Predicting the depths of Amino Acids using Neural Networks

By: Tanayaa Bhagdikar (20151144)

Supervisor : Dr. M.S. Madhusudhan, Indian Institute of Science Education & Research, Pune

Expert Advisor : Dr Pranay Goel, Indian Institute of Science Education & Research, Pune

CERTIFICATE

This is to certify that this dissertation entitled "**Predicting the depths of Amino Acids using Neural Networks**" towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by **Tanayaa Bhagdikar** at **I.I.S.E.R. Pune** under the supervision of **Dr. M.S. Madhusudhan, Dept. of Biology** during the academic year **2019-20**

Signature of Student **Signature of Supervisor**

DECLARATION

I hereby declare that the matter embodied in the report entitled **Predicting the depths of Amino Acids using Neural Networks** are the results of the work carried out by me at the **Department of Biology**, **I.I.S.E.R. Pune**, under the supervision of **Dr. M.S. Madhusudhan** and the same has not been submitted elsewhere for any other degree.

Signature of Student **Signature of Supervisor**

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The aim of our project was to predict the depths of all the amino acids in a given query sequence. Residue depth is an important parameter to study various other properties of the protein. To predict the same we used neural networks. We had earlier tried statistical methods to predict residue depth but failed. We extracted all information from the query sequence and used these as features for our neural network. We subsequently trained several networks and tweaked the input features to improve the accuracy. We also tried changing the number of nodes in every hidden layer and the number of hidden layers. Our final model had negligible accuracy. This was because the input features we could extract, only given the query was far too weakly correlated to the final residue depth. Other constraints of energy and structure would have probably led to a better prediction.

INTRODUCTION

Proteins are among the most abundant and vital molecules in any living system. The function of any protein can be deduced by studying its 3 dimensional structure, which depends on its amino acid sequence. Different amino acids have different preferences as to where they are present in a protein [Figure 1]. Charged amino acids such as Lysine and Arginine prefer to be present on the surface to interact with the solvent, while uncharged amino acids are generally buried.

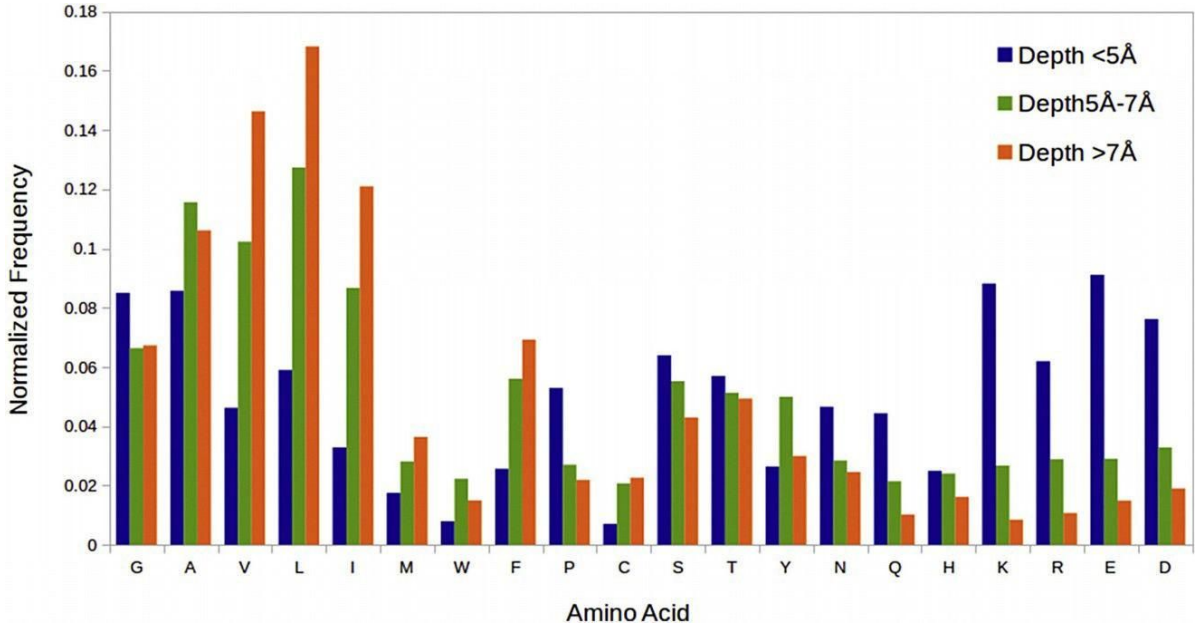

Figure 1: Different amino acids prefer different environments

The environment an amino acid is present in can be quantified by a parameter called residue depth. Residue depth is computed as the distance between the amino acid and the bulk solvent[3]. Residue depth is useful in predicting small molecule binding sites, cavities, pKa of amino acids and the effect of deleterious mutations in proteins[1,2,4]. Hence, prediction of amino acid depths can help in determining the structure of the protein and help in predicting the other properties as mentioned above.

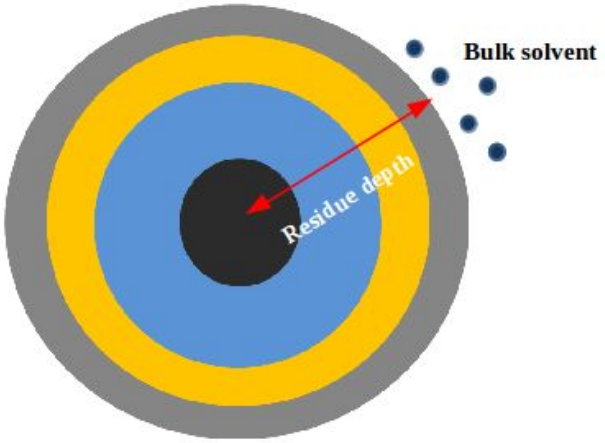

Figure 2: Computation of residue depth

Protein structure prediction is an important problem in Biology because there is a huge disparity between the number of known sequences and the number of known structures. The current version of UniProt contains 133507323 sequence entries, yet the protein data bank has only 146266 protein structures[7]. Crucially, the number of unique folds these structures can be classified into is only 1195. There are several methods of protein structure prediction in use including homology modelling, threading and *ab initio* prediction methods.

In this project, we will be using the method of Threading for protein structure prediction and subsequent depth prediction. Threading works by sliding an unknown query sequence through all known protein folds and scoring the models so generated. The best scoring model is the predicted fold. Predicting the fold correctly, hence, will directly allow us to infer the depths of the query sequence.

We first obtained the set of all known protein folds from HOMSTRAD [5]. Out of 2841 possible folds, we decided to use 1752 single-domain folds as our template folds.

Initially we tried purely statistical methods to predict residue depth. We studied a non-redundant set of 968 proteins to calculate the propensities of each amino acid being at different depths. The structures of these proteins were obtained from the RCSB database. A 30% redundant set was obtained using PISCES [6]. It was also ensured that the structures obtained had no missing residues. Finally, we used DEPTH to calculate the depths of each residue in a given protein structure– and use these observations for scoring.

We computed the probability for each amino acid to occur at a particular depth, using above data. The scores (probabilities) were computed by taking into account the fluctuation in depths of an amino acid. Given one observation, we assign scores to all the depths that residue can be at by multiplying the bin width (i.e. 0.1) with the cumulative distribution function of the Gaussian at that point.The scores for each bin are successively added up, accounting for all depths a given residue is observed at.

We further normalized these scores such that given a certain depth interval, all the individual scores of all amino acids summed up to 1. We observed that hydrophobic amino acids such as Alanine, Valine and Proline had higher scores at higher depths – indicative of the nature of their side chains while hydrophilic amino acids like Serine, Asparagine and Glutamine had higher scores at lower depths [Figure 3].

