MASTER THESIS No. 3028

Using Graph Neural Networks to Separate Haplotypes in Assembly Graphs

Filip Wolf

Zagreb, May 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3028

Using Graph Neural Networks to Separate Haplotypes in Assembly Graphs

Filip Wolf

Zagreb, May 2022

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3028

Pristupnik: **Filip Wolf (0036510053)**

Studij: Računarstvo

Profil: Računarska znanost

Mentor: prof. dr. sc. Mile Šikić

Zadatak: **Korištenje graf neuronskih mreža za odvajanje haplotipa u grafovima sastavljanja**

Opis zadatka:

Cilj diploidnog de novo sastavljanja genoma jest ne samo rekonstruirati genomsku sekvencu pojedinca, već i odvojiti dva haplotipa, po jedan naslijeđen od svakog roditelja. Čak i nakon godina istraživanja i brojnih pokušaja, pouzdan alat za ovu vrstu problema nije konstruiran. Međutim, s najnovijom HiFi tehnologijom sekvenciranja, jedan smo korak bliže rješenju. U ovom projektu, prvi korak je korištenje asemblera Raven za konstruiranje grafova sastavljanja iz diploidnih podataka, u kojima čvorovi predstavljaju sekvence koje pripadaju pojedinim haplotipovima, a bridovi predstavljaju preklapanju među tim očitanjima. Idući korak jest konstrukcija modela dubokog učenja za predikciju bridova koji povezuju čvorove iz različitih haplotipova. Micanje tih čvorova iz grafa sastavljanja bi problem diploidnog sastavljanja pojednostavilo na dva zasebna problema haploidnog sastavljanja genoma. Učenje treba biti napravljeno na sintetičkom skupu podataka simuliranom iz genoma bakterija i kvasaca. Evaluacija treba biti napravljena na stvarnim očitanjima bakterija i kvasaca sekvenciranih PacBio HiFi tehnologijom. Rješenje treba biti implementirano u Pythonu koristeći Pytorch ili sličnu biblioteku za duboko učenje. Kod treba biti dokumentirati koristeći komentare i razvijati prema Google Python Style Guide ako je moguće. Cijeli programski proizvod potrebno je postaviti na GitHub pod jednu od OSI odabranih licenci.

Rok za predaju rada: 27. lipnja 2022.

CONTENTS

1. Introduction

Traditionally, the focus of *de novo* genome assembly has always been on the reconstruction of an individual's genome from its numerous broken-up fragments obtained after sequencing [13]. We will here, however, focus on a different application of *de novo* genome assembly: haplotype separation. Every individual's genome is composed of both a mother's and a father's genome. Because we inherit both parent's chromosomes, the genetic material gets mixed between them, but some regions stay together in the form of genes. The term haplotype refers both to these inherited regions, as well as all of the genes of a parent on a chromosome [26]. In this thesis, we will use the term haplotype in the latter sense. By separating the two haplotypes from a genome, we can determine which parent contributed to what genes. This has a wide range of applications, from ancestry tests to finding hereditary diseases [23].

We will try to do this using novel algorithms from the field of *deep learning* (DL). Bioinformatics has long been dominated by algorithms that employ complex heuristics and expert knowledge to find solutions to the problems researchers face. This is however slowly changing. More and more research is being done using deep learning to solve problems in bioinformatics, foregoing the laborious process of feature engineering and extensive human intervention. First, DL was employed only for finding dense and abstract representations of genome features, but it later started to completely replace the previously mentioned algorithms. It has contributed tremendously to the field in recent years and shows no signs of stopping, the most notable achievement being the solution to the protein folding problem which previously wasn't solvable for 50 years [14]. Still, there remains a long way to go before DL becomes completely standard within the field. Thus, this Thesis is concerned with applying recent DL techniques in order to solve the problem of separating the two haplotypes in an existing genome. This could not only potentially improve performance and reduce the necessity for human experts, but also bring bioinformatics to a wider range of people [18].

1.1. Bioinformatics

Bioinformatics is an interdisciplinary field of research that has had a tremendous impact on humanity in the last few decades. Since the completion of the Human Genome Project [16] [22], the cost of sequencing a human genome has fallen exponentially. We can now reliably sequence a human genome for less than a $1000 [21], all thanks to recent advances in sequencing technology, as well as the accompanying algorithms. We will here briefly explain the general pipeline of genome sequencing, as well as the file formats used in the process.

