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, June 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, June 2022

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

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 [15]. 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 [35]. 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 [31].

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 [17]. 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 [24].

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 [21] [30], the cost of sequencing a human genome has fallen exponentially. We can now reliably sequence a human genome for less than a $1000 [29], 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 and assembly (section 1.1.1), as well as the file formats used in the process (section 1.1.2).

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 [27] and the human genome is around 6.4 Gb long [26], 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 (Chapter 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 (section 3.1). 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 (section 2.2.2). 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

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

specifying which edges connect what parent's haplotype. An example of lines starting with an L is in the following:

```
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:

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

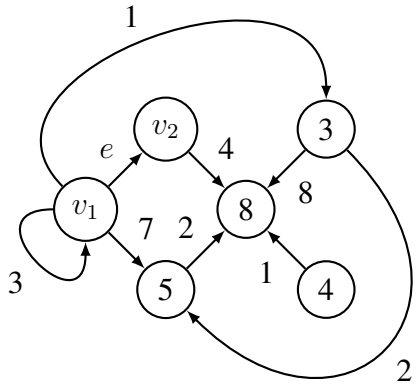

Figure 1.1: An example of a directed graph with node and edge weights. Aside from every edge being directed, loops are also present.

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. A representation of such a graph can be seen in Figure 1.1. 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 [37]. Aside from this, both the nodes and edges have weights assigned to them that can represent various information about the graph.

In 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

In this section, we will go over the basics of deep learning (section 1.3.1), the main tool used for the experiments in this thesis, followed with an overview of graph neural networks (section 1.3.2) and all its different variants we used.

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 en-

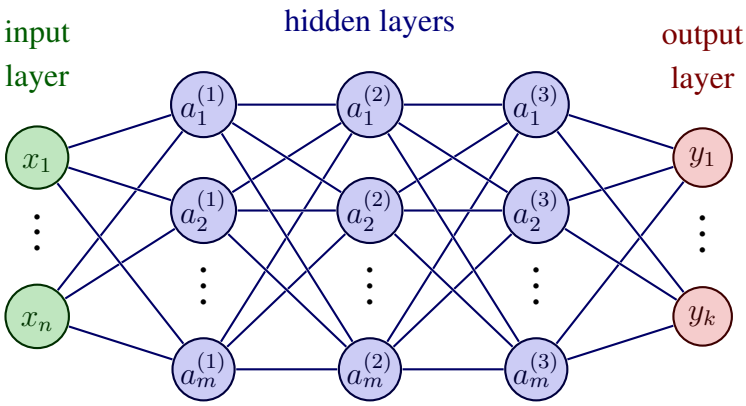

Figure 1.2: Example of a fully connected neural network. The green nodes represent the input to the network of size n. The blue nodes represent the hidden layers, with each connection between them having trainable weights. They are of size m with the numbers in brackets representing the layer index. Finally, the red nodes represent the output of the network of size k. Taken from [28].

deavor, from healthcare to entertainment [8]. And it is still growing day by day on its mission to become the standard way of handling almost all data [33].

Artificial Neural Networks

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. An example of a fully connected neural network can be seein in Figure 1.2. 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* [14] 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* [22], 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.

Loss Function & Stochastic Gradient Descent

To successfully explain how ANNs work, we need to introduce two concepts: a loss function and backpropagation. When we use our deep learning model to learn from data, we pass the data 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. The loss function

we used is described in section 4.1.3. These gradients are then propagated back through the network using *backpropagation*. But, before explaining the backpropagation algorithm, we first need to explain a related, but simpler concept in the form of *Stochastic Gradient Descent* (SGD). In SGD, to find a function's minimum value, we simply nudge all the weights of the function in the direction opposite the function's gradients. This will bring the weights closer to an optimal position as the gradient points in the opposite direction of the function minimum. This is represented in the following equation:

$$w = w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla Q_i(w)$$

where w are the function weights, n is the number of data samples, η is the learning rate (section 3.3.1) and $\nabla Q_i(w)$ is the gradient of the function Q over weights w.

Now, SGD can also come in multiple forms. True SGD, as in the above equation, calculates these weight changes by first calculating function loss on all data we have available, averages it and only then updates weights. This is very accurate, but also rather slow. On the other end of the spectrum, we can update weights after we process every piece of data. This is very fast and has a lower chance of ending up in local optima, but is also too imprecise for fine-tuning network performance. Due to this, we use a compromise of the two in the form of *mini-batches*. Instead of using only one or all data, we use a batch of data. Using larger batches is slower and more precise, while using smaller batches is faster and less precise. That being said, most often, we are limited by memory constraints while determining batch size.

Backpropagation

We can now move on to the backpropagation algorithm. In essence, it is the idea of SGD applied to neural networks. It tells every weight in our network how to change in order to better predict our data. By correctly propagating these gradient values back through the network, which backpropagation does, we can successfully make our network learn from data [12].

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) [34]. Most modern graph neural networks work on the principle of *message passing* [11]. A node accumulates information from adjacent nodes and the edges connecting them and uses it to update its own weights. The information is represented in the form of node and edge features, multidimensional vectors of data that represent some information about the nodes and edges. 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 [13] [6].

We used three different GNNs in this thesis, which we will describe in the following sections. We will start with the most basic GNN, the Graph Convolutional Network (section 1.3.2), follow them up with Graph Attention Networks (section 1.3.2), and finally finish with the most complex network type, the Edge Graph Attention Network (section 1.3.2).

Graph Covolutional Networks

The first network we used was the simplest and oldest one, and it bases its computations on Graph Convolutional Networks (GCN) [3]. 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* [2] to improve its performance. The Graph Attention Networks (GAT) [7] we used is an updated version of the original GAT [4] 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:

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

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

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 instances 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. It is also worth mentioning that we can specify a number of so-called attention heads. Attention heads are simply multiple attention mechanisms at work at the same time. The final attention score is calculated by concatenating or averaging these different attention values. This is done to improve training stability and ultimately performance.

