

Diploma Project

**Solving the problem of Protein Secondary Structure Prediction
using Machine Learning Techniques**

Stefanos Englezou



University of Cyprus
**Department of Computer
Science**

Department of Computer Science

May 2026

UNIVERSITY OF CYPRUS
Faculty of Pure and Applied Sciences
Department of Computer Science

**Solving the problem of Protein Secondary Structure Prediction
using Machine Learning Techniques**

Stefanos Englezou

Supervisor:

Dr. Chris Christodoulou

The Thesis was submitted in partial fulfilment of the requirements for obtaining the Computer Science degree of the Department of Computer Science of the University of Cyprus

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (<https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/>) and with full accountability on my part for the accuracy, integrity and validity of all content.

Acknowledgements

Firstly, I want to thank my family and friends for their continuous support, encouragement and patience throughout this journey. Their understanding and belief in me provided the energy and motivation needed to overcome the challenges and complete this dissertation.

I would especially like to thank my supervisor, Dr. Chris Christodoulou, for proposing the subject of this dissertation which gave me the opportunity to explore a new field of interest. His continuous feedback and guidance throughout this year were invaluable.

I would also like to express my appreciation to Dr. Vasilis Promponas for his valuable ideas, scientific input and the many productive discussions throughout this year. The ongoing exchange of feedback with both Dr. Christodoulou and Dr. Promponas played a central role in helping me understand the objectives of this thesis. Their experience and insights were important in strengthening the quality of my research.

In addition, I would like to thank Dr. Michalis Agathokleous for his helpful suggestions, comments and continued support during this work. His feedback and recommendations were greatly appreciated.

Finally, I would like to acknowledge everyone who, directly or indirectly, contributed to this thesis and supported me throughout my studies. Their encouragement and assistance are sincerely appreciated.

Abstract

Protein Secondary Structure Prediction (PSSP) is a fundamental problem in computational biology, providing a critical stepping stone toward understanding full three-dimensional protein conformation and enabling downstream applications in drug design and disease research. While recent deep learning methods have demonstrated strong performance, many rely on computationally expensive Multiple Sequence Alignment (MSA) generation, which is time-consuming, database-dependent and ineffective for proteins with few known homologs.

This thesis investigates an alignment-free approach to PSSP using frozen per-residue embeddings from the Evolutionary Scale Modelling 2 (ESM-2), which is a Protein Language Model (PLM) as input to an encoder-only Transformer architecture. Rather than treating each 1280-dimensional embedding as a single vector, the model partitions embeddings into non-overlapping token slices, enabling self-attention to capture intra-embedding relationships. Sinusoidal positional embeddings are applied to each slice, serving both as positional encodings and as an implicit regulariser against overfitting to the strong ESM-2 signal. A seed ensemble of three identically structured models with different weight initialisations is combined with a two-stage post-processing pipeline consisting of a Random Forest classifier operating on sliding windows of prediction probabilities, followed by deterministic rules that remove structurally implausible short segments.

The proposed system is evaluated on four benchmark datasets in both 3-state and 8-state classification. On CB513, the base ensemble achieves 86.80% Q3 and 75.10% Q8, outperforming NetSurfP-2.0 by over 2 percentage points and matching AttSec within 0.30 pp Q3 despite AttSec using the substantially larger PLM model. On CASP12 Free Modelling targets, the model achieves 81.40% Q3, exceeding all evaluated baselines including AttSec (80.80%) and NetSurfP-2.0 (81.00%). Post-processing consistently improves segment-level

quality, producing SOV gains of up to 10 points on CASP12 and 4 points on TS115 with negligible impact on residue-level accuracy. In 8-state classification, the largest gains occur for minority structural classes including π -helices (I), 3_{10} -helices (G) and bends (S), which are typically underrepresented in standard Q8 reporting.

Hyperparameter analysis of the post-processing stage shows that probability-based remains stable across window sizes, while hard-label inputs degrade in 8-state classification at smaller windows. Additional analysis demonstrates that the Transformer encoder learns structured, head-specialised attention patterns and produces a substantially more class-separable feature space than the original ESM-2 embeddings.

Implementation analysis confirmed that the Transformer encoder learns structured, head-specialised attention patterns and produces a class-separable feature space that is substantially more structured than the raw ESM-2 representations, demonstrating that the encoder contributes meaningfully beyond feature passthrough.

These results establish that high-quality PSSP is achievable from frozen PLM embeddings alone, without any evolutionary profile computation and that the proposed architecture is competitive with or superior to current state-of-the-art MSA-based methods across all evaluated benchmarks.

Table of Contents

Acknowledgements.....	3
Abstract.....	4
Table of Contents.....	6
List of Figures.....	9
List of Tables	11
List of Abbreviations	13
1 Introduction.....	16
1.1 Problem Statement.....	16
1.2 Problem Importance	17
1.3 Thesis Overview	19
2 Background.....	20
2.1 Biological Background.....	20
2.1.1 Amino Acids	20
2.1.2 Protein Levels of Organization	23
2.1.3 Homologous Proteins	28
2.1.4 Dictionary of Secondary Structure of Proteins.....	28
2.2 PSSP Problem.....	29
2.2.1 Experimental Methods	29
2.2.2 Computational History	30
2.2.3 Multiple Sequence Alignment and Evolutionary Profiles.....	31
2.2.4 Post-AlphaFold Era and Limitations.....	33
2.3 Datasets and Evaluation Metrics	35
2.3.1 Datasets	35
2.3.2 Evaluation Metrics	37
2.4 Neural Network Foundations.....	39
2.4.1 Origins of Artificial Neural Networks	39
2.4.2 McCulloch-Pitts and Perceptron	40
2.4.3 Multi-Layer Perceptron.....	42
2.4.4 Gradient Descent and Backpropagation.....	44
2.4.5 Attention and Transformers	47
2.4.6 ESM-2 Protein Language Model.....	50
2.4.7 Training Techniques.....	52
2.4.8 Ensemble Methods	56
2.4.9 Decision Tree and Random Forest.....	57

3	Related Work	60
3.1	Direct Comparison Methods.....	60
3.1.1	NetSurfP-2.0.....	60
3.1.2	NetSurfP-3.0.....	60
3.1.3	AttSec.....	61
3.2	Acknowledged State-of-the-Art Methods	62
3.2.1	SPOT-1D-LM	62
3.2.2	Porter 6.....	63
3.3	Post-Processing Lineage.....	63
3.4	Other Transformer-based PSSP Methods.....	65
4	Methodology	66
4.1	High Level Overview	66
4.2	Dataset Preparation.....	66
4.3	Input Representation.....	68
4.4	Base Model Architecture	70
4.5	Filtering Methods	74
4.5.1	Random Forest Post-Processor.....	74
4.5.2	External Rules	76
4.5.3	Combined Pipeline	77
4.6	Training Procedure	78
4.7	Hyperparameter Optimization	79
4.7.1	Tree-structured Parzen Estimator Sampler.....	80
4.7.2	Median Pruner	80
5	Experiments and Results.....	81
5.1	Hyperparameter Tuning Results.....	81
5.2	Results in 3-State.....	82
5.2.1	Cross-Validation Results.....	82
5.2.2	Cross-Validation Results with Post-Processing	85
5.2.3	Training Curves.....	87
5.2.4	Evaluation Results in 3-State	90
5.2.5	Window Size Experiment in 3-State	96
5.3	Results in 8-State.....	99
5.3.1	Cross-Validation Results.....	99
5.3.2	Cross-Validation Results with Post-Processing Filtering	102
5.3.3	Training Curves.....	105
5.3.4	Evaluation Results in 8-State	107
5.3.5	Window Size Experiment in 8-State	114

5.4	Implementation Analysis.....	117
5.4.1	Attention Analysis.....	117
5.4.2	Mean-Pooled Vectors.....	120
5.5	Comparison.....	123
6	Conclusions and Future Work.....	126
6.1	Conclusions and Takeaways.....	126
6.2	Limitations.....	130
6.3	Future Work.....	131
7	References.....	134
	Appendices.....	144
	Appendix I — Mapping Module.....	144
	Appendix II — ESM-2 Embedding Extraction.....	145
	Appendix III — Transformer Encoder Implementation	149
	Appendix IV — External Rules Module.....	165
	Appendix V — Hyperparameter Tuning Script.....	166
	Appendix VI — Train Utilities Module.....	170
	Appendix VII — Extraction of Predictions Module	176
	Appendix VIII — Cross-Validation Pipeline.....	182
	Appendix IX — Training and Evaluation Pipeline.....	189
	Appendix X — Configuration	199

List of Figures

Figure 2.1. Structure of an amino acid, showing the amino group, α -carbon, side chain (R group).....	20
Figure 2.2. The twenty-one standard amino acids classified by side chain properties. From [26], CC0 1.0.....	21
Figure 2.3. Formation of a peptide bond between two amino acids through a condensation reaction, with the resulting C–N bond highlighted. From [25], CC BY-SA 4.0.....	22
Figure 2.4. The amide or peptide bond planes are joined by the tetrahedral bonds of the α -carbon. From [30], CC BY 3.0.....	24
Figure 2.5. Ramachandran plot and areas of secondary structures. From [31], CC BY-SA 3.0.	25
Figure 2.6. Alpha-Helix and Beta-Pleated Sheet secondary structures. From [33], CC BY 4.0.	26
Figure 2.7. Levels of Protein Organization. From [35], CC BY 4.0.....	27
Figure 2.8. Anatomy of Biological Neuron. From [90], CC BY-SA 4.0.	40
Figure 2.9. The Perceptron. From [93], CC BY-SA 3.0.	41
Figure 2.10. A Multi-Layer Perceptron with two active layers (one hidden and one output layer).	43
Figure 2.11. Example of attention weights of Layer 1/ Head 5	47
Figure 2.12. Decision Tree and Random Forest illustration. From [122], CC BY-SA 4.0	59
Figure 4.1. Proposed Architecture for PSSP	73
Figure 5.1. Training and Validation Q3 accuracies of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.	87
Figure 5.2. Training and Validation Cross-Entropy losses on 3-State of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.	88
Figure 5.3. Training and Validation Q3 accuracies of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the PISCES dataset.	88

Figure 5.4. Training and Validation Cross-Entropy losses on 3-State of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the PISCES dataset.	89
Figure 5.5. Confusion Matrix for 3-State Classification in CASP13 after applying RF→ER ordering.	94
Figure 5.6. Confusion Matrix for 3-State Classification in CB513 after applying RF→ER ordering.	94
Figure 5.7. Confusion Matrix for 3-State Classification in TS115 after applying RF→ER ordering.	95
Figure 5.8. Confusion Matrix for 3-State Classification in CASP12 after applying RF→ER ordering.	95
Figure 5.9. Average Q3 across all benchmarks (CASP12, TS115, CB513 and CASP13), testing multiple window sizes.	97
Figure 5.10. Average SOV'99 across all benchmarks (CASP12, TS115, CB513 and CASP13), testing multiple window sizes.	97
Figure 5.11. Training and Validation Q8 accuracies of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.	106
Figure 5.12. Training and Validation Cross-Entropy losses on 8-State of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.	106
Figure 5.13. Confusion Matrix for 8-State Classification in CB513 after applying RF.	110
Figure 5.14. Confusion Matrix for 8-State Classification in TS115 after applying RF.	112
Figure 5.15. Confusion Matrix for 8-State Classification in CASP12 after applying RF.	113
Figure 5.16. Average Q8 across all benchmarks (CASP12, TS115 and CB513), testing multiple window sizes.	115
Figure 5.17. Average SOV across all benchmarks (CASP12, TS115 and CB513), testing multiple window sizes.	115
Figure 5.18. Randomly initialized attention weights throughout all the encoder layers and heads (3-State Classification).	119
Figure 5.19. Trained attention weights throughout all the encoder layers and heads (3-State Classification).	119
Figure 5.20. PCA of Mean-Pooled vectors in 3-State Classification in comparison to a randomly initialized transformer.	122

Figure 5.21. PCA of Mean-Pooled vectors in 8-State Classification in comparison to a randomly initialized transformer.....	122
--	-----

List of Tables

Table 2.1. All eight secondary structure types.	29
Table 2.2. Smooth activation functions used in ANNs.	43
Table 5.1. The selected hyperparameters for the base model in both 3-state and 8-state tasks.	81
Table 5.2. 5-fold Cross-Validation Results in PISCES dataset (3-State Classification), ...	83
Table 5.3. 7-fold Cross-Validation Results in NetSurfP dataset (3-State Classification), ...	84
Table 5.4. 5-fold Cross-Validation Results in PISCES dataset (3-State Classification), using an ensemble of	85
Table 5.5. 7-fold Cross-Validation Results in NetSurfP dataset (3-State Classification), using an ensemble of	86
Table 5.6. Evaluation Results in CASP13 after combining multiple filtering methods	91
Table 5.7. Evaluation Results in CB513 after combining multiple filtering methods	91
Table 5.8. Evaluation Results in TS115 after combining multiple filtering methods	91
Table 5.9. Evaluation Results in CASP12 after combining multiple filtering methods	92
Table 5.10. 7-fold Cross-Validation Results on Q8, SOV'99 and SOV _{refine} in NetSurfP dataset	99
Table 5.11. Detailed 7-fold Cross-Validation Results on SOV'99 for each state in NetSurfP dataset	100
Table 5.12. Detailed 7-fold Cross-Validation Results on SOV _{refine} for each state in NetSurfP dataset	101
Table 5.13. 7-fold Cross-Validation Results on Q8, SOV'99 and SOV _{refine} in NetSurfP dataset (8-State Classification), ...	103
Table 5.14. Detailed 7-fold Cross-Validation Results on SOV'99 for each state in NetSurfP dataset (8-State Classification), ...	104
Table 5.15. Detailed 7-fold Cross-Validation Results on SOV _{refine} for each state in NetSurfP dataset (8-State Classification), ...	104

Table 5.16. Q8, SOV'99 and SOV _{refine} Results in CB513 after adding post-processing filtering	107
Table 5.17. Detailed SOV'99 Results in CB513 after adding post-processing filtering	107
Table 5.18. Detailed SOV _{refine} Results in CB513 after adding post-processing filtering	107
Table 5.19. Q8, SOV'99 and SOV _{refine} Results in TS115 after adding post-processing filtering	108
Table 5.20. Detailed SOV'99 Results in TS115 after adding post-processing filtering	108
Table 5.21. Detailed SOV _{refine} Results in TS115 after adding post-processing filtering	108
Table 5.22. Q8, SOV'99 and SOV _{refine} Results in CASP12 after adding post-processing filtering.....	109
Table 5.23. Detailed SOV'99 Results in CASP12 after adding post-processing filtering.....	109
Table 5.24. Detailed SOV _{refine} Results in CASP12 after adding post-processing filtering....	109
Table 5.25. Comparison with Direct and Transformer-based methods on independent tests of CB513, TS115 and CASP12.....	125

List of Abbreviations

Abbreviation	Definition
AdaGrad	Adaptive Gradient
Adam	Adaptive Moment Estimation
AdamW	Adaptive Moment Estimation with Decoupled Weight Decay
ANN	Artificial Neural Network
AUROC	Area Under the Receiver Operating Characteristic curve
BiGRU	Bidirectional Gated Recurrent Unit
BiLSTM	Bidirectional Long Short-Term Memory
BLAST	Basic Local Alignment Search Tool
BRNN	Bidirectional Recurrent Neural Network
CASP	Critical Assessment of protein Structure Prediction
CBRNN	Convolutional-Bidirectional Recurrent Neural Network
CNN	Convolutional Neural Network
Cryo-EM	Cryo-Electron Microscopy
DSSP	Dictionary of Secondary Structure of Proteins
ER	External Rules
ESM-1b	Evolutionary Scale Modelling 1b
ESM-2	Evolutionary Scale Modelling 2
FFN	Feed-Forward Network
FM	Free Modelling
GELU	Gaussian Error Linear Unit
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HFO	Hessian-Free Optimization

HMM	Hidden Markov Model
IDR	Intrinsically Disordered Region
LCE	Categorical Cross-Entropy Loss
LN	Layer Normalization
LSTM	Long Short-Term Memory
MHA	Multi-Head Attention
MLM	Masked Language Modelling
MLP	Multi-Layer Perceptron
MSA	Multiple Sequence Alignment
MSE	Mean Squared Error
NMR	Nuclear Magnetic Resonance
PCA	Principal Component Analysis
PDB	Protein Data Bank
PLM	Protein Language Model
Post-LN	Post-Layer Normalization
Pre-LN	Pre-Layer Normalization
ProtT5	Protein Text-to-Text Transfer Transformer
PSA	Pairwise Sequence Alignment
PSI-BLAST	Position-Specific Iterated BLAST
PSSM	Position-Specific Scoring Matrix
PSSP	Protein Secondary Structure Prediction
Q	Per-Residue Accuracy
Q3	Three-State Per-Residue Accuracy
Q8	Eight-State Per-Residue Accuracy
RBF	Radial Basis Function
ReLU	Rectified Linear Unit

RF	Random Forest
RMSProp	Root Mean Square Propagation
RNN	Recurrent Neural Network
RoPE	Rotary Positional Embeddings
SGD	Stochastic Gradient Descent
SGDR	Stochastic Gradient Descent with Warm Restarts
SOV	Segment Overlap Score
SOV'99	Segment Overlap, 1999 revision
SOVrefine	Refined Segment Overlap
SVM	Support Vector Machine
TBM	Template-Based Modelling
TPE	Tree-structured Parzen Estimator
ViT	Vision Transformer

1 Introduction

1.1 Problem Statement

Proteins are complex biological macromolecules composed of subunits called amino acids. Their functions include building and repairing muscles and other tissues, transporting molecules, sending signals, supporting the immune system and many more [1, pp. 357–359, 3483], [2]. Proteins are assembled according to instructions encoded in deoxyribonucleic acid, specifically within regions known as protein-coding genes. In humans, there are approximately 20,000 protein-coding genes [1, pp. 416–417]. Alternative splicing and other processes generate protein variants, or isoforms, each with a slightly different functionality [3, p. 1076]. With an enormous number of human cells in the body [4] and with scientists estimating up to 10 trillion proteins in each human cell [5], it is apparent that proteins play a crucial role in all biological processes.

The final three-dimensional structure of a protein (referred to as its conformation), can be described at four levels of organization: primary, secondary, tertiary and quaternary structure [2], [3, pp. 106–107]. The primary structure is the linear sequence of amino acids, the secondary structure captures local folding patterns such as α -helices and β -pleated sheets, the tertiary structure is the full three-dimensional fold and the quaternary structure describes assemblies of multiple polypeptide chains [2]. It is this tertiary structure that ultimately determines a protein's biological function, as its three-dimensional shape governs binding sites, interactions and activity [6].

The first advancements of studying protein structure began in 1951, when Pauling and Corey predicted helical and sheet conformations for the protein polypeptide backbone, even before the first protein structure was determined [7], [8]. The first protein structure, myoglobin, was determined by X-ray crystallography in 1958, confirming Pauling's predicted secondary

structures existed [9]. By 1973, Christian B. Anfinsen had developed his thermodynamic hypothesis of protein folding, stating that, at least for a small globular protein in its standard physiological environment, the native structure is determined only by the protein's amino acid sequence [10], [11]. By this discovery, a central question emerged: “*Can we infer the three-dimensional structure of a protein from its amino acid sequence?*”, that is, the prediction of its secondary and tertiary structure from its primary structure?

Tertiary protein structures are resolved by experimental methods like X-ray crystallography. Nevertheless, these techniques have many disadvantages, namely their time-consuming and expensive nature [12], [13], [14], [15]. Experiment costs are typically in the range of \$100,000–\$300,000 per structure [15]. Despite the challenges, all successfully experimentally derived structures are stored in the Protein Data Bank (PDB) [16]. Even though the PDB contains a huge number of sequences, it covers only a fraction of all known proteins [17], [18], highlighting the critical need for computational methods that can accurately predict protein structures for any given sequence. To address tertiary structure prediction, the problem must first be approached through its precursor, namely Protein Secondary Structure Prediction (PSSP).

1.2 Problem Importance

Protein structure prediction is one of the most important objectives pursued by computational biology and directly addresses Levinthal's paradox [19]: the observation that finding the native folded state of a protein by random search among all possible conformations would take longer than the age of the universe, yet proteins fold in seconds or less. A small but physically reasonable energy bias against locally unfavourable states is sufficient to overcome this

paradox, revealing that folding follows a guided energy landscape rather than random exploration [20].

Understanding protein structure is crucial for medicine and drug design, as Structure-based Drug Design relies on three-dimensional structural data of protein targets [21]. Since such 3D structures are typically obtained from experimental techniques that are time-consuming and expensive, computational methods that can predict structural information have become essential for accelerating the drug discovery process. Furthermore, protein misfolding, aggregation and accumulation are common pathogenic mechanisms in neurodegenerative disorders. Alzheimer's disease is closely associated with abnormal forms of beta-amyloid (A β) and tau proteins, whereas Parkinson's disease is linked to misfolded α -synuclein protein. Misfolding of these proteins leads to the formation of toxic aggregates in specific brain regions and these pathological species can progressively spread to connected areas, contributing to widespread neurodegeneration [22], [23]. These examples illustrate why accurate prediction of protein structure and the secondary structure elements that govern misfolding propensity, remains a problem of direct biomedical relevance.

Taken together, these considerations highlight that Protein Secondary Structure Prediction is not merely a historical stepping stone toward tertiary-structure methods, but a practically important task in its own right: it supports the interpretation of folding mechanisms, enables structural and functional annotation when full 3D structures are unavailable and provides scalable structural information that underpins biomedical applications such as drug discovery and the study of protein misfolding in disease.

1.3 Thesis Overview

This thesis is organised into six chapters, guiding the reader from the biological and computational foundations of PSSP to the proposed methodology, experimental evaluation and final conclusions.

Chapter 2 provides the necessary background on protein structure, PSSP, evolutionary information, standard datasets and evaluation metrics and the neural network concepts underlying the proposed architecture, including attention mechanisms, Protein Language Models, ensemble methods and tree-based classifiers.

Chapter 3 reviews the most relevant related work, including state-of-the-art PSSP systems, Transformer-based approaches and previous post-processing frameworks relevant to this thesis.

Chapter 4 presents the proposed methodology, describing the overall two-stage ensemble architecture, dataset preparation, the use of frozen Protein Language Model embeddings, the Transformer base model, the post-processing pipeline, the training procedure and hyperparameter optimisation.

Chapter 5 presents the experimental evaluation, including cross-validation and benchmark results for both 3-state and 8-state prediction, post-processing analysis, implementation analysis of the learned representations and attention behaviour and comparison against reference methods.

Finally, Chapter 6 summarises the main findings of the thesis, discusses its limitations and outlines directions for future work.

2 Background

2.1 Biological Background

2.1.1 Amino Acids

Amino acids are the building blocks (or monomers) that compose a protein (or polymer). They are also referred to as residues. Each amino acid has a standard structure, which consists of a central carbon atom known as the alpha (α) carbon connected to a hydrogen atom, a carboxyl group, an amino group and the R group or side chain [24], [25]. An example of amino acid structure is shown in Figure 2.1.

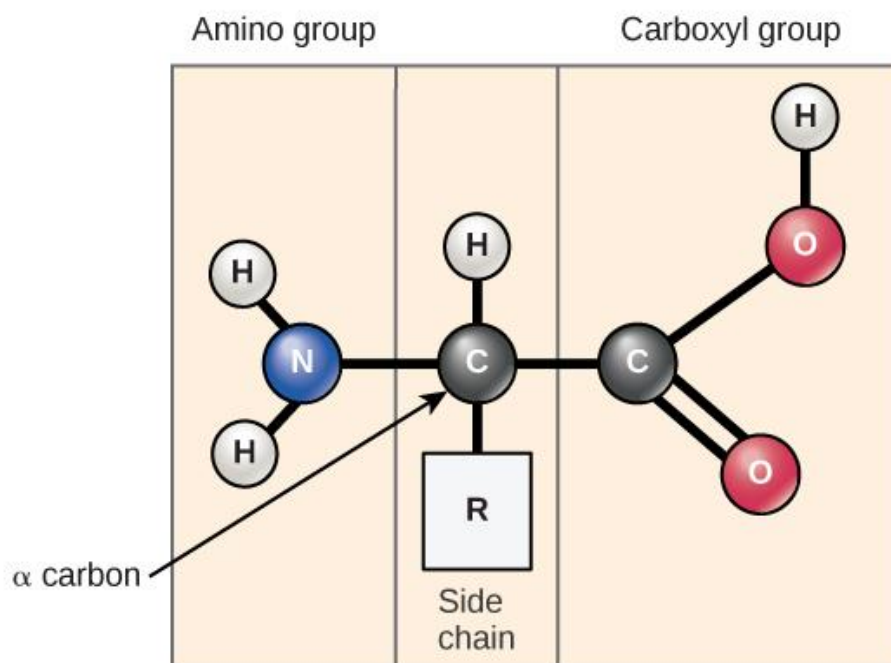


Figure 2.1. Structure of an amino acid, showing the amino group, α -carbon, side chain (R group) and carboxyl group. From [25], CC BY-SA 4.0.

There are 22 amino acids present in proteins, each with a unique side chain. Two of these are considered non-standard due to their rare occurrence in organisms: Selenocysteine (Sec, U) and Pyrrolysine (Pyl, O). Because of their rarity, Pyrrolysine is commonly excluded, leaving

21 amino acids typically recognized. Each amino acid's characteristics are based on the chemical properties of the side chain (R group). All amino acids can be grouped into categories based on chemical criteria: polarity/hydrophobicity and charge (Figure 2.2) [25].

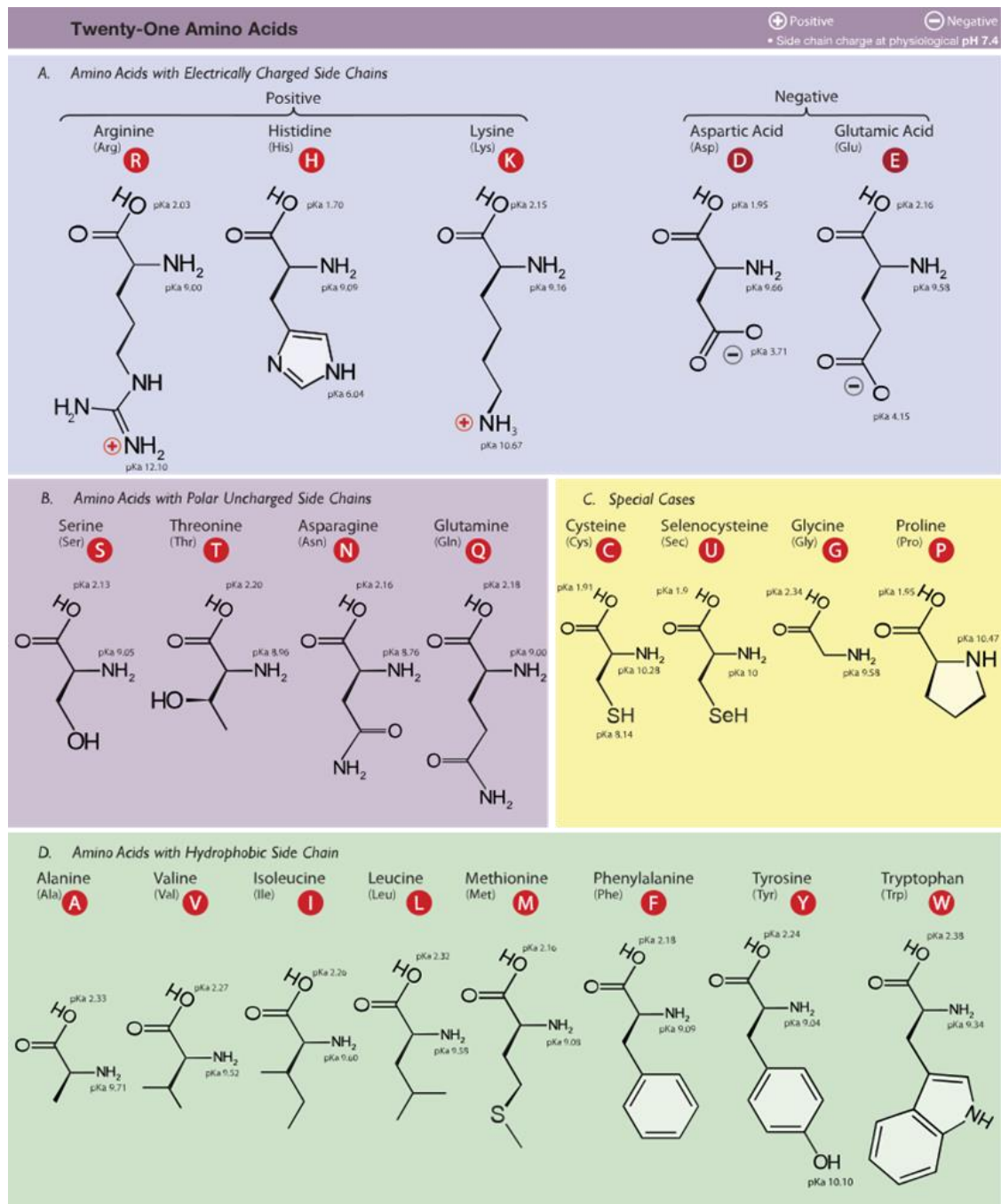


Figure 2.2. The twenty-one standard amino acids classified by side chain properties. From [26], CC0 1.0.

The sequence and number of amino acids influence the shape, size and function of the protein. Amino acids are linked together by peptide bonds (Figure 2.3), which involve the connection of one carboxyl group of one amino acid with the amino group of another. This repeating process resulting in a chain of connected amino acids is called a polypeptide. It is important to note that proteins and polypeptides are not the same; in fact, proteins are assembled by one or more polypeptides. In a polypeptide, the amino acid residue at the end with the free amino group is called the amino-terminal (N-terminal) and on the other end with the free carboxyl group is called the carboxyl-terminal (C-terminal) [1, pp. 379–381], [25].

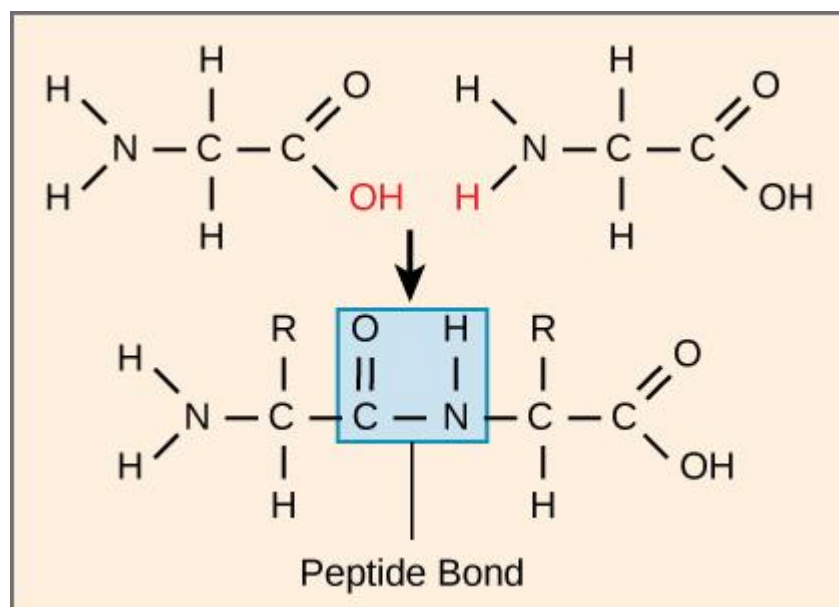


Figure 2.3. Formation of a peptide bond between two amino acids through a condensation reaction, with the resulting C–N bond highlighted. From [25], CC BY-SA 4.0.

2.1.2 Protein Levels of Organization

The primary structure is defined as the linear sequence of amino acids which we described as being formed with peptide bonds. The specific amino acids and their position in the sequence determine the factors for which portions of the protein fold closely together to form the protein's final three-dimensional shape, which we refer to as folding structures [3, pp. 106–107]. It is important to note that two polypeptide chains with the same amino acid sequence can result in different forms [27]. Whereas the primary structure of a protein is determined by the covalently (share of electron pairs between atoms) linked residues in the polypeptide backbone, the rest of the structures depend on weaker noncovalent forces like hydrogen bonds, ionic, van der Waals and hydrophobic interactions [3, pp. 148–149].

The secondary structure is the first step in the folding process that a protein takes to assume its native structure. This folding is held together predominately with hydrogen bond formation within elements of the polypeptide backbone. The two common patterns observed in this structure are the alpha-helices and the beta-pleated sheets. Where these portions are not repetitive, they are called random coils [28], [29]. For PSSP, these 3 categories break into smaller ones, which are mentioned in Section 2.1.4.

The formation of hydrogen bonds stems from the peptide bond itself, where six atoms, the oxygen, carbon, nitrogen and hydrogen atoms of the peptide group, as well as the adjacent α -carbons, lie in a plane called amide or peptide bond plane. Each residue technically has two planes which rotate: one linking the α -carbon with the carbonyl carbon and the other linking the α -carbon with the amide nitrogen. Their rotation is measured by the dihedral angles psi (ψ) and phi (ϕ) respectively (Figure 2.4). Note that the positive values of ϕ and ψ correspond to clockwise rotation viewed from the α -carbon [3, p. 150].

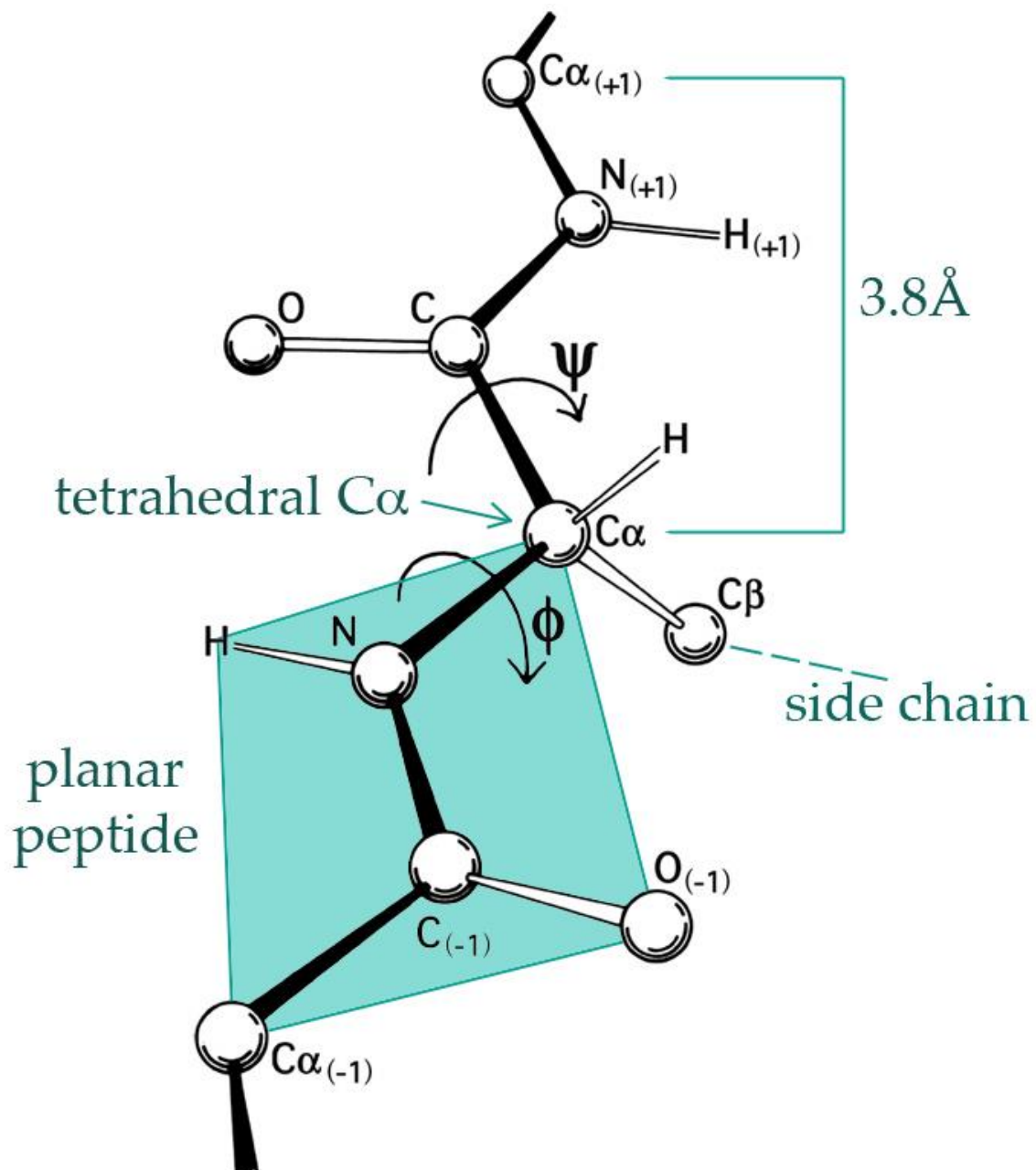


Figure 2.4. The amide or peptide bond planes are joined by the tetrahedral bonds of the α -carbon. From [30], CC BY 3.0.

Some combinations of ϕ and ψ angles are sterically forbidden because they produce unfavourable atomic overlaps. Valid pairs of these angles can be illustrated using a Ramachandran plot which shows clustered values that correspond to specific secondary structures [3, p. 151]. This plot is typically constructed by measuring these angles for multiple residue cases, creating a dense and accurate graphical representation (Figure 2.5).

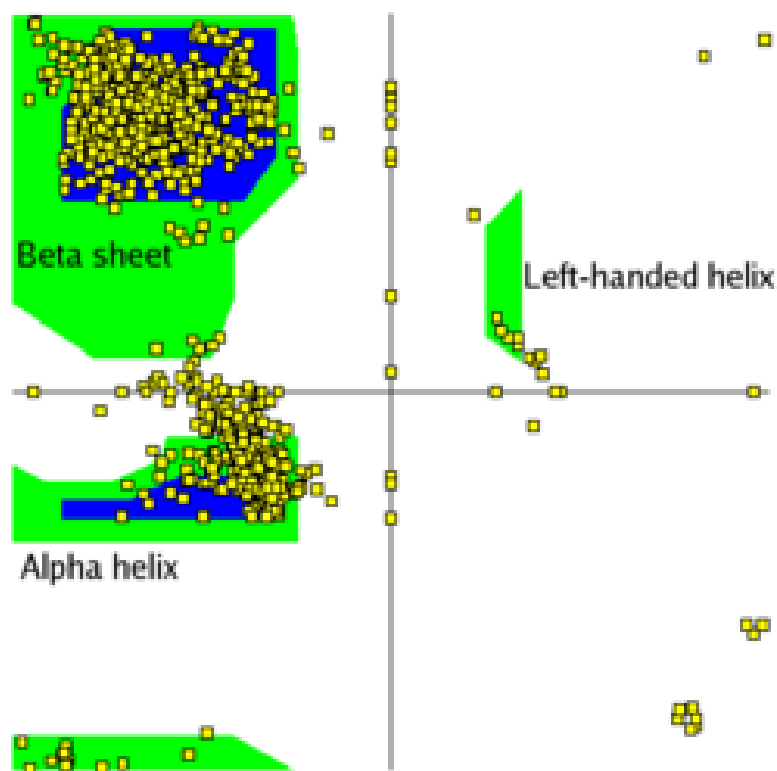


Figure 2.5. Ramachandran plot and areas of secondary structures. From [31], CC BY-SA 3.0.

In these allowed conformations, the carbonyl oxygen (δ^-) of one peptide bond is positioned sufficiently close to the amide hydrogen (δ^+) of another to permit hydrogen bond formation through electrostatic attraction. The repetitive ϕ and ψ angle combinations therefore lead to regular patterns of hydrogen bonding, forming structural motifs such as α -helices, in which hydrogen bonds form within a single polypeptide chain and β -pleated sheets, as hydrogen bonds form between adjacent chain segments [1, pp. 498–504], [3, p. 154], [28].

The α -helix is a common secondary structure in which the polypeptide backbone coils into a right-handed spiral, with side chains projecting outward. Each turn contains approximately 3.6 amino acid residues and spans 5.4 Å (0.54 nm). The structure is stabilized by intrachain hydrogen bonds between the carbonyl oxygen of residue i and the amide hydrogen of residue $i + 4$ (Figure 2.6). This arrangement is favoured by ϕ angles around -60° and ψ angles between

Further folding of the secondary structure by non-covalent bonds eventually creates a three-dimensional structure known as the tertiary structure. This level of organization integrates secondary structural elements by primarily R group interactions, producing the fully folded and biologically functional state of the protein [34]. Tertiary structures can be represented by one or more protein domains, regions that fold independently into three-dimensional units [6]. Since tertiary structure is ultimately determined by the primary sequence, accurately predicting secondary structural elements (the intermediate level of organization) provides a critical stepping stone towards understanding and modelling full three-dimensional protein conformation [6] [34]. Finally, the quaternary structure describes the assembly of multiple polypeptide chains, termed subunits for the overall stabilization of the protein [34]. All the levels of organization of the protein mentioned are shown in Figure 2.7.

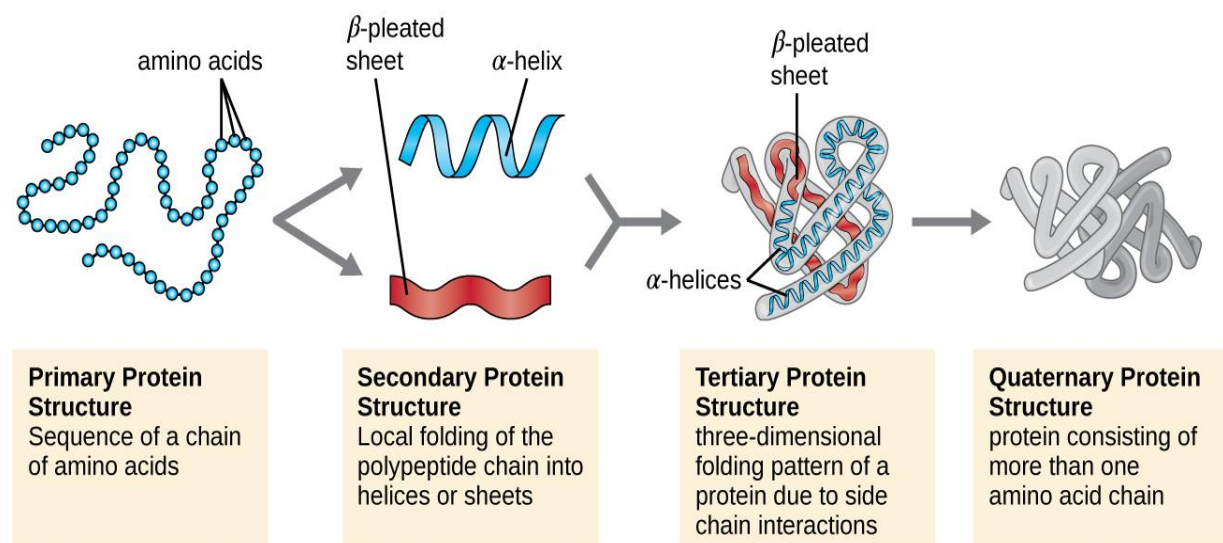


Figure 2.7. Levels of Protein Organization. From [35], CC BY 4.0.

2.1.3 Homologous Proteins

As seen later in Section 2.2, evolutionary information for the PSSP problem is vital. This evolutionary information specifically, is retrieved by finding protein sequence pairs that share a significant degree of similarity, structural resemblance or functionality and are defined as homologous. Homologous proteins can be further subdivided into orthologous and paralogous proteins whereas orthologous proteins can be from different species while paralogous proteins can be from a single species [3, p. 128].

2.1.4 Dictionary of Secondary Structure of Proteins

As mentioned in Chapter 1, this dissertation focuses particularly on the prediction of secondary structures using computational methods. The most common structures that can be observed are the α -helix, the β -pleated sheet and random coil. These classes, however, break down into 8 patterns that were observed experimentally. The Dictionary of Secondary Structure of Proteins (DSSP), developed by Kabsch and Sander [36], became the standardized algorithm for assigning secondary structure from experimentally derived 3D atomic coordinates. DSSP does not predict secondary structure, it assigns it from known structures. This standardization was crucial because it provided consistent training labels for all computational PSSP prediction methods. The secondary structure is assigned based on hydrogen bonding patterns such as those initially proposed by Pauling et al. in 1951 [7], [8]. The eight types of secondary structure that DSSP defines are an extension of the three types mentioned earlier. Specifically, the class of helices extends to three types: 3-turn helix (3_{10} -helix), the 4-turn helix (α -helix) and 5-turn helix (π -helix). The class of β -sheets spans into the extended strand in parallel and/or anti-parallel β -sheet conformation and the isolated β -bridge and finally the coil class extends to hydrogen bonded turn (3, 4 or 5 turn), bend and coil [36]. All the types are shown in the table below.

DSSP class	8-state Symbols	3-class Symbols
α -helix	H	H
3_{10} -helix	G	H
π -helix	I	H
strand- β	E	E
bridge- β	B	E
turn	T	C
bend	S	C
coil	C	C

Table 2.1. All eight secondary structure types.

2.2 PSSP Problem

2.2.1 Experimental Methods

Before the era of computational prediction, secondary structure of proteins was determined exclusively through experimental techniques such as X-ray crystallography, Nuclear Magnetic Resonance (NMR) spectroscopy and Cryo-Electron microscopy (Cryo-EM) [37]. Each technique provides information from which the three-dimensional atomic coordinates of a protein can be derived and then used by algorithms such as DSSP to assign the secondary structures [36]. These methods however share many limitations, as they are expensive [15] and slow in determining structures [12], [13], [14], with many experiments resulting in failures and inaccuracies [38], [39], [40], [41]. Despite these challenges, all successfully experimentally derived structures are stored in the Protein Data Bank [16]. However, since DSSP [36] assigns secondary structure labels purely by reading 3D atomic coordinates, any inaccuracies inherent in the experimental models propagate directly into erroneous ground truth labels. Furthermore, the PDB covers only a fraction of known protein sequences [17], [18], highlighting the critical need for computational PSSP methods.