1.1.1. The Process of Sequencing and Assembly

In an ideal world, we would extract a human genome in the form of DNA from a cell, input it into a sequencer, and get a complete and accurate sequence of DNA as output which we could immediately use for further study. However, unfortunately this is not (yet) the case and it is hard to predict when this might become possible. Due to this, we have to make due with sequencers that can only output genomes in the form of thousands of fragmented reads, at maximum about 10 kilobases (kb) long, with shorter read sequencers sequencing reads at lengths of around 150 base pairs (bp). This process is called *shotgun sequencing*. An average genome is much longer than that, e.g. the yeast genome is around 12 Mb long [20] and the human genome is around 6.4 Gb long [19], so after sequencing, we need to assemble these short reads into longer ones before going deeper into analysis.

This process of finding longer reads is the first step in the process of *assembly*. These longer reads are called *contigs*. After this is done, we move on to looking for overlaps between the contigs which we use to create a graph in which each node represents a read, and each edge represents an overlap between reads. An overlap can be described as a match between two contigs' *ends*, that is, the contig's prefix (start) and suffix (end). The length of this overlap can indicate a stronger similarity, and therefore stronger link between two contigs. Finally, we are tasked with finding the longest possible path on this graph to connect the individual contigs and form a complete genome.

1.1.2. File Formats

As a specific subfield that combines computer science and genetics, bioinformatics uses multiple file formats that are unique to it. Mostly used for storing representations of genomic data, they offer us a simpler way to work with such data, as well as modify it. In the following sections, we will briefly describe these file formats, as well as state their usage in this project.

FASTA & FASTQ

The most common file format encountered while working with genomic data is the FASTA format, designated by its .fasta or .fa file extension. It is a text-based format for storing nucleotide sequences and their information. Base pairs of amino acids are stored using letters (A - adenine, C - cytosine, G - guanine, T - thymine) in the form of a sequence. Every sequence starts with a description line, which is designated with a ">" symbol at the start of it. The description line can contain various information about the sequence, such as sequence name, sequence length, sequence statistics, etc. In our case, for the yeast genomic data, it specified the species and chromosome to which the sequences belonged to. After generating the reads (2) for our data, the description contained data such as the read number, strand information, position on the chromosome, length and the chromosome to which it belonged to, as well as if its a mutated read or not. Sometimes, we may also encounter a FASTQ file. It is identical to a FASTA file, but with added sequence quality score information added after every sequence. In the following, we can see an example of a FASTA file filled with reads:

```
>read=1,reverse,position=1004750-1015277,length=10527,
    NC_001139.9
AAGCTTGCAGATTTATTAACAGTTCAAACGAGTTTGGCTGATAATGCTCGTGCAGGTATTG
```

CSV

After generating various information about our dataset, we would store it in a CSV (comma separated values) file. That way, it could be easily read by some of the tools we used. A CSV file can be thought of as a table, where values in a row are separated with a comma sign. The file starts with a header with column names that are also separated with commas.

Another reason why we use the CSV file format is because of the Raven assembler outputting a CSV file with information about sequence overlaps. It is stored in the file in the following way. We start in a descending order of read name. A line with an even index number represents an original read, while a line with an odd index number line represents a *virtual* read, i.e. a *reverse complement* of the original read. A reverse complement represents the same sequence as the original, but in reverse order and with the base pairs replaced with their complements (A - T, C - G). The original and the reverse complement essentially represent the same genomic data, but on different strands of the DNA. An example of such a file is in the following:

```
6549 [3274] LN:i:9216 RC:i:1,6412 [3206] LN:i:11360 RC:i
    :1,1,39678 8937 0 0.992832
6413 [3206] LN:i:11360 RC:i:1,6548 [3274] LN:i:9216 RC:i
    :1,1,39679 11081 0 0.992832
```

```
6647 [3323] LN:i:10656 RC:i:1,6638 [3319] LN:i:10224 RC:i
    :1,1,39680 10347 0 0.990291
6639 [3319] LN:i:10224 RC:i:1,6646 [3323] LN:i:10656 RC:i
    :1,1,39681 9915 0 0.990291
```

First, we have the index of the read. Then, in square brackets, we have the index of the original read (each virtual read forms a pair with an original read). Then, we have some read information, followed by the second read and its own information. Lastly, we have three fields that represent the overlap length, the weight of the overlap (here not used) and its score, measured in percentage of the overlap match respectively. Each line essentially represents an edge in our graph between two nodes. The first n lines in the file, where n is the number of reads, represent the overlap between the original and virtual file.

