Abstract

Because of the importance of determining protein structure and the growing number of known protein sequence, protein structure prediction has been a critical goal in bioinformatics. As a part of protein structure prediction, protein secondary structure prediction plays an important role in ab initio protein structure prediction and protein fold recognition. In this project, we attempted to predict protein secondary structure with neural networks. On the basis of ordinary neural networks, we used a neural network system that contains two-level networks, and we also utilized multiple sequence alignment technique to improve the prediction. Eventually, we achieved around 80% accuracy which is quite close to state-of-the-art technologies like PSIPRED and JPred. Besides, a program has been developed in Python and Theano for making experiments and measures more convenient. Based on the Theano implementation, it will be easy to apply deep learning technology to predict protein secondary structure in the near future.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background & Objectives

In this chapter, we will discuss the background of protein secondary structure prediction and neural networks, the objectives of this project, the challenge the project faces and the research methods I have followed to do the work.

1.1 Background

1.1.1 Protein Structure Prediction

Protein structure prediction is the prediction of the three-dimensional structure of a protein from its amino acid sequence. To be specific, it is the prediction of the folding and the secondary, tertiary and quaternary structure of a protein from its primary structure.

For analysing and understanding the functions of proteins at a molecular level, it is usually necessary to determine their three-dimensional structures. However, experimental methods like X-ray crystallography, NMR spectroscopy and dual polarisation interferometry for determining protein structure are so time-consuming and expensive that they can be applied to only a fraction of the proteins produced by large-scale protein sequencing projects.

In contrast to the slower increase in the number of known protein structures, an exploding number of known protein sequences have been produced every year due to the development of genetic engineering. It has been widely accepted that protein folding is a physical process that the amino acid sequence contains sufficient information to determine the three-dimensional structure of a protein [10]. Consequently, scientists started to predict protein structures by the amino acid sequence decades ago.

Moreover, on a practical level, protein structure prediction is extremely essential in the fields of protein engineering and rational drug design. Due to the influential role of protein structure prediction, it has become one of the most important goals pursued by bioinformatics and theoretical chemistry.

1.1.2 Protein Secondary Structure Prediction

Proteins consist of 20 different specific amino acids and have four levels of structure - primary, secondary, tertiary and quaternary structure, as shown in Figure 1.1. Obviously, protein secondary structure prediction is the prediction of the secondary structures of proteins based on their primary sequences.

Protein Secondary Structure Prediction (PSSP) is a fundamental problem in protein structure prediction. Understanding the formation of regular local structures such as α-helices and β-strands within a single protein sequence is an essential intermediate step on the way to predicting the full three-dimensional structure of a protein. It is possible to derive a number of possible tertiary structures if the secondary structure of a protein is known.

Accurate secondary structure prediction cannot only improve ab initio protein structure prediction, but also improve fold recognition, classification of structural motifs, and refinement of sequence alignments.

A lot of early attempts utilized statistical approaches to predict secondary structures of proteins, but often failed to achieve accuracy higher than 65% [20]. However, machine learning methods, especially neural networks, have proved to be effective for such prediction. Judging by the published results in the protein biochemistry literature, neural networks have produced the most accurate secondary structure predictions for the majority of the past decade, starting with the earliest published work [18]. They could raise accuracy to more than 70%, even though the accuracy has been increasing slowly over 80% at this stage.

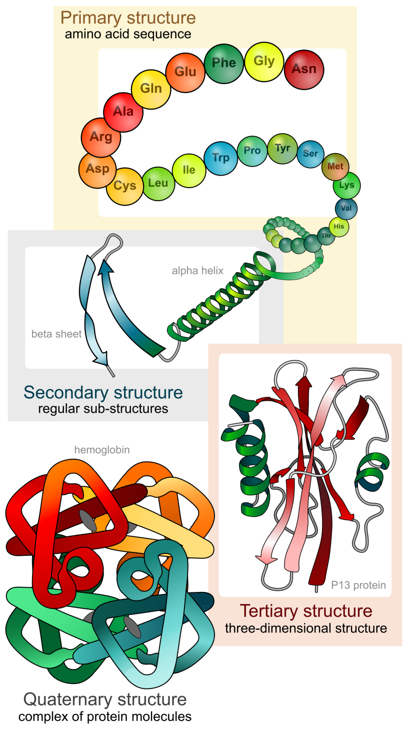


Figure 1.1: Protein Structure
source from: `www.aly-abbara.com/livre_gyn_obs/images/testosterone_DHT.html`

1.1.3 Neural Networks

Neural networks, which are more accurately called *artificial neural networks*, are some famous powerful methods of machine learning. Neural networks are inspired by biological neural networks in the brain of animals. It can express non-linear models and be usually used in some models with complex relations between inputs and outputs. Neural networks have been applied very successfully in many fields like handwritten recognition, speech recognition and computer vision.

Feed-forward neural networks, which are also called multilayer perceptions, are the most well-known type of neural networks. The architecture of a feed-forward neural network usually looks like Figure 1.2.

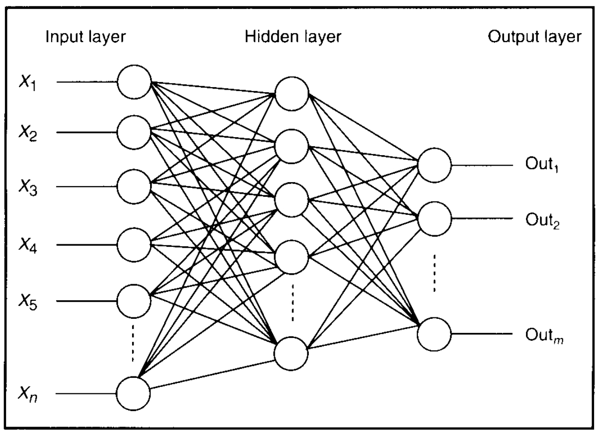


Figure 1.2: A Feed-Forward Neural Network
source from: `http://mechanicalforex.com`

As shown in Figure 1.2, neural networks normally consist of a couple of *neurons*, which can communicate with each other. The neurons denote some operations called *activation functions*, and the connections between neurons denote numeric *weights*. The neurons make up an input layer, an output layer and one or more hidden layers. When computing, data flow goes from the input layer to the hidden layer, and then the output layer, becoming the results we want.

1.2 Objectives

The overall goal of this project is to use neural networks to predict the secondary structures of proteins. There are already some famous methods for PSSP, we will try to understand them and attempt different ways to improve them as much as possible. In order to facilitate the test, we may need a program to do experiments repeatedly. To be specific, we have the following objectives:

1. Understand the typical methodology of protein secondary structure prediction.
2. To implement a program which can predict the secondary structure of proteins.
3. Based on the typical methodology, try to improve the accuracy.
4. Do experiments and compare the results of the program with other state-of-the-art methods.

1.3 Challenges

The challenges or difficulties of this project concentrate on two points.

Firstly, for improving the accuracy, it is necessary to master the knowledge of protein structure and protein structure prediction. Sometimes it is also required to use some professional tools to extract the features of protein sequences.

Secondly, Many experts in this field have been already done lots of attempts. However, the accuracy has been at 80% to 82% over a long time. It might be hard to make a breakthrough especially in a solo project.

1.4 Research Method

For understanding the basic methodology to predict the secondary structures of proteins with neural networks, we started this project with some milestones in history of PSSP, which are some classic methods that got some higher accuracies for the first time in history.

After that, I tried to develop a program with my own method in order to validate how my method is and improve it easily. This called for a high requirement for the quality of my application. Hence, I had to apply some software development methods and tools to make my programs reliable, reusable and easy to experiment repeatedly.

1.4.1 Development Method

Comparing with the heavy waterfall software development, it is better to use some agile methods in this project for some reasons. First, the waterfall development is too ponderous to use in this tiny and solo research project. Then, the requirement is unpredictable in this project because it is necessary to modify the application for various experiments and improving methods continually. Accordingly, iterative agile models could be more suitable for this project.

However, it is still difficult and evil to use some native agile models, like XP or Scrum due to the scale of this project. Instead, I only used some techniques and principles during the development process:

- Iterate faster to respond to requirement changes.
- Write self-documented code to abandon unnecessary documents.
- Refactoring frequently to keep the extensibility of programs.

1.4.2 Development Environment

In order to realize the goal, I have used some tools like Git and Vim to develop and track the version of the code.

Git is a distributed version control system, which can be used locally and efficiently. I edited files in Vim, an editor which can modify code extremely quickly, with some plugins that could check the grammar automatically, browse and open project files fast and cooperate well with Git.

The project has been open-source and hosted on Github (`https://github.com/junshuai/PSSPred`).

Chapter 2

Methods

This chapter will describe the concepts of artificial neural networks, the basic approach for PSSP, neural networks design and multiple sequence alignment, which is an important technique to extract features.

2.1 Biological Neural Networks

2.1.1 Biological Inspiration

What the original important goal of machine learning is to implement artificial intelligence. Just like the name *machine learning*, to teach machines how to learn, is to give machines intelligence. In this respect, like a lot of great human inventions, artificial neural networks become the greatest algorithms in the field of machine learning through simulating organisms, which is the masterpiece of the creator.

The intelligence of organisms originates from the neural system, which consists of biological neural networks. As the basic units of biological neural networks, a neuron is a cell that processes and transmits stimulation in a neural system. As seen in Figure 2.1, it possesses a soma, dendrites, and an axon. They have different functions: a soma can deal with the information it got; dendrites take charge of propagating the stimulation received from other neurons; an axon transmits information to other neural cells. There are around 10^{11} neurons in the brains of human beings. By transportation of stimulation by dendrites and axons, and the disposal of cell bodies, neurons build a system that can help humans to complete physiological activity, recognize things and communicate with others.

From the perspective of computer programming, we can consider dendrites as inputs, an axon as an output, and a soma as a function that can accept some inputs and produce an output. In this way, we could use computers to simulate the structures of neural networks, that is an artificial neural network.

