CAFA 5 Protein Function Prediction

Name - Krishnansh Agarwal
Roll No. - 210532
Department - EE
Email - krishnansh21@iitk.ac.in
Kaggle ID - krishnanshagarwal05
Leaderboard Rank - 304

---

1. Introduction to the problem

CAFA (Critical Assessment of Protein Function Annotation) is an experiment, an ongoing, global, community-driven effort to evaluate and improve the computational annotation of protein function. The goal of the competition is to train ML models that can correctly predict the protein functionality from the given protein sequences and some data related to them.

A protein's functionalities are described by attaching certain annotations to it, picked up from a large and organised dataset of such annotations. This data set is called the Gene Ontology (GO) which is a large resource developed in bioinformatics and is easy for performing computational analysis. It describes genes with respect to three aspects:

- **Molecular functions (MFO)**: Describe the molecular activities of the gene
- **Cellular Components (CCO)**: locations where gene perform their process
- **Biological Process (BPO)**: These refer to larger processes than the activities performed by individual genes and are cause by the activities of multiple genes

The data is represented by 3 acyclic directed graphs where each node contains GO terms. See the Gene Ontology Overview for more details.

The protein sequences are provided in FASTA format files. All of these were retrieved from the UniProt data set

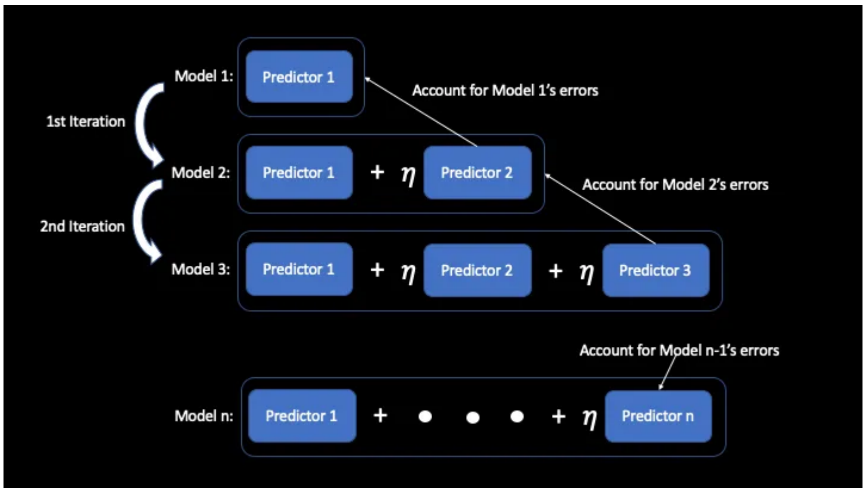
2. Literature Review

BioPython: Biopython is a Python library specifically designed for bioinformatics, which involves the analysis and manipulation of biological data. It provides a wide range of functionalities for tasks such as sequence analysis, protein structure analysis, genomic data processing, and more. In the CAFA challenge the protein sequence data is provided in the FASTA format which is a text-based format for representing either nucleotide sequences or amino acid (protein) sequences, in which nucleotides or amino acids are represented using single-letter codes. BioPython provides parsers that help in easily extracting info on sequences out of these files. It also provides sequence alignment and matching algorithm's implementation, such as BLAST and Clustalw alignment program. It also provides various other tools for performing common operations on sequences, such as translation, transcription and weight calculations.
Documentation of BioPython: https://biopython.org/wiki/Documentation

BLAST (Basic Local Alignment Search Tool): BLAST is a widely used algorithm that compares a query sequence against a database of known protein sequences to find similar sequences. It generates a similarity score based on sequence alignment and statistical significance. It uses some heuristic algorithm to give scores to rapidly search for local alignment matching rather than doing an exhaustive search.
Blast gives a non-ML naive approach for protein function prediction which is the current baseline in CAFA.
BLAST tool - https://blast.ncbi.nlm.nih.gov/Blast.cgi
Insight to Blast -
https://www.nature.com/scitable/topicpage/basic-local-alignment-search-tool-blast-29096/
Diamond is a sequence aligner implementation through which we can apply the BLAST algorithm. It is an optimised implementation that greatly speeds up pairwise alignment of proteins and translated DNA at 100x-10,000x speed of BLAST.
Github - https://github.com/bbuchfink/diamond
A kaggle competition implementation for BLAST -
https://www.kaggle.com/code/geraseva/diamond

Multilabel Classification using XGBoost: The CAFA problem is a multilabel classification problem where every protein sequence is assigned multiple GO