GFA

Another important file Raven outputs is a Graphical Fragment Assembly (GFA)[1] file. It contains similar information to the CSV file, but with some notable additions, namely, instead of just specifying read overlaps, it also contains the whole sequences it uses in the assembly process. It also includes the original read names specified before assembly that contain mutation information. Each line in the file starts with an identifier, listed in the following table:

Type	Comment
#	Comment
H	Header
S	Segment
L	Link
C	Containment
P	Path
W	Walk

For our purposes, the file only contained lines starting with the letters S and L. S denotes a segment (sequence) used in the assembly, along with most of its information that was present before assembly (this is notable because it is missing in the CSV file). The lines starting with an L contain lines about sequence overlaps in a similar manner to the CSV file, with the addition of overlap length information and mutation info the later being crucial for specifying which edges connect what parent's haplotype. An example of lines starting with an L is in the following:

---

[1]https://gfa-spec.github.io/GFA-spec/GFA1.html

```
L read=4158,reverse,position=395590-406211,length=10621,
    NC_001139.9|mutated + read=4884,reverse,position
    =383506-395832,length=12326,NC_001139.9|mutated + 101M
L read=4832,forward,position=802633-815655,length=13022,
    NC_001139.9|mutated - read=4165,reverse,position
    =793088-803478,length=10390,NC_001139.9|mutated + 733M
```

Miscellaneous

Aside from the mentioned file formats, we created numerous files of our own that either didn't have a suffix, or simply ended with a *.txt* extension, for easier reading and writing. Those files were of an unspecified format and contained temporary information abut the graph, such as overlap lengths, parent affiliation, and similar.

1.2. Thesis Task

The task we are presented with here is slightly different compared to standard genome sequencing. Instead of just finding a path through the graph in order to assemble a complete genome, we are instead tasked with removing edges in the graph that connect contigs belonging to separate parents. By doing this, we are essentially separating the two haplotypes that constitute a genome. To give more insight into this, we will explain what graphs are and how we use them.

1.2.1. Graphs

Graphs are data structures that can be defined with a set of nodes (vertices) V and a set of edges connecting them E. Formally, this can be written as:

$$G = (V, E)$$

where:

$$E \subseteq \{(x, y) | (x, y) \in V^2 \text{ and } x \neq y\}$$

The above definition is an example of a directed graph, meaning that the edges only go in one direction, which is the type of graph we will work with. For instance, if edge e is connecting nodes v_1 and v_2 in the direction $v_1 \rightarrow v_2$, then this means that node v_1 is connected to node v_2, but not the other way around. This is done because edge direction can

encode both suffix - prefix and prefix - suffix overlaps, which is an useful information for us [27].

I this thesis, after sequencing and assembly (1.1.1), we obtain a list of reads connected to each other via overlaps. The reads can be thought of as nodes in a graph, while edges are the links between the nodes. By building such structures, we can more easily use existing graph theory insights to remove unnecessary edges and separate the two haplotypes.

1.3. Deep Learning

1.3.1. Basics

In the last decade, deep learning has grown from a niche research area to one of the largest fields withing computer science. It is now actively employed in virtually every human endeavor, from healthcare to entertainment [7]. And it is still growing day by day on its mission to become the standard way of handling almost all data [24].

In deep learning, we use data processing structures called *artificial neural networks* (ANNs) to extract useful information from our data and learn to predict an outcome, such as some feature of the data or a target class. ANNs do this by adjusting learnable *weights* defined for every neuron in our network. Due to these weights, neural networks are much denser structures when compared to previous machine learning methods and can be referred to as *universal function approximators* [12] due to their ability to, with large enough networks, approximate any function. This gives them unprecedented performance on previously unsolvable tasks, but due to their abstract structure, makes them somewhat difficult to interpret. Some networks, like *convolutional neural networks* [17], don't suffer from this problem as much and can produce some quite intuitive visualizations. On the other hand, some networks, like the ones we will use here, cannot be visually meaningfully interpreted.

