MACHINE LEARNING FOR SECONDARY STRUCTURE ELEMENTS CLASSIFICATION OF AMINO ACIDS IN PROTEINS

A SENIOR PROJECT

SUBMITTED TO THE COLLEGE OF ENGINEERING

OF TENNESSEE STATE UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

JARED BAUMANN, MINA WAHIB, AND ROSALINE TEP

DECEMBER 2021

To the College of Engineering,

We are submitting a senior project by ”Jared Baumann, Mina Wahib, and Rosaline Tep” entitled ”Machine Learning for Secondary Structure Elements Classification of Amino Acids in Proteins”. We recommend that it be accepted in partial fulfillment of the requirements for the degree, Bachelor of Science in Computer Science.

Technical Advisor, Dr. Ali Sekmen

Course Instructor, Dr. Ali Sekmen

Department Chair, Dr. Ali Sekmen

Accepted for the College of Engineering:

S. Keith Hargrove, Ph.D.

Dean of the College of Engineering

To my friends and family,

my support system.

ABSTRACT

This project serves to implement neural network techniques for inferring secondary structure classes from data about the Cα backbone of an amino acid. It's purpose is not to determine the entire protein structure, but instead serve as a stepping stone for these efforts. This system is implement such that it will be easy to utilize, and provide fast and accurate predictions about secondary structures. In order to accomplish this task, we implemented multiple networks that can be used to predict structures, including Fully Connected Neural Networks (FCNNs) and Deep Convolutional Neural Networks (DCNNs) to make predictions both for the secondary structure classification of a single amino acid through its Cα characteristics as well as the transition type if applicable. For both cases were where able to exceed 80% accuracy, with the simple secondary structure classification reaching an accuracy of 93%, and the transition type reaching an accuracy of 84%.

ACKNOWLEDGMENTS

We would like to thank our advisor, Dr. Ali Sekmen, for providing steadfast support and motivation to assume this work. We thank him for trusting in our abilities, work ethic, and having patience to work with us throughout the project. Without the help of him, this project would not have been able to be accomplished. Thank you for everything you have given us, it is more than appreciated. This work was partially supported by DOD grant W911NF-20-100284. We thank the Department of Defense for this support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

CHAPTER 1

INTRODUCTION

In biology, it is pivotally important to know the structure of proteins. However, the primary structure is all that can be established with certainty at the moment, with secondary, tertiary, and quaternary structures being out of reach in the vast majority of situations due to the significant computing effort necessary to derive them. Whilst tertiary and onward are still out of reach, derivation of secondary structure from existing data via the utilization of machine learning techniques could prove viable. This report explores the protein-structure prediction capabilities of different machine learning techniques.

1.1 Motivation

With the wide gap in the amount of proteins with known secondary structures and those with only their primary structures known, the need for a way to visualize and predict the higher level structures of proteins expands. This is a difficult task to take on for humans. However, with machine learning techniques, like deep convolutional neural networks (DCNNs) that are designed to predict output using structures meant to simulate the function of the human brain, the secondary structures of proteins can be predicted more accurately since their primary structures are already known. This gap, as well as the vast potential of medical advancements and lack of capability, are the motivations behind using machine learning techniques in order to predict secondary protein structures.

1.2 Problem Statement

Proteins are fundamental for the principal physiological operations of life within every system in the human body. Proteins are made up of long amino acid chains that are crucial for many different tasks such as synthesizing DNA, sending chemical signals, and providing structural support [8]. As modern medicine becomes increasingly important, the need to understand the structures that amino acids make up in proteins grows as well. Predicting these structures can vastly assist in visualizing the three-dimensional models of the proteins. This prediction can aid in determining protein shape which directly helps to understand a protein's specific function. The shape and function of a protein are key to unlocking immense medical potential, such as creating new proteins to further the aid of cancer treatment [7]. In this project, a machine learning based system will be developed to determine the secondary structure of each amino acid in a protein.

1.3 Literature Review

This section provides some terminology and summarizes some of the existing work related to the project. To begin, the requisite biological components are discussed, followed by the machine learning fundamentals.

1.3.1 DNA

DNA is short for deoxyribonucleic acid, and it is the molecule that contains the genetic code of organisms. It can be found in all living cells, from plant and animal cells to those in protists and bacteria.DNA can be found in each cell in the organism and provides the cells a "blueprint" for each protein it needs to make. This code is described three bases at a time by your cells in order to build proteins that are required for growth and survival. A gene is a sequence of DNA that contains the instructions for making a protein. Each set of three of base pairs indicated a different amino acid, which are the building blocks of proteins. The base pairs T-G-G, for example, designate the amino acid tryptophan, while the base pairs G-G-G designate the amino acid glutamine [2]. DNA has a

double helix structure (Like shown in Figure 1.1) and has four nitrogenous bases called Adenine, Thymine, Guanine, and Cytosine. When a nucleotide binds to its corresponding nucleotide that is called a base pair. These pairs sit on a sugar-phosphate backbone.

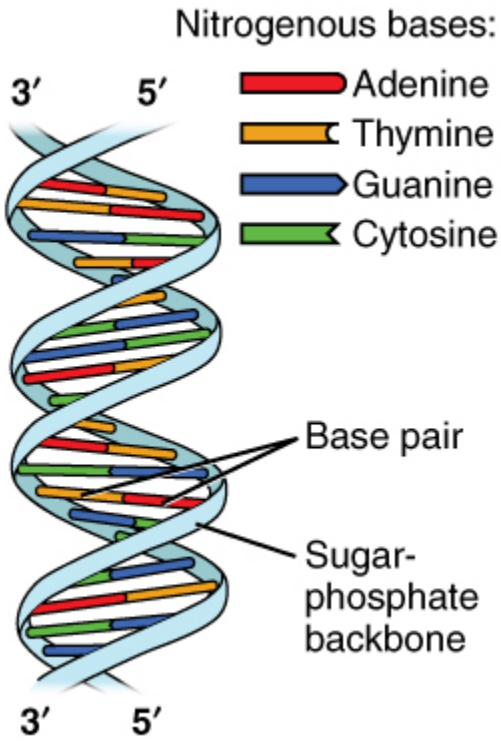

Figure 1.1: DNA double helix

1.3.2 Genetics/mRNA

Genetics is the study of genes and inheritance in living organisms. This part of science has an interesting history, extending from the nineteenth century when researchers started to concentrate on how creatures acquired characteristics from their parents, to the current day when the "source code" of living things can be perused letter-by-letter [3]. Messenger RNA (mRNA) is a sub-type of RNA. An mRNA molecule carries a portion of the DNA code to other parts of the cell for processing. mRNA is created during transcription. During the transcription process, a single strand of DNA is decoded by RNA polymerise, and mRNA is synthesized.

1.3.3 Proteins/Amino Acids

Proteins are large, complex molecules that play many critical roles in the body. They do most of the work in cells and are required for the structure, function, and regulation of the body's tissues and organs. Amino acids are organic compounds that combine to form proteins. Amino acids and proteins are the building blocks of life. When proteins are digested or broken down, amino acids are left. The human body uses amino acids to make proteins to help repair body tissue.

1.3.4 Structures of Proteins: Primary and Secondary

The primary structure of a protein alludes to the sequence of amino acids in the polypeptide chain. The primary structure is held together by peptide bonds that are made during the cycle of protein biosynthesis. The secondary structure of a protein is whatever regular structures occur as the polypeptide folds into its functional three-dimensional form due to interactions between surrounding or nearby amino acids. Hydrogen bonds (often shortened to H-bonds) develop between local groups of amino acids in a section of the polypeptide chain, resulting in secondary structures. The α-helix and β-pleated sheet form because of hydrogen bonding between carbonyl and amino groups in the peptide backbone [8]. Example of an α-helix and a β-sheet can be seen in Figure 1.2.

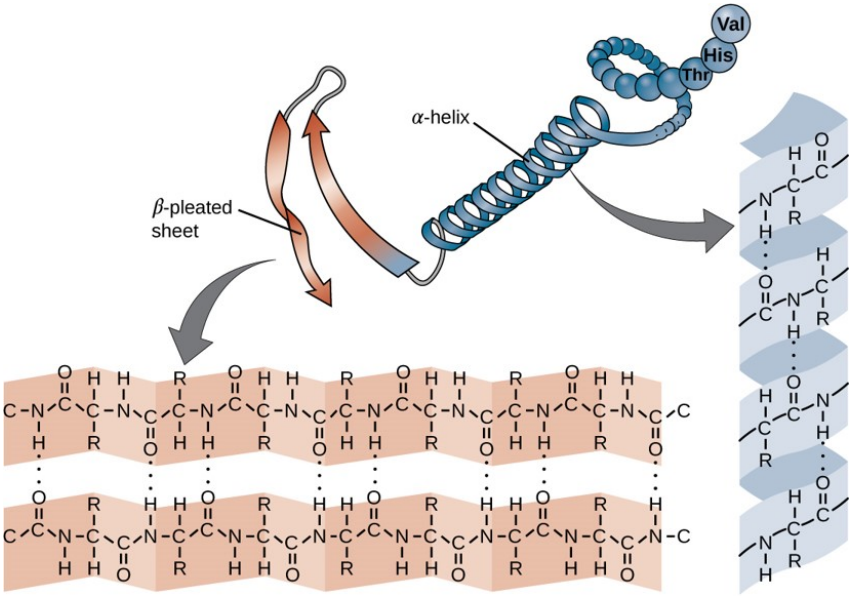

Figure 1.2: Protein Secondary Structures: α-helix and β-sheet

1.3.5 Basics of Machine Learning

Machine learning has become something of a buzzword in many data-driven fields. While it is by no means a new field, with the term originally being coined in 1952 by Arthur Samuel [5], it has had a surge in popularity in recent times due to several factors. With the wealth of available data sets and exponential increases in computing power over the last couple of years, the interest in this field has soared. At the highest level, machine learning is a simple process: rather than telling a machine strictly what to do, like in conventional programming, it instead "teaches" a machine to make predictions from the data it is fed [6], which allows for tasks that are generally hard to be programmatically quantized to be solved.

1.3.6 Machine Learning Techniques

While teaching is a relatively familiar concept to most, not everything is directly transferable from humans to machines. As previously stated, at the highest level, machine learning is a simple process in which a machine is made to learn from data. While this sounds simple in principle, in practice this means devising a method in which to both let a machine learn, and to accomplish "learning". The former can be broadly broken into 4 main categories: supervised machine learning, unsupervised machine learning, semi-supervised machine learning, and reinforcement machine learning [11].

Supervised Learning

In simple terms, supervised learning is teaching by example: one provides a large amount of data that has been pre-labeled by a human to learn from. While this variety of learning is highly effective, it also requires the most input data, and consequently, the most human work.

Unsupervised Learning

Unsupervised is the exact opposite of supervised learning requiring no training data, though rather than classifying data, it is instead used to derive structures and inferences from data. Unsupervised

learning can further be broken down into two further distinct categories: Clustering and Association. Clustering attempts to group or "cluster" data, breaking a set of data into its component groupings. Association, on the other hand, tries to find general rules/associations [1]:

> *"An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y."*

Semi-Supervised Learning

Semi-supervised is the between, meant to have the positives of both unsupervised and supervised simultaneously. These are relatively new in comparison to the other three learning techniques, but with the advent of such solutions as Generative Adversarial Networks (GANs), these semi-supervised systems can prove extremely valuable [6].

Reinforcement Learning

The last, and most complex is reinforcement learning. Reinforcement learning is effectively learning through maximization: the machine learns by experimenting until its able to get the highest "score" [11]. By attempting to minimize cost/maximize score, reinforcement learners are able to acquire desired behaviors based on simple feedback.

1.3.7 Machine Learning for Bioinformatics

Neural Networks (NN)

Neural networks are the base for deep learning, which is a subfield of machine learning, which are designed to mimic the structure of the human brain [4]. Neural networks are designed to intake data, analyze and learn from the data, and then teach itself to find patterns within the data. From there, neural networks can predict similar data sets and outputs. If something is wrong, the network goes back to correct itself and learns again. Some common neural network applications are facial recognition, image translation, and music suggestion predictions.

Deep Convolutional Neural Networks (DCNN)

A deep convolutional neural network is a type of neural network that is most commonly applied to image processing problems. It is what computers use to identify objects in an image. They can also be used for language processing as well. NN's in general are meant to build data off of each other so that there is no guess work involved in prediction. However, in DCNN's, data is treated as "spatial". Instead of neurons being collected to all neurons (like in NN's), neurons are only associated with those close to them [9].

Random Forest

Classification algorithms in data science include logistic regression, support vector machines, naive Bayes classifiers, and decision trees. The random forest classifier, on the other hand, is near the top of the classifier hierarchy [12]. As the name implies, a random forest is made up of a huge number of individual decision trees that work together as an ensemble, where the trees can protect each other from individual errors. Each tree in the random forest produces a class prediction, and the class with the most votes becomes the prediction of the model. Yiu states that,

> *"A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models."*

1.4 Research Goal and Objectives

1.4.1 Research Goal

The research goal of this project is to develop machine learning algorithms to determine secondary structures element types of amino acids in proteins based on $C\alpha$ backbone features.

1.4.2 Associated Objectives

The associated objectives are listed as:

1. Understand the feature sets that are commonly used in literature for $C\alpha$-based amino acid classification. In order to achieve this, get familiar with Pandas library that allows to manipulate data files using Python programming language.

2. Understand various machine learning techniques that can be used for classification of $C\alpha$s. The techniques include ordinary Neural Networks (NNs), Deep Convolutions Neural Networks (DCNNs), and Random Forest. In order to achieve this, get familiar with TensorFlow library using Python programming language.