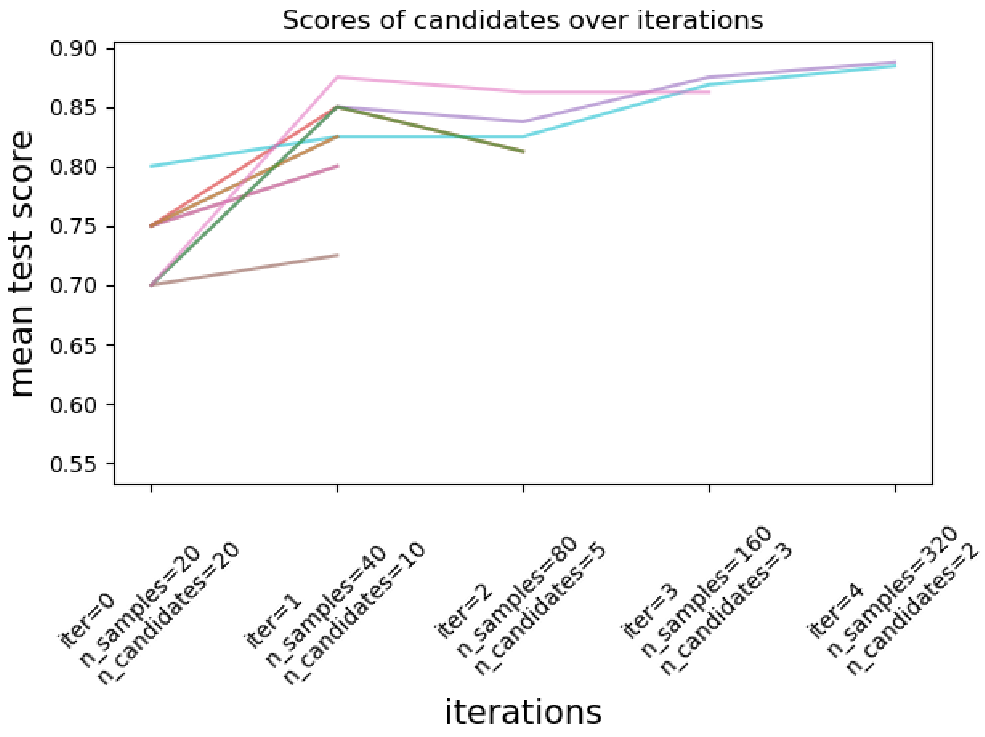
terms acting as the labels. We chose to train decision trees using the XGBoost algorithm for this classification problem. The labels, that are GO terms, are interrelated through a hierarchical tree. When a GO term is associated with a protein then so are its ancestors. This might be captured by decision trees where the higher splits divide the protein into more general classes while going down the tree they become more specific. The XGBoost model is a gradient based optimised algorithm for forming an ensemble of decision trees. It improves the splits made by the current tree at different levels by calculating the gradient of the loss function.
The following image encaptures gradient based learning of subsequent decision trees


Model 1:
Predictor 1
Account for Model 1's errors
1st Iteration
Model 2:
Predictor 1 + η Predictor 2
Account for Model 2's errors
2nd Iteration
Model 3:
Predictor 1 + η Predictor 2 + η Predictor 3
Account for Model n-1's errors
Model n:
Predictor 1 + ● ● ● + η Predictor n

A good intuition on this is provided by - https://towardsdatascience.com/the-intuition-behind-gradient-boosting-xgboost-6d5eac844920
The input to the XGBoost model cannot be a protein sequence obtained from the FASTA file. We need to obtain feature vectors of the sequences. T5 embeddings, Probert embeddings and ESM embeddings are some of the popular embeddings in the CAFA competition that were better than the SOTA embeddings. We used the T5 embeddings for the XGBoost model. Grandmaster Sergei Fironov has the credit for generating them. Dataset for obtaining them https://www.kaggle.com/datasets/sergeifironov/t5embeds

HyperBandSearchCV: A brute force approach for hyperparameter tuning is by listing out a set of values for all hyperparameters and then training and evaluating the model on all the combinations. This can be done using scikit learn's GridSearchCV**.**
But when the search space is large then this will take a lot of time and a model with a lot of weights makes it worse. The hyperBandSearchCV approach speeds this process up with almost no loss in the precision of picking out the best params. This approach is similar to another approach called the halvingGridSearchCV. These methods first train all the possible combinations(called candidates) for a small subset of dataset and small

number of epochs. Then they discard half of the candidates based on performance and in each successive iteration they double the dataset and epochs till finally we are left with only one candidate. The following paper introduced hyperBandSearchCV.
The following image shows how the algorithm eliminates the poor performing candidates on subsequent iterations



https://www.jmlr.org/papers/volume18/16-558/16-558.pdf
And this paper has been implemented and open sourced at
https://github.com/thuijskens/scikit-hyperband
A good explanation of hyperparameter tuning is give at
https://scikit-learn.org/stable/modules/grid_search.html#searching-for-optimal-parameters-with-successive-halving
The following is also a good blog for hyperparameter tuning of XGBoost
https://towardsdatascience.com/binary-classification-xgboost-hyperparameter-tuning-scenarios-by-non-exhaustive-grid-search-and-c261f4ce098d
The following gives good intuition about different hyperparameters of XGBoost
https://www.kaggle.com/code/prashant111/a-guide-on-xgboost-hyperparameters-tuning

