FACULTY OF BIOSCIENCE ENGINEERING

FACULTY OF SCIENCES

PROTEIN SECONDARY STRUCTURE PREDICTION USING TRANSFORMER NETWORKS

word count: 24 360

Lotte Pollaris

Student ID: 0153456

Supervisor: Prof. Dr. Willem Waegeman

Tutor: Ir. Jim Clauwaert

A dissertation submitted to Ghent University in partial fulfilment of the requirements for the degree of master in master of Science in Bioinformatics: Bioscience Engineering.

Academic year: 2019 - 2020

GHENT UNIVERSITY

PREFACE

After finishing my Bachelor's degree in Bioscience Engineering, I knew I wasn't made for lab work, and lab work wasn't made for me. Still fascinated by life sciences, and with a passion for programming, I decided to dive into unknown waters, and choose for the Masterin Bioinformatic. It was an interesting path, sturdy at times, but in the end, I am happy with everything I've learned along the way.

In my first Master year, I took Protein Chemistry by prof. dr. Els Van Damme and Predictive Modelling by prof. dr. Willem Waegeman. Both courses made me wonder: would it be possible to combine two of my main interests, proteins and machine learning, in my master dissertation? Prof. dr. Willem Waegeman was as so flexible as to create an extra bioinformatics subject for me, for which I am very grateful, and ir. Jim Clauwaert immediately showed great enthusiasm.

Both Willem and Jim were of great help to me throughout the process. Jim, thank you for always being there when I needed you. Your knowledge helped me find my way through the deep learning labyrinth as did your extensive text correction. Willem, thank you for being so involved, touching upon new concepts and providing the necessary structure.

I spent my first semester this year abroad in Colombia, where dr. Sergio Pulido helped me start up my thesis, and shared his knowledge about python and machine learning in general. I want to thank Gaetan De Waele for helping me every time I found myself battling the GPU server, and Brent Van De Wiele and Niels De Mayer for proofreading. My parents have provided much-needed logistic and mental support, especially during these Corona-times. Thanks to my brother, for all the table-tennis-Covid-breaks. And to Milan, my boyfriend, for the genuine interest he's been showing in what I was doing, and for always believing in me.

CONTENTS

SAMENVATTING

Secundaire eiwitstructuurpredictie is een belangrijk deelprobleem op het gebied van eiwitstructuurpredictie in het algemeen. De secundaire eiwitstructuur helpt de 3D-eiwitstructuren en hun functies te voorspellen. De laatste jaren heeft deze onderzoekstak grote sprongen gemaakt door gebruik te maken van machinaal leren. Vooral modellen uit de natuurlijke taalverwerking, zoals convolutionele en recurrente neurale netwerken, werden hiervoor gebruikt. Recent hebben Transformer-netwerken ingrijpende wijzigingen veroorzaakt op het gebied van natuurlijke taalverwerking, omdat ze de oude modellen op heel wat vlakken overtreffen. Als gevolg daarvan worden recurrente neurale netwerken amper nog gebruikt.

In deze masterproef worden Transformer-modellen uit de natuurlijke taalverwerking aangepast zodat ze secundaire eiwitstructuren kunnen voorspellen. Verschillen tussen tekst en eiwitstructuren worden besproken en de bijhorende aanpassingen van de modellen worden uitgevoerd. Verschillende datasets om het model te trainen, valideren en testen worden gemaakt en grondig geanalyseerd. De prestatie van de modellen wordt diepgaand onderzocht, en vergeleken met de huidige state of the art.

Beschikbaarheid en implementatie

Alle code nodig om de datasets en de modellen te recreëren kan gevonden worden op:

`https://github.ugent.be/bw26master/2019_PollarisLotte.git`

ABSTRACT

Protein secondary structure prediction is a long-standing subproblem in the field of protein structure prediction. It plays an important role in improving predictions for protein 3D structures and functions. Over the last years, advances in this field were made using machine learning models based on CNNs and BRNNs with LSTM. Before, these types of models were mainly used in the field of Natural Language Processing (NLP). Recently, the Transformer model has brought drastic changes to the NLP field, significantly reducing RNN use . The advantages of Transformer networks over BRNNs with LSTM are numerous.

In this master thesis, Transformer models are adapted to predict protein secondary structure. Differences between text and protein sequences are discussed and necessary adjustments to the models are made. Different datasets to train and test the models are created and thoroughly analysed. The created models' performances are analysed in depth and compared to the current state-of-the-art in the field.

Availability and implementation

All code to recreate the data and the models is available at:

`https://github.ugent.be/bw26master/2019_PollarisLotte.git`

LIST OF ABBREVIATIONS

A	Alanine
aa	Amino-acid(s)
AI	Artificial intelligence
ANN	Artificial neural network
BRNN	Bidirectional recurrent neural network
C	Carbon atom or Cysteine
CNN	Convolutional neural network
Cryo-EM	Cry-electron microscopy
D	Aspartic acid
DNA	Deoxyribonucleic acid
DSSP	Define secondary structure of proteins
E	Glutamic acid
F	Phenylalanine
G	Glycine
GRU	Gated recurrent units
H	Hydrogen atom or Histidine
I	Isoleucine
K	Lysine
kDa	Kilo Dalton
KSDSSP	Kabsch and Sander algorithm for defining the secondary structure of proteins
L	Leucine
LSTM	Long short term memory
M	Methionine

N	Nitrogen atom or Asparagine
NGS	Next generation sequencing
NLP	Natural language processing
NMR	Nuclear magnetic resonance
mRNA	Messenger RNA
MSE	Mean squared error
P	Proline
pdb	Protein data bank
PSSM	Position specific scoring matrix
Q	Glutamine
R	Organic side chain or arginine
ReLU	Rectified linear unit
RNA	Ribonucleic acid
RNN	Recurrent neural network
S	Serine
seq	Sequence
SGD	Stochastic gradient descent
T	Threonine
tRNA	Transport RNA
V	Valine
W	Tryptophan
X	Rare amino-acid, unknown amino-acid
Z	Gap

CHAPTER 1

PROTEINS, THE BUILDING BLOCKS OF LIFE

1.1 Introduction

When looking at life, one sees different organisms standing, walking or floating around. At first glance, a plant does not look like a monkey. When looking at it on a smaller scale, it seems that they both have more in common than one might think. Both organisms are built from cells. Different kinds of cells for different kinds of functions, although the standard built up of all cells is the same.

Proteins are indispensable for cells to function. They are complex macromolecules, which can differ a lot in size and purpose. Hemoglobin, for example, allows us to respirate. Lipases and sucrases digest food. Lymphocytes are of great importance in immune systems.

Understanding how proteins work leads to a better understanding of life itself. Understanding how proteins of pathogens invade our body can facilitate the discovery of drugs inhibiting them. If we knew how a certain enzyme can degrade fats, we could possibly adapt it to clean spots off of clothes. This way we have been able to create proteins that make leather softer, work as detergents, can be used as drugs, produce high fructose corn syrup, bio converse cellulose... (Du et al., 2013).

A vast amount of today's challenges could be solved by understanding proteins better: exploring or adapting enzymes so they can digest oil (and thus clean the sea after oil leaks), or create bioplastic. In the pharmaceutical industry, understanding proteins is very important to create drugs or define drug targets. Illnesses like Alzheimer's, Parkinson, Huntington, and Cystic Fibrosis are believed to be caused by misfolded proteins, so a thorough understanding of protein folding can give more insights into these illnesses.

The best way of understanding proteins is to determine their 3D structure, as this is directly related to their function.

1.2 The structure of proteins

Thanks to next generation sequencing techniques (NGS), the sequencing of DNA became easier, which resulted in the release of a huge amount of protein sequences (Schuster, 2008). The amount of known protein structures doesn't follow this trend completely. At this moment, a thousandfold more protein sequences than protein structures are known, mainly because it is much cheaper and faster to sequence DNA than to determine protein structures.

Proteins are macromolecules built from amino-acids (aa). The general structure of aa is given in Figure 1.1. The backbone is the same for all aa, consisting out of an amino group, a central carbon (C) atom and a carboxyl group. Most proteins are built out of 20 different aa, which only differ in the organic side chain (R groups). These R groupshave different physiochemical properties. They can differ in polarity, charge, hydrophobicity et cetera. The kind and order of the aa are determined by the DNA sequence. Every triplet of basepairs codes for one certain aa. Their order in the protein is called the primary structure of the protein. In order to be able to tell something about the function of the protein, the structure of the protein needs to be known. However, efforts that elicit the protein function from its structure have been unsuccesful.

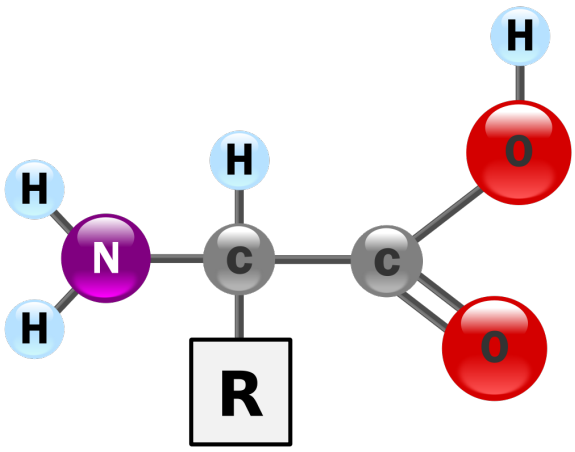


Figure 1.1: The general structure of an aa. The central C atom is flanked by an amino group and a carboxyl group, making up the backbone of the aa. The R group differs between amino acid types.

The structure of the protein is divided into four levels. This is illustrated in Figure 1.2.

- **Primary structure**: the sequence of the aa, the building blocks of proteins. This sequence is determined by the DNA sequence. As DNA sequencing becomes more accessible, the amount of primary structures available rises almost exponentially (Branden and Tooze, 2012). These sequences can't tell a lot about the biology of the protein (Branden and Tooze, 2012). Similar aa sequences do not necessarily have a similar protein function.

- **Secondary structure**: the regular structural elements. These structural elements are formed by hydrogen bonds, created between the hydrogen donors in the nitrogen part ($N-H$) and the hydrogen-acceptors in the carboxyl group ($C=O$) of the aa backbone (see Figure 1.1). These structures stabilize the protein. Different types of secondary structures are known: the α-helix, β-sheet (also called pleaded sheet) and random coil are the most important ones. More information is given in Section 1.2.1.

- **Tertiary structure**: the 3D pattern of the folded protein. In contrast to the first two structures, the tertiary structure is directly related to the function of the protein. This can be seen as the 3D packing of the secondary structure. Sometimes, motifs can be recognized in the way these secondary structures are organized. The tertiary structure of a protein is stabilized by different interactions, such as Van der Waals forces, disulfide bonds or hydrophobic interactions (Branden and Tooze, 2012; Van Damme, 2018).

- **Quaternary structure**: the interaction of different protein monomers. Different individual protein molecules can assemble together to form a functional protein cluster. Often, multiple protein monomers will interact to form a functional polymer. An example of such a protein cluster is hemoglobin (Lukin et al., 2003).

1.2.1 Secondary structures: In detail

Secondary protein structures are stabilized by hydrogen bonds between relatively small parts of the protein. The hydrogen bonds are formed between the atoms of the backbone of the protein. It is important to know that mankind invented the concept of secondary structures. Although these structures are defined well, differences in secondary structure allocations within proteins can occur, even when the tertiary structure is known. There tends to be small inequalities between what different algorithms define as an α-helix or β-sheet. Most of the time this is because it is not clear whether an aa at the end of an α-helix or β-sheet is still part of the secondary structure. For this reason, the predictive limit of existing algorithms is 88-90 % (Rost,

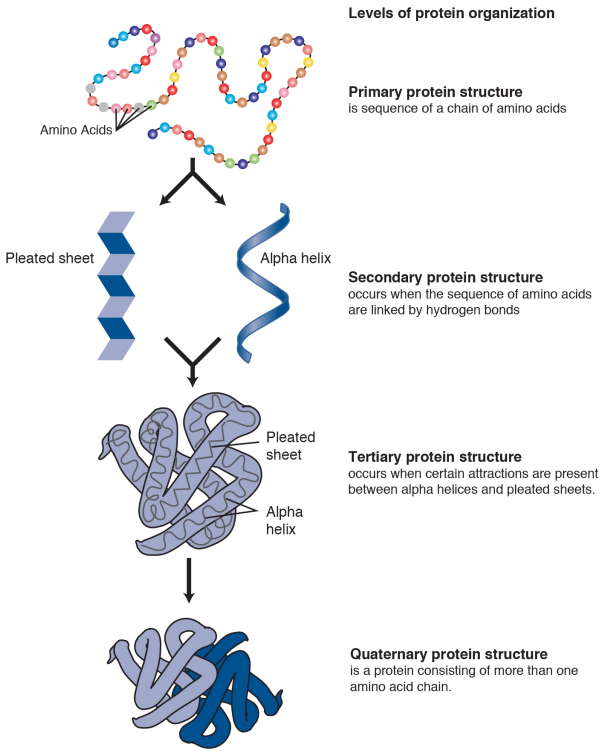


Figure 1.2: The different levels within protein structure. Courtesy: National Human Genome Research Institute

2001). The current practice is using the algorithm of the Dictionary of Protein Secondary Structure (DSSP). However, for submissions in the protein databank (PDB), the database that contains all known protein structures, it isn't required to use this algorithm (Berman et al., 2003). Therefore, variations within this database occur.

There are two systems for defining secondary protein structures. The first system uses three different types: α-helix, β-sheet and coil. In Figure 1.3, a protein is visualised, distinguishing the different secondary structures.

The first type of secondary structure is the **α-helix**. In these helices, the $C = O$ of the aa on position n has a hydrogenbond with the $N - H$ of the aa on position $n + 4$, which is the aa four positions downstream of aa n in the protein sequence. Some aa are more commonly found in α-helices than others. Proline, for example, does not fit inside a helix, whereas Alanine, Leucine, Methionine, and Glucine occur often in α-helices. Helical structures are often found in fibrous and globular proteins (Rehman and Botelho, 2018).

The second type of secondary protein structure is the **β-sheet**. A β-sheet is built from different β-strands. β-strands stabilize each other with hydrogen bonds and do not

need to be sequentially close to each other. This means that the aa composition at the end of the protein possibly influences the secondary structure at the beginning of the protein. β-sheets can be parallel or anti-parallel. A parallel sheet is defined as a sheet where all strands are directed in the same direction (all N-C or all C-N) whereas in anti-parallel sheets the directions of the strands alter within the sheet.

In this classification system, every part of the protein that isn't an α-helix or a β-sheet is called a coil. Those regions are less structured. Nonetheless, there are still some structural elements present in coil regions. For this reason, DSSP proposed eight classes of secondary protein structures (Kabsch and Sander, 1983). This way of classifying the secondary structure provides more information about the actual 3D formation of the protein.

In the DSSP classification system, three types of helices are defined: the 3_{10}-helix, α-helix and π-helix. The 3_{10}-helix has hydrogen bonds between positions n and n+3, where the π-helix has hydrogen bonds between positions n and n+5 (Fodje and Al-Karadaghi, 2002). The α-helix is way more abundant than the other two, and the only one regularly observed in natural proteins.

Next to these three helices and the β-strand, the bridge, turn and bend are defined. If an aa isn't part of any of these seven categories, it is put in the category 'others'.

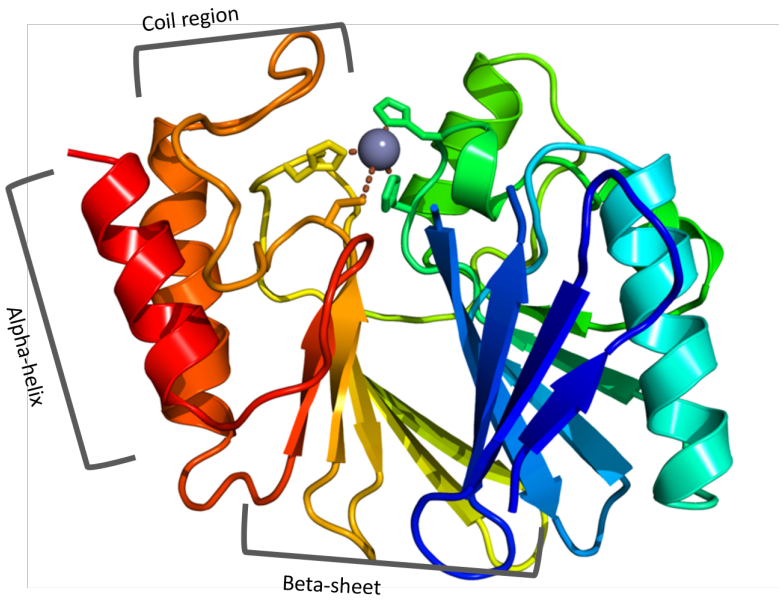


Figure 1.3: The protein structure of metallo-beta-lactamase in cartoon representation. The α-helices are visualised by helical structures and the β-sheets by arrows. Coil regions are visualised as thin lines. It can be clearly seen that α-helices exist out of consecutive aa. β-sheets are built from different strands, where aa within a strand are consecutive, but different strands aren't.

1.3 Protein Folding

Every molecule wants to minimize its chain entropy. This fact drives proteins to their folded states. Reaching a folded state is possible because the energy landscape of proteins is funnel-shaped, as can be seen in Figure 1.4. For this reason, most unfolded conformations have a high energy state, and only a few low energy folded structures are achievable. In order to reach those low energy structures, most proteins pass through intermediate folding states, present as local minima in the folding energy landscape. It is not sure that the native state of a protein (their energetically stable 3D structure) is the state of the protein with the lowest energy. Next to the lowest energy, the accession of the fold plays an important role too. Global minima that are extremely hard to attain will almost never be reached, although they would be favorable in terms of energy levels.

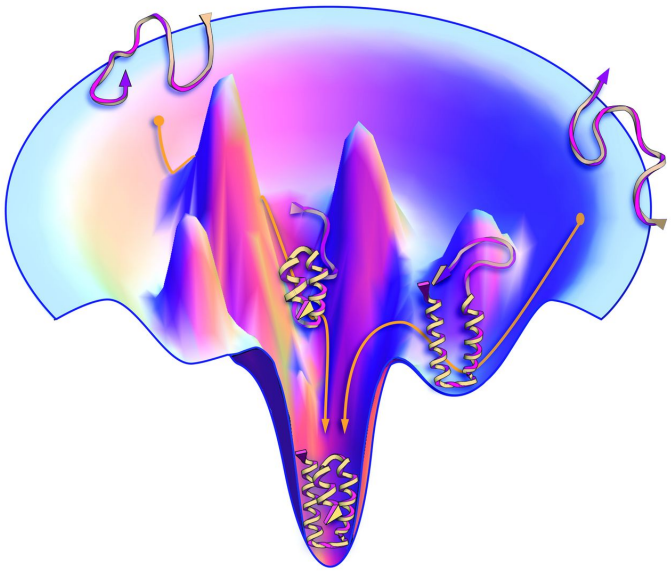


Figure 1.4: The energy landscape of a protein. This energy landscape is funnel-shaped, only a couple of conformations have a low energy level. Unfolded or partially folded proteins have high potential energies. The folding occurs through trajectories (Dill and MacCallum, 2012)

Often, proteins have multiple native states. Influenced by the environment or possible interaction partners, proteins might change conformation. We shouldn't look at proteins as being a rigid structure. The structure of a protein is flexible and can change with the environment, increasing the difficulty to determine the protein structure.

DNA contains all information to create proteins correctly. Parts of the DNA are transcribed to messenger RNA (mRNA) after which this mRNA is translated to an aa sequence, a process mediated by ribosomes and transport RNA (tRNA). However, this aa sequence is only a 1D structure. How cells create a fully functioning protein out of this

1D structure is still not completely known (Hartl and Hayer-Hartl, 2009; Van Damme, 2018). Only a fraction of the proteins fold to their native state independently. Most proteins are helped by chaperones, a class of proteins that bind to and stabilize unfolded or partially folded proteins (Gething and Sambrook, 1992).

In vitro, numerous proteins fold to their native state independently. All information for the tertiary structure of a protein is contained in the polypeptide chain (Anfinsen, 1973). Even though mankind has not succeeded yet in extracting this information,the possibility does make us hopeful.

1.4 Protein structure: Experimental determination

As the structure of proteins cannot yet be derived directly from the sequence based on biological explanations, new techniques have been developed. Three of these techniques are commonly used at the moment, all with their advantages and disadvantages: X-ray crystallography, nuclear magnetic resonance (NMR) and cryo-electron microscopy (cryo-EM).

X-ray crystallography

For this technique, the protein is crystallized and then irradiated with X-rays. These rays produce a series of spots: reflections. The crystal is turned around, and so, the reflection pattern changes. These reflection patterns are analyzed in combination with available chemical information for this protein. Together, the location of every electron can be determined. Using that information, atoms can be allocated. Interpreting these reflection patterns is a really specialized and difficult job, as the interpretation of the diffraction pattern is mathematically complex, data processing problems occur and numerous processing steps need to be completed (Smyth and Martin, 2000). At this moment, 89 % of the known protein structures are obtained using this technique. The advantages are that it can be used for all protein sizes, and that the obtained resolution of the structure is generally high.

However, X-ray crystallography is not suitable for all proteins as it can be extremely hard to obtain high-quality crystals. The obtained structure of the protein is rigid. As stated before, proteins change conformation, so this is an oversimplification. In total, it takes a couple of years and a lot of money to obtain the X-ray data of a protein, making it very expensive, time-consuming and infeasible to obtain all protein structures this way (Drenth, 2007).

Nuclear magnetic resonance spectroscopy

The second common method used to obtain protein structures is NMR. For this method, a pure and highly concentrated protein in solution is used. The proteins is placed in an external magnetic field, so that the H-atoms undergo a chemical shift. Depending on the size of the chemical shift, the position of the H-atom in the molecule can be determined. This technique is used to determine 9% of the protein structures. Two advantages of this method are the fact no crystals need to be obtained and that the structure of the protein is determined in solution. Because obtained in solution, the representation of the structure in vivo is better and different conformations of the same protein are created at the same time. As such, the protein can be modeled as a dynamic molecule. Next to that, NMR can be used to model interaction affinity and see which proteins interact (Nwanochie and Uversky, 2019).

The biggest disadvantage is the size limit, NMR is therefore mainly used for small molecules. When big proteins are present, H-atoms are superabundant, and interpreting the spectra will become infeasible. For this reason, the size limit of molecules for NMR spectra is 250 kDa (Wider and Wüthrich, 1999) (Wüthrich, 1990).

