Multi-Task LSTM Representation Extraction for Protein Function Prediction

Liam D. Eloie

Supervised by:

Prof. David Jones *Dr. Cen Wan*

This report is submitted as part requirement for the

MSc Degree in Machine Learning

at

University College London

Department of Computer Science

University College London

April 10, 2019

Abstract

Protein function prediction is an important field within bioinformatics that aims to employ machine learning techniques on protein sequence data in order to infer the possible functions of proteins in an efficient and cost-effective manner; essential tools that can ultimately be used in the field of drug delivery and discovery. Recently, deep learning has had a significant impact for many prediction methods within the field of bioinformatics, but has seen less success with protein function prediction due to the sparse, multi-labelled nature of the labels. Not only that, but there is a lack of protein data to successfully train models due to the strict rules that are imposed on the data to overcome protein-similarity testing bias. In this thesis, a novel protein function prediction method is presented, combining the desirability of deep learning whilst utilising the traditional machine learning techniques that have seen success in this field. More specifically, a bi-directional LSTM is used to encode sequence-dependent functional representations of the biophysical attributes of protein sequences, which are then used to train an array of binary SVM classifiers, one for each of the protein functions considered. It was found that using these functional-representations as input to a SVM performed better than using the LSTM as a classifier, helping to alleviate the imbalanced nature of the protein function data. Unfortunately, these feature-representations were not powerful enough to out-perform a SVM trained on the raw global features of the protein sequences. This suggests that further research needs be conducted to investigate the use of additional sequential features, as well as more complicated recurrent neural networks, to see whether it is possible to build up powerful functional-representations that are more informative than the raw global features of the protein sequences.

Contents

Chapter 1

Introduction

The study of biological processes are crucial for understanding the internal workings of our bodies on both a macro and microscopic level, not only fulfilling our natural curiosity to learn more about how the world works, but also offering practical implications within the medical field, such as: providing better protection and detection of diseases, as well as advancing drug delivery and discovery. Since the discovery of DNA and the central dogma of molecular biology, a lot of attention has been focused on understanding proteins and what kind of functions they serve. Traditionally, the study and exploration of the possible functions of a protein required doing a large-scale experiment, which often turns out to be time-consuming and costly; unable to scale to the true scope of the protein function problem. More recently, as sequencing technology advances and protein sequence data becomes more prevalent, biologists have turned to computational methods to help answer these questions.

A first look at protein function prediction involved sequence similarity-based methods, which prove very successful among homologous proteins, but unfortunately, these methods do not work very well when considering the functions of orphan or distant proteins. As such, efforts turned to feature-based machine learning methods, using supervised learning on the biophysical attributes of the protein sequences, resulting in the development of the widely used state-of-the-art protein function predictor, FFPred.

With the advent and success of deep learning, researchers have now turned their efforts to finding ways of using the incredibly powerful tools this field has to offer to learn and discover more about proteins, successfully developing deep neural prediction models for the secondary structure, disorder, and subcellular-localisation of a protein. Conversely, due to the incredibly sparse, imbalanced, and correlated multi-labelled nature of protein functions, combined with the strict rules protein data has to abide by in order to develop a fair and powerful classifier, deep learning techniques have seen less success for the protein function prediction problem.

As a consequence, this thesis will investigate using deep recurrent neural networks, namely an array of multi-task bi-directional LSTMs, on protein sequence data to identify whether they can be used to isolate signal from the proteins and use this for protein function prediction. Extending from this idea and its major limitations, a novel approach for protein function prediction has been proposed; combining the power of deep learning whilst using the notorious traditional feature-based techniques to overcome such limitations. Specifically, features of protein sequences will be transformed into functional feature representations encoding rich long-term dependencies from a LSTM, which will be used as input to an array of FFPred-like SVMs. Finally, a similar array of SVMs will be trained using the raw features of protein sequences providing a suitable baseline to compare the power of using the high-level encoded features.

The first section of this thesis will be a comprehensive overview of the theory needed to understand the problem of protein function prediction. It will cover the basics of what a protein is, how protein function is categorised, as well as the history and current efforts of solving the protein function prediction task. Next, there will be a section on the machine learning tools that will be used to tackle the problem of protein function prediction. This includes an overview of neural networks and how they work, as well as the specific type of neural network that will be used –

the Long-Short-Term Memory network, and the advanced techniques that are used to speed-up the training of these networks. Lastly, this section will conclude with a description of the methods on hyper-parameter tuning as well as a discussion concerning the imbalanced class problem and the metrics that will be used to deal with these types of problems in a fair manner.

The second section will cover the methodology of my project. This will include: a description of the data that was used and where it came from; how the label-space was reduced to simplify the overall problem; and an overview of how the deep-learning representations will be extracted and how they will be used to trained an array of SVM classifiers. Specifically, this will describe: how features were chosen and extracted; how the LSTM and SVM models were constructed; and how hyper-parameter tuning was conducted. Furthermore, there will be a discussion on the how three experiments were designed to evaluate and compare the proposed hypothesis. Finally, there will be a paragraph explaining any considerations that had to be made due to computational as well as time constraints.

Finally, the last sections will be about the results of the experiments, any discussion related to those results, as well as a section to address potential future work.

Chapter 2

Background

2.1 Protein Function Prediction

2.1.1 What is a Protein?

The complexities of life can be boiled down into a countable number of simpler constituents. Among these biological building blocks are proteins – large macromolecules that are responsible for several thousand different biological and chemical functions required for creating and maintaining the balance of life. More specifically, proteins govern most of the work within the cells in a living organism which give rise to the structure, function, and regulation of the body's tissues and organs. Proteins are large biological molecules with complicated dynamics, and without considering the simpler units they are made up from, it can be exceedingly difficult to start understanding how they work and what purposes they may serve.

As such, proteins are made up of many long chains of small organic compounds called amino acids. All amino acids are made up from an amine group, a carboxyl group, and a side chain that gives each amino acid their unique characteristics – there are twenty possible side chains, resulting in twenty unique amino acids in nature. Due to these particular characteristics, long chains of amino acids interact in complicated ways, giving rise to interesting structures within the proteins that ultimately have an effect on their function.

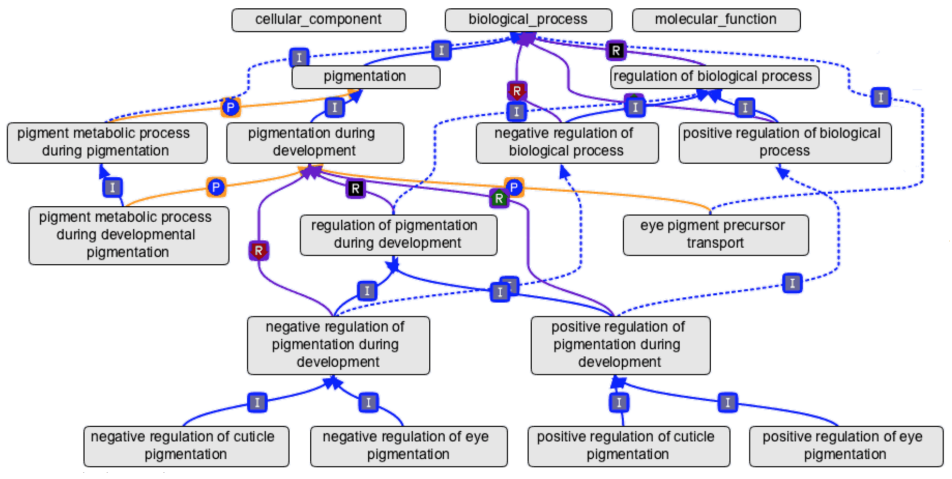
cellular_component
biological_process
molecular_function
pigmentation
regulation of biological process
pigment metabolic process during pigmentation
pigmentation during development
negative regulation of biological process
positive regulation of biological process
pigment metabolic process during developmental pigmentation
regulation of pigmentation during development
eye pigment precursor transport
negative regulation of pigmentation during development
positive regulation of pigmentation during development
negative regulation of cuticle pigmentation
negative regulation of eye pigmentation
positive regulation of cuticle pigmentation
positive regulation of eye pigmentation

Figure 2.1: A graph of the Gene Ontology DAG, showing some of the high-level nodes in the tree.[1]

By analysing the sequence of amino acids that make up a protein in a strategic way, it is possible to infer many properties of that protein, such as its structure, function, and even how it may interact with other proteins. This approach has become popular within the domain of computational biology for efficiently and inexpensively understanding the role of different proteins.

2.1.2 Gene Ontology

In humans alone, there are around 20,000 different proteins that help regulate the cells in our bodies. Beginning to find out what purpose each and every protein serves is an extremely difficult task – one should first come up with an ontology of classifying such objects. That is, a representation of what is known about protein functions and the relationships between them. Francis Crick introduced the idea of the Central Dogma of Molecular Biology; DNA makes RNA, which in turn makes proteins through the processes of transcription and translation [1]. This idea ultimately suggests that for each protein, there is a specific gene within our DNA that encodes for it. Knowledgeable of this fact, and the need for an ontology, Ashburner et al. proposed the idea of *Gene Ontology* (GO) [2]. Gene Ontology is an attempt to unify the representation of gene and gene product attributes across all species, providing and maintaining annotations of gene products–therefore protein functions–and enabling easy functional interpretation of experimental data.

The Gene Ontology takes on the structure of a directed acyclic graph with three main roots, which define the sub-ontologies of GO: *cellular component, molecular function*, and *biological process*. These domains are high-level representations of protein functions and each cover a wide range of proteins – collectively covering all proteins. As the branches progress down the levels of the tree, the functional representation of the nodes become more specific and each individually cover a smaller subset of proteins. An example of the GO tree can be seen in Figure 2.1. Each GO term (node) in the GO tree is associated with attributes that define that gene product. Among the most important attributes are:

- term name - a word, or string of words, that label the gene product,
- identifier - a unique alphanumeric identifier,
- namespace - the domain in which the gene product derives from,
- description - an explanation of what function the gene product serves.

For example, a typical node in the GO may look something like this:

```
id: GO:0000016
name: lactase activity
namespace: molecular_function
def: "Catalysis of the reaction: lactose + H2O=D-glucose
+ D-galactose."
synonym: "lactase-phlorizin hydrolase activity" BROAD
[EC:3.2.1.108]
synonym: "lactose galactohydrolase activity" EXACT [EC:3.2.1.108]
xref: EC:3.2.1.108
xref: MetaCyc:LACTASE-RXN
xref: Reactome:20536
is_a: GO:0004553 ! hydrolase activity, hydrolyzing O-glycosyl
compounds
```

[1]http://www.geneontology.org/page/ontology-structure

2.1.3 Critical Assessment of Protein Function Annotation Algorithms

With the fast development of protein function annotation algorithms, it is imperative to have a way of testing these algorithms in a non-bias setting to understand how well these tools operate in practice. The Critical Assessment of Protein Function Annotation Algorithms (CAFA) is a timed challenge, designed for measuring the large-scale assessment of computational methods dedicated to predicting protein function [3]. The way CAFA works is as follows:

- CAFA organisers release a large number of protein sequences that have yet to be experimentally annotated.
- Competitors apply their models to predict the function of these proteins, either by associating them with GO terms, or Human Phenotype Ontology.
- After a prediction deadline, several months pass, in which some proteins whose functions were experimentally unknown during the competition have received experimental verification.
- These proteins are used to create a benchmark, in which the methods are tested and compared.

2.1.4 Function Prediction

Protein function prediction is a very important and on-going research topic within bioinformatics. Due to the progression of high-throughput sequencing techniques, the number of protein sequences in databases such as: Swiss-Prot [4] and PDB [5] have been exponentially increasing. Illumina, one of many competitive sequencing companies, produces a suite of sequencers which are able to sequence upwards of 30x the human genome in less than 30 hours [6]. As a result of this exponential increase, more than 80% of sequences in UniProtKB release 2015_01 lack assignments for at least one of the three GO sub-ontologies, causing a gap between the number of sequences available and the number of sequences annotated to emerge,

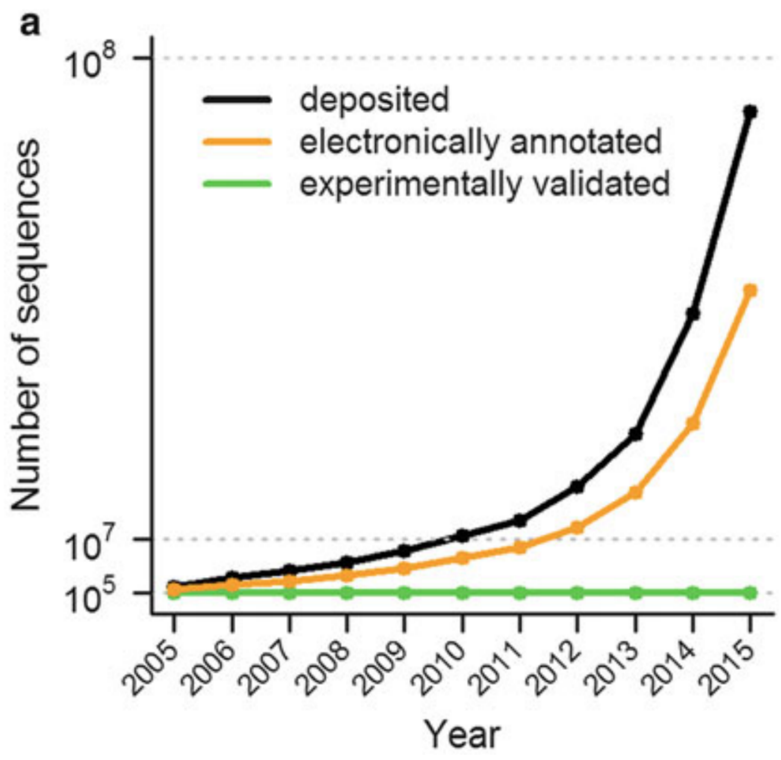

Figure 2.2: A plot demonstrating the exponential increase in the number of amino acids sequences (black) compared to the linear increase in the number of experimentally validated supported GO term assignments (green). The orange line shows the increase in the number of electronically annotated GO term assignments.

demonstrated by the plot shown in Figure 2.2 [7]. Experimental methods for function prediction, such as: gene knockout, targeted mutations, and the inhibition of gene expression [8] are low-throughput – requiring large experimental and human efforts to analyse a single protein, which has become intractable for scaling to the amount of sequences available nowadays. It has become necessary, thus desirable, to develop computational techniques to analyse and predict the function of these protein sequences to try to bridge this gap, tackle the intractability of classical methods, and reduce the amount of efforts required to successfully annotate sequences of proteins.

Sequence-based function prediction methods can be broken down into three classes:

- homology-based: proteins with similar sequences are usually homologous, and thus have a similar function. These methods test for the similarity between a target protein and proteins from large databanks to infer protein functions.

- sequence motif-based: identifying domains with a query sequence to provide evidence for the likely functions.
- feature-based: use specific features of protein sequences, whether that be particular motifs or some other property of the proteins, such as its length, amino-acid composition, or secondary-structure, to build a function approximator that is able to infer protein functions.

Among the first successful attempts at predicting protein function were sequence-homology based similarity methods such as FASTA [9] and BLAST [10]. These systems use approximate sequence-alignment techniques to annotated protein sequences within databanks in search for proteins that are homologous, or similar in terms of sequence-structure, to the protein being annotated. This idea is motivated by the belief that proteins with similar sequences evolved from a common-ancestor, and therefore should have similar functions [11]. Unfortunately, these methods are not fool proof due to the divergent evolution of a protein, specifically when the duplicate of the original gene adopts a different function [12]. The introduction of protein-structure can be incorporated into the sequence-based methods to help overcome this problem, as in many cases the evolution of a protein retains the folding pattern long after the sequence similarity becomes undetectable [13].