To successfully explain how ANNs work, we need to introduce two concepts: a loss function and backpropagation. When use our deep learning model to learn from data, we pass it through our network and compare the output to a previously defined true value by using a *loss function*. A loss function abstracts the error of our network prediction to a single number which is then used to calculate gradients in respect to our data. These gradients are then propagated back through the network using *backpropagation*. In essence, the backpropagation algorithm tells every weight in our network how to change in order to better predict our data. If we imagine our data as a 2-dimensional function on a plane, a gradient is the information of how steeps the function is at any specified point. This steepness value tells the network weights how much they should change, while its sign specifies the direction of change. By correctly propagating these gradient values back through the network, which

backpropagation does, we an successfully make our network learn from data [10].

1.3.2. Graph Neural Networks

While standard ANNs are great at predicting simple data with no underlying structure (or at least one that isn't known), to successfully make our network learn from assembly graphs, we will need something more refined. Yes, it is true that we can simply represent our graph in the form of a 1-dimensional vector, but we then lose precious structural information about the contig overlaps. By using networks more tailor-made for data representation on graphs, we can take advantage of the graph's underlying structure. The networks in question are called *Graph Neural Networks* (GNNs) [25]. Most modern graph neural networks work on the principle of *message passing* [9]. A node accumulates information from adjacent nodes and the edges connecting them and uses it to update its own weights. By repeating this process enough times, we can converge to a stable solution. This can be represented wit the following equation.

Let $x_v \in \mathbf{R}^{d_1}$ be the feature for node v, and $w \in \mathbf{R}^{d_2}$ be the feature for edge (u, v). The message passing paradigm defines the following node-wise and edge-wise computation at step $t + 1$:

Edge-wise: $m_e^{(t+1)} = \phi(x_v^{(t)}, x_u^{(t)}, w_e^{(t)}), (u, v, e) \in \mathcal{E}$.

Node-wise: $x_v^{(t+1)} = \psi(x_v^{(t)}, \rho(\{m_e^{(t+1)} : (u, v, e) \in \mathcal{E}\}))$.

In the above equations, ϕ is a message function defined on each edge to generate a message by combining the edge feature with the features of its incident nodes; ψ is an update function defined on each node to update the node feature by aggregating its incoming messages using the reduce function ρ.

The GNN can be though of as an extension of CNNs. A CNN takes an image's local neighborhood and extracts information from it. It does this using convolution *filters*, which take a certain amount of pixels in a neighborhood and multiply them with weights. Now, the size of this filter is predefined and cannot be changed. For instance, it can have a size of 3 x 3 or 5 x 5. If we were to create such a filter for use on graphs, we would simply designate the central weight of the filter to be the node we are currently looking at, and the surrounding weights would be its neighboring nodes. As we can see, this would limit us to graphs where nodes had a constant number of neighbors, or graphs where we could look only at a limited number of neighbors. GNNs do not have this limitation. The ψ function takes all nodes in a central node's neighborhood into account equally [11] [5].

We used three different GNNs in this thesis, which we will describe in the following sections.

Graph Covolutional Networks

The first network we used was the simplest and oldest one, and it bases its computations on Graph Convolutional Networks (GCN) [2]. It aggregates information from neighboring nodes and the edges connecting them and uses them to update the central node's information. It can be defined as follows:

$$h_i^{(l+1)} = \sigma(b^{(l)} + \sum_{j \in \mathcal{N}(i)} \frac{e_{ji}}{c_{ji}} h_j^{(l)} W^{(l)})$$

Here, $\mathcal{N}(i)$ represents all the neighboring nodes of node i, e_{ji} is the scalar weight of the edge connecting nodes i and j, $c_{ji} = \sqrt{|\mathcal{N}(j)|}\sqrt{|\mathcal{N}(i)|}$, and σ is an activation function. $b^{(l)}$, $h_j^{(l)}$ and $W^{(l)}$ are the networks bias at step l, features of node j at step l and weight of the network at step l respectively. The aggregated information is used to update the features h of node i at step $(l+1)$. We can see that the calculations use are fairly straightforward, as they can be summed up by simply multiplying the network weight and node features and adding bias before passing it through an nonlinearity. A reader familiar with deep learning may notice that this is similar to an iteration of the backpropagation algorithm, but applied to the graph domain.

