

DIPLOMA PROJECT

**PROTEIN SECONDARY STRUCTURE PREDICTION**  
**USING BIDIRECTIONAL RECURRENT NEURAL**  
**NETWORKS AND**  
**HESSIAN FREE OPTIMIZATION**

KONSTANTINOS CHARALAMPOUS

UNIVERSITY OF CYPRUS



DEPARTMENT OF COMPUTER SCIENCE

MAY 2018

**UNIVERSITY OF CYPRUS**  
**DEPARTMENT OF COMPUTER SCIENCE**

**Protein Secondary Structure Prediction**  
**Using Bidirectional Recurrent Neural Networks**  
**And**  
**Hessian Free Optimization**

**Konstantinos Charalampous**

Supervisor  
Chris Christodoulou

The Diploma Project was submitted for partial completion of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

May 2018

## **Acknowledgments**

Firstly, I would like to express my sincere gratitude to my advisor Dr. Christodoulou Chris for the continuous support of my study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

I would also like to thank the assistant professor and head of the Bioinformatics Research Laboratory in the Department of Biological Sciences, Dr. Prompona Vasili, whose knowledge regarding the proteins part of the dissertation was crucial for the conduction of this dissertation.

Finally, I would like to thank the doctorate student Michali Agathokleous, as well as my fellow students Antrea Dionysiou and Panayioti Dimitriou for their constant support and help throughout the whole past year of conducting together this research.

## Summary

This dissertation focuses on the problem of protein secondary structure prediction, a problem that mainly concerns the fields of Computer Science and Biology.

Proteins are an integral part of the human body and every living organism. Studying the structure and functions of proteins facilitate the process of manufacturing food supplements, drugs and antibiotics to further evolve the quality of life and healthiness of people forward. The study of existing proteins is the key for treating diseases and solving a number of biological problems, especially nowadays when technology has made the process computationally easier, faster and significantly cheaper.

Despite the fact that for millions of proteins, the primary structure is well documented, only for a small fraction of those we know the secondary and tertiary structure. This is because the current state-of-the art methodologies and instruments for protein structure determination are extremely costly in terms of both money and time. This is incredibly serious, since the primary structure on its own, tells nothing about the actual function of the protein. This resulted in the emergence of a number of computational techniques and algorithms that attempt to predict the secondary and tertiary structure of a protein, given its primary, which do so significantly faster and cheaper.

For the purpose of this dissertation, a Bidirectional Recurrent Neural Network (BRNN) was implemented, trained with the Hessian Free Optimization to predict the secondary structures of proteins. The motivation of this project is to use HFO on a complex problem like PSSP, which was not done before, to test the theoretical superior performance in terms of execution times. The results of this network was an overall accuracy of 78.15% for a single fold with ensembles and 76.52 for 10-fold cross validation without ensembles, which is extremely promising, considering the current best methods and algorithms result in accuracies that are in between the 84-85% range (Wang et al., 2016).

## Περίληψη

Ο στόχος της παρούσας εργασίας είναι η προσπάθεια επίλυσης του προβλήματος πρόβλεψης της δευτεροταγούς δομής πρωτεϊνών, ένα πρόβλημα το οποίο αφορά κυρίως τους κλάδους της Πληροφορικής και Βιολογίας.

Οι πρωτεΐνες είναι ένα αναπόσπαστο κομμάτι του ανθρώπινου σώματος καθώς και κάθε ζωντανού οργανισμού. Μελετώντας την δομή και την λειτουργία των πρωτεϊνών, διευκολύνεται η διαδικασία ανάπτυξης διαφόρων συμπληρωμάτων διατροφής, φαρμάκων και αντιβιοτικών τα οποία μπορούν να εξελίξουν ραγδαία την ποιότητα ζωής και την γενικότερη υγεία των ανθρώπων. Η μελέτη των γνωστών πρωτεϊνών είναι το κλειδί για την εύρεση θεραπειών σοβαρών ασθενειών καθώς και για την επίλυση σημαντικών βιολογικών προβλημάτων. Αυτό ισχύει ειδικότερα στα σημερινά δεδομένα, όπου η τεχνολογία έχει καταφέρει να κάνει τις απαιτούμενες διαδικασίες υπολογιστικά ευκολότερες, γρηγορότερες και σημαντικά φθηνότερες.

Παρόλο το γεγονός ότι για εκατομμύρια πρωτεΐνες, η πρωτοταγής δομή τους είναι αρκετά ικανοποιητικά καταγεγραμμένες, μόνο για ένα πολύ μικρό κομμάτι από αυτές είναι γνωστή η δευτεροταγής και τριτοταγής δομή τους. Αυτό οφείλεται στο γεγονός ότι οι υπάρχουσες μεθοδολογίες και όργανα για εξακρίβωση της δομής των πρωτεϊνών είναι εξαιρετικά δαπανηρές, τόσο σε θέμα χρημάτων όσο και στο χρόνο που απαιτείται για να ολοκληρωθούν. Αυτό είναι πολύ σοβαρό πρόβλημα, διότι η πρωτοταγής δομή από μόνη της δεν παρέχει αρκετή πληροφορία για εξακρίβωση της λειτουργίας μια πρωτεΐνης. Αυτό είχε σαν αποτέλεσμα την εμφάνιση υπολογιστικών μεθοδολογιών και τεχνικών, οι οποίες προσπαθούν δεδομένου της πρωτοταγής δομής μια πρωτεΐνης, να προβλέψουν την δευτεροταγής της.

Στα πλαίσια της παρούσας Διπλωματικές εργασίας, υλοποιήθηκε ένα Νευρωνικό Δίκτυο αμφίδρομης ανάδρασης, εκπαιδευμένο με Hessian Free Optimization, με σκοπό την πρόβλεψη της δευτεροταγής δομής πρωτεϊνών. Ο σκοπός της έρευνας είναι να εφαρμόσουν τον αλγόριθμο HFO σε ένα πιο δύσκολο πρόβλημα όπως το PSSP, για να εξεταστεί η θεωρητική ανώτερη επίδοση του όσων αφορά τον χρόνο εκτέλεσης. Το αποτέλεσμα ήταν μια ολική ακρίβεια της τάξης του 78,15% για ένα fold με χρήση ensembles, και 76,52% με 10-fold cross validation,

χωρίς την χρήση ensembles το οποίο είναι εξαιρετικά υποσχόμενο, δεδομένου ότι οι καλύτερες μεθοδολογίες και αλγόριθμοι για το πρόβλημα κυμαίνονται γύρω στο 84-85%.

# Contents

<b>Chapter 1</b>	<b>Introduction.....</b>	<b>1</b>
1.1	The Importance and Purpose of PSSP	2
1.2	Previous work on PSSP	4
<hr/>		
<b>Chapter 2</b>	<b>Background Information.....</b>	<b>8</b>
2.1	Biology Background	8
2.1.1	The Biological Role of Proteins	8
2.1.2	Amino Acids	9
2.1.3	Protein Structure	14
2.1.3.1	Primary Structure	15
2.1.3.2	Secondary Structure	15
2.1.3.3	Tertiary and Quaternary Structure	16
2.2	Artificial Neural Networks (ANN) Background	18
2.2.1	ANN Origins	18
2.2.2	ANN Variants	20
<hr/>		
<b>Chapter 3</b>	<b>Dataset .....</b>	<b>36</b>
3.1	PSSP Metrics	37
3.2	Protein Databases and Dictionary of Secondary Structure of Proteins	38
3.3	Training/Testing Set and Cross Validation	39
3.4	Dataset Format	40
3.5	The CB513 Dataset	41
3.6	Data Encoding and Multiple Sequence Alignment profiles	42
3.7	Sliding Window	44
3.8	Ensembles	45
3.9	Filtering	46
<hr/>		

<b>Chapter 4</b>	<b>Methodology.....</b>	<b>49</b>
	4.1 Selecting a suitable ANN for PSSP	50
	4.2 Bidirectional Recurrent Neural Network (BRNN)	52
	4.3 Hessian Free Optimization (HFO)	54
	4.4 System Implementation	64

---

<b>Chapter 5</b>	<b>Results and Discussions.....</b>	<b>67</b>
	5.1 Feedforward Neural Network Experimentations	68
	5.2 Recurrent Neural Network Experimentations	73
	5.3 BRNN Experimentations	80
	5.4 Cross Validation, Filtering and Ensembles	87

---

<b>Chapter 6</b>	<b>Conclusion and Future Work.....</b>	<b>93</b>
	6.1 Conclusion	94
	6.2 Future Work	95

---

<b>References .....</b>	<b>96</b>
-------------------------	-----------

<b>Annex A .....</b>	<b>A-1</b>
----------------------	------------

<b>Annex B .....</b>	<b>B-1</b>
----------------------	------------

<b>Annex C .....</b>	<b>C-1</b>
----------------------	------------

<b>Annex D .....</b>	<b>D-1</b>
----------------------	------------

<b>Annex E .....</b>	<b>E-1</b>
----------------------	------------



# Chapter 1

## Introduction

---

1.1 The Importance and Purpose of PSSP	2
1.2 Previous work on PSSP	4

---

## **1.1. The Importance and Purpose of PSSP**

Proteins are an integral part of every living organism. In the human body, there are more than 30,000 unique proteins, which perform a vast array of important functions inside the cells. They are responsible for DNA replicating and defending against infections, as well as for many other functions required to sustain life.

They consist of organic compounds called amino acids connected to each other in long chains. Each protein differentiates from another in structure and in function, depending on the serial sequence of its amino acids. This is because the amino acids that make up a protein interact with each other, which causes the protein to fold into a specific three-dimensional structure. The structure is always the same for a specific protein, under certain conditions, and this is what determines its function.

Studying the structure and functions of proteins facilitate the process of manufacturing food supplements, drugs and antibiotics to further evolve the quality of life and healthiness of people forward. The study of existing proteins is the key for treating diseases and solving a number of biological problems, especially nowadays when technology has made the process computationally easier, faster and significantly cheaper.

In order to facilitate the process of studying proteins, a hierarchical approach has been established to better observe the structure of the proteins in the various phases of their formation. There are four layers of organization, which are the primary structure, the secondary structure, the tertiary structure and finally, the quaternary structure. The primary structure is the linear sequence of the amino acids, namely the order in which amino acids appear in the protein when unfolded. The secondary structure defines the way local segments of a protein are oriented in space, while the tertiary structure is the three-dimensional shape of a protein, when the amino acid chain is folded, and is the one that determines the specific function of a protein. Finally, a number of tertiary structures folding together forms a quaternary structure.

Despite the fact that for millions of proteins, the primary structure is well documented, only for a small fraction of those we know the secondary and tertiary structure. This is because the current state-of-the-art methodologies and instruments for protein structure determination are incredibly costly in terms of both money and time. This is incredibly serious, since the primary structure on its own, tells nothing about the actual function of the protein. This resulted in the emergence of a number of computational techniques and algorithms that attempt to predict the secondary and tertiary structure of a protein, given its primary, which do so significantly faster and cheaper.

One of those techniques used on this problem - PSSP (Protein Secondary Structure Prediction) is the use of Machine Learning algorithms. These algorithms are designed based on computational statistics and mathematical optimization techniques, which give computer systems the ability to learn patterns and idiosyncrasies of data, with the goal of being able to predict and classify new ones. There are a number of machine learning algorithms that have been used over time on this problem (which are discussed in the subsequent chapter); however, the focus of this dissertation is on Artificial Neural Networks (ANN). More specifically, a bidirectional recurrent neural network (BRNN), similar to the one used by Baldi (1999), trained with the Hessian Free Optimization (HFO) (Martens 2010) was developed and optimized on this problem.

The benefit of using BRNN for this problem is very clear. The way biological proteins fold in local segments (secondary structure) depends solely on the interactions and bonds that are formed by the neighboring amino acids. Consequently, a network, which is designed to take into account the amino acids located on either side (bidirectional) of the specific amino acid being classified, is bound to give a better prediction.

The original BRNN by Baldi was trained using the backpropagation algorithm (Werbos 1974) and resulted in extremely good predictions at the time. However, it is a relatively slow algorithm which suffers from problems like overfitting and

getting stuck in local minima. Using Second Order Optimization Algorithms, which are named so because they use second order derivatives (Hessian Matrix), additional information is calculated by the network, which heavily improves the optimization process in terms of both speed and accuracy. However, calculating and using the Hessian Matrix is often prohibiting due to its extremely large memory requirements. HFO addresses the memory issues by not calculating the Hessian Matrix (H) but the product  $Hu$  ( $u$  is an arbitrary vector), which is mathematically possible with a number of techniques and costs just as much as a gradient calculation. This, with combination of a number of other things, discussed on Chapter 4, made HFO computationally possible and accurate.

## 1.2 Previous Work on PSSP

There is more than half a century's worth of work on the PSSP problem. A number of machine learning algorithms have been developed and optimized for this specific problem over the years, which resulted recently in accuracies >90% (Shangxin et al. 2018, Magnan et al. 2014) in the Q3 accuracy score (Equation 2.1.) that essentially mark the problem solved. However, the algorithms that managed to achieve such high accuracies (>85%) have all used additional information and structural templates from databases, called sequence-based structural similarity of a protein. This makes the learning process and performance extremely better, relatively to the more pure machine learning algorithms. Without relying on these structural templates, the three-state accuracy is now at 82-84%, which is still good, considering the complexity of the problem, however there is still room for improvement, considering the theoretical limit of the three state prediction of around 88-90% (Rost, Burkhard 2001).

$$Q = 100 \frac{1}{n} \sum_{i=1}^n m_i$$

**Equation 1.1.:** Equation measuring the accuracy of protein secondary structure predictions, where  $n$  is the number of amino acid residues and  $m_i$  takes the value of 1 if the predicted value of the  $i^{\text{th}}$  amino acid residue is correct and 0 otherwise

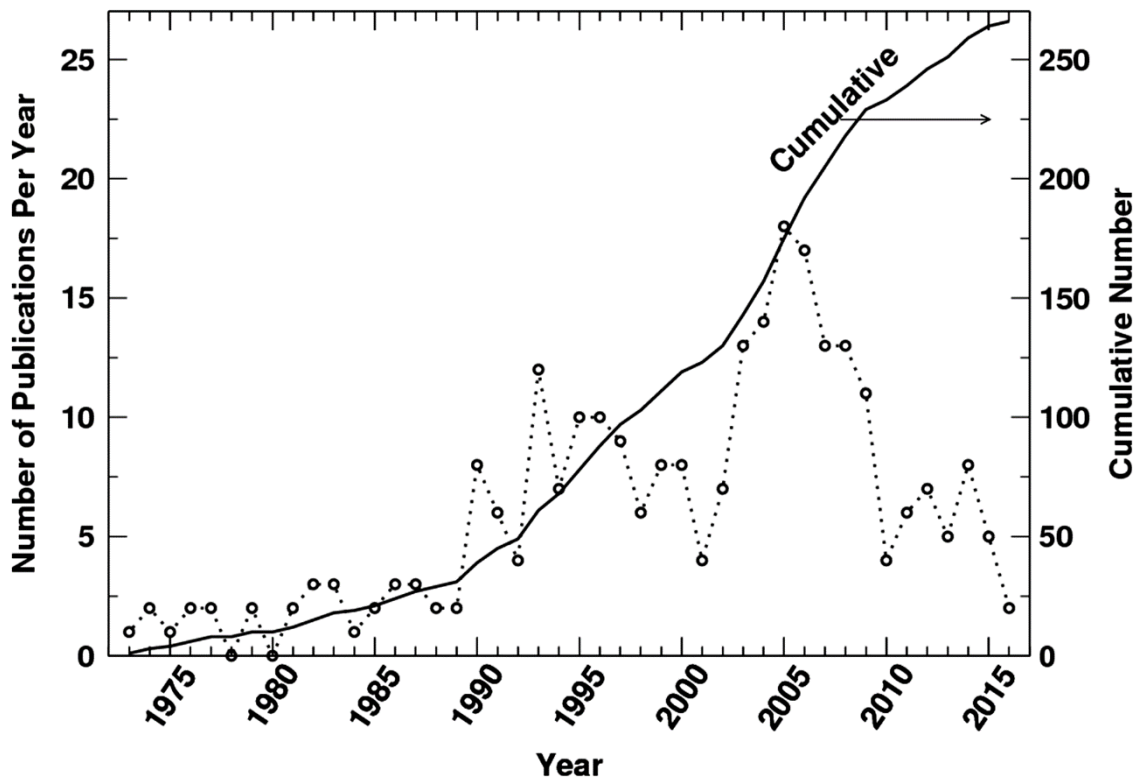


Figure 1.1: Number of publications for PSSP per year (Yang et al. 2016)

Observing the figure 1.1 it is clear that despite its long history it was only until the 90's that PSSP started getting more attention. That is because some major breakthroughs were achieved during that period which resulted in gradually increasing the three-state accuracy of the problem significantly.

While most methods were around the 60-63% Q3 before the 90's, in 1993 Rost and Sander (Rost & Sander, 1993) achieved accuracy of 69.7%. Their predictor was a fully connected feed forward Neural Network with early training stopping conditions and made use of averaging ensembles by training multiple models. In 1994, the same people released an automatic mail server for PSSP, called PHD (Rost et al., 1994) with accuracy of 71.4%.

Five years later, in 1999, two other predictors achieved accuracies of 76% and 76.5%. The first one was a Bidirectional Recurrent Neural Network trained with Backpropagation by Baldi (Baldi, 1999) and the second was PSIPRED by David

T.Jones (Jones, 1999) based on the position specific scoring matrices generated by PSI-BLAST.

In 2007, the Structural Property prediction with Integrated Neural Network by Ofer Dor and Yaoqi Zhou (Dor & Zhou, 2007) achieved 80%, using multiple sequence alignment (MSA), representative amino acid properties, a slow learning rate, overfitting protection, and an optimized sliding-window size.

In 2015, the Integrated Deep neural network 2 (SPIDER2) by Heffernan (Heffernan et al., 2015) achieved 82% using local backbone angles, solvent accessible surface area of proteins and iterative deep learning.

Finally, in 2016, the Deep Convolution Neural Field network (DeepCNF) by Wang SPeng JMa JZ (Wang et al., 2016) achieved the highest documented accuracy, without relying on the structural templates, of 84%.

# Chapter 2

## Background

---

2.1 Biology Background	
2.1.1 The Biological Role of Proteins	8
2.1.2 Amino Acids	9
2.1.3 Protein Structure	14
2.1.3.1 Primary Structure	15
2.1.3.2 Secondary Structure	15
2.1.3.3 Tertiary and Quaternary Structure	16
2.2 Artificial Neural Networks (ANN) Background	
2.2.1 ANN Origins	18
2.2.2 ANN Variants	20

---

## 2.1 Biology Background

### 2.1.1 The Biological Role of Proteins

Proteins are large, complex molecules made up of hundreds to thousands of smaller units called amino acids, which are attached to one another in long chains. Proteins are responsible for most of the functions within organisms and this is what classifies each protein into a specific type. For example, there are structural proteins, which strengthen cells, tissues and organs and defense proteins, namely the antibodies, which help organisms fight infection, heal damaged tissue and evade predators.

Table 2.1 lists the most important functions of proteins, which reflect the importance of proteins in nearly the entirety of an organism.

Type	Function Description	Example
<b>Enzyme</b>	Enzymes build and break down molecules. They are critical for growth, digestion, and many other processes in the cell. Without enzymes, chemical reactions would happen too slowly to sustain life.	Lactase
<b>Messenger</b>	Messenger proteins transmit signals to coordinate biological processes between different cells, tissues, and organs.	Growth Hormone
<b>Structural</b>	Structural proteins strengthen cells, tissues, organs, and more.	Collagen
<b>Transport</b>	Transport proteins move molecules and nutrients around the body and in and out of cells.	Hemoglobin
<b>Storage</b>	Storage proteins store nutrients and energy-rich molecules for later use.	Gluten
<b>Signaling</b>	Signaling proteins allow cells to communicate with each other.	Insulin
<b>Regulatory</b>	Regulatory proteins bind DNA to turn genes on and off.	Androgen, Estrogen
<b>Sensory</b>	Sensory proteins help us learn about our environment. They help us detect light, sound, touch, smell, taste, pain, and heat.	Opsin
<b>Motor</b>	Motor proteins keep cells moving and changing shape. They also transport components around inside cells.	Dynein, Kinesin
<b>Defense</b>	Defense proteins help organisms fight infection, heal damaged tissue, and evade predators.	Antibodies

**Table 2.1: Types of proteins and their function (<http://learn.genetics.utah.edu>, 2018, May 5)**



In the human body, proteins are created mostly through the consumption of foods. When food, which contains proteins, is consumed, the digestive system breaks it down into amino acids, which enter the blood stream. The cells then gather the necessary amino acids from the blood stream, to create the proteins it requires to perform any of the vast array of functions possible. A diet poor of proteins results in few amino acids entering the blood stream which weakens the immune system, causes exhaustion, dizziness and possibly a number of other very serious diseases. This is because the cells do not have enough amino acids to create the proteins required for each of the functions necessary to sustain the human body.


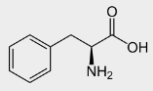
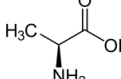
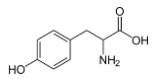
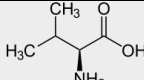
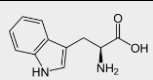
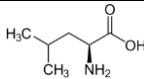
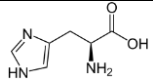
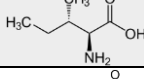
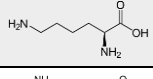
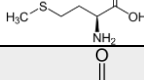
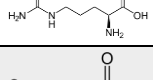
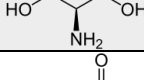
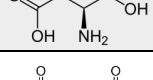
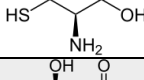
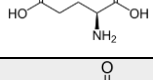
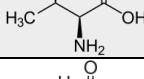
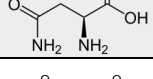
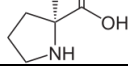
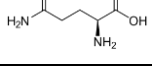
Consequently, understanding the significant role of proteins in all aspects of living organisms is important. However, what is necessary is to understand the core structure and function of each protein, in order to facilitate the process of creating food supplements, drugs and antibiotics to further evolve the quality of life and healthiness of people forward. The study of existing proteins, is the key for treating diseases and solving a number of biological problems, especially nowadays when technology has made the process computationally easier and significantly faster.

### **2.1.2 Amino Acids**

Amino acids, or as they are often called, the building blocks of life are the sole component of proteins. There are more than five hundred (500) naturally occurring amino acids known, but only twenty (20) appear in the genetic code and in the formation of proteins (Table 2.2). Consequently, those amino acids are called the essential amino acids and are found in most, but not all proteins.

All amino acids are composed by one functional group of amine ( $-NH_2$ ) and carboxyl ( $-COOH$ ), along with a side chain, the R group, specific to each amino acid. The unique side chain is what differentiates amino acids in their physical and chemical properties. Moreover, depending on the chemistry of their side chain, amino acids are classified into three (3) different categories. The first and

largest group of amino acids has nonpolar side chains, while the second has polar side chains, which are uncharged. The third one has amino acids with positive and negative charges on their side chain. This is extremely critical to the protein structure, since these side chains can interact and bond with one another based on their chemistry, which forms the specific part of the protein in a certain shape. This means that the sequence and location of amino acids in a particular protein determines where the bends and folds occur in its three-dimensional structure, (which is discussed later). Finally, every single amino acid has its amino group positively charged and its carboxylic group negatively charged. This facilitates the sequential connection between amino acids with covalent bonds.

	Amino Acid	Abbreviation		Structure		Amino Acid	Abbreviation		Structure
1.	Glycine	Gly	G		11.	Phenylalanine	Phe	F	
2.	Alanine	Ala	A		12.	Tyrosine	Tyr	Y	
3.	Valine	Val	V		13.	Tryptophan	Trp	W	
4.	Leucine	Leu	L		14.	Histidine	His	H	
5.	Isoleucine	Ile	I		15.	Lysine	Lys	K	
6.	Methionine	Met	M		16.	Arginine	Arg	R	
7.	Serine	Ser	S		17.	Aspartate	Asp	D	
8.	Cysteine	Cys	C		18.	Glutamate	Glu	E	
9.	Threonine	Thr	T		19.	Asparagine	Asn	N	
10.	Proline	Pro	P		20.	Glutamine	Gln	Q	

**Table 2.2: List of all the 20 essential amino acids (Hausman & Cooper, 2004).**

The way amino acids connect to each other is by peptide bonds, in units as small as two or three amino acids, called dipeptides and tripeptides respectively, or in much longer chains called polypeptides, forming a protein molecule. This process

is called condensation reaction and it extracts a water molecule as it joins the amino group of one amino acid and the carboxyl group of a neighboring amino acid. What remains of each amino acid after the junction, is called amino acid residue.

Figure 2.1 illustrates the core structure of all amino acids and the process of protein formation.

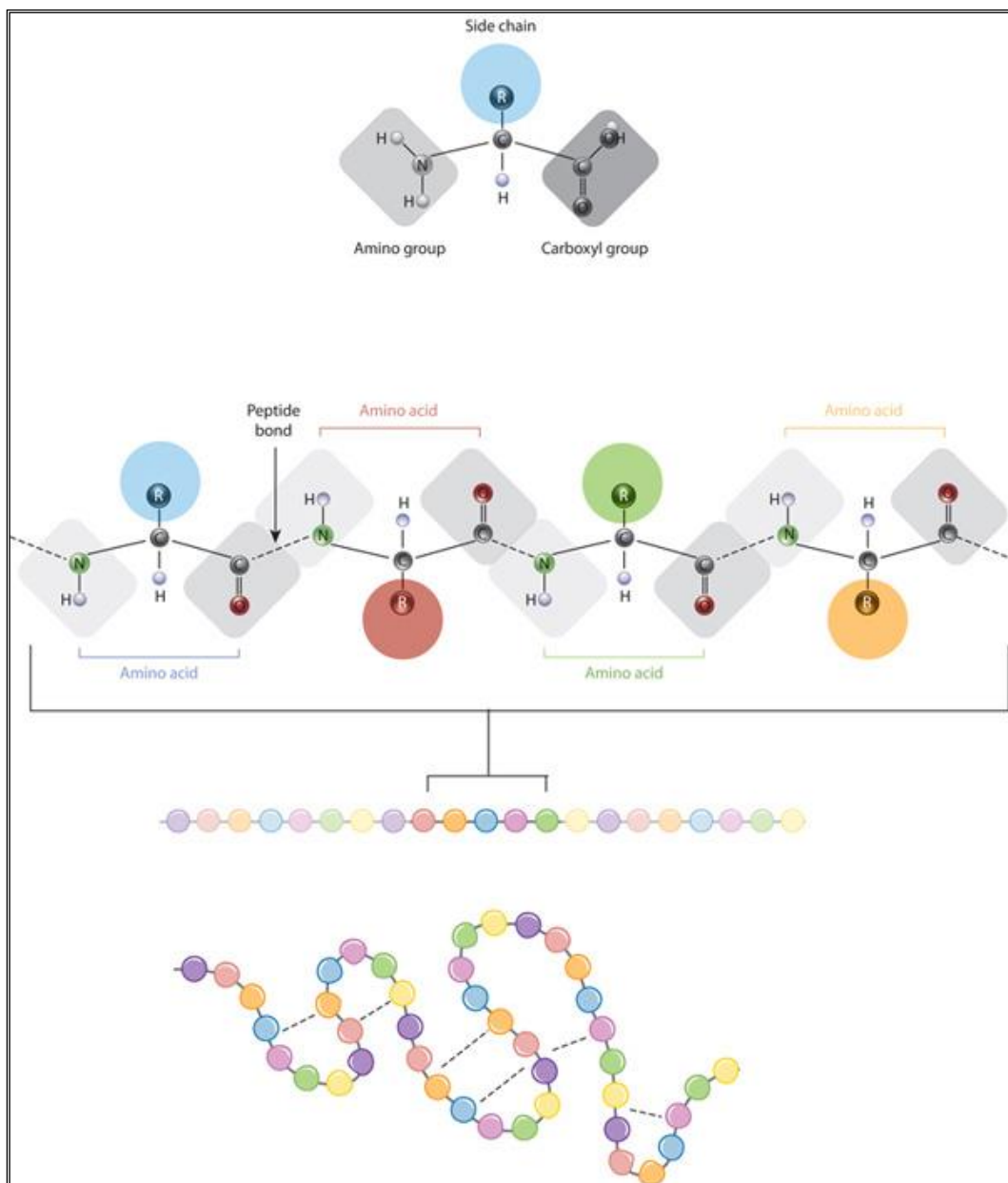
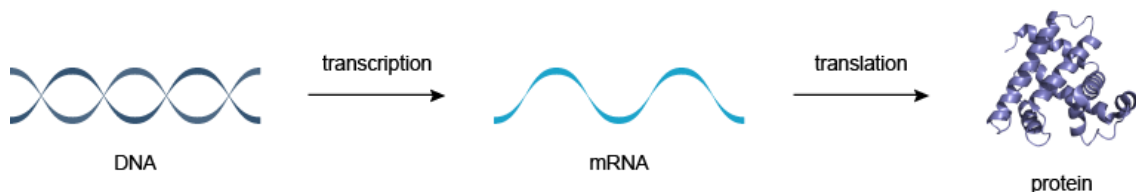


Figure 2.1: Amino acid structure and protein conformation (Nature Education 2010)

Each amino acid is abbreviated into a single (or triple) character from the English alphabet, meaning the amino acid sequence of a polypeptide can be represented as a sequence of characters. This sequence is considered to be the primary structure of the protein, which is discussed in detail later in section 2.1.3.1 As a result, any change in the sequence of the polypeptide, leads to the formation of a completely different protein, along with a completely different set of properties and functionalities.

The way each protein is assembled is encoded in the genes of an organism, the DNA. More specifically, the unique amino acid sequence, which forms a protein, is specified by the nucleotide sequence of the gene encoding that protein. In the case of the human genome, there are around thirty-thousand (30,000) genes, each of which encodes a single, unique protein.

The way it works is that the “DNA makes RNA” through a process called transcription and the “RNA makes proteins” through a process called translation. This constitutes The Central Dogma of Molecular Biology, which is illustrated in figure 2.2.



**Figure 2.2: The Central Dogma of Molecular Biology: DNA makes RNA makes proteins**  
(Nucleic Acids Book, [www.atdbio.com](http://www.atdbio.com), 2018, May 5)

The genetic code is basically a set of nucleotide triplets, called codons. Each combination of a triplet designates an amino acid, and since there are four (4) unique nucleotides (adenine - A, uracil - U, guanine - G, and cytosine -C), the total number of triplets that can be arranged is sixty-four ( $4^3 = 64$ ). However, there are only twenty (20) amino acids that can be encoded naturally, which means some amino acids can be described by more than one codon, or some codons do not encode any amino acids. Those codons, which do not encode any amino

acids, are called the stop codons and serve as a termination signal for the translation process, meaning that when one is found, the polypeptide, or the protein, translated up to that point is released. Figure 2.3 illustrates an example of the translation from DNA to protein (the first few amino acids for the alpha subunit of the protein hemoglobin), while figure 2.4 examines the full table of codons, along with the amino acid or the stop signal they encode.

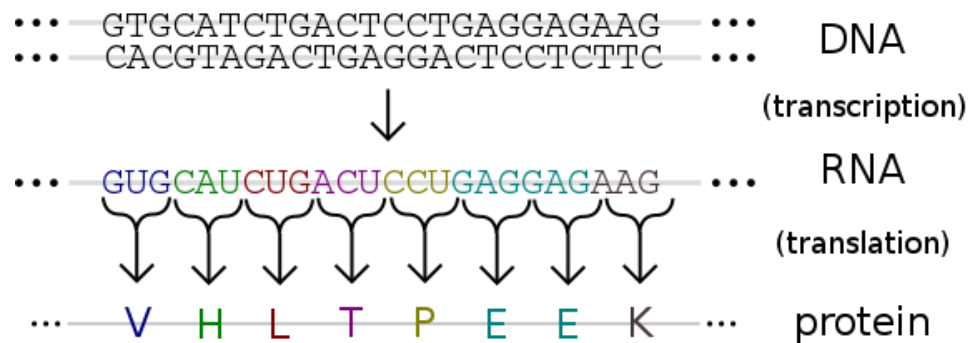


Figure 2.3: Example of the central dogma. The first few amino acids for the alpha subunit of hemoglobin (Madprime, 2006)

		Second nucleotide				
		U	C	A	G	
First nucleotide	U	UUU Phe UUC UUA Leu UUG	UCU UCC Ser UCA UCG	UAU Tyr UAC UAA STOP UAG STOP	UGU Cys UGC UGA STOP UGG Trp	U C A G
	C	CUU CUC Leu CUA CUG	CCU CCC Pro CCA CCG	CAU His CAC CAA Gln CAG	CGU CGC Arg CGA CGG	U C A G
	A	AUU Ile AUC AUA AUG Met	ACU ACC Thr ACA ACG	AAU Asn AAC AAA Lys AAG	AGU Ser AGC AGA Arg AGG	U C A G
	G	GUU GUC Val GUA GUG	GCU GCC Ala GCA GCG	GAU Asp GAC GAA Glu GAG	GGU GGC Gly GGA GGG	U C A G

Figure 2.4: The amino acids specified by each codon. (Nature Education 2014)

### 2.1.3 Protein Structure

In order to facilitate the process of studying proteins, a hierarchical approach has been established to better observe the structure of the proteins in the various phases of their formation, discussed previously in section 2.1.2. There are four layers of organization, which are the primary structure, the secondary structure, the tertiary structure and finally, the quaternary structure (Figure 2.5). It is important to note that this organization of many hierarchical structures is strictly used to make things easier for people to understand how proteins are formed. In organisms, proteins have one single structure, which is three-dimensional.

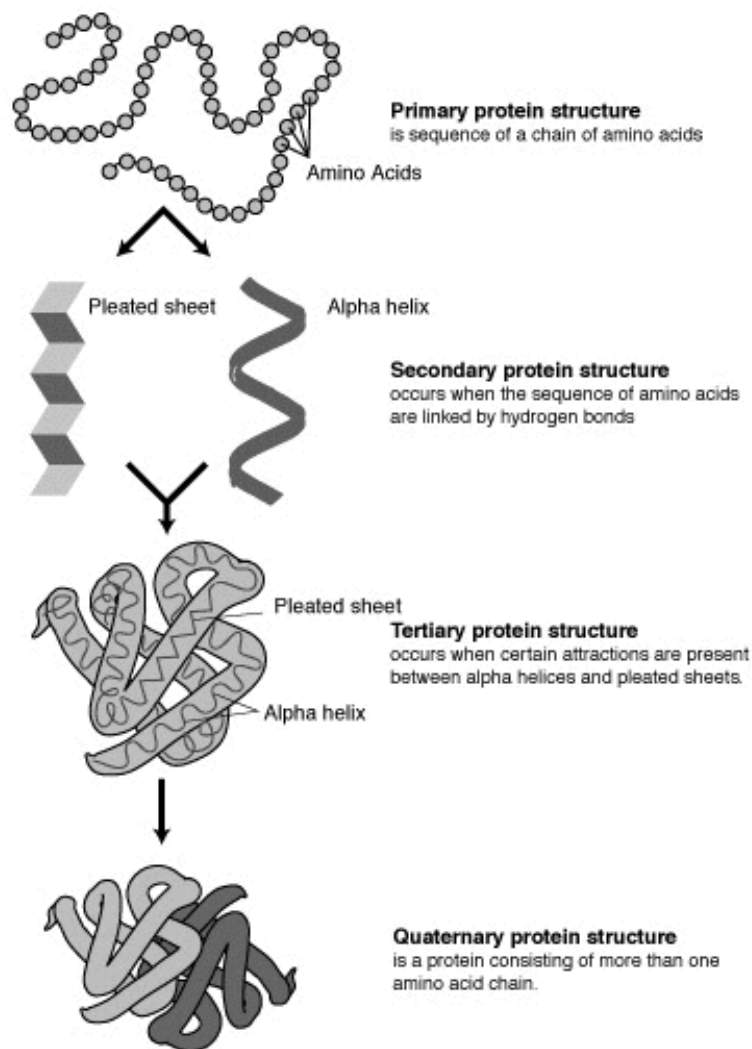


Figure 2.5: Layers of protein structure (Madison 2009)

### **2.1.3.1 Primary Structure**

The primary structure of a protein is the discrete sequence of amino acids, which is basically the linear succession of amino acids in the protein, if its three-dimensional structure was to be unfolded. Using the one-character amino acid abbreviations (Table 2.2), countless possibilities of protein formation exist. However, only a tiny subset of them has actually been studied and most of the information that exists today about proteins is about their primary structure. This is because the primary structure of a protein can easily be translated from the genetic material, though no useful information regarding its function can be extracted from it. However, various learning algorithms can be applied to it, to accurately predict cheaply its secondary and tertiary structure, which is also the main focus of this dissertation. The benefits of the secondary and tertiary structure are discussed subsequently, in sections 2.1.3.2 and 2.1.3.3 respectively

### **2.1.3.2 Secondary Structure**

The secondary structure is the three-dimensional form of local segments of proteins. The most common method of describing the secondary structure of proteins was defined by the Dictionary of Protein Secondary Structure, or DSSP (Kabsch et al., 1983) in short. Single character codes are used, based on hydrogen bond patterns, to define the eight (8) types of secondary structure that the DSSP classifies. These are the  $\alpha$ -helix (H), 3-helix (G),  $\pi$ -helix (I),  $\beta$ -strand (E),  $\beta$ -bridge (B),  $\beta$ -turn (T), bend (S), and random coil (C) for residues which are not in any of the other conformations. This last designation is unfortunate as no portion of protein three-dimensional structure is truly random and it is usually not a coil. A number of "other" secondary structures types have been proposed; however, they represent a small fraction of residues and may not be a general structural principle of proteins. It is common to group these eight (8) categories into three (3) to describe the nature of the shape of the specific local segment of the protein. First, the helix conformations that obviously contain the first three categories (H, G, I), and have helical form, the sheet conformations that contain

the  $\beta$ -strand (E) and  $\beta$ -bridge (B) categories, and finally Coil conformations which contain everything else.

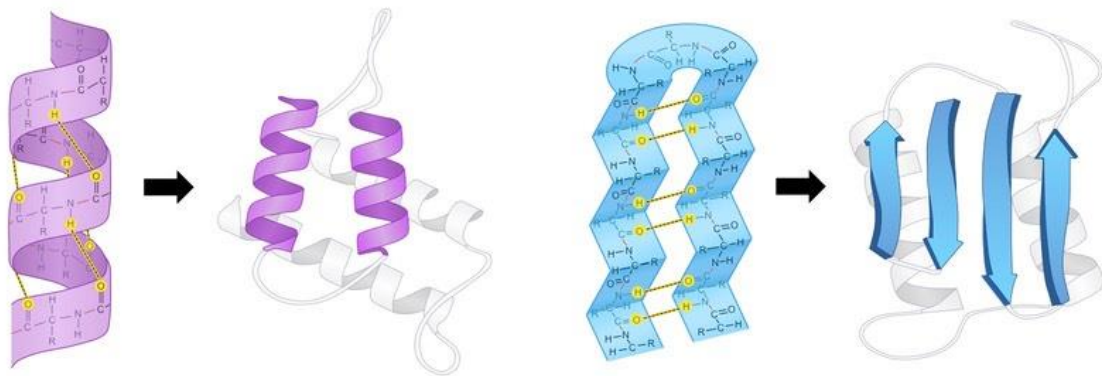


Figure 2.6: alpha helices form spirals (left) and beta-pleated sheets arrows (right) (<http://ib.bioninja.com.au>, 2018, May 5)

### 2.1.3.3 Tertiary and Quaternary Structure

The tertiary structure is the way the polypeptide chain coils and turns to form a complex molecular three-dimensional shape. This structure is what actually defines the functions and properties of the protein. Despite its great significance only for a very small portion of known proteins, there is a documented and fully defined tertiary structure. This is because of the very expensive experimental procedures required and it is still today a very important problem. Under certain conditions, such as protein temperature or pH change, the original three-dimensional structure is destroyed and its properties and biological functions are altered, despite of the fact that the amino acid sequence is still the same. This confirms that the 3D structure of the protein is what defines its function and not the amino acid sequence it is made up of. However, under normal conditions, both secondary and tertiary structures remain the same for each protein, since the linear sequence of amino acids (primary structure) is always the same and the following structures are developed through the interactions between the R groups of the amino acids.



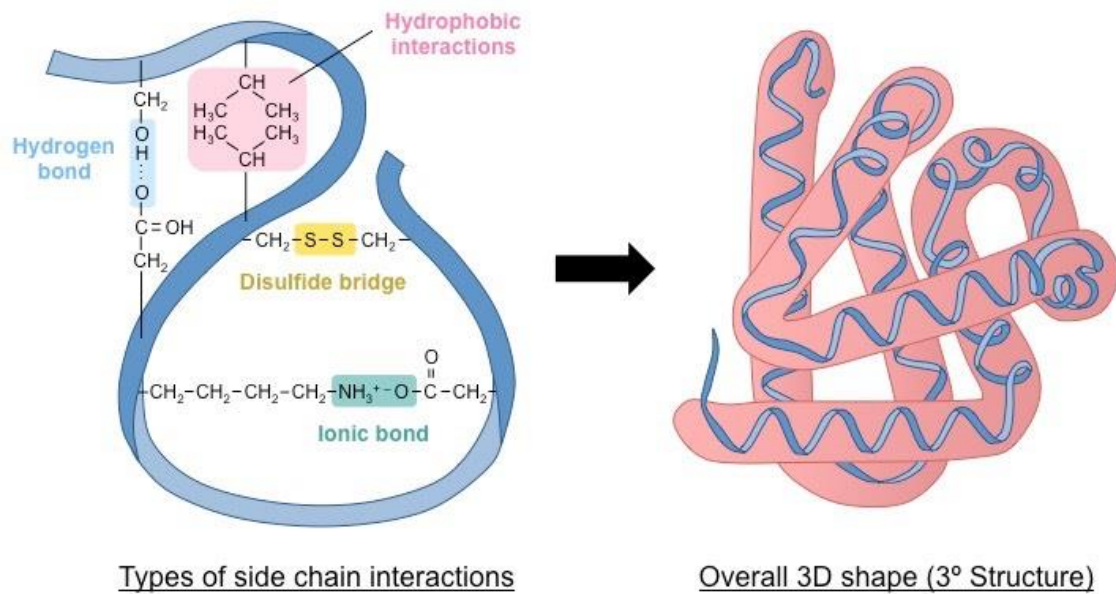


Figure 2.7: The formation of protein tertiary 3D structure (<http://ib.bioninja.com.au>, 2018, May 5)

The Quaternary Structure of a protein forms by multiple tertiary structures folding together.



Figure 2.8: The Quaternary Structure of a protein (McKinnon, 2003)

## **2.2 Artificial Neural Networks (ANN) Background**

### **2.2.1 Artificial Neural Networks (ANN) Origins**

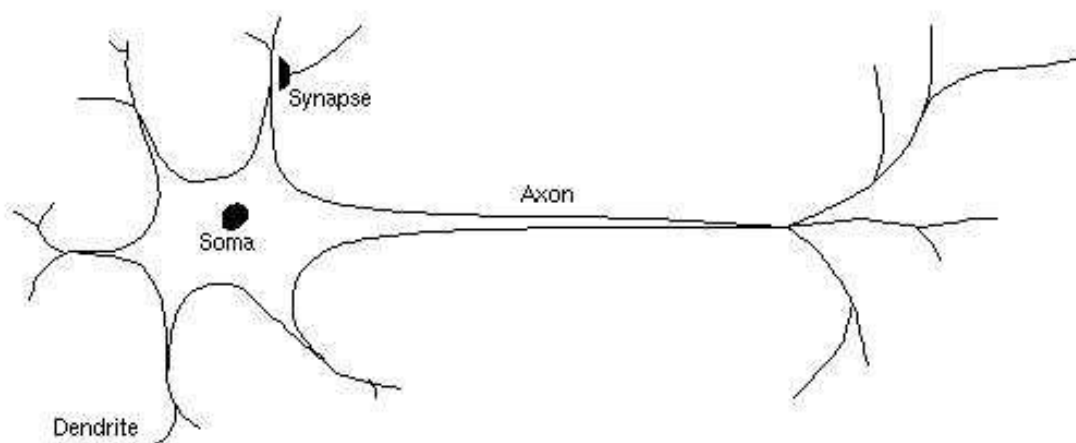
Artificial Neural Networks (ANN) are computing systems based on biological neural networks, which learn patterns and tasks without explicit programming. The term 'learn' varies based on the application it is used. However, a learning system could be summarized as a system that progressively improves its performance on a task given the metrics of the application. The learning is being accomplished by feeding and training the network on examples and data related to a specific task, with the goal of ultimately being able to identify the nature of newer, never before seen by the network data. For example, in financial applications, an ANN could be trained on historic stock market data with the goal of predicting future stock prices.

The popularity of ANN mainly resulted because of its theoretically extremely good properties. First of all, the Multilayer Perceptron (MLP), one of the most basic classes of ANN, is a Universal Function Approximator (Hornik et al., 1989). This essentially means that in theory ANN can reproduce all human intelligence and can solve any problem imaginable, with the assumption that everything can be reduced and modeled into a specific mathematical function. Moreover, they have the ability to extract useful information from inconsistent and noisy data and are able to generalize well from previous examples. They have minimal computational requirements when fully trained and due to their architecture, they can take advantage of parallelism, which significantly improves its training speed, comparing to the traditional serial computations. Finally, they are extremely suited to solve problems that are ill defined or problems that require enormous amount of processing.

Artificial Neural Networks were originally created to mimic and solve problems in the same way that a human brain would. This is reflected by the core architecture of an artificial neural network. In order to fully comprehend the similarities

between artificial and biological neural networks a brief explanation of how both networks are assembled and work follows.

The biological neural network is a collection of neurons that receive, process and transmit information between each other, through electrical and chemical signals via specialized connections called synapses. The ANN is essentially identical in terms of architecture, having nodes (artificial neurons), a simplified version of biological neurons in terms of functionalities, and edges, or connections instead of synapses, which transmit signals from one artificial neurons to another connected to it. A biological neuron consists of three (3) main components. The cell body (soma), the axons and the dendrites (Figure 2.9). Signals are transmitted through the axons and are received by dendrites, which in turn transmit it to the cell body. Finally, the cell body is responsible to process the aggregated signals, namely to add them together. Finally, providing that value exceeds a predefined threshold, the neuron fires another signal to some other connected neuron. Similarly, ANN work with the same concept, which is discussed in detail on the following chapters.



**Figure 2.9: Structure of a Biological Neuron (cs.stanford.edu, 2018, May 5)**

However, as ANN began gaining popularity, the attention was slowly shifted away from replicating the human brain and the biological neural networks. The need for matching and solving specific tasks, lead to the development of various ANN architectures, some of which deviate majorly from its initial biologically inspired

nature. For example, the need for image recognition, lead to the development of an alternative ANN called Convolutional Neural Network (CNN) (Lecun et al., 2015) which integrates a preprocessing module that is able to extract complex but useful features from images. These features are then fed into a classic ANN, a fully connected feedforward MLP (section 2.2.1), to classify and label the initial image. There are countless other variations of ANN to solve other specific tasks like speech recognition, machine translation and playing video games, some of which are discussed in the following section.

### 2.2.1 ANN Variants

#### McCulloch και Pitts (McP)

The foundation of all Artificial Neural Networks, proposed by Warren McCulloch and Walter Pitts in 1943 (McCulloch & Pitts, 1943), also known as a Binary Threshold Unit (Figure 2.10). The model aimed to replicate in its simplest form the structure and function of a single biological neuron of a neural network in the human brain. In the biological terms discussed earlier, an input vector takes the place of the 'dendrites', which feeds the artificial neuron the signals, by performing multiplications with the weight values. The artificial neuron then sums those signals and transmits the added value to a threshold function, the Step (or Heaviside) function (Figure 2.11). There, if the value exceeds a certain threshold value, it outputs an output signal of 1, otherwise an output signal of 0 (Equation 2.1) (similarly to biological neurons). Therefore, it can only be used for binary classification.

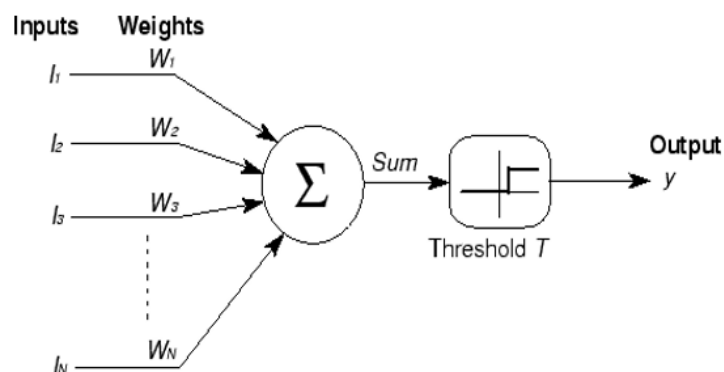
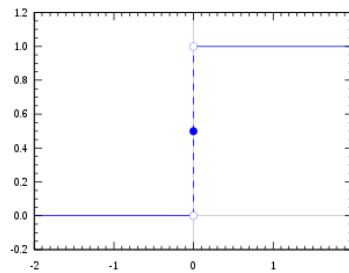


Figure 2.10: The artificial neuron of McCulloch and Pitts (1943)



**Figure 2.11.: The Step or Heaviside Function**

$$y = \begin{cases} 1 & \text{if } w \cdot x > s \\ 0 & \text{otherwise} \end{cases}$$

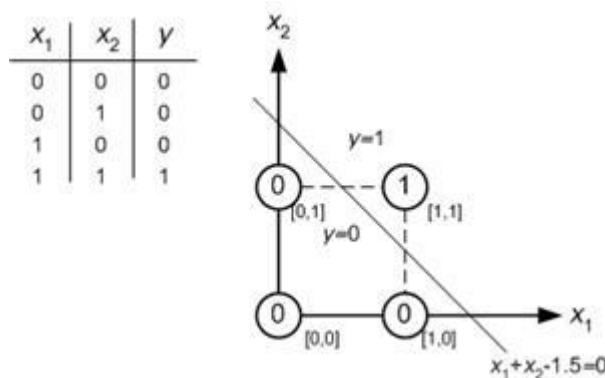
**Equation 2.1:** The output  $y$  of the network where  $x$  is the input vector,  $w$  the weight vector,  $w \cdot x$  the dot product and  $s$  the threshold

The way they classify inputs depends on the weights of the connections as well as on the threshold value. In a simple two-dimensional scenario (2D input vector), the decision line is that of the Equation 2.2.

$$x_2 = -\left(\frac{w_1}{w_2}\right)x_1 + \left(\frac{s}{w_2}\right)$$

**Equation 2.2:** Decision line of a 2D input vector

For example in the case of trying to classify the AND gate, the model would have to have weights of  $W = [1, 1]$  and threshold of  $S = 1.5$  (this is just an example, there are infinite other ways to solve this). The way the decision line would be is that of figure 2.12 and inputs would be classified depending on whether they are above the decision line (Class 1) or below (Class 0).



**Figure 2.12.: Decision Line of AND gate**

## Perceptron Learning Algorithm

The way McP neurons learn is through a learning algorithm called Perceptron (Rosenblatt 1957). The idea is to present input and desired output to the network, calculate the output for that input and in the case of misclassification (output is 1 but should be 0, or vice versa) adapt the weights accordingly (Algorithm 2.1). Although, this initially seemed promising, it was quickly proven that it could only solve linearly separable patterns, that is, patterns where a hyperplane can be found on space that can separate them. For example it could solve perfectly problems like the OR gate, since a straight line can separate the two classes. However, in problems like the XOR gate, where more than a straight line is needed to separate the classes, this algorithm failed (Figure 2.13). Moreover, there was no way to distinguish between outputs that are closer to the desired class due to the binary nature of the Heaviside function, which made the learning process more difficult, and impossible in more complicated scenarios of combining multiple perceptrons. This led to more sophisticated algorithms and networks like the Multilayer perceptron (MLP) and the backpropagation algorithm, which are discussed subsequently.

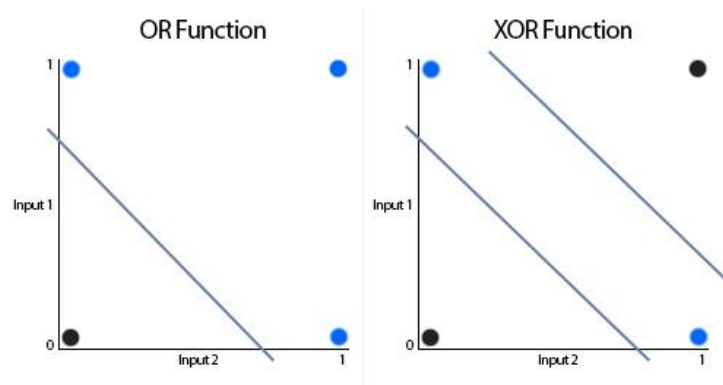
### Perceptron Learning Algorithm

1. Initialize weights and threshold randomly.
2. Present input and desired output.
3. Calculate actual output (Equation 2.1).
4. Adapt weights:

if output 0, should be 1:  $w_i(t+1) = w_i(t) + \eta \cdot x_i(t)$   
if output 1, should be 0:  $w_i(t+1) = w_i(t) - \eta \cdot x_i(t)$   
if output is correct :  $w_i(t+1) = w_i(t)$

where  $0 \leq \eta \leq 1$  the learning rate, controlling the adaptation rate.

**Algorithm 2.1: Perceptron Learning Algorithm**



**Figure 2.13: Linear Separability on OR gate vs Linear Inseparability on XOR Gate**

## **Multilayer Perceptron (MLP)**

Multilayer Perceptrons (MLP), or the ‘vanilla’ neural networks as they are often referred as, are the most popular and well-known variants of ANN. They are layered feedforwards networks, consisting of multiple (slightly different version of) McCulloch and Pitts neurons (Figure 2.14). The difference between true McP neurons and MLP neurons is that while the former strictly uses a threshold activation function (the Heaviside step function), the latter uses any arbitrary activation function (Table 2.3) This means that, while McP can only perform binary classification, MLP can either perform classification or regression, depending on the activation function used. In addition, activation functions serve as a way to distinguish between outputs that are closer to the desired class, which gives an indication of the scale by which to adjust the weights to have better predictions.

They have an input layer, an output layer and at least one hidden layer, with one or multiple neurons each. Each layer, apart from the output, has an independent ‘bias’ neuron unit, which basically helps fit the predictions better to the data, with a constant input value of 1. The hidden layers have multiple properties and it is where most of the information for the learning process is being stored and processed. Each neuron unit in the first hidden layer defines a new decision line that separates classes and patterns (Figure 2.13). Moreover, adding a second hidden layer leads to the formation of arbitrary complex decision shapes that are

capable of separating any classes. Consequently, no more than two hidden layers are needed in a network (Kolmogorov Theorem).

The process for calculating the output signal is feed forward and is similar to the one of McP. Neurons in the input layer, feed their values to the first hidden layer, where based on the activation function, they output a signal per hidden neuron (Equation 2.3). Those signals are in turn fed as inputs to the next hidden or output layer, where the process is repeated until no layer is left.

$$y = s(w^T x + b)$$

**Equation 2.3:** The output  $y$  of a single neuron where  $x$  is the input vector for that neuron,  $w$  the weight vector,  $w^T \cdot x$  the dot product,  $b$  the threshold and  $s$  the arbitrary activation function

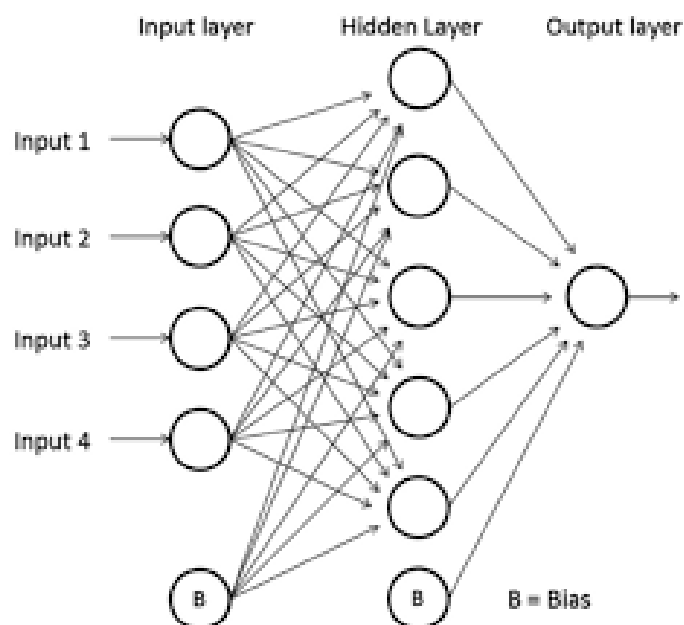


Figure 2.14 Multi-layer perceptron with 1 hidden layer



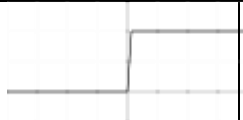
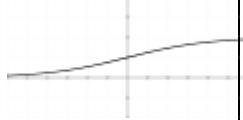
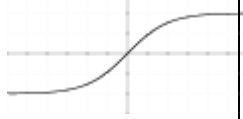

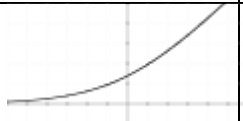

Name	Plot	Equation	Derivative	Range
Heaviside		$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ ? & \text{if } x = 0 \end{cases}$	$\{0,1\}$
Logistic / Sigmoid		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0,1)$
TanH		$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = \frac{1}{x^2 + 1}$	$(-1,1)$
Rectified linear unit (ReLU)		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$	$[0,\infty)$
SoftPlus		$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$	$(0,\infty)$
Gaussian		$f(x) = e^{-x^2}$	$f'(x) = -2xe^{-x^2}$	$(0,1)$

Table 2.3 : List of some of the most important activation functions.

## Gradient Descent

Gradient descent, also known as steepest descent, is a mathematical optimization algorithm for finding the minimum of a function. It is probably one of the most used algorithms in training Artificial Neural Networks. The way gradient descent works is, similarly to its name, given a point, it takes steps proportional to the negative of the gradient of the function at the current point.

An error signal is defined (Equation 2.4) which describes how well or how bad the network has managed to classify the input patterns. The goal is to minimize this function, so as many inputs as possible be correctly classified.

$$E = \frac{1}{2} \sum_j (t_{pj} - o_{pj})^2$$

**Equation 2.4: Mean Square Error (MSE) function, where t target output, o actual output, p denotes the pattern and j the neuron**

The main idea is to apply the Gradient Descent algorithm to the MSE function in order to minimize the error, namely the difference between the desired and actual outputs. What this means is to make a change in the weight vectors, proportional to the negative of the derivative of the error in the current pattern with respect to each weight.

$$\Delta w_{ij} = -n \frac{\partial E_p}{\partial w_{ij}}$$

**Equation 2.5: Weight change based on gradient descent where n learning rate**

### **Backpropagation (BP)**

Using the gradient descent method as is, the weights of the last hidden layer to the output layer are only possible to be adjusted. This is because, a desired output must be known in order to calculate (and minimize) the error, which is obviously not known in the hidden layers. Backpropagation addresses this issue by, as the name suggests, back-propagating the error from one layer, starting from the output layer, to the previous one. Two passes from all neurons are needed to achieve this. A forward pass, where given an input, the error is calculated (Equation 2.4). Then, at a second pass, the backward pass, the error is back propagated to the previous layers, adjusting the weights accordingly. This process is repeated until all patterns have been fed into the network enough times to minimize the error at a point where is small enough depending on the problem or enough epochs (the number of times all patterns have been fed into the network) have passed. (Algorithm 2.2).

### Backpropagation

```
Repeat:
  For each pattern :
    // Forward Pass
    Calculate the output
    // Backward Pass
    For each layer j, starting at the output:
      For each unit i:
        // Compute the error
        If output neuron:  $\delta_{ij} = y_{ij}(1 - y_{ij})(d_{ij} - y_{ij})$ 
        If hidden neuron:  $\delta_{ij} = y_{ij}(1 - y_{ij}) \sum \delta_{ik} \cdot W_{jk}$ 
        For each weight to this unit:
          Compute and apply  $\Delta w$ 
      Compute total error
    Increment epoch counter
Until small enough error or epoch counter exceeded
```

**Algorithm 2.2:** The Backpropagation algorithm, where  $d_{ij}$  is the error signal of neuron  $i$  of layer  $j$ ,  $y_{ij}$  is the actual output of neuron  $i$  of layer  $j$  and  $d_{ij}$  is the target output of neuron  $i$  of layer  $j$ . The  $\delta_{ik}$  is the  $\delta_{ij}$  but in the previous iteration of the algorithm

## Recurrent Neural Network (RNN)

Recurrent Neural Networks have the same structure as the Multilayer Perceptron Networks with one major difference. While MLP networks are feedforward, RNN have recurrent inputs, meaning the output of a hidden or output layer is fed back as input to itself or to another previous layer. The main idea behind this, is to create some sort of memory for the network which enables the output to be dependent not only from the current input, but also on a sequence of input data that were processed on previous iterations. Consequently, these ANN are used mainly on dynamic problems, namely time series predictions or predictions where the sequence of data is very important.

Some of the most popular RNN architectures are the one created by Jordan (1986) and the one created by Elman (1990) (Figure 2.15). The Jordan Network feeds its output to a context layer, which has connections to the hidden layer as well as back to itself, while the Elman Network feeds its hidden layer output to a context layer, which connects back to the hidden layer.

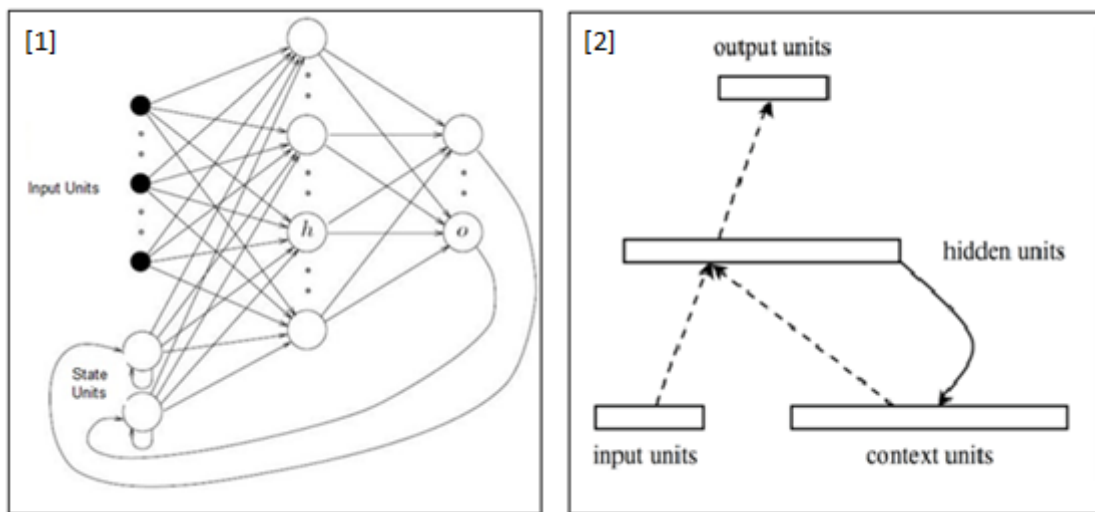


Figure 2.15: Jordan Network (Left), Elman Network (Right)

## Backpropagation through time (BPTT)

Recurrent neural networks share a lot of similarities with the Multilayer Perceptron Networks. However, the recurrent connections make it difficult for the standard BP to work on this architecture. As a result, Mozer in 1989 (Mozer, 1989), developed a technique to unfold the network in time to enable the standard BP to work (Figure 2.16). What this means, is that when a recurrent layer is found, create as many copies of it as time stamps (number of input-output pairs), each of which has the same parameters. Those layers have as inputs, recursively the output of the previous time stamp, as well as the new input data of the current time stamp. The BPTT algorithm by Werbos (1990) sums the errors of each time stamp until the end of the input window, and recursively moves back in the unfolded network, adjusting the weights.

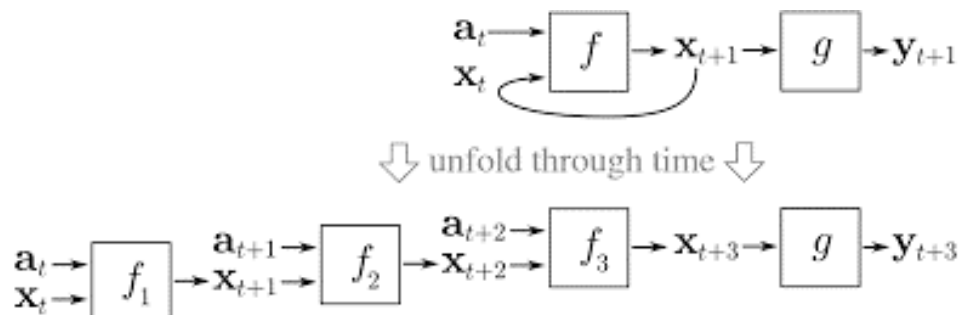


Figure 2.16: Unfolding RNN through time (Headlessplatter, 2010)

## Line Search

Line search is an iterative method, used to find a minimum  $x^*$  of an objective function (*in the case of ANN,  $x$  are the weights of the network and the error function is the objective function.*)

In the simplest terms, equation 2.6 shows the basic components in calculating the next iteration of  $\mathbf{x}$ , where  $\mathbf{d}$  is the search direction and  $\mathbf{a}$  is the step size, which determines how far  $\mathbf{x}$  should move along that direction. With simple gradient descent, the search direction is the negative gradient of the error function, and the step size an arbitrary learning rate. If the step size is too big, the objective function might move far away from the minimum. If it is too small, the updates get too small which can either make the optimization process significantly slower or force the objective into a local minimum. Consequently, it is very important to determine an optimal step size of each search direction at each iteration. As a result, line search tries to find the optimal step size, which minimizes an objective function in a specific search direction at each iteration.

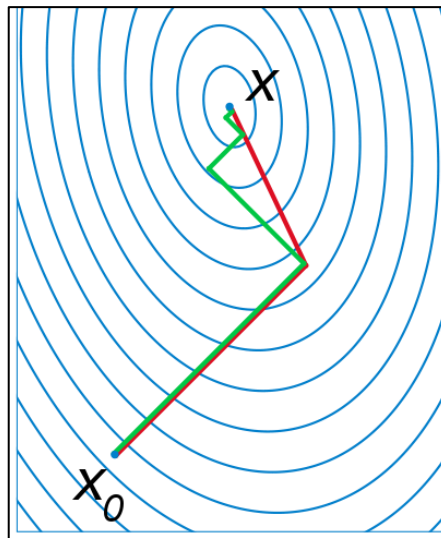
$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{a}_n \mathbf{d}_n$$

**Equation 2.6: Weights update, where  $\mathbf{a}$  is the step size and  $\mathbf{d}$  the search direction**

A naïve approach of finding the step size is to proceed along the search direction in small steps, evaluating the error function until it starts increasing (Hush and Salas, 1988). However, there are many variations of line search, much more efficient, robust and accurate (Press et al., 1992; Charalambous, 1992).

## Conjugate Gradient (CG)

In order to apply line search to optimize the step size and minimize the error function at each iteration, a descent search direction must first be determined. In the case of Gradient Descent, the search direction is the negative gradient of the error function at each new position. However, this is generally not a good choice of direction. Successive gradient directions, lead to the problem illustrated in figure 2.17 (green) in which the weights oscillate on successive steps while making little to no progress towards the minimum.



**Figure 2.17: Gradient Descent (Green) and Conjugate Gradient (Red) convergence with optimal step size (Alexandrov 2007)**

The Conjugate gradient algorithm addresses the problem by choosing directions in each iteration that do not interfere with each other or undo some of the progress made previously. More specifically, in an  $N$ -dimensional problem, CG guarantees a solution in  $N$  steps, with each step attaining the minimum in its direction. Figure 2.17 illustrates CG convergence on a 2-dimensional problem, in just two steps.

The algorithm, which describes how it works in detail follows:

### Conjugate Gradient

1. Initialize weight vector  $w_0$  randomly, set  $i=0$
2. Evaluate the gradient vector  $g_i$ , and set the initial search direction  $d_i = -g_i$
3. Use Line Search to find best step size  $a$ , which minimizes the function  $f(w_i + ad_i)$
4. Update weights  $w_{i+1} = w_i + ad_i$
5. Test stopping conditions
6. Evaluate new gradient vector  $g_{i+1}$
7. Evaluate new search direction  $d_{i+1} = -g_{i+1} + \beta_i d_i$ , where  $\beta_i$  is given by one of:

$$\beta_i = \frac{g_{i+1}^T (g_{i+1} - g_i)}{g_i^T g_i} \quad (\text{Polar and Ribiere})$$

$$\beta_i = \frac{g_{i+1}^T g_{i+1}}{g_i^T g_i} \quad (\text{Fletcher and Reeves})$$

8. Set  $i=i+1$  and go to step 3

**Algorithm 2.3: Conjugate Gradient Method (Bishop, 1995)**

## **Newtons's Method**

Newton's method is an iterative method originally created for finding approximations to the roots of real-valued functions. However, this method can also be used in optimization theory, to find a minimum or maximum of a function  $f(x)$ . The derivative of the function at these points is obviously zero, so the local minima and maxima can be found by applying Newton's Method to the derivative of the function to be optimized. In second-degree polynomials which are quadratic in nature, information of the second derivative of the function would be needed to work with, which essentially makes Newton's method a second-order optimization algorithm. This usage of second-order derivative results in a

significantly faster and more accurate convergence to the minimum, comparing to first-order optimization methods like gradient descent.

In a simple first-degree polynomial and 1-dimensional problem of a function  $f(x)$  and a sub-optimal initial solution  $x_0$ , Newton's method suggests the following:

1. Set  $x_i = x_0$
2. Find the equation of the tangent at  $x_i$
3. Find the point  $x_{i+1}$  at which the tangent line intersects with the x-axis
4. Find the projection of  $x_{i+1}$  on  $f(x)$
5. Set  $x_i = x_{i+1}$  and go to 2 until  $f(x_i) < \text{threshold}$

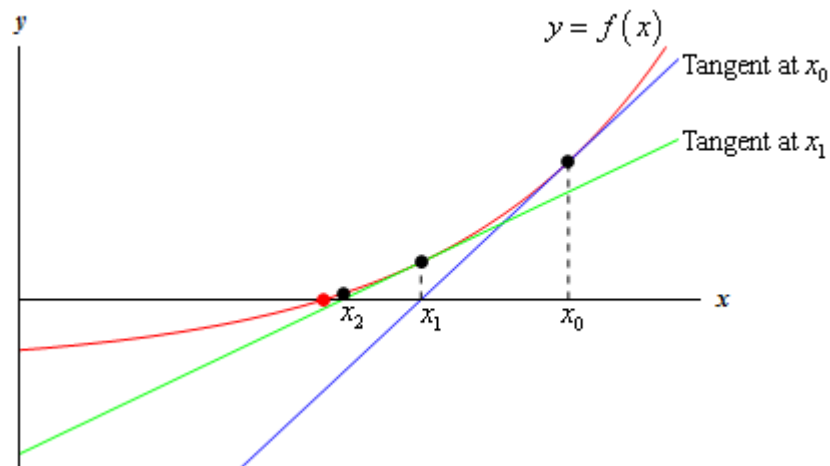


Figure 2.18: Newton's method in a first degree polynomial

The math behind this method is pretty simple. The equation of a point-slope line is

$$y - y_1 = m(x - x_1)$$

**Equation 2.7: The equation of a point-slope line**

Where  $m$  is the slope. This can be rewritten as

$$f(x) - f(x_1) = f'(x)(x - x_1)$$

**Equation 2.8: The equation of a point-slope line using derivative instead of slope**

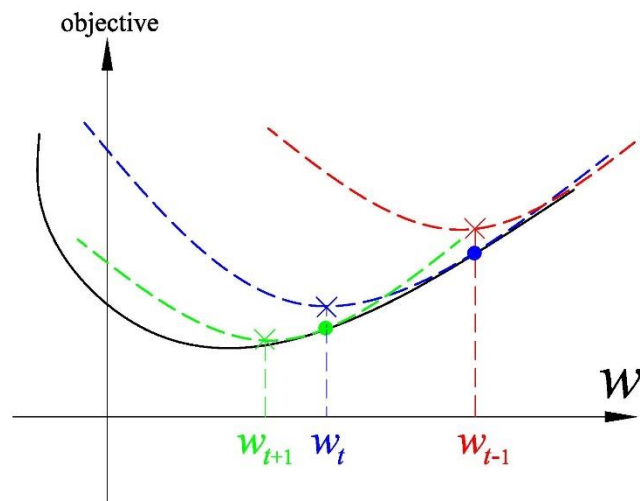


However,  $f(x_1) = 0$  (point of interaction at x-axis) which finally gives the update rule for  $x$  as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

**Equation 2.9: The update rule for optimizing the function**

However, this is just an illustrating example, used to gain the intuition behind the method of finding the roots of a function. What this method actually does in optimization theory is instead of using tangent lines at a current solution  $x$  as discussed earlier, it approximates the function  $f(x)$  by a local quadratic function around  $x$ , and take steps iteratively towards the minimum of that approximated function. This is repeated enough times to reach a certain threshold of the error or until a specific number of iterations have passed. Figure 2.19 illustrates the quadratic approximations around the weights at each iteration.



**Figure 2.19 Local Quadratic approximations (Rezamohammadighazi 2014)**

To approximate the function  $f(x)$ , the second-order Taylor expansion is being utilized.

$$f(x_0 + x) \approx f(x_0) + f'(x_0)x + f''(x_0)\frac{x^2}{2}$$

**Equation 2.10: The second series Taylor approximation**

Obviously, an optimal  $x$  needs to be chosen so the  $f(x_0 + x)$  is a minimum. In order to do that, Newton's method suggests to take the derivative of the Taylor series and set it equal to zero.

$$\frac{d(f(x_0) + f'(x_0)x + f''(x_0)\frac{x^2}{2})}{dx} = f'(x_0) + f''(x_0)x = 0 \Rightarrow x = -\frac{f'(x_0)}{f''(x_0)}$$

**Equation 2.11: The minimizer of Taylor's approximation**

Ideally, this  $x$  should have been the absolute minimum of  $f(x)$ . However, it is just the absolute minimum of the local approximation of  $f(x)$  around the initial solution of  $x_0$ . In order to get to the minimum of the objective function, we just repeat the process. This gives the final update rule for a 1-dimensional problem, which eventually converges to a minimum:

$$x_{n+1} = -\frac{f'(x_n)}{f''(x_n)}$$

**Equation 2.12: The update rule for optimizing the function  $f(x)$  for a 1D problem**

The problem is that this algorithm only works for objective functions with a single dimension ( $f: \mathbb{R} \rightarrow \mathbb{R}$ ).

If the objective function, has multiple dimensions ( $f: \mathbb{R}^n \rightarrow \mathbb{R}$ ), the algorithm is simply modified by replacing derivatives with gradients and second derivatives with Hessians (the matrix of second partial derivatives, figure 2.20)

$$x_{n+1} = -\frac{\nabla f(x_n)}{H(f)(x_n)}$$

**Equation 2.13: The final update rule for optimizing the function  $f(x)$  for multi-dimensional problem**

This is the final update rule, which is the one cited as the Newton's method.

$$H(e) = \begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 e}{\partial w_n \partial w_1} & \frac{\partial^2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix}$$

**Figure 2.20: The Hessian matrix of the error function in regards with the weights**

This method seems to be extremely efficient and fast computationally, since unlike Gradient Descent, it does not fit a plane (derivative) at a solution and move forward on that plane (using the learning rate / step size) but fits a quadratic approximation around that solution and directly finds the minimum of that curvature.

However, as the parameters of the function increase, it gets computationally impossible to calculate and store the entire hessian matrix of the function. This is why the standard Newton's method cannot be applied to Neural Networks where there are thousands to millions of parameters. However, with some modifications, a different variant of the algorithm can be derived which would make the Hessian calculation unnecessary and thus possible to apply it to ANN training (Hessian Free Optimization) (Martens 2010), discussed in section 4.2.

## Chapter 3

### Data Processing

---

3.1 PSSP Metrics	37
3.2 Protein Databases and Dictionary of Secondary Structure of Proteins	38
3.3 Training/Testing Set and Cross Validation	39
3.4 Dataset Format	40
3.5 The CB513 Dataset	41
3.6 Data Encoding and Multiple Sequence Alignment (MSA) profiles	42
3.7 Sliding Window	44
3.8 Ensembles	45
3.9 Filtering	46

---

### 3.1 PSSP Metrics

The focus of the PSSP problem is to predict accurately – to some extent – the secondary structure of a protein, given its primary. As a result, proteins with both structures known have been used to train and test the ANN used in this dissertation. As input, they have the primary structure of a protein encoded in some format, and as output, the predicted secondary structure, in a format, which is consistent to the actual secondary structure format of DSSP, discussed subsequently.

In order to measure the accuracy of the models trained, two (2) different metrics were used in this dissertation, which are the most common for the PSSP problem. First, the Q3 accuracy, which simply measures the number of correctly classified amino acids, divided by the number of total amino acids (Equation 1.1). However, this method does not measure how well each separate class is predicted and how good is the general structure of the complete predicted protein.

As a result, a different metric, the Segment Overlap (SOV) (Rost et al., 1994) score is also applied to address this problem. This method, instead of comparing each amino acid in a row, it compares segments of classes. For example, if in the correct secondary structure, there are four (4) helices, followed by two (2) coils and another four (4) helices but in the predicted structure there are simple ten (10) helices in a row the two metrics would produce significantly different accuracies. Indeed, the Q3 accuracy would be 80%, while the SOV score would be just 48. Note that SOV originally was not a percentage, since it could produce values outside of the 0-100 range. However, a modified definition of SOV (Zemla et al., 1999), fixed this problem using normalization techniques.

## 3.2 Protein Databases and Dictionary of Secondary Structure of Proteins (DSSP)

There are millions of documented proteins in various protein databases such as Protein Information Resource (iProClass), Protein Data bank in Europe (PDBe), Protein Data bank in Japan (PDBj) and RCSB Protein Data Bank. In those databases, information regarding protein names, length, structures (primary, secondary, tertiary and quaternary) exists, as well as many other biological information related to proteins. Those databases were used to extract protein information to create the datasets used in PSSP.

The Dictionary for Secondary Structure of Proteins (DSSP) (Kabsch et al., 1983) defined a standardized format of categorizing the secondary structures of a protein. In this format, there are eight (8) different classes of secondary structures, based on their shape and they are represented by a capital English letter. There are the  $\alpha$ -helix (H), 3-helix (G),  $\pi$ -helix (I),  $\beta$ -strand (E),  $\beta$ -bridge (B),  $\beta$ -turn (T), bend (S), and random coil (C) (table 3.1) for residues which are not in any of the other conformations. This last designation is unfortunate as no portion of protein three-dimensional structure is truly random and it is usually not a coil. A number of "other" secondary structures types have been proposed; however, they represent a small fraction of residues and may not be a general structural principle of proteins. It is common to group these eight (8) categories into three (3) to describe the nature of the shape of the specific local segment of the protein, which is the way they are categorized in this dissertation. First, the helix (H) conformations that obviously contain the first three categories (H, G, I), and have helical form, the sheet (E) conformations that contain the  $\beta$ -strand (E) and  $\beta$ -bridge (B) categories, and finally Coil (C) conformations which contain everything else.

Secondary Structure	8 class code	3 class code
$\alpha$ -helix	H	H
3-helix	G	
$\pi$ -helix	I	
$\beta$ -strand	E	E
$\beta$ -bridge	B	
$\beta$ -turn	T	C
bend	S	
Random coil	C	

**Table 3.1. Matrix with the abbreviations of the secondary structures grouped in 8 and 3 classes**

### 3.3 Training/Testing Set and Cross Validation

The way that learning algorithms work, and more specifically supervised methods like ANN, is they have a set of data, called the training dataset whose purpose is to train the model to recognize their patterns and identify each training example to a given class. This is achieved by a learning algorithm like those discussed on Section 2.2. In order to evaluate if the model has the ability to generalize those patterns and that it does not just learn them by heart, another set of data is used, called the test dataset, completely different from the training one, which simply measures how well the network has classified the new, never before seen data. Generally a good way to split a given dataset on training and testing sets, is by the 80-20 rule, meaning 80% of the total data are used for training while the remaining 20% for testing, but depending on the problem, different splitting percentages may produce better results.

However, most of the times, this is not good enough, since the data on a single test set may not give a good indication on how well the model generalizes new data. For this reason, a method called k-fold cross validation (Figure 3.1) is often used to address this issue. What this method suggests, is to split the data on k folds evenly, and train k different models with each model having a unique fold

as testing and the remaining k-1 folds as training. The average accuracy of each model is the cross validation accuracy.

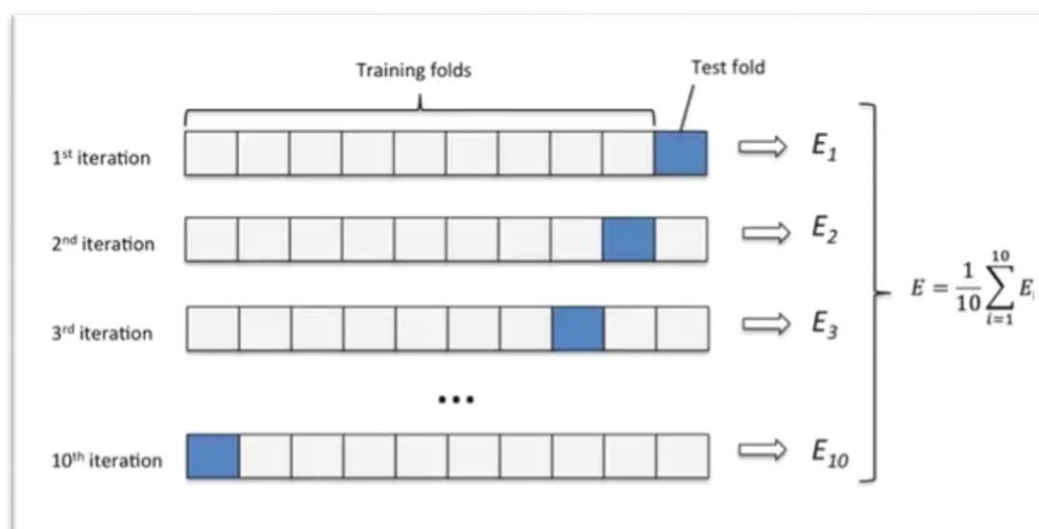
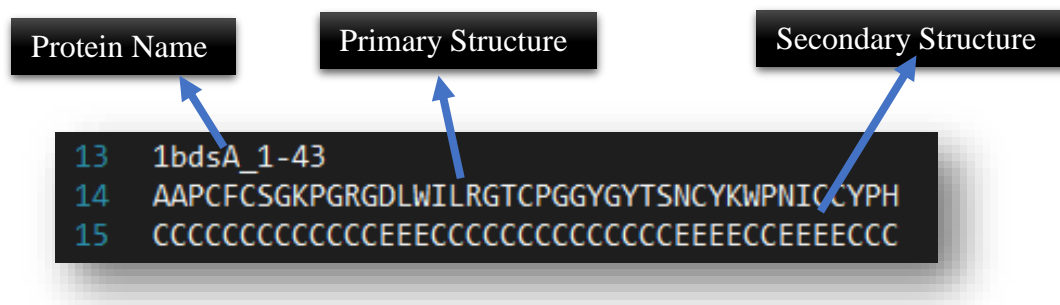


Figure 3.1: 10-fold cross validation with error/accuracy being the average of each iteration

### 3.4 Dataset Format

The datasets used in this dissertation for the purpose of training and validating ANN for the prediction of the secondary structure of proteins consist of three (3) lines per protein. The first line of each triplet has the protein name, which is useful in a later stage, for including additional information to the network beside the primary structure, using the Multiple Sequence Alignment (MSA) (Rost and Sander, 1993) profiles (discussed subsequently). The second line has the primary structure of the protein, which is essentially a sequence of amino acids, each of which is encoded in a single English character (Table 2.2) as discussed on Section 2.1. The final, third line has the correct secondary structure of the protein, which the model aims to predict.





**Figure 3.2: An example of a protein representation in the training set files**

Generally, in machine learning and ANNs, the performance of the predictive model heavily depends on the quality of the data being utilized. It is very critical to the success of the machine learning solution, to create a dataset, which is well selected and prepared. In order to ensure this, a process often called data selecting and data cleaning has to be performed. In the case of PSSP, there are many common datasets which have been created over the years, all of which have followed this process. In this dissertation, the dataset used is CB513 (Cuff and Barton 1999), which consists of 513 unique proteins. A very brief explanation of the process, which was required to create it, follows.

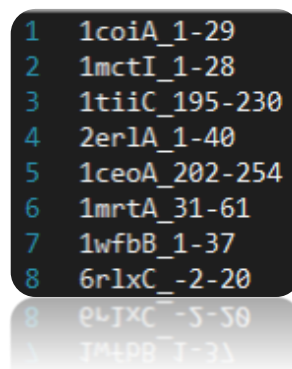
### 3.5 The CB513 Dataset

The origin of the CB513 (Cuff and Barton 1999) dataset was the dataset of Heinz-Uwe Hobohm (Pdb\_Select25, 2009) in 2009. This dataset originally contained 4019 proteins, with maximum similarity per protein pair of 25%. This is incredibly important in order to avoid a problem called selection bias, where the data sample is not truly random and there is no even representation of all classes of the problem. In selection bias, the trained model learns some classes better than others, which results in poor classification/prediction on patterns in the testing set, which belong to a poorly represented class on the training dataset.

From the initial 4019 proteins, only 513 finally remained. This is due to three main reasons. First, proteins had to be in the PDB database and be encoded in the

DSSP format. Second, the secondary structure of those proteins should have been determined by the X-Ray crystallography method or by the NMR (Nuclear magnetic resonance) method. Finally, there was some additional specific requirements, regarding both the structure of amino acids in a protein as well as the clarity of the structure determination by the X-Rays, Those conditions had to be set and followed, in order to create a dataset which would actually be useful for the PSSP problem without negatively influencing the classifications.

Finally, due to some problems in the MSA profiles (discussed subsequently), eight (8) distinct proteins (figure 3.3) had to be excluded from the training sets. Those proteins had MSA profiles with every value being 0, which would negatively affect the learning process, which was why it was decided for the purpose of this dissertation, to remove them altogether.



```
1 1coiA_1-29
2 1mctI_1-28
3 1tiiC_195-230
4 2er1A_1-40
5 1ceoA_202-254
6 1mrtA_31-61
7 1wfbB_1-37
8 6r1xC_-2-20
8 6r1xC_-5-50
1 1mrtA_31-61
```

Figure 3.3. A list of the name of the 8 excluded proteins

### 3.6 Data Encoding and Multiple Sequence Alignment (MSA) profiles

In ANN, and in most machine learning algorithms, the input and output of the model should be encoded and normalized in a real numbered value between zero and one (0-1) or between minus one and one (-1,1) depending on the range of the activation function used (Table 2.3). This is because all training examples in the dataset should be of equal importance. Having inputs with extremely high values, the network will learn and adjust to those examples in a way that it would be difficult for inputs with significantly smaller values to overcome. In the case of PSSP, the idea originally was to encode the input in twenty (20) artificial neurons,

with each neuron representing a unique amino acid (Table 2.2). The amino acid being examined at each iteration would have its neuron take the value of 1 while the rest the value of 0. This is called Orthogonal Encoding (Agathokleous 2009) (Figure 3.4). As intuitive as this method was, it did not give enough information to the network. As a result, a new method was suggested, which made use of Multiple Sequence Alignment (MSA) profiles.

A	10000000000000000000	M	00000000001000000000
C	01000000000000000000	N	00000000000100000000
D	00100000000000000000	P	00000000000010000000
E	00010000000000000000	Q	00000000000001000000
F	00001000000000000000	R	00000000000000100000
G	00000100000000000000	S	00000000000000010000
H	00000010000000000000	T	00000000000000001000
I	00000001000000000000	V	00000000000000000100
K	00000000100000000000	W	00000000000000000010
L	00000000010000000000	Y	00000000000000000001

Figure 3.4: Orthogonal encoding of amino acids

A lot of proteins have an evolutionary relationship with each other, by which they share a linkage and are descended from a common ancestor. Due to their evolutionary relationship, these proteins are supposed to have the same secondary and tertiary structure (Rost and Sander, 1993). As a result, MSA suggests aligning the amino acids of those proteins together, and encode in each position of their sequence the probability of each amino acid appearance. Figure 3.5 illustrates a simple example of the process.



Figure 3.5. Process of MSA profiling

In the case of adjusting those profiles for ANN use, a slightly more complicated encoding is needed. Figure 3.6 illustrates this in a simple matrix. In the first line, there are the 20 possible amino acids in a protein and in the first column the amino acid sequence of the specific protein being examined. Each line has the probabilities of which amino acid would appear in that specific position in the protein sequence. In the case of the example of the figure 3.5, the third from last line would have 0s in all 20 positions of the amino acids except from the V amino acid where it would have 89% (8/9) and the E amino acid 11% (1/9). Note that each line should add up to 100, and before feeding them into the network they should be divided by 100 in order to be in the 0-1 range. Using this encoding, each amino acid instead of having a single value of 1 in its 20 positions, it has multiple positive values summed up to 1 which should give a better indication to the network on how to predict its structure.

	V	L	I	M	F	W	Y	G	A	P	S	T	C	H	R	K	Q	E	N	D
N	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0	16	7	16	40	10
K	1	1	1	1	0	0	0	4	0	0	0	4	0	1	5	77	1	0	3	0
C	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0
P	1	1	0	0	1	0	2	22	5	48	4	2	1	7	1	2	0	1	1	1

Figure 3.6: MSA profile matrix (Christodoulou 2010)

In order to create these matrices, for each protein in the datasets, its corresponding encoding in MSA profiles was extracted from the HSSP (Homology-derived Secondary Structure of Proteins) Database.

### 3.7 Sliding Window

The structure of proteins in specific positions of the amino acid sequence, depends heavily on the bonds and interactions formed from neighboring amino acids. In order to capture this relation, instead of feeding into the network a single amino acid, a good idea is to feed multiple amino acids together. This is called a

sliding window. How this windows works, is essentially feeding into the network a fixed number (window size) of successive inputs concatenated, with the desired output varying depending on the application. For example, in time series prediction, the desired output of the window is usually the desired output of the immediate training example following the window. This is because, in time series problems, the output of a given time stamp depends solely on the previous time stamps.

However, in the case of PSSP, the secondary structure of an amino acid depends on the neighboring amino acids on either side of it. Consequently, in this case, the middle element of the window is the one being predicted and it is its desired output which is the desired output of the window. It is important to note that at each iteration the window slides by just one position and not by its whole size. So for example, in a given ANN trained with a window size of three (3), in the time stamp  $t$ , the inputs would be  $x_{(t-1)}$  to  $x_{(t+1)}$ . In the following iteration, the inputs would have been  $x_{(t)}$  to  $x_{(t+2)}$  and not  $x_{(t+2)}$  to  $x_{(t+5)}$ .

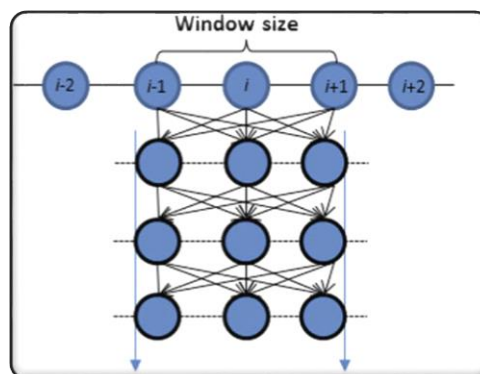


Figure 3.7: An example of an ANN trained with a window of size 3.(Wang et al. 2016)

### 3.8 Ensembles

In machine learning, a good way to improve the performance of your learning model is to make use of a method called ensemble learning. What this method suggests is instead of training just one model and get a single prediction, train multiple and combine in some way the results.

There are a number of ensemble methods, some of which are more advanced and complex than others. In this dissertation, a relatively simple and basic one was applied. It is called an averaging ensemble and essentially what it does is, as the name suggests, averages the outputs of its models. More specifically, in a scenario where there is a number of different models which were trained for the PSSP problem, the ensembling process works as follows. For each input/output pair, calculate the output for each model and classify it into one of the three classes available (H, E, and C). Using the 'winner takes all' method, take the results of each model, and the class with the most representations is the final class of the specific input. In the case of a tie, an arbitrary class of those participating in the tie is selected.

This way, random errors which might have occurred in some models are averaged out, which results in ultimately slightly better predictions, given the simplicity of this ensemble method. In more advanced ensembles, significant improvement may be achieved but they are usually computationally more expensive and time consuming.

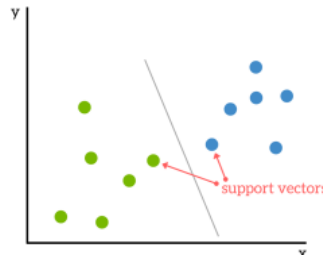
### **3.9 Filtering**

Another way to improve the performance of a predictive model is to apply post-processing filtering. The filtering can either be generic by applying another learning algorithm on the existing predictions or problem specific (Kountouris et al., 2012). In this dissertation, both methods were applied which resulted in slightly better results in the quality of the predictions (SOV score) or in the final raw accuracies (Q3 Score).

In the first case, different training and testing sets are being created based on the results of the original learning algorithm. These sets are basically the original sets, with the only difference being that instead of having as inputs the amino acids, they have the class of each amino acid which was predicted by the original model. Those sets are then used to train a separate learning model, which slightly

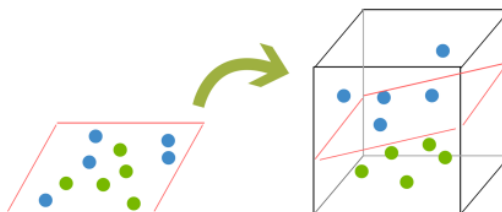
improves the results mostly in the Q3 accuracy. In this dissertation the learning model used to filter the data was the Support Vector Machines (SVM) (Cristianini et al., 2000).

The SVM algorithm, like Neural Networks, is a supervised algorithm, mainly used for classification. The way it works is essentially based on the idea of finding hyperplanes that best divide a dataset into classes, as shown in the image below.



**Figure 3.8. Dividing two linearly separable classes**

However, if the data are not linearly separable, as discussed in section 2.2, SVM tries to map the data into a higher dimension, using non-linear kernel functions that simply compute inner products, which is extremely cheap and effective. After this non-linear transformation into a higher dimension, the data are more likely to become linearly separable as illustrated in figure 3.9



**Figure 3.9. Transforming the feature vectors non-linearly to higher dimensions, results in the data becoming linearly separable**

In the second case, a specific set of external rules are being applied which are problem specific to the PSSP problem. Those rules were derived by empirical observations and mainly aim to fix the quality of the predictions instead of the overall accuracy.

More specifically, they include the following rules:

1. Single 'H' or 'E' are replaced with 'C'
2. Sequence 'HEEH' is replaced with 'HHHH'
3. Sequence 'HEH' is replaced with 'HHH'
4. Sequence 'HH' is replaced with 'CC'

Where H, E and C are the three predicting classes of table 3.1.

These rules are computationally extremely cheap since they consist of simple conditional statements, which improve the SOV score, while occasionally decreasing slightly the Q3 score.



## Chapter 4

### Methodology

4.1 Selecting a suitable ANN for PSSP	50
4.2 Bidirectional Recurrent Neural Network (BRNN)	52
4.3 Hessian Free Optimization (HFO)	54
4.4 System Implementation	64

## 4.1 Selecting a suitable ANN for PSSP

The way the PSSP problem in this dissertation is defined (Chapter 3) makes it a classification problem suitable for ANN training. There are a number of ANN architecture options, each of which has its benefits, which facilitate different types of classification problems. For example, for problems with just two linearly separable classes, a simple perceptron would be the best option for it, given its extreme simplicity, which results in a superior performance in terms of training time. However, as the complexity of the problem rises, more complex architectures and learning algorithms are required for effective predictions, such as MLPs and RNNs trained with their respective BP algorithms (Section 2.2).

The way biological proteins fold in local segments (secondary structure) depends solely on the interactions and bonds that are formed by the neighboring amino acids. A network, which is designed to take into account this information, namely to have as input not just the amino acid being classified, but also the amino acids located on either side of it, in a way that makes sense, is theoretically bound to give a better prediction. As a result, a different variation of ANN from those discussed in section 2.2 has been used which makes use of this information.

The way this new network is designed is essentially combining existing network architectures of feedforward MLP and recurrent ANN discussed earlier. More specifically, it is composed of three (3) separate modules/networks, two of which are recurrent networks with the other one being a simple feedforward MLP (Figure 4.1). The first recurrent network has as input the sequence of amino acids preceding the amino acid being classified while the second has the sequence of amino acids following it. This creates a bidirectional memory for the network, which facilitates the correlation of amino acids located on either side of the one being predicted, hence its name of Bidirectional Recurrent Neural Network (BRNN), originally proposed and designed by Baldi in 1999 (Baldi et al., 1999).



used to calculate the final output of the network, which is the final predicted secondary structure of the amino acid being examined (Equation 4.1).

$$Y_t = n(F_t, I_t, B_t)$$

**Equation 4.1.:** The output of the network, where  $n()$  is realized by the MLP,  $F_t$  is the output of the Forward RNN,  $I_t$  is the current input of the network and  $B_t$  is the output of the Backward RNN.

The BRNN has as input in time  $t$  a sequence (input window) of amino acids which is used to better classify the amino acid located in the center of this sequence. The time in this case is essentially the location of the amino acid in the primary structure of the protein. As a result, the time stamps of the network are limited to the 0-T range where T is the length of the primary structure of the protein. The input window has a fixed size for the entirety of the training process, which is always centered on the amino acid at location  $t$ .

The amino acids in the input window which precede and follow the amino acid in location  $t$ , are fed into two non-linear functions  $\varphi()$  (Equation 4.2) and  $\beta()$  (Equation 4.3), which are realized by the Forward and Backward RNN respectively. The forward RNN has as input the sequence of amino acids which is in the input window and precedes the amino acid  $t$  with a left-to-right order. Respectively, the backward RNN has as input the sequence of amino acids that is in the input window and follows the amino acid  $t$  with a right-to-left order. Being recurrent networks, they both have an additional input, which is their own output of the previous iteration. Consequently, these networks form some sort of a bidirectional memory for the network with which it can correlate the sequential relation between successive amino acids.

The way the data are fed into each recurrent network can either be one by one amino acid, or as a sequence of amino acids which has a fixed length, smaller than the number of amino acids that precede or follow the centered amino acid in the input window. In the edge cases, namely for  $t < 0$  or  $t > (\text{total number of}$

amino acids) a padding of zero-valued vectors are used for each position outside of the allowed range.

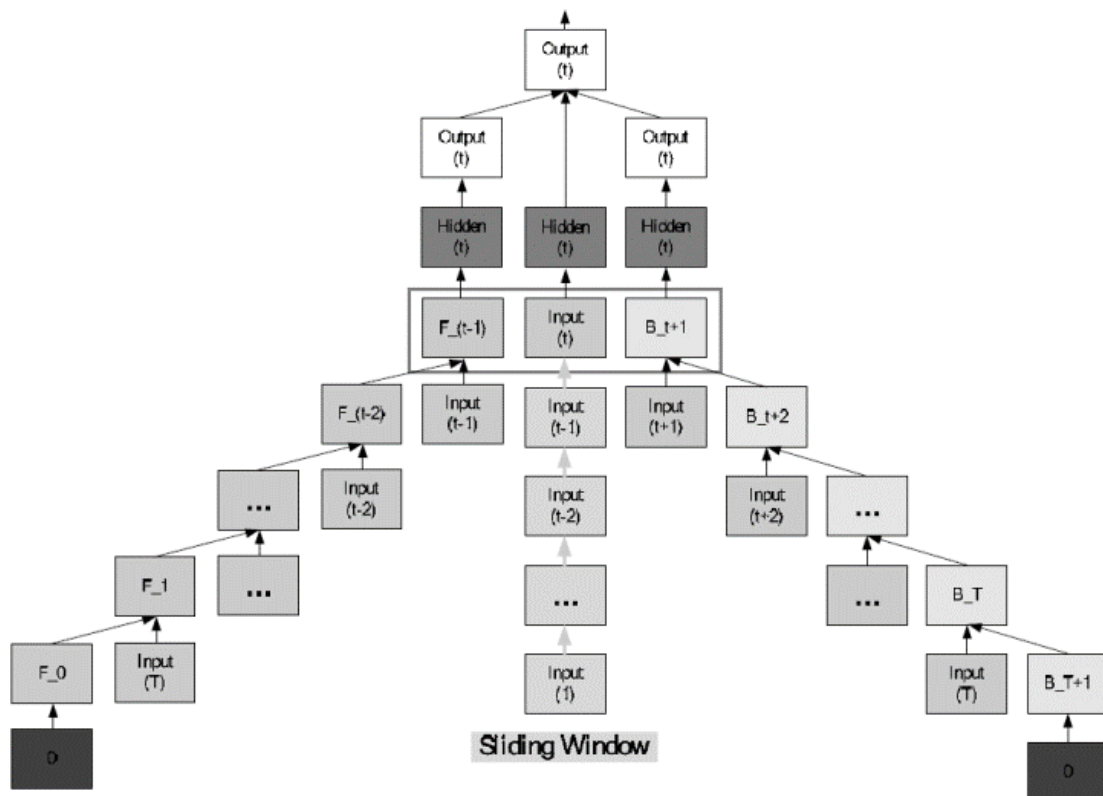
The unfolded network in time, which illustrates the successive inputs to the network, is presented in figure 4.2.

$$F_t = \varphi(F_{t-1}, U_t)$$

**Equation 4.2:** The output of the forward RNN, which is a non-linear function of its previous output ( $F_{t-1}$ ) and its current input, encoded by the unit  $U_t$  of figure 4.1.

$$B_t = \beta(B_{t+1}, U_t)$$

**Equation 4.3:** The output of the backward RNN, which is a non-linear function of its previous output ( $B_{t+1}$ ) and its current input, encoded by the unit  $U_t$  of figure 4.1.



**Figure 4.2.** The BRNN unfolded in time (Agathokleous, 2009)

## 4.3 Hessian Free Optimization (HFO)

### 4.3.1 Introduction to HFO

Hessian Free Optimization (HFO) (Martens, 2010) is a second order optimization algorithm of real-valued objective functions. It is a variation of the standard Newton's method, discussed in section 2.2, which uses local quadratic approximations to generate update proposals. As mentioned in that section, on problems with high dimensionality, namely large neural networks with many hidden layers, first order optimization algorithms like Gradient Descent can be extremely slow and ineffective. This is due to a problem called the Vanishing Gradient. In Gradient Descent, the updates are proportional to the gradient of the error function back propagated through the layers. Each time it is backpropagated, the gradient decreases, meaning that for many-layered or recurrent ANNs, the gradient becomes vanishingly small which results in the front layers having close to zero information on how to update their weights, meaning slow to completely ineffective training.

The advantage of using a second order optimization algorithm like Newton's method or HFO is that these algorithms consider the curvature of the error surface (Hessian Matrix) in their optimization process which results in extremely better step-wise performance. More specifically, instead of fitting a plane at an initial solution and then determining the step-wise jump like first order algorithms, second order methods find a tightly fitting quadratic curve at that point and directly find the minimum of that curvature, which is supremely fast and efficient.

However, computing the Hessian Matrix for a large ANN with thousands to millions of parameters is not always possible due to the extremely high memory requirements needed to store it. This is why, while there have been a number of Newton's variations like Newton-CG, CG-Steihaug, Newton-Lanczos (Nash, 1984), and Truncated Newton (Nash 2000), none of them have been applied effectively to machine learning and neural networks, or their applications have been extremely limited (Martens and Sutskever, 2012).

The Hessian Free method, however proposes solutions to these memory requirements, which enable it to be effective for Neural Network training. First of all, it does not compute and store the whole Hessian Matrix ( $H$ ), but instead just the dot product of it with an arbitrary vector ( $u$ ) ( $Hu$ ), using mathematical methods like finite differences which cost as much as a single gradient evaluation. This works really well for HFO since it does not require explicit use of the Hessian but rather many dot products with it and arbitrary vectors. Secondly, the local quadratic objectives, which second-order methods approximate, can be efficiently optimized using the linear conjugate gradient (CG), discussed in section 2.2 in order to compensate for the lack of the Hessian Matrix needed in Newton's method. While the CG method needs  $N$  iterations to converge, where  $N$  is the number of parameters of the network, there are a number of stopping criteria, which terminate it at early stages when significant progress in the minimization process has been made. This is extremely important since it is clearly impractical to wait for a complete CG convergence when there is a very low margin of further minimization.

It is important to note that even though in HFO no Hessian Matrix is calculated there are no approximations done and the  $Hu$  product is calculated accurately. The only difference between HFO and Newton's method is that while standard Newton's method performs a complete optimization to the approximated quadratic, HFO does not via the un-converged CG discussed earlier (Martens, 2010). However, the efficiency related benefits of avoiding a full Hessian Matrix calculation and inversion are clear and more than make up for the extremely small difference in accuracy by the not fully converged CG.

Finally, although the  $Hu$  product can be calculated efficiently and accurately, it is not the one usually used in HFO. Instead, the  $Gu$  product is used, where  $G$  is the Gauss-Newton matrix, an approximation of the Hessian Matrix (Schraudolph, 2002). While it seems pointless to use an approximation instead of the correct curvature matrix when there is no problem in efficiency, Gauss-Newton avoids some of the problems that the Hessian may face, which cause the algorithm to

be completely ineffective. In fact, even when those problems do not occur, the use of the G matrix consistently results in better search directions utilizing half the memory and running twice as fast, comparing to the usage of the Hessian matrix (Martens, 2010)

### 4.3.2 Detailed Analysis of HFO

The HFO method is a minimization algorithm of a twice-differentiable objective function  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  with regards to a vector of parameters  $w \in \mathbb{R}^n$ . Like Newton's method, it is based on the idea of iteratively optimizing a sequence of local quadratic approximations of the objective function in order to produce updates to  $w$ .

In the simplest situation, given the previous parameters  $w_{t-1}$ , iteration  $t$  produces a new  $w_t$  by minimizing a local quadratic model  $M_{t-1}(\delta)$  of the objective  $f(w_{t-1} + \delta)$ , which is formed using gradient and curvature information local to  $w_{t-1}$  (Equation 4.4)

$$f(w_{t-1} + \delta) \cong M_{t-1}(\delta) = f(w_{t-1}) + \nabla f(w_{t-1})^T \delta + \frac{1}{2} \delta^T B_{t-1} \delta$$

**Equation 4.4: The local quadratic approximation of the objective  $f(w_{t-1} + \delta)$  where  $B_{t-1}$  is the curvature matrix, which is usually the Hessian (H)**

Minimizing the quadratic means that an optimal search direction  $\delta^*$  is found with which the new update is calculated based on Equation 4.5

$$w_t = w_{t-1} + \alpha \delta_t^*$$

**Equation 4.5: The new weight update, where  $\delta_t^*$  is the minimizer of the quadratic of equation 4.4 and  $\alpha \in [0, 1]$  is the step size, calculated by line search, discussed in section 2.2.**

Solving the system in equation 4.4 in order to find the minimizer  $\delta^*$  like proposed by the standard Newton's method is computationally impractical and for some



networks even impossible if you consider the complexity it requires of  $O(n^3)$  (Martens and Sutskever, 2012). In order to avoid this, the linear Conjugate Gradient (CG) (section 2.2) is being used which partially optimizes the quadratic  $M$ . The resulting approximate minimizer  $\delta^*$  is then used to update the weights  $w$  (Equation 4.5).

## Conjugate Gradient for HFO

Conjugate Gradient is a specialized optimizer created specifically for quadratic objectives of the form  $q(x) = \frac{1}{2}x^T Ax - b^T x$  where  $A \in \mathbb{R}^{n \times n}$  is positive definite ( $x^T Ax > 0 \ \forall$  non zero column vector  $x$ ) and  $b \in \mathbb{R}^n$ . To apply CG to equation 4.4 of the quadratic model, we take  $x = \delta$ ,  $A = B_{t-1}$  and  $b = \nabla f(w_{t-1})$ , noting that the constant term  $f(w_{t-1})$  can be ignored.

Conjugate Gradient in the worst case converges in  $N$  steps, however depending on the structure of the curvature matrix  $B$ , it often converges in significantly less iterations and even if it does not converge, it tends to make very good partial progress (Martens and Sutskever, 2012). In fact, there is a method called preconditioning, which accelerates the CG convergence by transforming the coordinate system using a preconditioning matrix  $P$ . The CG algorithm using preconditioning is described in algorithm 4.1.

In order for CG to terminate optimally, there are a number of stopping criteria, which balance the quality of the solution with the number of iterations required to obtain it. Martens proposed an approach, which measures the relative progress of optimizing  $M$ , computed as of equation 4.6 (Martens and Sutskever, 2012).

$$s_j = \frac{M(x_j) - M(x_{j-k})}{M(x_j)}$$

**Equation 4.6: The measurement of progress suggested by Martens, where  $x_j$  is the  $j^{\text{th}}$  iterate of CG and  $k$  is the size of the window over witch the progress is calculated. (Martens and Sutskever, 2012).**

CG can be terminated when  $s_j$  is below some constant value (e.g. 0.0001). However, deciding when to terminate can be an extremely more complex and complicated process and thus a number of more advanced stopping criteria are available, with some even having nothing to do with the value of  $M$ .

---

**Algorithm 4.1**    Preconditioned conjugate gradient algorithm (PCG)

---

**inputs:**  $b, A, x_0, P$   
 $r_0 \leftarrow Ax_0 - b$   
 $y_0 \leftarrow$  solution of  $Py = r_0$   
 $p_0 \leftarrow -y_0$   
 $i \leftarrow 0$   
**while** termination conditions do not apply **do**  
     $\alpha_i \leftarrow \frac{r_i^\top y_i}{p_i^\top Ap_i}$   
     $x_{i+1} \leftarrow x_i + \alpha_i p_i$   
     $r_{i+1} \leftarrow r_i + \alpha_i Ap_i$   
     $y_{i+1} \leftarrow$  solution of  $Py = r_{i+1}$   
     $\beta_{i+1} \leftarrow \frac{r_{i+1}^\top y_{i+1}}{r_i^\top y_i}$   
     $p_{i+1} \leftarrow -y_{i+1} + \beta_{i+1} p_i$   
     $i \leftarrow i + 1$   
**end while**  
**output:**  $x_i$

---

**Algorithm 4.1.** The preconditioned CG. Noting that for minimizing the HFO quadratic we have  $x = \delta$ ,  $A = B_{t-1}$  and  $b = \nabla f(w_{t-1})$ ,  $P$  the preconditioning matrix (Martens and Sutskever, 2012)

## Damping

The CG algorithm described previously, requires the curvature matrix  $B$  to be positive-definite. However, in the case of Neural Networks where the objective function is usually non-convex,  $B$  may not be positive-definite, which means that the minimizer of  $M$  may not exist and thus the CG method becomes not applicable. Moreover, the minimizer  $\delta^*$  of the quadratic approximation  $M$  can be very large and “aggressive” in the early stages of the optimization, which means that is often located far beyond the region where the quadratic approximation is reasonably trust-worthy. These are general problems of 2<sup>nd</sup> order optimization for which a method called ‘damping’ addresses.

Damping methods essentially restrict the optimization of  $M$  to a “trust region” by augmenting  $M$  with penalty terms, which are designed to encourage the minimizer of  $M$  to remain somewhere where  $M$  is a good approximation of the objective function.

There are a number of damping methods proposed by Martens, which are applicable to HFO (Martens and Sutskever, 2012). However, the one used in this dissertation is the Tikhonov Damping with the Levenberg-Marquardt heuristic (Nocedal and Wright, 1999).

Tikhonov regularization or Tikhonov damping is one of the most well-known damping methods, which works by penalizing the quadratic model by introducing an additional quadratic penalty term into the quadratic model  $M$ . Thus, instead of minimizing  $M$ , we minimize a “damped” quadratic

$$\hat{M}(\delta) \equiv M(\delta) + \frac{\lambda}{2} \delta^\top \delta = f(\theta) + \nabla f(\theta)^\top \delta + \frac{1}{2} \delta^\top \hat{B} \delta$$

**Equation 4.7: The new damped quadratic, where  $\hat{B} = B + \lambda I$  and  $\lambda \geq 0$  is a scalar parameter determining the “strength” of the damping.**

Picking a good value of  $\lambda$  is critical to the success of the Tikhonov damping. Picking a too high value of  $\lambda$  results in updates which resemble gradient descent with extremely small learning rate that essentially take away all the benefits of 2<sup>nd</sup> order optimizations discussed previously (Martens and Sutskever, 2012). Too small, and CG will aggressively optimize the quadratic, resulting in very large weight updates that may increase the objective instead of decreasing it. This can be clearly observed by the difference in CG iterations needed per HFO iteration, based on the initial damping value of  $\lambda$  in the experiments section of the dissertation.

Dynamically adjusting the value of  $\lambda$  during optimization is just as important however, in order for it to constantly keep up with the changing local curvature

properties of the objective function  $f$ . A good method for addressing this issue is the Levenberg-Marquardt heuristic. This heuristic defines a reduction ratio, which measures the ratio of the reduction in the objective produced by the update  $\delta$ , to the reduction predicted by the quadratic model.

$$\rho \equiv \frac{f(\theta_{k-1} + \delta_k) - f(\theta_{k-1})}{M_{k-1}(\delta_k)}$$

**Equation 4.8: The reduction ratio measuring the reduction of the objective function comparing to the quadratic**

When  $\rho$  is much smaller than 1, the quadratic model overestimates the amount of reduction needed, so the value of  $\lambda$  should increase in order for future updates to be more reliable and smaller, as discussed previously. Contrary, when  $\rho$  is closer to 1, the quadratic model has a decent minimizer and so  $\lambda$  can be reduced since there is some margin for allowing larger and more substantial updates.

More specifically the Levenberg-Marquardt heuristic proposes two explicit rules to dynamically adapt the value of  $\lambda$ :

1. If  $\rho > 3/4$  then  $\lambda \leftarrow 2/3\lambda$

2. If  $\rho < 1/4$  then  $\lambda \leftarrow 3/2\lambda$

else  $\lambda \leftarrow \lambda$

Despite the clear benefits of damping, it is important to note that they are very tricky and must be used with care. If they are overused, they produce extremely reliable updates, which are simultaneously useless since they are too small. Moreover, if they are not properly calibrated they can produce updates which give the best reductions of the objective function in early stages but may not result in the best global optimization performance in the end.

## Gauss-Newton Matrix

There is a significant problem, briefly mentioned previously, regarding the use of the Hessian as the curvature matrix. The problem is the inability to apply the CG algorithm to the quadratic model if the curvature matrix is not positive-definite, which the Hessian sometimes tends to be. While the damping methods address this issue in a way, there is a more direct solution to deal with this.

Instead of using the Hessian Matrix as the curvature matrix, another matrix can be used which is guaranteed to always be positive semi-definite. This new matrix is the generalized Gauss-Newton matrix, which is an approximation of the Hessian (Schraudolph, 2002). The benefits of using this matrix do not only lie in the fact that it is always positive semi-definite but actually in practice, it tends to work much better both in regards in efficiency and in performance, than the Hessian. This even applies to situations where Hessian is positive-definite and there is no problem in using it as the curvature matrix. However, the use of the Gauss-Newton matrix does not eliminate the need for damping, but when combining them both, HFO produces much better updates with significantly less damping.

## Evaluating the Hessian-Vector Multiplication

There have been many references in this dissertation stating that no explicit evaluation and storing of the Hessian is being done for HFO. Instead, dot products with the Hessian and arbitrary vectors  $v \in \mathbb{R}^n$  are being computed and utilized, which cost as much as a gradient evaluation.

If you consider the Hessian to be the Jacobian matrix (first order derivatives matrix) of the gradient, by the definition of directional derivatives, the  $H(w)v$  product is the directional derivative of the gradient  $\nabla f(w)$  in the direction  $v$ , which gives

$$H(w)v = \lim_{\varepsilon \rightarrow 0} \frac{\nabla f(w + \varepsilon v) - \nabla f(w)}{\varepsilon}$$

**Equation 4.9:**  $H(w)v$  is the directional derivate of the gradient in the direction  $v$

While this may imply a finite-differences algorithm for computing  $Hv$  at the cost of a single gradient evaluation, in practice finite-differences suffer from numerical errors, which are extremely undesirable in neural network training.

Consequently, another method is being used which avoids those errors. This method is called 'Forward Differentiation', originally proposed by Wengert (Wengert, 1964) and later adjusted to neural network training by Pearlmutter (Pearlmutter, 1994).

The idea behind forward differentiation is to make repeated use of the chain rule to the value of every node of the gradient, like in the BP algorithm described in section 2.2. More precisely an  $R_v(X)$  operator is defined, which denotes the directional derivative of  $X$  in direction  $v$ .

$$R_v X = \lim_{\varepsilon \rightarrow 0} \frac{X(w + \varepsilon v) - X(\theta)}{\varepsilon} = \frac{\partial X}{\partial w} v$$

**Equation 4.10:** The  $R_v(X)$  operator being the directional derivative of  $X$  in the direction  $v$

Since the  $R$  operator is a derivative operator, it obeys the usual rules of differentiation:

$$\begin{aligned} R_v(X + Y) &= R_v X + R_v Y && \text{linearity} \\ R_v(XY) &= (R_v X)Y + XR_v Y && \text{product rule} \\ R_v(h(X)) &= (R_v X)h'(X) && \text{chain rule} \end{aligned}$$

**Equation 4.11:** The standard rules of differentiation

By applying these rules recursively to the gradient calculation algorithm, in a way analogous to back-propagation, the Hv product can be efficiently computed.

Algorithm 4.2 shows the algorithm for a simple gradient evaluation, while Algorithm 4.3 shows the modification of the gradient algorithm by applying the rules of differentiation to compute the Hv product. Similarly, algorithm 4.4 shows the algorithm for the Gv product, which is similar to Hv but simpler.

---

```

input:  $y_0; \theta$  mapped to  $(W_1, \dots, W_{\ell-1}, b_1, \dots, b_{\ell-1})$ .
for all  $i$  from 0 to  $\ell - 1$  do
     $x_{i+1} \leftarrow W_i y_i + b_i$ 
     $y_{i+1} \leftarrow s_{i+1}(x_{i+1})$ 
end for
 $dy_\ell \leftarrow \partial L(y_\ell; t_\ell) / \partial y_\ell$  ( $t_\ell$  is the target)
for all  $i$  from  $\ell - 1$  downto 0 do
     $dx_{i+1} \leftarrow dy_{i+1} s'_{i+1}(x_{i+1})$ 
     $dW_i \leftarrow dx_{i+1} y_i^\top$ 
     $db_i \leftarrow dx_{i+1}$ 
     $dy_i \leftarrow W_i^\top dx_{i+1}$ 
end for
output:  $\nabla f(\theta)$  as mapped from  $(dW_1, \dots, dW_{\ell-1}, db_1, \dots, db_{\ell-1})$ .

```

---

**Algorithm 4.2:** An algorithm for computing the gradient of a feedforward neural network, where  $L(y_i; t_i)$  is one of the loss functions of table 4.1 (Martens and Sutskever, 2012)

---

```

input:  $v$  mapped to  $(RW_1, \dots, RW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1})$ 
 $Ry_0 \leftarrow 0$  (since  $y_0$  is not a function of the parameters)
for all  $i$  from 0 to  $\ell - 1$  do
     $Rx_{i+1} \leftarrow RW_i y_i + W_i Ry_i + Rb_i$  (product rule)
     $Ry_{i+1} \leftarrow Rx_{i+1} s'_{i+1}(x_{i+1})$  (chain rule)
end for
 $Rdy_\ell \leftarrow R \left( \frac{\partial L(y_\ell; t_\ell)}{\partial y_\ell} \right) = \frac{\partial \{ \partial L(y_\ell; t_\ell) / \partial y_\ell \}}{\partial y_\ell} Ry_\ell = \frac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2} Ry_\ell$ 
for all  $i$  from  $\ell - 1$  downto 0 do
     $Rdx_{i+1} \leftarrow Rdy_{i+1} s'_{i+1}(x_{i+1}) + dy_{i+1} R \{ s'_{i+1}(x_{i+1}) \}$  (product rule)
     $\quad \quad \quad = dy_{i+1} s''_{i+1}(x_{i+1}) Rx_{i+1}$  (chain rule)
     $RdW_i \leftarrow Rdx_{i+1} y_i^\top + dx_{i+1} Ry_i^\top$  (product rule)
     $Rdb_i \leftarrow Rdy_i$ 
     $Rdy_i \leftarrow RW_i^\top dx_{i+1} + W_i^\top Rdx_{i+1}$  (product rule)
end for
output:  $H(w)v$  as mapped from  $(RdW_1, \dots, RdW_{\ell-1}, Rdb_1, \dots, Rdb_{\ell-1})$ .

```

---

**Algorithm 4.3:** An algorithm for computing the  $H(w)v$  product in a feedforward neural network, where  $L(y_i; t_i)$  is one of the loss functions of table 4.1 (Martens and Sutskever, 2012)

---

```

input:  $RW_1, \dots, RW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1}$ .
 $Ry_0 \leftarrow 0$  ( $y_0$  is not a function of the parameters)
for all  $i$  from 1 to  $\ell - 1$  do
     $Rx_{i+1} \leftarrow RW_i y_i + W_i Ry_i + Rb_i$  (product rule)
     $Ry_{i+1} \leftarrow Rx_{i+1} s'_{i+1}(x_{i+1})$ 
end for
 $Rdy_\ell \leftarrow \frac{\partial^2 L(y_\ell; t_\ell)}{\partial y_\ell^2} Ry_\ell$ 
for all  $i$  from  $\ell - 1$  downto 1 do
     $Rdx_{i+1} \leftarrow Rdy_{i+1} s'_{i+1}(x_{i+1})$ 
     $RdW_i \leftarrow Rdx_{i+1} y_i^\top$ 
     $Rdb_i \leftarrow Rdx_{i+1}$ 
     $Rdy_i \leftarrow RW_i^\top dx_{i+1}$ 
end for
output:  $(RdW_1, \dots, RdW_{\ell-1}, Rb_1, \dots, Rb_{\ell-1})$ .

```

---

**Algorithm 4.4:** An algorithm for computing the  $G(w)v$  product in a feedforward neural network, where  $L(y; t)$  is one of the loss functions of table 4.1 (Martens and Sutskever, 2012)

Name	$L(z; t)$	$\nabla L(z; t)$	$L''(z; t)$
Squared error	$\frac{1}{2} \ p - t\ ^2$	$-(p - t)$	$I$
Cross-entropy error	$-t \log p - (1 - t) \log(1 - p)$	$-(p - t)$	$\text{diag}(p(1 - p))$
Cross-entropy error (multi-dim)	$-\sum_i [t]_i \log[p]_i$	$-(p - t)$	$\text{diag}(p) - pp^\top$

Table 4.1 : Typical losses with their derivatives and Hessians.

## 4.4 System Implementation

For the purpose of this dissertation, a library for the Hessian Free Optimization was used, which was implemented by in 2015 (Rasmussen, 2015) based on the papers of Martens and Sutskever (Martens, 2010; Martens and Sutskever, 2011). It made use of many optimization tricks that were suggested in those papers and the structure of the library was ideal for the needs of this dissertation.

First of all, both Feedforward and Recurrent Neural Networks were supported for HFO training, which were needed for implementing the Bidirectional network described in section 4.1. Moreover, the connections between the layers were not restrictive, meaning that it was possible to interconnect arbitrary layers, which was also necessary for constructing the BRNN connections. Finally, all the standard nonlinearities and loss functions were built-in, which is great for experimentation.



However, the RNN recursion was later discovered that it was possible only within a single layer, namely connections from a layer back to itself. For this reason, some functions of the library were modified to allow multi-layer recursion from arbitrary layers. Moreover, there was support for a single input layer (which is standard in ANN), but BRRN requires three different inputs for each separate network. As a result, the function which implements the forward pass was modified to allow the input layer to be split into three different vectors, with each vector being fed into a different hidden layer.

The way the RNN in this library is implemented requires the use of minibatches. In minibatch training, the training set is split into smaller batches (subsets) and the weight updates are done after all examples in a single batch are through. Generally, there are two more ways of training. The online method, where the weight updates are computed and applied every time a training example is fed into the network, and the batch method, where all the training examples in the dataset are used to calculate the weight update. Consequently, minibatching is somewhere in the middle of the two methods, trying to balance the benefits of both. Having a larger minibatch size (approaching the batch method), the convergence of the learning algorithm is usually more accurate but significantly slower and requires much more memory. Contrary, in a smaller minibatch size (approaching the online method), the convergence is usually faster but a lot less accurate.

In the case of PSSP, a minibatch size was chosen in a way to include as much information as possible, without splitting proteins in half, which would have a negative result in the predictions. More specifically, the number of training examples in a mini batch was chosen to be the number of amino acids in the longest amino acid chain of a protein included in the datasets used, which was 753. This way, all information regarding a single protein would be used to make the adjustments to the weights of the network. For smaller sized proteins, with less amino acids, a padding of 0s was added to even out the batches.

## Chapter 5

### Results and Discussion

---

5.1 Feedforward Neural Network Experimentations	68
5.2 Recurrent Neural Network Experimentations	73
5.3 Bidirectional Recurrent Neural Network Experimentations	80
5.4 Cross Validation, Filtering and Ensembles	87

---

## 5. Results and Discussion

In order to find the optimal parameters of the Bidirectional Recurrent network many experimentations have been conducted, using different values for the hyper-parameters.

Before experimenting with the final BRNN though, the built-in Feedforward and Recurrent neural networks had to be tested, in order to verify their ability to work and learn properly using HFO, the PSSP problem. Otherwise, there was no point in modifying them into a BRNN, since that would not work either.

It is also important to note that in order to get the best accuracy in a given model, it is critical to stop the learning process when the testing error starts getting higher instead of decreasing at each iteration. This is called overfitting where the model starts learning the training examples by heart and fails to generalize for unseen data. It can be clearly observed in figure 5.1 at iteration around 37 until the end. To address this issue, a condition was implemented to check whether the testing error at a given iteration is higher than the one at a number of iterations previously. After some experimentations, it was observed the fluctuations did not really last for more than 5-10 iterations before converging into overfitting or escaping a local minimum and start decreasing again. For this reason, the number of previous iterations for comparing the testing error was set at 10. However, the last 10 iterations of possible overfitting before being terminated have extremely bad effects on the final accuracy for the network. Consequently, the best weights up until a given iteration of the network are being stored in order to restore the network in its optimal iteration to counter the overfitting of those 10 iterations, when the training is over.

## 5.1 Feedforward Neural Network Experimentations

Beginning with the Feedforward network, the parameters that had to be tuned were fairly straightforward. As is standard in FF neural networks, the number of neurons in hidden layers as well as the nonlinearities and loss functions used are the major architectural parameters that need to be optimized. In terms of the HFO parameters, fortunately there are not many. There are mainly only two, which is the initial damping  $\lambda$  described in section 4.2 and the maximum number of Conjugate Gradient iterations, which was set at a fixed 500, but in practice rarely exceeded the 300 mark. Finally, the last parameter which had to be tuned for every type of network was the window size discussed in section 3.7.

The training for FFN was done using batch learning, meaning that every single training example was fed into the network before adjusting and updating the weights. This is because simple FFN do not require much memory and batch learning usually results in better performance.

Parameter	Value
Window Size	11
Number of Hidden Layers	1
Hidden Layer 1 Neurons	75
Nonlinearity	Softmax
Loss Function	Cross Entropy
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.1: The parameters for the first experiment on FFN

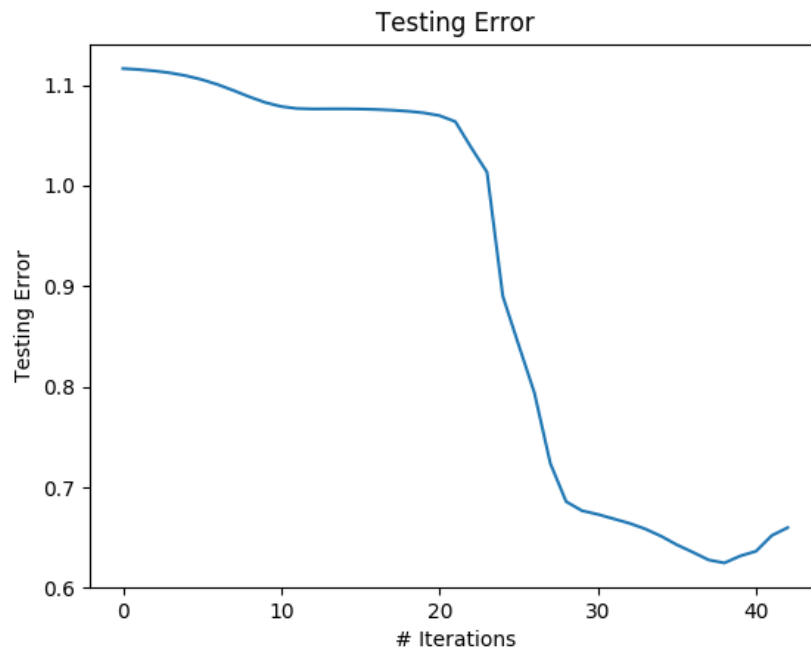


Figure 5.1: The graph of the testing error, with regards to the number of iteration

Beginning the experimentations with the default settings of the library, as mentioned in the table above and an initial window size of 11, it is clear that the algorithm works extremely well. The Q3 accuracy with these settings is **73.5%**, which not only validates the correctness of the implementation for the FFN, but also is a really decent accuracy overall, considering the simplicity of the feedforward network with a single hidden layer. However, as it turned out this combination of softmax nonlinearity and cross entropy loss function required an unexpected amount of memory, which made it impossible to experiment with more complex architectures (e.g. more neurons and hidden layers) or with a bigger window size.

As a result, the forthcoming experiments were conducted with the other nonlinearities available (ReLU and Logistic/Sigmoid) and with the Squared Error loss function.

Parameter	Value
Window Size	11
Number of Hidden Layers	1
Hidden Layer 1 Neurons	75
Nonlinearity	ReLU
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.2: The parameters for the second experiment on FFN

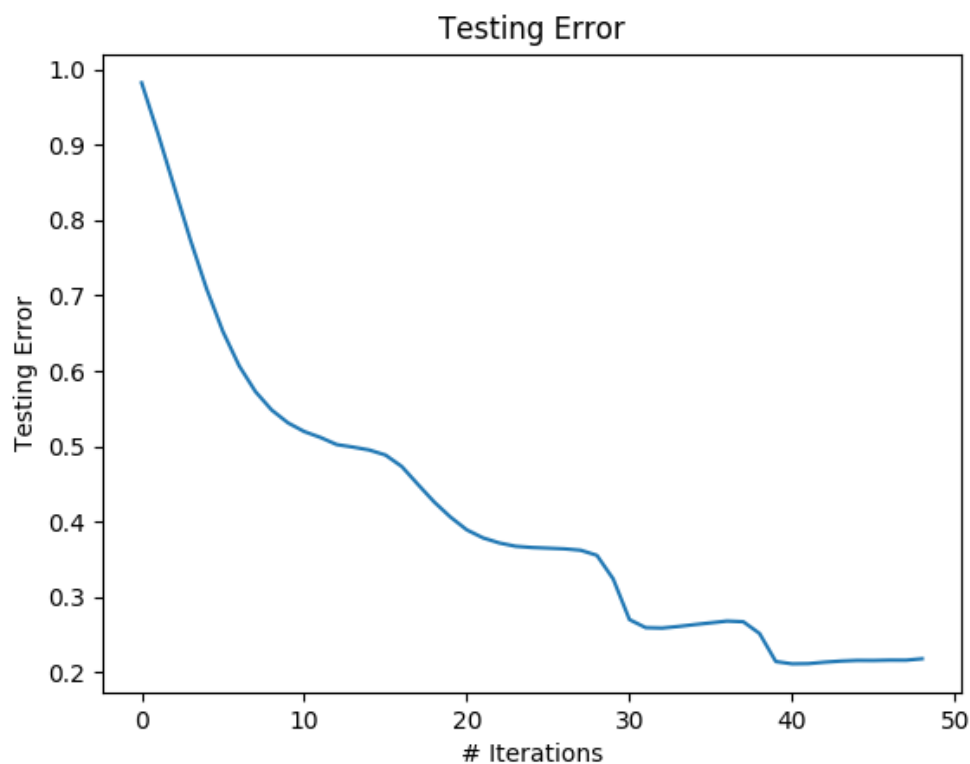


Figure 5.2: The graph of the testing error, with regards to the number of iteration

While it seems that the testing error was significantly reduced ( $\sim 0.24$ , Figure 5.2) by the ReLU functions comparing to the error in the first example ( $\sim 0.62$ , Figure 5.1) using the Softmax function, it is not true, since the errors in the two examples

are not comparable. In the first case the loss function used was the Cross entropy error and in the second the Squared Error, which both use different metrics to calculate the error of the network. In fact, the Q3 accuracy of this method is just **73.1%**, which is slightly lower than the **73.5%** of the first experiment.

Parameter	Value
Window Size	11
Number of Hidden Layers	1
Hidden Layer 1 Neurons	75
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.3: The parameters for the third experiment on FFN

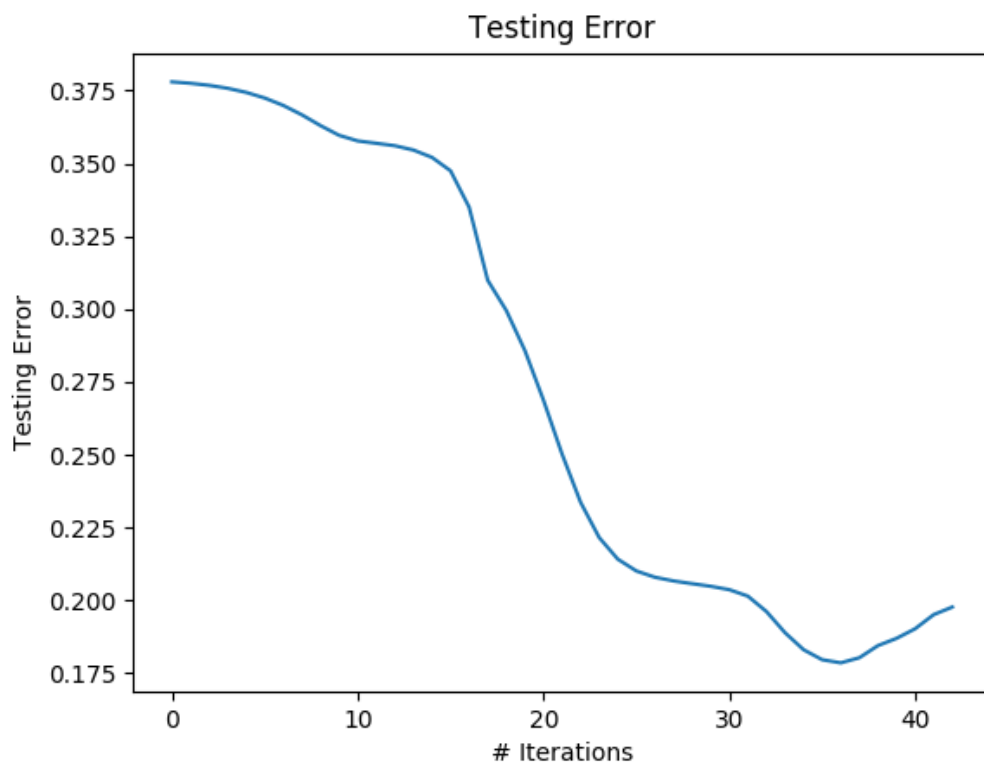


Figure 5.3: The graph of the testing error, with regards to the number of iteration

It is clear that the Logistic function works best with the Squared error for this problem (Table 5.3). The testing error was reduced to ~0.18 with the Q3 accuracy being **75.1%** (Figure 5.3).

Parameter	Value
Window Size	13
Number of Hidden Layers	1
Hidden Layer 1 Neurons	90
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	10
Maximum CG Iterations	500

Table 5.4: The parameters for the best experiment on FFN

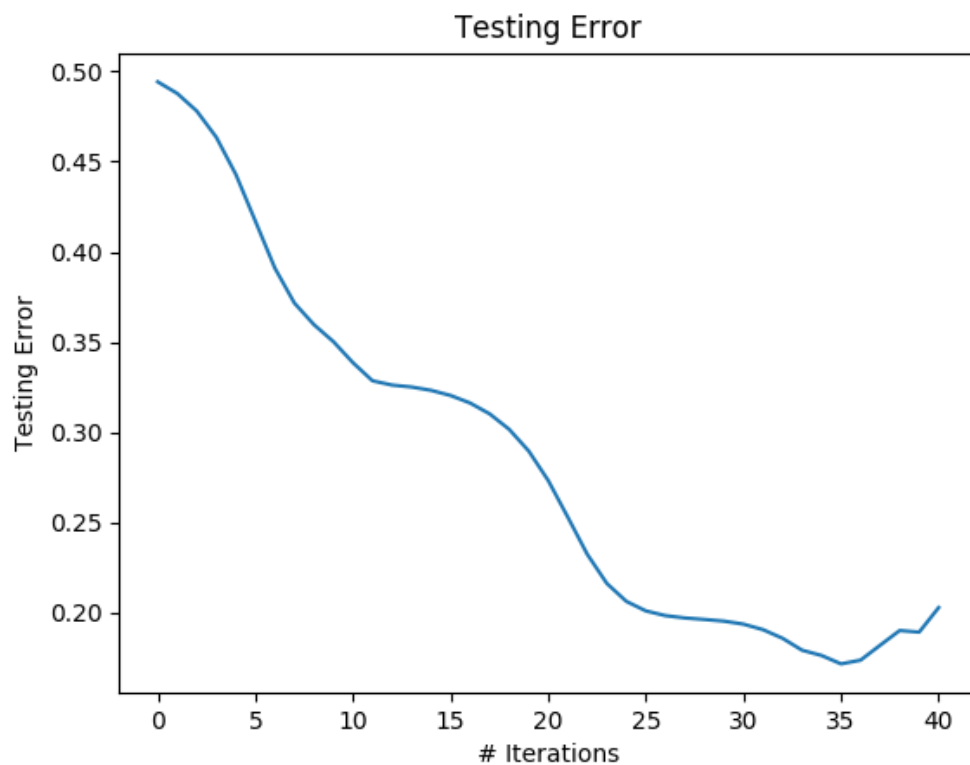


Figure 5.4: The graph of the testing error, in regards to the number of iteration



After many experimentations, which are not discussed individually since FFN is not the purpose of this dissertation, but rather BRRN, the best accuracy taken was **76.01%** using the above configurations (Table 5.4). More information about the neighboring amino acids are given to the network by having a slightly bigger window size, which required an increase in the number of hidden neurons as well in order to be able to store more complex relations. However, since the network by design is very simple, increasing the window too much results in worse results since it simply learns the training examples by heart and fails to generalize. The optimal window for FFN was found to be 13 and any bigger than that resulted in significantly worse results (72-74%)

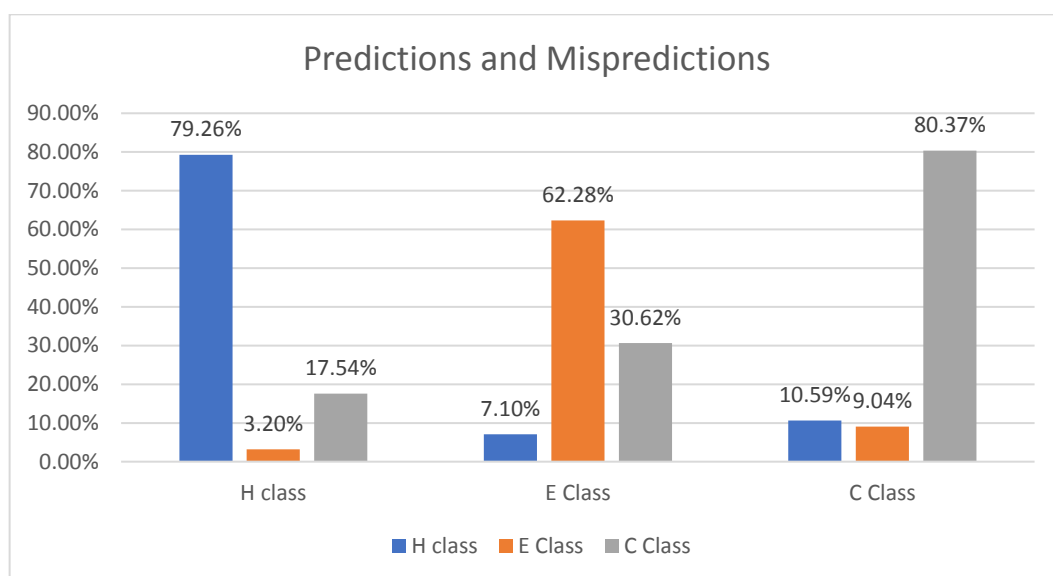


Figure 5.5: The confusion matrix, describing the predictions and mispredictions of the FFN.

However, observing figure 5.5, it is clear that the network, despite its relatively good accuracy, is unable to predict the E class effectively (62.28% comparing to ~80% of the other classes). A reason as to why this is happening could be that the 'E' class could potentially be formed with specific long-range dependencies and interactions with amino acids not close to the predicted one, which FFN, given its simplicity and relatively small window size, is unable to attain. Since every nonlinearity and loss function implemented seemed to work fine for FFN, it was time to test the Recurrent networks if they work just as well.

## 5.2 Recurrent Neural Network Experimentations

The way RNN are implemented in the library used is that each layer can be defined as recurrent by feeding its input back to itself, or not, by behaving as a standard feedforward layer. Therefore, along with the parameters that had to be tuned for the feedforward network, the RNN has to also define the layers which should be recurrent. The default is to make all layers except the input layer and output layer recurrent, which is what is being done in the following experimentations.

By design, recurrent neural networks are able to correlate and take into account previous examples in their predictions. For this reason, the input window has been modified to have the amino acid being predicted on the far right edge of the window. This way, the network would have input in a given time only the amino acids following it, since the ones preceding it would have already passed into the network and fed back into it as a recurrent input.

The training for RNN was done using minibatch learning, discussed in section 4.4. The way the error is calculated in this method is by summing all the individual errors from each example in a batch so it seems to be much higher than it actually is, comparing to the previous errors from FFN.

Parameter	Value
Window Size	15
Number of Hidden Layers	1
Hidden Layer 1 Neurons	90
Nonlinearity	ReLU / Logistic
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.5: The parameters for the experiment on RNN in terms of activation functions

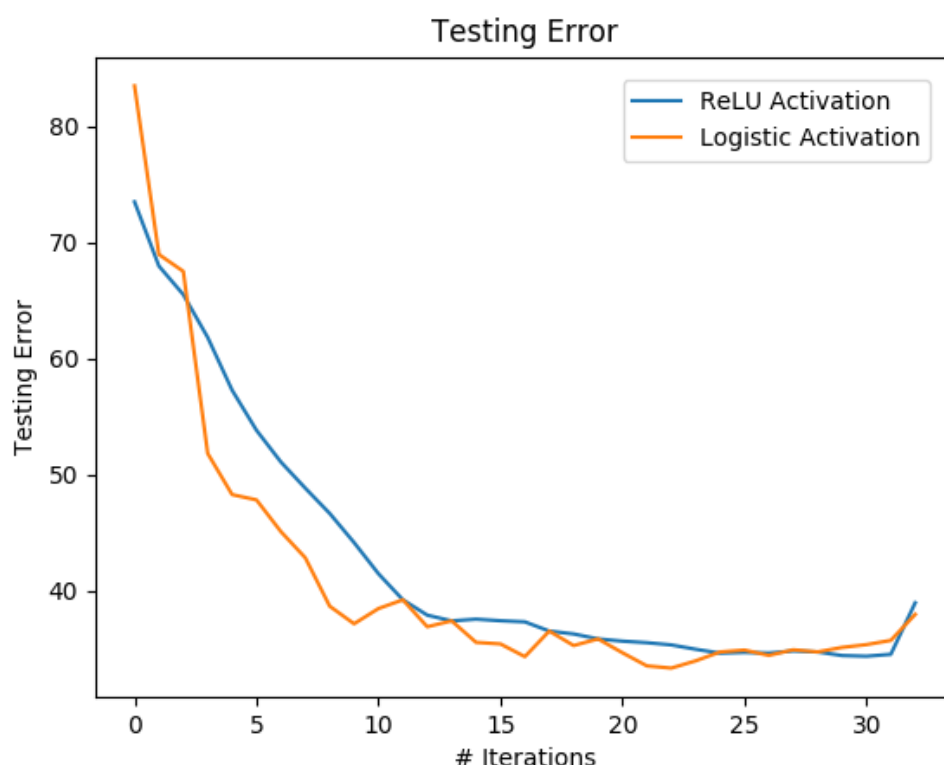


Figure 5.6: The graph of the testing error, with regards to the number of iteration for the ReLU and Logistic activation functions

Starting off with the default parameters again (Table 5.5), but not including the Softmax activation with the Cross Entropy error for the same reasons discussed earlier. Using the parameters on the table above, the testing error taken using both ReLU and Logistic activation functions with a window size of 15 was just 34.2 and 33.8 with final accuracies **66.5%** and **67.1%** respectively (Figure 5.6).

Despite the fact that RNN by design does take into account previous examples in its predictions, adjusting the window to not include them as input turns out that it was not a good idea. Given their importance in the determination of a specific amino acid's secondary structure, the RNN recursive memory is not enough to justify excluding those examples altogether from the input. For this reason, the forthcoming experiments were conducted using the window method done in FFN as well, namely having the amino acid in the center of the window being the one getting predicted, in order to include both following and preceding amino acids in the prediction of a single amino acid.

Parameter	Value
Window Size	11 / 15 / 21
Number of Hidden Layers	1
Hidden Layer 1 Neurons	75 / 90 / 110
Nonlinearity	ReLU
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.6: The parameters for the experiment on RNN in terms of window size and the ReLU activation function

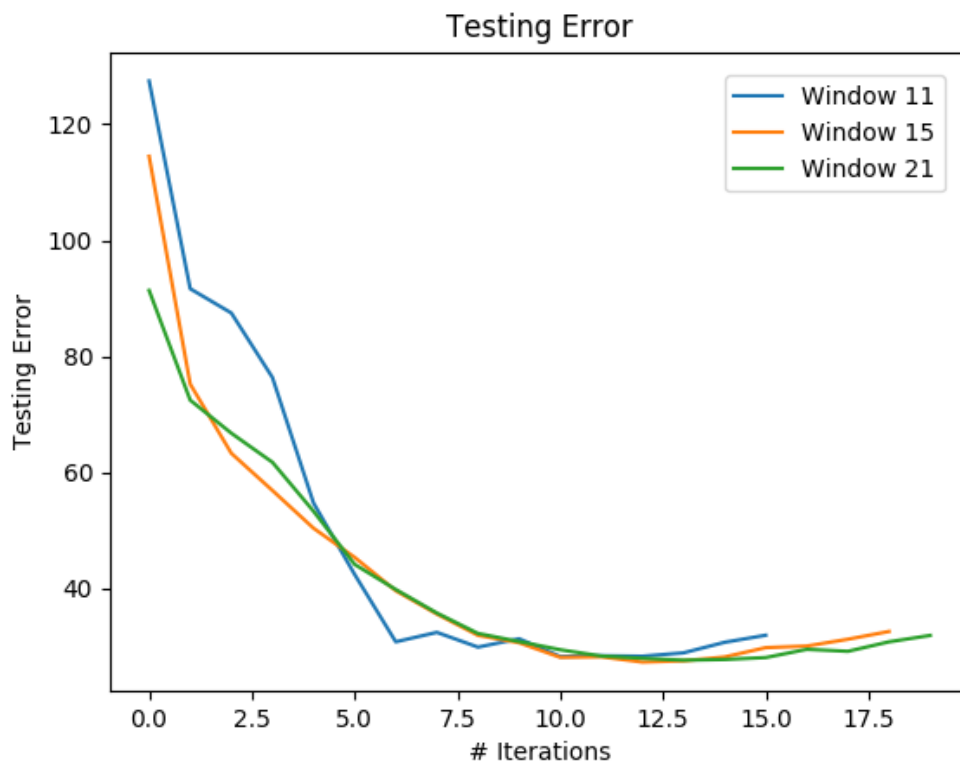


Figure 5.7: The graph of the testing error, with regards to the number of iteration for different window sizes and the ReLU activation

It is clear that adjusting the window sequence to have the amino acid being predicted in the center, greatly improves the performance of the network and the final accuracies. With the parameters used on the table 5.6, the testing error taken using the ReLU activation function with a window size of 11 was 26.4, with 15 it was 26.2 and finally with a window size of 21 it was 25.98 with final accuracies **74.5%**, **75.42%** and **75.82%** respectively (Figure 5.7). The best results were taken with a window size of 21 which is bigger than the one for FFN. This is because RNN has a slightly more complex architecture, with which it can calculate and store more complicated relations.

Parameter	Value
Window Size	11 / 15 / 21
Number of Hidden Layers	1
Hidden Layer 1 Neurons	75 / 90 / 110
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.7: The parameters for the experiment on RNN in terms of window size and the Logistic activation function

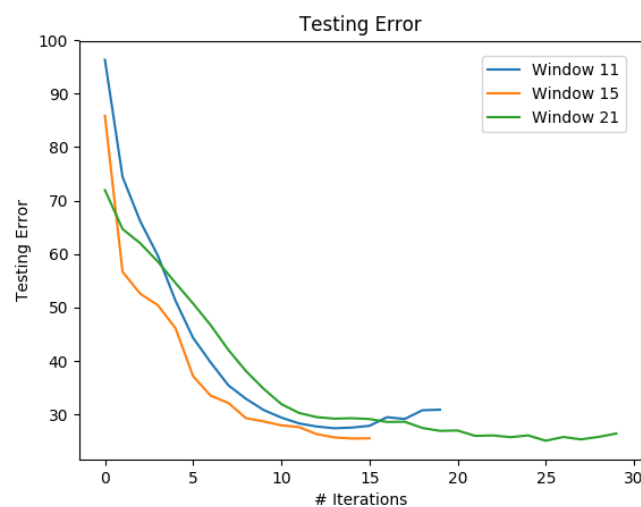


Figure 5.8: The graph of the testing error, with regards to the number of iteration for different window sizes and the Logistic activation function

Testing the network with the same parameters but with the Logistic activation function (Table 5.7), results in slightly better accuracies, about ~0.4-0.7% higher in each case, comparing with the ReLU function. More specifically, with a window size of 11 the accuracy was **74.9%**, with 15 it was **75.91%** and with 21 it was **76.21%** (Figure 5.8). It is clear that RNN was also implemented correctly for both activation functions and it is able to learn more than decently the PSSP problem, which is great, considering that both FFN and RNN networks have to be correct in order to be formed into the Bidirectional network.

Parameter	Value
Window Size	21
Number of Hidden Layers	1
Hidden Layer 1 Neurons	120
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	0.1
Maximum CG Iterations	500

Table 5.8: The parameters for the best experiment on RNN

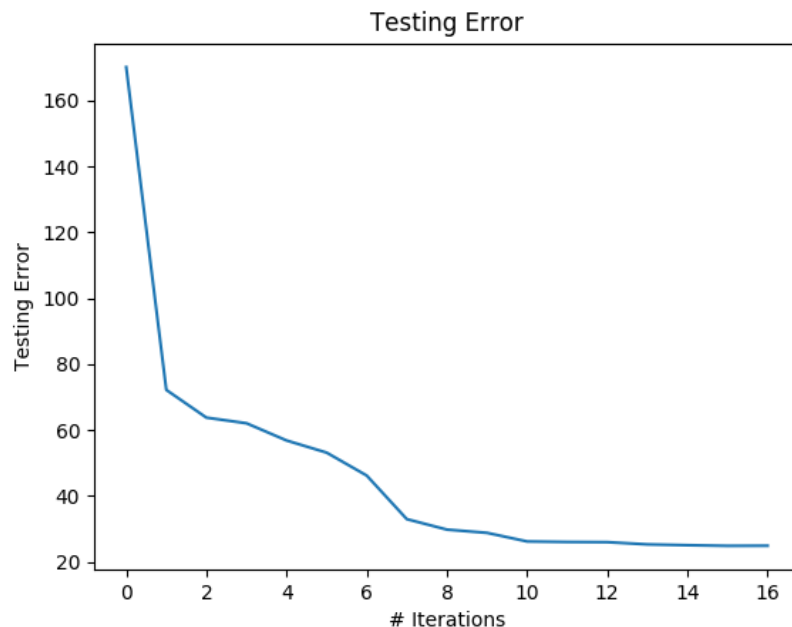


Figure 5.9: The graph of the testing error, with regards to the number of iteration for the best experiment on RNN

From the previous examples, it was clear that the logistic function, using a window size of 21 resulted in the best accuracies. Experimenting with the damping value  $\lambda$ , (Table 5.8) the optimal accuracy for RNN was **76.62%**. (Figure 5.9)

A significantly smaller damping value (0.1, comparing to the default of 45) gave the optimal results in both network architectures. This means that the quadratic approximations that HFO calculates are fairly reliable which results in the CG producing larger and substantial updates by being more aggressive, as discussed in section 4.3. This can be clearly observed by figure 5.10, which shows that not only does the  $\lambda$  value start at a lower point, but keeps decreasing using the Levenberg-Marquardt heuristic. This confirms that throughout the training the quadratic models are fairly accurate, which is reflected by the final accuracies. Otherwise, the damping would increase, producing smaller updates, which would produce much lower accuracies.

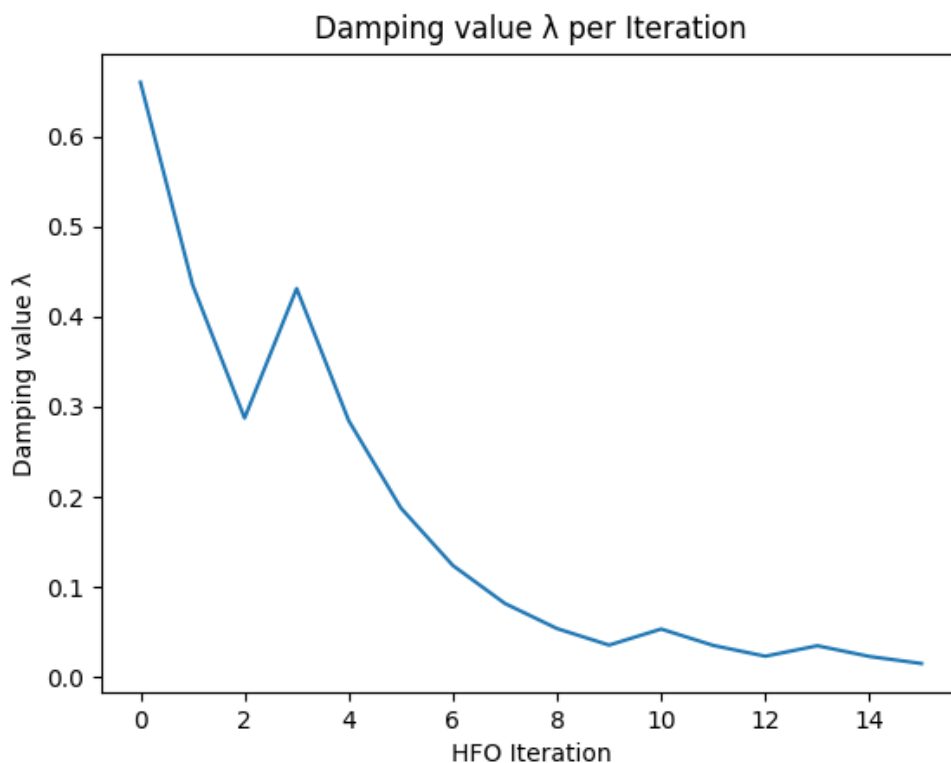


Figure 5.10: The graph of the the damping value of  $\lambda$ , with regards to the number of HFO iteration

### 5.3 Bidirectional Recurrent Neural Network Experimentations

After testing and validating the correct implementation and ability to learn the PSSP problem for both Feedforward and Recurrent network architectures, it was time to test the modifications described in section 4.4, which formed the BRNN.

In BRRN there are much more parameters that have to be tuned, since there are 3 individual networks with their hidden layers and neurons. Moreover, there is another window which is used for the recurrent networks to feed their input layers with a subsequence of the initial window's amino acids at each time stamp, as described in section 4.2. Moreover, since the Logistic nonlinearity consistently resulted in better results, comparing to the ReLU, it is the only one being used in the following experiments.

Parameter	Value
Window Size	21
Recurrent Window size	1 / 3 / 5
Backward Hidden Layer 1 Neurons	50
Backward Hidden Layer 2 Neurons	30
Forward Hidden Layer 1 Neurons	50
Forward Hidden Layer 2 Neurons	30
MLP Hidden Layer Neurons	80
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.9: The parameters for the experiment on BRNN in terms of recurrent window size



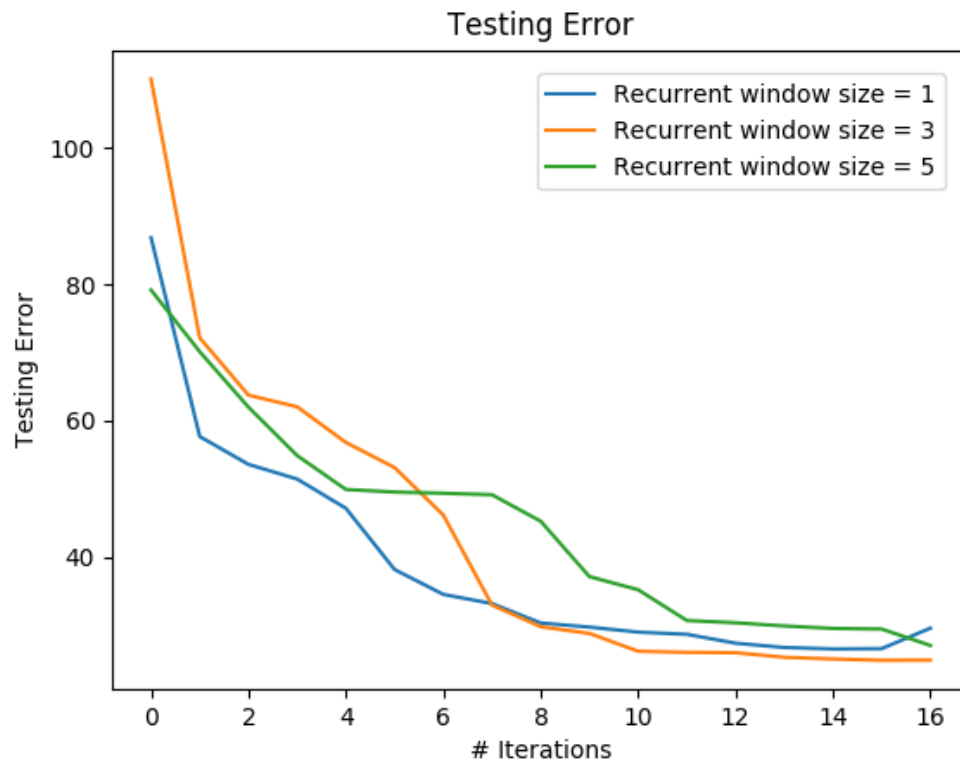


Figure 5.11: The graph of the testing error, with regards to the number of iteration in terms of recurrent window size

Starting the experiments with some arbitrary hidden layer sizes, and changing the recurrent window size in order to find an optimal in the range of 1 / 3 / 5 (Table 5.9) it is clear that the window size of 3 gives the better predictions (Figure 5.11). More specifically, the window size of 1 has accuracy of **74.81%**, the window size of 3 has **76.01%** and the window size of 5 has **73.97%**. Feeding too much information into the network at a time, results in even worse results than feeding it with significantly less. This is extremely important to highlight, since more information does not always mean better predictions.

Parameter	Value
Window Size	11 / 21 / 31
Recurrent Window size	3
Backward Hidden Layer 1 Neurons	50
Backward Hidden Layer 2 Neurons	30
Forward Hidden Layer 1 Neurons	50
Forward Hidden Layer 2 Neurons	30
MLP Hidden Layer Neurons	80
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	45
Maximum CG Iterations	500

Table 5.10: The parameters for the experiment on BRNN in terms of window size

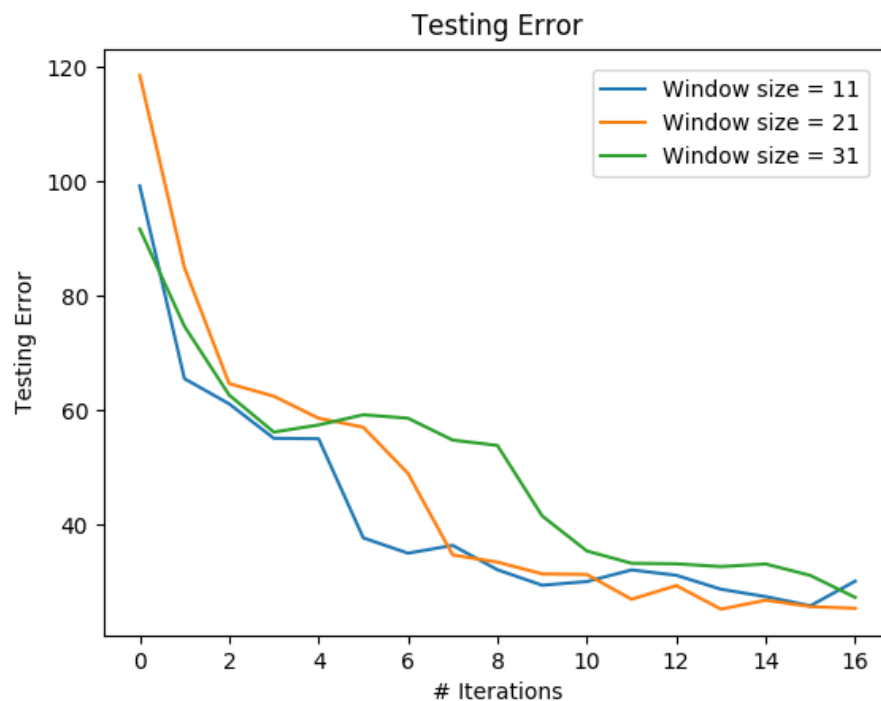


Figure 5.12: The graph of the testing error, with regards to the number of iteration in terms of the window size

Similar to the previous example, feeding too much information to the network does not always mean better performance with higher accuracies. For a window size of 11 (Table 5.10) the accuracy of the network was **74.85%**, for 21 it was **76.05%** and for 31 it was **74.73%**. (Figure 5.12)

Parameter	Value
Window Size	21
Recurrent Window size	3
Backward Hidden Layer 1 Neurons	50
Backward Hidden Layer 2 Neurons	30
Forward Hidden Layer 1 Neurons	50
Forward Hidden Layer 2 Neurons	30
MLP Hidden Layer Neurons	80
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	0.01 / 1 / 20 / 45
Maximum CG Iterations	500

Table 5.11: The parameters for the experiment on BRNN in terms of the damping parameter  $\lambda$

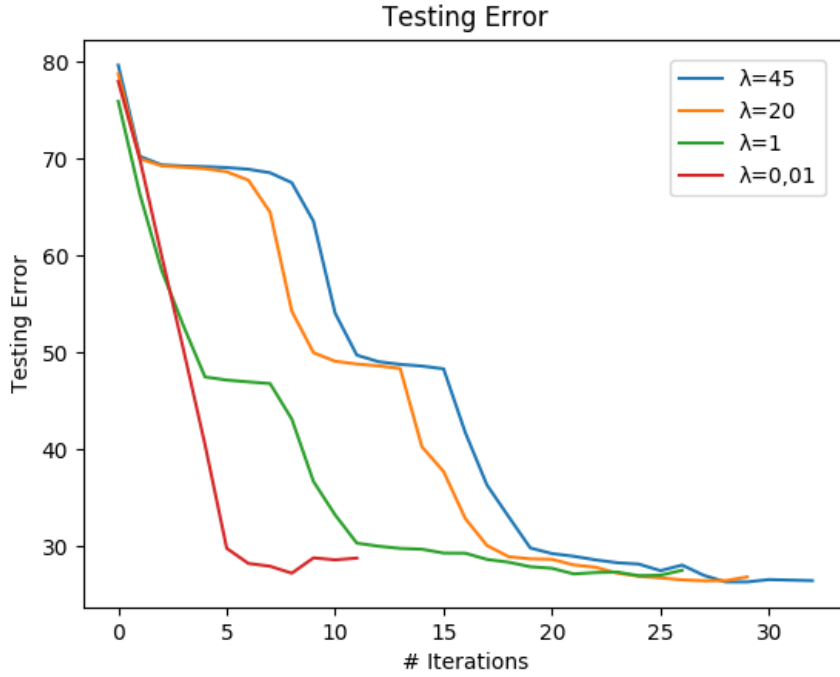


Figure 5.13: The graph of the testing error, with regards to the number of iteration in terms of the damping parameter  $\lambda$

Continuing with maybe the most important parameter in terms of both performance and execution time optimization that had to be tuned was the damping parameter  $\lambda$ . (Table 5.11). As discussed in section 4.3, picking a too high value of  $\lambda$  results in more reliable updates, which can be extremely small and inefficient. Too small, and CG will aggressively optimize the quadratic, resulting in very large weight updates. Figures 5.13, 5.14 and 5.15 clearly demonstrate this theory, where at a higher value of 45 the HFO computes less CG iterations per HFO iteration, which results in smaller updates and higher execution time. As the damping becomes lower, the updates become larger by calculating significantly more CG iterations and lowering the execution time. It is important to find a balance between the two, which in the case of this problem, the balance of the  $\lambda$  value was at 1. This resulted in a final accuracy of **76.45%**, compared to **75.82%**, **76.11%** and **76.07%** for damping 0.01, 20 and 45 respectively.

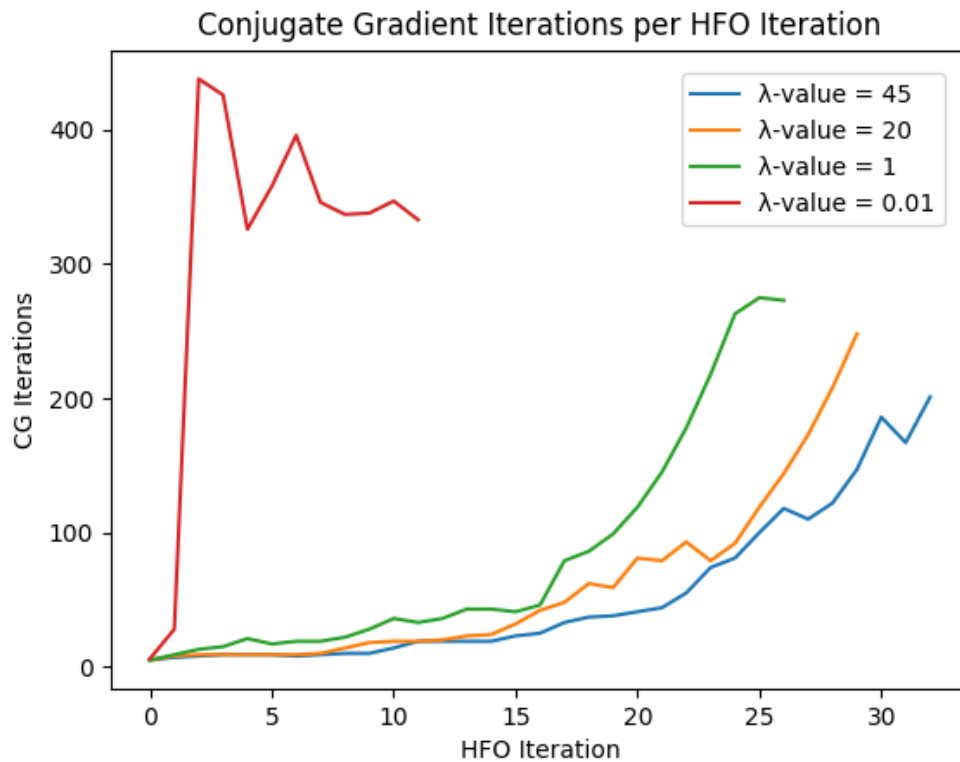


Figure 5.14: The graph of the CG iterations per HFO iteration with different values of damping



Figure 5.15: The graph of the execution times with different values of damping

Parameter	Value
Window Size	21
Recurrent Window size	3
Backward Hidden Layer 1 Neurons	25
Backward Hidden Layer 2 Neurons	13
Forward Hidden Layer 1 Neurons	25
Forward Hidden Layer 2 Neurons	13
MLP Hidden Layer Neurons	50
Nonlinearity	Logistic
Loss Function	Squared Error
Damping factor $\lambda$	1
Maximum CG Iterations	500

Table 5.12: The parameters for the best experiment on BRNN.

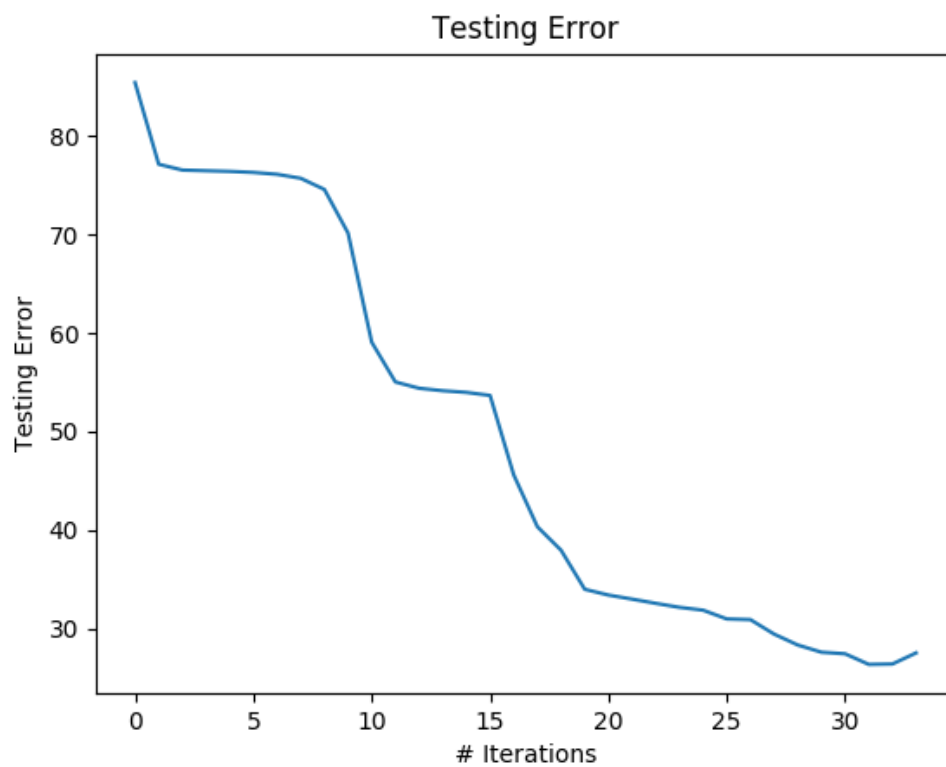


Figure 5.16: The graph of the testing error for the best experiment on BRNN.

The final parameters that were adjusted to produce the best results in BRNN were the number of neurons in each hidden layer for both the recurrent networks and the feedforward one (Table 5.12). It seems that the network does not require many neurons to make better predictions. This is probably because of the input, due to the MSA profiles, which are mostly 0s which the network can learn to encode in significantly less neurons. In fact, comparing to the previous experiments, they only require less than half of it, which significantly reduced the training duration from 10-11 hours depending on the damping value to just 7-8. The final accuracy of the network was 76.91%, which is extremely good.

## 5.4 Cross Validation, Filtering and Ensembles

However, good results taken from a single testing set does not indicate whether the network is indeed a good predicting model, or whether the testing set used was just favoring its performance. For this reason, the 10-fold cross validation is being used to validate the good generalization properties of it, as discussed in chapter 3.

The parameters used are the same of Table 5.12 which gave the best accuracies overall.

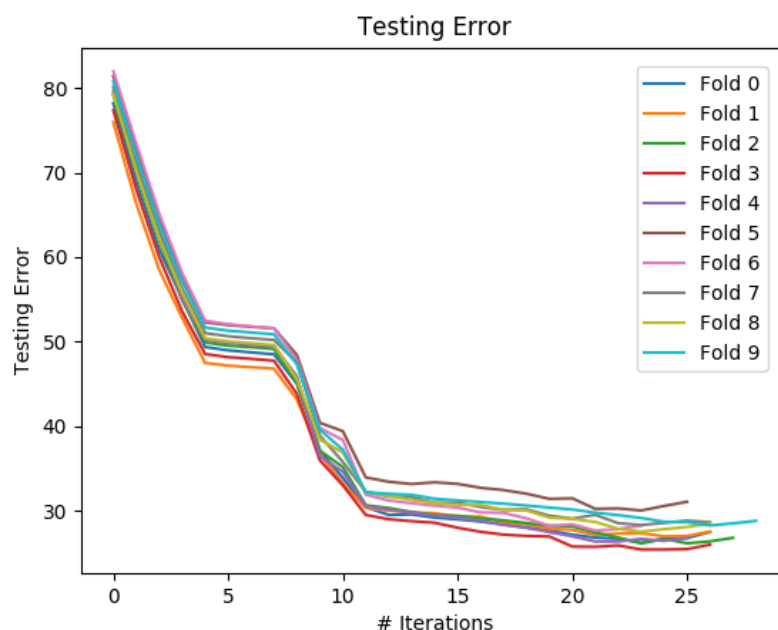


Figure 5.17: The graph of the testing error for all the folds in cross validation.