2.2.2 Computational History

All the computational methods that were applied to that problem can be described in five generations as explained by Ismi et al. [42]. The first generation was purely based on statistical observations, namely the Chou-Fasman method [32] in which they used propensities of amino acid residues with heuristic rules [43] but these approaches resulted in accuracies less than 60% which was not satisfying [44]. The second generation methods like Garnier-Osguthorpe-Robson method [45] and the Lim method [46] used neighbourhood residue information along with physiochemical features [47] and more sophisticated techniques like graph theory [48], neural networks [49], [50], logic-based machine learning [51] and nearest-neighbour methods [52], [53] but still the overall accuracy of these methods was still less than 65% [44]. The third generation is characterized by the usage of homologous sequences and Multiple Sequence Alignment (MSA) profiles [54], with more advanced machine learning models like Support Vector Machines (SVMs) [55], [56] and neural-network based models which turned out to be the most performative [43]. A fourth generation emerged around the 2000s, utilizing auxiliary input features along evolutionary information but the accuracy improvements of these approaches were small and could not exceed the 80% [42]. Finally, the fifth generation with deep learning architectures specifically for PSSP emerged in 2010s-2020s [42] finally exceeding the 80% accuracy. Some of the most notable methods in that period was SPIDER2 [57], DeepCNF [58], eCRRNN [59] and NetSurfP-2.0 [60]. These methods approached the theoretical accuracy limit of 88-90%, constrained by inconsistencies in secondary structure assignment among structural homologs [43].

The field underwent a revolutionary transformation in 2020 with the introduction of AlphaFold2 by DeepMind, predicting protein tertiary structures with high atomic-level accuracy [42], [61]. AlphaFold2 combines two core components: the Evoformer, which processes MSA and pairwise residue features and a Structure Module that generates full 3D

coordinates directly from the primary sequence [62]. AlphaFold2 bypasses intermediary tasks such as secondary structure prediction entirely, predicting 3D structure directly [42].

In the post-AlphaFold era, PSSP research shifted toward two complementary directions: using MSA directly as model input and exploiting Protein Language Models (PLMs), such as Evolutionary Scale Modelling 1b (ESM-1b), Evolutionary Scale Modelling 2 (ESM-2) and Protein Text-to-Text Transfer Transformer (ProtT5) as alignment-free sequence representations [42], [61]. Unlike MSA-based features, PLMs encode structural and evolutionary context directly from raw sequences without computationally expensive database searches, making them faster and applicable to proteins with few homologs [61]. State-of-the-art models such as NetSurfP-3.0 [63], SPOT-1D-LM [64] and Porter 6 [65] demonstrated that PLM-based PSSP reaches accuracy comparable to or surpassing traditional profile-based methods [42], [61].

2.2.3 Multiple Sequence Alignment and Evolutionary Profiles

Sequence alignment analyses the biological characteristics shared between nucleotide or protein sequences by comparing their similarity, extracting functional, structural and evolutionary information [66]. When two sequences are compared, the process is termed Pairwise Sequence Alignment (PSA) where aligning three or more sequences simultaneously is called Multiple Sequence Alignment (MSA). Both can further be identified as global which span the full length of all sequences or local, where only the most similar sub-regions are identified [66]. Global MSA is of relevance to computational protein analysis, forming the core input to models such as AlphaFold2 [62] and serving as the primary source of evolutionary signal for the third and later generations of PSSP methods.

2.2.3.1 Position-Specific Scoring Matrix

A Position-Specific Scoring Matrix (PSSM) is a matrix of size $L \times 20$, where L is the sequence length and 20 corresponds to the standard amino acid alphabet. Each entry at position (i, j) represents a log-odds score quantifying the evolutionary preference of residue j at sequence position i , computed from substitution frequencies observed across an MSA relative to a background frequency [67]. When an amino acid is highly conserved at a given position, it receives a large positive score while all others receive large negative scores. At weakly conserved positions, all residues receive scores near zero.

2.2.3.2 Profile Hidden Markov Model

A Hidden Markov Model (HMM) is a probabilistic model built directly from an MSA that captures both residue composition and insertion–deletion patterns at every position [68]. For each column of the MSA, the model stores emission probabilities (the likelihood of observing each of the 20 amino acids) computed as the normalised residue frequency across aligned sequences. Beyond residue composition, the model encodes transition probabilities between match, insert and delete states, capturing how likely a sequence is to proceed normally, skip a position, or insert an extra residue between positions [68]. This formal probabilistic basis allows profile HMMs to assign position-specific gap penalties rather than a flat uniform penalty, making them a more expressive representation of a protein family than PSSMs alone.

2.2.3.3 Sequence Search Tools

Since the dataset used in this work contains HMM profiles generated through evolutionary sequence search, it is necessary to introduce the tools through which such representations are produced. Basic Local Alignment Search Tool (BLAST) is a heuristic database search program to rapidly identify local sequence similarities [67]. Position-Specific Iterated Basic Local

Alignment Search Tool (PSI-BLAST) extends BLAST by iteratively constructing a PSSM from the hits of each round and using it as the query in subsequent rounds, substantially increasing sensitivity for detecting remote homologs [67]. HHblits further improves sensitivity by converting both the query and database entries into profile HMMs and performing profile-to-profile comparison which makes it a more expressive search than profile-to-sequence alignment, enabling the detection of homologs well below the 30% sequence identity threshold [69]. MMseqs2 achieves comparable sensitivity through a double k-mer prefilter and vectorized Smith–Waterman alignment, operating orders of magnitude faster and scaling efficiently to databases of hundreds of millions of sequences [70]. However, both tools require iterative database searches over large repositories, making MSA generation computationally expensive and dependent on the availability of sufficient homologous sequences for rich representation, a dependency that motivates the alignment-free PLM-based approach adopted in this work.

2.2.4 Post-AlphaFold Era and Limitations

Despite AlphaFold2's revolutionary accuracy in tertiary structure prediction it does not render PSSP and related one-dimensional annotation tasks obsolete [71]. Several fundamental limitations keep the problem of secondary structure prediction not only relevant but actively researched in the post-AlphaFold era.

AlphaFold2's performance is critically contingent on the availability of deep MSAs, requiring a minimum of approximately 30 effective homologous sequences to achieve reliable predictions [61]. This reliance becomes a hard barrier for orphan proteins (sequences lacking known homologs) which are estimated to constitute around 20% of all metagenomic protein sequences and approximately 11% of eukaryotic and viral proteins [42]. In such low-homology regimes, the quality of the MSA degrades, directly reducing prediction accuracy and moreover,

the computational cost of searching protein databases as these increase, increases as well [42], [61]. Lightweight PSSP predictors that bypass MSA entirely, particularly PLM-based models such as those using ESM-2 or ProtT5 embeddings, therefore remain essential for rapid, scalable annotation of sequences where AlphaFold2 cannot be reliably applied [42], [61].

Another limitation concerns intrinsically disordered regions (IDRs) which are protein segments that lack a fixed three-dimensional structure under physiological conditions [61], [71]. These regions are estimated to be present in 30–50% of the human proteome and their structural flexibility underlies critical biological roles in molecular recognition, signalling and regulation [61], [71]. Since AlphaFold2 was trained primarily on experimentally resolved, well-ordered structures from the PDB, it tends to assign low-confidence scores to disordered regions without providing meaningful structural or functional annotations for them [71].

Beyond disordered regions, AlphaFold2 is fundamentally designed to predict a single static conformation per protein, as it was trained on X-ray crystallography and cryo-EM structures that capture proteins in one dominant state [71]. This is a critical limitation for special proteins that switch folds, allosteric proteins and kinases, whose biological function depends on transitioning between multiple conformational states [72], [73]. Evidence suggests that correct predictions in these cases reflect training-set memorization rather than true structural inference [72], [73]. Furthermore, AlphaFold2 cannot reliably predict the structural effects of point mutations, since it lacks a database of structurally disruptive mutations and makes predictions based on wild-type or homologous sequence patterns [71].

Taken together, these limitations establish a clear post-AlphaFold rationale for continued PSSP research. AlphaFold2 bypasses secondary structure prediction as an intermediary step, but this bypass fails for the significant fraction of proteins lacking evolutionary data and PSSP predictors that do not rely on evolutionary information remain crucial precisely in those cases

[42]. Furthermore, 1D structural annotations including secondary structure, solvent accessibility (water interaction) and IDR labelling serve as interpretable, fast and computationally inexpensive features that inform downstream functional annotation pipelines at scale; a role AlphaFold2, with its high computational cost and MSA requirements, cannot fill [42], [61].

2.3 Datasets and Evaluation Metrics

2.3.1 Datasets

Deep Neural Network based implementations requires both a training and test datasets. Typically, these datasets can be obtained either from the PDB or use the benchmark datasets previously created by other researchers [42].

Reliable training data is a critical in PSSP as the choice of dataset directly determines the learning capacity of the model [44]. The dominant method for collecting a dataset from the PDB is the PISCES culling server [74], which fetches non-redundant subsets of the PDB under configurable structural-quality criteria, typically applying a sequence identity cutoff of 25–30% [42]. Several pre-assembled training collections have achieved widespread reuse: CB6133 (6,133 protein chains from CullPDB, collected by [75], its de-duplicated version CullPDB5926 and the filtered variant CB6133-filtered, which removes chains overlapping with CB513 to ensure a clean train/test split [42], [76]. Larger curated sets include TR4590 [57] and TR9993 [77], the SPOT-1D dataset (10,029 proteins, assembled by [78]) and ProteinNet [79].

Independent benchmark test sets are the primary vehicle for comparing PSSP methods across publications [42]. CB513, one of the oldest and most widely used benchmarks, was developed by [76] and consists of 513 protein chains comprising 84,107 residues, all with pairwise

sequence identity below 25% [44]. RS126, developed by [80], contains 126 non-homologous protein sequences and has served independently as a benchmark in earlier literature [44]. TS115, introduced by [43], contains 115 protein chains released after January 2016, providing natural temporal separation from most training [42], [61]. Additional benchmarks include TEST2016 (1,213 proteins) and TEST2018 (250 proteins), both introduced by [78] which have since become community standards for evaluating recent deep learning models [42].

The Critical Assessment of Protein Structure Prediction (CASP) is a prestigious competition held every two years since 1994, where researchers and modelers are tasked with predicting the structure of previously unpublished, experimentally determined proteins [61]. Because CASP targets are drawn from real, unsolved structures, including sequences with low homology to known structures and multi-domain proteins, they represent genuinely challenging and realistic benchmarks, making them inherently harder than standard curated datasets and thus a stringent test for any prediction method [61]. Targets are divided into two main categories based on the availability of structural templates: Template-Based Modelling (TBM), for targets where one or more structural templates can be identified by sequence search and Free Modelling (FM), for targets with no sequence-detectable template, with a borderline category TBM/FM for intermediate cases [81], [82]. Further refinement into TBM-easy, TBM-hard, FM/TBM and FM sub-categories is determined by combining sequence-structure similarity to known folds with server prediction performance [83]. Target domains with good server predictions and close template homologs are classified as TBM, while those with poor server predictions, representing fast-evolving homologs, structural analogues, or novel folds, are classified as FM or FM/TBM [83]. Within this framework, CASP14 marked a landmark moment, as AlphaFold2 demonstrated revolutionary performance, significantly outperforming existing methods [62], [84].

2.3.2 Evaluation Metrics

For evaluating and comparing PSSP methods, this thesis uses two families of metrics: Per-Residue Accuracy (Q score) and Segment Overlap (SOV). The Q score is the simplest and most widely adopted measure, defined as the fraction of residues correctly predicted across all states [44]:

$$Q_m = 100 \cdot \frac{1}{N_{res}} \sum_{i=1}^m M_{ii}$$

where $m = 3$ yields the three-state per-residue accuracy (Q3) and $m = 8$ yields the eight-state per-residue accuracy (Q8) respectively, N_{res} is the total number of residues and M_{ii} is the number of correctly predicted residues in state i . The per-state accuracy is similarly defined as:

$$Q_i = 100 \cdot \frac{M_{ii}}{obs_i}$$

where obs_i is the number of residues observed in state i [44]. Despite its popularity, Q3 leads to distorted prediction quality e.g. assigning the entire myoglobin chain as a single helix, for instance, yields $Q3 \approx 78\%$, exceeding the accuracy of many real methods despite being structurally unrealistic [85].

To address this, the Segment Overlap score operates at the level of continuous secondary structure segments rather than individual residues, tolerating small boundary deviations while penalising errors in segment interiors [85]. While originally defined by Rost et al. [86] it was seen that the measure was difficult to compare with other scores in percentage terms. Zemla et al. addressed this problem where the modified metric SOV'99 is defined as:

$$SOV = 100 \times \frac{1}{\sum_i N(i)} \sum_i \sum_{s(i)} \frac{\min OV(s_1, s_2) + \delta(s_1, s_2)}{\max OV(s_1, s_2)} \times \text{len}(s_1)$$

Where $\min OV(s_1, s_2)$ is the actual overlap length, $\max OV(s_1, s_2)$ is the total extent of the segment pair and $\text{len}(s_1)$ is the reference segment length (ground truth). The allowance (bonus) $\delta(s_1, s_2)$ is defined as [85]:

$$\delta(s_1, s_2) = \min \left\{ \begin{array}{l} \max OV(s_1, s_2) - \min OV(s_1, s_2) \\ \min OV(s_1, s_2) \\ \text{int}(0.5 \times \text{len}(s_1)) \\ \text{int}(0.5 \times \text{len}(s_2)). \end{array} \right\}$$

However, SOV'99 exhibits a fundamental limitation: its allowance is not guaranteed to grow monotonically as predictions improve, producing irregular score increments even for strictly better predictions [87]. Refined Segment Overlap (SOV_{refine}) tackles this by redesigning the allowance as:

$$\delta(s_1, s_2) = \delta(\text{all}) \times \frac{\text{len}(s_1)}{\text{len}(s_r)} \times \frac{\min OV(s_1, s_2)}{\max OV(s_1, s_2)}$$

where δ_{all} is the total allowance assigned to the whole reference sequence, defined as [87]:

$$\delta(\text{all}) = \lambda \times \frac{N_C}{\sum_{j=1}^{N_S} \left(\frac{\text{len}(s_j)}{\text{len}(s_r)} \right)^2}$$

where N_C is the number of states, N_S is the number of segments in the reference sequence s_r and λ is an adjustable scale parameter set to 0.5 in all experiments. This updated allowance ensures incremental scores as predictions improve and achieves higher correlation with 3D

structural quality measures [87]. In all experiments, Q3/Q8, SOV'99 and SOV_{refine} with $\lambda = 0.5$ are reported jointly to provide both residue-level and segment-level evaluation.

2.4 Neural Network Foundations

2.4.1 Origins of Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational model inspired by biological neural networks, designed to model the way the brain performs specific tasks [88, pp. 1–10]. The motivation for mimicking the brain stems from the fact that it outperforms digital computers on certain tasks such as pattern recognition, perception and motor control, with huge parallel computing that no sequential machine can replicate [89, p. 1]. Although ANNs do not attempt to be biologically realistic in detail, they inherit the abstract structure of the biological neuron to learn from experience [89, p. 1].

The brain is composed of approximately 10^{11} neurons, each consisting of a cell body (soma), tree-like fibers called dendrites that receive incoming signals and a single long fiber called the axon that transmits signals outwards to other neurons [88, pp. 6–10], [89, pp. 1–2]. Signal transmission across a synapse is a chemical process in which transmitter substances raise or lower the electrical potential of the receiving cell. When this potential reaches a threshold, an action potential is fired down the axon, after which the cell must wait through a refractory period before it can fire again [89, p. 3]. The brain's ability to adapt is captured by the concept of plasticity where the nervous system continuously reorganises its synaptic connections in response to experience and it is precisely this property that ANNs abstract and implement through their learning algorithms [88, pp. 1–2]. The anatomy of a biological neuron is illustrated in Figure 2.8.

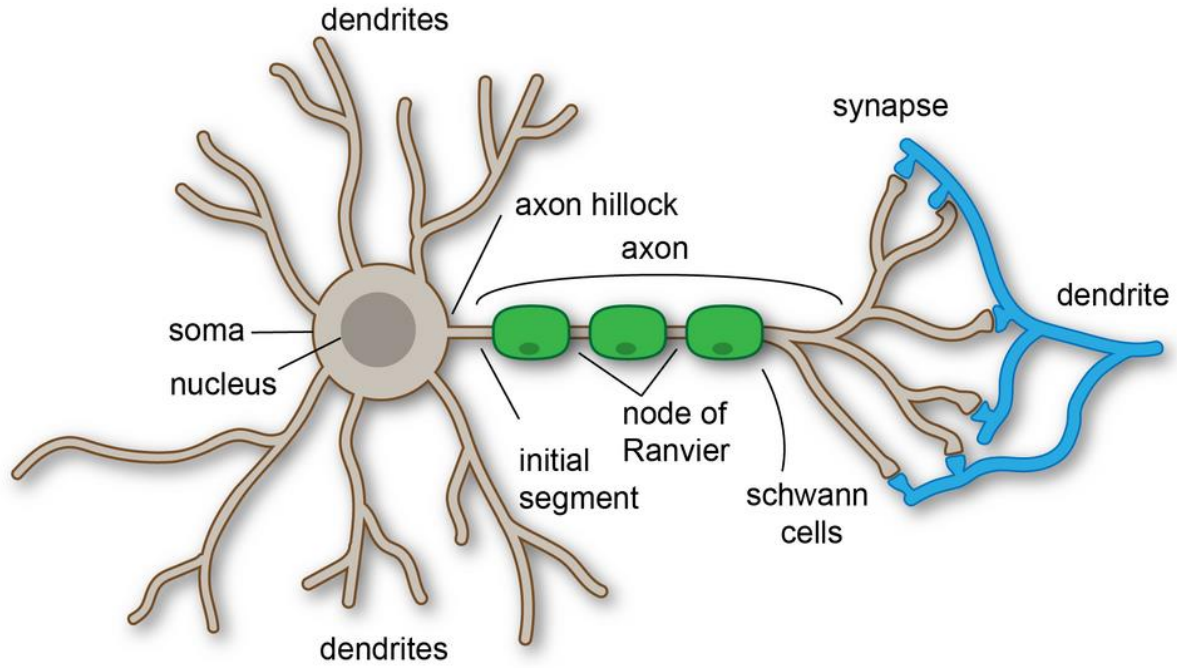


Figure 2.8. Anatomy of Biological Neuron. From [90], CC BY-SA 4.0.

2.4.2 McCulloch-Pitts and Perceptron

The first model to mimic the biological neuron was the McCulloch and Pitts model which acted as a binary threshold unit. That is, if the weighted sum of the inputs from each neuron j (n_j) at discrete time t exceeded a certain threshold μ_i the neuron fires: The output is 1, else is 0. The model uses the Heaviside or step function that decides whether the neuron fires or not [89, pp. 3–4], [91]. Specifically:

$$n_i(t + 1) = \Theta \left(\sum_j w_{ij} n_j(t) - \mu_i \right), \Theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

However, it is important to note that in this model, the weights must be set explicitly by hand. Thus, it did not include a learning procedure as biological neurons have in nature [89, pp. 3–

5]. The Rosenblatt Perceptron model [92] (Figure 2.9), built around the McCulloch-Pitts model, it addresses its problems by including continuous input and a learning algorithm (known as the perceptron learning algorithm).

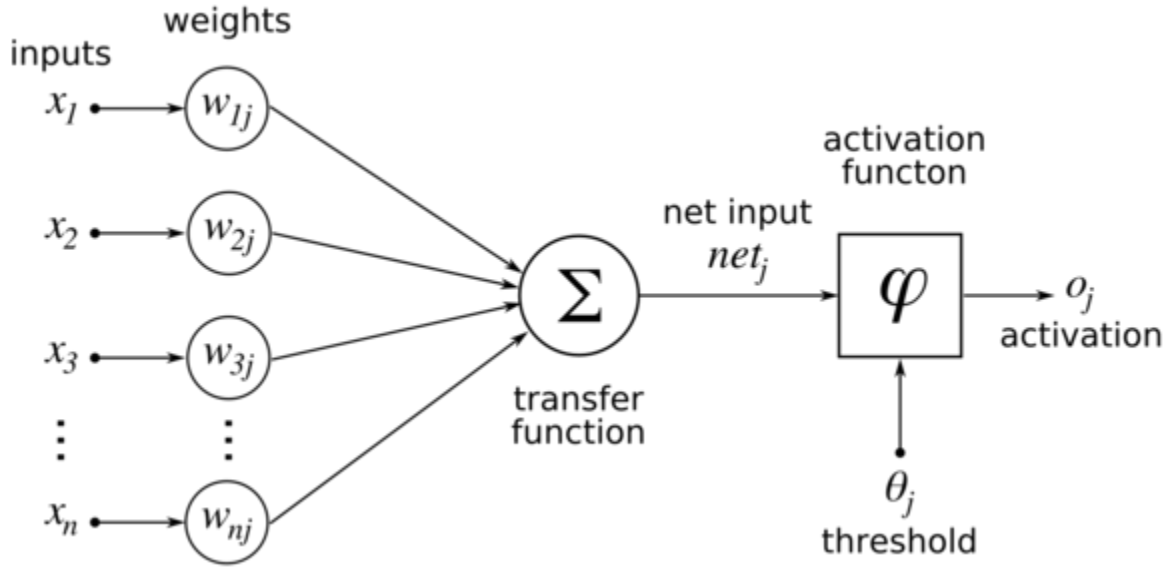


Figure 2.9. The Perceptron. From [93], CC BY-SA 3.0.

The perceptron learning algorithm operates iteratively over a set of labelled training examples to find a weight vector that correctly separates two linearly separable classes. The weights are initialised to zero and at each time step n the perceptron computes its output as:

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)], \text{sgn}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where $\mathbf{x}(n)$ is the input vector augmented with a bias term, $\mathbf{w}(n)$ is the weight vector and $\text{sgn}(x)$ is the signum function. If the response is incorrect, the weight vector is updated according to:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathcal{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathcal{C}_2 \end{cases}$$

where $d(n) \in \{+1, -1\}$ is the desired response and η is the learning rate. If the prediction is correct the weights remain unchanged and the process repeats until all training examples are classified correctly. The Perceptron Convergence Theorem guarantees that this procedure terminates in a finite number of steps provided the two classes are linearly separable [88, pp. 50–55]. However, a drawback of the Perceptron was that it could not work for non-linearly separable problems like the Exclusive OR (XOR) function proved by Minsky and Papert [94].

2.4.3 Multi-Layer Perceptron

This limitation was overcome with the induction of the Multi-Layer Perceptrons (MLPs) or Feed-Forward Networks (FFNs). These models are called feedforward cause information flows towards one direction without feedback connections (like Recurrent Neural Networks) [95, pp. 168–169].

Typically, the structure of the network consists of the input layer, one or more hidden layers and the output layer [95, pp. 168–169] (Figure 2.10). The input layer does not perform any computation as each neuron is simply one input feature and passes the value forward [95, p. 168]. The hidden layers are fully connected, meaning every neuron in one layer connects via a learned weight to every neuron in the next layer [88, pp. 21–23]. Each hidden neuron computes a weighted sum of its inputs, adds a bias term and passes the result through a non-linear activation function (e.g. Heaviside), which introduces the non-linearity necessary to learn complex patterns [95, pp. 172–173]. The function of the hidden neurons is that they act as feature detectors: the first hidden layer learns low-level features (e.g., simple boundaries), while deeper layers combine them into increasingly abstract representations [88, pp. 122–126].

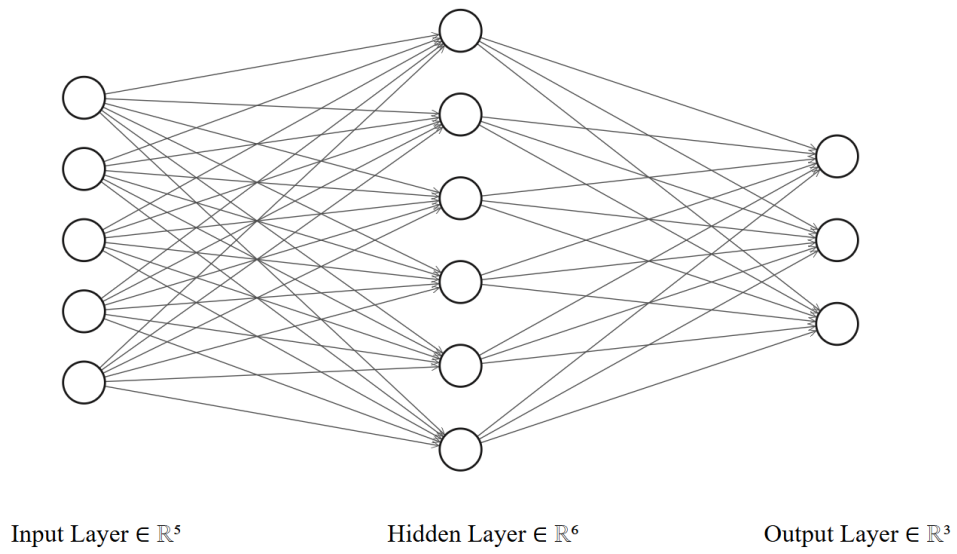


Figure 2.10. A Multi-Layer Perceptron with two active layers (one hidden and one output layer).

Activation Function	Formula
Sigmoid	$\sigma(z) = \frac{1}{1 + e^{-z}}$
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
ReLU	$f(z) = \max(0, z)$
GELU	$\text{GELU}(z) = \frac{z}{2} \left[1 + \text{erf} \left(\frac{z}{\sqrt{2}} \right) \right]$

Table 2.2. Smooth activation functions used in ANNs.

The theoretical justification for why MLPs can solve non-linearly separable problems is given by the Universal Approximation Theorem, which states that a feedforward network with a

linear output layer and at least one hidden layer with any non-linear activation function (as proven with the sigmoid activation) can approximate any non-linear function, provided sufficiently many hidden units are available [95, pp. 198–201], [96], [97].

As seen next, gradient descent learning rule assumes that activation functions are differentiable, but so far, we only mentioned the Heaviside and Signum functions which derivative is 0 everywhere and undefined for $x = 0$ and thus learning is impossible. The specific activation functions we are looking for are often called smooth activation functions [98]. These functions are continuous and differentiable (everywhere, or almost everywhere). Table 2.2 shows the most common activation functions.

2.4.4 Gradient Descent and Backpropagation

Gradient Descent or Steepest Descent is a weight adjustment rule with primary objective to approach the minimum point of a function whose value quantifies the discrepancy between the model's predictions and the ground truth labels. The function is referred to as loss or error function [88, Ch. 3]. The weight update rule is simply:

$$w_{ij}(t + 1) = w_{ij}(t) - \eta \frac{\partial E}{\partial w_{ij}}$$

where $w_{ij}(t)$ the weight value at time step t , η is the learning rate and $\frac{\partial E}{\partial w_{ij}}$ is the derivative of the loss function in respect to the weight w_{ij} (weight connection from neuron i to neuron j). The update rule above describes Stochastic Gradient Descent (SGD), in which $E = \ell_k$ is the loss computed on a single randomly sampled training example per update step. The mini-batch variant replaces this with the average loss over a random subset $\mathcal{B}$, i.e. $E = \frac{1}{B} \sum_{k \in \mathcal{B}} \ell_k$, which is the standard approach in modern deep learning as it enables Graphics Processing Unit (GPU) parallelisation while maintaining stable gradient estimates [99, Ch. 7].

In deep neural networks however, calculating that derivative isn't that straightforward. The backpropagation algorithm [100], provides an efficient method for computing the gradient of the loss function with respect to all network weights. The algorithm proceeds in three distinct phases. First, during the forward propagation phase, each neuron j computes its pre-activation $a_j = \sum_i w_{ji} z_i$, where w_{ji} is the weight from neuron i to neuron j and z_i denotes the output of neuron i (or the raw input x_i for the first layer). The post-activation output is then obtained by applying the activation function $z_j = h(a_j)$ [99, Ch. 8].

Second, during the error evaluation phase, the error signal at each output neuron k is initialised as $\delta_k = \partial E_n / \partial a_k$, which measures the sensitivity of the loss to the pre-activation at the output layer. Third, in the backward propagation phase, the error signal is propagated recursively through the hidden layers in reverse order. For each hidden neuron j , the delta is computed as [99, Ch. 8]:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

which applies the chain rule by weighting the error signals δ_k of the succeeding layer by the corresponding weights w_{kj} and the local derivative of the activation function $h'(a_j)$. Finally, the gradient of the loss with respect to each weight is evaluated as [99, Ch. 8]:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$$

This elegant result states that the gradient of any weight w_{ji} is simply the product of the error signal at the destination neuron δ_j and the activation of the source neuron z_i , making the

algorithm computationally efficient since each δ_j is computed once and reused for all weights entering neuron j [99, Ch. 8].

As for which loss functions are used, this depends on the task the model solves. For regression problems, one of the most common ones is the Mean Squared Error (MSE), also known as L2 loss [101] from calculating the Euclidean distance between the predicted output $\hat{y}_i$ and the ground truth y_i averaged by n samples, defined as:

$$\text{MSE}(\theta) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \frac{1}{n} \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2$$

While for classification problems, typically the Categorical Cross-Entropy Loss (LCE) is used, measuring the log loss of the output probability distribution vector $\hat{p}_i = [\hat{p}_{i,1}, \dots, \hat{p}_{i,C}]$ for sample i , defined as:

$$L(y_i, \hat{p}_i) = - \sum_{j=1}^C y_{i,j} \log(\hat{p}_{i,j})$$

where C is the number of all classes, $y_{i,j} \in \{0,1\}$ and $\sum_j y_{i,j} = 1$. Each output logit $z_i \in \mathbb{R}^C$ is transformed into a probability $\hat{p}_{i,j}$ through the softmax function (1) so that $\sum_j \hat{p}_{i,j} = 1$.

Averaging over n samples result in the final equation (2) [101].

$$\hat{p}_{i,j} = \frac{\exp(z_{i,j})}{\sum_{k=1}^C \exp(z_{i,k})} \quad (1)$$

$$\text{LCE}(\theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^C y_{i,j} \log(\hat{p}_{i,j}) \quad (2)$$

2.4.5 Attention and Transformers

Recurrent architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) process sequences step-by-step, which limits parallelisation and makes it difficult to capture dependencies between residues that are far apart in the primary chain but physically close in the folded protein. The attention mechanism, originally proposed to augment Recurrent Neural Network (RNN) encoder–decoders for neural machine translation [102], addresses this by allowing each position to directly weigh every other position in the sequence (Figure 2.11) [99, pp. 358–360]. Vaswani et al. [103] then showed that recurrence can be removed entirely: a model built purely on attention, the Transformer, attains state-of-the-art results while being fully parallelisable over the sequence dimension. These models are called that because they transform a set of embedding vectors in some representation space into a new space, keeping the dimensionality [99, pp. 357–358]. As the task requires a label at every residue position, we adopt an encoder-only Transformer the architecture underpinning BERT-style models [104] and ESM-2 [105] which preserves the full input sequence length and maps each token directly to a classification output, without the generative decoder component used in sequence-to-sequence tasks [103].

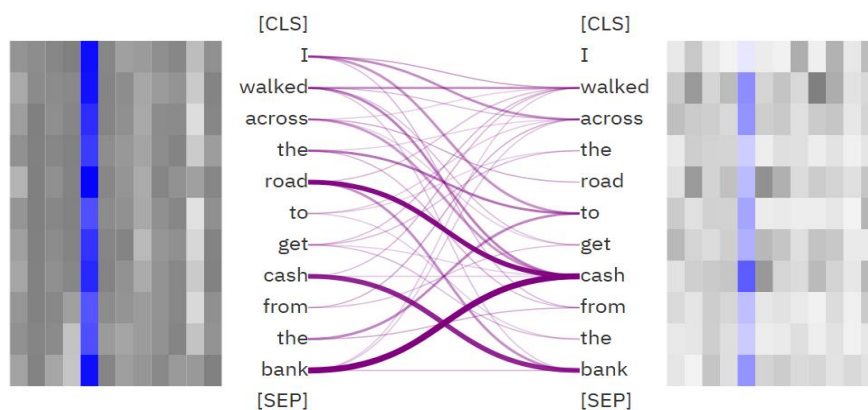


Figure 2.11. Example of attention weights of Layer 1/ Head 5

(Obtained from exBERT: <https://huggingface.co/spaces/exbert-project/exbert>)

2.4.5.1 Scaled Dot-Product Attention

To build intuition, attention can be viewed through the lens of information retrieval: a query is matched against descriptive representations (keys) to identify relevant items, whose underlying content (values) is then retrieved. Given an input sequence $X \in \mathbb{R}^{N \times D}$ of N residue embeddings of dimension D , the encoder learns three projections: $Q = XW_Q$, $K = XW_K$ and $V = XW_V$, mixing information across positions through scaled dot-product attention [99, Ch. 12], [103]:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

The softmax yields non-negative attention weights that sum to one along each row, so each output position is a convex combination of all value vectors. The $1/\sqrt{d_k}$ factor counteracts the growth in variance of the dot products as d_k increases, preventing the softmax from saturating into near-one-hot distributions where gradients vanish. Separate W_Q , W_K , W_V break the symmetry that pure self-attention would otherwise impose, which is essential when modelling asymmetric biological relationships between residues.

2.4.5.2 Multi-Head Self-Attention

Using a single attention function forces the model to collapse all types of dependencies into one distribution. In practice, different heads can specialise: some capture local secondary-structure motifs, others long-range contacts. Vaswani et al. [103] Multi-Head Attention (MHA) module therefore computes H attention functions in parallel, each with its own projections in a lower-dimensional subspace $d_k = d_v = D/H$ and concatenate the results [99, Ch. 12]:

$$\text{MHA}(X) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)W_O \quad (1)$$

$$\text{head}_h = \text{Attention} \left(XW_Q^{(h)}, XW_K^{(h)}, XW_V^{(h)} \right) \quad (2)$$

Because each head operates in a D/H dimensional subspace, the total computational cost is comparable to a single full-dimensional head.

2.4.5.3 Positional Encoding

Self-attention is permutation-equivariant: permuting the input tokens permutes the output identically, so the raw mechanism is blind to token order. Since the linear order of amino acids is biologically meaningful, positional information must be injected into the token embeddings [106] [99, pp. 371–374]. We follow the original sinusoidal scheme of [103] where for position n and dimension i :

$$r_{n,2i} = \sin \left(\frac{n}{10000^{2i/D}} \right), r_{n,2i+1} = \cos \left(\frac{n}{10000^{2i/D}} \right)$$

and r_n is added to the residue embedding. This encoding is bounded in $[-1,1]$, assigns a unique signature to every position and allows relative offsets to be expressed as linear functions of the encoding, which helps generalisation to sequence lengths beyond those seen in training.

2.4.5.4 Encoder Block

The encoder stacks L encoder blocks, each one of a multi-head self-attention sub-layer with a position-wise FFN,

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

applied identically and independently to each position. Each sub-layer is wrapped in a residual connection [107] and layer normalisation [108], which stabilises training for variable-length inputs (unlike batch normalisation [109], whose statistics are coupled to the batch dimension).

Two placements are standard: Post-LN: $\text{LN}(x + \text{Sublayer}(x))$, used in the original paper and Pre-LN: $x + \text{Sublayer}(\text{LN}(x))$, which Xiong et al., 2020 [110] show produces better-behaved gradients at initialisation and removes the need for learning-rate warm-up in many settings.

2.4.5.5 Vision Transformers

While Transformers were originally developed for Natural Language Processing, Dosovitskiy et al. [111] demonstrated that a pure Transformer encoder applied directly to sequences of image patches achieves state-of-the-art results on image classification benchmarks, giving rise to the Vision Transformer (ViT). Given an image $x \in \mathbb{R}^{H \times W \times C}$, ViT partitions it into $N = HW/P^2$ non-overlapping patches of size $P \times P$, flattens each into a vector $x_p \in \mathbb{R}^{P^2 C}$ and linearly projects the result into the model dimension D . Sinusoidal or learnable positional embeddings are added to each patch token to restore positional information lost by the permutation-equivariant attention mechanism [99, pp. 371–374], [111]. A learnable class token is prepended to the sequence; after passing through the encoder stack, this token's final representation is forwarded to an MLP classification head, as it aggregates global image context across all patch positions through self-attention [99, pp. 395–396].

2.4.6 ESM-2 Protein Language Model

Natural language processing demonstrated that transformer-based models pretrained on large text corpora via a masked language modelling (MLM) objective can produce contextual token representations that transfer effectively to diverse downstream tasks. The same paradigm was subsequently applied to protein sequences, treating amino acids as tokens in a molecular language, giving rise to Protein Language Models. ESM-2 is an encoder-only transformer family developed by Meta AI Research belonging to the ESM model lineage [105] [112]. It

was pretrained on approximately 65 million unique protein sequences from UniRef50 using the MLM objective: random amino acid positions are masked and the model learns to predict them from surrounding sequence context, with no explicit structural supervision of any kind [105], [112].

Despite this structurally unsupervised pretraining objective, ESM-2 exhibits emergent learning of sequence-structure relationships. Protein sequences inherently encode three-dimensional structural constraints through evolutionary co-variation: spatially proximal residues in the folded structure tend to co-evolve through compensatory mutations and this statistical signal becomes progressively embedded in the learned representations across training [105]. The resulting per-residue embeddings implicitly capture torsional propensities consistent with the Ramachandran plot distributions described in Section 2.1 where for instance, the helix-forming preference of alanine and the helix-breaking role of proline due to restricted ϕ rotation, as well as long-range co-evolutionary couplings between residues distant in sequence but proximate in three-dimensional space [112].

ESM-2 is available at multiple scales ranging from 8M to 15B parameters, all sharing the same encoder-only architecture comprising multi-head self-attention, position-wise feed-forward blocks, residual connections and layer normalisation. A notable architectural distinction from standard BERT-style models is the use of Rotary Positional Embeddings (RoPE) rather than absolute positional encodings, which encodes relative rather than absolute positional relationships and generalises more naturally to protein sequences of varying length [105]. These embeddings subsume much of the evolutionary signal traditionally derived from computationally expensive Multiple Sequence Alignment methods such as HHBlits, making them substantially richer input representations than raw one-hot amino acid encodings [105], [112].

2.4.7 Training Techniques

Training a deep network such as our Transformer encoder requires more than the vanilla gradient-descent rule introduced earlier. Finite biological datasets like CB513 and NetSurfP make overfitting and memorization the dominant failure mode, while the heterogeneous loss landscape of attention-based models demands an optimiser that adapts per-parameter and a schedule that stabilises convergence. This section describes four training techniques used in this work: Dropout, L2 / weight-decay regularisation, the Adaptive Moment Estimation with Decoupled Weight Decay (AdamW) optimiser and Cosine-Annealing learning-rate scheduling.

2.4.7.1 Dropout

Deep networks with many parameters tend to develop co-adaptation, where each layer learns to rely on a specific activation pattern from the previous layer. Dropout [113] breaks this by randomly zeroing each hidden unit with probability p on every forward pass during training, producing on each step a different thinned sub-network. The gradient then flows only through the surviving units, which effectively trains an exponentially large ensemble of sub-networks sharing weights. At inference, however, dropout is disabled and all units are active, so the incoming signal is proportionally larger. This inconsistency between training and inference time statistics causes the model's learned weights to be miscalibrated, meaning they were optimised for a weaker signal than what they receive at test time. One standard solution is the inverted dropout rule, which scales each surviving activation during training by $1/(1 - p)$ To compensate for the missing neurons by amplifying the signal to those that remain.[113].

$$h' = \begin{cases} 0 & \text{with probability } p \\ h & \\ \frac{h}{1 - p} & \text{otherwise} \end{cases}$$

2.4.7.2 L2 Regularisation and Weight Decay

A second way of controlling model complexity is to penalise the norm of the parameters directly. L2 regularisation adds a squared-norm penalty to the loss:

$$\tilde{\mathcal{L}}(\mathbf{w}) = \mathcal{L}_0(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

where $\mathcal{L}_0$ is the unregularised loss and $\lambda \geq 0$ is a hyper-parameter controlling regularisation strength. Substituting the derivative $\nabla \tilde{\mathcal{L}} = \nabla \mathcal{L}_0 + \lambda \mathbf{w}$ into the plain SGD update gives:

$$\mathbf{w}_{t+1} = (1 - \eta\lambda)\mathbf{w}_t - \eta\nabla\mathcal{L}_0(\mathbf{w}_t)$$

which is precisely the shrink-then-descend pattern known as weight decay. For plain SGD, therefore, L2 regularisation and weight decay are mathematically equivalent; for adaptive optimisers they are not, as discussed below [114].

2.4.7.3 From SGD to AdamW

An optimizer is an algorithm that iteratively adjusts a model's weights during training to minimize the loss function, like SGD. SGD uses a single global learning rate for every parameter, which is inefficient when the loss surface is poorly conditioned. A family of adaptive optimisers replaces η with a per-parameter effective rate derived from gradient history.

Adaptive Gradient (AdaGrad) [115] accumulates the sum of squared gradients and divides the update by its square root, so that parameters on steep directions receive smaller steps:

$$r_i^{(t)} = r_i^{(t-1)} + \left(\frac{\partial \mathcal{L}}{\partial w_i}\right)^2, \quad w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{r_i^{(t)} + \delta}} \frac{\partial \mathcal{L}}{\partial w_i}$$

Accumulating from the start of training causes the effective learning rate to decay monotonically, which slows late-stage learning. Root Mean Square Propagation (RMSProp) [116] replaces the cumulative sum with an exponential moving average,

$$r_i^{(t)} = \beta r_i^{(t-1)} + (1 - \beta) \left(\frac{\partial \mathcal{L}}{\partial w_i} \right)^2, 0 < \beta < 1$$

keeping adaptation responsive throughout training. Combining RMSProp's second-moment estimate with a momentum-style first-moment estimate yields Adaptive Moment Estimation (Adam) [99, Ch. 7], [117]:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \quad w_{t+1} = w_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

with typical values $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$ [99, Ch. 7]. Loshchilov and Hutter [114] observed that what most deep-learning libraries called "weight decay" inside Adam was actually L2 regularisation folded into the gradient g_t , which Adam then rescaled by $1/(\sqrt{\hat{v}_t} + \varepsilon)$. The regularisation strength therefore varies per parameter in an uncontrolled way, weakening it exactly for the large-gradient weights that most need it. AdamW fixes this by decoupling the decay term from the adaptive step entirely:

$$w_{t+1} = w_t - \alpha \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}} + \lambda w_t \right)$$

Here λ acts as a uniform, gradient-independent shrinkage applied after the adaptive update, recovering the true weight-decay semantics in the adaptive regime.

2.4.7.4 Learning-Rate Scheduling

A fixed learning rate forces a compromise: high values risk overshooting minima, while low values slow convergence. Learning-rate schedulers adapt α during training to get the best of both. A simple choice is step decay, which halves the rate every fixed number of epochs, but the abrupt transitions can destabilise adaptive optimisers.

This work uses Cosine Annealing [118], which decays the learning rate smoothly from an initial $\eta_{\max}$ down to a floor $\eta_{\min}$ along a half-cosine curve:

$$\eta_t = \eta_{\min} + \frac{1}{2} (\eta_{\max} - \eta_{\min}) \left(1 + \cos \frac{T_{\text{cur}}}{T_{\max}} \pi \right)$$

where T_{cur} is the current epoch within a cycle and $T_{\max}$ is the cycle length. The smooth profile combines large exploratory steps early in training with fine-grained refinement near convergence, pairing well with AdamW's coordinate-wise adaptivity.

Loshchilov and Hutter further introduce Stochastic Gradient Descent with Warm Restarts (SGDR) [118], which periodically resets $T_{\text{cur}} \leftarrow 0$ and raises η back to $\eta_{\max}$ each restart re-injects exploratory capacity, allowing the optimiser to escape poor local minima and traverse the loss landscape from a new initial point, while the ensemble of snapshots taken just before restarts often yields stronger final models.

2.4.8 Ensemble Methods

Ensemble learning refers to a methodology that combines several baseline models into a single, more powerful model than any individual constituent [119]. Because each base model captures different aspects of the data, their combination reduces the risk of overfitting and generalises better than any single learner, a property that has led to consistent performance gains across classification and regression tasks [119] [120, pp. 714–715].

2.4.8.1 Homogeneous and Heterogeneous Ensembles

Ensemble strategies differ in how they select and diversify base classifiers. A homogeneous ensemble consists of base classifiers of the same type, trained on different subsets of the data, where diversity arises from the variation in training samples rather than from the algorithm itself. A heterogeneous ensemble, by contrast, combines classifiers of different types trained on the same data, where diversity arises from the complementary inductive biases of the algorithms involved [121].

2.4.8.2 Sequential and Parallel Ensembles

Any ensemble framework can also be characterised by the dependency between its base learners. In a sequential ensemble, each successive model is trained to correct the residual errors of its predecessor, exploiting dependency between learners to progressively reduce bias. In a parallel ensemble, base learners are trained independently and simultaneously, with no data dependency between them; the key advantage is that errors across models differ and combining them allows the ensemble to average out individual mistakes [119].

2.4.8.3 Fusion Methods

Once base learners have produced their outputs, a fusion method determines how those outputs are combined. Voting methods, including hard (majority) voting, averaging and weighted averaging, aggregate base model predictions directly and are well-suited to classification tasks with comparable base model performance. Meta-learning, also known as stacking or learning to learn, instead trains a second-level learner on the outputs of the base models, allowing the meta-learner to discover non-linear relationships between base predictions and the true label [119], [121].

2.4.8.4 Bagging, Boosting and Stacking

The three most widely adopted ensemble frameworks each combine the characteristics above in a distinct way. Bagging (bootstrap aggregating), is a homogeneous parallel technique in which multiple classifiers of the same type are independently trained on different bootstrap replicates of the training data, reducing variance and mitigating overfitting. Boosting is a homogeneous sequential technique in which each model is trained to correct the errors of its predecessor. Stacking is a heterogeneous parallel technique in which base models of different types are trained on the same data and their predictions are passed as input features to a meta-learner that learns to optimally combine them [119], [121] [120, pp. 714–719].

2.4.9 Decision Tree and Random Forest

2.4.9.1 Decision Tree

A decision tree is built through a recursive, top-down partitioning process (Figure 2.12). Starting from the root node, which contains all training examples, the algorithm evaluates every available attribute and selects the one that yields the most informative split, branching the data

into child nodes accordingly [120, pp. 675–676]. This process repeats at each child node until a stopping condition is met: all examples in a node share the same label, no attributes remain, or a maximum depth constraint is reached [120, p. 678].