Graph Attention Networks

The second networks we used is more complex than a simple GCN, as it uses the *attention mechanism* [1] to improve its performance. The Graph Attention Networks (GAT) [6] we used is an updated version of the original GAT [3] called GATv2 and we did notice a slight increase in performance while using it. It can be defined as follows:

$$h_i^{(l+1)} = \sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} W_{right}^{(l)} h_j^{(l)}$$

where:

$$\alpha_{ij}^{(l)} = \text{softmax}_i(e_{ij}^{(l)})$$

$$e_{ij}^{(l)} = \vec{a}^{T^{(l)}} \text{LeakyReLU}(W_{left}^{(l)} h_i + W_{right}^{(l)} h_j)$$

Here, W_{right} is one of the two network weight matrices, h is a node feature matrix, α is an attention weight and $\vec{a}$ is the attention weight vector. We can see that the basic equation is similar to the GCN, but with an added attention weight. This weight is the main reason why the network performs better compared to the regular GCN. The attention mechanism works by selecting data during training, such as nodes, that it considers more important than

other data and thereby giving it more weight. This way, it increases model capacity and performance, while at the same time not directly increasing the network size.

This network differs from the original GAT only in the way $e_{ij}^{(l)}$ is calculated. Instead of using a single weight matrix W that is concatenated to the features h, we use two separate matrices in the form of W_{right} and W_{left}, giving the network more parameters and learning power.

Edge Graph Attention Network

The last network we used, and by far the most effective one, is the graph attention layer that handles edge features (EGAT) [15]. The main equation is the same as in GAT (1.3.2). The difference lies in how the attention scores e_{ij} are calculated. Instead of the usual way, the are obtained like this:

$$e_{ij} = \vec{a} f'_{ij}$$

$$f'_{ij} = \text{LeakyReLU}(A[h_i || f_{ij} || h_j])$$

Here, f_{ij} represents edge features, A is a weight matrix and $\vec{a}$ is a weight vector. This network greatly improves performance due to the fact that it better utilizes edge features. Instead of just using them as weights to scale edge importance, it handles them in the same manner of importance as the node features. This way, the network can find meaning in the edge weights much better compared to a regular GAT network.

2. Dataset

2.1. Introduction

When it comes to deep learning, data can often take a central role in determining the final outcome of the project. Although carefully crafting a predictive model is important, data quality can have a large impact on the model's performance. In this thesis, in order to train our model, we artificially created a dataset based on the brewer's yeast (*Saccharomyces cerevisiae*) genome. For most experimentation purposes, this dataset was only created using the first chromosome of the genome, as using the entire genome required the creation of a much larger dataset that would have significantly slowed down the training and testing process of different models. This larger dataset would have been necessary due to the fact that using the same number of reads for a larger number of chromosomes than just one would essentially dilute our available information for each chromosome, making our models difficult to train. This can be mitigated by the fact that different chromosomes are somewhat similar, as we will see in further experiments. Nevertheless, using only one chromosome proved effective enough for testing out different models, as we will see later in the results section.

2.2. Dataset Creation

2.2.1. Simulating & Mutating the Reads

To generate the dataset, we used a multi-step process. The yeast genome was stored in a simple FASTA file with the chromosomes written down in order. In the first step, we would simply extract the first chromosome from the FASTA file. In the next step, in order to simulate a fragmented chromosome like if it was obtained in the process of sequencing, we needed to generate artificial reads from the chromosome. This was done using the `seqrequester`[1] package and its `generate` option for generating reads. By setting the `-nreads` flag to the desired number, we could specify the number of reads the program would generate, and

---

[1]https://github.com/marbl/seqrequester

using the `-distribution` flag, we could choose the desired distribution for the reads (in our case `pacbio-hifi`). After this was done, we ended up with a file filled with simulated reads. We used a small C++ program[2] to mutate these reads into new ones of the same length with a mutation frequency of 0.01, meaning that every 100 base pairs, a base pair would get mutated into its complement base pair. Now, if the original simulated reads represented the mother's reads, these mutated ones could be thought of as the father's reads. Lastly, we combined them into a single file by simply concatenating them, and we were now ready for the final sequencing process where we would obtain contigs and their overlaps which we could use for training our model.

2.2.2. Assembling the Graph

We found that generating 5000 - 10 000 reads gave satisfactory results in combination with the Raven assembler [28]. This is due to the fact that Raven's `filter` option reduces the number of edges in the graph if a smaller value for the option is specified. While setting the option to 0.99, and using the previously mentioned number of reads, Raven will generate an appropriate number of nodes and edges for training. Setting it to anything lower would create datasets too small for training. Unfortunately, this very low filter value also meant that the overlaps between our contigs were also a minimum of 99%, making them less of a candidate for usage as edge features.