However, there is still an issue that must be addressed. In a hunman brain, vision, auditory and other sensory abilities are in the charge of different sections of the brain. Each section of the brain is considered relatively independent. This means, for building a neural network system, it

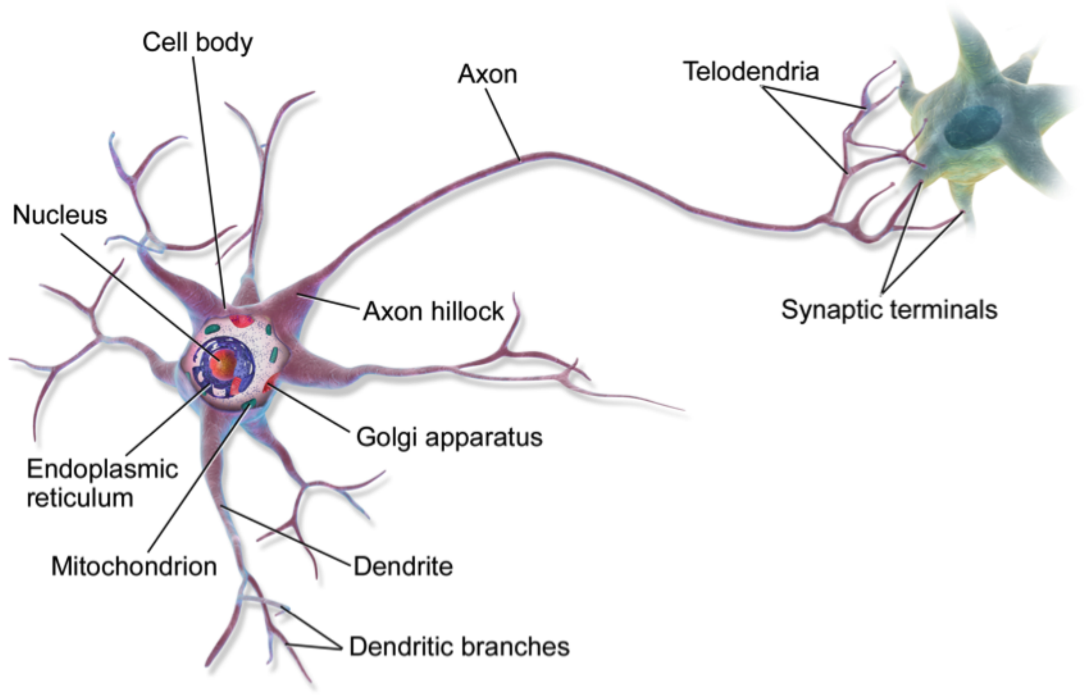


Figure 2.1: Neuron
source from: `http://en.wikipedia.org/wiki/Neuron`

is required to construct various sensors and processing units, also make them communicate with each other. It is nearly impossible to build this complex network system with our current level of technology.

A hypothesis has been raised later, and it is considered to be able to address this problem potentially. It is *one learning algorithm*. It assumes all neurons are similar with each other, and they learn to deal with disparate jobs by only one algorithm. It bases on a kind of experiments named *neural reconnection experiments*. Neuroscientists cut the wire from the ears to the auditory cortex, and re-wire it. The signal from the eyes to the optic nerve eventually gets routed to the auditory cortex. It turns out that the auditory cortex will learn to see. Similar experiments can be also applied in other sense organs. Accordingly, it is more than enough to find the only one learning algorithm for construction of neural networks.

The existing neural networks use a large amount of nodes represented as neurons, the edges between nodes stand for inputs and outputs. In each neuron node, a function has been defined as an *activation function*. Moreover, *weight* has been defined in each edge for simulating how synaptic connections affect the stimulation. When running, every node as inputs can transmit a value. A weighted sum of these values will be the input of the activation function in the cell body. Then, the neuron outputs a new value produced by the activation function to next neuron.

2.1.2 Single-Layer Perceptrons

A perceptron is a simplest artificial neural network, which can be an abstraction of a single neuron easily as shown in Figure 2.2.

A single-layer perceptron with n-dimensional inputs, has an output t:

$$t = f(\sum_{i=1}^{n} w_i x_i + b) = f(\mathbf{w}^T \mathbf{x}),$$

where $\mathbf{w} = [w_1\ w_2\ ...\ w_n\ b]^T$, $\mathbf{x} = [x_1\ x_2\ ...\ x_n\ 1]^T$, and the activation function $f(x)$ can have various forms, but usually an antisymmetric function. The range is often $[-1, 1]$ or $[0, 1]$. When the activation function is a sigmoid function, the single-layer perceptron can be considered as a logistic regression.

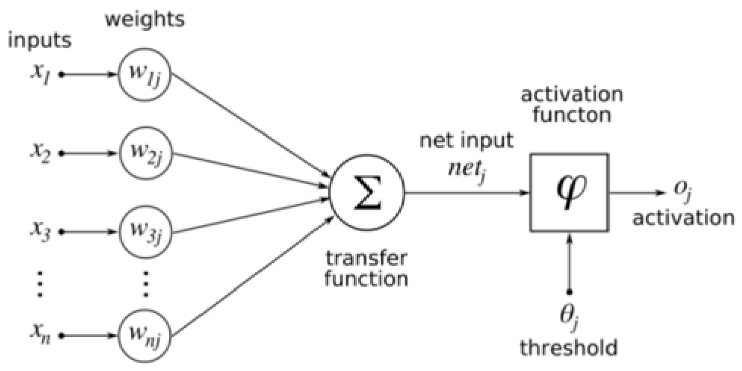


Figure 2.2: Perceptron
source from: `http://nl.wikipedia.org/wiki/Perceptron`

The single-layer perceptron is a linear classifier, which can solve linear problems. Yet, it cannot solve the problems that are not linearly separable.

2.1.3 Multi-Layer Perceptrons

Single-layer perceptron is a kind of feed-forward networks. There is another kind of feed-forward networks named a *multilayer perceptron* (MLP). Actually, it is quite common to regard feed-forward networks as a multilayer perceptron. Regularly, people do not distinguish these two concepts very much.

As seen in Figure 2.3, the multilayer perceptron consists of multiple neuron layers. They can be classified by the input layer, the hidden layer and the output layer. Sometimes, we will have more than one hidden layer. In each layer, you can recognize neuron nodes as perceptrons. We sometimes call each layer fully connected to the next layer, means each node in one layer connects with a certain weight w_{ij} to every node in the following layer.

Multilayer perceptrons can address non-linear problems with their complicated models. They were popular in the 1980s, applied in many fields such as handwritten recognition, speech recognition and computer vision. Even though they faced strong competition from the support vector machines since then, multilayer perceptrons have been renewed attention due to the development of deep learning recently.

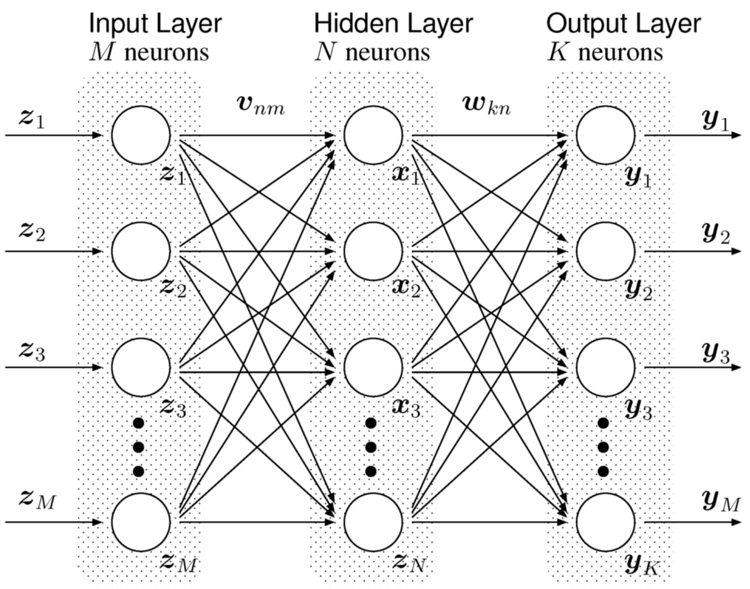


Figure 2.3: Multi-Layer Perceptron
source from: `http://www.mdpi.com/2078-2489/3/4/756`

2.2 Learning Algorithms

After defining the model, it is necessary to train the model before using it. Multilayer perceptrons are usually trained by using a backpropagation algorithm, which is essentially a gradient algorithm.

2.2.1 Loss Function

Before we use the gradient descent, we have to know the loss function. Loss function has been used to define the gap between predicted value and observed value. When we want to make our model fit data sets, the fact is that we don't have any specific objects to optimize, so we should choose a loss function to measure how much we fitted the model and data.

A well-known loss function is MSE, which is the *Mean Squared Error* function. It is of the form:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2.$$

In the problem of multiclass classification, it is very common to use the negative log-likelihood as the loss. This is equivalent to maximizing the likelihood of the data set $\mathcal{D}$ under the model parameterized by θ. Let us define the likelihood $\mathcal{L}$ and loss ℓ:

$$\mathcal{L}(\theta = \{W, b\}, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} \log(P(Y = y^{(i)} | x^{(i)}, W, b));$$
$$\ell(\theta = \{W, b\}, \mathcal{D}) = -\mathcal{L}(\theta = \{W, b\}, \mathcal{D}).$$

2.2.2 Gradient Descent

After defining the loss function, we will use the gradient descent to optimize the parameters. To understand gradient descent, we use a simple model with only two parameters. In this simple model, the relation between the value of the loss function and the parameters will look like a bowl as seen in Figure 2.4.

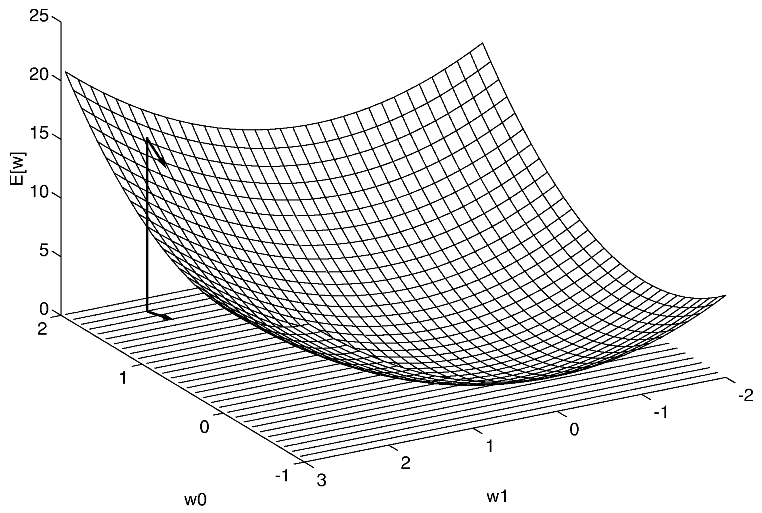


Figure 2.4: Gradient Descent
source from: `http://jmvidal.cse.sc.edu/talks/ann/graddesgrad.html`

After each prediction, we can be at the surface of the relation function in Figure 2.4. When we are searching a local minimum of a function, we take steps toward the direction of the negative of the gradient of the loss function at the current point. Then we will be growing closer to the local minimum.

2.2.3 Stochastic Gradient Descent

Gradient descent is a general strategy for searching a local minimum. However, the procedure to apply gradient descent to converge is quite slow when we are training a large-scale model. It is also not guaranteed that the global minimum can be found. A variation of gradient descent is *Stochastic gradient descent* (SGD), which could speed up the procedure of learning.