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) [18]. The main equation is the same as in GAT (section 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.

1.3.3. Other Functions

In this section, we will explain all the functions we use in our networks that aren't directly responsible for training, such as regularization functions and nonlinearities.

Nonlinearities

We use four different nonlinearities in our work. Three of them are used as nonlinearities between network layers, and one of them, the sigmoid function, is used for final classification of the edges. First, we are going to describe the network nonlinearities. The most basic one is the **ReLU** (rectified linear unit) [25], represented with the red line in Figure 1.3. It can be simply described with the following equation:

$$\text{ReLU}(z) = \max(0, z)$$

In other words, it lets through everything that is positive in a linear fashion, and sets everything else to zero. This is one of the most widely used nonlinearities. It has the same benefits as a sigmoid function or a tanh function (being differentiable), but is less computationally expensive and doesn't suffer from saturated weights for high input values. That being said,

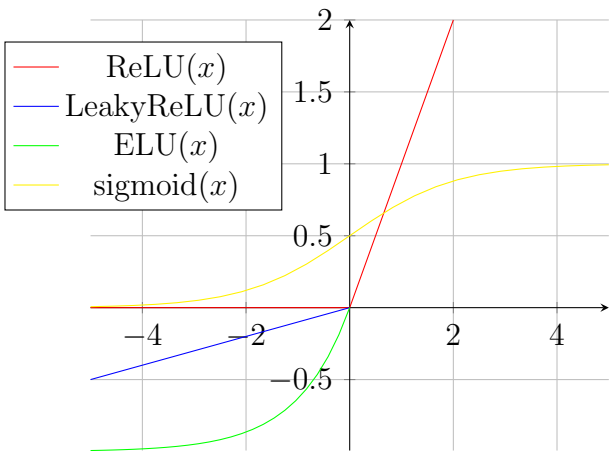

Figure 1.3: An example of all the different nonlinearities used in this thesis for model training. In red, we can see a regular ReLU function, which is valued at zero for $x < 0$. The blue line represents the LeakyReLU function, which lets a small value pass through it for $x < 0$, here value at $0.1 * x$. The green line represents the ELU function, which takes a exponential form for $x < 0$. We can observe that these functions are different only for $x < 0$. Lastly, the yellow line represents the sigmoid function. Unlike the other functions, it restricts both positive and negative values to range of $[0, 1]$.

it does suffer from the exploding gradients problem because, unlike the sigmoid or tanh, it does not restrict the size of the outputs.

A variant of the ReLU is the **LeakyReLU** [23], represented with the blue line in Figure 1.3. It only differs from ReLU in that it allows for a small non zero constant gradient α below zero. It tries to fix the limitation of ReLU where some neurons never express their values due to them always being negative.

Finally, we have the **ELU** (exponential linear unit) [9], represented with the green line in Figure 1.3. It can be described with the following equation:

$$\text{ELU}(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases}$$

For values greater than 0, it emits a constant output. But for any other value, it slowly smooths out bellow zero where it tends to the value $-\alpha$. Just like LeakyReLU, it also tries to negate the previously mentioned weakness of ReLU and can be used as a strong alternative to it.

Lastly, for the classification of the network outputs, we need to normalize them into a $[0, 1]$ range so they can be interpreted as probabilities. For this, we use the **sigmoid** function, represented with the yellow line in Figure 1.3 and described with the following equation:

$$S(x) = \frac{1}{1 + e^{-x}}$$

where x is the input to the function. It is worth noting that for cases where there are more

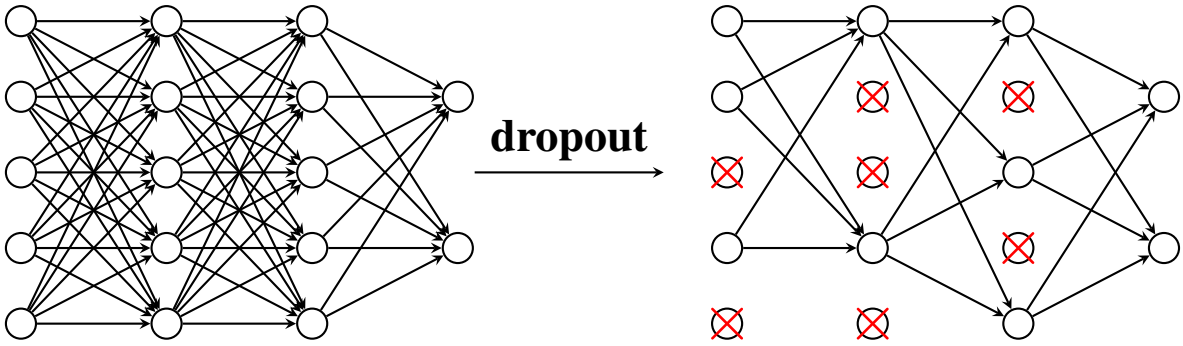

Figure 1.4: An example of dropout. On the network on the right, we can see that some nodes, as well as their connections, have been dropped. This leads to sparser network and more node specialization. Taken from [28].

than two classes present, we actually use the *softmax* function. It can be described with the following equation:

$$\sigma(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Regularization Functions

We primarily used three different regularization functions throughout our networks. In general, these functions help stabilize and speed up network training, as well as often improve performance.

Before imputing our data into the model, we **normalize** it, i.e. bring its values into a $[0, 1]$ range. This is done so that input data from different sources is all in the same range. If this isn't done automatically in the network, we do it with built-in functions. There are many different ways to normalize edge and node data. Input and output node degrees are normalized by subtracting the minimal respective node degree value from them and dividing them with the maximum value which was also reduced with the minimal value, a process known as min-max scaling. Formally, this can be written as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Normalizing edge weights is a bit more complicated. The method we use for this is represented with the following equation:

$$c_{ji} = \left(\sqrt{\sum_{k \in N(j)} e_{jk}} \sqrt{\sum_{k \in N(i)} e_{ki}} \right)$$

e_{jk} and e_{ki} are the weights of the edges from node j to node k and from node k to node i respectively. We divide all of the weights with the calculated weight c_{ji}. Intuitively, we

multiply the square roots of the sums of weights of the edges connected to the two nodes that are being connected by edge e_{ji}. It is worth noting that we usually did not use node degree normalization, as we would transform the node data into a higher abstract dimension anyway, where the individual node degrees would get lost.

