Computational Biology - Project One

Barney Evans
School of Electrical and
Computer Engineering
Southampton University
Southampton, Hampshire
Email: be1g21@soton.ac.uk

1. Introduction

Accurately predicting protein secondary structures is an important challenge in computational biology, vital for understanding protein functions and interactions. This project aims to enhance Qian et al.'s original model by integrating the single improvement of local encoding instead of one-hot encoding. Additionally, it extends the model to incorporate Bidirectional Recurrent Neural Networks (BRNN), representing a state-of-the-art technique.

2. Methodology

2.1. Data Extraction and Preprocessing

Extracting and processing protein sequence data is fundamental to the study of secondary structures and their functional interactions. Following the approach taken by Qian et al., this project starts with the extraction and formatting of amino acid sequences from the test and train datasets provided [1]. Each protein sequence is processed such that both amino acids and their respective structures are mapped to numerical indices.

2.1.1. Homologous Sequences. Qian et al. detail the initial steps taken to prepare their dataset, starting with the removal of homologous protein sequences (homologies) [1]. These homologies are proteins that share a high degree of sequence similarity due to a common evolutionary origin [2]. If a model is trained on homologous sequences, it might perform exceptionally well on the test set simply because it has seen very similar sequences during training. This issue is further addressed by Qian et al. stating "results were highly sensitive to homologies" [1].

The proposed solution, by Qian et al., was the use of Diagon Plots, where the test and train amino acid sequences are plotted on each axis respectively [1]. Any overlapping amino acids are then highlighted with a dot at the corresponding grid position. Continuous diagonal lines of dots indicate regions of high similarity or homology between the sequences. These plots allow researchers to visually identify and analyze homologous regions. By identifying these regions, Qian et al. [1]. could ensure that homologies were either removed or properly managed, minimizing bias and overfitting in their predictive models.

To identify homologies within the test and train set provided, Diagon Plots were created between each protein in the test and train sequence with a red dot indicating a match in the amino acid sequence. Analysis of these plots revealed no indication of a strong presence of homologous sequences with the majority producing sparse plots, an example being showcased within figure 1. Occasionally, some plots exhibited a higher level of matches but with no significant correlation, as seen within figure 2. For reference, a paper by Heslop-Harrison et al. showcases a strong existence of a homologous sequence within figure 8 [3], found within the appendix. Due to the lack of homologies within the dataset, there was no need in implementing a method to remove them, therefore the dataset was left untreated.

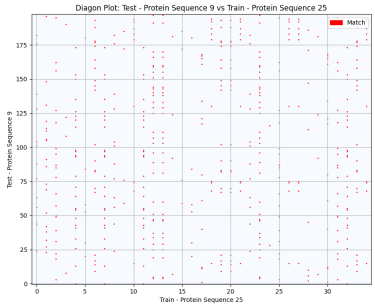

Figure 1: Protein sequence exhibiting a lack of matches.

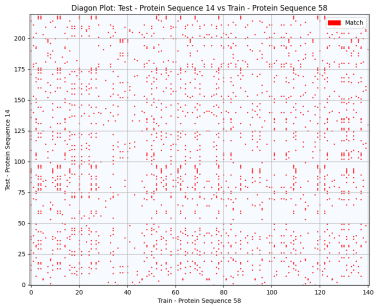

Figure 2: Protein sequence with an increased number of matches and minor correlations between matches.

2.1.2. Concatenation of Sequences. Following the identification of homologous sequences, the next step proposed by Qian et al. is the concatenation of protein sequences [1]. This process utilises spacers ('$') to replace all instances of the start and end for proteins, simplifying the input for the neural network. The spacers allow the clear distinction between protein sequences ensuring that this context is not lost within training.

By simulating the biological synthesis of proteins as continuous chains, concatenation provides a realistic scenario for neural network training, enhancing the model's generalization capabilities. This method preserves the integrity of the dataset by maintaining local context at sequence boundaries, ensuring accurate learning and predictions during the training phase [1].

2.2. Data Encoding and Context Window Implementation

2.2.1. Data Encoding. Following the concatenation of amino acid sequences, it is essential to convert this data into a format that can be processed by a neural network. This can be achieved using one-hot encoding, to vectorize each amino acid. Within this method each amino acid, previously designated an index, is represented by a list of size N, where N is the total number of amino acids with the addition of the space. This list is then filled with zeros except for the specific position relating to that index, which is filled with a one. For example, if there are twenty possible amino acids, and the amino acid in question is the third one in a standardized list, its vector will have a one in the third position and zeros elsewhere. The result is a fixed-size vector for each amino acid, which ensures that inputs to the neural network are uniform in size, thereby simplifying and enhancing the architecture of the network [1].

2.2.2. Context Window. Once the encoded data is successfully created, the next step is creating the context windows. The context window is a crucial feature used by the neural network due to its ability to capture surrounding information relative to a central amino acid, therefore enhancing the model's accuracy. Following the method denoted by Qian et al., a context window of size 13 will be used to capture sufficient local interactions between amino acids without introducing unwanted noise [1]. Additionally, the context windows will be placed randomly "to prevent erratic oscillations in the performance that occurred when the amino acids were sequentially sampled" [1]. To manage the edge cases where the context window is unable to read a full sequence, zero padding will be applied. This was a method, detailed by Lopez-del Rio et al (2020), which influences model performance by maintaining structural consistency and artificially fills the context window with zeros [4].