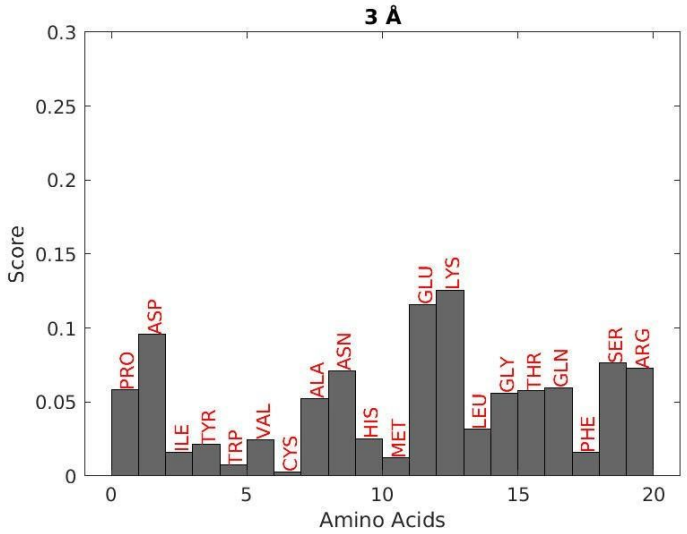

Fig 3(a) : Normalized scores of 20 residues at 3 Å

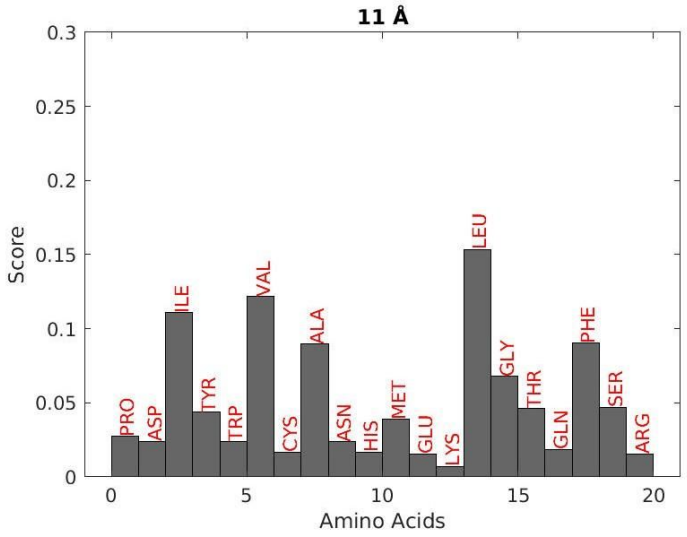

Fig 3(b): Normalized scores of 20 residues at 11 Å

Figure 3: Histograms depicting the scores i.e. probabilities of the 20 amino acids occurring at two different depths

The query sequence when threaded through different templates, was made to inherit the depths of the template residues and scored using our computed propensities, i.e. the probability that residue ‘X’ would be found at depth ‘d’, for a threaded model. The highest scoring model was the predicted fold.

However, upon testing this method, by using known fold sequences as our queries; we got very poor results. The predicted model was never the existing fold, in fact it was usually one of the poorest scoring folds.

Next, we thought that models where the depth distribution of a residue was most similar to its known distribution would be most favourable. So, now, for every threaded model, we further did a spline fit (using inherited depths, and known scores) to obtain the interpolated probability scores of each of the 20 amino acids across all depths, and compare these with the known values. We now penalize each model by computing the difference between interpolated values and known values. It is thought that the more likely a template is, the closer the interpolated values (for each residue) will be to its natural probability distribution – and such models get very low penalties (added to the scores).

We again tested the above method by using a set of “template” sequences as our query sequences. We found that the known model was never one of the top scoring models. Its percentiles ranged from 94 - 9 , the average percentile being 62. In order to check if the predicted model and the known model had any structural similarity, we used a topology independent structural superimposition tool called CLICK[10]. The structural overlap between the predicted and the actual model ranged between 16% - 39%.

There was clearly poor overlap between predicted and known fold . We thought this might be because we disregarded the effect of neighbours (and their depths) on the residue itself. It also might be because we did not check if there was unbalanced buried charge in any of these confirmations - such models should have been heavily penalized.

To take into account the various factors we thought would influence residue depth i.e. charge, environment around the residue etc, we decided to use Machine Learning, specifically Neural Networks.

Neural Networks are a class of Supervised Machine Learning Algorithms. These work by taking in as the input a set of “features” which are correlated to and determine the “target variable”. We briefly introduce the common terminologies associated with Neural Networks below:

(i) Neuron : A neuron is the most basic unit of a neural network. It takes in some input(s), processes it, and generates some output. The incoming inputs are basically multiplied by some “weights” and summed up. An “activation function” acts on this value and the output is transmitted to the next neuron(s). Neural Networks are essentially just layers of neurons, connected to one another, which finally predict the desired variable.

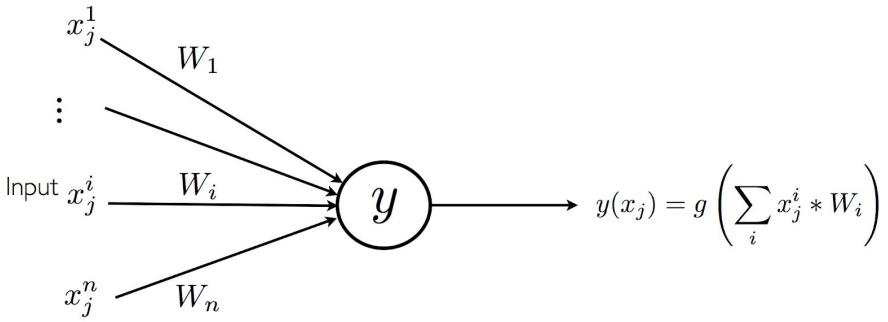

Figure 4: Schematic of a neuron

(ii) Weights : When the neuron receives an input, it multiplies that by a specific weight. For 'n' different inputs to one neuron; there will be 'n' different weights; one for every input. These weights are indicative of the importance of the input; i.e. an input considered more important, will be multiplied by a higher weight. During training a neural network, we initially assign random weights for all inputs - the purpose of training is to find out the optimal weights for each input; in order to produce the result. So, given an input x(i) to a node; it is transformed to W(i)*x(i); where W(i) is its weight.

(iii) Bias: A linear component is applied to the inputs as well. This basically helps to change the range of the quantity W(i)*x(i). The input x(i) is finally processed as W(i)*x(i)+ B.

(iv) Activation Function: This is the part where non-linearity is added to the network. Many real world prediction problems cannot be solved by classical regression techniques because all these models are based on linear algorithms. The activation function is a nonlinear function which acts on the processed quantity W(i)*x(i)+B and this final result is transmitted to the next neuron.

(v)Input / Output / Hidden Layer : The input layer receives the input and is the first layer of the network. It feeds this input to the first hidden layer. The hidden layer is called such because it is not visible to us. These are the processing layers, they

perform specific tasks on the incoming data and pass on the output generated by them to the next layer. The last hidden layer passes on its outputs to the Output Layer; which predicts / classifies the target variable.

(vi) Cost Function: The aim of the network is to predict the output value as close as possible to the known value. The Cost Function is a metric to measure the accuracy of the neural network. It essentially penalizes the network for predicting values deviating from the known values. During training, the network keeps trying to minimize the cost function and increase accuracy.

(vii) Gradient Descent : Gradient descent is an optimization algorithm for minimizing the cost. We start from a point x (which has some cost function J1), and take little steps Δx, and compute the cost at x+Δx (say J2). If J2<J1; we keep proceeding along this path to find a local minimum of the cost function.If J2>J1, we change directions and proceed down the path where the cost function keeps decreasing.

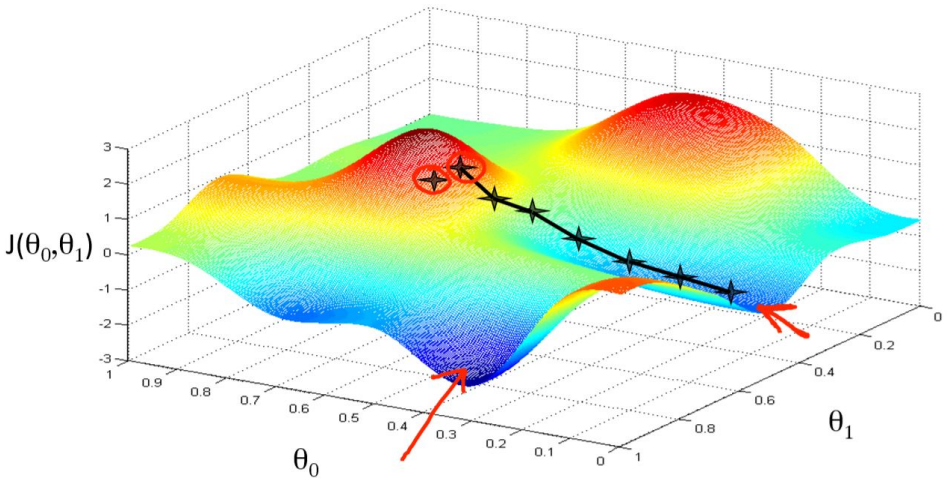

Figure 5: Gradient Descent

(viii) Learning Rate : The learning rate is the amount of minimization of the cost function in each iteration. The rate at which we descend towards the local minima of the cost function is the learning rate. The learning rate needs to be picked very carefully; too large and the network being trained will miss out on the minima, too slow and the network will take a very long time to converge.

(ix) Backpropagation : This is the crux of training a neural network. During training, at each iteration, the cost function is computed and the error of the network can be found. Backpropagation feeds back this error through the nodes along with the gradient of the cost function, at every point updating the weights so as to reduce this error. After every iteration, the weights get closer to their optimal values.

