasome



(f) Z-axis central slice of generated T20S proteasome

Figure 3.28: Visualized central slices of ground truth and generated structure for the T20S proteasome from the model capacity experiment.

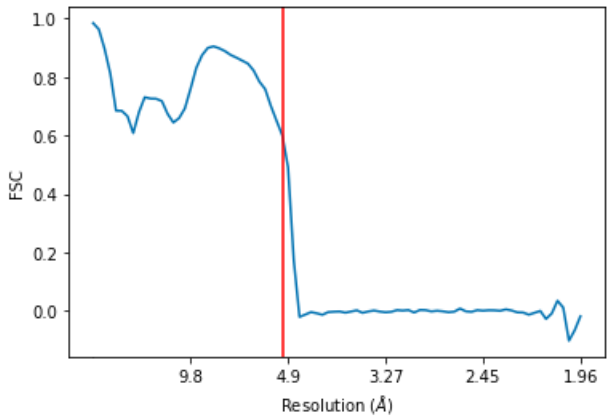
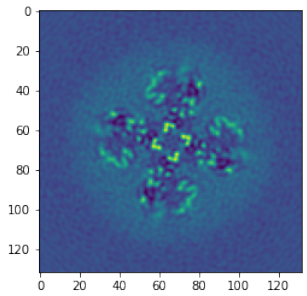
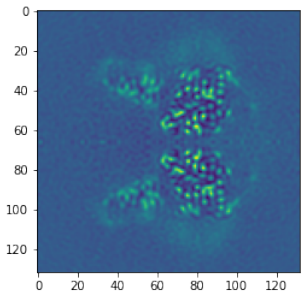


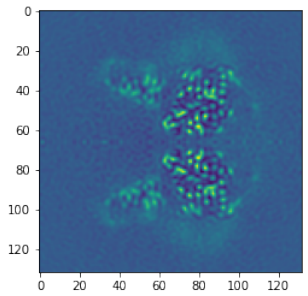
Figure 3.29: FSC curve for the generated 3D T20S proteasome structure of shape (200, 200, 200) when trained on a structure of shape (200, 200, 200). The resolution is 5.048484848484848 Å.



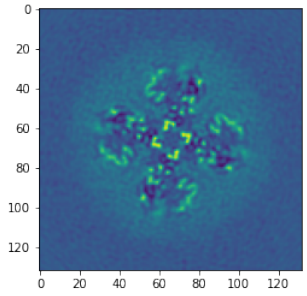
(a) X-axis central slice of ground truth TRPV1 proteasome



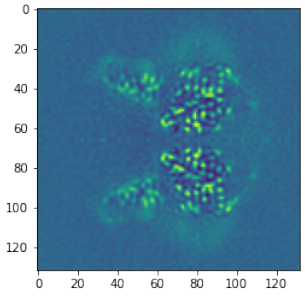
(b) Y-axis central slice of ground truth TRPV1 proteasome



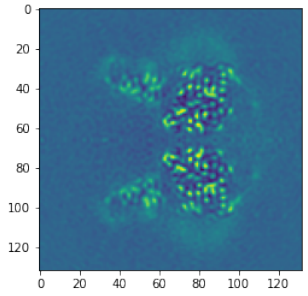
(c) Z-axis central slice of ground truth TRPV1 proteasome



(d) X-axis central slice of generated TRPV1 proteasome



(e) Y-axis central slice of generated TRPV1 proteasome



(f) Z-axis central slice of generated TRPV1 proteasome

Figure 3.30: Visualized central slices of ground truth and generated structure for the TRPV1 from the model capacity experiment.

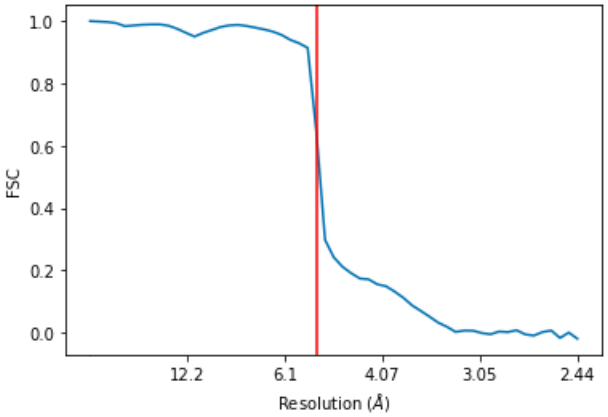
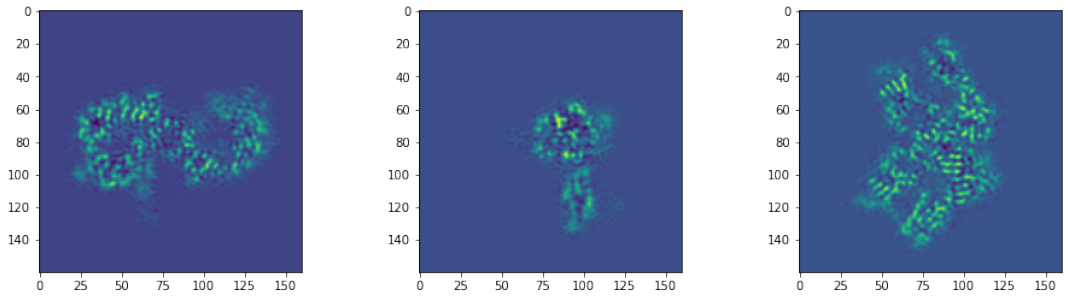


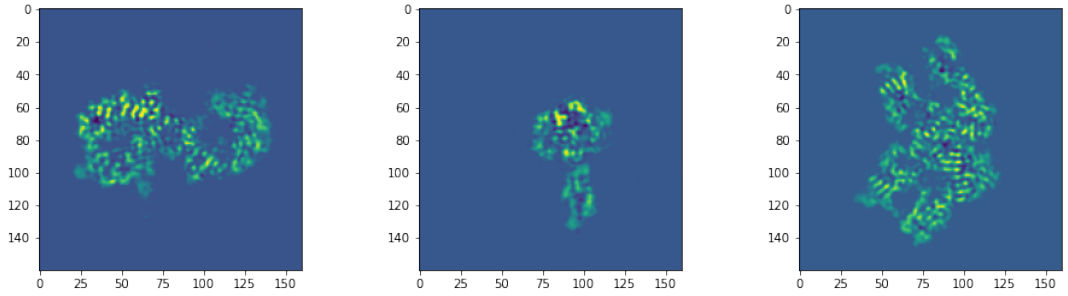
Figure 3.31: FSC curve for the generated 3D TRPV1 structure of shape (132, 132, 132) when trained on a structure of shape (132, 132, 132). The resolution is 5.255384615384616.