2.3. Model Design and Training

To most accurately follow the process that Qian et al. utilised to design their model a sequential model was chosen. This model was picked due to its ability to process sequential input data and ability to handle series of data points where the order is crucial (the amino acid sequence) [5]. This allowed the design of a neural network closely following the three-layered design detailed within Qian et al's. paper.

2.3.1. Neural Network Architecture. The neural network architecture used in this study consists of three layers: an input layer, a hidden layer, and an output layer.

Input Layer: The input layer consists of 273 nodes. This configuration is designed to replicate the window size (13 positions) multiplied by the 20 possible amino acids, plus an additional node for the spacer symbol.

Hidden Layer: The hidden layer is comprised of 40 nodes and connects to each node within the input layer. Qian et al.'s further specifies that the "state of each unit, s_i, has a real value in the range between 0 and 1" [1]. Although not specifically mentioned, the sigmoid activation function follows this logic and is therefore selected for the hidden layers.

Output Layer: The output layer contained 3 nodes, each representing one of the three possible classes: α-*helices*, β-*sheets*, and *coils*. Softmax activation is used in this layer to convert the network outputs into a probability distribution over the predicted classes, allowing results to be extracted using the argmax function. This choice is based on Qian et al.'s statement that "For a given input and set of weights, the output of the network will be a set of numbers

between 0 and 1. The secondary structure chosen was the output unit that had the highest activity level; this was equivalent to choosing the output unit that had the least mean-square error with the target outputs" [1]

See Figure 3 for a visual representation of the neural network architecture.

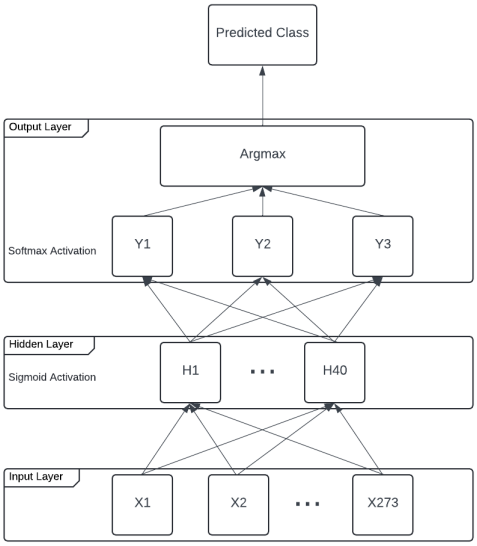

Figure 3: Neural Network Architecture implemented in the style by Qian et al. [1]

2.3.2. Training Configuration. Once the architecture had been decided the created, the model was then compiled with the loss function *categorical_crossentropy* due to its specialization in handling multi-class classification tasks. The weights in the neural network were randomly assigned with "values uniformly distributed in the range [-0.3, 0.3]" [1]. Furthermore, to prevent overfitting, Qian et al. implemented early stopping to "halt training upon no improvement in validation accuracy, ensuring the model generalizes well to unseen data" [1].

3. Model Evaluation

3.1. Performance Metrics

To evaluate the algorithm, performance metrics are used to gauge the model's ability to produce correct results. The focus will be on the accuracy (Q_3) of the model due to its emphasis in Qian et al.'s paper [1]. Additionally, the confusion matrix will be utilised to provide a detailed breakdown of correct and incorrect predictions for each class.

3.1.1. Accuracy. Accuracy measures the percentage of correctly predicted amino acid secondary structures, encompassing α-*helices*, β-*sheets*, and *coils*.

$$Q_3 = \frac{P_\alpha + P_\beta + P_{\text{coil}}}{N} \times 100 \tag{1}$$

where:

- P_α is the number of correctly predicted α-*helices*,
- P_β is the number of correctly predicted β-*sheets*,
- P_{coil} is the number of correctly predicted coils,
- N is the total number of predicted secondary structures.

The highest accuracy achieved was 0.634, with an average accuracy of 0.631, close to Qian et al.'s reported accuracy of 0.643. Figure 4 presents the average training and test accuracy across 10 runs over 34 epochs (the average number of epochs specified by the stop early algorithm). The training accuracy shows a steady increase, peaking at around 0.67, whereas the test accuracy plateaus around 0.63. This indicates that the model is overfitting due its improvement on the training set while failing to generalize to the test set. The overfitting seen in Qian et al.'s model aligns with findings from Riis et al., who also highlighted it as a significant issue in their analysis [1] [6]. This will be discussed further within the next section.

Figure 4: Comparison of training and validation accuracy across 33 epochs utilising Qian et al.'s model

4. Single Model Improvement

4.1. Literature Review

4.1.1. Complexities in Adapting Old Models to New Computational Methods. Due to many modern algorithms utilizing deep learning and requiring significant changes to the neural network architecture,

finding a single improvement to Qian et al.'s model is challenging. Ismi et al. highlight this evolution by stating, "In recent years, deep neural networks have become the primary method for protein secondary structure prediction" with 'favored deep learning methods, such as convolutional neural networks, recurrent neural networks, inception networks, and graph neural network "[7]. Jiang et al. further emphasize the complexity of these modern algorithms, noting that deep neural networks necessitate multiple, diverse hidden layers and datasets that include 'information on solvent accessible surface area, backbone angles, and dihedrals" [8]. This indicates that large-scale changes to both the neural network architecture and the preprocessing steps are required, complicating the integration of simple, singular improvements. Therefore a focus on literature shortly preceding Qian et al.'s model is required, leading to Riis et al.'s paper.