Cryo-electron microscopy

Cryo-electron microscopy is a new technique to determine protein structures. In order to obtain these structures, electrons are shot at a molecule frozen in solution. Although the technique was developed in the ’70s, it has only been used in protein chemistry in recent years. Before, the resolution of the technique wasn’t sufficient. In 2017, the nobel prize was given to Jacques Dubochet, Joachim Frank and Richard Henderson for their role in the development of this technique. The advantages are the speed (it is faster than the other techniques), the lack of size limit, and the easiness to deduce flexible structures. The main drawback is the resolution, which can not yet reach the standard set by X-ray crystallography (Nwanochie and Uversky, 2019).

No optimal technique to experimentally determine the protein structure is currently available. The method that is fast, cheap and has a high resolution still needs to be invented. As an alternative, protein modeling and protein structure prediction are often performed to determine protein structures.

1.5 (Secondary) protein structure prediction

As stated before, the prediction of protein structures is one of the big remaining problems within biology. Solving this problem can help drug design (Ronin et al., 2018; Ibrahim et al., 2019) and the design of enzymes in general considerably. It can also help in analyzing disease-causing mutations (Fiedorczuk et al., 2016).

A lot of methods to predict protein structures start from a template. In these methods, the structure of a new protein is resolved using the known structure of a similar protein. This is called homology. Very similar protein sequences can have a similar structure, but this isn't always the case (Kaczanowski and Zielenkiewicz, 2010). When using homology modeling techniques, the prediction of the unknown protein will be biased towards the used template, which isn't desired. Most proteins don't have close homologs of which the structure is already determined. In this case, no templates are available, so no homology modeling can be performed.

Predicting the protein structure de novo (without templates) remains challenging. A technique to simplify this problem is the divide and conquer technique. With this technique, you split up the big problem in smaller problems thatdemand simpler solutions. The best-known smaller problem for protein structure prediction is the prediction of secondary protein structure (Yang et al., 2016). Multiple secondary structures are packed together to form structural folds. These folds make up the classification of tertiary structures. This way, protein secondary structure prediction can help determining the 3D structure of the protein

Next to this, the secondary structure influences the way and the speed of the folding process (Zhou and Karplus, 1999; Plaxco et al., 1998). Also, secondary structures are more conserved than sequences and have a more direct link to function. For this reason, predicted secondary structures are useful in protein sequence alignment and protein function prediction (Zhou and Zhou, 2005; Godzik et al., 2007). Disease-causing mutations are often located in α-helices or β-sheets (Yue et al., 2005). The predictions of secondary structural elements can help the experimental determination of protein structures (Fiedorczuk et al., 2016).

Next to secondary structure prediction, some models try to facilitate the protein structure prediction by predicting backbone angles, because they provide an in-depth description of protein local confirmation (Hanson et al., 2019; Gao et al., 2018). Other models try to predict solvent accessibility or residue-residue contacts, amongst other characteristics (Kurgan and Miri Disfani, 2011).

In recent years, de novo secondary protein structure prediction happens with machine learning (a clear definition and explanation of this concept is given in chapter 3). As a substantial amount of protein structures is determined experimentally, data to train machine learning models is available. Most models in literature use deep neural network architectures to predict secondary structures (Torrisi et al., 2018; Wang et al., 2016). The machine learning landscape evolves extremely fast; new and better models pop up like mushrooms. One field within machine learning that might be of interest for secondary protein structure prediction is Natural Language Processing (NLP).

1.6 Natural language processing (NLP)

According to Bird et al. (2009), natural language processing is a broad field of research: "*We will take Natural Language Processing — or NLP for short – in a wide sense to cover any kind of computer manipulation of natural language. At one extreme, it could be as simple as counting word frequencies to compare different writing styles. At the other extreme, NLP involves "understanding" complete human utterances, at least to the extent of being able to give useful responses to them.*"

In the last years, a lot of research is done in order to use machine learning and especially deep learning, in the field of NLP (Socher et al., 2012), from translation (Cho et al., 2014) to understanding spoken language (Mesnil et al., 2013). The interest in these fields is high because automatic translation and speech recognition are tools with very direct and broad application fields. Language is extremely flexible and quite messy which complicates NLP. A word can have different meanings or translations, depending on its context. This context can go sentences back. The rules that govern language are poorly understood. Citing the blog of Chris Nicholson: "*So it's probably more fitting to think of sentences folding like proteins in three-dimensional space, with one part of a phrase curling around to touch another part with which it has a particularly strong affiliation, each word a molecule pearling on a polymer.*" This indicates clearly why deep learning algorithms designed for natural language processing could work too for protein folding. Both problems are sequencing problems in which order is important and the lengths of the sentences aren't fixed. The mechanisms that rule them aren't understood completely yet and the outcome can be depending on the very broad context. Adapting translation algorithms in order to fold proteins could be a good idea.

CHAPTER 2

AIMS

As stated in Chapter 1, the prediction of protein structures based on sequence remains an important problem. Knowledge of protein structure can contribute to innovation in numerous fields. Protein secondary structure prediction is a long-standing subproblem in the field of protein structure prediction, and can help predicting protein structures as well as protein function. All recent models that predict protein secondary structure based on sequence are based on machine learning models. This dissertation's main goal is to create new machine learning models which are able to accurately predict these protein secondary structures, and to compare their performance to the current state-of-the-art. A first step in this process is the generation of datasets to train the machine learning models. A schematic overview of this thesis' structure is shown in Figure 2.1. Two main parts can be distinguished: data retrieval and model creation.

2.1 Data retrieval

A first aim of this master dissertation is to create clean databases for model training. This is necessary because available databases lack in transparency and/or completeness. Two databases are created: one only containing sequence information for all proteins in the database, and one containing evolutionary information about all proteins. Due to time constraints, it was impossible to generate an entirely new evolutionary database for this master dissertation; hence, a literature-based database is used. These datasets are filtered extensively to delete artefacts. Afterwards, a thorough analysis of the datasets is performed. Possible biases and imbalances are highlighted, and ways to handle them are suggested. This is discussed in Chapter 4.

2.2 Model creation

A second aim of this master dissertation is the creation of a new type of machine learning model to predict protein secondary structure. Chapter 3 mainly focuses on

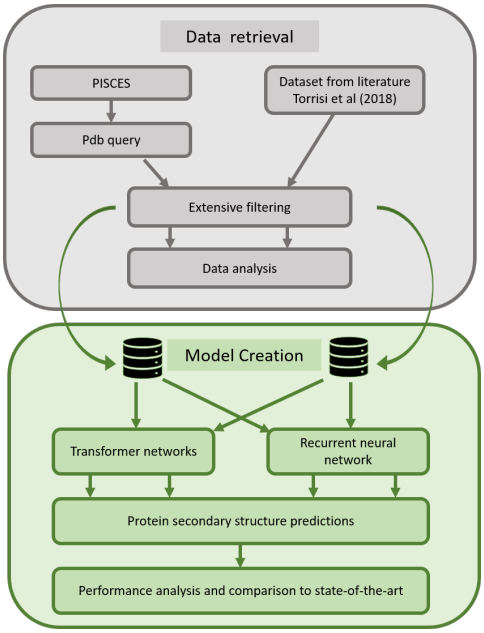


Figure 2.1: A schematic overview of the pipeline of this master dissertation.

machine learning concepts necessary to understand all principles. Additionally, this chapter provides an overview of the current state-of-the-art, its advantages and its shortcomings.

These current protein secondary structure prediction practices are almost always based on techniques used in NLP, such as convolutional and recurrent neural networks. NLP is a very active research field within machine learning. Recently, the Transformer model, which is a new type of NLP model, has outperformed existing models in numerous tasks (Vaswani et al., 2017). In Section 3.8, the mechanism behind this model is discussed and possible reasons for it outperforming previous models are examined.

As these Transformer models are designed for NLP problems, adaptations to the model need to be made. The necessary adaptations are defined, possible Transformer models are tested, and different Transformer architectures are constructed and compared.

The adapted Transformer models are then evaluated, and their performance is thoroughly compared to the current practice in literature, for which an extra model is created. This extra model's architecture is based on the model architectures in current practices – a combination of Recurrent Neural Networks and convolutional neural networks – so the same data can be used to train both types of models, which results in an unbiased comparison. The predicted protein secondary structures are completely analysed in Chapter 5. To conclude, further prespectives are highlighted in Chapter 6.

CHAPTER 3

MACHINE LEARNING: USEFUL TECHNIQUES

3.1 Introduction

Nowadays, machine learning is a hot topic. Some people think that, with machine learning, in a couple of years the earth will be ruled by cyborgs. Machine learning can be useful, but cyborg-presidents most probably will not rule the world in 50 years. In the news, machine learning was given attention when it was used to create self-driving cars, and when it defeated the best GO player in the world. Machine learning has applications in day-to-day life too: virtual assistants such as Siri or Alexa, prediction of traffic jams, ads on social media, spam filtering, product recommendations, et cetera.

An interesting definition of machine learning was given by Arthur Samuel (1959) : "*Machine learning is the study that gives computer the ability to learn without being explicitly programmed.*" We can look at a machine learning algorithm as a fancy pattern finder that can label data. Imagine a conveyor-belt of garbage that needs to be sorted. This could be done by people, who recognize paper/glass/residual waste and sort it accordingly. On the other hand, an explicit program could be written that defines the different types of garbage: the different classes. For every piece of waste, it then determines to which class it belongs, based on what the programmer entered into the system. Machine learning works differently. In machine learning, data is given to the program. The algorithm ingests pieces of garbage data, along with their classes. This set of data is called the training data. This data will be used by the algorithm to learn. The more data is present in the training data, the better the algorithm will learn. The algorithm trains by recognizing patterns in the training data, for example, the garbage that belongs in the glass section is often bottle-shaped and heavier or paper waste is often light and white. These patterns will be used to predict the class of future garbage. Machine learning models are often more precise than explicit

programs because they are more flexible and will be able to learn data properties the explicit programmer forgot or never knew about.

This is only the case if the training data fed to the algorithm is of good quality. If mistakes are present in the training data, the algorithm will have difficulties classifying correctly. The machine learning algorithm can at most be as good as the data provided.

Before using this machine-learning algorithm to sort our garbage, we might want to check how accurately it performs. How often does it manage to sort well? A score metric needs to be provided. This can be easily done by measuring accuracy. An accuracy of 90 % means that 90 % of the garbage is sorted well. Sometimes accuracy is a bit too plain to reflect the model's performance. If misclassifying paper trash as residual waste is less a of a problem than misclassifying glass as residual waste, this should be taken into account in the scoring metric. For this reason, different scoring metrics exist.

3.2 (Un)supervised learning

Two big groups of algorithms exist within machine learning: supervised learning and unsupervised learning.

Imagine having a lot of apples. For each apple, you have data: the size of the apple, the sweetness, the texture et cetera. This data gives a representation of the apple. Together, they form the features of the apple, making up your dataset. The goal is to divide the apples into two stacks: one for green and one for red apples. However, this computer can not see colors. So a model that extracts the information out of the features needs to be created. In order to do so, a set of apples is selected: the training data. In supervised learning, the labels (e.g. the color of the apples) are known for this set of apples. This information is used for creating models that predict the color of unknown apples (apples that were not present in the training set) given their features (size, sweetness,...). These models extract patterns out of the features and use those to classify. For example, green apples are sourer than red ones or red apples are bigger. When the right model is provided with data of apples it has never seen before, it could be able to predict the color of those apples well when the correlated features of those apples are known. This way, the computer can create two stacks of apples based on color. This can be seen on the left part of Figure 3.1. Examples of supervised machine learning models are linear/ logistic classification, random forests, support vector machines and neural networks (nn's).

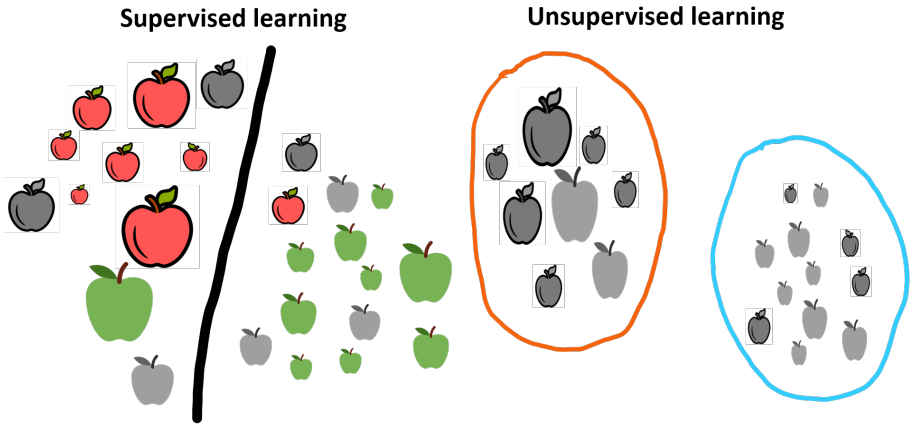


Figure 3.1: The difference between supervised and unsupervised learning. Left: supervised learning: the training data is labeled: colored. Based on this data a model can be trained. For the non-training data (grey), the color is predicted based on which side of the black line the apples are on. Right: unsupervised learning: no labels are known. Apples are clustered based on the similarity of their feature representation.

Now imagine that the data provided is collected by a colorblind person. This person could measure all features, but could not determine the color of the apples. So the data is unlabeled, for none of the apples (data points) in the training set the label (color) is known. Without labeled data, it is impossible to create a model that predicts the label (the color for the apples). Nonetheless, information can be extracted. Models used on unsupervised data recognize patterns in the data and group the data based on those patterns. An illustration of this principle can be observed in the right part of Figure 3.1. The underlying inherent structure or distribution of the data will be modeled. This way, we can explore and learn about the data. Examples of this are groupings or associations within the data. In unsupervised learning, there is no clear right or wrong. Checking on these computer algorithms is more difficult. It isn't possible to decide based on which criteria the grouping will happen, it can be based on taste, shape, size, or most probably a mixture of all those properties. Unsupervised algorithms thus cannot be used to create two stacks of apples based on color, but they can create two stacks based on similarity. Examples of unsupervised learning are clustering, tSNE and autoencoders.

Within supervised learning methods, two big groups of problems can be distinguished: regression and classification. A regression problem is a problem where a continuous output value needs to be predicted, for example the price of a house. A classification problem is a problem where a class needs to be predicted. For example, predicting apple color.

Sometimes, instances need to be classified into more than two classes. When for every instance only one class needs to be predicted out of multiple, this kind of prob-

lem is called a multiclass classification problem. When for some instances more than one class needs to be predicted out of multiple, the problem is called a multilabel classification problem. Both have their own specific solving strategies.

3.3 Deep learning

Machine learning evolved a lot in the last 60 years. Since 2006, deep learning, a branch of machine learning, is becoming more and more popular. Within deep learning, both algorithms for supervised and unsupervised learning are available (Deng et al., 2014).

According to LeCun et al. (2015), "Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction." The models in deep learning are more complex in construction than, for example, linear or logistic regression. For this reason, they can detect more complex, non-linear patterns. Deep learning has applications in many fields, like speech recognition (Amodei et al., 2016), object detection (Ouyang et al., 2015), genomics (Clauwaert et al., 2018) et cetera.

In deep learning, data can be processed in its raw form, which is a huge advantage compared to other machine learning techniques. Previous techniques often required careful feature extraction. With deep learning, a complete audio file can be given as input, whereas with other techniques specific informative frequencies have to be chosen, as only meaningful features could be fed to the model. Deep learning algorithms learn which frequencies are meaningful during training. This can be seen in the middle column of Figure 3.2.

Deep learning techniques can extract those meaningful features by using representation learning on multiple levels. This happens by using simple, non-linear modules. Every module can transform a representation into a higher-level representation. When multiple modules are combined, complex representations can be learned. This can be clearly seen in Figure 3.2. The rightmost column shows that first, simple features are learned from the data. These simple features are then fed to extra modules in order to learn higher-level feature representations, which can consequently be mapped in order to achieve the requested output (Goodfellow et al., 2016).

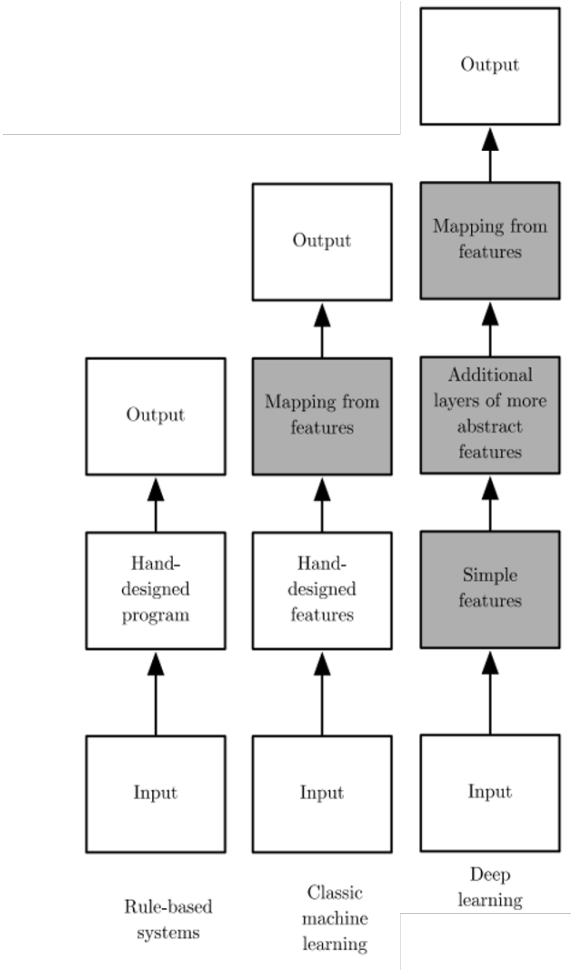


Figure 3.2: Flowcharts representing different parts of different groups of machine learning models. Shaded boxes indicate components that are able to learn from data (Goodfellow et al., 2016).

3.4 Artificial neural networks

Deep feed-forward neural networks are quintessential models in deep learning (Goodfellow et al., 2016). An understanding of these simple deep learning models is necessary to understand the principles of more advanced deep learning models. Artificial Neural Networks (ANN's) are models that are loosely based on how human neurons work. In biological neural nets, every neuron is connected to thousands of other neurons, sending and receiving signals. ANN's aren't as big and complex as biological neural networks and the neurons are often called nodes. Connections between nodes are named edges. Nodes that are connected by an edge can transmit signals. In feed-forward networks, these can only send information from input to output.

In ANN's, nodes are organized in layers. Normally, neurons of one layer only connect to the nodes of the previous and the next layer, by edges. The value of a node is a

linear combination of the values of the nodes of the previous layer, weighted by the edges.

A basic example of an ANN is visualized in Figure 3.3 and will be used to explain the concept. The different colored circles represent nodes, where the arrows are the edges. An ANN is normally read from left to right, the direction of the edges. Every ANN contains an input layer (blue) and an ouput layer (red). The input layer holds all inputs given to the system, and the output layer is responsible for showing the resulting output. The inputs can be raw data. In this example, the input layer (a_0) consists of three blue circles: nodes. Each of these nodes is connected to every node in the second (green) layer of the network, this is called a fully connected layer.

The second layer of the ANN is the first hidden layer. All layers between the input and the output layer are called hidden layers. This layer can extract simple features from the data. Multiple nodes are present per layer to extract multiple interactions. The nodes of the first hidden layer are given as input for the second hidden layer (yellow). This again is a fully connected layer. This layer can extract more complex features, in higher orders. These complex features are used to calculate the output layer (red). This layer is responsible for producing the final result.

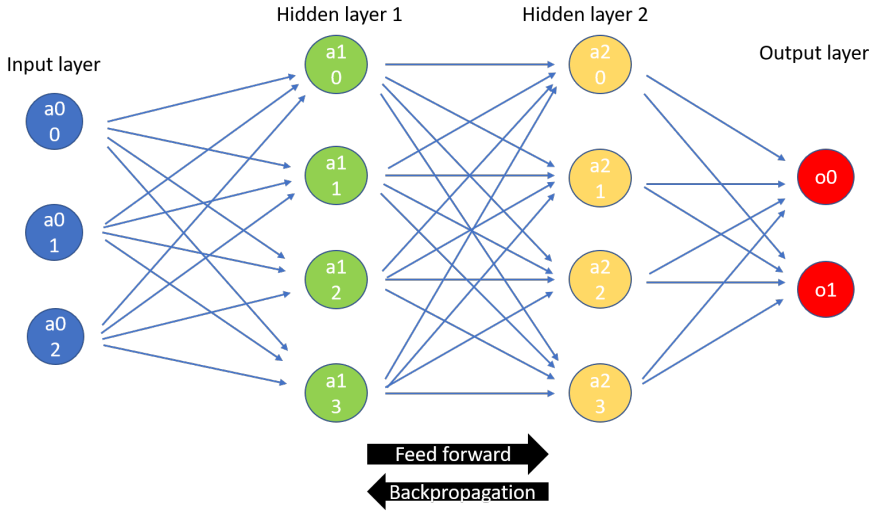


Figure 3.3: a schematic representation of a basic ANN

The nodes, in each layer of the ANN, are used as a representation for the same functions. Every node is a linear combination of the previous nodes, weighted by the egdes. A bias is added, and lastly an activation function is executed. This gives following equation for hidden layer one:

$$\mathbf{a^1} = \sigma(W^0 * \mathbf{a^0} + \mathbf{b^0}) = \sigma(\mathbf{z^1}) \tag{3.1}$$

Where $\boldsymbol{\alpha^1}$ represents all nodes in hidden layer one. A linear combination over all nodes of the previous layer ($\boldsymbol{\alpha^0}$) is calculated, where W^0 is a matrix containing all weights, corresponding to the edges. $\mathbf{b^0}$ is the added bias. σ represents the activation function. This activation is used to regulate the range of values a node can contain. More importantly, they add non-linearity to the ANN. In this example the sigmoid function is used, but other activation functions exist, such as rectified linear unit (reLu) and tanh.

This is how signals are transmitted through ANN's. The weights and biases of an ANN can be tuned, this is how an ANN learns. When you start, the weights and biases are initiated randomly, so it happens only by chance that the outputs are predicted correctly. When an ANN is trained, the weights and biases are tweaked for every training set, and in this way, the model can adapt.

In order for an ANN to adapt in the right way, it needs to know what is wrong and what is right, so it can learn from its mistakes. This is done by optimizing a cost function. A cost function results in a low value when the ANN can predict the output right, and a high value when the ANN makes wrong predictions. A straightforward cost function that can be used is the mean squared error (MSE) (LeCun et al., 2015). When the true label is represented by t, the predicted otput by O and n training samples are used, this function is:

$$Cost_{MSE} = \frac{1}{n}\sum_{k=1}^{n}(O^k - t^k)^2 \tag{3.2}$$

The bigger the absolute difference between the predicted output and the ground truth, the bigger the value returned by the cost function. Depending on the problem, other cost functions might be a better fit (Zhang and Sabuncu, 2018). The minimization of this function happens by adapting the weights and biases, as stated before. How these are adapted, is determined by the the partial derivativeof the cost fuction with respect to all model weights. In order to calculate these, backpropagation is used.