(a) X-axis central slice of ground truth rNLRP1 proteasome

(b) Y-axis central slice of ground truth rNLRP1 proteasome

(c) Z-axis central slice of ground truth rNLRP1 proteasome



(d) X-axis central slice of generated rNLRP1 proteasome

(e) Y-axis central slice of generated rNLRP1 proteasome

(f) Z-axis central slice of generated rNLRP1 proteasome

Figure 3.32: Visualized central slices of ground truth and generated structure for the rNLRP1 from the model capacity experiment.

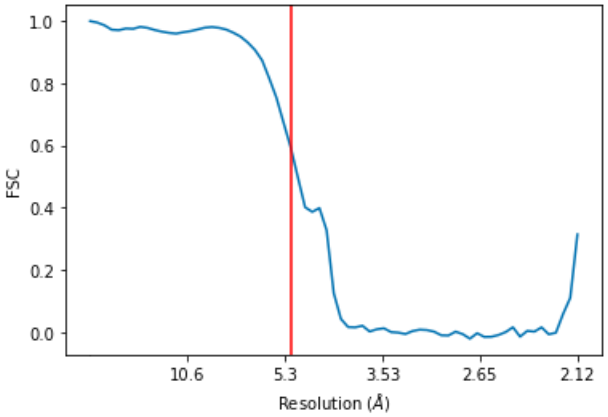


Figure 3.33: FSC curve for the generated 3D rNLRP1 structure of shape (160, 160, 160) when trained on a structure of shape (160, 160, 160). The resolution is 5.1485714285714295 Å.

The slices from the generated representations shown in Figures 3.28, 3.30, and 3.32 seem to have the same pattern as the ground truths but have different colors. This is the similar to the results from the interpolation experiment presented in Section 3.4.4.2 and once again indicates that the models were capable of interpolating the relative density patterns of the protein structures but not the correct density values themselves.

The FSC curves for all proteins, shown in Figures 3.29, 3.31, and 3.33, seem to still be as poor as they were in the interpolation experiment. Table 3.11 presents the resolutions from this model capacity experiment as well as the resolutions from the interpolation experiment.

	T20S	TRPV1	rNLRP1
Non-interpolated Downsampled Output Resolution (Å)	2.29	2.57	3.18
Interpolated Output Resolution (Å)	5.81	5.26	6.36
Non-interpolated Non-downsampled Output Resolution (Å)	5.10	5.26	5.15

Table 3.11: Comparison of approximate resolutions for generated non-interpolated downsampled structures, and interpolated structures, and non-interpolated non-downsampled structures.

Observing this table, it seems that the resolutions from this experiment are better than the resolutions from the interpolation experiment but are still insufficient. The improvement in resolution is possibly due to one or more of the following reasons:

- Cropping to reduce empty space learned during training.
- Lack of interpolation done in this experiment.
- Overfitting slightly more to the structure due to the reduced shape from cropping.

Regardless of the cause, it can be concluded that the SIREN models used for training were insufficient in representing larger protein structures. This is likely due to the model being too small.

3.4.7 Algorithmic Interpolation Experiment

The experiment described in this section investigates if the poor interpolated representations generated in the interpolation experiment, presented in Section 3.4.4, are possibly the result of the trained models being unable to interpolate effectively.

3.4.7.1 Method

To investigate how well the trained SIREN models interpolate, the models trained in the interpolation experiment described in Section 3.4.4 were used to generate structures of the same shape as the downsampled proteins used to train the models. The generated structures were then interpolated to the same shape as the non-downsampled proteins using bicubic interpolation. Bicubic interpolation is a common algorithmic interpolation method and should poorly interpolate protein structures due to the how complex the protein structures are. The structures interpolated using bicubic interpolation was then evaluated against the structures generated in the interpolation experiment by comparing their FSC curves and resolutions. If the resultant resolutions are better than that of the resolutions from the interpolation experiment, then it is possible that one of the reasons for the poor results from the interpolation experiment was the models interpolating poorly.

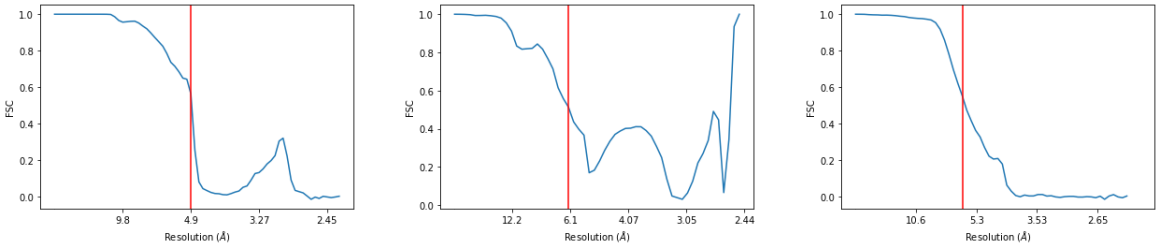
3.4.7.2 Results

The FSC curves for the bicubic interpolated T20S, TRPV1, and rNLRP1 structures are shown in Figure 3.34. The corresponding resolutions for T20S, TRPV1, and rNLRP1 are 4.9 Å, 6.210909090909091 Å, and 6.006666666666668 Å respectively.

Table 3.12 compares the resolutions acquired from interpolating with bicubic interpolation to the resolutions acquired from interpolating using the SIREN model from Section 3.4.4.

	T20S	TRPV1	rNLRP1
SIREN Interpolated Output Resolution (Å)	5.81	5.26	6.36
Bicubic Interpolated Output Resolution (Å)	6.21	6.21	6.01

Table 3.12: Comparison of approximate resolutions for generated non-interpolated downsampled structures, and interpolated structures, and non-interpolated non-downsampled structures.



