Report On Differential Dynamic Programming for Structured Prediction and Attention

By Ladji Idrissa FOFANA

March 26, 2023

Abstract

The article proposes a new approach to using dynamic programming (DP) algorithms in neural networks to solve structured prediction tasks. Traditional attention mechanisms can be computationally expensive, especially for long input sequences. To address this issue, the authors propose to smooth the max operator in DP recursion with a strongly convex regularizer that allows them to relax both the optimal value and solution of combinatorial problems and turn them into differentiable operators. The paper provides theoretical insight on how these smoothed DP operators can be applied through inference of graphical models and two particular instantiations are showcased (Viterbi algorithm for sequence prediction and DTW algorithm for time-series alignment) experimentally evaluated on several NLP tasks.

1 Summary of the Related Work

1.1 DP Differentiable Operators using Max-Smoothed Semirings

This paper presents a new approach for incorporating dynamic programming (DP) layers into neural networks using differentiable operators. The authors propose a framework called DPΩ, which is a smoothed relaxation of the original DP algorithm. Instead of using the traditional semiring operations, they introduce the smoothed max operator, which is defined by a strongly convex regularizer. The $\max_\Omega$ operator is differentiable everywhere with a Lipschitz continuous gradient, and the gradient of $\max_\Omega$ can be found using Danskin's theorem. The paper focuses on two regularizers, the negentropy and the squared ℓ_2 norm, which satisfy all the properties of the $\max_\Omega$ operator. The authors explore the use of negative entropy for Ω, which allows them to recover existing CRF-based works from a different perspective. They also investigate the use of squared ℓ_2 norm for Ω, which leads to the development of new algorithms whose expected solution is sparse. The authors provide a method for backpropagating gradients through DPΩ and ∇DPΩ, which allows for the creation of differentiable DP layers that can be incorporated into neural networks trained end-to-end. The gradient of DPΩ, denoted as ∇DPΩ, is equal to the expected trajectory of a certain random walk, and can be used as a sound relaxation to the original dynamic program's solution. Overall, this paper presents a comprehensive introduction to DP algorithms and a novel approach for using DP layers in neural networks.

1.1.1 Dynamic Programic Problem Formulation

The problem is set up on a directed acyclic graph (DAG) $G = (V, E)$ with unique source v and sink v nodes. The nodes in V are assigned weights $\theta \in \mathbb{R}^{V \times V}$, with $\theta_{v_v=1}$ and $\theta_{uv} = -\infty$ if $uv \notin E$.

The goal is to find the path with the highest score among all paths $v_* \to v^*$. To solve this, each node $v \in V$ is identified with its number $i = 1, 2, \ldots, m$ in topological order. The algorithm proceeds as follows:

- Initialize $F_1(\theta) = 0$
- For $j = 2, 3, \ldots, m$, compute $F_j(\theta)$ we compute $F_j(\theta) \leftarrow \max i : j \in G_i \{\theta ij + F_i(\theta)\}$
- The optimal path can be found by backtracking through $(F_j(\theta))_{j=1}^m$.
- The optimal path is $v_* \to v_{i_1} \to v_{i_2} \to \cdots \to v_{i_{k-1}} \to v^*$, where $i_1, i_2, \ldots, i_{k-1}$ are the indices of the nodes in the optimal path in topological order.

The key idea is to compute the optimal solution to a subproblem and then use this solution to solve the original problem. The subproblems are solved in a bottom-up manner, starting from the smallest subproblem and building up to the largest. The solution to the original problem is obtained by combining the solutions to the subproblems.

In 1952, Bellman showed that Dynamic Programming Problems can be formulated as linear problems. Specifically, he demonstrated that the value obtained from solving the DP problem, denoted by $DP(\theta)$, is equivalent to solving a linear program, denoted by $LP(\theta)$:

$$LP(\theta) = \max \pi \sum_{u \in V} \sum_{v \in V} \theta_{uv} 1_{uv \in \pi} = \max_{\mathbf{y} \in \mathcal{Y}} \langle \theta, \mathbf{y} \rangle$$

where $\mathcal{Y}$ is the set of binary matrices representing all paths from the source node v to the sink node v. However, there are some limitations to using $LP(\theta)$ as an alternative to solving DP problems:

- $LP(\theta)$ may not be differentiable unless its solution is unique.
- The optimal solution $\mathbf{y}^*(\theta)$ obtained from $LP(\theta)$ may be a discontinuous map.