After Raven is done with the assembly process, it outputs two files: a GFA file and CSV file. They both contain information about the contigs and their overlaps suitable in a format for generating a graph. They also contained different information in each, so to fully utilize all the graph information we have available, we needed to use both files. From the GFA file, we extract information about which contig belongs to the mutated reads and which one to the original simulated ones, as well as contig overlap length information. We extract them to a separate file for easier analyzing. Now, as we previously mentioned, our task is to separate the contigs into the father's and the mother's genome. In other words, if we are presented with a graph where nodes represent contigs and edges their overlaps, we need to remove edges between contigs that don't belong to the same parent. So for each edge, we specify if it's "correct" (i.e. it connects two contigs belonging to the same parent) or "incorrect" (it connect two contigs belonging to different parents). We now have a dataset where each edge has a label, as well as a feature in the form of overlap length. We can now use this as input to our model to generate a dataset fit for training.

---

[2]Courtesy of R. Vaser

3. Implementation

3.1. Technology Stack

The main programming language used for this thesis was Python[1], specifically, Python version 3.9. Aside from this, we also used Bash[2] in order to build some scripts to speed up repetitive tasks. Most of the core functionalities of this project were implemented using two Python libraries: PyTorch and DGL. PyTorch[3] probably needs no introduction. It is currently one of the most popular deep learning library used by millions due to its simplicity and versatility [8]. It is free and open source and maintained by Facebook's AI Research lab. In this project, it was mostly used for its *tensor* data structure, its implementation of nonlinearities such as the ReLU and ELU, and loss functions such as cross entropy loss and others. It is also used as the underlying architecture for the deep learning layers we used, which will be described in the next paragraph.

The Deep Graph Library[4] (DGL) [4] is an free and open source deep learning library primarily aimed at the graph neural networks domain. It is maintained by a diverse team of contributors, most of the stemming from the Amazon Web Services team. In this project, it proved itself as a crucial addition to the list of used tools due to its numerous graph oriented functions. We used it mainly for the following features. Firstly, it allowed us to effortlessly and efficiently create graphs from our yeast dataset that were then used for training our models. Secondly, its numerous implemented GNN layers were easy to set up and test out, allowing us to more quickly find the best model for our data. Lastly, due to it being built on the previously mentioned PyTorch library, it had seamless integration with it and could use many of PyTorch's built in functions to help us in training.

Aside from this, we also used a few other libraries to help us in some tasks. We will here shortly list them and describe them.

- NumPy[5] - a popular Python library focused on efficient and easy mathematical cal-

---

[1]https://www.python.org/

[2]https://www.gnu.org/software/bash/

[3]https://pytorch.org/

[4]https://www.dgl.ai/

[5]https://numpy.org/

culations. We mostly used it for its implementation of large arrays

- Pandas[6] - a Python data science library with numerous data oriented features. We used it for its CSV saving and loading capabilities

- Scikit-learn[7] - the most popular machine learning library for Python. We used its easy to use performance metrics, such as F1 score and accuracy

- TensorBoard[8] - a Python library used for easy visualization of the training process and its metrics

3.2. Structure

The project contained three main files as well as some helper scripts. In the following sections, we will describe these.

3.2.1. DGLDataset

The first thing to do was generate a dataset from the data obtained after sequencing. For this, we used DGLs `dgl.data.DGLDataset` class, which we inherit. It has multiple purposes, after generating a dataset we can save it and load it by calling the `save` and `load` functions. We can also check if there already is a previously saved dataset using the `has_cache` function. We can also get an instance of the dataset by calling the `__getitem__` function, as well as its length using the `__len__` function. But by far its most important purpose is generating the dataset we will use for training. We do this by loading the previously generated node and edge features and using them to generate a graph. We also encode label information, edge features, and node features. Node features are generated by taking the in and out degrees of every node.