More interesting that normalization is **dropout** [1] [36]. Dropout is one of the most popular regularization techniques as it often greatly improves network performance, but at the cost of training speed. It works by randomly omitting some neurons from the neural network along with their connections with some probability p. By doing this, neurons get more specialized in detecting different features in the data and don't co-adapt as much. In practice, these neurons aren't actually removed from the network, but instead their weights are simply reduced by multiplying them with p, thereby lessening their importance during training and subsequent inference, which has the same effect as removing them but is less computationally expensive. We used a value for p of 0.2 for regular GCN layers (section 1.3.2), and a value of 0.5 for GAT layers (section 1.3.2). A visualization of dropout can be seen in Figure 1.4.

Lastly, we employed **batch normalization** [16]. As previously described, we normalize input values so that the network can train more easily. However, during training, in deeper layers, some of that normalization is lost and neurons have to chase a moving target in order to learn properly from data. This slows down training speed ad reduces performance. To alleviate this, batch normalization is used to normalize the outputs of neurons during training so that the next layers can always expect the same range of values as input, thereby stabilizing and greatly improving training times. It also allows for a much wider range of learning rate values to be used without making the model diverge. As a side effect, by making the layer activations less dependent on the current batch, it adds some noise to training, and thereby acts as a regularizer to the network.

2. Dataset

In this chapter, we will go over the dataset we used for our models. First, we will give an introduction to the dataset we chose to use and why we did so (section 2.1), and then explain how we generated our own dataset for use in our models (section 2.2).

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 chapter (chapter 4).

2.2. Dataset Creation

There ar a lot of readily available datasets for experimenting on graphs and genomic data, but for our intents and purposes, we required a custom dataset created from our own data. We will explain in detail how we created this dataset and how we used it in our experiments.

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 (section 1.1.2) 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 using the `-distribution` flag, we could choose the desired distribution for the reads (in our case `pacbio-hifi`). We did this n times, where n is the number of graphs we wanted to generate for training, usually 100. After this was done, we ended up with n files 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 the father's and mother's reads from each of the $2n$ (n for the father's and n for the mother's reads) files into a single file by concatenating them, which left us with n files that were ready for the final sequencing process where we would obtain contigs and their overlaps which we could use for training our model.

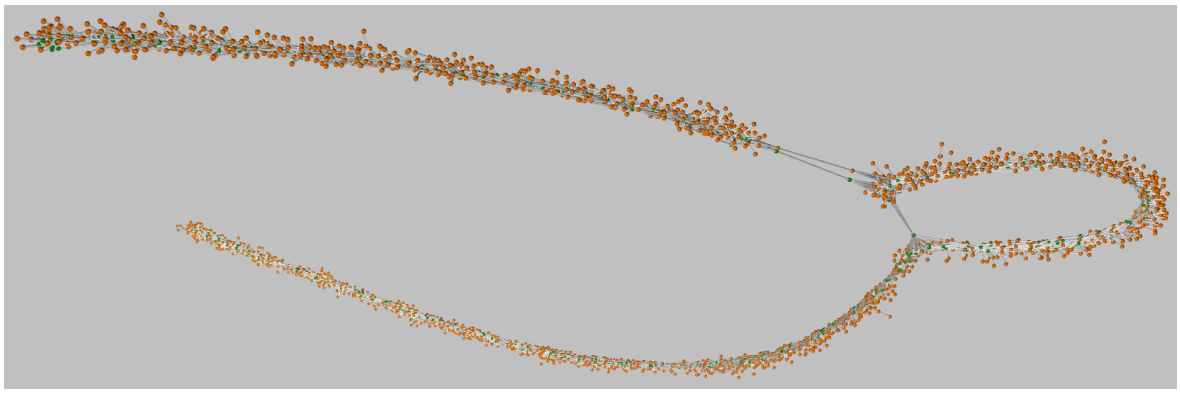

Figure 2.1: Example of a graph obtained after sequencing and assembly. Orange nodes represent mutated reads, while green nodes represent the original reads. We would generate in surplus of 100 of these graphs in order to train our model. This graph was obtained using the Graphia[3] graphing tool.

[1] https://github.com/marbl/seqrequester
[2] Courtesy of R. Vaser
[3] https://graphia.app/

2.2.2. Assembling the Graph

We found that generating 5000 - 10 000 reads gave satisfactory results in combination with the Raven assembler [38]. 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. We won't go into detail about why this is so, as it is unclear to us as well.

After Raven is done with the assembly process, it outputs two files: a GFA file (section 1.1.2) and CSV file (section 1.1.2). 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. An example of a graph obtained after assembly can be seen in Figure 2.1. 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.

3. Implementation

In this chapter, we will go over all the different software components we used for our experiments. First, we will go over the software we used for our deep learning models, as well as all the tools we used to help us (section 3.1). After that, we will explain the structure of our model and how all the different components come together (section 3.2).

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 [10]. 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 (section 1.3.3), 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) [5] 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 (section 1.3.2) 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

[1] https://www.python.org/
[2] https://www.gnu.org/software/bash/
[3] https://pytorch.org/
[4] https://www.dgl.ai/

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 calculations. 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. Code 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 for the

[5] https://numpy.org/
[6] https://pandas.pydata.org/
[7] https://scikit-learn.org/stable/
[8] https://www.tensorflow.org/tensorboard

gradients calculation. An example of the EGAT (section 1.3.2) model definition code 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):
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 the network in the following way. First, we initalize a linear layer that transforms the node features (number of in and out node degrees) into a higher dimension of size `node_features`. Then, we define four EGAT layers. The first layer is of the largest width `hidden_dim` (both for node and edge features), and subsequent layers divide this size by two in each step. Only the first layer used a `num_heads` number of attention heads, the rest used only one. This is done to improve training 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. This layer works by taking the new calculated 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.