The guiding criterion at each split is Information Gain, which measures the reduction in uncertainty a candidate split produces. Uncertainty is quantified using Entropy, borrowed from information theory:

$$H(S) = - \sum_{x \in X} p(x) \log_2 p(x)$$

where S is the current dataset and $p(x)$ is the proportion of samples belonging to class x . A node where all examples share the same label has $H = 0$, while maximum entropy occurs when all classes are equally represented [120, pp. 679–680]. Information Gain for a candidate attribute A is then defined as:

$$\text{IG}(S, A) = H(S) - \sum_{t \in T} p(t) H(t)$$

where T are the subsets produced by splitting S on attribute A , $p(t)$ is the proportion of examples in subset t and $H(t)$ is its entropy [120, pp. 679–680]. The attribute maximising Information Gain is selected as the splitting criterion at each node. This formulation is closely related to the Cross-Entropy loss used to train neural networks, as both measure the divergence between a predicted distribution and the true label distribution.

2.4.9.2 Random Forest

A key limitation of individual decision trees is their sensitivity to the specific training data; small perturbations can produce substantially different trees, leading to high variance. Random Forest addresses this by constructing a bagging ensemble of decision trees, where each tree is trained on a bootstrap replicate of the training data and, at each split, only a random subset of features is considered as candidate attributes. After all trees are constructed, predictions are combined via majority voting for classification. The random feature subsampling decorrelates individual trees, ensuring that the ensemble benefits from variance reduction beyond what bootstrapping alone provides [120, pp. 714–716] (Figure 2.12).

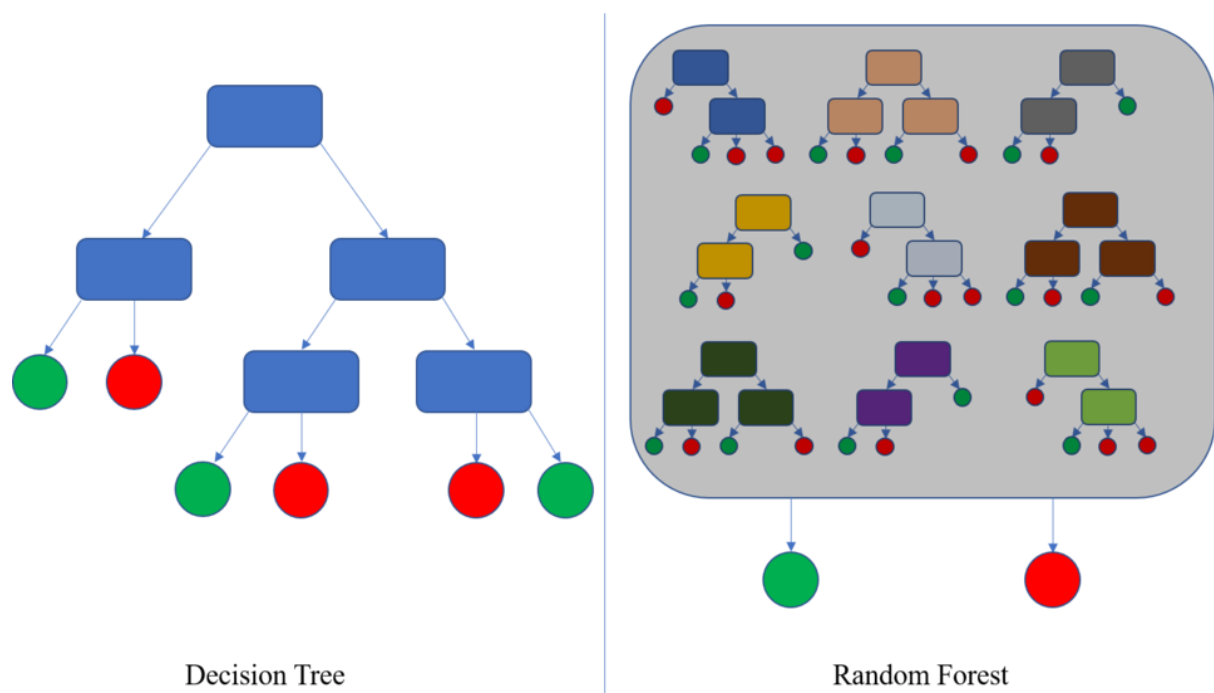


Figure 2.12. Decision Tree and Random Forest illustration. From [122], CC BY-SA 4.0

3 Related Work

3.1 Direct Comparison Methods

This section describes the three methods against which the present work is numerically benchmarked. All three are trained on the same NetSurfP-2.0 training dataset used in this thesis and evaluated on the independent test sets TS115 and CASP12, whose results are reported in the comparison tables of Chapter 5.

3.1.1 NetSurfP-2.0

NetSurfP-2.0 [60] is a multi-task deep learning framework for the joint prediction of secondary structure, relative solvent accessibility, backbone dihedral angles (Ramachandran angles) and disorder from a protein's primary sequence. It represents the ceiling of the profile-based paradigm and serves as the primary baseline for quantifying the accuracy gain obtained by substituting MSA-derived features with PLM embeddings. The architecture of two parallel 1D convolutional layers, following a two-layer Bidirectional LSTM (BiLSTM). The output then is fed into separate MLPs for each prediction task. The input representation encodes each residue as a 50-dimensional vector derived from evolutionary profiles generated by HHBlits [69] or MMseqs2 [123] searches, encoding position-specific amino acid probabilities computed from aligned homologous sequences.

3.1.2 NetSurfP-3.0

NetSurfP-3.0 [63] is the direct successor to NetSurfP-2.0. It replaces the HHBlits-generated input representation with embeddings from the ESM-1b PLM [124], while retaining the

identical Convolutional Neural Network (CNN) with BiLSTM downstream prediction head of its predecessor. ESM-1b is an encoder-only Transformer with 650M parameters pre-trained on 250 million protein sequences from UniRef50 using a masked language modelling objective, producing a 1280-dimensional per-residue embedding that implicitly encodes evolutionary and structural context without requiring a database search step. By eliminating MSA generation, NetSurfP-3.0 reduces inference time compared to NetSurfP-2.0, while achieving comparable accuracy. The authors noted sub-optimal performance when replacing the CNN + BiLSTM with a shallow Transformer encoder (2 layers, 8 heads) over ESM-1b embeddings [63]. However, this ablation used only 2 encoder layers and provides no detail on how the raw embeddings were fed to the encoder, making it an insufficient basis for concluding that Transformer encoders are unsuitable for PLM-based PSSP in general.

NetSurfP-3.0 is the most directly relevant baseline in this thesis: it uses the identical training set, the same three benchmark test sets and a closely related PLM input strategy. The primary distinctions of the present work are the substitution of ESM-1b with the ESM-2 (650M, 33 layers, 1280-dimensional) and the adoption of a Transformer encoder as the downstream architecture rather than a BiLSTM.

3.1.3 AttSec

AttSec [125] introduces a fundamentally different approach to exploiting PLM representations compared to NetSurfP-3.0. Rather than passing residue embeddings through sequential convolutional and recurrent layers, AttSec extracts the self-attention maps produced by multiple Transformer encoder layers and processes this stack of pairwise attention features through a 2D convolution to capture local hydrogen bond patterns. The input to the encoder are embeddings from the frozen ProtT5-XL-U50 checkpoint model [126].

Their results represent the highest published performance among no-evolutionary-information models evaluated on this benchmark combination, surpassing both NetSurfP-3.0 and the ProtT5-XL-U50 standalone baseline. AttSec therefore sets the primary accuracy target for both Q3 and Q8 against which the Transformer model proposed in this thesis is compared.

3.2 Acknowledged State-of-the-Art Methods

This section describes two methods that represent the current state of the art in alignment-free PSSP but are not directly compared against the results of this thesis, as they use different training sets and evaluation benchmarks. In cases where benchmark names coincide (such as CASP12 and CASP13) it could not be verified that the underlying protein subsets are identical to those used in this work and thus a fair comparison cannot be assumed.

3.2.1 SPOT-1D-LM

SPOT-1D-LM [64] is an ensemble predictor that combines the embeddings of two PLMs: ESM-1b [124] and ProtT5-XL-U50 [126] concatenated with one-hot encoding. This combined representation is processed by an ensemble of three architectures: a two-layer BiLSTM, a Multi-Scale Residual Network and a hybrid Multi-Scale Residual LSTM, whose outputs are averaged to produce the final prediction. The model is trained on 38913 proteins without any MSA generation. The central claim of the work is that combining embeddings from two complementary language models closes the accuracy gap between alignment-free and profile-based methods, achieving comparable performance. On TEST2018 it achieves 86.74% Q3 and 76.47% Q8, while on TEST2020 set performance drops to 79.82% Q3 and 66.68% Q8. On their own filtered Free Modelling subsets, SPOT-1D-LM achieves 77.45% Q3 on CASP12-FM and 82.15% Q3 on CASP13-FM [64]. SOV is not reported across any of these benchmarks.

3.2.2 Porter 6

Porter 6 [65] is the latest iteration of the Porter family of predictors, using ESM-2 [105] embeddings as its sole input fed into a Convolutional-Bidirectional RNN (CBRNN) architecture. The model is an ensemble of five CBRNN predictors generated via 5-fold cross-validation, trained on PDB80, comprising approximately 55,500 non-redundant sequences filtered at 80% sequence identity. Porter 6 uses the same ESM-2 model variant (650M parameters) as the present thesis, making it architecturally the closest published reference point for the embedding choice. On an independent 2024 benchmark of 692 proteins Porter 6 achieves 84.56% Q3 and 74.18% Q8 [65]. SOV is not reported. It is acknowledged here as a state-of-the-art reference rather than a direct comparison method, since it reports results on independent test sets not shared with this work.

3.3 Post-Processing Lineage

This section describes the series of prior works from the University of Cyprus research group that this thesis directly extends. Each work built upon the post-processing methodology of its predecessor, forming a clear line of development in applying increasingly powerful filtering techniques to PSSP.

Kountouris et al. [127] performed a comparative study of post-processing filters on top of a Bidirectional RNN (BRNN) ensemble. They evaluated an array of classification algorithms like Naïve Bayes, Support Vector Machine and the Radial Basis Function (RBF) network acting as post-processing filtering on top of the base models' predictions. Additionally, they examined whether empirical/external rules (ER) can be used as filtering instead of an algorithm. Their rules were derived from two sets, one from Salamov and Solovyev [128] and the other set from

Wood and Hirst [129]. The conclusions of this work were that additional filtering improved significantly the SOV metric in most cases and that some classifiers increase the per-residue accuracy, whereas others decrease it. Logistic function, the Multilayer Perceptron and the Support Vector Machines were found to be superior from other methods in terms of both Q3 and SOV score.

Following the above work, Dionysiou et al. [130] and Charalampous et al. [131] expanded this approach to Convolutional Neural Network and Feed-Forward Neural Network architectures respectively, both employing an SVM as their post-processing filtering algorithm. Dionysiou et al. [130] trained their CNN using standard SGD, while Charalampous et al. [131] further introduced a second-order optimization algorithm, Hessian-Free Optimization (HFO), demonstrating its effectiveness for the PSSP problem.

Finally, Chatzimiltis et al. [132] in their work employed a variant of the HFO algorithm, the Subsampled Hessian Newton method within a CNN architecture, combining both external rules and filtering algorithms. Specifically, the filtering algorithm used was a Random Forest classifier and the empirical ruleset derived from Salamov and Solovyev [128], proving that both strategies can be combined further increasing the per-residue accuracy and SOV metric. They also evaluated how different window sizes affect the performance of the model, proving a monotonic increase in the metrics as the window increases.

3.4 Other Transformer-based PSSP Methods

This section describes two additional transformer-based PSSP methods that are relevant to the methodology of this thesis but are not directly comparable, as both rely on alignment-based input features and are evaluated on different benchmarks.

MFTrans [133] proposes a multi-feature fusion network that integrates four distinct input representations: raw sequence, MSA-derived features, evolutionary information and language model embeddings, fusing them through a cosine-similarity-based multi-view strategy. The architecture employs two Transformer components: a pre-trained MSA Transformer for extracting evolutionary features from sequence alignments, and a Transformer encoder-decoder for enhancing the fused representation prior to a hybrid CNN and Bidirectional GRU (BiGRU) network. It is trained on CullPDB, comprising 5,600 sequences filtered at 25% sequence identity. MFTrans reports Q3 accuracy across all benchmarks; Q8 and SOV are not reported. On CB513 it achieves 88.34% Q3, on CASP12 84.83%, on CASP13 82.47%, on CASP14 84.31%, on TEST2016 87.17% and on TEST2018 88.42% [133]. The key distinction from the present thesis is that MFTrans relies on MSA profiles as a core input feature, whereas this work is entirely alignment-free.

SADGRU-SS [134] proposes a novel architecture combining self-attention with asymmetric MLP-GRU blocks and a dense block, using evolutionary-information-based features (PSSM and HMM profiles) as inputs. The model is trained on CB6133-filtered, comprising 5,534 sequences with CB513 redundancy removed. The self-attention mechanism is applied at the front of the network to enrich the input representation before the recurrent processing stages, and ablation experiments confirm that this placement outperforms other configurations [134]. Similarly to MFTrans, SADGRU-SS relies on alignment-based features, positioning it as a complementary reference point to this thesis.

4 Methodology

The implementation of the methodology presented in this section was conducted using the PyTorch [135] framework and is shown in the respective Appendices. However, it is highly recommended to check the code in the GitHub repository, for newly updated and refined code (source: <https://github.com/seggles01/PSSP-Implementation-BSc-Thesis.git>).

4.1 High Level Overview

The proposed methodology follows a two-stage, ensemble-based pipeline. First, an encoder-only Transformer is trained on frozen ESM-2 embeddings to perform residue-level secondary structure prediction. For each task (3-state and 8-state), an ensemble of three base models is constructed by training the same architecture with different random initialisations. At inference time, their per-residue class probability vectors are averaged to produce a single, seed-ensemble prediction for each protein. In the second stage, this averaged probability sequence is passed through a Random Forest post-processor operating on sliding window, followed (in the 3-state setting only) by external rules that remove structurally implausible short segments. Conceptually, this corresponds to a lightweight stacking ensemble, where the Transformer models act as first-level learners and the Random Forest serves as a meta-learner that refines their predictions using local contextual information, with the external rules providing a final deterministic correction layer.

4.2 Dataset Preparation

As mentioned in Chapter 2, high quality data for training and testing the model is essential for the model to recognize meaningful patterns and generalize on unseen data. Fortunately, in this work, two sets of data are used.

The first set, provided by the previous work from the University of Cyprus Research group [130], [131], [132] included the PISCES and CASP13 datasets. The PISCES dataset was one of the two training datasets used and was extracted from the PISCES culling server [74] containing 8632 protein sequences with maximum similarity per protein 25%, resolution cutoff of 3.0 Å and an R-factor cutoff of 1.0 [132]. The CASP13 dataset contains 40 sequences with 13 of them matching at least one protein in the PISCES dataset, however, prior work produced results both on the full CASP13 dataset and the reduced version (removed identical sequences), showing minor change in performance [132]. Both datasets of this set are labelled as 3-class classification.

The second set is online available by the NetSurfP team [60], [63] (source: <https://services.healthtech.dtu.dk/services/NetSurfP-3.0/>) including their own training set, CB513, CASP12 and TS115. The NetSurfP train set, being the second training set in this work, includes 12185 sequences from the Protein Data Bank, culled and selected by the PISCES server [74] with 25% sequence similarity clustering threshold and a resolution of 2.5 Å or better. To avoid overfitting, any sequence that had more than 25% identity to any sequences in the test datasets CB513, CASP12, TS115 was removed, as well as peptide chains with less than 20 residues, leaving 10837 sequences. The CASP12 dataset consists of 21 sequences, filtered exclusively from the Free Modelling targets [60], [63]. The TS115 and CB513 test datasets are untouched. The datasets of this set are labelled as 8-class classification, which allows to extract both 3-state and 8-state results via mapping the specialized class into the general one.

These datasets also include a range of structural and evolutionary features. Hidden Markov Model profiles, derived from MSAs, are provided as input features, while backbone dihedral angles (ϕ and ψ), along with absolute and relative solvent-accessible surface areas, are used as target labels, as in models such as NetSurfP [60], [63]. Despite the availability of evolutionary information through HMM profiles, this study does not incorporate them, in order to avoid the

computational cost and dependency associated with MSA generation and instead leverages Protein Language Models as a more efficient alternative.

Furthermore, it is important to note that within each dataset set, sequence overlap between training and test splits has been explicitly verified. The PISCES training set was curated to have no sequence similarity with the CASP13 benchmark, as also noted in prior work by Chatzimiltis et al. [132]. Similarly, the NetSurfP training set has been verified by its authors to contain no sequences overlapping with the CASP12, TS115 and CB513 benchmarks [60], [63]. However, cross-dataset independence, that is, between the training set of one source and the benchmarks of the other, cannot be fully verified, as no such cross-checking was performed by either party. For this reason, all experiments strictly respect the original (training set, benchmark) pairings, ensuring that reported results remain internally valid and directly comparable to prior works.

4.3 Input Representation

Each input protein sequence is processed through the pretrained ESM-2 model [105] (checkpoint facebook/esm2_t33_650M_UR50D) to obtain per-residue contextual embeddings. This variant model consists of total 650M parameters, 33 transformer layers, 20 attention heads and produces 1280-dimensional vectors. It was loaded via the Hugging Face transformers library [136] in inference mode with all parameters frozen instead of fine-tuning it along with the base model. For each input sequence of length L , a single forward pass through the frozen ESM-2 encoder yields a tensor of shape $(L+2) \times 1280$, corresponding to all residues plus the prepended beginning of sentence and appended end of sentence boundary tokens [105]. The boundary token embeddings are discarded, leaving a per-residue embedding matrix $H \in \mathbb{R}^{L \times 1280}$, where each row encodes a single residue's contextual representation.

This matrix is passed directly as input to the downstream architecture. The scripts responsible for the extraction and saving of the embeddings are in Appendices I and II.

Choosing this specific variant allows three things: Firstly, to balance the computational cost and the representation quality of each protein in contrast to other variants. Secondly, to align with established PLM-based implementations like Porter 6 [65] which also uses the same variant. Thirdly, we use a variant that outputs 1280-dimensional vectors and has 650M parameters, the same architectural configuration as ESM-1b [124], thus comparisons with methods like NetSurfP3.0 [63] ensures that any performance differences reflect architectural and training choices rather than disparities in embedding dimensionality or model capacity.

As for the choice to freeze the ESM-2 encoder, while parameter-efficient fine-tuning has been shown to improve downstream performance across diverse protein prediction tasks [137], the gain for secondary structure prediction specifically is minimal. Schmirler et al. [137] explicitly note this as an 'insignificant gain', with improvements of at most 0.8 percentage points in Q3 accuracy and attribute this to secondary structure information already being encoded in the frozen embeddings during unsupervised pretraining. Similarly, Saadat & Fellay [138] report only marginal Area Under the Receiver Operating Characteristic (AUROC) improvements for structural features (α -helix, β -strand, turn) when comparing fine-tuned to frozen ESM-2 classifiers. Given these findings, the decision to use frozen ESM-2 embeddings is not only computationally practical but also specifically justified for the secondary structure prediction task.

4.4 Base Model Architecture

This section describes the single model architecture used in this work. It should be noted that the initial architecture aimed to build was ViT based [111], instead it can be described best as an encoder-only Transformer.

The input in the encoder is the 1280-dimensional vector embedding extracted from ESM-2 for one residue in the protein. This vector is sliced into a set number of token vectors e.g. If the number of tokens set is T , then the 1280 vector is transformed into a matrix $I \in \mathbb{R}^{T \times D}$ where $D = \frac{1280}{T}$. Sinusoidal positional encoding is added into the vectors and forwarded to the encoder layers. In each encoder block, tokens pass through multiheaded self-attention and the FFN sublayers each having a residual connection and Post-Layer-Normalization (Post-LN). Each FFN sublayer consists of two layers with Rectified Linear Unit (ReLU) activation function, as defined by Vaswani et al. [103]. After passing through all the layers, the self-attended tokens are then averaged and passed into the final-MLP head that classifies the secondary structure on that specific residue. This MLP consists of two hidden layers and Gaussian Error Linear Unit (GELU) activation, derived from the ViT architecture [111]. The full model implementation is presented in Appendix III and the module for extracting the model's predictions is in Appendix VII.

The use of only one embedding per residue arises from the fact that each 1×1280 vector extracted from ESM-2 already encodes both local residue context and global sequence-level dependencies through the self-attention mechanism [105], making additional features redundant. Including more features per residue would increase computational and memory overhead without proportional gain in representational quality. Rather than treating this vector as a single opaque input, it is partitioned into T non-overlapping slices, each serving as an independent token. This decomposition allows the self-attention mechanism to explore intra-

embedding structure, capturing relationships between different subspaces of the representation and identifying more complex compositional patterns within the ESM-2 embedding itself.

Sinusoidal positional embeddings were added to each slice to indicate their relative order within the partitioned embedding. Although ESM-2 [105] internally uses RoPE, these encode the sequential position of each amino acid residue within the protein chain, information already injected into the values of the 1280-dimensional output vector. They carry no information about the ordering of subspaces within that vector. The sinusoidal embeddings applied here therefore serve a distinct and complementary role: encoding the positional order of slices within the decomposed representation, which the self-attention mechanism would otherwise treat as a permutation-invariant set.

Beyond their positional role, the sinusoidal embeddings also had an important regularisation effect. Early experiments conducted without positional embeddings revealed that the model converged rapidly in the first few epochs but exhibited severe overfitting, with the strong representational signal from ESM-2 allowing the model to fit the training distribution closely without learning robust structural patterns. Notably, dropout and weight decay alone were insufficient to resolve this issue. The addition of sinusoidal embeddings introduces a small but consistent perturbation to each token vector which, in combination with dropout and weight decay, caused the model to converge more slowly and exhibit more stable generalisation. The sinusoidal embeddings therefore serve a dual purpose in this architecture: a principled positional encoding that preserves slice order and an implicit regulariser that prevents the model from over-relying on the raw ESM-2 signal.

As described in Section 2.4.5.5, a standard ViT assumes image-like input and employs a convolutional or linear patch extraction step that maps patches to a fixed projection dimension, along with a learnable class token prepended to the patch sequence for classification [111]. The

architecture presented here handles the analogous operation differently in two respects. First, patch creation is performed manually by partitioning the 1280-dimensional ESM-2 embedding into the T non-overlapping slices of equal size, without a subsequent linear projection. This is equivalent to the patch embedding step in ViT, but without the additional learned transformation. Since the ESM-2 vectors already reside in a rich representational space, further projection would add parameters without adding representational benefit. Second, rather than using a learnable class token for classification, mean pooling is applied over the T attended slice tokens. Previous experiments with the class token showed consistently worse performance, likely due to a distributional mismatch with slice tokens from the ESM-2 embedding. While these tokens follow a well-formed distribution shaped by deep transformer processing, the randomly initialised token starts as an outlier and disrupts early attention. Mean pooling avoids this by aggregating all tokens symmetrically, resulting in more stable and accurate performance.

A diagram of the architecture is shown in Figure 4.1. The model is defined by the embedding dimension (`embed_dim`, default 1280), which is divided into a fixed number of tokens (default 5). The transformer encoder is configured by the number of attention heads (`num_heads`), encoder layers (`layers`) and the Feed-Forward Network scaling factor (`d_ff_multiplier`). The final MLP head is controlled by `mlp_multiplier`. Additional parameters include the number of output classes (`num_classes`) and the dropout rate (`dropout` $\in [0, 1]$).

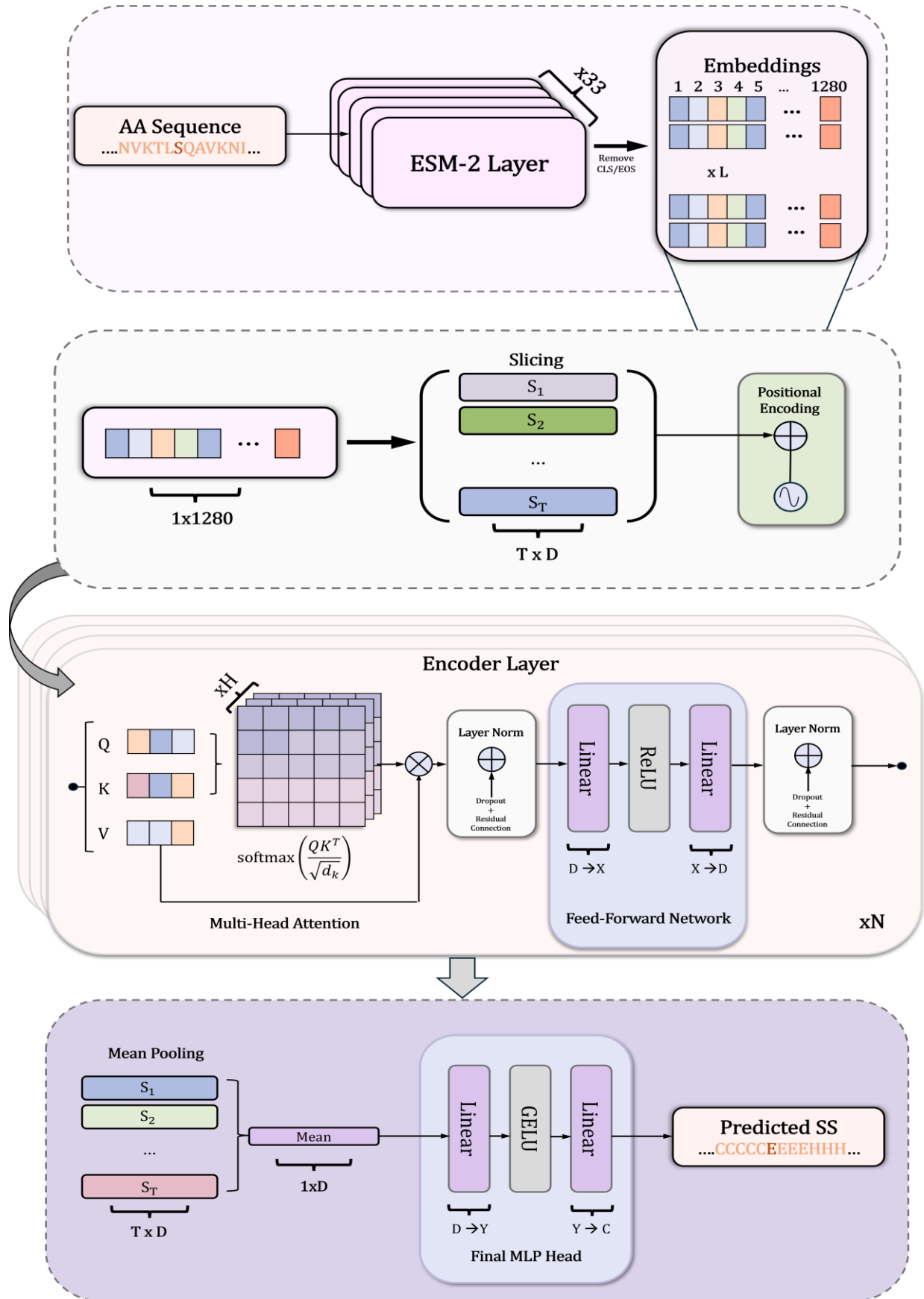


Figure 4.1. Proposed Architecture for PSSP

4.5 Filtering Methods

As described Chapter 3, this work applies two complementary post-processing strategies on top of the base model's predictions: a learned classifier (Random Forest) and a set of external rules derived from Salamov and Solovyev [128]. This section describes the concrete implementation of each strategy.

4.5.1 Random Forest Post-Processor

Post-processing filtering is implemented by training a secondary classifier on the outputs of the base model. The base ensemble consists of three independently trained models initialised with different random seeds and trained on the full training set. For each sample, their predictions are averaged to produce ensemble probability estimates, which are then converted into fixed-length sliding window representations used to construct the training data for the Random Forest (RF) post-processor. The trained post-processor is subsequently applied to ensemble-averaged predictions on the unseen test set to produce the final secondary structure assignments. Since proteins vary in length, a sliding window of fixed width is applied over the per-residue predictions, so that every sample presented to the classifier has a uniform dimensionality. Based on the findings of Chatzimiltis et al. [132], window sizes in the range of 21–31 residues provide the best performance, with a monotonic improvement observed as the window grows. Thus, a window of 31 is used throughout this work for consistency with that prior study.

A key design choice concerns both the representation of the base model's predictions fed into the post-processor and the architecture of the post-processor itself. Kountouris et al. [127] used raw class probabilities as input to a filter trained on top of a BRNN base model, whereas Chatzimiltis et al. [132] used hard predicted class indices derived from a CNN. Notably, the

former found that an RBF kernel SVM achieved the best filtering performance, while the latter reported that a Random Forest was the most effective classifier. This divergence suggests that the optimal post-processing classifier is not a universal property, meaning it depends not only on the representation it receives, but also on the base model architecture it operates on.

Probabilistic outputs are preferred here because they carry strictly more information than class indices: even when the predicted class is the same, a distribution such as $p_H = 0.4$, $p_E = 0.3$, $p_C = 0.3$ reveals substantial ambiguity that a windowed classifier can exploit, whereas a single index discards this uncertainty entirely.

Accordingly, the Random Forest is adopted as the post-processing classifier for both 3-state and 8-state prediction. This choice is motivated by three converging factors. First, it is the most recent and directly relevant precedent in this lineage, reported by Chatzimiltis et al. [132] on a comparable PSSP post-processing setting. Second, Chatzimiltis et al. [132] identified a window size of 31 as optimal for the RF classifier and this configuration is adopted here for consistency and comparability. Third, both prior works in this lineage employed a RBF kernel SVM as the primary alternative classifier, whose extensive computational requirements were explicitly noted by Kountouris et al. [127], who excluded it from further combination experiments on those grounds. A Random Forest is substantially more lightweight and faster to train, making it practical to retrain independently for both 3-state and 8-state prediction without significant computational overhead.

The Scikit-learn [139] Random Forest post-processor is configured as follows. Following Chatzimiltis et al. [132], `n_estimators = 100` defines the number of decision trees grown in the ensemble, providing sufficient diversity without excessive training cost and `max_depth = 25` imposes a limit on tree growth, preventing expansion that would lead to memorisation rather than generalisable patterns. `max_features = "sqrt"` restricts each split to a randomly drawn

square-root-sized subset of features, deliberately decorrelating the trees so their errors do not share the same systematic biases. `min_samples_leaf` enforces a minimum occupancy threshold on leaf nodes, acting as a lightweight structural regulariser against overfitting. Finally, `class_weight = "balanced"` rescales each class's contribution to training inversely proportional to its frequency which is crucial in the PSSP setting where the Coil class dominates and without which the classifier would systematically under-predict minority classes such as Strand and the rare 8-state sub-classes.

4.5.2 External Rules

Prior to the adoption of learning-based post-processors, Salamov and Solovyev [128] introduced a set of three deterministic correction rules applied as a final post-processing step, observing that residue-level predictors frequently produce structurally unrealistic short segments such as isolated single-residue helices or strands that are rarely observed in experimental structures. The first rule converts all predicted helices of length one or two residues to coil, except for EHE, which is instead converted to EEE (e.g. CCHHEE $\rightarrow$ CCCCEE, EEHHEE $\rightarrow$ EEEEE). The second rule converts all predicted strands of length one residue to coil (e.g. HHEHH $\rightarrow$ HHCHH). The third rule converts all predicted strands of length two residues that are surrounded by helical residues on both sides to helix (e.g. HHEHH $\rightarrow$ HHHHHH). Despite their simplicity, these three rules produced a consistent improvement of approximately 0.2% in Q3 accuracy in Salamov and Solovyev's original pipeline and their combination with a learned classifier was confirmed by Kountouris et al. and Chatzimiltis et al. to yield the highest overall accuracy and SOV gains. The implementation of the external rules module is provided in Appendix IV.

The external rules are applied exclusively to 3-state predictions and are deliberately excluded from the 8-state pipeline. The three dominant classes in 8-state prediction C (coil), H (α -helix)

and E (β -strand) account for the vast majority of residues in any dataset and carry the most weight in the Q8 accuracy metric. Enforcing structural rules that predominantly correct transitions among these dominant classes would inflate the overall accuracy while systematically suppressing predictions of the minority classes G (3_{10} -helix), I (π -helix), B (β -bridge), T (turn) and S (bend). In the extreme, a model that never predicts any rare class can approach Q3-level accuracy on the Q8 metric if only $\sim 5\%$ of residues belong to those classes, making any reported gain an artefact of class imbalance rather than a genuine improvement in structural discrimination. Any apparent accuracy gain from applying these rules to Q8 would therefore reflect a collapse toward the dominant class distribution rather than a meaningful improvement in the model's ability to resolve the full 8-state alphabet.

4.5.3 Combined Pipeline

In this work, both filtering strategies are combined, examining whether a specific ordering performs best. Based on the findings of Chatzimiltis et al. [132], applying the Random Forest first before the external rules (RF $\rightarrow$ ER) proved to be better than the inverse order (ER $\rightarrow$ RF). In the RF $\rightarrow$ ER ordering, the Random Forest operates on the original ensemble probability vectors and produces a predicted label sequence, upon which the external rules are then applied as a purely downstream string-level correction. The ER $\rightarrow$ RF ordering requires special handling, however, since the Random Forest expects probability vectors as input while the external rules operate on hard labels. To reconcile this, a hybrid probability representation is constructed: positions where an external rule fires are replaced by a one-hot vector encoding the corrected class with certainty 1.0, while positions unaffected by the rules retain their original soft probability distributions. This hybrid tensor is passed through the sliding window feature construction and into the Random Forest, allowing the ER-corrected certainty to propagate as a contextual signal through the window to neighbouring positions.

4.6 Training Procedure

In this section, we describe the main components of the training procedure used for the base model. In all experiments, the cross-entropy loss function was used for optimisation which is the standard for multi-class classification and the batch size was set to 256. For reproducibility we used for each model a fixed seed (42, 123 and 1337) across all executions. The training utilities, training and evaluation pipeline and configuration are provided in Appendices VI, IX and X respectively.

Early stopping was employed to prevent unnecessary training once the model stopped improving. Specifically, a patience counter monitored the validation loss across epochs where if the loss did not decrease for a fixed number of consecutive epochs, the training process was terminated. The patience and maximum epoch values varied between the 3-state and 8-state tasks with the 8-state configuration assigned higher patience and more epochs, reflecting the greater structural complexity of the task and the correspondingly longer convergence time required to resolve the finer distinctions among eight secondary structure classes. The exact values used at each stage are detailed in the respective sections.

Training was performed using the AdamW optimiser [114]. The choice of AdamW is not only theoretically cleaner than standard Adam with L2 regularisation, but structurally appropriate for a Transformer encoder. Zhang et al. [140] analyse the Hessian spectra of Transformer models and identify block heterogeneity: different parameter blocks: attention projections, FFN layers and embeddings exhibit dramatically different curvature, a property absent in CNNs. A single global learning rate cannot accommodate these simultaneously, whereas Adam-style coordinate-wise rates adapt to each block independently. The authors show that even exhaustive SGD tuning cannot close the gap with AdamW on Transformer tasks, while the two optimisers perform comparably on convolutional networks.

Across all training schemes, the beta parameters were set to $\beta_1 = 0.9$ and $\beta_2 = 0.95$, respectively. A reduced second moment decay rate of $\beta_2 = 0.95$, rather than the conventional 0.999, was adopted to limit the effective gradient variance window, ensuring that the running estimate remains representative of a smaller number of recently observed residues to compensate the structural heterogeneity of protein chains in the training set, where accumulating statistics over a large number of residues risks averaging gradient information across fundamentally different structural contexts.

The learning rate and weight decay were treated as hyperparameters and tuned using the Optuna framework [141], as described in the following section. To further stabilise training, gradient clipping was applied with a maximum norm of 1.0 to prevent exploding gradients during backpropagation.

A Cosine Annealing learning rate scheduler without restarts was employed to gradually decrease the learning rate throughout training, providing a smooth decay that helps stabilise optimisation in later training stages. The minimum learning rate was fixed at 10^{-6} [118]. Restarts were omitted since the training runs are relatively short and bounded by early stopping; periodic resets would interfere with the convergence signal monitored by the patience counter, potentially resetting the learning rate just as the model approaches a good minimum.

All experiments were implemented using the PyTorch framework [135] and executed on an NVIDIA V100 GPU. Training was performed using 32-bit floating-point precision without mixed-precision optimisation to ensure numerical stability.

4.7 Hyperparameter Optimization

Hyperparameter tuning was performed using Optuna [141], a Bayesian optimisation framework that intelligently navigates the search space rather than exhaustively enumerating

combinations. Folds 0, 1 and 2 from NetSurfP [60] were used for training in each trial, with fold 3 serving as the fixed validation set to measure generalisation. Optuna was applied separately for both 3-state and 8-state classification, with 30 trials in each case. Each trial trained the model for 20 and 30 epochs for the 3-state and 8-state tasks respectively, with the best validation accuracy across all epochs reported as the trial's objective value. The hyperparameter tuning script is provided in Appendix V.

4.7.1 Tree-structured Parzen Estimator Sampler

Optuna used the Tree-structured Parzen Estimator (TPE) sampler [142] with multivariate modelling enabled [143]. Rather than treating each hyperparameter independently, multivariate TPE models the joint probability distribution over all parameters simultaneously, capturing interactions between them [143]. The first 5 trials were random exploration to build an initial baseline, after which TPE used the accumulated trial history to propose parameter combinations more likely to improve upon the current best [141], [142].

4.7.2 Median Pruner

The Median Pruner was used to terminate unpromising trials early [141]. After a warmup period of the first 30% of epochs, during which all trials run unconditionally to allow the model to leave random initialisation, the pruner evaluates each trial's validation accuracy every 2 epochs and compares it against the median accuracy of all previously completed trials at the same epoch. If a trial falls below this median, it is pruned immediately, freeing GPU resources for more promising configurations. The first 5 trials are never pruned regardless of performance, ensuring the median estimator has sufficient data before making pruning decisions [141].

5 Experiments and Results

All the experiments shown in this section were conducted using the PyTorch [135] framework and is shown in the respective Appendices. However, it is highly recommended to check the code in the GitHub repository for newly updated and refined code (source: <https://github.com/seggle01/PSSP-Implementation-BSc-Thesis.git>).

5.1 Hyperparameter Tuning Results

Hyperparameter optimisation was performed using Optuna [141] with a TPE sampler [142], [143] over 30 trials, as described in Section 4.7. The search was conducted independently for Q3 and Q8 classification, with each trial evaluated on a held-out validation fold. The selected hyperparameters for both tasks are summarised in Table 5.1.

	3-State	8-State
Hyperparameter	Value	Value
embed_dim	1280	1280
tokens	5	10
d_ff_multiplier	4	5
num_heads	16	8
layers	6	7
mlp_multiplier	7	6
dropout	0.1	0.1
Learning Rate	1e-4	2e-4
Weight Decay	1.8e-5	1.3e-3

Table 5.1. The selected hyperparameters for the base model in both 3-state and 8-state tasks.

The embedding dimension of 1280 is fixed across both tasks, reflecting the output dimensionality of the ESM-2 650M encoder. The number of tokens is the only deliberate architectural asymmetry between the two configurations: Q3 uses 5 tokens, producing a 5×5 attention matrix over embedding segments, while Q8 uses 10, yielding a 10×10 matrix. This provides finer-grained pairwise interactions between segments of the residue representation, a design choice motivated by the greater structural complexity that 8-state classification demands.

The Q8 model also converged to a higher learning rate ($2e-4$ vs. $1e-4$) and a substantially larger weight decay ($1.3e-3$ vs. $1.8e-5$), suggesting that the 8-state task benefits from stronger regularisation to avoid overfitting on the more fine-grained label space. The deeper feed-forward multiplier (5 vs. 4) and additional transformer layer (7 vs. 6) are consistent with the increased model capacity needed for the harder classification problem.

5.2 Results in 3-State

This section presents the results of this work's model in the 3-State Classification task. First, the k-fold cross-validation results on both PISCES and NetSurfP training sets are shown. Then the cross-validation results after applying the post-processing algorithm are presented, followed by the evaluation results on independent test sets using combinations of post-processing and external rules.

5.2.1 Cross-Validation Results

Table 5.2 and Table 5.3 report the 5-fold and 7-fold cross-validation results on the PISCES and NetSurfP datasets respectively, using an ensemble of three models trained with different random seeds (42, 123 and 1337). The cross-validation pipeline is provided in Appendix VIII.

The PISCES ensemble achieves a mean Q3 accuracy of 86.62% with a standard deviation of just 0.13, indicating highly stable performance across folds. The SOV'99 and SOV_{refine} scores follow the same pattern of consistency, with standard deviations of 0.54 and 0.60 respectively. At the per-class level, helix (SOV'99_H = 86.82) is predicted with the highest segment-level accuracy, followed by strand (82.66) and coil (74.70). This ordering is consistent with the well-established observation in the PSSP literature that α -helices, being longer and more regularly structured, are easier to segment correctly than coil regions, which are structurally irregular and often short. The SOV_{refine} scores are uniformly lower than their SOV'99 counterparts across all classes, which is expected given that SOV_{refine} applies a length-dependent penalty ($\lambda=0.5$) that more harshly penalises short, fragmented predictions.

Fold	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
0	86.50	80.30	73.50	85.90	81.30	74.30	81.20	74.80	65.20
1	86.60	81.40	74.70	87.10	83.70	75.00	82.60	77.30	66.00
2	86.70	81.30	74.80	87.20	82.30	75.30	83.00	76.00	66.50
3	86.50	80.60	73.90	86.80	82.80	73.70	82.20	76.60	64.80
4	86.80	81.50	74.80	87.10	83.20	75.20	82.60	76.80	66.10
Mean	86.62	81.02	74.34	86.82	82.66	74.70	82.32	76.30	65.72
<i>Std</i>	<i>0.13</i>	<i>0.54</i>	<i>0.60</i>	<i>0.54</i>	<i>0.92</i>	<i>0.68</i>	<i>0.69</i>	<i>0.96</i>	<i>0.70</i>

Table 5.2. 5-fold Cross-Validation Results in PISCES dataset (3-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337).

The NetSurfP ensemble achieves a marginally higher mean Q3 accuracy of 87.17% compared to PISCES (86.62%), a difference of 0.55 percentage points. However, this residue-level gain does not extend to segment-level metrics. SOV'99 drops to 76.21 (vs. 81.02) and SOV_{refine} to 69.39 (vs. 74.34), representing substantial deficits of approximately 5 points in both metrics. The per-class breakdown reveals that the degradation is most severe in the coil class, where SOV'99_C falls to 67.21 (vs. 74.70) with a standard deviation of 3.77, suggesting high fold-to-

fold instability in coil segment prediction. Helix remains the best-predicted class ($\text{SOV}'99_{\text{H}} = 85.16$), but the advantage over strand (83.14) narrows considerably compared to PISCES.

Fold	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
0	87.00	77.00	70.40	84.50	81.80	69.50	80.10	75.40	61.10
1	87.40	79.10	72.30	87.60	83.40	71.00	83.10	76.80	62.50
2	87.70	77.10	70.60	86.30	84.00	67.50	81.80	77.60	59.50
3	86.90	76.20	69.10	82.40	83.50	68.80	77.20	76.40	60.20
4	86.80	74.30	67.20	85.20	82.60	63.80	80.00	75.40	55.60
5	87.50	77.70	71.10	84.50	84.50	69.50	79.90	78.00	61.20
6	86.90	72.10	65.00	85.60	82.20	60.40	80.40	74.80	52.20
Mean	87.17	76.21	69.39	85.16	83.14	67.21	80.36	76.34	58.90
<i>Std</i>	<i>0.35</i>	<i>2.33</i>	<i>2.52</i>	<i>1.63</i>	<i>0.98</i>	<i>3.77</i>	<i>1.82</i>	<i>1.20</i>	<i>3.67</i>

Table 5.3. 7-fold Cross-Validation Results in NetSurfP dataset (3-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337).

Despite NetSurfP containing a larger and more stringently filtered dataset (10,837 sequences, 2.5 Å resolution cutoff), PISCES achieves consistently higher SOV scores across both SOV'99 (81.02 vs. 76.21) and SOV_{refine} (74.34 vs. 69.39). While NetSurfP yields marginally higher Q3 accuracy, the difference of 0.55 percentage points is negligible and insufficient to compensate for the substantial deficit in segment-level metrics. This discrepancy can be attributed in part to the stricter resolution cutoff used in NetSurfP (2.5 Å versus 3.0 Å for PISCES), which yields sharper and more realistic DSSP boundary assignments. In practice, this makes NetSurfP a harder dataset: the model is exposed to very precise secondary-structure transitions resulting in lower SOV scores, despite similar or slightly higher Q3.

5.2.2 Cross-Validation Results with Post-Processing

Table 5.4 and Table 5.5 report the 5-fold and 7-fold cross-validation results on PISCES and NetSurfP respectively, after applying Random Forest post-processing with a sliding window of 31 residues on top of the three-seed ensemble predictions.

For PISCES, the RF raises mean Q3 accuracy from 86.62% to 86.82% (+0.20 pp): a modest but consistent residue-level improvement. Larger gains appear at the segment level: SOV'99 improves from 81.02 to 83.06 (+2.04) and SOV_{refine} from 74.34 to 76.28 (+1.94). At the per-class level, helix benefits most (SOV'99_H: +2.38). Crucially, the RF also reduces cross-fold variance: SOV'99 standard deviation narrows from 0.54 to 0.44 and SOV_{refine} from 0.60 to 0.47, confirming that segment correction acts as a stabilising step that makes predictions more consistent across folds regardless of which partition is held out.

Fold	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
0	86.70	82.50	75.60	88.90	84.30	75.30	84.20	77.70	66.30
1	86.90	83.50	76.70	89.60	85.50	76.70	85.00	79.10	67.60
2	86.80	83.20	76.50	89.40	84.20	76.70	85.00	77.80	67.90
3	86.70	82.70	76.00	89.00	84.50	75.90	84.30	78.20	67.00
4	87.00	83.40	76.60	89.10	84.80	77.10	84.50	78.30	67.90
Mean	86.82	83.06	76.28	89.20	84.66	76.34	84.60	78.22	67.34
<i>Std</i>	<i>0.13</i>	<i>0.44</i>	<i>0.47</i>	<i>0.29</i>	<i>0.52</i>	<i>0.73</i>	<i>0.38</i>	<i>0.55</i>	<i>0.69</i>

Table 5.4. 5-fold Cross-Validation Results in PISCES dataset (3-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337) and after applying Random Forest post-processing filtering with window size of 31.