Formally, the function can be written as $\max : \mathbb{R}^d \longrightarrow \mathbb{R}$, where $\theta \longmapsto \max_{i=1}^{d} \theta_i$. This can also be written as the supremum of the inner product between the vector θ and a vector x in the unit simplex in $\mathbb{R}^d$, denoted as Δ^d. The unit simplex Δ^d is defined as the set of non-negative vectors x whose components sum to 1, i.e., $\Delta^d = x : |x|_1 = 1, x \geq 0$.

The **max** function is used in many optimization problems, as it often arises as a subproblem in optimization formulations. It is worth noting that the function is differentiable almost everywhere except on negligible sets, but its optimal solution is generally non-differentiable.

1.1.2 Smoothness of the maxima

The maximum function $\max_\Omega$ has several useful properties. It is increasing, meaning that if $\theta_1 \leq \theta_2$, then $\max_\Omega(\theta_1) \leq \max_\Omega(\theta_2)$, and it is also translation invariant, which means that $\max_\Omega(\mathbf{1}c + \theta) = c + \max_\Omega(\theta)$ for any $c \in \mathbb{R}$. Another important property of $\max_\Omega$ is that it is permutation invariant, i.e., $\max_\Omega(\pi\theta) = \max_\Omega(\theta)$ for any permutation P with $\Omega \circ P = \Omega$. If any element of the input vector θ is equal to $-\infty$, then the corresponding partial derivative of $\max_\Omega(\theta)$ is equal to 0.The function $\max_\Omega$ takes a vector θ in $\mathbb{R}^d$ as input and returns the maximum element of θ subject to a constraint defined by the regularizer Ω. Specifically, the constraint requires that the solution x lies in the unit simplex Δ^d, which is the set of non-negative vectors whose elements sum to 1. The function $\max_\Omega$ is defined as $\max_\Omega(\theta) = \sup_{x \in \Delta^d} \langle x, \theta \rangle - \Omega(x)$ using the Smooth-Max formulation. The regularizer Ω being strongly convex ensures a unique minimum and a unique solution x^θ for every input θ. The resulting properties of $\max_\Omega$ are listed as follows: x^θ is unique, the gradient of $\max_\Omega$ at θ is Lipschitz-continuous, and the Hessian of $\max_\Omega$ exists almost everywhere. Furthermore, $\max_\Omega(\theta)$ is a useful approximation in optimization problems involving $\max_\Omega(\theta)$ since it is not far from the simpler $\max(\theta)$ function.

1.1.3 Smoothing the LP and DP Operator

The problem of finding the maximum value of a function over a given set is a common optimization problem that arises in various fields. In this section, we discuss two optimization problems: the linear program and the Bellman iterations, and their smoothed versions.

For the linear Program (LP), the authors suppose to have a set of feasible solutions $\mathcal{Y}$ and a linear function $f : \mathcal{Y} \to \mathbb{R}$, defined as $f(\mathbf{y}) = \langle \theta, \mathbf{y} \rangle$, where θ is a vector of coefficients. Then, the linear program for θ is defined as:

$$\mathrm{LP}_\Omega(\theta) = \max_{\mathbf{y} \in \mathcal{Y}} \langle \theta, \mathbf{y} \rangle$$

The LP can be solved using standard optimization techniques, such as linear programming solvers. However, the LP may not be computationally tractable for large-scale problems.

To overcome the limitations of the LP, we can use a regularizer function Ω to smooth the optimization problem. The regularizer function Ω should be strongly convex and have a minimum at the uniform distribution over $\mathcal{Y}$, denoted by $\mathbf{u}$. Then, we define the smoothed version of the LP as:

$$\begin{aligned}
\max \Omega(f(\mathcal{Y})) &= \max_{\Omega} \left(\max_{y \in \mathcal{J}} \langle \theta, \boldsymbol{J} \rangle \right) \\
&= \max_{\Omega} \left(\sup_{y \in \mathcal{J}} (\theta, \boldsymbol{J}) - \Omega(\boldsymbol{J}) \right) \\
&= \max_{\Omega} \left(\mathrm{LP}\, \Omega(\theta) - \Omega(\mathbf{u}) \right)
\end{aligned}$$

where $\Omega(\mathbf{u})$ is a constant that does not affect the optimization. Note that the smoothed LP can be solved using the same optimization techniques as the LP, and the solution depends on the choice of the regularizer function Ω.
The Bellman iterations are a dynamic programming algorithm used to solve optimization problems on graphs, such as shortest path or maximum flow. The algorithm starts from the nodes with no incoming edges and propagates the optimal values through the graph using the Bellman equation. The dynamic programming solution is obtained when the values converge. Let $G = (V, E)$ be a directed acyclic graph with m nodes and n edges, where $V = 1, \ldots, m$ and $E \subseteq V \times V$. Let G_i denote the set of nodes that have edges to node i, and let $F_i(\theta)$ be the optimal value at node i with parameter θ. Then, the Bellman equation is given by:

$$F_j(\theta) \leftarrow \max i : j \in G_i \left(\theta_{ij} + F_j(\theta)\right), \quad F_1(\theta) \leftarrow 0$$

$$\mathrm{DP}_\Omega(\theta) \leftarrow F_m(\theta).$$

where θ_{ij} is the cost of the edge from node i to node j. The algorithm computes the optimal values $F_i(\theta)$ in topological order of the graph, starting from the nodes with no incoming edges. The dynamic program for θ is defined as the maximum value over all possible paths from node 1. Both LP and DP have their limitations. LP is intractable due to the exponential size of the set $\mathcal{Y}$, which makes finding the maximum value of θ over all possible outcomes computationally expensive. On the other hand, DP is tractable with a computational complexity of $\mathcal{O}(|E|)$, but it requires solving a dynamic programming problem, which may not always be feasible. However, smoothed LP and DP with Lipschitz continuous gradients are useful in many applications, particularly in machine learning.

1.1.4 Computation of the Gradient in Smoothed Relaxations of DP using Differentiable Operators and Backpropagation

To compute $\nabla_\theta \mathrm{DP}\Omega(\theta)$ in the Smoothed Relaxations of DP using Differentiable Operators and Backpropagation, one can use backpropagation along the reverse-topological order of G. In the forward-pass, while computing $F_i(\theta)$, one can obtain $q_i(\theta) = \nabla \max \Omega\left(\theta_i + F(\theta)\right) \in \Delta^m$. Assuming $F_k(\theta) = -\infty$ for all k after i, in the backward-pass, one can compute the gradient recursively by propagating the derivatives from the output layer to the input layer using the chain rule. Specifically, in reverse-topological order $j = m \ldots 1$, one can compute $\nabla\theta\mathrm{DP}\Omega(\theta)$ as $(w_{ij})\, i, j = 1 \ldots m \in \mathbb{R}^{V \times V}$, where $w_{ij} = \bar{w}_i q_{ij}$ if $i \in G_j$ else 0, and $\bar{w}_j = \sum_{i \in G_j} w_{ij}$ if $j \neq m$ else 1.

The matrix $Q(\theta) = (q_i(\theta))_{i=1}^m$ is a transition matrix for backward random walks from $v^* = m$ back to $v^* = 1$, where $\mathbb{P}(i \to j) = q_{ij}(\theta)$ if $i \in G_j$. According to [Mensch and Blondel, 2018], the gradient $\nabla_\theta \mathrm{DP}\Omega$ is the expected path of the random walk, i.e., $\nabla\theta\mathrm{DP}\Omega = \mathbb{E}\mathbf{y} \sim Q(\theta)\mathbf{y}$. Additionally, as $\gamma \to 0$, $\nabla_\theta \mathrm{DP}_{\gamma\Omega}(\theta) \underset{\gamma \to 0}{\longrightarrow} \mathbf{y}^*(\theta) \in \partial \mathrm{LP}(\theta)$, which means that the gradient converges to the optimal solution.

1.2 Applications of Smoothed Relaxations in Structured Prediction and Neural Machine Translation

The authors introduce the Vit-Omega operator for applying dynamic programming to the computational graph of the Viterbi algorithm for sequence prediction. The problem involves tagging a sequence of vectors in $\mathbb{R}^D$ with the most probable output sequence, modeled as finding the highest-scoring path on a treillis. The potentials are organized as a $T \times S \times S$ real tensor. The authors provide pseudo-code for the Vit-Omega operator, as well as gradient and Hessian-product computations. The authors then apply the proposed layers, $\mathrm{DP}\,\Omega(\boldsymbol{\theta})$ and $\nabla\mathrm{DP}\Omega(\boldsymbol{\theta})$, to structured prediction tasks such as named entity recognition and time-series alignment, where the losses are classified into convex and non-convex categories. Furthermore, the authors propose a segmentation attention layer and a structured attention mechanism that use differentiable dynamic programming to compute attention distributions over the input sequence. They show how to backpropagate gradients through these mechanisms and demonstrate their effectiveness with an LSTM encoder and decoder for French to English translation. We provide further details and results in the following :

1.2.1 Experiments Results

The first experiment evaluates the performance of different losses and regularizations on the CoNLL 2003 dataset using a character LSTM and pretrained embeddings. The results show that with proper parameter selection, all losses perform similarly, but entropy-regularized losses perform slightly better on three out of four languages. The ℓ_2^2-regularized losses yield sparse predictions, which helps to identify ambiguous predictions more easily.

Ω	Loss	English	Spanish	German	Dutch
Negentropy	Surrogate	90.80	**86.68**	77.35	**87.56**
	Relaxed	90.47	86.20	**77.56**	87.37
ℓ_2^2	Surrogate	**90.86**	85.51	76.01	86.58
	Relaxed	89.49	84.07	76.91	85.90

The results showed that all losses performed similarly in terms of F1-score, but entropy-regularized losses performed slightly better on three out of four languages. The ℓ_2^2-regularized losses produced sparse predictions, while entropy regularization produced dense probability vectors, making it easier to identify ambiguous predictions. The results are presented in Table 1, along with previous results from Lample et al. (2016).

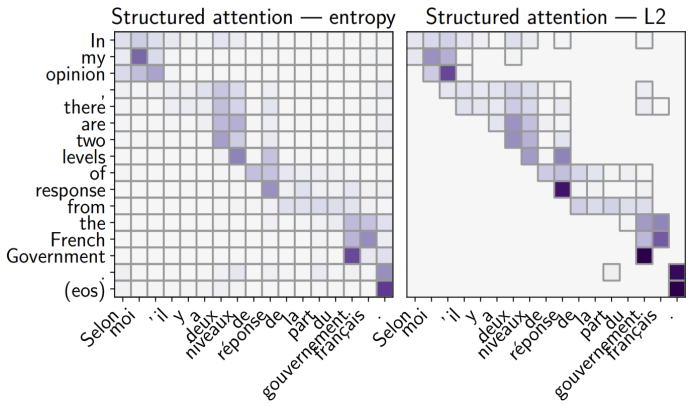
In the second experiment, the authors use their framework to perform supervised audio-to-score alignment on the Bach 10 dataset. The dataset contains 10 music pieces with audio tracks, MIDI transcriptions, and annotated alignments between them. The authors transform the audio tracks into a sequence of audio frames and represent the associated score sequence by a one-hot vector. Each pair of sequences is associated with an alignment matrix. The authors define a discrepancy matrix between the elements of the two sequences, setting the cost between an audio frame and a key to be the log-likelihood of this key given a multinomial linear classifier. They use a softmax function to compute the likelihood and a negative log function to compute the cost. The discrepancy matrix is used as input to the proposed algorithm to perform audio-to-score alignment.

Linear model	Train	Test
End-to-end trained	$\mathbf{0.17 \pm 0.01}$	$\mathbf{1.07 \pm 0.61}$
Pretrained	1.80 ± 0.14	3.69 ± 2.85
Random $\boldsymbol{\theta}$	14.64 ± 2.63	14.64 ± 0.29

They use a feature extractor to transform the audio tracks into a sequence of audio frames and represent the associated score sequence with a one-hot vector. They define a discrepancy matrix between the two sequences based on the log-likelihood of each key given a multinomial linear classifier. They predict a soft alignment using DTW and define a relaxed loss based on the area between the true alignment matrix and the predicted alignment matrix. They perform a leave-one-out cross-validation and report mean absolute deviation on both train and test sets. Their end-to-end technique outperforms a baseline approach of learning the multinomial classifier beforehand and they demonstrate that end-to-end training produces alignments that are visually closer to the ground truth.

1.3 Generalizing Structured Attention

The paper presents a method to generalize segmentation layers to any Ω and efficiently backpropagate through them. The representation of $\boldsymbol{y}$ is a tensor $\boldsymbol{Y} \in 0, 1^{T \times 2 \times 2}$, and the potentials are a tensor $\boldsymbol{\theta} \in \mathbb{R}^{T \times 2 \times 2}$. The gradient $\nabla \operatorname{Vit} \Omega(\boldsymbol{\theta})$ is equal to the expected matrix $\boldsymbol{E} \in \mathbb{R}^{T \times 2 \times 2}$, and the marginals are obtained by marginalizing that matrix.



The experiments demonstrate the use of structured attention layers with an LSTM encoder and decoder to perform French to English translation using data from a 1 million sentence subset of the WMT14 FR-EN challenge. It also illustrates an example of the attention map obtained with negentropy and ℓ_2^2 regularizations. The results showed that introducing structure and sparsity provides enhanced interpretability with comparable performance.

2 The Paper Assessment

The article presents a novel approach for integrating dynamic programming algorithms into neural networks, making them differentiable and improving performance in structured prediction and attention tasks. For us, this paper is generally sound, well-written, and original, but there are some limitations, such as the experiments being limited to a few tasks and datasets and the high computational cost of the method. Beside, the authors have provided sufficient detail for reproducibility, but additional documentation and tutorials could be helpful.
Additionally, some Future research could explore reducing computational costs and applying the method to other structured prediction tasks, as well as combining it with other deep learning techniques (This is what we will try to do in the new experiment session).

3 New Experiments

In this new experiment, we aims to apply machine learning and AI techniques to a classification problem in bioinformatics. The focus is on proteins, which are essential biomolecules in living organisms that have various functions such as chemical reactions and structural support. The challenge is to classify 6,111 protein sequences and their corresponding graph structures into 18 different classes based on their location and function using the Cellular Component ontology. The approach is two-fold, using both graph analysis and natural language processing techniques to understand the protein sequences and structures.

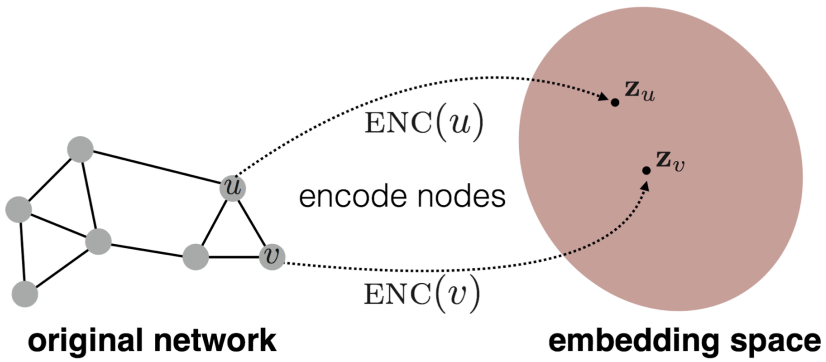
The performance of our models will be assessed using the logarithmic loss measure. This metric is defined as the negative log-likelihood of the true class labels given a probabilistic classifier's predictions.

3.1 Preliminaries

we start with Graph Representation Learning, followed by setting up our notation and presenting the ideas of message passing neural networks and the main family of GNNs, introducing Neighborhood Sampling in GCNs.

3.2 Graph Representation Learning :

Graph Representation Learning is a method for integrating information about a graph's structure into a machine learning model. The goal is to encode the structural information of the graph into a low-dimensional vector space (also known as an embedding space) in order to preserve the geometric relationships of the original network. This is achieved by learning a mapping function that embeds nodes or entire subgraphs from the non-euclidean space to the vector space. The key idea is that nodes that are close to each other in the original network should also remain close to each other in the embedding space, while pushing unconnected nodes further apart. This approach allows for end-to-end learning, where features can be learned through a loss function during the training process. The below diagram illustrates the mapping process, where an encoder maps nodes u and v to low-dimensional vectors z_u and z_v. (reference: stanford-cs224w)



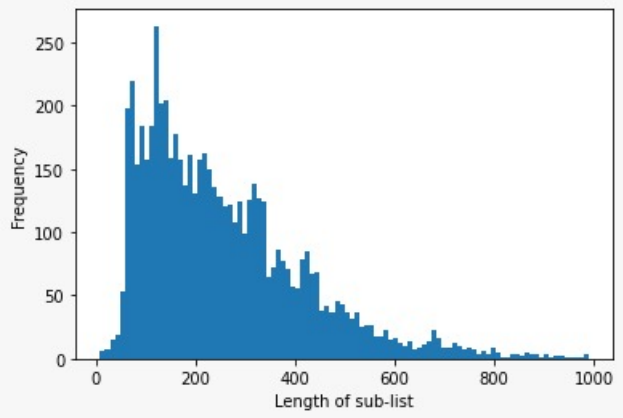
3.3 Overview on Undirected Graphs and Adjacency Matrices

$G = (V, E)$ is an undirected graph consisting of a set of nodes V and a set of edges E. n is the number of nodes in the graph and m is the number of edges. The neighborhood of a node $v \in V$ is denoted by $N(v)$ and the degree of a node v is denoted by $deg(v) = |N(v)|$. The adjacency matrix $A \in \mathbb{R}^{n \times n}$ of the graph G is a symmetric matrix used

to encode edge information, where $A_{i,j}$ is the weight of the edge between the i^{th} and j^{th} node in the graph, or 0 if no such edge exists.

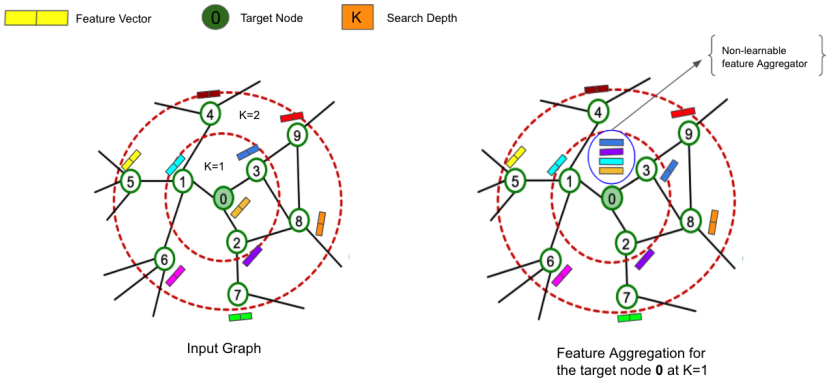
4 Data exploration

The dataset used in this study contains 6,111 proteins represented as undirected graphs with edge attributes provided in "edge attributes.txt" and node attributes provided in "node attributes.txt". The dataset also includes graph indicators in "graph indicator.txt" and class labels in "graph labels.txt". The goal of the study is to predict the class labels of proteins that belong to the test set, with a total of 18 different classes based on their location and function. The logarithmic loss measure will be used to assess the performance of the models.



4.1 Methodology

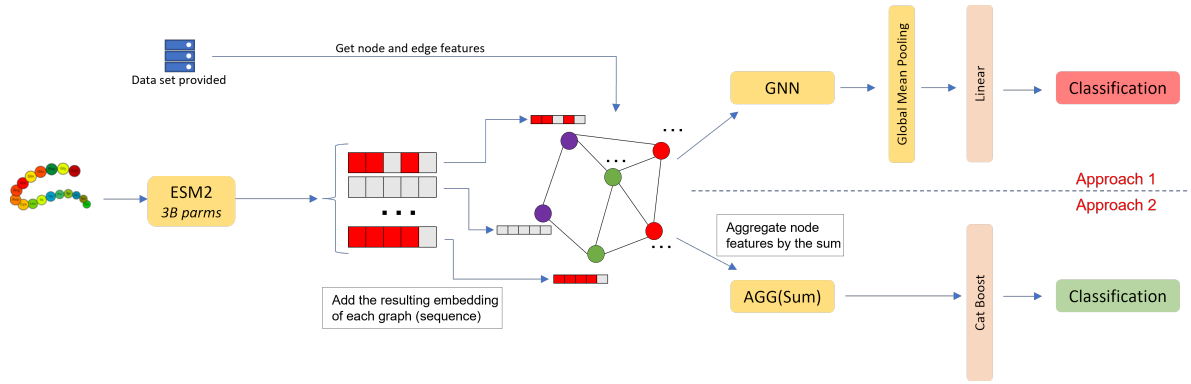
4.1.1 Graph Approach :



The second approach we proposed in this study is based on graph representation of proteins, this approach aims to capture the structural information of the protein by representing it as a graph and applying various graph neural network models. In this approach, we used node attributes and edge attributes to classify proteins. We used Graph Attention Networks (GATs) which is a specific type of graph neural network that can handle graph-structured data by using self-attention mechanism. Additionally, we also used other graph neural network models such as Graph Convolutional Networks (GCN), general Graph Neural Networks (GNN) and GraphSAGE. However, for the GraphSAGE model, we faced some implementation complications with the neighbor data loader, which may have affected the results obtained from this model.

These models are well-suited for this task as they are designed to handle graph-structured data and can capture the structural information of the protein. The goal of this approach is to capture the structural information of the protein by representing it as a graph and apply these models to classify the proteins based on the structural information. We used the information from both nodes and edges attributes as input to these models and the goal is to improve the performance by leveraging the graph structure information of the proteins.

4.1.2 NLP Approach :



we proposed several approaches to classify protein sequences. One approach used natural language processing techniques such as Fasttext and TF-IDF to convert amino acid sequences into numerical vectors. We also used dimensionality reduction techniques such as PCA and Auto-Encoder to capture the most important features of the embeddings and reduce noise. Another approach utilized a LSTM network to capture long-term dependencies in sequential data. We also used pre-trained models such as ProtBERT and ESM-2 to extract embeddings from the protein sequences and fine-tuned them for classification. In addition, we trained a ProtCNN model and a transformer to capture the contextual information of the protein and classify them based on their sequence information. We aimed to improve the model's performance by fine-tuning with an additional classification head.

4.1.3 Mixture Approach :

In our third approach, we combined the first two approaches to classify proteins using both protein sequences and graph structure information. We utilized a pre-trained embedding model (ESM) with different versions to extract embeddings from the protein sequences and concatenated them with node embeddings to feed them into the graph neural network models used in the second approach. To improve performance, we added mean pooling and dropout layers with a probability of 0.2. We also chose to use 2 layers for the Graph Convolutional Network (GCN) model. Our aim was to capture both the sequence and structural information of the proteins and improve the classification performance by combining them.

5 Evaluation

After evaluating different approaches for protein sequence classification, we determined that simple embedding methods and traditional classifiers were not effective, and we opted for more advanced approaches such as training a transformer from scratch and using pre-trained embedding models like ESM. Incorporating ESM embeddings as inputs for graph-based approaches improved performance, and we fine-tuned ESM2 using a classification head and ensemble classifiers like CatBoost. For training, we used the Adam optimizer with a learning rate of $1.e^{-4}$ to minimize multi-class log loss. We experimented with different GNN architectures and found that GCN with 2 layers performed the best. We used a dropout rate of 0.2 and a batch size of 32 to prevent overfitting. Our best results were achieved using ESM2 with 3 billion parameters and the CatBoost algorithm.

5.1 Evaluation metrics

In evaluating our model, we primarily rely on two metrics: accuracy and logarithmic loss function. The logarithmic loss function, also known as log-likelihood, measures the performance of a probabilistic classifier in predicting the true class labels. It is defined as the negative log-likelihood of the predicted class labels.

Specifically, the multi-class log loss is defined as :

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log\left(p_{ij}\right)$$

5.2 Results

Models	ACC Train%	Loss Train	ACC Val%	Loss Val
TF-IDF + SVM	81	0.29	56	1.32
ProtCNN	92	0.62	44	1.42
Transformer	68	0.67	42	1.80
ESM + GCN	79	0.55	70	1.07
ESM2 + Catboost	82	0.40	75	0.75
ProtBert	75	0.49	67	1.43
Lstm	58	1.53	42	1.98

The table shows the accuracy and loss values of different models during the training and validation phases. The best-performing model in terms of accuracy is ESM2 + Catboost, with 82% accuracy on the training data and 75% accuracy on the validation data. The other models, including ProtCNN, Transformer, ESM + GCN, ProtBert, and Lstm, have lower accuracy values on both training and validation data. Therefore, ESM2 + Catboost appears to be the most suitable model for the given classification task.

5.3 Setup configuration

In order to implement our solution, we utilized a cloud-based configuration that included a powerful 16 GB GPU and 40 GB of RAM. This allowed us to efficiently run our GNN models and process large amounts of data. We chose to use the PyTorch Geometric library for implementing GNNs, as it is a powerful and easy-to-use tool that provides a wide range of functionalities for working with graph-structured data. This library also provides a number of pre-built GNN models, which we could use as a starting point for our own implementation.

5.4 Conclusion

In summary, this new experiment aimed to address the problem of protein classification using graph representations. We conducted the given article and a literature review to survey existing methods and their limitations in this field. Then we proposed three frameworks that utilize concepts from NLP, graph convolutional networks and graph attention mechanisms to effectively capture the structural information of proteins and improve classification performance. The effectiveness of these framworks is demonstrated through this experiment on a dataset of 6,111 proteins, and their results were compared to state-of-the-art methods. Overall, the study showed that utilizing large data sets, high-performance computing, and a combination of different techniques and models can lead to improved the model performance.

Bibliography

(1) Bellman, R. (1952).
On the theory of dynamic programming.
Proceedings of the National Academy of Sciences, 38(8):716–719.

(2) Mensch, A. and Blondel, M. (2018).
Differentiable Dynamic Programming for Structured
Prediction and Attention.
In 35th International Conference on Machine Learning,
volume 80 of Proceedings of the 35th International
Conference on Machine Learning, Stockholm, Sweden

(3) A. Elnaggar, M. Heinzinger, C. Dallago, G. Rihawi, Y. Wang, L. Jones, T. Gibbs, T. Feher, C. Angerer, D. Bhowmik, and B. Rost, "Prottrans (2013)
Towards cracking the language of life's code through self-supervised
deep learning and high performance computing," bioRxiv,.

(4) T. Mikolov, K. Chen, G. Corrado, and J. Dean (2013)
Efficient estimation of word representations in vector space,"

(5) P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, (2016)
"Enriching word vectors with subword information".