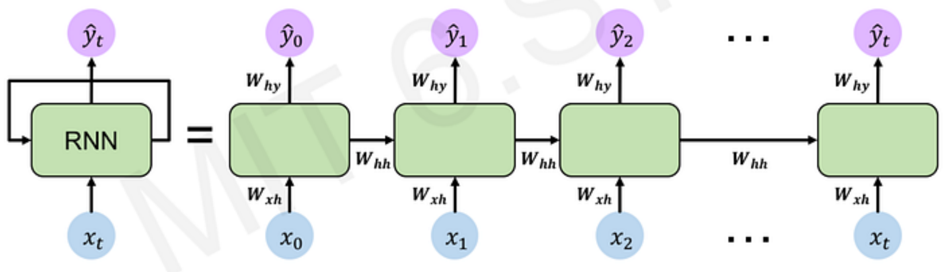
Feature Extraction and Dimensionality Reduction: Feature Extraction aims to reduce the number of features in a dataset by creating new features from the existing ones (and then discarding the original features). These new reduced sets of features should then be able to summarise most of the information contained in the original set of features. This not only helps in better visualising and understanding large dimensional data but also helps in speeding up our model and sometimes prevents overfitting. Some of the methods used for feature extraction are:

a) **PCA(Principal Component Analysis):** Principal Component Analysis is an unsupervised linear transformation technique used for feature extraction and dimensionality reduction. It identifies patterns in data based on the correlations between different data points. More technically, it aims to find the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one. Basically we choose the top k eigen vectors of the covariance matrix and then project our data point to reduced space formed by these k vectors. In the reduced space, as is expected in a feature space, the same kind of data will be clustered together.
b) **Autoencoders:** Auto encoders provide a popular approach for reducing the dimensions of the data. An autoencoder is an unsupervised multilayered ANN consisting of three parts: encoder, code, decoder. This ANN is trained on the input data and has to give the same input data as the output. The encoder layer is expected to map the input data on a smaller space (of the code) called the latent space and the decoder learns to reconstruct the data from this latent space. After the training we can remove the decoder part and the encoder part performs dimensionality reduction for us.

The following are some good blogs for learning about PCA and auto encoders:
https://towardsdatascience.com/principal-component-analysis-for-dimensionality-reduction-115a3d157bad
https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

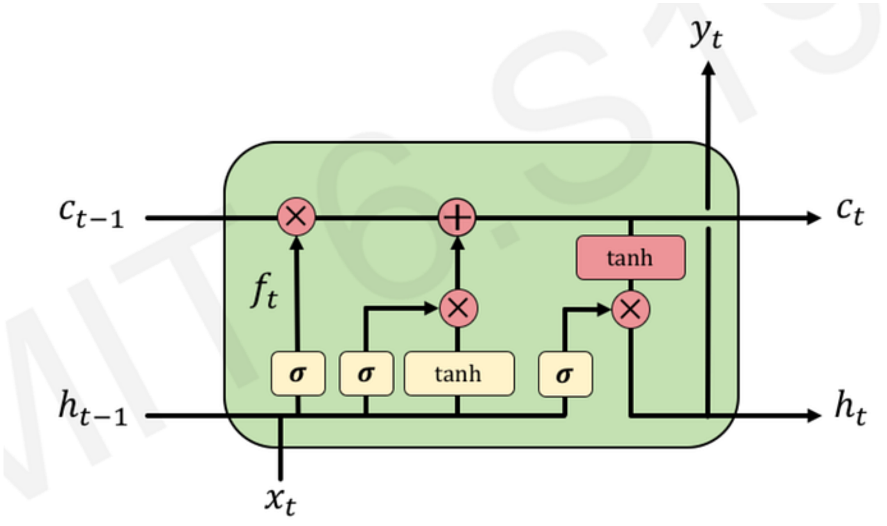
RNN, LSTM and BiLSTM: Since the problem at hand involves sequential data, RNNs and LSTM seem to be the obvious choice for this task.

RNNs keep a hidden state which basically captures the data from the sequence it has seen so far. They keep on updating this hidden state and use it to predict output from the next neuron in the sequence. All of the neurons share the same weights. Parameter sharing enables the network to generalise to different sequence lengths. But this leads to the problem of exploding and vanishing gradients because the gradients are also the same and on multiplication to calculate the net gradient they increase or decrease exponentially.



LSTM proposes a solution to these problems. They have a special cell state called Ct, where information flows and three special gates: the forget gate, the input gate, and the output gate. The forget gate decides what information is not important and can be forgotten, input gate decides what info should be stored and the output gate decides what information should be shown as the output at any point of the sequence processing. The LSTMs are unidirectional, that is info flows only in one direction. They can only have information of the past sequence and not the future.