4.1.2. Local Encoding. As previously noted, the issue of overfitting in Qian et al.'s model is a significant challenge. Riis et al. identify this specific problem as being caused by the "huge number of free parameters to be estimated from the data" [1] [6]. To reduce these parameters, Riis et al. suggest adaptive encoding, more commonly known as local encoding [6]. Riis et al. break this encoding method into four steps with the first step encoding each amino acid is using two-three real numbers, between the values of zero and one, instead of the traditional orthogonal encoding, which uses 21 binary numbers. Riis et al. does not suggest how these values are determined; however, a paper by W. Taylor specifies that the secondary structure can be expressed using two physicochemical properties: molecular volume and hydrophibicity [9]. Physicochemical encoding captures more detailed and biologically significant characteristics of amino acids, potentially improving the predictive power when utilising them [9].

4.2. Implementation

To implement these physicochemical properties, we determined the molecular weights from this website [10]. Due to the large collection of different hydrophobicity scales, it was necessary to identify the optimal scale [11]. This was done post-implementation by running the system with each scale and selecting the scale with the highest accuracy. The table below showcases the accuracy values for each scale.

To closely follow Riis et al. for the remaining steps, firstly, a new hidden layer is required for weight sharing [6]. This is the process of ensuring that the same weights are used for each position in the window of amino acids, thereby enforcing consistency in the encoding across the entire window. Specifically, for each position j in the window, the weights connecting the input amino acids to the hidden units are shared,

TABLE 1: Test Accuracies for Different Hydrophobicity Scales

Hydrophobicity Scale	Best Test Accuracy
kd	0.640
hh	0.641
mf	0.604
tt	0.630
ww	0.593

meaning $w_{ij}^k = w_{ij}^l$ for all k and l.

Following this, the network's architecture is adapted to ensure the input layer corresponds to the decreased vector size. This includes determining the optimal number of nodes within the new hidden layer. An iterative process of training and validation is used to identify the number of nodes that maximizes predictive accuracy while minimizing overfitting resulting with 60 nodes. The new architecture for the neural network can be seen within Figure 5

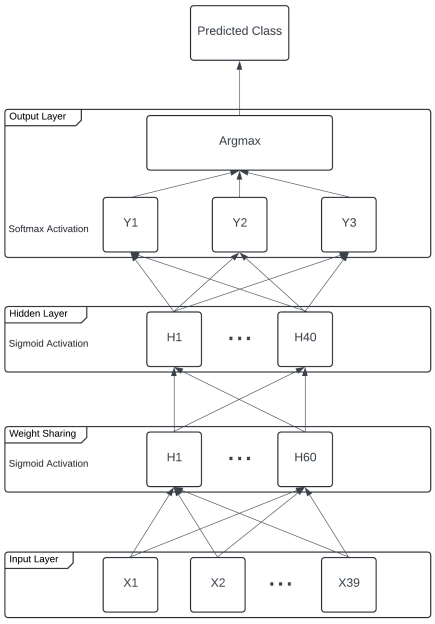

Figure 5: Neural Network Architecture implemented with local encoding

4.3. Evaluation

The model's highest accuracy was now 0.641, representing a 0.007 increase in accuracy of the previous model. With Figure 6 showcasing the resultant graph. Figure 6, in comparison to Figure 4 showcases an improvement in overfitting and accuracy with the previous model showing a diversion between the training and test

accuracies early on. The improved model showcases a steady climb, although sometimes uneven, between test and train accuracies. This is due to the integration of physicochemical property encoding, which captures essential attributes of the dataset more effectively than one-hot encoding. This encoding method considers the chemical and physical characteristics of the entities, providing a more biologically significant feature set. This shift not only leads to better model understanding of the underlying patterns but also enhances its ability to generalize from training to unseen data.

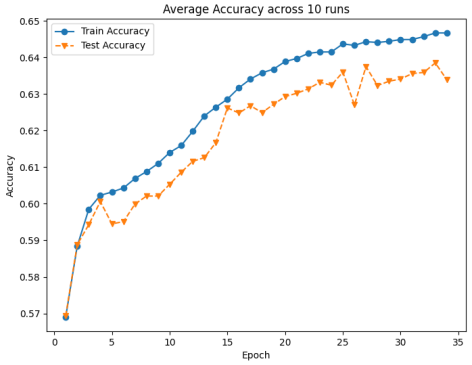

Figure 6: Comparison of training and validation accuracy across 34 epochs, for the improved model

5. Unseen Data

To test the base and improved algorithms on unseen data, Jing et al. state that the Critical Assessment of techniques for protein Structure Prediction (CASP) are highly regarded datasets within protein structure prediction [12] [13]. Furthermore these datasets do not contain any homologous sequences ensuring compatibility with Qian et al.'s model [1]. Out of the possible CASP datasets, CASP12 was selected due to it resulting in the highest benchmark accuracies within Jing et al.'s paper [13] [12].