For NetSurfP, Q3 improves from 87.17% to 87.41% (+0.24 pp). The segment-level gains are larger than for PISCES: SOV'99 improves from 76.21 to 79.29 (+3.08) and SOV_{refine} from 69.39 to 72.37 (+2.98). The RF's largest gain is in helix, where SOV'99_H rises from 85.16 to 88.59 (+3.43), narrowing the helix gap with PISCES from 1.66 to 0.61 points. Coil remains the most challenging class (SOV'99_C: 70.19), though this represents a +2.98 improvement over the base model. Fold 6 remains a persistent outlier across both the base model and the post-processed pipeline, recording SOV'99 75.00 and SOV_{refine} 68.00 (the lowest of any fold), suggesting its structural atypicality is not correctable by windowed post-processing. The larger overall gains in NetSurfP relative to PISCES are consistent with the higher baseline fragmentation in the uncorrected results: the RF's smoothing mechanism is most beneficial precisely where the base model produces the most uncertain predictions.

Fold	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
0	87.20	80.00	73.20	88.30	84.60	71.80	83.80	78.10	63.20
1	87.60	81.60	74.80	89.80	86.00	73.70	85.30	79.20	65.00
2	87.90	79.80	73.20	89.70	86.40	70.00	85.00	79.80	61.90
3	87.20	79.60	72.40	87.20	86.00	71.40	82.10	78.90	62.80
4	87.10	78.00	70.80	87.80	84.90	68.40	82.60	77.70	59.90
5	87.70	81.00	74.20	88.40	86.00	73.00	83.70	79.50	64.50
6	87.20	75.00	68.00	88.90	84.70	63.00	83.70	77.30	54.90
Mean	87.41	79.29	72.37	88.59	85.51	70.19	83.74	78.64	61.74
<i>Std</i>	<i>0.31</i>	<i>2.21</i>	<i>2.32</i>	<i>0.95</i>	<i>0.75</i>	<i>3.63</i>	<i>1.16</i>	<i>0.95</i>	<i>3.46</i>

Table 5.5. 7-fold Cross-Validation Results in NetSurfP dataset (3-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337) and after applying Random Forest post-processing filtering with window size of 31.

5.2.3 Training Curves

This section presents the training dynamics of the fully trained models used for evaluation on the independent test sets. Unlike the cross-validation phase, where models were trained on subsets of the data, here all available training data from each dataset is used, with an 80/20 random split applied internally to monitor validation loss for early stopping. The total number of epochs and the T_{max} parameter of the cosine annealing scheduler are kept at 50 for 3-state, consistent with the cross-validation configuration, to ensure that the learning rate schedule follows the same annealing trajectory. The early stopping patience is reduced from 10 to 5 epochs to avoid prolonged training on the complete dataset where overfitting risk is lower due to the larger training pool. The saved checkpoint in each case is the one achieving the lowest validation loss across all epochs.

Figures 5.1-5.4 show the training and validation loss and accuracy curves for the PISCES and NetSurfP fully trained models across the three seeds (42, 123, 1337) respectively.

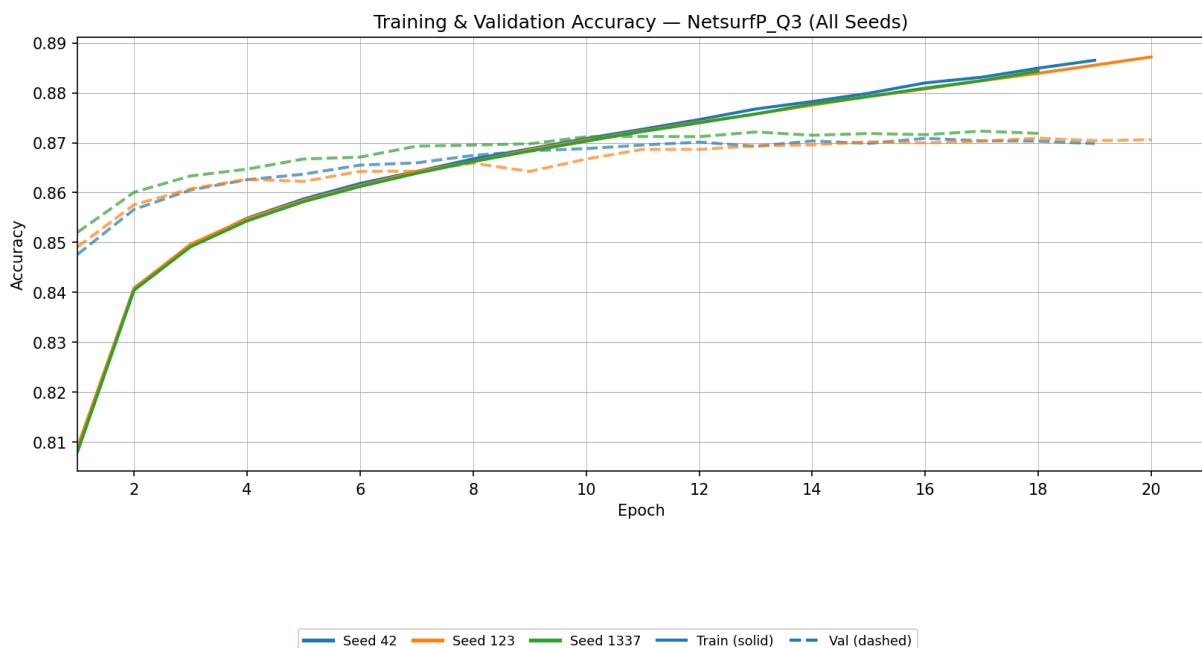


Figure 5.1. Training and Validation $Q3$ accuracies of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.

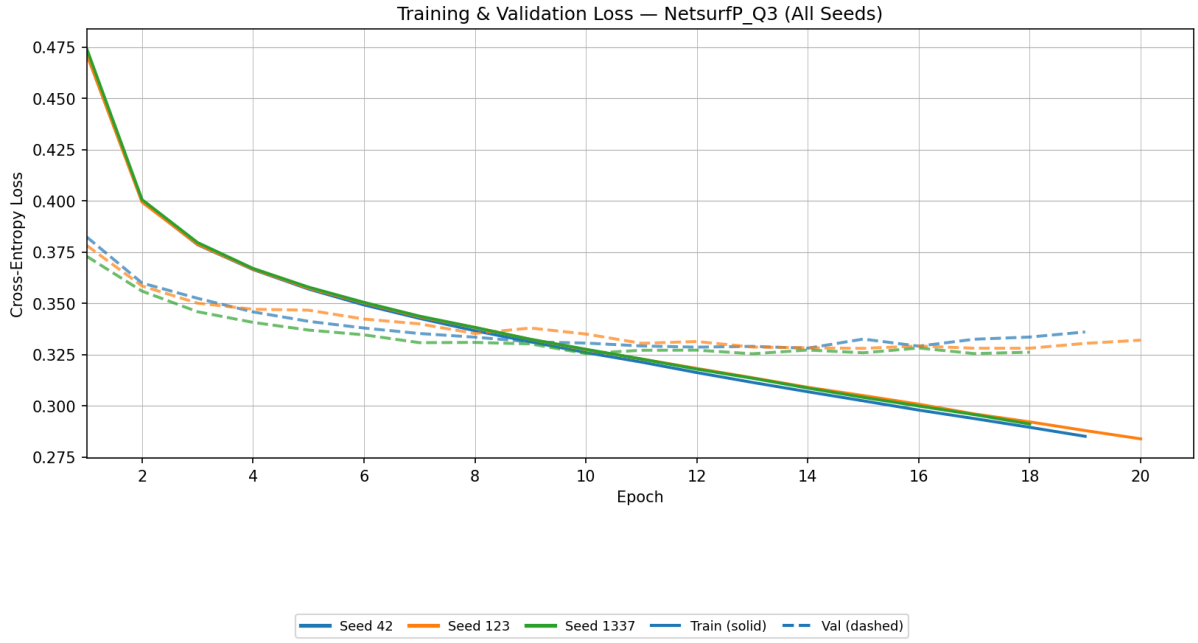


Figure 5.2. Training and Validation Cross-Entropy losses on 3-State of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.

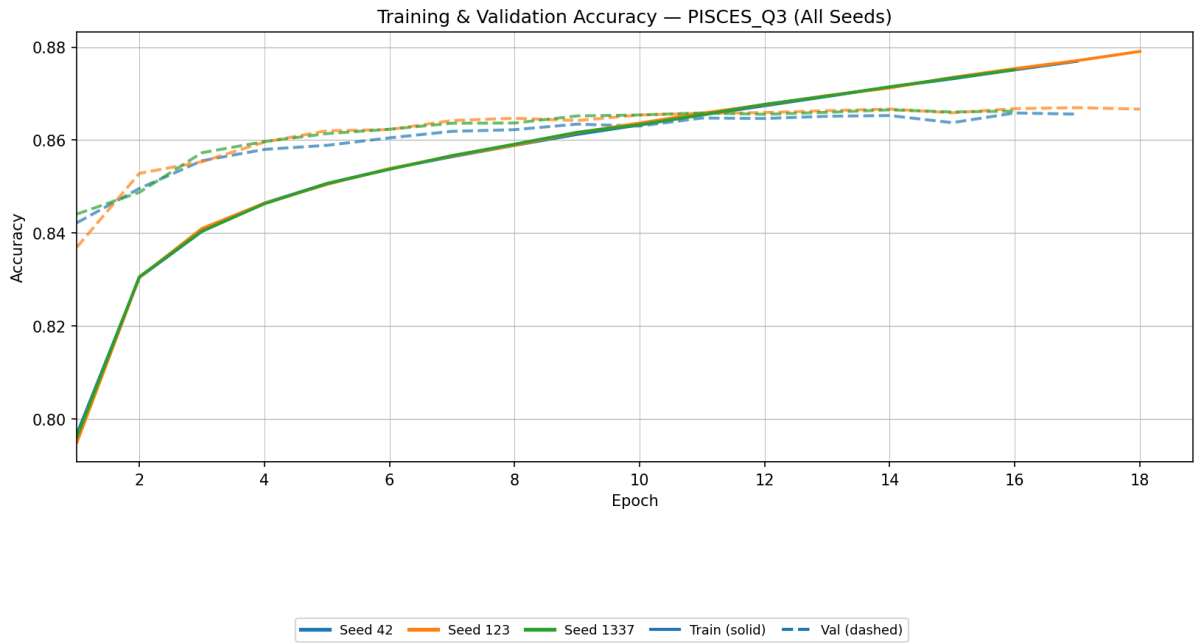


Figure 5.3. Training and Validation Q3 accuracies of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the PISCES dataset.



Figure 5.4. Training and Validation Cross-Entropy losses on 3-State of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the PISCES dataset.

In both cases, the cosine annealing schedule produces a gradual, monotonic decay without sharp oscillations and early stopping triggers consistently in the final training epochs, suggesting the models reach near-optimal generalisation without significant overfitting. The accuracy curves plateau well before early stopping fires, confirming that the patience reduction does not truncate training prematurely.

5.2.4 Evaluation Results in 3-State

Tables 5.6–5.9 report the evaluation results on CASP13, CB513, TS115 and CASP12 respectively, for all filtering pipeline variants. The base ensemble is progressively combined with the Random Forest post-processor and the external rules in two orderings: ER→RF and RF→ER, to assess the effect of both individual components and their combination, as well as whether application order matters. The script responsible for the training and evaluation of the model is in Appendix IX.

Across CASP13, CB513 and TS115, the Ensemble + RF + ER configuration consistently delivers the best balance of SOV and Q3 performance. On CB513 it improves SOV'99 by +3.40 and SOV_{refine} by +3.20 with only +0.10 pp Q3 gain. On TS115 it yields SOV'99 +4.00 and SOV'99_C +4.80 while Q3 remains unchanged at 87.20%. CASP13 shows more modest but consistent gains (+2.60 SOV'99, +2.20 SOV_{refine}, no Q3 change), with helix benefiting most across all three benchmarks (+3.70 on CB513, +3.20 on TS115, +2.80 on CASP13 in SOV'99_H), consistent with helix segments being long and regular enough for the RF to reliably extend and clean their boundaries. The two application orders produce near-identical results differing by at most 0.9 SOV points, confirming the two components act on complementary error types: the RF corrects probabilistically uncertain segment boundaries while the ER eliminates structurally implausible short segments and neither undoes the other's corrections. This complementarity, combined with the consistency of gains across all three benchmarks, is what motivates selecting Ensemble + RF + ER as the reference pipeline.

Model	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
Seed Ensemble	84.60	76.00	68.10	84.70	79.90	67.80	78.10	72.70	58.40
Ensemble + RF	84.80	78.60	70.40	87.80	81.80	70.10	80.90	74.30	60.70
Ensemble + ER	84.30	78.00	69.80	86.10	81.00	70.10	79.60	73.50	60.30
Ensemble + ER + RF	84.20	77.90	69.60	85.90	82.10	69.80	79.30	74.50	59.80
Ensemble + RF + ER	84.60	78.60	70.30	87.50	81.70	70.20	80.70	74.10	60.40

Table 5.6. Evaluation Results in CASP13 after combining multiple filtering methods
(3-State Classification and RF window size =31).

Model	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
Seed Ensemble	86.80	76.70	70.10	86.60	80.40	68.60	82.70	73.80	60.30
Ensemble + RF	87.00	79.10	72.50	90.30	83.40	70.20	86.10	76.70	62.00
Ensemble + ER	86.60	79.40	72.50	89.20	81.60	71.50	85.20	74.60	62.80
Ensemble + ER + RF	86.60	79.20	72.40	89.10	82.60	70.90	85.10	75.80	62.30
Ensemble + RF + ER	86.90	80.10	73.30	90.30	83.40	71.80	86.30	76.40	63.10

Table 5.7. Evaluation Results in CB513 after combining multiple filtering methods
(3-State Classification and RF window size =31).

Model	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
Seed Ensemble	87.20	79.20	71.80	85.90	82.60	72.50	81.00	75.10	63.10
Ensemble + RF	87.30	81.40	73.90	88.20	85.30	74.30	83.00	77.90	65.00
Ensemble + ER	87.10	83.20	75.60	88.90	83.70	78.10	83.90	75.90	68.20
Ensemble + ER + RF	87.10	82.90	75.30	88.90	84.50	77.30	83.70	77.00	67.40
Ensemble + RF + ER	87.20	83.20	75.60	89.10	85.10	77.30	83.90	77.60	67.70

Table 5.8. Evaluation Results in TS115 after combining multiple filtering methods
(3-State Classification and RF window size =31).

CASP12 stands out as the most challenging benchmark. The base ensemble achieves only SOV'99 54.00 and SOV_{refine} 46.70 (substantially lower than on any other benchmark) reflecting the structural novelty of FM targets. Post-processing delivers its largest absolute gains here: Ensemble + RF + ER improves SOV'99 by +10.00 and SOV_{refine} by +8.70, with SOV'99_C improving by +11.30, driven primarily by the ER's elimination of short spurious helix and strand insertions in the predominantly coil-rich FM targets. ER+RF marginally leads RF+ER in SOV'99 (64.20 vs. 64.00) (a difference of 0.20 points that is negligible in practice) confirming that the two orderings remain largely interchangeable even on the hardest benchmark and that the overall gains on CASP12 are driven by the complementary action of both components rather than their order.

Model	Accuracy Q3 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)	SOV'99 _H	SOV'99 _E	SOV'99 _C	SOV _{ref,H}	SOV _{ref,E}	SOV _{ref,C}
Seed Ensemble	81.40	54.00	46.70	80.10	76.80	41.00	73.00	67.20	34.00
Ensemble + RF	81.30	62.00	53.70	82.90	77.70	50.00	75.70	67.70	41.70
Ensemble + ER	81.90	63.30	54.70	82.80	77.50	51.80	75.60	67.40	43.10
Ensemble + ER + RF	81.90	64.20	55.70	81.70	78.00	53.20	74.50	68.10	44.50
Ensemble + RF + ER	81.40	64.00	55.40	83.40	78.10	52.30	76.30	67.80	43.60

*Table 5.9. Evaluation Results in CASP12 after combining multiple filtering methods
(3-State Classification and RF window size =31).*

Figures 5.5–5.8 display the confusion matrices for 3-state secondary structure classification across the four independent benchmark datasets (CASP13, CB513, TS115 and CASP12) obtained after applying the RF→ER post-processing pipeline. Each matrix is a 3×3 grid whose rows correspond to the true DSSP-derived secondary structure class (Helix H, Strand E, Coil C) and whose columns correspond to the predicted class.

The most consistent observation is that direct confusion between helix (H) and strand (E) is almost non-existent. The corresponding off-diagonal cells remain negligible on every benchmark, with only 15 and 34 misclassified residues on CASP13, 157 and 162 on CB513, 46 and 47 on TS115 and 23 and 12 on CASP12, in each case well below 1% of the respective class. This reflects the strong geometric and hydrogen-bonding distinction between the two regular structures, which the frozen ESM-2 representation evidently encodes, so that the model rarely commits the kind of gross structural misassignment that would be most damaging for downstream use.

Instead, almost all residue-level errors are routed through the coil (C) class. For both helix and strand, the dominant error is misassignment to coil; on CB513, for example, 3,586 of the 3,743 helix errors and 3,923 of the 4,085 strand errors fall into coil, while the coil row distributes its own errors roughly evenly between helix and strand (3,060 against 3,093 on CB513 and 995 against 729 on TS115). Coil therefore behaves as an ambiguity sink, with the residual mistakes concentrated at the transitions between regular segments and their flanking coil regions, exactly where the experimental DSSP assignment is itself least clear-cut. This is precisely the error mode that the segment-level post-processing is designed to address and explains why the RF→ER pipeline improves SOV while leaving Q3 essentially unchanged: it refines segment boundaries rather than reclassifying residues between the two regular structural classes.

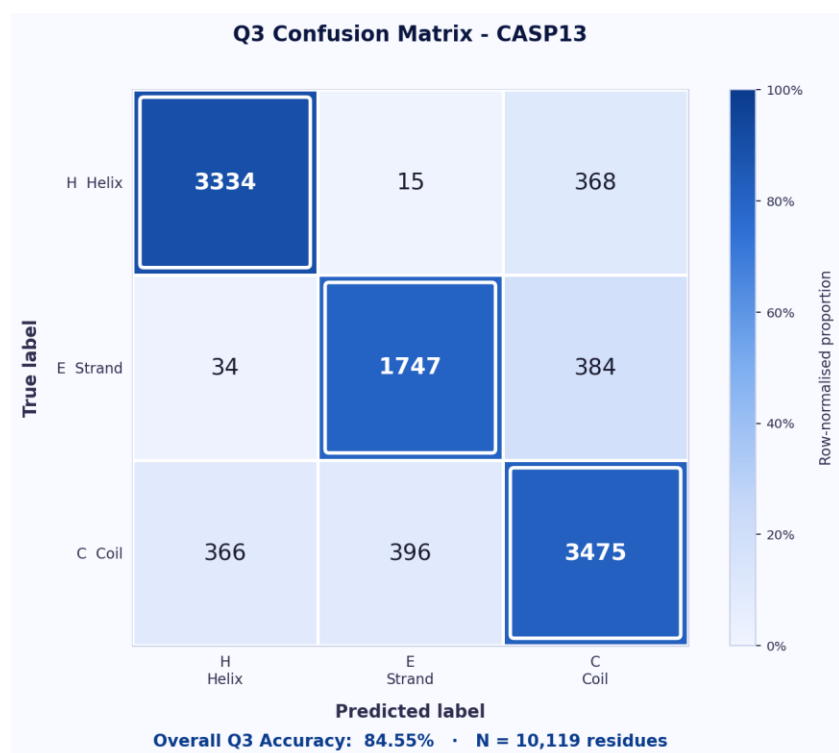


Figure 5.5. Confusion Matrix for 3-State Classification in CASP13 after applying RF→ER ordering.

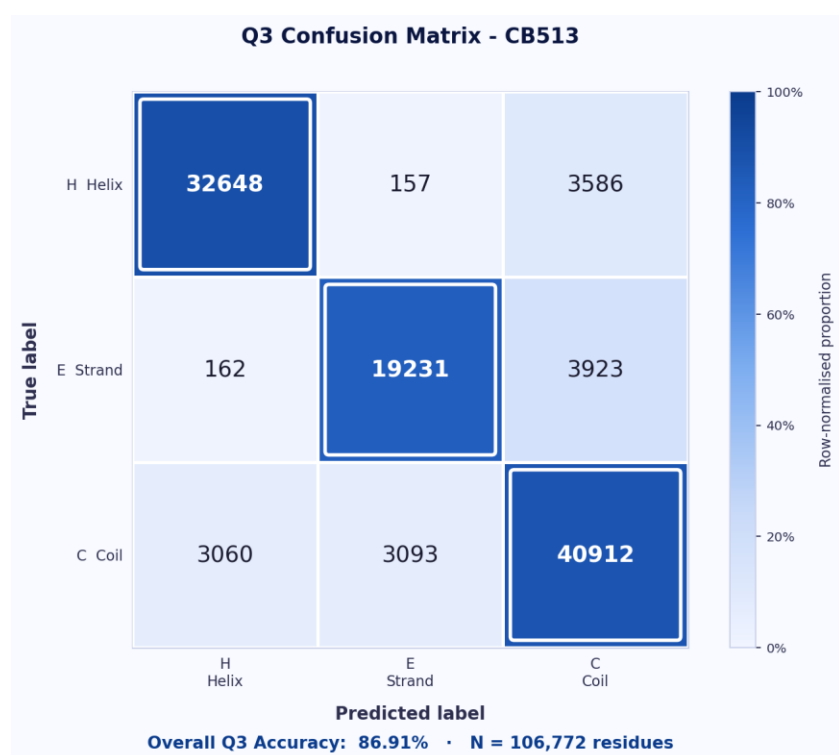


Figure 5.6. Confusion Matrix for 3-State Classification in CB513 after applying RF→ER ordering.

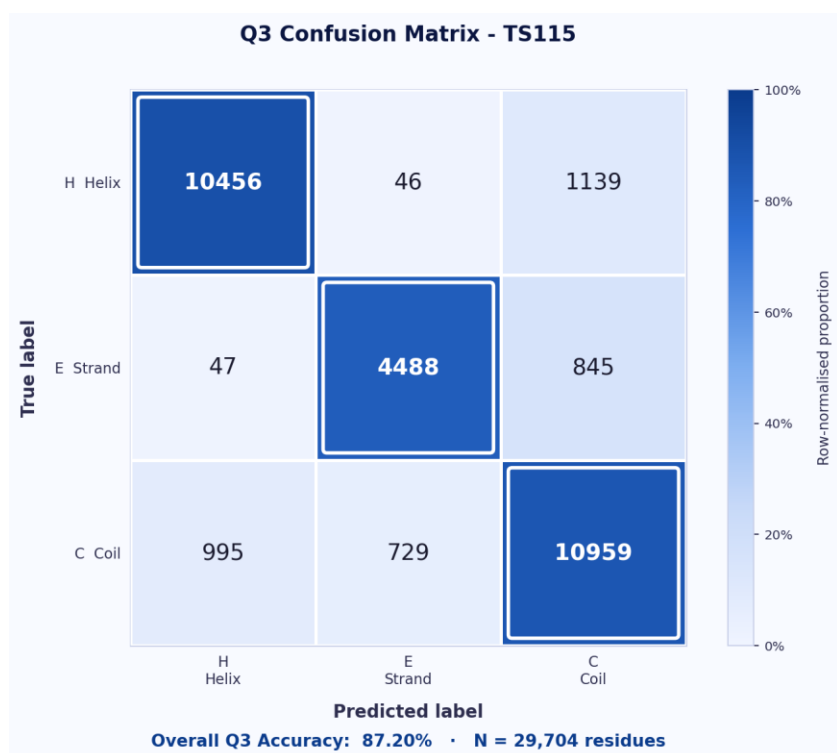


Figure 5.7. Confusion Matrix for 3-State Classification in TS115 after applying RF→ER ordering.

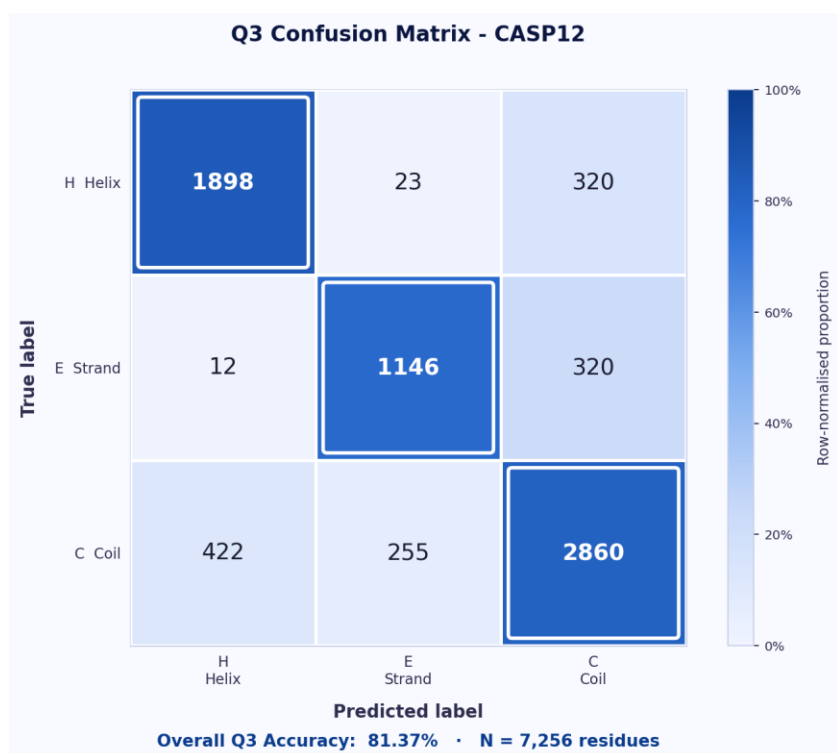


Figure 5.8. Confusion Matrix for 3-State Classification in CASP12 after applying RF→ER ordering.

5.2.5 Window Size Experiment in 3-State

This section examines the effectiveness of increasing the post-processing window size in 3-State classification. Furthermore, it analyses whether the input format of the post-processor plays a crucial role. Chatzimiltis et al. [132] used hard labels as input, keeping the feature dimension the same as the window size (e.g. for window size 31, there are 31 features with a class index at each position) while Kountouris et al. [127] use the output probabilities, increasing the input dimension by the number of states (e.g. for window size 31, we have $31 \times C$ features where C are all the classes). As discussed in Chapter 4, probabilities provide uncertainty which by definition hard labels lack, making them more informative.

Figure 5.9 reports the average Q3 accuracy across CASP13, CB513, TS115 and CASP12 for multiple window sizes for both when probabilities or hard labels are used to the post-processor. Similarly, Figure 5.10 reports the average SOV'99 across CASP13, CB513, TS115 and CASP12 for multiple window sizes for both when probabilities or hard labels are used to the post-processor.

It should be noted that CASP13 results are produced by the PISCES-trained base model while the remaining three benchmarks use the NetSurfP-trained base model. The RF post-processor operates solely on the output probability distributions of the base model, with no access to or dependence on the training provenance of those probabilities. The choice of base model training set (PISCES vs NetSurfP) is therefore architecturally irrelevant to what this experiment measures. Whether different training data produces different probability distributions is a question about base model quality, not about the RF's window size sensitivity; investigating whether PISCES or NetSurfP produces better probability estimates is a separate experimental question outside the scope of this analysis.

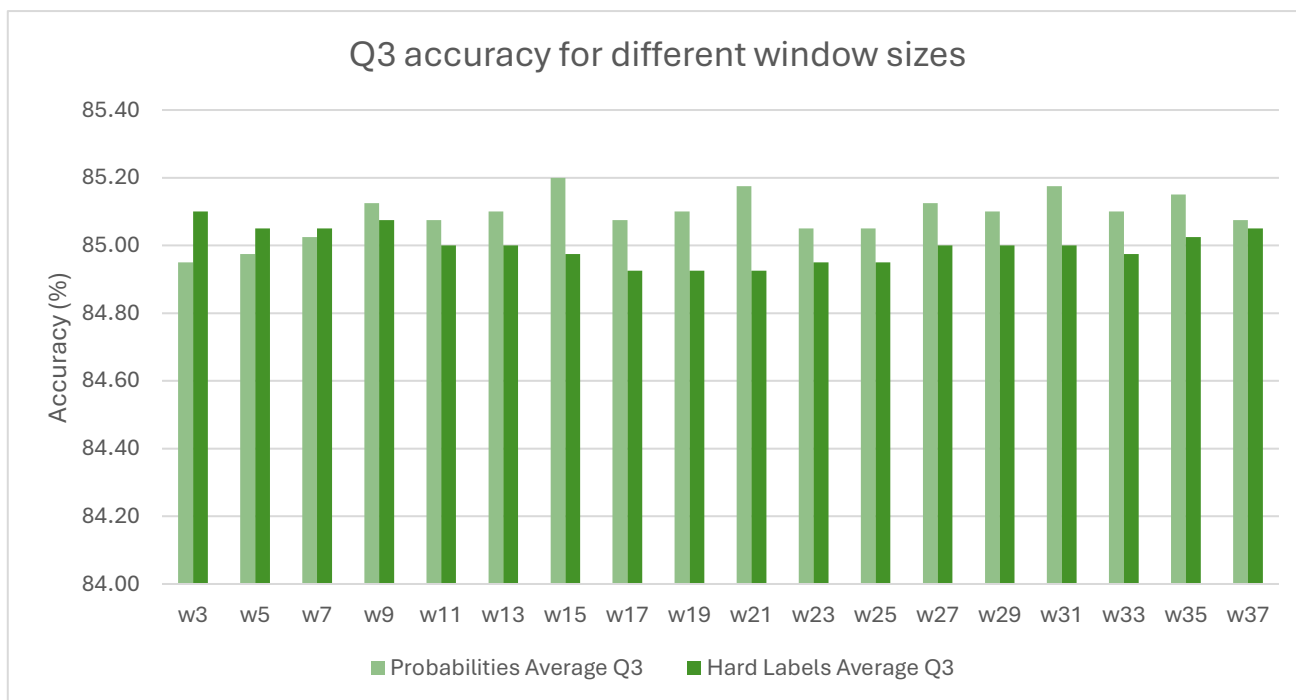


Figure 5.9. Average Q3 across all benchmarks (CASPI2, TS115, CB513 and CASPI3), testing multiple window sizes.

For each window both input formats are compared: probabilities or hard labels.

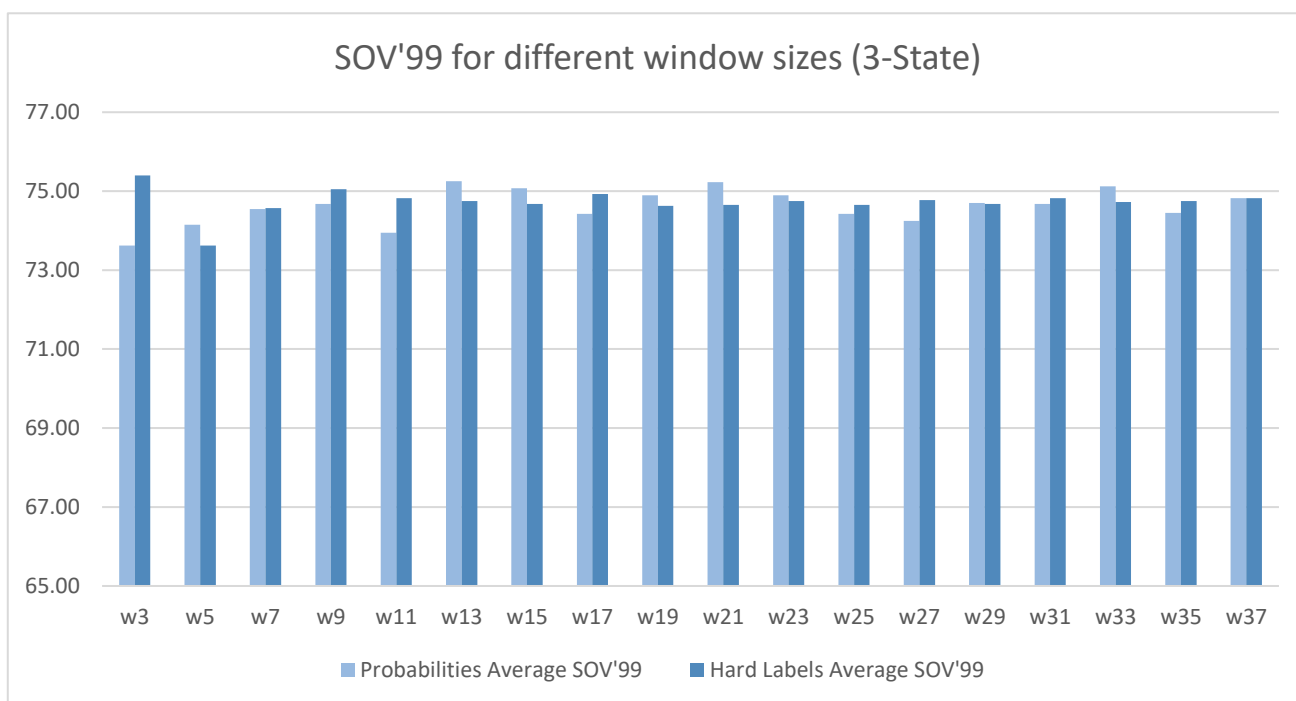


Figure 5.10. Average SOV'99 across all benchmarks (CASPI2, TS115, CB513 and CASPI3), testing multiple window sizes.

For each window both input formats are compared: probabilities or hard labels.

Examining Figure 5.9, per-residue accuracy remains largely stable across all tested window sizes regardless of input format, with differences between configurations confined to less than 0.1 percentage points in most cases. This contrasts with the monotonic increase in Q3 accuracy reported by Chatzimiltis et al. [132] as window size grows, which is not reproduced here. A likely explanation is that the base model's ESM-2 embeddings already encode rich contextual information across the full sequence, leaving little residual positional context for the sliding window to exploit at the residue level.

Turning to Figure 5.10, the probability-based input yields small but consistent improvements in SOV'99 over hard labels in most window size configurations, supporting the expectation that soft distributions provide a richer signal for segment-level refinement. However, the magnitude of these gains remains modest and does not grow systematically with window size. SOV'99 is notably more sensitive to input format than Q3 accuracy, suggesting that the additional uncertainty information carried by probabilities is primarily useful for correcting segment boundaries rather than reclassifying individual residues. Overall, neither increasing the window size nor switching input format produces the clear, monotonic gains reported by Chatzimiltis et al. [132], indicating that the dominant source of performance in this pipeline lies in the base model rather than the post-processing configuration.

These findings suggest that, for 3-state classification, a smaller window size could have been adopted without meaningful loss in either Q3 accuracy or SOV'99 and keeping the boost a post-processing algorithm gives. Similarly, hard labels would have served as a computationally cheaper and equally effective input representation, since the base model's strong probabilistic outputs do not translate into a meaningful advantage at the segment level in this setting. The window size of 31 and probability inputs are retained throughout this work for consistency with prior work in this lineage and to enable direct comparability, but the present results indicate that these choices are not critical to performance in the 3-state case.

5.3 Results in 8-State

5.3.1 Cross-Validation Results

Table 5.10 reports the overall Q8 accuracy, SOV'99 and SOV_{refine} across the 7-fold cross-validation on NetSurfP, while Table 5.11 and Table 5.12 provide the per-class SOV'99 and SOV_{refine} breakdowns respectively. Again, using an ensemble of three models trained with different random seeds (42, 123 and 1337). Implementation is in Appendix VIII.

The ensemble achieves a mean Q8 accuracy of 77.41% with a standard deviation of 0.61 and mean SOV'99 and SOV_{refine} scores of 68.04 and 67.06 respectively. The notably higher standard deviations in both SOV metrics (2.66 and 2.97) relative to the accuracy standard deviation (0.61) indicate that fold-to-fold variation is primarily at the segment level rather than the residue level: a pattern already observed in the Q3 NetSurfP results. Fold 6 is the consistent outlier across all three metrics, recording the lowest accuracy (76.90%), SOV'99 (63.30) and SOV_{refine} (61.80), suggesting an atypically difficult held-out partition.

Fold	Accuracy Q8 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)
0	77.00	69.00	68.50
1	77.70	70.70	70.00
2	78.30	69.50	68.60
3	77.10	67.80	66.60
4	76.80	65.80	64.50
5	78.10	70.20	69.40
6	76.90	63.30	61.80
Mean	77.41	68.04	67.06
<i>Std</i>	<i>0.61</i>	<i>2.66</i>	<i>2.97</i>

Table 5.10. 7-fold Cross-Validation Results on Q8, SOV'99 and SOV_{refine} in NetSurfP dataset (8-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337).

Turning to the comparison against Q3 NetSurfP, the transition to 8-state prediction incurred reductions of 9.76 points in accuracy ($87.17\% \rightarrow 77.41\%$) and 8.17 points in SOV'99 ($76.21 \rightarrow 68.04$), yet $\text{SOV}_{\text{refine}}$ dropped by only 2.33 points ($69.39 \rightarrow 67.06$). The small degradation in $\text{SOV}_{\text{refine}}$ relative to the much larger drops in Q3 accuracy and SOV'99 suggests that the dominant error mode in 8-state prediction consists of within-family label confusions, such as misclassifying α -helices (H) as 3_{10} -helices (G), or turns (T) as bends (S), rather than a fundamental breakdown in segment boundary localisation. Since these substituted classes share similar structural neighbourhoods, $\text{SOV}_{\text{refine}}$'s monotonically growing allowance function appropriately credits partial segment overlap, whereas SOV'99 penalises such errors more harshly by assigning zero allowance to non-overlapping segments. This pattern indicates that the model has successfully learned the geometry of secondary structure transitions but requires greater discriminative capacity to resolve the finer distinctions that define the 8-state scheme.

Fold	SOV'99 _G	SOV'99 _H	SOV'99 _I	SOV'99 _B	SOV'99 _E	SOV'99 _S	SOV'99 _T	SOV'99 _C
0	36.20	88.00	32.40	8.60	86.50	34.80	60.40	53.90
1	41.10	90.50	30.10	13.00	86.30	34.40	60.40	56.90
2	40.50	90.20	34.70	8.80	87.80	38.20	61.50	52.10
3	37.50	84.80	31.20	7.80	86.80	34.80	59.50	54.10
4	38.80	87.30	26.30	6.40	85.50	33.60	58.60	48.50
5	39.40	87.60	31.60	9.50	87.80	37.70	59.00	55.90
6	34.30	88.20	32.20	8.00	85.10	33.30	59.50	43.50
Mean	38.26	88.09	31.21	8.87	86.54	35.26	59.84	52.13
<i>Std</i>	2.42	1.91	2.58	2.06	1.04	1.93	0.99	4.68

Table 5.11. Detailed 7-fold Cross-Validation Results on SOV'99 for each state in NetSurfP dataset (8-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337).

Fold	SOV _{ref,G}	SOV _{ref,H}	SOV _{ref,I}	SOV _{ref,B}	SOV _{ref,E}	SOV _{ref,S}	SOV _{ref,T}	SOV _{ref,C}
0	38.00	87.00	33.40	8.70	85.50	37.10	62.20	52.60
1	42.40	89.40	31.00	13.20	85.10	36.80	61.90	55.40
2	41.60	88.90	35.70	8.90	86.30	40.20	62.90	50.60
3	38.60	83.30	31.80	7.90	85.00	36.60	60.70	52.00
4	39.60	85.50	26.60	6.50	83.50	35.50	59.60	46.50
5	40.80	86.30	32.30	9.60	86.40	39.90	60.60	54.40
6	34.90	86.20	31.90	8.00	82.80	34.80	60.10	41.40
Mean	39.41	86.66	31.81	8.97	84.94	37.27	61.14	50.41
Std	2.54	2.07	2.76	2.10	1.35	2.06	1.21	4.91

Table 5.12. Detailed 7-fold Cross-Validation Results on SOV_{refine} for each state in NetSurfP dataset (8-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337).

The per-class results reveal a pronounced stratification of prediction difficulty across the eight DSSP states. α -helix (H) is by far the best-predicted class, achieving mean SOV'99_H of 88.09 and SOV_{ref,H} of 86.66, consistent with its regularity and prevalence in training data. Extended strand (E) follows closely at SOV'99_E = 86.54 and SOV_{ref,E} = 84.94, benefiting from similarly regular hydrogen bonding geometry.

At the other extreme, β -bridge (B) is the most poorly predicted state, with mean SOV'99_B of just 8.87 and SOV_{ref,B} of 8.97 and standard deviations of approximately 2.1 in both metrics. This near-zero performance is expected: isolated β -bridges are single-residue states by DSSP definition, meaning there is no segment to overlap and no allowance to accumulate, making SOV scores structurally uninformative for this class. Similarly, 3_{10} -helix (G) achieves 38.26 and π -helix (I) achieves 31.21, both reflecting their rarity in training data and short typical segment lengths which limit the accumulation of overlap allowance.

Coil (C) exhibits the highest instability, with $\text{SOV}'99_C = 52.13 \pm 4.68$ and $\text{SOV}_{\text{ref}, C} = 50.41 \pm 4.91$, the largest standard deviations of any class. This mirrors the Q3 findings and reinforces that coil, as a structurally heterogeneous catch-all category in DSSP, is inherently the hardest class to predict in a segment-coherent manner regardless of the number of states used. Turn (T) at 59.84 and bend (S) at 35.26 complete the middle tier, with both classes suffering from short segment lengths and frequent confusion with one another due to their overlapping structural definitions in DSSP.

5.3.2 Cross-Validation Results with Post-Processing Filtering

Table 5.13 reports the 7-fold cross-validation results on NetSurfP for 8-state classification after applying Random Forest post-processing with a window of 31 residues. The RF improves segment-level metrics while producing a small regression in Q8 accuracy: $\text{SOV}'99$ rises by 1.92 points and $\text{SOV}_{\text{refine}}$ by 2.01, whereas Q8 drops by 0.71 pp. This pattern contrasts with the 3-state case, where residue-level accuracy remains essentially unchanged. In 8-state space, the post-processor sometimes “over-corrects” within the helix family: it may convert conservative α -helix (H) predictions into 3_{10} -helix (G) or π -helix (I) to enforce locally consistent segments. While this improves segment continuity and SOV, it can introduce new H/G/I mismatches relative to the DSSP labels, slightly reducing 8-state residue-level accuracy without affecting 3-state Q3.

Fold	Accuracy Q8 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)
0	76.10	70.60	70.20
1	76.90	71.90	71.30
2	77.60	71.00	70.20
3	76.20	70.30	69.20
4	76.10	68.10	66.90
5	77.40	72.50	71.80
6	76.60	65.30	63.90
Mean	76.70	69.96	69.07
<i>Std</i>	<i>0.62</i>	<i>2.48</i>	<i>2.78</i>

Table 5.13. 7-fold Cross-Validation Results on Q8, SOV'99 and SOV_{refine} in NetSurfP dataset (8-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337) and after applying Random Forest post-processing filtering with window size of 31.

Table 5.14 and Table 5.15 break down the per-state SOV'99 and SOV_{refine} gains respectively. The RF delivers larger improvements on the minority classes, which are chronically under-predicted due to severe class imbalance: SOV'99_I improves by 21.69 points, SOV'99_G by 16.65, SOV'99_B by 12.93 and SOV'99_S by 12.40. These classes together represent a small percentage of all residues, meaning they have negligible impact on overall SOV metrics, yet they are structurally significant. The two dominant classes, H and E, improve more modestly (SOV'99_H: +2.18, SOV'99_E: +1.76), consistent with the base model already producing stable predictions for them. Coil is the only class that regresses (SOV'99_C: -2.20), a predictable counterpart to the minority class recovery: coil regions bordering extended G and I segments are partially absorbed during propagation, representing an acceptable trade-off given the substantially larger gains in the structurally informative helix sub-classes. Standard deviations remain high for coil (4.31 for SOV'99, 4.63 for SOV_{refine}), confirming that fold-level instability in this class is structural rather than correctable by windowed post-processing.

Fold	SOV'99 _G	SOV'99 _H	SOV'99 _I	SOV'99 _B	SOV'99 _E	SOV'99 _S	SOV'99 _T	SOV'99 _C
0	54.30	89.90	51.10	21.50	87.90	47.20	62.30	51.00
1	55.70	91.00	49.10	23.40	88.40	48.00	62.50	53.50
2	57.10	91.00	57.80	21.30	89.60	49.30	64.00	50.00
3	53.30	89.00	51.70	20.50	88.90	47.00	62.70	52.10
4	54.70	89.50	52.90	22.10	87.20	46.60	62.30	47.30
5	55.30	90.90	52.40	21.00	88.90	48.60	63.10	54.00
6	54.00	90.60	55.30	22.80	87.20	46.90	63.60	41.60
Mean	54.91	90.27	52.90	21.80	88.30	47.66	62.93	49.93
<i>Std</i>	<i>1.25</i>	<i>0.81</i>	<i>2.86</i>	<i>1.03</i>	<i>0.91</i>	<i>1.00</i>	<i>0.67</i>	<i>4.31</i>

Table 5.14. Detailed 7-fold Cross-Validation Results on SOV'99 for each state in NetSurfP dataset (8-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337) and after applying Random Forest post-processing filtering with window size of 31.

Fold	SOV _{ref,G}	SOV _{ref,H}	SOV _{ref,I}	SOV _{ref,B}	SOV _{ref,E}	SOV _{ref,S}	SOV _{ref,T}	SOV _{ref,C}
0	54.20	88.90	50.30	21.80	86.60	48.60	63.30	51.00
1	55.50	89.90	48.70	23.80	86.90	49.60	63.20	53.20
2	56.30	89.80	56.90	21.60	87.90	50.20	64.50	49.30
3	52.70	87.40	49.90	20.90	86.90	47.70	63.00	51.20
4	53.80	87.80	51.30	22.50	85.10	47.40	62.40	46.30
5	55.00	89.70	51.60	21.30	87.40	49.70	63.70	53.60
6	52.80	88.80	53.20	23.10	84.70	47.50	63.30	40.40
Mean	54.33	88.90	51.70	22.14	86.50	48.67	63.34	49.29
<i>Std</i>	<i>1.35</i>	<i>0.99</i>	<i>2.70</i>	<i>1.04</i>	<i>1.18</i>	<i>1.17</i>	<i>0.65</i>	<i>4.63</i>

Table 5.15. Detailed 7-fold Cross-Validation Results on SOV_{refine} for each state in NetSurfP dataset (8-State Classification), using an ensemble of three base-models with different weight initializations (seeds 42, 123 and 1337) and after applying Random Forest post-processing filtering with window size of 31.