3. Implement an ordinary NN architecture using MNIST dataset.

4. Implement a DCNN architecture using MNIST dataset.

5. Implement an ordinary NN architecture using $C\alpha$ feature set available as a CSV file.

6. Implement a DCNN architecture using $C\alpha$ feature set available as a CSV file.

7. Compare the performances of ordinary NN and DCNN architectures.

1.5 Chapter Designations

This project is laid out as follows: chapter 2 provides a comprehensive requirement analysis, chapter 3 provides architectural details and design, chapter 4 discusses implementation, chapter 5 presents the experimental results, and chapter 6 covers conclusions.

CHAPTER 2

REQUIREMENT ANALYSIS

This section goes into the functional and non-functional requirements of this senior project. Functional requirements are mandatory requirements that define the functionality of the software. They describe what the program or product does and focus on user requirement. Non-functional requirements are non-mandatory and help verify the performance of the software. They describe how the program or product works and focus on the user's expectation and experience. Functional requirements describe what the software does while non-functional requirements describe how the software will do it.

2.1 Functional Requirements

1. A Fully Connected Neural Network (FCNN) will be developed using MNIST handwritten digits dataset.
   (a) The handwritten digit images will be fed into FCNN without any feature extraction.
   (b) The dataset will be divided into three groups: 50,000 images for training, 10,000 images for validation, and 10,000 images for testing.
2. A Deep Convolutional Neural Network (DCNN) will be developed using MNIST handwritten digits dataset.
   (a) The handwritten digit images will be fed into DCNN without any feature extraction.
   (b) The dataset will be divided into three groups: 50,000 images for training, 10,000 images for validation, and 10,000 images for testing.
3. A comparison of performance for FCNN and DCNN will be provided.

(a) Each dataset will be compared side-by-side for processing time and accuracy.

4. A self-contained software system will be developed that can read and manipulate Comma Separated Values (CSV) files.

   (a) The CSV file will be the protein dataset provided by the course instructor.

      i. The protein dataset provided in CSV format includes over 4,000 proteins and about 850,000 amino acids.

      ii. Each amino acid includes over 25 features.

   (b) The system will the read the entire CSV dataset into a data frame.

   (c) The system will allow the user to query certain parts of the dataset.

5. A FCNN will be developed using $C\alpha$ features provided in the CSV file to determine Secondary Structure Element (SSE) types.

   (a) The FCNN will have at least two hidden layers.

   (b) The SSE types will include helices, sheets, and loops.

6. A DCNN will be developed using $C\alpha$ features provided in the CSV file to determine Secondary Structure Element (SSE) types.

   (a) The DCNN will have various kernels and convolutional layers in addition to at least two hidden layers.

   (b) The SSE types will include helices, sheets, and loops.

7. Principal Component Analysis (PCA) will be used to determine the optimal number of features from the protein dataset.

   (a) The PCA will allow the system to work within a lower dimensional space.

   (b) Singular Value Decomposition (SVD) will be used after creating a data matrix.

2.2 Non-Functional Requirements

1. The system will be implemented in Python 3.

   Rationale: Python and its associated libraries are heavily used in state-of-the-art machine learning applications. It is critical for the senior project groups to understand the state-of-the-art, and the technologies they are built upon.

2. The system will be implemented using several key libraries including Pandas, Numpy, and TensorFlow.

   Rationale: Pandas is a key library used for data processing and analysis. Numpy is great for dealing with large numerical datasets. TensorFlow combines machine learning and deep learning models, including processes like obtaining data, training models, making predictions, and refining results.

3. The system will perform with at least 80% prediction accuracy.

4. The system should be able to be utilized without requiring intimate knowledge of computer science principles.

   Rationale: Working under the assumption this system is used for its intended purpose, the primary audience for this software would be those working in the biology field. As such, anyone in this field, including those without a computer background, should reasonably be able to operate this software.

5. The system should be able to operate on both large proteins (those containing more than 1000 amino acids) and small proteins (those containing fewer than 100 amino acids).

   Rationale: As there are many different variations in size among proteins, the system should be flexible enough to adapt to the size constraints of the data.

6. The system should be able to run in the Google Colaboratory environment.

   Rationale: Google Colaboratory (”Colab” for short) helps to facilitate sharing any code written in the ubiquitous ipython notebook format.

CHAPTER 3

DESIGN

This chapter goes through the specifics of the neural network system's architectural design. The architecture overview displays all of the subsystems required to run the system. The detailed design section goes through each subsystem in greater depth. This section describes the input, training, testing, learning, and application. The overall system architecture is shown below in Figure 3.1.

3.1 Architectural Design

The system utilizes data from two input sources in its operation, the instructor-provided CSV used to generate the neural network models, and $C\alpha$ data provided to the predictor model by an end user. As previously stated, the CSV is used solely for model generation. Predictor input is fed into the generated networks, and the system outputs the predicted secondary structure types from the given data. In theory, the end user would be able to input similar data (amino acid characteristics including $C\alpha$ data), and the system would be able to determine the secondary structure on its own based on the information gathered from the aforementioned CSV.

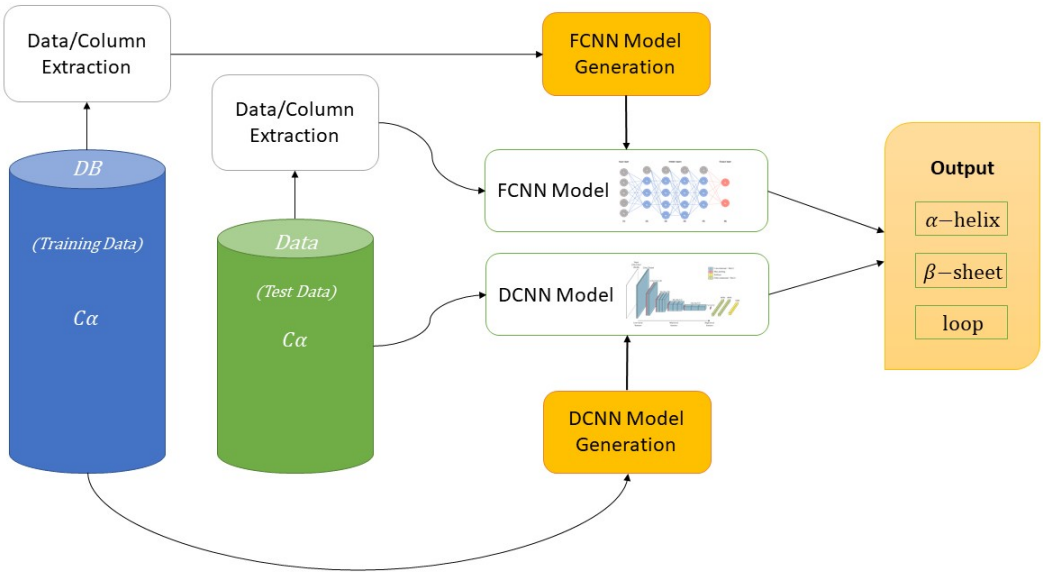

Figure 3.1: Overall System Architecture

3.2 Detailed Design

This section explains each component step that constructs the system. The CSV file provided by the instructor is used to generate a functional model to determine the secondary structure types of an amino acid based on its $C\alpha$ characteristics.

3.2.1 Data Preprocessing and Extraction

Before the CSV data can be used for training, the data must be preprocessed, extracting useful characteristic information, filtering out rows with NaN values, and quantizing/normalizing the data so that it can be fed into the network. The first stage selectively filters columns based on which ones are most useful in prediction. This selection will be made using principal component analysis (PCA) and singular value decomposition (SVD). To filter out NaN values (caused by missing/blank cells) the built-in Pandas dropna function will be utilized. Finally, the values have to both be quantized in the case of non-numeric columns (such as amino acid type and secondary structure type), and normalized (ensuring each of the value are within a confined input space).

3.2.2 Neural Network Training

Neural network training is accomplished by feeding the preprocessed data into the two neural network models: a deep convolutional neural network (DCNN), and a fully connected neural network (FCNN). The preprocessed data is split into three blocks, the first two blocks containing 10000 $C\alpha$s will function as the test and validation sets, the rest will be utilized as training data. With n layers, neural networks will have n-1 hidden units and 1 output unit. The are hidden layers from 1 to n-1, which is, every layer except the last one. The final one is known as the output layer. These layers are made up of units. In addition, these units are made up of weights and activation functions.

3.2.3 Validation

The validation stage will be used to ensure the models are constructed in the best possible way to both have good accuracy, as well as to run in a reasonable amount of time. Both models will be run using varying numbers of layers, neurons, filters (in the case of the DCNN), and training epochs in order to derive the best possible network to meet these goals. Ideally, this stage will determine the best and most precise combination of values that will gives the highest accuracy prediction.

3.2.4 Predictor

Following the model production and validation, the final component is the actual predictor generated from the preceding components. This part of the model uses the generated ”ideal” networks to predict secondary structures from newly provided $C\alpha$ data, and exports them in an easy to parse format. This component should be primarily what an end user should interact with. All other components should be hidden and not require direct interaction.

CHAPTER 4

IMPLEMENTATION

The implementation of the secondary structure inference system's design will be explained in this chapter. All of the implementation's functionality will be discussed, and a small selection of source code fragments will be provided to aid comprehension. The implementation is divided into three sections: data preparation, network construction, and network assessment. All three of these implementation components will be further separated by component functionalities.

4.1 Data Preparation

The data preparation component takes in the training data, preprocesses it, and splits it into training, testing, and validation data to be used in network construction. The software library in Python called 'Pandas' is what is used to firstly load the data, but it's also used for some of the data manipulation in general. In Listing 4.1, the first line is the import statement, and the second line is loading the CSV (read_csv).

```
1 import pandas as pd
2 df = pd.read_csv('ProteinData.csv')
```

Listing 4.1: Code for Loading CSV Using Pandas

4.1.1 Filtering out unnecessary/bad data

After the dataframe loads the given CSV file using Pandas, the system then filters out the 'bad' data. 'Bad' data is defined as any data that is irrelevant or unnecessary to the overall accuracy of the network and its goal. In Listing 4.2, the only features kept are those that are labeled with an 'f' (determined by the instructor), the neighbors, the protein ID, and the SStype (or alternatively NSStype (Explained in subsection 4.1.3) in the case of the seven class network). All of this un-

necessary data is removed using the ‘drop’ command, which simply deletes the unwanted columns (whose names are extracted using a REGEX (REGular EXpression, a string processing function used to extract or detect patterns in strings) on the list of column names). A ‘dropna’ line is also utilized to get rid of any NaN values, which stands for ”not a number” values (which is present where missing or empty values occurred in the CSV).

```
import re
prot_features.drop(columns=filter(lambda x: not
    re.match(r'f\d+|neighbors|ProteinID|SStype', x) ,prot_features.columns), inplace=True)
prot_features.dropna(inplace=True)
```

Listing 4.2: Code for Removing Unnecessary Columns and Bad Data

4.1.2 Data Splitting and Format Conversion

One of the most important stages in data preparation is splitting the data. The test data used is a list of the protein ID’s provided by the instructor that will be used to generate the testing set. This list is in Listing 4.3. A function called ‘breakdown’ (Listing 4.4) is the utilized to drop the Protein ID (as it won’t be used from here on), pop out the SStype (removes the ‘SStype’ column from the dataframe, and returns the column as output) (NSStype in seven class) and hold it under the variable name ‘labels’. The ‘labels’ variable is then categorized to numerical values, and converted into a column vector using the ‘to_categorical’ function, and returned as an array of vectors. This is then done to all sets.

```
test_data = ["4OH7", "5YDE", "2OPC", "6YDR", "6NZS", "1EAR", "2FP1", "2Z6R",
             "2OIT", "5JUH", "4B2O", "2JDA", "3LFK", "1Z6N", "6P8O", "5UEB",
             "5YDE", "3V4K", "4ZDS", "4WKA"]

test_data_df = prot_features[prot_features["ProteinID"].isin(test_data)]
train_data_df = prot_features[~prot_features["ProteinID"].isin(test_data)]
train_data_df, val_data_df = train_test_split(train_data_df, test_size=0.1)
```

Listing 4.3: Code for Splitting the Data

```
def breakdown(df : pd.DataFrame):
    df = df.copy()
```

```
    df.drop(columns=["ProteinID"], inplace=True)
    labels = df.pop("NSStype")
    labels = labels.astype('category').cat.codes
    return np.array(df), tf.keras.utils.to_categorical(labels)

x_train, y_train = breakdown(train_data_df)
x_test, y_test = breakdown(test_data_df)
x_val, y_val = breakdown(val_data_df)
```

Listing 4.4: Code for Breaking Down and Converting the Data

4.1.3 Column generation for neighboring secondary structures

For generating the Neighboring Secondary Structure Type (NSStype) column, first a copy of the full dataframe is created using the 'copy' function. From the copy, a list of pairs generated from the 'Num' column and 'SStype' column (using the zip function on the two columns, which generates such a list) is extracted. These pairs are utilized in determining the secondary structure sequences. From this list of pairs, a list of 'Next secondary structures' is generated by iterating through the list utilizing the map function. In the map function a lambda function is used. This lambda functions checks that the following secondary structure exists and is in-fact next-in-sequence, producing the required list. Following this, a similar procedure is used to generate a list of 'previous secondary structures'. After both the 'next' and 'previous' lists are generated, they are both then combined with the normal secondary structure type, producing list of tuples of previous, current, and next secondary structures for each amino acid, with each tuple then converted to a set through the 'set' function to remove repeats, they then are each sorted alphabetically using the 'sorted' function, and then are each combined into strings, generating the appropriate sequence to indicate the relevant class. Finally, this list is converted into a pandas series using the 'pd.Series' function, and written to a new column named 'NSStype' in the copied dataframe. The full code for this can be seen in Listing 4.5.

```
prot_features = df.copy()
sstypes = list(zip(prot_features['Num'], prot_features['SStype']))
```

```
nsstypes = list(map(lambda x: sstypes[x[0] + 1][1] if x[0] + 1 < len(sstypes) and
    sstypes[x[0] + 1][0] == x[1][0] + 1 else np.nan, enumerate(sstypes)))
psstypes = list(map(lambda x: sstypes[x[0] - 1][1] if x[0] - 1 > 0 and sstypes[x[0] - 1][0]
    == x[1][0] - 1 else np.nan, enumerate(sstypes)))
ngsstypes = [set(filter(lambda x: type(x) is str, [i, j, k])) for i, j, k in zip(nsstypes,
    psstypes, (l for _, l in sstypes))]
prot_features['NSStype'] = pd.Series(['',join(sorted(list(i))) if i else np.nan for i in
    ngsstypes])
del sstypes
```