	<b>Q3 (%)</b>	<b>QH (%)</b>	<b>QE (%)</b>	<b>QC (%)</b>	<b>SOV</b>
<b>Fold0</b>	76.81	79.11	69.72	79.37	70.01
<b>Fold1</b>	74.91	71.02	68.12	<b>80.1</b>	71.02
<b>Fold2</b>	76.32	74.02	69.01	78.2	71.58
<b>Fold3</b>	76.02	78.01	68.12	76.52	71.02
<b>Fold4</b>	75.72	76.52	70.02	77.01	73.54
<b>Fold5</b>	75.01	78.52	68.51	75.12	70.92
<b>Fold6</b>	<b>77.01</b>	<b>79.11</b>	68.12	78.78	72.41
<b>Fold7</b>	75.95	77.91	<b>71.74</b>	75.03	<b>73.68</b>
<b>Fold8</b>	74.75	76.42	67.25	77.12	70.36
<b>Fold9</b>	75.52	77.14	71.12	74.15	73.22
<b>Average</b>	<b>75.8</b>	76.74	69.17	77.14	71.78

Table 5.13: The results of the 10-fold cross validation for the overall Q3 accuracy, the accuracy for each class and the overall SOV of each fold as well as the average results for all folds.

Observing figure 5.17, illustrating the training process for all folds, it seems that each fold learns with a similar pace. It is clear that the model works for all folds, however the overall accuracy dropped by about 1% to 75.8%, which is expected since the model was optimized for the first fold. Slightly different parameters would produce better results for the other folds which could potentially improve the overall accuracy for the cross validation. However, due to the high amounts of execution time, it was not possible to find the optimal parameters that would benefit all the folds overall.

Table 5.13 gives a better insight on the quality of the results, instead of a simple Q3 accuracy. Observing the accuracies for each class as well as the SOV accuracy, it is clear that while the Helix and Coil classes are getting predicted



fairly accurately, the sheet conformations not so much, which negatively affects the overall SOV performance of the network.

However, as discussed in Chapter 3 there are a number of ways to improve either the Q3 accuracy using post-processing filtering with SVMs, or the SOV accuracy using the external rules.

	<b>Q3 (%)</b>	<b>QH (%)</b>	<b>QE (%)</b>	<b>QC (%)</b>	<b>SOV</b>
<b>Fold0</b>	77.26	79.52	69.92	<b>79.12</b>	69.82
<b>Fold1</b>	76.12	74.02	68.01	79.02	70.76
<b>Fold2</b>	76.91	75.02	69.51	78.11	71.42
<b>Fold3</b>	77.01	79.23	69.12	76.72	71.31
<b>Fold4</b>	76.12	76.82	69.92	77.13	73.14
<b>Fold5</b>	75.94	78.91	68.11	75.92	70.75
<b>Fold6</b>	<b>77.41</b>	<b>79.33</b>	68.54	78.81	72.31
<b>Fold7</b>	76.22	77.61	<b>71.94</b>	76.03	<b>73.81</b>
<b>Fold8</b>	75.35	76.51	68.25	77.11	70.61
<b>Fold9</b>	76.82	79.14	70.12	75.15	72.12
<b>Average</b>	<b>76.52</b>	77.61	69.34	77.31	71.61

Table 5.14: The results of the 10-fold cross validation for the overall Q3 accuracy, the accuracy for each class and the overall SOV of each fold as well as the average results for all folds, after applying the SVM filtering

Applying the SVM filtering for each fold in the cross validation results, an increase of about 0.7% was achieved for the overall Q3 accuracy, while the SOV decreased by just under 0.2 (Table 5.14). It is important to find a balance between the overall accuracy of the network and the SOV accuracy, which are both significant.

	<b>Q3 (%)</b>	<b>QH (%)</b>	<b>QE (%)</b>	<b>QC (%)</b>	<b>SOV</b>
<b>Fold0</b>	76.91	79.81	69.52	<b>79.40</b>	70.51
<b>Fold1</b>	75.91	74.12	67.84	79.14	71.32
<b>Fold2</b>	76.42	75.32	69.47	78.33	71.99
<b>Fold3</b>	76.57	79.31	68.52	76.81	71.83
<b>Fold4</b>	76.01	76.89	69.81	77.17	73.51
<b>Fold5</b>	75.59	78.99	67.97	76.01	71.42
<b>Fold6</b>	<b>76.94</b>	<b>79.41</b>	68.01	78.91	72.83
<b>Fold7</b>	76.11	77.71	<b>71.21</b>	76.52	<b>74.01</b>
<b>Fold8</b>	75.22	76.71	67.58	77.23	71.04
<b>Fold9</b>	76.51	79.22	70.01	75.27	72.57
<b>Average</b>	<b>76.22</b>	77.75	68.99	77.48	72.1

Table 5.15: The results of the 10-fold cross validation for the overall Q3 accuracy, the accuracy for each class and the overall SOV of each fold as well as the average results for all folds, after applying the external rules.

Applying the external rules filtering for each fold in the cross validation results, an increase of about 0.5 was achieved for the overall SOV accuracy, while the Q3 decreased by just 0.3 (Table 5.15).

When deciding what is important in the final predictions of the secondary structure of a protein, whether that would be the raw accuracy overall or the individual accuracy of each class, or just the segment overlap, describing the general structure of the protein, it is critical to choose a suitable post processing filtering method. For the first cases, the SVM filter usually results in better overall predictions, while the empirical external rules are better for a better general structure instead of just individual amino acids.

Finally, the last method used to improve the performance was the ensembles. However, since there was not enough time to create multiple ensembles for each fold, the process was applied only for a single fold (Fold 0).

The ensemble files used for figure 5.18 were the results after using SVMs, while the files for figure 5.19 were the results after using the external rules.

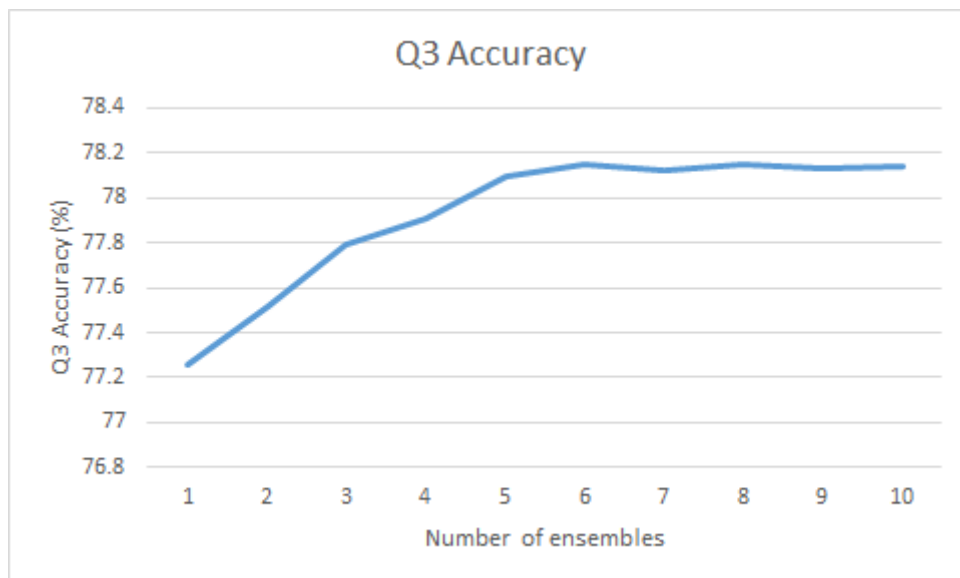


Figure 5.18: The graph of the Q3 accuracy, depending on the quantity of ensemble files used

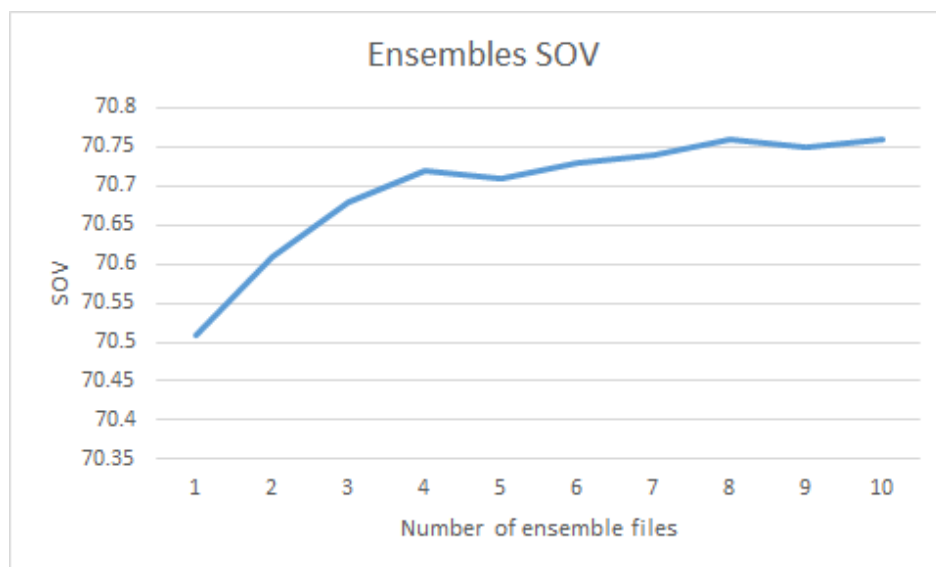


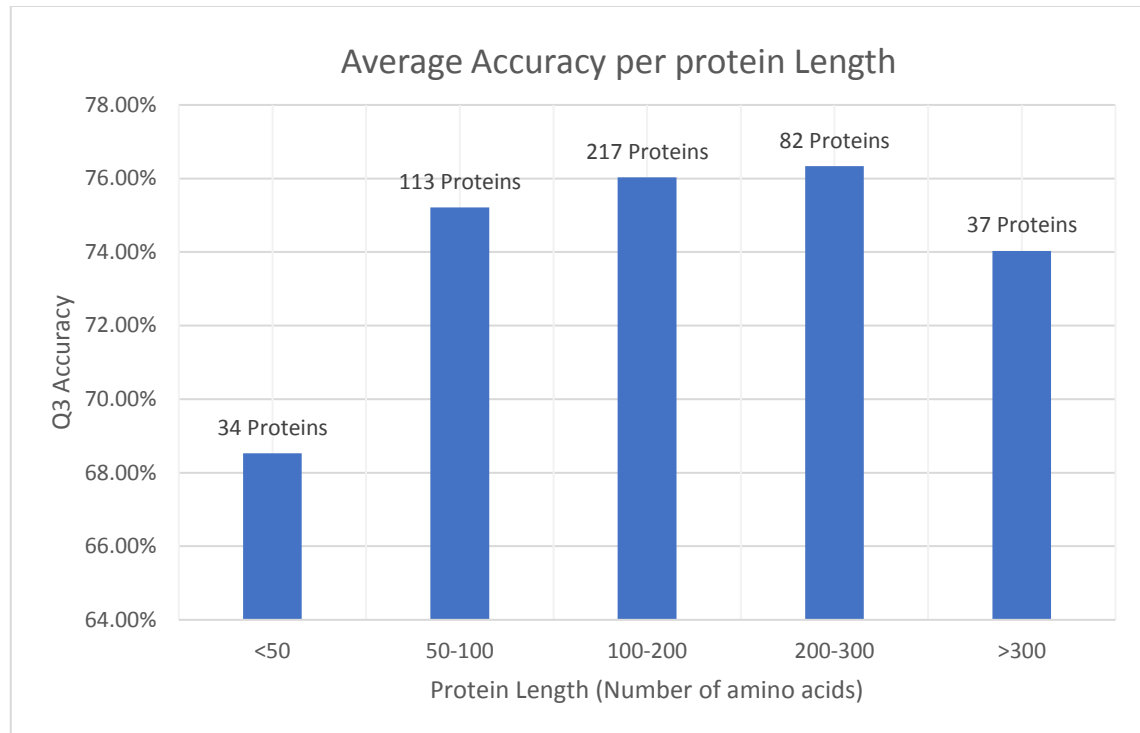
Figure 5.19: The graph of the SOV accuracy, depending on the quantity of ensemble files used

Observing figures 5.18 and 5.19, the accuracies in both cases improved for the first 4-6 ensemble files but then they remain relatively the same. This is important to highlight, since creating a huge set of files for ensembling purposes will not drastically improve the predictions, since the predicting model is the same, which more or less has the same predicting patterns at every completed training iteration. However, ensembles do pick up on random misclassifications, but ultimately only slightly improve the overall performance. In the case of this problem, ensembles improved the Q3 accuracy by about ~1% to 78.15 while the SOV accuracy by about ~0.25 to 70.76.

**Figure 5.20: The 10 worst-predicted proteins (Protein name / Q3 accuracy / Secondary structure / Predicted Secondary structure)**

**Figure 5.21: The 10 best-predicted proteins (Protein name / Q3 accuracy / Secondary structure / Predicted Secondary structure)**

found in both figures, which makes it hard to conclude in a specific reasoning behind why those specific proteins are the ones being the best/worst-predicted ones.



**Figure 5.22: The Average Q3 accuracy in terms of protein length.**

Finally, observing the average accuracy per protein length of Figure 5.22 it is clear that the network fails to predict accurately the proteins which have a smaller size (<50) at an average ~68.5% Q3 accuracy comparing to proteins which are larger (50-300) at an average ~76% Q3 accuracy. Interestingly, when proteins are significantly larger (>300) the accuracy falls about 2%, which does not necessarily mean that there could be a specific reasoning behind this, but it's probably due to the small sample of proteins (only 37). The fact that there is also a small sample for smaller proteins (34), however, does not justify the significantly lower accuracies of it too.

# Chapter 6

## Conclusion and Future Work

---

6.1 Conclusion	95
6.2 Future Work	96

---

## 6.1 Conclusion

The purpose of this dissertation was the problem of Protein Secondary Structure Prediction (PSSP), using learning algorithms that aim to predict the secondary structure of a protein based on its primary. The importance of this lies on the fact that the experimental methods and instruments that actually determine it are incredible costly, whereas learning methods are not. Moreover, being able to predict the functions of a protein through its secondary structure, enables the manufacturing of pharmaceutical drugs, food complements and antibiotics.

In this thesis, a Bidirectional Recurrent Neural Network (BRNN) was trained using the Hessian Free Optimization (HFO). The results taken were extremely promising at about **78.15%** Q3 accuracy using a single fold with ensembles and SVM filtering and about **76.52%** using 10-fold cross validation with SVM filtering but not ensembles, due to the high amounts of training time. Finally, the highest SOV score achieved for cross validation was **72.1** which is fairly decent. It is important to highlight that even though some of published results have higher accuracies (84-85%), they use different datasets, which are much larger than CB 513, which means that they are not completely comparable. However, there is still room for improvement, discussed in the following section.

	Q3 (%)	SOV	Q3 w/SVM(%)	Q3 w/ External Rules (%)	SOV w/SVM	SOV w/ External Rules	Q3 w/ Ensembles (%)	SOV w/ Ensembles
<b>10-fold Cross Validation</b>	75.58	71.78	<b>76.52</b>	76.22	71.61	<b>72.1</b>	-	-
<b>One fold</b>	76.81	70.01	77.26	76.91	69.82	70.51	<b>78.15</b>	<b>70.76</b>

Table 6.1 The final results of the dissertation

## 6.1 Future Work

Even though the results were fairly good, considering they were just 6-7% lower than the current best Q3 accuracies, there is still some margin for improvement.

First of all, the complete dataset used was split into ten folds for the purpose of the 10-fold cross validation. Therefore, necessarily, one of those folds was used to optimize the parameters for the model. However, the correct thing to do is to have a separate file for this process, containing proteins which are not included in any of the folds used for the cross validation. This is because in order to have an objective validation for the good generalization properties of the model, it is necessary to not have the optimized fold influence the final results of the cross validation. For this reason, the dataset should be split into 11 folds instead, where the addition fold should be used solely for optimizing the parameters and not be included in the cross validation.

Moreover, the results taken from feedforward networks were extremely promising, given their simple architecture and superior performance in execution time. Therefore, a proper methodology should be used to validate its performance using cross validation. Moreover, additional filtering, similar to the one used for BRNN in this dissertation (SVM, external empirical rules and ensembles), should also be applied to find the limits in the accuracies of the network. Finally, given the extremely promising results taken from Convolutional Neural Networks which used a simple Gradient descent optimizer for its feedforward network in the end (Διονυσίου, 2018), there is some possibility that even better results could be achieved if the HFO optimizer was used instead.

Due to lack of time and the extremely high amounts of training time required for BRNNs, no ensembling was done for the cross validation. This could potentially increase the overall accuracy by ~1-1.5%. Moreover, the averaging ensemble method used is the simplest form of ensembles. Using a more complex ensemble like AdaBoost (Rätsch et al., 2001) or Random Forest (Chan et al, 2008) could



also potentially increase the final performance of the network. Finally, in this thesis the ensemble files used were all derived from the same network trained with the same parameters. Combining different architecture trained with different learning algorithms which store different kinds of relations, could also increase the quality of the results.

The current implementation used for BRNN and HFO, required extremely large amounts of memory. This prevented the training of an existing, significantly larger dataset, derived from PISCES, which is a protein sequence culling server (Wang et al, 2003). In general, training a learning algorithm with a much larger dataset often results in better predictions. Consequently, it is important to try to optimize the implementation to require less memory or use a machine with higher memory capabilities in order to get results with HFO on a much larger dataset.

Finally, the benefits of using HFO lie on its superior performance in terms of execution times. In order to verify this, proper comparisons should be conducted with other second order algorithms like Scaled Conjugate Gradient, which are implemented on BRNN as well. (Agathokleous, 2016)

## References

- Agathocleous, M., Christodoulou, C., Promponas, V., Kountouris, P. and Vassiliades, V. (2016). Training Bidirectional Recurrent Neural Network Architectures with the Scaled Conjugate Gradient Algorithm. Artificial Neural Networks and Machine Learning - ICANN 2016, Lecture Notes in Computer Science, ed. by A. E. P. Villa, P. Masulli and A. J. P. Rivero, Springer-Verlag, 9886, pp. 123-131.
- Baldi, P., Brunak, S., Frasconi, P., Soda, G., & Pollastri, G. (1999) Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11), pp. 937-946.
- Bishop, C.M. (1995). Neural networks for pattern recognition. Oxford University Press, Oxford, UK
- Chan, J. C. W., & Paelinckx, D. (2008). Evaluation of Random Forest and Adaboost tree-based ensemble classification and spectral band selection for ecotope mapping using airborne hyperspectral imagery. *Remote Sensing of Environment*, 112(6), pp. 2999-3011.
- Charalambous, C. (1992). Conjugate gradient algorithm for efficient training of artificial neural networks. *Circuits, Devices and Systems, IEE Proceedings G*, 139(3), pp. 301-310
- Cristianini, N., & Shawe-Taylor, J. (2000). An introduction to Support Vector Machines. Cambridge University Press, New York, NY, USA
- Dor, O. and Zhou, Y. (2006). Achieving 80% ten-fold cross-validated accuracy for secondary structure prediction by large-scale training. *Proteins: Structure, Function, and Bioinformatics*, 66(4), pp.838-845.

Heffernan, R., Paliwal, K., Lyons, J., Dehzangi, A., Sharma, A., Wang, J., Sattar, A., Yang, Y. and Zhou, Y. (2015). Improving prediction of secondary structure, local backbone angles and solvent accessible surface area of proteins by iterative deep learning. *Scientific Reports*, 5, 11476.

Hornik, K., Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), pp.359-366.

Hush, D. R. and Salas, J. M., (1988). Improving the learning rate of back-propagation with the gradient reuse algorithm, *Proceedings of the IEEE 1988 International Conference on Neural Networks*, San Diego, CA, USA, vol.1, pp. 441-447

Jones, D. (1999). Protein secondary structure prediction based on position-specific scoring matrices. *Journal of Molecular Biology*, 292(2), pp.195-202.

Joosten, R., Te Beek, T., Krieger, E., Hekkelman, M., Hooft, R., Schneider, R., Sander, C. and Vriend, G. (2010). A series of PDB related databases for everyday needs. *Nucleic Acids Research*, 39(Database), pp.D411-D419.

Kabsch, W. and Sander, C. (1983). Dictionary of protein secondary structure: Pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers*, 22(12), pp.2577-2637.

LeCun, Y., Bengio, Y. and Hinton, G. (2015). Deep learning. *Nature*, 521(7553), pp.436-444.

Kountouris, P., Agathocleous, M., Promponas, V., Christodoulou, G., Hadjicostas, S., Vassiliades, V. and Christodoulou, C. (2012). A Comparative Study on Filtering Protein Secondary Structure Prediction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3), pp.731-739.

Magnan, C. and Baldi, P. (2014). SSpro/ACCpro 5: almost perfect prediction of protein secondary structure and relative solvent accessibility using profiles, machine learning and structural similarity. *Bioinformatics*, 30(18), pp.2592-2597.

Martens, J. (2010) Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*, Bottou, L., and Littman, M., (eds.), pp. 735-742.

Martens, J., Sutskever, I. (2012). Training Deep and Recurrent Networks with Hessian-Free Optimization. In: Montavon G., Orr G.B., Müller KR. (eds) *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science, vol 7700. Springer, Berlin, Heidelberg

Nash, S. G. (1984). Newton-type minimization via the Lanczos method. *SIAM Journal on Numerical Analysis*, 21(4), 770-788.

Nash, S.G. (2000). A survey of truncated-newton methods. *Journal of Computational and Applied Mathematics*, 124(1), pp 45–59.

Nocedal, J., & Write, S. SJ (1999). *Numerical optimization*. Springer-Verlag, New York

Pearlmutter, B.A. (1994). Fast exact multiplication by the hessian. *Neural Computation*, 6(1): pp.147–160.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing* (Second ed.). Cambridge University Press. Cambridge, MA

Rasmussen, D. (2015), Hessian-free optimization for deep networks, GitHub repository, <https://github.com/drasmuss/hessianfree>, May, 2018.

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), pp.7558-7562.

Rost, B., Sander, C. and Schneider, R. (1994). PHD-an automatic mail server for protein secondary structure prediction. *Bioinformatics*, 10(1), pp.53-60.

Rätsch, G., Onoda, T., & Müller, K. R. (2001). Soft margins for AdaBoost. *Machine learning*, 42(3), pp. 287-320.

Schraudolph, N. (2002). Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation*, 14(7), pp.1723-1738.

Wang, G., & Dunbrack Jr, R. L. (2003). PISCES: a protein sequence culling server. *Bioinformatics*, 19(12), pp. 1589-1591.

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

Wengert, RE. (1964). A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8), pp.463–464.

Xie, S., Li, Z. and Hu, H. (2018). Protein secondary structure prediction based on the fuzzy support vector machine with the hyperplane optimization. *Gene*, 642, pp.74-83.

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*, p.bbw129.

Zemla, A., Venclovas, Č., Fidelis, K., & Rost, B. (1999). A modified definition of Sov, a segment-based measure for protein secondary structure prediction

assessment. *Proteins: Structure, Function, and Bioinformatics*, 34(2), pp. 220-223.

Αγαθοκλέους Μ., (2009). Πρόβλεψη Δευτεροταγούς Δομής Πρωτεϊνών με την χρήση Νευρωνικών Δικτύων Αμφίδρομης Ανάδρασης, Προπτυχιακή διπλωματική, Τμήμα Πληροφορικής Πανεπιστήμιο Κύπρου, Λευκωσία.

Διονυσίου Α., (2018). Πρόβλεψη Δευτεροταγούς Δομής Πρωτεϊνών Με Χρήση Συνελκτικών Νευρωνικών Δικτύων Σε Συνδυασμό Με Φίλτρα Gabor Και Support Vector Machines, Προπτυχιακή διπλωματική, Τμήμα Πληροφορικής Πανεπιστήμιο Κύπρου, Λευκωσία.

Πετρίδου Ε., (2015) Πρόβλεψη Δευτεροταγούς Δομής Πρωτεϊνών, Μεταπτυχιακή διατριβή, Τμήμα Πληροφορικής Πανεπιστήμιο Κύπρου, Λευκωσία.

Χριστοδούλου, Γ. (2010). Διερεύνηση Μεθόδων Εκπαίδευσης Νευρωνικών Δικτύων Αμφίδρομης Ανάδρασης για Πρόβλεψη Δευτεροταγούς Δομής Πρωτεϊνών, Προπτυχιακή διπλωματική, Τμήμα Πληροφορικής Πανεπιστήμιο Κύπρου, Λευκωσία.

## Annex A

### BRNN.py

```
import hessianfree as hf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import storeProteins
import datetime
import os
import random
import pickle
import sys

def trainRNN(trainingData, trainingOutput, testingData, testingOutput,
window):

    optimizer = hf.opt.HessianFree(CG_iter=250, init_damping=45)
    rnn = hf.RNNNet(
        shape=[20 * window, 120, 3],
        layers=hf.nl.Logistic(),
        loss_type=[hf.loss_funcs.SquaredError(),
                    hf.loss_funcs.StructuralDamping(1e-4,
optimizer=optimizer)], debug=False,
        rng=np.random.RandomState(0))

    rnn.run_epochs(trainingData, trainingOutput,
optimizer=optimizer, test=(testingData,
testingOutput),
        max_epochs=100,
        test_err=hf.loss_funcs.SquaredError())

    return rnn

def trainFFN (trainingData, trainingOutput,testingData, testingOutput
, window, plots=True,seed=0,fold=0):

    ff = hf.FFNet([20 * window, 75, 3],
        layers=hf.nl.Logistic(),
        loss_type=hf.loss_funcs.SquaredError())
    pre = seed
    ff.run_epochs(trainingData, trainingOutput,
        optimizer=hf.opt.HessianFree(CG_iter=500,
init_damping=45),
        max_epochs=150, plotting=plots,
file_output="Results/Fold"+str(fold)+"/BRNN_PLOT_"+str(pre),test=(test
ingData, testingOutput))
    return ff

def trainBRNN(trainingData, trainingOutput,testingData, testingOutput,
window, plots=True,seed = 0,fold=0,d=45):
```

```

optimizer = hf.opt.HessianFree(CG_iter=500, init_damping=d)
brnn = hf.BRNNNet([20 * 3, 20 * 13, 20 * 3, 65, 30, 30, 15, 15, 3],
                  layers=hf.nl.Logistic(),
                  windowC=13, windowF=3, windowB=3, windowA=21,
                  loss_type=[hf.loss_funcs.SquaredError(),
                             hf.loss_funcs.StructuralDamping(1e-4,
optimizer=optimizer)],
                  conns={0: [4], 1: [3], 2: [5], 4: [6], 5: [7], 3:
[8], 6: [8], 7: [8]}},
                  rec_layers=[4, 5])

pre = seed
brnn.run_epochs(trainingData, trainingOutput,
               optimizer=optimizer,
               max_epochs=150, plotting=plots,
file_output="Results/Fold"+str(fold)+'damping ' + '{0:.5f}'.format(d)
+ "/BRNN_PLOT_"+str(pre), test=(testingData, testingOutput),
test_err=hf.loss_funcs.SquaredError(),)
return brnn

def getFFNAccuracy(testingData,testingOutput,ff,proteins, plots =
True,fold=0):
    outputs = ff.forward(testingData)[-1]
    accuracy = 0
    count = [0,0,0]
    countTotal = [0,0,0]
    correctProts = ['C', 'E', 'H']
    f = open("Results/Fold" + str(fold)+ "/" +
datetime.datetime.now().strftime("BRNN_%Y%m%d%H%M%S")+ '.txt', "w+")
    np.set_printoptions(precision=3, suppress=True)
    c = len(proteins[2])
    counter = 0
    pos =0
    for i in range(len(outputs)):
        max = outputs[i][0]
        index = 0
        prot = ""
        if (outputs[i][1] > max):
            max = outputs[i][1]
            index = 1
        if (outputs[i][2] > max):
            max = outputs[i][2]
            index = 2
        if testingOutput[i][index] == 1:
            accuracy +=1
            count[index] += 1
        prot += correctProts[index]
        counter += 1
        if counter == c:
            proteins[pos * 4 + 3] = prot
            prot = ""
            counter = 0
            pos += 1
            if i<len(outputs)-1:
                c = len(proteins[pos*4+2])
            f.write(str(outputs[i])[1:-1] + ' ' +
str(testingOutput[i])[1:-1] + '\n')
            countTotal[np.where(testingOutput[i]==1)[0][0]] =
countTotal[np.where(testingOutput[i]==1)[0][0]]+1
            acc = accuracy * 100 / (len(testingOutput))

```



```

print("Accuracy: ", accuracy * 100 / (len(testingOutput)), "%")
print("Accuracy Coil: ", count[0]*100.0/ countTotal[0], "%")
print("Accuracy E: ", count[1] * 100.0 / countTotal[1], "%")
print("Accuracy H: ", count[2] * 100.0 / countTotal[2], "%")
return acc,proteins

def getRNNAccuracy(testingData, testingOutput, rnn,
proteins,plots=True,fold=0,d = 45):
    outputs = rnn.forward(testingData)[-1]
    accuracy = 0
    count = [0,0,0]
    countTotal = [0,0,0]
    correctProts = ['C','E','H']
    f = open("Results/Fold" + str(fold)+ "/damping " + str(d) + '/' +
datetime.datetime.now().strftime("BRNN_%Y%m%d%H%M")+ '.txt', "w+")
    np.set_printoptions(precision=3, suppress=True)
    for i in range(len(outputs)):
        prot = ""
        for j in range(len(outputs[i])):
            if (1 not in testingOutput[i][j]):
                break;
            max = outputs[i][j][0]
            index = 0
            if (outputs[i][j][1] > max):
                max = outputs[i][j][1]
                index = 1
            if (outputs[i][j][2] > max):
                max = outputs[i][j][2]
                index = 2
            if testingOutput[i][j][index] == 1:
                accuracy += 1
                count[index] += 1
            if testingOutput[i][j][0] == 1:
                countTotal[0] +=1
            elif testingOutput[i][j][1] == 1:
                countTotal[1] +=1
            elif testingOutput[i][j][2] == 1:
                countTotal[2] +=1
            f.write(str(outputs[i][j])[1:-1] + ' ' +
str(testingOutput[i][j])[1:-1] + '\n')
            prot +=correctProts[index]
        proteins[i*4+3] = prot

    acc = accuracy * 100 / (countTotal[0]+countTotal[1]+countTotal[2])
    print("Accuracy: ", accuracy * 100 /
(countTotal[0]+countTotal[1]+countTotal[2]), "%")
    print("Accuracy Coil: ", (count[0]*100.0/ countTotal[0]) if
(countTotal[0] > 0) else 100, "%")
    print("Accuracy E: ", count[1] * 100.0 / countTotal[1] if
countTotal[1] >0 else 100, "%")
    print("Accuracy H: ", count[2] * 100.0 / countTotal[2] if
countTotal[2] >0 else 100, "%")

    return acc, proteins

def createSet(file>window,batch=None,cutoff=None):
    open_file = open(file, "r")

```

```

lines = open_file.readlines()

protObj = storeProteins.storeProteins()
size = 1
prot = lines[0:]
proteins = protObj.readProteins(prot, window - 1, batch, cutoff)
print("Helix: ", protObj.h)
print("Extended strand: ", protObj.e)
print("Coil: ", protObj.c)
print("Zeroes(padding): ", protObj.padding)
data = []
pos = 0
i = 0
temp = []
if cutoff is not None:
    batch = cutoff*20+(window-1)*20
    while i <= (len(protObj.data) - 20 * (window)):
        data.append(protObj.data[i:i + 20 * window])
        if (batch is not None and i> 0 and ((i+ (window) * 20 ) %
(batch*size) == 0)):
            i += int(window) * 20
            temp.append(data)
            data= []
            continue
        elif (protObj.endOfProtein[pos] == i + (window) * 20):
            pos += 1
            i += int(window) * 20
            continue;
        i += 20

if batch is None:
    data = np.array(data)
    output = np.array(protObj.yt)
else:
    output = []
    for i in range(0, len(protObj.yt), protObj.maxProtein*size):
        output.append(protObj.yt[i:i+protObj.maxProtein*size])
    output = np.array(output)
    if len(data) > 0:
        print(len(data))
        print(protObj.maxProtein*size - len(data))
        temp.append(data)
        print("remaining")
    data = np.array([np.array(t) for t in temp])
    return data, output, proteins

window = 21
#cb 513
batch_size = 753*20+(window-1)*20
cutoff = None

fold = 0
damping = 1

startTime = datetime.datetime.now().strftime("%H:%M %Y-%m-%d")
print(startTime)
pre = random.randint(1,1000000)

#These files for full dataset

```

```

testingData,testingOutput, proteinsTest = createSet("TestSets/testSet"
+ str(fold),window,batch_size,cutoff)
trainingData,trainingOutput, proteinsTrain =
createSet("TrainingSets/trainSet"+str(fold),window,batch_size,cutoff)

#These files for just one protein for testing
#trainingData,trainingOutput,proteinsTrain =
createSet("TrainingSets/msaProteinsTrainBigDataset_afterProcess.txt",w
indow,batch_size,cutoff)
#testingData,testingOutput, proteinsTest =
createSet("TrainingSets/msaProteinsTrainBigDataset_afterProcess.txt",w
indow,batch_size,cutoff)

#ffn = trainFFN(trainingData, trainingOutput, testingData,
testingOutput, window,seed=pre)
#rnn = trainRNN(trainingData, trainingOutput, testingData,
testingOutput, window)
rnn = trainBRNN(trainingData, trainingOutput, testingData,
testingOutput, window,seed=pre,fold=fold,d = damping)

#Set the weights to the weight of the best epoch for ffn
'''ffn.W = ffn.best_W.copy()
np.save("Results/Fold" + str(fold) + "/damping " + str(damping) + '/'
+ str(pre) + "_weights.npy", ffn.W)
with open("Results/Fold" + str(fold) + "/damping " + str(damping) +
'/' + "/" + str(pre) + "_brnn_settings.pkl", "wb") as f:
    pickle.dump(ffn.shape, f)
accTest,proteinsTest = getFFNAccuracy(testingData, testingOutput, ffn,
proteinsTest,fold=fold)
accTrain, proteinsTrain= getFFNAccuracy(trainingData, trainingOutput,
ffn, proteinsTrain,fold=fold)
endTime = datetime.datetime.now().strftime("%H:%M %Y-%m-%d")

print(endTime)
outputs = ffn.forward(trainingData)[-1]
outputs2 = ffn.forward(testingData)[-1]'''

#Set the weights to the weight of the best epoch for rnn
rnn.W = rnn.best_W.copy()
np.save("Results/Fold" + str(fold) + "/damping " +
'{:.5f}'.format(damping) + '/' + str(pre) + "_weights.npy", rnn.W)
with open("Results/Fold" + str(fold) + "/damping " +
'{0:.5f}'.format(damping) + '/' + "/" + str(pre) +
"_brnn_settings.pkl", "wb") as f:
    pickle.dump(rnn.shape, f)

accTest,proteinsTest = getRNNAccuracy(testingData, testingOutput, rnn,
proteinsTest,fold=fold,d=damping)
accTrain, proteinsTrain= getRNNAccuracy(trainingData, trainingOutput,
rnn, proteinsTrain,fold=fold,d=damping)
endTime = datetime.datetime.now().strftime("%H:%M %Y-%m-%d")
print(endTime)
outputs = rnn.forward(trainingData)[-1]
outputs2 = rnn.forward(testingData)[-1]

# Save predictions

```

```

f= open('Results/Fold' + str(fold) + "/damping " +
'{0:.5f}'.format(damping) + '/' + 'train-' + str(accTrain) + str(pre)
+ ".txt", "w+")
f2= open('Results/Fold' + str(fold) + "/damping " +
'{0:.5f}'.format(damping) + '/' + 'test-' + str(accTest) + str(pre)
+ ".txt", "w+")

for p in proteinsTrain:
    f.write(p + '\n')

for p in proteinsTest:
    f2.write(p + '\n')

```

## Annex B

### storeProteins.py

```
##
# Read data from file and organize them into input
# and target output
##
from numpy import *

class storeProteins:

    def __init__(self):
        self.sizeOfAminoacids = 0 # for training data
        self.sizeOfCleanAminoacids = 0 # for targets
        self.aminoacids = []
        self.data = []
        self.yt = []
        self.maxProtein = 753
        self.h = 0
        self.c = 0
        self.e = 0
        self.extra = 0
        self.padding = 0
        self.endOfProtein = []

    ##
    # Read data from file and finds there MSA representation. Adds
    zeros, equal to the window size, at the beginning
    # and the end of the protein. Returns an array with the output
    format of the program
    # (Protein name\nPrimary Structure\nCorrect secondary structure\n
    +\n) where + is to be replaced with the predicted
    # secondary structure. If the sequence has unknown symbols i.e.
    '!' it removes them from the sequence.
    ##
    def readProteins(self, lines, window, batch_size=None, cutoff=None):
        wrongProteins = ['lcoiA_1-29', 'lmctI_1-28', 'ltiiC_195-
230', '2erlA_1-40', 'lceoA_202-254', 'lmrtA_31-61', 'lwfbB_1-37', '6rlxC_
2-20']

        proteins = []
        leadingZeros = zeros((1, window * 10)) # half window frond and
back
        if cutoff is not None:
            self.maxProtein = cutoff
        for i in range(0, len(lines), 3):

            name = lines[i].rstrip()
            if (name in wrongProteins):
                print (name)
                continue;
            proteins.append(name)

            secondary = lines[i + 2].rstrip()
            proteins.append(lines[i + 1].rstrip().replace('!', '')) #
protein's first structure
```

```

        proteins.append(secondary.replace('!', '')) # protein's
secondary structure
        proteins.append('') # protein's predicted secondary
structure which will be added later in the + field

        try:
            msa = loadtxt("msaFiles/" + name + '.hssp')
        except FileNotFoundError:
            print(name, " protein not found")

        # followingZeros = zeros((1, (753-(len(msa))) * 20)) #
padding

        protein = (msa * 1.0) / 100.0

        if batch_size is not None:
            protein = concatenate((protein, tile(zeros(20),
(self.maxProtein-len(protein),1))))
            if cutoff is not None:
                self.extra = cutoff - (len(protein) % cutoff)

                protein = concatenate((protein,tile(zeros(20),(cutoff
- (len(protein) % cutoff),1))))
                #protein = append(protein, zeros(20)*(cutoff -
(len(protein) % cutoff)))
                temp = protein.copy()
                for i in range(cutoff, len(protein), cutoff):
                    temp = concatenate((temp[:i + int(i / cutoff - 1)
* window], protein[i - int(window / 2):i],
temp[i + int(i / cutoff - 1) *
window:i + int(i / cutoff - 1) * window + int(window / 2)],
temp[i + int(i / cutoff - 1) *
window:]))
                    #temp = concatenate((temp[:i+int(i/cutoff-
1)*window],protein[i-int(window/2):i] , temp[i+int(i/cutoff-
1)*window:]))
                protein = temp

        # Placing half windows on the front and the back of a
protein makes the predicted amino acid being on the center of each
window

        self.data = append(self.data, leadingZeros)
        self.data = append(self.data, protein)
        self.data = append(self.data, leadingZeros)

        #self.data = append(self.data, leadingZeros)
        #self.data = append(self.data, leadingZeros)
        if cutoff is None:
            self.endOfProtein = append(self.endOfProtein,
len(self.data))
        else:
            for i in range(cutoff,len(protein),cutoff):
                self.endOfProtein = append(self.endOfProtein,
len(self.data-len(protein)+i))
            # self.data = append(self.data, followingZeros) # padding

        self.sizeOfAminoacids += len(msa) + window
        # self.sizeOfAminoacids += 753 + window
        self.sizeOfCleanAminoacids += len(msa)

```

```

        # self.sizeOfCleanAminoacids += 753
        self.normalizeOutput(secondary, window, batch_size, cutoff)
        # self.normalizeOutput(secondary, window, followingZeros)

    return proteins

##
# Converts the secondary structure class in to neuron activation.
##
def normalizeOutput(self, secondary,
window, batch_size=None, cutoff=None):

    se = secondary.split()

    for i in range(0, window, 1):
        # self.yt.append([0, 0, 0])
        self.aminoacids.append("10")
    count = 0
    for t in se[0]:

        if t == "C":
            self.yt.append([1, 0, 0])
            self.c += 1
            self.aminoacids.append(t)

        elif t == "E":
            self.yt.append([0, 1, 0])
            self.e += 1
            self.aminoacids.append(t)

        elif t == "H":
            self.yt.append([0, 0, 1])
            self.h += 1
            self.aminoacids.append(t)

        else:
            count += 1
            continue # ignores if there is a ! or anything except
the three above

    if batch_size is not None:
        self.padding += self.maxProtein-len(se[0])+count
        for i in range(self.maxProtein-len(se[0])+count):
            self.yt.append([0,0,0])

    if cutoff is not None:
        for i in range(self.extra):
            self.yt.append([0,0,0])

```

## Annex C

### Bffnet.py

```
"""Implementation of feedforward network, including Gauss-Newton
approximation
for use in Hessian-free optimization.

.. codeauthor:: Daniel Rasmussen
<daniel.rasmussen@appliedbrainresearch.com>

Based on
Martens, J. (2010). Deep learning via Hessian-free optimization. In
Proceedings
of the 27th International Conference on Machine Learning.
"""

from __future__ import print_function

from collections import defaultdict, OrderedDict
import pickle
import warnings

import numpy as np
import math
import hessianfree as hf

class BFFNet(object):
    """Implementation of feed-forward network (including
    gradient/curvature
    computation).

    :param list shape: the number of neurons in each layer
    :param layers: nonlinearity to use in the network (or a list
    giving a
        nonlinearity for each layer)
    :type layers: :class:`~.nonlinearities.Nonlinearity` or `list`
    :param dict conns: dictionary of the form `{layer_x:[layer_y,
    layer_z],
        ...}` specifying the connections between layers (default is to
        connect in series)
    :param loss_type: loss function (or list of loss functions) used
    to
        evaluate network
    :type loss_type: :class:`~.loss_funcs.LossFunction` or `list`
    :param dict W_init_params: parameters passed to
    :meth:`~.init_weights`
        (see parameter descriptions in that function)
    :param bool use_GPU: run curvature computation on GPU (requires
        PyCUDA and scikit-cuda)
    :param load_weights: load initial weights from given array or
    filename
    :type load_weights: `str` or :class:`~numpy:numpy.ndarray`
    :param bool debug: activates expensive features to help with
    debugging
    :param rng: used to generate any random numbers for this network
    (use
```



```

        this to control the seed)
:type rng: :class:`~numpy:numpy.random.RandomState`
:param dtype: floating point precision used throughout the network
:type dtype: :class:`~numpy:numpy.dtype`
"""

def __init__(self, shape, layers=hf.nl.Logistic(), conns=None,
              loss_type=hf.loss_funcs.SquaredError(),
W_init_params=None,
              use_GPU=False, load_weights=None, debug=False,
rng=None,
              dtype=np.float64):

    self.debug = debug
    self.shape = shape
    self.n_layers = len(shape)
    self.dtype = np.float64 if debug else dtype
    self.mask = None
    self._optimizer = None
    self.rng = np.random.RandomState() if rng is None else rng

    # note: this isn't used internally, it is just here so that an
    # external process with a handle to this object can tell what
epoch
    # it is on
    self.epoch = None

    self.inputs = None
    self.targets = None
    self.activations = None
    self.d_activations = None

    # initialize layer nonlinearities
    if not isinstance(layers, (list, tuple)):
        if isinstance(layers, hf.nl.Nonlinearity) and
layers.stateful:
            warnings.warn("Multiple layers sharing stateful
nonlinearity, "
                           "consider creating a separate instance
for each "
                           "layer.")
        layers = [layers for _ in range(self.n_layers)]
        layers[0] = layers[1] = layers[2] = hf.nl.Linear()

    if len(layers) != len(shape):
        raise ValueError("Number of nonlinearities (%d) does not
match "
                           "number of layers (%d)" %
                           (len(layers), len(shape)))

    self.layers = []
    for t in layers:
        if isinstance(t, str):
            # look up the nonlinearity with the given name
            t = getattr(hf.nl, t)()
        if not isinstance(t, hf.nl.Nonlinearity):
            raise TypeError("Layer type (%s) must be an instance
of "
                           "nonlinearities.Nonlinearity" % t)
        self.layers += [t]

```

```

# initialize loss function
self.init_loss(loss_type)

# initialize connections
if conns is None:
    # set up the feedforward series connections
    conns = {}
    for pre, post in zip(np.arange(self.n_layers - 1),
                        np.arange(1, self.n_layers)):
        conns[pre] = [post]

self.conns = OrderedDict(sorted(conns.items(), key=lambda x:
x[0]))
# note: conns is an ordered dict sorted by layer so that we
can
# reliably loop over the items (in compute_offsets and
init_weights)

# maintain a list of backwards connections as well (for
efficient
# lookup in the other direction)
self.back_conns = defaultdict(list)
for pre in conns:
    for post in conns[pre]:
        self.back_conns[post] += [pre]

        if pre >= post:
            raise ValueError("Can only connect from lower to
higher "
                             "layers (%s >= %s)" % (pre,
post))

# add empty connection for first/last layer (just helps smooth
the code
# elsewhere)
self.conns[self.n_layers - 1] = []
self.back_conns[0] = []

# compute indices for the different connection weight matrices
in the
# overall parameter vector
self.compute_offsets()

# initialize connection weights
if load_weights is None:
    if W_init_params is None:
        W_init_params = {}
        self.W = self.init_weights(
            [(self.shape[pre], self.shape[post])
             for pre in self.conns for post in self.conns[pre]],
            **W_init_params)
else:
    if isinstance(load_weights, np.ndarray):
        self.W = load_weights
    else:
        # load weights from file
        self.W = np.load(load_weights)

    if len(self.W) != np.max(list(self.offsets.values())):

```

```

        raise IndexError(
            "Length of loaded weights (%s) does not match
expected "
            "length (%s)" % (len(self.W),
np.max(list(self.offsets.values()))))

        if self.W.dtype != self.dtype:
            raise TypeError("Loaded weights dtype (%s) doesn't
match "
                            "self.dtype (%s)" % (self.W.dtype,
self.dtype))

        # initialize GPU
        if use_GPU:
            try:
                import pycuda
                import skcuda
            except Exception as e:
                print(e)
                raise ImportError("PyCuda/scikit-cuda not installed. "
                                "Set use_GPU=False.")

        hf.gpu.init_kernels()

        self.use_GPU = use_GPU

    def run_epochs(self, inputs, targets, optimizer,
                   max_epochs=100, minibatch_size=None, test=None,
                   test_err=None, target_err=1e-6, plotting=False,
                   file_output=None, print_period=1):
        """Apply the given optimizer with a sequence of (mini)batches.

        :param inputs: input vectors (or a
:~.nonlinearities.Plant` that
            will generate the input vectors dynamically)
        :type inputs: :class:`~numpy:numpy.ndarray` or
            :class:`~.nonlinearities.Plant`
        :param targets: target vectors corresponding to each input
vector (or
            None if a plant is being used)
        :type targets: :class:`~numpy:numpy.ndarray`
        :param optimizer: computes the weight update each epoch (see
            optimizers.py)
        :param int max_epochs: the maximum number of epochs to run
        :param int minibatch_size: the size of the minibatch to use in
each epoch
            (or None to use full batches)
        :param tuple test: tuple of (inputs,targets) to use as the
test data
            (if None then the same inputs and targets as training will
be used)
        :param test_err: a custom error function to be applied to
            the test data (e.g., classification error)
        :type test_err: :class:`~.loss_funcs.LossFunction`
        :param float target_err: run will terminate if this test error
is
            reached
        :param str file_output: output files from the run will use
this as a

```

```

        prefix (if None then don't output files)
        :param bool plotting: if True then data from the run will be
output to
        a file, which can be displayed via dataplotter.py
        :param int print_period: print out information about the run
every `x`
        epochs
        """

        test_errs = []
        self.best_W = None
        self.best_error = None

        # compute test error
        if test is None:
            test_in, test_t = inputs, targets
        else:
            test_in, test_t = test[0], test[1]

        prefix = "HF" if file_output is None else file_output
        minibatch_size = minibatch_size or inputs.shape[0]
        plots = defaultdict(list)
        self.optimizer = optimizer

        if isinstance(optimizer, hf.opt.SCG):
            if test_err is None:
                err = self.error(self.W, test_in, test_t)
            else:
                output = self.forward(test_in, self.W)
                err = test_err.batch_loss(output, test_t)
            test_errs += [err]
        for i in range(max_epochs):
            self.epoch = i
            printing = print_period is not None and (i % print_period
== 0 or

                                                    self.debug)

            #printing = False;
            print("=" * 40)
            print("epoch", i)

            # run minibatches
            indices = self.rng.permutation(inputs.shape[0])
            #indices = range(inputs.shape[0])
            for start in range(0, inputs.shape[0], minibatch_size):
                # generate minibatch and cache activations
                self.cache_minibatch(
                    inputs, targets, indices[start:start +
minibatch_size])

                # validity checks
                if self.inputs.shape[-1] != self.winA*20:
                    raise ValueError(
                        "Input dimension (%d) does not match number of
input "
                        "nodes (%d)" % (self.inputs.shape[-1],
self.shape[0]))
                if self.targets.shape[-1] != self.shape[-1]:
                    raise ValueError(
                        "Target dimension (%d) does not match number
of "

```

```

        "output nodes (%d)" % (self.targets.shape[-1],
                                self.shape[-1]))

    assert self.activations[-1].dtype == self.dtype
    # compute update
    print(prefix)
    update = optimizer.compute_update(printing, prefix)
    # compute update SCG
    # update =
optimizer.compute_update(printing, inputs, targets, indices[start:start +
minibatch_size])
    assert update.dtype == self.dtype

    # apply mask
    if self.mask is not None:
        update[self.mask] = 0

    # update weights
    self.W += update

    # invalidate cached activations (shouldn't be
necessary,
    # but doesn't hurt)
    #self.activations = None
    #self.d_activations = None
    #self.GPU_activations = None

    if test_err is None:
        err = self.error(self.W, test_in, test_t)
    else:
        output = self.forward(test_in, self.W)
        err = test_err.batch_loss(output, test_t)
    test_errs += [err]
    if isinstance(optimizer, hf.opt.SCG):
        print("Object of SCG")
        comparison = 2 * optimizer.delta * (test_errs[-2] -
test_errs[-1]) / np.power(optimizer.phi, 2)
        print(test_errs[-2], test_errs[-1])
        if comparison >= 0:
            if test_errs[-1] < target_err:
                break # done!

        vector = np.copy(self.W)
        f_old = test_errs[-1]
        self.activations, self.d_activations =
self.forward(inputs[indices[start:start + minibatch_size]],
self.W,
deriv=True)
        self.activations = [np.asarray(a,
dtype=self.dtype)
                            for a in self.activations]
        self.d_activations = [np.asarray(a,
dtype=self.dtype)
                              for a in
self.d_activations]
        self.r_new = -self.calc_grad()

```

```

        self.success = True
        self.lamb_ = 0

        if (i+1) % self.W.shape[0] == 0:
            optimizer.grad_new = optimizer.r_new
        else:
            beta = (np.dot(optimizer.r_new,
optimizer.r_new) - np.dot(optimizer.r_new, optimizer.r)) /
optimizer.phi
            optimizer.grad_new = optimizer.r_new + beta *
optimizer.grad

        if comparison > 0.75:
            optimizer.lamb = 0.5 * optimizer.lamb
        else:
            optimizer.lamb_ = optimizer.lamb
            # end

        if comparison < 0.25:
            optimizer.lamb = 4 * optimizer.lamb
    if printing:
        print("test error", test_errs[-1])
    print("test error", test_errs[-1])
    # save the weights with the best error
    if self.best_W is None or test_errs[-1] < self.best_error:
        self.best_W = self.W.copy()
        self.best_error = test_errs[-1]

    # dump plot data
    if plotting:
        plots["update norm"] += [np.linalg.norm(update)]
        plots["W norm"] += [np.linalg.norm(self.W)]
        plots["test error (log)"] += [test_errs[-1]]

        if hasattr(optimizer, "plots"):
            plots.update(optimizer.plots)

        with open("%s_plots.pkl" % prefix, "wb") as f:
            pickle.dump(plots, f)

    # dump weights
    '''if file_output is not None:
        np.save("%s_weights.npy" % prefix, self.W)'''

    # check for termination
    if test_errs[-1] < target_err:
        if print_period is not None:
            print("target error reached")
            break
    if test is not None and i > 10 and test_errs[-5] <
test_errs[-1]:
        if print_period is not None:
            print("overfitting detected, terminating")
            break

def forward(self, inputs, params=None, deriv=False):
    """Compute layer activations for given input and parameters.

    :param inputs: input vectors (passed to first layer)

```

```

        :type inputs: :class:`~numpy:numpy.ndarray`
        :param params: parameter vector (weights) for the network
        (defaults to
            ``self.W``)
        :type params: :class:`~numpy:numpy.ndarray`
        :param bool deriv: if True then also compute the derivative of
the
            activations
        """

        params = self.W if params is None else params

        if isinstance(inputs, hf.nl.Plant):
            inputs.reset()

        activations = [None for _ in range(self.n_layers)]
        if deriv:
            d_activations = [None for _ in range(self.n_layers)]
            original = inputs.copy()
            for i in range(self.n_layers):
                if i == 0:
                    if isinstance(inputs, hf.nl.Plant):
                        inputs = inputs(None)
                    else:
                        inputs = original[:,0:self.shape[0]]
                elif i == 1:
                    inputs = original[:,self.shape[0]:self.shape[0] +
self.shape[1]]
                elif i==2:
                    inputs = original[:,self.shape[0] +
self.shape[1]:self.shape[0] + self.shape[1] + self.shape[2]]
                else:
                    inputs = np.zeros((inputs.shape[0], self.shape[i]),
dtype=self.dtype)
                    for pre in self.back_conns[i]:
                        W, b = self.get_weights(params, (pre, i))
                        inputs += np.dot(activations[pre], W)
                        inputs += b
                        # note: we're applying a bias on each connection
to a
                        # neuron (rather than one for each neuron). just
because
                        # it's easier than tracking how many connections
there are
                        # for each layer (but we could do it if it becomes
                        # important).
                    activations[i] = self.layers[i].activation(inputs)

                if deriv:
                    d_activations[i] = self.layers[i].d_activation(inputs,
activations[i])
            for i, a in enumerate(activations):
                if not np.all(np.isfinite(a)):
                    raise OverflowError("Non-finite nonlinearity
activation "
                                "value (layer %d) \n %s" %
                                (i, a[not np.isfinite(a)]))

        if deriv:
            return activations, d_activations

```

```

        return activations

def error(self, W=None, inputs=None, targets=None):
    """Compute network error.

    :param W: network parameters (defaults to ``self.W``)
    :type W: :class:`~numpy:numpy.ndarray`
    :param inputs: input vectors (defaults to the cached
(mini)batch for
        current epoch)
    :type inputs: :class:`~numpy:numpy.ndarray`
    :param targets: target vectors (defaults to the cached
(mini)batch for
        current epoch)
    :type targets: :class:`~numpy:numpy.ndarray`
    """

    W = self.W if W is None else W
    inputs = self.inputs if inputs is None else inputs

    # get outputs
    if (W is self.W and inputs is self.inputs and
        self.activations is not None):
        # use cached activations
        activations = self.activations
    else:
        # compute activations
        activations = self.forward(inputs, W)

    # get targets
    if isinstance(inputs, hf.nl.Plant):
        # get targets from plant
        targets = inputs.get_vecs()[1]
    else:
        targets = self.targets if targets is None else targets

    # note: np.nan can be used in the target to specify places
    # where the target is not defined. those get translated to
    # zero error in the loss function.
    error = self.loss.batch_loss(activations, targets)

    return error

def cache_minibatch(self, inputs, targets, minibatch=None):
    """Pick a subset of inputs and targets to use in minibatch,
and cache
    the activations for that minibatch."""

    if minibatch is None:
        minibatch = np.arange(inputs.shape[0])

    if not isinstance(inputs, hf.nl.Plant):
        # inputs/targets are vectors
        self.inputs = inputs[minibatch]
        self.targets = targets[minibatch]

        # cache activations
        self.activations, self.d_activations =
self.forward(self.inputs,

```



```

self.W,

deriv=True)
    else:
        # input is a dynamic plant
        if targets is not None:
            raise ValueError("Cannot specify targets when using
dynamic "
                                "plant to generate inputs (plant
should "
                                "generate targets itself)")

        # run plant to generate batch
        inputs.shape[0] = len(minibatch)
        self.activations, self.d_activations =
self.forward(inputs, self.W,

deriv=True)
        self.inputs, self.targets = inputs.get_vecs()

        # cast to self.dtype
        if self.inputs.dtype != self.dtype:
            warnings.warn("Input dtype (%s) not equal to self.dtype
(%s)" %
                            (self.inputs.dtype, self.dtype))
        self.inputs = np.asarray(self.inputs, dtype=self.dtype)
        self.targets = np.asarray(self.targets, dtype=self.dtype)
        self.activations = [np.asarray(a, dtype=self.dtype)
                             for a in self.activations]
        self.d_activations = [np.asarray(a, dtype=self.dtype)
                               for a in self.d_activations]
        self.d2_loss = self.loss.d2_loss(self.activations,
self.targets)

        # allocate temporary space for intermediate values, to save on
        # memory allocations
        self.tmp_space = [np.zeros(a.shape, self.dtype)
                           for a in self.activations]

        if self.use_GPU:
            # TODO: we could just allocate these on the first timestep
and
            # then do a copy rather than an allocation after that, if
this
            # ever became a significant part of the computation time
            self.load_GPU_data()

def load_GPU_data(self):
    """Load data for the current epoch onto GPU."""

    from pycuda import gpuarray

    # clear out old data (this would happen eventually on its own,
but by
    # doing it first we make sure there is room on the GPU before
    # creating new arrays)
    if hasattr(self, "GPU_W"):
        del self.GPU_W
        del self.GPU_activations

```

```

        del self.GPU_d_activations
        del self.GPU_d2_loss
        del self.GPU_tmp_space

self.GPU_W = gpuarray.to_gpu(self.W)
self.GPU_activations = [gpuarray.to_gpu(a)
                        for a in self.activations]
self.GPU_d_activations = [gpuarray.to_gpu(a)
                        for a in self.d_activations]
self.GPU_d2_loss = [gpuarray.to_gpu(a) if a is not None else
None
                    for a in self.d2_loss]
self.GPU_tmp_space = [gpuarray.empty(a.shape, self.dtype)
                    for a in self.activations]

@staticmethod
def J_dot(J, vec, transpose_J=False, out=None):
    """Compute the product of a Jacobian and some vector."""

    # In many cases the Jacobian is a diagonal matrix, so it is
    # efficient to just represent it with the diagonal vector.
    # function just lets those two be used interchangeably.

    if J.ndim == 2:
        # note: the first dimension is the batch, so ndim==2 means
        # this is a vector representation
        if out is None:
            # passing out=None fails for some reason
            return np.multiply(J, vec)
        else:
            return np.multiply(J, vec, out=out)
    else:
        if transpose_J:
            J = np.transpose(J, (0, 2, 1))

        if out is None:
            # passing out=None fails for some reason
            return np.einsum("ijk,ik->ij", J, vec)

        if out is vec:
            tmp_vec = vec.copy()
        else:
            tmp_vec = vec

        return np.einsum("ijk,ik->ij", J, tmp_vec, out=out)

def calc_grad(self):
    """Compute parameter gradient."""

    for l in self.layers:
        if l.stateful:
            raise TypeError("Cannot use neurons with internal
state in "
                           "a one-step feedforward network; use "
                           "RNNet instead.")

    grad = np.zeros_like(self.W)

```

```

# backpropagation
# note: this uses the cached activations, so the forward
# pass has already been run elsewhere

# compute output error for each layer
error = self.loss.d_loss(self.activations, self.targets)

error = [np.zeros_like(self.activations[i]) if e is None else
e
          for i, e in enumerate(error)]

deltas = [np.zeros_like(a) for a in self.activations]
#change deltas 0 to reflect the new weights
# backwards pass
for i in range(self.n_layers - 1, -1, -1):
    for post in self.conns[i]:
        error[i] += np.dot(deltas[post],
                           self.get_weights(self.W, (i,
post))) [0].T)

        W_grad, b_grad = self.get_weights(grad, (i, post))
        np.dot(self.activations[i].T, deltas[post],
out=W_grad)
        np.sum(deltas[post], axis=0, out=b_grad)

        if i > 0:
            self.J_dot(self.d_activations[i], error[i],
transpose_J=True,
                        out=deltas[i])

grad /= self.inputs.shape[0]

return grad

def check_grad(self, calc_grad):
    """Check gradient via finite differences (for debugging)."""

    eps = 1e-6
    grad = np.zeros_like(calc_grad)
    inc_W = np.zeros_like(self.W)
    for i in range(len(self.W)):
        inc_W[i] = eps

        error_inc = self.error(self.W + inc_W, self.inputs,
self.targets)
        error_dec = self.error(self.W - inc_W, self.inputs,
self.targets)
        grad[i] = (error_inc - error_dec) / (2 * eps)

        inc_W[i] = 0
    try:
        assert np.allclose(calc_grad, grad, rtol=1e-3)
    except AssertionError:
        print("calc_grad")
        print(calc_grad)
        print("finite grad")
        print(grad)
        print("calc_grad - finite grad")
        print(calc_grad - grad)
        print("calc_grad / finite grad")

```

```

        print(calc_grad / grad)
        input("Paused (press enter to continue)")

def calc_G(self, v, damping=0, out=None):
    """Compute Gauss-Newton matrix-vector product."""

    if out is None:
        Gv = np.zeros(self.W.size, dtype=self.dtype)
    else:
        Gv = out
        Gv.fill(0)

    # R forward pass
    R_activations = [np.zeros_like(a) for a in self.activations]
    for i in range(1, self.n_layers):
        for pre in self.back_conns[i]:
            vw, vb = self.get_weights(v, (pre, i))
            Ww, _ = self.get_weights(self.W, (pre, i))

            R_activations[i] += np.dot(self.activations[pre], vw,
                                       out=self.tmp_space[i])
            R_activations[i] += vb
            R_activations[i] += np.dot(R_activations[pre], Ww,
                                       out=self.tmp_space[i])

        self.J_dot(self.d_activations[i], R_activations[i],
                  out=R_activations[i])

    # backward pass
    R_error = R_activations

    for i in range(self.n_layers - 1, -1, -1):
        if self.d2_loss[i] is not None:
            # note: R_error[i] is already set to R_activations[i]
            R_error[i] *= self.d2_loss[i]
        else:
            R_error[i].fill(0)

        for post in self.conns[i]:
            W, _ = self.get_weights(self.W, (i, post))

            R_error[i] += np.dot(R_error[post], W.T,
                               out=self.tmp_space[i])

            W_g, b_g = self.get_weights(Gv, (i, post))
            np.dot(self.activations[i].T, R_error[post], out=W_g)
            np.sum(R_error[post], axis=0, out=b_g)

        if i > 0:
            self.J_dot(self.d_activations[i], R_error[i],
                      out=R_error[i], transpose_J=True)

    Gv /= len(self.inputs)

    Gv += damping * v # Tikhonov damping

    return Gv

def GPU_calc_G(self, v, damping=0, out=None):
    """Compute Gauss-Newton matrix-vector product on GPU."""

```

```

from pycuda import gpuarray

if out is None or not isinstance(out, gpuarray.GPUArray):
    Gv = gpuarray.zeros(self.W.shape, self.dtype)
else:
    Gv = out
    Gv.fill(0)

if not isinstance(v, gpuarray.GPUArray):
    GPU_v = gpuarray.to_gpu(v)
else:
    GPU_v = v

# R forward pass
R_activations = self.GPU_tmp_space

for i in range(self.n_layers):
    R_activations[i].fill(0)
    for pre in self.back_conns[i]:
        vw, vb = self.get_weights(GPU_v, (pre, i))
        Ww, _ = self.get_weights(self.GPU_W, (pre, i))

        hf.gpu.dot(self.GPU_activations[pre], vw,
                    out=R_activations[i], increment=True)
        hf.gpu.iadd(R_activations[i], vb)
        hf.gpu.dot(R_activations[pre], Ww,
                    out=R_activations[i], increment=True)

    hf.gpu.J_dot(self.GPU_d_activations[i], R_activations[i],
                  out=R_activations[i])

# backward pass
R_error = R_activations

for i in range(self.n_layers - 1, -1, -1):
    if self.GPU_d2_loss[i] is not None:
        # note: R_error[i] is already set to R_activations[i]
        R_error[i] *= self.GPU_d2_loss[i]
    else:
        R_error[i].fill(0)

    for post in self.conns[i]:
        W, _ = self.get_weights(self.GPU_W, (i, post))
        W_g, b_g = self.get_weights(Gv, (i, post))

        hf.gpu.dot(R_error[post], W, transpose_b=True,
                    out=R_error[i], increment=True)

        hf.gpu.dot(self.GPU_activations[i], R_error[post],
                    transpose_a=True, out=W_g)

        hf.gpu.sum_cols(R_error[post], out=b_g)

    if i > 0:
        hf.gpu.J_dot(self.GPU_d_activations[i], R_error[i],
                      out=R_error[i], transpose_J=True)

# Tikhonov damping and batch mean
Gv._axpbyz(1.0 / len(self.inputs), GPU_v, damping, Gv)

```

```

        if isinstance(v, gpuarray.GPUArray):
            return Gv
        else:
            return Gv.get(out, pagelocked=True)

    def check_J(self):
        """Compute the Jacobian of the network via finite
        differences."""

        eps = 1e-6
        N = self.W.size

        # compute the Jacobian
        J = [None for _ in self.layers]
        inc_i = np.zeros_like(self.W)
        for i in range(N):
            inc_i[i] = eps

            inc = self.forward(self.inputs, self.W + inc_i)
            dec = self.forward(self.inputs, self.W - inc_i)

            for l in range(self.n_layers):
                J_i = (inc[l] - dec[l]) / (2 * eps)
                if J[l] is None:
                    J[l] = J_i[... , None]
                else:
                    J[l] = np.concatenate((J[l], J_i[... , None]),
axis=-1)

            inc_i[i] = 0

        return J

    def check_G(self, calc_G, v, damping=0):
        """Check Gv calculation via finite differences (for
        debugging)."""

        # compute Jacobian
        J = self.check_J()

        # second derivative of loss function
        L = self.loss.d2_loss(self.activations, self.targets)
        # TODO: check loss via finite differences

        G = np.sum([np.einsum("aji,aj,ajk->ik", J[l], L[l], J[l])
                    for l in range(self.n_layers) if L[l] is not
None], axis=0)

        # divide by batch size
        G /= self.inputs.shape[0]

        Gv = np.dot(G, v)
        Gv += damping * v

        try:
            assert np.allclose(calc_G, Gv, rtol=1e-3)
        except AssertionError:
            print("calc_G")
            print(calc_G)

```

```

        print("finite G")
        print(Gv)
        print("calc_G - finite G")
        print(calc_G - Gv)
        print("calc_G / finite G")
        print(calc_G / Gv)
        input("Paused (press enter to continue)")

    def init_weights(self, shapes, coeff=1.e-4, biases=0,
init_type="sparse"):
        """Weight initialization, given shapes of weight matrices.

        Note: coeff, biases, and init_type can be specified by the
        `W_init_params` dict in :class:`.FFNet`. Each can be
        specified as a single value (for all matrices) or as a list
        giving a
        value for each matrix.

        :param list shapes: list of (pre,post) shapes for each weight
matrix
        :param float coeff: scales the magnitude of the connection
weights
        :param float biases: bias values for the post of each matrix
        :param str init_type: type of initialization to use (currently
supports
            'sparse', 'uniform', 'gaussian')
        """
        # if given single parameters, expand for all matrices
        if isinstance(coeff, (int, float)):
            coeff = [coeff] * len(shapes)
        if isinstance(biases, (int, float)):
            biases = [biases] * len(shapes)
        if isinstance(init_type, str):
            init_type = [init_type] * len(shapes)

        W = [np.zeros((pre + 1, post), dtype=self.dtype)
            for pre, post in shapes]

        for i, s in enumerate(shapes):
            if init_type[i] == "sparse":
                # sparse initialization (from martens)
                num_conn = 15

                for j in range(s[1]):
                    # pick num_conn random pre neurons
                    indices = self.rng.choice(np.arange(s[0]),
                        size=min(num_conn,
s[0]),
                        replace=False)

                    # connect to post
                    W[i][indices, j] = self.rng.randn(indices.size) *
coeff[i]
                elif init_type[i] == "uniform":
                    W[i][:,1] = self.rng.uniform(-coeff[i] /
np.sqrt(s[0]),
                    coeff[i] / np.sqrt(s[0]),
                    (s[0], s[1]))
                elif init_type[i] == "gaussian":

```

```

        W[i][:-1] = self.rng.randn(s[0], s[1]) * coeff[i]
    else:
        raise ValueError("Unknown weight initialization (%s)"
                          % init_type)

    # set biases
    W[i][-1, :] = biases[i]

W = np.concatenate([w.flatten() for w in W])

return W

def compute_offsets(self):
    """Precompute offsets for layers in the overall parameter
    vector."""

    self.offsets = {}
    offset = 0
    for pre in self.conns:
        for post in self.conns[pre]:
            n_params = (self.shape[pre] + 1) * self.shape[post]
            self.offsets[(pre, post)] = (
                offset,
                offset + n_params - self.shape[post],
                offset + n_params)
            offset += n_params

    return offset

def get_weights(self, params, conn):
    """Get weight matrix for a connection from overall parameter
    vector."""

    if conn not in self.offsets:
        return None

    offset, W_end, b_end = self.offsets[conn]
    W = params[offset:W_end]
    b = params[W_end:b_end]
    '''if (conn[0] == 0):
        return W.reshape((self.shape[conn[1]],
self.shape[conn[1]])), b
    else:'''
    return W.reshape((self.shape[conn[0]], self.shape[conn[1]])),
b

def init_loss(self, loss_type):
    """Set the loss type for this network to the given
    :class:`~.loss_funcs.LossFunction` (or a list of functions can
    be
    passed to create a :class:`~.loss_funcs.LossSet`)."""

    if isinstance(loss_type, (list, tuple)):
        tmp = loss_type
    else:
        tmp = [loss_type]

    for t in tmp:
        if not isinstance(t, hf.loss_funcs.LossFunction):

```



```

        raise TypeError("loss_type (%s) must be an instance of
"
                        "LossFunction" % t)

    # sanity checks
    if (isinstance(t, hf.loss_funcs.CrossEntropy) and
        np.any(self.layers[-1].activation(
            np.linspace(-80, 80, 100)[None, :]) <= 0)):
        # this won't catch everything, but hopefully a useful
warning
        raise ValueError("Must use positive activation
function "
                        "with cross-entropy error")
    if (isinstance(t, hf.loss_funcs.CrossEntropy) and
        not isinstance(self.layers[-1], hf.nl.Softmax)):
        warnings.warn("Softmax should probably be used with "
                        "cross-entropy error")

    if isinstance(loss_type, (list, tuple)):
        self.loss = hf.loss_funcs.LossSet(loss_type)
    else:
        self.loss = loss_type

    def run_epoch_SCG(self, inputs, targets, optimizer,
                      max_epochs=100, minibatch_size=None, test=None,
                      test_err=None, target_err=1e-6, plotting=False,
                      file_output=None, print_period=1):
update
        """A stripped down version of run_epochs that just does the
        without any overhead.

        Can be used for optimizers where the cost to compute an update
is
        very cheap, in which case the overhead (e.g., computing test
error,
        saving weights, outputting data for plotting, etc.) becomes
        non-negligible.
        """

        if test is None:
            test_in, test_t = inputs, targets
        else:
            test_in, test_t = test[0], test[1]

        sigma0 = 1.e-6
        lamb = 1.e-6
        lamb_ = 0
        vector = self.W
        self.cache_minibatch(
            inputs, targets, range(inputs.shape[0]))
        grad_new = -self.calc_grad()
        r_new = grad_new
        success = True
        for i in range(max_epochs):
            r = np.copy(r_new)
            grad = np.copy(grad_new)
            mu = np.dot(grad, grad)
            if success:
                success = False
                sigma = sigma0 / math.sqrt(mu)

```

```

        temp = self.calc_grad()
        self.W = vector + sigma*grad
        self.activations, self.d_activations =
self.forward(inputs,
vector + sigma * grad,
deriv=True)

        self.activations = [np.asarray(a, dtype=self.dtype)
                             for a in self.activations]
        self.d_activations = [np.asarray(a, dtype=self.dtype)
                               for a in self.d_activations]
        s = (self.calc_grad() - temp) / sigma
        delta = np.dot(grad.T, s)

#end
    # scale s
    zetta = lamb - lamb_
    s += zetta * grad
    delta += zetta * mu

    if delta < 0:
        s += (lamb - 2 * delta / mu) * grad
        lamb_ = 2 * (lamb - delta / mu)
        delta -= lamb * mu
        delta *= -1
        lamb = lamb_
    # end

    phi = np.dot(grad.T, r)
    alpha = phi / delta

    vector_new = vector +alpha*grad
    output = self.forward(test_in, vector)
    f_old = test_err.batch_loss(output, test_t)

    output = self.forward(test_in, vector_new)
    f_new = test_err.batch_loss(output, test_t)
    print("epoch ",i,":",f_new)
    comparison = 2 * delta * (f_old - f_new) / np.power(phi,
2)

    if comparison >= 0:
        if f_new < target_err:
            break # done!

        vector = vector_new
        f_old = f_new
        self.W = vector
        self.activations, self.d_activations =
self.forward(inputs,
vector,
deriv=True)

        self.activations = [np.asarray(a, dtype=self.dtype)
                             for a in self.activations]
        self.d_activations = [np.asarray(a, dtype=self.dtype)
                               for a in self.d_activations]
        r_new = -self.calc_grad()

```

```

        success = True
        lamb_ = 0

        if (i+1) % self.W.shape[0] == 0:
            grad_new = r_new
        else:
            beta = (np.dot(r_new, r_new) - np.dot(r_new, r)) /

phi            grad_new = r_new + beta * grad

            if comparison > 0.75:
                lamb = 0.5 * lamb
            else:
                lamb_ = lamb
            # end

            if comparison < 0.25:
                lamb = 4 * lamb

            # compute update

        self.W = vector_new

@property
def optimizer(self):
    return self._optimizer

@optimizer.setter
def optimizer(self, o):
    self._optimizer = o
    o.net = self

```

## Annex D

### Brnnet.py

```
"""Implementation of recurrent network, including Gauss-Newton
approximation
for use in Hessian-free optimization.

.. codeauthor:: Daniel Rasmussen
<daniel.rasmussen@appliedbrainresearch.com>

Based on
Martens, J., & Sutskever, I. (2011). Learning recurrent neural
networks with
hessian-free optimization. Proceedings of the 28th International
Conference on
Machine Learning.
"""

from __future__ import print_function

import numpy as np

import hessianfree as hf

class BRNNet(hf.BFFNet):
    """Implementation of recurrent deep network (including
    gradient/curvature
    computation).

    :param list rec_layers: indices of layers with recurrent
    connections
    (default is to make all except first and last layers
    recurrent)
    :param dict W_rec_params: parameters used to initialize recurrent
    weights (passed to :meth:`~.init_weights`)
    :param tuple truncation: a tuple `(n,k)` where backpropagation
    through
    time will be executed every `n` timesteps and run backwards
    for `k`
    steps (defaults to full backprop if None)

    See :class:`~.FFNet` for the remaining parameters."""

    def __init__(self, shape, rec_layers=None, W_rec_params=None,
        windowC=0, windowB=0, windowF=0, windowA=0,
        truncation=None, **kwargs):

        # define recurrence for each layer (needs to be done before
        super
        # constructor because this is used in compute_offsets)
        if rec_layers is None:
            # assume all recurrent except first/last layer
            rec_layers = np.arange(1, len(shape) - 1)
        self.rec_layers = rec_layers
        self.winC = windowC
        self.winB = windowB
```

```

self.winF = windowF
self.winA = windowA
# super constructor
super(BRNNNet, self).__init__(shape, **kwargs)

self.truncation = truncation

# add on recurrent weights
if kwargs.get("load_weights", None) is None and
len(rec_layers) > 0:
    if W_rec_params is None:
        W_rec_params = dict()
    self.W = np.concatenate(
        (self.W, self.init_weights([(self.shape[1+2],
self.shape[1])
                                for l in
range(self.n_layers)
                                if l in rec_layers],
                                **W_rec_params)))

def forward(self, inputs, params=None, deriv=False,
init_activations=None,
init_state=None):
    """Compute layer activations for given input and parameters.

    :param inputs: input vectors (passed to first layer)
    :type inputs: :class:`~numpy:numpy.ndarray`
    :param params: parameter vector (weights) for the network
    (defaults to
    ``self.W``)
    :type params: :class:`~numpy:numpy.ndarray`
    :param bool deriv: if True then also compute the derivative of
the
    activations
    :param list init_activations: initial values for the
activations in
    each layer
    :param list init_state: initial values for the internal state
of any
    stateful nonlinearities
    """

    # input shape = [minibatch_size, seq_len, input_dim]
    # activations shape = [n_layers, minibatch_size, seq_len,
layer_size]

    params = self.W if params is None else params
    if isinstance(inputs, hf.nl.Plant):
        # reset the plant
        # TODO: allow the initial state of plant to be set?
        inputs.reset()

    batch_size = inputs.shape[0]
    sig_len = inputs.shape[1]
    activations = [np.zeros((batch_size, sig_len, 1),
dtype=self.dtype)
                    for l in self.shape]

    # temporary space to minimize memory allocations

```

```

tmp_space = [np.zeros((batch_size, 1), dtype=self.dtype)
              for l in self.shape]

if deriv:
    d_activations = [None for _ in self.layers]

for i, l in enumerate(self.layers):
    # reset any state in the nonlinearities
    l.reset(None if init_state is None else init_state[i])

W_recs = [self.get_weights(params, (i+2, i))
           for i in range(self.n_layers)]
#original = inputs.copy()
for s in range(sig_len):
    for w in range(int(self.winA/2)):
        for i in range(self.n_layers):

            if i == 0:
                ff_input = inputs[:,s,w*20:w*20+self.shape[0]]
            elif i == 1:
                #ff_input = original[:, s,
self.shape[0]:self.shape[0] + self.shape[1]]
                ff_input = inputs[:, s, 20*(int(self.winA/2)-
int(self.winC/2)):20*(int(self.winA/2)+int(self.winC/2)+1)]
            elif i == 2:
                ff_input = inputs[:, s, (self.winA-
(w+self.winB)) * 20:(self.winA-w) * 20]
                #ff_input = np.flip(ff_input, 1)
            else:
                # compute feedforward input
                ff_input = np.zeros_like(activations[i][:, s])
                for pre in self.back_conns[i]:
                    W, b = self.get_weights(params, (pre, i))

                    ff_input += np.dot(activations[pre][:, s],
W,
                                out=tmp_space[i])

                    ff_input += b

            # recurrent input
            if i in self.rec_layers:
                if s > 0:
                    rec_input = np.dot(activations[i+2][:, s -
1]*0.5,
                                W_recs[i][0],
                                out=tmp_space[i])

                elif init_activations is None:
                    # apply bias input on first timestep
                    rec_input = W_recs[i][1]
                else:
                    # use the provided activations to
initialize the
                    # 'previous' timestep
                    rec_input =
np.dot(init_activations[i+2]*0.5,
                                W_recs[i][0],
                                out=tmp_space[i])

            else:
                rec_input = 0
                # apply activation function

```

```

        activations[i][:, s] =
self.layers[i].activation(ff_input +
rec_input)
        # compute derivative
        if deriv:
            d_act = self.layers[i].d_activation(ff_input +
rec_input,
activations[i][:, s])
        if d_activations[i] is None:
            # note: we can't allocate this array ahead
            # because we don't know if d_activations
            # returning diagonal vectors or matrices
            d_activations[i] = np.zeros(
                np.concatenate(([batch_size],
                                d_act.shape[1:]))),
                dtype=self.dtype)
            d_activations[i][:, s] = d_act

    for i, a in enumerate(activations):
        if not np.all(np.isfinite(a)):
            raise OverflowError("Non-finite nonlinearity
activation "
                                "value (layer %d) \n %s" %
                                (i, a[not np.isfinite(a)]))

    if deriv:
        return activations, d_activations

    return activations

def calc_grad(self):
    """Compute parameter gradient."""

    grad = np.zeros_like(self.W)
    W_recs = [self.get_weights(self.W, (l+2, 1))
               for l in range(self.n_layers)]
    batch_size = self.inputs.shape[0]
    sig_len = self.inputs.shape[1]

    # temporary space to minimize memory allocations
    tmp_act = [np.zeros((batch_size, 1), dtype=self.dtype)
               for l in self.shape]
    tmp_grad = np.zeros_like(grad)

    if self.truncation is None:
        trunc_per = trunc_len = sig_len
    else:
        trunc_per, trunc_len = self.truncation

    for n in range(trunc_per - 1, sig_len, trunc_per):
        # every trunc_per timesteps we want to run backprop

        deltas = [np.zeros((batch_size, 1), dtype=self.dtype)
                  for l in self.shape]
        state_deltas = [None if not l.stateful else

```

```

        np.zeros((batch_size, self.shape[i]),
                  dtype=self.dtype)
        for i, l in enumerate(self.layers)]

    # backpropagate error
    for s in range(n, np.maximum(n - trunc_len, -1), -1):
        # execute trunc_len steps of backprop through time

        error = self.loss.d_loss([a[:, s] for a in
self.activations],
                                self.targets[:, s])
        error = [np.zeros_like(self.activations[i][:, s]) if e
is None
                  else e for i, e in enumerate(error)]

        for l in range(self.n_layers - 1, -1, -1):
            for post in self.conns[l]:
                error[l] += np.dot(deltas[post],
                                self.get_weights(self.W,
                                                    (l,
post)) [0].T,
                                out=tmp_act[l])

                # feedforward gradient
                W_grad, b_grad = self.get_weights(grad, (l,
post))
                W_tmp_grad, b_tmp_grad =
self.get_weights(tmp_grad,
                                (l,
post))
                W_grad += np.dot(self.activations[l][:, s].T,
                                deltas[post], out=W_tmp_grad)
                b_grad += np.sum(deltas[post], axis=0,
out=b_tmp_grad)

            # add recurrent error
            if l-2 in self.rec_layers:
                error[l] += np.dot(deltas[l-2], W_recs[l-
2] [0].T,
                                out=tmp_act[l])

            # compute deltas
            if not self.layers[l].stateful:
                self.J_dot(self.d_activations[l][:, s],
error[l],
                                transpose_J=True, out=deltas[l])
            else:
                d_input = self.d_activations[l][:, s, ..., 0]
                d_state = self.d_activations[l][:, s, ..., 1]
                d_output = self.d_activations[l][:, s, ..., 2]

                state_deltas[l] += self.J_dot(d_output,
error[l],
                                transpose_J=True,
                                out=tmp_act[l])
                self.J_dot(d_input, state_deltas[l],
transpose_J=True,
                                out=deltas[l])

```



```

        self.J_dot(d_state, state_deltas[l],
transpose_J=True,
                    out=state_deltas[l])

        # gradient for recurrent weights
        if l-2 in self.rec_layers:
            W_grad, b_grad = self.get_weights(grad, (l, l-
2))

            W_tmp_grad, b_tmp_grad =
self.get_weights(tmp_grad,
                    (l,
l-2))

            if s > 0:
                W_grad += np.dot(self.activations[l][:, s
- 1].T,
                    deltas[l-2],
out=W_tmp_grad)

            else:
                # put remaining gradient into initial bias
                b_grad += np.sum(deltas[l-2], axis=0,
                    out=b_tmp_grad)

        grad /= batch_size

    return grad

def check_grad(self, calc_grad):
    """Check gradient via finite differences (for debugging)."""

    eps = 1e-6
    grad = np.zeros_like(calc_grad)

    sig_len = self.inputs.shape[1]
    if self.truncation is None:
        trunc_per = trunc_len = sig_len
    else:
        trunc_per, trunc_len = self.truncation

    inc_W = np.zeros_like(self.W)

    for n in range(trunc_per, sig_len + 1, trunc_per):
        start = np.maximum(n - trunc_len, 0)

        # the truncated backprop gradient is equivalent to running
the
        # network normally for the initial timesteps and then just
changing
        # the parameters for the truncation period.  so that's
what we're
        # simulating here.
        if start > 0:
            prev = self.forward(self.inputs[:, :start], self.W)
            init_a = [p[:, -1] for p in prev]
            init_s = [l.state.copy() if l.stateful else None
                for l in self.layers]
        else:
            init_a = None
            init_s = None

        for i in range(len(self.W)):

```

```

        inc_W[i] = eps

        out_inc = self.forward(self.inputs[:, start:n], self.W
+ inc_W,
                                init_activations=init_a,
                                init_state=init_s)
        out_dec = self.forward(self.inputs[:, start:n], self.W
- inc_W,
                                init_activations=init_a,
                                init_state=init_s)

        error_inc = self.loss.batch_loss(out_inc,
                                          self.targets[:,
start:n])

        error_dec = self.loss.batch_loss(out_dec,
                                          self.targets[:,
start:n])

        grad[i] += (error_inc - error_dec) / (2 * eps)

        inc_W[i] = 0

    try:
        assert np.allclose(calc_grad, grad, rtol=1e-3)
    except AssertionError:
        print("calc_grad")
        print(calc_grad)
        print("finite grad")
        print(grad)
        print("calc_grad - finite grad")
        print(calc_grad - grad)
        print("calc_grad / finite grad")
        print(calc_grad / grad)
        input("Paused (press enter to continue)")

def calc_G(self, v, damping=0, out=None):
    """Compute Gauss-Newton matrix-vector product."""

    if out is None:
        Gv = np.zeros(self.W.size, dtype=self.dtype)
    else:
        Gv = out
        Gv.fill(0)

    batch_size = self.inputs.shape[0]
    sig_len = self.inputs.shape[1]

    # temporary space to minimize memory allocations
    tmp_act = [np.zeros((batch_size, l), dtype=self.dtype)
                for l in self.shape]
    tmp_grad = np.zeros_like(Gv)

    # R forward pass
    R_states = [None if not l.stateful else
                np.zeros((batch_size, self.shape[i]),
dtype=self.dtype)
                for i, l in enumerate(self.layers)]
    R_activations = self.tmp_space
    for a in R_activations:

```

```

        a.fill(0)

    v_recs = [self.get_weights(v, (l+2, 1))
               for l in range(self.n_layers)]
    W_recs = [self.get_weights(self.W, (l+2, 1))
               for l in range(self.n_layers)]
    Gv_recs = [self.get_weights(Gv, (l+2, 1))
                for l in range(self.n_layers)]
    v_ff = dict([(conn, self.get_weights(v, conn))
                  for conn in self.offsets])
    W_ff = dict([(conn, self.get_weights(self.W, conn))
                  for conn in self.offsets])
    Gv_ff = dict([(conn, self.get_weights(Gv, conn))
                  for conn in self.offsets])

    for s in range(sig_len):
        for l in range(self.n_layers):
            R_act = R_activations[l][:, s]

            # input from feedforward connections
            for pre in self.back_conns[l]:
                vw, vb = v_ff[(pre, l)]
                Ww, _ = W_ff[(pre, l)]

                R_act += np.dot(self.activations[pre][:, s], vw,
                                out=tmp_act[l])
                R_act += vb
                R_act += np.dot(R_activations[pre][:, s], Ww,
                                out=tmp_act[l])

            # recurrent input
            if l in self.rec_layers:
                if s == 0:
                    # bias input on first step
                    R_act += v_recs[l][1]
                else:
                    R_act += np.dot(self.activations[l+2][:, s -
1],
                                v_recs[l][0], out=tmp_act[l])
                    R_act += np.dot(R_activations[l+2][:, s - 1],
                                W_recs[l][0], out=tmp_act[l])

            if not self.layers[l].stateful:
                self.J_dot(self.d_activations[l][:, s], R_act,
out=R_act)
            else:
                d_input = self.d_activations[l][:, s, ..., 0]
                d_state = self.d_activations[l][:, s, ..., 1]
                d_output = self.d_activations[l][:, s, ..., 2]

                R_states[l] = self.J_dot(d_state, R_states[l])

                R_states[l] += self.J_dot(d_input, R_act,
out=tmp_act[l])
                self.J_dot(d_output, R_states[l], out=R_act)

        # R backward pass
        if self.truncation is None:
            trunc_per = trunc_len = sig_len
        else:

```

```

trunc_per, trunc_len = self.truncation

R_error = [np.zeros((batch_size, 1), dtype=self.dtype)
            for l in self.shape]
R_deltas = [np.zeros((batch_size, 1), dtype=self.dtype)
            for l in self.shape]

for n in range(trunc_per - 1, sig_len, trunc_per):
    for i in range(self.n_layers):
        R_deltas[i].fill(0)
        if R_states[i] is not None:
            R_states[i].fill(0)

    for s in range(n, np.maximum(n - trunc_len, -1), -1):
        for l in range(self.n_layers - 1, -1, -1):
            if self.d2_loss[l] is not None:
                np.multiply(self.d2_loss[l][:, s],
                           R_activations[l][:, s],
                           out=R_error[l])
            else:
                R_error[l].fill(0)

            # error from feedforward connections
            for post in self.conns[l]:
                R_error[l] += np.dot(R_deltas[post],
                                     W_ff[(l, post)][0].T,
                                     out=tmp_act[l])

            # feedforward gradient
            W_g, b_g = Gv_ff[(l, post)]
            W_tmp_grad, b_tmp_grad =

self.get_weights(tmp_grad,
                 (l,
                 post))

            W_g += np.dot(self.activations[l][:, s].T,
                          R_deltas[post], out=W_tmp_grad)
            b_g += np.sum(R_deltas[post], axis=0,

out=b_tmp_grad)

            # add recurrent error
            if l-2 in self.rec_layers:
                R_error[l] += np.dot(R_deltas[l-2], W_recs[l-
2][0].T,
                                     out=tmp_act[l])

            # compute deltas
            if not self.layers[l].stateful:
                self.J_dot(self.d_activations[l][:, s],
                R_error[l],
                           transpose_J=True, out=R_deltas[l])
            else:
                d_input = self.d_activations[l][:, s, ..., 0]
                d_state = self.d_activations[l][:, s, ..., 1]
                d_output = self.d_activations[l][:, s, ..., 2]

                R_states[l] += self.J_dot(d_output,
                R_error[l],
                           transpose_J=True,
                           out=tmp_act[l])

```

```

        self.J_dot(d_input, R_states[l],
transpose_J=True,
                    out=R_deltas[l])
        self.J_dot(d_state, R_states[l],
transpose_J=True,
                    out=R_states[l])

        # recurrent gradient
        if l-2 in self.rec_layers:
            W_g, b_g = Gv_recs[l-2]
            W_tmp_grad, b_tmp_grad =
self.get_weights(tmp_grad,
                    (l,
l-2))

            if s > 0:
                W_g += np.dot(self.activations[l][:, s -
1].T,
                    R_deltas[l-2],
out=W_tmp_grad)

            else:
                b_g += np.sum(R_deltas[l-2], axis=0,
out=b_tmp_grad)

        Gv /= batch_size

        Gv += damping * v # Tikhonov damping

        return Gv

def load_GPU_data(self):
    """Load data for the current epoch onto GPU."""

    from pycuda import gpuarray

    def split_axes(array, n=1):
        # split a multidimensional array into a corresponding list
of lists
        # along the first n axes (this is used so that
array.__getitem__
        # isn't called repeatedly, as it is somewhat expensive for
# gpuarrays)
        if n == 1:
            return [a for a in array]

        return [split_axes(a, n - 1) for a in array]

    # clear out old data (this would happen eventually on its own,
but by
    # doing it first we make sure there is room on the GPU before
# creating new arrays)
    if hasattr(self, "GPU_W"):
        del self.GPU_W
        del self.GPU_activations
        del self.GPU_d_activations
        del self.GPU_d2_loss
        del self.GPU_tmp_space
        del self.GPU_states
        del self.GPU_errors
        del self.GPU_deltas

```

```

self.GPU_W = gpuarray.to_gpu(self.W)

# rearrange GPU data so that signal is the first axis (so
# that each time step is a single block of memory in
GPU_calc_G)
self.GPU_activations = [
    split_axes(gpuarray.to_gpu(np.ascontiguousarray(
        np.swapaxes(a, 0, 1))), 1)
    for a in self.activations]

self.GPU_d_activations = [
    split_axes(gpuarray.to_gpu(np.ascontiguousarray(
        np.rollaxis(np.swapaxes(a, 0, 1), -1, 1))), 2)
    if self.layers[i].stateful else
    split_axes(gpuarray.to_gpu(np.ascontiguousarray(
        np.swapaxes(a, 0, 1))), 1)
    for i, a in enumerate(self.d_activations)]

self.GPU_d2_loss = [
    split_axes(gpuarray.to_gpu(np.ascontiguousarray(
        np.swapaxes(a, 0, 1))), 1)
    if a is not None else None for a in self.d2_loss]

self.GPU_tmp_space = [split_axes(gpuarray.empty((a.shape[1],
                                                    a.shape[0],
                                                    a.shape[2]),
                                                    self.dtype),
1)
                        for a in self.activations]

# pre-allocate calc_G arrays
batch_size = self.inputs.shape[0]
self.GPU_states = [[gpuarray.empty((batch_size,
self.shape[i]),
                                dtype=self.dtype) for _ in
range(2)]

                        if l.stateful else None
                        for i, l in enumerate(self.layers)]
self.GPU_errors = [gpuarray.empty((batch_size, 1),
                                dtype=self.dtype)
                    for l in self.shape]
self.GPU_deltas = [gpuarray.empty((batch_size, 1),
                                dtype=self.dtype)
                    for l in self.shape]

def GPU_calc_G(self, v, damping=0, out=None):
    """Compute Gauss-Newton matrix-vector product on GPU."""

    from pycuda import gpuarray

    if out is None or not isinstance(out, gpuarray.GPUArray):
        Gv = gpuarray.zeros(self.W.shape, dtype=self.dtype)
    else:
        Gv = out
        Gv.fill(0)

    if not isinstance(v, gpuarray.GPUArray):
        GPU_v = gpuarray.to_gpu(v)
    else:
        GPU_v = v

```

```

batch_size = self.inputs.shape[0]
sig_len = self.inputs.shape[1]

# R forward pass
R_states = self.GPU_states
R_activations = self.GPU_tmp_space
for i in range(self.n_layers):
    R_activations[i][0].base.fill(0)
    if R_states[i] is not None:
        R_states[i][0].fill(0)

v_recs = [self.get_weights(GPU_v, (1, 1))
           for l in range(self.n_layers)]
W_recs = [self.get_weights(self.GPU_W, (1, 1))
           for l in range(self.n_layers)]
Gv_recs = [self.get_weights(Gv, (1, 1))
            for l in range(self.n_layers)]
v_ff = dict([(conn, self.get_weights(GPU_v, conn))
              for conn in self.offsets])
W_ff = dict([(conn, self.get_weights(self.GPU_W, conn))
              for conn in self.offsets])
Gv_ff = dict([(conn, self.get_weights(Gv, conn))
               for conn in self.offsets])

for s in range(sig_len):
    for l in range(self.n_layers):
        R_act = R_activations[l][s]

        # input from feedforward connections
        for pre in self.back_conns[l]:
            vw, vb = v_ff[(pre, l)]
            hf.gpu.dot(self.GPU_activations[pre][s], vw,
                       out=R_act, increment=True)
            hf.gpu.iadd(R_act, vb)
            hf.gpu.dot(R_activations[pre][s], W_ff[(pre,
1)][0],
                       out=R_act, increment=True)

        # recurrent input
        if l in self.rec_layers:
            if s == 0:
                # bias input on first step
                hf.gpu.iadd(R_act, v_recs[l][1])
            else:
                hf.gpu.dot(self.GPU_activations[l][s - 1],
                           v_recs[l][0], out=R_act,
increment=True)
                hf.gpu.dot(R_activations[l][s - 1],
W_recs[l][0],
                           out=R_act, increment=True)

        if not self.layers[l].stateful:
            if not isinstance(self.layers[l], hf.nl.Linear):
                # note: this requires a memory allocation if
                # d_activations is non-diagonal
                hf.gpu.J_dot(self.GPU_d_activations[l][s],
R_act,
                           out=R_act)
        else:

```

```

        d_input = self.GPU_d_activations[l][s][0]
        d_state = self.GPU_d_activations[l][s][1]
        d_output = self.GPU_d_activations[l][s][2]

        # note: we're doing this weird thing with two
R_states

        # in order to avoid doing a copy every time
        i = s % 2
        hf.gpu.J_dot(d_state, R_states[l][i],
                    out=R_states[l][1 - i])
        hf.gpu.J_dot(d_input, R_act, out=R_states[l][1 -
i],
                    increment=True)
        hf.gpu.J_dot(d_output, R_states[l][1 - i],
out=R_act)

        # R backward pass
        if self.truncation is None:
            trunc_per = trunc_len = sig_len
        else:
            trunc_per, trunc_len = self.truncation

        R_error = self.GPU_errors
        R_deltas = self.GPU_deltas

        for n in range(trunc_per - 1, sig_len, trunc_per):
            for i in range(self.n_layers):
                R_deltas[i].fill(0)
                if R_states[i] is not None:
                    R_states[i][n % 2].fill(0)

            for s in range(n, np.maximum(n - trunc_len, -1), -1):
                for l in range(self.n_layers - 1, -1, -1):
                    if self.GPU_d2_loss[l] is not None:
                        hf.gpu.multiply(self.GPU_d2_loss[l][s],
                                        R_activations[l][s],
out=R_error[l])

                    else:
                        R_error[l].fill(0)

                # error from feedforward connections
                for post in self.conns[l]:
                    W, _ = W_ff[(l, post)]
                    hf.gpu.dot(R_deltas[post], W, out=R_error[l],
                                transpose_b=True, increment=True)

                # feedforward gradient
                W_g, b_g = Gv_ff[(l, post)]
                hf.gpu.dot(self.GPU_activations[l][s],
R_deltas[post],
                                out=W_g, transpose_a=True,
increment=True)

                hf.gpu.sum_cols(R_deltas[post], out=b_g,
                                increment=True)

                # add recurrent error
                if l in self.rec_layers:
                    hf.gpu.dot(R_deltas[l], W_recs[l][0],
out=R_error[l],
                                transpose_b=True, increment=True)

```



```

        # compute deltas
        if not self.layers[l].stateful:
            hf.gpu.J_dot(self.GPU_d_activations[l][s],
R_error[l],
                                out=R_deltas[l],
transpose_J=True)
        else:
            d_input = self.GPU_d_activations[l][s][0]
            d_state = self.GPU_d_activations[l][s][1]
            d_output = self.GPU_d_activations[l][s][2]

            i = s % 2
            hf.gpu.J_dot(d_output, R_error[l],
out=R_states[l][i],
                                increment=True, transpose_J=True)
            hf.gpu.J_dot(d_input, R_states[l][i],
out=R_deltas[l],
                                transpose_J=True)
            hf.gpu.J_dot(d_state, R_states[l][i],
                                out=R_states[l][1 - i],
transpose_J=True)

        # recurrent gradient
        if l in self.rec_layers:
            if s > 0:
                hf.gpu.dot(self.GPU_activations[l][s - 1],
R_deltas[l], out=Gv_recs[l][0],
                                transpose_a=True,
increment=True)
            else:
                hf.gpu.sum_cols(R_deltas[l],
out=Gv_recs[l][1],
                                increment=True)

        # Tikhonov damping and batch mean
        Gv._axpbyz(1.0 / batch_size, GPU_v, damping, Gv)

        if isinstance(v, gpuarray.GPUArray):
            return Gv
        else:
            return Gv.get(out, pagelocked=True)

    def check_J(self, start=0, stop=None):
        """Compute the Jacobian of the network via finite
differences."""

        eps = 1e-6
        N = self.W.size

        # as in check_grad, the truncation is equivalent to running
the network
        # normally for the initial timesteps and then changing the
parameters,
        # so that's what we do here to compute the Jacobian

        if start > 0:
            prev = self.forward(self.inputs[:, :start], self.W)
            init_a = [p[:, -1] for p in prev]
            init_s = [l.state.copy() if l.stateful else None

```

```

        for l in self.layers]
    else:
        init_a = None
        init_s = None

    if stop is None:
        stop = self.inputs.shape[1]

    # compute the Jacobian
    J = [None for _ in self.layers]
    inc_i = np.zeros_like(self.W)
    for i in range(N):
        inc_i[i] = eps

        inc = self.forward(self.inputs[:, start:stop], self.W +
inc_i,
                                init_activations=init_a,
init_state=init_s)
        dec = self.forward(self.inputs[:, start:stop], self.W -
inc_i,
                                init_activations=init_a,
init_state=init_s)

        for l in range(self.n_layers):
            if start > 0:
                inc[l] = np.concatenate((prev[l], inc[l]), axis=1)
                dec[l] = np.concatenate((prev[l], dec[l]), axis=1)

            J_i = (inc[l] - dec[l]) / (2 * eps)
            if J[l] is None:
                J[l] = J_i[... , None]
            else:
                J[l] = np.concatenate((J[l], J_i[... , None]),
axis=-1)

            inc_i[i] = 0

    return J

def check_G(self, calc_G, v, damping=0):
    """Check Gv calculation via finite differences (for
debugging)."""

    sig_len = self.inputs.shape[1]
    if self.truncation is None:
        trunc_per = trunc_len = sig_len
    else:
        trunc_per, trunc_len = self.truncation

    G = np.zeros((len(self.W), len(self.W)), dtype=self.dtype)

    for n in range(trunc_per, sig_len + 1, trunc_per):
        start = np.maximum(n - trunc_len, 0)

        # compute Jacobian
        # note that we do a full forward pass and a partial
backwards
        # pass, so we only truncate the backwards J matrix
        J = self.check_J(0, n)
        trunc_J = self.check_J(start, n) if start > 0 else J

```

```

        # second derivative of loss function
        L = self.loss.d2_loss([a[:, :n] for a in
self.activations],
                                self.targets[:, :n])
        # TODO: check loss via finite differences

        G += np.sum([np.einsum("abji,abj,abjk->ik", trunc_J[l],
L[l], J[l])
                                for l in range(self.n_layers) if L[l] is not
None],
                                axis=0)

        # divide by batch size
        G /= self.inputs.shape[0]

        Gv = np.dot(G, v)
        Gv += damping * v

    try:
        assert np.allclose(calc_G, Gv, rtol=1e-3)
    except AssertionError:
        print("calc_G")
        print(calc_G)
        print("finite G")
        print(Gv)
        print("calc_G - finite G")
        print(calc_G - Gv)
        print("calc_G / finite G")
        print(calc_G / Gv)
        input("Paused (press enter to continue)")

def compute_offsets(self):
    """Precompute offsets for layers in the overall parameter
vector."""

    ff_offset = super(BRNNNet, self).compute_offsets()

    # offset for recurrent weights is end of ff weights
    offset = ff_offset
    for l in range(self.n_layers):
        if l in self.rec_layers:
            self.offsets[(l+2, l)] = (
                offset,
                offset + self.shape[l+2] * self.shape[l],
                offset + (self.shape[l+2] + 1) * self.shape[l])
            offset += (self.shape[l+2] + 1) * self.shape[l]

    return offset - ff_offset

```

## Annex E

Readme.txt

The Requirements to run this library are:

- python 3.5
- numpy 1.9.2
- matplotlib 1.3.1
- optional: scipy 0.15.1, pycuda 2015.1.3, scikit-cuda 0.5.1, pytest 2.7.0

To run the code make sure you have all the requisites mentoned, navigate to the folder of BRNN.py, open a terminal and enter:

```
python BRNN.py
```