(a) FSC curve of generated T20S structure interpolated using bicubic interpolation

(b) FSC curve of generated TRPV1 structure interpolated using bicubic interpolation

(c) FSC curve of generated rNLRP1 structure interpolated using bicubic interpolation

Figure 3.34: FSC curve of generated structures interpolated using bicubic interpolated for T20S (a), TRPV1 (b), and rNLRP1 (c).

From Table 3.12, it can be seen that the bicubic interpolation performed worse than the SIREN interpolation for the T20S and TRPV1 proteins but performed slightly better for the rNLRP1 protein. This can also be observed in the FSC curves presented in Figures 3.34a and 3.34b. The FSC values for the T20S and TRPV1 proteins fluctuate greatly past a certain resolution, which indicates that bicubic interpolation interpolates poorly as spatial frequencies approach the Nyquist rate. This can be compared to the FSC plots generated from interpolating with SIREN presented in Figures 3.23, 3.25, and 3.27 which only feature small fluctuations. Meanwhile, the FSC plot for the rNLRP1 structure interpolated using bicubic interpolation has very few fluctuations as can be seen in Figure 3.34c. The improved results the bicubic interpolated rNLRP1 structure has over the SIREN interpolated rNLRP1 structure is possibly due to the rNLRP1 protein having a structure that is conducive to effective interpolation using bicubic interpolation. Although it should be noted that all the resolutions from the bicubic interpolations are ultimately still very poor.

Thus it can be concluded that the SIREN models likely interpolated better than bicubic interpolation, which indicates that the SIREN models' ability to interpolate is likely not the primary cause of the poor interpolation results from the interpolation experiment.

3.4.8 Discussion of 3D Case Experiments

Based on the experimental results acquired from investigating the 3D case, various deductions and conclusions can be made. First, based on the results of the memory experiment shown in Section 3.4.3.2, it is evident that SIREN models have the potential to be extremely memory efficient means of representing protein structures, regardless of interpolation. Second, based on the results of the interpolation experiment described in Section 3.4.4, it seems that training SIREN models to effectively interpolate to larger and more detailed structure sizes is a nontrivial, difficult task. Third, the bottlenecks to effective interpolation using SIREN is not necessarily the architecture's ability to

interpolate, rather it's likely the architecture's ability to represent larger proteins in general. This is indicated by the model capacity experiment and the algorithmic interpolation experiment detailed in Sections 3.4.6 and 3.4.7 respectively. Thus it is likely that the number of parameters in the model must be increased by increasing the number of hidden features per layer and/or increasing the number of layers. However, the hyperparameter investigation experiment from Section 3.4.5 indicated that increasing the number of hidden features per layer does not have a significant impact on the resolution. Thus it can be inferred that rather than increasing the number of hidden features per layer, increasing the number of layers in the SIREN model is likely the key to improving the resolution of protein structures interpolated using SIREN, as well as the resolution of larger protein structures represented using SIREN in general. This is further corroborated by the results of the initial hyperparameter tuning experiment detailed in Section 3.4.1.2, where it was shown that increasing the number of hidden layers can have a significant effect on 3D protein representations.

Chapter 4

Conclusion

This thesis has demonstrated that SIREN models and neural implicit representations demonstrate great potential in being extremely efficient means of storing finite protein representations. SIREN models have also demonstrated potential for atomic resolution via interpolation of larger protein representations. However, this thesis has also shown that training the SIREN models to effectively represent larger proteins and interpolate protein representations to larger, more details structures is a difficult task. The investigations performed in this thesis have shown that increasing the number of layers in the SIREN model is likely the key to training SIREN models that can successfully represent larger proteins and effectively interpolate to higher resolution structures.

As of now, SIREN models are not ready to be used for atomic resolution representations in cryo-EM. Based on the conclusions of this thesis, future work that can be done to eventually develop atomic resolution representations using SIREN models. First, the hyperparameters of the SIREN models can be further tuned with an emphasis on increasing the number of layers in the model. Second, potential architecture changes can be made to the SIREN models to better suit implicit neural representations of protein structures. One such change is passing the SIREN model Fourier transformed inputs akin to Fourier feature networks. This can be done by splitting the real and imaginary parts of Fourier transformed densities and using these pairs of values as the ground truths. The model would then output the real and imaginary parts of a density value which can then be combined to form the Fourier transformed density at the given coordinate. Alternatively two SIREN models can be trained where one learns to generate the real part of the density values and the other learns to generate the imaginary part. Alternative model initializations can also be explored. One such alternative is training the models on a downsampled structure, and then using the optimized weights as an initialization for training on a larger form of the same protein structure. If SIREN models reach the point where they can be used for atomic resolution representations of 3D protein structures, SIREN models can then be explored for potential use in representing protein dynamics such as learning the physical reactions of protein structures to thermodynamic forces.

References

[1] E. Callaway, "Revolutionary cryo-EM is taking over structural biology", *Nature*, vol. 578, no. 201, 2020.

[2] A. Punjani and D. J. Fleet, "3D Variability Analysis: Resolving continuous flexibility and discrete heterogeneity from single particle cryo-EM," 2020.

[3] M. V. Heel and M. Schatz, "Fourier shell correlation threshold criteria," *Journal of Structural Biology*, vol. 151, no. 3, pp. 250–262, 2005.

[4] M. Tanaka, "TOP," *JEOL*, 15-May-2020. [Online]. Available: `https://www.jeol.co.jp/en/words/emterms/search_result.html?keyword=Fourier%20Shell%20Correlation%2C%20FSC`. [Accessed: 24-Jan-2021].

[5] "11. Correlation and regression," *The BMJ: leading general medical journal*, 24-Nov-2020. [Online]. Available: `https://www.bmj.com/about-bmj/resources-readers/publications/statistics-square-one/11-correlation-and-regression`. [Accessed: 24-Jan-2021].

[6] J. R. Feathers, K. A. Spoth, and J. C. Fromme, "Surpassing the physical Nyquist limit to produce super-resolution cryo-EM reconstructions," bioRxiv, 2019.

[7] S. Koho, G. Tortarolo, M. Castello, T. Deguchi, A. Diaspro, and G. Vicidomini, "Fourier ring correlation simplifies image restoration in fluorescence microscopy," *Nature Communications*, vol. 10, no. 1, 2019.

[8] EMBL-EBI Cellular Structure and 3D Bioimaging Team, "EMPIAR-10025," Electron Microscopy Public Image Archive (EMPIAR). [Online]. Available: https://www.ebi.ac.uk/pdbe/emdb/empiar/entry/10025/. [Accessed: 31-Jan-2021].