Often, particular sub-sequences within a protein can give a powerful indicator of its specific functions. Such sub-sequences, known as *motifs*, are conserved across a set of proteins belonging to the same family, and thus are candidates for functional sites within a protein. A useful resource to find domains within a query sequence is Pfam, which is a protein families database containing a large collection of protein families, represented by multiple sequence alignments and hidden Markov models [14]. Examples of motifs within a sequence would be: ligand binding sites, DNA binding sites, and sites that promote interactions with other proteins – giving a strong clues as to the function of a protein [15, 16]. Applications of these methods were first carried out by Hannenhalli and Russell on protein sets that have no close homologues. They found a 20% increase in accuracy using their methods on these protein sets, which achieved an accuracy of 96%, over the sequence-based similarity methods of BLAST, which achieved an accuracy of 74% [17].

Motif-based methods analyse the raw string of amino-acids in order to locate functional regions within a sequence, using them for the indication of possible functions in a protein. Alternatively, it is possible to first transform this sequence of amino-acids into more biological meaningful features – in essence, coming up with a functional representation of a protein which could prove more informative when it comes to function classification. These methods are known as feature-based and typically use machine learning algorithms such as SVMs and neural networks to predict the functions of a protein, given its sequence. Like motif-based methods, feature-based methods work well when no homologues to the protein sequence can be found. One of the first successful attempts was carried out by Jensen et al., where they developed a shallow neural-network based function predictor for cellular role as well as enzymatic function according to the EC classification system [18]. Sequence derived features such as predicted post translational modifications, protein sorting signals, and physical/chemical properties calculated from the amino acid composition were used. Using a neural network with a single hidden layer of 10-50 neurons, depending on the functional category, they were moderately successful in predicting functions of orphan proteins. Soon after, they used a similar approach to design a model that also covered a number of biologically, as well as pharmaceutically interesting categories in the GO classification system, achieving a recall rate of 70% for hormones and receptors function classes; well suited for gene discovery and assay selection purposes [19]. In 2008, Lobley et al. designed a feature-based function predictor using an array of SVMs for vertebrate proteomes, named FFPred [20]. Using the amino acid content, structural disorder, secondary structure, as well as a host of other biological informative properties of proteins, they were able to train a library of SVMs for predicting functions representing over 300 GO annotations. For each GO term, five SVMs using a RBF kernel were used to recognise feature patterns associated with the annotation term and was able to achieve credible function prediction for distant and orphan proteins over a broad range of GO terms included within the molecular function and biological process sub-ontologies. FFPred was further developed: using larger datasets of protein sequences and annotations; updated component feature predictors; as well as revised training procedures to create the FFPred 2.0 prediction server [21]. FFPred 2.0, still using SVM-based techniques, was able to predict a combined total of 442 GO annotations for the molecular function and biological process sub-ontologies and topped the rankings of the first CAFA experiment. In the conquest to provide a powerful function predictor for proteins, Domenico Cozzetto et al. further improved the FFPred model, using a larger SVM library that now extends its coverage to the cellular component GO sub-ontology for the first time [22]. Known as FFPred 3, this model now serves as the state-of-the-art when predicting the function of distant or orphan proteins, covering 597 GO annotation. FFPred 3 predictions achieve a higher precision value, for high recall values, than BLAST and naïve methods for all three GO sub-ontologies, as well as dominated the rankings for term-centric evaluations in the CAFA2 experiment [23].

As attention in machine learning has recently been moving over to deep neural methods, which have displayed a number of incredible results within many different fields including but not limited to: facial recognition [24], language translation [25], and learning-agents in reinforcement learning [26] – the interest in the application of these models within bioinformatics have also gained much attention. Recent efforts have included using deep convolutional neural networks and long short-term memory recurrent neural networks for protein secondary structure prediction [27, 28], protein disorder prediction [29], protein subcellular localization prediction [30], and protein contact prediction [31], all achieving significant results and usually becoming the new state-of-the-art models. Likewise, deep learning has also been applied to protein function prediction and has seen great success. Kulmanov et al. used a convolutional neural network to learn features through a representation embedding layer according to trigrams in the protein sequence, combined this with information on the protein structure and protein-protein interaction networks, and then designed a hierarchy of deep fully-connected layers that can refine features on each distinction present in the GO classification. Using this architecture, they saw significant improvements over baseline methods such as BLAST, especially for predicting cellular locations [32]. Rui Fa et al. investigated the usefulness of multi-task deep neural networks (MTDNN), where all GO terms share a hidden-layer representation, which branches off into a parallel stack of independent hidden layers for each of the GO terms to be classified. Interestingly, this architecture gives more accurate predictions than baseline methods for proteins with no close homologues due the exploitation of commonalities and differences between the prediction tasks [33]. The unfortunate downfall of these methods are that they generally require large sets of data due to the large number of parameters within the models, and are extremely computationally expensive, taking upwards of weeks to train. Unfortunately, datasets concerning protein function prediction usually require special tailoring to remove sequence-similarity between training and test sets, resulting in a reduced size in the data – as discussed in the FFPred paper.

2.2 Neural Networks

In recent years, due to their incredible ability to learn complex non-linear representations of input data, the idea of neural networks has gained a lot of traction, being used for a whole host of problems such as: predictive models, generative models, and control problems [34, 35]. Neural networks have existed as a concept since the fiftie, but due to the computational considerations, lack of data, and problems with backpropogating gradients through multiple hidden layers, they were considered inferior to more classical techniques such as decision trees and support vector

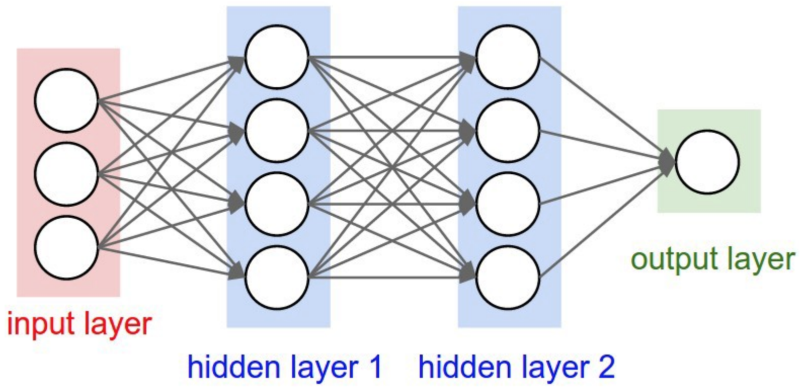

Figure 2.3: A schematic diagram of a neural network.[2]

machines. Post technological boom, with the development of powerful Graphical Processing Units (GPUs), being in the middle of the big data era, as well as the development of research in new theories and techniques, neural networks are now incredibly powerful tools in approximating functions and lead the forefront of most machine learning in production today.

Neural networks are a simple concept, very loosely borrowing from the ideas of how biological neural networks within our own brains work. A neural network is made up of subsequent layers of processing units called ’neurons’. These neurons have the ability to control the flow of information through the network, depending on what kind of signals are being passed to them. Each neuron in a layer receives a signal from every other neuron in the previous layer and that neuron will go on to pass a signal to every other neuron in the next layer. The input of a neural network will often be a stream of data and by carrying-out this process in a methodical way, the signals that reach the ends of the neural network are capable of containing high-level representations of this data, resulting in very powerful predictive and generative models. A depiction of a neural network can be seen in Figure 2.3

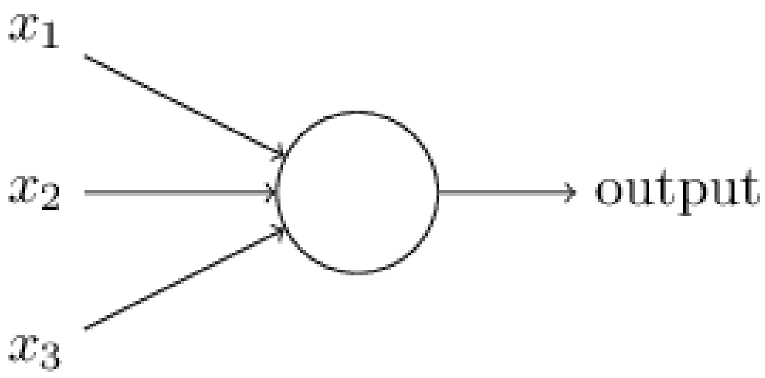

Figure 2.4: A schematic diagram of a perceptron with three input binary features and a single binary output.[3]

2.2.1 Perceptron

Before considering the dynamics of a modern neural network, it may prove useful to consider a simpler, special kind of neural network called a perceptron. The perceptron was theorised back in 1958 by Frank Rosenblatt [36]. A perceptron consists of two layers: an input layer containing multiple binary inputs and an output layer containing a single binary output. A sketch of a perceptron with three binary inputs and a single binary output can be seen in Figure 2.4. To compute the output of the perceptron, *weights* are defined for each input representing their relative importance. In this case, there would be the weights w_1, w_2, w_3 for the binary inputs x_1, x_2, x_3. These weights are used to form a linear combination of the input features and then the output of the neuron will either be 0 or 1 depending on whether the weighted sum is less than or greater than some threshold value. Mathematically, this can be expressed as:

$$\text{output} = \begin{cases} 0 & \text{if} \sum_i w_i x_i \leq \text{threshold} \\ 1 & \text{if} \sum_i w_i x_i > \text{threshold} \end{cases} \tag{2.1}$$

What makes a neural network so versatile is the ability to change these weights connecting the input features in order to choose when the output neuron activates or deactivates. Even more astonishing, it can be shown that a perceptron can be used

[2]https://hackernoon.com/challenges-in-deep-learning-57bbf6e73bb
[3]http://neuralnetworksanddeeplearning.com/chap1.html

to model a `NAND` gate and thus form a complete unit for universal computation [37].

Extending the idea of a perceptron, additional layers can be introduced between the input and output layers, called *hidden* layers. These networks are known as *Multi-Layer Perceptrons* (MLPs) [38]. These additional hidden layers allow it so more complex computation can take place, and therefore the network becomes even more versatile when it comes to designing outputs. In order to really make use of the versatility these networks possess for bigger problems, it is necessary to design learning algorithms that can automatically tune the weights of the networks to produce desired outputs, as the number of weights become intractable to design by hand.

Machine learning is a huge field, offering many different kinds of learning algorithms. Amongst the most popular ways to design learning algorithms is through gradient-based optimisation of a loss function [39]. That is, given some *metric* that measures the performance of a learning algorithm, the parameters of that algorithm should be adjusted such that the performance is maximised. In the case of the MLP, this means that the weights and biases of the network need to be dynamically changed to maximise how desirable the outputs of the network are. This is usually done by calculating the gradient of the loss function, given some change in the parameters, and then adjusting the parameters in the direction of maximum descent with respect to that gradient and the change in the parameters.

$$w_i = w_i + \frac{\partial L}{\partial w_i} \Delta w_i \tag{2.2}$$

Unfortunately, this learning procedure on a MLP is unable to learn effectively. This is because the outputs of the neurons are discrete, either 0 or 1. A small change in the parameters of the network can cause drastic changes within the outputs, resulting in unstable gradients for learning. Fortunately, there is a fix for this – *Sigmoid* neurons [40]. Sigmoid neurons, instead of being a discrete ”on” or ”off” governed by a threshold, have continuous values describing how activated they are. Small changes in the weights only induce small changes in the gradients, allowing for smooth weight updates. Like a perceptron, sigmoid neurons calculate the linear combination of input features – but instead of comparing this to a threshold, a logistic activation function is applied. This carefully crafted monotonic function, seen in Equation 2.3, squashes the output of a neuron into a range between 0 and 1, varying smoothly, providing a continuous approximation to the step-function.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.3}$$

Neurons are not limited to the logistic activation function to get the desirable properties for designing a learning algorithm, other smooth monotonic non-linear functions can also be used, such as the *Tanh* function or *ReLU* function [41]. It is these type of neurons that are used today in modern fully-connected neural networks, capable of automatically learning parameters through a process called *backpropagation* [42].

2.2.2 Feed-forward Fully-Connected Neural Network

By combining these ideas, a neural network can now be constructed. Given a set of data $S = \{(x_1, y_1), ..., (x_N, y_N)\}$, where $\mathbf{x}_i$ are the features of a data point and y_i is the corresponding label of those features, a neural network is to be designed to seek a function $f : X \rightarrow Y$. For this example, the architecture of the neural network in Figure 2.3 will be considered, where there are two hidden layers. The weight connecting neuron j in layer l to neuron k in layer $l+1$ will be denoted by w^l_{jk} and the bias in neuron k will be denoted by b^l_k. Additionally, the signal contained within neuron j in the l^{th} hidden layer will be denoted by h^l_j. From these notations, a matrix $W^{(l)}$ and a bias $\mathbf{b}^{(l)}$ can be defined that maps the hidden state of layer l into the hidden state of layer $l+1$, $\mathbf{h}^{(l)} \rightarrow \mathbf{h}^{(l+1)}$.

Forward Pass

First, the output of the neural network needs to be evaluated, given some features of the i^{th} data point, $\mathbf{x}_i$. This is called the *forward pass* and simply requires going through the neural network, sequentially applying the weights and biases throughout the layers on $\mathbf{x}_i$. Equation 2.4 shows how this process is carried out layer-by layer.

$$\begin{aligned}
\mathbf{h}^{(0)} &= \mathbf{x}_i \\
\mathbf{z}^{(1)} &= W^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)} \\
\mathbf{h}^{(1)} &= \sigma_a(\mathbf{z}^{(l)}) \\
\mathbf{z}^{(2)} &= W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)} \\
\mathbf{h}^{(2)} &= \sigma_a(\mathbf{z}^{(2)}) \\
\mathbf{z}^{(3)} &= W^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)} \\
\mathbf{h}^{(3)} &= \mathbf{o} = \sigma_a(\mathbf{z}^{(3)}) \qquad (2.4)
\end{aligned}$$

From this forward pass, a *loss function* is used to calculate how different the output of the neural network, $\mathbf{o}$, is from the actual label $\mathbf{y}_i$. Commonly, the loss function is the mean-squared error between the predicted and actual output, given in Equation 2.5 – or if the labels are binary one-hot encoded vectors, a common loss function is the binary-cross entropy function seen in Equation 2.6. If the difference between the two is zero, then the neural network perfectly predicts the label of the data point.

$$L(\mathbf{o}_i, \mathbf{y}_i) = \frac{1}{2}\sum_j (o_{ij} - y_{ij})^2 \qquad (2.5)$$

$$L(\mathbf{o}_i, \mathbf{y}_i) = -\sum_j y_{ij} \log o_{ij} \qquad (2.6)$$

Backward Pass

The idea of a backward pass is to calculate the gradient of the loss in respect to all the parameters in the network. This is so that the parameters can be adjusted using a *gradient descent* optimiser to try and reduce the loss to zero. By doing so, this would ensure that the neural network produces outputs that are close to the true labels. Simply put, a backwards pass is a series of applied chain rules, progressively accumulating gradients backwards through the network.