You may have noticed that in some layers, we multiply the input dimension for the EGAT layer with `num_heads`. This is done so that the number of attention heads is accounted for correctly while passing the results to the next network. The different layer outputs are *flattened*, i.e. concatenated into a smaller dimensional vector before being passed to the next layer. Aside from this, we also employ the *dropout* (section 1.3.3) mechanism to reduce overfitting (section 3.3.2).

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 (section 3.1). 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. After that, we define the optimizer for our model, and finally, we start the training process. This process is depicted in Figure 3.2.

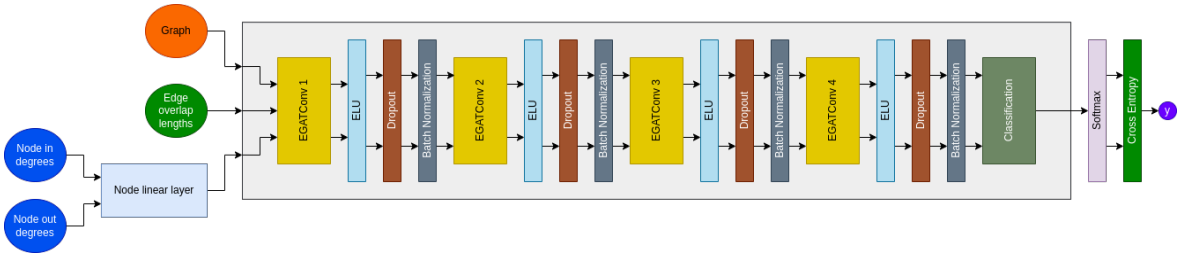

Figure 3.1: Graphical representation of the model architecture with EGAT layers. Different layers are represented with different colors and the main model is described within the gray box. This image was constructed using diagrams.net[9]

3.3. Model Description

In Figure 3.1, we can see the general structure of one of our models, specifically, the model using the EGAT layers (section 1.3.2). We will explain this model in detail as it is the most complicated one we used and it generalizes well to all other models we used.

In the first step, we take node features, node in and out degrees, as input to a linear layer. The dimension of each of these feature vectors is n, where n is the number of nodes in the current graph. The purpose of this linear layer is to increase the size of the features from two to a larger number m_1 (usually 64 or 128), thereby extracting useful information from them and successfully representing it with a suitable dimension.

We then have a $n \times m_1$ matrix, which we use as input to the first EGAT layer along with the node features we use (edge overlap lengths) and the graph structure. The EGAT layer uses these to perform its operations and outputs two vectors of size $n \times m_2 \times a$ and $e \times m_2 \times a$, where m_2 is the output size of the first EGAT layer (up to four times alrger than m_1), a is the number of attention heads (usually three) and e is the number of edges. The two vectors represent the learned features of the nodes and edges respectively. They are then passed through ELU activation function (section 1.3.3) which acts as a nonlinearity,

[9]https://www.diagrams.net/

followed with the dropout mechanism (section 1.3.3). This is then finally passed through batch normalization (section 1.3.3).

This set of operations represents one step of the network, which we perform four times. In each step, we reduce the m_2 dimension two times in each step and use only one attention head. After the final layer is done, we pass everything through a classification layer. This layer works by taking the last set of calculated node and edge features and concatenates them in a node feature, edge feature, node feature order. These are then used as input to a fully connected layer which finally outputs logits. To use them in a loss function, we pass them through a sigmoid function to get class probabilities.

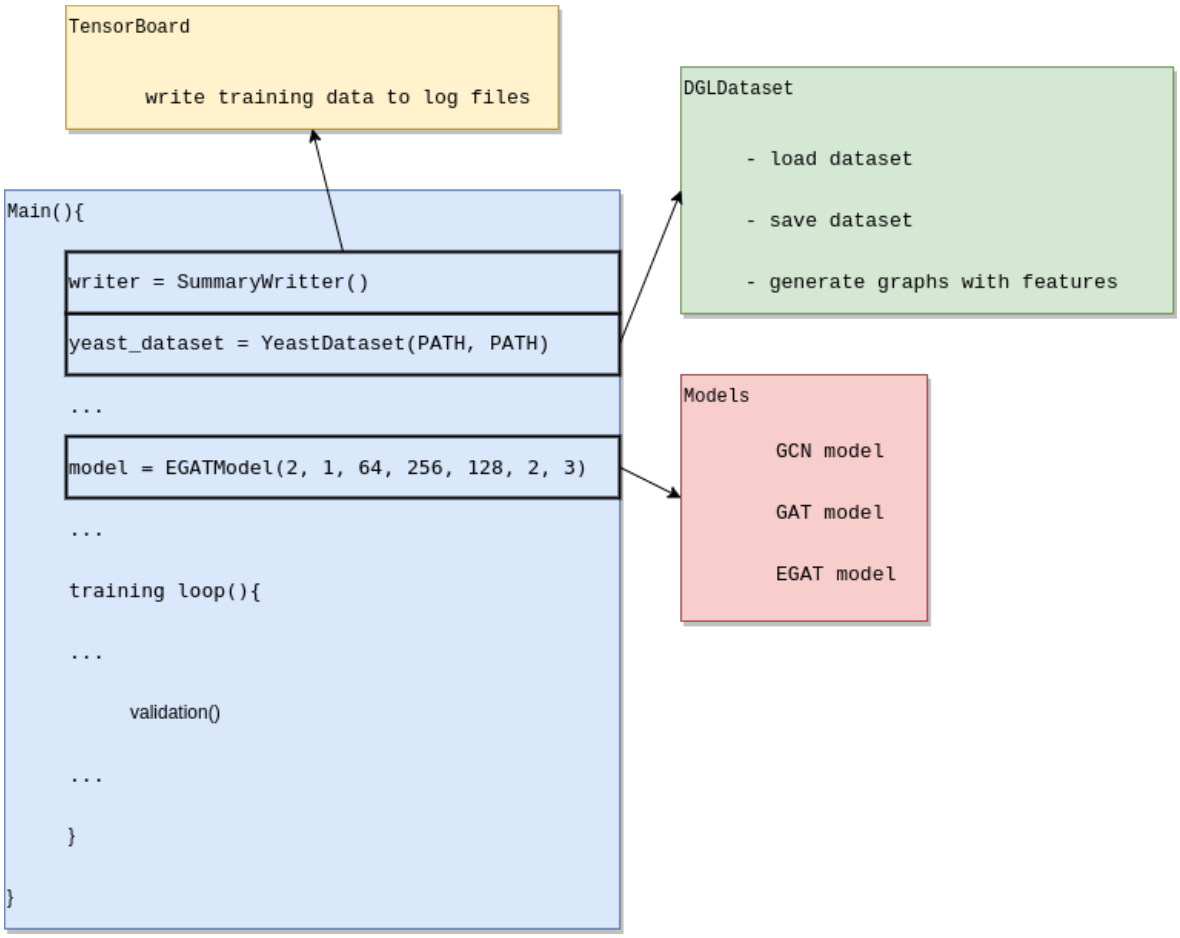

Figure 3.2: Illustration depicting the architecture of our system. All the different components are joined together in the Main program. This image was constructed using diagrams.net[10]

3.3.1. Optimizer

An important element of every machine learning system is the optimizer. An optimizer is a function that controls how the weights of our network are adjusted after we obtain the gradi-

[10]https://www.diagrams.net/

ents. The optimizer we used is the popular Adam (Adaptive Moment Estimation) optimizer [19]. It falls into the adaptive optimizers category and represents a nice balance between speed and performance due to using two different adaptive techniques: momentum and root mean square propagation (SMSProp). Recently, there has been more and more research about the advantages and disadvantages of adaptive optimization techniques compared to standard *Stochastic Gradient Descent* (SGD). However, it is still a good idea to first use Adam as it doesn't require hyperparameter tuning to offer good results [32]. Momentum accelerates gradient descent using the *exponentially weighted average* of the gradients and can be described with the following equations:

$$w_{t+1} = w_t - \alpha m_t$$

where

$$m_t = \beta m_{t-1} + (1 - \beta)\frac{\partial L}{\partial w_t}$$

Here, w_t and w_{t+1} are the current and new weights respectively and α is the learning rate. m_t and m_{t-1} are the current and previous moving averages respectively, while $\frac{\partial L}{\partial w_t}$ is the derivative of the loss function by the weights of the network. Lastly, β is the moving average parameter or decay value, usually 0.9. Essentially, momentum accumulates previous gradients to speed up its convergence towards an optimum. On the other hand, RMSProp can be described as:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}}\frac{\partial L}{\partial w_t}$$

where

$$v_t = \beta v_{t-1} + (1 - \beta)\left(\frac{\partial L}{\partial w_t}\right)^2$$

Here, v_t and v_{t-1} are the current and previous sum of squared gradients (moving averages) and ϵ is a small value added to avoid division by zero. RMSProp works the same way as momentum, but instead uses the squares of gradients and instead of multiplying α with v_t, it divides by it.

Finally, Adam is defined with the following set of equations:

$$m_i = \beta_1 m_i + (1 - \beta_1)\frac{\partial L}{\partial \theta_i}$$

$$v_i = \beta_2 v_i + (1 - \beta_2)\left(\frac{\partial L}{\partial \theta_i}\right)^2$$

then:

$$\widehat{m_l} = \frac{m_i}{1-\beta_1}, \widehat{v_l} = \frac{v_i}{1-\beta_2}$$

$$\theta_i = \theta_i - \frac{\alpha}{\sqrt{\widehat{v_l}} + \epsilon}\widehat{m_l}$$

It uses both the first and the second momentum which are divided by one minus the decay factor β to account for bias in the estimator. ϵ is a small value added in order to avoid division by zero and α is the learning rate. β_1 is almost always 0.9, while β_2 is usually 0.999. In the first iteration, the moving averagesa are set to zero. By using techniques from both SGD with momentum and RMSProp, Adam inherits both of their strengths. The main advantages are that Adam takes big enough steps to avoid local otpima, while simultaneously oscillating minimaly when it reaches the global optimum.

During training, we used a version od Adam implemented in PyTorch (section 3.1) with default parameters. The learning rate was set to 0.001, the β parameters were set to 0.9 an 0.999 respectively, ϵ was set to 1e-8 and weight decay wasn't used.

3.3.2. Training and Validation

After we generated our dataset of n graphs (100 for the dataset with only one chromosome), we used 99 graphs for training and one for validation. This is done to prevent *overfitting*. This can be explained in the following way. As our model trains on the training part of the dataset, it starts to memorize the data in our dataset by heart. This may sound good, but it actually leads to poor *generalization*, i.e., it cannot perform well on other similar data, only on the data it was trained on. To alleviate this, we separate a smaller validation dataset from the main one that the model doesn't train on. By validating our performance on it, we can get a more accurate state of the performance of our model. As the model starts overfitting, performance on the train part of the dataset will continue to improve, while on the validation part, it will start to worsen. We can then take the step where the model performed on the validation dataset the best as the best version of our model.

Aside from this, validation can also be used for tuning *hyperparameters* of our model, i.e. parameters that aren't adjuseted during training, but are set before training. In that case, as some of the information from the validation dataset leaks into the training, we also need to create a test dataset that will now fulfill the role that validation has fulfilled previously: generalization testing. In this thesis, we don't do this as most of the hyperparameters we used are set to default and we aren't concerned with top performance as much as testing out if the concept of haplotype separation works with GNNs and with what models.