5.1. Integration

To integrate the CASP 12 dataset into the current algorithm, the test and train dataset's provided were reformatted to that of Qian et al. dataset. These were then passed into the algorithms with no additional change required

5.2. Evaluation

To illustrate the performance when utilising both datasets, Table 2 showcases the highest accuracies for each dataset against each algorithm. From this table, it can be observed that both models have a decreased accuracy. This decrease is due to the complexity and diversity of the CASP datasets, which often include more challenging protein structures compared to previous datasets such as the one used by Qian et al. [14] [1]. This added complexity is due to the use of CASP within competitions to push the scopes required from new, modern secondary protein prediction models [14]. Therefore the dataset includes more nuances and finer detail that is not picked up by the current models.

Model	Test Accuracy	
	Original Dataset	CASP12 Dataset
Qian et al.'s	0.634	0.610
Improved	0.641	0.613
BRNN	-	0.678

TABLE 2: Comparison of Model Test Accuracies on Original and CASP12 Datasets. The Bidirectional Recurrent Neural Network (BRNN) model benchmark here uses one-hot encoding and additionally is tested and trained on the CASP 12 dataset. The other two models are trained on Qian et al.'s dataset and tested on the CASP 12 dataset.

6. BRNN Models

6.1. Literature Review

6.1.1. BRNN significance. From Table 2, we observe that the BRNN model, utilizing one-hot encoding, achieved an accuracy of 0.678 on the CASP12 dataset. This performance is higher compared to the models implemented following Qian et al.'s methodology and the implementation utilizing local encoding. This indicates the ability of BRNNs in predicting the secondary structure of amino acids more effectively than traditional neural networks. Which is further emphasized by Jing et al. with "the state-of-the-art performances by using the long short-term memory (LSTM) bidirectional recurrent neural networks" [12].

6.1.2. BRNN Overview. A Recurrent Neural Network (RNN) evolves on a traditional neural network by building up a memory of previous inputs, and taking these into account. It is specifically created to handle sequential data whereby previous context is important, such as secondary structure prediction. A BRNN builds upon this by processing data in both forward and backward directions simultaneously. This allows the BRNN to gather contextual information from both previous and future sequences [15].

Given these advancements, this study aims to extend the re-implementation of the Qian et al.'s model by integrating BRNNs [1]. This extension will retain the preprocessing stage, however implement the BRNN as an alternative to the current feed-forward neural network.

6.2. Methodology

To implement the BRNN, the method set out within Jing et al.'s paper was followed. This includes implementing "two BRNN layers with 256 LSTM cells and two fully connected (dense) layers with 1,024 and 512 nodes" [12]. This replaces the current architecture with the one seen within Figure X, although not stated within Jing et al.'s , sigmoid activation functions will be utilised to follow Qian et al.'s approach [12] [1].

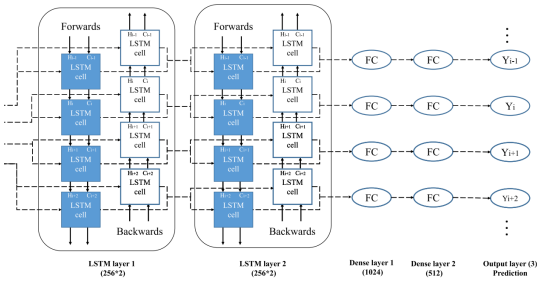

Figure 7: Architecture for BRNN model for secondary structure prediction, taken from Jing et al. [12]

6.3. Evaluation

Table 3 showcases the BRNN model tested and trained on the combination of the CASP 12 and Qian et al.'s dataset. Overall the BRNN model produces higher accuracies across the board but at a significant cost in processing speed. The number of parameters needed to train the model was 3,196,907 for the BRNN model and 11,083 for Qian et al.'s model. [1]. The increase in paramaters was the cause of the increased accuracy showcasing that the additional contextual information that was captured by the bi-directional model. Furthemore the BRNN model's performance on unseen datasets (CASP12) when trained on Qian et al.'s dataset (and vice versa) highlights its robustness and generalizabil- ity. The relatively high accuracy on unseen data (0.642) demonstrates that the model can generalize well beyond the specific dataset it was trained on. This is a crucial advantage for real-world applications where the model will encounter diverse and previously unseen protein sequences.

Training Dataset	Testing Dataset	Test Accuracy
CASP12	CASP12	0.671
CASP12	Qian et al.'s	0.662
Qian et al.'s	CASP12	0.642
Qian et al.'s	Qian et al.'s	0.662

TABLE 3: Comparison of the highest BRNN model test accuracies with different training and testing datasets. The model is trained on either the CASP 12 dataset or Qian et al.'s dataset and tested on either dataset accordingly.

7. Conclusion

This study successfully enhanced the prediction accuracy of protein secondary structures by improving upon Qian et al.'s original model. This original model achieved 0.643 and the implementation within this project resulted in 0.631. To make a single improvement on the model, local encoding was implemented in the form of physicochemical properties to represent more biologically relevant features. This lead to a small increase in the accuracy of the model going to 0.641. Further improvements were achieved by integrating BRNNs, which significantly outperformed traditional neural networks due to their ability to capture contextual information from both directions. The highest result from this method was 0.671 in comparison to the benchmark of 0.678.

8. Future Work

Following the basic implementation of the state-of-the-art BRNN networks using one-hot encoding, the next step would be to utilise different encoding schemes to achieve a higher test accuracy. Jin et al. state that on the CASP 12 benchmark, the use of PSSM encoding with a BRNN network results in test accuracies of 0.794 [12].

References

[1] Ning Qian and Terrence J. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202(4):865–884, 1988.