5.3.3 Training Curves

This section presents the training dynamics of the fully trained models used for evaluation on the independent test sets in 8-state classification. As in Section 5.2.3, all available NetSurfP training data is used with an 80/20 split to monitor validation loss for early stopping. The total number of epochs and the T_{max} parameter of the cosine annealing scheduler are set to 60, consistent with the cross-validation configuration for the Q8 task, reflecting the increased complexity of 8-state classification which requires a longer annealing trajectory to reach stable convergence. The early stopping patience is reduced from 15 to 10 epochs and the saved checkpoint is the one achieving the lowest validation loss across all epochs.

For the 8-state models, the longer training horizon of 60 epochs allows the loss and accuracy to converge more slowly than in the 3-state case, reflecting the greater difficulty of Q8 classification. As in the 3-state setting, the early stopping mechanism is sufficient to halt training once validation performance plateaus, preventing unnecessary overfitting while still capturing the point of best generalisation. Figure 5.11 and Figure 5.12 show the training and validation loss and accuracy curves for the NetSurfP Q8 fully trained models across the three seeds (42, 123, 1337).

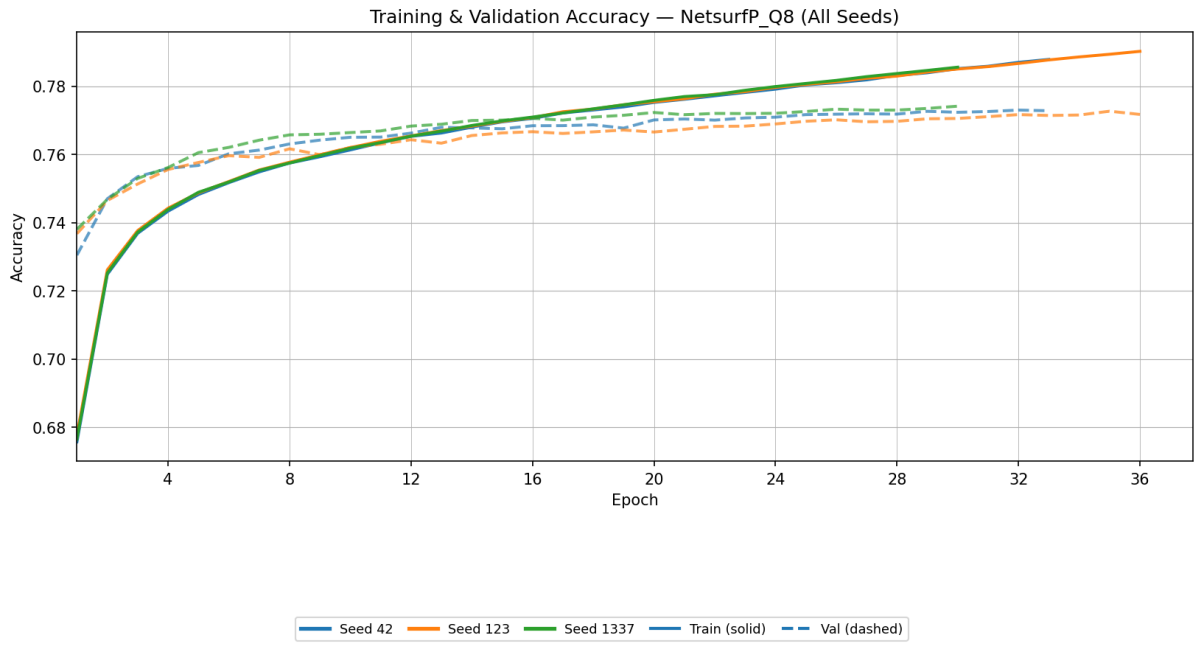


Figure 5.11. Training and Validation $Q8$ accuracies of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.

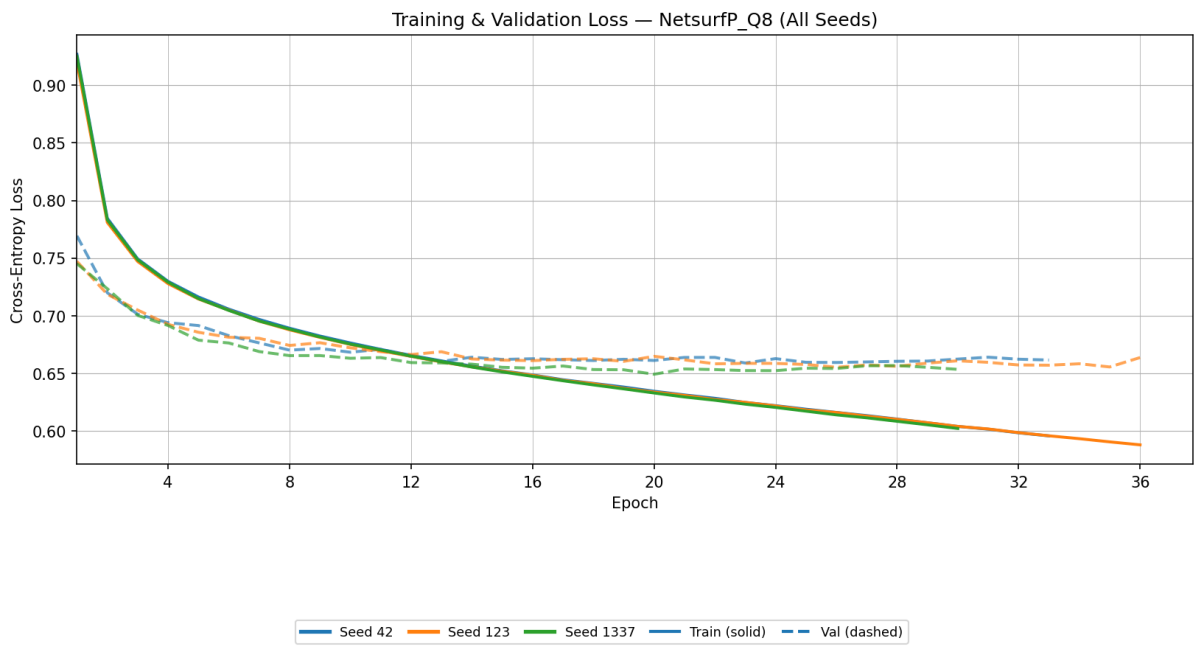


Figure 5.12. Training and Validation Cross-Entropy losses on 8-State of each of the three-models with randomly initialized weights (42, 123 and 1337), when trained on the 80/20 split of the NetSurfP dataset.

5.3.4 Evaluation Results in 8-State

Tables 5.16–5.24 report 8-state (Q8) and segment-level performance on CB513, TS115 and CASP12 before and after applying the RF post-processor. External rules were not applied for 8-State classification as justified in Section 4.5.2. On CB513, the RF slightly reduces Q8 accuracy from 75.10% to 74.60% but improves SOV'99 from 67.40 to 68.90 and SOV_{refine} from 67.80 to 69.40. Per-class SOV scores reveal that these gains are concentrated in the helix subclasses and minority states: SOV'99_G increases from 38.20 to 57.50, SOV'99_I from 32.80 to 54.40 and SOV'99_B from 7.60 to 20.90, with similar trends in SOV_{refine}. This indicates that the RF tends to balance short 3_{10} - and π -helix segments and rare β -bridges, improving segment continuity even though some within-family label substitutions (e.g. H $\rightarrow$ G/I) introduce additional 8-state residue-level errors.

Model	Accuracy Q8 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)
Seed Ensemble	75.10	67.40	67.80
Ensemble + RF	74.60	68.90	69.40

Table 5.16. Q8, SOV'99 and SOV_{refine} Results in CB513 after adding post-processing filtering (8-State Classification and RF window size =31).

Model	SOV'99 _G	SOV'99 _H	SOV'99 _I	SOV'99 _B	SOV'99 _E	SOV'99 _S	SOV'99 _T	SOV'99 _C
Seed Ensemble	38.20	90.80	32.80	7.60	84.60	32.80	60.30	51.30
Ensemble + RF	57.50	91.70	54.40	20.90	87.00	46.50	60.90	47.60

Table 5.17. Detailed SOV'99 Results in CB513 after adding post-processing filtering (8-State Classification and RF window size =31).

Model	SOV _{ref,G}	SOV _{ref,H}	SOV _{ref,I}	SOV _{ref,B}	SOV _{ref,E}	SOV _{ref,S}	SOV _{ref,T}	SOV _{ref,C}
Seed Ensemble	40.80	90.20	35.10	7.70	84.40	35.80	63.00	51.10
Ensemble + RF	58.50	91.20	54.80	21.50	86.50	48.90	62.60	48.90

Table 5.18. Detailed SOV_{refine} Results in CB513 after adding post-processing filtering (8-State Classification and RF window size =31).

On TS115, Q8 decreases marginally from 77.80% to 77.10%, but SOV'99 rises from 70.80 to 73.10 and SOV_{refine} from 69.50 to 72.00. As on CB513, the largest per-class improvements appear in the non-H canonical helix states and β -bridge: SOV'99_G increases from 33.30 to 50.40, SOV'99_I from 45.10 to 61.70 and SOV'99_B from 7.60 to 18.10, with SOV_{refine} showing closely matching gains. Helix (H) and strand (E) segments also benefit modestly (e.g. SOV'99_H 88.00→90.50, SOV'99_E 86.30→88.60), confirming that the post-processor improves segment coherence without materially altering the overall Q8 level.

Model	Accuracy Q8 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)
Seed Ensemble	77.80	70.80	69.50
Ensemble + RF	77.10	73.10	72.00

Table 5.19. Q8, SOV'99 and SOV_{refine} Results in TS115 after adding post-processing filtering

(8-State Classification and RF window size =31).

Model	SOV'99 _G	SOV'99 _H	SOV'99 _I	SOV'99 _B	SOV'99 _E	SOV'99 _S	SOV'99 _T	SOV'99 _C
Seed Ensemble	33.30	88.00	45.10	7.60	86.30	34.60	59.70	60.10
Ensemble + RF	50.40	90.50	61.70	18.10	88.60	46.10	61.60	58.70

Table 5.20. Detailed SOV'99 Results in TS115 after adding post-processing filtering

(8-State Classification and RF window size =31).

Model	SOV _{ref,G}	SOV _{ref,H}	SOV _{ref,I}	SOV _{ref,B}	SOV _{ref,E}	SOV _{ref,S}	SOV _{ref,T}	SOV _{ref,C}
Seed Ensemble	34.50	86.40	44.80	7.80	84.30	36.60	60.60	57.90
Ensemble + RF	49.50	89.10	60.10	18.20	86.40	47.00	61.90	57.60

Table 5.21. Detailed SOV_{refine} Results in TS115 after adding post-processing filtering

(8-State Classification and RF window size =31).

On CASP12, the RF has the strongest relative effect at the segment level: Q8 decreases slightly from 70.20% to 69.60%, but SOV'99 increases from 42.80 to 51.10 and SOV_{refine} from 40.50

to 48.50. The per-class SOV tables show particularly large gains in 3_{10} -helix, π -helix and coil: SOV'99_G grows from 30.80 to 52.90, SOV'99_I from 43.20 to 71.30 and SOV'99_C from 23.50 to 30.20, with corresponding SOV_{refine} improvements (e.g. 30.40→50.10 for G, 43.50→70.10 for I, 21.70→28.10 for C). These results mirror the 3-state CASP12 findings: on structurally novel FM targets the RF is especially effective at suppressing spurious short helices and β -elements, substantially improving segment-level agreement while leaving Q8 essentially unchanged.

Model	Accuracy Q8 (%)	SOV'99	SOV _{refine} ($\lambda=0.5$)
Seed Ensemble	70.20	42.80	40.50
Ensemble + RF	69.60	51.10	48.50

Table 5.22. Q8, SOV'99 and SOV_{refine} Results in CASP12 after adding post-processing filtering
(8-State Classification and RF window size =31).

Model	SOV'99 _G	SOV'99 _H	SOV'99 _I	SOV'99 _B	SOV'99 _E	SOV'99 _S	SOV'99 _T	SOV'99 _C
Seed Ensemble	30.80	80.70	43.20	8.10	80.30	25.00	51.20	23.50
Ensemble + RF	52.90	83.00	71.30	23.40	81.10	34.60	57.00	30.20

Table 5.23. Detailed SOV'99 Results in CASP12 after adding post-processing filtering
(8-State Classification and RF window size =31).

Model	SOV _{ref,G}	SOV _{ref,H}	SOV _{ref,I}	SOV _{ref,B}	SOV _{ref,E}	SOV _{ref,S}	SOV _{ref,T}	SOV _{ref,C}
Seed Ensemble	30.40	76.50	43.50	8.10	75.90	25.50	50.90	21.70
Ensemble + RF	50.10	79.20	70.10	23.70	76.60	34.30	56.10	28.10

Table 5.24. Detailed SOV_{refine} Results in CASP12 after adding post-processing filtering
(8-State Classification and RF window size =31).

Figures 5.13-5.15 present the confusion matrices of the Ensemble + RF configuration on CB513, TS115 and CASP12 respectively. In contrast to the 3-state matrices, the diagonal is dominant only for the well-populated α -helix (H) and β -strand (E) states, which retain high recall on every benchmark, whereas the minority states are recovered far less reliably. This uneven diagonal is the residue-level origin of the gap between Q8 and Q3 reported throughout this work.

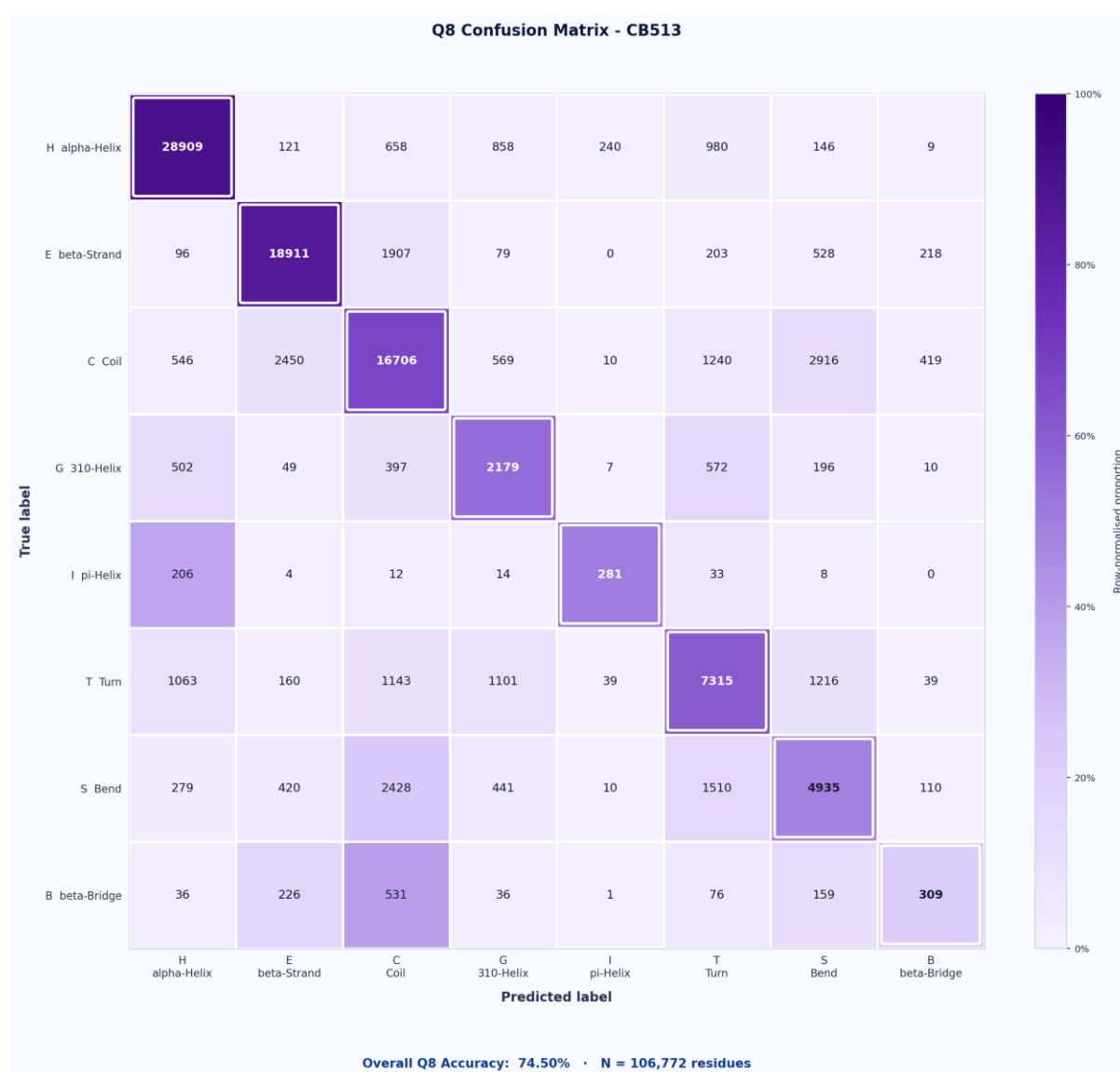


Figure 5.13. Confusion Matrix for 8-State Classification in CB513 after applying RF.

The decisive observation is that the misclassifications are not random but are concentrated within the two broad structural families that the 3-state mapping collapses together. Within the helix family, the rare 3_{10} -helix (G) and π -helix (I) are most often absorbed into the dominant α -helix (H): on CB513 the largest single error for true I is its reassignment to H (206 of 558 residues) and 502 true-G residues are likewise labelled H, with similar patterns in TS115 and CASP12. This is structurally expected, since 3_{10} - and π -helices are short helical variants in the general α -helix category (H) under the 3-state scheme of Table 2.1.

An analogous effect dominates the coil family (C, T, S, B). Bend (S) is most frequently reassigned to coil (2,428 residues on CB513, 516 on TS115 and 166 on CASP12) and isolated β -bridge (B) is the weakest state of all, with recall of only 22.5%, 18.6% and 25.8% on CB513, TS115 and CASP12, its residues being absorbed mainly into coil and strand ($B \rightarrow C = 531$ and $B \rightarrow E = 226$ on CB513). The recall ordering is therefore consistent across benchmarks: H and E are strongest, followed by coil, while the minority states G, I, T, S and B trail well behind. Crucially, cross-family confusion between the helix and strand families remains as small as in the 3-state case, with direct $H \leftrightarrow E$ exchanges of only 121 and 96 residues on CB513, 29 and 37 on TS115 and 21 and 16 on CASP12. The 8-state errors are thus predominantly fine-grained substitutions inside a family rather than gross structural misassignments, which explains why these same predictions yield much higher Q3 accuracy once the families are collapsed, and why the RF improves minority-class segment scores (G, I, B) at the cost of a small Q8 reduction: it reshuffles labels within a family to produce more coherent segments without disturbing the clean helix/strand separation.

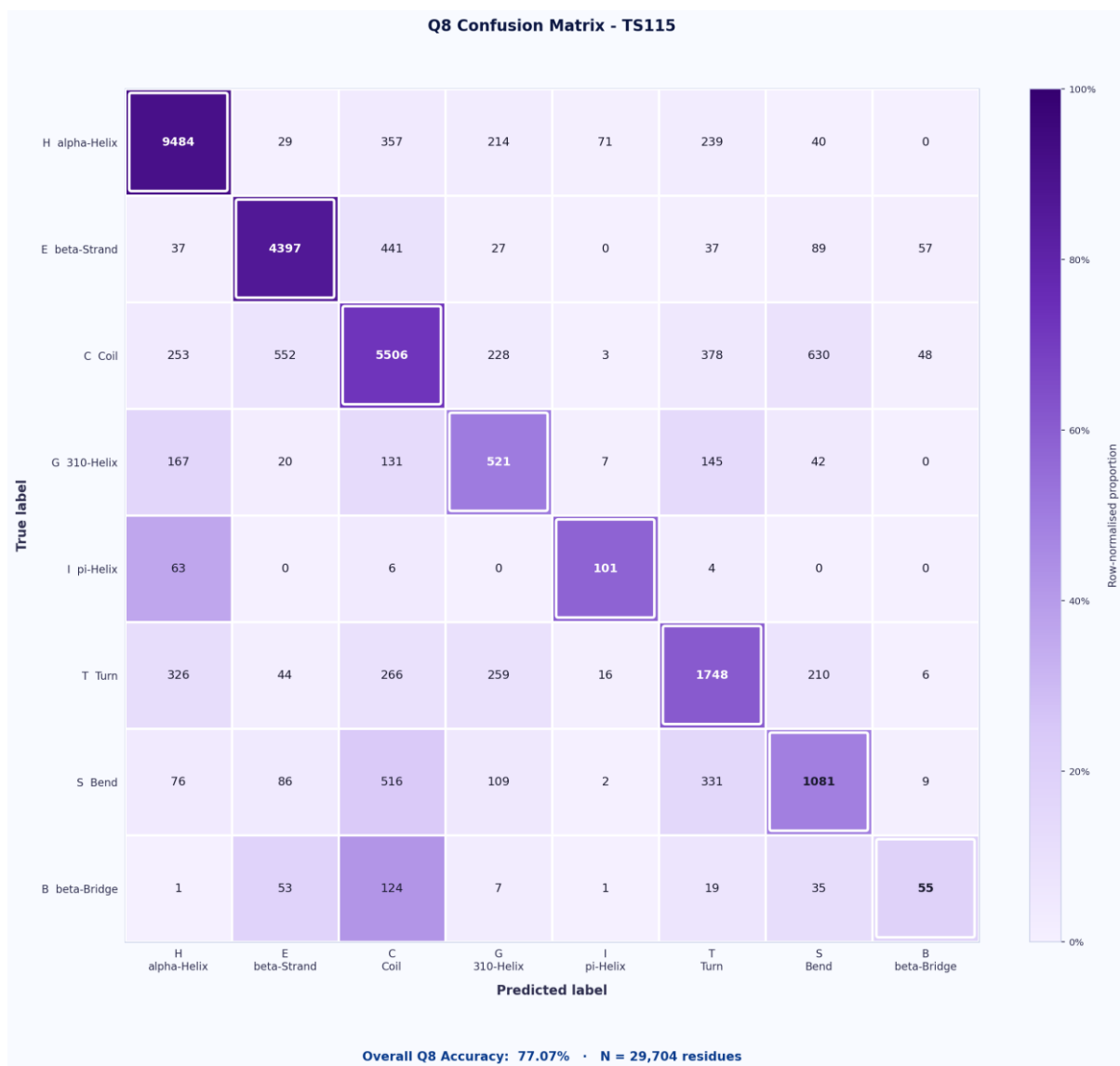


Figure 5.14. Confusion Matrix for 8-State Classification in TS115 after applying RF.

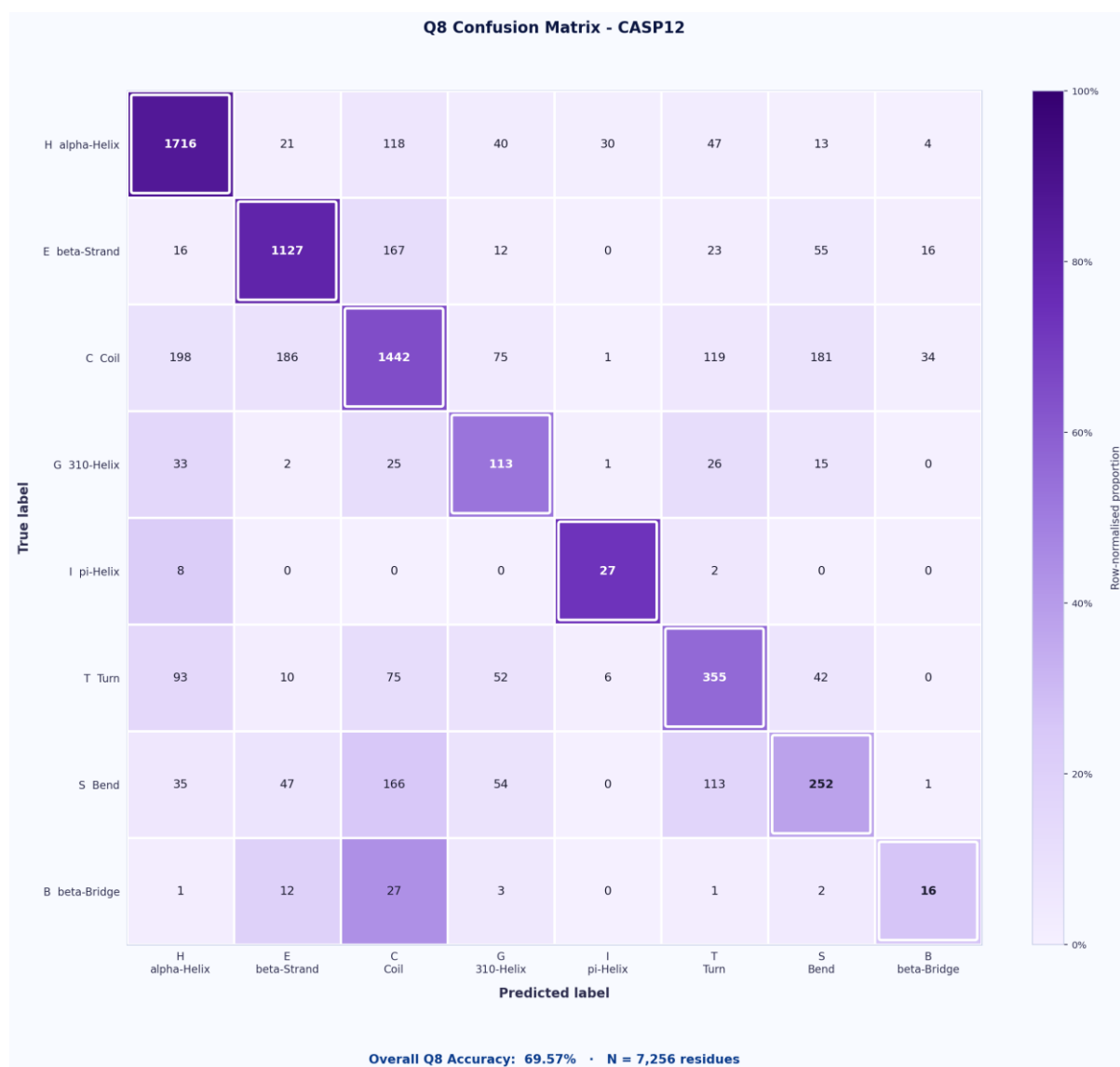


Figure 5.15. Confusion Matrix for 8-State Classification in CASP12 after applying RF.

5.3.5 Window Size Experiment in 8-State

This section examines the effect of window size and input format on post-processing performance in the 8-State classification setting. The results reveal a notably different pattern from the 3-State case, highlighting that the choice of input representation has a more pronounced impact when the label space is larger.

Figure 5.16 reports the average Q8 accuracy across the test sets for multiple window sizes under both input formats. Unlike the 3-State setting, hard label inputs exhibit a clear degradation at smaller window sizes, with Q8 accuracy dropping markedly before recovering as the window grows. Probability inputs, by contrast, remain stable across all tested window sizes, showing no comparable early-window degradation. At smaller window sizes, the gap between the two input formats reaches approximately 4 percentage points in Q8 accuracy, a substantially larger discrepancy than anything observed in the 3-State experiments. As the window size increases, hard label performance converges toward that of probabilities, eventually reaching comparable levels at the largest configurations tested.

Figure 5.17 reports the average SOV'99 across the test sets where the same broad pattern holds. Hard labels show an initial decrease followed by a recovery, while probabilities remain comparatively stable, though with slightly more variation across window sizes than observed in the Q8 accuracy metric. Crucially, the instability of hard labels at smaller windows is again evident in SOV'99, reinforcing that this behaviour is not an artefact of the residue-level metric but reflects a genuine segment-level degradation.

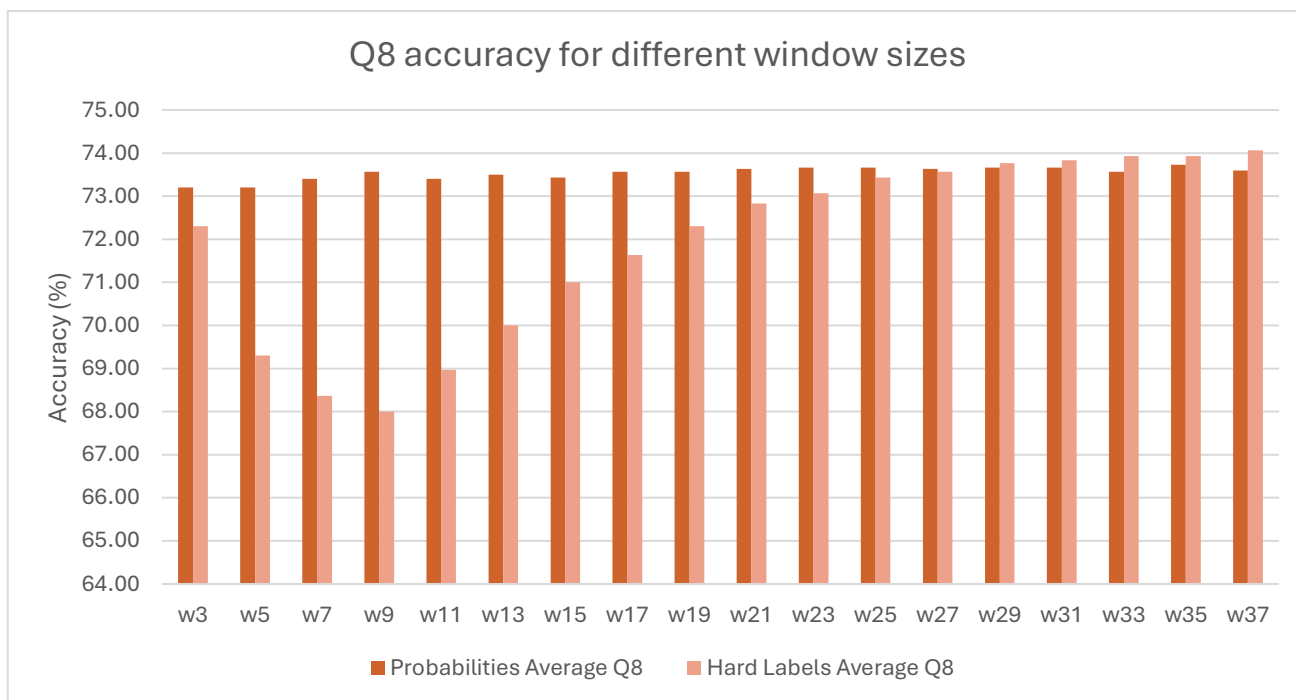


Figure 5.16. Average Q8 across all benchmarks (CASP12, TS115 and CB513), testing multiple window sizes.

For each window both input formats are compared: probabilities or hard labels.

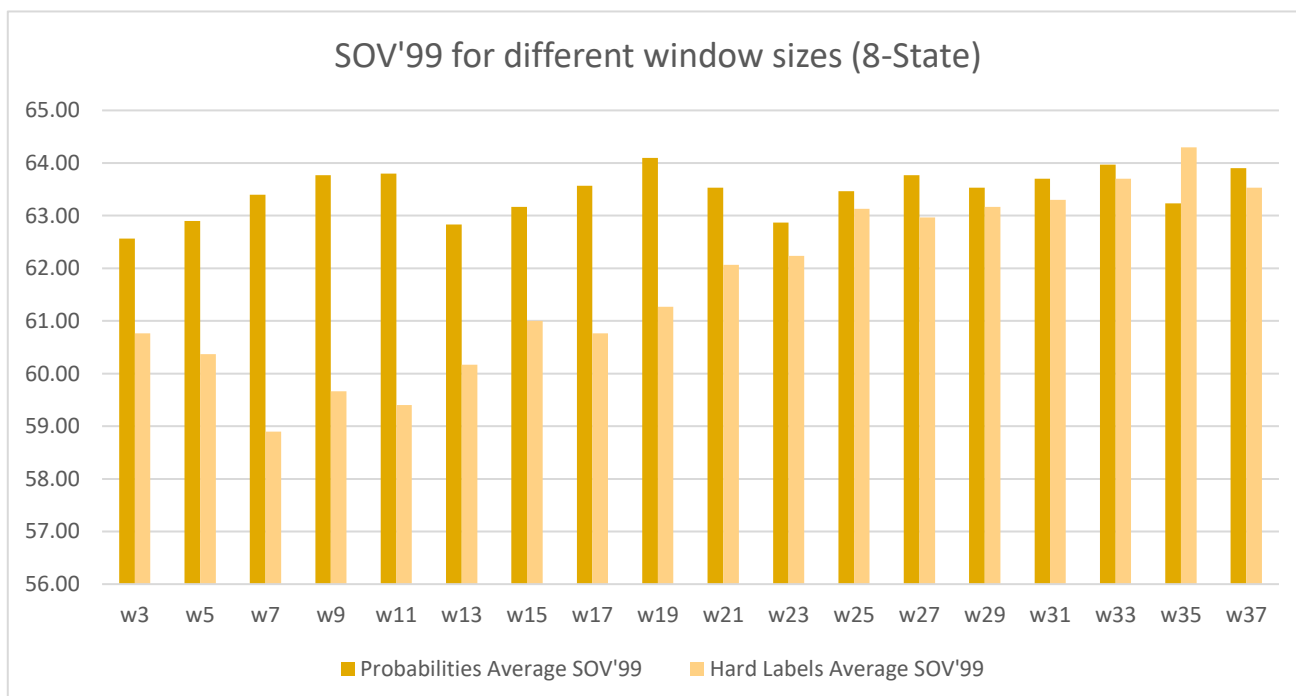


Figure 5.17. Average SOV across all benchmarks (CASP12, TS115 and CB513), testing multiple window sizes.

For each window both input formats are compared: probabilities or hard labels.

This divergence between input formats is more pronounced in the 8-State setting than in 3-State and admits a straightforward interpretation. With eight output classes, many of which correspond to rare structural states such as 3_{10} -helix, π -helix and β -bridge, a hard label at any given position discards the probability mass distributed across the remaining seven classes entirely. In a small window, the classifier has access to very few positions and therefore receives an impoverished, information-sparse representation when labels are used. Probability vectors, carrying the full output distribution at each position, provide enough signal for the classifier to operate reliably even in narrow windows. As the window grows, hard labels compensate through the sheer volume of positional context, eventually recovering the lost uncertainty information through breadth rather than depth. This explains both the early-window deficit and the late convergence observed in Figure 5.16 and Figure 5.17.

In contrast to the 3-state case, probability inputs are not merely a marginal refinement in the 8-state setting but a more important design choice, particularly when computational constraints favour smaller windows. Practitioners facing such constraints are therefore presented with a meaningful trade-off: a small window with probability inputs achieves stable performance at reduced computational cost, while a large window with hard labels can reach comparable accuracy at the expense of a wider context and the associated feature dimensionality. Both configurations are viable, but the probability-based approach offers more predictable behaviour across the full range of window sizes tested.

5.4 Implementation Analysis

This section analyses the proposed base model architecture. Specifically, the attention maps of the fully trained model are compared against a randomly initialized one in order to highlight the relationships developed between the tokens. Furthermore, the mean aggregation mechanism is examined to whether the mean vectors attended provide a separable space for the final MLP head to classify.

5.4.1 Attention Analysis

The attention map analysis was conducted on residue 40 of protein chain 5d9g-A, drawn from the CASP12 benchmark set. Figure 5.18 and Figure 5.19 display the self-attention weights across all encoder layers and heads for both the 3-state and 8-state models under random and trained initializations respectively. A clear qualitative difference is visible between the two conditions: the randomly initialized models produce diffuse, near-uniform attention distributions across tokens, indicating that no structured relationships between embedding subspaces have yet been established. In contrast, the trained models exhibit markedly more structured patterns, with distinct heads attending selectively to specific token pairs and this specialization becomes increasingly pronounced in deeper layers. These differences confirm that the self-attention mechanism learns non-trivial intra-embedding interactions during training, rather than collapsing to a trivial aggregation of the input slices.

An important observation is that the structural complexity of the attention patterns increases with depth. In the earlier encoder layers, attention tends to be broadly distributed, with heads capturing relatively coarse-grained positional or local relationships. As the network deepens, the patterns become progressively sharper and more differentiated across heads, suggesting that higher-level, more abstract intra-embedding dependencies are being captured. This layer-

wise progression is consistent with the general behaviour observed in Transformer models, where lower layers handle local feature interactions and upper layers integrate more global contextual information.

It should be noted that, unlike attention maps in natural language processing where tokens correspond to discrete words or sub words with well-defined semantic identity, the tokens here are fixed-dimensional slices of a continuous ESM-2 embedding vector. As such, the pairwise attention weights between these subspaces do not carry a directly interpretable linguistic or biochemical meaning. For this reason, the maps are best examined at the aggregate level across all layers and heads simultaneously, to identify global structural trends rather than attempting to assign meaning to individual attention values.

Finally, it is worth noting that several heads, in Figure 5.19, exhibit what are known as attention sinks: a phenomenon where a large fraction of attention weight collapses onto one or a small number of tokens [\[144\]](#). Attention sinks are a recognised pathology in Transformer models and may indicate that certain heads are not contributing meaningfully to the representation, their presence here is a point of concern that warrants further investigation in future work. While the broader community has not yet converged on a single consensus about the role of attention sinks, several recent studies argue that they can serve useful geometric or stabilizing functions rather than being inherently harmful [\[144\]](#).

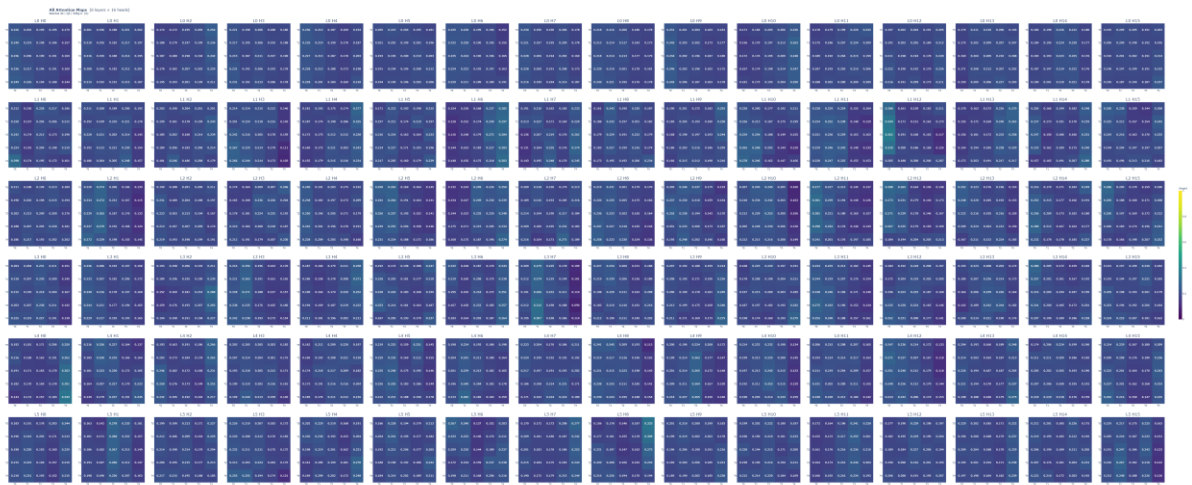


Figure 5.18. Randomly initialized attention weights throughout all the encoder layers and heads (3-State Classification).

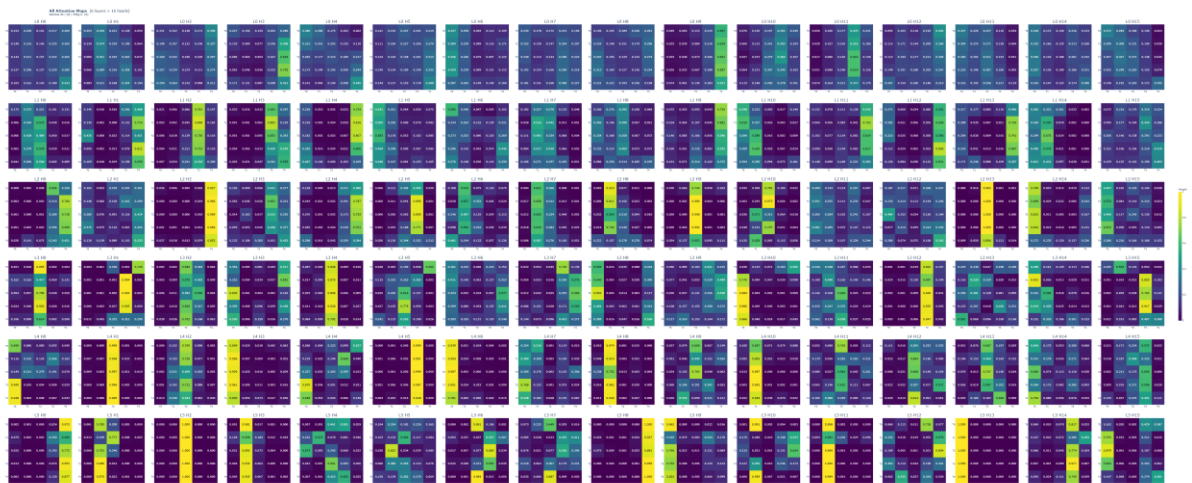


Figure 5.19. Trained attention weights throughout all the encoder layers and heads (3-State Classification).

5.4.2 Mean-Pooled Vectors

To assess whether the mean-pooled encoder representations remain class-discriminative despite the high inter-token similarity observed in the attention analysis, the representation space was visualised using Principal Component Analysis (PCA) [145]. To verify that the observed class-discriminative structure in the mean-pooled representations is genuinely produced by the Transformer encoder and not inherited directly from the ESM-2 embeddings, the PCA visualisation compares the raw ESM-2 embedding vectors against the trained Transformer output for the same set of residues, projected into the same principal component space.

It is important to note that comparing the Transformer's trained output against the output of a randomly initialised Transformer (as is commonly done in representation analysis) would be methodologically insufficient for this purpose. A randomly initialised encoder still processes the ESM-2 input through multiple attention layers and non-linear transformations before producing its output, meaning that any structural properties already present in the ESM-2 embeddings could persist, be amplified, or be distorted in ways that are artefacts of random weight values rather than a neutral baseline. To cleanly isolate the Transformer's contribution, the correct control is to examine the representations before the Transformer is applied at all: the raw, unmodified ESM-2 mean-pooled vectors. If these are already separable by class, any separation seen in the Transformer output cannot be attributed to the encoder. Only if the raw ESM-2 baseline is not separable does the Transformer's output separation constitute genuine evidence of learned structure.

The contrast in Figure 5.20 (Q3, 82.8% variance explained) is unambiguous. On the left, the raw ESM-2 embeddings form a dense, undifferentiated cloud: the three secondary structure classes overlap almost entirely with no visible spatial tendency for any class to separate from

the others. Despite the high explained variance in this projection, class structure is essentially absent. This confirms that the ESM-2 mean embedding, while rich in sequence-level information, does not by itself produce a representation space that is immediately separable along secondary structure class boundaries. On the right, the trained Transformer encoder output tells a sharply different story: Helix residues cluster in the lower-left quadrant, Strand residues in the lower-right and Coil residues in the upper region, with a spread roughly two orders of magnitude larger than the ESM-2 baseline. This dramatic expansion of the representational geometry, together with the clear class separation, constitutes direct empirical evidence that the Transformer encoder actively restructures the embedding space during training, learning to expose secondary structure class boundaries that are latent but not accessible in the raw ESM-2 vectors.

In the 8-state setting (Figure 5.21, 48.3% variance explained), the separation in the Transformer output is naturally weaker than in the 3-state case. The class-discriminative variance is distributed across more directions than two principal components can capture, reflecting the increased structural complexity of the 8-state alphabet. Nevertheless, the same qualitative contrast holds: the raw ESM-2 baseline is unstructured, while the trained encoder output shows coherent groupings by class. The lower explained variance should therefore be interpreted as a partial view of a richer, higher-dimensional separation rather than an absence of it.

It is worth noting that the role of the self-attention layers in this architecture may not be to enhance or enrich the ESM-2 representations in the traditional sense, but rather to summarise and reorganise the relationships between the intra-embedding subspaces produced by token slicing. That is, rather than introducing new information, the encoder may learn to selectively aggregate the content of different slice positions, redistributing emphasis across subspaces in a way that makes class-relevant structure accessible to the mean-pooling operation and the subsequent MLP head, just like a CNN would.

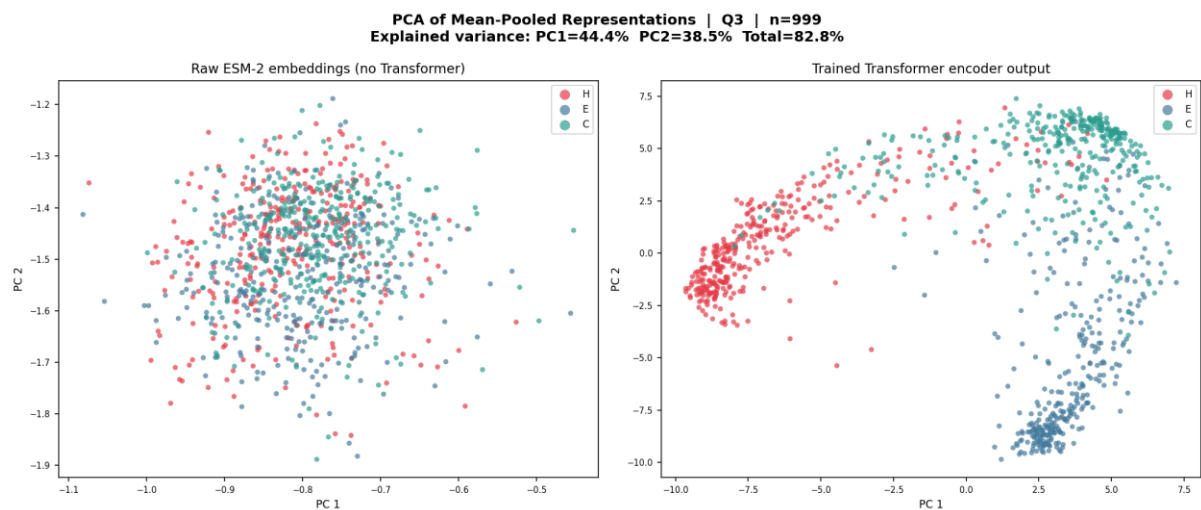


Figure 5.20. PCA of Mean-Pooled vectors in 3-State Classification in comparison to a randomly initialized transformer.