Image of SIngle LSTM unit:



Multiple of these units are chained one after the other

BiLSTM are nothing but two LSTMs working in parallel and storing information from both directions. So their state is updated not only from past data but also from the future sequence. In protein sequence alignment the sequence lying to the left and to the right are equivalent and should be equally considered when updating any neuron, so BiLSTMs are a better choice than LSTMs.

The following is a good blog for learning intuition and implementation of RNNs and LSTMs
https://ai.plainenglish.io/recurrent-neural-networks-for-multilabel-text-classification-tasks-d04c4edd50ae#8e5d

The following repo has some good tutorial on RNNs for sentiment analysis
https://github.com/bentrevett/pytorch-sentiment-analysis

The mentors also took a presentation and shared their ideas on how we can implement these for our specific problem. This is the link to the presentation.

3. Experimentation

A) **Exploratory Data Analysis (EDA) :** The first thing we did was to get familiar with the competition and the data. The training dataset CAFA provides consists of 5 important files -

- *Go-basic.obo*: GO graph data. Each node of the graph contains info on GO terms and relationships with other GO terms.
- *Train_sequences.fasta*: The list of proteins with unique ids, some meta info and sequence.
- *Train_taxonomy.tsv*: It contains the taxonomy ID of proteins
- *Train_term.tsv*: Contains mapping of the protein ids with the GO terms ids.
- *IA.txt:* Information Accretion for each term. This is used to weight precision and recall

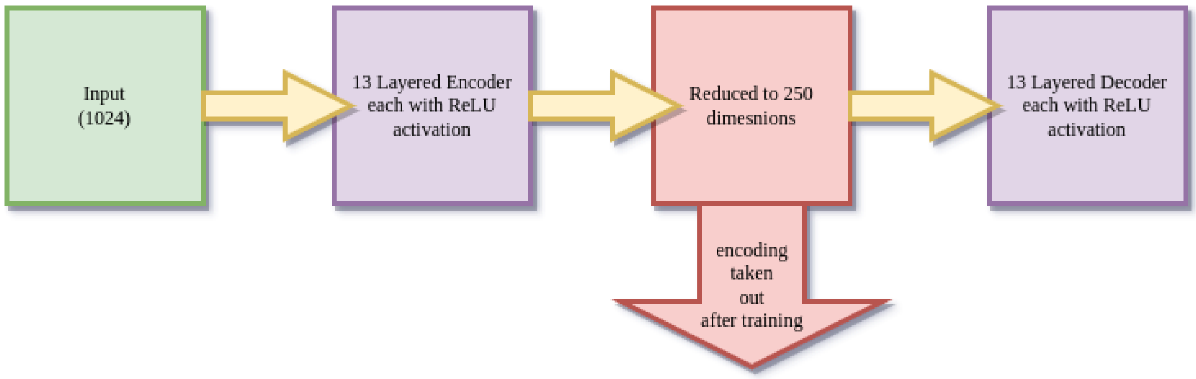
Our notebook for EDA is - https://www.kaggle.com/code/krishnanshagarwal05/notebooke07f311747/settings