3.2.2. Models

The models we used for training are defined by inheriting the `torch.nn.Module` class. In each model, we define the layers of the network, as well as the `forward` function. An example of the EGAT model is in the following:

```
class EGATModel(nn.Module):
    def __init__(self, node_features, edge_features,
        lin_dim, hidden_dim, out_dim, n_classes, num_heads):
```

---

[6]https://pandas.pydata.org/

[7]https://scikit-learn.org/stable/

[8]https://www.tensorflow.org/tensorboard

```
super(EGATModel, self).__init__()
self.lin_n = nn.Linear(node_features, lin_dim)
self.egat1 = EGATConv(lin_dim, edge_features,
    hidden_dim, hidden_dim, num_heads=num_heads)
self.egat2 = EGATConv(hidden_dim * num_heads,
    hidden_dim * num_heads, out_dim, out_dim,
    num_heads=1)
self.egat3 = EGATConv(out_dim, out_dim, int(
    out_dim / 2), int(out_dim / 2), num_heads=1)
self.egat4 = EGATConv(int(out_dim / 2), int(
    out_dim / 2), int(out_dim / 4), int(out_dim /
    4), num_heads=1)
self.classify = MLPPredictorEGAT(int(out_dim / 4)
    , n_classes)
self.dp = nn.Dropout(p=0.5)
```

We define this network in the following way. First, we initalize a linear layer that transforms the node features (in and out node degrees) into a higher dimension of `node_features`. Then, we define for EGAT layers. The first layer is of the hgihest size `hidden_dim` (both for node and edge features), and subsequent layers divide this size by two in each step. Only the first layer used `num_heads` attention heads, the rest use only one. This is done to improve trainign times, as this network proved much slower than the other ones we used. After all the EGAT layers are done with their computations, we pass everything through a classification layer. It works by taking the new calculayed node and edge features and concatenating them before passing them into a fully connected classification layer. The network then finally outputs a list of class probabilities for every edge in the graph.

3.2.3. Main

All of the previously described functions are connected in the `main` function. First, we define the function for logging parameters such as loss and accuracy for visualization using TensorBoard. Then, we define the dataset by calling the `DGLDataset` module and, if it previously hasn't been generated, create the dataset. We then define the model we will use for training with all its parameters and layer sizes. We define the optimizer, and finally, start the training process.

4. Conclusion

BIBLIOGRAPHY

[1] *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv, 2014.

[2] *Semi-Supervised Classification with Graph Convolutional Networks*. arXiv, 2016.

[3] *Graph Attention Networks*. arXiv, 2017.

[4] *Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks*. arXiv, 2019.

[5] *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*, 2021.

[6] *How Attentive are Graph Attention Networks?* arXiv, 2021.

[7] M. Chatterjee. Top 20 applications of deep learning in 2022 across industries, 2022.

[8] S. . Data. Most popular machine learning libraries – 2014/2021.

[9] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 06–11 Aug 2017.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] W. L. Hamilton. 2020.

[12] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[13] Illumina. Assembling novel genomes.

[14] E. R. P. A. e. a. Jumper, J. Highly accurate protein structure prediction with alphafold. *Nature*, 2021.

[15] J. M. B. A. M. R. S. K. D.-H. S. Kamiński K, Ludwiczak J. Rossmann-toolbox: a deep learning-based protocol for the prediction and design of cofactor specificity in rossmann fold proteins. *Brief Bioinform.*, 2022.

[16] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, and W. e. a. FitzHugh. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[17] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] S. Min, B. Lee, and S. Yoon. Deep learning in bioinformatics. *Briefings in Bioinformatics*, 18(5):851–869, 07 2016.

[19] NCBI. Homo sapiens.

[20] NCBI. Saccharomyces cerevisiae.

[21] NIH. The cost of sequencing a human genome, 2021.

[22] S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N.-C. Chen, H. Cheng, C.-S. Chin, W. Chow, L. G. de Lima, P. C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I. Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W. Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

[23] PacBio. Sequencing 101: ploidy, haplotypes, and phasing — how to get more from your sequencing data.

[24] Z. M. Research. Outlook on the global deep learning market size, share & growth 2022 - 2028 | estimated to achieve a revenue of $80769.6 million with growing at a cagr 38.3

[25] F. Scarselli, A. C. Tsoi, M. Gori, and M. Hagenbuchner. Graphical-based learning environments for pattern recognition. In A. Fred, T. M. Caelli, R. P. W. Duin, A. C. Campilho, and D. de Ridder, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 42–56, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[26] Scitable. haplotype.

[27] R. J. Trudeau. *Introduction to graph theory*. Stanford, 2017.

[28] R. Vaser and M. Šikić. Raven: a de novo genome assembler for long reads. *bioRxiv*, 2021.

Using Graph Neural Networks to Separate Haplotypes in Assembly Graphs

Abstract

Keywords:

Sažetak

Ključne riječi: