©Copyright 2019
Todor Kirilov Avramov

Deep Learning for Validating Resolution and Detecting Secondary Structure Elements of Proteins in 3D Cryo-Electron Microscopy Images

Todor Kirilov Avramov

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2019

Committee:

Dong Si

Wooyoung Kim

Munehiro Fukuda

Program Authorized to Offer Degree:
Computing and Software Systems

University of Washington

Abstract

Deep Learning for Validating Resolution and Detecting Secondary Structure Elements of Proteins in 3D Cryo-Electron Microscopy Images

Todor Kirilov Avramov

Chair of the Supervisory Committee:
Professor Dong Si
Computing and Software Systems

Cryo-electron microscopy (cryo-EM) is becoming the imaging method of choice for determining protein structures. Many atomic structures have been resolved based on an exponentially growing number of published three-dimensional (3D) cryo-EM density maps. However, the resolution value claimed for the reconstructed 3D density map has been the topic of scientific debate for many years. The Fourier Shell Correlation (FSC) is the currently accepted cryo-EM resolution measure, but it can be subjective, manipulated, and has its own limitations. This thesis proposes supervised deep learning methods to extract representative 3D features at high, medium and low resolutions from simulated protein density maps and build classification models that objectively validate resolutions of experimental 3D cryo-EM maps. Specifically, classification models based on dense artificial neural network (DNN) and 3D convolutional neural network (3D CNN) architectures are presented. The trained models can classify a given 3D cryo-EM density map into one of three resolution levels: high, medium, low. The DNN model achieved 92.73% accuracy and the 3D CNN model achieved 99.75% accuracy on simulated test maps. When tested on simulated maps at gradually varying resolutions, the two models identified the resolution boundaries between high, medium and low resolutions. The deep learning models clustered maps lower than 4-4.5Å in the high resolution class, maps between 5.0-8.5Å in the medium resolution, and maps at resolutions

>=9.0Å were classified as low resolution. Applying the DNN and 3D CNN models to thirty experimental cryo-EM maps achieved an agreement of 60.0% and 56.7%, respectively, with the author published resolution value of the density maps. These results suggest that deep learning can potentially improve the resolution evaluation process of cryo-EM maps but further work is needed to account for local variability of resolution as suggested by recent studies.

Detection of protein secondary structure elements (SSEs) to aid in the creation of accurate atomic models especially from medium resolution cryo-EM maps is another area of current research. Medium resolution experimental cryo-EM images lack detail and contain noise, and thus require additional computational and visualization analyses to fully determine protein structures. Most previous researches proposed prescriptive image-processing and pattern matching algorithms to locate α-helices and β-sheets in cryo-EM maps, but these methods were not fully automated and required subjective selection of parameters. This thesis explores a convolutional neural network model for end-to-end voxelwise segmentation of 3D cryo-EM density images. The 3D segmentation model, adapted from the U-Net architecture, was constructed in TensorFlow and it optimized a multi-class Tversky loss function with Adam optimization algorithm. The proposed 3D U-Net model was trained to segment a cryo-EM map by classifying each voxel as either being part of an α-helix, a β-sheet, a turn/loop, or background. For that purpose, we first introduce and describe a novel method to generate large amounts of labeled cryo-EM maps suitable for training deep learning models for secondary structure segmentation. The model achieved higher per-class and overall precision and recall rates than previous methods when tested on 3597 simulated cryo-EM density maps. The proposed method was also shown to reliably segment experimental cryo-EM maps. Finally, the 3D U-Net segmentation model was compiled into an executable program and integrated as a plug-in in the UCSF Chimera visualization and analysis system.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

Chapter 1: Introduction 1
1.1 Rationale 1
1.2 Specific Aims 3
1.3 Contributions and Thesis Overview 5

Chapter 2: Background and Related Work 9
2.1 Proteins 9
2.2 Cryo-Electron Microscopy Imaging 12
2.3 Secondary Structure Detection in Cryo-EM Protein Density maps 18
2.4 Convolutional Neural Networks 22
2.5 Semantic Segmentation with CNNs 23

Chapter 3: Resolution Validation of Three Dimensional Cryo-Electron Microscopy Density Maps 27
3.1 Methodology 27
3.2 Data Collection and Data Pre-processing 28
3.3 Model Training 29
3.4 Results 32
3.5 Discussion 39

Chapter 4: Detecting Secondary Structures in Protein Cryo-EM Density Maps . . 43
4.1 Data Collection and Preparation 43
4.2 The Proposed Deep Model 60
4.3 Results 66

4.4 Discussion 83

Chapter 5: Software Technology Package 84
5.1 SS Predictor 84
5.2 Discussion 88

Chapter 6: Conclusion and Future Work 90

Bibliography 94

Appendix A: Micro-Averaged Precision and Recall 102
A.1 Derivation and Equality Proof 102

Appendix B: Optimizing Tversky Loss Parameters 104
B.1 Confusion Matrices and Evaluation Metrics for 100 Test Maps 104

Appendix C: Source Code 106
C.1 Data Collection and Preparation 106
C.2 The Proposed Deep Model 115
C.3 SW Tool 124

LIST OF FIGURES

Figure Number Page

2.1 Amino Acid Structure and Peptide Bond Formation 11
2.2 Protein Secondary Structure: α-helix and β-sheet 13
2.3 Approximate Resolution Assessment of cryo-EM maps 17
2.4 Semantic Segmentation Network Architectures 26
3.1 Resolution Validation Experimental Design 28
3.2 DNN Architecture 31
3.3 3D CNN Architecture 32
3.4 DNN Validation Accuracy and Loss 33
3.5 3D CNN Validation Accuracy and Loss 34
4.1 Part of a PDB file 46
4.2 Splitting PDB files based on secondary structure 48
4.3 Centering PDB structures around origin 52
4.4 Centering PDB structures around origin 53
4.5 Simulated 3D Electron Density Maps from Segmented PDB Files 55
4.6 Simulated 3D Electron Density Maps at Different Resolutions 56
4.7 Structure of hdf5 files 59
4.8 3D U-Net Architecture 61
4.9 Effects of Loss Function Parameters on Validation Loss 69
4.10 Effects of Loss Function Parameters on Validation Accuracy 69
4.11 Evaluation Metrics at Different Tversky Loss Regularization Parameters . . 71
4.12 Segmentation of Simulated Map (pdb id: 1C3W) 74
4.13 Segmentation of Simulated Map (pdb id: 1BVP) 76
4.14 Segmentation of Simulated Map (pdb id: 1TIM) 78
4.15 Segmentation of Simulated Map (pdb id: 1LP4) 79
4.16 Experimental Map EMD-4391 Segmentation Results 81

5.1 Launching SSE Predictor in Chimera 86
5.2 GUI for the Chimera Plug-in 87
5.3 Automatic Display of Plug-In Results in Chimera Viewer 88

LIST OF TABLES

Table Number | Page

3.1 Map counts in each data subset used for resolution validation experiment. . . 29
3.2 Classification Results on Simulated Maps with Varying Resolution. 36
3.3 Classification Results of the 30 Experimental cryo-EM Density Maps 37
3.4 DNN Confusion Matrix 38
3.5 3D CNN Confusion Matrix 38
3.6 Performance Measures of the Classifiers 39

4.1 Protein density map counts per resolution in train and test data sets for secondary structure segmentation experiment. 60
4.2 1C3W Segmentation Results 74
4.3 1BVP Segmentation Results 76
4.4 1TIM Segmentation Results 77
4.5 1LP4 Segmentation Results 79
4.6 EMD-4391 Segmentation Results 82

B.1 Confusion Matrices and Evaluation Metrics for Models Trained with Different Tversky Loss Parameters 105

ACKNOWLEDGMENTS

I would like to thank my adviser Dr. Dong Si for his support and guidance throughout the duration of this research work. Professor Si was the instructor who first introduced me to the topic of Machine Learning in one of his classes, for which I will be forever grateful. His mentoring has helped me mature as a student, scientist and person. I have published two conference papers in collaboration with Dr. Si and I sincerely appreciate the impact he has had on my professional and personal development.

I wish to express sincere appreciation to University of Washington Bothell, where I have had the access to the computing resources that enabled me to do this research project, and to my employer Philips Healthcare for the generous financial assistance program.

I would also like to thank the many great instructors who taught the classes I took as part of my Master's Degree. Thanks go to my fellow graduate students, Albert Ng and Spencer Moritz, for sharing ideas and snippets of code as well as their review inputs and assistance with technical issues. The stress of school and research was alleviated through the fun conversations and jokes with them.

Foremost, I am thankful to my wife and my parents for their love and unquestioning support throughout this process.

DEDICATION

to my dear wife, Monise,

and my newborn daughter, Martina.

Chapter 1

INTRODUCTION

1.1 Rationale

Proteins are indisputably the most diverse and most intensively studied macromolecules in biology. Proteins perform diverse functions within all living organisms. The functional diversity of proteins arises from their three-dimensional (3D) structures. All proteins fold up in complex and unique 3D shapes, and understanding the folding mechanism of proteins is one of biology's grand challenges. The specific shape of a protein dictates its biochemical function, for example by allowing certain molecules to form chemical bonds at specific regions while physically preventing other molecules to fit in these binding sites. Therefore, knowledge about a protein's 3D structure is a prerequisite for understanding its biological function, and for the development of therapeutic protein drugs as well as drugs that inhibit harmful proteins (such as the proteins on virus surfaces). There are three research techniques that have been developed for aiding the studying of protein structures: X-ray crystallography, nuclear magnetic resondance (NMR) and cryo-electron microscopy (cryo-EM). Structural biologists have traditionally relied on X-ray crystallography to determine the 3D atomic structures of proteins. In recent years, cryo-electron microscopy has been proposed as an alternative method for determining structures of proteins, especially large proteins and other macromolecules that do not crystallize easily. Some large molecular complexes such as ribosomes and viruses have been successfully resolved to near-atomic resolutions through cryo-EM [6, 39, 51]. But structures of smaller proteins could not be fully resolved with the same level of detail for a long time. Since the cryo-EM technique was first developed in the 1970s until 2015, cryo-EM protein structures with high resolution <4Å (Angstrom, $1\text{Å} = 10^{-10}$ meter) could not be achieved. Therefore, structures of proteins could not be fully determined only

from cryo-EM images. Majority of published electron microscopy structures at lower resolutions require further computational and visualization analyses to aid in their interpretation. One approach to derive atomic structures from medium resolution (5-10Å) cryo-EM density images is to identify and interpret commonly found features in protein maps. Two common secondary structure elements (SSEs), α-helix and β-sheet, typically make more than half of protein structures so identifying them in electron-density maps can potentially reveal a large part of the protein and lead to building an atomic model. Therefore, a major area of focus in cryo-EM research is the development of accurate and automatic methods for identifying SSEs in medium resolution protein density maps.

Another limitation of the cryo-EM technique is the concept of resolution. Resolution in electron microscopy is not well defined. The current methods for specifying resolution of cryo-EM structures are subjective, not consistently applied and have other limitations. Authors of cryo-EM maps can submit structures in publicly available databases and claim a resolution value based on their own method of choice. There is not a gold-standard for resolution determination and reporting. With the increasing popularity of cryo-electron microscopy, there is currently a need to objectively quantify and validate the resolution of cryo-EM structures in order to properly interpret results.

Deep learning is an attractive software technology that is successfully being applied in many disciplines. While the science of deep learning is not new, it has gained fresh momentum from the availability of vast amounts of data and the capability to perform complex calculations fast. Today, deep learning excels in image classification and pattern recognition tasks. Inspired by the network of neurons in the nervous system and by how the human brain learns, neural networks are the most popular deep learning models. Neural networks are a type of fully trainable models that can capture nonlinear relationships between inputs and outputs. When trained with supervised learning algorithms, neural networks infer a mapping function based on many examples of input-output pairs; the same way a child learns to recognize and name an object after repeatedly being show examples of that object. In the training process, neural networks are designed to automatically capture information

from the input and associate it with a particular output. These characteristics of neural networks make them ideal candidates to be used as a means to addressing the current focus areas in cryo-EM research. Incorporating the latest ideas and the most recent deep learning advances for solving the secondary structure segmentation and resolution classification problems provide promising new avenues for advancing the current body of knowledge.

1.2 Specific Aims

This research aims to explore deep learning techniques for addressing the aforementioned challenges in cryo-EM research. Specifically, we discuss a deep learning framework for automatically segmenting medium resolution cryo-EM maps based on secondary structure elements, and the integration of that neural network in a standalone software tool. Separate deep learning networks for objectively quantifying and verifying resolution of cryo-EM structures are also proposed. The results of this research are presented through three specific aims:

1.2.1 Specific Aim 1

Use machine learning techniques to objectively quantify and validate the resolution of 3D cryo-EM density maps. Through this aim, we are specifically seeking to explore methods and neural network architectures that can objectively classify resolutions of experimental electron density maps. Using supervised learning techniques and electron density maps with known resolutions, it is hypothesized that neural network models can independently learn to distinguish cryo-EM structures at different resolutions. Experimental density maps can then be classified using the trained models and results compared with author claimed resolution values from public repositories. This thesis is an initial attempt to investigate the feasibility of using deep learning for resolution validation of cryo-EM density maps. The goals are to examine the implicatioins and limitations of such approach and to identify direction for future work. Results for this aim have already been published in the Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics [7].

1.2.2 Specific Aim 2

Design and train a neural network for segmentation of protein secondary structure elements in three-dimensional, medium resolution cryo-electron density maps. Evidence from the literature suggests that it is possible to identify two types of regularly repeating structural elements, α-helices and β-sheets, in protein cryo-EM structures with resolutions between 5Å and 10Å [8, 10, 42, 86]. It is therefore hypothesized that a deep neural network can be trained to more accurately detect protein secondary structures in cryo-EM images. In particular, this study aims to propose a specialized type of neural network that segments each individual voxel in a cryo-EM map and classifies it as either α-helix, β-sheet or other. For that purpose, we introduce a method to generate large sets of simulated 3D electron density maps labeled at each voxel that can be used for training deep learning semantic segmentation models. After training a model on simulated maps, this work evaluates and discusses predictive capabilities of the proposed segmentation neural network on experimental maps. The goal is to provide a fully automated solution for simultaneously detecting α-helices and β-sheets that has better accuracy than previously published methods.

1.2.3 Specific Aim 3

Integrate the secondary structure segmentation model in a user-friendly software technology package available to the scientific community. Deploying a deep learning solution is unique because just publishing the method or the network architecture is not enough. What is important for the deployment of deep learning solution is the final model with optimized parameters. Very often, fully trained deep learning solutions do not get integrated in easily usable tools. Hence, the user would need to obtain all the training data and repeat the entire time-consuming training procedure before the solution can be used. The aim of this work is to eliminate the need for end-users to repeat the training of the segmentation neural network model. Instead, the goal is to compile an executable

program with a simple interface that users can download and use to segment experimental cryo-electron density maps. This goal involves integrating the final optimized model and data processing steps from Aim 2 with an input pipeline from a popular software program for visualizing 3D protein structures.

1.3 Contributions and Thesis Overview

1.3.1 Contributions

This work makes a number of contributions regarding the two main challenges in cryo-EM research identified above. These contributions are summarized here and discussed in full detail in the following chapters:

- To our knowledge, this study of resolution estimation and validation of cryo-EM maps is the first to explore deep learning methods. One long-standing problem for the correct interpretation and analysis of protein structures from cryo-EM images is the subjectivity and inconsistent use of currently used methods for estimating resolution of reconstructed structures. This study applies the latest deep learning methodologies and proposes two different neural network models for consistent and objective resolution validation of cryo-EM density maps. Using simulated density maps, evidence is provided that the two deep learning models can independently learn resolution patterns from protein structures. This research also shows that the two deep learning models classified maps with similar resolution values as belonging to the same class. Applying these methods to experimental cryo-EM maps had low agreement with the resolution values claimed by the map deposition authors, suggesting that either 1) our approach to train on simulated maps was not adequate enough or 2) a possibility that claimed resolutions values are inaccurate. This study suggests future improvements that can further shift research focus in the direction of using deep learning methods for resolution validation of cryo-electron microscopy images.

- The method proposed in this theses for identification of secondary structure elements in protein electron density maps departs from traditional heuristic methods which have existed for a long time. Most previous methods are based on prescriptive image-processing and pattern matching algorithms and have several shortcomings: 1) they require careful selection of parameters which work for some but not all structures; 2) most were optimized to identify only alpha-helices or beta-sheets but not both simultaneously; 3) they do not explore all data to assist in detecting SSEs of new samples. This research work explores a special type of convolutional neural network for semantic segmentation that is fully automated and is capable of simultaneously associating each voxel in a cryo-em map with one of three types of SSEs (alpha-helix, beta-sheet, other turn/loops) or background/empty voxel. The proposed network extends the popular U-Net architecture from Ronneberger et al. by replacing all 2D convolutions, deconvolutions and other operations with their 3D equivalents [63]. This work for identifying protein secondary structure elements in medium resolution cryo-electron density maps is not the first solution to use convolutional neural networks but it is the first one to use the U-Net architecture. While other CNN approaches utilize patch-based training and segmentation, the 3D U-Net architecture presented here can segment a full image at the voxel level. We present evidence that our method achieves better results on simulated maps than previous methods. Furthermore, this work shows that the 3D U-Net model achieves good accuracy on an experimental density map. We use a large dataset of 14194 labeled maps to train our model, versus only 30 maps used in previous methods. The creation of such a large set was possible with a novel application of an existing method to simulate segmented electron density maps from existing solved 3D protein models and assign SSE label for each voxel. Regardless of the size of training data, voxels labeled as empty/background are present in disproportionately larger numbers over the other three classes. Since traditional cross entropy loss functions assert equal learning to each voxel, the most prevalent class would dominate the optimization of the model parameters during training. To that end, this research investigates the benefits

of using a Tversky loss function, originally presented for binary segmentation by Salehi et al., but extended for our four-class segmentation problem. Last but not least, this work presents the implementation of the 3D U-Net network in the latest version, as of this writing, of Google's machine learning Tensorflow framework. Using Tensorflow interfaces, this work leverages the latest field advances to build efficient data pipelines to train and evaluate the 3D U-Net deep learning model.

- Finally, this study introduces SS Predictor, a fully automated user friendly program for identifying α-helices, β-sheets and turn/loops in cryo-electron density maps. SS Predictor assigns labels to individual map voxels based on predictions from a pre-trained 3D U-Net model. SS Predictor offers considerable advantages over other methods in that it is easy to set-up and use as it does not require any user defined parameters or special software. The paper demonstrates the deployment of SS Predictor as an extension plug-in in the UCSF Chimera package. The plug-in offers an end-to-end solution for researchers in this field. The initial version of SS Predictor is supported on Windows operating system, but it can be easily released for other configurations.

1.3.2 Thesis Overview

The remainder of this thesis paper is organized as follows. Next chapter presents in-depth background material about the problems this thesis addresses, and highlights related work that provides context for the novelty of the research presented here. Chapters 3 through 5 form the core of the thesis, with each chapter focusing on each one of the specific aims identified above. Chapter 3 presents the deep learning approach and the results for validating resolution of three dimensional cryo-electron microscopy density maps. Chapter 4 is concerned with the detection of secondary structure elements in medium-resolution cryo-EM density maps. we first introduce a novel method for generating large sets of maps labeled at each voxel suitable for training a deep learning model. Then, we describe the data input pipeline for training a 3D convolutional neural network with U-Net architecture and a loss

function based on Tversky index. Finally, the performance of the proposed method is evaluated through the use of simulated and experimental density maps. Chapter 5 describes the deployment of the deep learning method for identification of secondary structure elements into an application, and the integration of that application into a widespread protein visualization software package. Chapter 6 concludes the work by summarizing the contributions and suggesting directions for future work.

Chapter 2
BACKGROUND AND RELATED WORK

This chapter summarizes previous research work and highlights the contributions that relate to the research presented in this thesis. The goal is to place our contributions in the proper context and to provide background and discussion of existing methods for the two areas of our research: cryo-EM resolution estimation and protein secondary structure detection in cryo-EM images. State-of-the-art deep learning methods are also presented. Finally, the chapter is concluded by summarizing how our research builds on this previous work to address the two main problems identified above.

2.1 Proteins

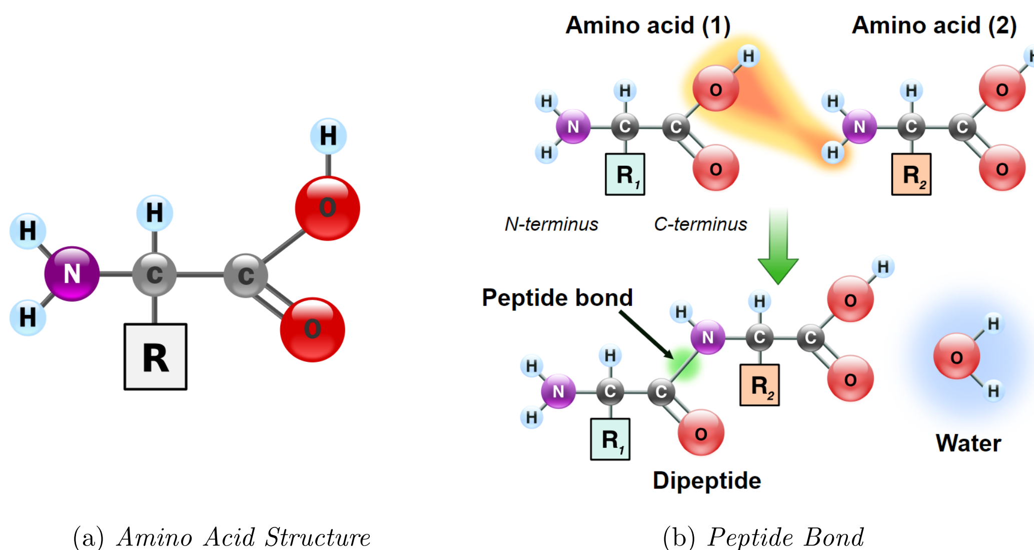
2.1.1 A Brief Overview of Protein Biochemistry

The one common molecule that cells of all living organisms contain, from humans to plants to bacteria, is deoxyribonucleic acid (DNA). DNA is considered the molecule of live because its long chain of nucleotides stores the information that makes organisms unique. But DNA in the cell's nucleus alone does not produce an organism; instead DNA encodes information to produce other molecules to carry out the functions in the cells of organisms. Specifically marked parts of the DNA molecule are copied to another complementary molecule called ribonucleic acid (RNA) in a process called transcription. After transcription and further processing in the nucleus, the RNA molecule exits through the nuclear membrane and enters the cell's cytoplasm where it attaches to one or more ribosome complexes via three of its nucleotides [1, 45]. Each time a new nucleotide triplet is attached, the ribosome selects and collects one of twenty naturally occurring, smaller organic molecules called amino acids. All twenty amino acids have a common structure with a central carbon atom (Cα) attached to

a hydrogen atom, an amine (-NH_2) group, a carboxyl (-COOH) group and a different side chain (-R group) specific to each amino acid (Figure 2.1a). The twenty amino acid molecules are found free floating in the cytoplasm and a chain is produced by linking together different amino acids as the RNA molecule slides through the ribosome three nucleotides at a time. In this process of translation, the order of amino acids is determined by the order of nucleotides in the RNA as the 3 nucleotides bound to the ribosome determine the identity of the selected amino acid. The link between two amino acids is formed via a peptide bond between the carboxyl group (C-terminus) of one amino acid and the amine group (N-terminus) of another (Figure 2.1b). The linked series of carbon, nitrogen and oxygen atoms in the peptide bond form the protein backbone. Once linked in a chain, an individual amino acid is called a residue, and the whole chain of amino acids is also referred to as polypeptide. Ribosomes synthesize polypeptides at a rate of up to 200 amino acids per second yielding proteins with several hundred to several thousand residues in length [1]. Proteins undergo post-translation modification which transforms the long chains of amino acids into mature and functional proteins.

2.1.2 Protein Structure

Proteins contain the same 20 naturally occurring amino acids but they perform many of the different functions in all living organisms, from unicellular bacteria to humans. Despite being made up of the same building blocks, proteins are as diverse and abundant as is life on Earth. Human cells are estimated to contain 1 to 3 billion different proteins while the per cell concentration varies from a few molecules up to 20 million [13, 55]. Some proteins contain less than 100 amino acids, while the largest known protein, titin, has a length of around 30,000 amino acids [58]. Proteins also fold in different structural configurations that facilitate their functions. In fact, what determines the function of a protein is its structure and not so much its chemical composition. There are four structure layers that are commonly used to describe the complex and diverse structures of proteins: primary, secondary, tertiary, quaternary. The primary structure refers to the chemical composition and the number and



(a) *Amino Acid Structure*

(b) *Peptide Bond*

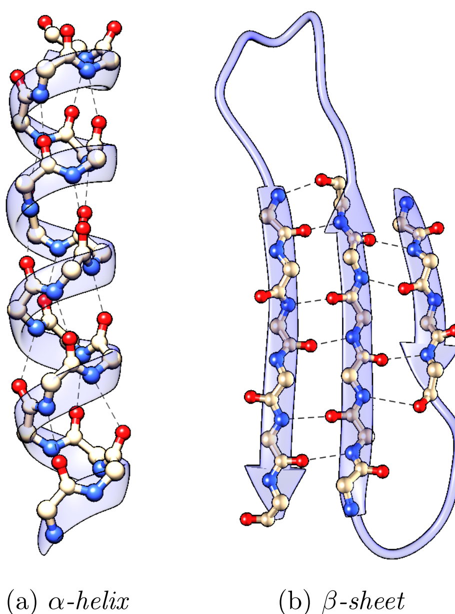
Figure 2.1: ***(a)*** *General structure of an amino acid.* ***(b)*** *Formation of a peptide bond between the carboxyl (-COOH) and amine (-NH_2) groups of two amino acids.*

sequence of amino acids in the polypeptide chain. Primary structure arises from the peptide bonds which are formed during protein biosynthesis. The primary structure of a protein is indirectly determined by the DNA sequence which encodes the particular number and order of amino acids. Secondary structure describes the twisting of the protein chain and the regularly repeating structural elements on the polypeptide backbone. Two main types of secondary structure elements (SSE), α-helix and β-strand or β-sheet, were first discovered by Linus Pauling in 1951 [59]. The α-helix and β-sheet structures are held together by interactions between atoms in neighboring backbone segments. A relatively strong electrostatic attraction called hydrogen bond occurs between hydrogen and oxygen atoms that make up the backbone. In the case of alpha-helices, a hydrogen bond forms between the N-H group and the C=O group in every 3-4 residues and that hydrogen bond holds the protein backbone in a spiral confirmation (Figure 2.2a). Beta-sheets are formed when N-H groups in a strand of several residues form hydrogen bonds with the C=O groups in the backbone of an

adjacent strand (Figure 2.2b). Alpha-helices and beta-sheets are the most prevalent motifs in proteins. According to SCOP (Structural Classification Of Proteins), a widely accepted classification schema, the majority of known protein structures can be broken into four major families: all alpha (α), all beta (β), alpha and beta (α/β), and alpha plus beta ($\alpha+\beta$) [35]. Connected by loops of different length, alpha helices and beta sheets fold into compact three dimensional shapes. This next level of a polypeptide chain structure is referred to as tertiary structure. Tertiary structure is stabilized by interactions between side chains of the same polypeptide, as well as interactions between side chains and the protein surroundings. The twenty different side chains of the amino acid residues have different properties and shapes which influence the tertiary structure of the protein. Some side chains are positively charged, some are negative, some are hydrophobic while others are hydrophilic. Because proteins exist in aqueous environments, proteins fold in ways to hide (bury) hydrophobic residues away from the surrounding water. A positively charged residue and a negatively charged residue may form a strong ionic bond with each other or with another surrounding atom. These ionic bonds and other strong interactions including hydrogen bonds lock and hold the entire tertiary structure of the protein. Quaternary structure, the next-higher and highest level of structure, describes the assembly of two or more polypeptide chains into a protein complex. The multiple chains in a protein complex can be the same or different but together they operate as a single functional unit held by the same interactions that hold the tertiary structure. Quaternary structure defines the number and relative positions of the polypeptide subunits in the complex. There are many possible organizations at each structural level which give rise to the overall structural diversity of proteins.

2.2 Cryo-Electron Microscopy Imaging

Cryo-electron Microscopy is one of the three principle methods used for determining the 3D atomic structures of proteins; the other two being X-ray crystallography and Nuclear Magnetic Resonance (NMR) spectroscopy. The goal of all three imaging techniques is to generate a high-quality and high-resolution, detailed protein macromolecular map that can



(a) *α-helix* (b) *β-sheet*

Figure 2.2: *Representative geometry of alpha-helix and beta-sheet structures. The structures are shown with a ribbon view and the backbone atoms are shown as colored spheres (red for oxygen, blue for nitrogen, and yellow for carbon). Hydrogen bonds between the N-H group and the C=O groups of the backbone are shown with dashed lines. Graphic was generated with UCSF Chimera [61].*

be used in conjunction with other biochemical experiments and computational methods to generate a structural atomic model. X-ray crystallography has been the gold-standard approach for solving protein structures for years [2]. Up to today, most protein structures with near-atomic or atomic resolutions have been determined by X-ray crystallography [83]. But large macromolecules that make up cell organelles such as ribosomes or those showing high degrees of flexibility and heterogeneity such as membrane proteins do not crystallize easily (or do not crystallize at all) and have proven to be a challenge for X-ray crystallography [22, 82]. On the other hand, NMR can provide unique information on dynamics and interactions of proteins, however, atomic structure determination is restricted to small complexes

with molecular weights of approximately <90 kDa (KiloDalton, 1kDa = 1000 grams/mole). Both X-ray crystallography and NMR techniques typically require large amounts of relatively pure sample, on the order of several milligrams.

Electron microscopy imaging relies on refraction and diffraction of radiation waves similar to conventional light microscopic imaging, but instead of using light photons as a source of radiation, higher-energy electron waves are used. The use of electron microscopy for 3D reconstruction of biological structures was first proposed by David DeRosier and Aaron Klug in 1968 [27]. In their paper titled "Reconstruction of three dimensional structures from electron micrographs", DeRosier and Klug obtained a 3D density map of the tail of a common virus using images from electron microscope. As a means to limit the damage of biological samples due to the high energy electron radiation, cryogenic electron microscopy was created when researchers proposed cooling down samples to very low temperatures [31, 71]. Thus, cryo-electron microscopy is a form of transmission electron microscopy in which the studied sample is frozen in a thin layer of a noncrystalline form of solid water, termed amorphous ice, at cryogenic temperatures — generally that of liquid ethane. Over the years, cryo-electron microscopy has evolved to encompass several different experimental methods such as cryo-electron tomography, single-particle analysis and electron crystallography. All of these cryo-EM methods are based on imaging specimens in cryogenic temperatures with transmission electron microscope. Today, cryo-EM methods are extensively used in structural biology to obtain the 3D structural information of macromolecular complexes at subnanometer or nanometer resolution depending on the technique used.

2.2.1 Single-Particle Cryo-Electron Microscopy

Single particle analysis (SPA) is a form of cryo-electron microscopy that allows acquisition of three-dimensional information of large and small macromolecules. In single-particle cryo-electron microscopy for protein structure determination, a beam of electrons is fired directly at a protein sample embedded in vitreous ice at cryogenic temperatures. Aqueous solution of the protein is first flash-frozen with liquid ethane (or liquid nitrogen) to hold the protein

molecules still in random orientations while preventing the formation of ice crystals [54]. The vitrified sample is then placed in the vacuum column of an electron microscope. A beam of electrons accelerated by a voltage potential passes through the sample interacting with the thousands of protein molueclues suspended in random orientations in the thin frozen film [40]. The idea of data collection in SPA is to make use of the fact that the protein molecule occurs in multiple copies with (essentially) identical structure, and that the orientations of these molecules are random, ranging the entire angular space with no major gaps. Direct electron detectors capture the emerging scattered electrons after they pass through the protein, forming a series of magnified single-particle 2D image projections (micrographs). Reconstruction software aligns, averages and combines the 2D image projections, yielding a 3D electron density map of the protein [23].

The popularity of SPA stems from the fact that it allows observation of biological specimens in their native environment, not stained or fixed in any way. In all cryo-EM methods, biological samples are processed in their native state and thus captured in their native conformations. This contrasts with X-ray crystallography, where the specimen needs to be crystallized, necessitating difficult and time-consuming procedures that typically place the biological samples into non-physiological environments, which can lead to functionally irrelevant conformational changes. Cryo-EM does not require the samples to crystallize and is therefore capable to image larger proteins. In fact, cryo-EM was first used to resolve large molecular complexes such as ribosomes [6, 39] and membrane proteins [32] with relatively good resolution details. In addition, cryo-EM is not as sensitive to impurities and requires much smaller amount of experimental sample than it is required to grow protein crystals (micrograms instead of milligrams) [30, 81]. However, cryo-EM has its own shortcomings. Due to reduced electron doses, electron micrographs have low signal-to-noise ratio (SNR) and poor contrast which cause the resolution of cryo-EM reconstructions to be worse than X-ray crystallography. Furthermore, cryo-electron microscopy does not work for very small proteins whose 2D image projections cannot be sorted and aligned. The lower size limit for imaging proteins with cryo-EM is estimated to be between 38kDA and 300kDa, while

such size limitation does not exist for X-ray crystallography [34, 40]. Instead of contrasting and comparing cryo-EM with X-ray crystallography, some researchers have proposed that combining both techniques can complement each other and reveal more detailed structural information than each one alone [80].

In the last few years, cryo-electron microscopy using single particle analysis method (termed in the following as cryo-EM or 3D EM) has gained much attention among structural biologists due to breakthrough advances that now have allowed this technique to resolve proteins at near-atomic resolutions similar to the ones provided by X-ray crystallography and NMR. Significant advances in sample preparation [3], detectors [40], motion-correction and image processing algorithms [67] have made cryo-EM suitable for imaging smaller proteins and led to the publications of the first cryo-EM structures at resolutions around 3.5Å in the 2013-2014 timeframe [47]. Since then, the development of the cryo-EM method has undergone a resolution revolution, culminating in the 2017 Nobel Prize in Chemistry [26]. Today, published 3DEM maps have resolutions approaching atomic (1-2Å) range for select few samples, and can be directly used to produce 3D protein models [50].

2.2.2 The Unresolved Problem of Resolution and Map Validation in Cryo-EM

Optical resolution is traditionally evaluated by the smallest distance at which two objects can be distinguished from one another. But this traditional resolution measure is not applicable for electron microscopy because obtained 3DEM maps are affected by noise that may distort this resolution metric. Nonetheless, resolution assessment and validation of cryo-EM maps is highly important for the correct interpretation and analysis of structural results. The concept of resolution in electron microscopy is not well defined and the currently used methods are subjective and not fully agreed on by the scientific community [60, 78]. One approach to approximate resolution of a 3DEM density map is to compare it with an X-ray crystallographic reconstruction with a known resolution [87]. This method has a clear limitation that the X-ray crystallographic reconstruction has to exist to do such comparison. The amount of detail in reconstructed EM maps can also be judged by the appearance of

common structural elements, such as α-helices, β-sheets or aromatic rings (Figure 2.3). This approach is approximate and inherently subjective, and only applicable when the amount of detail in the reconstructed density map is high. When the amount of detail in computed density map is low, there are no external measures or resolved secondary structure elements by which resolution can be judged.

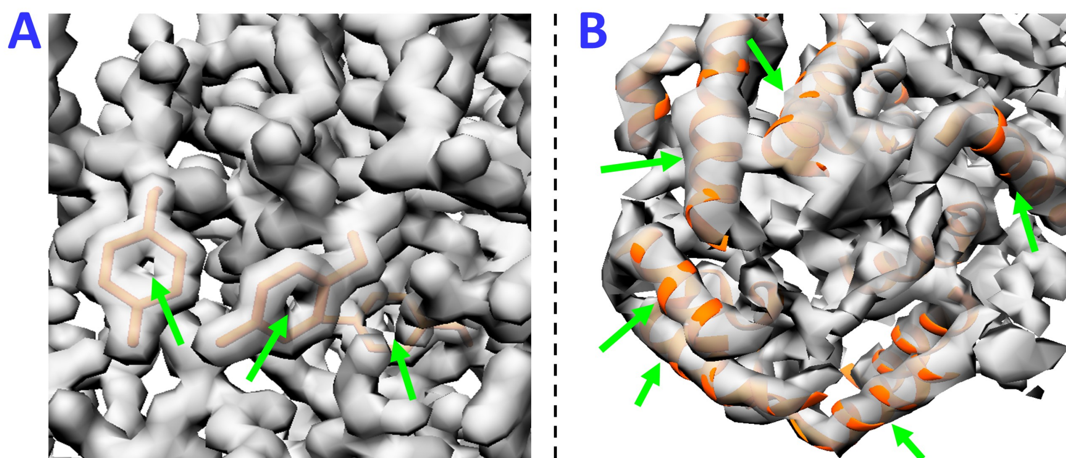


Figure 2.3: *Simulated cryo-electron density maps of two example proteins at 2Å and 6Å, (**a**) and (**b**) respectively, illustrate how resolution can be estimated by the appearance of common structural elements. (**a**) Density is thinned out in the center of aromatic rings, a feature typically observed in structures determined by x-ray crystallography at resolutions of 2Å. (**b**) In density maps at medium (5-10Å) resolutions, α-helices appear cylindrical in shape.*

Previous researches have proposed computational and statistical methods for estimating the resolution of reconstructed 3DEM structures. Current practices include Differential Phase Residual, Q-Factor, Spectral Signal-to-Noise Ratio (SSNR) and Fourier Ring/Shell Correlation [60, 48]. The most commonly used resolution measure today is the gold-standard Fourier Shell Correlation (FSC) procedure. The FSC calculates the similarity at different res-

olutions in frequency (Fourier) space between two independent 3D maps of the same protein, each calculated using one-half subset of the collected cryo-EM data [12]. The calculation produces an FSC curve as a function of the modulus of spatial frequency or resolution [60]. To derive a single resolution value from the FSC curve, a threshold criterion needs to be defined. The FSC threshold value is arbitrarily chosen as the point on the FSC curve at which the two reconstructions are considered consistent [12]. The threshold value and its interpretation remain a debated topic that has resisted a satisfactory solution. Some researchers have proposed various resolution thresholds (like FSC=0.143, FSC=0.5, and FSC=1/3) as indicators of the resolution limit while others argue that fixed-valued FSC thresholds are not appropriate for all reconstructions due to the FSC dependency on the symmetry and size of the structure [77]. Additionally, because FSC uses two reconstructed maps computed from two halves of the data, the final resolution value varies depending on the processing method and whether the source data are split into two halves before or after alignment [60, 66]. Therefore, many argue that FSC is a measure of the quality and consistency of the experiment and not a measure about the resolution and the physical details contained in the reconstructed map. Another issue of using the FSC as a tool to estimate the resolution is that it produces a single resolution value which does not account for locally variable resolution accros the density map. The resolution of cryo-EM maps is not usually homogeneous in magnitude and shows different values along the obtained map, as independently demonstrated by three research groups [21, 44, 79]. All these approaches require a considerable amount of processing time and in some cases their estimates are significantly different from one another.

2.3 Secondary Structure Detection in Cryo-EM Protein Density maps

Regardless of the resolution value, the ultimate goal of cryo-EM imaging is to build an atomic model that is consistent with the known sequence of the protein and its observed geometrical features. The identification of the two types of regularly repeating secondary structure elements (SSEs), α-helix and β-sheet, is a subset of this bigger problem. Once alpha-helices

and beta-sheets are reliably identified in cryo-EM images, an atomic model of the protein structure can be derived using *de novo* modeling techniques that match the secondary structure elements in the density images with those in the protein sequence [4, 5, 18, 28, 49]. Many tools for manual, semi-automatic, and automatic interpretation of electron-density maps and identification of secondary structure elements have been proposed. Due to their easily identifiable appearance as high-density continuous rods of 5Å to 7Å in diameter, alpha-helices in medium-resolution density maps were first identified by visual inspection [19, 85]. However, these primitive methods were fairly subjective as they relied on experience, bias and interpretation. Later work in protein secondary structure prediction relied on more quantifiable image processing and pattern recognition techniques. Jian et al. developed a program called *"helixhunter"* that could identify position, orientation and length of alpha-helices in three-dimensional structures from intermediate resolution electron cryomicroscopy maps [37]. *Helixhunter* models alpha-helices as cylinders and uses seven cylindrical parameters, whose values can be manually specified, as criteria for identifying a helix. It uses a cross-correlation search algorithm to find density segments that fit a prototypical alpha-helix cylindrical template. The method correctly identified 88% of the helices when tested on four simulated protein maps at 8Å resolution. In addition to not being fully automated, *Helixhunter* could not detect both alpha-helix and beta-sheet structures simultaneously. Another tool called *"SSEHunter"* proposed by Baker et al. allowed for simultaneous identification of both α-helices and β-sheets in density maps with resolutions between 10-5Å [11]. SSEHunter measures propensity of high-density areas to be α-helical or β-sheet by combining scores from three algorithms: prototypical helix correlation, skeletonization and local geometry predicates. SSEHunter performed very well on large structures; it had 99.3% accuracy identifying helices that were greater than 8 amino acids and 100% accuracy identifying beta-sheets with more than 2 strands when tested on both simulated and experimental cryo-EM density maps with resolutions between 6Å and 10Å. However, the tool identified less than 30% of small helices with less than four amino acids and sheets with two or fewer strands. There are other similar methods to Helixhunter and SSEHunter that search for cylinder-like

α-helical regions and plane-like regions for β-sheets that offer some improvements on certain aspects like the speed of identification [73, 74] and the level of manual processing [9]. All of these previous methods were based on prescriptive image-processing and pattern matching algorithms that required careful selection of multiple parameters. Most of the methods could identify one or the other secondary structure element but not both simultaneously. Another shortcoming is that these image processing methods do not explore all the existing data to assist in detecting SSEs of new samples.

2.3.1 SSE Detection with Deep Learning Methods

Machine learning models with better accuracy have since been proposed for detecting secondary structures in cryo-EM protein density maps. A trained Support Vector Machine (SVM) classifier showed that it is possible and effective to use available cryo-EM maps with known structures for learning to detect SSE in other unknown cryo-EM structures [69]. Si et al. used feature extraction and supervised learning to train an SVM model to detect both helix and sheet structures from background/other parts. The trained SVM model was combined with image post-processing techniques into a tool called *"SSELearner"*. SSELearner achieved high specificity and sensitivity for identifying the number of helix and beta-sheet structures in both simulated and experimental maps. SSELearner improved the accuracy of detecting medium-sized helices with lengths between 5 and 8 amino acids and beta-sheets containing two strands, but it still struggled to detect short alpha helices made of fewer than five residues. The SVM algorithm separated voxels based on only 5 cylinder- and plane-like structural features calculated from density values but did not consider other local or global features. The model indirectly predicted the locations of SSEs in the map by first identifying the Cα atoms in the protein backbone and then using empirical distance metrics from the Cα atoms of 2.5Å and 3Å in order to identify helix and sheet voxels, respectively.

More recently, fully trainable deep learning convolutional neural networks (CNN) have been extended to tasks involving electron density map segmentation. In a 2016 paper, Li et al. presented one such deep learning model for detecting α-helix and β-sheet structures from

cryo-EM images at 8Å resolution based on a deep convolutional neural network. Dilated convolution with residual learning approach was used to create the deep CNN classification model which was trained with 15 cryo-EM images with known secondary structures. The SSE detection performance was evaluated using 10 other cryo-EM protein structures which had not been used for the training. The presented convolutional neural network method outperformed prior methods on detecting secondary structures of proteins from volumetric density maps. The average sensitivity and specificity of the CNN reached 71.52% and 87.86% for α-helix identification and 76.04% and 91.87% for β-sheet identification [46]. This deep learning model also identified the C*alpha* atoms that belonged to each secondary structure and used a separate labeling algorithm to mark all voxels within 2.5Å and 3.5Å around the Cα atoms. This approach led to high rates of false positives and caused the model to even miss identify some Cα atoms. In addition, the model was trained with patches of a small number of cryo-EM maps (only 15) and while it yielded high specificity and sensitivity and high F1 scores, it is likely that using more samples for training could yield better predictive accuracy. The research group only examined the predictive capabilities of the CNN model on artificially simulated cryo-EM density images and did not consider experimental electron density maps. Furthermore, the researchers applied a post processing step that was not specifically designed for the CNN generated data. Instead, a post processing step that had been designed to remove false positive predictions from the earlier Support Vector Machine model was used. The post-processing technique was not effective for the CNN model and did not improve predictions.

Today, newer and more sophisticated deep learning libraries have been released and it is common and computationally possible to construct complex networks with high number of layers and filters that can make dense voxel-wise predictions and still be efficiently trained on large datasets. In the next section, a short description of some of the current state-of-the-art deep learning neural network architectures used for image segmentation is provided.

2.4 Convolutional Neural Networks

Neural networks are made of multiple layers of simple computational units, called neurons, connected in an acyclic graph. Each neuron represents a mathematical model of a biological neuron. In simple terms, biological neurons receive input signals from other neurons and produce an output signal based on the input. In computational neural networks, each neuron performs a mathematical operation on its inputs, most often multiplication (dot product) with a weight coefficient followed with an addition of a bias parameter. Scalar multiplication and addition are linear transformations therefore they can be used in neurons to capture only linear relationships between input and output. Oftentimes in nature, the relationship between input and output is more complex. To capture non-linear relationships between inputs and outputs, computational neurons are modeled with non-linear activation functions. Popular activation functions used in neural networks are Sigmoid, Tanh, Rectified Linear Unit (ReLU).

Neural networks organize neurons in interconnected stacked layers. Neurons are arranged in a single dimension within a layer and are not connected to one another, while neurons between two adjacent layers are typically fully connected pairwise. The first layer of a neural network is called the input layer, while the last layer is the output layer. In between the input and output layers, several hidden layers of neurons may be present. The often large number of hidden layers gives rise to the term "deep" in deep learning. The goal of training a neural network is to find the weights and biases of all neurons that can produce a correct output based on a given output. In supervised learning algorithms, the correct output is provided to the network along with the raw input. Then, parameters of the neurons are updated in an iterative process such that the error between the predicted output and the known, true output is minimized over many samples. Famous optimization algorithms commonly encountered in the literature include Stochastic Gradient Descent (SGD), Adaptive Gradient Algorithm (AdaGrad), and Adaptive Moment Estimation (Adam) [29, 41, 64].

Convolutional neural networks (ConvNets or CNNs) are a type of neural networks op-

timized to work well on 2D images and other higher dimensional input. ConvNets have similar stacked architecture of layers with neurons that have learnable weights and biases as ordinary neural networks, however, neurons in ConvNets are arranged in multiple dimensions within a layer. ConvNets contain at least one (but more often many) layers of neurons that perform convolution mathematical operations. The neurons in a convolutional layer are arranged as a sequence of filters and they are connected to only a small region of neurons in the layer before them. This connectivity constraint allows for parameter sharing between neurons and reduces the number of trainable parameters, while still allowing different filters to extract different features (like edges or other shapes) from the same input. Non-linear activation functions are used after convolutional layers to capture non-linear relationships between input and output. Thus, CNNs yield hierarchies of features through non-linear mappings between multiple stacked layers. In addition, ConvNets often have pooling layers inserted between convolutional layers to further reduce the amount of parameters and computation in the network. The final output of a CNN is often produced via a fully connected layer adapted to the particular application.

CNNs were first successfully used for whole-image recognition where static 2D images were classified into different classes [43]. These first CNN models were able to recognize a single object or a scene present in the image. ConvNets were then expanded to recognize and localize objects in smaller, coarse bounding windows within images [33]. Bounding box detection models can locate and classify multiple objects within the image and draw rectangles around them. More recently, ConvNets are being explored for semantic segmentation applications where models can make a dense prediction at every pixel. Next section summarizes the basic structures of CNN architectures for dense predictions, which serve as examples for building a neural network for segmenting 3D electron density maps.

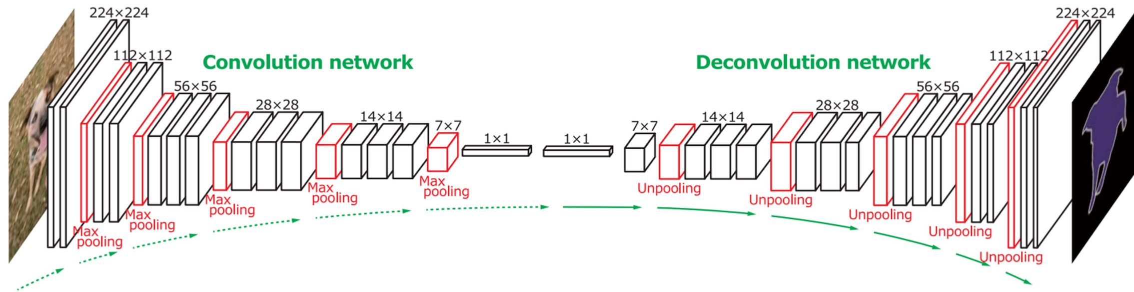
2.5 Semantic Segmentation with CNNs

A 2014 paper by a UC Berkeley research group presented a groundbreaking end-to-end convolutional network architecture for semantic segmentation [52]. The Fully Convolutional

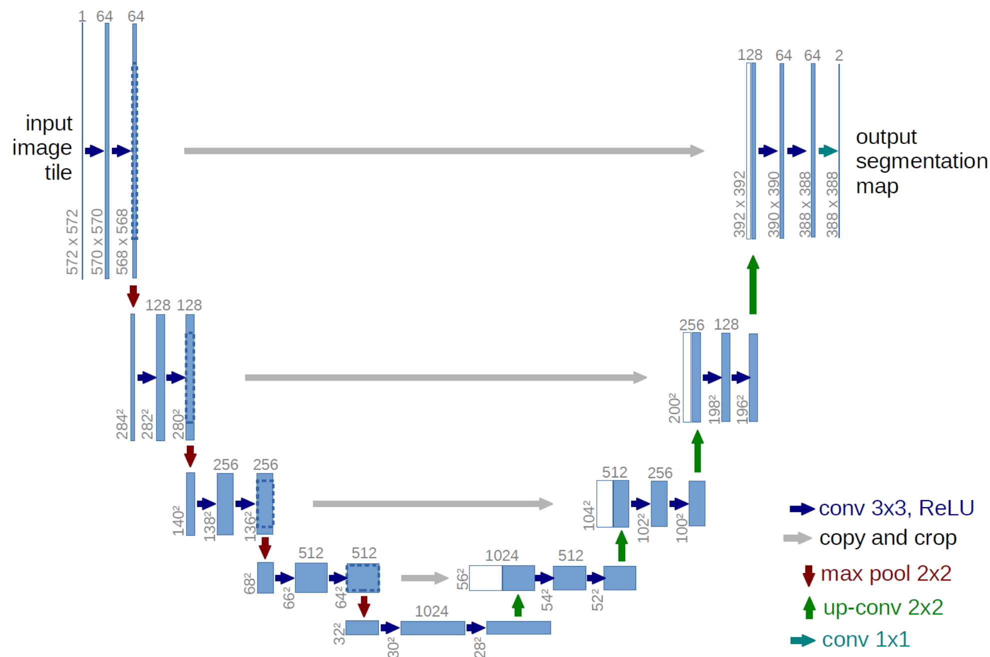
Network (FCN) architecture augments whole-image classification networks by replacing the final fully connected layers with convolutional layers. Additionally, FCN adds in-network upsampling layers. Another highlight of the FCN approach was the introduction of skip connections where activation from upper coarse layers are combined with lower, finer-stride layers. The convolutional part of an FCN is a conventional classification network; the original paper demonstrates the FCN adaptation of three well-studied, pre-trained image classification networks (AlexNet, VGG net and GoogLeNet). Extracted features from the pre-trained convolutional layers are up-sampled to the original input image size by appending transpose convolutional layers (also referred to as deconvolution). The added deconvolution layers have learnable parameters that are optimized during additional fine-tuning. That additional training was facilitated by the skip connections which combined local and global features. In addition to speeding up the training, the skip connections significantly improved the performance of the model by allowing it to make finer local predictions while respecting global structure. The FCN architecture achieved state-of-the-art segmentation performance on benchmark datasets at that time.

The original FCN architecture introduced the basic structure with two distinct convolution and deconvolution sub-networks that most semantic segmentation models are based on today. Because the convolutional layers are purposed to encode features from the input while the deconvolutional layers are purposed to decode the encoded features, this type of ConvNet architectures are also referred to as Encoder-Decoder networks. Later segmentation models attempt to address some of the shortcomings of FCN, like the coarseness of the predictions and the sensitivity to small objects. Some of these FCN inspired models include the U-Net [63] and *Deconvnet* [57] architectures (Figure 2.4). *Deconvnet* follows the FCN backbone by using convolution and max pooling for feature extraction and uses deconvolution and unpooling layers in reverse order to upsample images to original size, but avoids skip connections. Deconvnet achieved better results than FCN and was able to distinguish small objects, however, deconvnet required much longer training stage. The U-Net architecture consists of similar symmetrical contracting and expanding paths but adds convolutional

layers in the decoder. It also uses skip connections to capture both global and local features. The U-Net architecture was capable to segment 512x512 pixel images in less than one second, after a 10-hour training. The U-Net architecture won top honors by a large margin on cell tracking and segmentation of neuronal structures in electron microscopic stacks at the 2015 International Symposium on Biomedical Imaging (ISBI). These pioneering network architectures were initially designed to segment 2D images, but most recently, researchers have started adapting them for segmentation of 3D images, especially for medical application [36, 38, 53, 65]. Inspired by the 3D U-Net architectures in [36] and [65], this work explores 3D U-Net model for semantic segmentation of protein density maps based on α-helix and β-sheet secondary structure elements in Chapter 4.



(a) *Deconvnet*



(b) *U-Net*

Figure 2.4: ***(a)*** *Deconvnet architecture illustrates the FCN backbone by fusing traditional classification network with multiple deconvolution and unpooling layers to produce a probability map with the same size as the input. Image from Noh et al. (2015) [57].* ***(b)*** *U-Net architecture features symmetrical contracting and expanding paths and uses skip connections to produce fine-grained segmentations. Image from Ronneberger et al. (2015) [63].*

Chapter 3

RESOLUTION VALIDATION OF THREE DIMENSIONAL CRYO-ELECTRON MICROSCOPY DENSITY MAPS

3.1 Methodology

The goal of this research is to overcome the limitations of the Fourier Shell Correlation and other resolution estimation methods by presenting a deep learning approach for resolution validation of cryo-EM density maps. Deep learning neural networks have been shown to be capable to classify images by automatically extracting high representative global and local information. Therefore, we would like to test the hypothesis that neural networks can be used to objectively quantify the resolution of 3D cryo-EM density maps. For this purpose, we use supervised learning techniques to construct neural network models capable of learning to distinguish electron density maps at high (<5Å), medium (5-10Å) and low(>10Å) resolutions. The trained models are then used to evaluate simulated and experimental cryo-EM density maps. 3D electron density maps with different resolutions were first generated from solved 3D atomic structures using image processing software tools. To evaluate the hypothesis that deep learning models can extract and "learn" resolution features from 3D cryo-EM maps, datasets with simulated maps at three different resolution levels were created, pre-processed, and then used to train two neural networks. The two neural network architectures explored in this investigation were dense artificial neural network with non-linearity (DNN), and 3D convolutional neural network (3D CNN). These deep learning models were first trained on thousands of simulated maps with their corresponding known resolution label. The performance of the models was evaluated by comparing the predicted resolution category from the model with the known, simulated resolution of density maps that had not been used for training. In the second phase of this investigation, experimental 3D cryo-EM

electron density maps were used for testing, and the models were evaluated based on the agreement between the published resolution value and the predicted value from the model. The high-level representation of the experimental design is depicted in Figure 3.1.

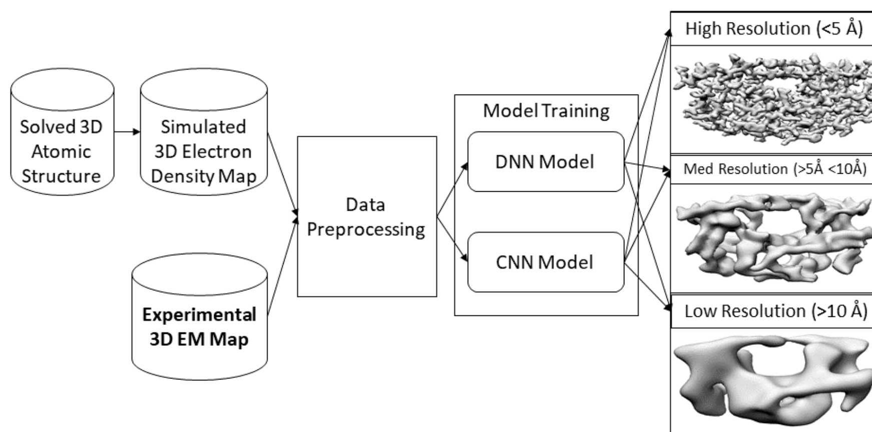


Figure 3.1: *3D electron density maps are simulated from solved 3D protein atomic structures and used to build deep learning classification models. The trained models are then used to classify experimental 3D cryo-EM density maps intro tree categories (high, medium, low resolution).*

3.2 Data Collection and Data Pre-processing

Solved 3D protein structures were downloaded from the publicly available Protein Data Bank (PDB) [15]. A query was used to select structures containing only single-chain proteins with molecular weights between 10 and 200kDa (KiloDalton). The query returned around 47000 results which were further reduced to 12671 structures by removing models with more than 40% structural similarity. Using the EMAN2 software package and the function "pdb2mrc" [70], the selected 12671 PDB structures were simulated to 3D electron density maps with 2.5Å, 7.5Å and 12.5Å resolutions, corresponding to high, medium, and low resolution, respectively. The "pdb2mrc" function models each atom as a 3D Gaussian distribution, and

the resolution value is the reciprocal of the half-width of that Gaussian distribution in Fourier space. The simulation box size was set to 64x64x64 and the sampling rate was 1Å/voxel. The electron density values of all maps were linearly normalized between 0 and 1 to ensure uniform intensity ranges across all proteins. The 38013 simulated normalized maps were uniformly and randomly split into training, validation and test sets with 60%-20%-20% ratio, respectively, and packaged into a single "hdf5" file. Each data subset was confirmed to contain approximately equal number of maps at the three different resolutions (Table 3.1). All the data pre-processing steps were performed using Python 3.5.2. Visualization of protein models and electron density maps was done in UCSF Chimera tool [61].

Table 3.1: Map counts in each data subset used for resolution validation experiment.

	Training	**Validation**	**Test**
High	7597	2496	2578
Medium	7661	2529	2481
Low	7549	2578	2544

3.3 Model Training

The neural network models were constructed and executed using the Python API of Tensorflow r1.5 software package on a Linux node with NVIDIA TITAN X (Pascal) with 12 GB GPU memory. The training dataset of simulated maps was input into the models along with their known resolution class. Starting from default randomly initialized state, the models optimized a loss function that measures the probability error between the predicted and known labels during training. A non-exhaustive random search of different hyperparameters was performed to identify the set of loss functions, optimization algorithms, activation functions, learning rate, batch size and number of epochs that produced the overall smallest training error, highest validation accuracy, and fastest training time. The best performing DNN and

3DCNN models are described in the next sections.

3.3.1 Dense Artificial Neural Network

Artificial neural networks (ANN) were first designed to model biological neural systems. They consist of multiple layers of neurons connected in an acyclic graph. In fully connected, dense artificial neural networks, all neurons in one layer are connected to all neurons in the next layer with an activation function applied along the connection. We construct one such network architecture for the resolution prediction of 3D EM maps. Our dense ANN contains an input layer and two hidden layers with 1000 and 100 neurons, respectively. The neurons in both hidden layers are followed by ReLU activation function. The purpose of the activation function is to build the nonlinear relationship between input and output. The final fully-connected output layer performed classification with 3 neurons, one for each resolution target class. The detailed configuration of the proposed DNN is provided in Figure 3.2.

The input layer with 64*64*64 neurons takes the voxels of a 3D map as input, multiplies the intensity values of each voxel with the weights of the 1000 neurons in the first fully connected layer, adds a bias, applies a ReLU mapping and feeds the output to the second hidden layer. The 100 neurons in the second hidden layer multiply, add a bias, and apply a ReLU nonlinearity to its input. The 3 neurons in the output layer map probabilities to each class, with a softmax normalization. The DNN model was trained with Gradient Descent Optimizer and constant learning rate of 0.01. Sparse softmax cross entropy with logits was used as the training loss function. The training batch size was 150 and early stopping was implemented to stop training after 50 iterations with no progress in minimizing the loss function.

3.3.2 3D Convolutional Neural Network

Stacked convolutional neural networks (CNNs) are currently the most popular model architecture for deep learning. This type of model architecture has been successfully applied in many image classification tasks. Convolutional networks have been shown to extract and

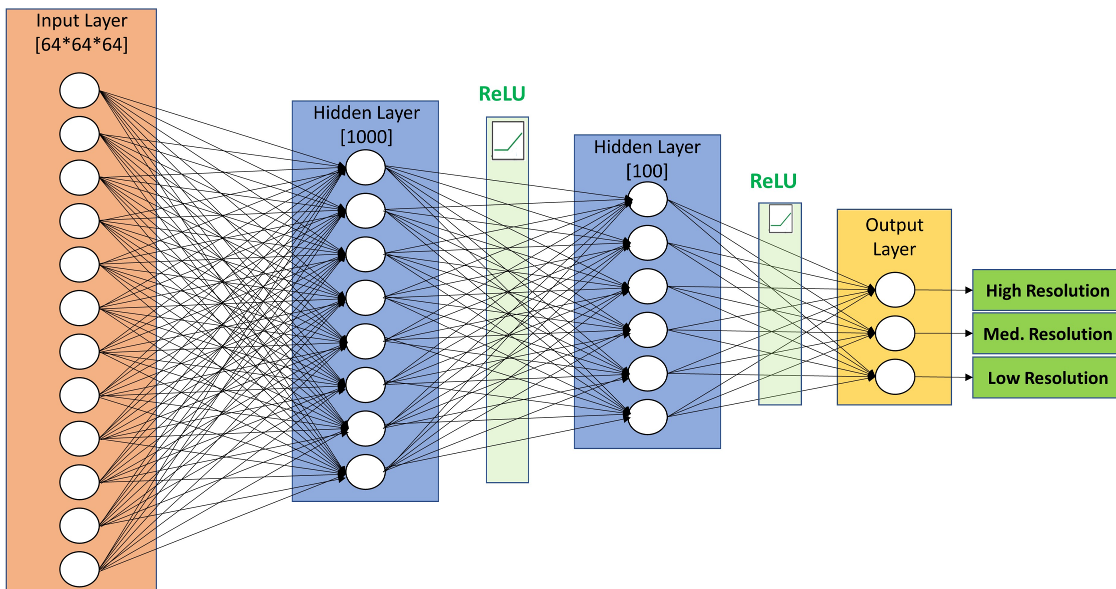


Figure 3.2: *The fully connected neurons in the DNN Architecture and the application of non-linear mapping functions transform an input map into a single label corresponding to high, medium and low resolution.*

learn higher-level features associated with a known label and then use these features to classify unseen images with unknown labels. Most of the previously published CNNs apply a series of filters to the raw pixel data of a 2D image. CNNs have achieved state-of-the-art performance on image and object recognition tasks [43, 84]. In this paper, we propose to use a 3D CNN architecture that uses three-dimensional convolutional filters applied to the three-dimensional electron density maps. Our 3D CNN contained stacked convolutional and pooling layers with different non-linear activation functions, a fully connected dense layer and a logits output layer (Figure 3.3). The 3D CNN network maps an input 3D map to one of the three resolution classes (high, medium, low). The first convolutional layer (3D CNN#1) applies 32 7x7x7 filters. The second (3D CNN#2) and third (3D CNN#3) convolutional layers apply 64 and 128 5x5x5 filters, respectively. The first two convolutional layers

are activated by an eLU activation function [25], and the third is activated by ReLU (not shown on figure). Zero-padding is used following each convolutional layer to preserve map dimensions. A max pooling operation with a 2x2x2 filter and stride of 2 is applied after each convolutional layer to downsample (reduce by half) the map data size. Dense layer with 1024 neurons activated by Tanh function and dropout regularization (rate = 0.4) performs classification on the features extracted by the convolutional and max pooling layers. The "logits layer" maps class probabilities with a softmax normalization and outputs one of the three resolution classes (0-high, 1-medium, 2-low). The 3D CNN model was trained with Gradient Descent Optimizer and learning rate of 0.01. Sparse softmax cross entropy with logits was used as the training loss function. The training batch size was 75.

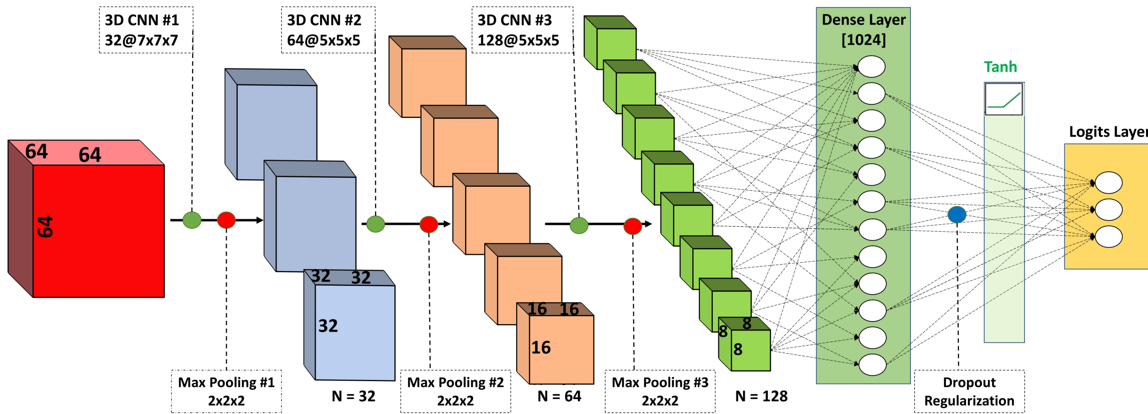


Figure 3.3: *3D CNN Architecture showing the layers of convolutional blocks of different size and the reduction of an input map to one of three output labels.*

3.4 Results

3.4.1 Model Metrics

The DNN model was trained for 90 epochs (triggered by early stopping). The accuracy and the loss function value were recorded for the validation set at each epoch. The validation accuracy increased to around 92%, and the loss value decreased to 0.19 as the training

progressed (Figure 3.4). After completion of the training stage, the performance of the DNN model was evaluated on a test data set containing simulated maps that were not included in the training process and were never seen by the trained model. The test accuracy of the DNN model was 92.73% on simulated maps.

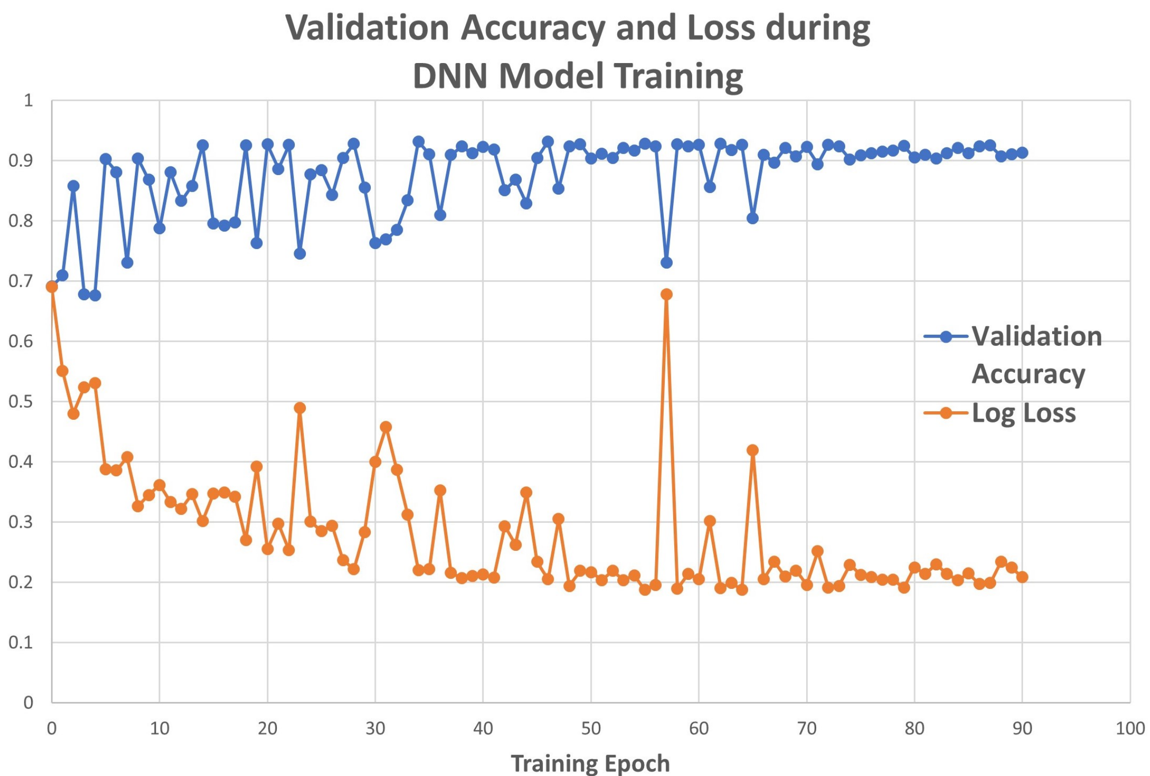


Figure 3.4: *DNN Validation Accuracy and Loss*

The training of the 3D CNN model was discontinued after 1500 epochs, at which point the optimization of the loss function was saturated. The validation accuracy and the loss function were recorded every 100 epochs during the training stage. The accuracy increased as training progressed and approached 100% at 1500 epochs, while the loss approached 0.01 (Figure 3.5). The test accuracy of the trained 3D CNN model was assessed on a data set containing simulated maps that were not included in the training process and were never

seen by the model. The test accuracy of the 3D CNN model was 99.75%.

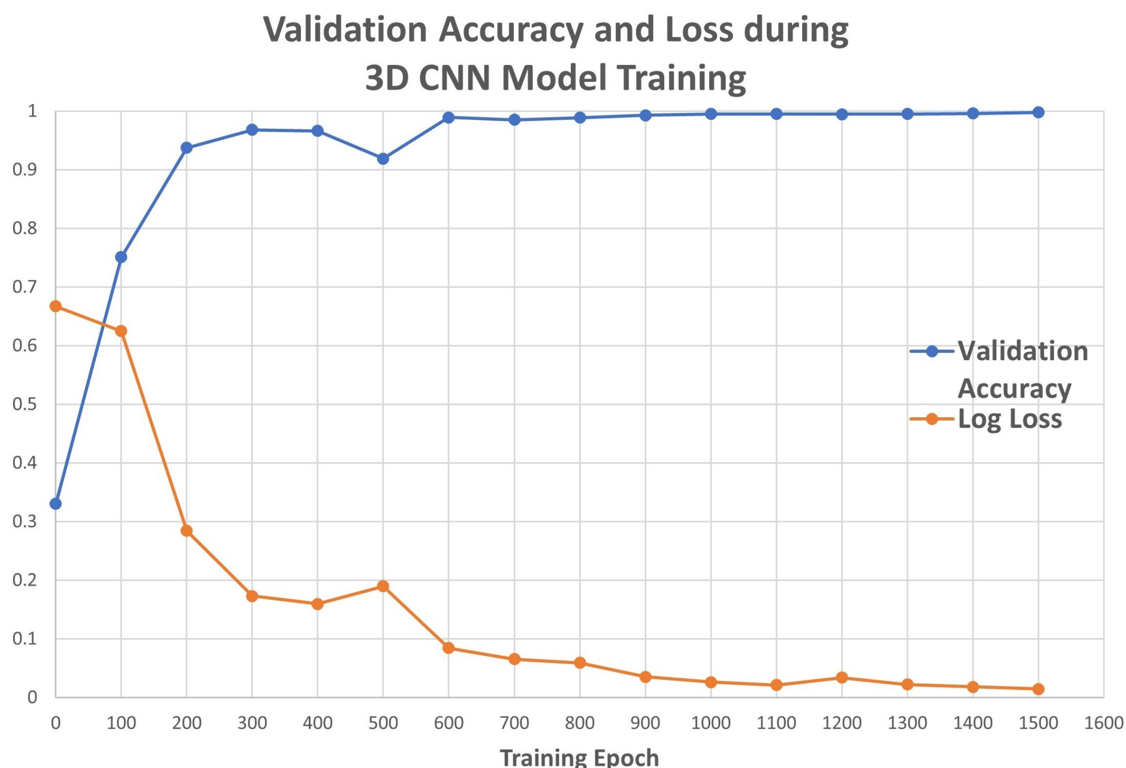


Figure 3.5: *3D CNN Validation Accuracy and Loss*

3.4.2 Performance on simulated cryo-EM density maps at varying resolutions

In addition to the 2.5Å, 7.5Å and 12.5Å maps contained in the validation data set, the trained models were tested on an unseen dataset containing 28 different protein maps simulated at resolutions between 1.5Å and 15.0Å at each 0.5Å increment. The protein PDB ID was chosen at random for each resolution and care was taken to not include any structures contained in the training set. The 28 maps were downloaded from the Protein Data Bank and simulated with the “pdb2mrc” function to a 64x64x64 box with 1Å voxel spacing. The simulated maps were additionally processed with the same normalization function as used for the training

dataset. The 28 unlabeled maps were input to the pre-trained DNN and 3D CNN classifiers. The predicted resolution classes of all 28 maps are listed in Table 3.2.

Our models classified maps with similar resolutions as belonging to the same class. This experiment of classifying simulated maps with gradually variable resolutions identified the classification boundaries of the trained models. Classification boundary is the resolution at which the model changes its output from one class to another. The DNN model classified all maps with resolutions <= 4.0Å as high, maps with resolutions between 4.5-8.5Å as medium, and all other maps with resolutions >= 9.0Å as low. The 3D CNN model classification boundaries between high-medium and medium-low resolution classes were at 4.5Å and 8.5Å, respectively. All maps predicted by the 3D CNN model followed these classification boundaries unlike the DNN model which had two outlying predictions; the 7.5Å map was classified as low resolution and the 10.5Å map was classified as medium resolution.

Table 3.2: Classification Results on Simulated Maps with Varying Resolution.

Simulated Resolution	DNN	3D CNN	Simulated	DNN	3D CNN
1.5 Å	High	High	8.5 Å	Medium	Medium
2.0 Å	High	High	9.0 Å	Low	Low
2.5 Å	High	High	9.5 Å	Low	Low
3.0 Å	High	High	10.0 Å	Low	Low
3.5 Å	High	High	10.5 Å	Medium	Low
4.0 Å	High	High	11.0 Å	Low	Low
4.5 Å	Medium	High	11.5 Å	Low	Low
5.0 Å	Medium	Medium	12.0 Å	Low	Low
5.5 Å	Medium	Medium	12.5 Å	Low	Low
6.0 Å	Medium	Medium	13.0 Å	Low	Low
6.5 Å	Medium	Medium	13.5 Å	Low	Low
7.0 Å	Medium	Medium	14.0 Å	Low	Low
7.5 Å	Low	Medium	14.5 Å	Low	Low
8.0 Å	Medium	Medium	15.0 Å	Low	Low

3.4.3 Performance on experimental cryo-EM density maps

Thirty published cryo-EM maps were downloaded from the EM Data Bank (www.emdatabank.org). The experimental maps were chosen based on their published resolution value; ten maps are claimed to have high resolutions ranging from 1.6Å to 2.9Å, ten had medium resolution between 6.0Å and 8.0Å, and ten had low resolution in the range from 11.0Å to 14.8Å. Table 3.3 lists the EM Data Bank (EMDB) IDs of the thirty experimental cryo-EM density maps and their claimed resolution values. The thirty experimental maps were preprocessed with the same normalization technique as used for the simulated maps. Because the trained models require an input of constant 64x64x64 size, an additional pre-

processing step was applied to the experimental maps. The experimental maps had variable dimensions ranging from 45 to 432 voxels. For the maps with dimensions greater than 64, central cropping operation was applied to remove the outer parts of the maps and retain the central 64x64x64 region. In cases where a map had a dimension lower than 64, zero-padding was applied at the end to increase it to 64. The post-processed experimental maps were input to the DNN and 3D CNN models. Both models classified each experimental map as either high, medium or low resolution, and the results are included in Table 3.3.

Table 3.3: Classification Results of the 30 Experimental cryo-EM Density Maps

EMDB ID	Resolution	DNN	3D CNN	EMDB ID	Resolution	DNN	3D CNN
8221	1.6Å	High	High	5751	8.0Å	High	Medium
2984	2.2Å	High	High	3340	7.2Å	Medium	Medium
2945	2.9Å	High	High	5101	8.0Å	Low	Low
6342	2.5Å	High	High	3168	7.4Å	Medium	Medium
8218	2.3Å	High	High	5653	7.3Å	High	High
8222	2.5Å	High	High	7089	13.2Å	Low	Low
8077	1.75Å	High	High	1601	14.1Å	High	High
8217	1.8Å	High	High	6013	14.8Å	Medium	Medium
8219	2.5Å	High	High	3368	13.0Å	Medium	Medium
6313	2.5Å	High	High	2862	12.5Å	Low	Low
3311	6.7Å	Medium	High	3098	11.0Å	Medium	Medium
5664	7.8Å	High	High	7471	12.5Å	Medium	High
6410	7.8Å	High	High	1711	13.0Å	Low	Low
4047	6.0Å	Medium	High	5169	11.0Å	Medium	Medium
1674	6.0Å	Medium	Medium	1981	14.2Å	High	High

We further measured the agreement between the unverified published resolution level and the resolution level predicted by the classification models. Table 3.4 and Table 3.5 show the confusion matrices for the DNN and 3D CNN models, respectively.

Table 3.4: DNN Confusion Matrix

DNN Confusion Matrix		Predicted Resolution		
		High	Medium	Low
Published Resolution	High	10	0	0
	Medium	4	5	1
	Low	2	5	3

Table 3.5: 3D CNN Confusion Matrix

3D CNN Confusion Matrix		Predicted Resolution		
		High	Medium	Low
Published Resolution	High	10	0	0
	Medium	5	4	1
	Low	3	4	3

The confusion matrices count the number of maps that truly belong to each class and were predicted to belong to that class. The DNN model prediction and the published resolution match for all 10 high resolution maps, 5 medium resolution maps, and 3 low resolution maps, yielding a 60.0% combined agreement for all maps. The 3D CNN model prediction and the published resolution match for all 10 high resolution maps, 4 medium resolution maps, and 3 low resolution maps, yielding a 56.7% combined agreement for all maps. The DNN model classified 16 of the 30 maps as high resolution, 10 as medium resolution and only 4 as low resolution. The 3D CNN model classified 18 maps as high resolution, 8 as medium resolution and 4 as low resolution.

To further assess the performance and predictive abilities of our classifiers, the confusion matrices were used to calculate the sensitivity, specificity, positive predictive value (PPV) and negative predictive value (NPV). The results included in Table 3.6 for the two multi-class classifiers are calculated according to the partial class membership method proposed by Beleitas et al. [14].

Table 3.6: Performance Measures of the Classifiers

		Sensitivity	Specificity	PPV	NPV
DNN Model	High	100.00%	70.00%	62.50%	100.00%
	Medium	50.00%	75.00%	50.00%	75.00%
	Low	30.00%	95.00%	75.00%	73.10%
CNN Model	High	100.00%	60.00%	55.60%	100.00%
	Medium	40.00%	80.00%	50.00%	72.70%
	Low	30.00%	95.00%	75.00%	73.10%

Sensitivity measures how well the classifiers recognize maps belonging to each resolution level while specificity measures the ability to recognize maps that do not belong to the given resolution group. PPV measures the probability a given map truly belongs to the resolution group predicted by the model. NPV measures the probability a map truly does not belong to a resolution group when the model predicts it does not belong to that resolution group. According to these results, the two classifiers have similar predictive capabilities except for the slightly better specificity and positive predictive value for high resolution maps of the DNN model over the 3D CNN model. Both are sensitive to high resolution maps but fail to recognize low resolution maps. The specificity for both classifiers is highest for low resolution maps and lowest for high resolution maps. For both classifiers, there is only a 50% certainty that a medium resolution prediction is correct as indicated by the PPV. However, the higher NPV for medium resolution suggests our models can better classify maps that are not medium resolution.

3.5 Discussion

Both DNN and 3D CNN classification models were successfully trained, and minimized loss functions close to zero. The DNN model converged in 90 epochs, faster than the 1500 epochs needed to train the 3D CNN model. The 3D CNN model had higher prediction accuracy

(99.75%) on the simulated test data set compared to the 92.73% of the DNN model. However, both the DNN and the 3D CNN models did not perform well on the experimental electron density maps. The two models had similar predictive capabilities as well as accuracies (60.0% for the DNN vs 56.7% for the 3D CNN) when evaluating the dataset of thirty published experimental maps. The large percentage of disagreement suggests that there are significant structural and volumetric differences between experimental and ideal (simulated) maps at each resolution level. More than half of the thirty experimental maps were classified as high resolution: 16 by the DNN model and 18 by the 3D CNN model. The large number of reported medium- and low-resolution maps that were incorrectly classified as high resolution suggests that our models might be oversensitive to high-resolution features. However, both models classified as low resolution the same map (EMDB-5101) which was claimed to have 8Å (medium) resolution in the EM Data Bank entry (see Table 3.3). This finding suggests a possibility that the author claimed resolution value may also be inaccurate.

Overall, these observations suggest that our approach to train on simulated maps was not adequate enough for the classification of experimental cryo-EM maps. Potential reasons of this issue are: 1) Simulated 3D electron density maps at different resolution levels from PDB models obtained by low-pass filtering do not accurately represent experimental cryo-electron microscopy maps. Note that these simulated maps are not affected by noise as they are obtained directly from atomic models; 2) Simulated maps present homogeneous resolutions as they were low-pass filtered to the same resolution value. Cryo-EM maps usually show inhomogeneous resolution values, therefore, different parts of the map exhibit different resolutions. To improve the accuracy of our deep learning approach, we need to account for local resolution variances within a map. A voxel-wise classification model may be more appropriate instead of a single-valued resolution classification.

Both models were trained to make coarse-grained classifications as only three resolution classes were considered, but the models could classify simulated maps with varying resolutions. The trained models consistently classified maps with similar resolutions to the same class. Our results also confirmed that comparable structural features exist in protein density

maps at close resolutions. The structural features at resolutions less than 4-4.5Å were picked up by the model as distinctly different than the features present at 5.0-8.5Å density maps. The trained classification models also "learned" to recognize the features, or the lack thereof, in protein density maps at low (>= 9.0Å) resolutions.

The input layers of the two proposed models require regular 3D voxel data with constant size which necessitated additional preprocessing (cropping and zero-padding) to transform the experimental 3D electron density maps into expected input dimensions. Proper voxelization of proteins into 3D density voxel maps was also required for the simulated maps used for training the models. Proteins can greatly vary in size and shape which makes determination of 3D voxel grid dimensions challenging. Selecting grid dimensions that are too small struggle to handle very large and/or irregular proteins and may result in maps that cut off a portion of the protein structure. Conversely, selecting very large grid dimensions may result in output that is unnecessarily voluminous with lots of zero intensity voxels, especially for density maps of small proteins. Additionally, the large grid size consumes a proportionally large amount of memory, which can slow down the performance and training of the neural network model. Our approach to address this issue was to use a 64x64x64 voxel grid and select relatively small proteins for the simulated dataset used for model training. Several simulated maps were visually inspected to confirm the 64x64x64 voxel box was large enough to fit the entire protein structure for the first two models. But such manual inspection was impractical for all 12671 proteins in the dataset leaving the possibility that some simulated maps were erroneously cut off.

Finally, the performance of supervised learning models is affected by the quality of training data. The simulated 3D maps were generated using a sampling rate of 1Å/voxel while the experimental maps had varying voxel spacing. In addition, the training dataset was selectively chosen to include small single-chained proteins, while the experimental maps were of various large, multi-chain proteins. These limitations in the training set are probable explanations why our models did not perform that well on the experimental density maps as they performed on the simulated maps. Furthermore, the simulated maps filled the 3D

volume of the 64x64x64 box and contained no noise outside of the molecule (density values were zero in the voxels where no atoms were present). However, experimental maps had varying box sizes not entirely filled by the molecule and the voxels outside of the protein atoms contained noise. The presence of noise in the experimental maps likely had a negative impact on the performance of the classifiers. It was also observed that the resolution values of the 30 experimental maps were determined by different resolution methods. Most of the published cryo-EM maps reported using the FSC method for calculating the map resolution, although with different FSC cut-off values (0.143 and 0.5). Other maps reported using a "Diffraction Pattern/Layerlines" method for determining the experimental resolution value [68]. There was no correlation between the predicted-published resolution agreement and the reported resolution method.

Chapter 4

DETECTING SECONDARY STRUCTURES IN PROTEIN CRYO-EM DENSITY MAPS

4.1 Data Collection and Preparation

The purpose of a machine learning system is to infer a mapping function between input and output. Segmenting and classifying different secondary structure elements in a protein is a supervised machine learning problem. In supervised learning algorithms, the target mapping function is estimated using labeled data. Data samples inputted to the algorithm need to be paired with their true output label during the training stage. After observing many examples of input and output pairs, the model infers a mapping function for all voxels that satisfies all training examples with the least amount of error. The generalizability of the learned mapping function is evaluated using a test dataset, separate from the set used during training. The output labels of the samples in the test dataset need to also be known, however, they are not inputted to the model during the testing stage. Instead, the trained model makes predictions based on the test input, and the correctness of the predictions are evaluated by comparing them to the test labels. Thus, the goals of data collection for the classification of secondary structure elements in proteins were to gather training and test sets of 3D electron density maps with each voxel labeled based on α-helices, β-sheets, and turn/loops. Although there are many experimentally obtained cryo-EM density maps at medium resolution, most of them contain noise and do not have a corresponding solved structure to identify which voxels make up α-helices, β-sheets, and turns/loops. Furthermore, the cost of labeling secondary structures in protein electron density maps is quite high because it requires months of expensive analysis by expert crystallographers. Therefore, real density maps are unsuitable for training. Instead, we propose a novel application of an existing

method to simulate segmented electron density maps from existing solved 3D protein models and generate labeled data suitable for training deep learning models.

4.1.1 Solved 3D Models from Protein Data Bank

Thousands of protein structures have been experimentally determined and 3D models with exact atom coordinates have been published in database archives. The first step in the data collection process was to download solved 3D protein structures from the publicly available Protein Data Bank (PDB) (https://www.rcsb.org/) [15]. The PDB database archives more than 147-thousand biological macro-molecular structures as of December 2018. Each one of the entries in the Protein Data Bank is assigned a unique and immutable 4-character PDB ID, which is commonly used throughout the scientific literature as the sole identifier of a PDB structure. A query on the Protein Data Bank website was used to select 3D models of single-chain proteins with molecular weights between 5 and 250kDa that contained 15 or more percent α-helix and β-sheet structures. The query returned 25199 PDB IDs which were saved to a text file in a list format. The query interface of the Protein Data Bank provides an additional query parameter to filter out similar structures based on sequence identity. When the query parameter was set to filter out structures with more than 30% sequence similarity, the query returned 4672 PDB IDs. Although the relationship between sequence similarity and structure similarity is complex, it was assumed that the 4672 structures are more representative overall of the structure varieties in the entire dataset. Therefore, when splitting the original list of 25199 PDB IDs into training and test sets, all 4672 PDB IDs were placed in the training set. The remaining 20527 PDB IDs were randomly shuffled and split into two parts, with 15487 structures going to the training set, and the other 5040 going to the test set. The result of this splitting procedure was the creation of two lists of PDB IDs stored as text files: the training list contained 20159 PDB IDs while the test list contained 5040 PDB IDs, yielding a 80%-20% train-test split. The 80:20 split is commonly used in Machine Learning [56] to ensure having a large enough training data set to minimize overfitting and a large enough test dataset to validate the generalizability of the trained

models.

4.1.2 PDB File Format

The Protein Data Bank stores three-dimensional protein structure data in a standardized PDB file format. The PDB file consists of a number of text lines. Each line of information is called a record, and each record is self-contained. However, the records in the PDB file must appear in a specific order defined by the worldwide Protein Data Bank (www.wwpdb.org) [16, 17]. Each record is divided into fixed-character length fields, with the first six-character field specifying the record type. Provided in Figure 4.1 is a partial PDB file showing some of the different types of records. All PDB files begin with a HEADER record and end with an END record. Following the HEADER is typically a section containing background information about the authors, the protein and the method used to determine the structure. A few specific records like TITLE and AUTHOR exist only once in a PDB file, while most other records may appear multiple times, often in groups or in the form of a list. The SEQRES records group identify the primary sequence (sequential arrangement) of amino acids making up the protein. The main structural information of the PDB file is represented as a list of consecutive ATOM records. Each ATOM record identifies the xyz-coordinate of a single atom in the protein. ATOM records also identify the name, type and symbol of the atom, as well as the amino acid name, chain identifier and sequence number the atom is a part of. In addition to identifying each atom and amino acid in the protein, the structure information in the *pdb* file also identifies the exact location of α-helices and β-sheets. HELIX and SHEET records specify the start and end amino acid residues that make up each α-helix and β-sheet and there is one record per each continuous helix/sheet (see Figure 4.1).

4.1.3 Downloading and Processing PDB Files

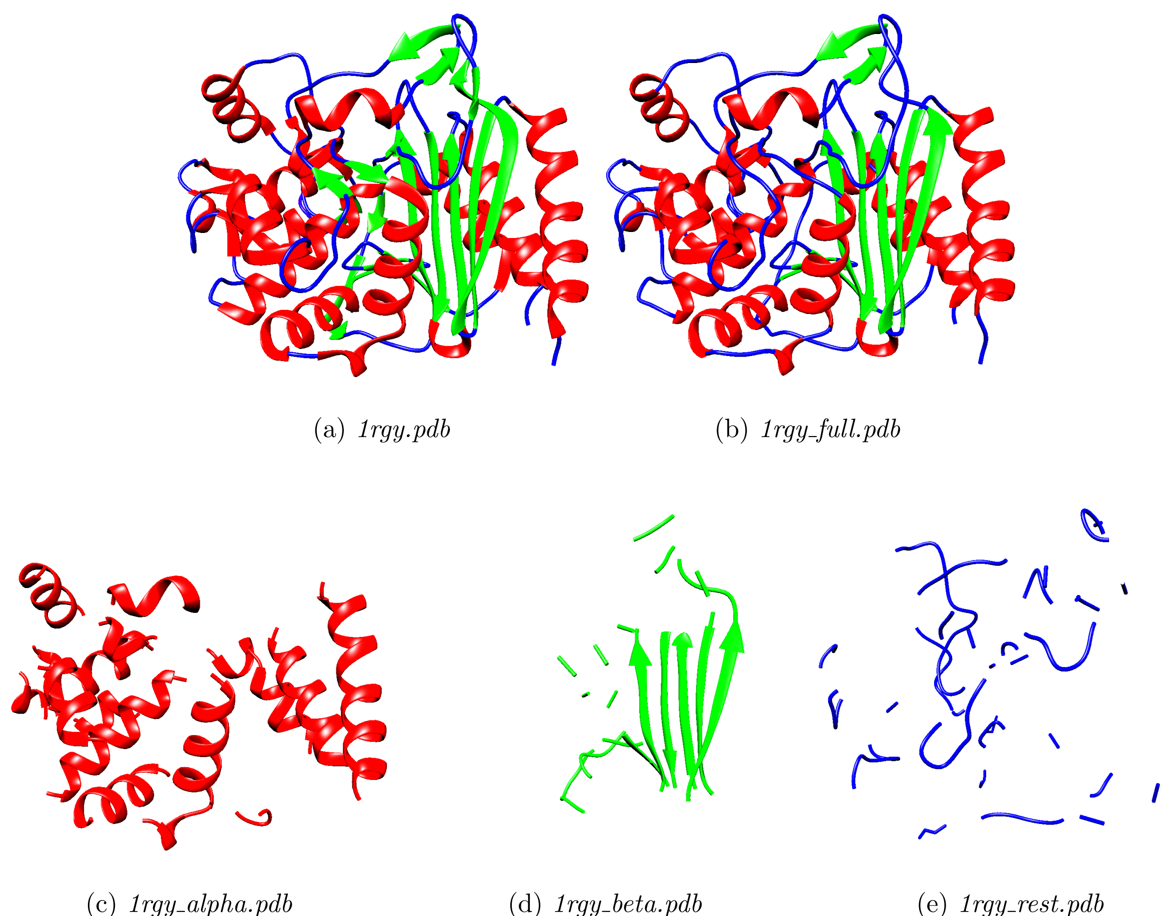
Downloading the structure files of the proteins was done with a Python script that utilized the FTP (File Transfer Protocol) Services provided by the PDB archive. The ***generate_pdb_files.py*** script was designed to read one-by-one the PDB IDs listed in the train

```
HEADER    LIGASE                                  01-DEC-97   1A0I
TITLE     ATP-DEPENDENT DNA LIGASE FROM BACTERIOPHAGE T7 COMPLEX WITH
...
AUTHOR    H.S.SUBRAMANYA,A.J.DOHERTY,S.R.ASHFORD,D.B.WIGLEY
...
REMARK   2 RESOLUTION.    2.60 ANGSTROMS.
...
REMARK 200  EXPERIMENT TYPE                : X-RAY DIFFRACTION
REMARK 200  DATE OF DATA COLLECTION        : AUG-95
REMARK 200  TEMPERATURE           (KELVIN) : 100
...
SEQRES   1 A  348  VAL ASN ILE LYS THR ASN PRO PHE LYS ALA VAL SER PHE
SEQRES   2 A  348  VAL GLU SER ALA ILE LYS LYS ALA LEU ASP ASN ALA GLY
SEQRES   3 A  348  TYR LEU ILE ALA GLU ILE LYS TYR ASP GLY VAL ARG GLY
...
HET    ATP  A   1      31
HETNAM     ATP ADENOSINE-5'-TRIPHOSPHATE
FORMUL   2  ATP    C10 H16 N5 O13 P3
FORMUL   3  HOH   *193(H2 O)
HELIX    1   1 GLU A   16  ALA A   26  1                                  11
HELIX    2   2 PRO A   61  LEU A   66  5                                   6
HELIX    3   3 VAL A   71  ASN A   78  1                                   8
HELIX    4   4 GLY A  104  ARG A  109  1                                   6
HELIX    5   5 LEU A  154  SER A  159  1                                   6
HELIX    6   6 THR A  167  TYR A  184  1                                  18
...
SHEET    1   A 5 SER A 195  VAL A 198  0
SHEET    2   A 5 LEU A  29  ILE A  33 -1  N  ALA A  31   O  TYR A 196
SHEET    3   A 5 LEU A 219  LYS A 222 -1  N  LYS A 222   O  ILE A  30
SHEET    4   A 5 GLY A 231  MET A 239 -1  N  MET A 239   O  LEU A 219
SHEET    5   A 5 PHE A   9  SER A  13  1  N  LYS A  10   O  GLY A 231
...
ATOM      1  N   VAL A   2     -10.053  -6.552  67.853  1.00 78.44           N
ATOM      2  CA  VAL A   2     -11.496  -6.584  67.619  1.00 79.00           C
ATOM      3  C   VAL A   2     -12.002  -7.947  68.142  1.00 79.82           C
ATOM      4  O   VAL A   2     -11.214  -8.539  68.885  1.00 79.76           O
ATOM      5  CB  VAL A   2     -11.850  -6.351  66.056  1.00 80.34           C
ATOM      6  CG1 VAL A   2     -10.934  -5.247  65.512  1.00 74.21           C
ATOM      7  CG2 VAL A   2     -11.726  -7.608  65.204  1.00 77.41           C
...
ATOM   2673  CB  ARG A 349       3.965 -24.934  31.484  1.00 77.17           C
ATOM   2674  CG  ARG A 349       3.392 -23.519  31.317  1.00 74.51           C
ATOM   2675  CD  ARG A 349       4.071 -22.762  32.401  1.00 71.90           C
ATOM   2676  NE  ARG A 349       3.950 -21.323  32.301  1.00 63.24           N
ATOM   2677  CZ  ARG A 349       4.960 -20.524  31.919  1.00 54.05           C
ATOM   2678  NH1 ARG A 349       6.167 -20.952  31.523  1.00 49.17           N
ATOM   2679  NH2 ARG A 349       4.677 -19.243  31.908  1.00 41.43           N
...
END
```

Figure 4.1: Examples of the types of records found in a PDB file (`1a0i.pdb`). Comprehensive PDB file format specification and version history is available from the wwPDB.

and test text files (see Appendix C.1.1 for source code). For each PDB ID, the program constructed corresponding FTP path and the Python command ***urlib.urlopen()*** was used to open the PDB structure file. The script collected all ATOM records and kept track of the the minimum and maximum values of the x, y, and z- coordinates as each PDB file was processed. The script used the minimum and maximum values of the x, y, and z- coordinates to filter out PDB IDs of large structures. Any structure file that had a difference between maximum and minimum coordinate value greater than 64 in any dimension was skipped and not processed further. The script generated two new text files with filtered lists of train and test PDB IDs of protein structures that had maximum dimensions less than or equal to 64. The need for this filtering is discussed in the next section. For those PDB IDs that were not filtered out, a separate function in the Python script parsed the HELIX and SHEET records in order to identify the position and sequence number of α-helices and β-sheets in each protein structure. The sequence numbers of the initial and terminal residues of the helices and sheets were stored and then used to separate ATOM records based on whether they made part of α-helix, β-sheet, or neither. The script wrote the ATOM records to separate *PDB* files. The ATOM records that were part of α-helices were written into one PDB file, and those that were part of a β-sheets into another file. ATOM records that were not part of either secondary structure element were written into third separate PDB file. The script wrote a fourth file with all ATOM records from the original PDB file. While writing the ATOM records to the separate files, the script modified the original xyz-coordinate values, shifting them so that all atoms are centered around the origin point (0,0,0). The shift amount in each direction was the half point between the minimum and maximum coordinate value in that direction. The need for this coordinate shift is also explained in the next section.

The four generated PDB files were saved in individual folders for each PDB ID, and each file name contained the PDB ID as well as a suffix ("_alpha", "_beta", "_rest", "_full") to indicate the type of ATOM records the file contained. The script-generated PDB files were validated by visualizing them in the UCSF Chimera program (see Figure 4.2). The type of representation of a protein structure in Figure 4.2 is called ribbon representation.



(a) *1rgy.pdb* (b) *1rgy_full.pdb*

(c) *1rgy_alpha.pdb* (d) *1rgy_beta.pdb* (e) *1rgy_rest.pdb*

Figure 4.2: *Separating the PDB file of an example protein (1rgy) based on secondary structure elements. Secondary structure elements are colored with different colors in Chimera: (red for α-helices, green for β-sheets, blue for turns/loops). Sub-figures illustrate the original and script-generated PDB files when opened and visualized in Chimera.*

(a) *Original 1rgy.pdb file as downloaded from PDB database.* ***(b)*** *1rgy_full.pdb file containing all ATOM records for the entire protein shifted and centered around origin point (0,0,0).*

(c) *1rgy_alpha.pdb file contains only the centered ATOM records that are part of α-helices.*

(d) *1rgy_beta.pdb file contains only the centered ATOM records that are part of β-sheets.*

(e) *1rgy_rest.pdb file contains centered ATOM records for protein loops and turns that do not make part of α-helices or β-sheets.*

Chimera was able to read and display the four generated files even though they contained only ATOM records and did not follow the exact specification of the PDB file format described in Section 4.1.2. Comparison between Figures 4.2a and 4.2b demonstrates that the protein structure is unaffected from the removal of non-ATOM records and the shifting of coordinates in the PDB file. Additionally, the figure confirms that an entire protein structure can be divided into segments containing only secondary structure elements. Minor differences in appearance between the sub-figures in Figure 4.2 can be attributed to the way Chimera draws the protein backbone as a ribbon. In ribbon view, α-helices are rendered as three dimensional spirals, β-sheets as flat arrows and turns/loops are displayed with thin lines, but their actual appearance on the screen varies slightly depending on the length of the segment. That is the reason why some short β-sheet segments in Figure 4.2d are displayed with thin lines.

Processing PDB Files in Parallel

The ***generate_pdb_files.py*** script was designed to run two concurrent processes in parallel in order to better utilize the multiple CPUs available on the host computer. The execution parallelism was achieved by using the ***multiprocessing*** Python package. The two Python processes performed the same functions but they were completely independent from each other and did not share same memory. One process worked on the PDB IDs from the test list file, while the other worked on processing the PDB files from the training list file. A timing experiment compared the execution time of the sequential and parallel versions of the script. Processing 20 PDB IDs took $16.05\,\text{s} \pm 0.09\,\text{s}$ for the sequential version of the script, and only $8.24\,\text{s} \pm 0.12\,\text{s}$ for the parallel version, a speed up of 1.95 times.

4.1.4 Simulating Electron Density Maps

There exist image processing software tools that can simulate 3D electron density maps from solved 3D atomic structures. EMAN2 is one such software package specifically designed for processing electron microscopy data [70]. Using EMAN2 and the function "*pdb2mrc*", the

segmented PDB structures were simulated to fixed-size 3D electron density maps at specific resolution. The "*pdb2mrc*" function takes a resolution value (measured in Angstroms) and models each atom in the PDB model with a 3D Gaussian distribution. The resolution is the reciprocal of the half-width of the Gaussian distribution in Fourier space. The electron density value for each atom in the corresponding position in the density map is calculated according to the Gaussian equation, $n_e e^{-(\frac{r}{k})^2}$, where n_e is the atomic number, r is the distance from the center of the atom, and k is a Fourier coefficient corresponding to a filter with half-width of 1/resolution. The EMAN2 function writes the 3D electron density maps as MRC files. MRC files are binary files widely used in the field of electron microscopy. MRC files contain a datablock of three-dimensional array of intensity values that represent the protein map, and a header with information and statistics about the data. The full specification of MRC file format is published by Cheng et al [24].

Each generated PDB file from the first step was inputted to the "*pdb2mrc*" function, and a simulated 3D electron density map was outputted as an"MRC" file. The "*pdb2mrc*" function also allows to specify the three-dimensional size of the simulated map as well as the size of each voxel. The simulation box was set to constant size of 64x64x64 voxels while the default 1Å/voxel sampling rate was used. The box size argument controls the size of the simulated map (dimensions of the datablock) and so it was chosen to be large enough in order to fit the entirety of most protein structures that were collected. However, it was observed that a large box size alone did not guarantee that no parts of the proteins extend beyond the edges of the simulation box. The default behavior of "pdb2mrc" function is to place the origin of the protein at the middle of the box. This meant that any atom coordinate in the PDB file greater than 32 or smaller than -32 would have electron density outside the simulation box. And because the origin of each protein is arbitrarily defined in the PDB files, an extra step was needed to shift the XYZ coordinates and center them around (0,0,0). This extra step was added during the collection and separation of the PDB files in the first step.

Figure 4.3 illustrates two simulated maps at 6Å resolution and box size of 64x64x64 vox-

els, but one was from the unaltered PDB file (4yml.pdb) and the other from the processed PDB file with shifted coordinates (4yml_full.pdb). The simulated electron density is cut out for the map generated from the original PDB file. The coordinate shift between the two structures is demonstrated by comparing the coordinate of the Cα atom of in the 105th and 60th residues: in the unaltered PDB file, the Cα atoms are at $(-43.992, 4.709, -2.011)$ and $(-32.722, -17.408, -16.182)$. Because the x-coordinate value is less than -32, those atoms lay outside the simulation box and the protein map is abruptly cut-out at the edge of the simulation box. The positions of the same two atoms after the shift are at $(-20.673, 11.288, 14.332)$ and $(-9.404, -10.829, 0.161)$ and within the simulation box. This shifting of coordinates ensured that the density maps of the entire proteins are contained within the simulation box. The coordinate shift alone, however, could not guarantee that density maps are not cut out. Some proteins structures were just too big to fit in the 64x64x64 box. That is why when the PDB files were processed in the previous step, a logic in the script filtered out structures spanning more than 64 voxels in any dimension. Figure 4.4 illustrates one such large protein that could not fit in the constant-size simulation box even with a coordinate shift and was therefore not included in the final training or testing data sets. After the removal of the PDB IDs of such large structures, the training set was reduced from 20159 to 14194 structures while 3597 remained from the originally collected 5040 structures in the test dataset. The 80%-20% training-test ratio was in effect unchanged after the removal of large structures.

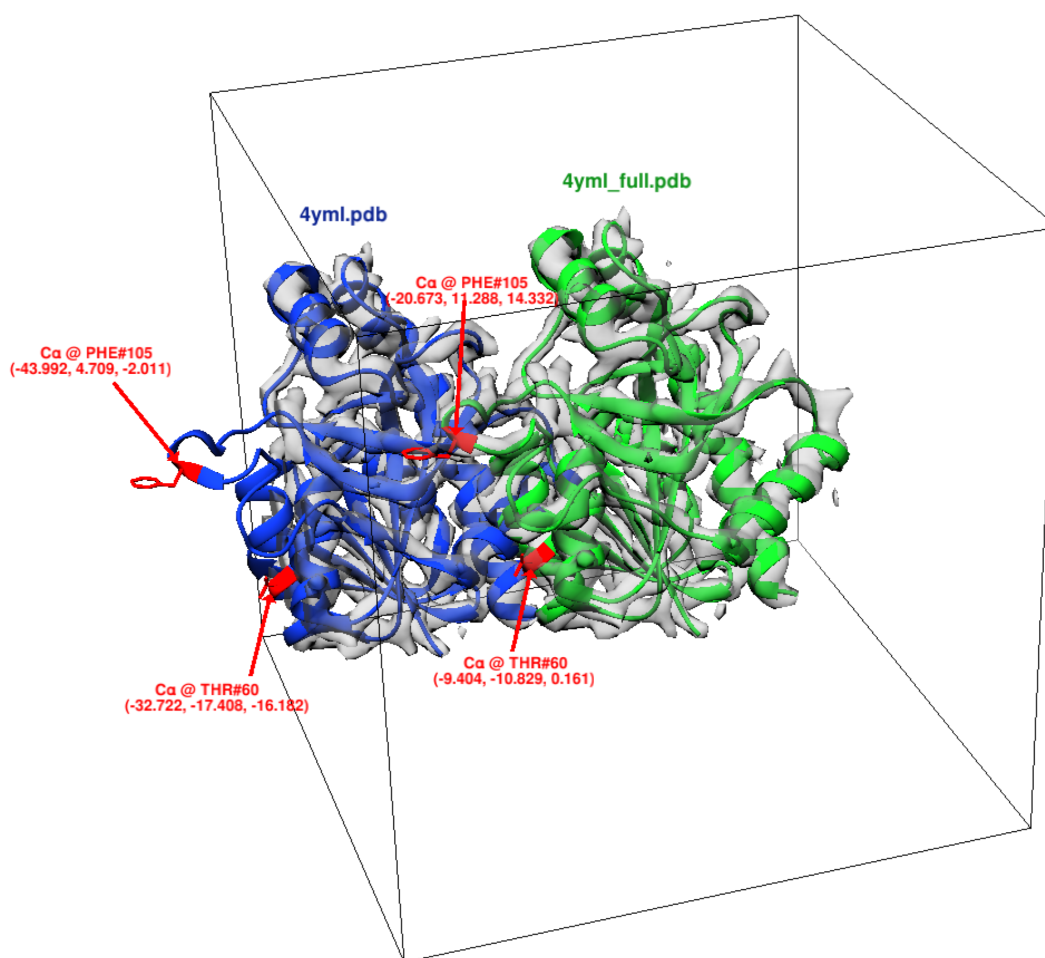


Figure 4.3: *Two EMAN2-simulated maps of the same protein, one generated from the original PDB file (4yml.pdb) and the other from the PDB file with shifted coordinates (4yml_full.pdb), demonstrate the issue of cut-out density maps and how shifting the atom coordinates relative to the (0,0,0) center of the simulation box fixes the issue. The simulated maps are displayed over the ribbon structures (blue for 4yml.pdb, green for 4yml_full.pdb) in Chimera with the outline cube showing the boundaries of the 64x64x64-voxel simulation box. Highlighted in red are two Cα atoms which are outside of the simulation box on the original structure (4yml.pdb) but inside the box after the shift.*

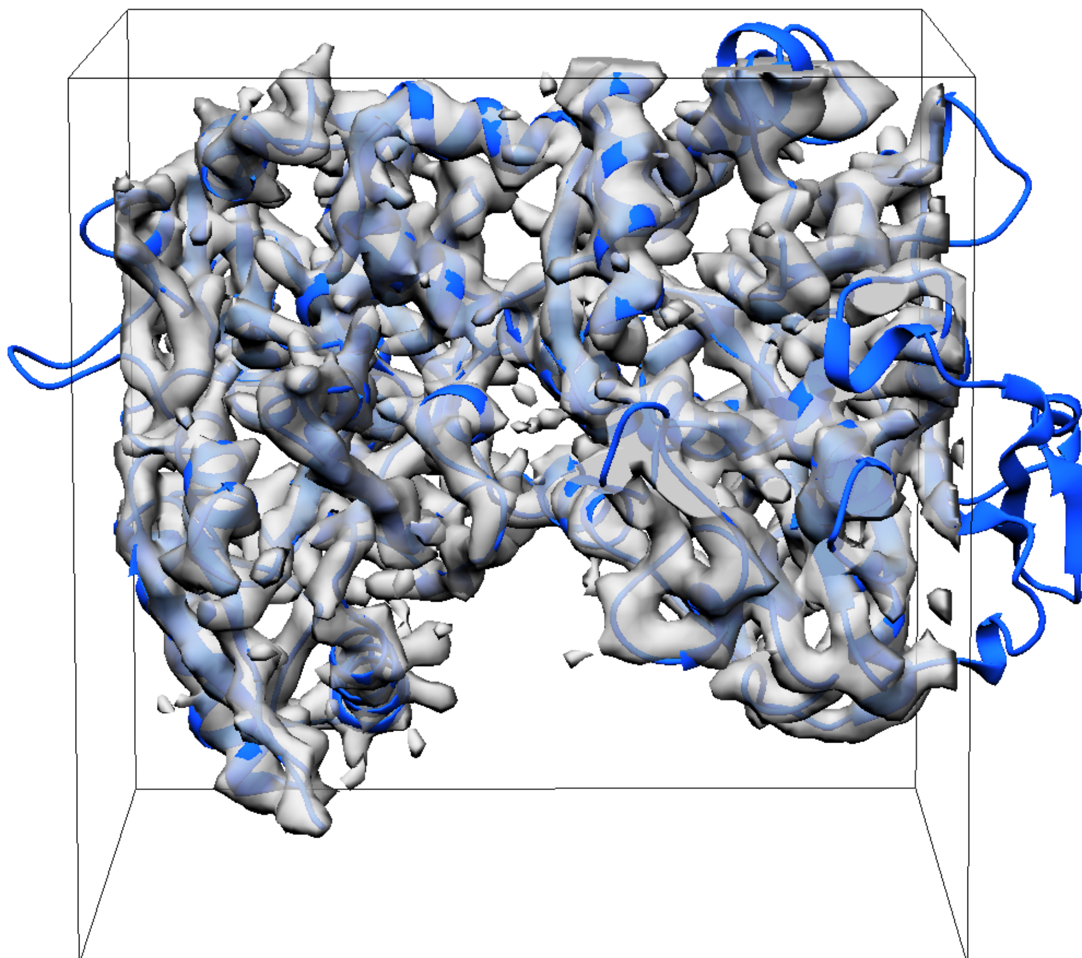


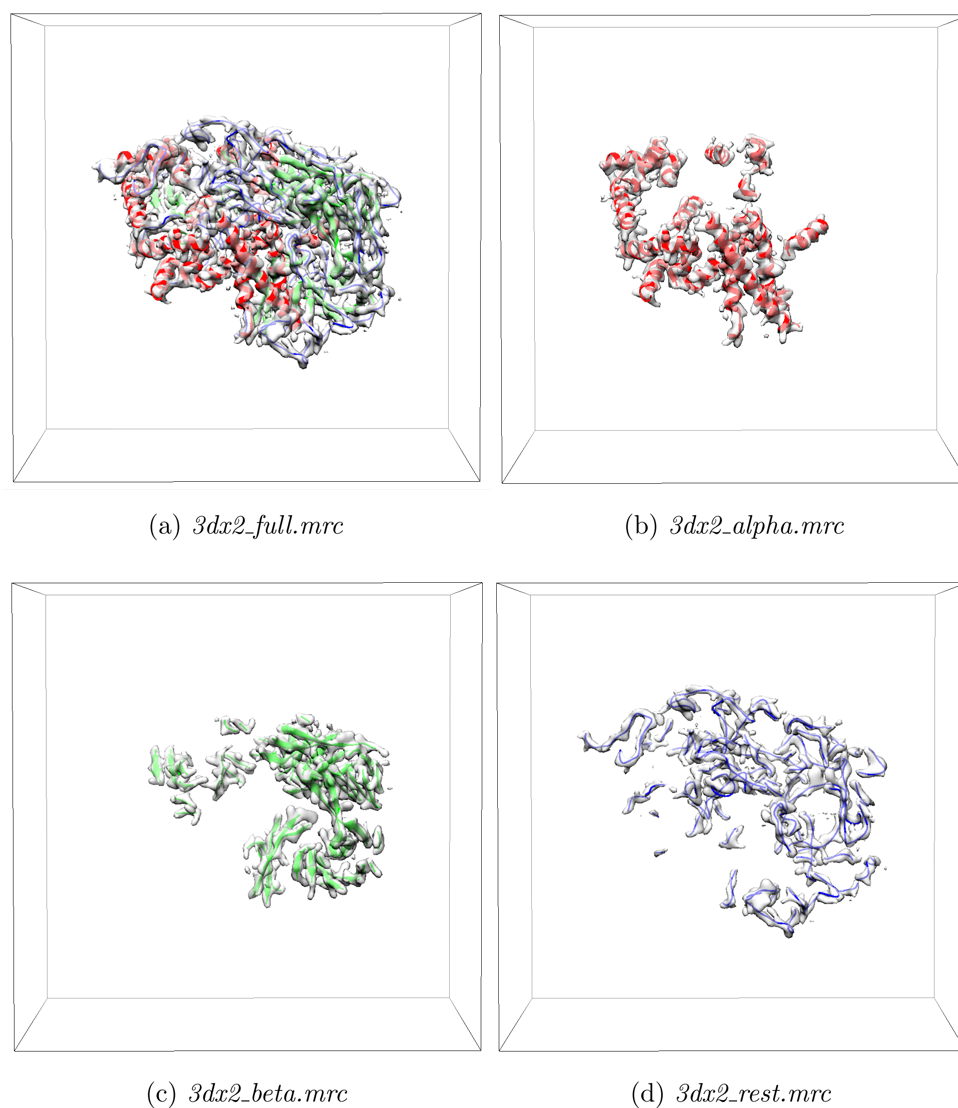
Figure 4.4: *Example of a protein structure (1b7z) which is too big to fit in the constant size simulation box even after shifting atom coordinates to the box center. Such large structures with any dimension greater than 64 were filtered out and not included in the training and test data sets.*

A standalone Python script generated MRC files from the 14194 training and 3597 test PDB files (see Appendix C.1.2 for source code). The ***generate_mrc_files.py*** script read the PDB IDs from the filtered list files (***test_pdbs_list.txt*** and ***train_pdbs_list.txt***) to generate the paths to the different PDB and MRC files, and used those paths as arguments

to the “pdb2mrc” function. The “pdb2mrc” function was called as a Python executable in a separate shell via the ***os.system()*** command. Additional arguments to the “pdb2mrc” command included the resolution, the box size and the voxel spacing. Four different resolution values in the medium resolution range were chosen for the density map simulations: 5Å, 6Å, 7Å, 8Å. Thus, a single PDB file was simulated to 4 different resolutions. The script stored the MRC files for the different resolutions in different subfolders, each named with the corresponding resolution value. Similar to the naming convention used for the PDB files, the script also generated a name for each MRC map file using the “_alpha”, “_beta”, “_rest”, “_full” suffices to indicate the PDB file from which it was simulated and the secondary structure it contained.

Visualization of Segmented Simulated Maps

The full simulated electron density map and the three partial maps of an example protein (PDB ID 3dx2) were opened as MRC files in Chimera and visualized on top of the PDB files from which they were generated (see Figure 4.5). EMAN2’s “pdb2mrc” function adequately modeled the electron density cloud of only the atoms specified in the partial PDB files. The partial alpha, beta and rest maps were generated by simulating the electron distribution of only the atoms making up α-helices, β-sheets and turns/loops, respectively. In addition, the “pdb2mrc” function created the partial maps with the same relative coordinates within the simulation box by preserving the coordinates of the atoms specified in the PDB files. All four maps contain the same number of voxels (64 in each dimension), but the density values are approximately zero for the voxels that did not correspond to the xyz-location of an atom. It can also be seen from Figure 4.5 that the electron densities of the three secondary structure elements appear differently. α-helix structures have tubular shape, while the β-strands have flatter and thinner tubular shapes adjacent to and nearly parallel to other similar tubes of density. Loops and turns are often made up of smaller and not as electron-rich residues, therefore they have reduced electron densities around them.



(a) *3dx2_full.mrc*

(b) *3dx2_alpha.mrc*

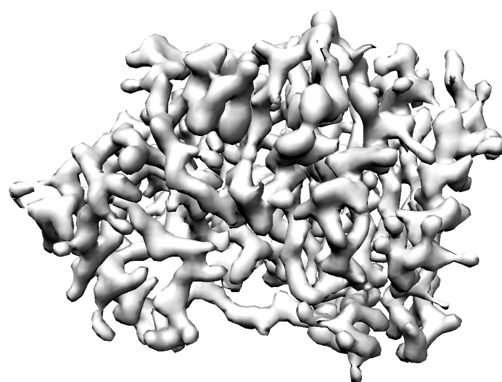
(c) *3dx2_beta.mrc*

(d) *3dx2_rest.mrc*

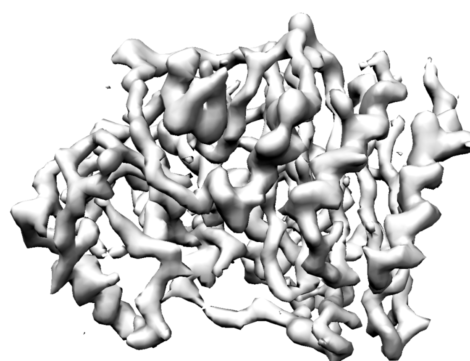
Figure 4.5: *Visualization of the 6Å-resolution electron density maps of an example protein (3dx2) simulated using EMAN2 software package and the function "pdb2mrc". Sub-figures illustrate the 4 generated MRC maps in Chimera, superimposed over the PDB structures from which they were generated.* ***(a)*** *Density map of the entire protein was generated by simulating the electron distribution of each atom listed in the PDB file.* ***(b)*** *Electron density map of the atoms making up only alpha-helices in the protein.* ***(c)*** *Electron density map of only the beta-sheets.* ***(d)*** *Electron density map of only the loop and turn regions.*

Map Simulation at Different Resolutions

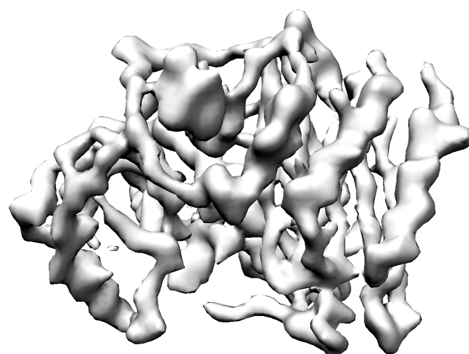
The electron density shapes of secondary structure elements appear a little differently based on the simulated resolution of the map (see Figure 4.6). At a resolution of 5Å, electron density of most side chains can be seen and the tubular shapes of alpha-helices and beta-sheets are obscured. Side chains are not resolved at lower than 6Å resolutions and the tubular shape of the alpha-helix backbone is especially pronounced. At lower resolutions, the adjacent beta-strands appear to merge together as a flat "sheet" of density, while loops have very thin electron density.



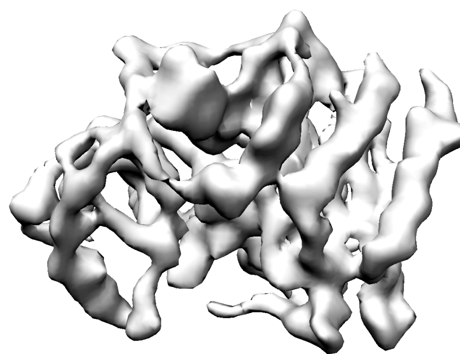
(a) *5Å resolution map of 1rgy*



(b) *6Å resolution map of 1rgy*



(c) *7Å resolution map of 1rgy*



(d) *8Å resolution map of 1rgy*

Figure 4.6: Hello World. **(a)** Hello World. **(b)** Hello World.

Generating MRC Files in Parallel

Similar to ***generate_pdb_files.py*** script, ***generate_mrc_files.py*** script was executed in parallel for the train and test data sets using the ***multiprocessing*** Python package. The parallel version of the script was 1.93 times faster than the sequential execution, taking $153.35\,\mathrm{s} \pm 0.91\,\mathrm{s}$ versus $295.80\,\mathrm{s} \pm 0.91\,\mathrm{s}$ to simulate 20 maps at the four different resolutions.

4.1.5 Generating Secondary Structure Labels

The next step in the data collection and preparation phase was the creation of labels for each simulated map. In order to train a neural network in supervised learning setting to segment density maps based on secondary structures, each input density map needs to be associated with a labeled map. Labeled maps need to specify the type of secondary structure element at each voxel location. For that purpose, a discrete voxel label was assigned to identify the secondary structure element for the corresponding location in the protein. But apart from recognizing each secondary structure element within the protein, the voxel labels also delineate the boundaries of the protein. Therefore, in a protein density map, a voxel can either be empty (i.e. background) or contain electron density from one of the three secondary structure elements. As a result, four values were used when assigning class labels at each voxel: 0 for background,1 for alpha-helix, 2 for beta-sheet and 3 for loops/turns. A Python script generated the voxel labels for the 17791 simulated maps in the training and test datasets (see Appendix C.1.3 for source code). For each protein PDB ID, the script randomly chose one resolution value from the four resolutions used for the simulation, constructed the directory paths to the simulated maps and loaded the MRC files from disk. Using the *mrcfile* Python library [20], the script extracted the three dimensional data arrays from the three segmented alpha-, beta-, rest- MRC files. The electron density values in the data arrays were first min-max normalized between 0 and 1, and then values in the 10th percentile were set to zero. The non-zero elements in the arrays corresponded to locations of secondary structure elements and the zero-valued voxels corresponded to non-structure

or background. The extracted arrays from the three maps and their values were then used to determine the discrete label (0,1,2, or 3) for each voxel. The script compared the values from the three arrays at each element and assigned a label value at the same index in a new, zero-initialized 64x64x64-dimensional array. The label value (1, 2, or 3) was chosen based on which of the three arrays contained the greatest, non-zero density value at that index. The greater-than comparison was needed because voxels located at the boundaries between two secondary structures contained non-zero electron densities in more than one of the source arrays.

Packaging Data into HDF5 File

The labeling algorithm was included in the same script that packaged the data arrays from MRC files into data files that could be stored on disk and thus eliminate the need to repeat this step. Pairs of the generated three-dimensional arrays with discrete labels and the data arrays from the full MRC file were stored and organized into hierarchical files to facilitate input to neural networks. The open source *hdf5* file format was chosen because of its compatibility with a range of computational platforms and capability to store large heterogeneous data types in index-based, contiguous disk blocks for fast input-output performance [75]. The Python script created two hdf5 files using the *h5py* package. One file (***train_data.hdf5***) held 14194 training maps and labels arrays and the other file (***test_data.hdf5***) stored 3597 map and label array pairs for testing. In addition to the map and labels arrays, the two *hdf5* files also stored the PDB IDs as well as the resolution values for each map entry. The test and train hdf5 files had different lengths but had otherwise identical structures composed of 4 datasets (see Figure 4.7). Similarly to the other data generation scripts, the labeling and hdf5-file creation script spawned two independent processes to work on the train and test datasets in parallel. However, it was observed that the parallel version of the script was not faster than the sequential version ($4.99\,\mathrm{s} \pm 1.70\,\mathrm{s}$ versus $4.61\,\mathrm{s} \pm 1.13\,\mathrm{s}$) when tested on a sample of 20 maps. This was likely due to the extensive data shuffle and disk write operations.

After the two hdf5 files were created, an auxiliary script parsed the files to confirm the integrity of the data. The files were confirmed to contain an almost equal number of maps from the 4 different resolutions (see Table 4.1).

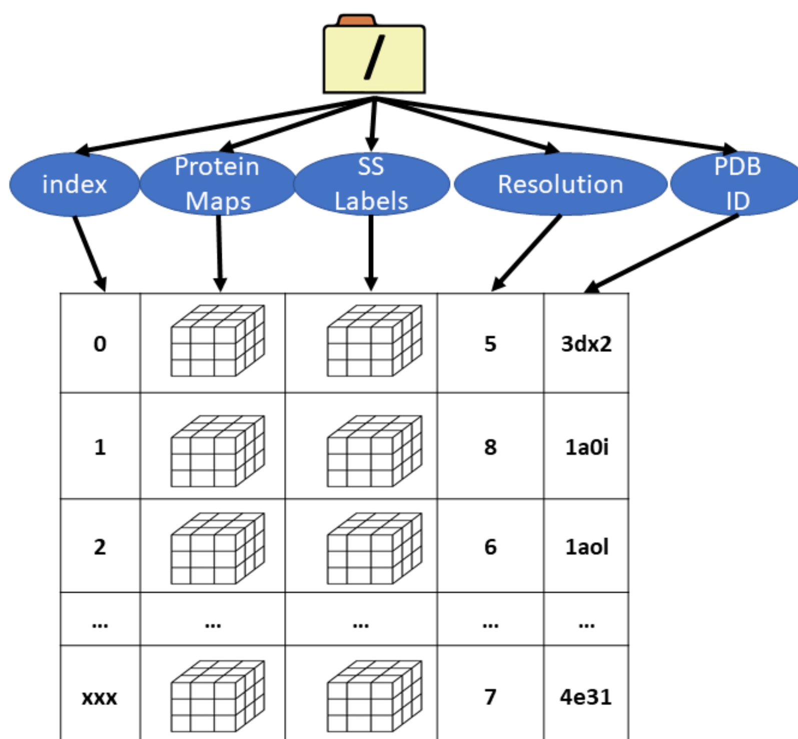


Figure 4.7: *The two hdf5 file containers held 4 indexed datasets of heterogeneous data objects. The resolution dataset stored values as 8-bit unsigned integers, the PDB ID dataset stored 4-character strings, while the protein maps and SS labels datasets stored three-dimensional arrays of 32-bit floating-point values. The two files had different lengths: train_data.hdf5 contained 14194 entries, while test_data.hdf5 contained 3597 entries. The sizes of the two files were 29 Gigabytes and 7.3 Gigabytes, respectively, when stored on disk.*

Resolution	*train_data.hdf5*	*test_data.hdf5*	Total
5Å	3582	894	**4476**
6Å	3574	868	**4442**
7Å	3508	919	**4427**
8Å	3530	916	**4446**
Total	**14194**	**3597**	**17791**

Table 4.1: Protein density map counts per resolution in train and test data sets for secondary structure segmentation experiment.

4.2 The Proposed Deep Model

4.2.1 Model Architecture

Inspired by Ronneberger, Fischer and Brox's original U-Net architecture [63] for segmentation of two-dimensional biomedical images, a three-dimensional model with U-Net style contracting and expanding paths is designed for protein secondary structure segmentation. Original two-dimensional U-Net convolution/deconvolution layers are replaced with three dimensional layers appropriate for segmenting 3D protein density maps. The numbers and sizes of filters in all convolutional and deconvolutional layers are adjusted based on a constant input layer with size 64x64x64-voxel cubes. Final probability mapping layers of the model are also modified to output four three-dimensional probability arrays, one for each secondary structure class, with the same size as the input map. The proposed 3D U-Net network architecture is illustrated in Figure 4.8.

The contracting part of the 3D U-Net model consists of repeated pairs of convolutional layers followed by a max pooling layer. The sizes of the convolutional filters remain the same (4x4x4) throughout, but the number of filters increases from 32 in the beginning layers to 512 at the deepest layer in the contracting half of the network. Zero padding is added to

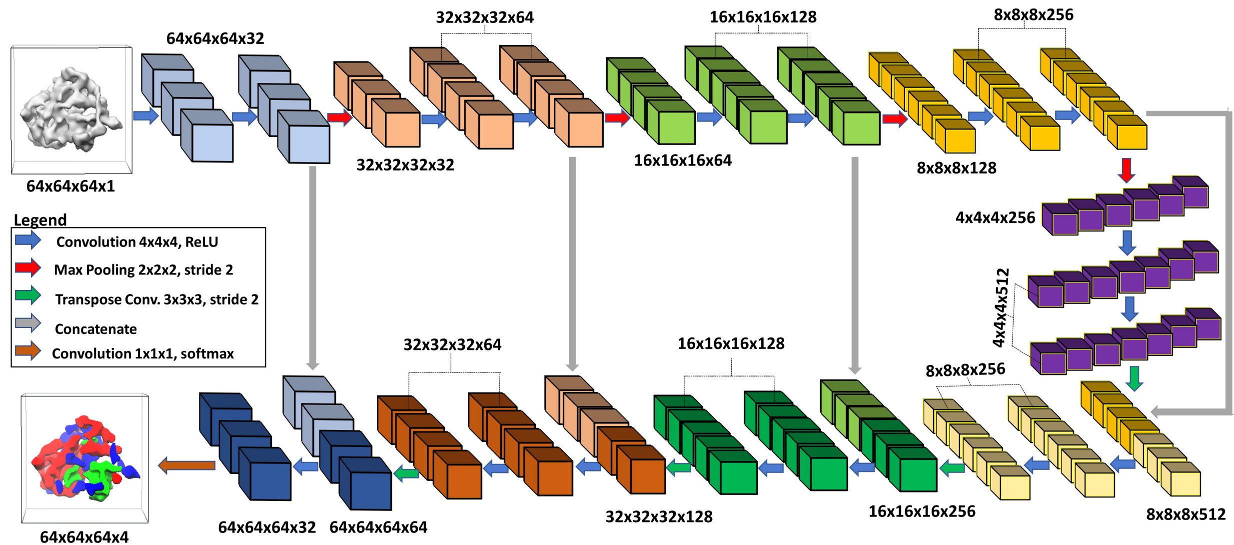


Figure 4.8: *3D U-Net Architecture showing the symmetrical convolution and deconvolution paths. The input 3D density map is processed in the direction of the arrows which are color coded for the different operations listed in the legend. The dimensions of the data at each layer is also shown above and below each layer. The network outputs 4 probability 64x64x64x maps, one for each possible class label. The label at a given voxel is determined by taking the maximum of the four probabilities.*

each convolution to preserve the dimensions of the output and ReLU activation function is applied to the output as a non-linear transformation. Volume dimensions are reduced by half after every two consecutive convolutional layers using max pooling layers with size 2x2x2 and stride 2. In the expanding arm of the network, transpose convolutions (deconvolutions) are used to upsample the volume. The number of filters in each transpose convolutional layer decreases from 256 to 32 before the final output layer. Sizes of those filters are kept constant (3x3x3) and the deconvolution operation is performed with stride 2 which causes the volume dimensions to double. The output of each transpose convolution is concatenated via skip connections with filters from upper layers in the contracting path. That concatenated output is processed by a pair of regular convolutional layers which also gradually reduce the number

of filters from 256 to 32. The final layer contains four 1x1x1 filters to represent each of the four possible classes (background, alpha, beta, turns/loops). Softmax activation function is used to turn the four 64x64x64 outputs into four probabilities in the range (0-1) whose total sum for each voxel is equal to 1. In order to get a single map with discrete labels, the class with maximum probability is assigned to the corresponding voxel. Each voxel in the single output 64x64x64 cube is labeled as either 0 for background, 1 for alpha-helix, 2 for beta-sheet, and 3 for rest/loop.

4.2.2 Model Implementation

This custom 3D U-Net model is constructed using Python 3.5.5 interface of Google's Machine Learning Framework called TensorFlow r1.12.0. Machine learning models are created in TensorFlow as dataflow graphs, where nodes in the graph represent mathematical operations, and edges are the data (n-dimensional arrays called tensors) consumed or produced by the computations. The 3D U-Net model logic is implemented as a dataflow graph, source code for which is provided in Appendix C.2.1. The model function takes as input raw data, known labels associated with that data and other parameters, and returns operations that can be used to perform training, evaluation, or prediction. This specific signature of the model function allows it to be encapsulated in an Estimator object. tf.Estimator is TensorFlow's latest high-level API for building complete machine learning models. Estimators provide intuitive methods for training and evaluating a model whereas low-level details such as building and initializing the model graph from the model function and providing a TensorFlow session environment for executing operations and evaluating tensors are abstracted out. Estimators also provide an easy mechanism to supply data for training or testing the model. Data input to Estimator is separate from the model which makes it easy to perform experiments with different data sets. Specifically, train and evaluate methods of the Estimator class accept custom input functions as source of the data. The next section describes the training and evaluation input pipelines for supplying protein density maps to the 3D U-Net Estimator.

4.2.3 Data Input Pipeline

A data input pipeline is specifically designed to feed 3D data arrays of electron density maps and their labels from the generated HDF5 files to the neural network. Data samples are pipelined into the network in batches using parallel extract-transform-load (ETL) process. The first two steps (Extract and Transform) require the use of CPU cores to read data from disk and perform shuffling and batching. The third step in the data pipeline is loading batched data onto the GPU for training or evaluating the model. TensorFlow's tf.data API is used to construct an efficient ETL input pipeline that optimizes CPU and GPU processing (see source code for the data pipeline in Appendix C.2.2). A Python generator object is first defined for accessing and reading data from the HDF5 files containing protein maps and their secondary structure labels. The constructor of the Generator class takes a file path to an HDF5 file as an argument and is able to read and return data from that file on demand. The generator behaves like an iterator but with improved and efficient performance due to lazy (on demand) generation. On-the-fly generation of data is particularly useful in this case due to the inability to load entire training and test datasets into memory. In addition, the generator class opens an HDF5 file once and keeps it open while remembering its position after each iteration call, which allows for reading next batches of data without the need to re-read data from the beginning each time. Using the tf.data.Dataset.from_generator() construction method, the Generator object is used to create TensorFlow dataset objects in two input functions, one for training and one for evaluation. The generator in the train input function is given the path to *train_data.hdf5* file while the evaluation input function uses *test_data.hdf5* as the source file. The two input functions produce a pipeline of batches of data elements instead of single elements. tf.data.Dataset.batch function combines consecutive elements into batches of size set by a hyperparameter. tf.data.Dataset.prefetch function buffers elements from the dataset making them available to load to the GPU as soon as GPU becomes available. Prefetching decouples CPU extract and transform tasks from the GPU so that while the GPU is training on the *i-th* iteration, the CPU is preparing data batch

for the *(i+1)-th* iteration. Only the train input function contains an additional call to repeat training data indefinitely but to randomize sequencing order of samples in the set at each repetition. The repeat and shuffle operation ensures good performance and strong ordering guarantees that all samples in the training data are used about the same number of times during training. However, the ***shuffle_and_repeat()*** function blurs the epoch boundaries as some maps could be repeated before other maps are trained on.

4.2.4 Tversky Loss

Deep learning models learn a mapping function between input and output by means of minimizing a loss function. In the case of voxelwise segmentation, a loss function needs to examine each voxel individually and compare the class predictions to the provided target labels. The results and performance of the model rely on a suitable choice of loss function and not only on the network architecture. This is particularly important when there is a strong class imbalance in the data. For the segmentation of secondary structure elements in cryo-electron density maps, the background/void voxels are the most prevalent in the data and heavily outweigh the number of alpha-, beta-, and turn/loop- voxels. Therefore, a voxelwise cross entropy loss function that averages over all voxels is not a suitable choice because it would assert equal learning to each voxel in the density map. To account for the voxel per class imbalance, this research adapts a loss function based on Tversky index originally proposed for binary classification by Salehi, Erdogmus and Gholipour [65]. Tversky index is an asymmetric similarity measure between predicted and ground truth labels. The Tversky index for a given class can be calculated using the following generalized equation of Dice (F1) Similarity Coefficient (DSC):

$$TI_c = \frac{\sum_{i=1}^{N} v_{ic} g_{ic}}{\sum_{i=1}^{N} v_{ic} g_{ic} + \alpha \sum_{i=1}^{N} v_{i\bar{c}} g_{ic} + \beta \sum_{i=1}^{N} v_{ic} g_{i\bar{c}}} \tag{4.1}$$

where the sums are over all N voxels, v_{ic} is the probability of voxel i be of voxel class c, $v_{i\bar{c}}$ is the probability voxel i is of non-class $\bar{c}$, g_{ic} is the ground truth probability of class label c,

and $g_{i\bar{c}}$ is the ground truth probability of label being non-class $\bar{c}$. The α and β parameters in the denominator of the Tversky index, not to be confused with or having any relationship to α-helix and β-sheets in the secondary structure segmentation problem, are regularization parameters that can be adjusted to measure asymmetric similarity by balancing false positive (FP) and false negative (FN) predictions. When $\alpha = \beta = 0.5$, Tversky index equals DSC, and when $\alpha = \beta = 1$, Equation 4.1 produces Tanimoto/Jaccard index which also equals to Intersection over Union (IoU) similarity measure [65].

The Tversky index in Equation 4.1 can be extended for multiple classes by summing the Tversky indexes calculated individually for each class. This strategy transforms the multi-class problem into multiple binary problems using *One-vs-All* strategy and it works if there is only one correct class per voxel. The equation for multi-class Tversky index is shown in the following formulation:

$$TI_{multiclass} = \sum_{c=1}^{C} TI_c \tag{4.2}$$

where C is the number of classes greater than two. Adapting the multi-class Tversky index to a loss function can then be viewed as minimization of the overlap between prediction and ground truth for each class:

$$TL = \sum_{c} (1 - TI_c) \tag{4.3}$$

This version of Tversky loss function does not weigh individual voxels equally but rather each class category is weighted equally. Each class no matter how many voxels are in that class can reduce the loss value by maximum of 1 unit. The maximum value of Equation 4.3 for N-class problem is N and the minimum is 0. The Tversky loss function is implemented for the protein secondary structure segmentation problem with four classes and is used to train the proposed 3D U-Net model (see Appendix C.2.3 for source code).

4.2.5 Model Training

The 3D U-Net model function, the data input pipeline and the Tversky loss function are wrapped together in a training routine which defines additional hyperparameters required for training the model. The training routine is defined in a separate Python program that also allows user input for several of the training parameters (see Appendix C.2.4 for source code). After trying three different optimization algorithms (Gradient Descent, Adagrad, and Adam) and different learning rates, it was determined that the 3D U-Net model best optimized the multi-class Tversky loss function with default initialized Adam optimizer and learning rate of 0.0001. At the beginning of training, the model parameters in each network layer were initialized at a default random state. Training batch size was set to 20 and the model was trained for 15000 iterations corresponding approximately to 21 epochs. At each iteration, the Adam optimizer updated the parameters of the model in the direction of decreasing gradient of the Tversky loss function. The value of the Tversky loss function was recorded every 500 training iterations by evaluating the 3597 maps in the test dataset. Also monitored every 500 iterations was the overall voxelwise validation accuracy which is defined as a fraction of total number of correctly classified voxels in all four classes divided by total number of voxels. Evaluation during training was done by passing an *InMemoryEvaluatorHook* object to the tf.Estimator's train method. 15000 train iterations with evaluation at every 500 iterations took 11.5 hours on a Linux node with a quad-core Intel Xeon CPU E5-2623 v4 @2.60GHz equipped with 64 GB RAM and NVIDIA GeForce GTX 1080 GPU with 12 GB memory.

4.3 Results

4.3.1 Evaluation Metrics

The following result sections report several different evaluation metrics in order to critically evaluate the performance of the proposed segmentation model. Because secondary structure segmentation can be viewed as four-class classification for each voxel, confusion matrices are used to determine true positive (TP), true negative (TN), false positive (FP) and false

negative (FN) counts for each class. The determination is made by comparing the voxels predicted by the classifier and the true voxel labels. The metrics are then calculated for each class according to these equations:

$$PPV(Precision) = \frac{TP}{TP + FP}$$
$$Recall(Sensitivity) = \frac{TP}{TP + FN}$$
$$Specificity(Selectivity) = \frac{TN}{TN + FP}$$
$$NPV = \frac{TN}{TN + FN}$$
$$DSC(F_1 score) = \frac{2TP}{2TP + FN + FP}$$
$$F_2 score = \frac{5TP}{5TP + 4FN + FP}$$

To compute combined metrics for all four classes, two averaging methods were considered: macro- and micro-averaging. Macro-averaging method first computes the metric independently for each class and then takes the arithmetic average over all classes, hence weighing each class equally. For example, macro-averaged precision for the four classes can be calculated using the following equation:

$$Precision_{macro-average} = \frac{Precision_{\alpha} + Precision_{\beta} + Precision_{turn/loop} + Precision_{void}}{4}$$

Micro-averaging on the other hand aggregates individual TP, TN, FP, FN contributions from all classes and use those aggregated values to compute an average combined metric. Therefore, micro-averaged precision can be calculated using the following equation:

$$Precision_{micro-average} = \frac{TP_{\alpha} + TP_{\beta} + TP_{turn/loop} + TP_{void}}{(TP_{\alpha} + TP_{\beta} + TP_{turn/loop} + TP_{void}) + (FP_{\alpha} + FP_{\beta} + FP_{turn/loop} + FP_{void})}$$

With strong class imbalance, micro-averaged statistics are often better indicators for the performance of the model. However, in a multi-class, *one-vs-all* setting, micro-averaged precision and recall metrics (and therefore F_1 and F_2 scores) are always the same. This is

because a false positive prediction with respect to one class is considered a false negative prediction with respect to the other classes, and the sum of all FP predictions equal the sum of all FN predictions which equal the sum of all false instances (see Proof in Appendix A.1).

4.3.2 Configuring the Parameters of Tversky Loss Function

To evaluate the effects of the Tversky Loss function on the training of the network, nine different instances of the 3D U-Net model were trained on the training dataset with different values of alpha and beta Tversky loss regularization parameters. The α parameter was varied from 0.1 to 0.9 while simultaneously varying the β parameter from 0.9 to 0.1, keeping the sum $\alpha+\beta$ equal 1 in all cases. Adam optimizer and learning rate of 0.0001 were used to train all nine models for 15000 iterations. All other training parameters (e.g batch size) were kept the same and no modifications to network parameters were made from one model to another. Each model was evaluated every 500 training steps on the test dataset by recording the value of the loss function and the total validation accuracy. Figure 4.9 shows the optimization of the loss function during training for each set of Tversky loss regularization parameters. In all cases, the loss function was saturated at around 15000 iterations, however the converged values were slightly different for each pair of α and β values. The lowest values of the loss function at each training step were achieved when $\alpha = 0.9$ and $\beta = 0.1$ (purple line in Figure 4.9). There were no significant differences between the converged values at lower α (higher β) values. However, the general trend from Figure 4.9 is that the loss function converged to a lower value with increasing α parameter value.

Total validation accuracy was also recorded for each model trained with varying α and β Tversky parameters. The graph in Figure 4.10 shows validation accuracy increasing as training progressed for all models and saturating at around 15000 iterations. However, the plots show that the values of the Tversky regularization parameters affected the performance accuracy of the models. Models trained with higher α values tended to converge to lower validation accuracy values and models trained with higher β (lower α) values were observed to produce higher total accuracy results.

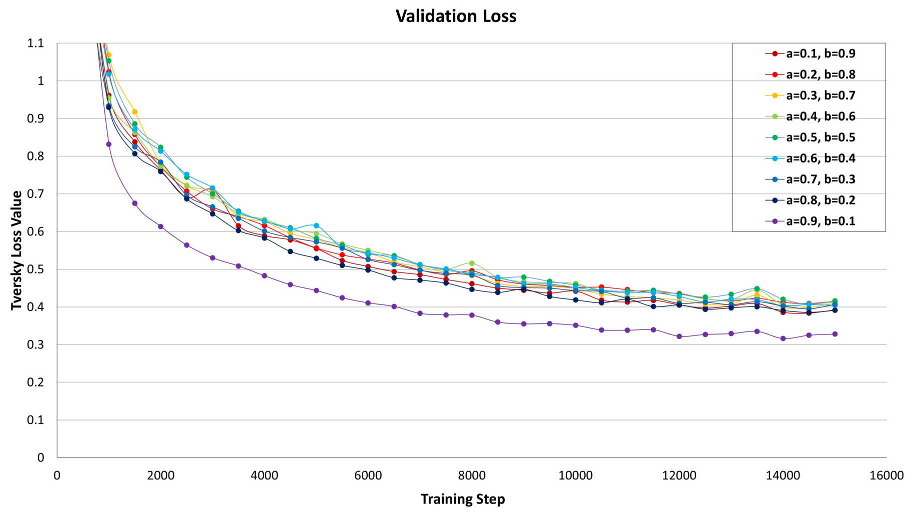


Figure 4.9: *Evaluating the effects of the Tversky regularization parameters on the validation loss during training of the 3D U-Net model. Tversky loss functions with higher α values converged to lower validation loss at the end of 15000 training iterations.*

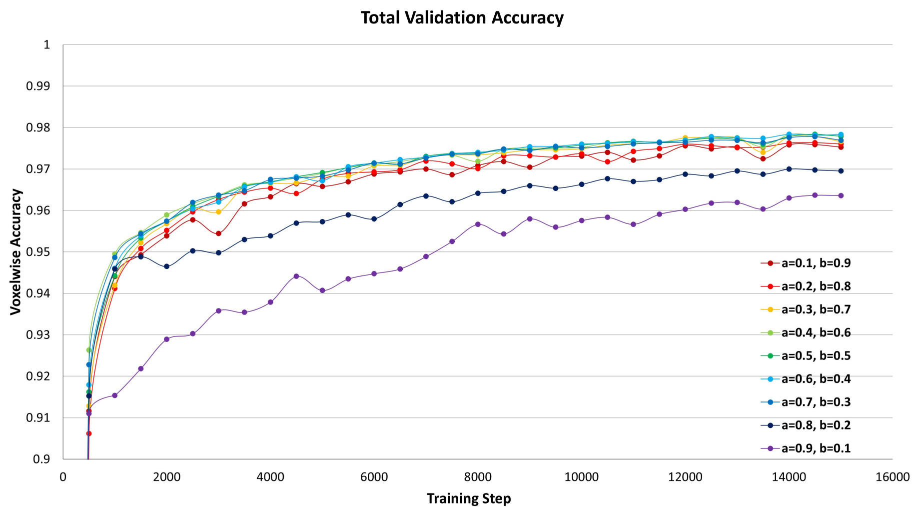


Figure 4.10: *Validation accuracy differences when changing values of Tversky regularization parameters. Models trained with higher α values had lower accuracy during training.*

Total validation accuracy and loss metrics alone are not sufficient to determine which Tversky parameters are most suitable for training the proposed 3D U-Net model for secondary structure segmentation. Total accuracy is a global measure that does not account for the class imbalance in the data and does not reveal how the model predicts each of the 4 classes. To better understand the performance of the 9 classification models trained with different Tversky parameters, confusion matrices were generated for each model on 100 simulated maps from the test dataset (see Appendix B.1). The confusion matrices compared the predicted results from the models with the ground truth labels for each voxel. True Positive (TP), False Positive (FP), False Negative (FN) and True Negative (TN) voxels were computed using *One-vs-All* strategy and the partial class membership method proposed by Beleitas et al. [14]. Additional performance metrics described in the previous section were calculated in order to determine which set of Tversky regularization parameters produced the best performing model. Macro-averaged statistics between all classes are plotted as a function of the α parameter in Figure 4.11. It can be seen from the plots that models trained with lower α (higher β) values yielded higher averaged recall, F2 and Specificity values for all classes. Precision was the only metric that increased with increasing the α parameter, while NPV changed very slightly between low and high α values. The F1 metric peaked at α=0.6 and dropped significantly at higher α values.

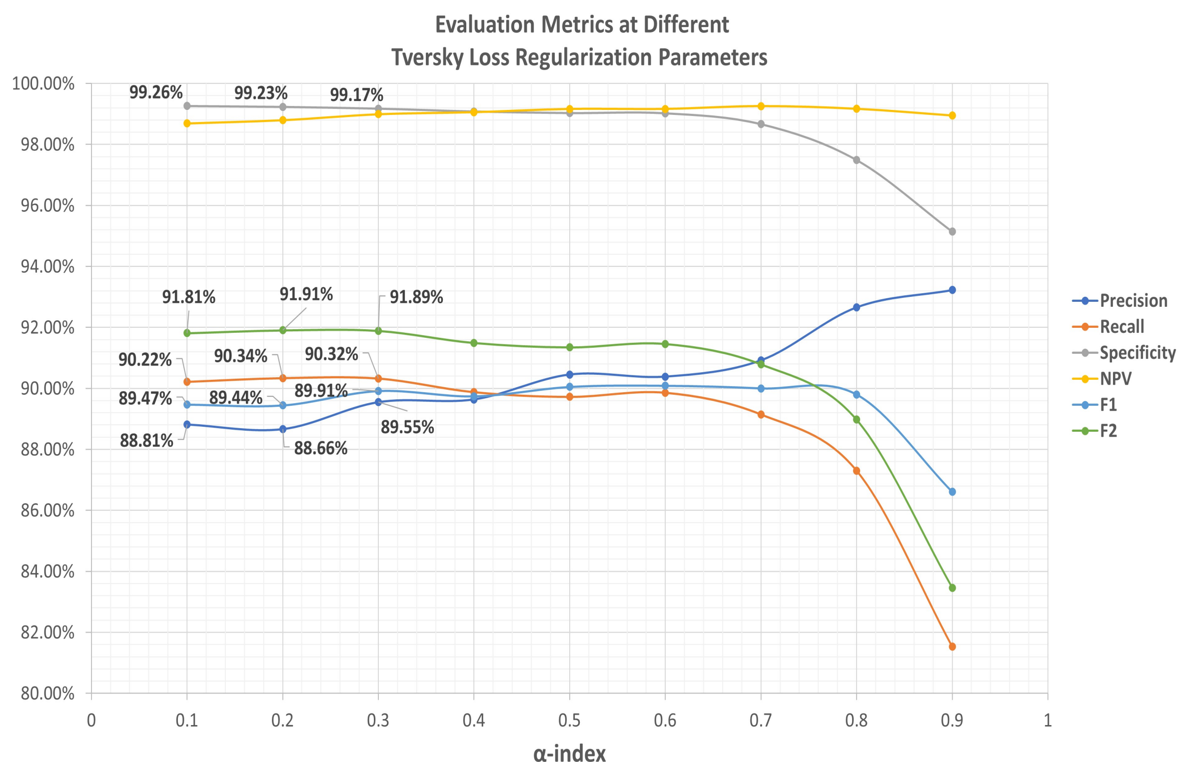


Figure 4.11: *Macro-averaged performance metrics on 100 simulated maps from the test set for different values of Tversky parameters α and β used in training the 3D U-Net. Metric values are displayed for α=0.1, 0.2 and 0.3 to illustrate the slightly better overall performance when α=0.2 and β=0.8.*

The goal of the proposed segmentation model is to accurately identify all 3 types of secondary structure elements (α-helices, β-sheets and turns/loops) in electron density maps. And because segmentation is done at the voxel level, it is important to have high selectivity and sensitivity even to very small structures but not at the expense of sacrificing precision. Figure 4.11 shows that the highest, and almost identical, recall rates, specificity and F2 scores are achieved with lower α-parameter values. Although there is no significant differences in the average metrics between the models trained with α=0.1, α=0.2 and α=0.3, a comparison of their confusion matrices in Appendix B.1 reveals that total number of predictions for each

class varied significantly. The higher average metrics for the model trained with α=0.3 were due to higher numbers of void and rest predictions at the expense of lower number of alpha and beta predictions. The model trained with α=0.1 had higher number of rest predictions at the expense of beta predictions which influenced the average precision and recall metrics. The model trained with α=0.2 provided the best balance between the alpha and beta class predictions. Furthermore, the highest average recall and F2 scores were achieved with α=0.2 parameter. Therefore, it was concluded that the best overall performing model for identifying secondary structure elements was the one trained with $\alpha = 0.2$ and $\beta = 0.8$.

4.3.3 SSE Segmentation on Simulated Density Maps

The ability of the 3D U-Net model to segment both α-helices and β-sheets was first tested on a set of simulated density maps. Similarly to the results presented in [11] and [37], the set of proteins chosen for this evaluation were representative of the four SCOP families. It was confirmed that three of the four PDB IDs used in the two referenced papers - (1C3W, α), (1TIM, α/β) and (1BVP, α lower domain and β upper domain) were not contained in the training or test datasets and as such were used in the assessment of the 3D U-Net segmentation model. The fourth protein evaluated in the previous studies, 1IRK, was found to be in the set used for training the 3D U-Net model. Therefore, another randomly chosen PDB ID (1LP4) from the (α+β) SCOP family is examined instead of 1IRK.

The four PBD structures were downloaded from the Protein Data Bank and electron density maps with size 64x64x64 were generated using EMAN2 program "*pdb2mrc*" with a sampling of 1Å/voxel. In order to demonstrate that the proposed 3D U-Net model can simultaneously classify maps in the medium resolution range (5Å-8Å), the four maps were simulated to one of the four different resolution levels (5Å, 6Å, 7Å or 8Å) chosen at random. Secondary structure labels for the four maps were generated using the same labeling technique as described above. The four simulated maps, without their true SSE labels, were input to the 3D U-Net Estimator. The estimator predicted each voxel in the four maps as either α-helix, β-sheet, rest/loop, or background. The predictions from the models were compared

against their corresponding true labels to compute confusion matrices and other performance measures described above. Furthermore, using the segmentation results from the model and the electron density values from the original map, predicted α-helix, β-sheet and rest/loop voxels were saved as electron density maps in separate MRC files. These predicted maps were opened and visualized in Chimera alongside the source PDB structure. Visual qualitative results and computed performance metrics for each one of the four simulated maps are presented in the following subsections.

1C3W

PDB ID: 1C3W is the atomic structure of the protein bacteriorhodopsin. Bacteriorhodopsin is a relatively small protein with only 222 residues and its structure contains seven long alpha-helices connected by short loops and a small beta-sheet. The atomic structure of the protein was originally determined by X-ray diffraction of crystals. SCOP annotates bacteriorhodopsin as an all-alpha protein. The 3D U-Net model was tested with an electron density map simulated at 5Å resolution from the 1C3W atomic structure. The confusion matrix and the voxelwise evaluation metrics for the segmentation of 1C3W are shown in Table 4.2. Due to its relatively small size, most of the voxels in the simulated 64x64x64 map were background and the model identified them with above 99% precision and recall. 22948 or 92.15% of the 24902 alpha voxels were correctly identified by the model and the majority of misclassified alpha-voxels (1717 of 1942) were predicted as being turns/loops. There is only one beta-sheet present in the structure which occupied a total of 1687 voxels and the model correctly identified 1652 of those voxels yielding a recall rate of 97.93% for the beta-sheet class. The lowest precision and recall scores were observed for the turn/loop class, 59.12% and 87.52% respectively. These lower scores impact the macro-averaged metrics which weigh each class equally. Micro-averaged F1 and F2 metrics, which take into account the inherent class imbalance, indicate the superb overall performance of the segmentation model on this simulated map. The segmentation accuracy of the model can be visually seen in Column D of Figure 4.12 which shows predicted maps overlaid on top of the true PDB structure. It

can be seen that the segmentation model correctly and accurately identified the locations of the seven alpha helices and one beta-sheet in the simulated map.

1C3W 5Å		predicted				Totals	Precision	Recall	Specificity	NPV	F1	F2
		void	alpha	beta	rest							
Expected	void	231734	496	84	140	232454	99.94%	99.69%	99.54%	97.62%	99.82%	99.74%
	alpha	102	22948	135	1717	24902	96.52%	92.15%	99.65%	99.18%	94.28%	92.99%
	beta	4	11	1652	20	1687	86.63%	97.93%	99.90%	99.99%	91.93%	95.44%
	rest	30	321	36	2714	3101	59.12%	87.52%	99.28%	99.85%	70.57%	79.85%
Totals		231870	23776	1907	4591	262144						
macro-average							**85.55%**	**94.32%**	**99.59%**	**99.16%**	**89.15%**	**92.00%**
micro-average							**98.82%**	**98.82%**	**99.61%**	**99.61%**	**98.82%**	**98.82%**

Table 4.2: 1C3W Segmentation Results

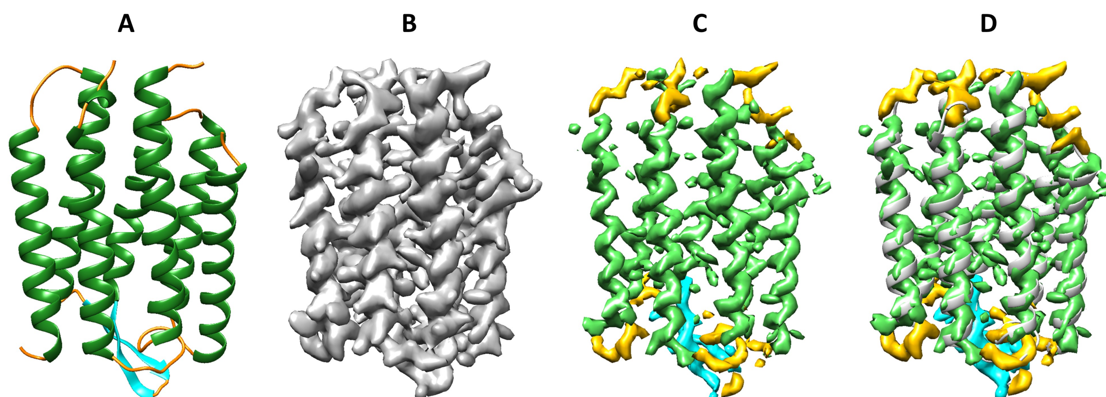


Figure 4.12: *Visual comparison of segmented map and ground truth PDB structure in Chimera.*

1BVP

Atomic structure of Bluetongue Virus VP7 is stored under PDB ID: 1BVP in the Protein Data Bank. It is a relative large protein with 2094 amino acids arranged in two domains: a lower domain made of alpha-helices, and an upper domain made of beta-sheets. Within each domain, the SSEs are connected with turns and loops of different lengths. A 6Å resolution map was simulated from the atomic model of Bluetongue Virus VP7 and used to test the proposed 3D U-Net segmentation model. Table 4.3 shows the confusion matrix and the segmentation results of the 1BVP simulated map. Because of its large size, less than half of the map voxels are background but the model still identified them with high precision and recall. The model correctly predicted 91.99% of all alpha-helix voxels present in the map but the precision of an alpha prediction was 76.74% due to a high number of misclassified turn/loop voxels. The model correctly identified very small α-helix structures present in the upper $\beta-$ domain. Around 23% of the β-sheet voxels in the upper domain were misclassified as turns/loops and another small percentage were classified as belonging to the alpha background classes, giving a beta-class recall score of 73.62%. Similarly, the model imprecisely predicted β-helix for a significant number of turn/loop voxels. The model struggled to identify more than 30% of the turn/loop voxels as they were often confused as alpha-helices and beta-sheets. And when the model predicted a turn/loop voxel, it was correct 79.57% of the time. The total voxelwise accuracy for 1BVP was 88.24%. The segmented maps from the model output and the atomic structure of 1BVP were overlaid in Chimera, providing a visual measure of the quality of segmentation (see Figure 4.13).

1BVP 6Å		predicted				Totals	Precision	Recall	Specificity	NPV	F1	F2
		void	alpha	beta	turn/loop							
Expected	void	127415	1066	396	951	129828	99.56%	98.14%	99.58%	98.20%	98.85%	98.42%
	alpha	85	45274	532	3325	49216	76.74%	91.99%	93.55%	98.06%	83.67%	88.47%
	beta	94	758	18903	5922	25677	74.83%	73.62%	97.31%	97.14%	74.22%	73.86%
	turn/loop	383	11900	5429	39711	57423	79.57%	69.16%	95.02%	91.65%	74.00%	71.01%
Totals		127977	58998	25260	49909	262144						
						macro-average	**82.67%**	**83.23%**	**96.37%**	**96.26%**	**82.68%**	**82.94%**
						micro-average	**88.24%**	**88.24%**	**96.08%**	**96.08%**	**88.24%**	**88.24%**

Table 4.3: 1BVP Segmentation Results

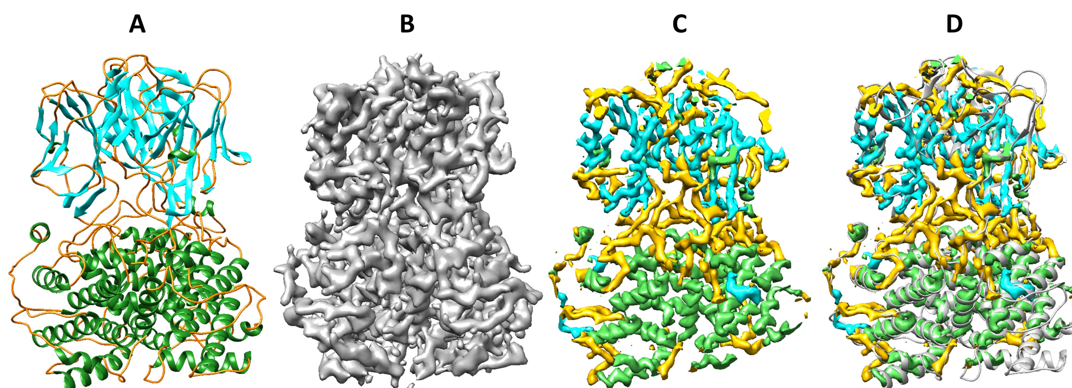


Figure 4.13: *Visual comparison of segmented map and ground truth PDB structure in Chimera.*

1TIM

Triose phospate isomerase (pdb id:1TIM) is an alpha/beta protein containing alternating alpha helices and beta sheets in its single chain of 248 amino acids. The atomic structure shows that the outer part of the protein is made of eleven alpha-helices while buried in the interior are eight beta-sheets arranged in the shape of a barrel. A simulated electron density map of 1TIM at 7Å resolution was used to test the ability of the proposed deep learning model to segment secondary structure elements. The segmentation results and model performance

metrics are summarized in Table 4.4. Approximately 84% of all voxels in the simulated map were background/void due to the small size of triose phosphate isomerase. Only very few background voxels were not classified by the model, and out of all background predictions made by the model, only 118 were incorrect. Around 12% of helix-voxels in the simulated map were not classified as such, and out of those most were misclassified as turns/loops. Similarly, a large number of beta-sheet voxels (1547) were misclassified as turns/loops. In total, 1 out of 3 turn/loop predictions by the model were incorrect. However, the recall rate for the turn/loop class was higher than the recall rates for alpha-helices and beta-sheets. Figure 4.14 shows another perspective of the segmentatioin accuracy of the the 3D U-Net model. In particular, Column D shows a comparison of the ground-truth X-ray structure and identified secondary structure elements by the 3D U-Net model. Closer look at the segmented density map reveals that the model identified all 11 alpha-helices on the surface of the protein. Figure 4.14 also reveals that beta-sheet voxels were correctly identified on the interior of the protein although their exact number could not be visually determined from the segmented map due to the lower resolution level.