[2] Pablo Mier, Antonio J. Pérez-Pulido, and Miguel A. Andrade-Navarro. Automated selection of homologs to track the evolutionary history of proteins. *BMC Bioinformatics*, 19(1):431, 11 2018.

[3] J. S. Heslop-Harrison and Trude Schwarzacher. Domestication, genomics and the future for banana. *Annals of Botany*, 100(5):1073–1084, Nov 2007. Research Support, Non-U.S. Gov't; Review.

[4] Angela Lopez del Rio, Maria Martin, Alexandre Perera-Lluna, and Rabie Saidi. Effect of sequence padding on the performance of deep learning models in archaeal protein functional prediction. *Scientific Reports*, 10(1):14634, 2020.

[5] Ridzuan. Building sequential models with keras: A comprehensive guide for deep learning. https://example.com/building-sequential-models-with-keras-a-comprehensive-guide-for-deep-learning, February 23 2023. Accessed: [Your Access Date].

[6] Søren Riis and Anders Krogh. Improving prediction of protein secondary structure using structured neural networks and multiple sequence alignments. *Journal of computational biology : a journal of computational molecular cell biology*, 3:163–83, 02 1996.

[7] Dewi Pramudi Ismi, Reza Pulungan, and Afiahayati. Deep learning for protein secondary structure prediction: Pre and post-alphafold. *Computational and Structural Biotechnology Journal*, 20:6271–6286, 2022. PubMed-not-MEDLINE.

[8] Qian Jiang, Xin Jin, Shin-Jye Lee, and Shaowen Yao. Protein secondary structure prediction: A survey of the state of the art. *Journal of Molecular Graphics and Modelling*, 76:379–402, 2017.

[9] William Ramsay Taylor. The classification of amino acid conservation. *Journal of Theoretical Biology*, 119(2):205–218, 1986.

[10] Promega Corporation. Amino acid structures, codes and reference information, 2024. Accessed: 2024-06-11.

[11] UCSF Computer Graphics Laboratory. Amino acid hydrophobicity, 2018. Accessed: 2024-06-11.

[12] Xiaoyang Jing, Qiwen Dong, Daocheng Hong, and Ruqian Lu. Amino acid encoding methods for protein sequences: A comprehensive review and assessment. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(6):1918–1931, 2020.

[13] Protein Structure Prediction Center. Casp12 protein structure prediction dataset, 2016. University of California, Davis. Accessed: 2024-06-10.

[14] Andriy Kryshtafovych, Torsten Schwede, Maya Topf, Krzysztof Fidelis, and John Moult. Critical assessment of methods of protein structure prediction (casp)—round xiii. *Proteins: Structure, Function, and Bioinformatics*, 87(12):1011–1020, 2019.