3.3.3. Hyperparameters

There are numerous important hyperparameters that can be adjusted to optimize training performance. However, due to time constraints and simplicity reasons, we mostly used default parameters in this thesis. We already mentioned the hyperparameters we used for the Adam optimizer (section 3.3.1), the network sizes we used (section 3.3), as well as the values used for dropout (section 1.3.3), but there are others that require an explanation. An important one is batch size (section 1.3.1). Batch size determines how much of our data enters our network during one step of training. In GNNs, it translates to the number of separate graphs in an instance of the dataset. The message passing algorithm (section 1.3.2) doesn't pass data over from separate graphs, so batches can be implemented very easily. That being said, the graphs we used for training were of sufficient size and using more than one graph during an instance of training simply wasn't necesary. In addition, some graphs appeared fragmented after assembly anyways, so they acted like mini-batches regardless.

4. Results

In this chapter, we will present the results obtained in the experiments we performed for this thesis. First, we will go over the experiments on only one chromosome (section 4.2), and then over the experiments on the whole genome.

4.1. Performance Metrics

Here, we will briefly explain all the different metrics for evaluating model performance during training that we used. These include accuracy (section 4.1.1), F1 score (section 4.1.2) and model loss (section 4.1.3).

4.1.1. Accuracy

Accuracy is a simple and reliable metric foe assessing model performance that is quite popular and widespread due to its interpretability and it being easy to understand. It can be described as the number of correctly classified examples divided by the total number of examples in the dataset. With that said, it does have limits to its use. For instance, in the case when the dataset is severely imbalanced, i.e. there are many more examples of one class, e.g. *true* compared to the other, e.g. *false*, it will always display a high performance figure by simply always guessing *true*. Fortunately, this is not the case for our dataset, as we have a fairly similar number of correct and incorrect examples. We can also define it with the following equation:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Accuracy is the number of *true positive* and *true negative* examples divided by all examples. In other words, it is the sum of everything our model classified correctly divided by both everything our model did and didn't classify correctly.

4.1.2. F1 score

F1 score is slightly more complex than the previously described accuracy score. It belongs to a family of scores known as *F-scores*, which can be described with the following equation:

$$\mathrm{F} = (1 + \beta^2) \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{(\beta^2 \cdot \mathrm{precision}) + \mathrm{recall}}$$

where

$$\mathrm{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$\mathrm{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

Precision measures how many of the examples we classified as positive are truly positive, while recall measures how many of the positive examples we actually classified as positive.

F1 score is a version of the F-score where the β parameter is set to 1, which in turn simplifies the equation to:

$$\mathrm{F}_1 = 2\frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

It can be described as the harmonic mean between the precision and recall metrics and is often used as a more data agnostic version of those metrics.

4.1.3. Loss Function

As our model falls into the classification category, for the loss function, we used *cross entropy loss*, also know as *logistic loss*. In works by taking the *logits* our model outputs and comparing them against the true labels. Logits are vectors of size n, where n is the number of classes we're predicting for every data entry. For instance, if we have 3 examples in our dataset that we need to classify and 7 classes, the size of our matrix would be 7×3. This matrix is compared to the *one-hot* encoded classes of the data. One-hot encoded vectors are also of size n, but with only their true class for the example set to 1, and everything else to 0. They are compared with the following equation:

$$L = -\sum_{i=1}^{n} t_i \log(p_i)$$

where t_i is either a 1 or a 0 (truth label) for class i, and p_i is the probability for class i. If we are only using two classes (which we indeed are), we can simplify this to *binary cross entropy loss* represented with the following equation:

$$L = -[t \log(p) + (1 - t) \log(1 - p)]$$

A smaller value is better, so by minimizing this loss, we can train our model. Due to its logarithmic nature, it heavily penalizes values straying from 0, which is the ideal value of the function [20].

It is also worth noting that the loss function doesn't take logits as input directly, but class probabilities instead, obtained by passing the logits through a softmax function (section 1.3.3).

4.2. Experiments on One Chromosome

The results of the experiments we performed on one chromosome are presented through three graphs: accuracy, F1 score and loss. We will only display validation graphs, as training graphs are often an unreliable indicator of performance due to them almost certainly overfitting. In addition, we will only display a few different experiments we performed for clarity, even though there were many more. Lastly, we only display the first 200 epochs of training, as all of the models reached their peak performance up to that point.

4.2.1. Accuracy & F1 Score

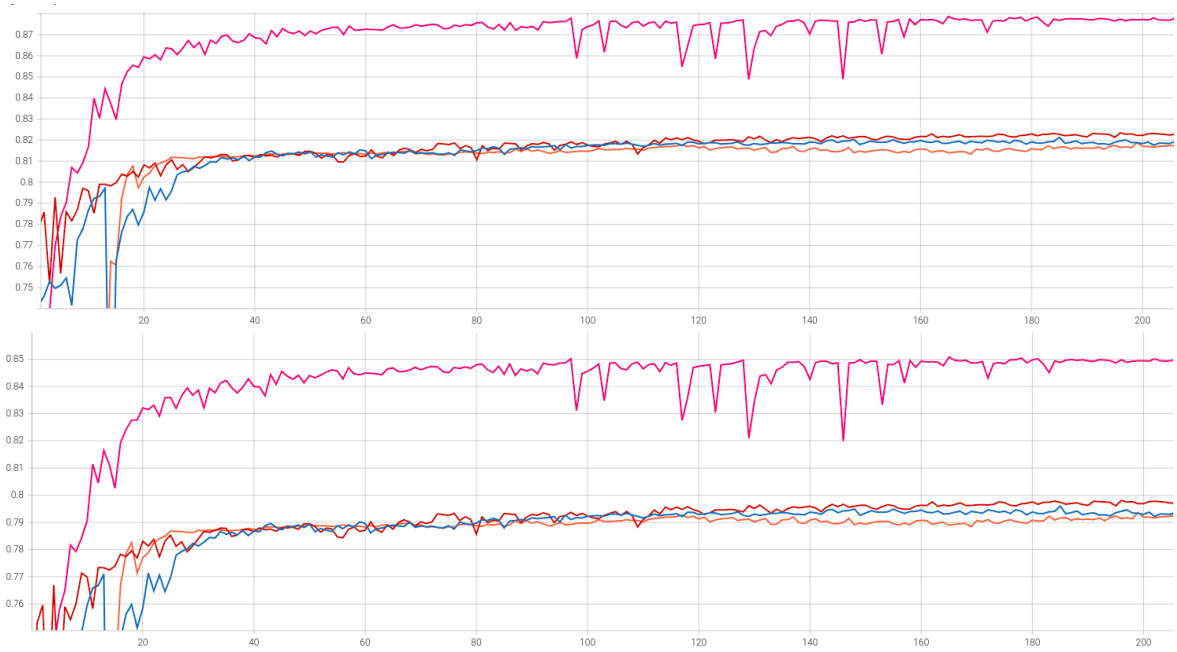

Figure 4.1: Two graphs representing the model training performance on the validation part of the dataset. Above, we have the accuracy graph and below it, we can see an F1 score graph. Epochs are displayed on the x axis, while on the y axis, we have the respective performance metric of either accuracy or F1 score. The graphs were obtained using TensorBoard (section 3.1).