1TIM 7Å		predicted				Totals	Precision	Recall	Specificity	NPV	F1	F2
		void	alpha	beta	turn/loop							
Expected	void	220036	317	72	457	220882	99.95%	99.62%	99.71%	97.99%	99.78%	99.68%
	alpha	26	20382	230	2516	23154	94.38%	88.03%	99.49%	98.85%	91.09%	89.23%
	beta	20	390	5984	1547	7941	88.94%	75.36%	99.71%	99.23%	81.59%	77.73%
	turn/loop	72	507	442	9146	10167	66.93%	89.96%	98.21%	99.59%	76.75%	84.16%
Totals		220154	21596	6728	13666	262144						
macro-average							**87.55%**	**88.24%**	**99.28%**	**98.91%**	**87.30%**	**87.70%**
micro-average							**97.48%**	**97.48%**	**99.16%**	**99.16%**	**97.48%**	**97.48%**

Table 4.4: 1TIM Segmentation Results

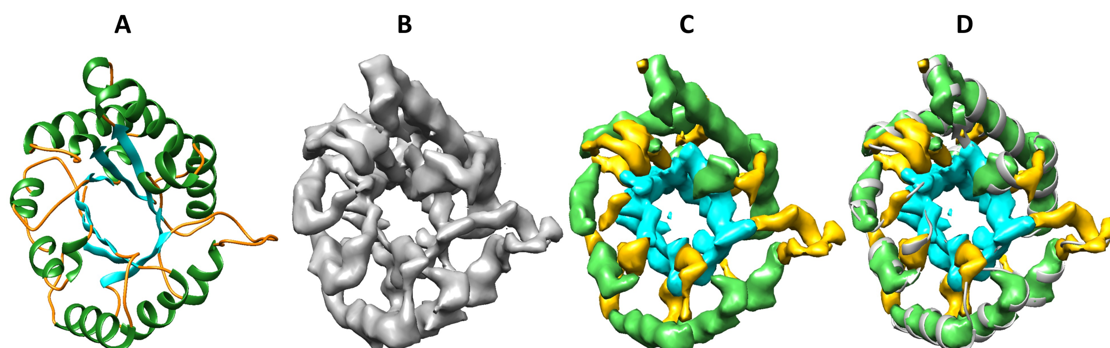


Figure 4.14: *Visual comparison of segmented map and ground truth PDB structure in Chimera.*

1LP4

PDB ID:1LP4 is a 332 residue long atomic structure of Protein kinase CK2. The structure of the protein contains segregated alpha and beta regions which make it a part of the alpha+beta SCOP family. The alpha region contains 20 *alpha*-helices of different lengths and the beta region is made of 7 mostly small β-sheets. An 8Å-resolution map was simulated from the PDB structure and was used to evaluate the segmentation model. Counts of predicted voxels for each class and their agreement with the ground truth labels are provided in Table 4.5 along with per-class and micro- and macro-averaged classification metrics. Because of the small protein size, most of the voxels in the simulated map were background. Almost all background voxels were identified correctly and only very few non-background voxels were misclassified, giving high precision and recall rates. Sensitivity to α-helices and β-sheets was 89.49% and 86.63%, respectively. The most number of alpha-helix and beta-sheet voxels were misclassifed as turns/loops. Similarly, 4544 turn/loop voxels were misclassifed as belonging to the alpha-helix class and 1147 were misclassifed as beta-sheets. The total precision and recall rates micro-averaged for all classes was 95.58%. The high agreement of the predicted voxels can be seen in Figure 4.15 Column D which shows the segmented maps from the

model overlaid on top of the true PDB structure for the protein.

1LP4 8Å		predicted				Totals	Precision	Recall	Specificity	NPV	F1	F2
		void	alpha	beta	turn/loop							
Expected	void	197639	594	218	619	199070	99.92%	99.28%	99.76%	97.78%	99.60%	99.41%
	alpha	32	29040	300	3079	32451	84.59%	89.49%	97.70%	98.50%	86.97%	88.46%
	beta	36	151	8361	780	9328	83.39%	89.63%	99.34%	99.62%	86.40%	88.31%
	turn/loop	85	4544	1147	15519	21295	77.61%	72.88%	98.14%	97.61%	75.17%	73.78%
Totals		197792	34329	10026	19997	262144						
						macro-average	**86.38%**	**87.82%**	**98.73%**	**98.38%**	**87.04%**	**87.49%**
						micro-average	**95.58%**	**95.58%**	**98.53%**	**98.53%**	**95.58%**	**95.58%**

Table 4.5: 1LP4 Segmentation Results

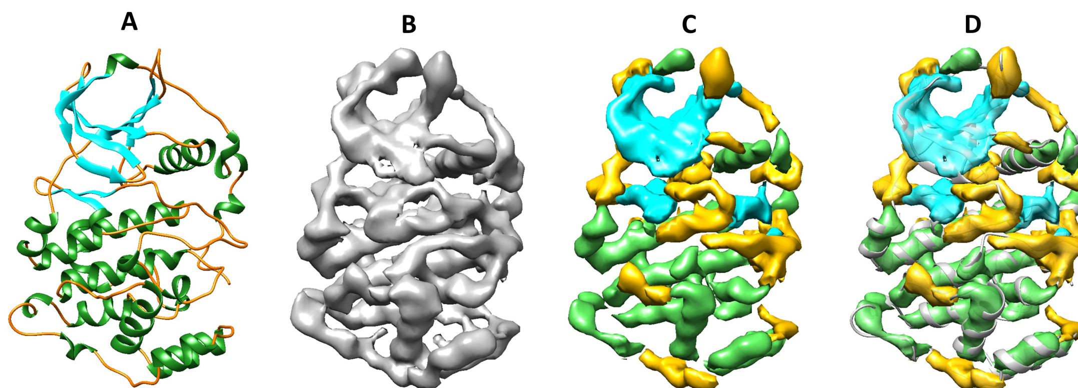


Figure 4.15: *Visual comparison of segmented map and ground truth PDB structure in Chimera.*

4.3.4 SSE Segmentation on Experimental Density Maps

To evaluate the hypothesis that a 3D U-Net model trained on simulated protein density maps can be used to identify secondary structure elements in real experimental maps, electron microscopy data for a mouse protein called P-glycoprotein(ABCB1) was downloaded from the EM Data Bank and tested with the trained segmentation model. The electron density map (EMDB ID EMD-4391) was reconstructed to 7.9Å resolution using cryo-EM Single Particle Analysis method [76]. The map deposition authors report that the resolution value was determined by FSC 0.143 cut-off method. The reconstructed map's dimensions are 156 voxels and the voxel spacing is 1.06Å. This map was chosen because a model PDB structure (PDB ID 6GDI) had been built from it and it was within the operating resolution range and voxel spacing of the trained 3D U-Net model. A Chimera ribbon representation of the PDB structure is shown in column A of Figure 4.16, while column B shows the

The map file of EMD-4391 was parsed using the *mrcfile* library to extract the 3D data array with density values. The electron density values were normalized using the same min-max normalization and threshold as used for preparing the training data. An additional processing step was applied to the data array in order to be able to pass it to the segmentation model which required input size of 64x64x64. The map was first reduced to size 128x128x128 by cropping 14 voxels from each side in each dimension, leaving the central portion of the map. Then, the array cube was split into 8 non-overlapping cubes of size 64x64x64. All 8 arrays were sequentially passed to the input layer of the 3D U-Net network for segmentation. After each smaller map was predicted by the model, the arrays were stitched back together based on the predicted secondary structure label to produce 3 segmented maps each with dimension 128x128x128. Figure 4.16 Column C shows the segmented maps in different colors for the different structure elements. Column D overlays the predicted results on the ribbon representation of the fitted PDB structure. The accuracy of the results can be qualitatively judged by comparing the locations of alpha-helices and beta-sheets from the fitted PDB structure with the locations identified by the model.

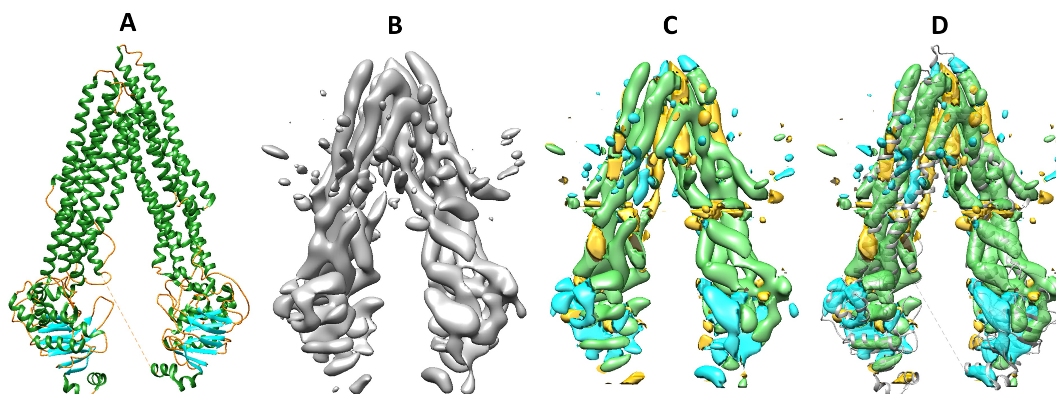


Figure 4.16: *Segmentation results of experimental map EMD-4391. Column A shows a ribbon diagram of PDB ID 6GDI. Column B shows the Chimera visualization of experimental density map of P-glycoprotein(ABCB1) (EMD-4391) at a 0.0572 threshold. In Column C, the results of secondary structure identification are shown with different colors for the voxels classified for each class: green for α-helix, cyan for β-sheet and orange for turns/loops. The PDB structure and the segmented results are superimposed in Column D.*

Calculating performance metrics on the segmentation results of the experimental map is not as straight forward as it is for simulated maps. The reason is there are no ground truth labels defined for each voxel in the experimental map. Ground truth labels were indirectly inferred from the PDB structure by first generating partial density maps from only the atoms identified in the 6GDI atomic model to be part of each corresponding secondary structure. To ensure maximum alignment and equal grid spacing with the experimental map, the partial maps for each SSE were created in Chimera using EMAN's pdb2mrc-based command *molmap* with argument -onGrid. The partial segmented maps were then used to create labels using the same procedure as described in Section 4.1.5. Because in a simulated map each atom is modeled as a 3D Gaussian distribution, all voxels contain at least some density from each secondary structure. Label assignment therefore depends on a threshold value below which voxels are considered to be empty/background. Threshold value 0.5 instead of 0.1 was used

to zero out the partial simulated maps before they were used to generate the labels for EMD-4391. The range of density values in the experimental map was between -0.15 and 0.22. These values were first normalized using min-max normalization method and then a threshold of 0.459, corresponding to raw density value of 0.02, was used to zero out voxels below that level. These special threshold levels were only required for this part in order to enable the comparison of segmentation results of an experimental map with labels generated from simulated maps. Counts of predicted voxels for each class and their agreement with the generated labels from the PDB structures are provided in Table 4.6 along with per-class and micro- and macro-averaged classification metrics.

EMD-4391 7.9Å		predicted				Totals	Precision	Recall	Specificity	NPV	F1	F2
		void	alpha	beta	turn/loop							
Expected	void	1756436	43286	57246	171604	2028572	99.78%	86.58%	94.43%	19.22%	92.72%	88.94%
	alpha	2705	24435	3398	17205	47743	34.56%	51.18%	97.74%	98.85%	41.26%	46.69%
	beta	260	40	3910	2029	6239	5.70%	62.67%	96.91%	99.89%	10.46%	20.91%
	turn/loop	855	2940	3997	6806	14598	3.44%	46.62%	90.84%	99.59%	6.41%	13.29%
Totals		1760256	70701	68551	197644	2097152						
						macro-average	**35.87%**	**61.76%**	**94.98%**	**79.39%**	**37.71%**	**42.46%**
						micro-average	**85.43%**	**85.43%**	**95.14%**	**95.14%**	**85.43%**	**85.43%**

Table 4.6: EMD-4391 Segmentation Results

4.4 Discussion

The 3D U-Net model was successfully trained and accuracy and loss values were shown to converge at around 15000 iterations. The values of regularization parameters in the Tversky loss function were shown to have an effect on the convergence of the model. It was observed that higher β (lower α led to higher sensitivity (recall) and lower precision across all classes. For this voxelwise segmentation problem it is more important to have better sensitivity in order to identify even small SSEs. The best overall performance with high recall without sacrificing too much precision was achieved with β=0.8 and α=0.2. The model achieved consistently high precision, recall and specificity rates when tested on four simulated protein structures from each of the four major SCOP families. Regardless of the shape, size or location, the model identified alpha-helix structures as being distinctively different than beta-sheet structures. Some highlights include the identification of two very small alpha-helices in the upper beta-domain of 1BVP protein. In most cases the model identified all present SSEs correctly, and if metrics were calculated based on number of instances (i.e. whole structure counts) versus number of correctly identified voxels, accuracy would be even higher. It was observed that alpha helices were more often misclassifed as turns/loops and turns/loops were more often misclassified as alpha helices. This can be due to one of two possibilities. One suggestion is that our labeling scheme may be improperly delineating the ends of alpha helices with the beginnings of turns/loops and therefore the model is inadequately trained. Second possibility is that some turns/loops may have similar cylindrical shape as alpha helices.

Chapter 5
SOFTWARE TECHNOLOGY PACKAGE

Training 3D U-Net model for secondary structure segmentation took more than 11 hours to complete on a dedicated GPU node. In addition, training required a large dataset of simulated electron density maps and their corresponding secondary structure labels at each voxel. Only after a model is properly trained, it can be used to predict other electron density maps, including experimental density maps from the EM Data Bank. However, researchers who wish to use this technology should not be expected to go through the burden of training a deep learning model from scratch. Therefore, for the third and final goal of this research project, an easy-to-use software application is created that integrates the best performing 3D U-Net segmentation model described and evaluated in the previous chapter.

5.1 SS Predictor

5.1.1 Creating Executable

Layered architecuture pattern is used for creating a Python application that deploys the deep learning model for protein structure segmentation. The first layer is a Python application that defines the public interface and serves as the main entry point for creating a *SS Predictor* executable (see source code in Appendix C.3.1). The Python program takes as an argument the file path to an MRC file. The application provides logic for loading the specified MRC file from disk and for parsing its contents using the *mrcfile* Python library [20]. The Python code also loads a saved pre-trained 3D U-Net segmentation model as a Tensorflow Estimator object and passes to its *predict()* method the data array from the MRC file. Special logic in the code first handles input maps that are not cubes and larger or smaller than 64 voxels. Data arrays of smaller maps are padded with zeros in each dimension that is less than 64.

Larger maps are either central cropped or padded with zeros to dimensions $n * 64$, where n is integer, after which they are broken down into n^3 cubic maps each having size 64x64x64. The application passes all n^3 data arrays to the model and stitches back segmentation results into a single map. After the trained model outputs predictions, the application writes to disk seperate MRC files with voxels for each structure element (α-helix, β-sheet, turn/loop). In order to run the application without the need of having Python and all required libraries installed, the python code was compiled to a standalone executable under 64-bit Windows using PyInstaller utility [72]. The following command was executed within Python 3.6.2 virtual environment at the root of a folder containing the *ss_predictor.py* file and subfolder named *"trained_model"* containing model files generated after training the 3D U-Net network:

```
pyinstaller --add-data ./trained_model/;./trained_model -w ss_predictor.py
```

The output of the PyInstaller was a single folder that bundled all dependencies and libraries needed to execute the created *ss_predictor.exe* application. The entire program was self-contained within the PyInstaller bundle and no additional modules, libraries or Python distributions were needed in order to execute the program.

5.1.2 Creating Chimera Plug-in

UCSF Chimera is a popular visualization and analysis software package for protein density maps. Chimera allows third-party developers to deploy extensions in the form of plug-ins. A Chimera extension is created to invoke the SS Predictor program from within the main Chimera window. The extension is integrated as a utility and it provides an end-to-end workflow for segmenting density maps that are opened and visualized in Chimera. In addition, the utility automatically opens and displays the predicted results after the input map is segmented by the 3D U-Net model. The SS Predictor extension is implemented as a Python package consisting of a set of layered modules in a single folder on the file system. Separate modules within the package define user interface code and processing logic. The package also contains standard initialization modules that automatically register the

extension with Chimera Extension Manager (source code for all Python modules are included in Appendix C.3.2). After providing the path to the folder containing all extension modules in Chimera's third-party plugin locations menu, the initialization modules are automatically executed and the SS Predictor extension becomes available. The SS Predictor plug-in can be launched from the Tools/Utilities menu or can be configured to be run from an icon on the toolbar (see Figure 5.1).

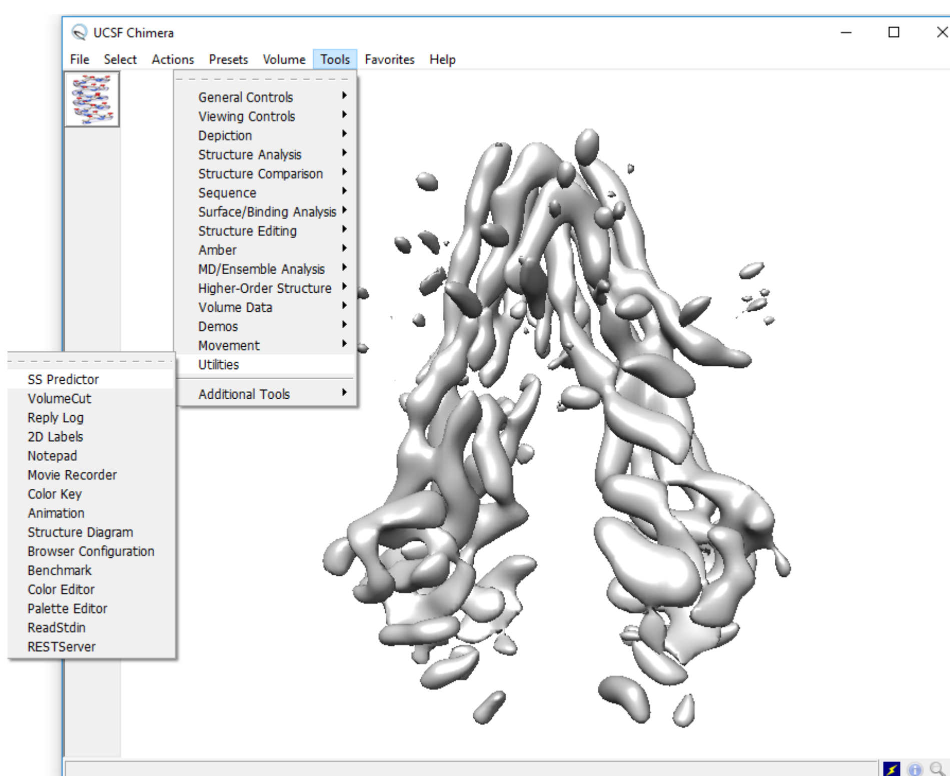


Figure 5.1: *Launching SS Predictor in Chimera can be done by navigating the Tools and Utilities menu.*

When a user activates the SS Predictor utility either from the menu or from the toolbar, a dialog shows up (Figure 5.2). The dialog is created using Chimera python package and serves as a simple graphical user interface for user to launch SS Predictor. The dialog contains a drop down menu to select the name of the MRC file which auto-populates if there is an

opened map. Once a valid MRC file is provided in the GUI and the user presses the Predict button, an intermediate layer of the plug-in generates a file path to the selected map and hands it to the SS Predictor executable as an argument. This intermediate layer knows the path to the executable and runs it in separate subprocess.

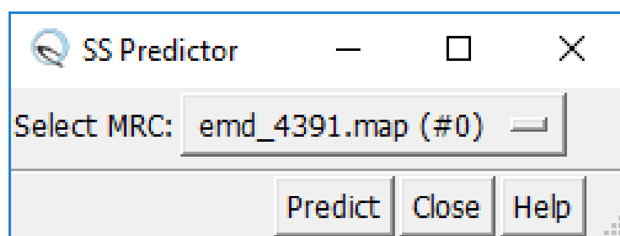


Figure 5.2: *Chimera Dialog for launching the SS Predictor utility.*

After waiting for the SS Predictor executable to complete in the background and produce results, plug-in logic opens the predicted segmented maps and displays them on the main Chimera viewing area. The segmented maps are automatically displayed with different colors and the color can be associated with the type of secondary structure by looking at the map names shown in Volume Viewer (see Figure 5.3).

Running the plug-in does not require installation of any special software, other than the plug-in itself, or hardware. The plug-in is supported on any regular Windows PC that supports the UCSF Chimera viewer. Processing time for segmenting a map in the plug-in, however, depends on the size of the map and the hardware configuration of the computer. Segmentation of the 156x156x156-voxel emd_4391 map takes 165 seconds on a laptop computer with Intel Core i7-7500 CPU @ 2.70GHz and 16GB RAM running 64-bit Windows 10 Operating System. However, only 21 seconds are needed for the SW tool to segment a 64x64x64 map on the same computer configuration. The 8x slower segmentation time is due to extra processing of emd_4391 for breaking it down into 8 smaller 64x64x64 segments and then for separately predicting each segment by the model before stitching them back into a single map.

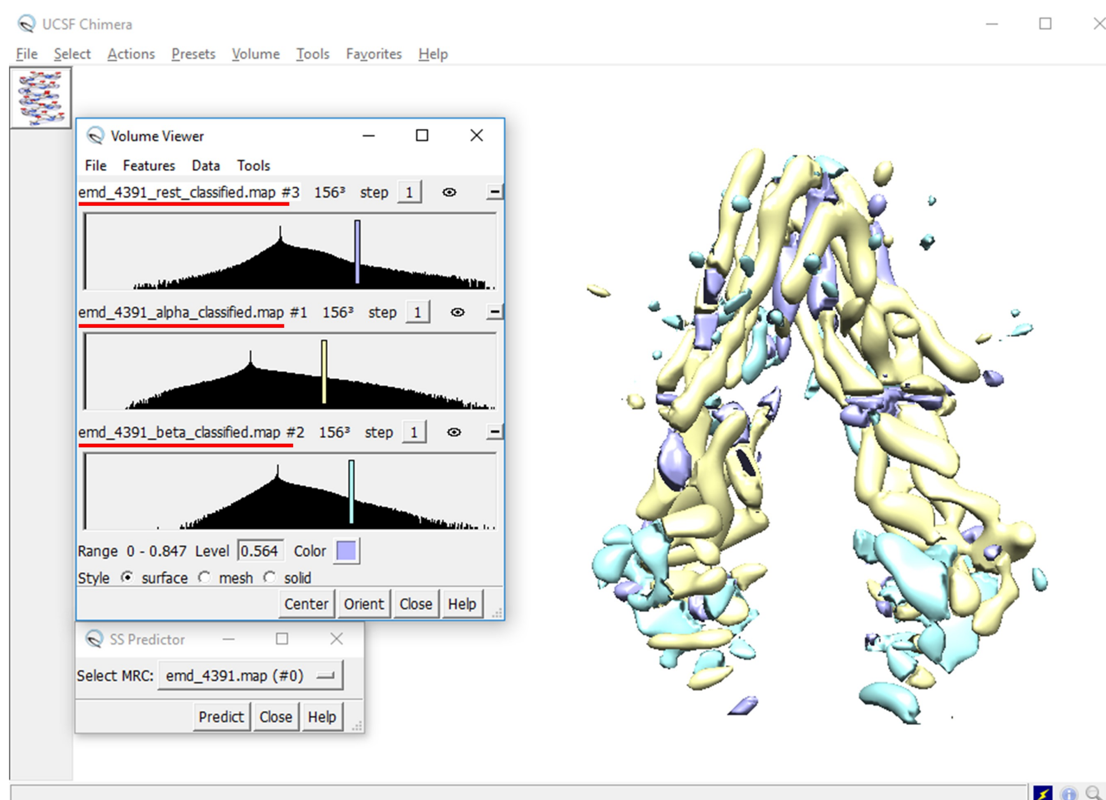


Figure 5.3: *After completion of the background executable, she SS Predictor extension automatically displays segmented map results in Chimera. Each map is shown in different color by default and each segment can be separately adjusted with controls in the Volume Viewer. Underlined in red are the names of the three segmented maps indicating the type of secondary structure element contained in each map.*

5.2 Discussion

The Chimera plug-in dialog serves as the top UI layer of the tool while the middle application layer relays input map to the deep learning model and fetches results from the model back to Chimera's user interface. The middle-tier layer does some pre-processing on data before passing it to the model. In particular, the application ensures that maps match the required input dimensions of 64x64x64 voxels. And in case input maps are bigger or smaller, the

application contains logic to break them in 64-sized patches or pad them with zeros. This alteration of input data, especially breaking down large maps into smaller maps, may have affected the performance of the model and needs to be further investigated. The middle-layer is also responsible for waiting the model to finish and for presenting the segmented maps in Chimera with different colors, overlaid over the original map. The core of the tool is a pre-trained deep learning model. The results produced by the tool depend solely on the quality of the trained model. This first version of the application references the 3D U-Net model trained for 15000 iterations with the training dataset and Tversky Loss parameters (α=0.2 and β=0.8) described in Chapter 4. Therefore, the quality of results returned by the tool should have similar accuracy and recall rates as presented in Chapter 4. The 3D U-Net model is quite complex and contains thousands of parameters across all layers. Still, a single 64x64x64 map can be segmented in as little as 20 seconds. Segmentation of larger maps by the tool may take a couple of minutes on a typical personal computer, depending on the size of the map and the hardware configuration of the computer.

Chapter 6

CONCLUSION AND FUTURE WORK

6.0.1 Specific Aim 1

We have demonstrated that deep learning models can learn resolution patterns from cryo-EM density maps. The two models presented in Chapter 3 predicted the resolution of simulated maps, as either high, medium or low, with high accuracy and precision. The three-class classifiers were further shown to correctly group together simulated maps with similar resolutions. Using experimental cryo-EM density maps and assuming the author-claimed published resolution value as the true value, the two models were shown to have specificity in the 60-95% range, although sensitivity lacked especially for medium- and low-resolution maps. The DNN and 3D CNN models both had positive and negative predictive values greater than 50% for all 3 resolution classes.

This paper is only the first step in the direction of using deep learning methods for resolution validation of cryo-electron density maps. The presented results suggest that deep learning techniques can be used to validate resolutions of 3D cryo-EM density maps. More experiments are necessary to explain why the resolution prediction of experimental cryo-EM maps had low and variable accuracies. Furthermore, future work will need to focus on improving the resolution prediction of experimental maps. Central cropping and zero-padding may affect prediction results, therefore other sizing approaches will need to be investigated. Further work is required to also investigate models that are agnostic to different input sizes. An example of one such network model is PointNet [62]. The PointNet CNN works with point clouds representing 3D geometry as input, thus avoiding the problem of selecting a rigid voxel grid size. The PointNet model has been shown to perform 3D object classification with better performance than models based on volumetric 3D input. PointNet architecture

may be applied for resolution classification of 3D cryo-electron microscopy density maps and it is likely to achieve better results than the ones presented here. Additionally, future deep learning models need to account for locally variable resolutions within a map. This can be achieved either by using smaller patch-based training and classification procedures or network architectures like the 3D U-Net that can do dense voxelwise classification.

This research only considered three different levels of resolution, but future work can extend the classification to finer resolution steps sizes, such as every 1Å. Future deep learning models may also benefit from using a more representative training data, sampled at different voxel spacings that more closely match the experimental maps in the EM Data Bank. An even better approach would be to use experimental density maps for training.

6.0.2 Specific Aim 2

Our results in Chapter 4 suggest that it is possible to use a set of cryo-EM maps for learning to detect SSE in other unknown and unseen cryo-EM maps. The proposed 3D U-Net was trained to classify each voxel of a protein density map as either being a part of α-helix, β-sheet, turn/loop or background. By doing so, the model was shown to identify the main secondary structure elements in a map. Using 4 simulated protein structures at between 5Å and 8Å resolutions, the model achieved combined voxelwise accuracy of 95.03%. And if accuracy was measured per each correctly identified secondary structure versus each correctly identified voxel, the number could be even higher. Experimental maps differ significantly from simulated maps as they come in different sizes and contain noise, which necessitated special pre-processing before they could be segmented by the proposed model. A test on an experimental map showed that the model can identify SSEs in the structure. However, performance metrics for experimental maps cannot be easily calculated due to unavailability of ground truth labels. Estimated results for the experimental map were calculated from labels generated from a fitted PDB structure and showed that recall rates were in the 50-60% range for each structure element and above 86% for background voxels.

Future work will attempt to improve the segmentation of experimental maps. A prereq-

uisite for that is to develop a reliable method to judge the quality of segmentation results of experimental maps. Also, evaluation needs to be done on more experimental maps. And because experimental maps can be bigger than the size of the input layer, different sizing techniques should be explored. For example, sliding window algorithm can be used to do classification on non-overlapping segments of the map. Other methods that could help in the segmentation of experimental maps include additional pre- and post-processing. Pre-processing can be implemented to filter out noise while post-processing algorithms like Conditional Random Field (CRF) can boost model performance by 1-2%.

Another area of focus is to optimize the current model by varying hyperparameters. It is unlikely that a significantly better performing model is identified but it is possible that a simpler model with fewer parameters produces similar results, in which case the simpler model would be preferred according to the principal of parsimony. Lastly, combining additional knowledge from other sources during training could produce a more robust model. Future deep learning models will benefit if experimental maps are included in the training data.

6.0.3 Specific Aim 3

A trained 3D U-Net model was successfully integrated in a standalone program called *SS Predictor*, which was further incorporated in the Chimera visualization package as a plug-in extension. The tool is fully-automated and there are no special parameters that users need to provide in order to segment a given map. The core of the tool is a pre-trained 3D U-Net model which has the flexibility to be distributed with, or separately from, the application itself. The developed *SS Predictor* tool is easy to set up and does not require any special software or hardware to run. However, the speed of the tool may be problematic when trying to segment very large maps on personal computers. Faster and more efficient prediction models can be incorporated into SS Predictor in the future. But segmentation is typically a lot faster on computers equipped with GPUs or multiple cores CPUs. Therefore, future work can attempt to reduce processing times by distributing segmentation of large maps on mul-

tiple parallel processes or threads. Another possibility is to evaluate whether the application and pre-trained model can be hosted on a cloud server rather than on the local host computer. This approach would allow researchers to upload their maps directly from Chimera and offload the computationally heavy-lifting to a more powerful server. After processing a user uploaded map, the server would return the segmented map results back to Chimera for visualization. This approach, limited mostly by the upload and download bandwidths, could potentially decrease the time of getting segmentation results. Finally, future work may extend deployments of SS Predictor to other Windows versions (32-bit, 64-bit) and other operating systems (Mac OS, Linux) to match the supported Chimera releases.

BIBLIOGRAPHY

[1] Ribosome | British Society for Cell Biology.

[2] Method of the Year 2015. *Nature Methods*, 13(1):1–1, January 2016.

[3] Marc Adrian, Jacques Dubochet, Jean Lepault, and Alasdair W. McDowall. Cryo-electron microscopy of viruses. *Nature*, 308(5954):32–36, March 1984.

[4] Kamal Al Nasr, Lin Chen, Dong Si, Desh Ranjan, Mohammad Zubair, and Jing He. Building the initial chain of the proteins through de novo modeling of the cryo-electron microscopy volume data at the medium resolutions. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine - BCB '12*, pages 490–497, Orlando, Florida, 2012. ACM Press.

[5] Kamal Al Nasr, Desh Ranjan, Mohammad Zubair, Lin Chen, and Jing He. Solving the Secondary Structure Matching Problem in Cryo-EM De Novo Modeling Using a Constrained K-Shortest Path Graph Algorithm. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):419–430, March 2014.

[6] A. Amunts, A. Brown, J. Toots, S. H. W. Scheres, and V. Ramakrishnan. The structure of the human mitochondrial ribosome. *Science*, 348(6230):95–98, April 2015.

[7] Todor K. Avramov and Dong Si. Deep Learning for Resolution Validation of Three Dimensional Cryo-Electron Microscopy Density Maps. In *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics - BCB '18*, pages 669–674, Washington, DC, USA, 2018. ACM Press.

[8] M. L. Baker, W. Jiang, F. J. Rixon, and W. Chiu. Common Ancestry of Herpesviruses and Tailed DNA Bacteriophages. *Journal of Virology*, 79(23):14967–14970, December 2005.

[9] Matthew L. Baker, Mariah R. Baker, Corey F. Hryc, Tao Ju, and Wah Chiu. Gorgon and pathwalking: Macromolecular modeling tools for subnanometer resolution density maps. *Biopolymers*, 97(9):655–668, September 2012.

[10] Matthew L. Baker, Wen Jiang, Brian R. Bowman, Z. Hong Zhou, Florante A. Quiocho, Frazer J. Rixon, and Wah Chiu. Architecture of the Herpes Simplex Virus Major

Capsid Protein Derived from Structural Bioinformatics. *Journal of Molecular Biology*, 331(2):447–456, August 2003.

[11] Matthew L. Baker, Tao Ju, and Wah Chiu. Identification of Secondary Structure Elements in Intermediate-Resolution Density Maps. *Structure*, 15(1):7–19, January 2007.

[12] Niccolò Banterle, Khanh Huy Bui, Edward A. Lemke, and Martin Beck. Fourier ring correlation as a resolution criterion for super-resolution microscopy. *Journal of Structural Biology*, 183(3):363–367, September 2013.

[13] M. Beck, A. Schmidt, J. Malmstroem, M. Claassen, A. Ori, A. Szymborska, F. Herzog, O. Rinner, J. Ellenberg, and R. Aebersold. The quantitative proteome of a human cell line. *Molecular Systems Biology*, 7(1):549–549, April 2014.

[14] Claudia Beleites, Reiner Salzer, and Valter Sergo. Validation of soft classification models using partial class memberships: An extended concept of sensitivity & co. applied to grading of astrocytoma tissues. *Chemometrics and Intelligent Laboratory Systems*, 122:12–22, March 2013.

[15] H. M. Berman. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, January 2000.

[16] Helen Berman, Kim Henrick, and Haruki Nakamura. Announcing the worldwide Protein Data Bank. *Nature Structural & Molecular Biology*, 10(12):980–980, December 2003.

[17] Helen Berman, Kim Henrick, Haruki Nakamura, and John L. Markley. The worldwide Protein Data Bank (wwPDB): ensuring a single, uniform archive of PDB data. *Nucleic Acids Research*, 35(suppl_1):D301–D303, 2007.

[18] Abhishek Biswas, Dong Si, Kamal Al Nasr, Desh Ranjan, Mohammad Zubair, and Jing He. Improved Efficiency in Cryo-EM Secondary Structure Topology Determination from Inaccurate DATA. *Journal of Bioinformatics and Computational Biology*, 10(03):1242006, June 2012.

[19] B. Böttcher, S. A. Wynne, and R. A. Crowther. Determination of the fold of the core protein of hepatitis B virus by electron cryomicroscopy. *Nature*, 386(6620):88–91, March 1997.

[20] Tom Burnley, Colin M. Palmer, and Martyn Winn. Recent developments in the *CCP-EM* software suite. *Acta Crystallographica Section D Structural Biology*, 73(6):469–477, June 2017.

[21] Giovanni Cardone, J. Bernard Heymann, and Alasdair C. Steven. One number does not fit all: Mapping local variations in resolution in cryo-EM reconstructions. *Journal of Structural Biology*, 184(2):226–236, November 2013.

[22] Elisabeth P Carpenter, Konstantinos Beis, Alexander D Cameron, and So Iwata. Overcoming the challenges of membrane protein crystallography. *Current Opinion in Structural Biology*, 18(5):581–586, October 2008.

[23] Marta Carroni and Helen R. Saibil. Cryo electron microscopy to determine the structure of macromolecular complexes. *Methods (San Diego, Calif.)*, 95:78–85, February 2016.

[24] Anchi Cheng, Richard Henderson, David Mastronarde, Steven J. Ludtke, Remco H.M. Schoenmakers, Judith Short, Roberto Marabini, Sargis Dallakyan, David Agard, and Martyn Winn. MRC2014: Extensions to the MRC format header for electron cryo-microscopy and tomography. *Journal of Structural Biology*, 192(2):146–150, November 2015.

[25] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv:1511.07289 [cs]*, November 2015. arXiv: 1511.07289.

[26] Daniel Cressey and Ewen Callaway. Cryo-electron microscopy wins chemistry Nobel. *Nature*, 550(7675):167–167, October 2017.

[27] D. J. De Rosier and A. Klug. Reconstruction of three dimensional structures from electron micrographs. *Nature*, 217(5124):130–134, January 1968.

[28] Hang Dou, Derek W. Burrows, Matthew L. Baker, and Tao Ju. Flexible Fitting of Atomic Models into Cryo-EM Density Maps Guided by Helix Correspondences. *Biophysical Journal*, 112(12):2479–2493, June 2017.

[29] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *The Journal of Machine Learning Research*, 12:2121–2159, July 2011.

[30] Joachim Frank. Single-particle reconstruction of biological macromolecules in electron microscopy – 30 years. *Quarterly Reviews of Biophysics*, 42(03):139, August 2009.

[31] R. M. Glaeser and K. A. Taylor. Radiation damage relative to transmission electron microscopy of biological specimens at low temperature: a review. *Journal of Microscopy*, 112(1):127–138, January 1978.

[32] Tamir Gonen, Yifan Cheng, Piotr Sliz, Yoko Hiroaki, Yoshinori Fujiyoshi, Stephen C. Harrison, and Thomas Walz. Lipid–protein interactions in double-layered two-dimensional AQP0 crystals. *Nature*, 438(7068):633–638, December 2005.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *arXiv:1406.4729 [cs]*, 8691:346–361, 2014. arXiv: 1406.4729.

[34] Richard Henderson. The potential and limitations of neutrons, electrons and X-rays for atomic resolution microscopy of unstained biological molecules. *Quarterly Reviews of Biophysics*, 28(02):171, May 1995.

[35] T. J. Hubbard, A. G. Murzin, S. E. Brenner, and C. Chothia. SCOP: a structural classification of proteins database. *Nucleic Acids Research*, 25(1):236–239, January 1997.

[36] Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 3d U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. *arXiv:1606.06650 [cs]*, June 2016. arXiv: 1606.06650.

[37] Wen Jiang, Matthew L. Baker, Steven J. Ludtke, and Wah Chiu. Bridging the information gap: computational tools for intermediate resolution structure interpretation. *Journal of Molecular Biology*, 308(5):1033–1044, May 2001.

[38] Konstantinos Kamnitsas, Christian Ledig, Virginia F. J. Newcombe, Joanna P. Simpson, Andrew D. Kane, David K. Menon, Daniel Rueckert, and Ben Glocker. Efficient Multi-Scale 3d CNN with Fully Connected CRF for Accurate Brain Lesion Segmentation. *Medical Image Analysis*, 36:61–78, February 2017. arXiv: 1603.05959.

[39] Heena Khatter, Alexander G. Myasnikov, S. Kundhavai Natchiar, and Bruno P. Klaholz. Structure of the human 80s ribosome. *Nature*, 520(7549):640–645, April 2015.

[40] Werner Kühlbrandt. Cryo-EM enters a new era. *eLife*, 3, August 2014.

[41] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. arXiv: 1412.6980.

[42] Yifei Kong, Xing Zhang, Timothy S. Baker, and Jianpeng Ma. A Structural-informatics Approach for Tracing β-Sheets: Building Pseudo-Cα Traces for β-Strands in Intermediate-resolution Density Maps. *Journal of Molecular Biology*, 339(1):117–130, May 2004.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[44] Alp Kucukelbir, Fred J Sigworth, and Hemant D Tagare. Quantifying the local resolution of cryo-EM density maps. *Nature Methods*, 11(1):63–65, January 2014.

[45] Mitch Leslie. There are millions of protein factories in every cell. Surprise, they're not all the same. *Science*, June 2017.

[46] Rongjian Li, Dong Si, Tao Zeng, Shuiwang Ji, and Jing He. Deep convolutional neural networks for detecting secondary structures in protein density maps from cryo-electron microscopy. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 41–46, Shenzhen, China, December 2016. IEEE.

[47] Xueming Li, Paul Mooney, Shawn Zheng, Christopher R Booth, Michael B Braunfeld, Sander Gubbens, David A Agard, and Yifan Cheng. Electron counting and beam-induced motion correction enable near-atomic-resolution single-particle cryo-EM. *Nature Methods*, 10(6):584–590, June 2013.

[48] Hstau Y. Liao and Joachim Frank. Definition and Estimation of Resolution in Single-Particle Reconstructions. *Structure*, 18(7):768–775, July 2010.

[49] Steffen Lindert, Nathan Alexander, Nils Wötzel, Mert Karakaş, Phoebe L. Stewart, and Jens Meiler. EM-Fold: De Novo Atomic-Detail Protein Structure Determination from Medium-Resolution Density Maps. *Structure*, 20(3):464–478, March 2012.

[50] Shian Liu, Johan Hattne, Francis E. Reyes, Silvia Sanchez-Martinez, M. Jason de la Cruz, Dan Shi, and Tamir Gonen. Atomic resolution structure determination by the cryo-EM method MicroED: Atomic Resolution Structure Determination. *Protein Science*, 26(1):8–15, January 2017.

[51] Zheng Liu, Fei Guo, Feng Wang, Tian-Cheng Li, and Wen Jiang. 2.9 Å Resolution Cryo-EM 3d Reconstruction of Close-Packed Virus Particles. *Structure*, 24(2):319–328, February 2016.

[52] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, Boston, MA, USA, June 2015. IEEE.

[53] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. *arXiv:1606.04797 [cs]*, June 2016. arXiv: 1606.04797.

[54] Jacqueline L. S. Milne, Mario J. Borgnia, Alberto Bartesaghi, Erin E. H. Tran, Lesley A. Earl, David M. Schauder, Jeffrey Lengyel, Jason Pierson, Ardan Patwardhan, and Sriram Subramaniam. Cryo-electron microscopy - a primer for the non-microscopist. *FEBS Journal*, 280(1):28–45, January 2013.

[55] Ron Milo. What is the total number of protein molecules per cell volume? A call to rethink some published values: Insights & Perspectives. *BioEssays*, 35(12):1050–1055, December 2013.

[56] Sharada P. Mohanty, David P. Hughes, and Marcel Salathé. Using Deep Learning for Image-Based Plant Disease Detection. *Frontiers in Plant Science*, 7, September 2016.

[57] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning Deconvolution Network for Semantic Segmentation. *arXiv:1505.04366 [cs]*, May 2015. arXiv: 1505.04366.

[58] C. A. Opitz, M. Kulke, M. C. Leake, C. Neagoe, H. Hinssen, R. J. Hajjar, and W. A. Linke. Damped elastic recoil of the titin spring in myofibrils of human myocardium. *Proceedings of the National Academy of Sciences*, 100(22):12688–12693, October 2003.

[59] L. Pauling, R. B. Corey, and H. R. Branson. The structure of proteins: Two hydrogen-bonded helical configurations of the polypeptide chain. *Proceedings of the National Academy of Sciences*, 37(4):205–211, April 1951.

[60] Pawel A. Penczek. Resolution measures in molecular electron microscopy. *Methods in Enzymology*, 482:73–100, 2010.

[61] Eric F. Pettersen, Thomas D. Goddard, Conrad C. Huang, Gregory S. Couch, Daniel M. Greenblatt, Elaine C. Meng, and Thomas E. Ferrin. Ucsf chimera–a visualization system for exploratory research and analysis. *Journal of Computational Chemistry*, 25(13):1605–1612, October 2004.

[62] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3d Classification and Segmentation. *arXiv:1612.00593 [cs]*, December 2016. arXiv: 1612.00593.

[63] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv:1505.04597 [cs]*, May 2015. arXiv: 1505.04597.

[64] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, September 2016. arXiv: 1609.04747.

[65] Seyed Sadegh Mohseni Salehi, Deniz Erdogmus, and Ali Gholipour. Tversky loss function for image segmentation using 3d fully convolutional deep networks. *arXiv:1706.05721 [cs]*, June 2017. arXiv: 1706.05721.

[66] Sjors H W Scheres and Shaoxia Chen. Prevention of overfitting in cryo-EM structure determination. *Nature Methods*, 9(9):853–854, September 2012.

[67] Sjors HW Scheres. Beam-induced motion correction for sub-megadalton cryo-EM particles. *eLife*, 3:e03665, August 2014.

[68] Dan Shi, Brent L Nannenga, Matthew G Iadanza, and Tamir Gonen. Three-dimensional electron crystallography of protein microcrystals. *eLife*, 2, November 2013.

[69] Dong Si, Shuiwang Ji, Kamal Al Nasr, and Jing He. A Machine Learning Approach for the Identification of Protein Secondary Structure Elements from Electron Cryo-Microscopy Density Maps. *Biopolymers*, 97(9):698–708, September 2012.

[70] Guang Tang, Liwei Peng, Philip R. Baldwin, Deepinder S. Mann, Wen Jiang, Ian Rees, and Steven J. Ludtke. EMAN2: An extensible image processing suite for electron microscopy. *Journal of Structural Biology*, 157(1):38–46, January 2007.

[71] K. A. Taylor and R. M. Glaeser. Electron Diffraction of Frozen, Hydrated Protein Crystals. *Science*, 186(4168):1036–1037, December 1974.

[72] PyInstaller Development Team. Pyinstaller, 2005-2019. http://www.pyinstaller.org/.

[73] Thomas C. Terwilliger. Rapid model building of α-helices in electron-density maps. *Acta Crystallographica Section D Biological Crystallography*, 66(3):268–275, March 2010.

[74] Thomas C. Terwilliger. Rapid model building of β-sheets in electron-density maps. *Acta Crystallographica Section D Biological Crystallography*, 66(3):276–284, March 2010.

[75] The HDF Group. Hierarchical data format, version 5, 1997-2018. http://www.hdfgroup.org/HDF5/.

[76] Nopnithi Thongin, Richard F Collins, Alessandro Barbieri, Talha Shafi, Alistair Siebert, and Robert C Ford. Novel features in the structure of P-glycoprotein (ABCB1) in the post-hydrolytic state as determined at 7.9Å resolution. *bioRxiv*, April 2018.

[77] Marin van Heel and Michael Schatz. Fourier shell correlation threshold criteria. *Journal of Structural Biology*, 151(3):250–262, September 2005.

[78] Marin van Heel and Michael Schatz. REASSESSING THE REVOLUTIONS RESOLUTIONS. *bioRxiv*, November 2017.

[79] Jose Luis Vilas, Josué Gómez-Blanco, Pablo Conesa, Roberto Melero, José Miguel de la Rosa-Trevín, Joaquin Otón, Jesús Cuenca, Roberto Marabini, José María Carazo, Javier Vargas, and Carlos Oscar S. Sorzano. MonoRes: Automatic and Accurate Estimation of Local Resolution for Electron Microscopy Maps. *Structure*, 26(2):337–344.e4, February 2018.

[80] Hong-Wei Wang and Jia-Wei Wang. How cryo-electron microscopy and X-ray crystallography complement each other: Cryo-EM and X-Ray Crystallography Complement Each Other. *Protein Science*, 26(1):32–39, January 2017.

[81] Liguo Wang and Fred J. Sigworth. Cryo-EM and Single Particles. *Physiology*, 21(1):13–18, February 2006.

[82] Gulnara Yusupova and Marat Yusupov. Ribosome biochemistry in crystal structure determination. *RNA*, 21(4):771–773, April 2015.

[83] Giuseppe Zanotti. Cryo-EM and X-Ray Crystallography: Complementary or Alternative Techniques? *NanoWorld Journal*, 2(2), 2016.

[84] Matthew D. Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, volume 8689, pages 818–833. Springer International Publishing, Cham, 2014.

[85] Z. H. Zhou. Seeing the Herpesvirus Capsid at 8.5Å. *Science*, 288(5467):877–880, May 2000.

[86] Z. H. Zhou, M. L. Baker, W. Jiang, M. Dougherty, J. Jakana, G. Dong, G. Lu, and W. Chiu. Electron cryomicroscopy and bioinformatics suggest protein fold models for rice dwarf virus. *Nature Structural Biology*, 8(10):868–873, October 2001.

[87] Z Hong Zhou. Towards atomic resolution structural determination by single-particle cryo-electron microscopy. *Current Opinion in Structural Biology*, 18(2):218–228, April 2008.

Appendix A

MICRO-AVERAGED PRECISION AND RECALL

A.1 Derivation and Equality Proof

To demonstrate that using *one-vs-all* method in the four-class protein secondary structure setting produces the same values for micro-averaged precision and recall, consider the following four individual confusion matrices that show the TP, TN, FP and FN counts with respect to each class:

		predicted			
		void	alpha	beta	turn/loop
expected	void	TP_v	FN_v	FN_v	FN_v
	alpha	FP_v	TN_v	TN_v	TN_v
	beta	FP_v	TN_v	TN_v	TN_v
	turn/loop	FP_v	TN_v	TN_v	TN_v

		predicted			
		void	alpha	beta	turn/loop
expected	void	TN_α	FP_α	TN_α	TN_α
	alpha	FN_α	TP_α	FN_α	FN_α
	beta	TN_α	FP_α	TN_α	TN_α
	turn/loop	TN_α	FP_α	TN_α	TN_α

		predicted			
		void	alpha	beta	turn/loop
expected	void	TN_β	TN_β	FP_β	TN_β
	alpha	TN_β	TN_β	FP_β	TN_β
	beta	FN_β	FN_β	TP_β	FN_β
	turn/loop	TN_β	TN_β	FP_β	TN_β

		predicted			
		void	alpha	beta	turn/loop
expected	void	TN_t	TN_t	TN_t	FP_t
	alpha	TN_t	TN_t	TN_t	FP_t
	beta	TN_t	TN_t	TN_t	FP_t
	turn/loop	FN_t	FN_t	FN_t	TP_t

When calculating micro-averaged precision and recall metrics, TP, FP and FN counts are first summed for all classes and then used in the corresponding equations:

$$Precision_{micro-averaged} = \frac{\sum_c TP_c}{\sum_c TP_c + \sum_c FP_c} \tag{A.1}$$

$$Recall_{micro-averaged} = \frac{\sum_c TP_c}{\sum_c TP_c + \sum_c FN_c} \tag{A.2}$$

where c is the class label.

It can be seen from the confusion matrices above that a false positive classifications with respect to one class are considered false negative classifications with respect to the other three classes. Summing over all classes produces the following corollary:

$$\sum_c FP_c = \sum_c FN_c = Total - \sum_c TP_c$$

Therefore, substituting for $\sum_c FP_c$ and $\sum_c FN_c$ in equations A.1 and A.2, we can conclude that

$$Precision_{micro-averaged} = Recall_{micro-averaged} = \frac{\sum_c TP_c}{Total}$$

Similarly, this proof can be extended to show that

$$F_{1micro-averaged} = F_{2micro-averaged} = \frac{\sum_c TP_c}{Total}$$

Appendix B

OPTIMIZING TVERSKY LOSS PARAMETERS

B.1 Confusion Matrices and Evaluation Metrics for 100 Test Maps

Numbers of predicted and expected voxels for all segmentation classes are shown in confusion matrices for each model trained with different Tversky loss parameters. 100 maps from the test data set were used for the evaluation. The ground-truth labels for 84.9% of the total 26214400 voxels were void, 5.3% alpha, 3.6% beta and 6.2% rest. Per-class metrics are calculated from the confusion matrices. Highlighted in gray are the global macro-averaged metrics between all classes.

		Predicted				**Precision**	**Recall**	**Specificity**	**NPV**	**F1**	**F2**
		void	alpha	beta	rest						
α=0.1 β=0.9	void	22117210	41187	21933	66776	99.92%	99.42%	99.54%	96.82%	99.67%	99.44%
	alpha	7421	1297853	12952	81413	84.89%	92.73%	99.07%	99.59%	88.64%	93.92%
	beta	4732	16336	797238	119453	86.84%	85.02%	99.52%	99.44%	85.92%	87.56%
	rest	6253	173407	85892	1364344	83.60%	83.71%	98.91%	98.92%	83.65%	86.33%
Totals		22135616	1528783	918015	1631986	**88.81%**	**90.22%**	**99.26%**	**98.69%**	**89.47%**	**91.81%**
α=0.2 β=0.8	void	22137548	34832	26261	48465	99.89%	99.51%	99.36%	97.30%	99.70%	99.48%
	alpha	10862	1284947	18191	85639	85.06%	91.81%	99.09%	99.54%	88.31%	93.16%
	beta	5304	18601	826766	87088	83.92%	88.16%	99.37%	99.56%	85.99%	90.19%
	rest	9304	172182	113982	1334428	85.78%	81.87%	99.10%	98.80%	83.78%	84.79%
Totals		22163018	1510562	985200	1555620	**88.66%**	**90.34%**	**99.23%**	**98.80%**	**89.44%**	**91.91%**
α=0.3 β=0.7	void	22168926	21986	17385	38809	99.81%	99.65%	98.92%	98.05%	99.73%	99.50%
	alpha	19068	1277563	18101	84907	86.89%	91.28%	99.22%	99.51%	89.03%	92.75%
	beta	8761	14463	818092	96443	85.48%	87.24%	99.45%	99.53%	86.35%	89.42%
	rest	15076	156337	103523	1354960	86.02%	83.13%	99.10%	98.88%	84.55%	85.88%
Totals		22211831	1470349	957101	1575119	**89.55%**	**90.32%**	**99.17%**	**98.99%**	**89.91%**	**91.89%**

Continued on next page

			Predicted				Precision	Recall	Specificity	NPV	F1	F2
			void	alpha	beta	rest						
α=0.4	β=0.6	void	22184066	19050	14458	29532	99.74%	99.72%	98.52%	98.41%	99.73%	99.48%
		alpha	25180	1267300	15134	92025	86.76%	90.54%	99.22%	99.47%	88.61%	92.14%
		beta	12324	15281	811576	98578	86.10%	86.54%	99.48%	99.50%	86.32%	88.84%
		rest	21378	159084	101428	1348006	85.96%	82.71%	99.10%	98.86%	84.30%	85.51%
Totals			22242948	1460715	942596	1568141	**89.64%**	**89.88%**	**99.08%**	**99.06%**	**89.74%**	**91.49%**
α=0.5	β=0.5	void	22200290	18080	7148	21588	99.67%	99.79%	98.17%	98.81%	99.73%	99.47%
		alpha	25256	1278635	10618	85130	86.66%	91.35%	99.21%	99.51%	88.95%	92.81%
		beta	19777	18501	787234	112247	89.30%	83.95%	99.63%	99.41%	86.54%	86.67%
		rest	27371	160187	76514	1365824	86.18%	83.80%	99.11%	98.93%	84.97%	86.45%
Totals			22272694	1475403	881514	1584789	**90.46%**	**89.72%**	**99.03%**	**99.16%**	**90.05%**	**91.35%**
α=0.6	β=0.4	void	22200238	16562	9559	20747	99.67%	99.79%	98.13%	98.81%	99.73%	99.46%
		alpha	27957	1282624	12647	76411	86.78%	91.64%	99.21%	99.53%	89.15%	93.05%
		beta	18139	15978	795888	107754	88.23%	84.87%	99.58%	99.44%	86.52%	87.44%
		rest	28154	162794	83936	1355012	86.86%	83.13%	99.17%	98.89%	84.96%	85.89%
Totals			22274488	1477958	902030	1559924	**90.39%**	**89.86%**	**99.02%**	**99.17%**	**90.09%**	**91.46%**
α=0.7	β=0.3	void	22221734	8896	5405	11071	99.38%	99.89%	96.53%	99.34%	99.63%	99.22%
		alpha	49082	1254124	11515	84918	88.04%	89.60%	99.31%	99.41%	88.82%	91.38%
		beta	32897	14080	793886	96896	88.82%	84.66%	99.60%	99.43%	86.69%	87.27%
		rest	55793	147357	83044	1343702	87.45%	82.44%	99.22%	98.84%	84.87%	85.30%
Totals			22359506	1424457	893850	1536587	**90.92%**	**89.15%**	**98.67%**	**99.26%**	**90.00%**	**90.79%**
α=0.8	β=0.2	void	22223355	10629	3768	9354	98.48%	99.89%	91.35%	99.35%	99.18%	98.22%
		alpha	93745	1248517	5211	52166	89.91%	89.20%	99.44%	99.39%	89.55%	91.07%
		beta	90885	9019	755543	82312	92.24%	80.57%	99.75%	99.28%	86.01%	83.78%
		rest	158343	120505	54574	1296474	90.01%	79.54%	99.41%	98.65%	84.46%	82.84%
Totals			22566328	1388670	819096	1440306	**92.66%**	**87.30%**	**97.49%**	**99.17%**	**89.80%**	**88.98%**
α=0.9	β=0.1	void	22238161	4683	1437	2825	96.83%	99.96%	81.63%	99.72%	98.37%	96.42%
		alpha	182553	1181345	3399	32342	90.88%	84.40%	99.52%	99.12%	87.52%	87.04%
		beta	186147	8155	687822	55635	92.74%	73.35%	99.79%	99.02%	81.91%	77.44%
		rest	360280	105726	49044	1114846	92.47%	68.40%	99.63%	97.94%	78.63%	72.96%
Totals			22967141	1299909	741702	1205648	**93.23%**	**81.53%**	**95.14%**	**98.95%**	**86.61%**	**83.46%**

Table B.1: Confusion Matrices and Evaluation Metrics for Models Trained with Different Tversky Loss Parameters

Appendix C

SOURCE CODE

C.1 Data Collection and Preparation

C.1.1 generate_pdb_files.py

```
import global_variables as gv
import file_paths_helper
import os
import shutil
import re
import gzip
import sys
from urllib.request import urlopen
import multiprocessing as mp


# Check if _full, _alpha, _beta, _rest PDB files already exist in the pdb_id folder
# If one or more are missing, return true. If extra files exist, deletes them and returns
    true
# If all 4 files are present, returns false.
def check_and_cleanup(pdb_id):
    pdb_folder = os.path.join(gv.folder_pdb, pdb_id)
    if not os.path.exists(pdb_folder):
        return True
    pattern = re.compile("^[^\s]{4}_(beta|alpha|rest|full).pdb")
    counter = 0
    for the_file in os.listdir(pdb_folder):
        if pattern.match(the_file):
            counter += 1
            continue
        file_path = os.path.join(pdb_folder, the_file)
        try:
            if os.path.isfile(file_path):
                os.remove(file_path)
            elif os.path.isdir(file_path):
```

```
30                  shutil.rmtree(file_path)
31          except Exception as e:
32              print(e)
33  
34      if counter != 4:
35          shutil.rmtree(pdb_folder)
36          return True
37      else:
38          return False
39  
40  
41  def download_and_center(pdb_id):
42      max_x = -sys.maxsize - 1
43      max_y = -sys.maxsize - 1
44      max_z = -sys.maxsize - 1
45      min_x = sys.maxsize
46      min_y = sys.maxsize
47      min_z = sys.maxsize
48      atom_lines = []
49      alpha_range = []
50      beta_range = []
51      ftp_path = os.path.join(gv.folder_ftp_input, pdb_id[1:3] + '/pdb' + pdb_id + '.ent.gz')
52      with gzip.open(urlopen(ftp_path)) as fp:
53          for line in fp:
54              if line.startswith(b"HELIX"):
55                  line = line.decode("utf-8")
56                  start_index = int(line[21:25])
57                  end_index = int(line[33:37]) + 1
58                  alpha_range.extend(list(range(start_index, end_index)))
59                  continue
60              if line.startswith(b"SHEET"):
61                  line = line.decode("utf-8")
62                  start_index = int(line[22:26])
63                  end_index = int(line[33:37]) + 1
64                  beta_range.extend(list(range(start_index, end_index)))
65                  continue
66              if line.startswith(b"ATOM"):
67                  line = line.decode("utf-8")
68                  x_coord = float(line[30:38])
69                  y_coord = float(line[38:46])
70                  z_coord = float(line[46:54])
71                  if x_coord > max_x:
```

```
72                      max_x = x_coord
73                  if x_coord < min_x:
74                      min_x = x_coord
75                  if y_coord > max_y:
76                      max_y = y_coord
77                  if y_coord < min_y:
78                      min_y = y_coord
79                  if z_coord > max_z:
80                      max_z = z_coord
81                  if z_coord < min_z:
82                      min_z = z_coord
83                  atom_lines.append(line)
84      x_shift = (max_x + min_x) / 2
85      y_shift = (max_y + min_y) / 2
86      z_shift = (max_z + min_z) / 2
87
88      # skip generating pdb files that have coordinate value greater than +/-32.5 (0.5*gv.
        box_size) because the simulated map will be outside of the box
89      if (max_x - min_x > gv.box_size+1) or (max_y - min_y > gv.box_size+1) or (max_z - min_z
        > gv.box_size+1):
90          print("\tSkipped Generating " + pdb_id + ".pdb because it is too large")
91          return False
92      else:
93          center_pdb(pdb_id, atom_lines, alpha_range, beta_range, x_shift, y_shift, z_shift)
94          return True
95
96
97  def center_pdb(pdb_id, atom_lines, alpha_range, beta_range, x_shift, y_shift, z_shift):
98      # create the pdb_id folder:
99      pdb_folder = os.path.join(gv.folder_pdb, pdb_id)
100     file_paths_helper.check_make_dir(pdb_folder)
101     pdb_paths = file_paths_helper.generate_all_pdb_file_paths(pdb_id)
102     with open(pdb_paths["full"], 'a') as full_pdb:
103         with open(pdb_paths["alpha"], 'a') as alpha_pdb:
104             with open(pdb_paths["beta"], 'a') as beta_pdb:
105                 with open(pdb_paths["rest"], 'a') as rest_pdb:
106                     for line in atom_lines:
107                         x_coord = float(line[30:38])
108                         y_coord = float(line[38:46])
109                         z_coord = float(line[46:54])
110                         new_x_coord = "{:8.3f}".format(x_coord - x_shift)
111                         new_y_coord = "{:8.3f}".format(y_coord - y_shift)
```

```
112                         new_z_coord = "{:8.3f}".format(z_coord - z_shift)
113                         replacement = new_x_coord + new_y_coord + new_z_coord
114                         shifted_line = '%s%s%s' % (line[:30], replacement, line[54:])
115                         full_pdb.write(shifted_line)
116                         residue_number = int(line[22:26])
117                         if residue_number in alpha_range:
118                             alpha_pdb.write(shifted_line)
119                         elif residue_number in beta_range:
120                             beta_pdb.write(shifted_line)
121                         else:
122                             rest_pdb.write(shifted_line)
123     print("Generated " + pdb_id + ".pdb")
124 
125 def generate_pdb_files(unfiltered_pdb_list, final_pdb_list):
126     with open(final_pdb_list, "w+") as filtered_file:
127         with open(unfiltered_pdb_list) as unfiltered_file:
128             for line in unfiltered_file:
129                 if line.startswith('#'):
130                     continue
131                 pdb_id = line.rstrip('\n').lower()
132                 if not check_and_cleanup(pdb_id):
133                     print("\tSkipped Generating " + pdb_id + ".pdb because it already exists
    ")
134                     filtered_file.write(pdb_id + '\n')  # still write to filtered PDB List
135                     continue
136                 if download_and_center(pdb_id):
137                     filtered_file.write(pdb_id + '\n')  # write to new filtered PDB List
138 
139 if __name__ == "__main__":
140     file_paths_helper.check_make_dir(gv.folder_pdb)
141     processes = [mp.Process(target=generate_pdb_files, args=(gv.path_train_pdbs_unfiltered,
    gv.path_train_pdbs_list,)),
142                  mp.Process(target=generate_pdb_files, args=(gv.path_test_pdbs_unfiltered,
    gv.path_test_pdbs_list,))]
143     for p in processes:
144         p.start()
145     for p in processes:
146         p.join()
147 
148     print("Done generating pdbs!")
```

Listing C.1: generate_pdb_files.py

C.1.2 generate_mrc_files.py

```
1 import file_paths_helper
2 import os
3 import global_variables as gv
4 import multiprocessing as mp
5
6
7 def generate_mrc_files(pdb_list):
8     with open(pdb_list) as file:
9         for line in file:
10            if line.startswith('#'):
11                continue
12            pdb_id = line.rstrip('\n').lower()
13            simulate_map(pdb_id)
14
15
16 def simulate_map(pdb_id):
17     pdb_paths = file_paths_helper.generate_all_pdb_file_paths(pdb_id)
18     for resolution in gv.resolutions:
19         mrc_paths = file_paths_helper.generate_all_mrc_file_paths(resolution, pdb_id)
20         for key in pdb_paths:
21             os.system('/home/user/EMAN2/bin/e2pdb2mrc.py --apix={} --res={} --box={} {} {}'.
    format(gv.apix, resolution, gv.box_size, pdb_paths[key], mrc_paths[key]))
22
23
24 if __name__ == "__main__":
25     file_paths_helper.make_mrc_resolution_dirs()
26
27     processes = [mp.Process(target=generate_mrc_files, args=(gv.path_train_pdbs_list,)),
28                  mp.Process(target=generate_mrc_files, args=(gv.path_test_pdbs_list,))]
29
30     for p in processes:
31         p.start()
32
33     for p in processes:
34         p.join()
35
36     print("Done generating MRC files!")
```

Listing C.2: generate_mrc_files.py

C.1.3 create_hdf5.py

```
import file_paths_helper
import global_variables as gv
import h5py
import numpy as np
import random
import mrcfile
from copy import deepcopy
import multiprocessing as mp


def create_hdf5(mode):
    if mode == "train":
        pdb_list = gv.path_train_pdbs_list
        h5file_name = gv.path_train_data
    elif mode == "test":
        pdb_list = gv.path_test_pdbs_list
        h5file_name = gv.path_test_data
    else:
        raise ValueError("Mode can only be train or test")

    proteins = []
    with open(pdb_list) as file:
        for line in file:
            if line.startswith('#'):
                continue
            pdb_id = line.rstrip('\n').lower()
            proteins.append(pdb_id)

    num_proteins = len(proteins)
    dataset_shape = (num_proteins, gv.box_size, gv.box_size, gv.box_size)

    h5file = h5py.File(h5file_name, "w")
    h5file.create_dataset("pdb_id", (num_proteins,), dtype=h5py.special_dtype(vlen=str))
    h5file.create_dataset("resolution", (num_proteins,), 'uint8')
    h5file.create_dataset("protein_maps", dataset_shape, np.float32)
    h5file.create_dataset("ss_labels", dataset_shape, np.float32)

    fill_hdf5(proteins, h5file)


```

```
41 def fill_hdf5(pdb_id_list, h5file):
42     for index, pdb_id in enumerate(pdb_id_list):
43         # Choose a random resolution level
44         resolution = random.choice(gv.resolutions)
45         # Get the file paths to all mrc files
46         mrc_file_paths = file_paths_helper.generate_all_mrc_file_paths(resolution, pdb_id)
47         protein_map, ss_label = generate_ss_labels(mrc_file_paths)
48         h5file["pdb_id"][index] = pdb_id
49         h5file["resolution"][index] = resolution
50         h5file["protein_maps"][index] = protein_map
51         h5file["ss_labels"][index] = ss_label
52     h5file.close()
53
54
55 def generate_ss_labels(mrc_file_paths):
56     full = get_mrc_data(mrc_file_paths["full"])
57     alpha = get_mrc_data(mrc_file_paths["alpha"])
58     beta = get_mrc_data(mrc_file_paths["beta"])
59     rest = get_mrc_data(mrc_file_paths["rest"])
60
61     labeled_data = np.full((gv.box_size, gv.box_size, gv.box_size), 0)
62     labeled_data[np.logical_and(alpha > beta, alpha > rest)] = gv.sse_to_class_label["alpha"
       ]
63     labeled_data[np.logical_and(beta > alpha, beta > rest)] = gv.sse_to_class_label["beta"]
64     labeled_data[np.logical_and(rest > alpha, rest > beta)] = gv.sse_to_class_label["rest"]
65
66     return full, labeled_data
67
68
69 def get_mrc_data(mrc_file_path):
70     # 1. Open the mrc file and deepcopy the data
71     map_file = mrcfile.open(mrc_file_path, mode='r')
72     min_value = map_file.header.dmin
73     max_value = map_file.header.dmax
74     data = deepcopy(map_file.data)
75     map_file.close()
76     # 2. Feature scaling (Min-Max Normalization)
77     data = (data - min_value) / (max_value - min_value)
78     # 3. Threshold value
79     data[data < gv.threshold] = 0
80     return data
81
```

```
82
83 if __name__ == "__main__":
84     processes = [mp.Process(target=create_hdf5, args=("train",)),
85                  mp.Process(target=create_hdf5, args=("test",))]
86
87     for p in processes:
88         p.start()
89
90     for p in processes:
91         p.join()
92
93     print("Done generating HDF5 files!")
```

Listing C.3: generate_hdf5.py

C.2 The Proposed Deep Model

C.2.1 3D_UNet.py

```
1  import global_variables as gv
2  import tensorflow as tf
3  from tensorflow.python.framework import ops
4  import loss_functions
5
6
7  def accuracy_per_class(labels, predictions, class_id, metrics_collections=None,
       updates_collections=None):
8      num_correct_predicted = tf.reduce_sum(
9          tf.cast(tf.logical_and(tf.equal(predictions, class_id), tf.equal(labels, class_id)),
        tf.int32))
10     category_accuracy = num_correct_predicted / tf.reduce_sum(tf.cast(tf.equal(labels,
       class_id), tf.int32))
11
12     mean_category_accuracy, update_category_op = tf.metrics.mean(category_accuracy)
13     if metrics_collections:
14         ops.add_to_collections(metrics_collections, mean_category_accuracy)
15     if updates_collections:
16         ops.add_to_collections(updates_collections, update_category_op)
17
18     return mean_category_accuracy, update_category_op
19
20
21 def network(features, labels, mode, params):
22     # Input Layer
23     # Reshape features to 5-D tensor: [batch_size, width, height, length, channels]
24     # Protein maps are 64x64x64, and have one channel
25     input_layer = tf.reshape(features, [-1, gv.box_size, gv.box_size, gv.box_size, 1])
26
27     # Convolution Block #1
28     conv1 = tf.layers.conv3d(inputs=input_layer, filters=32, kernel_size=(4, 4, 4), padding=
       "same", activation=tf.nn.relu)
29     conv1 = tf.layers.conv3d(inputs=conv1, filters=32, kernel_size=(4, 4, 4), padding="same"
       , activation=tf.nn.relu)
30     pool1 = tf.layers.max_pooling3d(inputs=conv1, pool_size=(2, 2, 2), strides=2)
31
32     # Convolution Block #2
33     conv2 = tf.layers.conv3d(inputs=pool1, filters=64, kernel_size=(4, 4, 4), padding="same"
```

```
    , activation=tf.nn.relu)
34  conv2 = tf.layers.conv3d(inputs=conv2, filters=64, kernel_size=(4, 4, 4), padding="same"
    , activation=tf.nn.relu)
35  pool2 = tf.layers.max_pooling3d(inputs=conv2, pool_size=(2, 2, 2), strides=2)
36
37  # Convolution Block #3
38  conv3 = tf.layers.conv3d(inputs=pool2, filters=128, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
39  conv3 = tf.layers.conv3d(inputs=conv3, filters=128, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
40  pool3 = tf.layers.max_pooling3d(inputs=conv3, pool_size=(2, 2, 2), strides=2)
41
42  # Convolution Block #4
43  conv4 = tf.layers.conv3d(inputs=pool3, filters=256, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
44  conv4 = tf.layers.conv3d(inputs=conv4, filters=256, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
45  pool4 = tf.layers.max_pooling3d(inputs=conv4, pool_size=(2, 2, 2), strides=2)
46
47  # Convolution Block #5
48  conv5 = tf.layers.conv3d(inputs=pool4, filters=512, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
49  conv5 = tf.layers.conv3d(inputs=conv5, filters=512, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
50
51  deconv1 = tf.layers.conv3d_transpose(inputs=conv5, filters=256, kernel_size=(3, 3, 3),
    strides=(2, 2, 2),
52                                       padding="same")
53  up6 = tf.concat([deconv1, conv4], axis=4)
54  conv6 = tf.layers.conv3d(inputs=up6, filters=256, kernel_size=(4, 4, 4), padding="same",
     activation=tf.nn.relu)
55  conv6 = tf.layers.conv3d(inputs=conv6, filters=256, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
56
57  deconv2 = tf.layers.conv3d_transpose(inputs=conv6, filters=128, kernel_size=(3, 3, 3),
    strides=(2, 2, 2), padding="same")
58  up7 = tf.concat([deconv2, conv3], axis=4)
59  conv7 = tf.layers.conv3d(inputs=up7, filters=128, kernel_size=(4, 4, 4), padding="same",
     activation=tf.nn.relu)
60  conv7 = tf.layers.conv3d(inputs=conv7, filters=128, kernel_size=(4, 4, 4), padding="same
    ", activation=tf.nn.relu)
61
```

```
62     deconv3 = tf.layers.conv3d_transpose(inputs=conv7, filters=64, kernel_size=(3, 3, 3),
       strides=(2, 2, 2), padding="same")
63     up8 = tf.concat([deconv3, conv2], axis=4)
64     conv8 = tf.layers.conv3d(inputs=up8, filters=64, kernel_size=(4, 4, 4), padding="same",
       activation=tf.nn.relu)
65     conv8 = tf.layers.conv3d(inputs=conv8, filters=64, kernel_size=(4, 4, 4), padding="same"
       , activation=tf.nn.relu)
66
67     deconv4 = tf.layers.conv3d_transpose(inputs=conv8, filters=32, kernel_size=(3, 3, 3),
       strides=(2, 2, 2),
68                                          padding="same")
69     up9 = tf.concat([deconv4, conv1], axis=4)
70     conv9 = tf.layers.conv3d(inputs=up9, filters=32, kernel_size=(4, 4, 4), padding="same",
       activation=tf.nn.relu)
71     conv9 = tf.layers.conv3d(inputs=conv9, filters=32, kernel_size=(4, 4, 4), padding="same"
       , activation=tf.nn.relu)
72
73     logits = tf.layers.conv3d(inputs=conv9, filters=4, kernel_size=(1, 1, 1), activation=tf.
       nn.softmax)
74
75     predicted_classes = tf.argmax(input=logits, axis=4)
76
77     # Prediction block:
78     if mode == tf.estimator.ModeKeys.PREDICT:
79         predictions = {"classes": predicted_classes,
80                        "probabilities": tf.nn.softmax(logits, name="softmax_tensor"),
81                        "logits": logits
82                        }
83         return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)
84
85     # Calculate loss and accuracy (for both TRAIN and EVAL modes)
86     loss = loss_functions.tversky_loss(labels, logits)
87     accuracy = tf.metrics.accuracy(labels=labels, predictions=predicted_classes, name='
       acc_op')
88
89     # Compute Evaluation Metrics
90     if mode == tf.estimator.ModeKeys.EVAL:
91         tf.summary.scalar('eval_accuracy', accuracy[1])
92         mean_per_class_accuracy = tf.metrics.mean_per_class_accuracy(labels,
       predicted_classes, num_classes=4)
93         void_class_accuracy = accuracy_per_class(labels, predicted_classes, class_id=0)
94         alpha_class_accuracy = accuracy_per_class(labels, predicted_classes, class_id=1)
```

```
95         beta_class_accuracy = accuracy_per_class(labels, predicted_classes, class_id=2)
96         rest_class_accuracy = accuracy_per_class(labels, predicted_classes, class_id=3)
97         metrics = {'eval_accuracy': accuracy,
98                    'mean_per_class_accuracy': mean_per_class_accuracy,
99                    'mean_alpha_accuracy': alpha_class_accuracy,
100                   'mean_beta_accuracy': beta_class_accuracy,
101                   'mean_rest_accuracy': rest_class_accuracy,
102                   'mean_void_accuracy': void_class_accuracy
103                   }
104        return tf.estimator.EstimatorSpec(mode=mode, loss=loss, eval_metric_ops=metrics)
105
106    assert mode == tf.estimator.ModeKeys.TRAIN
107    # Compute Batch Metrics during training:
108    num_correct_void_predicted = tf.reduce_sum(
109        tf.cast(tf.logical_and(tf.equal(predicted_classes, 0), tf.equal(labels, 0)), tf.
    int32))
110    num_correct_alpha_predicted = tf.reduce_sum(
111        tf.cast(tf.logical_and(tf.equal(predicted_classes, 1), tf.equal(labels, 1)), tf.
    int32))
112    num_correct_beta_predicted = tf.reduce_sum(
113        tf.cast(tf.logical_and(tf.equal(predicted_classes, 2), tf.equal(labels, 2)), tf.
    int32))
114    num_correct_rest_predicted = tf.reduce_sum(
115        tf.cast(tf.logical_and(tf.equal(predicted_classes, 3), tf.equal(labels, 3)), tf.
    int32))
116    batch_void_accuracy = num_correct_void_predicted / tf.reduce_sum(tf.cast(tf.equal(labels
    , 0), tf.int32))
117    batch_alpha_accuracy = num_correct_alpha_predicted / tf.reduce_sum(tf.cast(tf.equal(
    labels, 1), tf.int32))
118    batch_beta_accuracy = num_correct_beta_predicted / tf.reduce_sum(tf.cast(tf.equal(labels
    , 2), tf.int32))
119    batch_rest_accuracy = num_correct_rest_predicted / tf.reduce_sum(tf.cast(tf.equal(labels
    , 3), tf.int32))
120
121    tf.summary.scalar('batch_total_accuracy', accuracy[1])
122    tf.summary.scalar('batch_alpha_accuracy', batch_alpha_accuracy)
123    tf.summary.scalar('batch_beta_accuracy', batch_beta_accuracy)
124    tf.summary.scalar('batch_rest_accuracy', batch_rest_accuracy)
125    tf.summary.scalar('batch_void_accuracy', batch_void_accuracy)
126
127    # Configure the Train Op (for TRAIN mode)
128    if params['optimizer'] == 'GradientDescent':
```

```
129         optimizer = tf.train.GradientDescentOptimizer(learning_rate=params['learning_rate'])
130     elif params['optimizer'] == 'Adagrad':
131         optimizer = tf.train.AdagradOptimizer(learning_rate=params['learning_rate'])
132     else:
133         optimizer = tf.train.AdamOptimizer(learning_rate=params['learning_rate'])
134     train_op = optimizer.minimize(loss=loss, global_step=tf.train.get_global_step())
135     return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

Listing C.4: 3D_UNet.py

C.2.2 *data_pipeline.py*

```
import tensorflow as tf
import h5py
import global_variables as gv


class Generator:
    def __init__(self, file):
        self.file = file

    def __call__(self):
        with h5py.File(self.file, 'r') as hf:
            for map, label in zip(hf["protein_maps"], hf["ss_labels"]):
                yield map, label


def train_input_fn(batch_size):
    dataset = tf.data.Dataset.from_generator(Generator(gv.path_train_data), (tf.float32, tf.
    float32), (tf.TensorShape([gv.box_size, gv.box_size, gv.box_size]), tf.TensorShape([gv.
    box_size, gv.box_size, gv.box_size])))

    dataset = dataset.apply(tf.contrib.data.shuffle_and_repeat(buffer_size=250))
    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(1)
    return dataset


def eval_input_fn(batch_size):
    dataset = tf.data.Dataset.from_generator(Generator(gv.path_test_data), (tf.float32, tf.
    float32), (tf.TensorShape([gv.box_size, gv.box_size, gv.box_size]), tf.TensorShape([gv.
    box_size, gv.box_size, gv.box_size])))

    dataset = dataset.batch(batch_size)
    dataset = dataset.prefetch(1)
    return dataset
```

Listing C.5: data_pipeline.py

C.2.3 loss_functions.py

```python
import tensorflow as tf


def tversky_loss(y_true, y_pred):
    """Calculates tversky loss value for multi-class. Tversky index is calculated for each
    class separately and then summed.
    :param y_true: ground-truth labels
    :param y_pred: predicted labels
    :return: Tversky loss; range is between 0 and num_classes
    :Source: E. Moebel https://github.com/keras-team/keras/issues/9395
    """
    # alpha=beta=0.5 : dice coefficient
    # alpha=beta=1   : tanimoto coefficient (also known as jaccard)
    # alpha+beta=1   : produces set of F*-scores
    alpha = 0.2
    beta = 0.8

    # convert y_true to one-hot encoding
    y_true = tf.one_hot(tf.cast(y_true, tf.int32), 4)

    ones = tf.ones(tf.shape(y_true))
    p0 = y_pred  # probability that voxels are class i
    p1 = ones - y_pred  # probability that voxels are not class i
    g0 = y_true # ground truth probability that voxels are class i
    g1 = ones - y_true # ground truth probability that voxels are not class i

    numerator = tf.reduce_sum(p0 * g0, (0, 1, 2, 3))
    denominator = numerator + alpha * tf.reduce_sum(p0 * g1, (0, 1, 2, 3)) + beta * tf.
    reduce_sum(p1 * g0, (0, 1, 2, 3))

    # sum over the Tversky index for each class to get a single number
    tversky_index = tf.reduce_sum(numerator / denominator)

    num_classes = tf.cast(tf.shape(y_true)[-1], 'float32')
    return num_classes - tversky_index
```

Listing C.6: loss_functions.py

C.2.4 train.py

```
import argparse
import sys
import global_variables as gv
import importlib
import os
from datetime import datetime
import tensorflow as tf
import data_pipeline as data_pipeline
import pathlib

sys.path.append(gv.BASE_DIR)
sys.path.append(gv.folder_models)

parser = argparse.ArgumentParser()
parser.add_argument('--model', default='3D_UNet', help='Model name: 3D_UNet or ??? [default:
    3D U-Net]')
parser.add_argument('--optimizer', default='Adam', help='Adam, GradientDescent or Adagrad [
    default: Adam]')
parser.add_argument('--learning_rate', default=0.0001, type=float, help='Learning Rate for
    training optimizer [default: 0.0001')
parser.add_argument('--batch_size', type=int, default=20, help='Integer number of samples
    per batch [default:10')
ARGS = parser.parse_args()

MODEL = importlib.import_module(ARGS.model)  # import network model function
MODEL_FILE = os.path.join(gv.folder_models, ARGS.model + '.py')
LOSS_FILE = os.path.join(gv.BASE_DIR, 'loss_functions.py')

MODEL_DIR = os.path.join(gv.BASE_DIR, "tmp", datetime.now().strftime("%Y%m%d-%H%M%S"))
if not os.path.exists(MODEL_DIR):
    pathlib.Path(MODEL_DIR).mkdir(parents=True, exist_ok=True)
os.system('cp %s %s' % (MODEL_FILE, MODEL_DIR))  # bkp of model function
os.system('cp %s %s' % (LOSS_FILE, MODEL_DIR))
os.system('cp train.py %s' % MODEL_DIR)  # bkp of train procedure
os.system('cp data_pipeline.py %s' % MODEL_DIR)  # bkp of data pipeline

LOG_FOUT = open(os.path.join(MODEL_DIR, 'log_train.txt'), 'w')
LOG_FOUT.write(str(ARGS) + '\n')
LOG_FOUT.flush()

```

```
37
38 def log_string(out_str):
39     LOG_FOUT.write(out_str + '\n')
40     LOG_FOUT.flush()
41     print(out_str)
42
43
44 def train():
45     voxel_classifier = tf.estimator.Estimator(model_fn=MODEL.network, params={'optimizer':
       ARGS.optimizer, 'learning_rate': ARGS.learning_rate},
46                                               model_dir=MODEL_DIR)
47
48     # Test while train
49     evaluator = tf.contrib.estimator.InMemoryEvaluatorHook(estimator=voxel_classifier,
       input_fn=lambda: data_pipeline.eval_input_fn(batch_size=ARGS.batch_size), every_n_iter
       =500)
50
51     voxel_classifier.train(input_fn=lambda: data_pipeline.train_input_fn(batch_size=ARGS.
       batch_size), max_steps=15001, hooks=[evaluator])
52
53
54 if __name__ == "__main__":
55     tf.logging.set_verbosity(tf.logging.INFO)
56     tf.app.run(train)
57     LOG_FOUT.close()
```

Listing C.7: train.py

C.3 SW Tool

C.3.1 ss_predictor.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Given a path to a protein density map file (.mrc or .map), predicts secondary structure
elements using a trained 3D U-Net segmentation model. Writes 3 segmented map
files to a "predictions" folder: one file for each of the 3 SSE: alpha-helix, beta-sheet,
turns/loops.
This script is used to compile ss_predictor.exe with PyInstaller.
"""
__author__ = "Todor Avramov"

from copy import deepcopy
import global_variables as gv
import importlib
import mrcfile as mrc
import numpy as np
import operator
import os
import sys
import tensorflow as tf

def main(argv):
    if len(argv) != 2:
        print("Usage: ss_predictor.py mrc_file")
        sys.exit(-1)

    model_dir = gv.folder_trained_models
    sys.path.append(gv.BASE_DIR)
    sys.path.append(model_dir)

    # Load Data
    filename = os.path.basename(argv[1])
    pdb_id, extension = os.path.splitext(filename)

    map_file = mrc.open(argv[1], mode='r')
    min_density_value = map_file.header.dmin
    max_density_value = map_file.header.dmax
    x_start = map_file.header.nxstart
```

```
39     y_start = map_file.header.nystart
40     z_start = map_file.header.nzstart
41     voxel_size = map_file.voxel_size
42     num_voxels_x = map_file.header.nx
43     num_voxels_y = map_file.header.ny
44     num_voxels_z = map_file.header.nz
45 
46     data = deepcopy(map_file.data)
47     # 2. Normalize data
48     data = (data - min_density_value) / (max_density_value - min_density_value)
49     # 3. Change all values < threshold to 0
50     threshold = (0.02 - min_density_value) / (max_density_value - min_density_value)
51     data[data < threshold] = 0
52 
53     if not (num_voxels_x == num_voxels_y == num_voxels_z):
54         print("This is not a cube!!")
55         exit(-1)
56 
57     # all 3 dimensions are equal, we have a BOX!
58     multiplier = num_voxels_x // 64
59     remainder = num_voxels_x % 64
60     reconstruction_pad_width = 0
61 
62     if remainder == 0:
63         if multiplier == 1:
64             cropped_data = data
65         else:
66             cropped_data = central_crop(data, (multiplier * 64, multiplier * 64, multiplier
       * 64))
67     elif remainder <= 32:
68         # central crop
69         cropped_data = central_crop(data, (multiplier*64, multiplier*64, multiplier*64))
70         if remainder % 2 == 0:
71             reconstruction_pad_width = int(remainder/2)
72         else:
73             value = 64 - remainder
74             reconstruction_pad_width = [1 + value // 2, value // 2]
75     else:
76         assert(remainder > 32)
77         # pad with zero
78         if remainder % 2 == 0:
79             pad_width = int(remainder/2)
```

```
80                cropped_data = np.pad(data, pad_width, mode='constant')
81            else:
82                value = 64 - remainder
83                pad_width = [1 + value // 2, value // 2]
84                cropped_data = np.pad(data, pad_width, mode='constant')
85
86        print(cropped_data.shape)
87        predict_data = cubify(cropped_data, (64, 64, 64))
88        print(predict_data.shape)
89
90        # Create the Estimator
91        model_function = importlib.import_module("3D_UNet")  # import network model function
92        voxel_classifier = tf.estimator.Estimator(model_fn=model_function.network, model_dir=
          model_dir)
93
94        predict_input_fn = tf.estimator.inputs.numpy_input_fn(
95            x=predict_data,
96            y=None,
97            batch_size=1,
98            num_epochs=1,
99            shuffle=False)
100       predictions = voxel_classifier.predict(predict_input_fn)
101
102       map_number = 0
103       all_cubes_alpha = []
104       all_cubes_beta = []
105       all_cubes_rest = []
106       for prediction in predictions:
107           data = prediction["classes"].astype('float32')
108           for class_id in list(gv.sse_to_class_label.values()):
109               hits = np.equal(data, class_id).astype(int)
110               per_type = np.multiply(predict_data[map_number], hits).astype(np.float32)
111               if class_id == 1:
112                   all_cubes_alpha.append(per_type)
113               elif class_id == 2:
114                   all_cubes_beta.append(per_type)
115               elif class_id == 3:
116                   all_cubes_rest.append(per_type)
117               else:
118                   print("Invalid class_id=", class_id)
119                   sys.exit(-1)
120           map_number += 1
```

```
121     print("Done")
122
123     path = os.path.join(gv.folder_predictions, pdb_id)
124     if not os.path.exists(path):
125         os.makedirs(path)
126     combined = uncubify(np.asarray(all_cubes_alpha), (multiplier*64, multiplier*64,
        multiplier*64))
127     combined = np.pad(combined, reconstruction_pad_width, mode='constant')
128     newmrc = mrc.new(os.path.join(path, pdb_id + '_alpha_classified.map'), overwrite=True)
129     newmrc.set_data(combined)
130     newmrc.header.nxstart = x_start
131     newmrc.header.nystart = y_start
132     newmrc.header.nzstart = z_start
133     newmrc.voxel_size = voxel_size
134     newmrc.update_header_from_data()
135     newmrc.close()
136
137     combined = uncubify(np.asarray(all_cubes_beta), (multiplier*64, multiplier*64,
        multiplier*64))
138     combined = np.pad(combined, reconstruction_pad_width, mode='constant')
139     newmrc = mrc.new(os.path.join(path, pdb_id + '_beta_classified.map'), overwrite=True)
140     newmrc.set_data(combined)
141     newmrc.header.nxstart = x_start
142     newmrc.header.nystart = y_start
143     newmrc.header.nzstart = z_start
144     newmrc.voxel_size = voxel_size
145     newmrc.update_header_from_data()
146     newmrc.close()
147     print("Done")
148
149     combined = uncubify(np.asarray(all_cubes_rest), (multiplier*64, multiplier*64,
        multiplier*64))
150     combined = np.pad(combined, reconstruction_pad_width, mode='constant')
151     newmrc = mrc.new(os.path.join(path, pdb_id + '_rest_classified.map'), overwrite=True)
152     newmrc.set_data(combined)
153     newmrc.header.nxstart = x_start
154     newmrc.header.nystart = y_start
155     newmrc.header.nzstart = z_start
156     newmrc.voxel_size = voxel_size
157     newmrc.update_header_from_data()
158     newmrc.close()
159     print("Done")
```

```
160
161 def cubify(arr, newshape):
162     """Splits a larger array into multiple smaller arrays of shape newshape.
163     :param arr: array of shape (W,H,D)
164     :param newshape: shape (w,h,d)
165     :return: array of shape (N,w,h,d)
166     :href: https://stackoverflow.com/questions/42297115/numpy-split-cube-into-cubes
        /42298440#42298440
167     """
168     oldshape = np.array(arr.shape)
169     repeats = (oldshape / newshape).astype(int)
170     tmpshape = np.column_stack([repeats, newshape]).ravel()
171     order = np.arange(len(tmpshape))
172     order = np.concatenate([order[::2], order[1::2]])
173     # newshape must divide oldshape evenly or else ValueError will be raised
174     return arr.reshape(tmpshape).transpose(order).reshape(-1, *newshape)
175
176 def uncubify(arr, oldshape):
177     """Combines smaller array into one larger array. Opposite of what cubify does
178     :param arr: array of shape (N,w,h,d)
179     :param oldshape: shape (W,H,D)
180     :return: array of shape (W,H,D)
181     :href: https://stackoverflow.com/questions/42297115/numpy-split-cube-into-cubes
        /42298440#42298440
182     """
183     number_of_cubes, newshape = arr.shape[0], arr.shape[1:]
184     oldshape = np.array(oldshape)
185     repeats = (oldshape / newshape).astype(int)
186     tmpshape = np.concatenate([repeats, newshape])
187     order = np.arange(len(tmpshape)).reshape(2, -1).ravel(order='F')
188     return arr.reshape(tmpshape).transpose(order).reshape(oldshape)
189
190 def central_crop(img, bounding):
191     start = tuple(map(lambda a, da: a // 2 - da // 2, img.shape, bounding))
192     end = tuple(map(operator.add, start, bounding))
193     slices = tuple(map(slice, start, end))
194     return img[slices]
195
196 if __name__ == "__main__":
197     tf.app.run(argv=sys.argv)
```

Listing C.8: ss_predictor.py

C.3.2 Chimera Extension

```
import re
import chimera

from SSE_Predictor import *
```

Listing C.9: init.py

```
import chimera.extension

class SSEPredictor_EMO(chimera.extension.EMO):
    def name(self):
        return 'SS Predictor'
    def description(self):
        return 'Predicts Secondary Structure Elements in Cryo-EM Protein Maps'
    def categories(self):
        return ['Utilities']
    def icon(self):
        return self.path('ss_icon.png')
    def activate(self):
        self.module('gui').show_sse_predictor_dialog()

chimera.extension.manager.registerExtension(SSEPredictor_EMO(__file__))
```

Listing C.10: ChimeraExtension.py

```
1  import warnings
2  import chimera
3  from chimera.baseDialog import ModelessDialog
4  import VolumeViewer
5
6  class SSE_Predictor_Dialog(ModelessDialog):
7      title = 'SS Predictor'
8      name = 'SS Predictor'
9      buttons = ('Predict', 'Close',)
10     help = 'www.google.com'
11
12     def fillInUI(self, parent):
13         self.toplevel_widget = parent.winfo_toplevel()
14         self.toplevel_widget.withdraw()
15
16         parent.columnconfigure(0, weight = 1)
17
18         from CGLtk import Hybrid
19         import Pmw, Tkinter
20
21         from chimera.widgets import ModelOptionMenu
22         self.modelMenu = ModelOptionMenu(parent, labelpos="w",
23                                 label_text="Select MRC:")
24         self.modelMenu.grid(row = 1, column = 0, sticky = 'w')
25
26     def Predict(self):
27         import SSE_Predictor as predictor
28
29
30         if self.modelMenu.getvalue() is None or type(self.modelMenu.getvalue()) is not
       VolumeViewer.volume.Volume:
31           warnings.warn('Must select a valid MRC map.')
32           return
33
34
35         mrc_loaded_num = self.modelMenu.getvalue().id
36         path_mrc = str(chimera.openModels.list()[mrc_loaded_num].openedAs[0])
37         print path_mrc
38         predictor.predict(path_mrc)
39
40 def movement_mode_dialog(create = 0):
41   from chimera import dialogs
```

```python
42   return dialogs.find(SSE_Predictor_Dialog.name, create=create)
43
44 def show_sse_predictor_dialog():
45   from chimera import dialogs
46   return dialogs.display(SSE_Predictor_Dialog.name)
47
48 from chimera import dialogs
49 dialogs.register(SSE_Predictor_Dialog.name,
50                  SSE_Predictor_Dialog, replace = 1)
```

Listing C.11: gui.py

```python
1  import sys
2  import os
3  import subprocess
4  from chimera import runCommand
5
6  def predict(mrc_loaded_path): #, contour_level):
7
8      # Define the path to the executable.
9      current_folder = os.path.dirname(os.path.abspath(__file__))
10     binary_path = os.path.join(current_folder, 'ss_predictor')
11     exe_path = os.path.join(binary_path,'ss_predictor.exe')
12
13     # Set up args to pass to binary
14     args = (str(exe_path), mrc_loaded_path)
15
16     # Run the binary to predict map, wait for it to finish, print output
17     p = subprocess.Popen(args, stdout=subprocess.PIPE)
18     p.wait()
19     output = p.stdout.read()
20     print(output)
21
22     base = os.path.basename(mrc_loaded_path)
23     pdb_id, extension = os.path.splitext(base)
24
25     predictions_folder = os.path.join(binary_path, 'predictions', pdb_id)
26     map_files = [f for f in os.listdir(predictions_folder) if os.path.isfile(os.path.join(
       predictions_folder,f))]
27     for map_file in map_files:
28         runCommand('open ' + os.path.join(predictions_folder, map_file))
```

Listing C.12: SSE_Predictor.py