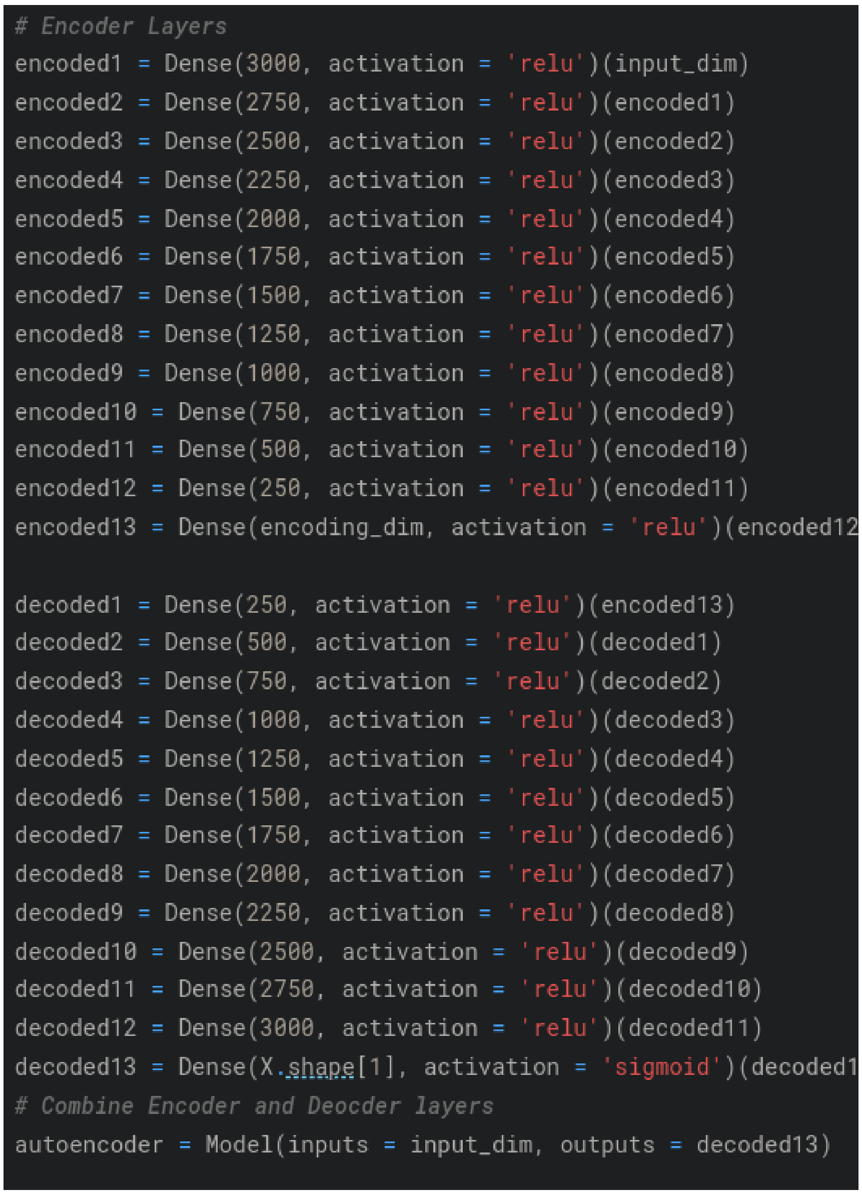
B) **Sequence similarity matching using BLAST algorithm :** This was a naive approach of similarity matching and sequence alignment of protein sequences on the argument that proteins with similar amino acid sequences have similar functions and structures. This similarity matching was implemented using the diamond sequence aligner. The protein fasta file which is provided in the competition was used for forming the database of the sequence aligner on which it will perform BLAST alignment for requested sequence. Then the unseen sequences are matched and top k matched sequences are obtained. We say that the associated GO terms with these sequences also belong to the unseen sequence and the number of times each term is repeated in the matched sequence gives the strength of this term being associated with the original sequence. So for every GO term(GO_i) we say its probability of being associated with original sequence = (# occurrence of GO_i on all matched sequences) / (total number of matched terms).

C) **Multilabel Classification with XGBoost :** The problem is clearly a multilabel classification problem where each protein sequence can be assigned different GO labels. We decided to train an ensemble of decision trees which felt intuitive for this problem. We trained it using the XGBoost method and used the T5 embeddings as was described in literature review. Subsequently, we made changes to enhance its performance. Additionally, we employed dimension reduction techniques and conducted hyperparameter tuning. After

all this, it did not provide great results. But this task helped us in understanding the dataset better.
The XGBClassifier algorithm from the scikit library was employed to construct the model. The train dataset was partitioned into an 80:20 split for training and evaluation purposes, respectively. This preliminary testing was conducted to avoid wastage of submissions. The roc_auc accuracy, F1 and Fmax scores were used to evaluate the model. As there are 40k GO labels, these would have been too many for our XGBoost model, so instead of considering all these labels we chose only the top 1500 most frequently occurring GO terms throughout the train set. This was an approach that was used by many others in the competition.
This is a link to our implementation notebook and this is the link to our evaluation and submission of the same. We used the hyperBandSearchCV and halvingGridSearchCV for hyper parameter tuning which are way faster than GridSearchCV as described in literature review. Since the implementation provided by scikit-learn does not support multi label classification we also had to make our own changes to bring in this support.
This is the notebook where we implemented hyperparameter tuning.