Listing 4.5: Code for Generation of the 'NSStype' Column

4.2 Network Construction

All of the models are sequential, which means that models are able to be generated in a layer-by-layer, linear fashion. Then, all the numerical values are rescaled to a confined range in order to increase consistency between features and increase output accuracy. This is done through the 'normalization' layer generated in Listing 4.6. What the normalization layer does is simple: based on the data provided in the 'adapt' method, it generates certain parameters that normalize each column in a row.

```
normalize = preprocessing.Normalization()
normalize.adapt(x_train)
```

Listing 4.6: Normalization Layer Construction Code

4.2.1 FCNN

In the fully connected case, the network is a sequential network constructed purely of 'Dense' (fully-connected) layers. What this means is that a given dense layer has all of the previous layers' outputs connected as input to all of the layers' neurons. The first argument of the 'Dense' constructor specifies the size in number of neurons, and the second 'activation' argument defines the activation function. An activation function determines the input preprocessing function which can be a number of functions. In this systems specific model rectified linear unit (a.k.a. relu which

truncates all negative values to zero) and softmax (which normalizes all layer inputs such that the sum total is equal to 1) are used. Another notable activation function is sigmoid (which compresses a range of $-\infty$ to ∞ to between 0 and 1), and although it is not utilize it in this project's code, it is a highly important and often utilized activation function. Relu and softmax are utilized specifically because relu is a highly performant activation function and softmax is utilized for the output layer, as the output layer should represent the confidence of each class prediction (e.g. a result of 0.8, 0.1, 0.1 means a prediction confidence of 80% for the first class, 10% for the second class, and 10% for the third class; There should not be a case where a sum of all confidences exceed 100%).

4.2.2 DCNN

In the deep convolutional case, convolutional layers are added in front of a fully convolutional network, as can be seen in Listing 4.7. First, however, before feeding into the convolutional layers, the data must be reshaped into the required shape (length by number of channels), which for the number of columns utilized (34) is 34x1 (34 columns, each being a single channel/number). Next there are the convolutional layers. Since the data is 1-Dimensional (single row of values) a 1D convolutional layer 'Conv1D' is used. The first argument for Conv1D is number of filters (filters are learnable weights that store a single template or pattern that the convolutional layer uses to 'scan' over the data), the second argument is the size of the filters (aka the kernel size), the 'input_shape' argument defines the input shape, and finally the 'activation' argument works the same for convolutional as with dense layers. The final step before feeding into the fully connected portion of the network is the 'flatten' function, which re-converts the data back into the simple row structure required by the dense layers. Of note, while the particular network only utilizes convolutional layers, a majority of convolutional networks will also include maxpooling layers, which are a useful downsampling tool (meaning they reduce the size of their input); however, due to the relatively small size of the input, it was determined that they would not provide any significant benefit in the case.

```
1 prot_model = tf.keras.Sequential([
2   normalize,
```

```
  layers.Reshape((34, 1)),
  layers.Conv1D(64, 4, activation='relu', input_shape=(34, 1)),
  layers.Conv1D(64, 4, activation='relu'),
  layers.Flatten(),
  layers.Dense(51, activation='relu'),
  layers.Dense(17, activation='relu'),
  layers.Dense(51, activation='relu'),
  layers.Dense(17, activation='relu'),
  layers.Dense(len(y_train[0]), activation='softmax')
])
```

Listing 4.7: Neural Network Model (7-class, Convolutional)

4.2.3 Model Compilation

Before the model can be trained, it must first have its 'compile' method called (shown in Listing 4.8). The compile method simply configures a network for the training stage by setting certain important parameters and functions to be utilized. First function specified is the 'loss' function; the loss function in a neural network predicts error in the network and is used in computing the mathematical gradient which is then used to update the weights of the neural network through a process called 'gradient descent'. The gradient descent process simply moves the parameters in the opposite direction of the gradient, which mathematically means moving the fastest possible towards a minima (lowest point); the lowest point for a loss function represents best performance. Categorical crossentropy (which is the loss function used) is a loss function used in multi-class classification problems. These are problems in which an example may only belong to one of several potential categories, and the model must select which one. Formally, it is intended to measure the difference between two probability distributions. The next parameter is the optimizer function; the optimizer function is the function used to make the requisite changes based on the loss function results. For all of this system's networks the ADAM optimizer is used, which is a modified form of stochastic gradient descent (SGD). SGD simply takes a single or small subset of samples (rows) from the training data to derive the loss function results with small nudges to weights and biases applied to determine the mathematical gradient. The last argument is 'metrics' which simply de-

termines which metrics to show in the results for testing and validation; 'categorical_accuracy' simply shows the percentage of correct categorical guesses vs total number of guesses.

```
prot_model.compile(loss = tf.losses.CategoricalCrossentropy(),
    optimizer=tf.optimizers.Adam(), metrics=['categorical_accuracy'])
```

Listing 4.8: Neural Network Compilation Code

4.2.4 Training

To perform training, the 'fit' method of the model is utilized. In the arguments of the fit method, how training should be carried out is specified. For these networks, the first two arguments (which represent training data, input and expected output) are set to 'x_train' and 'y_train', which are the training data inputs and training labels/outputs for this network. Following this, the number of epochs is specified as a maximum of 100. An epoch in neural network training is an instance where each trial in the training data has had a chance to update the internal model parameters; basically, the number of epochs is the amount of times that the learning algorithm will tackle the whole training dataset. The next argument is batch size, which simply specifies how many samples to use in a gradient descent cycle (in this case, this is set to 500 samples). Following this, the 'validation_data' argument specifies the validation data. Validation data is used to check the performance of the network following every epoch; this is important, as it is used in determining the rate at which the model is improving. The last argument for fit is 'callbacks', in which an 'EarlyStopping' callback is set up. The 'EarlyStopping' callback makes the model automatically stop training once a certain number of epochs (in this case 4, since 'patience' is specified as 4) have occurred without improvements to the validation accuracy; This is useful as it allows a high maximum number of epochs, while ensuring that excessive numbers of epochs are not occurring unnecessarily.

```
prot_model.fit(x_train, y_train, epochs=100, batch_size=500, validation_data=(x_val,
    y_val), callbacks = [tf.keras.callbacks.EarlyStopping(patience = 4)])
```

Listing 4.9: Code to Perform Model Fitting

4.2.5 Network Architecture Search (NAS)

The last, and most important part of network construction is the network architecture search. What network architecture search does is run through many possible network configurations in order to find the best possible network to solve a given problem.

Network Constructor

The first part of the network architecture search is generating the model building function, which can be seen in Listing 4.10. The process for building the network consists of many simple steps. First a number 'n_sh', which represents the shape for FCNN portion of the network, is required. It is converted into the actual shape as follows: the number is converted into a base 'max_scale' number, through which the code iterates with each digit to which one is added followed by multiplying by 'n_base' to get the size of a layer (number of neurons). The result is then recorded to a list to be used later for adding the corresponding layers to the model. Following this, the sequential model is generated using 'tf.keras.Sequential' with no arguments, and has the normalization layer (whose generation process was explained in the intro of section 4.2) added using the 'add' method. Following this, a number of Conv1D layers, the quantity of which is specified in 'n_conv', is added with the number of filters being defined in 'n_filters' and kernel size being defined in 'sz_kernel'. Then a flatten layer is added, followed by a series of dense layers whose size and quantity were determined previously. Finally, the output dense layer is added, and the model is compiled and subsequently returned. This whole build function is then warped using the SciKit Learn 'KerasRegressor' wrapper, which is needed in order to utilize the build function in the search.

```
def model_build(n_base = 16, n_sh = 1, n_conv = 0, n_filters = 4, sz_kernel = 2, max_scale
    = 4, printlayers = False):
    n_sh_l = []
    for i in range(int(math.log(n_sh, max_scale))):
      n_sh_l.append((n_sh % max_scale + 1) * n_base)
      n_sh /= max_scale
      n_sh = int(n_sh)
    if not n_sh_l:
      n_sh_l = [n_base]
```

```
    test_model = tf.keras.Sequential()
    test_model.add(normalize)
    if printlayers:
      print('normalize,')
    if n_conv:
      test_model.add(tf.keras.layers.Reshape((34,1)))
      if printlayers:
        print('Reshape((34, 1)),')
    for i in range(n_conv):
      if printlayers:
        print(f'Conv1D({n_filters}, {sz_kernel}, activation=\'relu\'),')
      if not i:
        test_model.add(tf.keras.layers.Conv1D(n_filters, sz_kernel, activation='relu',
    input_shape=(34, 1)))
      else:
        test_model.add(tf.keras.layers.Conv1D(n_filters, sz_kernel, activation='relu'))
    if n_conv:
      test_model.add(tf.keras.layers.Flatten())
      if printlayers:
        print('Flatten(),')
    for i in n_sh_l:
      test_model.add(tf.keras.layers.Dense(i, activation='relu'))
      if printlayers:
        print(f'Dense({i}, activation=\'relu\'),')
    test_model.add(layers.Dense(len(y_train[0]), activation='softmax'))
    if printlayers:
      print(f'Dense(len(y_train[0]), activation=\'softmax\')')
    test_model.compile(loss = tf.losses.CategoricalCrossentropy(),
    optimizer=tf.optimizers.Adam(), metrics=['categorical_accuracy'])
    return test_model

kreg = tf.keras.wrappers.scikit_learn.KerasRegressor(model_build)
```

Listing 4.10: NAS Model Construction Code

Parameter Definition and Search

Following the construction function being defined, the actual search can then be performed. To do this, the 'RandomizedSearchCV' function is utilized as can be seen in Listing 4.11. The 'Ran-

domizedSearchCV' function has several arguments: the first being for the network constructor function, followed by the parameter distrbution, 'n_iter', and 'cv'.The parameter distribution argument specifies all the possible parameter values it can test with the model constructor. The 'n_iter' argument specifies the number of search iterations to be run. The 'cv' argument specifies the type of cross-validation (which combines average prediction accuracies to derive an estimate of model-predictive performance) to use (in this case is 3-fold, which means performing crossvalidation on 3 equal divisions of all the samples). The fit line then takes the specified search parameters and will randomly test numerous networks, each trained according to the train parameters it is passed (which are equivalent to the fit parameters utilized in normal training as seen in Listing 4.9).

```
param_distrib_conv = {
    'n_conv': range(3),
    'n_filters': [16, 32, 64],
    'sz_kernel': [4, 8, 17],
    'n_sh': range(1,4096),
    'n_base': [17]
}
rnd_srch = RandomizedSearchCV(kreg, param_distrib_conv, n_iter=50, cv=3)
rnd_srch.fit(x_train, y_train, epochs = 100, batch_size=500, validation_data = (x_val,
    y_val), callbacks = [tf.keras.callbacks.EarlyStopping(patience = 4)])
```

Listing 4.11: Code for Performing Search (NAS)

Outputting Results

After running the search, the best network found through the search process can be output. As shown in 4.12, the function starts by presenting the numeric parameters and the score of the best network found. Next the found network is fed to the 'build' function, building the model with a verbose flag ('printlayers=True') so that it prints out each of the layers as they are added. The summary command is also run on the generated network to print the network summary as well.

```
print(rnd_srch.best_params_, rnd_srch.best_score_)
bp = rnd_srch.best_params_
model_build(n_sh = bp['n_sh'], n_base = bp['n_base'], sz_kernel = bp['sz_kernel'],
```

```
4            n_filters = bp['n_filters'], n_conv = bp['n_conv'], printlayers =
    True).summary()
```

Listing 4.12: Code for Outputting NAS Search Results

4.3 Network Assessment

The last component for the system is network assessment. The network assessment component takes the constructed network post training, and evaluate the overall performance of the finalized network.

4.3.1 Accuracy validation

The primary and most important metric for the network is accuracy, which is found using the 'evaluate' function as seen in Listing 4.13. All the 'evaluate' function does is simply compare predictions with expected results. The evaluate function is fed the test data, along with the test set labels, and the 'verbose=2' argument which enables the progress bar. On execute it will return the testing accuracy of the trained model.

```
1 prot_model.evaluate(x_test, y_test, verbose=2)
```

Listing 4.13: Network Evaluation Code

4.3.2 Confusion matricies

```
1 print(prot_features['NSStype'].astype('category').cat.categories)
2 cat_names = prot_features['NSStype'].astype('category').cat.categories
3 tf.math.confusion_matrix(np.argmax(y_test, axis=1), np.argmax(prot_model.predict(x_test),
    axis=1))
```

Listing 4.14: Confusion Matrix Generation Code

Before generating the confusion matrix, the first line prints off the protein feature 'NSStype' (or SStype if three-class) categorized to numerical values. Then the collection of names is stored under

a variable called ‘cat_names’ for later use. For the third line, tensorflow’s ‘math.confusion_matrix’ function is used. The ‘math.confusion_matrix’ function outputs a matrix that shows the amount of predictions of a class were of each class (e.g. how many helices were guessed to be helices, how many were guessed to be loops, etc.) This is used to gauge performance, and determine possible points of confusion. The parameters utilized in the confusion_matrix function are np.argmax, which returns the indices of the max values. In both parameters, the max values are along a linear axis (1). The first parameter takes the np.argmax of the aforementioned ‘y_test’ array as the label for classification (which returns a list of class numbers for all test outputs). The second parameter takes the np.argmax of ‘prot_model’ predictions based on the input data (‘x_test’) array (The ‘predict’ function accepts ‘x_test’ as a single argument and returns the labels of the data based on the learned data from the model). With this, the output allows visualization of the result data in a neat, statistical way.