The first two graphs we are going to look at are an accuracy graph and an F1 score graph

(Figure 4.1 top, Figure 4.1 bottom). We are going to analyze both of them at the same time due to their similarity. There are four model results displayed here. In red, a GATv2 model (section 1.3.2) with dropout set to 0.5 and edge overlaps used as edge weights in the GCN layers is displayed. In blue, we have the same model, but without using edge overlaps. Their performance is nearly similar, proving that this way of using edge overlaps is rather inefficient. In orange, we use GCN model (section 1.3.2) with dropout set to 0.2 and edge overlaps used. We can see that it is of a very similar performance, meaning that the GAT layers extract little additional information from the model.

The standout model here is the pink model. It has greatly outperformed the rest of the models, with an accuracy of 87.9%. This is because here we used an EGAT model (section 1.3.2), which uses edge features much more effectively by incorporating them into the attention mechanism. However, we have to note that we had to use a much smaller model here due to slow training times. Nevertheless, it proved effective for the task at hand.

4.3. Experiments on the Whole Genome

In addition to accuracy, F1 score and loss, we also present recall and precision. These additional metrics might provide clarity in how the model trains on the provided data. The model we used for training was the EGAT model (section 1.3.2) with default parameters.

4.3.1. Accuracy, F1 Score & Loss

In Figure 4.2 we can see the graphs for accuracy (top), F1 score (middle) and model loss (bottom). We group these three graphs together due to them showing similar properties. Accuracy and F1 score exhibit almost the same graphs, while model loss is essentially an inverted version of the same graph. They are also grouped together due to them showing a nice example of overfitting. The model reaches its best performance fairly early during traing, around epoch 100, and then continues to progressively exhibit worse and wrose performance on the vaidation dataset. The best model had an F1 score of 0.8024, accuracy of 0.7911 and loss of 0.3868. This is not far off from the best performing model on only one chromosome, showing that different chromosomes exhibit similar features which our model could take leverage of.

4.3.2. Precision & Recall

In Figure 4.3 we can see the graphs for precision (top) and recall (bottom) for the validation part of the entire genome dataset. An interesting observatin we can immediately see is that at the beggining of training, the model has a near perfect recall with a fairly lower precision

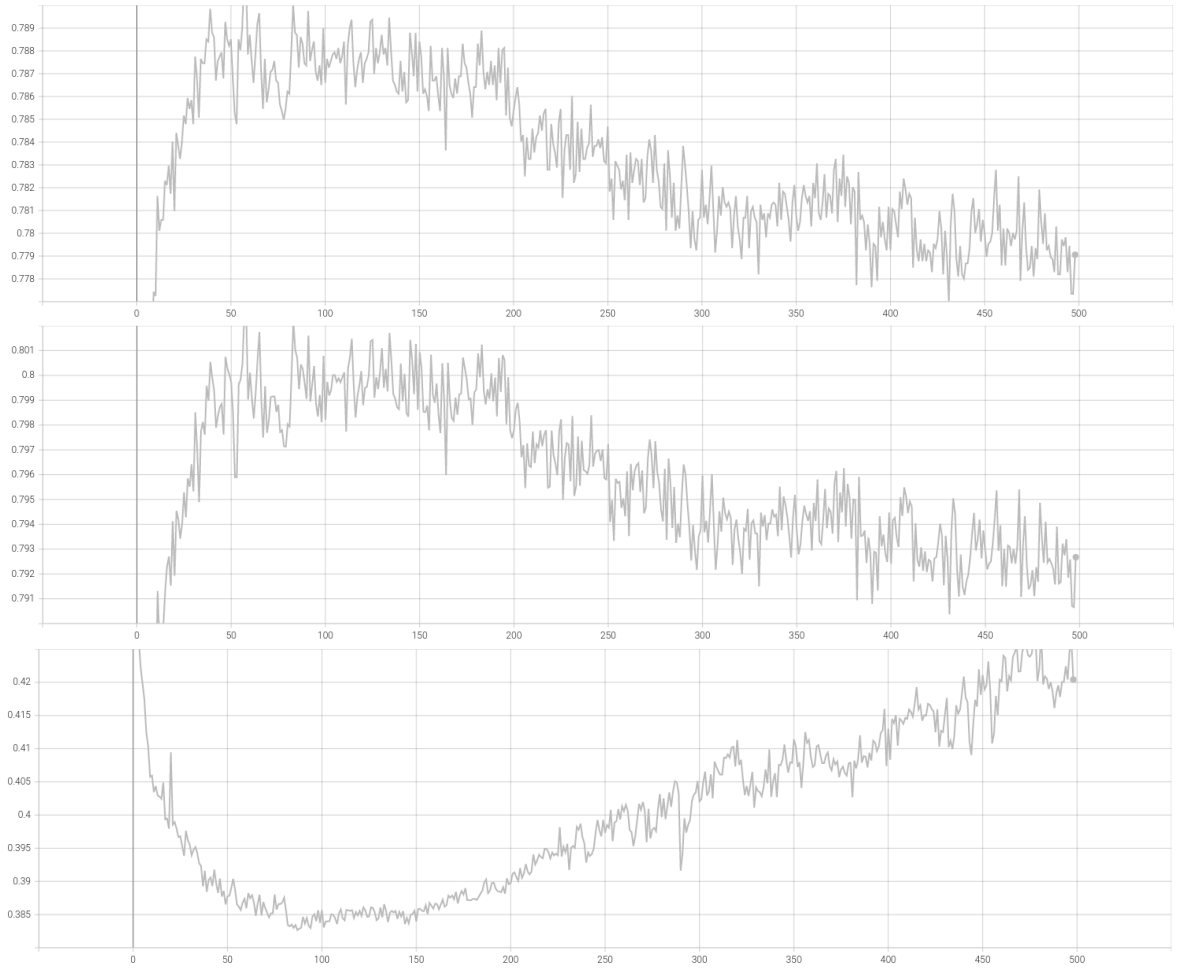

Figure 4.2: Graphs representing the accuracy (top), F1 score (middle) and model loss (bottom) for the validation part of the entire genome dataset. The model was trained for 500 epochs. The graphs were obtained using TensorBoard (section 3.1).