D) **Dimensionality Reduction using autoencoders :** Still the hyperparameter tuning of XGBoost was taking some time to report back the parameters so we used the technique of dimensionality reduction using autoencoders. We further reduced the size of the T5 embeddings that were the input to the model and this time the search time for tuning reduced. This is the notebook where we generate encodings of reduced dimensions and this is the notebook where we hypertune parameters using these reduced dimensions. We use 13 layered encoder and 13 layered decoder for this purpose and finally generated encodings of dimension 200. This helped in improving time but the scores remained almost the same with a non-significant increase. The autoencoder looks like:





```
# Encoder Layers
encoded1 = Dense(3000, activation = 'relu')(input_dim)
encoded2 = Dense(2750, activation = 'relu')(encoded1)
encoded3 = Dense(2500, activation = 'relu')(encoded2)
encoded4 = Dense(2250, activation = 'relu')(encoded3)
encoded5 = Dense(2000, activation = 'relu')(encoded4)
encoded6 = Dense(1750, activation = 'relu')(encoded5)
encoded7 = Dense(1500, activation = 'relu')(encoded6)
encoded8 = Dense(1250, activation = 'relu')(encoded7)
encoded9 = Dense(1000, activation = 'relu')(encoded8)
encoded10 = Dense(750, activation = 'relu')(encoded9)
encoded11 = Dense(500, activation = 'relu')(encoded10)
encoded12 = Dense(250, activation = 'relu')(encoded11)
encoded13 = Dense(encoding_dim, activation = 'relu')(encoded12

decoded1 = Dense(250, activation = 'relu')(encoded13)
decoded2 = Dense(500, activation = 'relu')(decoded1)
decoded3 = Dense(750, activation = 'relu')(decoded2)
decoded4 = Dense(1000, activation = 'relu')(decoded3)
decoded5 = Dense(1250, activation = 'relu')(decoded4)
decoded6 = Dense(1500, activation = 'relu')(decoded5)
decoded7 = Dense(1750, activation = 'relu')(decoded6)
decoded8 = Dense(2000, activation = 'relu')(decoded7)
decoded9 = Dense(2250, activation = 'relu')(decoded8)
decoded10 = Dense(2500, activation = 'relu')(decoded9)
decoded11 = Dense(2750, activation = 'relu')(decoded10)
decoded12 = Dense(3000, activation = 'relu')(decoded11)
decoded13 = Dense(X.shape[1], activation = 'sigmoid')(decoded1
# Combine Encoder and Deocder layers
autoencoder = Model(inputs = input_dim, outputs = decoded13)
```

Also the submissions were made only for one-fourth of the test data, since generating a submission file for all the data required 3 times (~ 35 - 40 GB) more RAM than is available on kaggle(13GB). So this also affected the submission score which was almost directly proportional to the size of the test set.

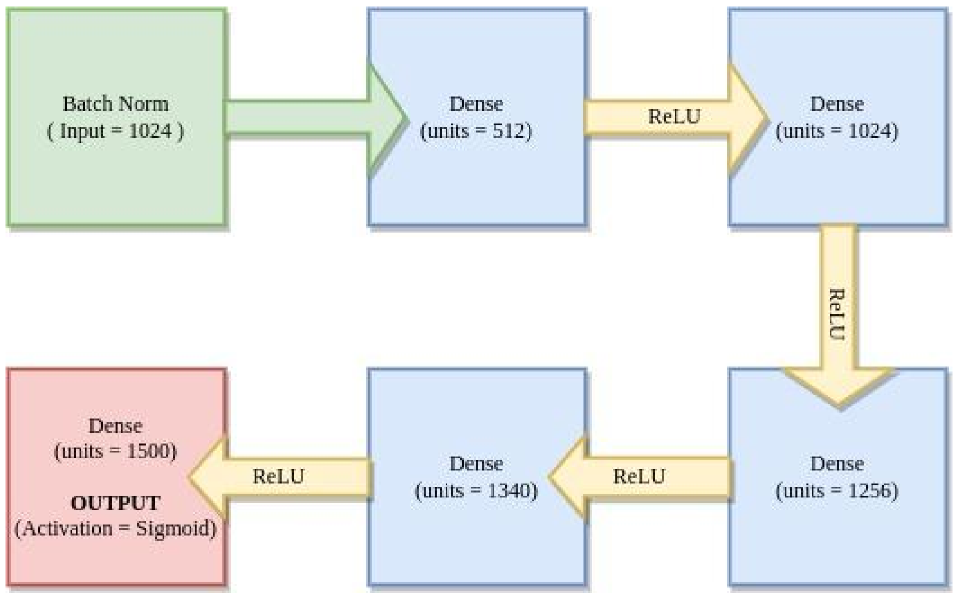
<u>Results Obtained -</u>

Even after spending a lot of effort and implementing all the above things the results for XGBoost were not very charming but the process provided us with this information needed to develop and test potential new approaches to the problem.
The following are the submission accuracies obtained for the XGBoost model:

Milestone	Accuracy (FMax by Submission)
Simple XGBoost with binary output	0.0026
XGBoost with probabilities by Softmax	0.04308
XGBoost after dimension reduction	0.04672
XGBoost with hyperparam tuning	0.05493

Note: These results suggest a potential score of 0.2 for the total dataset of 142,000 data points.