4.3.3 Receiver Operating Characteristic (ROC) Curve

An ROC curve is a simple graph showing the performance in terms of false positive rate (FPR) and true positive rate (TPR) for a range of different classification thresholds (classification thresholds simply are the threshold at which the positive case is assumed from a given output value). For generating the ROC Curve some prediction are needed to build the curve from. To do this, the network’s predictions are taken for the whole training set by using the models ‘predict’ method. A for loop is then used to generate the false positive and true positive rates from the ROC curve (using SciKit Learn’s ‘roc_curve’ function, which generates the TPR and FPR for a range of threshold values) for each class by comparing to the expected labels. The area-under-curve (AUC) value is also calculated using the ‘auc’ function also from SciKit Learn. For the micro average, the ROC and AUC are calculated by using the corresponding function on the ‘np.ravel’ flattened version of the matrix-of-curves (which contains the all FPR/TPR points for each class). To generate the macro average curve all FPRs are first aggregated by combining all FPR arrays using ‘np.concatenate’ followed by extracting each unique value from the array using the ‘np.unique’ function. The mean

TPR is derived for macro average by summing the points for each class using the np.interp function (which performs 1D interpolation) at all points along the curve (which correspond to each of the points on the previously generated 'all_fpr' array), and dividing them by the total number of classes. Lastly, the AUC is calculated for the macro average curve by running the 'auc' function with the calculated macro-average FPR and TPR. Finally, matplotlib is used to plot each of the curves, with a differing color and a corresponding label in the plot legend.

```
#Define line width for the curve
lw = 2

y_pred_roc = prot_model.predict(x_train)

y_train_roc = y_train.copy()

n_classes = len(y_test[0])

fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train_roc[:, i], y_pred_roc[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

fpr["micro"], tpr["micro"], _ = roc_curve(y_train_roc.ravel(), y_pred_roc.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at the points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
```

```
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.rcParams['figure.figsize'] = [10, 8]
plt.figure()
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'pink', 'darkgreen', 'maroon',
    'olive'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0}: {1} (area = {2:0.2f})'
             ''.format(i, cat_names[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.show()
```

Listing 4.15: ROC Curve Generation Code

CHAPTER 5

RESULTS

This chapter presents a comprehensive set of results that show how all functional and non-functional requirements are satisfied. It is also intended to present how various functions within the system work.

5.1 CSV Reading

In this project, the networks used Pandas to read the given CSV. Pandas being built on top of Python and being fast and reliable resulted in an efficient CSV reading system in only a few lines of code (example of its usage can be seen in Subsection 4.1.2).

5.2 MNIST Networks

Before beginning the process of generating networks for Cα based secondary structure determination, several simpler networks were set up based on the MNIST dataset [10]. This was done as the MNIST dataset is a sort-of "hello world" for machine learning applications. In particular, both a Fully Connected Neural Network (FCNN) as well as a Deep Convolutional Neural Network (DCNN) were constructed.

5.2.1 Fully Connected Neural Network

As stated in previous chapters, the simplest network approach utilized is the FCNN. For MNIST dataset, many of the normalization features explained in the previous chapter (specifically in Section 4.2) are not used, instead the network uses basic division to normalize the data. The built-in MNIST samples (available as tf.keras.datasets.mnist) are divided by 255, then fed into the network.

The network utilized in this case is a small, simple network with a flatten layer and 3 dense layers: a flatten layer to flatten the 28x28 matrix representation of the MNIST samples into a vector, two dense layers of sixteen with relu activation, and a dense layer of 10 neurons to work as output (the details of what these layers and activation function represent is discussed in Section 4.2). With this simple setup, this network was able to get 94.7% testing accuracy with an inference time of 43μs.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

Listing 5.1: FCNN Model for MNIST Dataset

5.2.2 Deep Convolutional Neural Network

For the DCNN a similar approach is taken to the FCNN, however, excluding the flattening stage utilized for the FCNN. The major difference is simply the utilization of two convolutional layers on top of the Fully-connected layers, with the removal of one of the dense layers (note: conv2d function much the same as conv1d, discussed in detail in Chapter 4.2.2, however, the kernel size '3' represents a 3x3 square in the 2d case). With this setup, this network was able to get 98.2% accuracy with an inference time of 490μs.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64,3,activation='relu', input_shape=(28,28,1)),
  tf.keras.layers.Conv2D(32, kernel_size=3, activation='relu'),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Listing 5.2: DCNN Model for MNIST Dataset

5.2.3 Comparison

While the DCNN produced obviously better results in comparison to the FCNN, the performance loss was quite significant. In general, its a trade-off between speed and accuracy. The better of the two options is dependent on situation.

Network Topology	Categorical Accuracy	Inference Time
FCNN	94.7%	43μs
DCNN	98.2%	490μs

Table 5.1: Comparison Table for the MNIST Neural Networks

5.3 Cα Networks

Once the example networks were successfully generated, Cα based secondary structure determination networks were constructed, the details of which were documented in Chapter 4. In total, four networks were constructed, a FCNN and a DCNN for both three class and seven class cases.

5.3.1 FCNN

3-Class FCNN

For the 3-class FCNN, several topologies were attempted in the early stages, originally settling on an all-dense topology with layers of 128, 64, 32, 16, 3 neurons in sequence. This network yielded a categorical accuracy of 92.1%; however when implementing NAS (as explained in Section 4.2.5) for the seven class case, NAS was chosen to be used for all networks as it should provide a more ideal network. The NAS provided a topology of dense layers containing 68, 68, 34, 51, 51, and 3 neurons in sequence. Multiple data splits were tested. Using the original 20-protein-test set (Table 5.2) with 10%-validation and 90%-train split under the NAS provided topology, the network had a categorical accuracy of 92.4%. Using an 80%-training, 10%-test, and 10%-validation split, the network had a categorical accuracy of 92.7%. Based on these results, the network ended up utilizing the 80-10-10 split.

20 Test Proteins				
40H7	5YDE	20PC	6YDR	6NZS
1EAR	2FP1	2Z6R	20IT	5JUH
4B20	2JDA	3LFK	1Z6N	6P80
5UEB	5YDE	3V4K	4ZDS	4WKA

Table 5.2: 20 Test Proteins Provided by Dr. Sekmen

With the chosen FCNN topology, and chosen test-training-validation split for the three-class case, the FCNN yields good results, with 92.7% accuracy (as previously stated), and an inference time of 42μs (tested on an Intel i9-9900k). The ROC curves (Figure 5.1) follow close to an ideal curve. Looking at the confusion matrix (Table 5.3), the majority of errors were for the β-sheet.

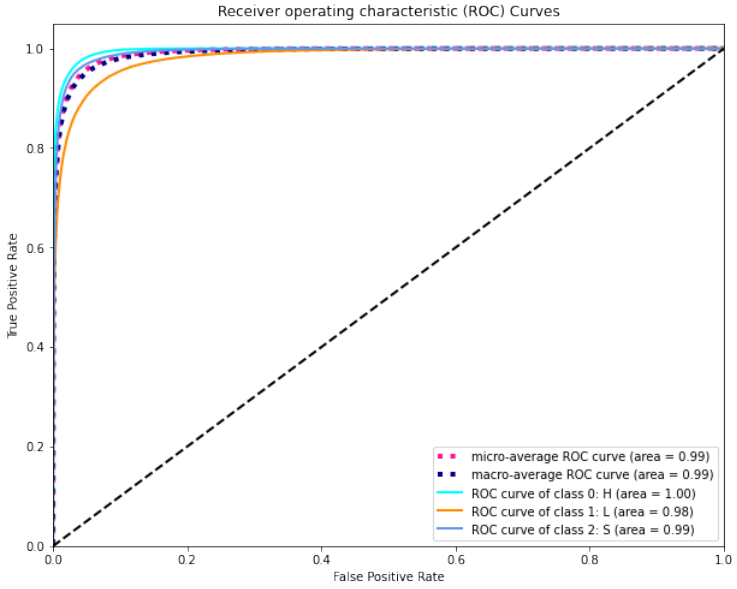

Figure 5.1: ROC Curve for 3-Class FCNN

	H	**L**	**S**
H	36842	1539	77
L	1580	25629	1394
S	45	1636	18061

Table 5.3: Confusion Matrix for 3-Class FCNN

7-Class FCNN

For the seven-class case, the network continued to utilize a similar set of selections to the previous, with the 8-10-10 split combined with a NAS generated topology, containing dense layers of 68, 68, 34, 51, and 7 neurons. The FCNN yields decent results, with 83.4% categorical accuracy, and an inference time of 42μs. Notably, ROC curve (Figure 5.2) still follows close to an ideal curve. Also noticeable, the confusion matrix (Table 5.4) presents the most confusion in the loop secondary structure.

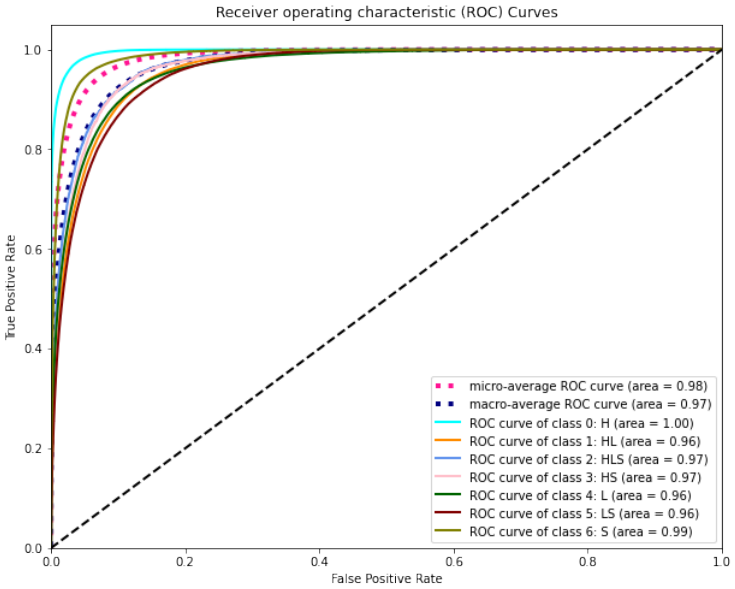

Figure 5.2: ROC Curve for 7-Class FCNN

	H	**HL**	**HLS**	**HS**	**L**	**LS**	**S**
H	31822	863	17	24	172	12	0
HL	1170	6204	10	22	1879	211	16
HLS	56	54	81	15	92	240	6
HS	80	131	12	117	38	320	56
L	500	1347	10	7	14201	1478	163
LS	36	119	33	60	1779	8970	1705
S	3	13	1	4	258	1372	11024

Table 5.4: Confusion Matrix for 7-Class FCNN

5.3.2 DCNN

3-Class DCNN

For the 3-class DCNN, the network did originally follow a similar process to the FCNN, with a custom chosen network (64-filter, kernel-size 3 convolutional layers followed by dense layers of 128,64,32,16, and 3 neurons), before ultimately using the NAS generated topology (2x64-filter, kernel-size 3 convolutional layers followed by dense layers of 17,34,34,17, and 3 neurons). While the custom network yielded 93% categorical accuracy, it was edged out by the NAS topology, which provided 93.2% categorical accuracy. The inference time for either was approximately 70μs.

The ROC curves for this case follow close to an ideal curve (Figure 5.3). As well, the majority of inaccurate guesses revolve around the loop secondary structure from the confusion matrix. (Table 5.5).

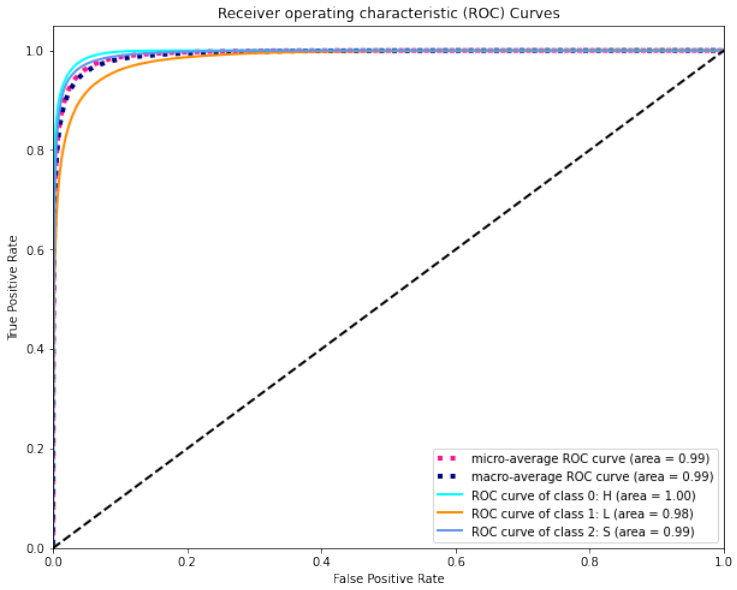

Figure 5.3: ROC Curve for 3-Class DCNN

	H	L	S
H	37359	1328	44
L	1704	25476	1436
S	47	1338	18071

Table 5.5: Confusion Matrix for 3-Class DCNN

7-Class DCNN

For the seven-class DCNN, rather than experiment around we directly utilized an NAS located topology of 2x64-filter, kernel-size 4 convolutional layers, followed by dense layers of 51, 17, 51, 17, and 7 neurons. This DCNN yields good results, with 84.3% accuracy, and an inference time of 64μs. The ROC curves still follows close to an ideal curve (Figure 5.4). Also seen from the confusion matrix, (Table 5.6) similar to the equivalent FCNN, that the most confusion is in the loop secondary structure.