A large sample of 999 residues were taken to balance all 3 classes (From TS115 dataset).

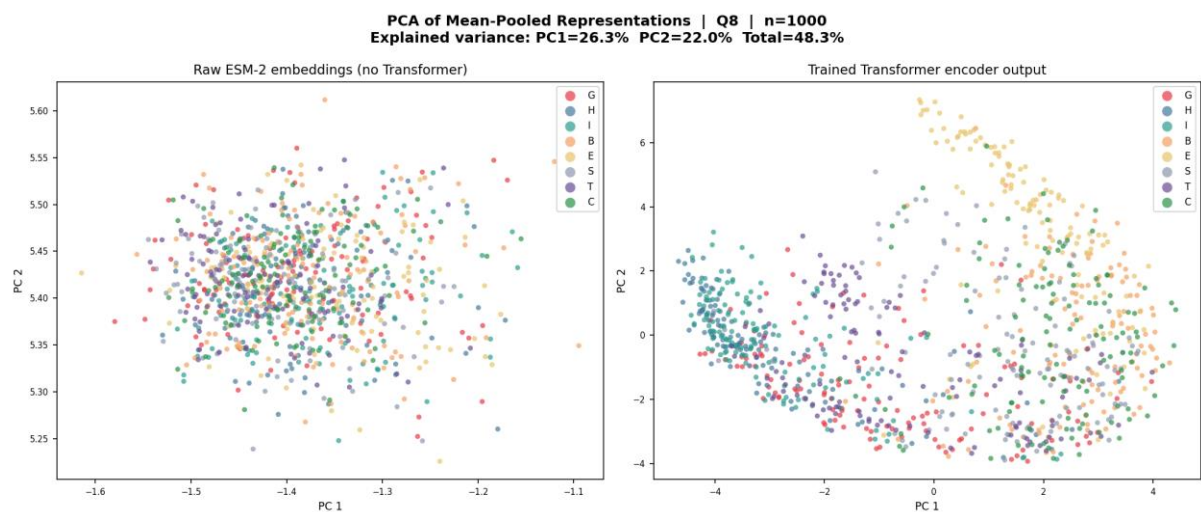


Figure 5.21. PCA of Mean-Pooled vectors in 8-State Classification in comparison to a randomly initialized transformer.

A large sample of 1000 residues were taken to balance all 8 classes (From TS115 dataset).

5.5 Comparison

Table 5.25 presents a direct comparison of the proposed method against established baselines on three independent benchmark datasets: CB513, TS115 and CASP12 across Q3 and Q8 accuracy. The baselines cover three distinct input paradigms: MSA-based evolutionary profiles (NetSurfP-2.0 [60], SADGRU-SS [134] via HHblits [69]), single-sequence PLM embeddings (NetSurfP-3.0 [63] via ESM-1b [124]) and a larger-scale PLM (AttSec [125] via ProtT5-XL-U50 [126]). For SADGRU-SS, only the CB513 test set is reported, and a fully rigorous head-to-head comparison would in principle require retraining SADGRU-SS on the same HHblits training data used in this work, since the original model was trained on the CB6133-filtered dataset. The proposed method uses ESM-2 [105] (650M, UR50D) embeddings, which are alignment-free and do not require any database search at inference time.

On CB513, the Seed Ensemble achieves 86.80% Q3, surpassing NetSurfP-2.0 (+2.30 pp) and NetSurfP-3.0 (+2.20 pp) by clear margins and matching AttSec within 0.30 pp despite AttSec relying on the considerably larger ProtT5-XL-U50 model. The RF post-processor raises Q3 to 87.00%, establishing the highest Q3 result among all methods on this benchmark. In Q8, the Seed Ensemble achieves 75.10%, matching AttSec's 75.20% within a negligible margin and substantially outperforming both NetSurfP variants. The slight Q8 decrease under RF post-processing (75.10% $\rightarrow$ 74.60%) is consistent with the within-helix-family label substitution trade-off and does not represent a deterioration in structural segment quality.

On TS115, the Seed Ensemble achieves 87.20% Q3, surpassing both NetSurfP-2.0 (+2.30 pp) and NetSurfP-3.0 (+1.60 pp) and remaining within 0.30 pp of AttSec. RF post-processing further improves Q3 to 87.30%. For Q8, the Seed Ensemble's 77.80% closely approaches AttSec's 78.50% and substantially outperforms both NetSurfP variants. The small Q8

regression after RF post-processing (77.80% $\rightarrow$ 77.10%) follows the same trade-off pattern observed on CB513.

On CASP12, which consists exclusively of Free Modelling targets and represents the most stringent evaluation setting, the Seed Ensemble achieves 81.40% Q3, surpassing all baselines including AttSec (80.80%) and NetSurfP-2.0 (81.00%). In Q8, the Seed Ensemble's 70.20% exceeds the NetSurfP variants and approaches AttSec (70.60%) to within 0.40 pp. These results are particularly noteworthy: CASP12 FM targets have no structural templates, meaning methods relying on evolutionary profiles benefit less from MSA enrichment and the alignment-free ESM-2 representation used here remains fully effective.

A fully rigorous comparison would require reporting SOV metrics alongside per-residue accuracy, as SOV scores penalise fragmented segment predictions that Q accuracy alone cannot detect. However, none of the direct comparison methods (NetSurfP-2.0, NetSurfP-3.0 and AttSec) report SOV scores in their published results, making a direct SOV-based comparison with these methods infeasible. The only prior work reporting SOV on overlapping benchmarks is eCRRNN [59], a 2018 MSA-based ensemble method. However, Shapovalov et al. [146] independently re-executed the eCRRNN software and found its reported CB513 accuracy overestimated by 3.0–3.8 percentage points, noting that eCRRNN does not specify a sequence identity cutoff between its training and test sets: a known source of accuracy overestimation in this literature. Given that SOV is inherently more sensitive to training-test overlap than per-residue accuracy, the corresponding SOV figures are likely overestimated by an equal or greater margin, rendering them unsuitable as a reference point. The SOV₉₉ and SOV_{refine} scores produced by the proposed system therefore constitute an independent evaluation dimension absent from most contemporary PSSP benchmarks.

Taken together, the results across all three benchmarks confirm that the proposed ESM-2-based architecture is competitive with or superior to the current state of the art in both Q3 and Q8 accuracy, while operating without any MSA computation at inference time.

Test Dataset	Method	Input Method	Accuracy Q3 (%)	Accuracy Q8 (%)
CB513				
	NetSurfP-2.0	HHBlits MSA profiles	84.50	71.30
	NetSurfP-3.0	ESM-1b	84.60	71.10
	AttSec	ProtT5-XL-U50	86.50	75.20
	SADGRU-SS	HHBlits MSA profiles	82.78	70.74
	Seed Ensemble	ESM-2 (UR50D)	86.80	75.10
	Ensemble + RF	ESM-2 (UR50D)	87.00	74.60
TS115				
	NetSurfP-2.0	HHBlits MSA profiles	84.90	74.00
	NetSurfP-3.0	ESM-1b	85.60	74.90
	AttSec	ProtT5-XL-U50	87.50	78.50
	SADGRU-SS	HHBlits MSA profiles	NaN	NaN
	Seed Ensemble	ESM-2 (UR50D)	87.20	77.80
	Ensemble + RF	ESM-2 (UR50D)	87.30	77.10
CASP12				
	NetSurfP-2.0	HHBlits MSA profiles	81.00	69.90
	NetSurfP-3.0	ESM-1b	79.10	66.90
	AttSec	ProtT5-XL-U50	80.80	70.60
	SADGRU-SS	HHBlits MSA profiles	NaN	NaN
	Seed Ensemble	ESM-2 (UR50D)	81.40	70.20
	Ensemble + RF	ESM-2 (UR50D)	81.30	69.60

Table 5.25. Comparison with Direct and Transformer-based methods on independent tests of CB513, TS115 and CASP12.

6 Conclusions and Future Work

6.1 Conclusions and Takeaways

This thesis investigated ESM-2 [105], [112] (650M parameters) embeddings as alignment-free input representations for Protein Secondary Structure Prediction, combined with an encoder-only Transformer and a two-stage post-processing pipeline. The proposed approach consistently matched or surpassed state-of-the-art MSA-based methods across all evaluated benchmarks (CB513, TS115, CASP12 and CASP13).

The results demonstrate that PLM embeddings alone are sufficient to support highly competitive secondary structure prediction without explicit evolutionary profile generation. The base model ensemble achieved Q3 accuracies of 86.80% (CB513), 87.20% (TS115) and 81.40% (CASP12), exceeding NetSurfP-2.0 [60] by over 2 percentage points on CB513 and TS115 without any evolutionary profile computation. Performance on CASP12 Free Modelling targets is particularly notable, demonstrating that ESM-2 representations remain effective even when homologous sequences are scarce, precisely where MSA-based methods are most constrained. These findings establish that high-quality secondary structure prediction can be achieved using frozen PLM embeddings alone, motivating a closer examination of the architectural and training choices that enabled this performance.

The architectural and optimisation experiments further clarified the role of the Transformer encoder within the proposed framework. Freezing ESM-2 is justified by the literature, as fine-tuning yields at most marginal Q3 gains (~ 0.8 pp) at substantially greater cost [137], [138]. Training used AdamW [114], cosine annealing [118], gradient clipping and early stopping. Hyperparameter optimisation via Optuna [141] identified distinct optimal configurations for Q3 and Q8: the Q8 model used a higher learning rate, larger weight decay, more tokens per

embedding and an additional encoder layer, reflecting the greater structural complexity of 8-state classification. While architectural optimisation was important, the choice of training dataset also proved to have a substantial effect on the structural quality of the predictions.

Although the NetSurfP dataset produced marginally higher residue-level Q3 accuracy during cross-validation (87.17% vs. 86.62%), the PISCES produced consistently superior segment-level SOV'99 (81.02 vs. 76.21) and SOVrefine (74.34 vs. 69.39) scores. This divergence demonstrates that Q3 accuracy alone is an insufficient criterion for training dataset selection and that segment-level metrics provide a more structurally meaningful basis for evaluation. The discrepancy between residue-level accuracy and segment-level structural quality further motivated the introduction of a dedicated post-processing stage aimed at improving prediction coherence and reducing fragmented secondary structure segments.

The proposed two-stage RF→ER post-processing pipeline consistently improved segment-level performance above the base ensemble across both 3-state and 8-state settings. In the 3-state case, the RF alone improved SOV'99 by 2.04 points (PISCES) and 3.08 points (NetSurfP) in cross-validation and the full RF→ER configuration delivered gains of up to +3.40 SOV'99 on CB513, +4.00 SOV'99 on TS115 and +2.60 SOV'99 on CASP13 compared to the base ensemble. External rules complementarily eliminated structurally implausible short segments and the RF→ER ordering consistently matched or outperformed its reverse (ER→RF) by up to 0.90 SOV'99 points, confirming it as the preferred configuration. The largest gains appeared on CASP12 FM targets (SOV'99 +10 points, SOV'99C +11.3 points), driven by the removal of spurious helix/strand insertions in the predominantly coil-rich FM sequences. Given these consistent segment-level improvements in the 3-state setting, the same post-processing strategy was subsequently evaluated for 8-state prediction.

Post-processing also delivered consistent improvements in the 8-state setting, where the RF was applied alone. On CB513, SOV'99 rose from 67.40 to 68.90 (+1.50) and SOV_{refine} from 67.80 to 69.40 (+1.60), while on TS115, SOV'99 improved from 70.80 to 73.10 (+2.30) and SOV_{refine} from 69.50 to 72.00 (+2.50). The most striking gains appeared on CASP12, where SOV'99 increased from 42.80 to 51.10 (+8.30) and SOV_{refine} from 40.50 to 48.50 (+8.00). Crucially, the most remarkable improvements were concentrated in the helix subclasses: on CB513, SOV'99 for 3_{10} -helix (G) rose from 38.20 to 57.50 (+19.3 points) and for π -helix (I) from 32.80 to 54.40 (+21.6 points), with closely matched gains on TS115 and CASP12. These gains reflect the RF's ability to enforce segment continuity within the helix family, consolidating the short, fragmented G and I predictions that the base model produces due to the rarity and brevity of these classes. The residue-level Q8 accuracy incurred only a small regression (e.g., 75.10% $\rightarrow$ 74.60% on CB513), representing a structurally acceptable trade-off given the substantially larger improvements in segment coherence.

Since the effectiveness of the RF stage depends strongly on the representation provided by the base model, different RF input formats were also investigated. In the 3-state setting, probability inputs produced modest SOV'99 gains over hard labels. In the 8-state setting, the difference was pronounced: hard labels degraded Q8 accuracy by up to 4 percentage points at smaller window sizes, while probability inputs remained stable across all configurations. This confirms that in the 3-state setting hard labels and smaller window sizes may be used since they are computationally and memory cheaper without losing severe performance. In the 8-state setting however while hard labels compensate on higher window sizes, probabilities with lower windows are preferred since they are more predictable and stable.

The implementation analysis provides additional insight into the internal behaviour of the model. Attention map analysis on a representative CASP12 residue for 3-State classification shows that training induces structured and head-specific attention patterns: while the randomly

initialised model produces diffuse and near-uniform attention distributions, the trained model exhibits selective and increasingly differentiated attention across heads and layers. This indicates that the Transformer encoder learns meaningful intra-embedding relationships. Mean-pooled encoder representations further indicate that the Transformer reshapes the original embedding space into a more structured and class-separable representation, consistent with earlier PCA findings, suggesting that the encoder contributes meaningfully to downstream classification rather than acting as a passive transformation layer.

Taken together, the benchmark results demonstrate that the proposed framework achieves performance competitive with and in several cases superior to, state-of-the-art MSA-based systems while avoiding the computational cost of evolutionary profile generation. On CB513, the base ensemble (86.80% Q3, 75.10% Q8) surpassed both NetSurfP [60], [63] variants by over 2 pp in Q3 and ~4 pp in Q8, matching AttSec [125] within 0.30 pp. On CASP12 FM targets, the model achieved 81.40% Q3, exceeding all baselines including AttSec (80.80%) and NetSurfP-2.0 (81.00%), confirming that alignment-free ESM-2 representations are a viable and computationally efficient alternative to MSA-based approaches.

6.2 Limitations

Despite the strong results obtained, several limitations of this work should be acknowledged.

First, comparison with baseline methods is restricted to Q3 and Q8 accuracy metrics, as published baselines (NetSurfP-3.0 [63] and AttSec [125]) do not report Segment Overlap (SOV) scores. This prevents a direct assessment of whether the SOV improvements observed in the RF post-processing stage represent genuine gains over these methods or whether they reflect differences in evaluation reporting rather than structural performance. Future benchmark work should therefore standardise the joint reporting of both residue-level (Q) and segment-level (SOV) metrics to enable consistent comparison across studies.

Second, the 8-state classification results highlight persistent difficulties in predicting rare DSSP classes. In particular, β -bridge (B) remains poorly captured due to its single-residue nature, while 310-helix (G) and π -helix (I) are systematically under-represented relative to their structural importance. Although class weighting was applied within the RF stage, it does not fully resolve the underlying imbalance present in PDB-derived datasets.

Third, the model was evaluated only on 3-state and 8-state DSSP classification. While these are standard benchmarks for PSSP, other structurally relevant tasks such as solvent accessibility prediction and IDR detection were not explored, despite being available within the NetSurfP dataset [60]. This limits the assessment of how generalisable the learned representations are across related structural prediction tasks.

Fourth, the post-processing window size (31 residues) was selected for consistency with prior work [132], but empirical results indicate that performance is only weakly sensitive to this choice. This suggests that the window size may not be optimal and could potentially be reduced without loss in accuracy, improving computational efficiency.

Fifth, the ensemble strategy is based on seed diversity, where models differ only in random initialisation. While this improves robustness, it produces limited functional diversity, as models tend to converge to similar solutions and exhibit correlated errors. This reduces the potential benefit of ensemble, particularly for downstream RF-based correction.

6.3 Future Work

Each of the above limitations directly motivates specific directions for future work.

First, the persistent underperformance on minority DSSP states suggests that future work should explore more targeted imbalance-handling strategies. This includes oversampling of rare secondary structure classes, augmentation of under-represented structural motifs and the use of loss functions explicitly designed to emphasise minority classes during training of the base model rather than only at the post-processing stage.

Building on this, a second direction is to extend the model beyond secondary structure prediction to additional structural annotation tasks. A multi-task learning framework jointly predicting secondary structure, solvent accessibility, dihedral angles and disorder could improve shared representation learning while also providing a more comprehensive evaluation of the encoder's learned structural information.

In a related architectural direction, the weak sensitivity of performance to post-processing window size suggests that smaller or more adaptive receptive fields may be sufficient. Future work could therefore investigate adaptive or learnable window sizes or alternatively replace fixed windows with attention-based or dynamic segmentation mechanisms to improve both efficiency and flexibility for long protein sequences.

A fourth direction concerns the enrichment of the input representation. Although ESM-2 embeddings provide rich contextual sequence information, explicit physicochemical properties such as polarity/hydrophilicity, charge and secondary structure propensities encode deterministic, local chemical constraints that PLMs capture only implicitly. Future work should investigate whether such features provide complementary signal and if so, identify a suitable fusion mechanism for integrating them with the PLM embeddings. A related question is whether reintroducing MSA-derived features such as PSSM or HMM profiles alongside PLM embeddings yields further gains, particularly for well-represented protein families.

A fifth direction is motivated directly by the PCA analysis of mean-pooled representations presented in Section 5.4.2 and is further supported by the patterns observed in the confusion matrices discussed in Section 5.2.4. The PCA visualisation reveals that misclassified residues in 3-state prediction follow a structured error pattern: incorrectly predicted helices are most likely assigned to coil and incorrectly predicted strands are similarly most likely assigned to coil, rather than to each other. This suggests that the dominant source of error is a helix-vs-coil and strand-vs-coil confusion, rather than a helix-vs-strand one. Future work could exploit this observation by decomposing the 3-state classification problem into two sequential binary classifiers: a first corrector that discriminates helix from coil and a second that discriminates strand from coil. Each binary classifier can be trained on the specific decision boundary it targets, potentially achieving sharper separation than a single 3-class model.

A sixth direction for future work involves investigating the use of second-order optimisation methods specifically within the final MLP classification head of the proposed architecture. In particular, Hessian-Free Optimization (HFO), previously explored in the post-processing lineage of PSSP methods [130], [131], [132], could be more effective than first-order optimisers such as AdamW [114]. Since most of the model parameters are contained within the Transformer encoder, restricting HFO to the comparatively small MLP head may make its computational cost practical while still improving convergence behaviour and local minima exploration.

Finally, to address the limitations of seed-based ensembles, future work should explore more structurally diverse ensemble strategies. This includes architectural variation, data-level diversity through bagging, or stochastic regularisation differences such as dropout configurations. Such diversity is expected to reduce error correlation between models and increase the effectiveness of the RF post-processing stage.

7 References

- [1] D. L. Nelson, M. M. Cox, and A. A. Hoskins, *Lehninger principles of biochemistry*, Eighth edition. Austin: Macmillan Learning, 2021.
- [2] A. LaPelusa and R. Kaushik, 'Physiology, Proteins', in *StatPearls*, Treasure Island (FL): StatPearls Publishing, 2026. Accessed: Apr. 18, 2026. [Online]. Available: <http://www.ncbi.nlm.nih.gov/books/NBK555990/>
- [3] R. H. Garrett and C. M. Grisham, *Biochemistry*, Sixth edition. Boston, MA, USA: Cengage Learning, 2017.
- [4] 'Cells by the Numbers | National Institute of General Medical Sciences'. Accessed: Apr. 18, 2026. [Online]. Available: <https://nigms.nih.gov/biobeat/2024/09/cells-by-the-numbers-2>
- [5] 'Proteins by the Numbers | National Institute of General Medical Sciences'. Accessed: Apr. 18, 2026. [Online]. Available: <https://nigms.nih.gov/biobeat/2025/01/proteins-by-the-numbers>
- [6] A. Ragupathi, A. J. Mastrogiannis, and N. Rahimi, 'Biochemistry, Tertiary Protein Structure', in *StatPearls*, Treasure Island (FL): StatPearls Publishing, 2026. Accessed: Apr. 18, 2026. [Online]. Available: <http://www.ncbi.nlm.nih.gov/books/NBK470269/>
- [7] L. Pauling, R. B. Corey, and H. R. Branson, 'The structure of proteins: Two hydrogen-bonded helical configurations of the polypeptide chain', *Proc. Natl. Acad. Sci.*, vol. 37, no. 4, pp. 205–211, Apr. 1951, doi: 10.1073/pnas.37.4.205.
- [8] L. Pauling and R. B. Corey, 'Configurations of Polypeptide Chains With Favored Orientations Around Single Bonds', *Proc. Natl. Acad. Sci.*, vol. 37, no. 11, pp. 729–740, Nov. 1951, doi: 10.1073/pnas.37.11.729.
- [9] J. C. Kendrew, G. Bodo, H. M. Dintzis, R. G. Parrish, H. Wyckoff, and D. C. Phillips, 'A Three-Dimensional Model of the Myoglobin Molecule Obtained by X-Ray Analysis', *Nature*, vol. 181, no. 4610, pp. 662–666, Mar. 1958, doi: 10.1038/181662a0.
- [10] C. B. Anfinsen, 'Principles that Govern the Folding of Protein Chains', *Science*, vol. 181, no. 4096, pp. 223–230, Jul. 1973, doi: 10.1126/science.181.4096.223.
- [11] G. Gambardella *et al.*, 'The Anfinsen Dogma: Intriguing Details Sixty-Five Years Later', *Int. J. Mol. Sci.*, vol. 23, no. 14, p. 7759, Jul. 2022, doi: 10.3390/ijms23147759.
- [12] I. J. Hirst, W. J. R. Thomas, R. A. Davies, and S. P. Muench, 'CryoEM grid preparation: a closer look at advancements and impact of preparation mode and new approaches', *Biochem. Soc. Trans.*, vol. 52, no. 3, pp. 1529–1537, Jun. 2024, doi: 10.1042/BST20231553.
- [13] U. Heinemann, G. Illing, and H. Oschkinat, 'High-throughput three-dimensional protein structure determination', *Curr. Opin. Biotechnol.*, vol. 12, no. 4, pp. 348–354, Aug. 2001, doi: 10.1016/S0958-1669(00)00226-3.
- [14] P. Klukowski, R. Riek, and P. Güntert, 'Rapid protein assignments and structures from raw NMR spectra with the deep learning technique ARTINA', *Nat. Commun.*, vol. 13, no. 1, p. 6151, Oct. 2022, doi: 10.1038/s41467-022-33879-5.
- [15] T. C. Terwilliger, D. Stuart, and S. Yokoyama, 'Lessons from Structural Genomics*', *Annu. Rev. Biophys.*, vol. 38, no. Volume 38, 2009, pp. 371–383, Jun. 2009, doi: 10.1146/annurev.biophys.050708.133740.
- [16] H. M. Berman *et al.*, 'The Protein Data Bank', *Nucleic Acids Res.*, vol. 28, no. 1, pp. 235–242, Jan. 2000, doi: 10.1093/nar/28.1.235.

- [17] H. M. Berman and S. K. Burley, 'Protein Data Bank (PDB): Fifty-three years young and having a transformative impact on science and society', *Q. Rev. Biophys.*, vol. 58, p. e9, 2025, doi: 10.1017/S0033583525000034.
- [18] The UniProt Consortium *et al.*, 'UniProt: the Universal Protein Knowledgebase in 2025', *Nucleic Acids Res.*, vol. 53, no. D1, pp. D609–D617, Jan. 2025, doi: 10.1093/nar/gkae1010.
- [19] C. Levinthal, 'How to fold gracefully', 1969. [Online]. Available: <https://api.semanticscholar.org/CorpusID:9923873>
- [20] R. Zwanzig, A. Szabo, and B. Bagchi, 'Levinthal's paradox.', *Proc. Natl. Acad. Sci.*, vol. 89, no. 1, pp. 20–22, Jan. 1992, doi: 10.1073/pnas.89.1.20.
- [21] L. Chen, J. K. Morrow, H. T. Tran, S. S. Phatak, L. Du-Cuny, and S. Zhang, 'From Laptop to Benchtop to Bedside: Structure-based Drug Design on Protein Targets', *Curr. Pharm. Des.*, vol. 18, no. 9, pp. 1217–1239, Mar. 2012, doi: 10.2174/138161212799436386.
- [22] O. Koszła and P. Sołek, 'Misfolding and aggregation in neurodegenerative diseases: protein quality control machinery as potential therapeutic clearance pathways', *Cell Commun. Signal.*, vol. 22, no. 1, p. 421, Aug. 2024, doi: 10.1186/s12964-024-01791-8.
- [23] G. Forloni *et al.*, 'Protein misfolding in Alzheimer's and Parkinson's disease: genetics and molecular mechanisms', *Neurobiol. Aging*, vol. 23, no. 5, pp. 957–976, Sep. 2002, doi: 10.1016/S0197-4580(02)00076-3.
- [24] M. J. Lopez and S. S. Mohiuddin, 'Biochemistry, Essential Amino Acids', in *StatPearls*, Treasure Island (FL): StatPearls Publishing, 2026. Accessed: Apr. 18, 2026. [Online]. Available: <http://www.ncbi.nlm.nih.gov/books/NBK557845/>
- [25] '2.5.4: Amino Acids', Biology LibreTexts. Accessed: Apr. 18, 2026. [Online]. Available: [https://bio.libretexts.org/Bookshelves/Microbiology/Microbiology_\(Boundless\)/02%3A_A_Chemistry/2.05%3A_Organic_Compounds/2.5.04%3A_Amino_Acids](https://bio.libretexts.org/Bookshelves/Microbiology/Microbiology_(Boundless)/02%3A_A_Chemistry/2.05%3A_Organic_Compounds/2.5.04%3A_Amino_Acids)
- [26] OrgoGuy2theRescue, *English: Table of 21 amino acids*. 2015. Accessed: Apr. 18, 2026. [Online]. Available: https://commons.wikimedia.org/wiki/File:AAs_table.png
- [27] T. Sanvictores and F. Farci, 'Biochemistry, Primary Protein Structure', in *StatPearls*, Treasure Island (FL): StatPearls Publishing, 2026. Accessed: Apr. 18, 2026. [Online]. Available: <http://www.ncbi.nlm.nih.gov/books/NBK564343/>
- [28] I. Rehman, M. Farooq, and S. Botelho, 'Biochemistry, Secondary Protein Structure', in *StatPearls*, Treasure Island (FL): StatPearls Publishing, 2026. Accessed: Apr. 18, 2026. [Online]. Available: <http://www.ncbi.nlm.nih.gov/books/NBK470235/>
- [29] '7.4: Secondary structure of proteins', Chemistry LibreTexts. Accessed: Apr. 18, 2026. [Online]. Available: [https://chem.libretexts.org/Bookshelves/Introductory_Chemistry/Introduction_to_Organic_and_Biochemistry_\(Malik\)/07%3A_Proteins/7.04%3A_Secondary_structure_of_proteins](https://chem.libretexts.org/Bookshelves/Introductory_Chemistry/Introduction_to_Organic_and_Biochemistry_(Malik)/07%3A_Proteins/7.04%3A_Secondary_structure_of_proteins)
- [30] Dcrjsr, *English: Diagram of protein backbone local geometry, with peptide plane indicated, and other features labeled. Modified by Jane Richardson from her figure in Anatomy & Taxonomy of Protein Structure*. 2012. Accessed: Apr. 18, 2026. [Online]. Available: https://commons.wikimedia.org/wiki/File:PhiPsi_drawing_with_plane_and_labels.jpg
- [31] O. regalis, *Ramachandran plot generated with coordinates from the human DNA clamp PCNA, PDB ID 1AXC, using VMD ([www.ks.uiuc.edu/Research/vmd/]) and the Ramaplot extension*. 2006. Accessed: Apr. 18, 2026. [Online]. Available: <https://commons.wikimedia.org/wiki/File:Ramaplot.png>
- [32] P. Y. Chou and G. D. Fasman, 'Prediction of protein conformation', *Biochemistry*, vol. 13, no. 2, pp. 222–245, Jan. 1974, doi: 10.1021/bi00699a002.

- [33] C. N. X. OpenStax, *Name: Biology*. 2016. Accessed: Apr. 18, 2026. [Online]. Available: https://commons.wikimedia.org/wiki/File:Figure_03_04_07.jpg
- [34] '13.4: Secondary, tertiary, and quaternary structure of proteins', Chemistry LibreTexts. Accessed: Apr. 18, 2026. [Online]. Available: https://chem.libretexts.org/Courses/Harper_College/General_Organic_and_Biochemistry_with_Problems_Case_Studies_and_Activities/13%3A_Proteins/13.04%3A_Secondary_tertiary_and_quaternary_structure_of_proteins
- [35] C. N. X. OpenStax, *Name:* 2016. Accessed: Apr. 18, 2026. [Online]. Available: https://commons.wikimedia.org/wiki/File:OSC_Microbio_07_04_proteinstr.jpg
- [36] W. Kabsch and C. Sander, 'Dictionary of protein secondary structure: Pattern recognition of hydrogen-bonded and geometrical features', *Biopolymers*, vol. 22, no. 12, pp. 2577–2637, 1983, doi: 10.1002/bip.360221211.
- [37] 'PDB101: Learn: Guide to Understanding PDB Data: Methods for Determining Structure', RCSB: PDB-101. Accessed: Apr. 20, 2026. [Online]. Available: <https://pdb101.rcsb.org/learn/guide-to-understanding-pdb-data/methods-for-determining-structure>
- [38] A. M. Davis, S. A. St-Gallay, and G. J. Kleywegt, 'Limitations and lessons in the use of X-ray structural information in drug design', *Drug Discov. Today*, vol. 13, no. 19, pp. 831–841, Oct. 2008, doi: 10.1016/j.drudis.2008.06.006.
- [39] E. D'Imprima and W. Kühlbrandt, 'Current limitations to high-resolution structure determination by single-particle cryoEM', *Q. Rev. Biophys.*, vol. 54, p. e4, 2021, doi: 10.1017/S0033583521000020.
- [40] A. Chari and H. Stark, 'Prospects and Limitations of High-Resolution Single-Particle Cryo-Electron Microscopy', *Annu. Rev. Biophys.*, vol. 52, no. 1, pp. 391–411, May 2023, doi: 10.1146/annurev-biophys-111622-091300.
- [41] M. Necelis, C. McDermott, M. Belcher Dufrisne, C. Baryames, and L. Columbus, 'Solution NMR investigations of integral membrane proteins: Challenges and innovations', *Curr. Opin. Struct. Biol.*, vol. 82, p. 102654, Oct. 2023, doi: 10.1016/j.sbi.2023.102654.
- [42] D. P. Ismi, R. Pulungan, and Afiahayati, 'Deep learning for protein secondary structure prediction: Pre and post-AlphaFold', *Comput. Struct. Biotechnol. J.*, vol. 20, pp. 6271–6286, Jan. 2022, doi: 10.1016/j.csbj.2022.11.012.
- [43] Y. Yang *et al.*, 'Sixty-five years of the long march in protein secondary structure prediction: the final stretch?', *Brief. Bioinform.*, vol. 19, no. 3, pp. 482–494, May 2018, doi: 10.1093/bib/bbw129.
- [44] Q. Jiang, X. Jin, S.-J. Lee, and S. Yao, 'Protein secondary structure prediction: A survey of the state of the art', *J. Mol. Graph. Model.*, vol. 76, pp. 379–402, Sep. 2017, doi: 10.1016/j.jmglm.2017.07.015.
- [45] J. Garnier, D. J. Osguthorpe, and B. Robson, 'Analysis of the accuracy and implications of simple methods for predicting the secondary structure of globular proteins', *J. Mol. Biol.*, vol. 120, no. 1, pp. 97–120, Mar. 1978, doi: 10.1016/0022-2836(78)90297-8.
- [46] V. I. Lim, 'Algorithms for prediction of α -helical and β -structural regions in globular proteins', *J. Mol. Biol.*, vol. 88, no. 4, pp. 873–894, Oct. 1974, doi: 10.1016/0022-2836(74)90405-7.
- [47] 'Protein Secondary Structure Prediction: A Review of Progress and Directions', *Curr. Bioinforma.*, vol. 15, no. 2, pp. 90–107, Jan. 2020, doi: 10.2174/1574893614666191017104639.
- [48] E. M. Mitchell, P. J. Artymiuk, D. W. Rice, and P. Willett, 'Use of techniques derived from graph theory to compare secondary structure motifs in proteins', *J. Mol. Biol.*, vol. 212, no. 1, pp. 151–166, Mar. 1990, doi: 10.1016/0022-2836(90)90312-A.

- [49] L. H. Holley and M. Karplus, 'Protein secondary structure prediction with a neural network.', *Proc. Natl. Acad. Sci.*, vol. 86, no. 1, pp. 152–156, Jan. 1989, doi: 10.1073/pnas.86.1.152.
- [50] H. Bohr *et al.*, 'Protein secondary structure and homology by neural networks The α -helices in rhodopsin', *FEBS Lett.*, vol. 241, no. 1–2, pp. 223–228, 1988, doi: 10.1016/0014-5793(88)81066-4.
- [51] S. Muggleton, R. D. King, and M. J. E. Stenberg, 'Protein secondary structure prediction using logic-based machine learning', *Protein Eng.*, vol. 5, no. 7, pp. 647–657, Oct. 1992, doi: 10.1093/protein/5.7.647.
- [52] T.-M. Yi and E. S. Lander, 'Protein Secondary Structure Prediction Using Nearest-neighbor Methods', *J. Mol. Biol.*, vol. 232, no. 4, pp. 1117–1129, Aug. 1993, doi: 10.1006/jmbi.1993.1464.
- [53] E. A. Kabat and T. T. Wu, 'The Influence of Nearest-Neighbor Amino Acids on the Conformation of the Middle Amino Acid in Proteins: Comparison of Predicted and Experimental Determination of β -Sheets in Concanavalin A', *Proc. Natl. Acad. Sci.*, vol. 70, no. 5, pp. 1473–1477, May 1973, doi: 10.1073/pnas.70.5.1473.
- [54] B. Rost and C. Sander, 'Improved prediction of protein secondary structure by use of sequence profiles and neural networks.', *Proc. Natl. Acad. Sci.*, vol. 90, no. 16, pp. 7558–7562, Aug. 1993, doi: 10.1073/pnas.90.16.7558.
- [55] J. J. Ward, L. J. McGuffin, B. F. Buxton, and D. T. Jones, 'Secondary structure prediction with support vector machines', *Bioinformatics*, vol. 19, no. 13, pp. 1650–1655, Sep. 2003, doi: 10.1093/bioinformatics/btg223.
- [56] S. Hua and Z. Sun, 'A novel method of protein secondary structure prediction with high segment overlap measure: support vector machine approach1', *J. Mol. Biol.*, vol. 308, no. 2, pp. 397–407, Apr. 2001, doi: 10.1006/jmbi.2001.4580.
- [57] R. Heffernan *et al.*, 'Improving prediction of secondary structure, local backbone angles and solvent accessible surface area of proteins by iterative deep learning', *Sci. Rep.*, vol. 5, no. 1, p. 11476, Jun. 2015, doi: 10.1038/srep11476.
- [58] S. Wang, J. Peng, J. Ma, and J. Xu, 'Protein Secondary Structure Prediction Using Deep Convolutional Neural Fields', *Sci. Rep.*, vol. 6, no. 1, p. 18962, Jan. 2016, doi: 10.1038/srep18962.
- [59] B. Zhang, J. Li, and Q. Lü, 'Prediction of 8-state protein secondary structures by a novel deep learning architecture', *BMC Bioinformatics*, vol. 19, no. 1, p. 293, Aug. 2018, doi: 10.1186/s12859-018-2280-5.
- [60] M. S. Klausen *et al.*, 'NetSurfP-2.0: Improved prediction of protein structural features by integrated deep learning', *Proteins Struct. Funct. Bioinforma.*, vol. 87, no. 6, pp. 520–527, Jun. 2019, doi: 10.1002/prot.25674.
- [61] W. Alanazi, D. Meng, and G. Pollastri, 'Advancements in one-dimensional protein structure prediction using machine learning and deep learning', *Comput. Struct. Biotechnol. J.*, vol. 27, pp. 1416–1430, Jan. 2025, doi: 10.1016/j.csbj.2025.04.005.
- [62] J. Jumper *et al.*, 'Highly accurate protein structure prediction with AlphaFold', *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021, doi: 10.1038/s41586-021-03819-2.
- [63] M. H. Høie *et al.*, 'NetSurfP-3.0: accurate and fast prediction of protein structural features by protein language models and deep learning', *Nucleic Acids Res.*, vol. 50, no. W1, pp. W510–W515, Jul. 2022, doi: 10.1093/nar/gkac439.
- [64] J. Singh, K. Paliwal, T. Litfin, J. Singh, and Y. Zhou, 'Reaching alignment-profile-based accuracy in predicting protein secondary and tertiary structural properties without alignment', *Sci. Rep.*, vol. 12, no. 1, p. 7607, May 2022, doi: 10.1038/s41598-022-11684-w.

- [65] W. Alanazi, D. Meng, and G. Pollastri, 'Porter 6: Protein Secondary Structure Prediction by Leveraging Pre-Trained Language Models (PLMs)', *Int. J. Mol. Sci.*, vol. 26, no. 1, Dec. 2024, doi: 10.3390/ijms26010130.
- [66] Y. Zhang, Q. Zhang, J. Zhou, and Q. Zou, 'A survey on the algorithm and development of multiple sequence alignment', *Brief. Bioinform.*, vol. 23, no. 3, p. bbac069, May 2022, doi: 10.1093/bib/bbac069.
- [67] S. Altschul, 'Gapped BLAST and PSI-BLAST: a new generation of protein database search programs', *Nucleic Acids Res.*, vol. 25, no. 17, pp. 3389–3402, Sep. 1997, doi: 10.1093/nar/25.17.3389.
- [68] S. R. Eddy, 'Profile hidden Markov models.', *Bioinformatics*, vol. 14, no. 9, pp. 755–763, Jan. 1998, doi: 10.1093/bioinformatics/14.9.755.
- [69] M. Remmert, A. Biegert, A. Hauser, and J. Söding, 'HHblits: lightning-fast iterative protein sequence searching by HMM-HMM alignment', *Nat. Methods*, vol. 9, no. 2, pp. 173–175, Feb. 2012, doi: 10.1038/nmeth.1818.
- [70] F. Kallenborn *et al.*, 'GPU-accelerated homology search with MMseqs2', *Nat. Methods*, vol. 22, no. 10, pp. 2024–2027, Oct. 2025, doi: 10.1038/s41592-025-02819-8.
- [71] L. Wang *et al.*, 'Overview of AlphaFold2 and breakthroughs in overcoming its limitations', *Comput. Biol. Med.*, vol. 176, p. 108620, Jun. 2024, doi: 10.1016/j.compbiomed.2024.108620.
- [72] D. Chakravarty, M. Lee, and L. L. Porter, 'Proteins with alternative folds reveal blind spots in AlphaFold-based protein structure prediction', *Curr. Opin. Struct. Biol.*, vol. 90, p. 102973, Feb. 2025, doi: 10.1016/j.sbi.2024.102973.
- [73] S. Park, S. Myung, and M. Baek, 'Advancing protein structure prediction beyond AlphaFold2', *Curr. Opin. Struct. Biol.*, vol. 90, p. 102985, Feb. 2025, doi: 10.1016/j.sbi.2025.102985.
- [74] G. Wang and R. L. Dunbrack Jr, 'PISCES: a protein sequence culling server', *Bioinformatics*, vol. 19, no. 12, pp. 1589–1591, Aug. 2003, doi: 10.1093/bioinformatics/btg224.
- [75] J. Zhou and O. Troyanskaya, 'Deep Supervised and Convolutional Generative Stochastic Network for Protein Secondary Structure Prediction', in *Proceedings of the 31st International Conference on Machine Learning*, PMLR, Jan. 2014, pp. 745–753. Accessed: Apr. 22, 2026. [Online]. Available: <https://proceedings.mlr.press/v32/zhou14.html>
- [76] J. A. Cuff and G. J. Barton, 'Evaluation and improvement of multiple sequence methods for protein secondary structure prediction', *Proteins Struct. Funct. Bioinforma.*, vol. 34, no. 4, pp. 508–519, 1999, doi: 10.1002/(SICI)1097-0134(19990301)34:4<508::AID-PROT10>3.0.CO;2-4.
- [77] R. Heffernan, K. Paliwal, J. Lyons, J. Singh, Y. Yang, and Y. Zhou, 'Single-sequence-based prediction of protein secondary structures and solvent accessibility by deep whole-sequence learning', *J. Comput. Chem.*, vol. 39, no. 26, pp. 2210–2216, 2018, doi: 10.1002/jcc.25534.
- [78] J. Hanson, K. Paliwal, T. Litfin, Y. Yang, and Y. Zhou, 'Improving prediction of protein secondary structure, backbone angles, solvent accessibility and contact numbers by using predicted contact maps and an ensemble of recurrent and residual convolutional neural networks', *Bioinformatics*, vol. 35, no. 14, pp. 2403–2410, Jul. 2019, doi: 10.1093/bioinformatics/bty1006.
- [79] M. AlQuraishi, 'ProteinNet: a standardized data set for machine learning of protein structure', *BMC Bioinformatics*, vol. 20, no. 1, p. 311, Jun. 2019, doi: 10.1186/s12859-019-2932-0.

- [80] B. Rost and C. Sander, 'Prediction of Protein Secondary Structure at Better than 70% Accuracy', *J. Mol. Biol.*, vol. 232, no. 2, pp. 584–599, Jul. 1993, doi: 10.1006/jmbi.1993.1413.
- [81] J. Moult, K. Fidelis, A. Kryshchuk, T. Schwede, and A. Tramontano, 'Critical assessment of methods of protein structure prediction (CASP)—Round XII', *Proteins Struct. Funct. Bioinforma.*, vol. 86, pp. 7–15, Dec. 2017, doi: 10.1002/prot.25415.
- [82] A. Kryshchuk, T. Schwede, M. Topf, K. Fidelis, and J. Moult, 'Critical Assessment of Methods of Protein Structure Prediction (CASP) – Round XIII', *Proteins*, vol. 87, no. 12, pp. 1011–1020, Dec. 2019, doi: 10.1002/prot.25823.
- [83] L. N. Kinch, A. Kryshchuk, B. Monastyrskyy, and N. V. Grishin, 'CASP13 Target Classification into Tertiary Structure Prediction Categories', *Proteins*, vol. 87, no. 12, pp. 1021–1036, Dec. 2019, doi: 10.1002/prot.25775.
- [84] A. Kryshchuk, T. Schwede, M. Topf, K. Fidelis, and J. Moult, 'Critical Assessment of Methods of Protein Structure Prediction (CASP) – Round XIV', *Proteins*, vol. 89, no. 12, pp. 1607–1617, Dec. 2021, doi: 10.1002/prot.26237.
- [85] A. Zemla, Č. Venclovas, K. Fidelis, and B. Rost, 'A modified definition of SOV, a segment-based measure for protein secondary structure prediction assessment', *Proteins Struct. Funct. Bioinforma.*, vol. 34, no. 2, pp. 220–223, 1999, doi: 10.1002/(SICI)1097-0134(19990201)34:2<220::AID-PROT7>3.0.CO;2-K.
- [86] B. Rost, C. Sander, and R. Schneider, 'Redefining the goals of protein secondary structure prediction', *J. Mol. Biol.*, vol. 235, no. 1, pp. 13–26, Jan. 1994, doi: 10.1016/S0022-2836(05)80007-5.
- [87] T. Liu and Z. Wang, 'SOV_refine: A further refined definition of segment overlap score and its significance for protein structure similarity', *Source Code Biol. Med.*, vol. 13, no. 1, p. 1, Apr. 2018, doi: 10.1186/s13029-018-0068-7.
- [88] S. S. Haykin and S. S. Haykin, *Neural networks and learning machines*, 3rd ed. New York: Prentice Hall, 2009.
- [89] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the theory of neural computation*. in Santa Fe Institute studies in the sciences of complexity, no. v. 1. Redwood City, Calif: Addison-Wesley Pub. Co, 1991.
- [90] C. Neveu, *English: The neuron contains dendrites that receives information, a cell body called the soma, an an axon that sends information. Schwann cells make activity move faster down axon. Synapses allow neurons to activate other neurons. The dendrites receive a signal, the axon hillock funnels the signal to the initial segment and the initial segment triggers the activity (action potential) that is sent along the axon towards the synapse. Please see learnbio.org for interactive version*. 2022. Accessed: Apr. 22, 2026. [Online]. Available: https://commons.wikimedia.org/wiki/File:Anatomy_of_neuron.png
- [91] W. S. McCulloch and W. Pitts, 'A logical calculus of the ideas immanent in nervous activity', *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943, doi: 10.1007/BF02478259.
- [92] F. Rosenblatt, 'The perceptron: A probabilistic model for information storage and organization in the brain', *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.
- [93] P. M. p87 Machine Learning, *Nederlands: Rosenblatt's Perceptron*. 2012. Accessed: Apr. 23, 2026. [Online]. Available: <https://commons.wikimedia.org/wiki/File:Rosenblattperceptron.png>
- [94] M. Minsky and S. Papert, *Perceptrons: an introduction to computational geometry*, Expanded ed. Cambridge, Mass: MIT Press, 1988.
- [95] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

- [96] G. Cybenko, ‘Approximation by superpositions of a sigmoidal function’, *Math. Control Signals Syst.*, vol. 2, no. 4, pp. 303–314, Dec. 1989, doi: 10.1007/BF02551274.
- [97] K. Hornik, M. Stinchcombe, and H. White, ‘Multilayer feedforward networks are universal approximators’, *Neural Netw.*, vol. 2, no. 5, pp. 359–366, Jan. 1989, doi: 10.1016/0893-6080(89)90020-8.
- [98] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, ‘Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark’, Jun. 28, 2022, *arXiv*: arXiv:2109.14545. doi: 10.48550/arXiv.2109.14545.
- [99] C. M. Bishop and H. Bishop, *Deep learning: foundations and concepts*. Cham: Springer, 2024.
- [100] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, ‘Learning representations by back-propagating errors’, *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: 10.1038/323533a0.
- [101] J. Terven, D. M. Cordova-Esparza, A. Ramirez-Pedraza, E. A. Chavez-Urbiola, and J. A. Romero-Gonzalez, ‘Loss Functions and Metrics in Deep Learning’, *Artif. Intell. Rev.*, vol. 58, no. 7, p. 195, Apr. 2025, doi: 10.1007/s10462-025-11198-7.
- [102] D. Bahdanau, K. Cho, and Y. Bengio, ‘Neural Machine Translation by Jointly Learning to Align and Translate’, May 19, 2016, *arXiv*: arXiv:1409.0473. doi: 10.48550/arXiv.1409.0473.
- [103] A. Vaswani *et al.*, ‘Attention Is All You Need’, Aug. 02, 2023, *arXiv*: arXiv:1706.03762. doi: 10.48550/arXiv.1706.03762.
- [104] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, ‘BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding’, May 24, 2019, *arXiv*: arXiv:1810.04805. doi: 10.48550/arXiv.1810.04805.
- [105] Z. Lin *et al.*, ‘Evolutionary-scale prediction of atomic-level protein structure with a language model’, *Science*, vol. 379, no. 6637, pp. 1123–1130, Mar. 2023, doi: 10.1126/science.ade2574.
- [106] P. Dufter, M. Schmitt, and H. Schütze, ‘Position Information in Transformers: An Overview’, Sep. 09, 2021, *arXiv*: arXiv:2102.11090. doi: 10.48550/arXiv.2102.11090.
- [107] K. He, X. Zhang, S. Ren, and J. Sun, ‘Deep Residual Learning for Image Recognition’, Dec. 10, 2015, *arXiv*: arXiv:1512.03385. doi: 10.48550/arXiv.1512.03385.
- [108] J. L. Ba, J. R. Kiros, and G. E. Hinton, ‘Layer Normalization’, Jul. 21, 2016, *arXiv*: arXiv:1607.06450. doi: 10.48550/arXiv.1607.06450.
- [109] S. Ioffe and C. Szegedy, ‘Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift’, Mar. 02, 2015, *arXiv*: arXiv:1502.03167. doi: 10.48550/arXiv.1502.03167.
- [110] R. Xiong *et al.*, ‘On Layer Normalization in the Transformer Architecture’, Jun. 29, 2020, *arXiv*: arXiv:2002.04745. doi: 10.48550/arXiv.2002.04745.
- [111] A. Dosovitskiy *et al.*, ‘An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale’, Jun. 03, 2021, *arXiv*: arXiv:2010.11929. doi: 10.48550/arXiv.2010.11929.
- [112] Z. Lin *et al.*, ‘Language models of protein sequences at the scale of evolution enable accurate structure prediction’, Jul. 21, 2022, *bioRxiv*. doi: 10.1101/2022.07.20.500902.
- [113] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, ‘Dropout: A Simple Way to Prevent Neural Networks from Overfitting’, *J. Mach. Learn. Res.*, vol. 15, no. 56, pp. 1929–1958, 2014.
- [114] I. Loshchilov and F. Hutter, ‘Decoupled Weight Decay Regularization’, Jan. 04, 2019, *arXiv*: arXiv:1711.05101. doi: 10.48550/arXiv.1711.05101.