3.4.1 Training a neural network: Gradient descent and backpropagation

The principle of gradient descent is used to optimize the cost function of an ANN. With this method, we can determine which weight(s) should be modified. The gradient of a function f (of several variables) is the first derivative of that function with respect to all its variables. This means that for every variable of f, the partial derivative is calculated. The gradient indicates the direction of the steepest ascent/descent of the function output for a given input. As the cost function needs to be minimized, the

descending direction of the gradient calculated for the cost function for a given input is followed. A gradient is represented as follows, where $x_1, ...; x_n$ are the variables of the function:

$$grad(f) = \nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_m}) \tag{3.3}$$

In ANN, this means that the gradient of the cost function is calculated with respect to all the weights and biases present in the model, as these are the variables. The direction of change is determined by the gradient and the size of the change is determined by the learning rate and the gradient. This learning rate is often fixed for the process but can be optimized as a hyperparameter. The bigger the learning rate, the faster the algorithm will learn, but the more chance there is of ending up with a sub-optimal set of weights and biases. When the learning rate is too small, they are prone to get stuck in local minima too. There exist models with adaptive learning rates (Plumb et al., 2005).

The size of a certain partial derivative at a certain point indicates how much influence that specific variable has on the ouput of the function. It gives an indication of the importance of that variable for the given input.

In order to calculate different partial derivatives present in the gradient, backpropagation is used. Backpropagation is called this way because the calculations of the partial derivatives happen from output to input by using the chain rule. This way, partial derivatives of the costfunction regarding all model parameters can be calculated and the complete gradient is obtained. The step size can be calculated by multiplying the gradient with the learning rate. The new parameter values equal the old parameter values - step size.

The gradient of the cost function can be calculated as an average value over all data points fed to the ANN. This would ask an enormous amount of partial derivatives to be calculated everytime all data is fed to the model, when working with a lot of data or big models. Especially when taking into account that often multiple repetitions of training (epochs), using the whole training set, happen. In order to speed up the training, stochastic gradient descent (SGD) is often used (LeCun et al., 2015). In this type of gradient descent, the model processes a few samples at a time. This way, only a few outputs and errors are generated. Based on those few, an average batch gradient is calculated, resulting in a noisy estimation of the real gradient. The weights/biases are adapted according to the noisy estimation. This method finds a good set of weights quickly, when compared to more elaborate techniques (Bottou and Bousquet, 2008).

The example used above and visualized in Figure 3.3 is a basic conformation of an ANN. More layers and more nodes per layer could be added. Next to feed-forward ANN's, feedback ANN's exist too. Convolutions can be added, or attention mechanisms. Depending on the problem, different types of adaptation are made to the basic ANN, and the possibilities are almost endless.

3.5 The possible flaws of ANN's

Neural networks, as described above, are very powerful tools in machine learning. These networks succeed in extracting features from raw data (Chen et al., 2016), dealing with missing data (Zhang et al., 2018b), extracting non-linear information or generalizing unseen relationships. But even ANN's aren't perfect.

One of the biggest flaws of ANN's is the fact that they are usually black boxes. Often, it isn't known why certain ANN's work well, and how exactly the ANN's extract their features (Mijwel, 2018). In a lot of applications, interpreting the models could help to gain knowledge. In the last years, techniques are created in order to illuminate the black box (Olden and Jackson, 2002; Yosinski et al., 2015). Nonetheless, ANN's are still more difficult to interpret than most other machine learning algorithms. When ANN's are, for example, created to predict secondary protein structures, it will be hard to gain knowledge about how the folding happens in vivo.

Gradient descent is used as an optimazation technique in different research fields. Gradient descent methods can be limited to the 'local minima' problem. With gradient descent, only descending steps are taken, but often, the lowest valley is at the other side of the mountain range. As the gradient descent algorithm does not walk up, this lowest valley will not be reached.

In machine learning, the goal is to create a model that does not only perform well on the data used for training, but also on new data. This means machine learning models should be able to generalize. If not, it will fail on predicting outputs for unseen data.

3.5.1 The problem of overfitting

When training the model, the cost function is minimized by learning about the training data and adapting weights. On this training data, the model isn't always perfect. We can compute a **training error**. In machine learning, the goal isn't to minimize this error, but the **generalization error** or **test error**. This error is defined as the expected error on a new input: an input that has not been used to train the model.

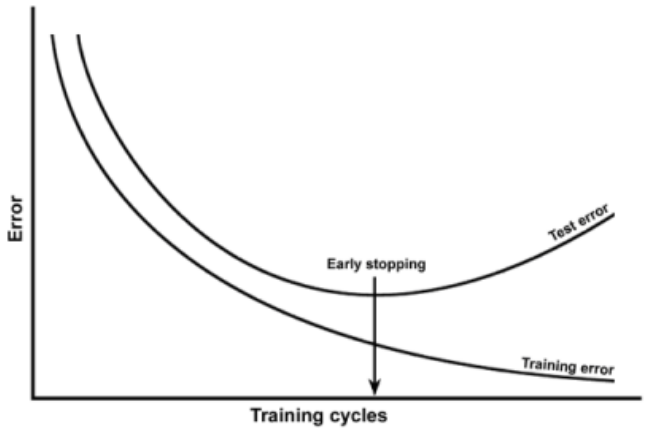


Figure 3.4: a scheme showing the train and test error in function of the amount of training cycles

Before the ANN is trained, the weights are assigned randomly. In every training cycle, the cost function is calculated, the gradients are calculated with backpropagation and a step towards a lower value of the cost function is made. As can be seen in Figure 3.4, both the train- and the test error drop steeply at the start of training, in the first training cycles. In this training phase, the model learns to recognize some general features, that are widely applicable. After a while, when the model continues its gradient descent, only the train error will drop. The test error will start to rise. This is because the model starts to learn very specific patterns about the training data: patterns that aren't generalizable. It will model the noise. This is called overfitting. The model fits the data so well that it isn't generalizable anymore. In order to prevent overfitting, the training of the model is stopped from the moment the generalization error starts to rise. Early stopping is indicated in Figure 3.4.

When the test set is used to tune any kind of hyperparameter, such as the amount of epochs the model should run, the results will be biased towards the testset. Consequently, the test set isn't independent anymore, and thus can't be used anymore to report performances. For this reason, a third dataset needs to be created: a validation set that is used to tune all necessary hyperparameters. To be able to choose the hyperparameters optimally and to prevent reporting wrong results, this validation set should be independent from both the training and the test set. This way, the test set can stay completely independent from the model. The training is stopped based on results on the validation set.

When a model isn't trained thoroughly, what happens when the training is stopped to prevent overfitting, most of the time the function will not have reached a local minimum on the trainingset. The best model for the train set probably won't be the

model that minimizes the test error. For this reason, ANN's don't often suffer from the 'local minimum' problem.

Overfitting can be delayed by training the model on more data. Normally, the more data provided to the network, the longer the model will be able to train without modelling noise. Also, the more parameters are present in the ANN, the bigger the chance on overfitting. This is why a lot of data is needed to train big ANN's. Another solution can be to penalize big weights (Tu, 1996). In general, overfitting is a big problem when using ANN's. Awareness of this problem should be present at all times.

3.5.2 Choosing the right architecture

Another drawback of ANN's is the difficulty of choosing the right ANN for a certain problem. Multiple decisions need to be made: will I use a standard feed-forward network or will I use more complex builds? How many layers will I use? How many nodes per layer will I use? Some of those more complex builds are discussed in Section 3.6. The way a certain ANN is selected is often based on trial and error (Mijwel, 2018; Gurcan et al., 2001). This makes the selection and training of an ANN a time consuming job. Some studies provide tools to find the optimal ANN architecture, but these tools are far from perfect (Gómez et al., 2009; Messer and Kittler, 1998). They can help to select the hyperparameters of a model, such as the number of hidden layer or nodes per hidden layer. Automatic tuning tools have hyperparameters too that need to be optimized (Goodfellow et al., 2016), only shifting the problem. Grid searches can help in finding the right architecture too, by evaluating model parameters in a certain range. Different ANN's can handle different types of input data, and this needs to be taken into account too.

Next to the network architecture, the activation function needs be chosen. In the example given in Section 3.3, the sigmoid activation function is used. Other activation functions that are used often are tanh and ReLu. Nowadays, ReLu is used the most, because it trains faster in networks with many layers. The reason for this is that ReLu can create real zeros in stead of values close to zero, which is very suitable when working with sparse data (Glorot et al., 2011). The disadvantage of ReLu compared to the other activation function is its non-differentiability at zero. The choice of the activation function can have a lot of influence on the performance of the ANN (Karlik and Olgac, 2011).

In machine learning, it is important to define the right cost function as well. As stated above, a cost function that is often used for regression problems MSE. However, this

function isn't always suitable. For example, regression and classification problems need different cost functions.

A cost function that is used very often in multiclass classification is the cross entropy loss function. This function takes into account only discrete outputs are possible. Also, using cross entropy loss speeds up the backpropagation. This function is defined as follows:

$$CE = -\sum_{i=1}^{C} t_i log O_i \tag{3.4}$$

Where C classes need to be predicted, t_i is the ground truth for class i and O_i is the output probability for class i. Often, this cross-entropy loss is combined with a softmax layer. This means that before calculating the loss function, the scores of the ANN are put through following function:

$$f(O)_i = \frac{e^{O_i}}{\sum_i^C e^{O_i}} \tag{3.5}$$

O_i from Equation 3.4 is then replaced by $f(O)_i$ from Equation 3.5. In the case of a multiclass classification problem, the labels are one-hot: for every output, there is only one correct class: t_p. For all other classes, Equation 3.4 equals zero. When combining Equation 3.4 and 3.5 and summing over all data points T taken into account for that step, the softmax-cost (cross-entropy+softmax) is calculated as follows:

$$-\frac{1}{T}\sum_{t=1}^{T} log \frac{\exp O_p^t}{\sum_i^C \exp O_i^t} \tag{3.6}$$

The cost function is the mean value over the loss functions for all datapoints. When preventing overfitting, the model is stopped when the average loss over the validation set is the lowest. This way, the model with the lowest value for the costfunction on an independent dataset is chosen.

When the different possible output classes don't all occur as frequently in the training set, the data is unbalanced. Unbalanced data influences the model performance. The less frequent occurring classes will be predicted less often. If the imbalance is big, it is possible to never predict the less frequent classes and still achieve high accuracies. For this reason, one might want to correct for class imbalance.This can be done be weighting the costfunction. Mistakes made on less frequent classes will be up-weighted proportionally to the imbalance. This way, all classes are given the same amount of attention, regardless their frequency of occurrence in the trainset. A more balanced model is trained.

The adaptation mechanism needs to be optimized as well. Generally, SGD with back-propagation is used to update the weights. In this case, the learning rate should be optimized. If this rate isn't suitable, the model will fail to optimize (Goodfellow et al., 2016). This is an important hyperparameter of the model to tune. An optimization algorithm that is used very often (Heffernan et al., 2017), is Adam. The algorithm uses stochastic gradient descent and can compute an adaptive learning rate. It is efficient, and little hyperparameter tuning is necessary (Kingma and Ba, 2014).

In order to evaluate the model predictions thoroughly, the right type of evaluation metric needs to be used. Model performance assesment is executed on the validation or test set. In multiclass classification problems, a prediction can be correct (a True positive: TP) or incorrect (an error E). When a prediction is incorrect, it might be interesting to know which class is predicted incorrectly. For this reason, confusion matrices are often created,as shown in Figure 3.5. These matrices indicate how often predictions are done right/wrong and if a prediction is done incorrect, it is visualises which error was made. This can give more insight into the models.

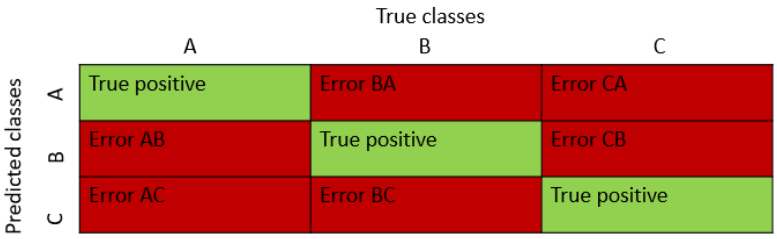


Figure 3.5: An illustrative example of a confusion matrix for multclass classification.

3.6 Advanced neural networks

3.6.1 Convolutional neural networks

Convolutional neural networks (CNN's) are types of neural networks that are designed for, and often used in, image recognition. An example of a CNN architecture is given in Figure 3.6. The ingenuity of CNN's is how it learns its features: by applying filters (also called kernels). These filters take a weighted sum over the neighborhood of a certain input. The weights are trained in such way that they can extract useful features, such as shapes. This filter moves over the complete input space and calculates one output value for every input. This way, the kernel captures temporal and spatial dependencies. Often, multiple kernels are used at the same time, in which case all kernels try to catch a different pattern. The outcomes of those different filters are stacked. Non-linearity is added with a ReLU operation. To reduce the dimensionality,

pooling is used after the convolution (Saha, 2018), where over a square of values, the maximal value (max pooling) or the mean value (average pooling) is calculated and used as input for the next layer. This is shown as well in Figure 3.6. This decreases the computational power needed to extract relevant features. The process of convolutions by using filters and pooling can be repeated, as in Figure 3.6.

Afterwards, the learned features are used to perform classification by putting it through a fully connected feed-forward layer. This way, non-linear combinations of the high-level features can be learned. Training is performed as usual, and both the weights of the convolutional filters as the weights of the feed-forward layer are optimized.

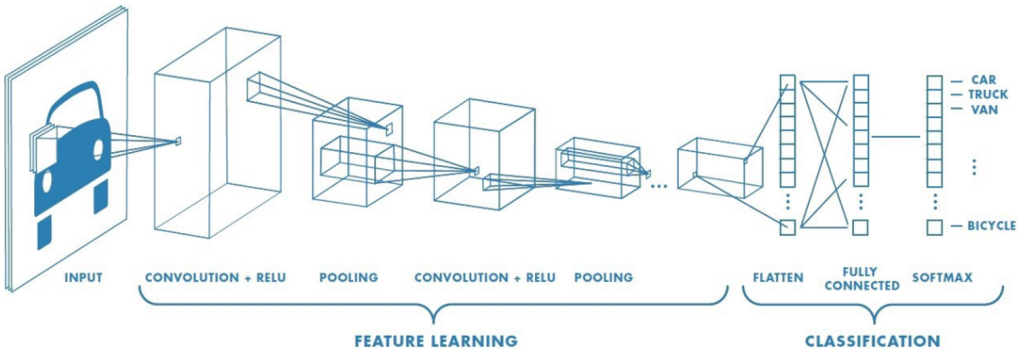


Figure 3.6: A schematic representation of a convolutional neural network (Saha, 2018).

3.6.2 Recurrent neural networks

Recurrent neural networks (RNN's) are used in sequence modelling. Unlike basic ANN's and CNN's, in RNN's, the output is not only dependent on the current input, but also on previous inputs. Most RNN's can process variable input lengths. As previous inputs can be taken into account, RNN's are suitable for processing data sets with temporal dependencies.

When feeding a sequence to a RNN, every element of the sequence is fed to the system separately. This is visualized in Figure 3.7. The input vectors **X** are the different sequence elements. At every timestep t, a hidden state (h) is calculated, and an output (y) is generated.

Hidden state t depends on hidden state t-1, and sequence element t. A weighted sum is calculated, where the weight matrices (U and W) are optimized during training.

$$\mathbf{h_t} = \sigma(W * \mathbf{h_{t-1}} + U * \mathbf{x_t} + b) \tag{3.7}$$

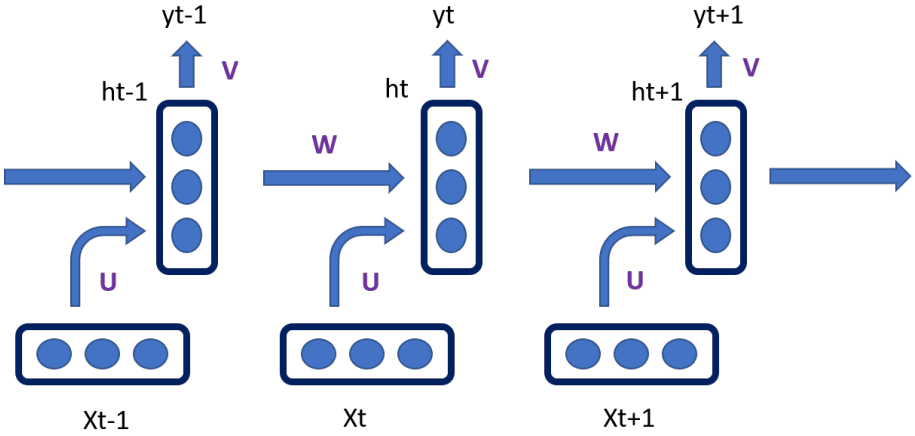


Figure 3.7: A schematic representation of a recurrent neural network. Input vectors x are fed to the system and used to calculate the hidden states (h). Based on those hidden states, outputvectors (y) are generated. Weight matrices U,V and W, used to calculate the hidden states and outputs, are drawn bold and purple. These vectors are optimized during training.

b is the bias and σ represents the activation function that is added to create non-linearity. The output vector y is calculated as follows for every timestep t:

$$y_t = O(V * \mathbf{h_t} + \mathbf{c}) \tag{3.8}$$

Where V is a weight matrix that is optimized during training. For every time step, the same weights (U,W,V) are used to update the hidden state. c is a constant, and O is an output function, for example a softmax layer.

RNN's can be used to predict a class for every element in the sequence. In that case, all output vectors are of interest. Sometimes, for example in spam detection, only one output class needs to be predicted for the whole sequence. Then, only the output vector of the last seuence element is taken into account, as this output can theoretically be influenced by all inputs.

This type of RNN's only use the information preceding the current time step. This is interesting when one might want to predict the next word in a sentence. When the goal is to predict the class of every word in a sentence, it might be interesting to use information of following time steps too. For this reason, **bidirectional RNN's** are used. When using this type of architecture, for every time step, two hidden states are calculated: one gathering information of preceding time steps and one gathering information of following time steps. This information is combined to calculate the output vector.

$$\overleftarrow{\mathbf{h_t}} = \sigma(\overleftarrow{W} * \overleftarrow{\mathbf{h}}_{t-1} + \overleftarrow{U} * \mathbf{x_t} + \overleftarrow{\mathbf{b}}) \tag{3.9}$$

$$\overrightarrow{\mathbf{h_t}} = \sigma(\overrightarrow{W} * \overrightarrow{\mathbf{h}}_{t-1} + \overrightarrow{U} * \mathbf{x_t} + \overrightarrow{\mathbf{b}}) \tag{3.10}$$

$$y_t = O(V[\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] + \mathbf{c}) \tag{3.11}$$

For classifying problems, cross-entropy loss is often used. The total error is the sum of the errors made at the different time steps. For an error made at a late time step, inputs and parameters far back in time might influence that error. When performing backpropagation to calculate the gradient, we have to propagate far back into time. When performing the chain rule, to proceed to previous states, we have to pass by all hidden states in between, and thus multiply with weight mattrix W everytime. When multiplying a value often with the weight matrix W, the value will decrease fastly, when W is small. This problem makes it very difficult to capture long term dependencies (Hochreiter, 1998). When W is larger than one, the value will explode, which needs to be avoided too. This means that states located far from the current state, won't influence the output much. RNN's suffer from short-term memory (Manning, 2019).

Often, one wants to track long term dependencies. So solutions for the short-term memory problem are created. Two adaptations to the model are often used to solve the problem: Long short term memories (LSTM's) and gated recurrent units (GRU's). These systems have gates that control the dataflow, making it easier for the gradient to reach states far back in time. This way the network can keep track of some information and forget about other. It decides which data is important and which data is not. Such networks are often used, for example in speech recognition (Nguyen, 2018).

3.6.3 Encoder-Decoder architecture

The encoder-decoder architecture is a network design pattern that is often used in NLP, especially in translating (Genthial et al., 2019). This architecture is the core of google's translation service (Wu et al., 2016). It has two main building blocks, visualized in Figure 3.8. The first element, the encoder, encodes the input into a fixed size vector. The working mechanism of the encoder is based on RNN's. At each time step t, the hidden state of time step t-1 is combined with the input value at time step t to compute the hidden state at timestep t. The hidden state at the last time step contains encoded information about all previous time steps. This is a fixed size

context vector **c**. It can be seen as a low dimensional representation of the whole input space (Manning, 2019).

This encoded context vector **c** is passed to the second building block: the decoder, as can be seen on Figure 3.8. Conditioned on the context vector **c**, the decoder generates an output sequence (Cho et al., 2014). At every time step t, the decoder is fed the hidden state of time step t-1 and the generated output at time step t-1. The first hidden state is the context vector **c**, which was created by the encoder.

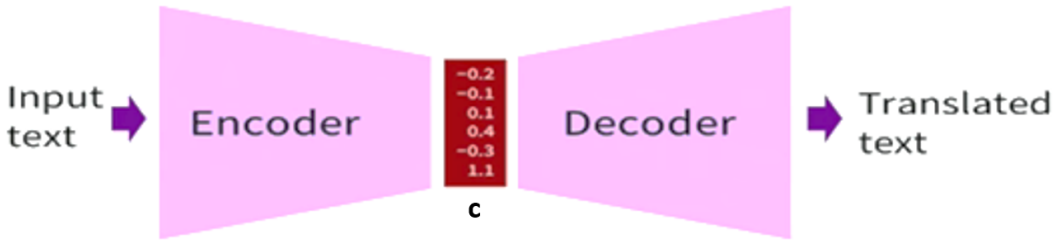


Figure 3.8: A schematic representation of an encoder-decoder architecture. The encoder encodes the input to a fixed size context vector **c**. The decoder uses this context vector to produce an output (Manning, 2019)

One might look at the context vector **c** as a vector that contains the grammar and information about the sentence construction. This information is used by the decoder to perform the translation. The whole model is optimized end-to-end by using back-propagation. Where the encoder is trained to extract the right information from the input sequence, the decoder is trained to capture the grammar and vocabulary of the output language. This results in a model that is more fluent, uses the context better and generalizes well.

When training an encoder-decoder model, the real output sequence is used to train the model. This way, mistakes can not stack. Of course, when testing the model, this can't be done, as normally the output sequence isn't known. In the test case, the previously predicted output value is used to predict the next one.

3.6.4 The attention mechanism

When performing a translation task using an encoder and decoder, all information about the input sequence is forced into one vector (the context vector **c**), causing a bottleneck for the information. Most probably, information on the beginning of the sentence is lost. Also, different parts of the input sequence are important for different parts of the output sequence, information that can't be learned using a simple encoder-decoder architecture. The attention mechanism provides a solution for this,

as at every step, the decoder can have a look at all encoder hidden states. It can decide at every decoder time step which hidden encoder states are relevant and which aren't. The relevant parts are used to predict the output. This way, more than only the context vector **c** of the encoder is used in the decoder (Genthial et al., 2019).

The output of the decoder at time step t will now be based on all hidden encoder states and the outputs. Based on the encoder hidden states, an attention output is calculated. The workflow for calculating this attention output is visualized in Figure 3.9. This attention output captures the relevant context for time step t from the original sentence.

To determine the relevant encoder hidden states, the dot product between the decoder hidden state of intereset ($\mathbf{s_t}$) and all encoder hidden states ($\mathbf{h_1}, ..., \mathbf{h_N}$) is calculated. This results in an attention score e^t for every encoder hidden state:

$$\mathbf{e^t} = [\mathbf{s}_t^T.\mathbf{h_1}, ..., \mathbf{s}_t^T.\mathbf{h_N}] \tag{3.12}$$

This attention scores will be higher for the hidden encoder states that are similar to the decoder hidden state, as they will have a higher value for the dot product. All equations are visualized in Figure 3.9. These attention scores are converted to fractions that sum to one using the softmax function:

$$\alpha^t = softmax(\mathbf{e^t}) \tag{3.13}$$

The values of α^t provide the attention distribution and show to which encoder hidden states is given the most attention. This is a representation of which encoder hidden states are the most relevant for the decoder hidden state t. In the example in Figure 3.9, most attention is given to the first encoder hidden state. As it is the first output element we are trying to predict, the first input element is probably relevant for this prediction. These elements of the attention distribution are used as weights to calculate a weigthed sum over the different encoder hidden states. The outcome of this weighted sum is called the attention output $\boldsymbol{\alpha_t}$:

$$\boldsymbol{\alpha_t} = \sum_{i=1}^{N} \alpha_i^t * \mathbf{h_i} \tag{3.14}$$

This attention output is then used to predict the output, often in combination with the decoder hidden states. This way, both information about the inputs as about the already generated outputs can be used to predict the next outputs.

Next to solving the the bottleneck problem and making it possible to focus on specific parts of the input in every decoder step, the attention mechanism also solves the short term memory problem, by creating shortcuts. By using attention, information can flow more directly. It doesn't have to pass so many hidden states anymore. Interpreting the attention step can give insights in the data (Manning, 2019). Attention can be thought of as a soft alignment. The words in the input sequence with a high attention score align with the current target word. Attention is better in describing long-term dependencies, making it the right fit for long sentences (Genthial et al., 2019).

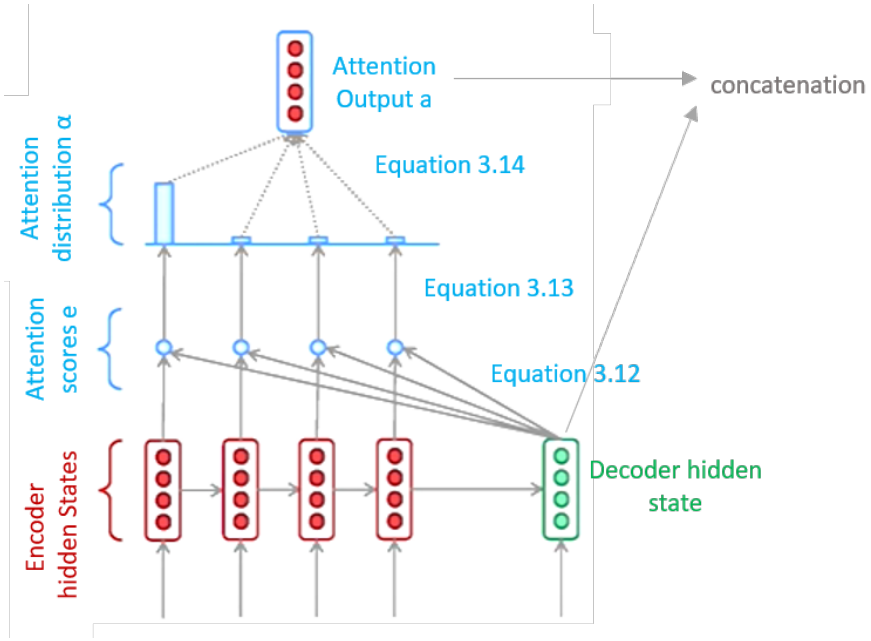


Figure 3.9: An overview of the attention mechanism. The dot product between all encoder hidden states and a specific decoder hidden state is taken. This results in a scalar for every encoder hidden state: the attention scores. These scores are put through a softmax function, providing the attention distribution. This distribution is used to calculate a weighted sum over the encoder hidden states, providing the attention output. This attention output is concatenated with the decoder hidden state. This Figure visualizes the attention mechanism for decoder hidden state at time step 1. For every time step, the decoder hidden state is different. This leads to a different attention distribution for every time step. Image adapted from Manning (2019)

The mechanism of attention can be generalized. To cite (Manning, 2019):" *Given a set of vector values, and a vector query, attention is a technique to compute a weighted sum of the values, dependent on the query*". In the previous example, the values were the encoder hidden states and the query was the decoder hidden state at the current time step.

The weighted sum can be seen as a selective summary of the information present in the values. The query determines on which values to focus. This way, a fixed size representation of the values can be created, depending on the query.

Different ways to calculate the attention scores exist. The dot product is the easiest one. Another option is to weigh the different values. This is called multiplicative attention.

3.7 Embeddings

For most machine learning models, the input of the model needs to be numerical. When dealing with NLP problems, this isn't evident. The input of a translation model is a sentence, and words aren't numerical. For the conversion of words into numerical vectors, multiple methods exist. These numerical vectors are called the embeddings of the words. Embeddings can be used to convert any type of symbolic representation into a numerical one.

A possible way to create embeddings is by using **one-hot encoding**. The vector representing the symbols has the same length as the total amount of possible different symbols. Each position in this vector corresponds to a specific symbol (Rong, 2014). For example, when converting colors to a numerical vector, the length of the vector would be the total amount of different colors present in the dataset. For each input, the location corresponding to the color of that value is one, where all other locations are valued zero. This method is very basic, but has its flaws. The more colors present in the dataset, the longer the embedding vector is. In translation, this becomes problematic, as the amount of possible words is huge. It will result in enormous models and the need for a lot of computational power when large one-hot encoding vectors are used. Next to that, no specific information is captured with one-hot encoding. From the numerical representation, it is not clear that orange and red are more similar than orange and green. For this reason, other methods exist too.

A second way of creating embeddings is by creating **feature vectors**. Every symbol has its specific vector representation, based on features. With colors, a vector of three elements could be used, where the elements represent the amount of yellow/red/blue needed to create the color. This way, all colors can be represented by only using a vector of three elements. Also, colors that are similar, have similar representation vectors. With words, these vectors typically have a size between 50 and 300. Sometimes, it is difficult to define those features. Often, they are chosen by hand, which is labor-intensive.

Embeddings based on context can be trained. The reasoning behind this concept is that words with similar meaning occur in similar contexts (Firth, 1957).There are different methods that take the context of words into account. Some methods, like GloVe (Pennington et al., 2014), base their context embedding on co-occurence

statistics from corpuses (texts). Words with similar co-occurence statistics will have similar word embeddings. Other methods use neural networks to train the embeddings. They, for example, train their embeddings to predict the word based on the context (common bag of words), and/or to predict the context based on the word (Skip gram). This is how word2vec works (Rong, 2014). For text, these models are trained on huge databases, for example, wikipedia pages. Training these contextual embeddings is time-intensive. For this reason, pre-trained libraries exist, so everyone can use those contextual embeddings.

Other deep learning can be used to create embeddings too. One possibility is to use the latent space of a variational autoencoder (VAE) as the embedding of the input. Another way is using 1D convolutions to create embeddings. This way, a sparse high dimensional input space can be converted to a denser, low dimensional feature space.

3.8 The Transformer

In 2017, Vaswani et al. (2017) introduced a new model to perform machine translation tasks: the Transformer. This model, which is solely based on attention, caused a small earthquake in the NLP research field. The Transformer outperforms previous NLP models substantially on translation tasks .

Transformer models are based on the principle of **self-attention**. Self-attention allows each element of the input sequence to look at all other elements in this input sequence and search for clues that can help it to create a more meaningful encoding. It is a way to look at which other sequence elements are relevant for the element that is currently processed. The Transformer can grab context from both before and after the currently processed element.

3.8.1 Self-attention: Into detail

When performing self-attention, three vectors need to be created for each element of the encoder input: the query vector, the key vector, and the value vector. These vectors are created by performing matrix multiplications between the input embedding vector using three unique weight matrices. A visualization of the whole process of self-attention is given by Figure 3.10.

After this, self-attention scores are calculated. When calculating self-attention scores for a given element, the dot products between the query vector of this element and the key vectors of all other input elements are calculated. To make the model math-

ematically more stable, these self-attention scores are divided by the root of the size of the vectors. Just as before, these scores are normalized with a softmax layer.

This attention distribution is then used to calculate a weighted sum of the value vectors, resulting in a vector **z** for every input element. Where in the attention principle explained before, the vector to calculate attention scores and to perform the weighted sum was the same, in self-attention two different vectors are created and used.

As the self-attention needs to be calculated for all elements (thus a query for every element), one formula can be created to calculate a Z matrix. The rows of this Z matrix are the **z** vectors for every sequence input element, giving the matrix a size length sequence* dimension QKV. How these z-vectors are calculated can be seen in Figure 3.10.

In the Transformer, multi-headed attention is executed. For every attention head, different weight matrices to calculate Q, K, and V are trained. Every attention head outputs a matrix Z. Different attention heads can capture different types of information. The different Z matrices of the different attention heads are concatenated. This matrix becomes big when multiple attention heads are used. To reduce dimensionality, an extra weight matrix W^0 is trained to condense the different attention heads into a matrix with the same size as one Z matrix. This way, the amount of data given to a next step in does not enlarge every time self-attention is performed.

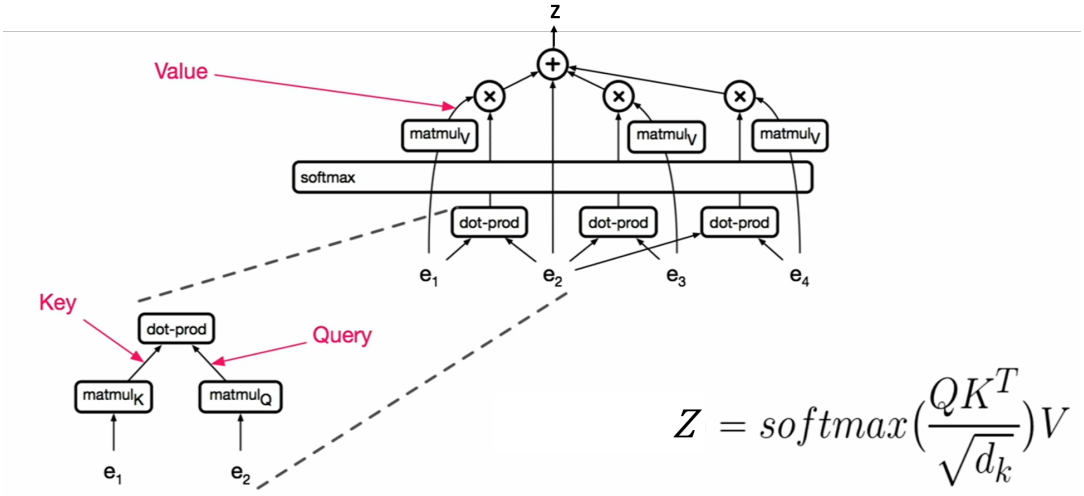


Figure 3.10: Schematic representation of the calculation of self-attention for e_2. The e_n-symbols represent the embedding vectors for all input elements. Matmul means matrix multiplication. Matrix multiplications are performed to calculate the Query, Key, and Value vectors. The dot product between K and Q vectors is calculated. These values are put through a softmax layer and used as weights for the corresponding value vectors. A weighted sum of all value vectors is calculated. This is the vector Z. Image adapted from Manning (2019).

3.8.2 The Transformer architecture

When performing self-attention, information about the order of the different elements within the sequence is lost. To address this problem, positional encodings are added to the embedding vectors. Every position has its unique positional encoding vector. These vectors follow a specific pattern, which the Transformer model can learn to recognize. This way, the model can take into account distances between the different elements.

Transformer models often have the encoder-decoder architecture, although this isn't necessarily so. The encoder is built out of different encoder layers which are all constructed in the same way. Such a layer is visualized in Figure 3.11. The positional encodings are added to the embedding vectors. Afterwards, self-attention is performed. Every self-attention layer is surrounded by a residual connection, summing up the output and input of the self-attention. This sum is normalized, and the normalized vectors are fed to a feed-forward sublayer. Every z-vector is fed separately to this feed-forward layer. The feed-forward layer is wrapped in a residual connection and the outcome is normalized too. Often, numerous encoder layers are piled to form the encoder. The output of the encoder is a fixed size vector for every element of the input sequence.

Just as the encoder, the decoder is built from of different decoder layers. In the decoder, a modified version of self-attention takes place. The query vector is only compared to the keys of previous output sequence elements. The elements further in the sequence aren't known yet, as they still have to be predicted. No information about these output elements may be used.

The whole Transformer model, including the decoder layers, is visualized in Figure 3.12. Next to a self-attention sublayer, a sublayer of encoder-decoder attention is present in the decoder, in which the decoder can examine the last z-vectors of the encoder, providing a fluent information transmission. The ultimate decoder sublayer is a feed-forward layer. All sublayers are packed in a residual connection. This built allows the decoder to examine all previously predicted outputs and all encoded input vectors to predict the next output. This way, information of the encoder is provided to the decoder, which could improve the predictive capacity.

The output vectors of the last decoder layer need to be processed to form the output. This is done by a combination of a feed-forward layer and a softmax function. The output corresponding to the highest probability will be the predicted output value for this time step.

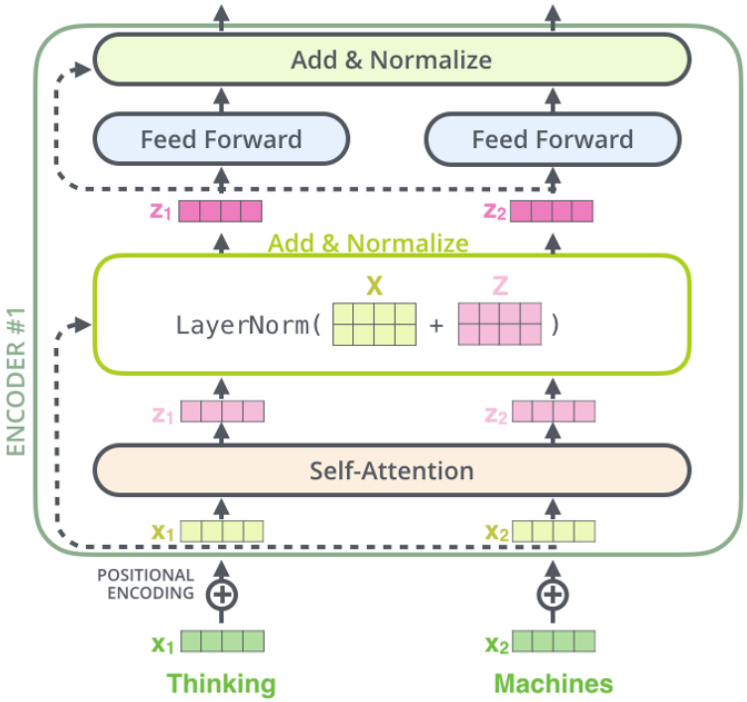


Figure 3.11: One encoder layer of a Transformer network. This Figure shows the different steps for an input sequence with two elements. x_1 and x_2 are the embedding vectors of these elements. The positional encodings are added to the embedding vectors. Afterwards, self-attention is performed, followed by an add and normalization step. The input and the output of the self-attention sublayer are added and normalized, resulting in a new z-vector. This new z-vector is fed to the feed-forward sublayer. Again, a residual connection and normalization sublayer are present. The outcome of this layer will then be the input for the next layer. Image from Alammar (2018).

When tasks other than translation need to happen, sometimes, only an encoder is used. This is, for example, the case for document classification (Adhikari et al., 2019) or name entity recognition (Yan et al., 2019). In this case, the encoded input vectors are the input of the feed-forward and softmax layer.

Transformer models have been extensively applied in different NLP fields, such as translation (Vaswani et al., 2017), document summarization (Zhang et al., 2019), speech recognition (Dong et al., 2018) and named entity recognition (Yan et al., 2019). These models have applications in the biological field too: predicting protein structure and function (Rives et al., 2019) or labeling DNA sequences (Clauwaert and Waegeman, 2019).

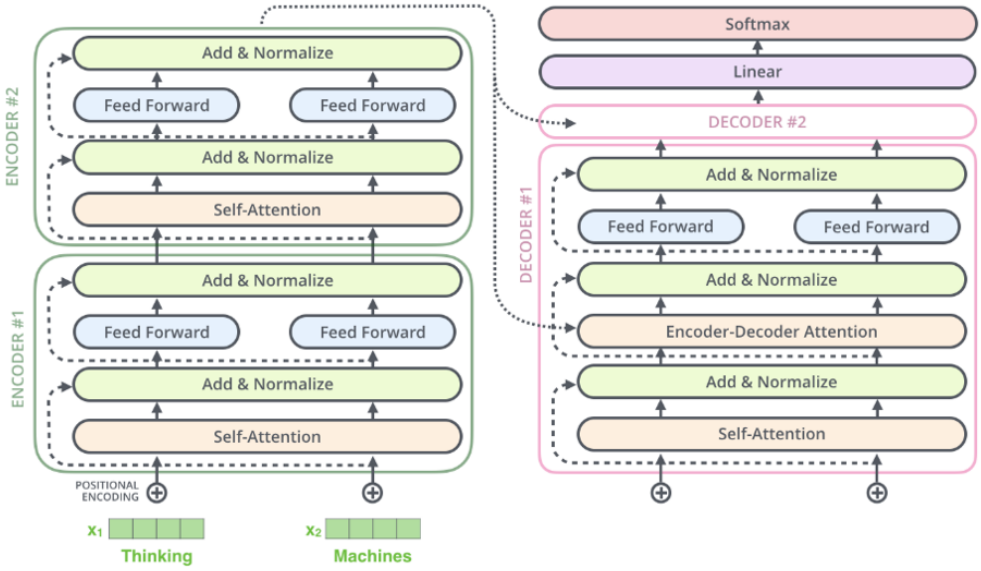


Figure 3.12: A schematic overview of a Transformer model. Different encoder and decoder layers are stacked. A decoder layer consists out of three sublayers: a self-attention sublayer, an encoder-decoder attention sublayer, and a feed-forward sublayer, all individually wrapped up in a residual connection. The output of the last decoder layer is put through a linear layer. To obtain output probabilities, a softmax layer is used at the end. Image from Alammar (2018).

3.9 Machine learning for protein secondary structure prediction

In this master dissertation, machine learning is used to predict secondary protein structures. This focus is chosen to make the problem feasible: more feasible than predicting 3D structures for proteins. In Section 1.5, the added value of protein secondary structure prediction is explained. Seen from a machine learning point of view, protein secondary structure prediction is a compelling problem.

In the case of secondary structure prediction, the used training data is labeled: for all aa present in the used proteins, we know whether they are part of an α-helix, a β-sheet or a coil region. This makes the problem a supervised problem. Unsupervised methods can still be used to explore the data. Secondary protein structure prediction is a multiclass classification problem. Exactly one class needs to be assigned to every element in the sequence, but there are multiple classes possible: three or eight, depending on which type of protein secondary structure is predicted.

Protein secondary structure prediction can be seen as a seq2seq problem, where the input is the protein sequence and the output is the secondary structure. In this setting, the input length is variable, as all proteins have different lengths. The length

of the output sequence is as long as the input sequence and equals the length of the protein.

In protein secondary structure prediction, the neighborhood of the aa is very important. Especially to predict β-sheets, a broad context is desired because these structures are formed between aa that are not necessarily close to each other in the sequence (Rost, 2001). For this reason, contexts spanning the whole protein could benefit the predictive power.

When predicting secondary protein structures de novo, only sequences with less than 30 % sequence identity should be used. If not, the model could base its predictions on homology (homology modelling), and this would bias the results.

3.9.1 Performances of secondary structure prediction models

Model performance for protein secondary prediction is measured in Q3 or Q8, depending on how detailed the predicted information is. Q3 and Q8 represent the percentage of residues that is correctly predicted (accuracy) and these measures are chosen because it is very easy to interpret from a biological perspective. When the three-state secondary structure information is predicted, Q3 is used. When eight states are taken into account, the Q8 score is used. In the last 25 years, the accuracy of secondary structure prediction models raised from 69,7 % (Q3 scores) using a simple ANN and sequence alignment (Rost and Sander, 1993) to 84 % using deep convolutional neural fields (Wang et al., 2016). Similar results are obtained in 2018 using BRNN's with LSTM (Torrisi et al., 2018). As stated before in Chapter 1, the theoretical limit of this accuracy is 88-90 %, because the secondary structures are defined ambiguously, especially on the borders. This possibly affects the predictions, as prediction algorithms report more mistakes at the boundaries of the structures (Wang et al., 2016). Also, the data tends to be noisy, as the same secondary structure predictor isn't always used.

3.9.2 Embeddings for protein secondary structure prediction

In the most recent models, different ways of converting the protein sequence to numerical values are used. Often, the models try to capture as much information as possible in their input features. Different techniques are used, and often multiple embedding types are combined.

One-hot encoding is often used. In this case, a vector of 20 elements is created for every position: one for every distinct aa. This results in sparse data. Some use

processes of NLP to convert these sparse vectors into dense representations that are learned during training (Zhang et al., 2018a), for example 1D convolutions. Heffernan et al. (2018) solely uses this type of embedding. The Q3 accuracy that has been reached in this paper, however, is only 72 %. Other methods, using more elaborate embeddings, can report considerably better accuracies.

Usually, feature vectors are used, because they can contain more information, for example, about the similarity of two aa. In recent years, virtually all algorithms make use of evolutionary data by constructing position-specific scoring matrices (PSSM's). This PSSM is a matrix with twenty columns and as many rows as the length of the sequence. The twenty rows represent the twenty different aa. To create PSSM's, programs like PSI-blast align the sequence of interest with (almost) all known protein sequences, creating multiple sequence alignments. PSI-blast works iteratively. After every alignement, the newly-added sequences are used to search for new alignments until no new alignments can be found (Bhagwat and Aravind, 2007). The structures of the aligned sequences aren't necessarily known. Over all aligned sequences, at all positions, the frequency of all unique aa is determined. These frequencies result in scores. The higher the frequency of a certain aa at a certain position, the higher score this aa at this position will get. This way, more information is used, as sequences for which the structure aren't determined yet are exploited too. This information is called evolutionary information, but not all aligned protein sequences are homologs. Similarity can be present due to convergent evolution too. PSI-blast doesn't distinguish between protein sequences that share common ancestors or protein sequences that just show sequence similarity.

When performing multiple sequence alignments with PSI-blast, one might need multiple hours to create the PSSM of a large protein (1000 aa) (Heffernan et al., 2018). For this reason, some methods make use of the HHBlits program (Remmert et al., 2012), which can generate multiple sequence alignments faster. This comes at the cost of fewer aligned sequences per protein.

Porter5 (Torrisi et al., 2018) uses both the PSI-blast and the HHblits alignment scores, as this improves the prediction. When the prediction needs to happen fast, only HHblits profiles can be used, only slightly decreasing the Q3 accuracy.

Also, the physical properties of every aa can be used to predict secondary structure. These properties include steric parameters (graph-shape index), polarizability, normalized van der Waals volume, hydrophobicity, isoelectric point, helix probability, and sheet probability (Zhang et al., 2018a). These last two features might bias the results, as those probabilities are calculated based on data that is used to predict the

protein secondary structures. This way, information about the output that needs to be predicted is given as input.

Most of the time, combinations of embeddings are used: one-hot encoding in combination with PSSM's (Gao et al., 2018; Wang et al., 2016) or physiochemical properties in combination with PSSM's (Heffernan et al., 2017). One method uses the PSSM's in combination with a generated dense representation started from the one-hot encoded vectors, the physiochemical properties and some extra scores (Zhang et al., 2018a). A conclusion might be that different kinds of information can be used. Guo et al. (2019) proves that the use of both the sequence features, and the PSSM matrix as input features improves the predictive power of the model.

3.9.3 Model architectures for protein secondary structure prediction

The models in literature follow a quite general pattern. Multiple models start with 1D convolutions to extract meaningful features and to capture local dependencies (Zhang et al., 2018a; Guo et al., 2019; Wang et al., 2016). Some use small 1D convolutions, whereas other models only use numerous 1D convolutions to predict secondary structure (Gao et al., 2018). To capture long-term dependencies, most models use bidirectional recurrent neural nets (BRNN's). This can be plain BRNN's (Torrisi et al., 2018), but often BRNN's with LSTM (Guo et al., 2019; Heffernan et al., 2017, 2018) or GRU (Zhang et al., 2018a) are used to capture more accurately long-term dependencies.

Often, some special features are added, such as residual connections (Gao et al., 2018), asymmetric convolutions (Guo et al., 2019), conditional random fields (Wang et al., 2016) or train multiple models that are combined at the end (Torrisi et al., 2018). To conclude, most models in literature use the same model architecture as base. Most models, however, adapt them by adding novelties.

CHAPTER 4

DATA EXPLORATION

In the figures of this Chapter, the different aa are represented by their one-letter abbreviation. The corresponding names can be found in the abbreviations list.

4.1 Data Collection

4.1.1 Data from the protein data bank

For this master dissertation, two datasets are created to train and test the predictive models. A first dataset is created by querying the pdb, the worldwide repository of information about biological 3D-structures. Although mainly containing information about protein structures, 3D-structures of nucleic acids and complex assemblies can be found, too. The pdb only contains experimentally-determined structures, models aren't allowed. When a new 3D structure of a protein is determined, this structure needs to be uploaded in the pdb before publication is possible, assuring the completeness of this database. The pdb is a curated data bank. For the proteins present in the data bank, the secondary structure is determined based on backbone angles and interactions. Mostly, the Kabsch and Sander algorithm for defining the secondary structure of proteins (KSDSSP) is used to perform this task (Kabsch and Sander, 1983). This algorithm defines the structures based on patterns in hydrogen bonds and geometrical features. The criteria are relatively simple. Although strongly recommended, the use of this algorithm isn't required. This can lead to inconsistencies, as different algorithms might define secondary structures differently.

On the 13th of May 2020, the pdb contained 163,949 protein structures. Often, multiple structures of the same protein are submitted to the data bank. These can be the same proteins interacting with different ligands, or with a small mutation. When such different structures of the same protein would be used to train or test the models, it would bias the results. It could lead to artificially-high accuracies on those specific proteins, as very similar data is fed multiple times to the model. The training set and the test set will not be independent anymore.

To create the first train and test set in this master dissertation, not all entries of the pdb were used. A strong selection happened. To select the right proteins, PISCES was used. PISCES is a protein culling server and can provide lists of proteins based on sequence identity and structural quality criteria (Wang and Dunbrack Jr, 2003). To create the dataset, only proteins showing less than 30% sequence similarity (determined by PSI-BLAST) and a resolution better than 2.5 Angstrom were used. The data was retrieved on 23/02/2020. This resulted in 17,062 protein structures, characterised by their pdb-id's. Afterwards, all pdb-ids that didn't have a corresponding secondary structure were deleted. When the secondary structure was longer than the sequence, a secondary structure was impossibly long, or secondary structures were overlapping, the protein was discarded. Also, proteins with unknown aa or extremely rare aa were excluded. Information about acetylations and nitrifications was removed, too. Also, to limit memory use, sequences longer than 1000 aa were discarded before training. This resulted in a dataset of 9,573 proteins that was used to train and test the models.

4.1.2 Data containing evolutionary information

As most algorithms to predict protein secondary structures in literature use evolutionary information to perform secondary structure prediction, such a dataset is obtained. Creating a PSSM with PSI-Blast can take up to 45 minutes per protein. It was thus unfeasible in this master dissertation to create such a database from scratch. Torrisi et al. (2018) provides a dataset containing evolutionary information. To create this dataset, protein sequences were retrieved from the pdb. For all retrieved sequences, PSSM's were created. Only proteins showing less than 25% sequence similarity are used, and no more than 10 undetermined aa per protein could be present. Additionally, all proteins longer than 1000 aa are discarded, for the same reason as before. For this master dissertation, the PSI-BLAST profiles are used. To add sequence information to the PSSM, the value of the PSSM column corresponding to the original aa (present in the original protein) is artificially clipped to one. This doesn't lead to a loss of information, as in PSSM's, the sum over the columns is one. This makes it possible for the algorithm to capture the real sequence information, without adding extra dimensionality to the input features. Using a database from literature makes it easy to compare performances. On the other hand, by using a provided dataset, it is very difficult to retrieve information about the dataset that wasn't provided. Also, bias is harder to detect, because of a possible lack of knowledge on how the dataset is created.

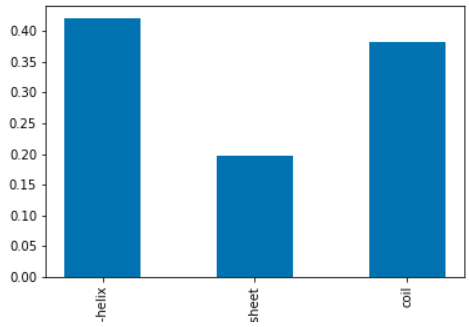
4.2 Data exploration on amino-acid level

There is a big overlap between the dataset containing PSSM's and the pdb data set which is solely based on sequence information, but they are weirdly enough not compatible. The same protein structures, characterised by the same protein identifier, often have a different length in the two databases. For other protein structures, small deviations in secondary structure or sequence can be detected. This wasn't expected, especially because both used the pdb dataset as a basis. This shows once more that the data is noisy, which most probably influences the results.

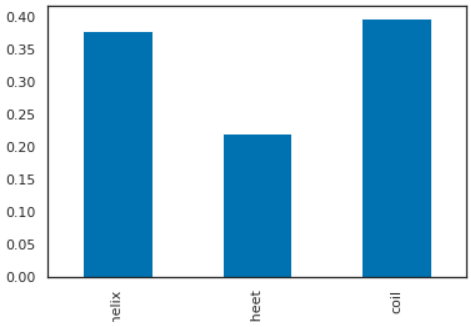
Both datasets defined above were examined. The data retrieved directly from the pdb contains 2,730,187 aa, and the training set retrieved from Torrisi et al. (2018) contains 3,797,426 aa. When examining the three-state secondary structures, as shown in Figure 4.1, an imbalance can be observed, where the β-sheet class is under-represented in both datasets. Around twice as many aa are present in a coil or α-helical structure than in a β-sheet. This imbalance in the output classes can and probably will influence the performance of the predictive models, and should be taken into account. When eight structures are considered, the imbalance is huge, as can be seen on Figure 4.1c. The eight structure information is only available for the evolutionary data. This huge imbalance asks for specialised models. Otherwise, underrepresented classes will be ignored. For this reason, no eight secondary structure predictions are performed in this master dissertation.

Next to output classes imbalance, imbalance in the input should be examined too. There are twenty frequently-occurring aa. In the evolutionary data, the non-frequent aa aren't filtered out, but are all represented by X. As can be seen in Figure 4.2, strong differences between aa can be observed. Some aa are more redundant than others. C, the abbreviation for cysteine, is the least common aa. A possible explanation for this could be that cysteine forms disulfide bonds. These bonds alter the protein structure a lot by increasing the rigidity and thus influencing the thermostability (Karshikoff et al., 2015). The plots for the two different datasets are similar, but small differences can be spotted. In the pdb dataset, protein glycine (G) occurs more often than protein valine(V), whereas this is the other way around for the evolutionary dataset. The imbalance has the same order of magnitude for both datasets.

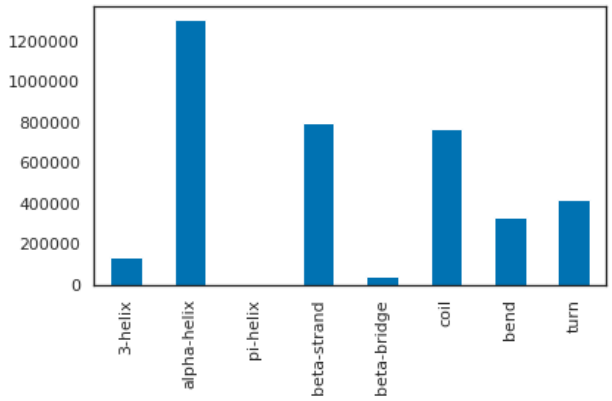
When combining both secondary structure and aa imbalance, even more striking differences can be observed, as shown in Figure 4.3. This plot is only shown for pdb data because the two datasets displayed very similar trends. Averaged over all proteins in the dataset, different aa appear with different frequencies in the different secondary structures. A first quick observation learns us that β-sheets are the least occurring



(a) class imbalance for pdb data

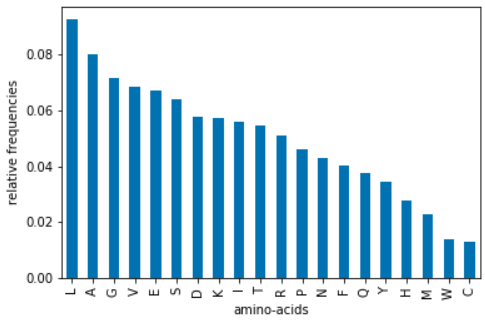


(b) class imbalance for evolutionary data

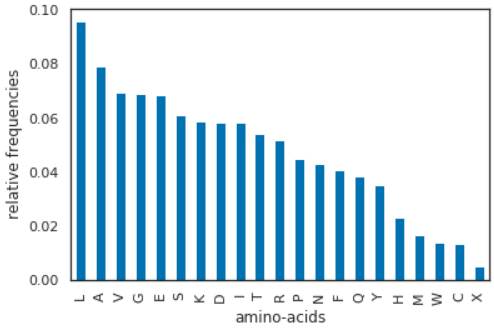


(c) eight classes imbalance for evolutionary data

Figure 4.1: a and b: For both datasets, these bar plots show the class imbalance for three-state secondary structures. For all aa in the datasets, the relative frequencies of appearance in the different secondary structures are shown. In both datasets, β-sheets are underrepresented. Aa occur (almost) twice as often in α-helices or coils than in β-sheets. c. Eight structure class imbalance for evolutionary dataset. The imbalance between different kinds of secondary structures is big. When creating predictive models, this imbalance should betaken into account.



(a) aa imbalance for pdb data



(b) aa imbalance for evolutionary data

Figure 4.2: For both datasets, the relative frequency of every aa is shown. The aa are ordered based on relative frequency. A clear imbalance can be observed. Some aa occur more than four times as often as other aa. This is another type of imbalance present in the dataset. Small differences between the two datasets can be observed.

secondary structure, as stated before. Some aa, like proline (P), mostly occur in coil regions. The reason for this is that proline disrupts helices and sheets, and can only be present at the borders of these structures. Alanine, for example, is often present in α-helices, as it has the propensity to stabilize helices (Rohl et al., 1999). When certain aa are frequently present in certain secondary structures, this influences the results. The algorithm will probably recognize this as a pattern and often classify an aa as the most frequent secondary structure for that aa. For example, the probability is high that when the wrong secondary structure is predicted for proline, this wrong class will be 'coil'.

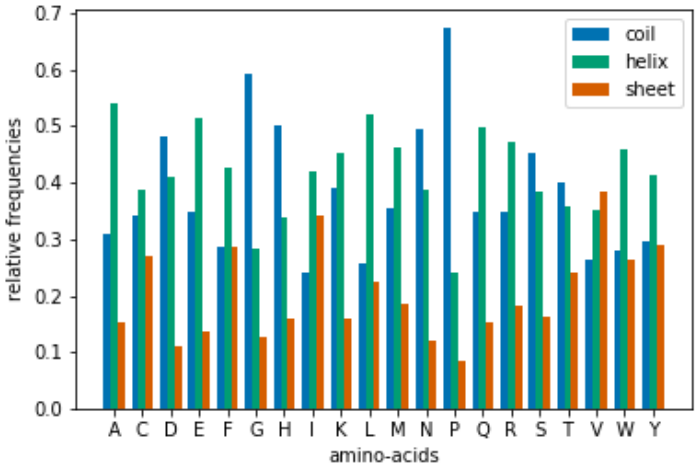
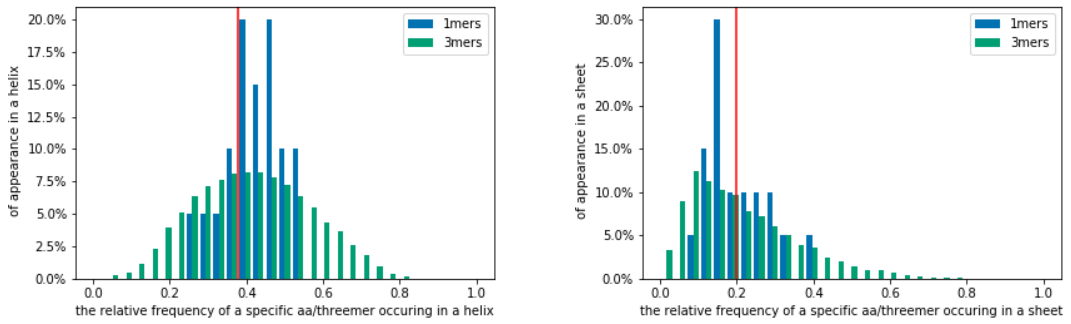


Figure 4.3: Plot for pdb data: For every aa, the relative frequency of occurring in the three possible protein secondary structures is shown. These frequencies were calculated using all proteins present in the dataset. Big differences can be observed. Proline(P) occurs in almost 70 % of the cases in a coil region of the protein, whereas Alanine (A) occurs in 50% of the cases in an α-helix.

Old methods for protein secondary structure prediction were based on relative frequencies. Only looking at the aa itself would result in inaccurate predictions. To improve performance, one should look at the neighborhood of the aa, and perform the secondary structure predictions based on this information. A simple implementation would be to look at one neighbor on each side. This way, threemers are formed, used to predict the secondary protein structure of the aa in the middle. All these unique threemers appear with different frequencies in the proteins and big differences are noticeable The least occurring threemer (CCW) is only present once in the whole pdb dataset. As cysteine (C) and tryptophan (W) are the least occurring aa, this isn't too surprising.

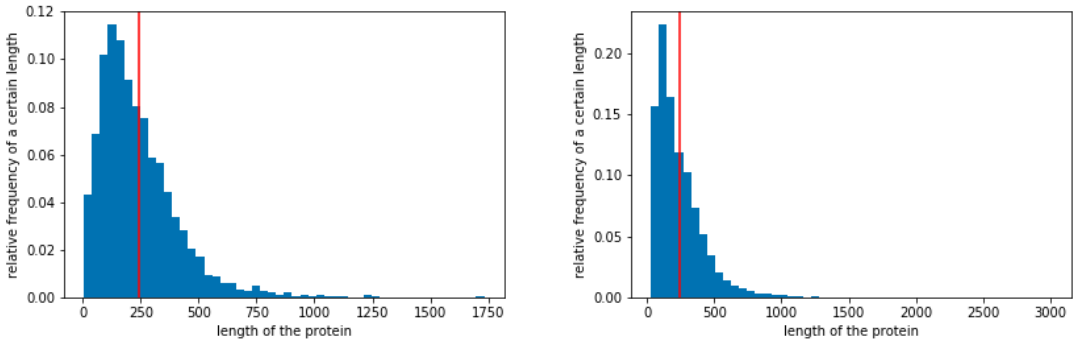
A visualisation of the occurrence distribution of all unique one- and threemers in secondary structures is made in Figure 4.4. When no information about an element in the sequence is given, there is a +- 40% chance that this element is located in an α-helix



(a) Helix (b) Sheet

Figure 4.4: Plots showing the occurrence distribution for aa and unique threemers in both helices and sheets for the pdb-data (both datasets showed similar results). The red line shows the overall chance of aa occurring in a helix/sheet. All unique onemers are looked at, and their relative frequency of occurring in a helix (left figure) or sheet (right figure) is calculated and plotted in blue. The x-axis represents this relative frequency, where the y-axis shows the percentage of unique aa that possesses this relative frequency. The same is done for threemers (where the left and right neighbor are taken into account). This is plotted in green.

and a +-20 % chance that it is located in a sheet. This is represented by the red line in the figures. More accurate estimations can happen when the aa is known. For each unique possible aa (20 in total), the relative frequency of it appearing in a helix/sheet is plotted in blue. For example, 15% of the aa appear in 42% of the cases in an α-helix. Differences between the different unique aa can be seen, and this information can be used. When the left and right neighbor of the element are known, even more information is available. For each unique possible threemer, the relative frequency of appearing in a helix/sheet is plotted in green. Certain threemers that are only present a couple of times can distort the results, as possibly, their frequency of occurrence isn't representative because not enough data was present. Only threemers occurring more than 100 times in the dataset have been included to create Figure 4.4. In practice, however, the differences between using all threemers and only using frequent threemers are negligible. It is clear that the green distributions are broader than the blue ones. Some threemers clearly appear often a specific secondary structure where others occur very rarely in a specific secondary structure. This means more information about protein secondary structure is present when using threemers. This indicates that looking at aa neighborhoods can help to predict the secondary structure of an aa. Expected patterns can be observed: the threemers with the lowest occurence in α-helices often contain proline(P) as central aa, and it has been shown that this aa breaks secondary structures. Threemers with a high relative occurence frequency in helices often contain the aa Leucine (L), methionine (M), and Glutamic acid (E).



(a) distribution on pdb data

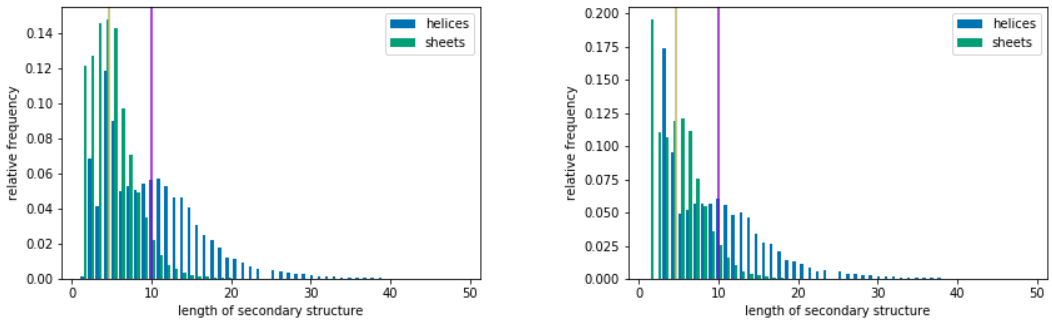
(b) distribution on evolutionary data

Figure 4.5: Plots showing the protein length distribution for both datasets. For both datasets, most proteins have lengths between 30 and 350 aa, where the mean length is shown by a vertical red line. The evolutionary dataset has outliers up to lengths of 3000 aa, whereas the longest protein in the pdb dataset is 1750 aa. Proteins longer than 1000 aa are rare.

4.3 Data exploration on protein level

The used datasets contain proteins and no single aa. Some properties of these proteins need to be examined to get a bit more insight into the data we are working with. First of all, the lengths of the proteins are explored, as shown in Figure 4.5. Different proteins have different lengths, but most proteins are built with 30 to 350 aa, with an average of 241 aa (evolutionary data) and 240 aa (pdb data). Proteins longer than 1000 aa are rare. To make the model computationally more efficient, all proteins longer than 1000 aa are left out. For the evolutionary dataset, this means losing 0.7% of the proteins, and for the pdb dataset 0.5%. This is worth the gain in computational speed and memory usage.

The number of secondary structures per protein is interesting, too. Do proteins contain a few long structures or a lot of short ones? First of all, the lengths of all secondary structures in all proteins are determined, as can be seen in Figure 4.6. Especially α-helices can be very long (up to 160 aa). Including those values in the plot made the plot unclear. For this reason, all structures longer than 50 aa aren't plotted (but they were used to calculate the mean length). On average, α-helices are about twice as long as β-strands. From the information present in the databases, it is impossible to determine which β-strands combine to form a β-sheet. The lengths of helices are spread between three and twenty, whereas the range for β-strands is smaller. The distributions are alike in both datasets. One clear difference can be detected. In the pdb-dataset, helices of a length lower than three are present. By definition, this is impossible. The shortest type of helix, the pi-helix, contains at least three aa. This means the data still contains mistakes. β-strands of 1 aa occur. This doesn't imply



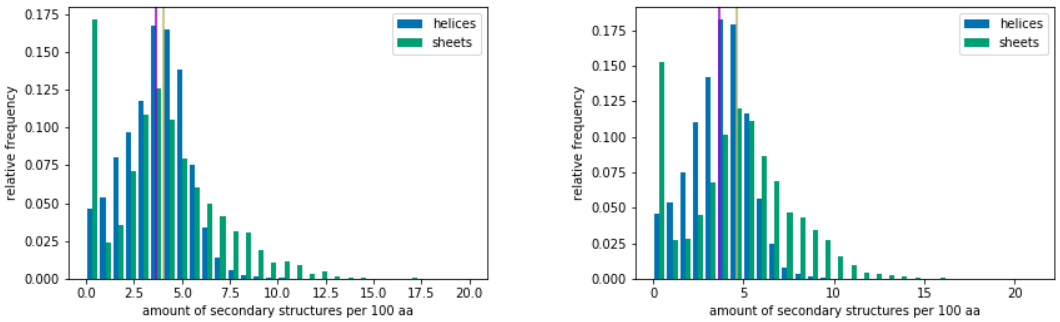
(a) distribution on pdb data

(b) distribution on evolutionary data

Figure 4.6: Plots showing the secondary structure length distribution for both datasets. In green, the length distribution for β-strands (forming β-sheets) is plotted, and the mean is indicated with a vertical khaki line. The length distribution for α-helices is showed in blue, and the mean is indicated with a vertical violet line. In both datasets, α-helices are substantially longer than β-strands (on average twice as long). Short helices occur regularly, but long β-strands are exceptional.

those sheets only contain one aa, because that would be impossible too. These 1 aa β-strands interact with other β-strand pieces located in other regions of the protein. β-strands are more spread out over the whole protein sequence, and interact with other β-strands that aren't necessarily located close to each other.

After looking at the length of secondary structures, the occurrence of these secondary structures in proteins is examined. To be able to compare proteins with different lengths, the occurrence of secondary structures is measured per 100 aa. The length of the structures isn't taken into account this time, only the amount of structures. The graphs for both datasets (as can be seen in Figure 4.7) show similar results. On average, there are slightly more β-strands than α-helices in proteins. The difference, however, is quite small. In total, (almost) twice as many aa are present in α-helices than in β-strands (as shown in Figure 4.1). When combining information from both Figure 4.6 and 4.7, the origin of this discrepancy is clear. α-helices and β-strands occur almost as regularly, but α-helices are on average about twice as long. It is a plausible hypothesis that longer structures are easier to predict, as there is less diversity in structural characteristics in that region, and a higher chance of detecting it. Another interesting observation is the difference in the distribution shape between the helices and the strands. The helix distribution only shows one peak, around the mean value. The sheet distribution, on the other hand, shows an extra peak around zero. This means quite a few proteins contain (almost) no β-sheets. Also, the sheet distribution has a bigger right tail than the helix distribution. Some proteins thus contain a lot of β-strands. Possibly, because they are smaller, and multiple β-strands are needed to form a β-sheet.



(a) distribution on pdb data

(b) distribution on evolutionay data

Figure 4.7: Plots showing the distribution of the number of secondary structures per 100 aa for both datasets. In green, the amount of β-sheets/100 aa is plotted, and the mean is indicated with a vertical khaki line. The number of α-helices/100 aa is indicated in blue, and the mean with a vertical violet line. There are slightly more β-sheets than helices in both datasets. Proteins with lots of β-sheets (>6 per 100 aa) occur regularly. Lots of helices in one protein is rarer.

4.4 Data exploration of the evolutionary information

The second dataset contains, next to information about the protein and the secondary structure, also PSSM's. The information captured in these PSSM's should be researched, too. The PSSM contains 22 columns, one for every common aa (20), one for all rare aa (X), and one for alignments with gaps (Z). For all aa, over all proteins, these columns of the PSSM indicate which alterations happen between the original sequence and the aligned sequences and how often they occur. This is visualised in Figure 4.8. In the columns of the heatmap, the aa in the protein, to which the alignments are created, are represented. The rows indicate the possible alterations: the values in the PSSM. A light color indicates this alteration happens often, a dark purple color indicates this alteration is very rare. The plot is created by averaging over all aa in all proteins present in the dataset. For example, a bright square can be noticed in column isoleucin (I) and row leucine(L). When isoleucine is present in the original sequence, to which the alignments are made, the aligned sequences often contain a leucine at this position. Although not necessarily so, the aligned (and thus similar) sequences are expected to have similar structure and similar function. The molecular structures of leucine and isoleucine are similar, as they are isomers. Isoleucine is branched, needing more space in the protein. In this space, Leucine can fit. The physiochemical properties of these two aa are alike, making them relatively well interchangeable. Probably, altering one into the other doesn't change the protein structure too much, leading in both cases to functioning proteins. Also, the codons coding for both aa are similar. This means a small mutation can lead to this

alteration. Possibly, during evolution, such mutations happened. Often, no selection against these mutations happened, as both of these proteins will function. This could explain why some alterations occur more often than others. The alteration leucine to isoleucine is rarer, probably because isoleucine needs more space.

Other frequent alterations are valine to isoleucine and leucine. Valine shares very similar physiochemical properties and codons with other two aa. Also, Glutamic acid (E) and Aspartic Acid (D) show frequent alterations with each other, for the same reasons. Little to none aa alter to rare/unknown aa, possibly because they are rare, and not often present in the aligned sequences. Loads of aa alter to leucine, possibly because leucine is the most common aa. The Z-column is completely dark-purple, because gaps are never a part of the original sequence, and thus never alter to something else. The order of the aa in this plot is based on their physiochemical properties. In general, values close to the diagonal are brighter. This means alterations happen more often between aa with similar physiochemical properties.

In Figure 4.9a, the frequency of alteration is plotted for all unique aa present in the original sequences. If this value is low, it indicates that in aligned sequences, this aa isn't altered often; the aa is conserved. For all aa, the alteration rates are relatively high. This is likely due to the huge amount of alignments created. For every protein, on average 14,000 alignments were created, and most probably, not all of these are very similar to the original sequence. This also explains the high values in Figure 4.9b, indicating the amount of different aa the original aa is altering to. If that many sequences are aligned to the original sequence, there is a relatively high probability that at least one of these aligned sequences contains a specific aa at a specific location. With PSI-blast, alignments are performed on previously-aligned sequences too, which can lead to aligning sequences that aren't similar anymore to the original one. The fact that a public database is used makes it impossible to check on this. Figure 4.9a clearly has one outlier: the rare/unknown aa. As alignments are made over different organisms, possibly those other organisms cannot use that specific rare aa. The aa with a relatively high conservation rate (proline, glycine, tryptophan) are aa with very specific shapes. Glycine is extremely small, tryptophan is big, and proline had a very specific side chain. None of them have other aa that are physiochemically similar, possibly explaining why they are more conserved.

Protein structure is more conserved than protein sequence (Rost and Sander, 1994). One might wonder if aa present in secondary structures are more conserved than aa present in unstructured regions. The adaptation rate is the highest when the original aa is present in an α-helix. The differences, however, are small: the alteration rate is +-69 % for aa located in α-helices and +- 66 % for aa located in β-sheets and coils. As different aa have different alteration rates and occur unequally in the

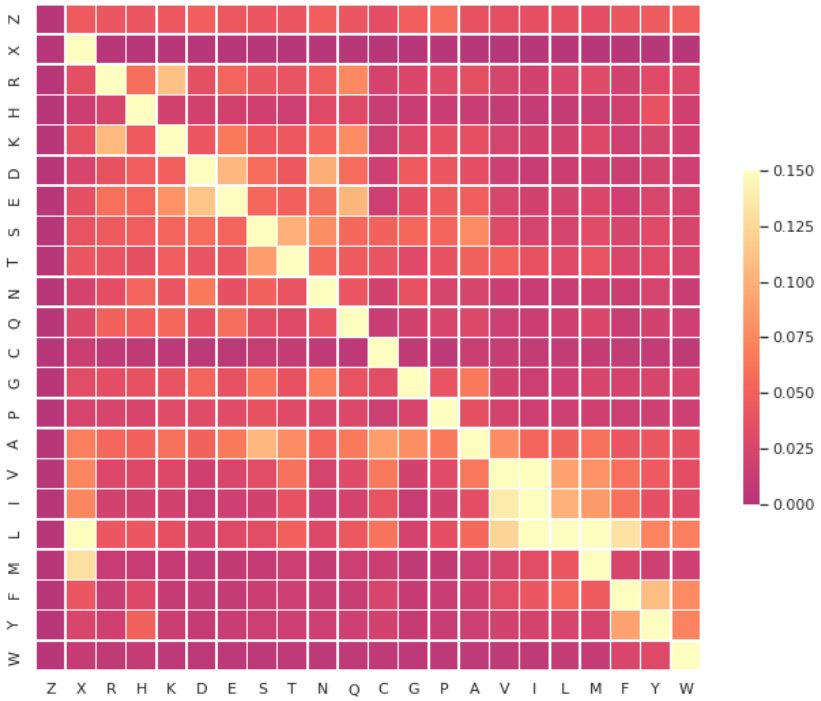
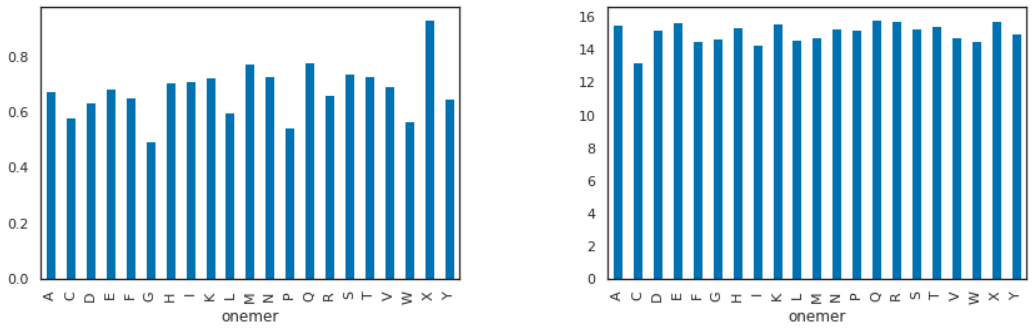


Figure 4.8: Heatmap showing the kind and frequency of alterations captured in the PSSM's. The columns represent the aa that were present in the original protein, to which the multiple alignments are created. The rows represent the columns of the PSSM's. High values are indicated in bright colors. The diagonal is completely bright. These diagonal values don't indicate alterations, but how often the original aa is preserved in the aligned sequences. On average, this is always the case in >15% of the instances. A bright color means the aa represented in the column often alters to the aa present in the rows in similar (aligned) sequences.

different secondary structures, this might be a confounding factor. For this reason, the alteration rate is calculated per original aa and per secondary structure, as can be seen on Figure 4.10. As previously shown in Figure 4.9a, differences between aa are noticeable. However, the differences between the secondary structures are small and no clear trends can be spotted. For some aa, the alteration rate is the highest when the original aa is present in a sheet, where for others it is higher in a helix.

It was to be expected that unstructured (coil) regions show higher alterations rates. This clearly isn't the case. Multiple explanations can be thought of. When certain regions are more conserved, it can be easier to find similar sequences. When more sequences are aligned, more variation can be present, especially because all aligned sequences are used to find more similar sequences. Also, there is no proof of secondary structure conservation over the different aligned sequences. The aa in the original sequence can be present in an α-helix, but that doesn't say anything about



(a) alteration rate per aa

(b) average amount of alterations per aa

Figure 4.9: a. For every unique aa in the original sequences, the relative frequency of its alteration is plotted, indicating the conservation of the different aa. The higher the bars, the less conserved the aa are. b. For every unique aa in the original sequence, the mean amount of alterations is calculated. High numbers indicate that often, this aa changed to numerous different aa in different aligned sequences.

the aa in the aligned sequences. Not enough information is available to make strong conclusions.

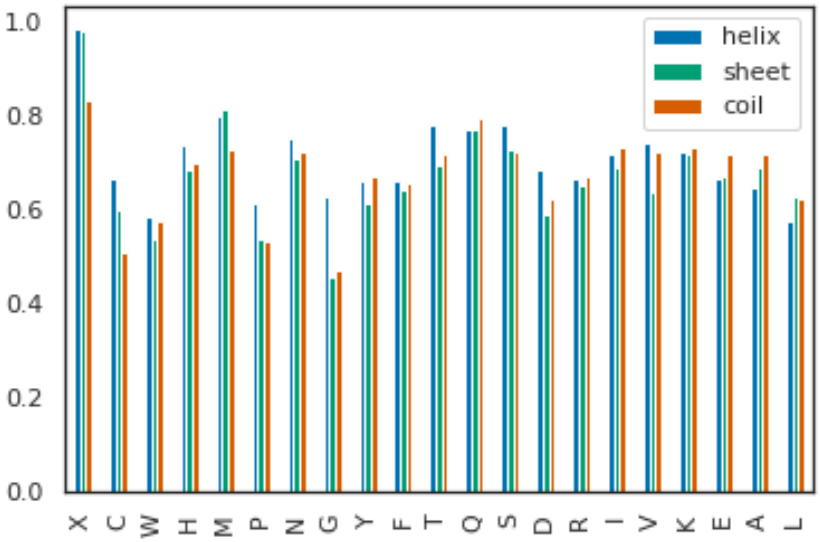


Figure 4.10: For each unique aa in the original sequences, the plot shows the average alteration rate per secondary structure. The secondary structure of an aa is defined by the secondary structure of the aa to which alignments are created. All the values in this plot are based on at least 4000 datapoints.

CHAPTER 5

PROTEIN SECONDARY STRUCTURE PREDICTION

5.1 Introduction

The main goal of the master dissertation is to develop a machine learning model that can predict protein secondary structures accurately. The state-of-the-art solutions to this problem are mostly based on BRNN's with LSTM. Until recently, applying RNN's (LSTM/GRU) were the way to go in NLP. In RNN's, sentences are processed word by word, making it impossible to train a model in parallel. The difficulty of capturing longterm dependencies is inherent to RNN's too. Similar problems were faced when using these types of models for protein secondary structure prediction: the models can't be parallelized and long-term dependencies are often missed. This is a possible reason why β-sheets are still the most difficult structure to predict (Gao et al., 2018), as their interactions occur more often between aa that are sequentially distant.

In the NLP field, Transformers overcome these problems. By the mechanism of self-attention, the model is non-sequential and thus easy to parallelize. Also, the Transformer models don't have long term dependencies problems because of this same mechanism. Important past information isn't lost, as in every step, the model has direct access to the whole input sequence. These characteristics possibly could improve protein secondary structure prediction too.

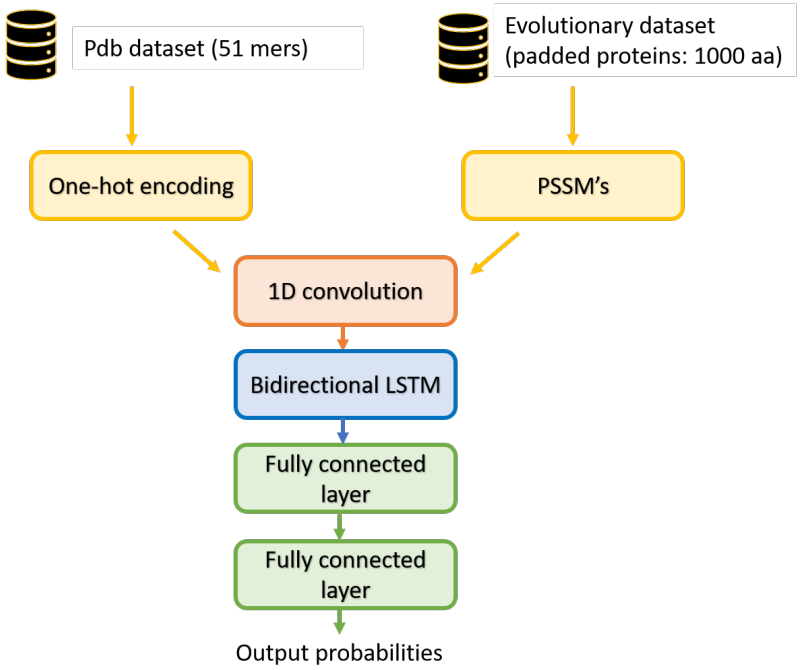
However, when using Transformers to predict protein secondary structure, some adaptations to the standard models are desirable. The protein vocabulary size is a lot smaller than the vocabulary size of a language, and on average, protein sequences are longer than sentences. To be able to compare the Transformer models accurately, BRNN models with LSTM are created too. Comparisons with literature are executed as well.

5.2 Model Architectures

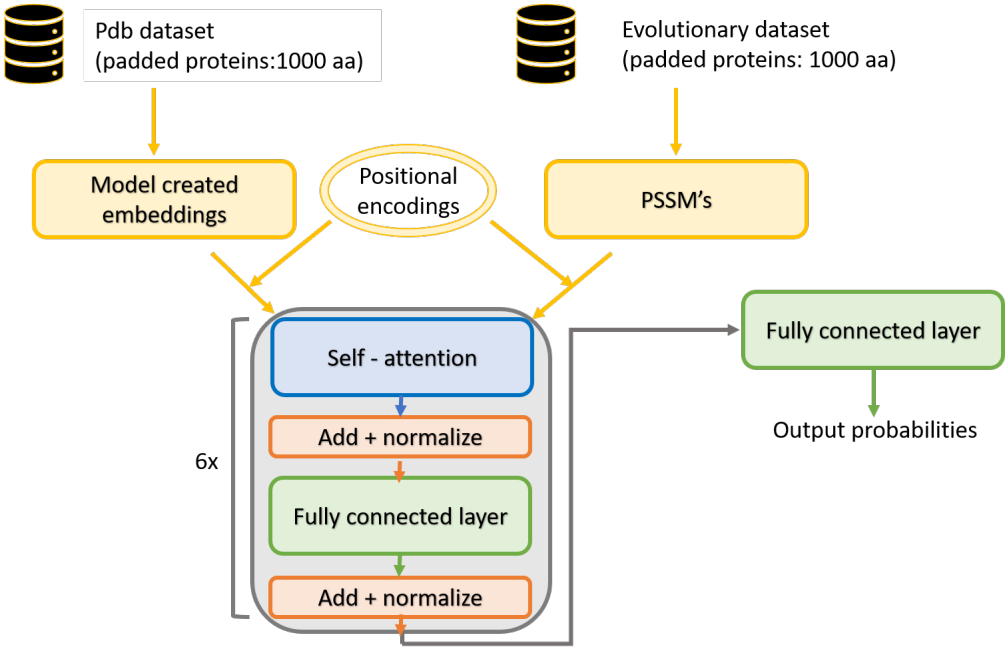
As most protein secondary structures prediction models in literature are based on BRNN in combination with 1D convolutions, such a model is created. A relatively basic architecture is obtained and displayed in Figure 5.1a. For both datasets, very similar models are created. Both models contain a 1D convolution to create dense representations for the input embeddings and to capture local dependencies. This building block is followed by a BRNN with LSTM to capture long term dependencies. There is chosen for LSTM because they made the model perform slightly better than GRU. As the last step, two fully connected layers were added, providing the output probabilities. The architecture of the model wasn't optimized, as this takes a lot of trial and error and these types of models weren't the main focus of the thesis.

Several Transformer models are created too. A schematic representation of their architecture is visualised in Figure 5.1b. Only an encoder is used, as the performed task is a multi-class classification task with limited output classes. For the pdb data, embeddings were created as a part of the model. This way, the representations of the different aa are optimized during training. Six encoder layers were used, where each encoder layer has the same layout. An encoder layer contains a self-attention layer (with six attention heads), followed by an residual connection and normalization layer, a fully connected feed-forward layer, and again an residual connection and normalization layer. The output of the previous encoder layer is the input of the next one. The output of the last encoder layer is fed to a fully connected layer. This layer produces the output probabilities. Some model parameters were tweaked during optimization. Modules were added or left out, to observe the effect of these changes. However, no full hyperparameter tuning is performed.

For both datasets, two types of Transformer models are used. The same overall architecture is used in both, but the self-attention module is constructed differently. The first, plain Transformer model uses the same self-attention as shown in Figure 3.10. To a second type of Transformer, 1D convolutions were added in the self-attention layer, as shown in Figure 5.2 (Clauwaert and Waegeman, 2019). By adding these 1D convolutions, local dependencies are captured directly. The protein vocabulary size only contains 20/21 different aa, resulting in a low input-complexity, much lower than the input-complexity of language. K-mers can be seen as equivalents for words (as both are built from letters). Adding convolutions in the self-attention layer is an elegant way to use kmer representation without losing resolution, as would be the case when kmers were given as input features. Different sizes of d_{conv} were examined, and a kernel size of 9 seemed to work best. This is a kernel size close to the mean length of an α-helix. Other types of convolutions over Q, K, and V were tested but weren't as



(a) BRNN model architecture using LSTM modules.



(b) Transformer model architecture.

Figure 5.1: a: Model architecture of BRNN models with LSTM modules. b: Model architecture for Transformer models.

successful as this approach. 1D convolutions over the input features didn't improve performance either.

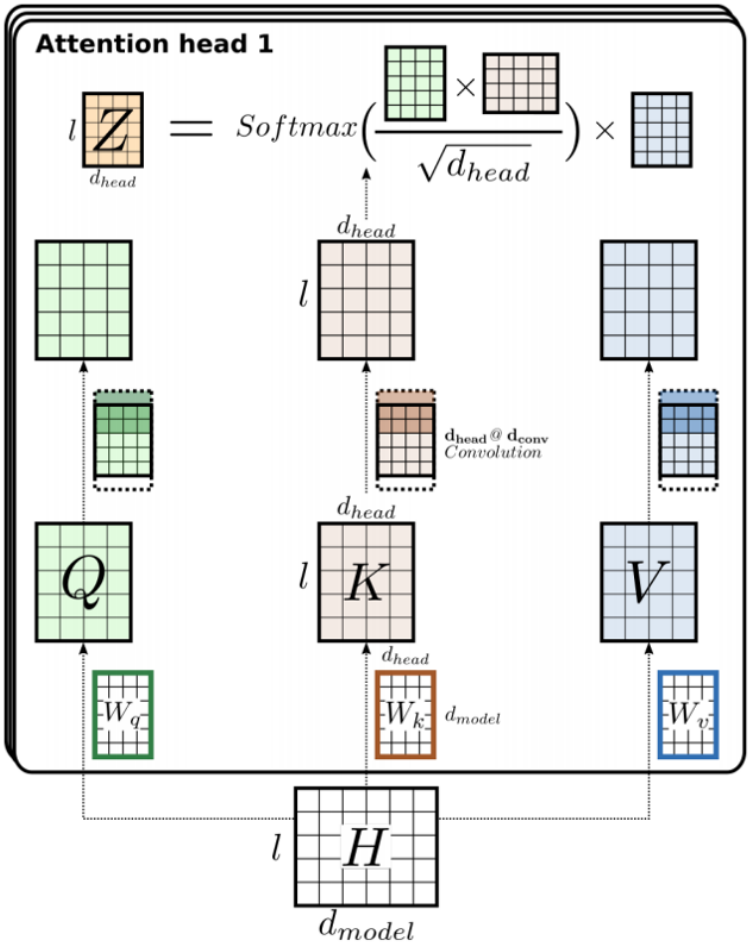


Figure 5.2: An overview of one self-attention head using convolutions. The **q**,**k** and **v** vectors are calculated as explained previously in Section 3.8.1. Calculating them for all l aa in the protein results in matrices Q, K, and V. Over these matrices, a single convolutional layer is applied. This layer contains as many kernels as d_{head}, and the matrix is padded. This way, the dimensions don't change by applying the convolutional layer. The vector representations **q**,**k** and **v** are transformed. The size of the convolution is d_{conv}, meaning the vector representation of every element is depending on d_{conv} neighboring aa. This way, local information about neighboring aa can be captured easily. Image from Clauwaert and Waegeman (2019).

To optimize the models, stochastic gradient descent (SGD) is used. The cross-entropy loss function is combined with the adam optimizer. Both weighted and unweighted cross-entropy loss are performed, and the outcomes are compared. A weighted version of the model was performed to correct for the output class imbalance. The learning rate is set to 0.0004. Independent train, test, and validation sets were created. Models were trained on a GPU-server until overfitting was observed. After every pass of the entire training set through the model (an epoch), the loss on the validation set is calculated. The model at the epoch with the lowest validation loss is selected. At

the end, Q3 accuracies are reported based on the performance of this selected model on the test set.

5.3 Model Performances

5.3.1 Models based on pdb data

Models that only get sequences as input will perform substantially worse than models that have PSSM's as inputs. This was shown in literature (Heffernan et al., 2017, 2018). Models with different input-embeddings should be compared separately. The models solely based on sequence data (from the pdb dataset) will achieve lower accuracies. These models are compared in this section. Different types of models are tested. The results can be seen in Table 5.1.

Over all models, BRNN models using LSTM have the lowest memory usage, because of the way the data is read into the model, which was done with 51mers instead of whole proteins. The Transformer models contain far less parameters than the BRNN (210,000-380,000 vs over 3,000,000). The size of the created Transformer model is in line with the similar Transformer models used in other biological settings (Clauwaert and Waegeman, 2019). Models used in NLP, however, typically contain a lot more parameters, up to 17 billion (Johnson, 2020), and are trained with enormous datasets. The fact the models created in this dissertation are small possibly can affect the performance of these models. Creating bigger models, however, was difficult without overloading the GPU-server. The Transformer models train faster than the BRNN model, albeit needing more training epochs. The BRNN with LSTM needs the less training epochs, mostly because of the way the data is read in.

When comparing the accuracies of the models created in this master dissertation, the weighted Transformer with convolutions performs the best. 71.53 % of the time, the secondary structures are predicted right. It seems weird that the balanced model performs best overall, because in this model, it isn't the overall performance that is optimized, but the performance on every class separately, as each class is looked at equally attentive. In practice, this means coil and helical regions are taken less into consideration in the balanced model, and sheets more. Probably, this is because when taking the less prominent classes, which are normally predicted worse, more into consideration, their prediction accuracy rises. This can counter the slightly worse prediction accuracy on the overrepresented classes. The validation loss and the test loss are very similar, showing that no hyperparameter overtuning happened. This was to be expected, as almost no hyperparameter tuning was performed. The Transformer

Table 5.1: Summary of different types of models used to predict protein secondary structure solely based on sequence information. The columns show how much memory was used when training the model, how many parameters the model archictecture contains, how long the training of the selected model with a specific architecture took, how many epochs the selected model is trained for, and the Q3 accuracy on the test set. The best values per column are indicated in bold. For the models from literature, only the accuracy was reported.

Model	Memory Use	# Parameters in model	Training time	Epochs	Accuracy
BRNN with LSTM	**807 MB**	3,297,477	568 s	**4**	67.25 %
Transformer	4,655 MB	**211,139**	**211 s**	25	53.51 %
Transformer Conv	4,655 MB	377,603	212 s	5	69.51 %
Transformer Conv balanced	4,655 MB	377,603	377 s	17	71.53 %
SPIDER3-Single (Heffernan et al., 2018)	\	\	\	\	**72.52 %**
PSIpred-Single	\	\	\	\	69.61 %

model without convolutions performed considerably worse, indicating the addition of convolutions to the self-attention mechanism improves the model. Comparing the performance of the BRNN with the LSTM model to the Transformer models is hard, as different types of data were used to train both models.

In Table 5.1, the accuracies of the created models are compared to models from literature and the current state-of-the-art. One should be careful when comparing performances, as not the same training and test set were used for these models. Spider3-single is based on a BRNN with LSTM, where psiPRED single is an old model from 1999, and is solely based on vanilla ANN's. The balanced Transformer model performs better than PSIpred-single and worse than SPIDER3-single. This Transformer model, however, isn't optimized completely yet. Hyperparameter tuning is only done to a very small extent, and the model is still relatively small. Enlarging the models could still lead to improvements, as no the overfitting of the model isn't very pronounced yet.

The SPIDER3-single model uses different iterations, where the predicted output of a previous iteration is appended to the input for the next iteration. It is shown that this improves the model performance. After one iteration, the model performs a Q3 accuracy of 71.8 %, which is very close to the accuracy of the balanced Transformer

model with convolutions. Adding iterations to the Transformer model could possibly improve predictions.

5.3.2 Models based on evolutionary data

Models created by using the evolutionary data are analysed, and the results are shown in Table 5.2. When training the models on the evolutionary data, the same data loader was used for all models.

Over the models Compared in Table 5.2, the BRNN with LSTM needs the less memory and contains the less model parameters. Most probably those two observations are linked. The memory complexity of Transformer networks is in the order of $0(n^2)$, as for self-attention (with n the length of the input sequence). This isn't the case in BRNN's, where it is $0(n)$. This can explain previous observations. For long sequences, BRNN's are less costly in terms of memory use. The Transformer models with convolutions need the less training time and training epochs. These two observations are linked too, as per epoch, Transformer models with convolutions need more time than the other models in this Table.

When comparing accuracies of the created models, the BRNN with LSTM performs best. The difference with the Transformer containing convolutions is small (0.51%). However, it still shows that the BRNN model with LSTM works better than the Transformer model. Both models are unoptimized, and adapting the model probably can improve the performances. Adaptations can greatly influence the performance, as illustrated by the difference in performance between the basic Transformer model and the Transformer model to which convolutions were added.

The fact that the Transformer models didn't outperform all other models can have multiple reasons. Transformer networks are created to analyse language. Languages show a great word complexity, and the input embeddings are typically 512/1024 integers. In our case, the embedding size is 10 for the pdb data and 22 for the evolutionary data. More features weren't necessary to distinguish the 20/21 aa. Possibly, Transformers need long and complex embeddings to perform optimally. A second reason could be the size of the Transformer models used in this master dissertation. They are way smaller than the models used in NLP. The size, however, is similar to the models used by Clauwaert and Waegeman (2019) on full prokaryotic genomes. Rives et al. (2019), the only (unpublished) paper using Transformers on proteins, models containing up to 708 million parameters are used, with embedding dimensions up to 1280. Training these models on the provided GPU wouldn't be possible. In the paper, only learning the embedding vectors already took four days.

Table 5.2: Summary of different types of models used to predict protein secondary structure solely based on sequence information. The columns show how much memory was used when training the model, how many parameters the model architecture contains, how long the training of the selected model with a specific architecture took, how many epochs the selected model is trained for, and the Q3 accuracy on the test set. The best values per column are indicated in bold. All accuracies in this table, including the accuracies of literature models, are calculated on the same test set, derived from (Torrisi et al., 2018). The star indicates this accuracy wasn't calculated on the whole dataset, as SPIDER3 can not deal with rare/ missing aa accounting for almost a third of the test set.

Model	Memory Use	# Parameters in model	Training Time	Epochs	Accuracy
BRNN with LSTM	**761 MB**	**95,497**	10,732 s	55	82.14 %
Transformer	4,657 MB	233,971	20,852 s	74	72.34 %
Transformer Conv	4,657 MB	312,775	5,713 s	21	81.63 %
Transformer Conv balanced	4,657 MB	312,775	**4,626 s**	**17**	81.29 %
Porter5 (Torrisi et al., 2018)	\	\	\	\	**84.2 %**
PSIPRED 4.01 (McGuffin et al., 2000)	\	\	\	\	82.06 %
RaptorX (Gao et al., 2018)	\	\	\	\	82.04%
Deep CNF (Wang et al., 2016)	\	\	\	\	81%
SPIDER3 (Heffernan et al., 2017)	\	\	\	\	83.15% *

Another possible reason is that normally, Transformer models are trained with enormous amounts of data. We can only provide the model 15,637 protein sequences, containing 3,649,741 aa. Language models are trained on billions of words. This likely affects the performance of the model too but is very difficult to solve. Possibly, different batches with sequentially similar proteins could be created. This way, the sequence identity cut-off of 30 % can be bypassed, and more sequence variation can be used to train the models. However, this won't lead to a much more diverse training set.

Proteins are very long sequences, whereas Transformers are mostly used on sentences that are not often longer than 100 words. Theoretically, the model should be able to capture all kinds of dependencies in such long sequences, due to the self-attention mechanism. Possibly, too much information is given to the model at every step, and the model isn't able to detect which information is useful. The selection may not be possible because the model is too small or because not enough data is provided.

In NLP tasks where whole texts need to be processed, sequences of 1000 words aren't enough. For such problems, the TransformerXL is created (Dai et al., 2019). With a clever trick, only a fixed amount of words are read at the time. Information about previous words, however, can still be accessed. As the maximum length of proteins used in this dissertation is 1000 aa, it didn't seem necessary to implement such architecture. When the input length would influence the model performances drastically, this type of architecture could provide a solution.

A clean comparison between the models created in this dissertation and models from literature can be done, as the dataset was retrieved from literature. Torrisi et al. (2018) used the same training set to train their models. Also, the same test set was used. With this specific test set, following models are evaluated too: PSIPRED 4.01 (McGuffin et al., 2000), RaptorX (Gao et al., 2018), Deep CNF (Wang et al., 2016) and SPIDER3 (Heffernan et al., 2017). When looking at Table 5.2, Porter5 outperforms all models. The BRNN with LSTM, created in this dissertation, is the third-best model, but the performance is close to what PSIPRED 4.01 and RaptorX perform. The balanced Transformer model with convolutions performs slightly worse, but still better than Deep CNF. SPIDER3 performs better than the models created in this master dissertation but isn't usable for all protein sequences, as it can not predict protein secondary structure for proteins containing rare or unknown aa.

The accuracies PSIPRED, RaptorX and DeepCNF show on this test set are much lower than what they report in the publications introducing the methods. DeepCNF, for example, reports accuracies around 84% on other datasets. The test set greatly in-

fluences the performance of the models, explaining the differences. The performance of the Porter5 model should be put in perspective, as the reported accuracies are created by the paper publishing this model. For this reason, it is very important the code used to train the models is available, which isn't the case for Torrisi et al. (2018). It is a clear trend that for all models, performances are always higher in the paper publishing the model. This, however, does not change the fact that SPIDER3 still performs better than the BRNN LSTM model created in this master dissertation.

5.3.3 Comparison of the datasets

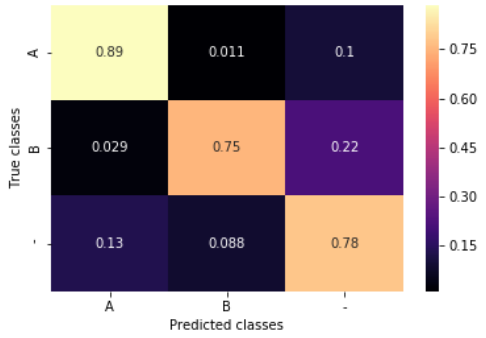
When comparing the performances on the different datasets (shown in Table 5.1 and 5.2), models based on evolutionary data clearly outperform models solely based on sequential data. This doesn't mean models solely based on sequential data can't be interesting. Generating PSSM's can take a lot of time, more time than a model needs to create a prediction. For most models, more than 99 % of the prediction time is used for calculating the PSSM's. Not using PSSM's could lead to extremely fast predictions that are a bit less accurate. Also, PSSM's can only be created when similar sequences are available. The majority of the protein sequences have few to no known homologs (Ovchinnikov et al., 2017), leading to (almost) no aligned sequences. No PSSM's can be created, and models that are trained based on evolutionary data will do a less good job predicting secondary structure solely based on sequence information (Heffernan et al., 2018). When time and similar sequences are available, models based on evolutionary data are the best option. When one of the two prerequisites is lacking, models solely based on sequence information provide a worthy alternative.

5.4 Prediction analysis and interpretation

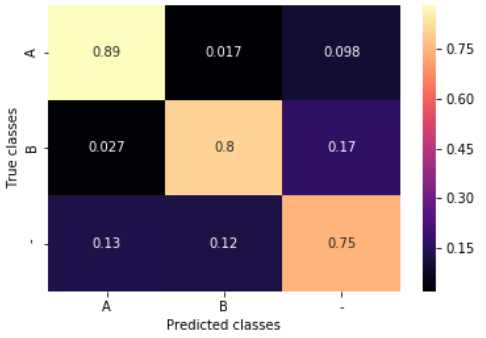
Only reporting the accuracy of a model does not tell us enough about the model's performance. A more in-depth analysis is executed. There is examined which mistakes are most frequent in the different models and which types of biases are existing. The analysis is done for models based on evolutionary data because those models perform best.

5.4.1 Confusion matrices

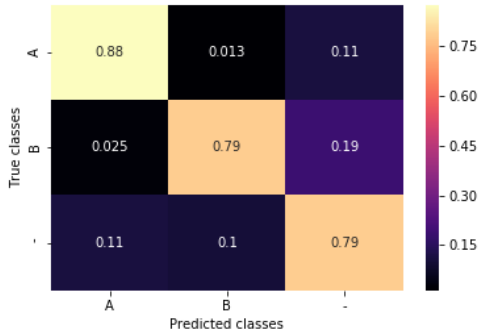
For all models created based on evolutionary data, the confusion matrices can be seen in Figure 5.3. On the diagonal axes, in bright colors, the true positives are indicated. The value indicates the relative frequency of the class being predicted correctly. For



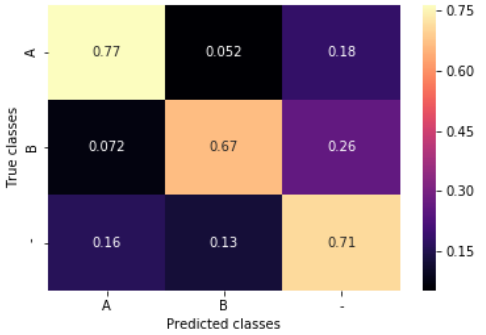
(a) Confusion matrix for Transformer model with convolutions



(b) Confusion matrix for weighted Transformer model with convolutions



(c) Confusion matrix for BRNN with LSTM



(d) Confusion matrix for Transformer model

Figure 5.3: Confusion matrices for all created evolutionary models, based on the test sets. The rows indicate the true classes and the columns the predicted classes. The confusion matrices show relative frequencies over the columns.

all different models, the prediction accuracy of the α-helix is the highest. For example, when using the Transformer model with convolutions (Figure 5.3a), if the aa is present in an α-helix in the protein structure, this will be predicted correctly in 89% of the cases. In 1.1 % of the cases, the α-helical structure will be wrongly predicted to be a β-sheet, and in 10 % of the cases, it will be wrongly predicted to be in a coil region.

This shows a second trend over all models. Predictive mistakes altering α-helices and β-sheets are relatively rare. Both structures are more often wrongly predicted to be in a coil region than in the other structured region. This is expected and can have multiple reasons. First of all, from a structural point of view, α-helices and β-sheets are more distinct to each other than to coil regions. Secondly, mistakes occur more often on the borders of secondary structures. Generally, structural regions are flanked by coil regions. This makes that the class 'coil' is predicted way more often than it appears in the sequences. In all confusion matrices, the sum over the third column (the column predicted class 'coil') is greatly larger than one (also when taking into account the relative importance of the different rows).

Only when the model is balanced, the predictions of β-sheets are more accurate than the prediction of coil regions. Most likely, this is because, in unbalanced models, coils appear more frequently than β-sheets in proteins. This gives the model more pressure to predict those structures correctly. With a balanced model, all classes are treated the same, regardless of their frequency of appearance. When adding weights to the cost function, a clear effect can be noticed. The accuracy of predicting β-sheets rises above the accuracy of predicting coil regions. This indicates coil regions are the most difficult to predict, possibly because they are the less structured and thus more diverse. When the model is balanced, more aa are wrongly predicted as β-sheet, because it was more emphasised during training, which is an unfortunate side-effect. Although α-helices are less emphasised when the model is balanced, the predictive performance on α-helices is constant.

When comparing the unbalanced Transformer model with convolutions to the BRNN with LSTM, the BRNN produces slightly more stable predictions over all classes. It predicts α-helices slightly worse, but β-sheets better. It seems less biased towards the overrepresented output classes. It gives the impression Transformers are struggling more to predict β-sheets correctly than BRNN's with LSTM.

5.4.2 Neighborhood dependence of accuracy

The location of the aa in the protein influences the prediction accuracy. This is shown in Figure 5.4. The aa were divided into four different categories, based on their location in the known protein structure: boundary of the protein, alone (meaning both neighbors have a different secondary structure), boundary of a secondary structure (meaning one neighbor has a different secondary structure) and internal (both neighbors have the same secondary structure). This was done based on true labels. So, there is no certainty these aa were predicted to be on the border of secondary structure when assigned this category. The average accuracies per category were calculated for all models based on evolutionary data. Different categories were present with different frequencies, as shown in Table 5.3. A huge imbalance can be observed, but none of the classes is so rare it would lead to unrepresentative results. Clear differences in accuracies between the different categories can be spotted in Figure 5.4.

All models show similar trends over the different neighbor categories. This indicates the trends are inherent to the data. For all models except for the plain Transformer model, the predictive accuracy on the protein borders is almost one, as virtually all proteins start and end with a coil. Recognizing this pattern is not too hard.

Table 5.3: Table indicating the amount of aa per neighborhood category

kind of neighborhood	amount of aa in this kind of neighborhood
boundary protein	6258
alone	11.103
boundary structure	187.845
internal	424.458

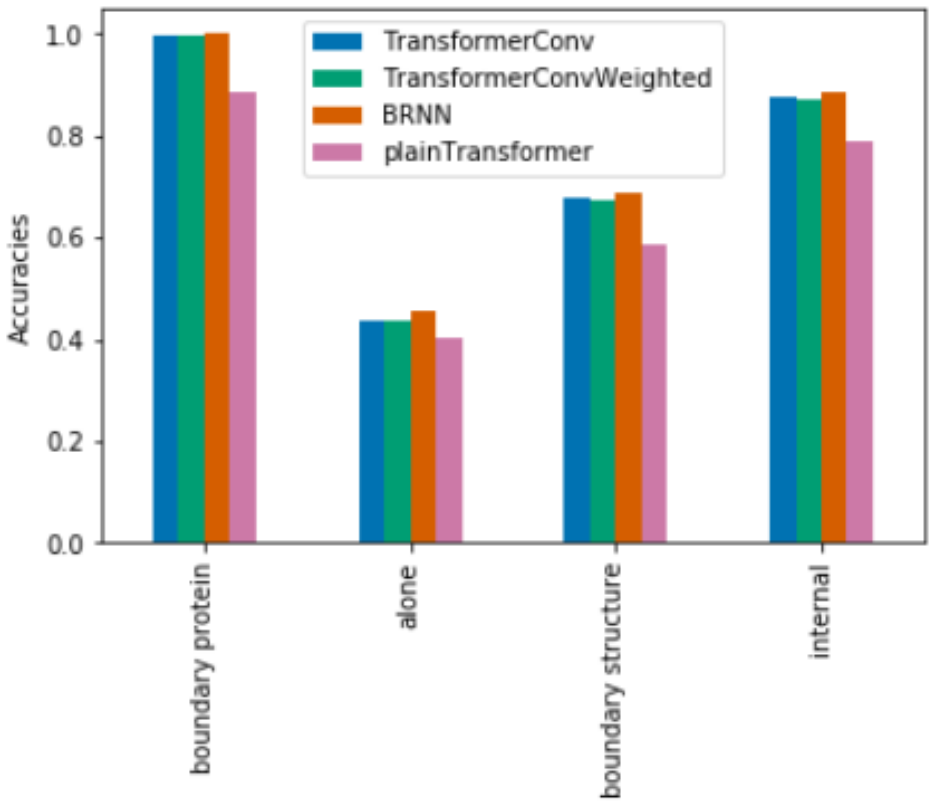


Figure 5.4: The aa are divided in different categories depending on their neighborhood in the protein. For all categories and all aa, the average accuracy is plotted.

Aa with two structural different neighbors in the original protein are the hardest to predict for the used models. Luckily, they are relatively rare (see Table 5.3). Their rarity possibly is the reason for the bad prediction accuracy too. As machine learning models recognize patterns, the pattern of an aa without structurally equal neighbors is so rare it isn't recognized as a pattern. Small structures in general might be harder to detect too.

Borders of secondary structures are predicted more accurately than the "alone" category, but still worse than aa that are located in the structures. It seems perfectly logical that mistakes are made more often at the borders of secondary structures. When a protein secondary structure is predicted wrong, the borders will be predicted wrong too. Also, aa without structural equal neighbors are rare, so a mistake of one aa in within secondary structure won't be made often. Next to that, the borders of protein secondary structures aren't defined clearly, introducing noisy structure ends that are difficult to predict correctly. This means protein secondary structures are fairly often predicted too long or too short, by one aa. Mistakes of more than one aa are rarer. A false split in such structure, however, will be very uncommon.

5.4.3 AA dependence of model performance

Different aa appear with different frequencies in the proteins and in the secondary structures. Differences in predictive performance per aa are expected. For all different models on evolutionary data, the performance per aa in the test set is shown in Figure 5.5. The aa are ordered based on frequency of occurrence. For all models, almost identical trends can be observed, and these trends are similar to what is reported in literature (Heffernan et al., 2018). It seems the trends are intrinsic to the data, and not depending on the used models. In general, a slight downwards trend can be spotted in the graph. When leaving out the X (as this is not a specific aa, but codes for rare/unknown aa), correlations around 0.6 between prediction accuracies and aa frequencies are recorded. The abundance of the aa affects the predictive power for the aa. This was to be expected. The more data available for a certain aa, the more emphasis the model gives to this aa, and the better it will have learned the behavior of the aa. Some valleys in Figure 5.5, that do not follow the trend described above, can be partially explained by looking at Figure 4.3. Elements such as C (cysteine), H(Histidine), S(serine), and T(Threonine) all have balanced appearance in all types of secondary structures, leading to an extra predictive challenge.

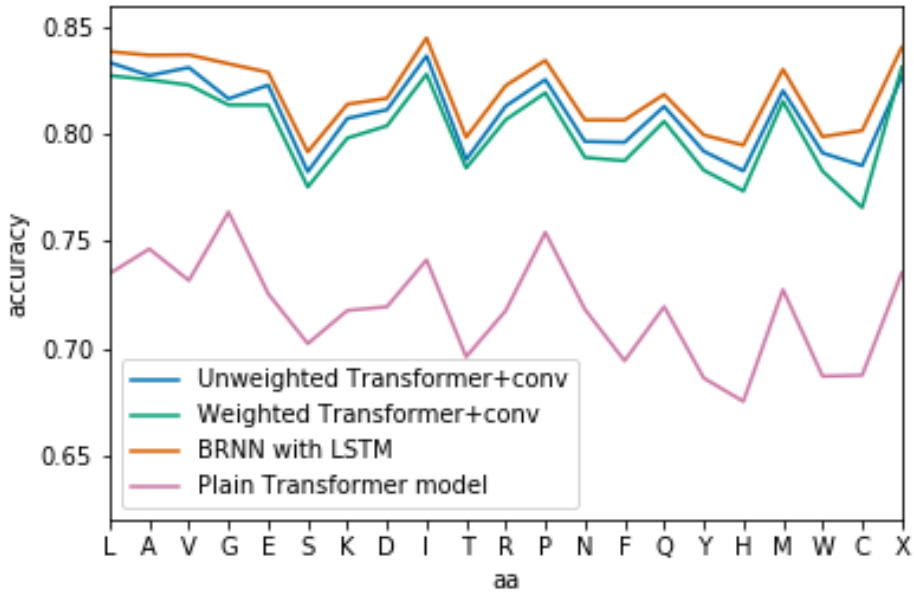


Figure 5.5: Accuracies per model and per aa for all models trained on the evolutionary dataset. The aa are ordered based on frequency of appearance in α-helices in the training set, where L is the most frequent aa in α-helices and X the less frequent. The accuracy is calculated based on the performance on the test set.

5.4.4 Popularity bias

Accuracy is calculated per unique aa and secondary structure. The results are shown in Figure 5.6 for BRNN model. The aa are ordered based on the relative frequency of occurrence in α-helices. In general, the blue bars are the highest, indicating α-helices are predicted most often correctly. However, for aa that are on average more occurring in coil regions, such as proline (P) and glycine(G) (as can be seen in Figure 4.3), these coil regions are predicted more accurately. This shows there is a popularity bias. The secondary structure in which the aa occurs most frequently will be predicted most accurately. This is to be expected, as the data leads the model towards these predictions. These effects work the other way around too. When the aa appears most often in coils (as proline does), it will often be wrongly be predicted as 'coil'. When for proline, the true class is β-sheet, in 54 % of the cases it will be predicted wrongly as 'coil'. For M (methionine), an aa that occurs often in β-sheets, true class β-sheet is only predicted wrongly as coil region in 16 % of the cases.

Countering this bias is difficult, and this effect is partly wanted, as it improves the overall performance. One should be aware of the effect when interpreting the predictions these models created.

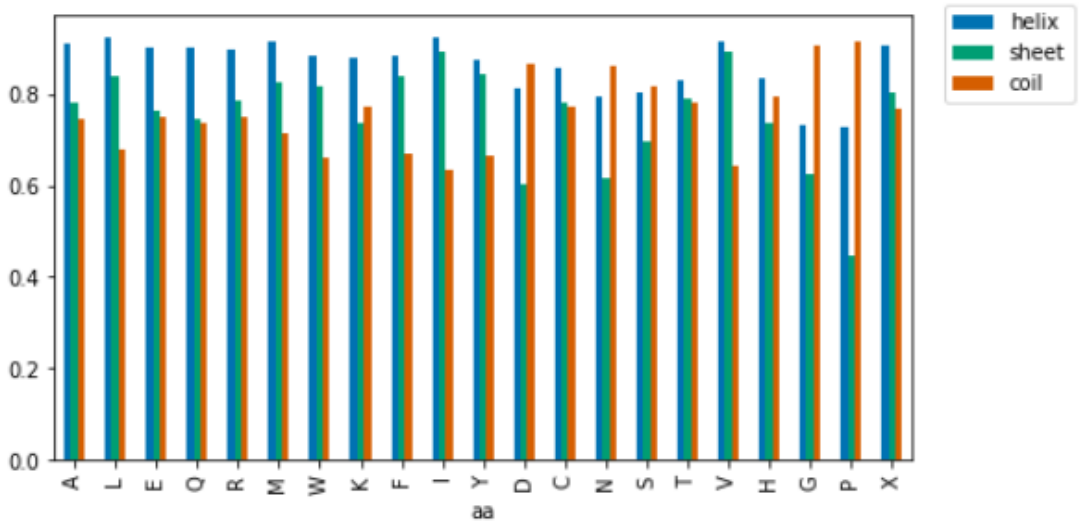


Figure 5.6: Accuracies per aa and per secondary structure are shown for the BRNN with LSTM model. All models showed extremely similar results. Aa in the plot are ordered based on their relative frequency of appearance in α-helices.

CHAPTER 6

FUTURE PERSPECTIVES AND CONCLUSIONS

6.1 Future perspectives

Data cleaning

As proposed by other studies, and according to the data exploration and filtering performed in this thesis, cleaning up the data yields the largest improvements within the field of protein secondary structure prediction using machine learning models. Due to artefacts, the highest possible accuracy with the current data is 88-90 %, while models perform 84 %. The room for improvement is mainly situated in the 88 to 100 % range. These improvements can only be achieved when the data is cleaned up. Firstly, all secondary structures in the pdb database should be determined with the same algorithm, which leads to a higher consistency. Secondly, all types of mistakes in the mmCIF pdb files should be corrected. With both adaptations, protein secondary structure prediction performance will rise tremendously. In addition to cleaner data, higher data quantity also contributes to training the models. More sophisticated ways of data management allow for different sequences with a sequence similarity >30 % to be used in different training rounds. With more data to train machine learning models, their prediction performance will most likely rise.

Adaptations to the created models

The models created in this thesis are relatively small compared to the Transformer models described in literature. Enlarging the models and using bigger input embeddings might improve the prediction performance. It should be noted that hyperparameter tuning can boost the model's performance, but is not extensively applied in this thesis because of its time-intensive nature. Using other types of Transformer models might increase the model's performance. Different types can be tested, such

as a complete encoder-decoder architecture, which adds decoder-attention. As the proteins' length possibly influences the model performance, Transformer XL elements can be built in. Another option is the implementation of a self-attention mechanism in a BRNN with LSTM model, blending the two types of models. This has already worked for text-classification (Jing, 2019). Other types of embedding vectors could be used, such as embedding vectors based on physiochemical properties. More interestingly, combinations of different embedding vectors can be used to train the model.

Applications in the protein domain

Transformer models with protein sequences or PSSMs for input can be applied beyond three-state secondary structure prediction. A first additional application could be eight-state secondary structure prediction. As shown in Chapter 4, these eight classes are extremely unbalanced. The model will have to be adapted to cope with this imbalance. Possibly, hierarchical classification can improve predictions. Furthermore, Transformer Networks can predict other protein properties, such as backbone angles, solvent accessibility and residue-residue contact maps. The combination of these features' prediction with protein secondary structures' prediction could improve the predictive performance of all properties, specifically in iterative learning, where previous iterations' outputs can be used for subsequent iterations.

Transformer models are used in NLP for enormous amounts of possible output classes (the vocabulary size of a language). They will probably accurately predict protein function based on sequence. Taking previously predicted protein secondary structures or other properties as additional input features into account will most likely improve protein function prediction.

6.2 Conclusions

In this thesis, two datasets to train models that predict protein secondary structure are created. Both datasets contain different proteins and different information about these proteins. These datasets are thoroughly examined; especially the dataset directly retrieved from the pdb server is found to be extremely noisy.

The models created in this thesis, based on those datasets, can compete with most models described in literature, especially when taking into account they have not yet been optimized completely, but they do not clearly outperform the models in literature. Based on this thesis' results, the considerable improvements Transformers

models caused in the NLP field cannot be extended towards protein secondary structure predictions. The models created in this thesis are relatively fast and broadly applicable. A further optimization of the created models might lead to Transformers playing a role in the future of protein bioinformatics.

BIBLIOGRAPHY

Adhikari, A., Ram, A., Tang, R., and Lin, J. (2019). Docbert: Bert for document classification. *arXiv preprint arXiv:1904.08398*.

Alammar, J. (2018). The illustrated transformer. *URL http://jalammar. github. io/illustrated-transformer*.

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. (2016). Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182.

Anfinsen, C. B. (1973). Principles that govern the folding of protein chains. *Science*, 181(4096):223–230.

Berman, H. M., Bourne, P. E., Westbrook, J., and Zardecki, C. (2003). The protein data bank. In *Protein Structure*, pages 394–410. CRC Press.

Bhagwat, M. and Aravind, L. (2007). Psi-blast tutorial. In *Comparative genomics*, pages 177–186. Springer.

Bird, S., Klein, E., and Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Bottou, L. and Bousquet, O. (2008). The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168.

Branden, C. I. and Tooze, J. (2012). *Introduction to protein structure*. Garland Science.

Chen, Y., Jiang, H., Li, C., Jia, X., and Ghamisi, P. (2016). Deep feature extraction and classification of hyperspectral images based on convolutional neural networks. *IEEE Transactions on Geoscience and Remote Sensing*, 54(10):6232–6251.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Clauwaert, J., Menschaert, G., and Waegeman, W. (2018). Deepribo: precise gene annotation of prokaryotes using deep learning and ribosome profiling data. *bioRxiv*, page 317180.

Clauwaert, J. and Waegeman, W. (2019). Novel transformer networks for improved sequence labeling in genomics. *bioRxiv*, page 836163.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Deng, L., Yu, D., et al. (2014). Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387.

Dill, K. A. and MacCallum, J. L. (2012). The protein-folding problem, 50 years on. *science*, 338(6110):1042–1046.

Dong, L., Xu, S., and Xu, B. (2018). Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888. IEEE.

Drenth, J. (2007). *Principles of protein X-ray crystallography*. Springer Science & Business Media.

Du, J., Lu, X., Long, Z., Zhang, Z., Zhu, X., Yang, Y., and Xu, J. (2013). In vitro and in vivo anticancer activity of aconitine on melanoma cell line b16. *Molecules*, 18(1):757–767.

Fiedorczuk, K., Letts, J. A., Degliesposti, G., Kaszuba, K., Skehel, M., and Sazanov, L. A. (2016). Atomic structure of the entire mammalian mitochondrial complex i. *Nature*, 538(7625):406.

Firth, J. R. (1957). A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*.

Fodje, M. and Al-Karadaghi, S. (2002). Occurrence, conformational features and amino acid propensities for the π-helix. *Protein Engineering, Design and Selection*, 15(5):353–358.

Gao, Y., Wang, S., Deng, M., and Xu, J. (2018). Raptorx-angle: real-value prediction of protein backbone dihedral angles through a hybrid method of clustering and deep learning. *BMC bioinformatics*, 19(4):100.

Genthial, G., Lucas Liu, L., Oshri, B., and Ranjan, K. (2019). Cs224n: Natural language processing with deep learning lecture notes: Part vi: Neural machine translation, seq2seq and attention.

Gething, M.-J. and Sambrook, J. (1992). Protein folding in the cell. *Nature*, 355(6355):33.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

Godzik, A., Jambon, M., and Friedberg, I. (2007). Computational protein function prediction: are we making progress? *Cellular and molecular life sciences*, 64(19-20):2505.

Gómez, I., Franco, L., and Jerez, J. M. (2009). Neural network architecture selection: can function complexity help? *Neural Processing Letters*, 30(2):71–87.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Guo, Y., Li, W., Wang, B., Liu, H., and Zhou, D. (2019). Deepaclstm: deep asymmetric convolutional long short-term memory neural models for protein secondary structure prediction. *BMC bioinformatics*, 20(1):341.

Gurcan, M. N., Sahiner, B., Chan, H.-P., Hadjiiski, L., and Petrick, N. (2001). Selection of an optimal neural network architecture for computer-aided detection of microcalcifications—comparison of automated optimization techniques. *Medical Physics*, 28(9):1937–1948.

Hanson, J., Paliwal, K., Litfin, T., Yang, Y., and Zhou, Y. (2019). 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*, 35(14):2403–2410.

Hartl, F. U. and Hayer-Hartl, M. (2009). Converging concepts of protein folding in vitro and in vivo. *Nature structural & molecular biology*, 16(6):574.

Heffernan, R., Paliwal, K., Lyons, J., Singh, J., Yang, Y., and Zhou, Y. (2018). Single-sequence-based prediction of protein secondary structures and solvent accessibility by deep whole-sequence learning. *Journal of computational chemistry*, 39(26):2210–2216.

Heffernan, R., Yang, Y., Paliwal, K., and Zhou, Y. (2017). Capturing non-local interactions by long short-term memory bidirectional recurrent neural networks for improving prediction of protein secondary structure, backbone angles, contact numbers and solvent accessibility. *Bioinformatics*, 33(18):2842–2849.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116.

Ibrahim, M. A., Abdelrahman, A. H., and Hassan, A. M. (2019). Identification of novel plasmodium falciparum pi4kb inhibitors as potential anti-malarial drugs: Homology

modeling, molecular docking and molecular dynamics simulations. *Computational biology and chemistry*, 80:79–89.

Jing, R. (2019). A self-attention based lstm network for text classification. In *Journal of Physics: Conference Series*, volume 1207, page 012008. IOP Publishing.

Johnson, K. (2020). Microsoft trains world's largest transformer language model.

Kabsch, W. and Sander, C. (1983). Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers: Original Research on Biomolecules*, 22(12):2577–2637.

Kaczanowski, S. and Zielenkiewicz, P. (2010). Why similar protein sequences encode similar three-dimensional structures? *Theoretical Chemistry Accounts*, 125(3-6):643–650.

Karlik, B. and Olgac, A. V. (2011). Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122.

Karshikoff, A., Nilsson, L., and Ladenstein, R. (2015). Rigidity versus flexibility: the dilemma of understanding protein thermal stability. *The FEBS journal*, 282(20):3899–3917.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kurgan, L. and Miri Disfani, F. (2011). Structural protein descriptors in 1-dimension and their sequence-based predictions. *Current Protein and Peptide Science*, 12(6):470–489.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

Lukin, J. A., Kontaxis, G., Simplaceanu, V., Yuan, Y., Bax, A., and Ho, C. (2003). Quaternary structure of hemoglobin in solution. *Proceedings of the National Academy of Sciences*, 100(2):517–520.

Manning, C. (2019). Natural language processing with deep learning cs224n/ling284. lecture 1-11. *Stanford University*.

McGuffin, L. J., Bryson, K., and Jones, D. T. (2000). The psipred protein structure prediction server. *Bioinformatics*, 16(4):404–405.

Mesnil, G., He, X., Deng, L., and Bengio, Y. (2013). Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In *Interspeech*, pages 3771–3775.

Messer, K. and Kittler, J. (1998). Choosing an optimal neural network size to aid a search through a large image database. In *BMVC*, pages 1–10. Citeseer.

Mijwel, M. (2018). Artificial neural networks advantages and disadvantages. *Retrieved from LinkedIn: https://www. linkedin. com/pulse/artificial-neural-net works-advantages-disadvantages-maad-m-mijwel*.

Nguyen, M. (2018). Illustrated guide to lstm's and gru's: A step by step explanation. *URL https://towardsdatascience. com/illustrated-guide-to-lstms-and-grusa-step-by-step-explanation-44e9eb85bf21*.

Nwanochie, E. and Uversky, V. N. (2019). Structure determination by single-particle cryo-electron microscopy: Only the sky (and intrinsic disorder) is the limit. *International journal of molecular sciences*, 20(17):4186.

Olden, J. D. and Jackson, D. A. (2002). Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological modelling*, 154(1-2):135–150.

Ouyang, W., Wang, X., Zeng, X., Qiu, S., Luo, P., Tian, Y., Li, H., Yang, S., Wang, Z., Loy, C.-C., et al. (2015). Deepid-net: Deformable deep convolutional neural networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2403–2412.

Ovchinnikov, S., Park, H., Varghese, N., Huang, P.-S., Pavlopoulos, G. A., Kim, D. E., Kamisetty, H., Kyrpides, N. C., and Baker, D. (2017). Protein structure determination using metagenome sequence data. *Science*, 355(6322):294–298.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Plaxco, K. W., Simons, K. T., and Baker, D. (1998). Contact order, transition state placement and the refolding rates of single domain proteins. *Journal of molecular biology*, 277(4):985–994.

Plumb, A. P., Rowe, R. C., York, P., and Brown, M. (2005). Optimisation of the predictive ability of artificial neural network (ann) models: a comparison of three ann programs and four classes of training algorithm. *European Journal of Pharmaceutical Sciences*, 25(4-5):395–405.

Rehman, I. and Botelho, S. (2018). Biochemistry, secondary protein structure. In *StatPearls [Internet]*. StatPearls Publishing.

Remmert, M., Biegert, A., Hauser, A., and Söding, J. (2012). Hhblits: lightning-fast iterative protein sequence searching by hmm-hmm alignment. *Nature methods*, 9(2):173.

Rives, A., Goyal, S., Meier, J., Guo, D., Ott, M., Zitnick, C. L., Ma, J., and Fergus, R. (2019). Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *bioRxiv*, page 622803.

Rohl, C. A., Fiori, W., and Baldwin, R. L. (1999). Alanine is helix-stabilizing in both template-nucleated and standard peptide helices. *Proceedings of the National Academy of Sciences*, 96(7):3682–3687.

Rong, X. (2014). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.

Ronin, C., Costa, D. M., Tavares, J., Faria, J., Ciesielski, F., Ciapetti, P., Smith, T. K., MacDougall, J., Cordeiro-da Silva, A., and Pemberton, I. K. (2018). The crystal structure of the leishmania infantum silent information regulator 2 related protein 1: Implications to protein function and drug design. *PloS one*, 13(3):e0193602.

Rost, B. (2001). Protein secondary structure prediction continues to rise. *Journal of structural biology*, 134(2-3):204–218.

Rost, B. and Sander, C. (1993). Improved prediction of protein secondary structure by use of sequence profiles and neural networks. *Proceedings of the National Academy of Sciences*, 90(16):7558–7562.

Rost, B. and Sander, C. (1994). Combining evolutionary information and neural networks to predict protein secondary structure. *Proteins: Structure, Function, and Bioinformatics*, 19(1):55–72.

Saha, S. (2018). A comprehensive guide to convolutional neural networks—the eli5 way. *Towards Data Science*, 15.

Samuel, A. (1959). July 1959. *Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development*, 3(3):210–29.

Schuster, S. C. (2008). Next-generation sequencing transforms today's biology. *Nature methods*, 5(1):16–18.

Smyth, M. and Martin, J. (2000). x ray crystallography. *Molecular Pathology*, 53(1):8.

Socher, R., Bengio, Y., and Manning, C. D. (2012). Deep learning for nlp (without magic). In *Tutorial Abstracts of ACL 2012*, pages 5–5. Association for Computational Linguistics.

Torrisi, M., Kaleel, M., and Pollastri, G. (2018). Porter 5: fast, state-of-the-art ab initio prediction of protein secondary structure in 3 and 8 classes. *bioRxiv*, page 289033.

Tu, J. V. (1996). Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes. *Journal of clinical epidemiology*, 49(11):1225–1231.

Van Damme, E. (2018). Protein chemistry.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Wang, G. and Dunbrack Jr, R. L. (2003). Pisces: a protein sequence culling server. *Bioinformatics*, 19(12):1589–1591.

Wang, S., Peng, J., Ma, J., and Xu, J. (2016). Protein secondary structure prediction using deep convolutional neural fields. *Scientific reports*, 6:18962.

Wider, G. and Wüthrich, K. (1999). Nmr spectroscopy of large molecules and multimolecular assemblies in solution. *Current opinion in structural biology*, 9(5):594–601.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Wüthrich, K. (1990). Protein structure determination in solution by nmr spectroscopy. *Journal of Biological Chemistry*, 265(36):22059–22062.

Yan, H., Deng, B., Li, X., and Qiu, X. (2019). Tener: Adapting transformer encoder for name entity recognition. *arXiv preprint arXiv:1911.04474*.

Yang, Y., Gao, J., Wang, J., Heffernan, R., Hanson, J., Paliwal, K., and Zhou, Y. (2016). Sixty-five years of the long march in protein secondary structure prediction: the final stretch? *Briefings in bioinformatics*, 19(3):482–494.

Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., and Lipson, H. (2015). Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*.

Yue, P., Li, Z., and Moult, J. (2005). Loss of protein structure stability as a major causative factor in monogenic disease. *Journal of molecular biology*, 353(2):459–473.

Zhang, B., Li, J., and Lü, Q. (2018a). Prediction of 8-state protein secondary structures by a novel deep learning architecture. *BMC bioinformatics*, 19(1):293.

Zhang, Q., Yuan, Q., Zeng, C., Li, X., and Wei, Y. (2018b). Missing data reconstruction in remote sensing image with a unified spatial–temporal–spectral deep convolutional neural network. *IEEE Transactions on Geoscience and Remote Sensing*, 56(8):4274–4288.

Zhang, X., Wei, F., and Zhou, M. (2019). Hibert: Document level pre-training of hierarchical bidirectional transformers for document summarization. *arXiv preprint arXiv:1905.06566*.

Zhang, Z. and Sabuncu, M. (2018). Generalized cross entropy loss for training deep neural networks with noisy labels. In *Advances in neural information processing systems*, pages 8778–8788.

Zhou, H. and Zhou, Y. (2005). Spem: improving multiple sequence alignment with sequence profiles and predicted secondary structures. *Bioinformatics*, 21(18):3615–3621.

Zhou, Y. and Karplus, M. (1999). Interpreting the folding kinetics of helical proteins. *Nature*, 401(6751):400.