In the standard gradient descent algorithm, the gradient for each iteration of weight updates is calculated based on all training examples; whereas in SGD, the gradient could be calculated based on a random subset of training examples. This could improve the efficiency in training particularly in case of large scale problems. It is also often used as a technique for overcoming the local optima problem for standard MLP.

2.3 Early Stopping

Neural networks are so powerful that they can represent very complex models. However, we don't usually know what model of data is and how complex networks we need build. Meanwhile, for addressing problems, we usually use an over powerful network so that we won't underfit the model. Hence, if we just train the networks again and again, it is almost inevitable that we will overfit the model.

Early stopping is a form of regularization for avoiding overfitting in gradient descent. It can also save the time of training. Figuring out when the best timing to stop training is a hot topic in machine learning.

There are a couple of ways to do early stopping. One of them is to use cross-validation. The simplest way is to stop the training as soon as the error on the validation set is higher than it was the last time it was checked. However, the validation set error does not evolve very smoothly in reality. Hence, we have to use some better stopping criteria [17].

To describe the criteria, we define some symbols first. Let E be the loss function, $E_{tr}(t)$ be the average training error over the past epoch, $E_{va}(t)$ is the corresponding error on the validation set. The value $E_{opt}(t)$ is defined to be the lowest validation set error obtained in epochs up to t:

$$E_{opt} = \min_{t' <= t} E_{va}(t').$$

Now we can define the generalization loss at epoch t which denotes the relative increase of the validation error:

$$GL(t) = 100 \cdot (\frac{E_{va}(t)}{E_{opt}(t)} - 1).$$

The first class of stopping criteria is **to stop as soon as the generalization loss exceeds a certain threshold**:

GL_α: stop after first epoch t with $GL(t) > \alpha$.

However, sometimes we hope the generalization loss will be "repaired" when the training error is decreasing rapidly. So we can take the speed of training error descent into account. We define a *training strip of length* k to be a sequence of k epochs. Then the training *progress* is:

$$P_k(t) = 1000 \cdot (\frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \cdot \min_{t'=t-k+1}^{t} E_{tr}(t')} - 1).$$

The second class of stopping criteria is **to stop as soon as the quotient of generalization loss and progress exceeds a certain threshold**:

$$PQ_\alpha\text{: stop after first end-of-strip epoch } t \text{ with } \frac{GL(t)}{P_k(t)} > \alpha.$$

Besides, we can stop the training when the generalization loss has been increasing for too long. The third class of stopping criteria is **to stop as soon as the generalization error increase in s successive strips**:

$$UP_s\text{: stop after epoch } t \text{ iff } UP_{s-1} \text{ stops after epoch } t-k \text{ and } E_{va}(t) > E_{va}(t-k);$$
$$UP_1\text{: stop after first end-of-strip epoch } t \text{ with } E_{va}(t) > E_{va}(t-k).$$

2.4 Network Design

In this section, we talk about the approach to designing a neural network system to predict the secondary structures of proteins, including the data, input/output encoding and the architecture of neural networks and neural network system.

2.4.1 Data: Sequences and Structures

The following is an example of an amino acid sequence of a protein and its corresponding secondary structure that we want to predict:

```
Amino Acid Sequences: LLRCRLPGGVITTKQWQAIDKFAGENTIYGSIRLTN
Secondary Structures: CCCCCCHHHCCCHHHHHHHHHHHHHCCCCCCCCCC
```

Let $s = (a_1, a_2, ..., a_n)$ be an amino acid sequence, where

$$a_i \in \mathbf{A} = \{A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$$

is a one-letter code denoting one of the 20 proteingenic amino acids.

Conventionally, there are three target classes - α-helix, β-strand and coil - defined by collapsing the original 8 structures defined in Dictionary of Protein Secondary Structure (DSSP) according to the following rules [3]:

DSSP classes:	Prediction class:
H, G	(H) helix
E	(E) strand
B, I, S, T, e, g, h	(C) coil

There is also another reason for using three prediction classes: generally, α-helices and β-strands are basically secondary structures that can determine functions of proteins. For most problems of proteins, it is enough to know which kind of secondary structures, α-helices, β-strands or coil. That can also simplify the complexity of protein secondary structure prediction.

Proteins fold into specific spatial conformations driven by a number of non-covalent interactions, such as hydrogen bonding, ionic interactions, Van der Waals forces, and hydrophobic packing. Therefore, it is a matter of course not to consider only a single residue on the predicted position, but also surrounding residues, which can be also considered as a window onto a short segment of protein chain centred on the residue to be predicted. A *window size* has been defined as the length of the window [3].

Hence, in this neural network, each input output pair whose window size is 19 is of the form:

```
Input:    **LLRCRLPGGVITTKQWQ
Output:            H
```

The extra symbol "$\star$" is used for windows that overlap the terminal of a chain.

2.4.2 Input/Output Encoding

In neural networks, both the residues and target classes are encoded in unary format [3]:

```
            A R N D C E Q G H I L K M F P S T W Y V
Ser:        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
Helix:      1 0 0
            H E C
```

Thus each pattern presented to the network comprises $n \times 20$ inputs for a window of size n. And a gap is represented by 20 zeros when the window overlaps the end of the chain.

2.4.3 Neural Network Architectures

Based on above approaches, we have a neural network with $n \times 20$ input nodes and 3 output nodes. The number of neurons in hidden layers is highly essential, and we will select it through experiments in Chapter 4. As for activation functions, we employ the sigmoid function in each neuron of hidden layers and output layers. A sigmoid function is a common-used activation function in classification problems. The sigmoid curve can be seen in Figure 2.5.

2.4.4 Second-Level Networks

The first-level networks are a sequence-to-structure networks which are trained to classify a central residue according to the surrounding residues. It does not take into account the consecutive

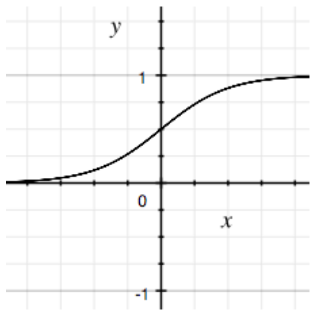


Figure 2.5: Sigmoid Function

patterns (i.e. the ambient secondary structures). We can build a second-level structure-to-structure network to express the correlation of consecutive patterns, like for a helix consisting of at least 3 consecutive patterns [19]. Experimentally, it can bring 1-2% improvement of accuracy. We will validate it in Chapter 4.

The architecture of second-level networks is similar with first-level networks. It has $n \times 3$ input nodes, a couple of hidden nodes and 3 output nodes, with sigmoid functions. Now we have built a prediction system with two-level neural networks.

2.4.5 Overview

In fact, this kind of two-level networks algorithm was originated by PHD Secondary Structure Prediction Method [19], as seen in Figure 2.6. This figure also does a good job of demonstrating our prediction system. In some state-of-the-art technologies such as PSIPRED and JPred, they both used a second-level structure-to-structure network to improve the prediction. However, even the similar approach we all used, it still has many possibilities due to various parameters and the network architecture, like different window sizes and various amounts of neurons in the hidden layer. This is also an important research goal of this project.

2.5 Multiple Sequence Alignment

Another important feature for PHD neural network system is the use of multiple sequence alignment results for protein secondary structure prediction. It has proved to be a quite efficient way to improve the prediction [8], and it usually bring 5-8% improvement on the prediction.

Multiple Sequence Alignment (MSA) is a sequence alignment of three or more biological sequences, generally protein, DNA or RNA. By using MSA, we can search a protein in a protein database to find proteins that have similar structures. Normally, homologous proteins could be found, counting the frequency of occurrence of each amino acid on every position for extracting features.

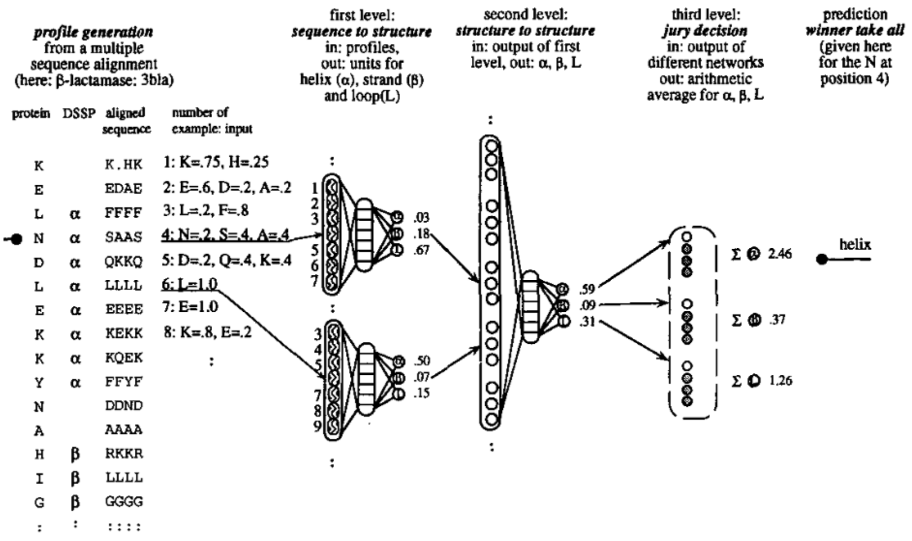
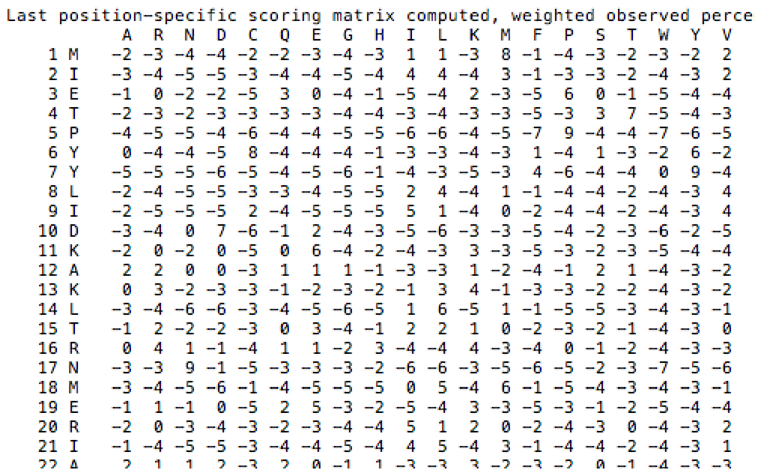

profile generation
from a multiple sequence alignment (here: β-lactamase: 3bla)
protein
DSSP
aligned sequence
number of example: input
1: K=.75, H=.25
2: E=.6, D=.2, A=.2
3: L=.2, F=.8
4: N=.2, S=.4, A=.4
5: D=.2, Q=.4, K=.4
6: L=1.0
7: E=1.0
8: K=.8, E=.2
first level: sequence to structure
in: profiles, out: units for helix (α), strand (β) and loop(L)
second level: structure to structure
in: output of first level, out: α, β, L
third level: jury decision
in: output of different networks
out: arithmetic average for α, β, L
prediction winner take all
(given here for the N at position 4)
Σ 2.46
Σ .37
Σ 1.26
helix

Figure 2.6: PHD Secondary Structure Prediction Method
source from: *Prediction of Protein Secondary Structure at Better than 70% Accuracy* [19]

2.5.1 PSI-BLAST

PSI-BLAST for Position-Specific Iterative BLAST (Basic Local Alignment Search Tool) is an algorithm for finding out similar protein sequences to a query sequence. It is a common tool for protein MSA in bioinformatics. It can construct a PSSM from the resulting alignment.

PSSM, or Position-Specific Scoring Matrix, is a type of scoring matrix used in protein BLAST searches, as seen in Figure 2.7. In a PSSM, the rows represent alignment positions and the columns represent amino acids. Its scores are generally shown as positive or negative integers. Positive scores indicate that the given amino acid substitution occurs more frequently in the alignment than expected by chance, while negative scores indicate that the substitution occurs less frequently than expected.



```
Last position-specific scoring matrix computed, weighted observed perce
           A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V
    1 M   -2 -3 -4 -4 -2 -2 -3 -4 -3  1  1 -3  8 -1 -4 -3 -2 -3 -2  2
    2 I   -3 -4 -5 -5 -3 -4 -4 -5 -4  4  4 -4  3 -1 -3 -3 -2 -4 -3  2
    3 E   -1  0 -2 -2 -5  3  0 -4 -1 -5 -4  2 -3 -5  6  0 -1 -5 -4 -4
    4 T   -2 -3 -2 -3 -3 -3 -3 -4 -4 -3 -4 -3 -3 -5 -3  3  7 -5 -4 -3
    5 P   -4 -5 -5 -4 -6 -4 -4 -5 -5 -6 -6 -4 -5 -7  9 -4 -4 -7 -6 -5
    6 Y    0 -4 -4 -5  8 -4 -4 -4 -1 -3 -3 -4 -3  1 -4  1 -3 -2  6 -2
    7 Y   -5 -5 -5 -6 -5 -4 -5 -6 -1 -4 -3 -5 -3  4 -6 -4 -4  0  9 -4
    8 L   -2 -4 -5 -5 -3 -3 -4 -5 -5  2  4 -4  1 -1 -4 -4 -2 -4 -3  4
    9 I   -2 -5 -5 -5  2 -4 -5 -5 -5  5  1 -4  0 -2 -4 -4 -2 -4 -3  4
   10 D   -3 -4  0  7 -6 -1  2 -4 -3 -5 -6 -3 -3 -5 -4 -2 -3 -6 -2 -5
   11 K   -2  0 -2  0 -5  0  6 -4 -2 -4 -3  3 -3 -5 -3 -2 -3 -5 -4 -4
   12 A    2  2  0  0 -3  1  1  1 -1 -3 -3  1 -2 -4 -1  2  1 -4 -3 -2
   13 K    0  3 -2 -3 -3 -1 -2 -3 -2 -1  3  4 -1 -3 -3 -2 -2 -4 -3 -2
   14 L   -3 -4 -6 -6 -3 -4 -5 -6 -5  1  6 -5  1 -1 -5 -5 -3 -4 -3 -1
   15 T   -1  2 -2 -2 -3  0  3 -4 -1  2  2  1  0 -2 -3 -2 -1 -4 -3  0
   16 R    0  4  1 -1 -4  1  1 -2  3 -4 -4  4 -3 -4  0 -1 -2 -4 -3 -3
   17 N   -3 -3  9 -1 -5 -3 -3 -3 -2 -6 -6 -3 -5 -6 -5 -2 -3 -7 -5 -6
   18 M   -3 -4 -5 -6 -1 -4 -5 -5 -5  0  5 -4  6 -1 -5 -4 -3 -4 -3 -1
   19 E   -1  1 -1  0 -5  2  5 -3 -2 -5 -4  3 -3 -5 -3 -1 -2 -5 -4 -4
   20 R   -2  0 -3 -4 -3 -2 -3 -4 -4  5  1  2  0 -2 -4 -3  0 -4 -3  2
   21 I   -1 -4 -5 -5 -3 -4 -4 -5 -4  4  5 -4  3 -1 -4 -4 -2 -4 -3  1
   22 A    2  1  1  2 -3  2  0 -1  1 -3 -3  3 -2 -3 -2  0 -1 -4 -3 -3
```

Figure 2.7: Position-Specific Scoring Matrix

2.5.2 Scaling Function

Another advantage to use PSSMs, is that we needn't to adjust anything of our neural network model. The format of PSSM is very suitable for our input encoding. But because the range of numbers in PSSMs is bigger than the range $[0, 1]$, we have to scale down the numbers by the scaling function. A simple piecewise function with a linear distribution for the intermediate values was suggested in SVMpsi [13]:

$$f(x) = \begin{cases} 0.0, & x < -5, \\ 0.5 + 0.1x, & -5 \leq x < 5, \\ 1.0, & x \geq 5. \end{cases}$$

And also a logistic function of the PSSM score can be used as in PSIPRED [12]:

$$f(t) = \frac{1}{1 + e^{-x}}.$$

2.6 Performance Measures

Measuring performance is to assess the quality of a particular prediction. It is not only for comparing this method with others, but also a support of how we improve predictions.

There are a couple of measures in protein secondary structure prediction. The most commonly used one is simply the percentage of correctly predicted residues, known as Q3. Another more complicated but effective measure of accuracy is the correlation coefficient [15].

Whichever measures we choose, it can be derived from a 3×3 (for three secondary structure types) confusion matrix A [19], with:

$$A_{ij} = \text{number of residues predicted to be in structure type } j \text{ and observed to be in type } i.$$

A example of a confusion matrix A can be seen as Table 2.1.

	α_{pred}	β_{pred}	L_{pred}
α_{obs}	7344	179	1085
β_{obs}	598	3205	1674
L_{obs}	1493	628	6889

Table 2.1: A Confusion Matrix

2.6.1 Percentage of Correctly Predicted Residues

With the accuracy table A, we can define the sums over the rows of A:

$$r_i = \sum_{j=1}^{3} A_{ij}, \quad \text{for } i = \alpha, \beta, L,$$

i.e. the number of residues observed to be in structure type i. And also the sums over the columns of A:

$$c_i = \sum_{j=1}^{3} A_{ji}, \quad \text{for } i = \alpha, \beta, L,$$

i.e. the number of residues predicted to be in structure type i. The sum over all elements of A is the number of residues in the data set used, abbreviated by m:

$$m = \sum_{i=1}^{3} r_i = \sum_{i=1}^{3} c_i.$$

We use the abbreviations:

$$Q_i = Q_i^{\%obs} = \frac{A_{ii}}{r_i} \times 100, \quad \text{for } i = \alpha, \beta, L,$$

which describe the percentage of residues correctly predicted to be in structure type i relative to those observed to be in type i. The percentages of residues correctly predicted to be in class i from all residues predicted to be in i are given by:

$$Q_i^{\%pred} = \frac{A_{ii}}{c_i} \times 100, \quad \text{for } i = \alpha, \beta, L.$$

Most people use the overall 3-state accuracy:

$$Q_3 = \frac{\sum_{i=1}^{3} A_{ii}}{m} \times 100,$$

or, in other words, the percentage of all correctly predicted residues.

2.6.2 Correlation Coefficient

The main problem with Q_3 as a measure is that it fails to penalise the network for over-predictions (e.g. non-helix residues predicted to be helix) or under-predictions (e.g. helix residues predicted to be non-helix). Hence, it is possible to achieve a Q_3 of around 50% merely by predicting everything to be coils. For addressing this problem, a useful measure of prediction accuracy is given by the correlation coefficient introduced by Matthews [15]:

$$C_i = \frac{p_i n_i - u_i o_i}{\sqrt{(p_i + u_i)(p_i + o_i)(n_i + u_i)(n_i + o_i)}}, \quad \text{for } i = \alpha, \beta, L,$$

with p_i being the number of patterns correctly assigned to structure type i; n_i the number of patterns correctly not assigned to structure type i; u_i the number of under-predictions, and o_i that of over-predictions.

In terms of the accuracy table A:

$$p_i = A_{ii}, \quad n_i = \sum_{j \neq i}^{3} \sum_{k \neq i}^{3} A_{jk}, \quad o_i = \sum_{j \neq i}^{3} A_{ji}, \quad u_i = \sum_{j \neq i}^{3} A_{ij}, \quad \text{for } i = \alpha, \beta, L.$$

The correlation coefficients for helix (C_α), strand (C_β) and coil (C_L) are in the range $+1$ (totally correlated) to -1 (totally anti-correlated). C_α, C_β and C_L can be combined in a single figure (C_3) by calculating the geometric mean.

2.6.3 Other Measures

There are also other measures which can measure the accuracy of predictions, like RI(Reliability Index) [19] and SOV(Segment Overlap) [21]. However, the Q_3 accuracy and the correlation coefficient have been used since the earliest published work appeared [18]. They are traditionally used to compare different methods in protein secondary structure prediction with neural networks. Thus, in this project, we only use the Q_3 accuracy and the correlation coefficient to measure performance of predictions.

Chapter 3

Implementation and Hyperparameter Settings

In this chapter, we will describe the data sets, how to get PSSM by PSI-BLAST tools, how to implement the application PSSPred and how to choose the hyperparameters by the grid search.

3.1 Data Sets

3.1.1 Data Source

The data set of this project came from *bamboo*, which is a Bayesian algorithm to predict protein secondary structures [14]. According to the description in the publication, the secondary structure data was derived from the ASTRAL SCOP 1.75 structure set [7] filtered at 95% sequence identity. This structure set consists of 15,470 individual protein sequences from the PDB (Protein Data Bank) whose length range from 22 to 1,419 amino acids and total 2,751,815 amino acids.

In some past methods, people used to use a protein dataset named CB513 and RS126 whose digits in the names stand for the number of proteins. Yet they are too small to gain high accuracy. Moreover, they were outdated since they were built almost 20 years ago. On the other hand, there is growing support for the claim that improving the prediction of PSSP requires larger datasets [6]. Hence, large data sets are selected in this project, and this also benefits the application of deep neural networks, which are possibly used in the near future.

3.1.2 Data Splitting

Since we want to use data sets not only for training, but also for the cross-validation, we have to split datasets into two parts: the training set and the validation set. In the project, 70% of datasets are used for training and 30% of datasets are used for validation. This is a quite common splitting scheme in machine learning, and we need not to consider the alternative too much on account of the large number of proteins in our datasets.

Moreover, we need to split the data set in half, because we need one half for first-level networks and one half for second-level networks.

The reason not to use the same training set between the first-level network and the second-level network is: if we use the same training set when training the second-level network, the results from the first-level network, which are used to train the second-level network, usually perform better than normal conditions. That will lead to the underfitting of the model.

3.2 BLAST+

In order to apply MSA to get features as inputs, it is necessary to use PSI-BLAST to calculate PSSMs. The famous tool of PSI-BLAST is from NCBI (The National Centre for Biotechnology Information of the United States) [4]. Because the online version of NCBI BLAST is limited to submit multiple queries at once and the number of proteins in the dataset we used is over than ten thousands, we have to download the standalone BLAST+ and the protein database, and do queries by ourselves.

This work was done by running PSI-BLAST algorithm for three iterations on the RefSeq Protein database. RefSeq is a comprehensive, integrated, non-redundant, well-annotated set of reference sequences including genomic, transcript, and protein from NCBI. That is the reason to choose RefSeq Protein Database. The data of RefSeq Protein database used is on 31, March 2015. There are about 50 million proteins in this database.

MSA has been applied in both training set and testing set. After using MSA, there are 15,179 proteins left in data sets in total, cause other 22 proteins failed to obtain results in PSI-BLAST. It didnt make a big effect since the number of absent proteins is too small to make an influence on training results. And the integrity of ASTRAL30 and CASP9, which are data sets for testing, is guaranteed.

3.3 PSSPred

PSSPred is a program written in Python by me for doing protein secondary structure prediction with neural networks easily. It is lightweight but powerful and efficient.

PSSPred has the following features:

1. Easy to experiment with different data sets and parameters;
2. Different measurements for evaluating the accuracy of predictions;
3. Early stopping mechanism.

The reason to develop a program rather than use third-party machine learning libraries for predicting is: there are also a couple of limits in prediction if we use a third-party library. It will be tough or impossible to modify some parameters like the learning rate, the number of epochs and even construct the suitable model we want. We have so many things to customize that it is better to develop a new program.

3.3.1 Python

Python is an excellent programming language that is able to let developer work quickly. Code in Python has a better readability than most other languages. The more important thing is, due to the integration feature of Python, which means that you can have a C implementation of the critical code, the program can still keep effective even if it is interpreted.

Python has so expansive library of open source data analysis tools that it can easily extract data and do computations in science research. What is more, there is an awesome library for deep learning named Theano that can do this job well.

3.3.2 Theano

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently [1] [2]. Theano usually refers to deep learning. Most people use it to do research related to deep learning. It has been powering large-scale computationally intensive scientific investigations since 2007. It is suitable for implementing neural networks and makes the employment of deep learning in this project possible.

3.3.3 Design

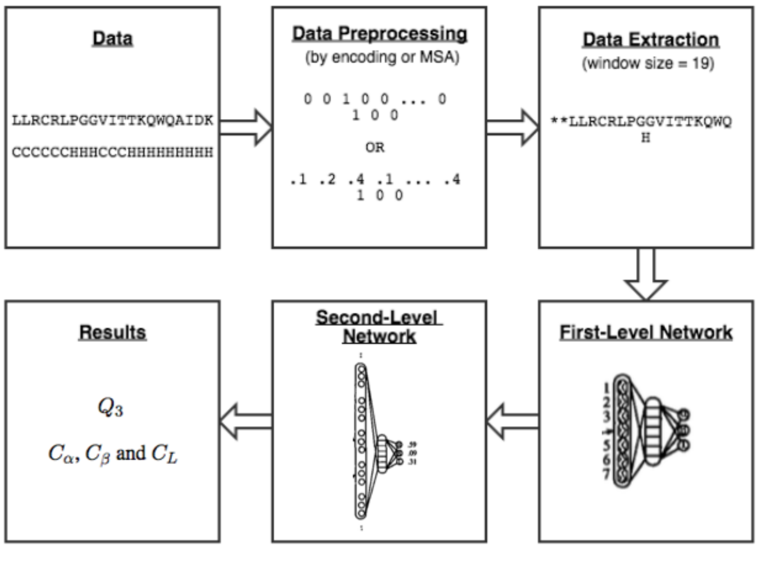


Figure 3.1: Flowchart

The procedure to predict the secondary structures has been shown in Figure 3.1. First, we preprocess the data by encoding it or running PSI-BLAST to get PSSMs. Next, we extract the features according to the window size. In Figure 3.1, the window size is 19, and the features will be an array whose length is $19 \times 20 = 380$. Then, we use the two-level networks to predict the secondary structures. Last, we can require any forms of the results.

When we train a model, what we should do is to predict first, and adjust the parameters according to the results of predictions. We will train the first-level network and the second-level network respectively. Because the inputs of second-level networks should be the prediction (i.e. outputs) of trained first-level networks.

In this program, we have three classes and one module basically: AccuracyTable class, StoppingCriteria class, MultilayerPerceptron class and dataloader module. The *AccuracyTable* class is for saving the confusion matrix, calculating and providing the results according to the measures. The *StoppingCriteria* class is used to decide when to stop the training. The *MultilayerPerceptron* class is responsible for training and predicting of our neural networks. The dataloader module is used to load data into the program, including applying MSA and scaling data. These classes and module are designed to be reusable and extensible so that developers can use it to do something else, not only protein secondary structure prediction.

3.4 Hyperparameter Settings

Before training the model, there is a problem of choosing a set of hyperparameters for a learning algorithm, such as a learning rate, the size of batch and the number of epochs. It is often to use a cross-validation dataset to estimate the generalization performance. Hyperprameters were selected based on the performance on the held-out validation set using grid search.

3.4.1 Grid Search

Grid search is a traditional way to optimize hyperparameters of a learning algorithm. It is easy to understand. For example, we have a hyperparameter constant C to optimize. Usually, the hyperparameters are continuous, but for performing grid search, we just use a finite set of some reasonable values. For example,

$$C \in \{100, 10, 1, 0.1, 0.01\}$$

Then we train the model with different values for parameter C and choose the best one who perform best in a cross-validation dataset.

Whilst we have more than one hyperparameters, like parameters C and γ, we could list their finite sets of optional values:

$$C \in \{100, 10, 1, 0.1, 0.01\}$$
$$\gamma \in \{1, 0.3, 0.1, 0.03, 0.01\}$$

We can train the model with different combinations of these two parameters (in this case, there will be $5 * 5 = 25$ combinations to be experimented), and find out the best combination.

3.5 Experiments

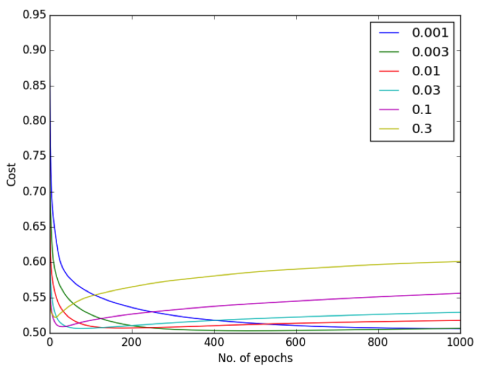
In this project, there are three important hyperparameters in the learning algorithm, which are the batch size for SGD, the number of epochs and the learning rate. However, aim to reduce the complexity, we could just set the two parameters of them without rigorous experiments. That is, to set the batch size to 20, and the number of epochs to 3000.

We can set the size of batch to 20, because in this project, we consider a protein which contains a number of input output pairs as a unit in a batch. So we do not worry about the batch size is too small to make the procedure of convergence unstable and inaccurate. By contrast, a simple experiment which uses 100 as the batch size has been done and the result is the procedure of convergence perform little better but slower much.

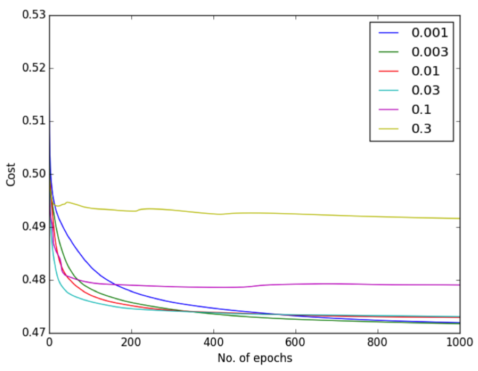
As for the number of epochs, it does not make much sense because we use the early stopping technique to shorten the time of training. The number of epochs in the program just decides on the maximal number of epochs to run. The only thing we should do is to set a large number as a limit (such as 3000 in this case) so that the program will not run for too long to go beyond our patience.

Hence, the most influential hyperparameter left for tuning is the learning rate. Grid search was done on the learning rate using the following set of values:

$$\alpha \in \{0.3, 0.1, 0.03, 0.01, 0.003, 0.001\}$$



(a) First-level Network



(b) Second-level Network

Figure 3.2: Comparison of learning curves for different learning rates. The y-axis indicates the value of the negative log-likelihood cost function during training.

Figure 3.2(a) shows the results of the grid search results for various learning rate for the first-level network.

By setting the optimized learning rate of the first-level network, we can test the learning rate for the second-level network, as shown in Figure 3.2(b)

Ultimately, 0.003 has been chosen to be the value of the learning rate by taking into account the good performance and the possibility that we may have a deeper architecture of the neural network.

Chapter 4

Results and Conclusions

This chapter will describe the results of experiments for various architectures, the impact of multiple sequence alignment, and compare this method with other PSSP methods. Expect the section that explores the impact of MSA, the PSSMs obtained from PSI-BLAST have been used as input features, and the learning rate of SGD in both two-layer networks has been set to 0.003 according to the results of experiments in Chapter 3.

4.1 Network Architecture

The architecture of our neural networks depends on the window size and the hidden layer size (i.e. the number of neurons in the hidden layer). In this section, we use the grid search to choose the most suitable window size and the hidden layer size for both neural networks.

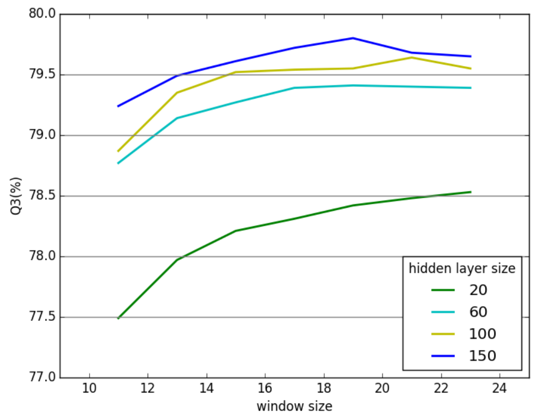


Figure 4.1: Comparison of performance on the validation set for various window sizes and hidden layer sizes in the first-level networks.

The Figure 4.1 shows the results of experiments. The window sizes of odd numbers from 11 to 23 were tested to determine the optimal window size for the first-level networks. The Q_3 accuracy increased toward a window size of 19, and dropped off for windows larger than 21. Eventually, a window size of 19 was selected.

As for the hidden layer size, the overall Q_3 accuracy increased as the sizes of hidden layers increased. It is supposed to continue to test the larger sizes of hidden layers. However, unfortunately, the memory of experimental cluster servers was out so that they cannot afford a larger experiment. A hidden layer size of 150 was selected eventually.

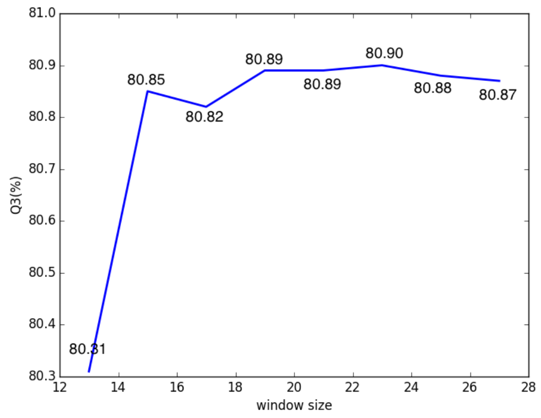


Figure 4.2: Comparison of performance on the validation set for various window sizes in the second-level networks.

For second-level structure , we set the same hidden layer size as the input layer size which equals $windowsize \times 3$. This practice can decrease the complexity of experiments. In Multilayer Perceptron, we regularly do not make the hidden layer size larger than the input layer size. In this case, the input layer size is small and very close to each other, so it does not make sense to adjust the hidden layer size much. As shown in Figure 4.2, it did not have much effect, even though we changed the window size between 19 to 27.

4.2 Impact of Second-Level Neural Networks

Although we can predict the secondary structures directly without the second-level neural network, we still choose to use a second-level neural network due to the improvements of the accuracy that it brings. We can see how much the improvements are through experiments, as shown in Table 4.1.

Measures	one-level network	two-level network
$Q_3(\%)$	79.70	80.90
C_α	0.74	0.76
C_β	0.68	0.71
C_L	0.62	0.64

Table 4.1: Comparison of the model performance on the validation set with and without second-level networks.

4.3 Impact of Multiple Sequence Alignment

The MSA is an important way to improve the prediction, we can see the improvement it brings though experiments, as shown in Table 4.2.

Measures	without MSA	with MSA
$Q_3(\%)$	69.44	80.90
C_α	0.57	0.76
C_β	0.49	0.71
C_L	0.49	0.64

Table 4.2: Comparison of the model performance on the validation set with and without the use of a multiple seuqnece alignment.

4.4 Model Evaluation Using Test Data

4.4.1 Model Selection

Through the above experiments, we selected a two-level network system with a 380-150-3 (i.e. the window size is 19, the hidden layer size is 150) first-level network and a 69-69-3 (i.e. the window size is 23) second-level network. The input features were extracted from the PSSMs obtained by MSA. We will evaluate the model with other state-of-the-art methods of PSSP.

4.4.2 State-of-the-Art Methods

PSIPRED and JPRED are some of the most known methods based on neural networks for protein secondary structure prediction.

PSIPRED is a simple and accurate secondary structure prediction method, incorporating two feed-forward neural networks which perform an analysis on the output obtained from PSI-BLAST (Position Specific Iterated - BLAST). Using a very stringent cross validation method to evaluate the method's performance, PSIPRED 3.2 achieves an average Q3 score of 81.6% [5].

JPred is a Protein Secondary Structure Prediction server and has been in operation since approximately 1998. JPred incorporates the Jnet algorithm in order to make more accurate predictions. In addition to protein secondary structure JPred also makes predictions on Solvent Accessibility and Coiled-coil regions (Lupas method). The current version of JPred (v4) claims that it has already got the secondary structure prediction accuracy of 82% [9].

We compared our method with PSIPRED and MSA-MP using the same test datasets. *MSA-MP* is a probabilistic method. To be specific, it is a Bayesian model based on the knob-socket model of protein packing in secondary structure [14].

4.4.3 Test Datasets

We have two test sets. They are both from bamboo [14], and the related description as following:

The first test set is the current release of SCOPe 2.03 data set [11] filtered at 30% sequence identity (ASTRAL30). In this ASTRAL30 set, we included the domains that are not included in 1.75 version and only included in 2.03 version. The transmembrane proteins were also exclude and this gave 2,794 domains. The data set integrity was further tested by breaking down into the actual segments. When the structure has missing residues, the chain was split into separate sequences and omitted in this study if a chain is shorter than 25 residues. This produced 3,344 chains with 523,332 amino acids. The second test set was created from the targets used in CASP9 experiments in 2010 [16]. The CASP9 set includes 147 structures, and the same cleanup procedure produced 203 chains with 23,298 amino acids.

It is worth mentioning that the Critical Assessment of Structure Prediction (CASP) is a community-wide experiment, which designs to benchmark the state-of-the-art of protein structure prediction in every two years since 1994. It almost becomes benchmark tests in Protein Structure Prediction.

4.4.4 Results

For evaluating the performance of this method, the results of experiments has been compared with two other secondary structure prediction methods: MSA-MP and PSIPRED. PSIPRED was tested by using two different sequence databases: the Protein Data Band (PDB) for PSIPRED-PDB and a non-redundant protein sequence database for PSIPRED-NR which produces the best results, coinciding with the benchmark for secondary structure prediction accuracy [14].

Dataset	MSA-MP	PSIPRED-PDB	PSIPRED-NR	Our Method
ASTRAL30	88	77	80	81
CASP9	74	75	81	80

Table 4.3: Overall Q3 accuracy (%) of MSA-MP, PSIPRED, and our method on ASTRAL30 and CASP9 test datasets

partial data source from: *Bayesian model of protein primary sequence for secondary structure prediction* [14]

As shown in Table 4.3, the accuracy of our method is close to the accuracy of PSIPRED-NR. For understanding what the difference is between the results of these two methods, we also had tests of the recall for each class of secondary structure. A *Recall* of a structure class can be understood as how many of this class has been predicted correctly.

As seen in Table 4.4 and Table 4.5, compared with PSIPRED-PDB, our method tended to recognize the structure coil as other structure.

4.5 Conclusions

Compared with other PSSP methods, the new neural network model was developed using a much larger training dataset with more hidden neurons. The performance on the independent test sets for our proposed model is comparable to the state-of-the-art methods. It still has potential to improve

	MSA-MP			PSIPRED-PDB			PSIPRED-NR			Our Method		
	H	E	C	H	E	C	H	E	C	H	E	C
H	90	1	6	77	5	9	79	3	6	88	3	12
E	1	85	5	3	66	8	1	68	6	2	79	12
C	9	14	89	20	29	83	20	29	88	10	18	76
Overall	88			77			80			81		

Table 4.4: Classification Q3 recall (%) of MSA-MP, PSIPRED, and our method on ASTRAL30 test dataset (H=Helix, E=Strand, C=Coil)
partial data source from: *Bayesian model of protein primary sequence for secondary structure prediction* [14]

	MSA-MP			PSIPRED-PDB			PSIPRED-NR			Our Method		
	H	E	C	H	E	C	H	E	C	H	E	C
H	73	7	11	77	6	10	82	2	7	87	4	12
E	7	67	9	4	62	10	1	69	7	2	78	13
C	20	26	81	19	32	80	17	29	86	11	18	75
Overall	74			75			81			80		

Table 4.5: Classification Q3 recall (%) of MSA-MP, PSIPRED, and our method on CASP9 test dataset (H=Helix, E=Strand, C=Coil)
partial data source from: *Bayesian model of protein primary sequence for secondary structure prediction* [14]

the accuracy. For example, we can try to increase sizes of hidden layers, because it may further improve the accuracy, as shown in the Figures 4.1. Besides, we can also add the number of hidden layers and attempt much deeper architecture.

Chapter 5

Critical Evaluation

As mentioned in the conclusion of the previous chapter, the project not merely predict the secondary structures of proteins successfully, the accuracy of the prediction is pretty good. Because of my lack of the knowledge of bioinformatics and even machine learning, the results are already beyond my original expectations.

I am satisfied with Python and Theano, they do increase the efficiency of both program development and the practical prediction of neural networks. Following the principle of agile, I always kept the good reusability, extensibility and maintainability of the PSSPred program.

One of the biggest challenges for this project is Multiple Sequence Alignment. Even though it may be common in bioinformatics, it took me much time to understand how to use PSI-BLAST and how to generate the PSSMs. The good news is that it is indeed worthwhile because it did bring a big improvement of accuracy.

However, there are something need to be improved. I am not very satisfied with the experimental settings. Because of the time constraint, I failed complete the full-scale experiments. For example, we should test more hidden neurons or more hidden layers. On the other hand, it will be better to get the averaged results of a couple of experiments, not only once. Fortunately, as I tested before, the error range of results is usually quite small.

For further development of this project, we can improve the experimental environment, attempt more neurons in hidden layers and two or more hidden layers of neural networks. Moreover, we can use a *jury decision* to combine the results of different neural networks as shown in PHD method [19]. Beyond that, with the rapid development of deep learning, we should have tried deep learning networks, such as convolutional neural networks and deep belief networks. Especially, in this respect, we have implemented artificial neural networks in Theano.

Appendices

Appendix A

Experimental Environment

Here we show the experiment environment of this project, including the hardware environment and the software environment.

1.1 Hardware

The program was running on the Cardiff Clusters of High Performance Computing (HPC) Wales. We need to submit the jobs of running code to a queue. The system distributes the jobs into different clusters. The clusters that run the code are the HPC Wales HTC sub-system. It comprises a total of 167 nodes and associated infrastructure designed for High Throughput Computing, specifically:

- 162 x BX922 dual-processor nodes, each having two, six-core Intel Westmere Xeon X5650 2.67 GHz CPUs and 36GB of memory, providing a total of 1994 Intel Xeon cores (with 3 GB of memory/core).
- 4 RX600 X7550 dual processor Intel Nehalem nodes, 2.00 GHz, each with 128 GB RAM.
- 1 RX900 X7550 node with 8 Nehalem processors, 2.00 GHz and 512 GB RAM.
- Total memory capacity for the system of 6.85 TBytes.
- 100 TBytes of permanent storage.

1.2 Software

HPC Wales continually updates application packages, compilers, communications libraries, tools, and math libraries. To facilitate this task and to provide a uniform mechanism for accessing different revisions of software, HPC Wales uses the modules utility.

The software applied in this project can be seen in the loaded module, as follows:

```
compiler/gnu-4.6.2
python/2.7.3-gnu-4.6.2
numpy/1.6.2-gnu-4.6.2
scipy/0.11.0-gnu-4.6.2
theano/0.7.0-python-2.7
```

Appendix B

PSSPred Usage

PSSPred has two commands: train and test. As their names imply, they are for training model and testing model respectively.

2.1 PSSPred

```
Usage: psspred [OPTIONS] COMMAND [ARGS]...

  Protein Secondary Structure Prediction

Options:
  -v, --verbose  Show the detail.
  -q, --quiet    Do not print any infomation.
  --version      Show the version and exit.
  --help         Show this message and exit.

Commands:
  test
  train
```

2.2 PSSPred-Train

```
Usage: psspred train [OPTIONS]

Options:
  --first-only                    Train the first-level network only.
  --second-only                   Train the second-level network only.
  -c, --config PATH               The config file to use instead of the
                                  default.
  -lr, --learning-rate FLOAT      Set the learning rate of training manually.
  -s, --save-training-progress FILENAME
                                  Save the training progress into a file
  --help                          Show this message and exit.
```

2.3 PSSPred-Test

```
Usage: psspred test [OPTIONS] FILENAME

Options:
  -1, --first PATH   The first-level network file.
  -2, --second PATH  The second-level network file.
  --help             Show this message and exit.
```

Appendix C

Source Code

```
#!/usr/bin/config python
# -*- coding: utf-8 -*-
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

import click
import datetime
import logging
import numpy as np
import theano
import theano.tensor as T

from itertools import product

try:
    import configparser
    import pickle
except ImportError:
    import ConfigParser as configparser
    import cPickle as pickle
    from itertools import izip as zip
    input = raw_input
    range = xrange


__version__ = 0.2


def floatX(X):
    return np.asarray(X, dtype=theano.config.floatX)


def piecewise_scaling_func(x):
    if x < -5:
        y = 0.0
    elif -5 <= x <= 5:
        y = 0.5 + 0.1*x
    else:
        y = 1.0
    return y
```

```
def encode_residue(residue):
    return [1 if residue == amino_acid else 0
            for amino_acid in ('A', 'R', 'N', 'D', 'C', 'E', 'Q', 'G', 'H',
                               'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W',
                               'Y', 'V')]


def encode_dssp(dssp):
    return [1 if dssp == hec else 0 for hec in ('H', 'E', 'C')]


def shared_dataset(data_xy, borrow=True):
    data_x, data_y = data_xy
    shared_x = theano.shared(floatX(data_x), borrow=borrow)
    shared_y = theano.shared(floatX(data_y), borrow=borrow)
    return shared_x, shared_y


def load_data(filename, window_size=19):
    logging.info('... loading data ("%s")' % filename)

    X = []
    Y = []
    with open(filename, 'r') as f:
        line = f.read().strip().split('\n')
        num_proteins = len(line) // 2

        for line_num in range(num_proteins):
            sequence = line[line_num*2]
            structure = line[line_num*2 + 1]

            double_end = [None] * (window_size // 2)
            unary_sequence = []
            for residue in double_end + list(sequence) + double_end:
                unary_sequence += encode_residue(residue)

            X += [
                unary_sequence[start: start+window_size*20]
                for start in range(0, len(sequence)*20, 20)
            ]

            Y += [encode_dssp(dssp) for dssp in structure]

    return shared_dataset([X, Y])


def load_pssm(filename, window_size=19, scale=piecewise_scaling_func):
    logging.info('... loading pssm ("%s")', filename)

    X = []
    Y = []
    with open(filename, 'r') as f:
        num_proteins = int(f.readline().strip())
        for __ in range(num_proteins):
            m = int(f.readline().strip())
            sequences = []
            for __ in range(m):
                line = f.readline()
                sequences += [scale(float(line[i*3: i*3+3]))
                              for i in range(20)]

            double_end = ([0.]*20) * (window_size//2)
            sequences = double_end + sequences + double_end
            X += [
                sequences[start:start+window_size*20]
                for start in range(0, m*20, 20)
            ]

            structure = f.readline().strip()
            Y += [encode_dssp(dssp) for dssp in structure]

    return shared_dataset([X, Y])


class AccuracyTable(object):

    def __init__(self, pred=None, obs=None):
        self.A = np.zeros(shape=(3, 3), dtype=float)
        if pred is not None and obs is not None:
            self.count(pred, obs)

    """
    ij = number of residues predicted to be in structure type j and observed
    to be in type i.
    """
    def count(self, pred, obs):
        for p, o in zip(pred, obs):
            self.A[o][p] += 1

    @property
    def Q3(self):
        return self.A.trace() / self.A.sum() * 100

    def C(self, i):
        if not 0 <= i < 3:
            raise ValueError('the argument i can only be 0(helix), 1(strand),'
                             '2(coil)')

        p = self.A[i][i]
        n = sum(self.A[j][k] if j != i and k != i else 0
                for j, k in product(range(3), repeat=2))
        o = sum(self.A[j][i] if j != i else 0 for j in range(3))
        u = sum(self.A[i][j] if j != i else 0 for j in range(3))
        return (p*n-o*u) / ((p+o)*(p+u)*(n+o)*(n+u))**0.5

    @property
    def Ch(self):
        return self.C(0)

    @property
    def Ce(self):
        return self.C(1)

    @property
    def Cc(self):
        return self.C(2)
```

```python
    @property
    def C3(self):
        return (self.Ch * self.Ce * self.Cc) ** (1./3)

    def __str__(self):
        res = ''
        for i in range(3):
            for j in range(3):
                res += str(self.A[i][j]) + '\t'
            res += '\n'
        return res


class StoppingCriteria(object):
    def __init__(self, k=5):
        self.t = 0
        self.k = k
        self.E_tr = [np.inf]
        self.E_va = [np.inf]
        self.E_opt = np.inf

    def append(self, E_tr, E_va):
        self.t += 1
        self.E_tr.append(E_tr)
        self.E_va.append(E_va)
        self.E_opt = min(self.E_opt, E_va)

    @property
    def generalization_loss(self):
        return 100. * (self.E_va[-1]/self.E_opt - 1)

    @property
    def training_progress(self):
        return 1000. * (sum(self.E_tr[-self.k:]) /
                        (self.k * min(self.E_tr[-self.k:])) - 1)

    def GL(self, alpha):
        """Stop as soon as the generalization loss exceeds a certain threshold.
        """
        return self.generalization_loss > alpha

    def PQ(self, alpha):
        """Stop as soon as quotient of generalization loss and progress exceeds
        a certain threshold
        """
        return self.generalization_loss / self.training_progress > alpha

    def UP(self, s, t=0):
        """Stop when the generalization error increased in s successive strips.
        """
        if t == 0:
            t = self.t
        if t - self.k < 0 or self.E_va[t] <= self.E_va[t - self.k]:
            return False
        if s == 1:
            return True
        return self.UP(s - 1, t - self.k)
```

```
def init_weights_sigmoid(shape):
    low = -np.sqrt(6./(shape[0]+shape[1])) * 4.
    high = np.sqrt(6./(shape[0]+shape[1])) * 4.
    values = np.random.uniform(low=low, high=high, size=shape)
    return theano.shared(floatX(values), borrow=True)


def init_weights(shape):
    values = np.random.randn(*shape)*0.01
    return theano.shared(floatX(values), borrow=True)


def init_bias(shape):
    values = np.zeros(shape, dtype=theano.config.floatX)
    return theano.shared(values, borrow=True)


class MultilayerPerceptron(object):
    def __init__(self, n_input, n_hidden, n_output):
        logging.info('... building model (%d-%d-%d)',
                     n_input, n_hidden, n_output)

        self.W_h = init_weights_sigmoid((n_input, n_hidden))
        self.b_h = init_bias(n_hidden)
        self.W_o = init_weights((n_hidden, n_output))
        self.b_o = init_bias(n_output)

        self.params = [self.W_h, self.b_h, self.W_o, self.b_o]

        self.X = T.matrix()
        self.Y = T.matrix()

        h = T.nnet.sigmoid(T.dot(self.X, self.W_h) + self.b_h)
        self.py_x = T.nnet.softmax(T.dot(h, self.W_o) + self.b_o)

        y = T.argmax(self.Y, axis=1)
        self.NLL = -T.mean(T.log(self.py_x)[T.arange(self.Y.shape[0]), y])
        self.L1 = T.sum(abs(self.W_h)) + T.sum(abs(self.W_o))
        self.L2_sqr = T.sum((self.W_h**2)) + T.sum((self.W_o**2))

    def train_model(self, X_train, Y_train, X_valid, Y_valid,
                    num_epochs=3000, learning_rate=0.001, batch_size=20,
                    L1_reg=0., L2_reg=0.):

        logging.info('... training model (learning_rate: %f)' % learning_rate)

        cost = self.NLL + L1_reg*self.L1 + L2_reg*self.L2_sqr

        grads = T.grad(cost=cost, wrt=self.params)
        updates = [[param, param - learning_rate*grad]
                   for param, grad in zip(self.params, grads)]

        start = T.lscalar()
        end = T.lscalar()

        train = theano.function(
            inputs=[start, end],
            outputs=cost,
            updates=updates,
            givens={
                self.X: X_train[start:end],
                self.Y: Y_train[start:end]
            }
        )

        validate = theano.function(
            inputs=[start, end],
            outputs=[cost, self.py_x],
            givens={
                self.X: X_valid[start:end],
                self.Y: Y_valid[start:end]
            }
        )

        m_train = X_train.get_value(borrow=True).shape[0]
        m_valid = X_valid.get_value(borrow=True).shape[0]

        stopping_criteria = StoppingCriteria()
        index = range(0, m_train+1, batch_size)

        y_valid = np.argmax(Y_valid.get_value(borrow=True), axis=1)
        for i in range(num_epochs):
            costs = [train(index[j], index[j+1]) for j in range(len(index)-1)]
            E_tr = np.mean(costs)

            E_va, py_x = validate(0, m_valid)
            y_pred = np.argmax(py_x, axis=1)
            A_valid = AccuracyTable(y_pred, y_valid)

            stopping_criteria.append(E_tr, E_va)
            logging.debug('epoch %3d/%d. Cost: %f  Validation: Q3=%.2f%% C3=%f'
                          ' (%.2f %.2f %.2f)',
                          i+1, num_epochs, E_tr, A_valid.Q3, A_valid.C3,
                          A_valid.Ch, A_valid.Ce, A_valid.Cc)

            if stopping_criteria.PQ(1):
                logging.debug('Early Stopping!')
                break

        return stopping_criteria

    def predict(self, X):
        start = T.lscalar()
        end = T.lscalar()
        return theano.function(
            inputs=[start, end],
            outputs=self.py_x,
            givens={self.X: X[start:end]}
        )


class Config(object):

    def __init__(self, profile, section):
        parser = configparser.RawConfigParser()
        parser.read(profile)
        self.train_file = parser.get(section, 'training_file')
        self.valid_file = parser.get(section, 'validation_file')
        self.window_size = parser.getint(section, 'window_size')
        self.hidden_layer_size = parser.getint(section, 'hidden_layer_size')
        self.learning_rate = parser.getfloat(section, 'learning_rate')
        self.num_epochs = parser.getint(section, 'num_epochs')
        if section == 'SECOND':
            self.network_file = parser.get(section, 'network_file')


def first_level(cfg, target):
    now = datetime.datetime.now()
    logging.info(now.strftime('%Y-%m-%d %H:%M:%S'))

    X_train, Y_train = load_pssm(cfg.train_file, window_size=cfg.window_size)
    X_valid, Y_valid = load_pssm(cfg.valid_file, window_size=cfg.window_size)

    input_layer_size = cfg.window_size * 20
    output_layer_size = 3

    classifier = MultilayerPerceptron(input_layer_size,
                                      cfg.hidden_layer_size,
                                      output_layer_size)

    result = classifier.train_model(X_train, Y_train, X_valid, Y_valid,
                                    cfg.num_epochs,
                                    learning_rate=cfg.learning_rate)

    if target is not None:
        for E_tr, E_va in zip(result.E_tr, result.E_va):
            target.write(str(E_tr) + ',' + str(E_va) + '\n')

    network_file = now.strftime('%Y%m%dT%H%M%S') + '.nn1'
    logging.info('... saving model in file (%s)', network_file)
    pickle.dump(classifier, open('output/' + network_file, 'wb'))

    return classifier


def second_level(cfg, fst_layer_classifier, target):

    def transform(x, m, window_size=17):
        double_end = [0.] * 3 * (window_size // 2)
        sequences = double_end + x.tolist()[0] + double_end
        return [sequences[index: index+window_size*3]
                for index in range(0, m*3, 3)]

    def get_XY(filename):
        X_data, Y_data = load_pssm(filename)
        m = X_data.get_value(borrow=True).shape[0]
        predict = fst_layer_classifier.predict(X_data)
        x = predict(0, m).reshape(1, m*3)
        x = transform(x, m, cfg.window_size)
        X = theano.shared(floatX(x), borrow=True)
        return X, Y_data

    now = datetime.datetime.now()
    logging.info(now.strftime('%Y-%m-%d %H:%M:%S'))
    if fst_layer_classifier is None:
        with open(cfg.network_file, 'rb') as f:
            fst_layer_classifier = pickle.load(f)

    X_train, Y_train = get_XY(cfg.train_file)
    X_valid, Y_valid = get_XY(cfg.valid_file)

    snd_layer_classifier = MultilayerPerceptron(cfg.window_size*3,
                                                cfg.hidden_layer_size,
                                                3)
    result = snd_layer_classifier.train_model(X_train, Y_train,
                                              X_valid, Y_valid,
                                              cfg.num_epochs,
                                              cfg.learning_rate)

    if target is not None:
        for E_tr, E_va in zip(result.E_tr, result.E_va):
            target.write(str(E_tr) + ',' + str(E_va) + '\n')

    network_file = now.strftime('%Y%m%dT%H%M%S') + '.nn2'
    logging.info('... saving model in file (%s)' % network_file)
    pickle.dump(snd_layer_classifier, open('output/' + network_file, 'wb'))


@click.group()
@click.option('--verbose', '-v', is_flag=True,
              help='Show the detail.')
@click.option('--quiet', '-q', is_flag=True,
              help='Do not print any infomation.')
@click.version_option(version=__version__)
def cli(verbose, quiet):
    """Protein Secondary Structure Prediction"""
    if quiet:
        level = logging.WARNING
    elif verbose:
        level = logging.DEBUG
    else:
        level = logging.INFO
    logging.basicConfig(format='', level=level)


@cli.command()
@click.option('--first-only', is_flag=True,
              help='Train the first-level network only.')
@click.option('--second-only', is_flag=True,
              help='Train the second-level network only.')
@click.option('--config', '-c', type=click.Path(exists=True),
              help='The config file to use instead of the default.')
@click.option('--learning-rate', '-lr', default=0.,
              help='Set the learning rate of training manually.')
@click.option('--save-training-progress', '-s', 'target', type=click.File('w'),
              help='Save the training progress into a file')
def train(first_only, second_only, config, learning_rate, target):
    if config is None:
        config = 'default.cfg'

    config4first = Config(config, 'FIRST')
    config4second = Config(config, 'SECOND')
    if learning_rate > 0:
        config4first.learning_rate = learning_rate
        config4second.learning_rate = learning_rate

    classifier = None
    if not second_only:
        classifier = first_level(config4first, target)
    if not first_only:
        second_level(config4second, classifier, target)


@cli.command()
@click.option('--first', '-1', type=click.Path(exists=True),
              help='The first-level network file.')
@click.option('--second', '-2', type=click.Path(exists=True),
              help='The second-level network file.')
@click.argument('filename', type=click.Path(exists=True))
def test(first, second, filename):

    def transform(x, m, window_size=17):
        double_end = [0.] * 3 * (window_size // 2)
        sequences = double_end + x.tolist()[0] + double_end
        return [sequences[index: index+window_size*3]
                for index in range(0, m*3, 3)]

    def get_XY(window_size=17):
        X_data, Y_data = load_pssm(filename)
        m = X_data.get_value(borrow=True).shape[0]
        predict = fst_layer_classifier.predict(X_data)
        x = predict(0, m).reshape(1, m*3)
        x = transform(x, m, window_size)
        X = theano.shared(floatX(x), borrow=True)
        return X, Y_data

    now = datetime.datetime.now()
    logging.info(now.strftime('%Y-%m-%d %H:%M:%S'))

    with open(first, 'rb') as f:
        fst_layer_classifier = pickle.load(f)

    with open(second, 'rb') as f:
        snd_layer_classifier = pickle.load(f)

    window_size = snd_layer_classifier.W_h.get_value(borrow=True).shape[0] // 3
    X_test, Y_test = get_XY(window_size)

    snd_layer_classifier.train_model(X_test, Y_test, X_test, Y_test, 1)
```

Annotated Bibliography

[1] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio, “Theano: new features and speed improvements,” Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[2] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: a CPU and GPU math expression compiler,” in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010, oral Presentation.

[3] Biochem.ucl.ac.uk. (2015) Protein secondary structure prediction with neural nets: The basics. [Online]. Available: http://www.biochem.ucl.ac.uk/~shepherd/sspred_tutorial/ss-pred-old.html

[4] Blast.ncbi.nlm.nih.gov. (2015) Protein blast: search protein databases using a protein query. [Online]. Available: http://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE=Proteins&PROGRAM=blastp&RUN_PSIBLAST=on

[5] M. Brylinski and J. Skolnick, “FINDSITE-metal: integrating evolutionary information and machine learning for structure-based metal-binding site prediction at the proteome level.” *Proteins*, vol. 79, no. 3, pp. 735–751, Mar. 2011.

[6] J. M. Chandonia and M. Karplus, “The importance of larger data sets for protein secondary structure prediction with neural networks.” *Protein science : a publication of the Protein Society*, vol. 5, no. 4, pp. 768–774, Apr. 1996.

[7] J.-M. Chandonia, G. Hon, N. S. Walker, L. Lo Conte, P. Koehl, M. Levitt, and S. E. Brenner, “The ASTRAL Compendium in 2004.” *Nucleic acids research*, vol. 32, no. Database issue, pp. D189–92, Jan. 2004.

[8] J. A. Cuff and G. J. Barton, “Application of multiple sequence alignment profiles to improve protein secondary structure prediction.” *Proteins*, vol. 40, no. 3, pp. 502–511, Aug. 2000.

[9] A. Drozdetskiy, C. Cole, J. Procter, and G. J. Barton, “JPred4: a protein secondary structure prediction server.” *Nucleic acids research*, p. gkv332, Apr. 2015.

[10] C. Floudas, H. Fung, S. McAllister, M. Mnnigmann, and R. Rajgaria, “Advances in protein structure prediction and de novo protein design: A review,” *Chemical Engineering Science*, vol. 61, no. 3, pp. 966 – 988, 2006, biomolecular Engineering. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0009250905002988

[11] N. K. Fox, S. E. Brenner, and J.-M. Chandonia, "SCOPe: Structural Classification of Proteins - extended, integrating SCOP and ASTRAL data and classification of new structures." *Nucleic acids research*, vol. 42, no. Database issue, pp. 304–309, 2014.

[12] D. T. Jones, "Protein secondary structure prediction based on position-specific scoring matrices," *Journal of molecular biology*, vol. 292, no. 2, pp. 195–202, Sept. 1999.

[13] H. Kim and H. Park, "Protein secondary structure prediction based on an improved support vector machines approach." *Protein engineering*, vol. 16, no. 8, pp. 553–560, Aug. 2003.

[14] Q. Li, D. B. Dahl, M. Vannucci, H. Joo, and J. W. Tsai, "Bayesian model of protein primary sequence for secondary structure prediction." *PloS one*, vol. 9, no. 10, p. e109832, 2014.

[15] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme." *Biochimica et biophysica acta*, vol. 405, no. 2, pp. 442–451, Oct. 1975.

[16] J. Moult, K. Fidelis, A. Kryshtafovych, and A. Tramontano, "Critical assessment of methods of protein structure prediction (CASP)–round IX." *Proteins*, vol. 79 Suppl 10, pp. 1–5, 2011.

[17] L. Prechelt, "Early Stopping - But When?" *Neural Networks Tricks of the Trade (2nd ed.) 201253-67*, vol. 7700, no. Chapter 5, pp. 53–67, 2012.

[18] N. Qian and T. J. Sejnowski, "Predicting the secondary structure of globular proteins using neural network models." *Journal of molecular biology*, vol. 202, no. 4, pp. 865–884, Aug. 1988.

[19] B. Rost and C. Sander, "Prediction of protein secondary structure at better than 70pp. 584 – 599, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022283683714130

[20] M. Spencer, J. Eickholt, and J. Cheng, "A Deep Learning Network Approach to ab initio Protein Secondary Structure Prediction," *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM*, vol. 12, no. 1, pp. 103–112, Aug. 2014.

[21] A. Zemla, C. Venclovas, K. Fidelis, and B. Rost, "A modified definition of Sov, a segment-based measure for protein secondary structure prediction assessment." *Proteins*, vol. 34, no. 2, pp. 220–223, Feb. 1999.