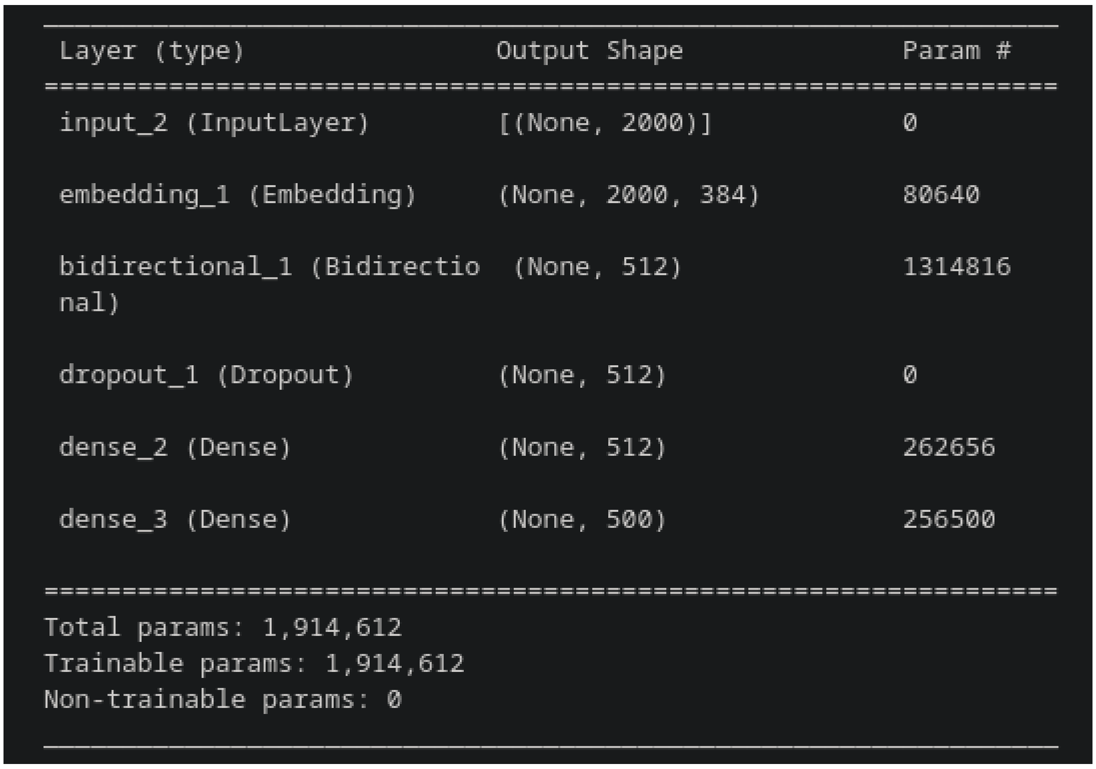
E) **Multi Layered Perceptron :** Next we moved onto deep learning and tried training a multi layered perceptron model. We used the T5 embeddings and the Probert embeddings as the input to our model. Only the top 1500 most frequent GO labels throughout the dataset was considered as was done in the XGBoost model as well. We tried different models by varying layers and units and using different activation and loss functions. Finally we used Adam optimizer along with binary_crossentropy loss. We also used AUC score as well as F1-score to evaluate the models performance on a test set made from splitting train data into train and validation.
Also we used an autoencoder to reduce the dimensions of the T5 embeddings as was done before for the XGBoost model which had shown not so significant changes in the predictions but an increased speed. This is the link where we implemented the MLP model.



Model	F1-Score
MLP + T5	0.40
MLP + ProtBert	0.36

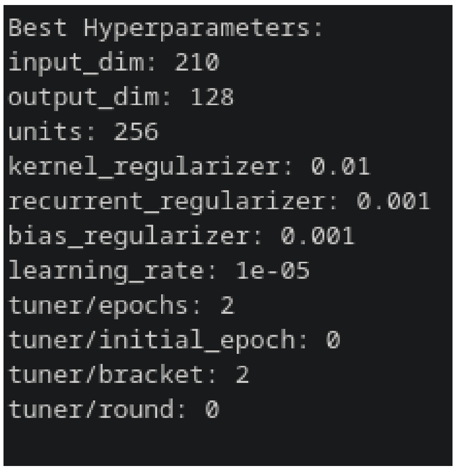
LSTM and BiLSTM : Due to the sequential nature of the dataset, LSTM and BiLSTM seemed a good choice for the problem. So we took references from different blogs and github repo to do our implementation. First we used a simple LSTM layer on which we fed the pretrained T5 and ProBert embeddings. This baseline model did not give good results at all. Then we tried adding some Dense layers in front of the LSTM layer. This was because we learned that LSTM is usually used to capture sequential information and then pass it to a deep network that then uses this info to make predictions. So we took reference from this notebook to add a similar MLP model in front of the LSTM. This worked better after we made some more changes to that implementation. This is our implementation for the LSTM model.
We took reference from the following repository.
https://github.com/jonad/Toxicity_comments/tree/master/packages

Next instead of using the pretrained embedding we decided to pass the sequences directly to the network and put an embedding layer before the LSTM. But since some of the sequences were too big we clipped and padded all sequences value to 1000 - 2000 since this is within where 90% of the sequences lie. This embedding layer would learn the representation of the sequences in a real vector space. Initially we kept the embeddings to 21 with the argument that there were 21 different amino acids in the dataset and so our embeddings layer would learn 21 dimensional vectors, each dimension corresponding to each amino acid. Also we changed the LSTM layer with a BiLSTM layer.
For this implementation we took reference from the following blog which also tries solving a similar problem of protein function prediction
https://towardsdatascience.com/protein-sequence-classification-99c80d0ad2df
We again added some Dense layers in front of the BiLSTM layer. The final output had a sigmoid activation and the binary_crossentropy was used as the loss function. We also added regularisation to the BiLSTM layer and dropout of the neurons as well. This was done so as to prevent overfitting. The model was not doing so good this time. Next we tried to fine tune the model using a manually written code then using the keras tuner which is a keras library for hyper parameter tuning of the deep networks. This also has an implementation of advanced search algorithms such as halving grid search and Bayseian search. After searching for the best hyperparameters for a long time, the model scoring improved but it was not so good and it seemed like it had reached its saturation. This is the notebook where we implemented the above model.
The final model looks like:



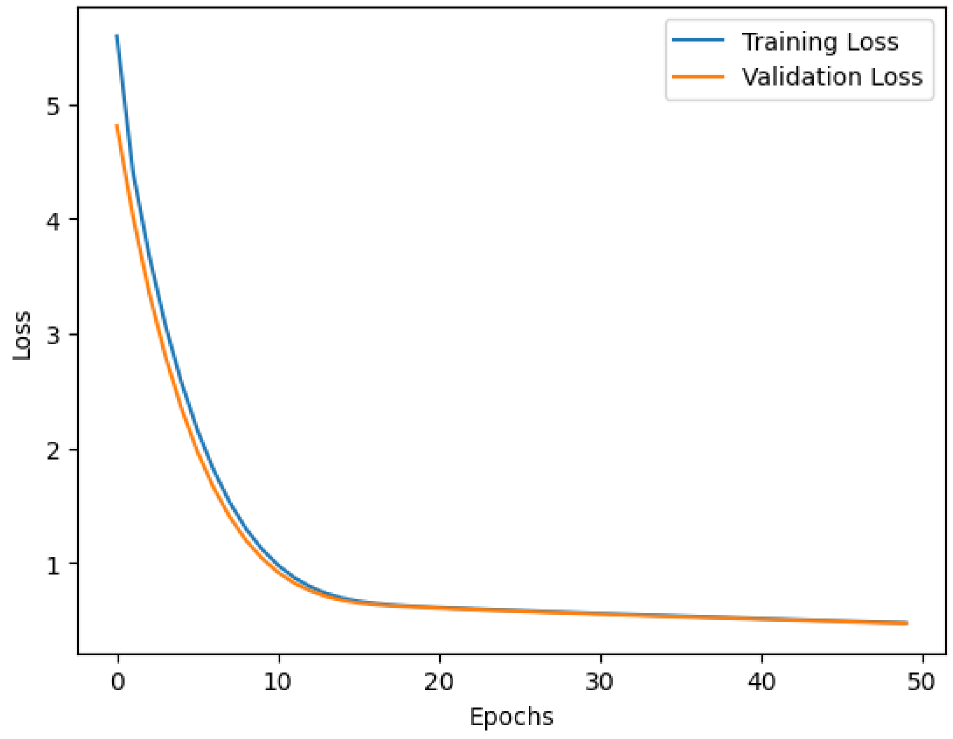
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 2000)]            0

 embedding_1 (Embedding)     (None, 2000, 384)         80640

 bidirectional_1 (Bidirectio  (None, 512)              1314816
 nal)

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 512)               262656

 dense_3 (Dense)             (None, 500)               256500

=================================================================
Total params: 1,914,612
Trainable params: 1,914,612
Non-trainable params: 0
_________________________________________________________________
```

Finally obtained best parameters:



```
Best Hyperparameters:
input_dim: 210
output_dim: 128
units: 256
kernel_regularizer: 0.01
recurrent_regularizer: 0.001
bias_regularizer: 0.001
learning_rate: 1e-05
tuner/epochs: 2
tuner/initial_epoch: 0
tuner/bracket: 2
tuner/round: 0
```

The training loss and validation loss (binary_crossentropy) of this model is



We obtained the following results:

Model	F1-Score
LSTM + ProBert	0.129
BiLSTM with embedding layer	0.116

The score obtained by the LSTM was underwhelming but next we are trying to implement an attention mechanism and GRU which usually promise to prove much better than the LSTM approach.

Finally this is the sheet where we have done all the tasks assigned to us https://docs.google.com/spreadsheets/d/1k1PeMh89COWt8ksnJBRCNRSYzluN9Zhn6BjAfW9iXJA/edit#gid=55360585

And this is the github repository where the code from all the teams is compiled https://github.com/BSBE-IITK/CAFA5-PFP