[9] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, G. Wetzstein, "Implicit Neural Representations with Periodic Activation Functions," 2020, *arXiv:2006.09661*.

[10] R. Hassan, A. Mohammed, J. Idrissi, M. Amine, G. Youssef, and E. Mohamed. (2016). ”Multilayer Perceptron: Architecture Optimization and Training,” *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 4, pp. 26-30, 10.9781/ijimai.2016.415.

[11] A. F. Agarap, ”Deep Learning using Rectified Linear Units (ReLU),” 2018, arXiv:1803.08375.

[12] M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng, “Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains,” 2020, arXiv:2006.10739.

[13] “Protein Data Bank in Europe,” The Electron Microscopy Data Bank. [Online]. Available: https://www.ebi.ac.uk/pdbe/emdb/. [Accessed: 13-Apr-2021].

[14] S. H. W. Scheres, “Decision letter: 2.8 Å resolution reconstruction of the Thermoplasma acidophilum 20S proteasome using cryo-electron microscopy,” 2015.

[15] “Fourier ring correlation of images,” Fourier ring correlation of images - tttrlib documentation, Jan-2019. [Online]. Available: `https://tttrlib.readthedocs.io/en/latest/auto_examples/imaging/plot_imaging_frc.html`. [Accessed: 25-Jan-2021].

Appendix A

Code for Importing Protein Structure

A.1 T20S Proteasome Import Code

```
!pip install gemmi
import os
import numpy as np
import scipy
import gemmi

# Getting test T20S map
if not os.path.exists('emd_6287.map.gz'):
    !wget ftp://ftp.ebi.ac.uk/pub/databases/emdb/structures/EMD-6287/
    map/emd_6287.map.gz
if not os.path.exists('emd_6287.map'):
    !gunzip 'emd_6287.map.gz'
map_path = 'emd_6287.map'

test_map = gemmi.read_ccp4_map(map_path)
test_map.setup()  # optional

raw_structure = np.array(test_map.grid, copy=False)

structure = np.expand_dims(raw_structure, axis=-1)
structure = structure[50:-50, 50:-50, 50:-50, :] # Cropping
```

```
structure = scipy.ndimage.zoom(structure, (0.5, 0.5, 0.5, 1)) #
↪ Downsampling
```

A.2 TRPV1 in Complex with DkTx and RTX Import Code

```
!pip install gemmi
import os
import numpy as np
import scipy
import gemmi

# Getting test TRPV1 map
if not os.path.exists('emd_8117.map.gz'):
    !wget ftp://ftp.ebi.ac.uk/pub/databases/emdb/structures/EMD-8117/
    map/emd_8117.map.gz
if not os.path.exists('emd_8117.map'):
    !gunzip 'emd_8117.map.gz'

test_map = gemmi.read_ccp4_map('emd_8117.map')
test_map.setup()  # optional

raw_structure = np.array(test_map.grid, copy=False)

structure = np.expand_dims(raw_structure, axis=-1)
structure = structure[30:-30, 30:-30, 30:-30, :] # Cropping
structure = scipy.ndimage.zoom(structure, (0.75, 0.75, 0.75, 1)) #
↪ Downsampling
```

A.3 rNLRP1-rDPP9 Import Code

```
!pip install gemmi
import os
import numpy as np
import scipy
import gemmi
```

```
# Getting test rNLRP1-rDPP9 map
if not os.path.exists('emd_30458.map.gz'):
    !wget ftp://ftp.ebi.ac.uk/pub/databases/emdb/structures/EMD-30458/
    map/emd_30458.map.gz
if not os.path.exists('emd_30458.map'):
    !gunzip 'emd_30458.map.gz'

test_map = gemmi.read_ccp4_map('emd_30458.map')
test_map.setup()  # optional

raw_structure = np.array(test_map.grid, copy=False)

structure = np.expand_dims(raw_structure, axis=-1)
structure = structure[40:-40, 40:-40, 40:-40, :] # Cropping
structure = scipy.ndimage.zoom(structure, (0.5, 0.5, 0.5, 1)) #
↪  Downsampling
```

Appendix B

Code for Producing FSC and FRC Plots

B.1 Code for generating FSC plot

The code used to produce the FSC plots is based on the FRC calculations code from ttrlib [15].
The px_to_A variable contains the pixel to Angstrom ratio, detailed in Section 3.4.

```
# Based on
↪  https://tttrlib.readthedocs.io/en/latest/auto_examples/imaging/plot_imaging_frc.ht

# Arguments
shell_size = 2
epsilon = 1e-7

original_image = structure
pred_image = final_siren_output

# Calculations
F1 = np.fft.fftn(original_image.squeeze() / np.sum(original_image),
↪  axes=(0,1,2))
F2 = np.fft.fftn(pred_image.squeeze() / np.sum(pred_image), axes=(0,1,2))

F1F2 = np.real(F1 * np.conj(F2))
F1_abs_squared = np.abs(F1)**2
F2_abs_squared = np.abs(F2)**2

nx, ny, nz = F1F2.shape
```

```
x = np.arange(-np.floor(nx / 2.0), np.ceil(nx / 2.0))
y = np.arange(-np.floor(ny / 2.0), np.ceil(ny / 2.0))
z = np.arange(-np.floor(nz / 2.0), np.ceil(nz / 2.0))

distances = []
numerators = []
F1_norms = []
F2_norms = []

for xi, yi, zi in np.array(np.meshgrid(x, y, z)).T.reshape(-1, 3):
    distances.append(np.sqrt((xi ** 2) + (yi ** 2) + (zi ** 2)))
    xi = int(xi)
    yi = int(yi)
    zi = int(zi)

    numerators.append(F1F2[xi, yi, zi])
    F1_norms.append(F1_abs_squared[xi, yi, zi])
    F2_norms.append(F2_abs_squared[xi, yi, zi])

bins = np.arange(0, np.sqrt(((nx // 2) ** 2) + ((ny // 2) ** 2) + ((nz // 2)
↪  ** 2)), shell_size)
F1F2_histogram, bin_edges = np.histogram(
    distances,
    bins=bins,
    weights=numerators
)
F1_norm_histogram, bin_edges = np.histogram(
    distances,
    bins=bins,
    weights=F1_norms
)
F2_norm_histogram, bin_edges = np.histogram(
    distances,
    bins=bins,
    weights=F2_norms
)
```

```
FSC_vals = F1F2_histogram / np.sqrt(F1_norm_histogram * F2_norm_histogram)

def convert_freq_to_resolution(frequencies, px_to_A):
    resolutions = 1 / frequencies
    resolutions[resolutions == np.inf] = 0
    resolutions *= px_to_A

    return resolutions

# Setting up x-axis
frequencies = np.linspace(0, 0.5, num=len(FSC_vals))
px_to_A = 1.06 # rNLRP1-rDPP9 dataset specific
resolutions = convert_freq_to_resolution(frequencies, px_to_A)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(frequencies, FSC_vals)
ax.set_xticklabels(np.around(convert_freq_to_resolution(ax.get_xticks(),
↪  px_to_A), 2))
negligible_threshold_check = np.where((FSC_vals < 0.5) == True)[0]
if len(negligible_threshold_check) > 0:
    resolution = resolutions[negligible_threshold_check[0] - 1]
    ax.axvline(x=frequencies[negligible_threshold_check[0] - 1],
    ↪  color='red')
    print("Resolution:", resolution)
# ax.set_xlabel('Frequency (1/px)')
ax.set_xlabel('Resolution (' + r'$\AA$' + ')')
ax.set_ylabel('FSC')
plt.show()
```