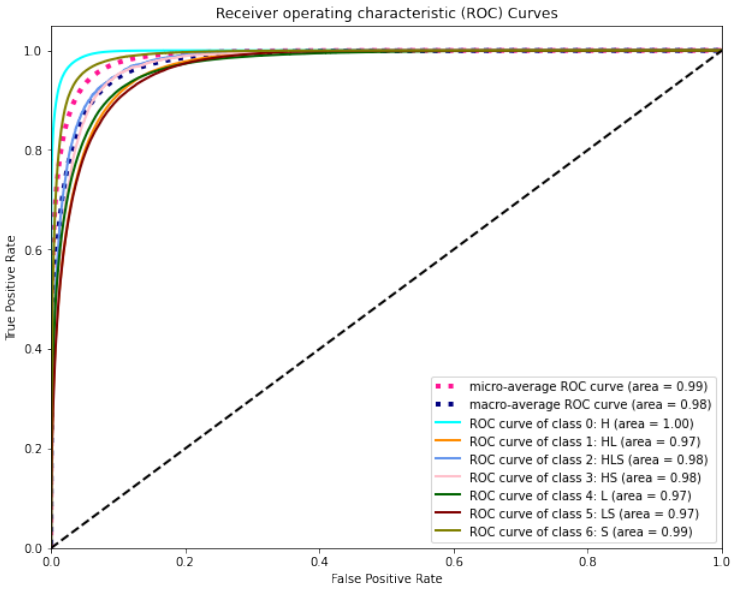

Figure 5.4: ROC Curve for 7-Class DCNN

	H	HL	HLS	HS	L	LS	S
H	31676	916	32	49	251	30	0
HL	1013	6440	49	40	1707	188	8
HLS	34	80	137	16	58	248	9
HS	58	104	13	166	33	304	61
L	313	1485	22	14	14267	1487	178
LS	24	124	95	66	1510	9327	1511
S	3	7	4	9	259	1238	11140

Table 5.6: Confusion Matrix for 7-Class DCNN

5.3.3 Comparison

For the FCNN networks, the network provided good accuracy, with high speeds. In general, the FCNNs were significantly faster in terms of single sample inferences, taking approximately 40μs on average in both cases. The DCNN networks sacrificed a bit of that speed to gain a few points of accuracy. In general, the gain for accuracy was approximately 1%, with an inference time increase of approximately 60-70%. When contrasting the cases based on class count, the FCNN to DCNN conversion for 3-class had a relative accuracy increase of 0.5% (with an actual increase of 0.5%) whilst taking approximately 166% the time, whereas the FCNN to DCNN conversion of 7-class had a relative accuracy increase of 1% (with an actual increase of 0.9%) whilst taking approximately 153% the time. In either case, the accuracy improvement is negligible, whilst the change in time required is significant. However, as this system is more than likely going to have less constraints in terms of runtime in the real word, the small accuracy improvement yielded by the DCNN is likely preferable, despite the extra time required for inferences.

Network Topology	Categorical Accuracy	Inference Time
3-Class FCNN, Original Topology, 20-Protein	92.1%	43μs
3-Class FCNN, NAS Topology, 20-Protein	92.4%	42μs
3-Class FCNN, NAS Topology, 80-10-10	92.7%	42μs
7-Class FCNN, NAS Topology, 80-10-10	83.4%	42μs
3-Class DCNN, Original Topology, 80-10-10	93.2%	71μs
3-Class DCNN, NAS Topology, 80-10-10	93.2%	70μs
7-Class DCNN, NAS Topology, 80-10-10	84.3%	64μs

Table 5.7: Comparison Table for the Cα Neural Networks

5.4 Principal Component Analysis

To determine whether the project would've been more efficient with less columns, Principal Component Analysis (PCA) was implemented. Aside from increasing efficiency, PCA highlights the most important aspects of variation in the columns and de-emphasizes the others. It does this by taking the dataset that varies on many dimensions and then looking on fewer dimensions. Singular Value Decomposition (SVD) was used as the form of dimension reduction. The formula for SVD is $A = U\Sigma V^T$, where V vectors are the potential principal components. To perform PCA, the range of continuous starting variables is standardized, correlations are found by computing the covariance matrix, the major components are determined by computing the eigenvectors and eigenvalues of the covariance matrix, and a feature vector is created to help select which main components to keep, and finally the data is recasted along the new axes of the major/useful components. Because SVD was used to implement PCA, the system was able to skip making a whole covariance matrix altogether. Using those principal components, special relations can be more easily read between the columns, and the system can then infer what these components could be saying about the NSStype as a whole.

SVD was implemented for PCA by using a numpy function called numpy.linalg.matrix_rank (there is also a numpy.linalg.svd function, but this one suits the specific need more). This function takes an input vector or stack of matrices, the threshold under which SVD values are zero (or considered zero), and a boolean function to signify if the matrix is Hermitian, meaning the matrix is symmetric if real-valued if this is equal to True, but it defaults to False. This function conveniently returns the matrix rank. The matrix rank is the number of singular values in the array that are greater than the threshold given in the parameter. After implementing this function, the result was a rank of 34, meaning that all 34 columns are independent, and as such, non-redundant and non-reducible. The exact implementation can be seen in Listing 5.3.

```
1 from google.colab import drive
2 drive.mount('/content/drive')
3 import pandas as pd
```

```
import numpy as np
import re
from sklearn.model_selection import train_test_split
df = pd.read_csv('drive/MyDrive/ProteinData.csv')
df.drop(columns=filter(lambda x: not re.match(r'f\d+|neighbors', x), df.columns),
    inplace=True)
df.dropna(inplace=True)
analysis_df, _ = train_test_split(df, train_size=2000)
analysis_mat = np.matrix(analysis_df)
print(f'Matrix Rank: {np.linalg.matrix_rank(analysis_mat)}')
u, s, vh = np.linalg.svd(analysis_mat, full_matrices=False)
smat = np.diag(s)
mat_reduced = np.dot(u, np.dot(smat, vh))
print(f'Reduced Matix: {mat_reduced}')
```

Listing 5.3: Code for Performing Principle Component Analysis (PCA)

5.5 Fulfillment of Non-Technical Requirements

The system was successfully implemented in Python 3. Python is known for being free and open source, being compatible with all major OS, having extensive libraries, and having data science support. All of these factors aided in the project's success. There were issues with speed (particularly with the preprocessing stages), as Python itself tends to be slow and not memory efficient, but those did not majorly interfere with the progress.

As previously mentioned, Pandas was fast and reliable and was a key library for the data processing and analysis. With native Python being inefficient with large data, Pandas was able to help make the CSV reading quick and efficient. Numpy was also a major tool used to manipulate the given dataset, allowing for fast mathematical operations. The CSV file had more than 80,000 rows, but Numpy made it more manageable to sift through the data efficiently. Tensorflow was the brains of the project, and combined with Pandas and Numpy, two Neural Networks were successfully created that far exceeded the goal of 80% accuracy. Tensorflow was utilized to obtain the data, creating training models, make predictions, and define results.

Using Google Colab was very beneficial for the project. Edits were able to be shared across

users easily. Google Colab facilitated sharing code in the iPython Notebook format, and this also made it much easier to read and understand as well with features like markdown and code blocks.

5.5.1 Graphical User Interface (GUI)

A Graphical User Interface was developed for the network by utilizing the Tkinter package for Python, seen in Figure 5.5. This package has many different components like buttons, listboxes, and built-in geometry to pack these components neatly. Firstly, a window was created that has four frames in it. The leftmost frame is where the user can drop a CSV file into, and it lists the file. Once double-clicked, the CSV loads into the right, topmost frame where the user can scroll vertically and horizontally to view their CSV file. This simultaneously fills the third frame, which holds a listbox of all of the column names in the given CSV file, and the user can choose which columns to use to test the neural network. The loaded dataframe also triggers a button that says ”Run CSV with These Chosen Columns” that the user can click to begin running through the epochs, which appears in the terminal, as seen in Figure 5.6. Overall, the use of the Tkinter package (Python-based) was a good choice as the networks themselves were Python-based, however, the documentation for certain objects was not always clear.

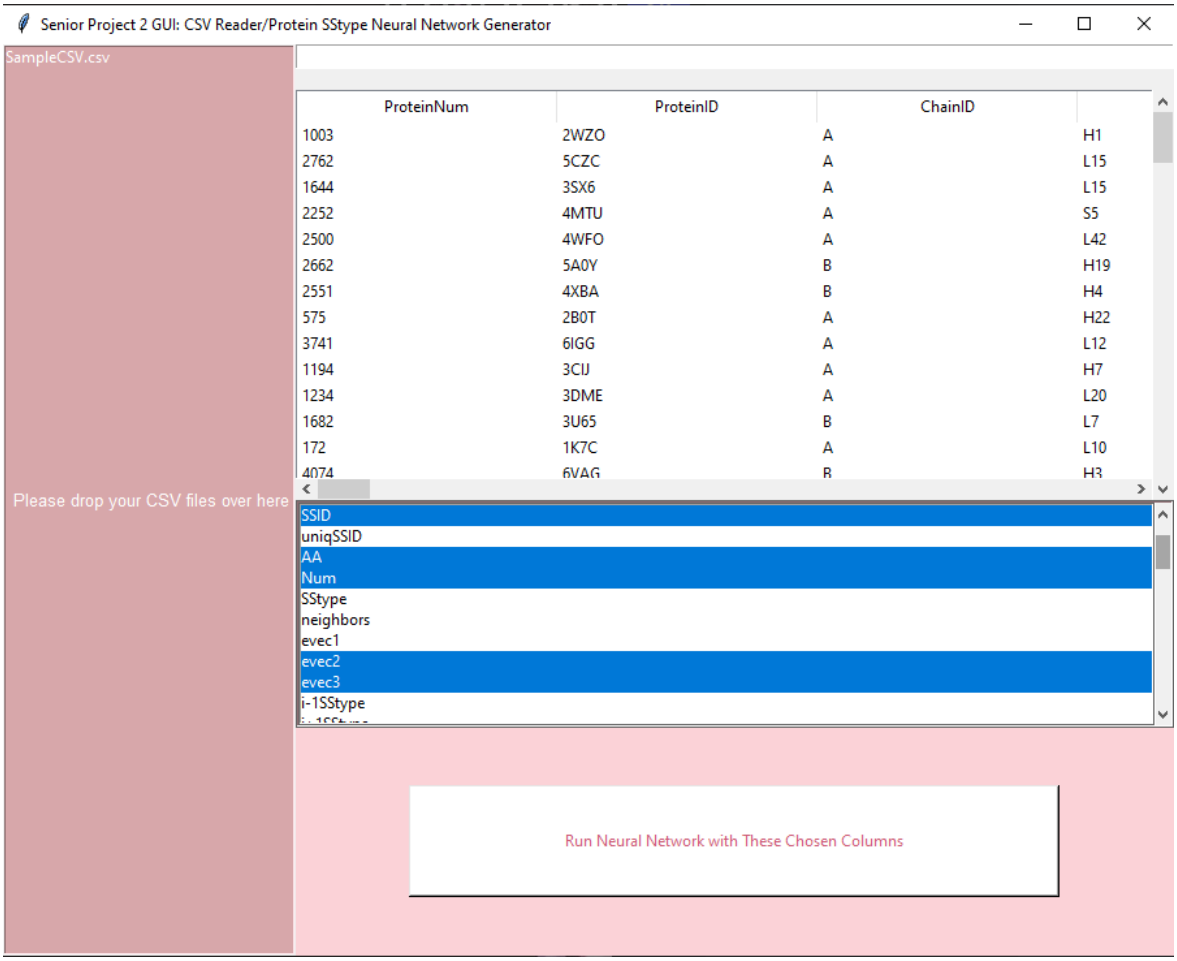

Figure 5.5: The Graphical User Interface (GUI)

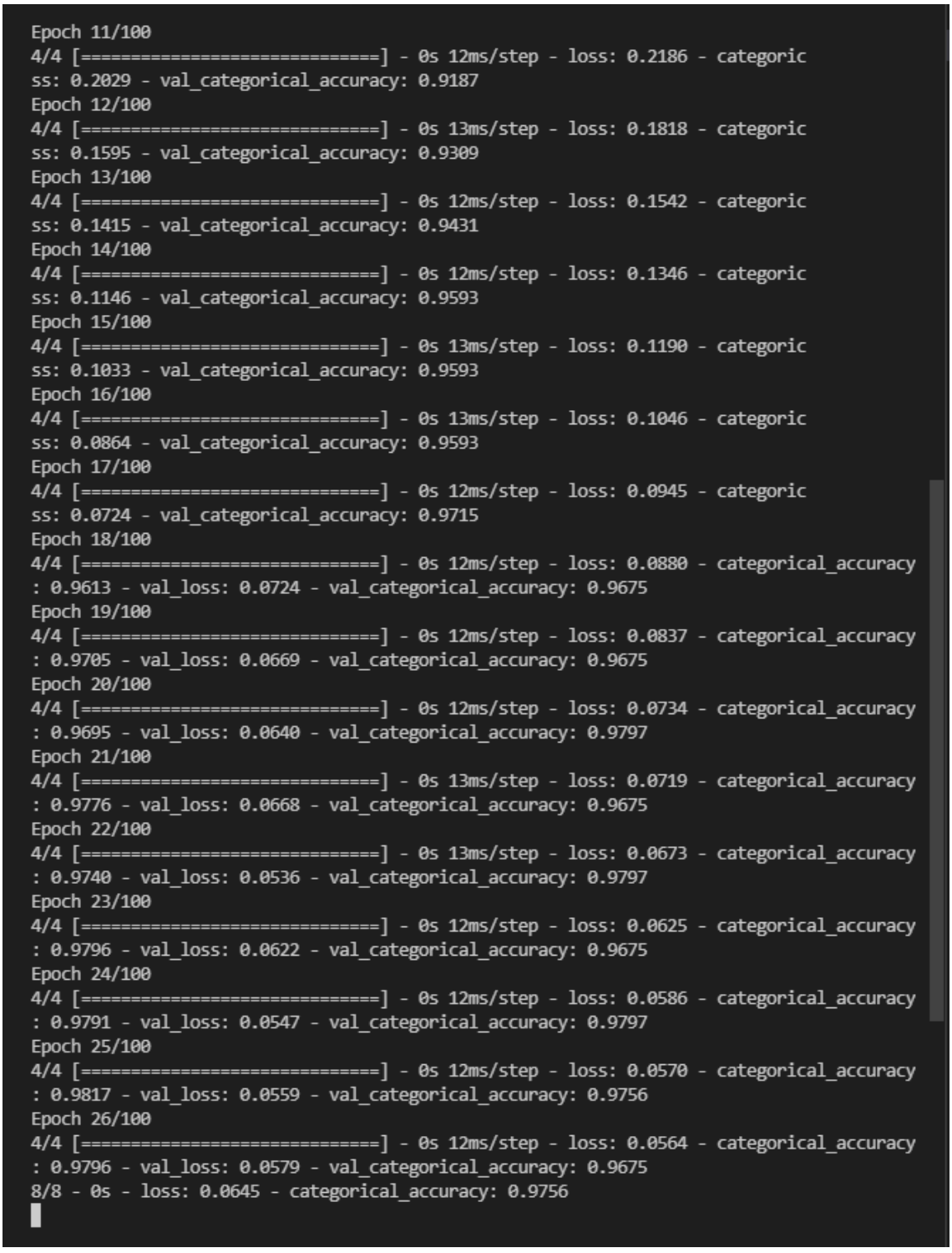

Figure 5.6: The Terminal After the Columns Have Been Chosen and the Neural Network Has Ran

CHAPTER 6

CONCLUSIONS

For this project, two neural networks were developed that were able to read CSV files and extract useful columns of Cα data in order to predict the secondary structure type of a particular protein. These networks were developed on Google Colab, an iPython notebook-based collaborative environment. Both of the developed networks were able to exceed the expected accuracy prediction of 80%.

Pandas, Numpy, and Tensorflow are already well-established data science packages for Python, and the success of the neural networks can be attributed to these packages. Although Pandas is not strictly a machine learning library, using it as a CSV reader and utilizing its useful Dataframe data structure made the preprocessing stage efficient. Numpy made the array computing easy and helped to install only the necessary computing tools. Tensorflow, being a famous machine-learning library, helped to successfully process the very large dataset of amino acid information (provided by Dr. Ali Sekmen). The overall network being created over Google Colab allowed the group to easily make changes and add comments over different devices.

The capacity to predict protein structures precisely based on their amino acid sequence would be a major boost to life sciences and medicine. It would greatly expedite attempts to comprehend the building components of cells, allowing for faster and more advanced drug development. Proteins are responsible for the majority of what occurs within cells. The 3D structure of a protein determines how it operates and what it does. For decades, laboratory experiments were the primary means of obtaining accurate protein structures. With the neural network, the secondary structure of a protein is able to be accurately predicted without having to conduct any laboratory experiments. Those unknown structures become more accessible and affordable through prediction from machine learning.

In conclusion, the team was able to learn different machine learning techniques and libraries in Python that were unfamiliar and were able to successfully create multiple neural networks that were able to predict proteins' secondary structures with over 80% accuracy. This was a challenge and learning experience for each of the team members, and we were able to solve a small portion of a larger real world problem that many scientists face today.

Appendices

APPENDIX 1

SOURCE CODES

A.1 3-Class FCNN

The code seen below generates, trains, and evaluates the 3-class fully connected neural network (FCNN).

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:



import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Make numpy values easier to read.
np.set_printoptions(precision=3, suppress=True)

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing


# ## Load CSV

# In[3]:


df = pd.read_csv('../ProteinData.csv')
df.head(25)


# ## Extract required columns
```

```
#

# In[4]:


prot_features = df.copy()
import re
prot_features.drop(columns=filter(lambda x: not
    re.match(r'f\d+|neighbors|ProteinID|SStype', x) ,prot_features.columns), inplace=True)
prot_features.dropna(inplace=True)
prot_features.head(5)


# ## Extract test data

# In[5]:


s801010 = True
if not s801010:
  test_data = ["4OH7", "5YDE", "2OPC", "6YDR", "6NZS", "1EAR", "2FP1", "2Z6R",
               "2OIT", "5JUH", "4B2O", "2JDA", "3LFK", "1Z6N", "6P8O", "5UEB",
               "5YDE", "3V4K", "4ZDS", "4WKA"]
  test_data_df = prot_features[prot_features["ProteinID"].isin(test_data)]
  test_data_df


# ## Extract remaining data

# In[6]:


if not s801010:
  train_data_df = prot_features[~prot_features["ProteinID"].isin(test_data)]
  train_data_df


# In[7]:


if s801010:
```

```
  train_data_df, val_data_df = train_test_split(prot_features, test_size=0.2)
  test_data_df, val_data_df = train_test_split(val_data_df, test_size=0.5)


# ## Split into training and test data

# In[8]:


train_data_df, val_data_df = train_test_split(train_data_df, test_size=0.1)


# ## Split out labels

# In[9]:


def breakdown(df):
    df = df.copy()
    df.drop(columns=["ProteinID"], inplace=True)
    labels = df.pop("SStype")
    labels = labels.astype('category').cat.codes
    return np.array(df), tf.keras.utils.to_categorical(labels)
x_train, y_train = breakdown(train_data_df)
x_test, y_test = breakdown(test_data_df)
x_val, y_val = breakdown(val_data_df)


# ## Create normalization layer
#

# In[10]:


#Normalization
normalize = preprocessing.Normalization()
normalize.adapt(x_train)


# ## Create network model
#
```

```
110
111 # In[11]:
112
113
114 # prot_model = tf.keras.Sequential([
115 #   normalize,
116 #   layers.Dense(128, activation='relu'),
117 #   layers.Dense(64, activation='relu'),
118 #   layers.Dense(32, activation='relu'),
119 #   layers.Dense(16, activation='relu'),
120 #   #layers.Dense(128, activation='relu'),
121 #   #layers.Dense(128, activation='relu'),
122 #   layers.Dense(3, activation='softmax')
123 # ])
124
125 prot_model = tf.keras.Sequential([
126   normalize,
127   layers.Dense(68, activation='relu'),
128   layers.Dense(68, activation='relu'),
129   layers.Dense(34, activation='relu'),
130   layers.Dense(51, activation='relu'),
131   layers.Dense(51, activation='relu'),
132   layers.Dense(3, activation='softmax')
133 ])
134
135 prot_model.compile(loss = tf.losses.CategoricalCrossentropy(),
        optimizer=tf.optimizers.Adam(), metrics=['categorical_accuracy'])
136
137
138 # ## Train the model, and evaluate its performance
139
140 # In[12]:
141
142
143 prot_model.fit(x_train, y_train, epochs=100, batch_size=500, validation_data=(x_val,
        y_val), callbacks = [tf.keras.callbacks.EarlyStopping(patience = 4)])
144 prot_model.evaluate(x_test, y_test, verbose=2)
145
146
147 # ## Create the confusion matrix
148 #
```

```
#

# In[13]:


print(prot_features['SStype'].astype('category').cat.categories)
tf.math.confusion_matrix(np.argmax(y_test, axis=1), np.argmax(prot_model.predict(x_test),
    axis=1))


# ## Performance Profiling

# In[14]:


import timeit
timeper = timeit.timeit(lambda:prot_model.predict(x_test[:1000]), number=50)/50000
print(f"Approximately {timeper}s per sample")


# ## ROC

# In[15]:


import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, auc, roc_auc_score
from itertools import cycle

lw = 2

y_pred_roc = prot_model.predict(x_train)

y_train_roc = y_train.copy()

n_classes = len(y_test[0])

fpr = dict()
tpr = dict()
roc_auc = dict()
```

```
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train_roc[:, i], y_pred_roc[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

fpr["micro"], tpr["micro"], _ = roc_curve(y_train_roc.ravel(), y_pred_roc.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

cat_names = prot_features['SStype'].astype('category').cat.categories

# Plot all ROC curves
plt.rcParams['figure.figsize'] = [10, 8]
plt.figure()
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0}: {1} (area = {2:0.2f})'
```

```
            ''.format(i, cat_names[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.show()
```

A.2 7-Class FCNN

The code seen below generates, trains, and evaluates the 7-class fully connected neural network (FCNN).

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:



import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Make numpy values easier to read.
np.set_printoptions(precision=3, suppress=True)

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing


# In[3]:


df = pd.read_csv('../ProteinData.csv')
df.head(25)

```

```
25
26 # In[4]:
27
28
29 prot_features = df.copy()
30 sstypes = list(zip(prot_features['Num'], prot_features['SStype']))
31 nsstypes = list(map(lambda x: sstypes[x[0] + 1][1] if x[0] + 1 < len(sstypes) and
       sstypes[x[0] + 1][0] == x[1][0] + 1 else np.nan, enumerate(sstypes)))
32 psstypes = list(map(lambda x: sstypes[x[0] - 1][1] if x[0] - 1 > 0 and sstypes[x[0] - 1][0]
       == x[1][0] - 1 else np.nan, enumerate(sstypes)))
33 ngsstypes = [set(filter(lambda x: type(x) is str, [i, j, k])) for i, j, k in zip(nsstypes,
       psstypes, (l for _, l in sstypes))]
34 prot_features['NSStype'] = pd.Series([''.join(sorted(list(i))) if i else np.nan for i in
       ngsstypes])
35 del sstypes
36 prot_features.iloc[160:170]
37
38
39 # In[6]:
40
41
42 prot_features.iloc[15:30]
43
44
45 # In[7]:
46
47
48 #prot_features = df.copy()
49 import re
50 prot_features.drop(columns=filter(lambda x: not
       re.match(r'f\d+|neighbors|ProteinID|NSStype', x) ,prot_features.columns), inplace=True)
51 prot_features.dropna(inplace=True)
52 prot_features.head(5)
53
54
55 # Extract test data
56
57 # In[9]:
58
59
60 s801010 = True
```

```
if not s801010:
  test_data = ["4OH7", "5YDE", "2OPC", "6YDR", "6NZS", "1EAR", "2FP1", "2Z6R",
               "2OIT", "5JUH", "4B2O", "2JDA", "3LFK", "1Z6N", "6P8O", "5UEB",
               "5YDE", "3V4K", "4ZDS", "4WKA"]
  test_data_df = prot_features[prot_features["ProteinID"].isin(test_data)]
  test_data_df.columns


# Extract remaining data

# In[10]:


if not s801010:
  train_data_df = prot_features[~prot_features["ProteinID"].isin(test_data)]
  train_data_df


# Split into training and test data

# In[11]:


if not s801010:
  train_data_df, val_data_df = train_test_split(train_data_df, test_size=0.1)


# ## 80-10-10 Optional split

# In[12]:


if s801010:
  train_data_df, val_data_df = train_test_split(prot_features, test_size=0.2)
  test_data_df, val_data_df = train_test_split(val_data_df, test_size=0.5)


# In[13]:


def breakdown(df : pd.DataFrame):
```

```
    df = df.copy()
    df.drop(columns=["ProteinID"], inplace=True)
    labels = df.pop("NSStype")
    labels = labels.astype('category').cat.codes
    return np.array(df), tf.keras.utils.to_categorical(labels)
x_train, y_train = breakdown(train_data_df)
x_test, y_test = breakdown(test_data_df)
x_val, y_val = breakdown(val_data_df)


# In[14]:


#Normalization
normalize = preprocessing.Normalization()
normalize.adapt(x_train)


# In[15]:


prot_model = tf.keras.Sequential([
  normalize,
  layers.Dense(68, activation='relu'),
  layers.Dense(68, activation='relu'),
  layers.Dense(34, activation='relu'),
  layers.Dense(51, activation='relu'),
  layers.Dense(len(y_train[0]), activation='softmax')
])
#{'sz_kernel': 4, 'n_sh': 290, 'n_filters': 64, 'n_conv': 2, 'n_base': 17}
# prot_model = tf.keras.Sequential([
#   normalize,
#   layers.Reshape((34, 1)),
#   layers.Conv1D(64, 4, activation='relu', input_shape=(34, 1)),
#   layers.Conv1D(64, 4, activation='relu'),
#   layers.Flatten(),
#   layers.Dense(51, activation='relu'),
#   layers.Dense(17, activation='relu'),
#   layers.Dense(51, activation='relu'),
#   layers.Dense(17, activation='relu'),
#   layers.Dense(len(y_train[0]), activation='softmax')
```

```
# ])

prot_model.compile(loss = tf.losses.CategoricalCrossentropy(),
    optimizer=tf.optimizers.Adam(), metrics=['categorical_accuracy'])


# In[16]:


prot_model.fit(x_train, y_train, epochs=100, batch_size=500, validation_data=(x_val,
    y_val), callbacks = [tf.keras.callbacks.EarlyStopping(patience = 4)])
prot_model.evaluate(x_test, y_test, verbose=2)


# ## Generate confusion matrix

# In[17]:


print(prot_features['NSStype'].astype('category').cat.categories)
cat_names = prot_features['NSStype'].astype('category').cat.categories
tf.math.confusion_matrix(np.argmax(y_test, axis=1), np.argmax(prot_model.predict(x_test),
    axis=1))


# ## Performance Profiling

# In[18]:


import timeit
timeper = timeit.timeit(lambda:prot_model.predict(x_test[:1000]), number=50)/50000
print(f"Approximately {timeper}s per sample")


# ## ROC

# In[19]:


import matplotlib.pyplot as plt
```

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, auc, roc_auc_score
from itertools import cycle

lw = 2

y_pred_roc = prot_model.predict(x_train)

y_train_roc = y_train.copy()

n_classes = len(y_test[0])


fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train_roc[:, i], y_pred_roc[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

fpr["micro"], tpr["micro"], _ = roc_curve(y_train_roc.ravel(), y_pred_roc.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.rcParams['figure.figsize'] = [10, 8]
plt.figure()
```

```
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'pink', 'darkgreen', 'maroon',
    'olive'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0}: {1} (area = {2:0.2f})'
             ''.format(i, cat_names[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.show()
```

A.3 3-Class DCNN

The code seen below generates, trains, and evaluates the 3-class deep convolutional neural network (DCNN).