[15] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

Appendix

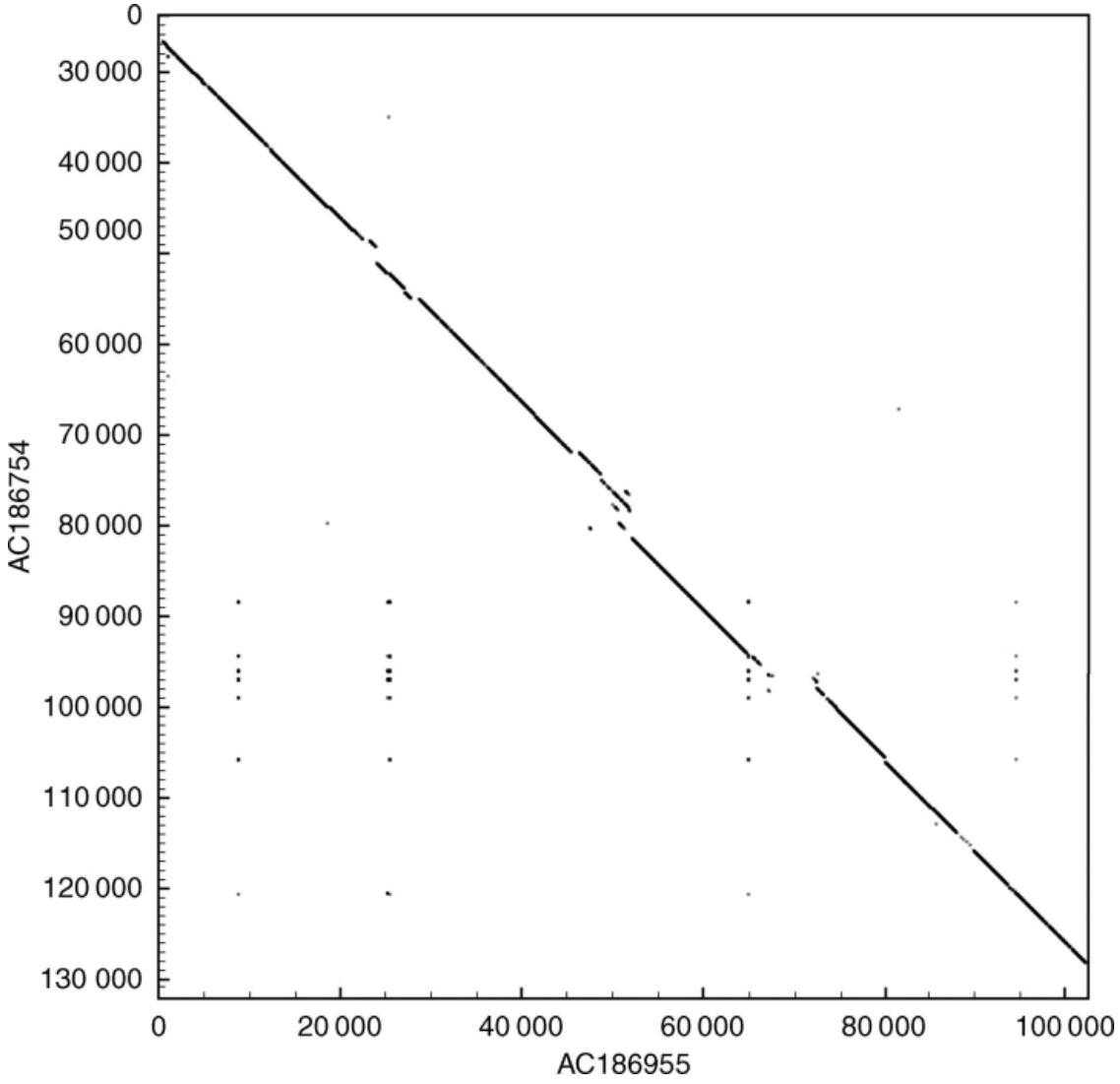

Figure 8: Homologous protein sequences

8.1. Preprocessing

8.1.1. Data Extraction.

```
train_file_path = r"C:\Users\evans\OneDrive - University of Southampton\Desktop\Year 3\Semester
    2\Computational Biology\Coursework\pythonProject5\Train_Test_Data\protein-secondary-structure
    .train.txt"
test_file_path = r"C:\Users\evans\OneDrive - University of Southampton\Desktop\Year 3\Semester 2\
    Computational Biology\Coursework\pythonProject5\Train_Test_Data\protein-secondary-structure.
    test.txt"
#train_file_path = r"C:\Users\evans\OneDrive - University of Southampton\Desktop\Year 3\Semester
    2\Computational Biology\Coursework\pythonProject5\Train_Test_Data\casp12_train.txt"
#test_file_path = r"C:\Users\evans\OneDrive - University of Southampton\Desktop\Year 3\Semester
    2\Computational Biology\Coursework\pythonProject5\Train_Test_Data\casp12_test.txt"

amino_acid_map = {
    'A': 0, 'C': 1, 'D': 2, 'E': 3, 'F': 4, 'G': 5, 'H': 6,
    'I': 7, 'K': 8, 'L': 9, 'M': 10, 'N': 11, 'P': 12, 'Q': 13,
    'R': 14, 'S': 15, 'T': 16, 'V': 17, 'W': 18, 'Y': 19,
```

```
    '$': 20
    }

structure_map = {'_': 0, 'e': 1, 'h': 2}

def load_and_separate_sequences(file_path):

    sequences = []
    current_sequence = []
    is_test_file = 'test' in file_path
    start_marker = '<>'
    end_marker = '<end>' if is_test_file else 'end'

    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            if line == start_marker:
                if current_sequence:
                    sequences.append(current_sequence)
                current_sequence = []
            elif line == end_marker:
                if current_sequence:
                    sequences.append(current_sequence)
                current_sequence = []
            elif line:
                parts = line.split()
                if len(parts) == 2 and parts[0] in amino_acid_map and parts[1] in structure_map:
                    amino_acid_idx = amino_acid_map[parts[0]]
                    structure_idx = structure_map[parts[1]]
                    current_sequence.append((amino_acid_idx, structure_idx))
                else:
                    print(f"Skipping malformed line: {line}")

        if current_sequence:
            sequences.append(current_sequence)

    return sequences

def concatenate_sequences(sequences, spacer='$'):

    concatenated_sequence = []
    for sequence in sequences:
        concatenated_sequence.extend(sequence)
        concatenated_sequence.append((amino_acid_map[spacer], structure_map['_']))
    return concatenated_sequence

train_sequences = load_and_separate_sequences(train_file_path)
test_sequences = load_and_separate_sequences(test_file_path)

train_concatenated = concatenate_sequences(train_sequences)
test_concatenated = concatenate_sequences(test_sequences)

#print(train_concatenated)

print("Concatenated Train Sequence Length:", len(train_concatenated))
print("Concatenated Test Sequence Length:", len(test_concatenated))
```

8.1.2. Additional Preprocessing.

```
from Ning_Qian.Pre_Processing.Data_Extraction import test_concatenated, train_concatenated
import numpy as np
from tensorflow.keras.utils import to_categorical

"""
Preprocessing Data
"""

def one_hot_encode(indices, num_classes):
    return to_categorical(indices, num_classes=num_classes)

def prepare_data(concatenated_data, num_amino_acids=21, num_structures=3):
```

```
    amino_acids = []
    structures = []
    for aa, struct in concatenated_data:
        amino_acids.append(aa)
        structures.append(struct)

    encoded_amino_acids = [one_hot_encode(aa, num_amino_acids) for aa in amino_acids]
    encoded_structures = [one_hot_encode(struct, num_structures) for struct in structures]

    data = np.array(encoded_amino_acids)
    labels = np.array(encoded_structures)

    return data, labels

def precompute_windows(data, labels, window_size=13):
    pad_width = window_size // 2
    padded_data = np.pad(data, ((pad_width, pad_width), (0, 0)), mode='wrap')
    num_samples = data.shape[0]


    random_indices = np.random.permutation(num_samples)
    windows = np.array([padded_data[i:i + window_size].flatten() for i in random_indices])
    return windows, labels[random_indices]

train_data, train_labels = prepare_data(train_concatenated)
test_data, test_labels = prepare_data(test_concatenated)

train_windows, train_window_labels = precompute_windows(train_data, train_labels)
test_windows, test_window_labels = precompute_windows(test_data, test_labels)
```