B.2 Code for generating FRC plot

The code used to produce the FRC plots uses the miplib library [7]. The miplib library does not have a Numpy friendly version of FSC and so in later experiments, FSC plots were generated using direct Numpy implementations, as shown in Appendix B.1. Note that the code in this

Appendix was altered slightly from the original miplib code to change the label of the output FRC plot.

```
#
↪  https://github.com/sakoho81/miplib/blob/10f14c8eccefcc0d90feea8dcea5b36a3801797b/m
# From above

def safe_divide(numerator, denominator):
    with np.errstate(divide="ignore", invalid="ignore"):
        result = numerator / denominator
        result[result == np.inf] = 0.0
        return np.nan_to_num(result)

#
↪  https://github.com/sakoho81/miplib/blob/10f14c8eccefcc0d90feea8dcea5b36a3801797b/m
# Slightly altered from code above to fix labels and similar

def __make_frc_subplot(ax, frc, title):
    """
    Creates a plot of the FRC curves in the curve_list. Single or multiple
↪  vurves can
    be plotted.
    """

    # # Font setting
    # font0 = FontProperties()
    # font1 = font0.copy()
    # font1.set_size('medium')
    # font = font1.copy()
    # font.set_family('sans')
    # rc('text', usetex=True)

    # Enable grid
    gridLineWidth = 0.2
    # ax.yaxis.grid(True, linewidth=gridLineWidth, linestyle='-',
    ↪  color='0.05')
```

```
# Axis labelling
xlabel = 'Frequency (1/px)'
ylabel = 'Correlation'
# ax.set_xlabel(xlabel, fontsize=12, position=(0.5, -0.2))
# ax.set_ylabel(ylabel, fontsize=12, position=(0.5, 0.5))
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)

ax.set_ylim([0, 1.2])

# Title
ax.set_title(title)

# Plot calculated FRC values as xy scatter.
y = frc.correlation["correlation"]
x = frc.correlation["frequency"]
x_axis = safe_divide(x, 2 * frc.resolution["spacing"])

ax.plot(x_axis, y, '^', markersize=6, color='#b5b5b3',
        label='FRC')

# Plot polynomial fit as a line plot over the FRC scatter
y = frc.correlation["curve-fit"]
ax.plot(x_axis, y, linewidth=3, color='#61a2da',
        label='Least-squares fit')

# Plot the resolution threshold curve
y = frc.resolution["threshold"]
res_crit = frc.resolution["criterion"]
if res_crit == 'one-bit':
    label = 'One-bit curve'
elif res_crit == 'half-bit':
    label = 'Half-bit curve'
elif res_crit == 'fixed':
    label = 'y = %f' % y[0]
else:
```

```
        label = "Threshold"

    if x[-1] < 1.0:
        x = np.append(x, 1.0)
        y = np.append(y, y[-1])

    x_axis = safe_divide(x, 2 * frc.resolution["spacing"])

    ax.plot(x_axis, y, color='#d77186',
            label=label, lw=2, linestyle='--')

    # Plot resolution point
    y0 = frc.resolution["resolution-point"][0]
    x0 = frc.resolution["resolution-point"][1] / (2 *
    ↪  frc.resolution["spacing"])

    ax.plot(x0, y0, 'ro', markersize=8, label='Resolution point',
    ↪  color='#D75725')

    verts = [(x0, 0), (x0, y0)]
    xs, ys = list(zip(*verts))

    ax.plot(xs, ys, 'x--', lw=3, color='#D75725', ms=10)
    # ax.text(x0, y0 + 0.10, 'RESOL-FREQ', fontsize=12)

    resolution = "The resolution is {} px.".format(
        frc.resolution["resolution"])
    ax.text(0.5, -0.3, resolution, ha="center", fontsize=12)

args_list = ("None \
--bin-delta=1 \
--frc-curve-fit-type=smooth-spline \
--resolution-threshold-criterion=fixed \
--frc-mode=two-image \
--resolution-threshold-value=0.5 \
--resolution-snr-value=1").split()
```

```
args = options.get_frc_script_options(args_list)

frc_results = FourierCorrelationDataCollection()

original_image = img
pred_image = final_siren_output

# Voxels for T20S is 0.98
# Spacing might indicate scale of pixels to um...
# spacing = 0.98
spacing = 1
frc_results[0] =
↪  frc.calculate_two_image_frc(Image(images=original_image.squeeze(),
↪  spacing=[spacing, spacing]), Image(images=pred_image.squeeze(),
↪  spacing=[spacing, spacing]), args)

# Plotting
plotter = frcplots.FourierDataPlotter(frc_results)
angle = 0
# plotter.plot_one(angle=angle)

plt.figure(figsize=(5, 4))
ax = plt.subplot(111)

__make_frc_subplot(ax, plotter.data[int(angle)], "FRC at angle %s" %
↪  str(angle))

plt.show()

# Use this code to change the label
#
↪  https://github.com/sakoho81/miplib/blob/10f14c8eccefcc0d90feea8dcea5b36a3801797b/m
```