score. This means that the model classifies most positive examples in the dataset as indeed positive, while classifying some negative examples incorrectly. As training continues the model starts trading recall for a higher precision value, and this carries on for the entirery of the 500 epochs. The model essentialy tries to balance these two metrics as best as it can, with the optimal balance found somewhere araound epoch 100. For that balance, recall stands at 0.9368, while precision is 0.7389. The dataset is of course not perfectly balanced, and this is somewhat reflected in these numbers, but it is also a testament to the fact that the model is simply better at finding incorrect edges, while simultaneously classifying some correct edges as incorrect. Balancing these parameters depends on the use case for such an application, but we can generalize that the model is quite succesful at separating haplotypes, but the haplotypes it separates will be somewhat incomplete.

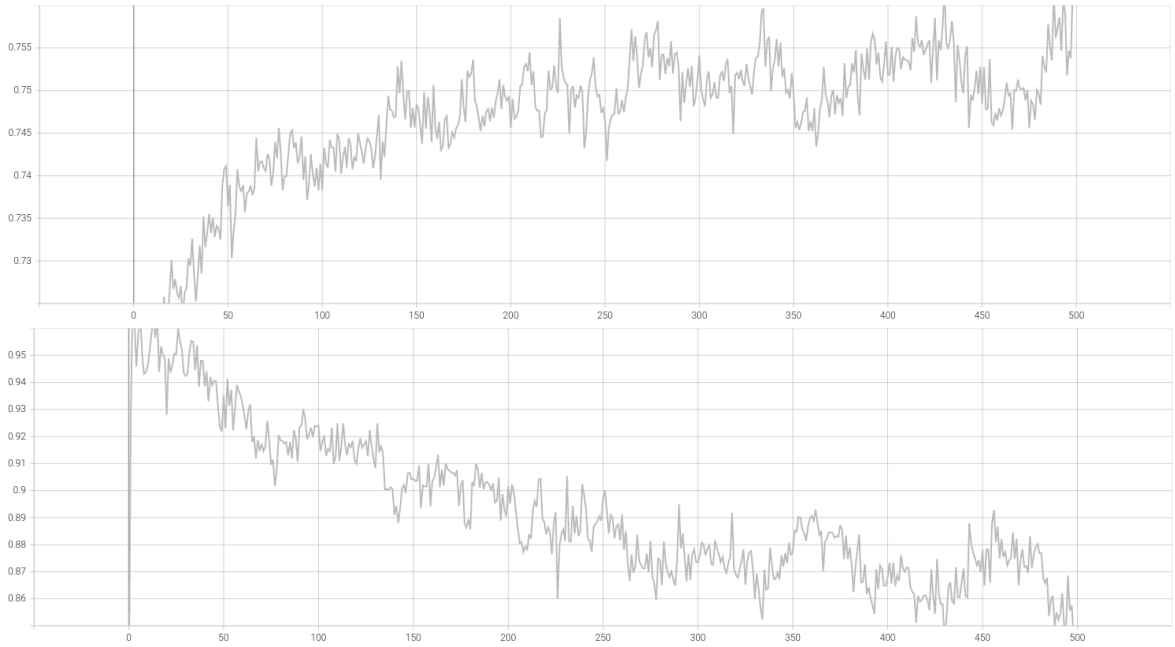

Figure 4.3: Graphs representing the precision (top) and recall (bottom) for the validation part of the entire genome dataset. The graphs were obtained using TensorBoard (section 3.1).

5. Conclusion

BIBLIOGRAPHY

[1] *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv, 2012.

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

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

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

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

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

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

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

[9] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2015.

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

[11] 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.

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

[13] W. L. Hamilton. 2020.

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

[15] Illumina. Assembling novel genomes.

[16] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

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

[18] 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.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

[20] K. E. Koech. Cross-entropy loss function, 2020.

[21] 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.

[22] 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.

[23] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

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

[25] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, pages 807–814, 2010.

[26] NCBI. Homo sapiens.

[27] NCBI. Saccharomyces cerevisiae.

[28] I. Neutelings. Neural networks.

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

[30] 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.

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

[32] S. Park. A 2021 guide to improving cnns-optimizers: Adam vs sgd, 2021.

[33] 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

[34] 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.

[35] Scitable. haplotype.

[36] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

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

[38] 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: Bioinformatics, Graph Neural Networks, GNN, Haplotype Separation, Deep Learning, Machine Learning

Korištenje graf neuronskih mreža za odvajanje haplotipa u grafovima astavljanja

Sažetak

Ključne riječi: Bioinformatika, Graf Neuronske Mreže, GNN, Odvajanje haplotipa, Duboko Učenje, Strojno Učenje