8.2. Qian's Model Configuration

```
from tensorflow.keras.models import Sequential
from Ning_Qian.Pre_Processing.More_Preprocessing import train_window_labels, train_windows,
    test_window_labels, test_windows
from tensorflow.keras.initializers import RandomUniform
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.callbacks import EarlyStopping
import time




num_amino_acids = 21
window_size = 13
input_dim = window_size * num_amino_acids




model = Sequential([
    Input(shape=(input_dim,)),
    Dense(40, activation='sigmoid', kernel_initializer=RandomUniform(minval=-0.3, maxval=0.3)),
    Dense(3, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

early_stopping = EarlyStopping(
)

model.summary()


# Train the model with early stopping
model.fit(
    train_windows, train_window_labels,
    epochs=200,
    batch_size=32,
```

```
    validation_data=(test_windows, test_window_labels),
    verbose=1,
    callbacks=[early_stopping]
)

test_loss, test_accuracy = model.evaluate(test_windows, test_window_labels)
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")
```

8.3. Local Encoding

8.3.1. Encoding Creation.

```
import json

amino_acid_properties = {
    'A': {'molecular_weight': 89,  'kd': 1.8,  'ww': -0.17, 'hh': 0.11,  'mf': 0.0,   'tt':
        0.38},
    'R': {'molecular_weight': 174, 'kd': -4.5, 'ww': -0.81, 'hh': 2.58,  'mf': 3.71,  'tt':
        -2.57},
    'N': {'molecular_weight': 132, 'kd': -3.5, 'ww': -0.42, 'hh': 2.05,  'mf': 3.47,  'tt':
        -1.62},
    'D': {'molecular_weight': 133, 'kd': -3.5, 'ww': -1.23, 'hh': 3.49,  'mf': 2.95,  'tt':
        -3.27},
    'C': {'molecular_weight': 121, 'kd': 2.5,  'ww': 0.24,  'hh': -0.13, 'mf': 0.49,  'tt':
        -0.30},
    'Q': {'molecular_weight': 146, 'kd': -3.5, 'ww': -0.58, 'hh': 2.36,  'mf': 3.01,  'tt':
        -1.84},
    'E': {'molecular_weight': 147, 'kd': -3.5, 'ww': -2.02, 'hh': 2.68,  'mf': 1.64,  'tt':
        -2.90},
    'G': {'molecular_weight': 75,  'kd': -0.4, 'ww': -0.01, 'hh': 0.74,  'mf': 1.72,  'tt':
        -0.19},
    'H': {'molecular_weight': 155, 'kd': -3.2, 'ww': -0.96, 'hh': 2.06,  'mf': 4.76,  'tt':
        -1.44},
    'I': {'molecular_weight': 131, 'kd': 4.5,  'ww': 0.31,  'hh': -0.60, 'mf': -1.56, 'tt':
        1.97},
    'L': {'molecular_weight': 131, 'kd': 3.8,  'ww': 0.56,  'hh': -0.55, 'mf': -1.81, 'tt':
        1.82},
    'K': {'molecular_weight': 146, 'kd': -3.9, 'ww': -0.99, 'hh': 2.71,  'mf': 5.39,  'tt':
        -3.46},
    'M': {'molecular_weight': 149, 'kd': 1.9,  'ww': 0.23,  'hh': -0.10, 'mf': -0.76, 'tt':
        1.40},
    'F': {'molecular_weight': 165, 'kd': 2.8,  'ww': 1.13,  'hh': -0.32, 'mf': -2.20, 'tt':
        1.98},
    'P': {'molecular_weight': 115, 'kd': -1.6, 'ww': -0.45, 'hh': 2.23,  'mf': -1.52, 'tt':
        -1.44},
    'S': {'molecular_weight': 105, 'kd': -0.8, 'ww': -0.13, 'hh': 0.84,  'mf': 1.83,  'tt':
        -0.53},
    'T': {'molecular_weight': 119, 'kd': -0.7, 'ww': -0.14, 'hh': 0.52,  'mf': 1.78,  'tt':
        -0.32},
    'W': {'molecular_weight': 204, 'kd': -0.9, 'ww': 1.85,  'hh': 0.30,  'mf': -0.38, 'tt':
        1.53},
    'Y': {'molecular_weight': 181, 'kd': -1.3, 'ww': 0.94,  'hh': 0.68,  'mf': -1.09, 'tt':
        0.49},
    'V': {'molecular_weight': 117, 'kd': 4.2,  'ww': -0.07, 'hh': -0.31, 'mf': -0.78, 'tt':
        1.46},
    'spacer': {'molecular_weight': 0,  'kd': 0,   'ww': 0,    'hh': 0,    'mf': 0,    'tt': 0}
}

def encode_properties(properties, scale):
    new_encoding = {}
    for aa, props in properties.items():
        weight = props['molecular_weight']
        hydro = props[scale]
        normalized_weight = (weight - 75) / (204 - 75)
        normalized_hydro = (hydro + 4.5) / (9)
        new_encoding[aa] = [normalized_weight, normalized_hydro]
    return new_encoding

scales = ['kd', 'ww', 'hh', 'mf', 'tt']
for scale in scales:
    encoded_data = encode_properties(amino_acid_properties, scale)
```

```
    file_name = f'amino_acid_properties_{scale}.json'
    with open(file_name, 'w') as json_file:
        json.dump(encoded_data, json_file, indent=4)
```