Appendix C

Code for Defining, Initializing, and Training SIREN Models

The code in this appendix is based on the Colab notebook provided by the authors of the SIREN paper, V. Sitzmann et al [9]. The notebook can be found here: `https://colab.research.google.com/github/vsitzmann/siren/blob/master/explore_siren.ipynb`.

```
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import numpy as np

class SineLayer(nn.Module):
    # See paper sec. 3.2, final paragraph, and supplement Sec. 1.5 for
    ↪  discussion of omega_0.

    # If is_first=True, omega_0 is a frequency factor which simply
    ↪  multiplies the activations before the
    # nonlinearity. Different signals may require different omega_0 in the
    ↪  first layer - this is a
    # hyperparameter.

    # If is_first=False, then the weights will be divided by omega_0 so as
    ↪  to keep the magnitude of
```

```
    # activations constant, but boost gradients to the weight matrix (see
    ↪ supplement Sec. 1.5)

    def __init__(self, in_features, out_features, bias=True,
                 is_first=False, omega_0=30):
        super().__init__()
        self.omega_0 = omega_0
        self.is_first = is_first

        self.in_features = in_features
        self.linear = nn.Linear(in_features, out_features, bias=bias)

        self.init_weights()

    def init_weights(self):
        with torch.no_grad():
            if self.is_first:
                self.linear.weight.uniform_(-1 / self.in_features,
                                            1 / self.in_features)
            else:
                self.linear.weight.uniform_(-np.sqrt(6 / self.in_features) /
                ↪ self.omega_0,
                                            np.sqrt(6 / self.in_features) /
                                            ↪ self.omega_0)

    def forward(self, input):
        return torch.sin(self.omega_0 * self.linear(input))

    def forward_with_intermediate(self, input):
        # For visualization of activation distributions
        intermediate = self.omega_0 * self.linear(input)
        return torch.sin(intermediate), intermediate

class Siren(nn.Module):
    def __init__(self, in_features, hidden_features, hidden_layers,
    ↪ out_features, outermost_linear=False,
```

```
                first_omega_0=30, hidden_omega_0=30.):
        super().__init__()

        self.net = []
        self.net.append(SineLayer(in_features, hidden_features,
                                  is_first=True, omega_0=first_omega_0))

        for i in range(hidden_layers):
            self.net.append(SineLayer(hidden_features, hidden_features,
                                      is_first=False,
                                      ↪ omega_0=hidden_omega_0))

        if outermost_linear:
            final_linear = nn.Linear(hidden_features, out_features)

            with torch.no_grad():
                final_linear.weight.uniform_(-np.sqrt(6 / hidden_features) /
                ↪ hidden_omega_0,
                                             np.sqrt(6 / hidden_features) /
                                             ↪ hidden_omega_0)

            self.net.append(final_linear)
        else:
            self.net.append(SineLayer(hidden_features, out_features,
                                      is_first=False,
                                      ↪ omega_0=hidden_omega_0))

        self.net = nn.Sequential(*self.net)

    def forward(self, coords):
        coords = coords.clone().detach().requires_grad_(True) # allows to
        ↪ take derivative w.r.t. input
        output = self.net(coords)
        return output, coords

    def forward_with_activations(self, coords, retain_grad=False):
```

```
        '''Returns not only model output, but also intermediate
        ↪  activations.
        Only used for visualizing activations later!'''
        activations = OrderedDict()

        activation_count = 0
        x = coords.clone().detach().requires_grad_(True)
        activations['input'] = x
        for i, layer in enumerate(self.net):
            if isinstance(layer, SineLayer):
                x, intermed = layer.forward_with_intermediate(x)

                if retain_grad:
                    x.retain_grad()
                    intermed.retain_grad()

                activations['_'.join((str(layer.__class__), "%d" %
                ↪  activation_count))] = intermed
                activation_count += 1
            else:
                x = layer(x)

                if retain_grad:
                    x.retain_grad()

            activations['_'.join((str(layer.__class__), "%d" %
            ↪  activation_count))] = x
            activation_count += 1

        return activations

# Define hyperparameters here
siren_model = Siren(in_features=3, out_features=1, hidden_features=512,
                    hidden_layers=7, outermost_linear=True, first_omega_0=30,
                    ↪  hidden_omega_0=30)
siren_model.cuda()
```

```
loss_vals = []

total_steps = 250
steps_til_summary = 10

optim = torch.optim.Adam(lr=1e-6, params=siren_model.parameters())

# model_input, ground_truth = next(iter(dataloader))
# model_input, ground_truth = model_input.cuda(), ground_truth.cuda()

for step in range(total_steps):

    epoch_loss = 0.0
    for batch, data in enumerate(dataloader):
        model_input, ground_truth = data
        model_input, ground_truth = model_input.cuda(), ground_truth.cuda()

        optim.zero_grad()

        model_output, coords = siren_model(model_input)
        loss = ((model_output - ground_truth)**2).mean()
        loss.backward()
        optim.step()

        epoch_loss += loss.item()

    epoch_loss /= len(dataloader)
    print("Epoch: %d, Loss %0.9f" % (step + 1, epoch_loss))
    loss_vals.append(epoch_loss)
    epoch_loss = 0.0
```

Appendix D

Code for Generating Coordinates and Combining Model Outputs to Representations using SIREN Models

The code in this appendix is based on the Colab notebook provided by the authors of the SIREN paper, V. Sitzmann et al [9]. The notebook can be found here: `https://colab.research.google.com/github/vsitzmann/siren/blob/master/explore_siren.ipynb`.