- [115] J. Duchi, E. Hazan, and Y. Singer, ‘Adaptive Subgradient Methods for Online Learning and Stochastic Optimization’, *J. Mach. Learn. Res.*, vol. 12, no. 61, pp. 2121–2159, 2011.
- [116] T. Tieleman and G. Hinton, ‘Lecture 6.5 — RMSProp: Divide the gradient by a running average of its recent magnitude’. 2012.
- [117] D. P. Kingma and J. Ba, ‘Adam: A Method for Stochastic Optimization’, Jan. 30, 2017, *arXiv*: arXiv:1412.6980. doi: 10.48550/arXiv.1412.6980.
- [118] I. Loshchilov and F. Hutter, ‘SGDR: Stochastic Gradient Descent with Warm Restarts’, May 03, 2017, *arXiv*: arXiv:1608.03983. doi: 10.48550/arXiv.1608.03983.
- [119] A. Mohammed and R. Kora, ‘A comprehensive review on ensemble deep learning: Opportunities and challenges’, *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 35, no. 2, pp. 757–774, Feb. 2023, doi: 10.1016/j.jksuci.2023.01.014.
- [120] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, Fourth edition. in Pearson series in artificial intelligence. Hoboken: Pearson, 2021.
- [121] M. A. Ganaie, M. Hu, A. K. Malik, M. Tanveer, and P. N. Suganthan, ‘Ensemble deep learning: A review’, 2021, doi: 10.48550/ARXIV.2104.02395.
- [122] Jeremybeauchamp, *English: A visual comparison between the complexity of decision trees and random forests*. 2020. Accessed: Apr. 29, 2026. [Online]. Available: https://commons.wikimedia.org/wiki/File:Decision_Tree_vs._Random_Forest.png
- [123] M. Steinegger and J. Söding, ‘MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets’, *Nat. Biotechnol.*, vol. 35, no. 11, pp. 1026–1028, Nov. 2017, doi: 10.1038/nbt.3988.
- [124] A. Rives *et al.*, ‘Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences’, *Proc. Natl. Acad. Sci.*, vol. 118, no. 15, p. e2016239118, Apr. 2021, doi: 10.1073/pnas.2016239118.
- [125] Y. Kim and J. Kwon, ‘AttSec: protein secondary structure prediction by capturing local patterns from attention map’, *BMC Bioinformatics*, vol. 24, no. 1, p. 183, May 2023, doi: 10.1186/s12859-023-05310-3.
- [126] A. Elnaggar *et al.*, ‘ProtTrans: Towards Cracking the Language of Life’s Code Through Self-Supervised Deep Learning and High Performance Computing’. 2021. [Online]. Available: <https://arxiv.org/abs/2007.06225>
- [127] P. Kountouris *et al.*, ‘A Comparative Study on Filtering Protein Secondary Structure Prediction’, *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 9, no. 3, pp. 731–739, 2012, doi: 10.1109/TCBB.2012.22.
- [128] A. A. Salamov and V. V. Solovyev, ‘Prediction of Protein Secondary Structure by Combining Nearest-neighbor Algorithms and Multiple Sequence Alignments’, *J. Mol. Biol.*, vol. 247, no. 1, pp. 11–15, Mar. 1995, doi: 10.1006/jmbi.1994.0116.
- [129] M. J. Wood and J. D. Hirst, ‘Protein secondary structure prediction with dihedral angles’, *Proteins Struct. Funct. Bioinforma.*, vol. 59, no. 3, pp. 476–481, May 2005, doi: 10.1002/prot.20435.
- [130] A. Dionysiou, M. Agathocleous, C. Christodoulou, and V. Promponas, ‘Convolutional Neural Networks in Combination with Support Vector Machines for Complex Sequential Data Classification’, in *Artificial Neural Networks and Machine Learning – ICANN 2018*, V. Kůrková, Y. Manolopoulos, B. Hammer, L. Iliadis, and I. Maglogiannis, Eds, Cham: Springer International Publishing, 2018, pp. 444–455.
- [131] K. Charalampous, M. Agathocleous, C. Christodoulou, and V. Promponas, ‘Solving the Protein Secondary Structure Prediction Problem With the Hessian Free Optimization Algorithm’, *IEEE Access*, vol. 10, pp. 27759–27770, 2022, doi: 10.1109/ACCESS.2022.3156888.

- [132] S. Chatzimiltis, M. Agathocleous, V. J. Promponas, and C. Christodoulou, ‘Post-processing enhances protein secondary structure prediction with second order deep learning and embeddings’, *Comput. Struct. Biotechnol. J.*, vol. 27, pp. 243–251, Jan. 2025, doi: 10.1016/j.csbj.2024.12.022.
- [133] Y. Chen, G. Chen, and C. Y.-C. Chen, ‘MFTrans: A multi-feature transformer network for protein secondary structure prediction’, *Int. J. Biol. Macromol.*, vol. 267, p. 131311, May 2024, doi: 10.1016/j.ijbiomac.2024.131311.
- [134] D. P. Ismi, R. Pulungan, and Afiahayati, ‘Self-attention and asymmetric multi-layer perceptron-gated recurrent unit blocks for protein secondary structure prediction’, *Appl. Soft Comput.*, vol. 159, p. 111604, Jul. 2024, doi: 10.1016/j.asoc.2024.111604.
- [135] A. Paszke *et al.*, ‘PyTorch: An Imperative Style, High-Performance Deep Learning Library’, 2019, *arXiv*. doi: 10.48550/ARXIV.1912.01703.
- [136] T. Wolf *et al.*, ‘Transformers: State-of-the-Art Natural Language Processing’, in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Q. Liu and D. Schlangen, Eds, Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. doi: 10.18653/v1/2020.emnlp-demos.6.
- [137] R. Schmirler, M. Heinzinger, and B. Rost, ‘Fine-tuning protein language models boosts predictions across diverse tasks’, *Nat. Commun.*, vol. 15, no. 1, p. 7407, Aug. 2024, doi: 10.1038/s41467-024-51844-2.
- [138] A. Saadat and J. Fellay, ‘Fine-tuning protein language models to understand the functional impact of missense variants’, *Comput. Struct. Biotechnol. J.*, vol. 27, pp. 2199–2207, Jan. 2025, doi: 10.1016/j.csbj.2025.05.022.
- [139] F. Pedregosa *et al.*, ‘Scikit-learn: Machine Learning in Python’, 2012, doi: 10.48550/ARXIV.1201.0490.
- [140] Y. Zhang, C. Chen, T. Ding, Z. Li, R. Sun, and Z.-Q. Luo, ‘Why Transformers Need Adam: A Hessian Perspective’, Oct. 21, 2024, *arXiv*: arXiv:2402.16788. doi: 10.48550/arXiv.2402.16788.
- [141] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, ‘Optuna: A Next-generation Hyperparameter Optimization Framework’, in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, in KDD ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 2623–2631. doi: 10.1145/3292500.3330701.
- [142] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, ‘Algorithms for Hyper-Parameter Optimization’, in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds, Curran Associates, Inc., 2011. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf
- [143] S. Watanabe, ‘Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance’, 2023, *arXiv*. doi: 10.48550/ARXIV.2304.11127.
- [144] V. Ruscio, U. Nanni, and F. Silvestri, ‘What are you sinking? A geometric approach on attention sink’, Aug. 04, 2025, *arXiv*: arXiv:2508.02546. doi: 10.48550/arXiv.2508.02546.
- [145] I. T. Jolliffe and J. Cadima, ‘Principal component analysis: a review and recent developments’, *Philos. Trans. R. Soc. Math. Phys. Eng. Sci.*, vol. 374, no. 2065, p. 20150202, Apr. 2016, doi: 10.1098/rsta.2015.0202.

- [146] M. Shapovalov, R. L. Dunbrack, and S. Vucetic, ‘Multifaceted analysis of training and testing convolutional neural networks for protein secondary structure prediction’, *PLOS ONE*, vol. 15, no. 5, p. e0232528, May 2020, doi: 10.1371/journal.pone.0232528.

Appendices

Appendix I — Mapping Module

This section presents the module responsible for converting the DSSP labels into predefined indices and vice versa, while also reducing the 8-state labels to a 3-state representation.

```
import numpy as np

##### DSSP label → integer mappings (Q3 / Q8) #####
def preprocess_tgt_Q3(seq: str):
    mapping = {'H': 0, 'E': 1, 'C': 2}
    return [mapping[aa] for aa in seq]

def preprocess_tgt_Q8(seq: str):
    # DSSP 8-state alphabet
    mapping = {
        'G': 0, # 3-turn helix (310 helix)
        'H': 1, # Alpha helix
        'I': 2, # 5-turn helix (pi helix)
        'B': 3, # Beta bridge
        'E': 4, # Extended strand (beta sheet)
        'S': 5, # Bend
        'T': 6, # Turn
        'C': 7 # Coil/loop
    }
    return [mapping[aa] for aa in seq]

##### Q8 → Q3 reduction (vectorised lookup table) #####
Q8_TO_Q3 = np.array([0, 0, 0, 1, 1, 2, 2, 2], dtype=np.int8)

def reduce_q8_to_q3(labels: np.ndarray) -> np.ndarray:
    """Map a Q8 integer label array to Q3 integer labels."""
    return Q8_TO_Q3[labels]

##### Integer → DSSP character (inverse of the mappings above) #####
def parse_int_to_q3(i: int) -> str:
    return {0: 'H', 1: 'E', 2: 'C'}[i]

def parse_int_to_q8(i: int) -> str:
    return {0: 'G', 1: 'H', 2: 'I', 3: 'B', 4: 'E', 5: 'S', 6: 'T', 7: 'C'}[i]
```

Appendix II — ESM-2 Embedding Extraction

This section describes the module for initializing ESM-2, extracting embeddings from a defined dataset, and storing them for subsequent use in training and evaluation.

```
import os
import torch
import numpy as np
import pandas as pd
from tqdm import tqdm
from preprocessing import preprocess_tgt_Q3, preprocess_tgt_Q8
from config import Config

##### Module-level configuration #####
cfg = Config()
device = cfg['DEVICE']
data_dir = cfg['PATHS']['data_dir']
embed_dir = cfg['PATHS']['embed_dir']

##### ESM-2 Protein Language Model initialisation #####
def init_ESM2(model_name):
    from transformers import AutoModel, AutoTokenizer
    print(f"Loading ESM-2 model ({model_name}).")

    # Instantiate tokenizer + frozen encoder
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    esm_model = AutoModel.from_pretrained(model_name).to(device)

    # Freeze parameters and set eval mode
    for param in esm_model.parameters():
        param.requires_grad = False
    esm_model.eval()

    return tokenizer, esm_model

##### Plain-text dataset parser (FASTA-like: name / seq / target triplets) #####
def create_dataset_as_df(file_path: str):
    assert isinstance(file_path, str), "file_path must be str."
    assert os.path.exists(file_path), f"File path doesn't exist: {file_path}"

    with open(file_path, 'r', encoding='utf-8') as f:
```

```

        text = f.read()

    rows      = []
    pid_count = {}
    seq_name = seq = tgt = expecting = None

    for raw_line in text.splitlines():
        line = raw_line.strip()
        if not line:
            continue

        if line.startswith('>'):
            raw_pid = line.replace(' ', '').replace('>', '')

            # Duplicate-PID handling
            if raw_pid in pid_count:
                pid_count[raw_pid] += 1
                seq_name = f"{raw_pid}-{pid_count[raw_pid]}"
                print(f" [duplicate PID] '{raw_pid}' → '{seq_name}'")
            else:
                pid_count[raw_pid] = 0
                seq_name = raw_pid

            expecting = 'seq'

            elif expecting == 'seq':
                seq      = line.replace(' ', '')
                expecting = 'target'
            elif expecting == 'target':
                tgt      = line.replace(' ', '')
                rows.append({'name': seq_name, 'seq': seq, 'tgt': tgt})
                expecting = None

    n_total = len(rows)
    n_unique = len(pid_count)
    print(f"Dataset loaded: {n_total} segments ({n_unique} unique PIDs,
{n_total - n_unique} duplicates renamed)")
    return pd.DataFrame(rows)

##### ESM-2 embedding extraction → compressed NPZ archive #####
def extract_embeddings_to_npz(dataset: pd.DataFrame, out_filename: str,
state: int):
    # Load PLM
    tokenizer, model = init_ESM2(model_name=cfg['ESM2']['model_name'])

    all_embeddings, all_labels, all_chain_ids = [], [], []

```

```

# Per-sample loop
for i in tqdm(range(len(dataset)), desc="Processing samples"):
    chain_id = dataset['name'].iloc[i]
    sequence = dataset['seq'].iloc[i]
    target = dataset['tgt'].iloc[i]

    # Map DSSP characters to integer labels
    if state == 3:
        labels = preprocess_tgt_Q3(target)
    elif state == 8:
        labels = preprocess_tgt_Q8(target)

    # Forward through ESM-2, dropping <BOS>/<EOS> rows
    with torch.no_grad():
        encoded = tokenizer(sequence,
return_tensors='pt').to(device)
        output = model(**encoded)
        embeddings = output.last_hidden_state[0, 1:-1, :] # [L,
1280]

        embeddings_np = embeddings.cpu().numpy().astype(np.float32)
        labels_np = np.array(labels, dtype=np.int8)

    # Sanity: per-residue alignment
    assert len(embeddings_np) == len(labels_np), \
        f"Embedding/label length mismatch for {chain_id}:
{len(embeddings_np)} vs {len(labels_np)}"

    all_embeddings.append(embeddings_np)
    all_labels.append(labels_np)
    all_chain_ids.append(np.array([chain_id] * len(labels_np)))

    del encoded, output, embeddings
    torch.cuda.empty_cache()

# Flatten across all proteins to a residue-level record
embeddings_arr = np.concatenate(all_embeddings, axis=0) # [N,
1280]
labels_arr = np.concatenate(all_labels, axis=0) # [N]
chain_ids_arr = np.concatenate(all_chain_ids, axis=0) # [N]

print(f"Total residues: {len(embeddings_arr)}")

output_path = out_filename + ".npz"
np.savez_compressed(output_path,

```

```

        embeddings = embeddings_arr,
        labels      = labels_arr,
        chain_ids   = chain_ids_arr)

    print(f"Saved: {output_path}")

##### Driver #####
def main():
    # List of datasets to extract PLM embeddings for
    datasets = [
        # Examples
        "PISCES_3class/CASP13.txt",
        "NetsurfP_3class/CB513.txt",
        "NetsurfP_8class/CB513.txt"
    ]

    for file_path in datasets:
        # Infer the state (3 or 8) from path affix
        state = None
        if '8class' in file_path:
            state = 8
        elif '3class' in file_path:
            state = 3
        else:
            raise Exception("Make sure filepath contains affix '_3class'
or '_8class'.")

        # Parse text file into dataframe
        dataset = create_dataset_as_df(os.path.join(data_dir,
file_path))

        filename      = file_path.split('/')[-1]
        base_name     = filename.split('.')[0]
        output_path   = os.path.join(embed_dir,
f"{base_name}_{state}class")
        os.makedirs(os.path.dirname(output_path), exist_ok=True)

        # Embed + save
        extract_embeddings_to_npz(
            dataset      = dataset,
            out_filename = output_path,
            state        = state
        )

        del dataset
        torch.cuda.empty_cache()

```

```

        print("All datasets processed successfully.")

if __name__ == "__main__":
    main()

```

Appendix III — Transformer Encoder Implementation

This section presents the PyTorch implementation of the predictive model used in this work, along with a general ensemble module class, that combines multiple base model classifiers and averages their output logits.

```

"""

```

Encoder Transformer for Protein Secondary Structure Prediction

This module implements an Encoder-Only Transformer architecture for predicting protein secondary structures from Protein Language Model (PLM) embeddings. Furthermore, an Ensemble class is implemented that combines multiple Encoder-Only models and averages their probabilities for the final output.

Author: Stefanos Englezou

Last Modified: 28/05/2026

```

"""

```

```

import torch
import torch.nn as nn
from math import sqrt, log
from config import Config

```

Module-level configuration

```

cfg          = Config()
device       = cfg["DEVICE"]
embed_dim    = cfg['ESM2']['embed_dim']

```

Base-model factory

```

def build_base_model(params):

```

```

    """

```

Instantiate a PSSPEncoder on the configured device from a hyperparameter dictionary.

Parameters

params : dict

Hyperparameter dictionary that MUST contain the following keys:
'embed_dim', 'tokens', 'd_ff_multiplier', 'num_heads', 'layers',
'mlp_multiplier', 'dropout', and 'num_classes'.

Returns

PSSPEncoder

A freshly initialised PSSPEncoder model moved to the global `device`.

"""

return PSSPEncoder(

embed_dim = params['embed_dim'],
 tokens = params['tokens'],
 d_ff_multiplier = params['d_ff_multiplier'],
 num_heads = params['num_heads'],
 layers = params['layers'],
 mlp_multiplier = params['mlp_multiplier'],
 dropout = params['dropout'],
 num_classes = params['num_classes']

).to(device)

PSSPEncoder -- Encoder-only Transformer for per-residue PSSP

class PSSPEncoder(nn.Module):

"""

The class of the encoder-transformer for protein secondary structure prediction.

The model takes the embedding from the PLM of a position, slices it into a number

of tokens, adds positional information to each token and through a number of layers

it performs self-attention. After capturing intra-relationships between the slices,

mean pooling is applied and forwarded to the final MLP head.

Attributes

tokens : int

The number of tokens the PLM embedding is sliced into.

token_dim : int

The dimension of each token (embed_dim // tokens).

pos_e : PositionalEncoding

Positional Embeddings Module.

layers : nn.ModuleList

List of Encoder Layer modules defined by a given number of blocks.

```

mlp_head : nn.Sequential
    Feed-forward network with two linear layers and GELU activation function.
"""
def __init__(self,
              embed_dim: int          = 1280,
              tokens: int              = 5,
              d_ff_multiplier: int     = 2,
              num_heads: int           = 4,
              layers: int              = 1,
              mlp_multiplier: int      = 2,
              num_classes: int         = 3,
              dropout: float           = 0.1):
    """
    Constructor of PSSPEncoder.

    Parameters
    -----
    embed_dim : int, default=1280
        The embedding dimension given by the Protein Language Model (PLM).
    tokens : int, default=5
        The number of equally sized tokens to slice the embedding vector.
    d_ff_multiplier : int, default=2
        The multiplier to be applied to the token dimension for the MLP inside
a Layer.
    num_heads : int, default=4
        The number of heads for the self-attention module.
    layers : int, default=1
        The number of layers in the Transformer.
    mlp_multiplier : int, default=2
        The multiplier to be applied to the token dimension for the final MLP
head.
    num_classes : int, default=3
        The number of classes to predict.
    dropout : float, default=0.1
        Value of dropout to be applied.
    """
    super().__init__()
    # Argument validation
    assert isinstance(embed_dim, int) and embed_dim > 0, "embed_dim must be a
positive integer."
    assert isinstance(tokens, int) and tokens > 0, "tokens must be a positive
integer."
    assert embed_dim % tokens == 0, "embed_dim must be divisible by tokens."
    assert isinstance(d_ff_multiplier, int) and d_ff_multiplier >= 1,
"d_ff_multiplier must be an integer starting from 1."
    assert isinstance(num_heads, int) and num_heads > 0, "num_heads must be a

```

```

positive integer."
    assert (embed_dim // tokens) % num_heads == 0, "token_dim (embed_dim //
tokens) must be divisible by num_heads."
    assert isinstance(layers, int) and layers >= 1, "layers must be an integer
starting from 1."
    assert isinstance(mlp_multiplier, int) and mlp_multiplier >= 1,
"mlp_multiplier must be an integer starting from 1."
    assert isinstance(num_classes, int) and num_classes in [3, 8],
"num_classes must be either 3 or 8"
    assert isinstance(dropout, float) and dropout >= 0.0 and dropout < 1.0,
"dropout must be a float in the range [0.0, 1.0)."
```

Derived dimensions

```

self.tokens      = tokens
self.token_dim   = embed_dim // tokens
d_ff             = d_ff_multiplier * self.token_dim
mlp_dim          = mlp_multiplier * self.token_dim
```

Components: positional encoding, stack, classification head

```

self.pos_e = PositionalEncoding(self.token_dim, tokens, dropout)
self.layers = nn.ModuleList([
    Layer(self.token_dim, d_ff, num_heads, dropout) for _ in range(layers)
])
self.mlp_head = nn.Sequential(
    nn.Linear(self.token_dim, mlp_dim),
    nn.GELU(),
    nn.Linear(mlp_dim, num_classes)
)
```

```

def forward(self, x: torch.Tensor):
    """
    Forward pass through PSSP Encoder.

    This method takes the given PLM embedding of a certain residue,
    reshapes it into token slices, adds absolute positional embeddings,
    then forwards through each layer to self-attend between the
    embedding slices. After that the mean is taken across the token
    dimension and passed into the final MLP head for classification.

    Parameters
    -----
    x : torch.Tensor
        Input tensor of shape (batch_size, embed_dim)
        with dtype torch.float32.

    Returns
    """
```

```

-----
tuple of (torch.Tensor, list)
    logits : torch.Tensor
        Output tensor of shape (batch_size, num_classes)
        containing the classification logits.
    encoder_attention_weights : list of torch.Tensor
        List of attention weight tensors, one per layer, each of
        shape (batch_size, num_heads, tokens, tokens).
"""
assert isinstance(x, torch.Tensor), "x must be a torch.Tensor."
assert x.dtype == torch.float32, "x must have dtype torch.float32."
assert x.dim() == 2, "x must be a 2D tensor of shape (batch_size,
embed_dim)."

# Slice PLM embedding into [batch_size, tokens, token_dim]
x = x.reshape(x.shape[0], self.tokens, self.token_dim)

# Add positional information
x = self.pos_e(x)

# Encoder stack
encoder_attention_weights = []
for layer in self.layers:
    x, attn_w = layer(x)
    encoder_attention_weights.append(attn_w)

# Mean-pool over tokens, classify
logits = self.mlp_head(torch.mean(x, dim=1))

return logits, encoder_attention_weights

##### Encoder block (multi-head self-attention + FFN, post-norm) #####
class Layer(nn.Module):
    """
    Transformer encoder layer.

    This class implements a single transformer encoder layer consisting
    of multi-head self-attention followed by a Feed Forward Network (FFN).

    Attributes
    -----
    attention : MultiHeadAttention
        Multi-head self-attention mechanism.
    mlp : nn.Sequential
        Feed-forward network with two linear layers and ReLU activation function.
    layer_norm_1 : nn.LayerNorm

```

```

        Layer normalization applied after attention.
    layer_norm_2 : nn.LayerNorm
        Layer normalization applied after MLP.
    dropout : nn.Dropout
        Dropout layer for regularization.
    """
    def __init__(self, d_model: int, d_ff: int, num_heads: int, dropout: float =
0.1):
        """
        Constructor of encoder layer.

        Parameters
        -----
        d_model : int
            Dimension of the model embeddings.
        d_ff : int
            Dimension of the feed-forward network hidden layer.
        num_heads : int
            Number of attention heads in multi-head attention.
        dropout : float, default=0.1
            Dropout probability for regularization. Must be between 0 and 1.
        """
        super().__init__()
        # Argument validation
        assert isinstance(d_model, int) and d_model > 0, "d_model must be a
positive integer."
        assert isinstance(d_ff, int) and d_ff > 0, "d_ff must be a positive
integer."
        assert isinstance(num_heads, int) and num_heads > 0, "num_heads must be a
positive integer."
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads."
        assert isinstance(dropout, float) and dropout >= 0.0 and dropout < 1.0,
"ddropout must be a float in the range [0.0, 1.0).".

        # Sub-modules
        self.attention = MultiHeadAttention(d_model, num_heads)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )
        self.layer_norm_1 = nn.LayerNorm(d_model)
        self.layer_norm_2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor):

```

```

"""
Forward pass through encoder layer.

This method applies the full encoder layer operations. It is
important to note that Layer-Normalization is applied after each
sub-layer (following post-norm approach).

The flow follows:
x -> MultiHeadAttention -> Dropout -> Add(x) -> LayerNorm -> MLP ->
Dropout -> Add -> LayerNorm -> output

Parameters
-----
x : torch.Tensor
    Input tensor of shape (batch_size, seq_len, d_model)
    with dtype torch.float32.

Returns
-----
tuple of (torch.Tensor, torch.Tensor)
    x : torch.Tensor
        Output tensor of shape (batch_size, seq_len, d_model)
        after applying attention and MLP with residual connections.
    attn_weights : torch.Tensor
        Attention weight tensor of shape
        (batch_size, num_heads, seq_len, seq_len).
"""
assert isinstance(x, torch.Tensor), "x must be a torch.Tensor."
assert x.dtype == torch.float32, "x must have dtype torch.float32."
assert x.dim() == 3, "x must be a 3D tensor of shape (batch_size, seq_len,
d_model)."

# Self-attention sub-layer
attn_out, attn_weights = self.attention((x, x, x))
x = self.layer_norm_1(x + self.dropout(attn_out))

# Feed-forward sub-layer
mlp_out = self.mlp(x)
x = self.layer_norm_2(x + self.dropout(mlp_out))

return x, attn_weights

##### Multi-Head Self-Attention #####
class MultiHeadAttention(nn.Module):
    """
    Multi-head self-attention mechanism.

```

This class implements the multi-head attention mechanism from "Attention is All You Need" (Vaswani et al., 2017). The input is projected into query, key, and value representations, which are then split into multiple heads. Scaled dot-product attention is computed in parallel for each head, and the results are concatenated and projected to produce the final output.

The attention mechanism allows the model to focus on different positions and representation subspaces simultaneously, capturing complex relationships in the input data.

Attributes

```
num_heads : int
    Number of attention heads.
d_k : int
    Dimension of each attention head (d_model / num_heads).
sqrt_d_k : float
    Square root of d_k used for scaling attention scores.
w_q : nn.Linear
    Linear projection for query.
w_k : nn.Linear
    Linear projection for key.
w_v : nn.Linear
    Linear projection for value.
w_o : nn.Linear
    Output projection after concatenating heads.
"""
```

```
def __init__(self, d_model: int, num_heads: int):
    """
```

```
        Constructor of multi-head attention module.
```

This constructor creates the linear projection layers for query, key, value, and output. The model dimension is split across multiple heads, with each head having dimension `d_k`. All projections are initialized without bias terms.

Parameters

```
d_model : int
    Dimension of the model embeddings. Must be a positive
    integer divisible by num_heads.
num_heads : int
    Number of attention heads. Must be a positive integer and
    d_model must be divisible by num_heads to ensure each head
```

```

        has integer dimension.
    """
    super().__init__()
    # Argument validation
    assert isinstance(d_model, int) and d_model > 0, "d_model must be a
positive integer."
    assert isinstance(num_heads, int) and num_heads > 0, "num_heads must be a
positive integer."
    assert d_model % num_heads == 0, "d_model must be divisible by num_heads."

    self.num_heads = num_heads
    self.d_k = d_model // num_heads
    self.sqrt_d_k = sqrt(self.d_k)

    # Query / Key / Value / Output projections (bias-free)
    self.w_q = nn.Linear(d_model, d_model, bias=False)
    self.w_k = nn.Linear(d_model, d_model, bias=False)
    self.w_v = nn.Linear(d_model, d_model, bias=False)
    self.w_o = nn.Linear(d_model, d_model, bias=False)

def scaled_dot_product_attention(self, Q, K, V):
    """
    Perform scaled dot-product attention.

    This method computes attention weights by taking the dot product
    of queries and keys, scaling by the square root of the key dimension,
    computing softmax, and finally multiplying by values.

    The attention formula is:
    Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V

    Parameters
    -----
    Q : torch.Tensor
        Query tensor of shape (batch_size, num_heads, tgt_len, d_k)
        with dtype torch.float32.
    K : torch.Tensor
        Key tensor of shape (batch_size, num_heads, src_len, d_k)
        with dtype torch.float32.
    V : torch.Tensor
        Value tensor of shape (batch_size, num_heads, src_len, d_k)
        with dtype torch.float32.

    Returns
    -----
    tuple of (torch.Tensor, torch.Tensor)

```

```

        attention_out : torch.Tensor
            Attention output of shape (batch_size, num_heads, tgt_len, d_k)
            representing the weighted combination of values based on
            attention weights.
        attention_weights : torch.Tensor
            Attention weight matrix of shape
            (batch_size, num_heads, tgt_len, src_len).
    """
    assert isinstance(Q, torch.Tensor), "Q must be a torch.Tensor."
    assert isinstance(K, torch.Tensor), "K must be a torch.Tensor."
    assert isinstance(V, torch.Tensor), "V must be a torch.Tensor."
    assert Q.dtype == torch.float32, "Q must have dtype torch.float32."
    assert K.dtype == torch.float32, "K must have dtype torch.float32."
    assert V.dtype == torch.float32, "V must have dtype torch.float32."
    assert Q.dim() == 4, "Q must be a 4D tensor of shape (batch_size,
num_heads, tgt_len, d_k)."
    assert K.dim() == 4, "K must be a 4D tensor of shape (batch_size,
num_heads, src_len, d_k)."
    assert V.dim() == 4, "V must be a 4D tensor of shape (batch_size,
num_heads, src_len, d_k)."

    # QK^T / sqrt(d_k)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / self.sqrt_d_k #
[batch_size, num_heads, tgt_len, src_len]

    # Row-wise softmax
    attention_weights = torch.softmax(scores, dim=-1)

    # Weighted sum over values
    attention_out = torch.matmul(attention_weights, V)

    return attention_out, attention_weights

def forward(self, QKV: tuple):
    """
    Forward pass through multi-head attention.

    This method applies multi-head attention by:
    1. Projecting inputs to query, key, value representations
    2. Splitting projections into multiple heads
    3. Computing scaled dot-product attention for each head
    4. Concatenating head outputs
    5. Applying final output projection

    For self-attention, all three inputs (query, key, value) are
    typically the same tensor.

```

Parameters

QKV : tuple of (torch.Tensor, torch.Tensor, torch.Tensor)

 Tuple of three tensors (query, key, value), each with
 dtype torch.float32. Query has shape
 (batch_size, tgt_len, d_model), key and value have shape
 (batch_size, src_len, d_model). For self-attention, all
 three are identical.

Returns

tuple of (torch.Tensor, torch.Tensor)

 output : torch.Tensor
 Output tensor of shape (batch_size, tgt_len, d_model)
 after applying multi-head attention and output projection.
 attention_weights : torch.Tensor
 Attention weight matrix of shape
 (batch_size, num_heads, tgt_len, src_len).

"""

Argument validation

assert isinstance(QKV, tuple) and len(QKV) == 3, "QKV must be a tuple of
three tensors (query, key, value)."

query, key, value = QKV

assert isinstance(query, torch.Tensor), "query must be a torch.Tensor."

assert isinstance(key, torch.Tensor), "key must be a torch.Tensor."

assert isinstance(value, torch.Tensor), "value must be a torch.Tensor."

assert query.dtype == torch.float32, "query must have dtype

torch.float32."

assert key.dtype == torch.float32, "key must have dtype torch.float32."

assert value.dtype == torch.float32, "value must have dtype

torch.float32."

assert query.dim() == 3, "query must be a 3D tensor of shape (batch_size,
tgt_len, d_model)."

assert key.dim() == 3, "key must be a 3D tensor of shape (batch_size,
src_len, d_model)."

assert value.dim() == 3, "value must be a 3D tensor of shape (batch_size,
src_len, d_model)."

batch_size, tgt_len, d_model = query.shape

src_len = key.shape[1]

Project + split heads → [batch_size, num_heads, seq_len, d_k]

Q = self.w_q(query).view(batch_size, tgt_len, self.num_heads,
self.d_k).transpose(1, 2)

K = self.w_k(key).view(batch_size, src_len, self.num_heads,

```

self.d_k).transpose(1, 2)
    V = self.w_v(value).view(batch_size, src_len, self.num_heads,
self.d_k).transpose(1, 2)

    # Scaled dot-product attention
    attention_out, attention_weights = self.scaled_dot_product_attention(Q, K,
V) # [batch_size, num_heads, tgt_len, d_k]

    # Concatenate heads → [batch_size, tgt_len, d_model]
    attention_out = attention_out.transpose(1,
2).contiguous().view(batch_size, tgt_len, d_model)

    # Output projection
    return self.w_o(attention_out), attention_weights

```

Sinusoidal Positional Encoding (Vaswani et al., 2017)

```

class PositionalEncoding(nn.Module):
    """
    Sinusoidal positional encoding.

    This class implements fixed sinusoidal positional encodings as
    described in "Attention is All You Need" (Vaswani et al., 2017).
    The encodings are added to input embeddings to provide absolute
    position information to the transformer.

    Attributes
    -----
    pe : torch.Tensor
        Sinusoidal positional encoding buffer of shape
        (1, max_len, d_model). Registered as a non-trainable buffer.
    dropout : nn.Dropout
        Dropout layer applied after adding positional encodings.
    """
    def __init__(self, d_model: int, max_len: int, dropout: float = 0.1):
        """
        Constructor of sinusoidal positional encoding.

        This constructor precomputes the sinusoidal positional encoding
        matrix and registers it as a buffer. The encoding uses sine
        for even dimensions and cosine for odd dimensions, with
        frequencies decreasing geometrically across dimensions.

        Parameters
        -----
        d_model : int
            Dimension of the model embeddings. Positional encodings

```

```

        have the same dimension and are added element-wise.
    max_len : int
        Maximum sequence length supported. Determines the number
        of precomputed positional encoding vectors.
    dropout : float, default=0.1
        Dropout probability applied after adding positional
        encodings. Must be between 0 and 1.
    """
    super().__init__()
    # Argument validation
    assert isinstance(d_model, int) and d_model > 0, "d_model must be a
positive integer."
    assert isinstance(max_len, int) and max_len > 0, "max_len must be a
positive integer."
    assert isinstance(dropout, float) and dropout >= 0.0 and dropout < 1.0,
"d dropout must be a float in the range [0.0, 1.0)."

    self.dropout = nn.Dropout(p=dropout)

    # Precompute the sinusoidal table and register as buffer
    pe = torch.zeros(max_len, d_model)
    position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-log(10000.0)
/ d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    self.register_buffer('pe', pe.unsqueeze(0)) # [1, max_len, d_model]

def forward(self, x: torch.Tensor):
    """
    Add positional encodings to input embeddings.

    This method adds the precomputed sinusoidal positional
    encodings to the input embeddings and applies dropout.

    Parameters
    -----
    x : torch.Tensor
        Input embeddings of shape (batch_size, seq_len, d_model)
        with dtype torch.float32.

    Returns
    -----
    torch.Tensor
        Embeddings with positional information added, of shape
        (batch_size, seq_len, d_model), after dropout.

```

```

    """
    assert isinstance(x, torch.Tensor), "x must be a torch.Tensor."
    assert x.dtype == torch.float32, "x must have dtype torch.float32."
    assert x.dim() == 3, "x must be a 3D tensor of shape (batch_size, seq_len,
d_model)."

    # Add positional information and apply dropout
    return self.dropout(x + self.pe[:, :x.size(1), :])

##### Seed Ensemble (averages logits across independently trained models)
#####
class EnsemblePSSP(nn.Module):
    """
    Ensemble of PSSP Encoder models for protein secondary structure prediction.

    This class combines multiple PSSP Encoder models and averages their output
    probabilities to produce a final prediction. Each model is independently
    loaded from a saved state dictionary and set to evaluation mode with frozen
    parameters. The ensemble approach reduces variance and improves robustness
    compared to a single model.

    Attributes
    -----
    ensemble : nn.ModuleList
        List of PSSP Encoder models loaded from the provided state dictionaries.
    """
    def __init__(self, state: str, state_dict_paths: list[str]):
        """
        Constructor of EnsemblePSSP.

        This constructor instantiates a set of PSSP Encoder models using the
        hyperparameters defined in the configuration for the given Q value.
        Each model is then loaded with its corresponding state dictionary,
        set to evaluation mode, and its parameters are frozen to prevent
        any gradient updates during inference.

        Parameters
        -----
        state : str
            Selector that picks which hyperparameter block to load from the
            configuration file. Must be one of ``"3class"`` or ``"8class"``;
            the constructor looks up ``MODEL_CONFIG[f"{state}_params"]`` to
            obtain ``tokens``, ``d_ff_multiplier``, ``num_heads``, ``layers``,
            ``mlp_multiplier``, ``dropout`` and ``num_classes`` for every
            member of the ensemble.
        state_dict_paths : list of str

```

```

        List of file paths to the saved state dictionaries, one per
        model in the ensemble. The number of models in the ensemble
        equals the length of this list.
    """
    super().__init__()
    # Argument validation
    assert isinstance(state, str), "state must be str"
    assert isinstance(state_dict_paths, list) and len(state_dict_paths) > 0,
"state_dict_paths must be a non-empty list."
    assert all(isinstance(p, str) for p in state_dict_paths), "All entries in
state_dict_paths must be strings."

    # Retrieve hyperparameters from configuration
    params = dict(cfg["MODEL_CONFIG"][f"{state}_params"])
    params.setdefault('embed_dim', embed_dim)

    # Instantiate one PSSP Encoder per checkpoint
    self.ensemble = nn.ModuleList(
        PSSP Encoder(
            embed_dim      = params['embed_dim'],
            tokens          = params['tokens'],
            d_ff_multiplier = params['d_ff_multiplier'],
            num_heads       = params['num_heads'],
            layers          = params['layers'],
            mlp_multiplier  = params['mlp_multiplier'],
            dropout         = params['dropout'],
            num_classes     = params['num_classes']
        )
        for _ in range(len(state_dict_paths))
    )

    # Load each checkpoint, set eval mode, freeze parameters
    for model, path in zip(self.ensemble, state_dict_paths):
        state_dict = torch.load(path, map_location=device)
        model.load_state_dict(state_dict)
        model.eval()
        for param in model.parameters():
            param.requires_grad = False

def forward(self, x: torch.Tensor):
    """
    Forward pass through the ensemble.

```

This method runs inference through each PSSP Encoder model in the ensemble independently and averages their output logits to produce a single prediction. All models are evaluated under `torch.no_grad()`

since their parameters are frozen.

Parameters

`x : torch.Tensor`

Input tensor of shape (batch_size, embed_dim)
with dtype torch.float32.

Returns

`torch.Tensor`

Averaged logits tensor of shape (batch_size, num_classes)
representing the mean prediction across all ensemble models.

"""

`assert isinstance(x, torch.Tensor), "x must be a torch.Tensor."`

`assert x.dtype == torch.float32, "x must have dtype torch.float32."`

`assert x.dim() == 2, "x must be a 2D tensor of shape (batch_size,
embed_dim)."`

`# Collect per-model logits`

`outputs = []`

`with torch.no_grad():`

`for model in self.ensemble:`

`outputs.append(model(x)[0]) # [B, num_classes]`

`# Average across the ensemble dimension`

`return torch.stack(outputs, dim=0).mean(dim=0) # [num_models, B, C] ->`

`[B, C]`

Appendix IV — External Rules Module

This section presents the module responsible for applying empirical/external rules to the base model predictions.

```
import re

##### Salamov-Solovyev Q3 post-processing rules #####

# Rule 1: remove implausibly short helices
def rule_1_Q3(predictions: str) -> str:
    # EHE → EEE (single H between strands becomes strand)
    result = re.sub(r"(?<=E)H(?=E)", "E", predictions)
    # Otherwise convert all 1-2 residue helices to coil
    result = re.sub(r"(?<!H)H{1,2}(?!H)", lambda m: "C" *
len(m.group()), result)
    return result

# Rule 2: remove isolated single-residue strands
def rule_2_Q3(predictions: str) -> str:
    return re.sub(r"(?<!E)E(?!E)", "C", predictions)

# Rule 3: convert HEEH (strand-of-2 flanked by helices) to HHHH
def rule_3_Q3(predictions: str) -> str:
    return re.sub(r"HEEH", "HHHH", predictions)

##### Combined application #####
def apply_all_q3_rules(s: str) -> str:
    """Apply all three Q3 rules sequentially: rule1 → rule2 → rule3."""
    return rule_3_Q3(rule_2_Q3(rule_1_Q3(s)))
```

Appendix V — Hyperparameter Tuning Script

This section presents the module responsible for the hyperparameter tuning of the model presented in this work, using the Optuna framework. It applies to both the 8-state and 3-state configurations.

```
import torch
import torch.nn as nn
import numpy as np
import optuna
import os, gc
from pathlib import Path
from train_utils import *
from preprocessing import reduce_q8_to_q3
from pssp_encoder import PSSPEncoder
from config import Config

##### Module-level configuration #####
cfg = Config()
device = cfg['DEVICE']
embed_dir = cfg['PATHS']['embed_dir']
embed_dim = cfg['ESM2']['embed_dim']
tuning = cfg['TUNING']
batch_size = cfg['TRAINING_CONFIG']['batch_size']

n_trials = cfg['TUNING_TRIALS']
tuning_dir = Path("./optuna_results")
tuning_dir.mkdir(exist_ok=True)

##### Objective factory #####
def make_objective(run_config_name: str, run_config):
    tcfg = tuning[run_config_name]
    initial_state = run_config_name.split('_')[1] # "3class" or "8class"
    state = initial_state
    if cfg['USE_8CLASS'] == run_config_name:
        state = "8class"
    epochs = run_config['epochs']
    tokens = cfg['MODEL_CONFIG'][f"{initial_state}_params"]['tokens']
    n_classes = cfg['MODEL_CONFIG'][f"{initial_state}_params"]['num_classes']
    token_dim = embed_dim // tokens
    use_8class = cfg['USE_8CLASS'] == run_config_name
```

```

# Single-fold loader (shared cache across trials)
def _load(fold_idx):
    name = f"{tcfg['fold_prefix']}{fold_idx}_{state}.npz"
    print(name)
    X, Y, _ = load_npz_dataset(os.path.join(embed_dir, name))
    if use_8class:
        Y = reduce_q8_to_q3(Y)
    return X, Y

# Pre-load: training folds concatenated + validation fold
print(f"\nLoading data for {run_config_name}...")
X_train = np.concatenate([_load(i)[0] for i in tcfg['train_folds']], axis=0)
Y_train = np.concatenate([_load(i)[1] for i in tcfg['train_folds']], axis=0)
X_val, Y_val = _load(tcfg['val_fold'])
print(f" Train: {len(X_train):,} residues | Val: {len(X_val):,} residues")

# Per-trial objective
def objective(trial):
    set_seed(cfg['DEFAULT_SEED'])

    # Search space
    d_ff_multiplier = trial.suggest_int('d_ff_multiplier', 4, 8)
    num_heads = trial.suggest_categorical(
        'num_heads',
        [h for h in [4, 8, 16] if token_dim % h == 0]
    )
    layers = trial.suggest_int('layers', 6, 8)
    mlp_multiplier = trial.suggest_int('mlp_multiplier', 4, 10)
    dropout = trial.suggest_float('dropout', 0.1, 0.3)
    lr = trial.suggest_float('lr', 5e-5, 5e-4, log=True)
    weight_decay = trial.suggest_float('weight_decay', 1e-5, 5e-3, log=True)

    # Instantiate model with the proposed configuration
    model = PSSPEncoder(
        embed_dim = embed_dim,
        tokens = tokens,
        d_ff_multiplier = d_ff_multiplier,
        num_heads = num_heads,
        layers = layers,
        mlp_multiplier = mlp_multiplier,
        num_classes = n_classes,
        dropout = dropout
    ).to(device)

    trial_params = {'learning_rate': lr, 'weight_decay': weight_decay}
    optimizer, scheduler, loss_fn = build_optimizer_scheduler(model, trial_params,

```

epochs)

```
# Train with pruner-driven early stopping
try:
    best_val_acc = train(
        model=model, optimizer=optimizer,
        scheduler=scheduler, loss_fn=loss_fn,
        train_data=(X_train, Y_train),
        val_data=(X_val, Y_val),
        epochs=epochs,
        patience=epochs,    # pruner handles early stopping, not patience
        trial=trial
    )
except torch.cuda.OutOfMemoryError:
    print(f"[{run_config_name}] Trial {trial.number} OOM -- pruning")
    raise optuna.TrialPruned()
finally:
    del model, optimizer, scheduler
    torch.cuda.empty_cache()
    gc.collect()

return best_val_acc

return objective

##### Study orchestration (TPE + Median pruner, persisted to SQLite) #####
def run_study(run_config_name: str, run_config):
    tcfg          = tuning[run_config_name]
    state         = run_config_name.split('_')[1]
    epochs        = cfg['TRAINING_CONFIG'][f"{state}_epochs"]
    warmup_steps  = max(5, epochs // 3)

    print(f"\n{'='*70}")
    print(f"Hyperparameter Tuning -- {run_config_name}  ({epochs} epochs, {n_trials}
    trials)")
    print(f"{'='*70}\n")

    # Create / resume an Optuna study backed by SQLite
    study = optuna.create_study(
        storage          = f"sqlite:/// {tuning_dir / tcfg['db_name']}",
        study_name       = tcfg['study_name'],
        load_if_exists   = True,
        direction        = "maximize",
        pruner           = optuna.pruners.MedianPruner(
            n_startup_trials = 5,
            n_warmup_steps   = warmup_steps,
```

```

        interval_steps    = 2
    ),
    sampler = optuna.samplers.TPESampler(
        n_startup_trials    = 5,
        multivariate        = True,
        warn_independent_sampling = True
    )
)

# Run trials
study.optimize(
    make_objective(run_config_name, run_config),
    n_trials        = n_trials,
    show_progress_bar = True,
    gc_after_trial  = True
)

# Best-trial summary
print(f"\n[{run_config_name}] Best trial : {study.best_trial.number}")
print(f"[{run_config_name}] Best val acc: {study.best_value:.4f}")
print(f"[{run_config_name}] Best params :")
for k, v in study.best_params.items():
    print(f"    {k}: {v}")

return study.best_params

##### Entry point #####
if __name__ == "__main__":
    results = {}
    for run_config_name, run_config in tuning.items():
        results[run_config_name] = run_study(run_config_name, run_config)

    print(f"\n{'='*70}")
    print("All tuning complete!")
    for name, params in results.items():
        print(f"\n{name}: {params}")

```

Appendix VI — Train Utilities Module

This section presents the Python module containing training utilities needed for training and evaluating the model.

```
import torch
import torch.nn as nn
import os, json, random
import optuna
import numpy as np
from config import Config
from torch.optim.lr_scheduler import CosineAnnealingLR
from preprocessing import reduce_q8_to_q3
from tqdm import tqdm

##### Module-level configuration #####
cfg = Config()
device = cfg['DEVICE']
batch_size = cfg['TRAINING_CONFIG']['batch_size']

os.makedirs(cfg['PATHS']['cv_models'], exist_ok=True)
os.makedirs(cfg['PATHS']['training_curves'], exist_ok=True)

##### Reproducibility #####
def set_seed(seed=None):
    if seed is None:
        raise ValueError("A seed must be explicitly provided for reproducibility.")
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

##### Fold concatenation helper #####
def concat_folds(loaded_folds: dict, fold_names: list):
    """Concatenate X, Y arrays from a subset of loaded folds."""
    X = np.concatenate([loaded_folds[n][0] for n in fold_names], axis=0)
    Y = np.concatenate([loaded_folds[n][1] for n in fold_names], axis=0)
    return X, Y
```

```

##### Optimiser / scheduler / loss factory #####
def build_optimizer_scheduler(model, params: dict, epochs: int):
    """Build AdamW + CosineAnnealingLR + CrossEntropyLoss from a params
    dict."""
    # AdamW with  $\beta_1=0.9$ ,  $\beta_2=0.95$ 
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr = params['learning_rate'],
        weight_decay = params['weight_decay'],
        betas = (cfg['TRAINING_CONFIG']['beta_1'],
        cfg['TRAINING_CONFIG']['beta_2'])
    )
    # Cosine annealing without restarts
    scheduler = CosineAnnealingLR(optimizer, T_max=epochs,
    eta_min=cfg['TRAINING_CONFIG']['eta_min'])
    loss_fn = nn.CrossEntropyLoss()
    return optimizer, scheduler, loss_fn

##### NPZ dataset loading #####
def load_npz_dataset(npz_path: str):
    """Load a saved npz dataset into memory."""
    data = np.load(npz_path)
    embeddings = data['embeddings'] # [N, 1280] float32
    labels = data['labels'] # [N] int8
    chain_ids = data['chain_ids'] # [N] str
    data.close()
    return embeddings, labels, chain_ids

def load_folds(fold_names: list, embed_dir: str, use_8class: bool =
False):
    """
    Load a list of .npz fold files, optionally remapping Q8→Q3.
    Returns a dict {name: (X, Y, ids)} and skips missing files with a
    warning.
    """
    loaded = {}
    for name in fold_names:
        try:
            X, Y, ids = load_npz_dataset(os.path.join(embed_dir, name))
            if use_8class:
                Y = reduce_q8_to_q3(Y)
            loaded[name] = (X, Y, ids)
            print(f" {name}: {len(X):,} residues")
        except FileNotFoundError:
            print(f" WARNING: {name} not found, skipping.")
    return loaded

```

```

##### Batch iteration over residue-level NPZ arrays #####
def fetch_batch_from_npz(offset, embeddings, labels, total_samples):
    # Exhausted
    if offset >= total_samples:
        return None, None

    # Slice next batch and transfer to GPU
    end_offset = min(offset + batch_size, total_samples)
    batch_embeddings =
torch.from_numpy(embeddings[offset:end_offset]).float().to(device)
    batch_labels =
torch.from_numpy(labels[offset:end_offset].astype(np.int64)).to(device)
    return batch_embeddings, batch_labels

def batch_iterator(embeddings, labels):
    total = len(embeddings)
    offset = 0
    while offset < total:
        batch_embeddings, batch_labels = fetch_batch_from_npz(offset,
embeddings, labels, total)
        if batch_embeddings is None or batch_embeddings.size(0) == 0:
            break
        yield batch_embeddings, batch_labels
        offset += batch_embeddings.size(0)

##### Single train / validation phase #####
def run_phase(model, dataloader_fn, loss_fn, optimizer=None, desc=""):
    is_train = optimizer is not None
    model.train(is_train)

    total_loss, correct, total = 0.0, 0, 0
    ctx = torch.enable_grad() if is_train else torch.no_grad()

    with ctx:
        with tqdm(dataloader_fn(), desc=desc, unit="batch",
dynamic_ncols=True) as pbar:
            for inputs, targets in pbar:
                # Forward (PSSPEncoder returns (logits, attn_weights))
                logits = model(inputs)[0]
                loss = loss_fn(logits, targets)

                # Backward + clipped step
                if is_train:
                    optimizer.zero_grad()
                    loss.backward()

```

```

        torch.nn.utils.clip_grad_norm_(model.parameters()),
1.0)
        optimizer.step()

        # Running metrics
        batch_loss = loss.item()
        total_loss += batch_loss * logits.size(0)
        correct    += (logits.argmax(-1) ==
targets).sum().item()
        total      += logits.size(0)

        pbar.set_postfix(loss=f"{batch_loss:.4f}",
acc=f"{correct/total:.4f}")
        del logits, loss

    return total_loss / total, correct / total

##### Training loop with early stopping + best-loss checkpointing
#####
def train(model, optimizer, scheduler, loss_fn,
        train_data, val_data,
        epochs, patience, model_dir=None,
        save_name=None, trial=None):

    X_train, Y_train = train_data
    X_val,   Y_val    = val_data

    best_val_acc, min_val_loss    = 0.0, float('inf')
    patience_counter, best_epoch  = 0, 0
    train_losses, val_losses      = [], []
    train_accuracies, val_accuracies = [], []

    for e in range(epochs):
        # Training epoch
        def train_loader():
            return batch_iterator(X_train, Y_train)
        train_loss, train_acc = run_phase(
            model, train_loader, loss_fn, optimizer,
            desc=f"Epoch {e+1}/{epochs} Train"
        )
        scheduler.step()

        # Validation epoch
        def val_loader():
            return batch_iterator(X_val, Y_val)
        val_loss, val_acc = run_phase(

```

```

        model, val_loader, loss_fn,
        desc=f"Epoch {e+1}/{epochs} Validation"
    )

    # Optuna pruning hook
    if trial is not None:
        trial.report(val_acc, e)
        if trial.should_prune():
            raise optuna.TrialPruned()

    if val_acc > best_val_acc:
        best_val_acc = val_acc

    print(f"Epoch {e+1}/{epochs} | "
          f"Train {train_acc:.4f} | Val {val_acc:.4f} | "
          f"Loss {val_loss:.4f} | Best {best_val_acc:.4f}")

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    train_accuracies.append(train_acc)
    val_accuracies.append(val_acc)

    # Checkpoint on best validation loss + early-stopping
    if val_loss < min_val_loss:
        min_val_loss = val_loss
        best_epoch = e + 1
        patience_counter = 0
        if trial is None and save_name is not None:
            torch.save(model.state_dict(), os.path.join(model_dir,
f"{save_name}.pth"))
            print(f" ✓ Best model saved (loss {min_val_loss:.4f}) → "
{save_name}.pth")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f"Early stop at epoch {e+1} -- best epoch "
{best_epoch})
                break

    save_training_curves(save_name, train_losses, val_losses,
                        train_accuracies, val_accuracies,
                        out_dir=cfg['PATHS']['training_curves'])

    return best_val_acc

##### Training curve persistence (JSON) #####

```

```

def save_training_curves(save_name, train_losses, val_losses,
                        train_accs, val_accs,
out_dir="training_curves"):
    """Save loss/accuracy history to
    {out_dir}/{save_name}_curves.json."""
    if save_name is None:
        return

    os.makedirs(out_dir, exist_ok=True)
    data = {
        "train_loss":    train_losses,
        "val_loss":      val_losses,
        "train_accuracy": train_accs,
        "val_accuracy":  val_accs,
    }
    out_path = os.path.join(out_dir, f"{save_name}_curves.json")
    with open(out_path, "w") as f:
        json.dump(data, f, indent=2)
    print(f"Training curves saved → {out_path}")

```

Appendix VII — Extraction of Predictions Module

This section presents the module that extracts the model predictions and saves them into two files of FASTA format. It is also responsible for creating the windowed dataset needed for the post-processing algorithm to train.

```
import torch
from tqdm import tqdm
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import os
from train_utils import load_npz_dataset, batch_iterator
from preprocessing import parse_int_to_q3, parse_int_to_q8,
reduce_q8_to_q3
from config import Config

##### Module-level configuration #####
cfg      = Config()
device   = cfg['DEVICE']
batch_size = cfg['TRAINING_CONFIG']['batch_size']
embed_dir = cfg['PATHS']['embed_dir']

os.makedirs(cfg['PATHS']['cv_fasta'], exist_ok=True)

##### Ensemble logit extraction (softmax probabilities, residue-
flat) #####
def extract_logits(model: nn.Module, filenames: list[str],
run_config_name: str):
    """
    Load the given .npz files, run inference, return flat arrays:
        logits    : [N, Q]    softmax probabilities
        labels    : [N]       int8 true labels
        chain_ids : [N]       str residue-level chain IDs
    """
    embeddings_list, labels_list, chain_ids_list = [], [], []

    # Concatenate all requested folds into a single flat record
    for name in filenames:
        try:
            X, Y, ids = load_npz_dataset(os.path.join(embed_dir, name))
            if cfg['USE_8CLASS'] == run_config_name:
```

```

        Y = reduce_q8_to_q3(Y)
        embeddings_list.append(X)
        labels_list.append(Y)
        chain_ids_list.append(ids)
        print(f"  Loaded {name}: {len(X)} residues")
    except FileNotFoundError:
        print(f"  WARNING: {name} not found, skipping.")

embeddings = np.concatenate(embeddings_list, axis=0)  # [N, 1280]
labels      = np.concatenate(labels_list,            axis=0)  # [N]
chain_ids   = np.concatenate(chain_ids_list,         axis=0)  # [N]

# Batched ensemble inference (averaged logits → softmax)
model.eval()
all_logits = []
with torch.no_grad():
    for batch_emb, _ in tqdm(batch_iterator(embeddings, labels),
desc="Inference", unit="batch"):
        probs = F.softmax(model(batch_emb), dim=-1)  # [B, Q]
        all_logits.append(probs.cpu().numpy())

logits = np.concatenate(all_logits, axis=0)  # [N, Q]
return logits, labels, chain_ids

##### Sliding-window feature construction (chain-aware, zero-
padded) #####
def build_windowed_dataset(logits: np.ndarray, labels: np.ndarray,
                           chain_ids: np.ndarray, window_size: int):
    assert window_size % 2 == 1, "window_size must be odd"
    half = window_size // 2
    N, Q = logits.shape

    # Locate chain boundaries (so windows never cross proteins)
    change_mask = np.concatenate([[True], chain_ids[1:] != chain_ids[:-
1]])
    chain_starts = np.where(change_mask)[0]
    chain_ends   = np.concatenate((chain_starts[1:], [N]))

    X_list, y_list = [], []

    # Per-chain windowing
    for start, end in zip(chain_starts, chain_ends):
        chain_logits = logits[start:end]  # [L, Q]
        chain_labels = labels[start:end]  # [L]
        L            = end - start

```

```

    # Zero-pad on both sides: [L + 2*half, Q]
    padded = np.zeros((L + 2 * half, Q), dtype=np.float32)
    padded[half: half + L] = chain_logits

    # Vectorised sliding window → [L, window_size, Q]
    shape = (L, window_size, Q)
    strides = (padded.strides[0], padded.strides[0],
padded.strides[1])
    windows = np.lib.stride_tricks.as_strided(padded, shape=shape,
strides=strides)

    # Flatten last two dims → [L, window_size * Q]
    X_list.append(windows.reshape(L, window_size * Q).copy()) #
.copy() to own memory
    y_list.append(chain_labels.astype(np.int64))

    return np.concatenate(X_list, axis=0), np.concatenate(y_list,
axis=0)

##### Base-model / ensemble predictions → per-chain FASTA #####
def get_predictions_to_fasta(
    model: nn.Module,
    npz_path: str,
    out_pred_fasta: str,
    out_true_fasta: str,
    Q: int,
    run_config_name: str
):

    # Pick label decoder for Q3 / Q8
    if Q == 3:
        int_to_ss = parse_int_to_q3
    elif Q == 8:
        int_to_ss = parse_int_to_q8
    else:
        raise ValueError(f"Invalid Q: {Q}")

    # Load flat residue-level arrays
    embeddings, labels, chain_ids =
load_npz_dataset(os.path.join(embed_dir, npz_path))

    model.eval()

    # Batch inference, argmax decoding
    all_preds = []
    with torch.no_grad():

```

```

        for batch_emb, _ in tqdm(batch_iterator(embeddings, labels),
desc="Inference", unit="batch"):
            logits = model(batch_emb)
            preds = torch.argmax(logits, dim=1).cpu().tolist()
            all_preds.extend(preds)

all_preds = np.array(all_preds, dtype=np.int64) # [N]

# Locate chain boundaries to reconstruct per-chain FASTA
N = len(chain_ids)
change_mask = np.concatenate(([True], chain_ids[1:] != chain_ids[:-
1]))
chain_starts = np.where(change_mask)[0]
chain_ends = np.concatenate((chain_starts[1:], [N]))

# Write predictions + ground truth FASTA
with open(out_pred_fasta, "w") as pred_f, open(out_true_fasta, "w")
as true_f:
    for start, end in tqdm(zip(chain_starts, chain_ends),
total=len(chain_starts), desc="Writing FASTA", unit="chain"):
        chain_id = chain_ids[start]
        chain_preds = all_preds[start:end]
        chain_labels = labels[start:end]

        pred_ss = "".join(int_to_ss(p) for p in chain_preds)
        if cfg['USE_8CLASS'] == run_config_name:
            true_ss = "".join(int_to_ss(reduce_q8_to_q3(int(l)))) for
l in chain_labels)
        else:
            true_ss = "".join(int_to_ss(int(l)) for l in
chain_labels)

        pred_f.write(f">{chain_id}\n{pred_ss}\n")
        true_f.write(f">{chain_id}\n{true_ss}\n")

    print(f"Predictions written → {out_pred_fasta}")
    print(f"True labels written → {out_true_fasta}")

##### Random Forest predictions → per-chain FASTA #####
def get_rf_predictions_to_fasta(
    rf, # trained RandomForestClassifier
    logits: np.ndarray, # [N, Q] softmax probs from ensemble
    labels: np.ndarray, # [N] true labels
    chain_ids: np.ndarray, # [N] residue-level chain IDs
    window_size: int,
    out_pred_fasta: str,

```

```

        out_true_fasta: str,
        Q: int
    ):
        # Pick label decoder
        if Q == 3:
            int_to_ss = parse_int_to_q3
        elif Q == 8:
            int_to_ss = parse_int_to_q8
        else:
            raise ValueError(f"Invalid Q: {Q}")

    N = len(chain_ids)
    half = window_size // 2

    # Chain boundary detection
    change_mask = np.concatenate(([True], chain_ids[1:] != chain_ids[:-
1]))
    chain_starts = np.where(change_mask)[0]
    chain_ends = np.concatenate((chain_starts[1:], [N]))

    os.makedirs(os.path.dirname(out_pred_fasta) or ".", exist_ok=True)
    os.makedirs(os.path.dirname(out_true_fasta) or ".", exist_ok=True)

    # Per-chain windowing + RF prediction + FASTA write
    with open(out_pred_fasta, "w") as pred_f, open(out_true_fasta, "w")
as true_f:
        for start, end in tqdm(
            zip(chain_starts, chain_ends),
            total=len(chain_starts),
            desc="Writing RF FASTA", unit="chain"
        ):
            chain_logits = logits[start:end]      # [L, Q]
            chain_labels = labels[start:end]      # [L]
            L = end - start
            chain_id = chain_ids[start]

            # Build windowed features for this chain
            padded = np.zeros((L + 2 * half, logits.shape[1]),
dtype=np.float32)
            padded[half: half + L] = chain_logits

            shape = (L, window_size, logits.shape[1])
            strides = (padded.strides[0], padded.strides[0],
padded.strides[1])
            X_chain = np.lib.stride_tricks.as_strided(
                padded, shape=shape, strides=strides

```

```

).reshape(L, -1).copy()          # [L, window_size * Q]

preds = rf.predict(X_chain)      # [L]

pred_ss = "".join(int_to_ss(p)   for p in preds)
true_ss = "".join(int_to_ss(int(l)) for l in chain_labels)

pred_f.write(f">{chain_id}\n{pred_ss}\n")
true_f.write(f">{chain_id}\n{true_ss}\n")

print(f"RF Predictions written → {out_pred_fasta}")
print(f"RF True labels written → {out_true_fasta}")

```

Appendix VIII — Cross-Validation Pipeline

This section presents the Python module responsible for the cross-validation setup with and without post-processing algorithm usage, saving in each case the predictions to FASTA format files.

```
import torch
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from train_utils import *
from preprocessing import reduce_q8_to_q3
from pssp_encoder import build_base_model, EnsemblePSSP
from save_predictions import *
import os, gc, joblib
from config import Config

##### Module-level configuration #####
cfg = Config()

device      = cfg['DEVICE']
embed_dim   = cfg['ESM2']['embed_dim']
seeds       = cfg['SEEDS']
cv_runs     = cfg['CV_RUNS']
rf_config   = cfg['RF_CONFIG']
window_size = cfg['WINDOW_SIZE']

cv_rf_dir = cfg['PATHS']['cv_rf_dir']
embed_dir = cfg['PATHS']['embed_dir']
os.makedirs(cfg['PATHS']['cv_rf_dir'], exist_ok=True)

##### Single outer fold: nested 80/20 train/val on the held-in
folds #####
def train_single_fold(fold_idx, fold_filenames, state, loaded_folds,
run_config, seed):
    params = cfg['MODEL_CONFIG'][f"{state}_params"].copy()
    params['embed_dim'] = embed_dim
    epochs = cfg['TRAINING_CONFIG'][f"{state}_epochs"]
    set_seed(seed)

    # Held-out (outer) test fold
    test_fold_name = fold_filenames[fold_idx]
```

```

    # Build the inner train+val pool from the remaining folds
    inner_fold_names = [name for i, name in enumerate(fold_filenames) if
i != fold_idx]
    X_inner, Y_inner = concat_folds(loader_folds, inner_fold_names)
    n_inner = len(X_inner)

    # 80/20 random split inside the inner pool
    idx = np.arange(n_inner)
    np.random.shuffle(idx)
    split = int(cfg['TRAINING_CONFIG']['split'] * n_inner)
    train_idx = idx[:split]
    val_idx = idx[split:]

    X_train, Y_train = X_inner[train_idx], Y_inner[train_idx]
    X_val, Y_val = X_inner[val_idx], Y_inner[val_idx]

    # Build model / optimiser / scheduler
    model = build_base_model(params)
    optimizer, scheduler, loss_fn = build_optimizer_scheduler(model,
params, epochs)

    print(f"\n{'='*70}")
    print(f"Fold {fold_idx+1}/{run_config['n_folds']} | Test (held-out):
{test_fold_name}")
    print(f"{'='*70}")

    # Train on the inner train/val split
    train(
        model=model, optimizer=optimizer,
        scheduler=scheduler, loss_fn=loss_fn,
        train_data=(X_train, Y_train),
        val_data=(X_val, Y_val),
        epochs=epochs,
        model_dir=cfg['PATHS']['cv_models'],
        patience=run_config['patience'],

save_name=f"{'_'.join(test_fold_name.split('_')[:2])}_{state}_{seed}"
    )

    # Evaluate on the held-out outer fold
    X_test, Y_test, _ = loader_folds[test_fold_name]

    def test_loader():
        return batch_iterator(X_test, Y_test)

    test_loss, test_acc = run_phase(

```

```

        model, test_loader, loss_fn,
        optimizer=None,
        desc=f"Fold {fold_idx+1} Test"
    )

    print(f"Fold {fold_idx+1} | Test Acc: {test_acc:.4f} | Test Loss:
{test_loss:.4f}")

    del model, optimizer, scheduler, X_inner, Y_inner
    gc.collect()
    torch.cuda.empty_cache()
    return test_acc

##### Full k-fold x multi-seed cross-validation driver #####
def run_cross_validation(run_config_name, run_config):
    # Take initial state of configuration
    initial_state = run_config_name.split('_')[1]
    state = initial_state

    # If configuration is NetsurfP_3class use the 8class files
    if cfg['USE_8CLASS'] == run_config_name:
        state = "8class"

    fold_filenames = [
        f"{run_config['fold_prefix']}{i}_{state}.npz"
        for i in range(run_config['n_folds'])
    ]

    # Load all folds once into memory
    loaded_folds = {}
    for name in fold_filenames:
        try:
            X, Y, ids = load_npz_dataset(os.path.join(embed_dir, name))
            if cfg['USE_8CLASS'] == run_config_name:
                Y = reduce_q8_to_q3(Y)
            loaded_folds[name] = (X, Y, ids)
            print(f"Fold {name} contains {len(X)} residues")
        except FileNotFoundError:
            print(f"Fold {name} was not found. Skipping...")

    print(f"Folds loaded: {len(loaded_folds.keys())}")
    if len(loaded_folds.keys()) <= 1:
        raise Exception("Cannot continue. Need at least 2 loaded
folds.")

    # Repeat across seeds (default: 42, 123, 1337)

```


3: Do k-fold Cross-Validation for each configuration using Random Forest

```
    post-processing and save the fasta predictions.
    """
    codes = [1,2,3]

    for code in codes:
        # Code 1: pure base-model k-fold CV
        if code == 1:
            for run_config_name, run_config in cv_runs.items():
                run_cross_validation(run_config_name, run_config)

        # Code 2: seed-ensemble inference → FASTA
        elif code == 2:
            for run_config_name, run_config in cv_runs.items():
                print(f"\nGenerating FASTA: {run_config_name} |
{run_config['n_folds']}")

                initial_state = run_config_name.split('_')[1]
                state = initial_state
                if cfg['USE_8CLASS'] == run_config_name:
                    state = "8class"

                fold_filenames =
[f"{run_config['fold_prefix']}{i}_{state}.npz"          for i in
range(run_config['n_folds'])]
                fold_prefixes =
[f"{run_config['fold_prefix']}{i}_{initial_state}"      for i in
range(run_config['n_folds'])]

                for filename, prefix in zip(fold_filenames,
fold_prefixes):
                    # Collect per-seed checkpoints
                    state_dict_paths =
[os.path.join(cfg['PATHS']['cv_models'], f"{prefix}_{seed}.pth") for
seed in seeds]

                    # Skip if any seed model is missing
                    missing = [p for p in state_dict_paths if not
os.path.exists(p)]
                    if missing:
                        print(f" WARNING: Missing model files for
{filename}: {missing}")
                        continue

                    ensemble = EnsemblePSSP(state=initial_state,
```

```

state_dict_paths=state_dict_paths).to(device)

        out_pred = os.path.join(cfg['PATHS']['cv_fasta'],
f"{prefix}_pred.fasta")
        out_true = os.path.join(cfg['PATHS']['cv_fasta'],
f"{prefix}_true.fasta")

        get_predictions_to_fasta(ensemble, filename,
out_pred, out_true, int(initial_state[0]), run_config_name)

    # Code 3: train RF post-processor per fold + FASTA
    elif code == 3:
        for run_config_name, run_config in cv_runs.items():
            initial_state = run_config_name.split('_')[1]
            state = initial_state
            if cfg['USE_8CLASS'] == run_config_name:
                state = "8class"

            n_folds = run_config['n_folds']
            fold_filenames =
[f"{run_config['fold_prefix']}{i}_{state}.npz" for i in
range(n_folds)]
            fold_prefixes =
[f"{run_config['fold_prefix']}{i}_{initial_state}" for i in
range(n_folds)]

            for fold_idx in range(n_folds):
                prefix = fold_prefixes[fold_idx]
                test_filename = fold_filenames[fold_idx]
                train_filenames = [fold_filenames[j] for j in
range(n_folds) if j != fold_idx]

                state_dict_paths = [
                    os.path.join(cfg['PATHS']['cv_models'],
f"{prefix}_{seed}.pth")
                    for seed in seeds
                ]

                ensemble = EnsemblePSSP(state=initial_state,
state_dict_paths=state_dict_paths).to(device)

                # Extract ensemble probabilities on train + test
                train_logits, train_labels, train_cids =
extract_logits(ensemble, train_filenames, run_config_name)
                test_logits, test_labels, test_cids =
extract_logits(ensemble, [test_filename], run_config_name)

```

```

        # Build chain-safe windowed feature matrices
        X_train, y_train =
build_windowed_dataset(train_logits, train_labels, train_cids,
window_size)

        X_test, y_test =
build_windowed_dataset(test_logits, test_labels, test_cids,
window_size)

        print(f"Fold {fold_idx}: train={X_train.shape[0]:,}
| test={X_test.shape[0]:,}")

        # Train Random Forest post-processor
        rf = RandomForestClassifier(**rf_config, n_jobs=-1,
random_state=cfg['DEFAULT_SEED'])
        rf.fit(X_train, y_train)

        acc = (rf.predict(X_test) == y_test).mean()
        print(f" Fold {fold_idx}: RF Test Acc = {acc:.4f}")

        # RF + per-chain FASTA outputs
        rf_out = os.path.join(cv_rf_dir,
f"{prefix}_rf_w{window_size}.pkl")
        joblib.dump(rf, rf_out)
        print(f" Fold {fold_idx}: RF saved → {rf_out}")

        out_pred = os.path.join(cfg['PATHS']['cv_fasta'],
f"{prefix}_rf_pred.fasta")
        out_true = os.path.join(cfg['PATHS']['cv_fasta'],
f"{prefix}_rf_true.fasta")
        get_rf_predictions_to_fasta(rf, test_logits,
test_labels, test_cids, window_size, out_pred, out_true,
int(initial_state[0]))

        gc.collect()
        torch.cuda.empty_cache()

if __name__ == "__main__":
    main()

```

Appendix IX — Training and Evaluation Pipeline

This module implements the final training and evaluation pipeline for the PSSP system responsible for all configurations. The evaluation predictions with or without usage of post-processing strategies are saved to FASTA format files.

```
import torch
import torch.nn as nn
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from train_utils import *
from preprocessing import reduce_q8_to_q3
from pssp_encoder import EnsemblePSSP, build_base_model
from save_predictions import *
import os, gc, joblib
from config import Config

##### Module-level configuration #####
cfg = Config()
device      = cfg['DEVICE']
embed_dir   = cfg['PATHS']['embed_dir']
embed_dim   = cfg['ESM2']['embed_dim']
seeds       = cfg['SEEDS']
final_runs  = cfg['FINAL_TRAINING']
rf_config   = cfg['RF_CONFIG']
window_size = cfg['WINDOW_SIZE']

for d in [cfg['PATHS']['evaluation_models'],
          cfg['PATHS']['evaluation_fasta'],
          cfg['PATHS']['evaluation_rf_dir']]:
    os.makedirs(d, exist_ok=True)

##### Full-data training (all folds combined, 80/20 split for early
stopping) #####
def full_train(run_config_name, run_config):
    initial_state = run_config_name.split('_')[1]
    state         = initial_state
    if cfg['USE_8CLASS'] == run_config_name:
        state = "8class"

    fold_filenames = [
        f"{run_config['fold_prefix']}{i}_{state}.npz"
```

```

        for i in range(run_config['n_folds'])
    ]

    print(f"\n{'='*70}")
    print(f"Final Training: {run_config_name} | {run_config['n_folds']}
folds")
    print(f"{'='*70}")

    # Load all folds and concatenate into a single training corpus
    loaded_folds = {}
    for name in fold_filenames:
        try:
            X, Y, ids = load_npz_dataset(os.path.join(embed_dir, name))
            if cfg['USE_8CLASS'] == run_config_name:
                Y = reduce_q8_to_q3(Y)
                loaded_folds[name] = (X, Y, ids)
                print(f" {name}: {len(X):,} residues")
        except FileNotFoundError:
            print(f" WARNING: {name} not found, skipping.")

    X_all = np.concatenate([loaded_folds[n][0] for n in fold_filenames
if n in loaded_folds], axis=0)
    Y_all = np.concatenate([loaded_folds[n][1] for n in fold_filenames
if n in loaded_folds], axis=0)
    del loaded_folds

    params =
cfg['MODEL_CONFIG'][f"{initial_state}_params"].copy()
    params['embed_dim'] = embed_dim
    epochs =
cfg['TRAINING_CONFIG'][f"{initial_state}_epochs"]

    # Train one model per seed (skip if checkpoint already exists)
    for seed in seeds:
        save_name = f"{run_config_name}_{seed}"
        model_path = os.path.join(cfg['PATHS']['evaluation_models'],
f"{save_name}.pth")

        if os.path.exists(model_path):
            print(f" Seed {seed}: model exists, skipping →
{model_path}")
            continue

        print(f"\n=== Seed {seed} ===")
        set_seed(seed)

```

```

# Random 80/20 split (val used for early stopping)
n      = len(X_all)
idx    = np.arange(n)
np.random.shuffle(idx)
split  = int(cfg['TRAINING_CONFIG']['split'] * n)

X_train, Y_train = X_all[idx[:split]], Y_all[idx[:split]]
X_val,   Y_val   = X_all[idx[split:]], Y_all[idx[split:]]

model = build_base_model(params)
optimizer, scheduler, loss_fn = build_optimizer_scheduler(model,
params, epochs)

train(
    model=model, optimizer=optimizer,
    scheduler=scheduler, loss_fn=loss_fn,
    train_data=(X_train, Y_train),
    val_data=(X_val, Y_val),
    epochs=epochs,
    model_dir=cfg['PATHS']['evaluation_models'],
    patience=run_config['patience'],
    save_name=save_name
)

del model, optimizer, scheduler, X_train, Y_train, X_val, Y_val
gc.collect()
torch.cuda.empty_cache()

del X_all, Y_all
gc.collect()

##### Top-level driver: four pipeline codes #####
def main():
    """
    Codes:
    1: Train final models on all folds for each config / seed.
    2: Ensemble inference on evaluation sets → save FASTA.
    3: Extract logits from training folds → train RF → save RF.
    4: RF + ER post-processing on eval sets → save FASTA.
    """
    codes = [1,2,3,4]

    for code in codes:
        # Code 1: full-data training per seed
        if code == 1:
            for run_config_name, run_config in final_runs.items():

```

```

full_train(run_config_name, run_config)

# Code 2: seed-ensemble inference → FASTA (base predictions)
elif code == 2:
    for run_config_name, run_config in final_runs.items():
        initial_state = run_config_name.split('_')[1]
        state = initial_state
        if cfg['USE_8CLASS'] == run_config_name:
            state = "8class"

        state_dict_paths = [
            os.path.join(cfg['PATHS']['evaluation_models'],
f"{run_config_name}_{seed}.pth")
            for seed in seeds
        ]
        missing = [p for p in state_dict_paths if not
os.path.exists(p)]
        if missing:
            print(f" WARNING: missing models for
{run_config_name}: {missing}")
            continue

        ensemble = EnsemblePSSP(state=initial_state,
state_dict_paths=state_dict_paths).to(device)

        print(f"\n{'='*70}")
        print(f"Code 2 | {run_config_name} | eval FASTA")
        print(f"{'='*70}")

        for eval_set in run_config['evaluation']:
            eval_npz = f"{eval_set}_{state}.npz"
            out_pred =
os.path.join(cfg['PATHS']['evaluation_fasta'],
f"{run_config_name}_{eval_set}_pred.fasta")
            out_true =
os.path.join(cfg['PATHS']['evaluation_fasta'],
f"{run_config_name}_{eval_set}_true.fasta")

            get_predictions_to_fasta(ensemble, eval_npz,
out_pred, out_true, int(initial_state[0]), run_config_name)
            print(f" {eval_set}: saved → {out_pred}")

        del ensemble
        gc.collect()
        torch.cuda.empty_cache()

```

```

# Code 3: train RF post-processor on full training corpus
elif code == 3:
    for run_config_name, run_config in final_runs.items():
        initial_state = run_config_name.split('_')[1]
        state = initial_state
        if cfg['USE_8CLASS'] == run_config_name:
            state = "8class"

        rf_out = os.path.join(cfg['PATHS']['evaluation_rf_dir'],
f"{run_config_name}_rf_w{window_size}.pkl")
        if os.path.exists(rf_out):
            print(f" {run_config_name}: RF exists, skipping →
{rf_out}")
            continue

        state_dict_paths = [
            os.path.join(cfg['PATHS']['evaluation_models'],
f"{run_config_name}_{seed}.pth")
            for seed in seeds
        ]
        missing = [p for p in state_dict_paths if not
os.path.exists(p)]
        if missing:
            print(f" WARNING: missing models for
{run_config_name}: {missing}")
            continue

        ensemble = EnsemblePSSP(state=initial_state,
state_dict_paths=state_dict_paths).to(device)

        print(f"\n{'='*70}")
        print(f"Code 3 | {run_config_name} | training RF on full
train set")
        print(f"{'='*70}")

        train_filenames = [
            f"{run_config['fold_prefix']}{i}_{state}.npz"
            for i in range(run_config['n_folds'])
        ]

        train_logits, train_labels, train_cids = extract_logits(
            ensemble, train_filenames, run_config_name
        )

        del ensemble
        gc.collect()

```

```

torch.cuda.empty_cache()

# Build chain-safe windowed features
X_train, y_train = build_windowed_dataset(
    train_logits, train_labels, train_cids, window_size
)
print(f" {X_train.shape[0]:,} residues |
feat_dim={X_train.shape[1]}")

# Fit Random Forest
rf = RandomForestClassifier(**rf_config, n_jobs=-1,
random_state=cfg['DEFAULT_SEED'])
rf.fit(X_train, y_train)
joblib.dump(rf, rf_out)
print(f" RF saved → {rf_out}")

del rf, X_train, y_train, train_logits, train_labels,
train_cids
gc.collect()

# Code 4: RF + ER post-processing on eval sets → FASTA
elif code == 4:
    from preprocessing import parse_int_to_q3
    from external_rules import apply_all_q3_rules

    for run_config_name, run_config in final_runs.items():
        initial_state = run_config_name.split('_')[1]
        state = initial_state
        Q = int(initial_state[0])

        rf_path =
os.path.join(cfg['PATHS']['evaluation_rf_dir'],
f"{run_config_name}_rf_w{window_size}.pkl")
        if not os.path.exists(rf_path):
            print(f" {run_config_name}: RF not found at
{rf_path}, skipping.")
            continue

        state_dict_paths = [
            os.path.join(cfg['PATHS']['evaluation_models'],
f"{run_config_name}_{seed}.pth")
            for seed in seeds
        ]
        missing = [p for p in state_dict_paths if not
os.path.exists(p)]
        if missing:

```

```

        print(f" WARNING: missing models for
{run_config_name}: {missing}")
        continue

    ensemble = EnsemblePSSP(state=initial_state,
state_dict_paths=state_dict_paths).to(device)
    rf = joblib.load(rf_path)

    print(f"\n{'='*70}")
    print(f"Code 4 | {run_config_name} | Q{Q} | RF + ER eval
FASTA")
    print(f"{'='*70}")

    for eval_set in run_config['evaluation']:
        eval_npz = f"{eval_set}_{state}.npz"

        eval_logits, eval_labels, eval_cids =
extract_logits(
            ensemble, [eval_npz], run_config_name
        )

        # RF only
        out_pred =
os.path.join(cfg['PATHS']['evaluation_fasta'],
f"{run_config_name}_{eval_set}_rf_pred.fasta")
        out_true =
os.path.join(cfg['PATHS']['evaluation_fasta'],
f"{run_config_name}_{eval_set}_rf_true.fasta")
        if not (os.path.exists(out_pred) and
os.path.exists(out_true)):
            get_rf_predictions_to_fasta(
                rf, eval_logits, eval_labels, eval_cids,
                window_size, out_pred, out_true, Q
            )
            print(f" [{eval_set}] rf: saved → {out_pred}")

        # ER / ER→RF / RF→ER variants (Q3 only)
        if Q == 3:
            # Reconstruct per-chain prob arrays and pred
strings for ER variants
            N = len(eval_cids)
            change_mask = np.concatenate(([True],
eval_cids[1:] != eval_cids[:-1]))
            chain_starts = np.where(change_mask)[0]
            chain_ends = np.concatenate((chain_starts[1:],
[N]))

```

```

unique_cids = [eval_cids[s] for s in
chain_starts]

er_pred_seqs = []
er_rf_pred_seqs = []
rf_er_pred_seqs = []
true_seqs = []

half = window_size // 2

for start, end in zip(chain_starts, chain_ends):
    probs = eval_logits[start:end] #
[L, Q] already softmax
    labels = eval_labels[start:end] #
[L]

    L = end - start

    pred_str = "".join(parse_int_to_q3(int(p))
for p in probs.argmax(axis=-1))
    true_str = "".join(parse_int_to_q3(int(l))
for l in labels)

    true_seqs.append(true_str)

    # Ensemble + ER

er_pred_seqs.append(apply_all_q3_rules(pred_str))

    # Ensemble + ER → RF (hybrid one-hot probs)
    Q3_MAP = {"H": 0, "E": 1, "C": 2}
    er_str = apply_all_q3_rules(pred_str)
    er_probs = probs.copy()
    for i, (orig, ruled) in
enumerate(zip(pred_str, er_str)):
        if orig != ruled:
            er_probs[i] = 0.0
            er_probs[i, Q3_MAP[ruled]] = 1.0

    padded = np.zeros((L + 2 * half, Q),
dtype=np.float32)

    padded[half: half + L] = er_probs
    shape = (L, window_size, Q)
    strides = (padded.strides[0],
padded.strides[0], padded.strides[1])
    X_er =
np.lib.stride_tricks.as_strided(padded, shape=shape,
strides=strides).reshape(L, -1).copy()

```

```

er_rf_preds = rf.predict(X_er)

er_rf_pred_seqs.append("".join(parse_int_to_q3(int(p)) for p in
er_rf_preds))

# Ensemble + RF → ER
padded_rf = np.zeros((L + 2 * half, Q),
dtype=np.float32)
padded_rf[half: half + L] = probs
X_rf =
np.lib.stride_tricks.as_strided(padded_rf, shape=shape,
strides=strides).reshape(L, -1).copy()
rf_preds = rf.predict(X_rf)
rf_str = "".join(parse_int_to_q3(int(p))
for p in rf_preds)

rf_er_pred_seqs.append(apply_all_q3_rules(rf_str))

# FASTA for each ER variant
for method, pred_seqs in [
    ("er", er_pred_seqs),
    ("er_rf", er_rf_pred_seqs),
    ("rf_er", rf_er_pred_seqs),
]:
    out_pred =
os.path.join(cfg['PATHS']['evaluation_fasta'],
f"{run_config_name}_{eval_set}_{method}_pred.fasta")
    out_true =
os.path.join(cfg['PATHS']['evaluation_fasta'],
f"{run_config_name}_{eval_set}_{method}_true.fasta")

    with open(out_pred, "w") as fp,
open(out_true, "w") as ft:
        for cid, pred_seq, true_seq in
zip(unique_cids, pred_seqs, true_seqs):
            fp.write(f">{cid}\n{pred_seq}\n")
            ft.write(f">{cid}\n{true_seq}\n")
        print(f" [{eval_set}] {method}: saved →
{out_pred}")

del eval_logits, eval_labels, eval_cids
gc.collect()

del ensemble, rf
gc.collect()
torch.cuda.empty_cache()

```

```
gc.collect()
torch.cuda.empty_cache()

if __name__ == "__main__":
    main()
```

Appendix X — Configuration

This section gives the general Config Python class used through all the preceding modules, along with the config.json file.

```
import json
from pathlib import Path

##### Singleton configuration loader #####
class Config:
    _instance = None

    def __new__(cls, path=None):
        # Lazy-instantiate on first access
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            p = path or Path(__file__).parent / "config.json"
            with open(p) as f:
                data = json.load(f)
            cls._instance.__dict__.update(data)
            return cls._instance

    # Mapping-style access (cfg['KEY'])
    def __getitem__(self, key):
        return getattr(self, key)

    def __contains__(self, key):
        return hasattr(self, key)

{
    "DEVICE": "cuda",
    "SEEDS": [42, 123, 1337],
    "DEFAULT_SEED": 42,
    "ESM2": {
        "model_name": "facebook/esm2_t33_650M_UR50D",
        "embed_dim": 1280
    },
    "PATHS": {
        "data_dir": "./data",
        "embed_dir": "./data/embeddings_dir",

```

```

    "tuning_dir": "./optuna_results",
    "cv_models": "./weights/cross_validation",
    "evaluation_models": "./weights/evaluation",
    "training_curves": "./training_curves",
    "cv_fasta": "./fasta_results/cross_validation",
    "evaluation_fasta": "./fasta_results/evaluation",
    "cv_rf_dir": "./rf/cross_validation",
    "evaluation_rf_dir": "./rf/evaluation"
},
"MODEL_CONFIG": {
    "3class_params": {
        "tokens": 5,
        "d_ff_multiplier": 4,
        "num_heads": 16,
        "layers": 6,
        "mlp_multiplier": 7,
        "dropout": 0.1,
        "learning_rate": 0.0001,
        "weight_decay": 1.8e-05,
        "num_classes": 3
    },
    "8class_params": {
        "tokens": 10,
        "d_ff_multiplier": 5,
        "num_heads": 8,
        "layers": 7,
        "mlp_multiplier": 6,
        "dropout": 0.1,
        "learning_rate": 0.0002,
        "weight_decay": 0.0013,
        "num_classes": 8
    }
},
"TRAINING_CONFIG": {
    "split": 0.8,
    "3class_epochs": 50,
    "8class_epochs": 60,
    "beta_1": 0.9,
    "beta_2": 0.95,
    "eta_min": 1e-06,
    "batch_size": 256
},
"TUNING_TRIALS": 30,
"TUNING": {
    "NetsurfP_3class": {
        "train_folds": [0, 1, 2],

```

```

        "val_fold": 3,
        "epochs": 20,
        "fold_prefix": "train_fold",
        "study_name": "pssp_3class",
        "db_name": "study_3class.db"
    },
    "NetsurfP_8class": {
        "train_folds": [0, 1, 2],
        "val_fold": 3,
        "epochs": 30,
        "fold_prefix": "train_fold",
        "study_name": "pssp_8class",
        "db_name": "study_8class.db"
    }
},
"USE_8CLASS": "NetsurfP_3class",
"CV_RUNS": {
    "PISCES_3class": {
        "patience": 10,
        "n_folds": 5,
        "fold_prefix": "pisces_fold"
    },
    "NetsurfP_3class": {
        "patience": 10,
        "n_folds": 7,
        "fold_prefix": "train_fold"
    },
    "NetsurfP_8class": {
        "patience": 15,
        "n_folds": 7,
        "fold_prefix": "train_fold"
    }
},
"FINAL_TRAINING": {
    "PISCES_3class": {
        "patience": 5,
        "n_folds": 5,
        "fold_prefix": "pisces_fold",
        "evaluation": [
            "CASP13"
        ]
    },
    "NetsurfP_3class": {
        "patience": 5,
        "n_folds": 7,
        "fold_prefix": "train_fold",

```

```

        "evaluation": [
            "CB513",
            "CASP12",
            "TS115"
        ]
    },
    "NetsurfP_8class": {
        "patience": 10,
        "n_folds": 7,
        "fold_prefix": "train_fold",
        "evaluation": [
            "CB513",
            "CASP12",
            "TS115"
        ]
    }
},
"WINDOW_SIZE": 31,
"RF_CONFIG": {
    "n_estimators": 100,
    "max_depth": 25,
    "max_features": "sqrt",
    "min_samples_leaf": 2,
    "class_weight": "balanced"
}
}

```