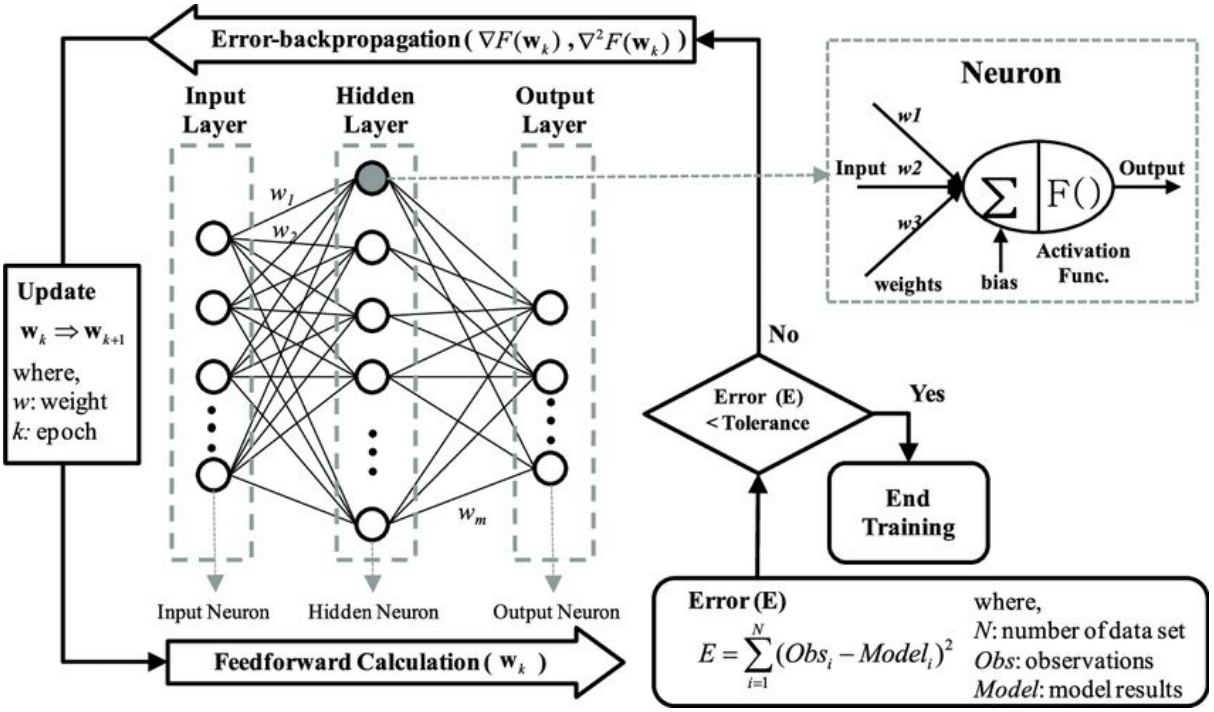

Figure 6: Schematic of a Neural Network

In this project, we will extract “features” or input variables from a given query sequence and use a neural network to predict the target variable, residue depth.

We initially split this problem into training 20 different neural networks, one for each of the 20 essential amino acids. Subsequently, we also tried using just one network and training all the data in one go.

METHODS

There were 3 steps involved in this project:

(i) Obtaining relevant data

(ii) Feature selection and setting up (training) the neural networks, on the training dataset

(iii) After achieving good accuracy, validate our models by making predictions, on the testing dataset

(i) Obtaining relevant data

We first obtained a set of 2841 protein folds from HOMSTRAD, a database of protein folds which classifies them by structural similarity. Out of these, we picked 1751 single-domain protein folds as our template folds, to be threaded through. We picked these single-domain folds by assuming that any fold more than 250 residues long was usually a multi-domain protein fold. Next, we used DEPTH to compute the depths of all the residues along the protein fold.

Out of 1751 folds, we picked 1393 sequences to be our training data, and the remaining to be our testing data

To be specific, we used 2 datasets to train our neural networks. One consisted of the microenvironments of known fold sequences, their features and subsequent depths. This dataset had 128204 rows, each referring to a specific residue present in one of the 1752 single - domain template folds. In this dataset, there was no “threading” taking place, all the rows represented “true” datapoints, known to occur in nature.

The other dataset consisted of 1.3 crore datapoints. These included “true” datapoints as well as “threaded” datapoints. We threaded the 1393 “training queries” through all 1752 folds and obtained the features at every point in all the models generated.

(ii) Feature Selection

I. Microenvironment: The microenvironment of a residue consists of its 6 closest neighbours, within 5Å of it (including sequential ones). These interact with the central residue through Hydrogen bonds and other side-chain interactions. To study the correlation between microenvironment and depths for all residues, we used a set of 4379 proteins, with Sequence Identity<=35%. We wrote a script to find out all the neighbours of a given residue within a distance cutoff of 5Å. After scanning the output text files, we discarded microenvironments which had less than 6 neighbours, for the purposes of our study. This we could do because these microenvironments, with fewer than 6 neighbours were a very small proportion of the entire dataset.

For a given residue, we score all its microenvironments (from the above data) on the basis of their chemical nature. We score each microenvironment as the mean of the kd scale hydrophobicity values of the neighbours comprising it. Furthermore, we plot heatmaps depicting the instances of that residue occuring in a microenvironment having a certain kd score, to depict the probability of the central residue, occurring in such an environment.

Figure 7: Heatmaps depicting the incidences of residue depth vs. kd scores of microenvironments for each of the 20 amino acids. Values scale up by 100

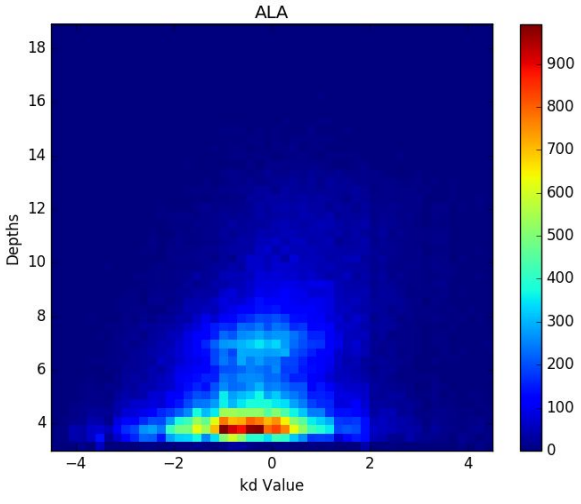

Fig 7(a): Heatmap for ALA

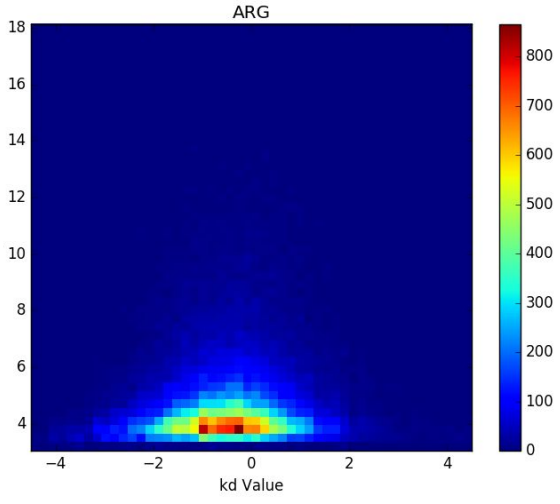

Fig 7(b): Heatmap for ARG

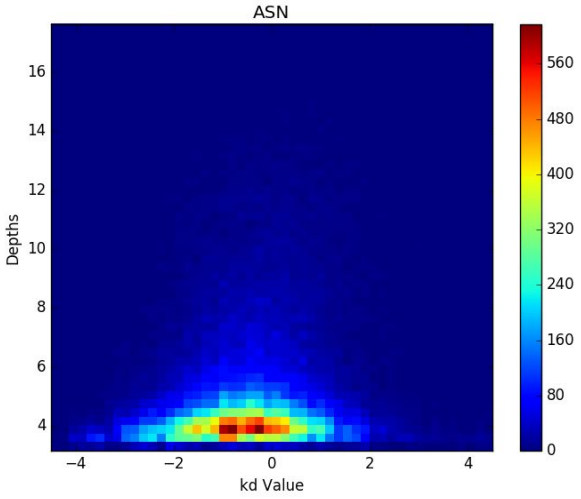

Fig 7(c): Heatmap for ASN

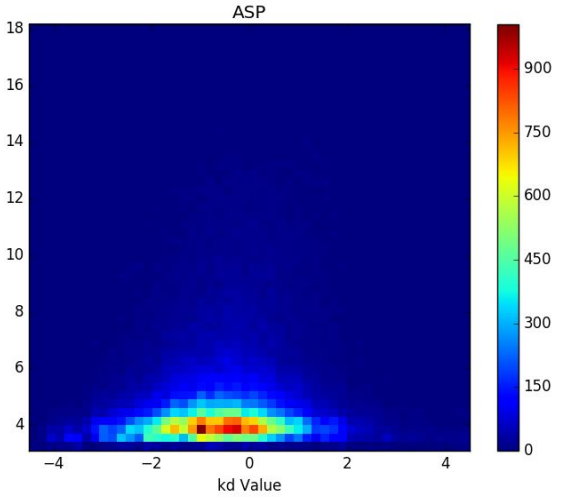

Fig 7(d): Heatmap for ASP

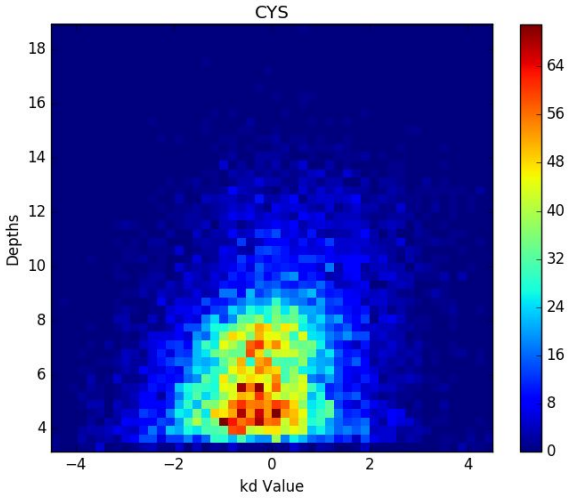

Fig 7(e): Heatmap for CYS

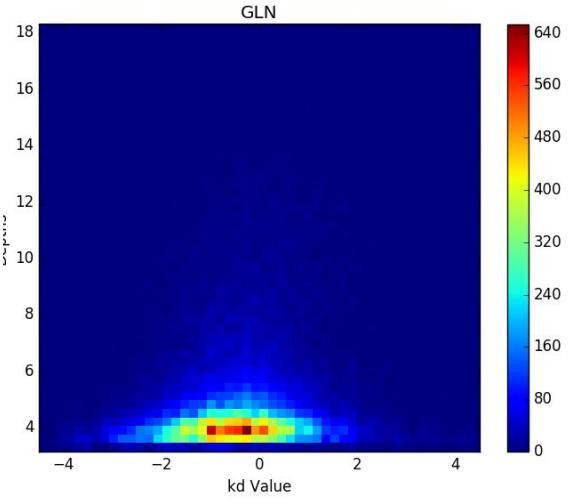

Fig 7(f): Heatmap for GLN

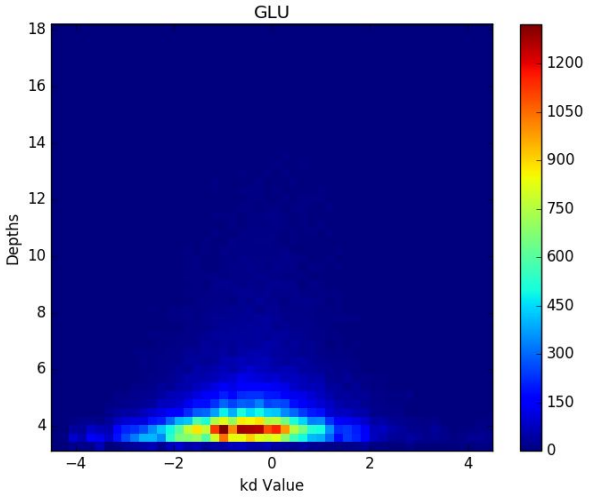

Fig 7(g) : Heatmap for GLU

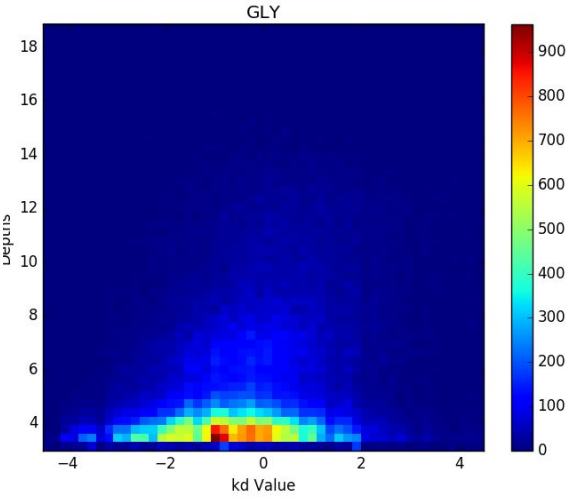

Fig 7(h) : Heatmap for GLY

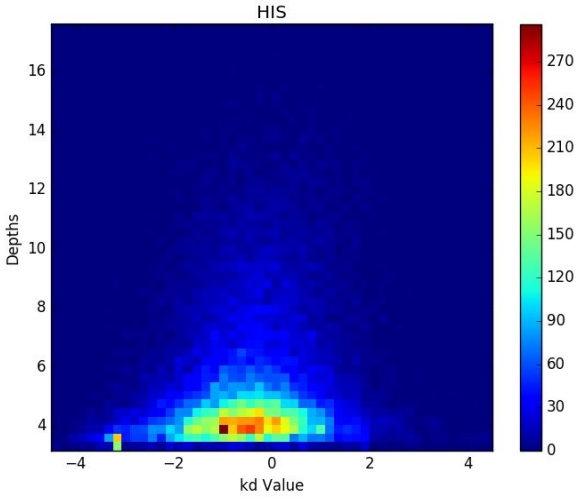

Fig 7(i): Heatmap for HIS

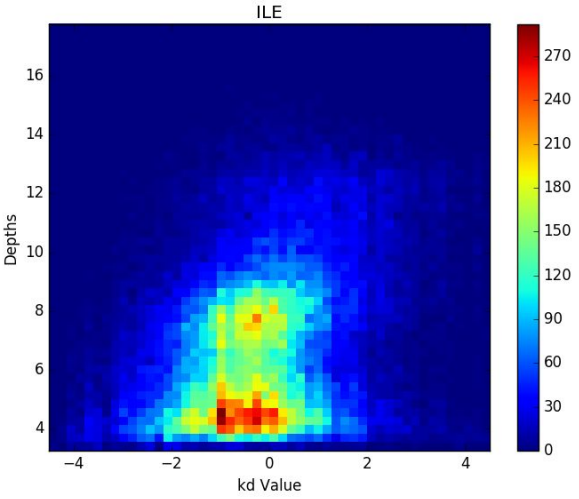

Fig 7(j): Heatmap for ILE

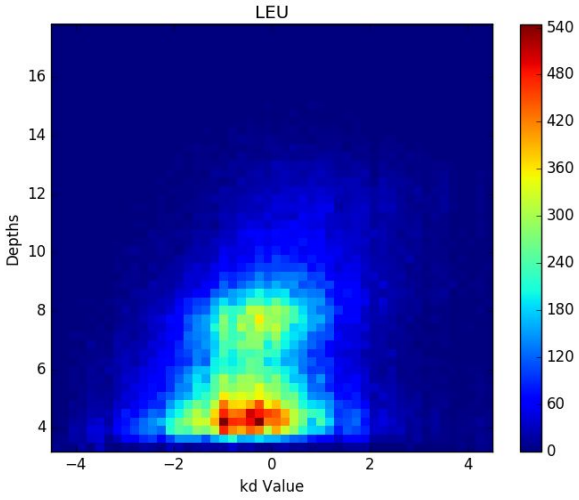

Fig 7(k): Heatmap for LEU

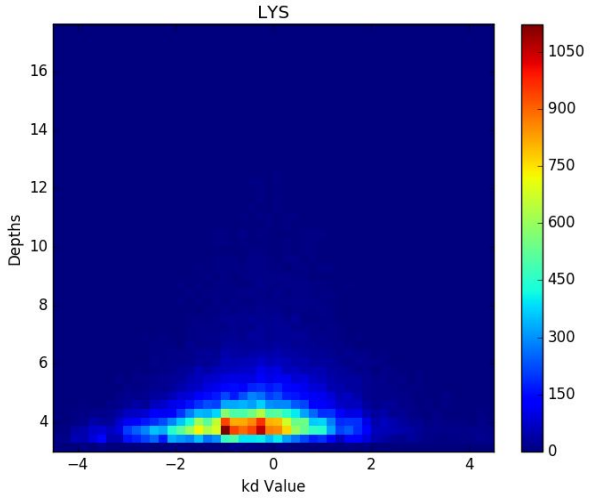

Fig 7(l): Heatmap for LYS

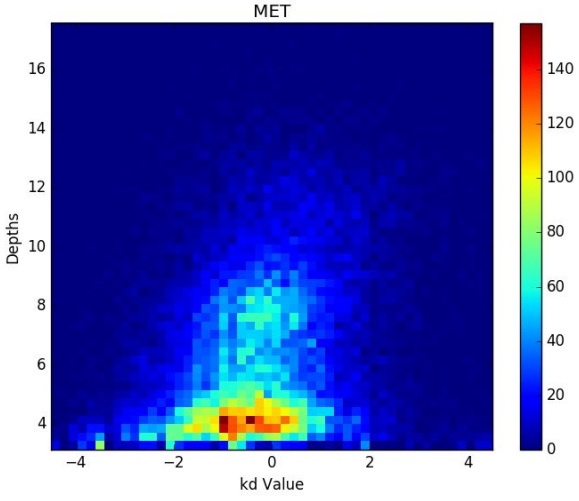

Fig 7(m) : Heatmap for MET

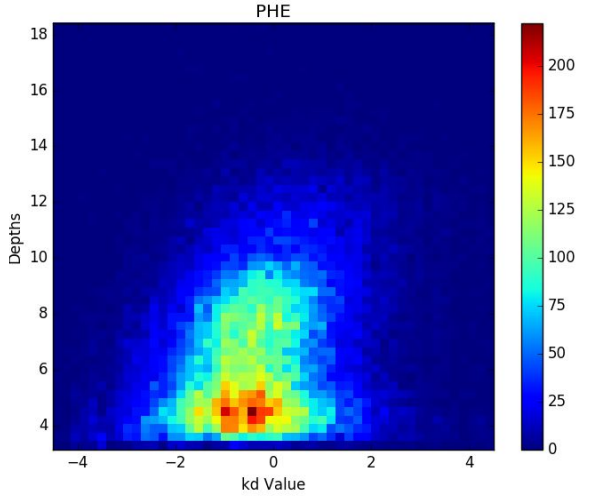

Fig 7(n) : Heatmap for PHE

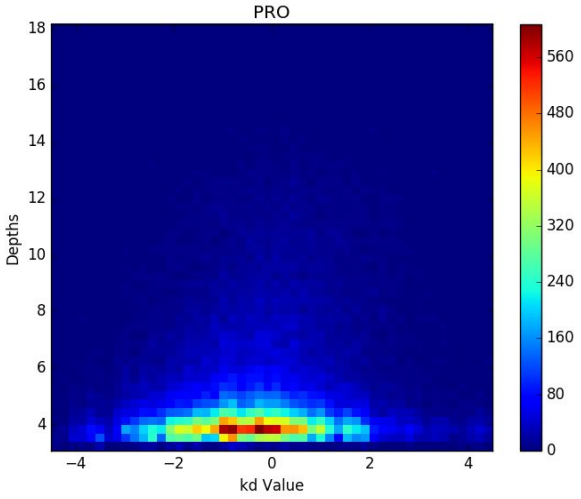

Fig 7(o): Heatmap for PRO

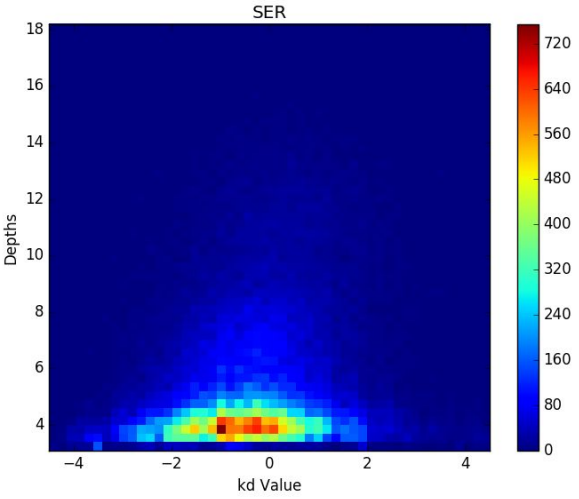

Fig 7(p): Heatmap for SER

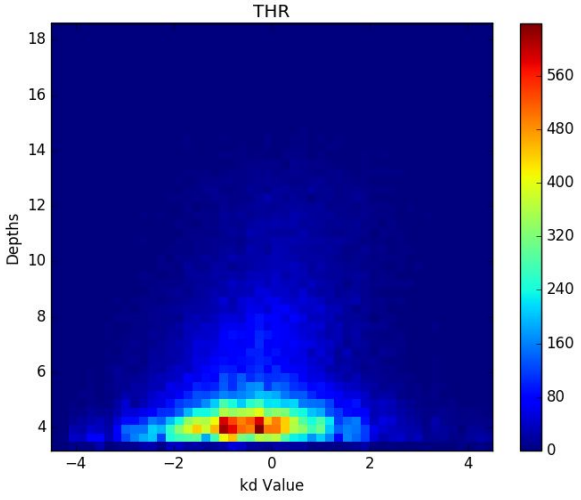

Fig 7(q): Heatmap for THR

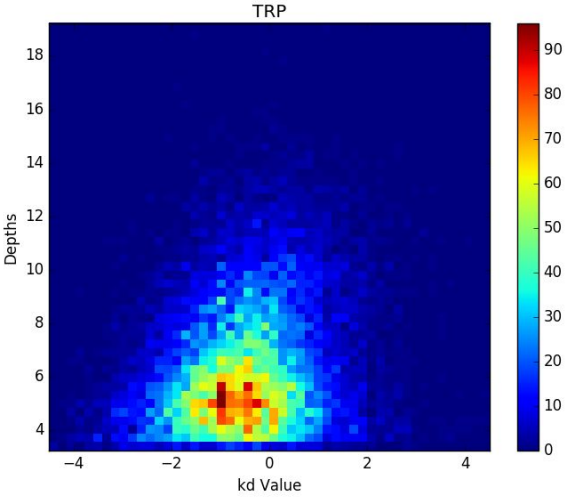

Fig 7(r): Heatmap for TRP

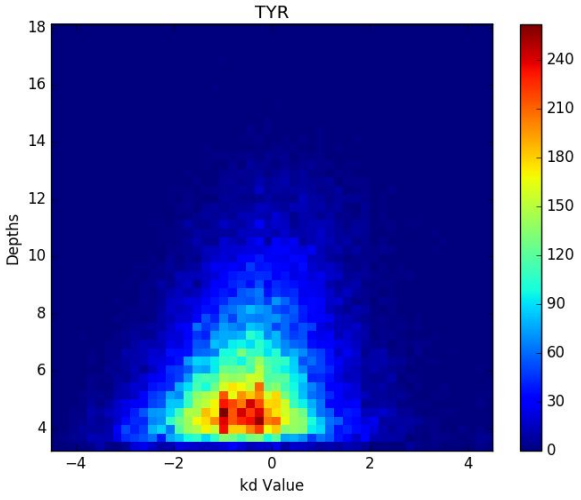

Fig 7(s) : Heatmap for TYR

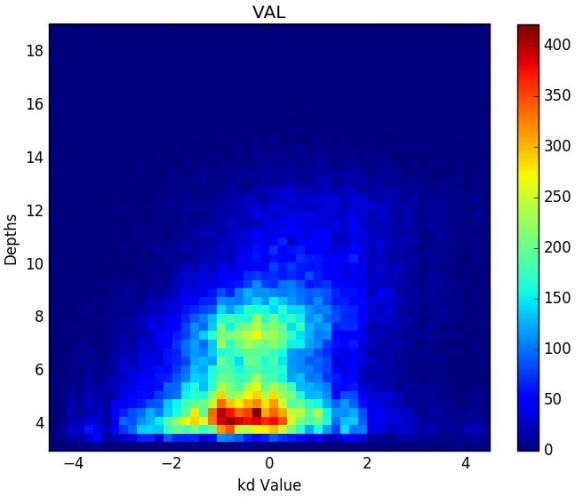

Fig 7(t) : Heatmap for VAL

We then tried to find the relation between the mean kd score of the microenvironment and its depth for all the 20 amino acids, using a simple regression model (a polynomial fit, as a linear model would clearly not work) [Table 1].

Residue	**Datapoints**	**Coefficient of Determination**
ALA	107793	0.070
ARG	55011	0.013
ASP	65866	0.019
ASN	48561	0.022
CYS	13904	0.043
GLN	40534	0.018
GLU	75643	0.017
GLY	84285	0.057
HIS	26382	0.031
ILE	64477	0.084
LEU	102242	0.069
LYS	65316	0.021
MET	25377	0.070
PHE	45612	0.051
PRO	50605	0.017
SER	66411	0.033
THR	60893	0.031
TRP	15519	0.040
TYR	40309	0.034
VAL	79585	0.077

Analyzing the above results, we thought that we might have lost out information while averaging out the scores of the 6 closest neighbours. So, we decided to try setting up a neural network with only the 6 closest neighbours' kd scores as the 6 input features to see if that gave better results.

We analyzed the results from running the above Neural Networks and concluded that more features would be required in order to predict depth accurately. So we extracted all the other features available from the query sequences.

II. pKa and pKc values (of the central residue) - This feature was a proxy for residue charge. We assumed that charged amino acids would generally prefer to be on the outside of a protein fold. If it occured in the interior, it would have to be nullified by other residues in its neighbourhood; to ensure stability.

II. pKa and pKc values (of the sequential neighbours) - The closest neighbours of a given residue are normally its sequential neighbours. Hence, their charge (or lack of it) would impact the position (and depth) of the central residue.

We used the following standard pKa and pKc values of amino acids [Table 2].

Residue	**pKa**	**pKc**
ALA	2.34	9.69
ARG	2.17	9.04
ASP	1.88	9.60
ASN	2.02	8.80
CYS	1.96	8.18
GLN	2.17	9.13
GLU	2.19	9.67

GLY	2.34	9.60
HIS	1.82	9.17
ILE	2.36	9.60
LEU	2.36	9.60
LYS	2.18	8.95
MET	2.28	9.21
PHE	1.83	9.13
PRO	1.99	1.83
SER	2.21	9.15
THR	2.09	9.10
TRP	2.83	9.39
TYR	2.20	9.11
VAL	2.32	9.62

Table 2 :pKa and pKc values of the 20 amino acids

III. Molar Mass of central residue : This feature served as a proxy for residue size. We are not sure how it would affect final residue depth. Following were the standard values we used for residue masses.

Residue	**Molar mass (g/mol)**
ALA	89.09
ARG	174.20
ASP	133.10
ASN	132.12
CYS	126.15
GLN	146.15
GLU	147.13

GLY	75.07
HIS	155.16
ILE	137.18
LEU	131.18
LYS	146.19
MET	149.21
PHE	165.19
PRO	115.13
SER	105.09
THR	119.12
TRP	204.23
TYR	181.19
VAL	117.15

Table 3 : Molar Masses of the 20 amino acids

IV. Number Density of central residue - We defined the number density as the number of neighbours a residue had within 5Å of it. This would range from six (which was the minimal cutoff we imposed for selecting microenvironments) to as high as 10.

Software - All the Neural Networks were run on Tensor Flow, version 1.1.5.0.
We used a dense, deep feed-forward neural network built using Keras. A ‘dense’ network is a network where every node in a given layer is connected to all the other nodes in the next layer. We used a deep feed-forward network because our problem was essentially a prediction, not a classification problem - with defined input features.
The loss function we used was the standard Mean Squared Error (M.S.E.) metric.

M.S.E. is defined as the sum of squares of the error. The error here was the gap between predicted residue depth and known residue depth.
The optimizer we used to train the network was Adam, with its default parameters.

RESULTS & DISCUSSION

We approached the problem in two ways. In the first method, we split our training datasets into 20 different datasets, corresponding to each of the 20 amino acids. We train 20 different neural networks. In the second method, we just use one neural network to train the entire dataset (and predict depths of all 20 types of residues).

We first present the results for Dataset 1 - comprising strictly of those microenvironments known to exist in nature, i.e. fold microenvironments.

(I) As our features for this neural network, we just used “labelled” microenvironments. The input was a (1*20) vector specifying the neighbours of a given central residue.
We trained 20 separate neural networks, each with 2 Hidden Layers. The input layer had 20 nodes, the First Hidden Layer had 128 nodes, the Second Hidden Layer had 64 nodes, and the Output Layer had 1 neuron.

Residue	**Datpoints**	**Accuracy**
ALA	9929	1.0072e-04
ARG	6128	0
ASP	7074	0
ASN	5575	0
CYS	3113	0
GLN	4772	0
GLU	8387	0
GLY	9061	0
HIS	3126	0
ILE	7419	1.3479e-04
LEU	11232	0

LYS	8454	0
MET	2470	0
PHE	5080	0
PRO	5563	0
SER	7640	0
THR	7338	0
TRP	1732	0
TYR	4614	0
VAL	9506	0
All	128204	0

Table 4: Results of the training the Neural Networks, one for each amino acid, using labelled microenvironments, on the True Dataset

We then changed the architecture to 3 Hidden layers, which had 256, 128 and 64 nodes respectively. The results remained unchanged. They remained the same even when we further changed the network to a sparser one with 64, 64, 64 nodes in each of the three hidden layers. We thought this was probably because here we made a clear distinction between each of the 20 amino acids. But chemically amino acids like Arginine and Lysine are fairly similar. It was quite likely that microenvironments comprising 6 Arginines or 6 Lysines would be extremely similar chemically (given a specific central residue) and lead to similar depths. However, in the above implementation we treated them as entirely different.

(II) For the subsequent networks, we did not distinguish between amino acids, and trained the neural network on the whole dataset. This was because, as we observed above, there were very few representatives for rare amino acids. Also, as we saw from the heatmaps in the ‘Methodology’ section; microenvironment by itself was not

a good indicator of residue depth. So we trained one neural network, and implemented it with different combinations of all our input features.

We label the features as follows -
(a) 6 kd Values characterizing microenvironment
(b) 2 pKa values of sequential neighbours
(c) 2 pKc values of sequential neighbours
(d) pKa value of central residue
(e) pKc value of central residue
(f) Molar Mass of central residue
(g) Number density

The notation we will be using to describe the neural network architecture is (#neurons HL1, #neurons HL2, …,1). So, (64, 64, 64) corresponds to a neural network with 3 Hidden Layers and one neuron in the output layer. The number of neurons in the input will depend on the features being used. If a different activation function is used, it will be mentioned. Otherwise, we use ReLu activation.

Features	**Architecture**	**Accuracy**
All	(64)	0.0035
All	(128, 64)	0.0035
All	(64, 64); sigmoid	0.0037
All	(64, 64, 64)	0.0035
All	(64, 64, 64); sigmoid	0.0034
All	(512, 245, 128)	0.0034
All	(64, 64, 64)	0.0034
(a)	(64, 64, 64)	0.0019
(a)	(64, 64)	0.0019
(a)	(64)	2.34 * (10^-5)

(a), (g)	(64, 64, 64)	0.0036
(a), (b), (c) , (d)	(64, 64, 64)	0.0034
(a), (b), (c) , (d)	(64, 64)	0.0035
(b), (c)	(64)	2.34 * (10^-5)
(b)	(64)	2.34 * (10^-5)
(c)	(64)	2.34 * (10^-5)
(b)	(64)	2.34 * (10^-5)
(c)	(64)	2.34 * (10^-5)

Table 5: Results of the training the Neural Networks, on the entire True Dataset, with different combinations of different features

All the above implementations yielded exceedingly poor results. Most basic neural networks generally start with an accuracy of 30-40%. Here the accuracy barely touched 1%. This implied one of the two things - either we had far too few data points, considering the complexity of the prediction or that we simply had features that were not strongly correlated to the target variable. If it was the second case which was true, that would mean we could not; using only the features extracted from a query sequence, predict residue depth.

In order to see if a larger amount of training data would give us better results we used our second dataset, this time consisting of threaded microenvironments and other features intrinsic to each amino acid. This dataset had approximately 1.3 crore datapoints. Even splitting it into 20 different clusters would have sufficient representatives for all amino acids.

(I) Trained 20 separate neural networks. The input feature used was just labelled microenvironments i.e. a (1*20) vector characterizing the microenvironment.

Residue	Datpoints	Accuracy
ALA	87077	0
ARG	570841	0
ASP	531659	0
ASN	679841	0
CYS	403785	0
GLN	452907	0
GLU	825760	0
GLY	878505	0
HIS	275143	0
ILE	617536	0
LEU	924399	0
LYS	924399	0
MET	234345	0
PHE	435104	0
PRO	539700	0
SER	729031	0
THR	671281	0
TRP	159961	0
TYR	385832	0
VAL	809281	0
All		0

Table 6: Results of the training the Neural Networks, one for each amino acid, using labelled microenvironments, on the Threaded Dataset

(II) We then trained a single neural network, using combinations of different features and different architectures on the threaded dataset [Table 7]. The input features correspond to the same labels as mentioned before.

Features	**Architecture**	**Accuracy**
All	(64)	0
All	(128, 64)	0
All	(64, 64); sigmoid	0
All	(64, 64, 64)	0
All	(64, 64, 64); sigmoid	0
All	(512, 245, 128)	0
All	(64, 64, 64)	0
(a)	(64, 64, 64)	0
(a)	(64, 64)	0
(a)	(64)	0
(a), (g)	(64, 64, 64)	0
(a), (b), (c) , (d)	(64, 64, 64)	0
(a), (b), (c) , (d)	(64, 64)	0
(b), (c)	(64)	0
(b)	(64)	0
(c)	(64)	0
(b)	(64)	0
(c)	(64)	0

Table 7: Results of the training the Neural Networks, on the entire Threaded Dataset, with different combinations of different features

As we can see, the results for all the models that we trained, had zero (~ 10^(-3) - 10^(-5)) accuracy. This result implies that if we were to randomly assign a depth value between 0-20 Å (binned at 0.5 Å), to each residue in the query sequence, even that algorithm would work better than the current neural network. For a deep feedforward neural network, predicting a continuous target variable, many thousands of datapoints should have sufficed for training. Yet, whichever dataset we used, we got extremely poor results.

The results of Dataset 2 could still be justified, by saying that there were a lot of “false positives” in the dataset - a lot of microenvironments were possible for the residue to exist in; but only few were found in nature, the others never occured in nature and hence the neural network performed poorly (mismatch between input features and target variable, depth). But even the results for Dataset 1 - which consisted of only microenvironments known to exist, “true positives”, the results were not any better.

The only explanation could be that the input features we used were very weakly correlated to the final residue depths. Indeed, when we studied residue microenvironment and depth correlation, the R^2 value were of the order 10^(-2) - showing poor dependence of residue depth on its environment. We used a few other features, which were inherent characteristics of that residue (molar mass, pKa, pKc values) - yet these by themselves did not significantly impact depth, as proven in our previous studies (where we had predicted depth using probabilities and background depth profiles). The sequential neighbours and number density, also proved to be irrelevant to predicting depth.

Throughout this exercise, we neglected two factors - the energy and structure constraints of the resulting protein fold (being threaded into). We had no way of finding these out, given only the query sequence. We hence conclude that, purely on basis of query sequence alone, we cannot predict the residue depths of that sequence and so deduce its protein fold. External energy constraints or structural

information would be required to do the same.

REFERENCES

1. Kuan Pern Tan, Thanh Binh Nguyen, Siddharth Patel, Raghavan Varadarajan and M. S. Madhusudhan (July 2013). “Depth: a web server to compute depth, cavity sizes, detect potential small-molecule ligand-binding cavities and predict the pKa of ionizable residues in proteins”. *Nucl. Acids Res.* 41 (W1): W314-W321. doi: 10.1093/nar/gkt503

2. Kuan Pern Tan, Raghavan Varadarajan, M. S. Madhusudhan (2011). “DEPTH: a web server to compute depth and predict small-molecule binding cavities in proteins”. *Nucleic Acids Research.* doi: 10.1093/nar/gkr356

3. Suvobrata Chakravarty and Raghavan Varadarajan (1999). “Residue depth: a novel parameter for the analysis of protein structure and stability”. *Structure*. Vol 7, Pages: 723-732.

4. Farheen N., Sen N., Nair S., Tan K.P., Madhusudhan M.S. (2017).“Depth dependent amino acid substitution matrices and their use in predicting deleterious mutations”. *Prog Biophys Mol Biol.* 128:14-23., doi: 10.1016/j.pbiomolbio.2017.02.004.

5. Mizuguchi K., Deane C.M. Blundell, T.L. & Overington J.P., (1998). "HOMSTRAD: a database of protein structure alignments for homologous families". *Protein Science.* 7, 2469-2471.

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

7. The UniProt Consortium (2017)."UniProt: the universal protein knowledgebase". *Nucleic Acids Res*. 45: D158-D169

8. Murzin A. G., Brenner S. E., Hubbard T., Chothia C. (1995). "SCOP: a structural classification of proteins database for the investigation of sequences and structures". *J. Mol. Biol.* 247, 536-540

9. Kyte J., Doolittle R.F. (1982). "A simple method for displaying the hydropathic character of a protein". *J. Mol. Bio.*157, 105-132

10. Nguyen M.N., Tan K.P., Madhusudhan M.S. (2011) . "CLICK - Topology independent comparison of biomolecular 3D structures". *Nucleic Acids Res*., doi:10.1093/nar/gkr393

APPENDIX

```
# Script to compute the Euclidean distance b/w any 2 atoms in a
given PDB file

import os
import sys
from math import pow

# Obtain the pdb file
fle=sys.argv[1]
print fle
residue_pos=[]
with open('%s'%fle) as f:
# The following snippet extracts the relevant positions of amino
acids in the sequence ( there is a lot more information in the pdb
apart from this )
    for line in f.readlines():
        if str(line[:4])=="ATOM":
            pos=int(line[23:26].strip())
            if pos not in residue_pos:
                residue_pos.append(pos)

# FOR loop iterates over all positions along the chain
for rno in range(0, len(residue_pos)):
    central_pos=residue_pos[rno]
    print central_pos
    env=[]
    pos_chk=[]
    with open('%s'%fle) as f:
        lines=f.readlines()
        f.close()
        for line1 in lines:
            if "ATOM" in line1:
                if int(line1[23:26].strip()) == central_pos:
                    # Extract central residue
                    central_residue=line1[17:20]

                    #Obtain its co-ordinates
                    xcen=float(line1[31:38].strip())
                    ycen=float(line1[39:46].strip())
```

```
                    zcen=float(line1[47:54].strip())


                    for line2 in lines:
                        if "ATOM" in line2:
                            # Extract other amino acids, excepting
central residue
                            ngh_pos=int(line2[23:26].strip())
                            # IF loop to avoid picking the central
residue itself ( "ATOM" lines have multiple entries for a single
residue )
                            if central_pos != ngh_pos:
                                # Extract neighbour coordinates
                                xngh=float(line2[31:38].strip())
                                yngh=float(line2[39:46].strip())
                                zngh=float(line2[47:54].strip())

                                # Compute distance netween central
residue and the other residue

dist=float(pow(float(pow((xcen-xngh),2)+pow((ycen-yngh),2)+pow((zc
en-zngh),2)),0.5))

                                # Distance cutoff
                                if dist<= float(5.00):
                                    residue=line2[17:20].strip()
                                    # IF loop to prevent same
residue to feature in the microenvironment more than once
                                    if ngh_pos not in pos_chk:
                                        pos_chk.append(ngh_pos)
                                        env.append(residue)


    # Write data to a new text file
    with open('%s.txt'%fle[:5], "a") as o:
        o.write('%s\t'%central_residue)
        o.write('%s\t'%env)
        o.write('%s\n'%pos_chk)
```

Script 1: To find out the neighbours of a residue within 5Å

```
# Script to find out the different types of microenvironments a
specific residue can occupy - and the depths associated with each
kind of microenvironment
# Script for ALA

import sys
import re
import linecache
import numpy as np

fle=sys.argv[1]

# Load dictionary of kD scale. Use this scale to determine the
chemical nature of the ALA microenvironment

kd_scale={'ILE':'4.5', 'VAL':'4.2', 'LEU':'3.8', 'PHE':'2.8',
'CYS':'2.5', 'MET':'1.9', 'ALA':'1.8', 'GLY':'-0.4', 'THR':'-0.7',
'SER':'-0.8', 'TRP':'-0.9', 'TYR':'-1.3', 'PRO':'-1.6',
'HIS':'-3.2', 'GLU':'-3.5', 'GLN':'-3.5', 'ASP':'-3.5',
'ASN':'-3.5', 'LYS':'-3.9', 'ARG':'-4.5'}

print(fle)
pdb=fle[:5]
#print pdb
with open('ALA_env.txt', "a") as o:
    o.write('%s\n'%pdb)

res='ALA'
l_no=0
indices=[]
# Purpose : Obtain indices of all lines that correspond to ALA
microenvironments. Need this list in order to avoid traversing the
entire file and eating up a lot of time
with open("%s"%fle) as f:
    for line_no, line in enumerate(f):
        if 'ALA' in line.partition('\t')[0].strip():
            indices.append(line_no)

# List of which unique ALA microenvs are present
#print track
#print indices
```

```
i=0
for i in range(0,len(indices)):
# Determine file line
    line1=str(linecache.getline('%s'%fle, int(indices[i]+1)))
    pre1=line1.partition('\t')[-1].partition('\t')[0].strip()
    pre2=re.sub("'","", pre1)
    microenv=pre2.strip('][').split(', ')
    #print microenv

    envscore=0
    for j in range(0, len(microenv)):
        key=microenv[j]
    #    print key
        envscore=envscore+float(kd_scale[key])

    finscore=envscore/len(microenv)

    with open("%s-residue.depth"%fle[:5]) as o:
        lines=o.readlines()
        line2=lines[int(indices[i]+1)]
        mark=line2.partition('\t')[0].partition(':')[-1].strip()
        central_res=str(res+":"+str(mark))
        depth=[]
        values=line2.split()
        all_atom=values[2].strip()
        main_chain=values[4].strip()
        side_chain=values[6].strip()
        depth.append(float(all_atom))
        depth.append(float(main_chain))
        depth.append(float(side_chain))

        #print central_res, microenv, depth, finscore

        with open('ALA_env.txt', "a") as o:
            o.write('%s\t'%central_res)
            o.write('%s\t'%microenv)
            o.write('%s\t'%finscore)
            o.write('%s\n'%depth)
```

Script 2: Extract individual microenvironment data for each residue

```
#Script to extract microenvironment and other features
import os
import sys
import numpy as np
import linecache
import csv

# Get input query, compute its length
pre=str(sys.argv[1])
fold=sys.argv[2]
name = pre.split('\t')[0].strip()
query = pre.split('\t')[-1].strip()
print(name, query)
l=len(query)
scan={'A':'0', 'R':'1', 'N':'2', 'D':'3', 'C':'4', 'E':'5',
'Q':'6', 'G':'7', 'H':'8', 'I':'9', 'L':'10', 'K':'11', 'M':'12',
'F':'13', 'P':'14', 'S':'15', 'T':'16', 'W':'17', 'Y':'18',
'V':'19'}

# Input data regarding kd values of AA
kd_scale={'I':'4.5', 'V':'4.2', 'L':'3.8', 'F':'2.8', 'C':'2.5',
'M':'1.9', 'A':'1.8', 'G':'-0.4', 'T':'-0.7', 'S':'-0.8',
'W':'-0.9', 'Y':'-1.3', 'P':'-1.6', 'H':'-3.2', 'E':'-3.5',
'Q':'-3.5', 'D':'-3.5', 'N':'-3.5', 'K':'-3.9', 'R':'-4.5'}

# Input data regarding pKa ( amino and carboxyl ) values of AA
pKa_carboxyl={'I':'2.36', 'V':'2.32', 'L':'2.36', 'F':'1.83',
'C':'1.96', 'M':'2.28', 'A':'2.34', 'G':'2.34', 'T':'2.09',
'S':'2.21', 'W':'2.83', 'Y':'2.2', 'P':'1.99', 'H':'1.82',
'E':'2.19', 'Q':'2.17', 'D':'1.88', 'N':'2.02', 'K':'2.18',
'R':'2.17'}

pKa_amino={'I':'9.6', 'V':'9.62', 'L':'9.6', 'F':'9.13',
'C':'8.18', 'M':'9.21', 'A':'9.69', 'G':'9.6', 'T':'9.10',
'S':'9.15', 'W':'9.39', 'Y':'9.11', 'P':'10.6', 'H':'9.17',
'E':'9.67', 'Q':'9.13', 'D':'9.6', 'N':'8.8', 'K':'8.95',
'R':'9.04'}

# Input data regarding molar mass of AA
Molar_Mass={'I':'137.18', 'V':'117.15', 'L':'131.18',
'F':'165.19', 'C':'121.15', 'M':'149.21', 'A':'89.09',
```

```
'G':'75.07', 'T':'119.12', 'S':'105.09', 'W':'204.23',
'Y':'181.19', 'P':'115.13', 'H':'155.16', 'E':'147.13',
'Q':'146.15', 'D':'133.10', 'N':'132.12', 'K':'146.19',
'R':'174.20'}

# Start Threading through all single-domain templates.
# Multiple models per template possible, if
len(query)<len(template)

os.chdir('threads')
template=np.loadtxt('%sthread.txt'%fold, delimiter=',')
os.chdir('..')
# Obtain length of template
i=np.size(template,axis=0)
if int(l) <= int(i):
    os.chdir('neigh')
    with open('%sfin.txt'%fold) as f:
        lines=f.readlines()
# Read characters from input query, and thread them through all
possible positions in one go
        for char in range(1,l-1):
            permissible_pos=i-char
            residue=query[char]

# Variables a,b determine permissible positions occupied during
threading
            a=min(l-char,permissible_pos)
            b=max(l-char,permissible_pos)
            for line in lines:

idx=line.partition('\t')[0].partition(',')[-1].partition(',')[-1]
                if int(idx) in range(a,b+1):
# Obtain microenvironment

neighbours=line.partition('\t')[-1].partition('\t')[0].strip()
                    env=neighbours.strip('][').split(', ')

distances=line.partition('\t')[-1].partition('\t')[-1].strip('][')
.split(', ')
                    indices=sorted(range(len(distances)),
key=lambda k: distances[k])
```

```
                    fin_env1= list(map(env.__getitem__, indices))
                    fin_env2=[]
# Below loop transforms template indices to corresponding query
indices
                    for ele in fin_env1:
                        if int(ele) in
range(int(int(idx)-int(l-char)+1),int(int(idx)+int(char)+1)):
                                fin_env2.append(ele)
                    no_density=len(fin_env2)
                    fin=[]
                    for res in fin_env2:
                        diff=int(int(res)-int(idx))
                        res= int(char)-int(diff)
                        fin.append(res)

# Determine 6 closest neighbours, obtain the other parameters.
Also generate labelled data, stored in array "matrix"
                    if no_density>= 6:
                        matrix=np.zeros(shape=(1,20))
                        amino_acid1=query[int(fin[0])]
                        kd1=kd_scale[amino_acid1]

matrix[0,int(scan[amino_acid1])]=matrix[0,int(scan[amino_acid1])]+
int(1)
                        amino_acid2=query[int(fin[1])]
                        kd2=kd_scale[amino_acid2]

matrix[0,int(scan[amino_acid2])]=matrix[0,int(scan[amino_acid2])]+
int(1)
                        amino_acid3=query[int(fin[2])]
                        kd3=kd_scale[amino_acid3]

matrix[0,int(scan[amino_acid3])]=matrix[0,int(scan[amino_acid3])]+
int(1)
                        amino_acid4=query[int(fin[3])]
                        kd4=kd_scale[amino_acid4]

matrix[0,int(scan[amino_acid4])]=matrix[0,int(scan[amino_acid4])]+
int(1)
                        amino_acid5=query[int(fin[4])]
                        kd5=kd_scale[amino_acid5]
```

```
matrix[0,int(scan[amino_acid5])]=matrix[0,int(scan[amino_acid5])]+
int(1)
                        amino_acid6=query[int(fin[5])]
                        kd6=kd_scale[amino_acid6]

matrix[0,int(scan[amino_acid6])]=matrix[0,int(scan[amino_acid6])]+
int(1)
                        seq1=query[char-1]
                        seq2=query[char+1]
                        pK11=pKa_carboxyl[seq1]
                        pK12=pKa_amino[seq1]
                        pK21=pKa_carboxyl[seq2]
                        pK22=pKa_amino[seq2]
                        mm=Molar_Mass[residue]
                        pK_c=pKa_carboxyl[residue]
                        pK_a=pKa_amino[residue]
                        depth_aa=round(template[int(idx)-1,0],2)
                        depth_mc=round(template[int(idx)-1,1],2)

# Obtain outputs, one file for labelled data, the other file for
all features
                        with open('input.csv', 'a') as f:
                            writer = csv.writer(f)
                            writer.writerow([name, fold, char,
kd1, kd2, kd3, kd4, kd5, kd6, pK11, pK12, pK21, pK22, mm, pK_c,
pK_a, no_density, depth_aa, depth_mc])
                        with open('input_labelled.csv', 'a') as o:
                            writer = csv.writer(o)
                            writer.writerow([name, fold, char,
matrix[0,0], matrix[0,1], matrix[0,2], matrix[0,3], matrix[0,4],
matrix[0,5], matrix[0,6], matrix[0,7], matrix[0,8], matrix[0,9],
matrix[0,10], matrix[0,11], matrix[0,12], matrix[0,13],
matrix[0,14], matrix[0,15], matrix[0,16], matrix[0,17],
matrix[0,18], matrix[0,19],  mm, depth_aa, depth_mc])

os.chdir('..')
```

Script 3: Extract features by threading the query through folds

```
from google.colab import drive
drive.mount('/content/drive')

# The NN script
import tensorflow
from sklearn.preprocessing import MinMaxScaler
from matplotlib import pyplot as plt
import pandas as pd
import seaborn as sb


#Read input data, split into features and target vectors
path='/content/drive/My Drive/Thesis_TB/input.csv'
training_data=pd.read_csv(path)
print(training_data.head())

#Scale the features
#scaler = MinMaxScaler(feature_range=(0, 1))
#scaled_train = scaler.fit_transform(training_data)

#multiplied_by = scaler.scale_[14]
#added = scaler.min_[14]
#scaled_train_df = pd.DataFrame(scaled_train,
columns=training_data.columns.values)

X_train=training_data.drop([training_data.columns[0],
training_data.columns[1],training_data.columns[2],
training_data.columns[16], training_data.columns[17]], axis=1)
print(X_train.head())
#X1_train=training_data[[training_data.columns[0],
training_data.columns[1], training_data.columns[2],
training_data.columns[3], training_data.columns[4],
training_data.columns[5]]]
#print(X1_train.head())


#Molar_Mass={'I':'137.18', 'V':'117.15', 'L':'131.18',
'F':'165.19', 'C':'121.15',
#            'M':'149.21', 'A':'89.09', 'G':'75.07', 'T':'119.12',
'S':'105.09',
#            'W':'204.23', 'Y':'181.19', 'P':'115.13',
```

```
'H':'155.16', 'E':'147.13', 'Q':'146.15',
#            'D':'133.10', 'N':'132.12', 'K':'146.19',
'R':'174.20'}

#RES=training_data.loc[training_data[training_data.columns[23]] ==
146.19]
#print(len(RES.index))

#RES_train=RES.drop([RES.columns[0], RES.columns[1],
RES.columns[2], RES.columns[23], RES.columns[24],
RES.columns[25]], axis=1)
Y_train=training_data[[training_data.columns[16]]]
print(Y_train.head())


from tensorflow.keras.models import Sequentia'l
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Dense

model = Sequential([
   Dense(128, activation='relu', input_shape=(14,)),
   Dense(64, activation='relu'),
   Dense(1, activation='linear'),
])

model.compile(optimizer='adam', loss='mean_squared_error',
metrics=['accuracy'])

hist=model.fit(X_train, Y_train, epochs=20)
```

Script 4: The Neural Network Script