A recursive formulae for the calculation of the gradients through the network can be setup by first considering the component form of the input sum of a neuron k in layer l, $z_k^{(l)}$, the corresponding hidden state, $h_k^{(l)}$, and the next input sum of a neuron m in layer $l+1$.

$$z_k^{(l)} = \sum_j w_{kj}^{(l)} h_j^{(l-1)} + b_k^{(l)} \tag{2.7}$$

$$h_k^{(l)} = \sigma_a(z_k^{(l)}) \tag{2.8}$$

$$z_m^{(l+1)} = \sum_k w_{mk}^{(l+1)} h_k^{(l)} + b_m^{(l+1)} \tag{2.9}$$

Consider a perturbation in the weights $w_{kj}^{(l)}$, how would the loss change as a consequence of this small change? The small change in these weights would first cause a change in $z_k^{(l)}$, causing a change in $h_k^{(l)}$, causing a further change in $z_m^{(l+1)}$ for all m. Therefore using the chain rule, the change in the loss in respect to the weights can be written as:

$$\begin{aligned}
\frac{\partial L}{\partial w_{kj}^{(l)}} &= \frac{\partial L}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}} \\
&= \frac{\partial L}{\partial h_k^{(l)}} \frac{\partial h_k^{(l)}}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}} \\
&= \left[\sum_m \frac{\partial L}{\partial z_m^{(l+1)}} \frac{\partial z_m^{(l+1)}}{\partial h_k^{(l)}} \right] \frac{\partial h_k^{(l)}}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}}
\end{aligned} \tag{2.10}$$

By evaluating all derivatives, excluding $\frac{\partial L}{\partial z_m^{(l+1)}}$, the derivative with respect to the weights $w_{kj}^{(l)}$ becomes:

$$\frac{\partial L}{\partial w_{kj}^{(l)}} = \left[\sum_m \frac{\partial L}{\partial z_m^{(l+1)}} w_{mk}^{(l+1)} \right] \sigma_a'(z_k^{(l)}) h_j^{(l-1)} \tag{2.11}$$

The *error signal* will now be defined, which is just the rate of change in the loss function with respect to the input sum of a neuron k in layer l, $z_k^{(l)}$.

$$\delta_k^{(l)} = \frac{\partial L}{\partial z_k^{(l)}} \tag{2.12}$$

Equation 2.10 allows us to rewrite this error signal in terms of its effects through the next layer of the network.

$$\delta_k^{(l)} = \left[\sum_m \frac{\partial L}{\partial z_m^{(l+1)}} w_{mk}^{(l+1)}\right] \sigma_a'(z_k^{(l)}) \tag{2.13}$$

Therefore, a recursion relationship for the error signal in layer l in terms of the error signals in layer $l+1$ can be seen by noticing that $\frac{\partial L}{\partial z_m^{(l+1)}}$ is just the error signal of neuron m in layer $l+1$, giving rise to

$$\delta_k^{(l)} = \left[\sum_m \delta_m^{(l+1)} w_{mk}^{(l+1)}\right] \sigma_a'(z_k^{(l)}) \tag{2.14}$$

Since the error signals can be used to calculate the gradient of the loss in respect to both the weights and biases in a specific layer (Equation 2.15), this recursive relationship allows the gradient of the loss with respect to all the parameters to be calculated fast and efficiently by back propagating the error through the layers.

$$\begin{aligned} \frac{\partial L}{\partial w_{kj}^{(l)}} &= \frac{\partial L}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}} = \delta_k^{(l)} h_j^{(l-1)} \\ \frac{\partial L}{\partial b_k^{(l)}} &= \frac{\partial L}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_k^{(l)}} = \delta_k^{(l)} \end{aligned} \tag{2.15}$$

By vectorising these equations to allow simultaneous calculations of gradients and error signals in a layer for computational efficiency, the procedure of backpropagation through a network of L layers to retrieve all necessary gradients can be written as:

1. $\delta^{(L)} = \nabla_{h^{(L)}} L \odot \sigma_a'(\mathbf{z}^{(L)})$

2. $\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(\mathbf{z}^{(l)})$

3. $\nabla_{W^{(l)}} L = \delta^{(l)} \cdot (\mathbf{h}^{(l-1)})^T$

4. $\nabla_{\mathbf{b}^{(l)}} L = \delta^{(l)}$

5. Repeat steps 2 - 5 until $l = 1$

Once all the gradients have been calculated, the weights and biases can be updated using gradient descent, moving them in the direction that minimises the loss.

$$W^{(l)} := W^{(l)} - \alpha \nabla_{W^{(l)}} L \tag{2.16}$$

$$\mathbf{b}^{(l)} := \mathbf{b}^{(l)} - \alpha \nabla_{\mathbf{b}^{(l)}} L \tag{2.17}$$

$$\tag{2.18}$$

where α is a hyperparameter that governs the size of the step to be taken in the direction of the gradients of the loss.

2.2.3 Recurrent Neural Networks and the Vanishing Gradient Problem

There are many different types of neural-network architecture depending on the type of problem one is working on. For example, when working with image data, it proves useful to use a *Convolutional Neural Network* which is able to capture informative local structures within an image by performing convolutions and max pooling operations, which leads to a richer feature representation and ultimately a more powerful model [43]. Likewise, when working with sequential data it is useful to be able to capture sequential dependencies within the data. This can be done with a *Recurrent Neural Network* or more commonly known as a *RNN* [44]. RNNs, like feedforward neural networks, use linear combinations of features in the data, followed by a non-linear transformation to build high-level representations which can lead to very powerful function approximations. Unlike the feedforward networks, RNNs involve the hidden state of the previous time-step when computing the representations in the next layer. This allows information from the sequence to propagate through the network. As a result, RNNs have a functional memory component that is able to capture rich time-dependent information – resulting in a much more powerful model for sequential data.

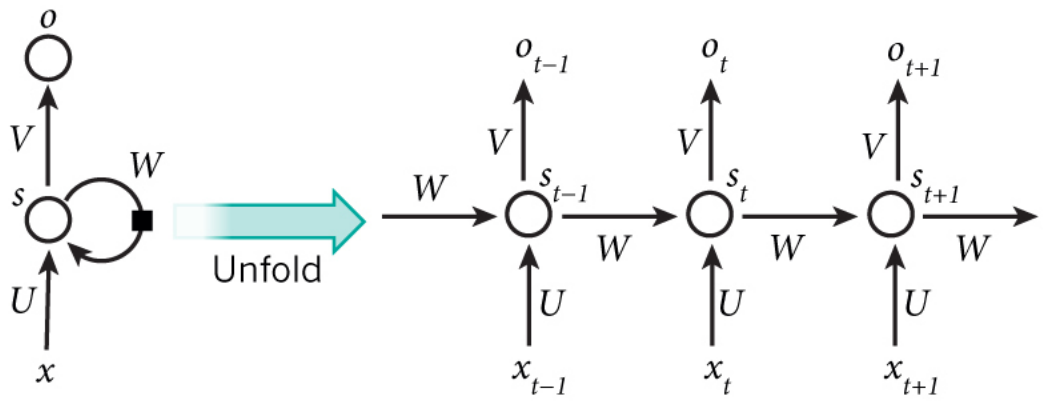

Figure 2.5: A schematic diagram of a RNN, both folded and unfolded, representing how the data x is transformed using weights w into the hidden states s and ultimately a prediction y.[4]

A schematic diagram for a RNN can be seen in Figure 2.5 and the computations for a forward pass through the network can be seen in Equation 2.19.

$$\begin{aligned} s_t &= \tanh(Ux_t + Ws_{t-1}) \\ o_t &= \text{softmax}(Vs_t) \end{aligned} \tag{2.19}$$

where x_t represents the features at each time step, s_{t-1} denotes the hidden state of the previous time step, s_t denotes the hidden state at the current time step, and o_t denotes the output of the network at time t. U is a weight matrix that acts on the features x_t to form a linear combination that partially contributes to the next hidden state. Likewise, W is a weight matrix that forms a linear combination of the previous hidden state that partially contributes to the next hidden state, and V is a weight matrix that forms a linear combination of the current hidden state mapping to an output. Note that the parameters U, V, and W are shared across every time step – this ensures that the number of parameters does not grow linearly with the

[4]http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

number of time steps. Another useful consequence is that it allows the network to easily generalise to sequences with varying length.

The performance of the network is measured using the cross-entropy loss function at every time step t, as seen in Equation 2.20.

$$\begin{aligned} E(o_t, \hat{y}_t) &= -o_t \log \hat{y}_t \\ E(o, \hat{y}) &= \sum_t E_t(o_t, \hat{y}_t) \end{aligned} \tag{2.20}$$

When performing backpropagation, it is simple to compute the gradients $\frac{\partial E_T}{\partial U}$ and $\frac{\partial E_T}{\partial V}$ as the gradients only have to go through a single layer. Unfortunately, when computing the gradient with respect to W, the gradients have to be backpropagated through the entire sequence. Since sequences are generally quite long, gradients have to be backpropagated through many layers. This leads to the *vanishing gradient* effect – long-term dependencies are lost through the shrinking of gradients through time. This can easily be seen when explicitly writing out the form of $\frac{\partial E_T}{\partial W}$ by applying the chain rule with care.

$$\begin{aligned} \frac{\partial E_T}{\partial W} &= \sum_{t=0}^{T} \frac{\partial E_T}{\partial o_T} \frac{\partial o_T}{\partial s_T} \frac{\partial s_T}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial W} \\ &= \frac{\partial E_T}{\partial o_T} \frac{\partial o_T}{\partial s_T} \sum_{t=0}^{T} \left(\prod_{j=t+1}^{T} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_t}{\partial W} \end{aligned}$$

Focusing attention to the product of the $t-T$ Jacobian matrices, $\frac{\partial s_j}{\partial s_{j-1}}$, it is evident that the 2-norm of each element in the matrix has an upper bound of 1. This is because the derivative of tanh is bounded between 0 and 1. As a result, the larger the number of terms in a sequence, the bigger the difference between $T-t$, and therefore faster the derivative exponentially decays to zero. This means that the change in a hidden state far-down in the sequence will have no real effect on the current hidden state. Razvan Pascanu et al. provide an even more extensive overview on the difficulty of training recurrent neural networks [45].

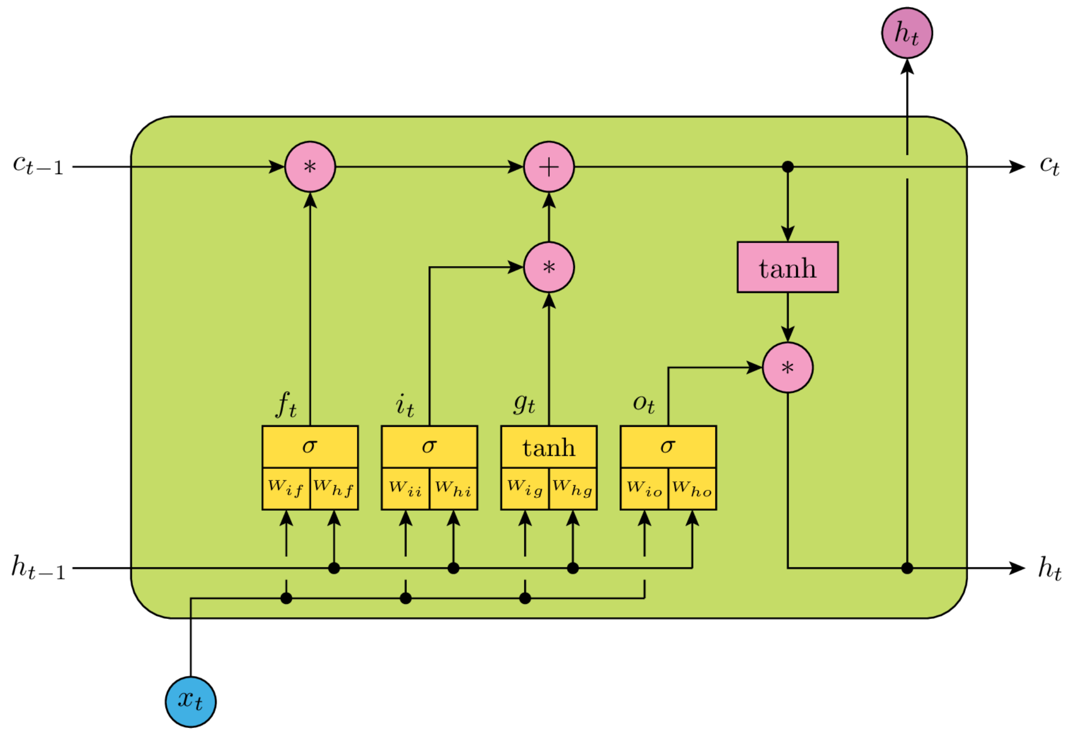

Figure 2.6: A LSTM memory cell demonstrating how information flows through three distinct gates: forget, input, and output in order to pass relevant information through the current and hidden states of the network.[5]

2.2.4 Long-Short Term Memory

Aware of these difficulties, Schmidhuber et al. wanted to come up with an architecture designed to drastically reduce the effect of vanishing gradients and able to efficiently learn long-range dependencies within sequential data. Their solution was the Long-Term Short Memory (LSTM) network [46]. Instead of having each hidden node in a RNN be a single activation function, each time-step of the LSTM is a memory cell – a module containing a cell state and a circuit of gates able to control the flow of data – a diagram of this cell can be seen in Figure 2.6. Specifically, there are three gates: the *forget* gate, the *input gate*, and the *output gate*. Each gate is a function of the new input data and the previous hidden state and acts a filter through various steps of the process. The role of each gate is as follows:

- The forget gate chooses which information is irrelevant and will be removed from the cell state.

[5]https://medium.com/@andre.holzner/lstm-cells-in-pytorch-fab924a78b1c

- The input gate chooses which information will be added to the cell state.
- The output gate decides what values from the cell state will be carried over to the hidden state output.

As well as the computation of the gates, a candidate cell state, $\tilde{c}_t$, is also computed. This state represents the possible configuration of information that has the potential of being carried over to the current state of the cell. Mathematically, the internal workings of an LSTM cell can be expressed within the following expressions.

$$\begin{aligned}
f_t &= \sigma_g(W_{if}x_t + W_{hf}h_{t-1} + b_f) \\
i_t &= \sigma_g(W_{ii}x_t + W_{hi}h_{t-1} + b_i) \\
o_t &= \sigma_g(W_{io}x_t + W_{ho}h_{t-1} + b_o) \\
\tilde{c}_t &= \tanh\big(W_{ig}x_t + W_{hg}h_{t-1} + b_c\big) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}$$

- f_t is the forget gate, with its corresponding weights and biases, W_{if}, W_{hf}, b_f,
- i_t is the input gate, with its corresponding weights and biases, W_{ii}, W_{hi}, b_i,
- o_t is the output gate, with its corresponding weights and biases, W_{io}, W_{ho}, b_o,
- $\tilde{c}_t$ is the candidate cell state at time t
- c_t is the current state at time t.
- h_t is the hidden state at time t.

Notice that the current state at time t is a combination of the elementwise product between the forget gate with the previous cell state and the elementwise product of the input gate with the candidate cell state of the current time. Concisely, the current state is just a selection of information from the previous cell state and the candidate cell states governed by the input and forget gate.

Looking back at the RNN vanishing gradient problem, the culprit was the recursive derivative term $\frac{\partial s_t}{\partial s_{t-1}}$. As a result of these gates, the recursive derivative at any time step now evaluates to either values that are greater than 1 or values that are in the range $[0,1]$. By choosing the parameters of the gates carefully, this property can be exploited to maintain the gradients from vanishing. Essentially, the gates can–and will–learn the appropriate parameters that reduce this effect. This is unlike the vanilla-RNN architecture where the recursive derivative at every time step is either always greater than 1 or always in the range $[0,1]$.

2.2.5 Multi-Label Learning in Deep Neural Networks

Generally speaking, a lot of problems within machine learning are single-labelled, multi-classification problems. These are problems where there exists a single answer out of multiple possible answers. Examples of these types of questions would be:

- Is this image a picture of a cat or a dog?
- What number is this handwritten digit?
- Does a person have a certain type of disease or not?

How about problems that supersede single-labelled problems and instead have multiple possible labels? Such as the examples:

- Classify all the objects in an image.
- What attributes does this human face have?
- What functions does a certain protein serve?

These are known as *multi-class, multi-label* classification problems – ones in which the problems have multiple answers that have to be identified. Formally, these problems are defined by the following. Given input data $X = (x_1, ..., x_n)$, the output of a model will be a vector

$$\mathbf{y} \in \mathbf{R}^n : y_i \in \{0,1\}$$

In order to learn models for multi-class, multi-label classification problems, similar machine learning techniques are used. Unfortunately, these techniques tend to struggle to learn powerful models, due to the natural sparsity and imbalance of the label-space.

2.2.6 Advanced Techniques

The optimisation of a neural network, as shown above, is quite a simple procedure which leads to a powerful learning algorithm. In order to make the most out of the data given and to speed up the learning procedure, a number of components within the neural network learning procedure can be improved. This includes how the weights and biases in the network are initialised, how the data flows through the networks, and how the parameters of the network are updated.

Xavier Initialisation

Initialisation of parameters within a neural network is an important aspect, ultimately governing how long a neural network will take to converge, or if it will converge at all. If the parameters of the network are initialised to values that are too small, the input signal to the neurons will eventually drop to a low value and will no longer be useful. This is because the variance of the input signal will start to diminish as it passes through each layer of the network due to the nature of the activation functions used. Likewise, if the parameters are too large, then the neurons will become saturated – causing the gradients to be close to zero and learning to come to a halt. Therefore, it is important that the parameters of the network are within a reasonable range before learning begins.