D.1 Code for generating training coordinates and densities using SIREN models

. The variable structure is assumed to contain a Numpy array with the protein structure to use for training.

```
import numpy as np
import skimage
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import gc

def get_mgrid(sidelen, dim=2):
    '''Generates a flattened grid of (x,y,...) coordinates in a range of
    ↪  -1 to 1.
```

```
    sidelen: int
    dim: int'''
    tensors = tuple(dim * [torch.linspace(-1, 1, steps=sidelen)])
    mgrid = torch.stack(torch.meshgrid(*tensors), dim=-1)
    mgrid = mgrid.reshape(-1, dim)
    return mgrid

class ImageFitting(Dataset):
    def __init__(self, img, sidelength):
        super().__init__()
        img = torch.from_numpy(img)
        self.pixels = img.reshape(-1, 1)
        self.coords = get_mgrid(sidelength, 3)
        # Second argument indicates number of dimensions, i.e.: set to 2
        ↪  for 2D protein slices


    def __len__(self):
        return len(self.pixels)

    def __getitem__(self, idx):
        return self.coords[idx], self.pixels[idx]

# Clears memory
#################
torch.cuda.empty_cache()
gc.collect()
#################

particle_slice = ImageFitting(structure, output_dims)
dataloader = DataLoader(particle_slice, batch_size=256, pin_memory=True,
↪  num_workers=0)

siren_model = Siren(in_features=3, out_features=1, hidden_features=512,
                    hidden_layers=7, outermost_linear=True, first_omega_0=30,
                    ↪  hidden_omega_0=30)
```

```
siren_model.cuda()
```

D.2 Code for generating representations of desired shape using SIREN Models

The variable desired_shape is assumed to contain an integer indicating the desired shape of the output model representation. For example, desired_shape = 300 will output a representation of shape (300, 300, 300) for the 3D case. siren_model is assumed to contain the trained SIREN model.

```
import numpy as np
import skimage
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset

def get_mgrid(sidelen, dim=2):
    '''Generates a flattened grid of (x,y,...) coordinates in a range of
    ↪  -1 to 1.
    sidelen: int
    dim: int'''
    tensors = tuple(dim * [torch.linspace(-1, 1, steps=sidelen)])
    mgrid = torch.stack(torch.meshgrid(*tensors), dim=-1)
    mgrid = mgrid.reshape(-1, dim)
    return mgrid

input_coords = get_mgrid(desired_shape, 3)
# Second argument indicates number of dimensions, i.e.: set to 2 for 2D
↪  protein slices
dataloader = DataLoader(input_coords, batch_size=1024, pin_memory=True,
↪  num_workers=0)

all_coords = []
all_densities = []
```

```
num_processed = 0
num_input = 0
for batch, model_input in enumerate(dataloader):
    clear_output()
    print(batch, '/', len(dataloader))

    model_input = model_input.cuda()

    model_output, coords = siren_model(model_input)
    all_coords.append(coords.cpu().detach().numpy().squeeze())
    # all_coords.append(model_input.cpu().detach().numpy().squeeze())
    all_densities.append(model_output.cpu().detach().numpy().squeeze())
    num_processed += len(model_output.cpu().detach().numpy().squeeze())
    num_input += len(model_input)

all_coords = np.concatenate(all_coords, axis=0)
all_densities = np.concatenate(all_densities, axis=0)

all_coords_transformed = np.stack(np.meshgrid(*(3 * [np.linspace(0,
↪  desired_shape - 1, desired_shape)])), axis=-1).reshape(-1,
↪  3).astype(int)

all_coords_z = all_coords_transformed[:, 0]
all_coords_y = all_coords_transformed[:, 1]
all_coords_x = all_coords_transformed[:, 2]

final_siren_output = np.zeros((desired_shape, desired_shape, desired_shape))
final_siren_output[all_coords_y, all_coords_z, all_coords_x] = all_densities
```

Appendix E

Code for Defining, Initializing, and Training Fourier Feature Networks

The code in this appendix is based on the Colab notebook provided by the authors of the Fourier feature networks paper, M. Tancik et al [12]. The notebook can be found here: `https://github.com/tancik/fourier-feature-networks/blob/master/Demo.ipynb`. Note that in the notebook from the authors uses the machine learning library JAX. The code presented here is a custom port of the JAX code in the notebook to PyTorch.

E.1 Code for generating training coordinates and densities using Fourier feature networks

The variable test_slice is assumed to contain a Numpy array with the protein structure to use for training. In this code, only the 2D case is considered.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
from tqdm.notebook import tqdm as tqdm

random_seed = 0
np.random.seed(random_seed)
torch.manual_seed(0)
```

```
c = [test_slice.shape[0]//2, test_slice.shape[1]//2]
r = 256 // 2
img = test_slice[c[0]-r:c[0]+r, c[1]-r:c[1]+r]

plt.imshow(img)
plt.show()

# Create input pixel coordinates in the unit square
coords = np.linspace(0, 1, img.shape[0], endpoint=False)
x_test = np.stack(np.meshgrid(coords, coords), -1)
test_data = [[x_test, img]]
train_data = [[x_test[::2,::2], img[::2,::2]]]

# Fourier feature mapping
def input_mapping(x, B):
  if B is None:
    return x
  else:
    x_proj = (2.*np.pi*x) @ B.T
    return np.concatenate([np.sin(x_proj), np.cos(x_proj)], axis=-1)

# PyTorch network definition
class fourier_feature_network(nn.Module):
    def __init__(self, num_layers, num_channels, in_channels,
    ↪  out_channels=3):
        # input_size is tensor.numel() or numpy.size

        super(fourier_feature_network, self).__init__()

        self.initial_dense_layer = nn.Linear(in_channels, num_channels)
        hidden_dense_layers = []
        for layer_idx in range(num_layers - 2):
            curr_layer = nn.Linear(num_channels, num_channels)
            hidden_dense_layers.append(curr_layer)

        self.hidden_dense_layers = nn.ModuleList(hidden_dense_layers)
```

```
        self.final_dense_layer = nn.Linear(num_channels, out_channels)

        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.initial_dense_layer(x)
        for layer_idx, curr_dense_layer in
        ↪  enumerate(self.hidden_dense_layers):
            x = curr_dense_layer(x)
            x = self.relu(x)

        x = self.final_dense_layer(x)
        x = self.sigmoid(x)

        return x

class fourier_feature_dataset(torch.utils.data.Dataset):
    def __init__(self, data, B):
        self.x = [curr_data[0] for curr_data in train_data]
        self.y = [curr_data[1] for curr_data in train_data]
        self.B = B

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        fourier_features = torch.from_numpy(input_mapping(self.x[idx],
        ↪  self.B))
        out_image = torch.from_numpy(self.y[idx])

        return fourier_features, out_image

def custom_loss(x, y):
    return 0.5 * nn.MSELoss()(x.float(), y.float())
```

```
def calculate_PSNR(loss):
    # jit(lambda params, x, y: -10 * np.log10(2.*model_loss(params, x,
    ↪  y)))
    return -10 * np.log10(loss.detach().cpu().numpy() * 2)

def train_model(network_size, learning_rate, iters, B, train_data,
↪  test_data):

    train_dataset = fourier_feature_dataset(train_data, B)
    test_dataset = fourier_feature_dataset(test_data, B)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=1,
    ↪  shuffle=False)
    test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1,
    ↪  shuffle=False)

    fourier_model = fourier_feature_network(network_size[0],
    ↪  network_size[1], train_dataset[0][0].shape[-1], 1)
    fourier_model = fourier_model.float()

    # GPU handling
    #
    ↪  https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    if torch.cuda.device_count() > 1:
        fourier_model = nn.DataParallel(fourier_model)
    fourier_model = fourier_model.to(device).cuda()

    optimizer = optim.Adam(fourier_model.parameters(), lr=learning_rate)

    x_vals = []
    train_psnrs = []
    test_psnrs = []
    prediction_images = []

    # print(model.module1.fc1.weight.type())
```

```
for epoch in range(iters):
    for idx, data in enumerate(train_loader):
        fourier_features, labels = data

        optimizer.zero_grad()

        fourier_features = fourier_features.float()
        fourier_features = fourier_features.to(device).cuda() # For gpu
        labels = labels.to(device).cuda()

        model_outputs = fourier_model(fourier_features)
        loss = custom_loss(model_outputs, labels)
        loss.backward()
        optimizer.step()

    # if epoch % 25 == 0:

    clear_output(wait=True)
    print(str(epoch) + '/' + str(iters))

    train_psnrs.append(calculate_PSNR(loss))

    test_loss = 0
    for test_idx, test_data in enumerate(test_loader):
        test_features, test_labels = test_data

        test_features = test_features.float()
        test_features = test_features.to(device).cuda()
        test_labels = test_labels.to(device).cuda()

        test_loss += custom_loss(fourier_model(test_features),
        ↪  test_labels)
    test_loss /= len(test_loader)

    test_psnrs.append(calculate_PSNR(test_loss))
```

```
            #
            ↪  prediction_images.append(fourier_model(test_dataset[0][0].float()))
            x_vals.append(epoch)

        prediction_images.append(fourier_model(test_dataset[0][0].float()))

        return {
            'state': fourier_model.state_dict(),
            'train_psnrs': train_psnrs,
            'test_psnrs': test_psnrs,
            'pred_imgs': prediction_images,
            'xs': x_vals
        }

network_size = (4, 256)
learning_rate = 1e-4
iters = 2000

mapping_size = 256

B_dict = {}
# Standard network - no mapping
B_dict['none'] = None
# Basic mapping
B_dict['basic'] = np.eye(2)
# Three different scales of Gaussian Fourier feature mappings
B_gauss = np.random.normal(loc=0, scale=1, size=(mapping_size, 2))
for scale in [1., 10., 100.]:
  B_dict[f'gauss_{scale}'] = B_gauss * scale

# This should take about 2-3 minutes
outputs = {}
for k in tqdm(B_dict):
  outputs[k] = train_model(network_size, learning_rate, iters, B_dict[k],
  ↪  train_data, test_data)
```

```
# Show final network outputs

plt.figure(figsize=(24,4))
N = len(outputs)
for i, k in enumerate(outputs):
    plt.subplot(1,N+1,i+1)
    plt.imshow(outputs[k]['pred_imgs'][-1].cpu().detach().numpy())
    plt.title(k)
plt.subplot(1,N+1,N+1)
plt.imshow(img)
plt.title('GT')
plt.show()

# Plot train/test error curves

plt.figure(figsize=(16,6))

plt.subplot(121)
for i, k in enumerate(outputs):
    plt.plot(outputs[k]['xs'], outputs[k]['train_psnrs'], label=k)
plt.title('Train error')
plt.ylabel('PSNR')
plt.xlabel('Training iter')
plt.legend()

plt.subplot(122)
for i, k in enumerate(outputs):
    plt.plot(outputs[k]['xs'], outputs[k]['test_psnrs'], label=k)
plt.title('Test error')
plt.ylabel('PSNR')
plt.xlabel('Training iter')
plt.legend()

plt.show()
```

Appendix F

Code for Generating Amplitude Plot

The variable original_image is assumed to contain a Numpy array with a protein reconstruction. Either the original structure or the reconstructed structure can be used. In this code, only the 2D case is considered.

```
import numpy as np
import matplotlib.pyplot as plt

# Arguments
shell_size = 2
epsilon = 1e-7

# Calculations
image_fft = np.fft.fftshift(np.fft.fft2(image.squeeze()))
raw_amplitudes = np.abs(image_fft)

n = image.squeeze().shape[0]
freq_interval = shell_size / n
freq_intervals = np.array([freq_idx * freq_interval for freq_idx in
↪  range(1, int(0.5 / freq_interval) + 1)])
amplitudes = np.zeros_like(freq_intervals)
num_amplitudes = np.zeros_like(freq_intervals)

center_point =
↪  scipy.ndimage.measurements.center_of_mass(np.ones_like(image.squeeze()))
grid_y, grid_x = np.mgrid[0:output_dims, 0:output_dims]
```

```
grid_y = (grid_y - center_point[0]) / output_dims
grid_x = (grid_x - center_point[1]) / output_dims
distances = np.hypot(grid_x, grid_y)
distance_vals = np.unique(distances)
distance_vals = distance_vals[distance_vals < 0.5]
for spatial_freq in distance_vals:
    freq_idx = np.where(freq_intervals == freq_intervals[spatial_freq <=
    ↪  freq_intervals].min())[0][0]
    curr_amplitudes = raw_amplitudes[distances == spatial_freq]
    amplitudes[freq_idx] += np.sum(curr_amplitudes)
    num_amplitudes[freq_idx] += len(curr_amplitudes)

amplitudes /= num_amplitudes

# Plotting (example of there being neglibile amplitudes)
plt.plot(freq_intervals, amplitudes)
negligible_threshold_check = np.where((amplitudes < 0.25) == True)[0]
if len(negligible_threshold_check) > 0:
    plt.axvline(x=freq_intervals[negligible_threshold_check[0]],
    ↪  color='red')
plt.xlabel('Frequency (1/px)')
plt.ylabel('Amplitude')
plt.show()
```