8.3.2. Encoding Amino Acids.

```
    import numpy as np
import json


with open('../Neural_Network_Architecture/amino_acid_properties_hh.json', 'r') as f:
    amino_acid_encoding = json.load(f)


for aa in amino_acid_encoding:
    amino_acid_encoding[aa] = np.array(amino_acid_encoding[aa])

amino_acids = 'ACDEFGHIKLMNPQRSTVWY'
one_hot_to_amino_acid = {i: aa for i, aa in enumerate(amino_acids)}
one_hot_to_amino_acid[20] = 'spacer'


def local_encoding(one_hot_windows, num_features):
    num_windows, input_dim = one_hot_windows.shape
    window_size = input_dim // 21
    local_encoded_windows = np.zeros((num_windows, window_size * num_features))

    for i, window in enumerate(one_hot_windows):
        for j in range(window_size):
            one_hot_vector = window[j * 21: (j + 1) * 21]
            amino_acid_index = np.argmax(one_hot_vector)
            amino_acid = one_hot_to_amino_acid[amino_acid_index]
            local_encoded_vector = amino_acid_encoding[amino_acid]
            local_encoded_windows[i, j * num_features: (j + 1) * num_features] =
                local_encoded_vector

    return local_encoded_windows
```

8.4. Model Architecture

```
import numpy as np
import json
import tensorflow as tf
from tensorflow.keras.models import Sequential
from Ning_Qian.Pre_Processing.More_Preprocessing import train_window_labels, train_windows,
    test_window_labels, test_windows
from Ning_Qian.Pre_Processing.Local_Encoding import local_encoding
from Ning_Qian.Pre_Processing.Weight_Sharing import WeightSharingLayer
from tensorflow.keras.initializers import RandomUniform
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt

num_features = 2

train_windows_local_encoded = local_encoding(train_windows, num_features)
test_windows_local_encoded = local_encoding(test_windows, num_features)

print("Shape of train_windows_local_encoded:", train_windows_local_encoded.shape)
print("Shape of test_windows_local_encoded:", test_windows_local_encoded.shape)

print("First few encoded train windows:", train_windows_local_encoded[:5])
print("First few encoded test windows:", test_windows_local_encoded[:5])

input_dim = train_windows_local_encoded.shape[1]
units = 68

model = Sequential([
    Input(shape=(input_dim,)),
```

```
    WeightSharingLayer(units=units, num_features=num_features),
    Dense(40, activation='sigmoid', kernel_initializer=RandomUniform(minval=-0.3, maxval=0.3)),
    Dense(3, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model.summary()

early_stopping = EarlyStopping(
)

history = model.fit(
    train_windows_local_encoded, train_window_labels,
    epochs=200,
    batch_size=32,
    validation_data=(test_windows_local_encoded, test_window_labels),
    verbose=2,
    callbacks=[early_stopping]
)

print("Training history:", history.history)

test_loss, test_accuracy = model.evaluate(test_windows_local_encoded, test_window_labels)
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")

best_accuracy = 0
best_model_history = None

if test_accuracy > best_accuracy:
    best_accuracy = test_accuracy
    best_model_history = history

    plt.figure(figsize=(8, 6))
    plt.plot(best_model_history.history['accuracy'], label='Train Accuracy', marker='o',
        linestyle='-')
    plt.plot(best_model_history.history['val_accuracy'], label='Test Accuracy', marker='v',
        linestyle='--')
    plt.title('Graph representing the accuracy at each epoch over 10 runs')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()
    plt.grid(True)
    plt.show()
```

8.5. BRNN

8.5.1. Model Architecture.

```
import numpy as np
import time
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Bidirectional, Input
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.initializers import RandomUniform
from Ning_Qian.Pre_Processing.More_Preprocessing import train_window_labels, train_windows,
    test_window_labels, test_windows

num_amino_acids = 21
window_size = 13
input_dim = window_size * num_amino_acids

train_windows = train_windows.reshape(-1, window_size, num_amino_acids)
test_windows = test_windows.reshape(-1, window_size, num_amino_acids)

model = Sequential([
    Bidirectional(LSTM(256, return_sequences=True), input_shape=(window_size, num_amino_acids)),
    Bidirectional(LSTM(256)),
    Dense(1024, activation='sigmoid'),
    Dense(512, activation='sigmoid'),
    Dense(3, activation='softmax')
```

```
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

early_stopping = EarlyStopping(
)

model.summary()

model.fit(
    train_windows, train_window_labels,
    epochs=200,
    batch_size=32,
    validation_data=(test_windows, test_window_labels),
    verbose=1,
    callbacks=[early_stopping]
)

test_loss, test_accuracy = model.evaluate(test_windows, test_window_labels)
print(f"Test Accuracy: {test_accuracy:.4f}, Test Loss: {test_loss:.4f}")
```