Xavier initialisation attempts to do just this, keeping the variance of the data through each layer constant. That is, given some input to a neuron $\mathbf{x}$ and the output before the activation function is applied, $\mathbf{z}$:

$$\mathbf{V}(\mathbf{x}) = \mathbf{V}(\mathbf{z})$$

It can be shown that this is achieved if the weights in the network are initialised according to a Gaussian distribution with a variance of $\frac{1}{N_i}$, where N_i is the number of input connections going into neuron i, as seen in Equation 2.21 [47].

$$w_i \sim \mathbf{N}\left(0, \frac{1}{N_i}\right) \tag{2.21}$$

Batch Normalisation

Normalising the inputs of a neural network is a common procedure that is known to help with the efficiency of training and reducing the chances of getting stuck in a local optima. Motivated by this idea, Sergey Ioffe et al. proposed doing this not only for the input layer, but also for all the other hidden layers – this is known as *batch normalisation* [48]. This technique ultimately improves the performance and stability of neural networks, and makes it possible for more sophisticated deep learning architectures that contain many layers work in practice. Other benefits include: making networks train faster, allowing higher learning rates and providing regularisation.

Dropout

Overfitting is a huge problem in machine learning, concerning the idea of a model learning the nuances of the data it has been trained on, particularly the noise – unable to generalise well to unseen data. Regularisation methods have been developed to combat overfitting, such as adding an L1 or L2 norm penalty to the loss function [49]. If given unlimited compute power, the best way to regularise a model is to build an ensemble of that model for all possible settings of the parameters and then averaging over the predictions, weighting each setting by its posterior probability given the training data. This can be approximated in smaller models, but unfortunately would be way too computational expensive to implement with neural networks. An inexpensive and efficient way of approximately combining exponentially many different neural network architectures, and therefore preventing overfitting to a neural network, is through a technique called *dropout*, proposed by Nitish Srivastava et al. [50]. Dropout refers to the idea of probabilistically 'dropping' neurons out of the architecture, along with its incoming and outgoing connections during training. This forces the network to learn many different representations of the same data, whilst trying to minimise the loss function. Therefore, the network becomes more versatile, less likely to overfit to training data, and generalises better to unseen data.

Improved Optimisers

Performing updates on the network weights, by choosing one random example at a time is known as *Stochastic Gradient Descent* (SGD). Although SGD may eventually get the work done, generally it is quite slow to converge and may have problems falling into sub-optimal local minima. SGD has also been found to be very slow to learn particularly when the gradients lie along a long-narrow valley, resulting in oscillations back-and-forth perpendicular to the direction through the valley. To overcome this short-coming, Rumelhart et al. introduced a second term to the update equation that incorporates the gradient at the previous time-step, as well as the current time step, called *momentum* [51]. Only having a single hyperparameter–the learning rate–although simple, can also lead to a few problems. It can be difficult to set the learning rate, as a value too small will take far too long to converge, whilst a value too large will cause continual random exploration or even divergence. When working with high-dimensional non-convex objective functions, as is most common when using neural networks, a single learning rate may be too small in some dimensions, but too large in other dimensions. A naïve approach to mitigate this problem is by having a learning rate for each dimension, but this soon becomes intractable when dealing with millions of parameters, as often is the case in deep neural networks. A more efficient way is by having a single initial learning rate, and scaling proportional to the size of the gradient in each dimension. This method, first proposed by Duchi et al., is known as *AdaGrad* [52]. The most popular optimiser nowadays is *Adaptive Moment Estimation*, known as Adam, which combines the benefits of both momentum and AdaGrad [53].

2.3 Support Vector Machine

A support vector machine is a machine learning algorithm whose objective is to find the maximal separating hyperplanes between different classes of data points [54]. For linearly separable binary-classed data, the problem can be formulated as the following:

$$\begin{aligned} &\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ &\text{subject to: } y_i(\mathbf{w}^T x_i + b) \geq 1, i = 1, ..., N \end{aligned} \tag{2.22}$$

where $\mathbf{w}$, b are the parameters of the separating linear hyperplane. If the data is linear, but not completely separable, *slack variables* can be introduced to relax the linearly separable assumption, relaxing the constraints. This allows the SVM to willingly misclassify some data points for the greater good. The new formulation takes on the following form:

$$\begin{aligned} &\min_{\mathbf{w}} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N} \xi_i \\ &\text{subject to: } y_i(\mathbf{w}^T x_i + b) \geq 1 - \xi_i, \xi_i \geq 0, i = 1, ..., N \end{aligned} \tag{2.23}$$

where ξ_i are the slack variables, one for each data point, and C is the slack penalty parameter that controls the relaxation of the slack variables – acting as a regularisation parameter. The greater the value of C, the more relaxed the classifier becomes to misclassifications. In the case of non-linear data, a simple linear hyperplane as formulated above is not enough to provide a good classification between the the classes. Fortunately, it is possible to project the data into a higher dimensional space using a kernel, where a linear hyperplane can separate the data. After a hyperplane has been found in that higher-dimension, the data is transformed back into

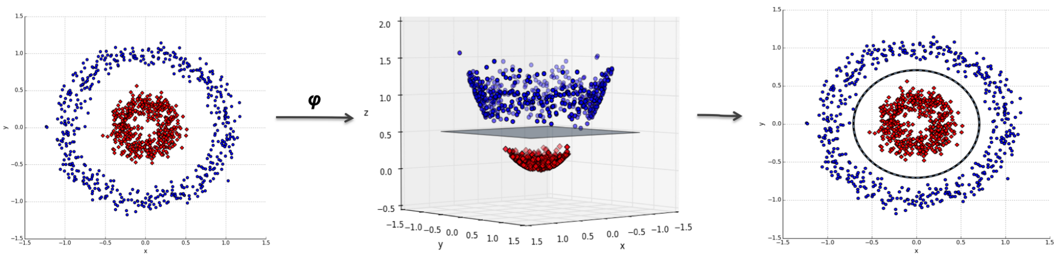

Figure 2.7: A non-linear binary classification problem. Left-hand side: plot of the two-dimension input data. Middle: plot of the transformed feature data with linear separating hyperplane. Right-hand side: Features space and hyperplane transformed back into input space to create a non-linear decision boundary.[6]

its original dimension. Figure 2.7 demonstrates this transformation procedure with a hypothetical kernel.

There are many different types of kernels that can be used with support vector machines, depending on the problem one is working on. Among the two most popular kernels that are used are: the Linear kernel and the Radial Basis Function kernel. The RBF kernel, shown in Equation 2.24, projects features in some input space into a feature space spanning an infinite number of dimensions, where γ is a hyperparameter known as the kernel coefficient which controls the width of the Gaussian.

$$K(\mathbf{x},\mathbf{x}') = \exp\left(-\gamma\|\mathbf{x}-\mathbf{x}'\|^2\right) \tag{2.24}$$

Generally, it is recommended that a linear kernel is tried first when training a SVM as it is a lot faster to compute – although, it is accepted that a non-linear kernel such as the RBF kernel will yield better results than a linear kernel. In fact, it has been proven that the linear kernel is a degenerate version of the RBF kernel and therefore will never perform better than a properly tuned RBF kernel [55].

Other hyperparameters that are important to take into consideration when dealing with imbalanced data are the *class weights* [56]. These hyperparameters are used to weight the slack penalty parameter by the classes of the data, such that

[6]http://beta.cambridgespark.com/courses/jpm/05-module.html

$C_k = w_k * C$, where w_k is a weight that is usually inversely proportional to class k's frequency. The idea behind this is to increase the penalty for misclassifying minority classes, as a way of preventing the classifier from being overwhelmed by the majority class. Usually, these class weights are set to 'balanced', which means that the class weights are equal to:

$$w_k = \frac{n_{\text{samples}}}{n_{\text{samples},k} n_{\text{classes}}} \tag{2.25}$$

where n_{samples} are the total number of samples, $n_{\text{samples},k}$ are the number of samples in class k, and n_{classes} are the total number of classes.

2.4 Machine Learning Practices

2.4.1 Cross Validation

When training a learning algorithm, there are many things to take into consideration to make sure the model being designed is both a powerful and robust predictor. One of the most important aspects of training is the idea of *hyperparameter tuning*. Hyperparameters are parameters within a machine learning model that are associated with the prior distribution of the problem and are set before the training of the model. These are different from parameters of the model, which describe the underlying dynamics of the system under analysis, which are derived through training. As such, a natural question to ask is how does one choose appropriate hyperparameters for their model?

One of the answers to this question is *cross validation*. The simplest form of this idea is called *hold-out* validation and works by splitting the data of a model into three sets: a training set, a validation set, and a testing set. An array of models are then trained using different sets of hyperparameters using the training set, and then they are evaluated using the data in the validation set. The model with the hyperparameters which gave the best performance on the validation set are then chosen and a new model is trained on the combined training and validation sets using these hyperparameters. Once the model has finished training, it is evaluated on the hold-out test set. The idea of having an intermediate validation set to fine-tune the parameters is to make sure that the model has not chosen hyperparameters to overfit to the test set – ensuring generality.

If a dataset only has a small number of data points, then splitting it into three sets could be harmful to the performance of a model. It is also known that deep learning models need a large amount of data to train effectively – the more data, the better the model can generalise to the true distribution the data follows. Another form of cross-validation, called *k-fold cross validation*, takes this into account and makes use of all of the data when training, tuning, and evaluating. Instead of having a single fixed validation set, a split of K partitions are made within the training set. A model is then trained with $K-1$ of these partitions with specific parameters, using the one remaining partition for validation. Each partition then takes a turn being the validation set, whilst the remaining $K-1$ partitions are used to train the model. This essentially allows the model to be trained and tested K times using different subsamples of the data, allowing for the robustness of a model to be tested. The hyperparameters between the models can now be compared in terms of not only their mean performances, but how their performances vary across different partitions of data, allowing for a more informed decision to be made. More analyses of these two cross validation methods can be found in R. Kohavi's paper [57].

2.4.2 Overview of Imbalanced Classes

Datasets come in all kinds of shapes and sizes and one very important property of a dataset is whether the classes are balanced or imbalanced. For example, take a dataset that contains images which are of 50% cats and 50% dogs, this would be a balanced dataset. On the other hand, a medical dataset containing records on whether a person has or does not have a certain type of a rare disease is an example of an imbalanced dataset. It is likely that this dataset contains more records of healthy people, perhaps 99%, than it does people with the disease, 1%. Training a classifier on the balanced image dataset is relatively easy, as a sensible metric to evaluate the performance of the model would be the accuracy – how many times did the model correctly predict the images as a percentage. Unfortunately, this metric would not work on the imbalanced dataset. Consider a model that is trying to predict whether a person has or does not have this rare type of disease. If the model only predicts that a person does not have the disease, and the accuracy metric is used, it will be correct 99% of the time. Clearly this is not a good model, as it does not provide any informative predictions and is biased towards the most frequent class in the dataset. As such, it is important to use an appropriate metric when dealing with imbalanced data to remove this bias in order to produce a useful predictor.

2.4.3 Precision and Recall

A better way to measure the performance of an imbalanced dataset is to take into account the true positives, true negatives, false positives, and false negatives. Briefly speaking, these are defined as:

- True positive - number of times a model correctly predicted a positive class.
- True negative - number of times a model correctly predicted a negative class.
- False positive - number of times a model incorrectly predicted a positive class.
- False negative - number of times a model incorrectly predicted a negative class.

One way to take these measures into account is by using the precision and recall metrics [58]. The precision is a measure on how often the classifier correctly predicts positive cases, whilst the recall is the ability of a model to find all the relevant cases within a dataset. In other words, they can be described in terms of the true positives, true negatives, false positives, and false negatives as seen directly below.

$$\text{precision} = \frac{tp}{tp + fp}$$
$$\text{recall} = \frac{tp}{tp + fn}$$

These metrics are particularly useful if you would like to maximise one, or the other. For example, in the case of classifying a rare disease, it would be preferable to maximise the recall whilst accepting a lower precision, if the follow-up expenses of further examinations are not too costly. Building the model in this way would ensure that people who have the disease will be correctly identified.

2.4.4 F_1 and $F_{\mathbf{max}}$ Scores

When working with an imbalanced dataset, it is sometimes desirable to train a model using an optimal blend of precision and recall. This can be achieved by taking the harmonic mean between the precision and recall, producing a metric known as the F_1 *score* [59].

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} \tag{2.26}$$

Using the harmonic mean between the precision and recall introduces an important property, namely that it penalises extreme values harshly, ensuring that the precision and recall do not deviate too far from each other. In fact, this is a special form of the generalise F-score metric given in Equation 2.27, which gives equal weighting to both the precision and recall. More emphasis can be put on one or the other by adjusting the parameter, β. The range of the F-score is defined by $F_\beta \in [0,1]\ \forall\ \text{precision, recall}$ where a F-score of 1 would be a perfect model.

$$F_\beta = (1+\beta^2)\frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{2.27}$$

More generally, given a target protein i and some decision threshold $t \in [0,1]$, the precision and recall for the individual protein can be defined as:

$$\text{precision}_i(t) = \frac{\sum_f I(f \in P_i(t) \wedge f \in T_i)}{\sum_f I(f \in P_i(t))} \tag{2.28}$$

$$\text{recall}_i(t) = \frac{\sum_f I(f \in P_u(t) \wedge f \in T_i)}{\sum_f I(f \in T_i)} \tag{2.29}$$

The precision-recall space can then be constructed, for a fixed threshold t, by averaging precision and recall across all proteins.

$$\text{precision(t)} = \frac{1}{m(t)} \sum_{i=1}^{m(t)} \text{precision}_i(t) \tag{2.30}$$

$$\text{recall(t)} = \frac{1}{n} \sum_{i=1}^{n} \text{recall}_i(t) \tag{2.31}$$

where $m(t)$ is the number of proteins in which at least one prediction was made above the threshold t, and n is the number of proteins in a target set.

Again, a single metric can be found by taking the harmonic mean between precision and recall to retrieve a F_1 score. This time, the threshold t will be varied until the largest F score is obtained. This is called the F_{max} score [3].

$$F_{\text{max}} = \max_t \left[\frac{2 \cdot \text{precision}(t) \cdot \text{recall}(t)}{\text{precision}(t) + \text{recall}(t)} \right] \tag{2.32}$$

Chapter 3

Methodology

3.1 Overview

Deep learning models often contain a large number of parameters, requiring vast amounts of data to train effectively. Due to the restrictions on the data for protein function prediction–particularly, making sure that it contains an adequate number of orphan proteins and distant proteins–large corpora of protein sequences are not available for this task. Additionally, it is a commonly known fact that machine learning models struggle to learn sparse multi-labelled tasks. As a result, it has proved difficult to train an effective classifier using deep learning–and as it stands, the state-of-the-art model transforms the problem into an array of single-labelled binary classification tasks using SVMs. With this in mind, it would be interesting to somehow incorporate the power of deep learning whilst still using the techniques that are most effective for predicting the functions of proteins.