```
#!/usr/bin/env python
# coding: utf-8

# ## Import everything
#
#
#

# In[2]:
```

```
10
11
12 import pandas as pd
13 import numpy as np
14 from sklearn.model_selection import train_test_split
15
16 # Make numpy values easier to read.
17 np.set_printoptions(precision=3, suppress=True)
18
19 import tensorflow as tf
20 from tensorflow.keras import layers
21 from tensorflow.keras.layers.experimental import preprocessing
22
23
24 # ## Load CSV
25
26 # In[3]:
27
28
29 df = pd.read_csv('../ProteinData.csv')
30 df.head(25)
31
32
33 # ## Extract required columns
34 #
35
36 # In[4]:
37
38
39 prot_features = df.copy()
40 import re
41 prot_features.drop(columns=filter(lambda x: not
       re.match(r'f\d+|neighbors|ProteinID|SStype', x) ,prot_features.columns), inplace=True)
42 prot_features.dropna(inplace=True)
43 prot_features.head(5)
44
45
46 # ## Extract test data
47
48 # In[6]:
49
```

```
50
51 s801010 = True
52 if not s801010:
53   test_data = ["4OH7", "5YDE", "2OPC", "6YDR", "6NZS", "1EAR", "2FP1", "2Z6R",
54               "2OIT", "5JUH", "4B2O", "2JDA", "3LFK", "1Z6N", "6P8O", "5UEB",
55               "5YDE", "3V4K", "4ZDS", "4WKA"]
56   test_data_df = prot_features[prot_features["ProteinID"].isin(test_data)]
57   test_data_df
58
59
60 # ## Extract remaining data
61
62 # In[7]:
63
64
65 if not s801010:
66   train_data_df = prot_features[~prot_features["ProteinID"].isin(test_data)]
67   train_data_df
68
69
70 # ## Split into training and test data
71
72 # In[8]:
73
74
75 if not s801010:
76   train_data_df, val_data_df = train_test_split(train_data_df, test_size=0.1)
77 else:
78   train_data_df, val_data_df = train_test_split(prot_features, test_size=0.2)
79   test_data_df, val_data_df = train_test_split(val_data_df, test_size=0.5)
80
81
82 # ## Split out labels
83
84 # In[9]:
85
86
87 def breakdown(df):
88     df = df.copy()
89     df.drop(columns=["ProteinID"], inplace=True)
90     labels = df.pop("SStype")
```

```
    labels = labels.astype('category').cat.codes
    return np.array(df), tf.keras.utils.to_categorical(labels)
x_train, y_train = breakdown(train_data_df)
x_test, y_test = breakdown(test_data_df)
x_val, y_val = breakdown(val_data_df)


# ## Normalization layer creation
#

# In[10]:


normalize = preprocessing.Normalization()
normalize.adapt(x_train)


# ## Create network model
#

# In[13]:


# model = tf.keras.models.Sequential([
#   normalize,
#   tf.keras.layers.Reshape((34, 1)),
#   tf.keras.layers.Conv1D(64, 3, activation='relu', input_shape=(34, 1)),
#   tf.keras.layers.MaxPool1D(),
#   #tf.keras.layers.Dense(128, activation='relu'),
#   tf.keras.layers.Flatten(),
#   tf.keras.layers.Dense(128, activation='relu'),
#   tf.keras.layers.Dense(64, activation='relu'),
#   tf.keras.layers.Dense(32, activation='relu'),
#   tf.keras.layers.Dense(16, activation='relu'),
#   tf.keras.layers.Dense(3, activation='softmax')
# ])

model = tf.keras.models.Sequential([
  normalize,
  tf.keras.layers.Reshape((34, 1)),
  tf.keras.layers.Conv1D(64, 4, activation='relu'),
```

```
  tf.keras.layers.Conv1D(64, 4, activation='relu'),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(17, activation='relu'),
  tf.keras.layers.Dense(34, activation='relu'),
  tf.keras.layers.Dense(34, activation='relu'),
  tf.keras.layers.Dense(17, activation='relu'),
  tf.keras.layers.Dense(len(y_train[0]), activation='softmax')
])

model.compile(optimizer=tf.optimizers.Adam(), loss=tf.losses.CategoricalCrossentropy(),
    metrics=['categorical_accuracy'])


# ## Train the model

# In[15]:


gpus = tf.config.list_physical_devices('GPU')
# print(gpus[0])
if gpus:
  with tf.device('/device:GPU:0'):
    model.fit(x_train, y_train, epochs=100, batch_size=500, validation_data=(x_val, y_val),
    callbacks=[tf.keras.callbacks.EarlyStopping(patience = 4)])
else:
  model.fit(x_train, y_train, epochs=100, batch_size=500, validation_data=(x_val, y_val),
    callbacks=[tf.keras.callbacks.EarlyStopping(patience = 4)])


# ## Create the confusion matrix
#

# In[16]:


print(prot_features['SStype'].astype('category').cat.categories)
tf.math.confusion_matrix(np.argmax(y_test, axis=1), np.argmax(model.predict(x_test),
    axis=1))


# ## Performance Profiling
```

```
169
170 # In[25]:
171
172
173 import timeit
174 timeper = timeit.timeit(lambda:model.predict(x_test[:1000]), number=50)/50000
175 print(f"Approximately {timeper}s per sample")
176
177
178 # ## ROC
179
180 # In[37]:
181
182
183 import matplotlib.pyplot as plt
184 from sklearn.preprocessing import LabelBinarizer
185 from sklearn.metrics import roc_curve, auc, roc_auc_score
186 from itertools import cycle
187
188 lw = 2
189
190 y_pred_roc = model.predict(x_train)
191
192 y_train_roc = y_train.copy()
193
194 n_classes = len(y_test[0])
195
196 fpr = dict()
197 tpr = dict()
198 roc_auc = dict()
199 for i in range(n_classes):
200     fpr[i], tpr[i], _ = roc_curve(y_train_roc[:, i], y_pred_roc[:, i])
201     roc_auc[i] = auc(fpr[i], tpr[i])
202
203 fpr["micro"], tpr["micro"], _ = roc_curve(y_train_roc.ravel(), y_pred_roc.ravel())
204 roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
205
206 # First aggregate all false positive rates
207 all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))
208
209 # Then interpolate all ROC curves at this points
```

```
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

cat_names = prot_features['SStype'].astype('category').cat.categories

# Plot all ROC curves
plt.rcParams['figure.figsize'] = [10, 8]
plt.figure()
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0}: {1} (area = {2:0.2f})'
             ''.format(i, cat_names[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.show()
```

```
251
252 # ## Evaluate the model
253
254 # In[38]:
255
256
257 model.evaluate(x_test, y_test, verbose=2)
```

A.4 7-Class DCNN

The code seen below generates, trains, and evaluates the 7-class deep convolutional neural network (DCNN).

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # In[4]:
5
6
7  import pandas as pd
8  import numpy as np
9  from sklearn.model_selection import train_test_split
10
11 # Make numpy values easier to read.
12 np.set_printoptions(precision=3, suppress=True)
13
14 import tensorflow as tf
15 from tensorflow.keras import layers
16 from tensorflow.keras.layers.experimental import preprocessing
17
18
19 # In[5]:
20
21
22 df = pd.read_csv('../ProteinData.csv')
23 df.head(25)
24
25
26 # In[6]:
27
```

```
28
29 prot_features = df.copy()
30 sstypes = list(zip(prot_features['Num'], prot_features['SStype']))
31 nsstypes = list(map(lambda x: sstypes[x[0] + 1][1] if x[0] + 1 < len(sstypes) and
       sstypes[x[0] + 1][0] == x[1][0] + 1 else np.nan, enumerate(sstypes)))
32 psstypes = list(map(lambda x: sstypes[x[0] - 1][1] if x[0] - 1 > 0 and sstypes[x[0] - 1][0]
       == x[1][0] - 1 else np.nan, enumerate(sstypes)))
33 ngsstypes = [set(filter(lambda x: type(x) is str, [i, j, k])) for i, j, k in zip(nsstypes,
       psstypes, (l for _, l in sstypes))]
34 prot_features['NSStype'] = pd.Series([''.join(sorted(list(i))) if i else np.nan for i in
       ngsstypes])
35 del sstypes
36 prot_features.iloc[160:170]
37
38
39 # In[8]:
40
41
42 prot_features.iloc[15:30]
43
44
45 # In[9]:
46
47
48 #prot_features = df.copy()
49 import re
50 prot_features.drop(columns=filter(lambda x: not
       re.match(r'f\d+|neighbors|ProteinID|NSStype', x) ,prot_features.columns), inplace=True)
51 prot_features.dropna(inplace=True)
52 prot_features.head(5)
53
54
55 # Extract test data
56
57 # In[11]:
58
59
60 s801010 = True
61 if not s801010:
62   test_data = ["4OH7", "5YDE", "2OPC", "6YDR", "6NZS", "1EAR", "2FP1", "2Z6R",
63               "2OIT", "5JUH", "4B2O", "2JDA", "3LFK", "1Z6N", "6P8O", "5UEB",
```

```
                "5YDE", "3V4K", "4ZDS", "4WKA"]
  test_data_df = prot_features[prot_features["ProteinID"].isin(test_data)]
  test_data_df.columns


# Extract remaining data

# In[12]:


if not s801010:
  train_data_df = prot_features[~prot_features["ProteinID"].isin(test_data)]
  train_data_df


# Split into training and test data

# In[13]:


if not s801010:
  train_data_df, val_data_df = train_test_split(train_data_df, test_size=0.1)


# ## 80-10-10 Optional split

# In[14]:


if s801010:
  train_data_df, val_data_df = train_test_split(prot_features, test_size=0.2)
  test_data_df, val_data_df = train_test_split(val_data_df, test_size=0.5)


# In[15]:


def breakdown(df : pd.DataFrame):
    df = df.copy()
    df.drop(columns=["ProteinID"], inplace=True)
    labels = df.pop("NSStype")
```

```
    labels = labels.astype('category').cat.codes
    return np.array(df), tf.keras.utils.to_categorical(labels)
x_train, y_train = breakdown(train_data_df)
x_test, y_test = breakdown(test_data_df)
x_val, y_val = breakdown(val_data_df)


# In[16]:


#Normalization
normalize = preprocessing.Normalization()
normalize.adapt(x_train)


# In[17]:


# prot_model = tf.keras.Sequential([
#   normalize,
#   layers.Dense(68, activation='relu'),
#   layers.Dense(68, activation='relu'),
#   layers.Dense(34, activation='relu'),
#   layers.Dense(51, activation='relu'),
#   #layers.Dense(128, activation='relu'),
#   #layers.Dense(128, activation='relu'),
#   layers.Dense(len(y_train[0]), activation='softmax')
# ])

prot_model = tf.keras.Sequential([
  normalize,
  layers.Reshape((34, 1)),
  layers.Conv1D(64, 4, activation='relu', input_shape=(34, 1)),
  layers.Conv1D(64, 4, activation='relu'),
  layers.Flatten(),
  layers.Dense(51, activation='relu'),
  layers.Dense(17, activation='relu'),
  layers.Dense(51, activation='relu'),
  layers.Dense(17, activation='relu'),
  layers.Dense(len(y_train[0]), activation='softmax')
])
```

```
146
147 prot_model.compile(loss = tf.losses.CategoricalCrossentropy(),
        optimizer=tf.optimizers.Adam(), metrics=['categorical_accuracy'])
148
149
150 # In[18]:
151
152
153 prot_model.fit(x_train, y_train, epochs=100, batch_size=500, validation_data=(x_val,
        y_val), callbacks = [tf.keras.callbacks.EarlyStopping(patience = 4)])
154 prot_model.evaluate(x_test, y_test, verbose=2)
155
156
157 # ## Generate confusion matrix
158
159 # In[19]:
160
161
162 print(prot_features['NSStype'].astype('category').cat.categories)
163 cat_names = prot_features['NSStype'].astype('category').cat.categories
164 tf.math.confusion_matrix(np.argmax(y_test, axis=1), np.argmax(prot_model.predict(x_test),
        axis=1))
165
166
167 # ## Performance Profiling
168
169 # In[20]:
170
171
172 import timeit
173 timeper = timeit.timeit(lambda:prot_model.predict(x_test[:1000]), number=50)/50000
174 print(f"Approximately {timeper}s per sample")
175
176
177 # ## ROC
178
179 # In[21]:
180
181
182 import matplotlib.pyplot as plt
183 from sklearn.preprocessing import LabelBinarizer
```

```
from sklearn.metrics import roc_curve, auc, roc_auc_score
from itertools import cycle

lw = 2

y_pred_roc = prot_model.predict(x_train)

y_train_roc = y_train.copy()

n_classes = len(y_test[0])

fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_train_roc[:, i], y_pred_roc[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

fpr["micro"], tpr["micro"], _ = roc_curve(y_train_roc.ravel(), y_pred_roc.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(n_classes):
    mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

# Finally average it and compute AUC
mean_tpr /= n_classes

fpr["macro"] = all_fpr
tpr["macro"] = mean_tpr
roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
plt.rcParams['figure.figsize'] = [10, 8]
plt.figure()
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
```

```
                ''.format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
               ''.format(roc_auc["macro"]),
         color='navy', linestyle=':', linewidth=4)

colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'pink', 'darkgreen', 'maroon',
    'olive'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='ROC curve of class {0}: {1} (area = {2:0.2f})'
             ''.format(i, cat_names[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curves')
plt.legend(loc="lower right")
plt.show()
```