The proposed idea is to train a multi-label LSTM on sequential biophysical features of the proteins, in the hopes that the network will be able to encode sequence-dependent functional representations. If the sequential features are informative enough to provide this signal, the functional feature-representations of the protein sequence that give rise to this signal will be extracted and used as a feature-embedding to train a SVM for protein function prediction. The hypothesis is that by using these LSTM embeddings, which contain high-level abstractions of the protein

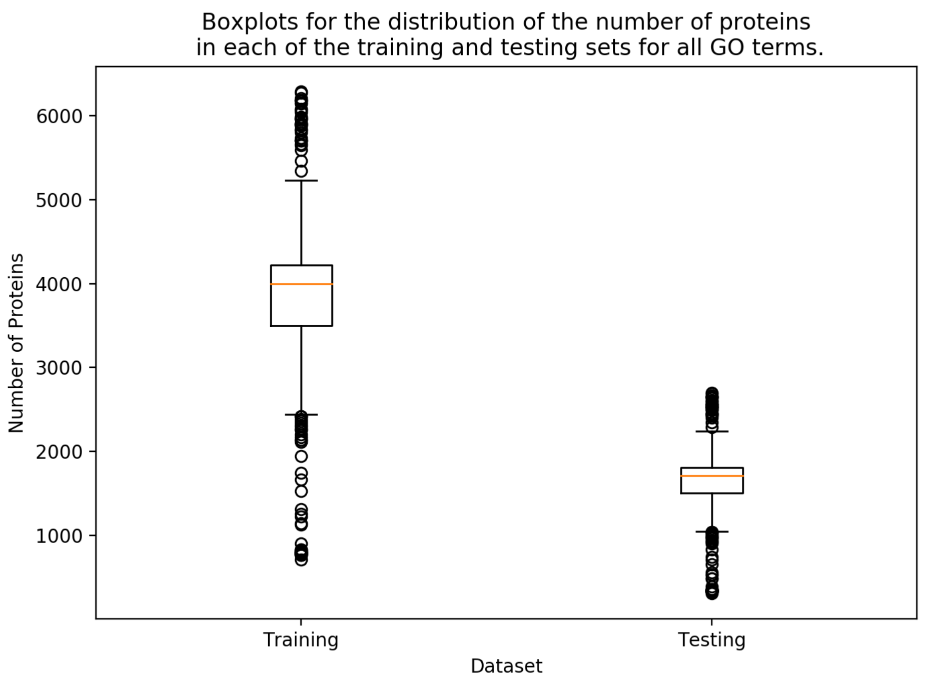

Figure 3.1: A boxplot showing the distributions in the number of proteins in the training and testing sets for all of the GO terms.

sequences, the decision boundary between the positive and negative proteins of a given function will become more distinctive and easier to discriminate between. The SVMs will then be used as a way to transform the multi-labelled problem into a set of single binary-output problems, which will help alleviate the sparsity in the label-space.

3.2 Data

Data is an incredibly important part of the machine learning process – without data, there would be no model to learn. Not only that, but it is important that the data is managed and dealt with in a methodical and correct manner, as it is very easy to come across bias that could render the learned model's performance obsolete. Therefore, this section will give an overview of the kind of data used in this thesis and how it was partitioned in such away to avoid testing bias.

The data used in this thesis is the same protein data that was used for FFPred

2.0 and 3. Specifically, the training procedures employing the term definitions and relationships defined in the GO OBO flat file released on 2015-02-03, the annotations for human proteins in UniProt-GOA released on 2015-04-02 and in UniProtKB release 2015_02, and the UniRef90 release 2015_02 for sequence similarity searches, which can all be obtained by running a script found on the psipred GitHub (https://github.com/psipred).

The protein data in its raw form consists of a directory of 23095 proteins, containing information on 13 feature groups. Examples of these feature-groups include:

- Files generated from running PSIBLAST on the sequence,
- Structure prediction from running PSIPRED,
- Disorder prediction from running DISOPRED,
- Sub-cellular location prediction from running WoLF PSORT.

Along with this information about all the protein sequences, a directory of files containing a specific training and testing set of proteins for each GO term is provided, outlining all the proteins that have been assigned to that specific function. These training and testing sets are carefully designed in order to make sure that similar protein sequences are not found within both the sets. Otherwise, this would cause a testing bias when evaluating on the test set, leading to false performance evaluations. Each experiment will ultimately be trained and tested using these sets for each GO term, which will be known as `Protein GO sets`. A boxplot showing the distribution of the number of proteins for the training and testing sets for each of the GO terms can be seen in Figure 3.1.

The data used to train the LSTMs for feature extraction will be slightly different. Since we are ultimately going to extract these representations to use as inputs to a SVM, as a way of transforming the problem from a multi-label one to an array of single binary-outputs, it is not necessary to form these 'special' data sets for each

GO term. Also considering that deep learning models generally require a lot of data to train effectively, these LSTMs will be trained using as much protein data as possible. It is not as simple as just passing the whole protein data set into a single LSTM to build up a representation, as this contains proteins in the Protein GO test sets for each specific GO term. Doing this would cause the LSTM representations to contain information about the test proteins, and therefore cause testing bias when evaluating the SVMs performance on these test sets. Instead, when training the LSTM for each GO term, the data set used will be the whole protein data set with the protein test set of the specific GO term removed. Unfortunately, this means that instead of only having one LSTM to encode representations, there will have to be a LSTM for each GO term under consideration. These types of data sets will be known as `LSTM training sets`.

3.3 Reducing the number of labels in the dataset

Proteins have a large amount of potential functions according to the GO. Analysis of the 23095 proteins in the dataset shows that there are a total of 999 different GO terms and on average each protein is only assigned to 30 different functions. This makes the protein function prediction label-space incredibly sparse and imbalanced, leading to great difficulty when trying to train a classifier. Also, it should be noted, as mentioned in the previous section, a LSTM needs to be trained for each GO term considered, in order to prevent testing bias within the evaluation of the SVM. Therefore, reducing the number of labels under consideration is desirable and there are a few ways in which this can be done. First, each of the GO sub-ontologies can be considered individually – that is, a separate model for molecular function, biological process, and cellular component can be created. Naturally, by doing this, one must be considerate about the reduction in the amount of proteins that can be used for training data for each model. Fortunately, since proteins share many functions across all domains, only a few thousand proteins are independently redistributed across the domains, resulting in only a small reduction from the original dataset. Table 3.1 shows how the number of GO terms are reduced, as well as

	Domain			
	MF	**BP**	**CC**	**Total**
Number of Proteins	17964	19258	18435	23095
Number of GO Terms	158	723	118	999
Average No. Assigned	5	20	5	30

Table 3.1: A table representing the number of proteins, number of GO terms, and the average number of assigned GO terms to each protein across the different GO sub-ontologies

the number of remaining proteins in each model, after the domains are considered separately.

A second, more effective way, to reduce the number of the labels is by considering the GO hierarchical structure. Recall that GO is a directed acyclic graph of functional terms and as such each term in the graph have children nodes, except for the end-leaves. These end-leaf nodes express the most detail about the function of a protein out of all of its ancestor nodes. For example, a protein could express the 'lactase activity' function, which in the GO graph follows the path: molecular function $\rightarrow$ catalytic activity $\rightarrow$ hydrolase activity $\rightarrow$ hydrolase activity, acting on glycosyl $\rightarrow$ hydrolase activity, hydrolyzing $\rightarrow$ lactase activity. Anything above lactase activity in the graph contains less information about the function, so is considered degenerate. Therefore, locating and removing all degenerate functions within the dataset would reduce the total number of labels, whilst preserving all the information about the function of the protein. There are many different levels in which the degeneracy threshold can be set, but for simplicity only the lowest-level GO terms will be considered – any protein that lacks the presence of one of these lowest-level GO terms will be removed from the dataset. Table 3.2 shows the effect on carrying out this procedure, showing a considerable decrease in the number of GO terms that need to be considered in each domain. The table also shows the number of proteins that remain in each domain after removing the proteins that do not have any end-leaf node GO terms assigned to them. Quite surprisingly, only a few thousand proteins were removed from each domain, leaving a nice amount of data for the training of the LSTM models.

	Domain			
	MF	**BP**	**CC**	**Total**
Number of Proteins	13146	15449	16276	20541
Number of GO Terms	77	381	82	540
Average No. Assigned	1	5	3	9

Table 3.2: A table representing the number of proteins, number of GO terms, and the average number of assigned GO terms to each protein across the different GO sub-ontologies after degenerate GO terms have been removed.

3.4 Model Design

An overview of the entire model can be seen in Figure 3.2. The first step will be to extract the sequential features for each protein sequence by parsing the raw protein data files. LSTMs will be trained using these extracted raw features for each GO term for function prediction. Once the LSTMs have been sufficiently trained, the raw features for the proteins that make up the Protein GO sets will then be parsed through their corresponding trained LSTM to create LSTM-feature representations for that specific GO term. Using these transformed feature representations, a SVM will be trained for each GO term, similar to FFPred's model, to predict whether or not a protein possesses this function.

3.5 Extracting Sequential Features

For each protein, an array of raw data files about the properties of the sequence have been supplied. This includes information on the secondary structure prediction, disorder prediction, subcellular location prediction, BLAST prediction data, etc. Unfortunately, many of these properties are not sequential, so will not be considered when building the LSTM-feature representation of the proteins. Instead, these non-sequential features can be considered alongside the LSTM representations during the training of the SVM, which will not be considered here.

A few sequential features that can be considered within the data files are:

- Amino acid sequence - the 21 different amino acids that make up the sequence.

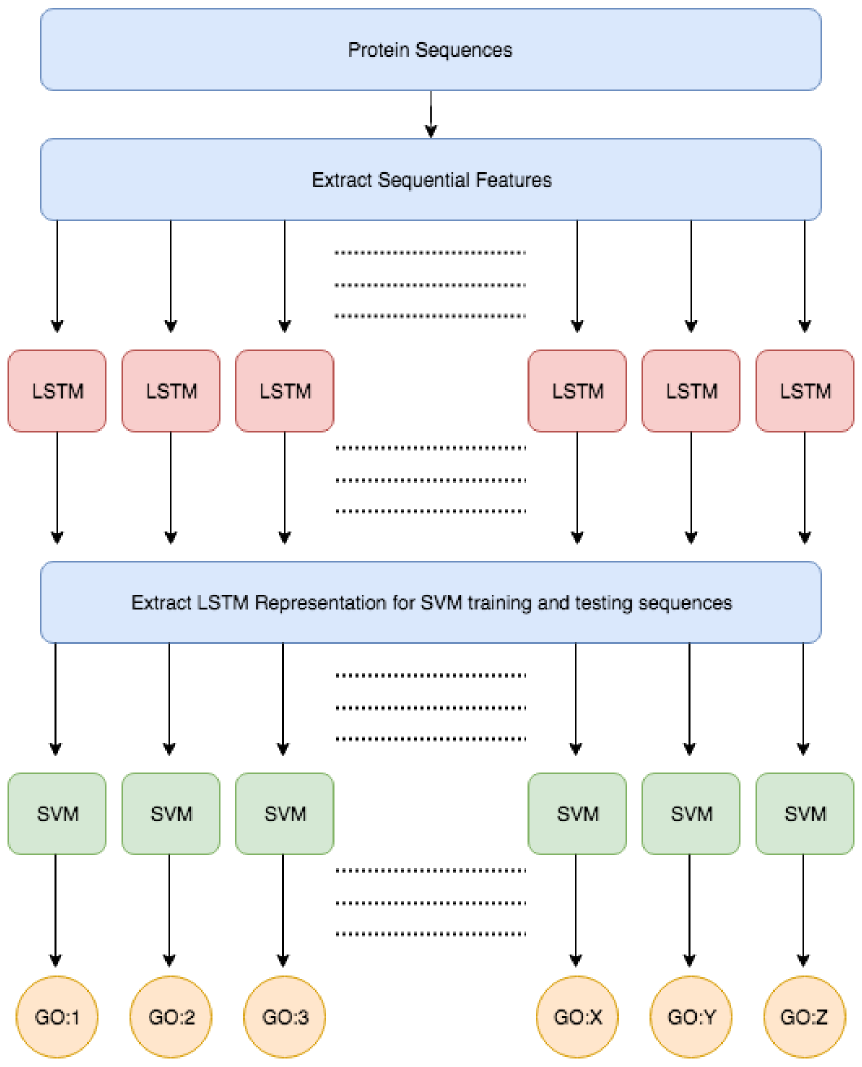
Protein Sequences
Extract Sequential Features
LSTM
LSTM
LSTM
LSTM
LSTM
LSTM
Extract LSTM Representation for SVM training and testing sequences
SVM
SVM
SVM
SVM
SVM
SVM
GO:1
GO:2
GO:3
GO:X
GO:Y
GO:Z

Figure 3.2: A schematic diagram of the pipeline process of the model. First, sequential features are extracted from the protein sequences. A LSTM for each GO term is then trained using these features. The protein sequences are then passed through these LSTMs to extract a high-level feature-representation, which are then passed into their corresponding SVMs for the function prediction of each GO term.

- Secondary structure prediction - for each amino acid, probabilities for that part of the sequence being either a coil, helix, or extended strand in parallel and/or anti-parallel -sheet conformation are given.
- Protein disorder prediction - for each amino acid, a probability is given indicating whether that region is disordered or not.
- N-Glycosylation prediction - for each amino acid, a binary output 1 or 0 is given depending on whether that region is predicted as a N-glycosylation site or not.
- O-Glycosylation prediction - for each amino acid, a binary output is given depending on whether that region is predicted as O-glycosylation site or not.

Due to computational considerations and time constraints, not all sequential features can be used when training the LSTMs. Preliminary testing shows that the PSIPRED secondary structure features offer the most valuable information when building a classifier, and as such, this will be the sequential feature we will use to build our LSTMs.

3.6 Construction of the Multi-label LSTMs

3.6.1 Overview

The type of model that will be considered for the learning of functional-feature representations will be a bi-direction LSTM. This is well suited to this problem, as it will allow temporal structures throughout the protein sequences to be captured within the encoded representations as well as allow for long-term dependencies within the sequences to be taken into account.

The LSTM is a parametric model, meaning that the number of parameters throughout the model for different sets of data remains constant. Consequently, this means that the inputs to the LSTM must all be the same size. Protein sequences are not all the same length, so will be unable to satisfy this requirement without some sort of modification. A common way to deal with this problem is to pad all the sequences

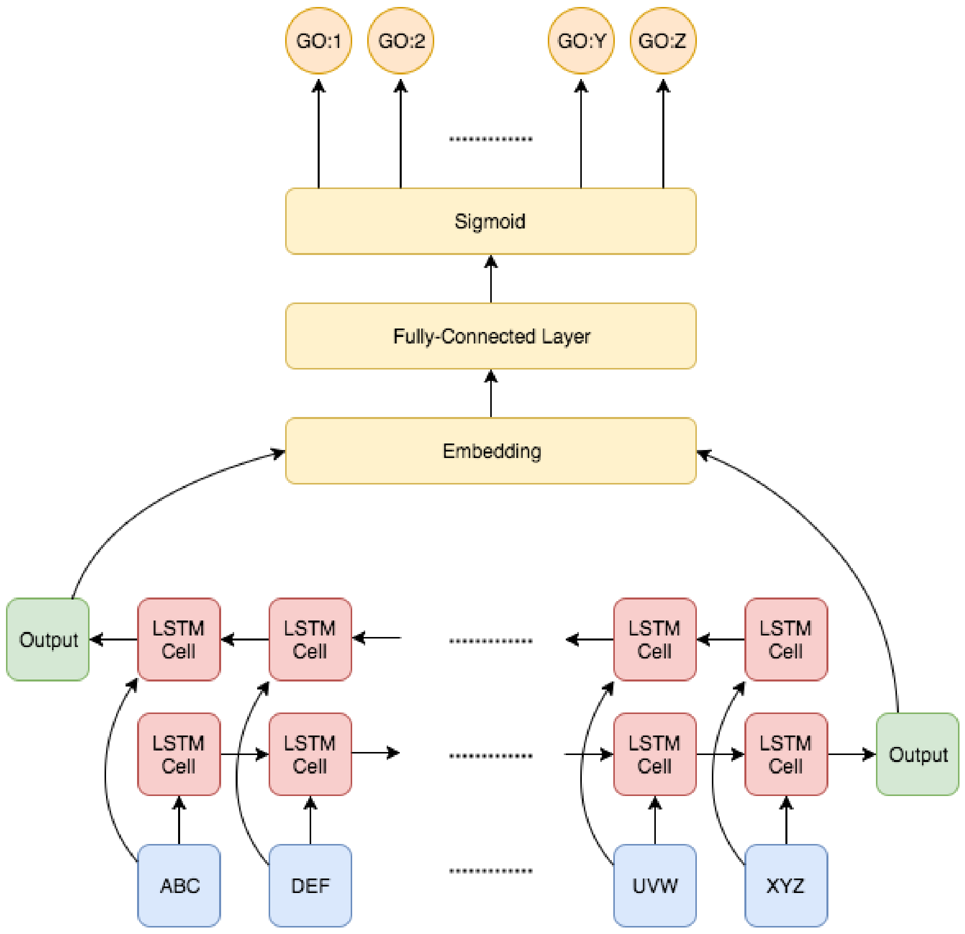

Figure 3.3: A schematic diagram of the LSTM model that will be used for building up high-level feature-representations.

to the length of the longest sequence in the data set, making them all the same size and able to be passed through the LSTM. The unfortunate downside to padding is that it introduces a lot of unnecessary noise that could potentially hinder the overall performance of the model.

Once the raw sequential features are in an appropriate format, they will be passed into a LSTM. An important consideration is the number of ’time-steps’ each LSTM cell takes as input. Although LSTMs have better gradient retention than vanilla RNNs, they are not completely rid of the vanishing gradient problem and often only remember up to 100 steps of a sequence. The average length of a protein in this data set is around 450 amino acids long, therefore considering only a single amino acid for each step of the LSTM would be detrimental to its performance. As a result, amino acids can be grouped together by concatenating their features into one input of a LSTM cell, this allows more information about the sequence to be preserved through the LSTM. A natural question to then ask is, what is the best time-step to use? As such, the time-step will become an important hyperparameter to be tuned in the training process.

When the LSTM embedding of a protein sequence has been calculated, it will be passed through a fully-connected layer, mapping it to an output with a dimension equal to the number of GO terms under consideration. A sigmoid activation function will then be applied to convert the outputs into probabilities and subsequently a decision can be made on each of the GO terms for that protein sequence. Although we are only interested in extracting the LSTM-representation for each protein sequence, it is necessary to pass this through the fully-connected layer to produce an output, so backpropagation can be performed in order to train the network to produce these powerful representations.

3.6.2 PyTorch

Deep Learning models can become incredibly complicated, requiring a large amount of effort to program from scratch. Fortunately, a number of deep learning frameworks have become readily available making the network building procedure considerably more easy. PyTorch, one such example of a deep learning framework, is an open-sourced machine learning library in Python that offers fast and flexible experimentation [60]. PyTorch offers a high-performance environment to build and create machine learning models, using automatic differentiation at its core to easily and efficiently carry-out backpropagation in even the most complicated models across many devices (CPU and GPU).

Pythonic in its design, PyTorch makes it very easy to create machine learning models. At the base of all neural network modules is the `nn.Module` object, containing all the necessary variables and methods to build a machine learning model. All neural methods should inherit from this object, defining its architecture within the constructor and overriding the `forward` method to define what computation should be carried out when performing a forward pass. Once the model class has been created and instantiated, only a few more lines of boilerplate code need to be written in order to setup which optimiser and loss function is going to be used during training. PyTorch offers a large number of readily-available optimisers and loss functions callable from modules. Once the algorithm has been setup, it is just a matter of passing the data through in batches and calling methods to perform backpropagation and an update in the weights. Note, that it is important that for each batch, the gradients that were accumulated from the last batch are reset before performing backpropagation. Otherwise, the update step will not correctly adjust the weights of the parameters to minimise the loss function. The basic outline for this procedure can be seen below:

```
class Model(nn.Module):
    def __init__(self, network_parameters):
        # Instantiate network parameters
    def forward(self, data):
        # Define computation for a forward pass
        return output
# Instantiate model
model = Model(network_parameters)
# Define optimiser on the model parameters.
optimiser = torch.Optimiser(model.parameters())
# Define the loss function.
criterion = nn.LossFunction()
# For each epoch
for epoch in range(epochs):
    # For each batch in the data
    for batch in training_data :
        # Features and labels
        X, y = batch
        # Forward pass of features
        output = model(X)
        # Calculate loss between output and labels.
        loss = criterion(output, y)
        # Reset the optimiser gradients to zero
        optimizer.zero_grad()
        # Retrieve parameter gradients.
        loss.backward()
        # Use gradients to adjust parameters.
        optimizer.step()
```

Using this procedure, a bi-directional LSTM, like that seen in Figure 3.3, was setup with the additions of Xavier initialisation of the parameters, batch-normalisation between the layers and dropout on the fully-connected layer. Within the forward method of the LSTM, a special function called `pad_packed_sequence` was used on the data before passing them through the network, allowing variable-sized sequences to be accepted as input. This ensures that the padding within the sequences will not contribute to the update in the parameters, reducing the noise that would otherwise incur. The full implementation of this code can be found on GitHub, link supplied in the Appendix.

3.6.3 Hyperparameters

During training, there will be many important hyperparameters to consider:

- The number of hidden layers,
- The number of neurons in each layer,
- The number of time-steps for each LSTM cell,
- The dropout probability value,
- Which gradient descent optimiser to use.

As mentioned previously, a LSTM for each considered GO term will have to be created in order to overcome testing bias. Unfortunately, this results in the training and testing of hundreds of LSTMs, which is an incredibly computationally expensive task. For each set of hyperparameters, the number of LSTMs to be trained grows exponentially and therefore, time does not permit the search of a large hyperparameter space.

To reduce the hyperparameter space, the first decision will be to choose which hyperparameters are the most important to test, giving rise to the largest effect on the final encoded representation. Intuitively, these were chosen to be:

- The number of hidden layers,
- The number of neurons in each layer,
- The number of time-steps for each LSTM cell.

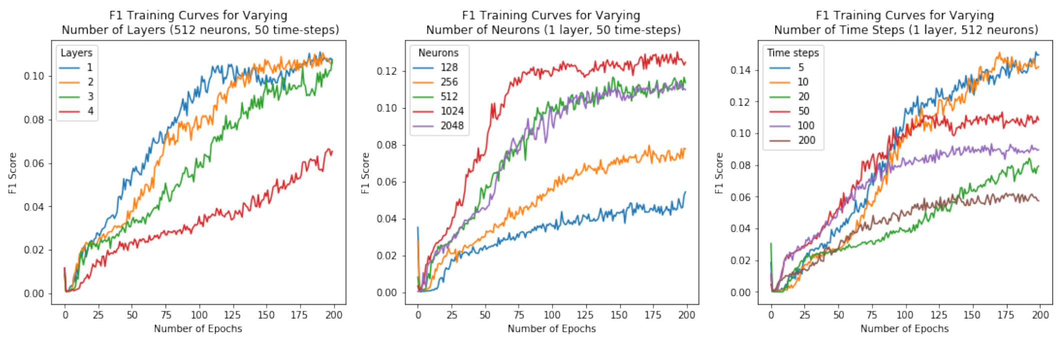

Figure 3.4: F_1 training curves for the hyperparameter preliminary tests. Left: varying number of layers. Middle: varying number of neurons. Right: varying number of time-steps

Additionally, further reduction in the hyperparameter space was performed by carrying out preliminary tests in search for an appropriate search-space domain. By using the whole protein sequence data, with a 70:30 training-test split, the following networks were trained to test the approximate performance of different sets of hyperparameters.

- Fixed neurons (512), fixed time-step (50), and varying layers (1, 2, 3, 4).
- Fixed layers (1), fixed time-step (50), and varying number of neurons (128, 256, 512, 1024, 2048)
- Fixed neurons (512), fixed layers (1), and varying time-step (5, 10, 20, 50, 100, 200).

These experiments will give us an approximate idea on how the networks behave as the hyperparameters are varied. Since the hyperparameters are varied independently from each other, this method is not directly conclusive for finding the best hyperparameters for the final experiments, but it should serve as an indicator on what kind of range the parameters should take.

The F_1 training curves for each of the experiments can be seen in Figure 3.4. It was found that generally, lower values of time-steps seem to perform best. As the number of neurons in a layer increase, the performance of the model also increases up until a point (2048 neurons), suggesting that perhaps the neurons are not being saturated and the model is unable to fully learn the data. Interestingly, adjusting the number of layers in the network has no great effect on the performance, until four layers are used, where a considerable drop in performance was found. This is perhaps from the neural network possessing too many parameters, which are unable to be effectively trained, given the amount of data available.

When choosing the range for each hyperparameter, it also should be noted that there is an increase in computational expense when: the number of time-steps are decreased, the number of neurons in a layer are increased, and if the number of layers are increased.

3.7 Construction of the SVMs

Once the LSTMs for each specific GO term has been trained, the SVM sequence data in the Protein GO data sets, corresponding to each of the GO terms, are passed through the model and the feature embeddings are extracted from the last time-step of the LSTM. These feature embeddings are used to train and test the SVMs.

3.7.1 Scikit-Learn

Scikit-Learn is a popular Python package used for various classification, regression and clustering algorithms including support vector machines [61]. Not only does it allow the easy use of classical machine learning algorithms, but it also provides a simple way of performing a cross-validation hyperparameter search. Scikit-Learn makes it incredibly simple to instantiate and train a model, requiring only two lines of code. Performing a cross-validation hyperparameter grid search requires two more lines of code to specify the dictionary of hyperparameters to search, as well as the method that takes in the model and the dictionary of hyperparameters to perform the grid search. The general procedure can be seen below:

```
model = sklearn.svm.SVC()
hyperparameter_dict = dict(hyperparameters)
clf = GridSearchCV(estimator=model, param_grid=hyperparameter_dict)
clf.fit(X_train, y_train)
```

Once the cross-validation hyperparameter grid-search has been carried out, the best performing set of hyperparameters for each GO term are used to train the final SVM models, which are evaluated on the testing data set.

3.7.2 Hyperparameters

As with the training of the LSTMs, hyperparameter selection is a crucial part when designing the SVMs. For each SVM, there are many different hyperparameters that can be considered, for example:

- The type of kernel to be used (RBF, Linear),
- The penalty parameter on the error, C.
- The kernel coefficient of the RBF kernel, γ.
- Balanced or imbalanced class weights.

Since the SVM is a much smaller model than the LSTM, a hyperparameter space reduction does not need to take place, as each model is fairly quick to train. The range of the hyperparamters that will be used are those that were used in the FFPred 3.0 experiments.

3.8 Experimental Design

3.8.1 Evaluation of the Predictive Performance of LSTM Classifier Using the Sequential Features

The first test to be conducted is whether a LSTM with the architecture discussed previously and shown in Figure 3.3 can be used for the classification of protein functions given a protein sequence. Ideally, we would like to see whether a LSTM can pick up on any signal using the sequential features of the protein sequences and whether this signal can be used to make informative decisions about the function of proteins.

A LSTM network will be trained for each GO term considered, predicting a single binary output on whether a protein has the corresponding function or not. The networks will be trained using the sequential psipred feature and undergo a hyperparameter search concerning the number of neurons in each layer, as well as the number of time-steps considered for each LSTM cell. This hyperparameter search will be conducted using a 70:30 train-test split of the Protein GO training data sets and the models will ultimately be evaluated on the Protein GO test data sets.

3.8.2 Evaluation of the Predictive Performance of SVM Classifier Using the LSTM Representations

The second test to be conducted is the core idea of this thesis. That is, whether functional feature representation of a LSTM can capture the long-term dependencies within protein sequences and whether this can be used to train a powerful SVM classifier for protein function prediction. Similar to the previous experiment, a LSTM network will be trained for each GO term considered, but instead of a binary output, it will have a multi-label output containing all considered GO terms. This is so the final learned representations are enhanced with as much protein data as possible and that the correlation between all GO terms can be captured. A hyperparameter search will be performed using a 70:30 train-test split on the LSTM training data sets and the best chosen hyperparameters for each of the GO terms will be used to train the final LSTM networks. Once these networks have been trained, the protein sequences in each of the Protein GO sets will be parsed through the corresponding networks and the feature-embeddings from the last time-step of the LSTM will be extracted. These extracted representations will be used for the training, tuning, and testing of the SVMs for each GO term. hyperparameter tuning of the SVMs will take place using five-fold cross validation.

3.8.3 Evaluation of the Predictive Performance of SVM Classifier Using the Raw Features

The final experiment to be conducted is training and testing SVM classifiers using the raw features of the protein sequences. The raw features under consideration will be those predicted by psipred, concerning the secondary structure of the protein sequences. This will allow for fair testing when comparing to the performance of the SVM classifier using the LSTM representations. Similar to the previous experiment, each SVM will undergo hyperparameter tuning using five-fold cross validation derived from the training data of the Protein GO sets. Using the best hyperparameters for each GO term, a final SVM will be trained using the whole of the training data and then evaluated on the test data in the Protein GO sets.

3.9 Compute and Parallelisation

Deep learning models are generally computationally expensive and are typically run on Graphical Processing Units (GPUs), where matrix computation can be run in parallel over many cores. Since the experiments conducted require the training and testing of many hundreds of LSTMs, how these models should be run in parallel needs to be considered. Although there is access to multiple GPUs, most of these are occupied by other members of the research group. As a result, the experiments to be conducted still require serially running hundreds of LSTMs on GPUs, taking upwards of weeks to finish. Not only that, but the GPUs are manually reset every so often without warning, further increasing the amount of time for the experiments to finish. Taking these factors into account, it was decided to run the experiments over a cluster of CPUs instead. Although CPUs possess less computational power than GPUs, many hundreds of models could be trained in parallel due to the vast number of units given, drastically speeding up the overall run-time. Furthermore, these CPU clusters are more stable than the GPUs; only going down for maintenance every so often, resulting in a much longer up-time.

3.10 Consideration of Constraints

Due to the nature of the project, as well as the short amount of time that it spans, a number of constraints had to be to considered in order to make sure that the experiments could be conducted and completed in time. One of the major constraints was the number of GO terms considered when training the LSTMs, due to the incredible computational expense. For this reason, a decision was made to only build models for the GO terms that are part of the biological process GO domain, as this has proven to be the most difficult for protein function predictors to perform on. Additionally, time only permitted for the completion of 121 of the 381 GO terms that exist in this domain.

Another critical constraint was the large reduction in the hyperparameter space for the LSTM models. The only two parameters that will be considered in the final experiments are the number of neurons in each layer and the length of time-step considered for each LSTM cell. Within these hyperparameters, only a small number of values will be tested, as a new LSTM has to be trained for each hyperparameter set, resulting in a very large computational problem. Not only that, but small-values of time-steps and large number of neurons have to be avoided due to the exponential growth in the amount of time for the networks to train – even though an increase in performance would be expected.

Finally, the LSTM hyperparameters will be tuned using a train-test grid-search, rather than a k-fold cross validation, which will ultimately lead to a less robust model and perhaps a sub-optimal performance. Usually, these types of models will be tuned using specifically crafted folds, making sure the models are robust as possible, not succumbing to protein-similarity bias in the test sets as a result of having similar proteins in both the training and test sets. Unfortunately, given the time constraints, it is not sensible to train k (usually 10), LSTMs for each GO term for each hyperparameter set.

Chapter 4

Results

4.1 Predictive Performance of LSTM Classifier Using the Sequential Features

The first experiment conducted was training a LSTM for each GO term in the protein SVM GO data sets by using the sequential psipred features and using it as a classifier. This is to see whether a signal can be captured from the sequential features of a protein sequence alone. Each of the LSTMs were trained for 200 epochs using the Adam optimiser, batch normalisation, Xavier initialisaton, and dropout with a probability of 0.5. A hyperparameter grid-search was performed using a train-test split with the following hyperparameter search-space:

- Number of Neurons: [256, 512]
- Number of Time-Steps: [10, 20, 50]

Figure 4.1 shows the performance of the LSTM classifier using the raw sequential psipred features on 121 GO terms in the form of a bar chart and a boxplot. As it can be seen, the LSTM is able to pick up on a signal and use it for the classification for most of the GO terms. Specifically, a F_1 score above 0.6 was achieved in 17 of the GO terms, which would be considered successful classifications. Overall, the classifier resulted in a mean F_1 score of 0.23. Although the psipred features were enough to pick up on a signal for a lot of the GO terms, around 30% of the GO terms have a F_1 score of 0, which is what we would expect due to the imbalanced nature

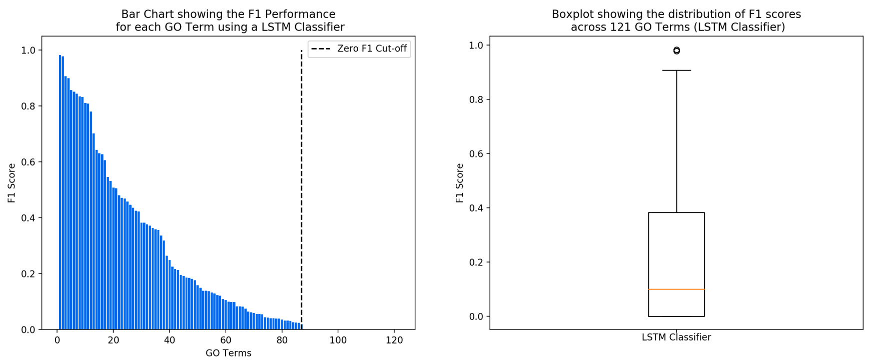

Figure 4.1: Left: A bar chart showing the performance of the LSTM classifier, in terms of the F_1 score, for 121 GO terms. Right: A boxplot displaying the distribution of the F_1 scores across the 121 GO terms.

of the problem and the lack of data to sufficiently train a LSTM on the protein SVM GO sets.