A.5 Principle Component Analysis

The code seen below performs basic PCA actions, including calculating matrix rank, as well as the SVD.

```
#!/usr/bin/env python
# coding: utf-8

# In[20]:


import pandas as pd
import numpy as np
import re
from sklearn.model_selection import train_test_split

```

```

# In[21]:


df = pd.read_csv('../ProteinData.csv')
df.drop(columns=filter(lambda x: not re.match(r'f\d+|neighbors', x), df.columns),
    inplace=True)
df.dropna(inplace=True)


# In[22]:


analysis_df, _ = train_test_split(df, train_size=2000)
analysis_mat = np.matrix(analysis_df)
analysis_mat


# In[23]:


np.linalg.matrix_rank(analysis_mat)


# In[24]:


u, s, vh = np.linalg.svd(analysis_mat, full_matrices=False)


# In[25]:


u.shape, s.shape, vh.shape


# In[29]:


smat = np.diag(s)
mat_reduced = np.dot(u, np.dot(smat, vh))
```

```
mat_reduced
```

A.6 Graphical User Interface

The code seen below presents the Graphical User Interface with selectable parameters to be used for prediction with the 3-Class Neural Network in determining Secondary Structure Type of a protein.

```

import tkinter as tk
from pathlib import Path
from tkinter import ttk
from tkinter.constants import BOTTOM

from TkinterDnD2 import DND_FILES, TkinterDnD
from tkinter import *

###### Neural Network Imports

import pandas as pd
import tensorflow as tf
import numpy as np
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing
from sklearn.model_selection import train_test_split

import sys
sys.path.append("/path/to/script/file/directory/")



class Application(TkinterDnD.Tk):
    def __init__(self):
        super().__init__()

        self.title("Senior Project 2 GUI: CSV Reader/Protein SStype Neural Network Generator")
        self.main_frame = tk.Frame(self)
        self.main_frame.pack(fill="both", expand="true")
```

```
        self.geometry("900x700")
        self.search_page = SearchPage(parent=self.main_frame)
```

```
class DataTable(ttk.Treeview):
    def __init__(self, parent):
        super().__init__(parent)
        scroll_Y = tk.Scrollbar(self, orient="vertical", command=self.yview)
        scroll_X = tk.Scrollbar(self, orient="horizontal", command=self.xview)
        self.configure(yscrollcommand=scroll_Y.set, xscrollcommand=scroll_X.set)
        scroll_Y.pack(side="right", fill="y")
        scroll_X.pack(side="bottom", fill="x")
        self.stored_dataframe = pd.DataFrame()


        # Treeview
        self.neural_net = NeuralNetwork(parent)
        self.neural_net.place(rely=0.50, relx=0.25, relwidth=0.75, relheight=0.50)

    def set_datatable(self, dataframe):
        self.stored_dataframe = dataframe
        self._draw_table(dataframe.head(100))



    def _draw_table(self, dataframe):
        self.delete(*self.get_children())
        columns = list(dataframe.columns)
        self.__setitem__("column", columns)
        self.__setitem__("show", "headings")

        for col in columns:
            self.heading(col, text=col)

        df_rows = dataframe.to_numpy().tolist()
        for row in df_rows:
            self.insert("", "end", values=row)
```

```
        return None


    def find_value(self, pairs):
        # pairs is a dictionary
        new_df = self.stored_dataframe
        for col, value in pairs.items():
            query_string = f"{col}.str.contains('{value}')"
            new_df = new_df.query(query_string, engine="python")
        self._draw_table(new_df)


    def reset_table(self):
        self._draw_table(self.stored_dataframe)


class SearchPage(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)


        self.file_names_listbox = tk.Listbox(parent, selectmode=tk.SINGLE, bg="#D7A7AA",
    fg="white")
        self.file_names_listbox.place(relheight=1, relwidth=0.25)
        self.file_names_listbox.drop_target_register(DND_FILES)
        self.file_names_listbox.dnd_bind("<<Drop>>", self.drop_inside_list_box)
        self.file_names_listbox.bind("<Double-1>", self._display_file)
        self.file_names_listbox.opening = Label(self.file_names_listbox, text="Please drop
    your CSV files over here", background="#D7A7AA", fg="white", font=("Arial", 10)
                )
        self.file_names_listbox.opening.place(relx=1.0,rely=0.5,anchor=E)


        self.search_entrybox = tk.Entry(parent)
        self.search_entrybox.place(relx=0.25, relwidth=0.75)
        self.search_entrybox.bind("<Return>", self.search_table)

        # Treeview
        self.data_table = DataTable(parent)
        self.data_table.place(rely=0.05, relx=0.25, relwidth=0.75, relheight=0.45)

```

```
        self.path_map = {}
    
    
    
    
    def drop_inside_list_box(self, event):
        file_paths = self._parse_drop_files(event.data)
        current_listbox_items = set(self.file_names_listbox.get(0, "end"))
        for file_path in file_paths:
            if file_path.endswith(".csv"):
                path_object = Path(file_path)
                file_name = path_object.name
                if file_name not in current_listbox_items:
                    self.file_names_listbox.insert("end", file_name)
                    self.path_map[file_name] = file_path

    def _display_file(self, event):
        file_name = self.file_names_listbox.get(self.file_names_listbox.curselection())
        path = self.path_map[file_name]
        df = pd.read_csv(path, error_bad_lines=False, engine ='python')
      #  if not df.empty:
        #    Button(root, text="Choose Columns to Use", bg="#E19B9F", fg="white",
    command=null).pack(side=BOTTOM)
        self.data_table.set_datatable(df)
        self.data_table.neural_net.choose_columns(df)


    def get_dataframe(self, dataframe):
        return self.data_table



    def _parse_drop_files(self, filename):
        size = len(filename)
        res = []  # list of file paths
        name = ""
        idx = 0
        while idx < size:
            if filename[idx] == "{":
                j = idx + 1
```

```
                while filename[j] != "}":
                    name += filename[j]
                    j += 1
                res.append(name)
                name = ""
                idx = j
            elif filename[idx] == " " and name != "":
                res.append(name)
                name = ""
            elif filename[idx] != " ":
                name += filename[idx]
            idx += 1
        if name != "":
            res.append(name)
        return res

    def search_table(self, event):
        # column value. [[column,value],column2=value2]....
        entry = self.search_entrybox.get()
        if entry == "":
            self.data_table.reset_table()
        else:
            entry_split = entry.split(",")
            column_value_pairs = {}
            for pair in entry_split:
                pair_split = pair.split("=")
                if len(pair_split) == 2:
                    col = pair_split[0]
                    lookup_value = pair_split[1]
                    column_value_pairs[col] = lookup_value
            self.data_table.find_value(pairs=column_value_pairs)


class NeuralNetwork(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)

        #yscrollbar = Scrollbar(parent)
        #yscrollbar.pack(side = RIGHT, fill = Y)
        #UGLY CHECKBOX vvvv
        self.neural_window = tk.Listbox(parent, selectmode=tk.SINGLE, bg="#ecb7bf",
```

```
    fg="white")
192         #BETTER LISTBOX vvvv
193         #self.neural_window = tk.Listbox(parent, selectmode=MULTIPLE, bg="black",
    fg="white", yscrollcommand = yscrollbar.set)
194         self.neural_window.place(rely=0.50, relx=0.25, relwidth=0.75, relheight=0.25)
195         self.neural_window_bottom = tk.Frame(parent, bg="#Fbd2d7")
196         self.neural_window_bottom.place(rely=0.75, relx=0.25, relwidth=0.75, relheight=0.25)
197
198         #self.column_choices = set(["SStype"])
199         #Application.add_scroll(self)
200
201
202     def genLambda(self, col):
203         return lambda: self.column_choices.add(col)
204
205
206
207
208
209     #LISTBOX VER
210     def choose_columns(self, dataframe):
211         #sbar = Scrollbar(self.neural_window, orient=VERTICAL,
    command=lbox.view).pack(side=RIGHT, fill=Y)
212         lbox = Listbox(self.neural_window, selectmode=MULTIPLE, height=40, width=109,
    listvariable=StringVar(value=list(dataframe.columns)))
213         lbox.pack(side='left', fill='y')
214         sbar = Scrollbar(self.neural_window, orient=VERTICAL, command=lbox.yview)
215         sbar.pack(side=RIGHT, fill=Y)
216         lbox.config(yscrollcommand=sbar.set)
217
218         Button(self.neural_window_bottom, text = "Run Neural Network with These Chosen
    Columns", height=5, width=70, command=lambda: self.neural_network(dataframe,
    lbox.curselection()), bg= "white", fg= "#CD5E77").pack(expand= YES)
219
220
221
222     def neural_network(self, dataframe, column_choices):
223
224         prot_features = dataframe.copy()
225         choices = set()
226         for x in column_choices:
```

```
            choices.add(dataframe.columns[x])
        choices.add("SStype")
        prot_features.drop(columns=filter(lambda x: x not in choices,
    prot_features.columns), inplace=True)
        prot_features.dropna(inplace=True)

        for choice in choices:
            if pd.api.types.is_string_dtype(prot_features[choice]):
                prot_features[choice] = prot_features[choice].astype('category').cat.codes


        train_data_df, val_data_df = train_test_split(prot_features, test_size=0.2)
        test_data_df, val_data_df = train_test_split(val_data_df, test_size=0.5)

        def breakdown(df : pd.DataFrame):
            df = df.copy()
            labels = df.pop("SStype")
            labels = labels.astype('category').cat.codes
            return np.array(df), tf.keras.utils.to_categorical(labels)
        x_train, y_train = breakdown(train_data_df)
        x_test, y_test = breakdown(test_data_df)
        x_val, y_val = breakdown(val_data_df)

        normalize = preprocessing.Normalization()
        normalize.adapt(x_train)

        prot_model = tf.keras.Sequential([
            normalize,
            layers.Reshape((len(choices)-1, 1)),
            layers.Conv1D(64, 2, activation='relu'),
            layers.Conv1D(64, 2, activation='relu'),
            layers.Flatten(),
            layers.Dense(51, activation='relu'),
            layers.Dense(17, activation='relu'),
            layers.Dense(51, activation='relu'),
            layers.Dense(17, activation='relu'),
            layers.Dense(len(y_train[0]), activation='softmax')
            ])

        prot_model.compile(loss = tf.losses.CategoricalCrossentropy(),
    optimizer=tf.optimizers.Adam(), metrics=['categorical_accuracy'])
```

```
266         prot_model.fit(x_train, y_train, epochs=100, batch_size=500,
    validation_data=(x_val, y_val), callbacks = [tf.keras.callbacks.EarlyStopping(patience
    = 4)])
267         prot_model.evaluate(x_test, y_test, verbose=2)
268
269
270
271
272
273 if __name__ == "__main__":
274     root = Application()
275     root.mainloop()
```

BIBLIOGRAPHY

[1] Jason Brownlee. *Supervised and Unsupervised Machine Learning Algorithms*. https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/. Accessed June 20, 2021.

[2] *DNA: What is, DNA, Example*. https://www.healthline.com/health/what-is-dna. Accessed June 21, 2021.

[3] *DNA: What is, DNA, Example*. https://biologydictionary.net/genetics/.html. Accessed June 21, 2021. 2013.

[4] IBM Cloud Education. *What are Neural Networks?* https://www.ibm.com/cloud/learn/neural-networks. Accessed June 22, 2021. 2020.

[5] Keith D. Foote. *A Brief History of Machine Learning*. https://www.dataversity.net/a-brief-history-of-machine-learning/. Accessed June 20, 2021.

[6] Nick Heath. *What is machine learning? Everything you need to know*. https://www.zdnet.com/article/what-is-machine-learning-everything-you-need-to-know/. Accessed June 20, 2021.

[7] Kelly Malcolm. *Designing New Proteins Could Lead to Cancer Treatment*. https://labblog.uofmhealth.org/lab-report/designing-new-proteins-could-lead-. Accessed June 13, 2021. Feb. 2019.

[8] *Proteins*. https://courses.lumenlearning.com/boundless-biology/chapter/proteins/. Accessed June 13, 2021. 2013.

[9] Sumit Saha. *A Comprehensive Guide to Deep Convolutional Neural Networks - The ELI5 Way*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. Accessed June 22, 2021. 2018.

[10] *THE MNIST DATABASE of handwritten digits*. http://yann.lecun.com/exdb/mnist/. Accessed June 22, 2021.

[11] *What is Machine Learning? A Definition.* https://www.expert.ai/blog/machine-learning-definition/. Accessed June 20, 2021.

[12] Tony Yiu. *Understanding Random Forest*. https://towardsdatascience.com/understanding-random-forest-58381e0602d2. Accessed June 22, 2021. June 2019.