4.2 Predictive Performance of SVM Classifier Using the LSTM Representations

The next experiment conducted was training a SVM for each GO term in the protein SVM GO data sets using the representations extracted from a LSTM trained on the LSTM training data sets. Likewise to the LSTM classifier, each LSTM was trained for 200 epochs using the Adam optimiser, batch normalisation, Xavier initialisation, and dropout with a probability of 0.5 as well as using the same hyperparameter tuning method and search-space. Figure 4.2 shows the F_1 performance distribution over the hyperparameter space as a series of boxplots for training the LSTM representations. It can be seen that in general, when the number of neurons in a layer increases, the performance of the LSTMs increase. This result is in agreement with the preliminary results carried out earlier. Interestingly, it can also be seen that as we decrease the number of amino acids that are taken into consideration for each time step of the LSTMs, the variance increases across the GO terms. After the LSTM representations have been trained and the best-set of hyperparameters have been chosen for each GO term, the protein GO SVM sequences are passed through their corresponding networks and their feature-representations extracted.

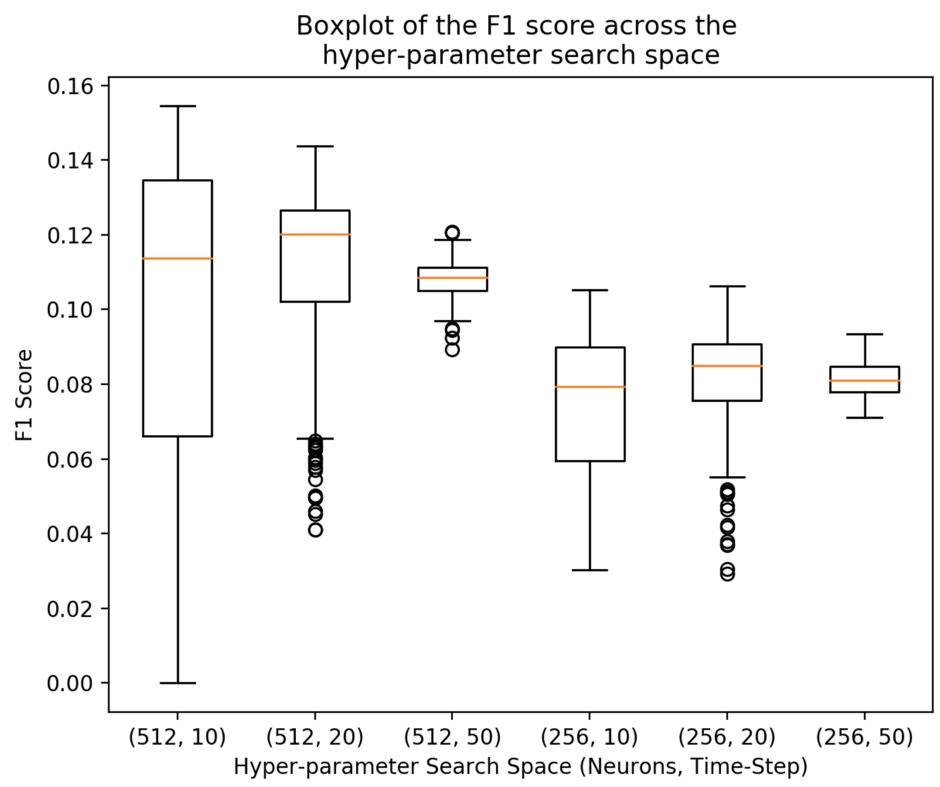

Figure 4.2: A series of boxplots displaying the distributions of F1 scores over all GO terms considered for different settings of the hyperparameters.

Once the representations have been extracted, a SVM classifier is trained using the corresponding representations for each GO term. A five-fold cross validation is conducted to choose the best hyperparameters using the following hyperparameter search space:

- Penalty Coefficient C: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]
- Kernel: [Linear, RBF]
- Gamma (if RBF kernel): [0.0001, 0.001, 0.01, 0.03, 0.1, 0.2, 0.5, 3, 10]
- Class weight: [Balanced, Non-Balanced]

Figure 4.3 shows a scatter plot comparing the F_1 performances of the LSTM classifier and the SVM classifier using the LSTM representations for each GO term. Points below the diagonal line correspond to GO terms that are classified better

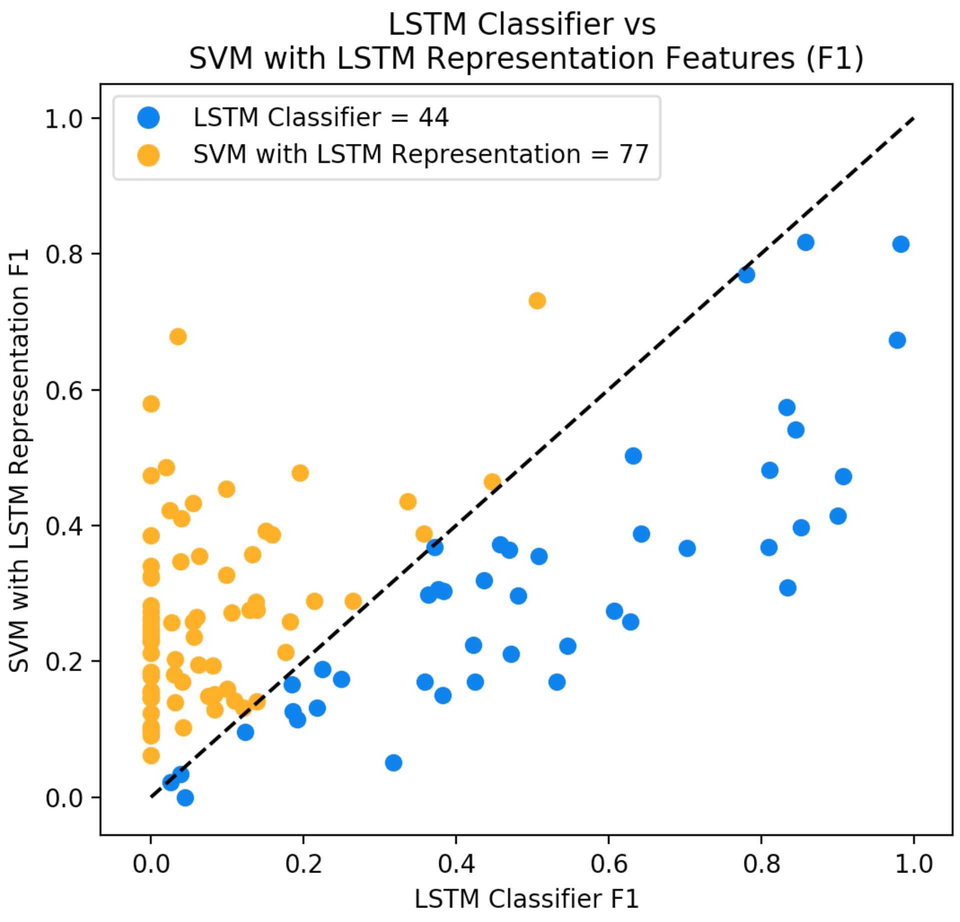

Figure 4.3: A scatter plot comparing the performance of the LSTM classifier and the SVM classifier using the LSTM representations for each GO term. It can be seen that overall, the SVM classifier using the LSTM representations perform better than the LSTM classifier for 64% of the GO terms.

using the LSTM classifier, whilst the points above the line correspond to the GO terms that are classified better using the SVM classifier trained on the extracted LSTM feature-representations. The plot shows that using the SVM classifier with LSTM representations achieves a better classification in 64% of the GO terms – resulting in a signal for all of the GO terms with an average F_1 score of 0.28. A non-parametric statistical test known as the Wilcoxon signed-rank test was carried out to analyse the statistical significance in the difference between the two methods. A p-value of 0.007 was found, suggesting a strong statistically significant difference between the two methods.

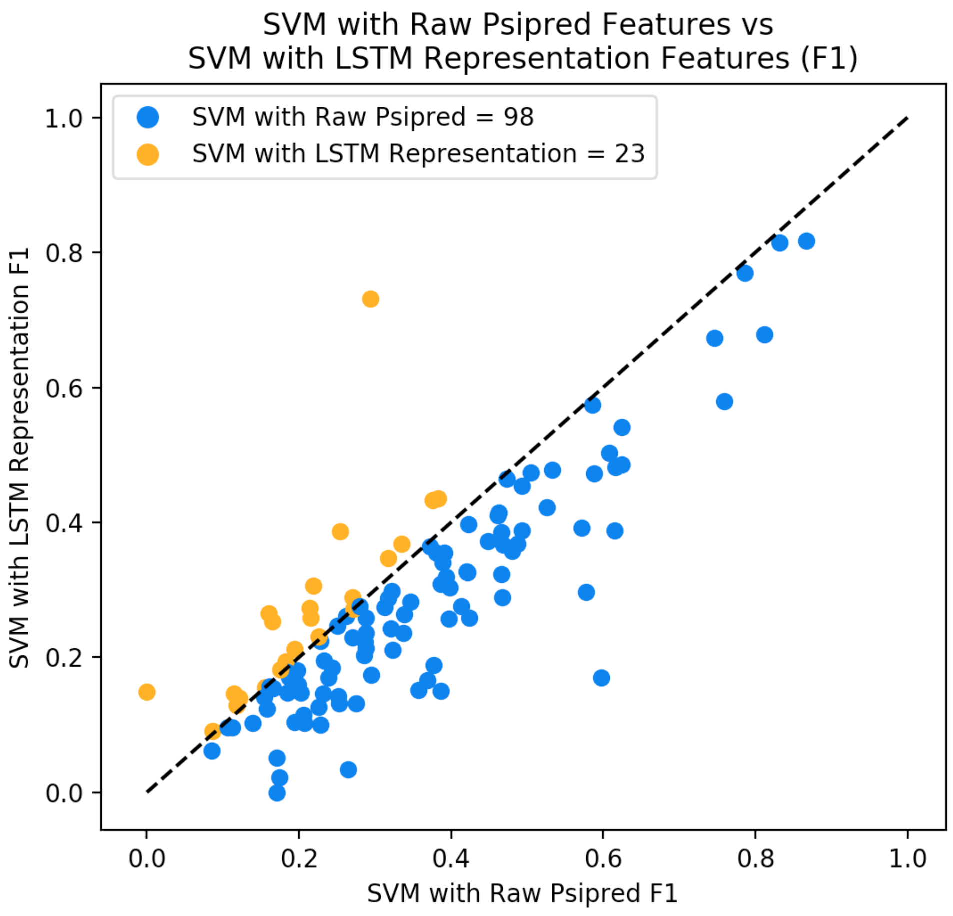

Figure 4.4: A scatter plot comparing the F_1 performances of each GO term between the SVM trained with the raw psipred features and the SVM trained with the LSTM represenations.

4.3 Predictive Performance of SVM Classifier Using the Raw Features

Extracting the LSTM representation from the protein sequences and using them to train a SVM classifier performs better than a stand-alone LSTM classifier, but how does this compare to a SVM trained with the raw secondary structure features of a sequence generated by psipred? The last experiment to be conducted was training a SVM using the raw psipred features for each of the 121 GO terms. Similar to the previous experiment, the SVMs were trained using a five-fold cross validation using the same hyperparameter search space. This is to make sure that the tests are being conducted fairly in order for the methods to be comparable.

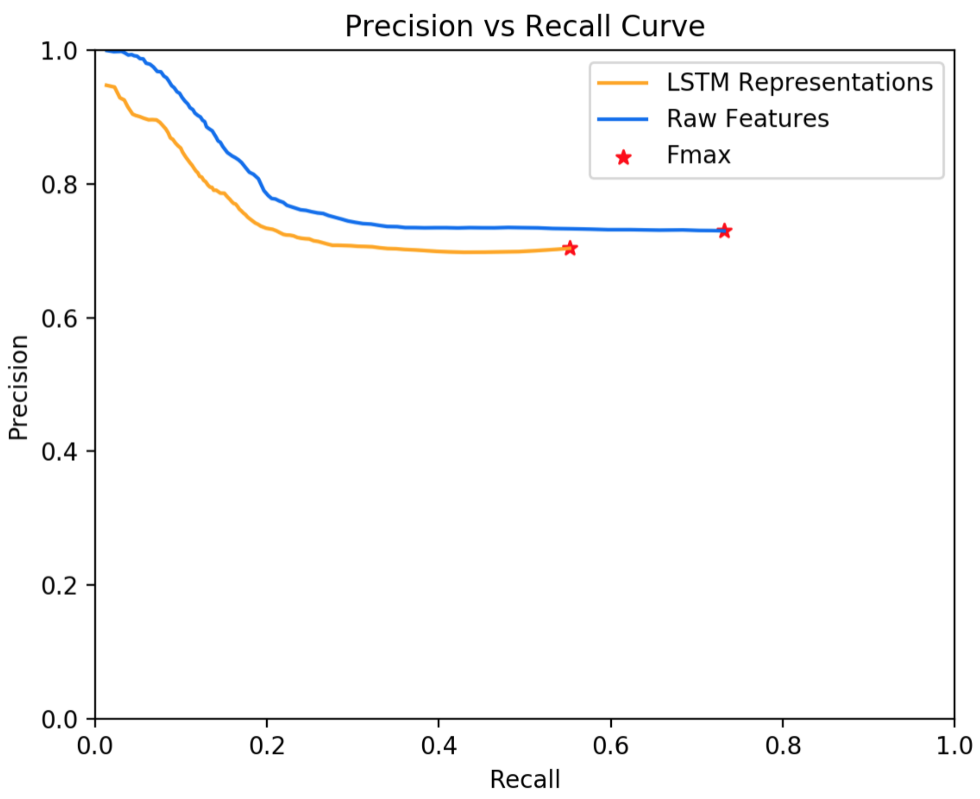

Figure 4.5: The precision-recall curves for varying thresholds for the SVM using the raw features (blue) and the SVM using the LSTM representations (orange). The red stars indicate the $F_{\max}$ score, 0.62 for the LSTM representations and 0.73 for the raw features.

Figure 4.4 compares the two experiments in a scatter plot between the F_1 scores for each GO term considered. It can be inferred by the graph that on average the SVMs trained using the raw psipred feature perform better than the SVMs trained using the LSTM representations. In fact, the average F_1 score was found to be 0.34 for the raw psipred features, compared to 0.28 achieved by the LSTM representations. Like previously, a Wilcoxon signed-rank test was performed, resulting in a p-value of 6×10^{-12} – suggesting a statistical significance between the two methods. Additionally, as can be seen in Figure 4.5, the average precision and recall for different thresholds for each method were computed and it was found that for every value of threshold, the raw features out-performed the LSTM representations, resulting in an $F_{\max}$ score of 0.73 and 0.62 respectively. Although the SVMs trained using the raw psipred features perform better overall, it is interesting to note that 20% of the GO terms are better classified using the LSTM representation.

Chapter 5

Discussion and Conclusion

To conclude, the experiments conducted found that it is possible to capture informative signals about the function of a protein by using a LSTM. Using these signals for the classification of protein function prediction using a LSTM classifier was found to be ineffective for many of the GO terms, resulting in a F_1 score of zero for 30% of the them. This suggests that using a LSTM as a classifier for this problem is prone to overfitting, unable to overcome the imbalanced nature of the problem, and ultimately succumbing to the prior distributions of the labels. Instead, by extracting the functional LSTM representations that give rise to these signals and passing them through a SVM model, an overall improvement in terms of the F_1 score was found. Specifically, the average F_1 score saw a statistically significant improvement of 0.05, resulting in signal to be captured for all GO terms considered. This is unsurprising, as it is widely accepted that SVMs are well-adapted for handing the imbalance class problem by varying the class weights and slack penalty accordingly. Even though this deep-learning-svm hybrid method was found to perform better than the stand-alone LSTM classifier, the performance of using a SVM with only the raw features of the protein sequence was significantly better. Overall, this method achieved a F_1 score 0.06 higher than using the functional feature embeddings, resulting in better classification in 81% of the GO terms. The raw psipred features are known to be extremely informative of the functions of a protein, and it is quite possible that a LSTM is unable to capture this information from just the predictive sequential output of psipred. Moreover, due to computational and time

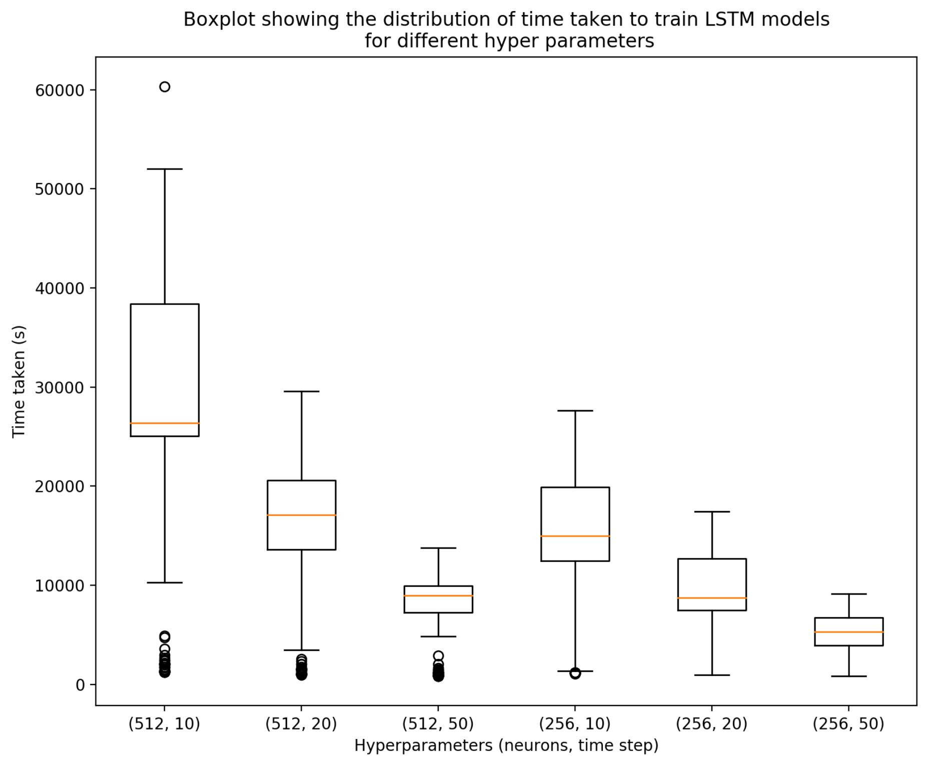

Figure 5.1: Boxplots displaying the distributions of the time taken for the LSTM models to train, given different sets of hyper-parameters (number of neurons and number of time-steps).

constraints, it is possible that the lack of performance could be caused by the small hyperparameter search space – not taking into account larger numbers of neurons or varying other important hyperparameters such as dropout or using different types of optimisers. The boxplots shown in Figure 5.1 shows the distributions of the time taken for LSTMs to train in seconds, given different sets of hyperparameters, demonstrating just how long it takes for the LSTMs to train. It can clearly be seen that as the number of neurons increase or as the number of time-steps decreases, there is a large increase in the amount of time it takes to train these networks, justifying why only a small hyperparameter space was considered during this project. Ultimately, it is possible that the embeddings derived from the LSTM were not used to their full potential, resulting in less informative representations.

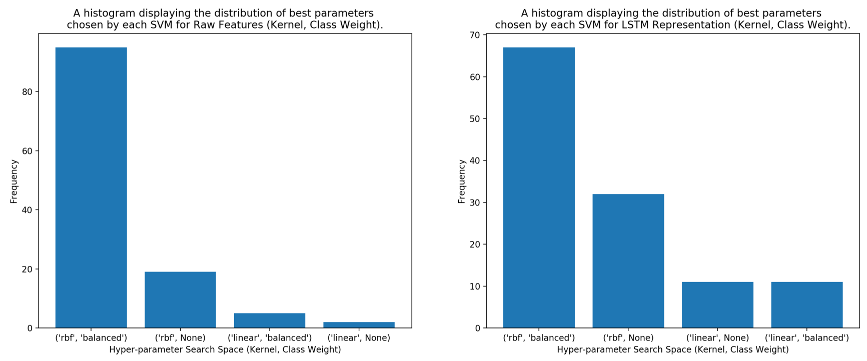

Figure 5.2: A bar chart displaying the proportion of GO terms that performed best using specific types of kernels, as well as whether the class weight was balanced or imbalanced.

Although, overall, the LSTM representations could not out-perform the raw features of the sequences, it is interesting to note and investigate the relative differences between the models and how the feature-space of the proteins are transformed when passed through the LSTM. The first interesting observation is the difference in the number of GO terms that use a 'balanced' class weight for the best performance between the two different models. Figure 5.2 displays this difference and outlines the fact that the LSTM representations use less balanced class weights than the raw features, suggesting that the transformation helps alleviate the imbalanced nature of the labels. Further observation of the hyperparameters, in the form of a heat-map between the slack penalty and kernel coefficient, shows the SVMs trained with the LSTM representations require a lower slack penalty parameter than those trained with the raw features, suggesting that the data becomes more easily separable once it has been transformed by the LSTM. It also suggests that the hyperparameter search conducted for the SVMs were sufficient, as the frequency-distribution is centred in the middle of the search-space.

In the future, it would be intriguing to see how the LSTM representations would perform on rest of the biological process domain, as well as the other two GO sub-ontologies, and whether they would be able to out-perform the raw features or not. Future work should involve the addition of more sequential features, such as

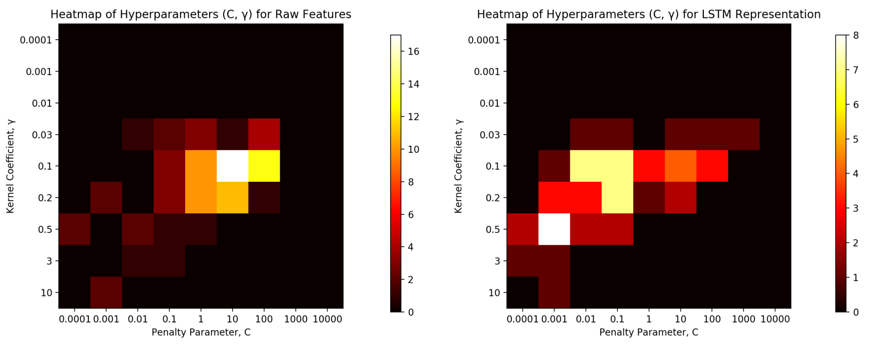

Figure 5.3: Heat-maps displaying the number of GO terms that performed best for a given the kernel coefficient and penalty parameter pair. Left: SVM using the raw features. Right: SVM using the LSTM representations.

the: disorder, amino acid sequence, or glycosylation sites, to see whether more information can be encode within the LSTM representations to build a more powerful classifier. Furthermore, research into using a larger hyperparameter search space for the LSTMs considered in this work to see whether the representations derived from these simple networks are enough to out-perform the SVM trained on the raw features. Finally, research into more complicated recurrent neural networks, such as a convolutional-LSTM would be interesting – hopefully being able to capture the sequence-dependencies between local groups of amino acids through convolutional operations, giving rise to even more powerful functional representations that could be used for the function prediction of proteins.

Appendix A

GitHub - Protein Function Prediction

All the code necessary for the extraction of raw features, the training of the models, and the analysis of the data and results can be found on GitHub at `https://github.com/Usefulmaths/MSc-Project-Protein-Function-Prediction`.

Bibliography

[1] Francis Crick. Central dogma of molecular biology. *Nature*, 227(5258):561, 1970.

[2] Michael Ashburner, Catherine A Ball, Judith A Blake, David Botstein, Heather Butler, J Michael Cherry, Allan P Davis, Kara Dolinski, Selina S Dwight, Janan T Eppig, et al. Gene ontology: tool for the unification of biology. *Nature genetics*, 25(1):25, 2000.

[3] Predrag Radivojac, Wyatt T Clark, Tal Ronnen Oron, Alexandra M Schnoes, Tobias Wittkop, Artem Sokolov, Kiley Graim, Christopher Funk, Karin Verspoor, Asa Ben-Hur, et al. A large-scale evaluation of computational protein function prediction. *Nature methods*, 10(3):221, 2013.

[4] Brigitte Boeckmann, Amos Bairoch, Rolf Apweiler, Marie-Claude Blatter, Anne Estreicher, Elisabeth Gasteiger, Maria J Martin, Karine Michoud, Claire O'donovan, Isabelle Phan, et al. The swiss-prot protein knowledgebase and its supplement trembl in 2003. *Nucleic acids research*, 31(1):365–370, 2003.

[5] Helen M Berman, John Westbrook, Zukang Feng, Gary Gilliland, Talapady N Bhat, Helge Weissig, Ilya N Shindyalov, and Philip E Bourne. The protein data bank, 1999–. In *International Tables for Crystallography Volume F: Crystallography of biological macromolecules*, pages 675–684. Springer, 2006.

[6] Jason A Reuter, Damek V Spacek, and Michael P Snyder. High-throughput sequencing technologies. *Molecular cell*, 58(4):586–597, 2015.

[7] Christophe Dessimoz and Nives Škunca. *The Gene Ontology Handbook*. Springer, 2017.

[8] Robert F. Weaver. *Molecular biology*. McGraw-Hill, 2002.

[9] William R Pearson. [5] rapid and sensitive sequence comparison with fastp and fasta. 1990.

[10] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[11] Leonardo de Oliveira Martins and David Posada. Proving universal common ancestry with similar sequences. *Trends in evolutionary biology*, 4(1), 2012.

[12] John A Gerlt and Patricia C Babbitt. Can sequence determine function? *Genome Biology*, 1(5):reviews0005–1, 2000.

[13] James C Whisstock and Arthur M Lesk. Prediction of protein function from protein sequence and structure. *Quarterly reviews of biophysics*, 36(3):307–340, 2003.

[14] Alex Bateman, Lachlan Coin, Richard Durbin, Robert D Finn, Volker Hollich, Sam Griffiths-Jones, Ajay Khanna, Mhairi Marshall, Simon Moxon, Erik LL Sonnhammer, et al. The pfam protein families database. *Nucleic acids research*, 32(suppl_1):D138–D141, 2004.

[15] Peer Bork and Eugene V Koonin. Protein sequence motifs. *Current opinion in structural biology*, 6(3):366–376, 1996.

[16] Jimmy Y Huang and Douglas L Brutlag. The emotif database. *Nucleic acids research*, 29(1):202–204, 2001.

[17] Sridhar S Hannenhalli and Robert B Russell. Analysis and prediction of functional sub-types from protein sequence alignments1. *Journal of molecular biology*, 303(1):61–76, 2000.

[18] L Juhl Jensen, Ramneek Gupta, Nikolaj Blom, D Devos, J Tamames, Can Kesmir, Henrik Nielsen, Hans Henrik Stærfeldt, Krzysztof Rapacki, Christopher Workman, et al. Prediction of human protein function from post-translational modifications and localization features. *Journal of molecular biology*, 319(5):1257–1265, 2002.

[19] Lars Juhl Jensen, Ramneek Gupta, H-H Staerfeldt, and Søren Brunak. Prediction of human protein function according to gene ontology categories. *Bioinformatics*, 19(5):635–642, 2003.

[20] Anna E Lobley, Timothy Nugent, Christine A Orengo, and David T Jones. Ffpred: an integrated feature-based function prediction server for vertebrate proteomes. *Nucleic acids research*, 36(suppl_2):W297–W302, 2008.

[21] Federico Minneci, Damiano Piovesan, Domenico Cozzetto, and David T Jones. Ffpred 2.0: improved homology-independent prediction of gene ontology terms for eukaryotic protein sequences. *PLoS One*, 8(5):e63754, 2013.

[22] Domenico Cozzetto, Federico Minneci, Hannah Currant, and David T Jones. Ffpred 3: feature-based function prediction for all gene ontology domains. *Scientific reports*, 6:31865, 2016.

[23] Yuxiang Jiang, Tal Ronnen Oron, Wyatt T Clark, Asma R Bankapur, Daniel DAndrea, Rosalba Lepore, Christopher S Funk, Indika Kahanda, Karin M Verspoor, Asa Ben-Hur, et al. An expanded evaluation of protein function prediction methods shows an improvement in accuracy. *Genome biology*, 17(1):184, 2016.

[24] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. Deep face recognition. In *BMVC*, volume 1, page 6, 2015.

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[27] Søren Kaae Sønderby and Ole Winther. Protein secondary structure prediction with long short term memory networks. *arXiv preprint arXiv:1412.7828*, 2014.

[28] Sheng Wang, Jian Peng, Jianzhu Ma, and Jinbo Xu. Protein secondary structure prediction using deep convolutional neural fields. *Scientific reports*, 6:18962, 2016.

[29] Jack Hanson, Yuedong Yang, Kuldip Paliwal, and Yaoqi Zhou. Improving protein disorder prediction by deep bidirectional long short-term memory recurrent neural networks. *Bioinformatics*, 33(5):685–692, 2016.

[30] Jose Juan Almagro Armenteros, Casper Kaae Sønderby, Søren Kaae Sønderby, Henrik Nielsen, and Ole Winther. Deeploc: prediction of protein subcellular localization using deep learning. *Bioinformatics*, 33(21):3387–3395, 2017.

[31] David T Jones and Shaun M Kandathil. High precision in protein contact prediction using fully convolutional neural networks and minimal sequence features. *Bioinformatics*, 1:8, 2018.

[32] Maxat Kulmanov, Mohammed Asif Khan, and Robert Hoehndorf. Deepgo: predicting protein functions from sequence and interactions using a deep ontology-aware classifier. *Bioinformatics*, 34(4):660–668, 2017.

[33] Rui Fa, Domenico Cozzetto, Cen Wan, and David T Jones. Predicting human protein function with multi-task deep neural networks. *PloS one*, 13(6):e0198216, 2018.

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[35] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[36] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] Gerhard X Ritter and Peter Sussner. An introduction to morphological neural networks. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 4, pages 709–717. IEEE, 1996.

[38] Matt W Gardner and SR Dorling. Artificial neural networks (the multilayer perceptron)a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998.

[39] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[40] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[41] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[42] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[44] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh*

Annual Conference of the International Speech Communication Association, 2010.

[45] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[47] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[48] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[49] Steven J Nowlan and Geoffrey E Hinton. Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493, 1992.

[50] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[51] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[52] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[53] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[54] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

[55] S Sathiya Keerthi and Chih-Jen Lin. Asymptotic behaviors of support vector machines with gaussian kernel. *Neural computation*, 15(7):1667–1689, 2003.

[56] Gary King and Langche Zeng. Logistic regression in rare events data. *Political analysis*, 9(2):137–163, 2001.

[57] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[58] Kai Ming Ting. Precision and recall. In *Encyclopedia of machine learning*, pages 781–781. Springer, 2011.

[59] Yutaka Sasaki et al. The truth of the f-measure. *Teach Tutor mater*, 1(5):1–5, 2007.

[60] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[61] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.