Predicting Satisfiability in SAT3 and Graph Coloring Problems

using Graph Transformers Networks (GTNs) and Long Short-Term Memory Networks (LSTMs)

Course: Deep Learning

Author: *Tatiana Boura*

Instructor: *Theodoros Giannakopoulos*
Date: June 2023

Contents

1 Introduction

A *satisfiability problem (SAT problem)* is a family of problems, where given a Boolean expression written using only the logical connectives AND, OR and NOT, variables and parentheses, we examine if there is some truth-assignment to the variables that will make the entire expression true. A **3SAT** problem is a special case of SAT problems, where the boolean expression should have a very strict form : the conjunctive normal form (CNF). A CNF is a conjunction of one or more clauses, where a clause is a disjunction of literals; in other words, it is a sequence of ANDs of ORs. In the 3SAT problem, every disjunction consists of 3 literals. An example of a 3SAT problem instance is the following,

$$(x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_3 \vee x_4 \vee x_2)$$

The 3SAT problem is one of the Karp's 21 NP-complete problems and is used as the starting point to prove that the other problems (graph coloring, clique detection, dominating set, vertex cover problems etc.) are also NP-Complete. These problems can be converted to one another in polynomial time.

Many algorithms have been developed in order to prove the (un)satisfiability of the 3SAT problems and point a truth assignment for their variables. However, all of them have an *exponential worst time complexity* as a function of the problems's size. The state-of-the-art solvers can solve many large instances using heuristics, but still very large instances cannot be solved. For this reason, some researchers (e.g. [1], [2], [3]) started exploring Machine Learning and, more specifically, Deep Learning methods in order to solve the problem, as these algorithms are capable of generalizing and possibly perform well on instances that classic solvers cannot handle. Most deep learning methods that were implemented for this cause use different Graphical Neural Network (GNN) architectures and have very promising results. In this assignment we will be trying **Graph Transformers**, since they have incorporated label propagation and allow us to use edge attributes. Nonetheless, most papers focus on general satisfiability problems and not on 3SAT problems that we will study in this assignment, so the results you will see here apply to the latter. Additionally, in this assignment our focus is on predicting the satisfiability of an instance and not on finding a truth assignment of the variables. The process of translating the problem to a graph, the selection of the model, its training and tuning as well as the performance of it on different evaluation sets will be discussed in *Section 2*.

Other than the graph formulation of the problem, one interesting approach is to view the problem as a sequence of clauses. Formulating the data as a series allows us implement a **Long Short-Term Memory Network (LSTM)** to predict the satisfiability of the instance. In *Section 3* we do exactly that and discuss the process thoroughly. Also, we compare the two methods and try to explain why the Graph Transformer architecture has an overall better performance.

As already mentioned, there is a connection between 3SAT problems and others belonging to the same family. The graph coloring problem is one of them. A **3-coloring problem** for undirected graphs is an assignment of colors to the nodes of the graph such that two adjacent vertices have a different color assigned to them and at most 3 colors are used to color the graph. In *Section 4* we present the pre-trained best model that solves their problem using transfer learning and discuss its results.

2 Solving the 3SAT problem using a Graph Transformer Network

In this section, we present a Graph Transformer model that predicts the satisfiablitiy of 3SAT problems. For this to happen, we first illustrate how the problem can be modelled as a graph, then we introduce GNN's broad family of neural networks and after that we dive into the training process, which includes the model's architecture, the selection of the optimal model and conclude with the illustration of the model's performance on two different evaluation sets.

2.1 Problem formulation as a graph

There are many ways that a 3SAT instance is modelled as a graph. The modeling that was chosen in the implementation is the one suggested in [1]. More specifically, given a problem instance with k clauses and n variables, we construct the undirected graph $G = (V, E)$ where the vertices V are the literals (the variables and their negation) and the clauses of the CNF. This means that $|V| = 2 \cdot n + k$. Concerning the edges, there are two types of them: the first one connects each variable with its negation and the second one connects each literal with every clause it appears in. For example, in Figure 1 is illustrated the modeling of the problem instance $S = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_3 \vee \neg x_1 \vee x_2)$ as a graph after following the previous instructions.

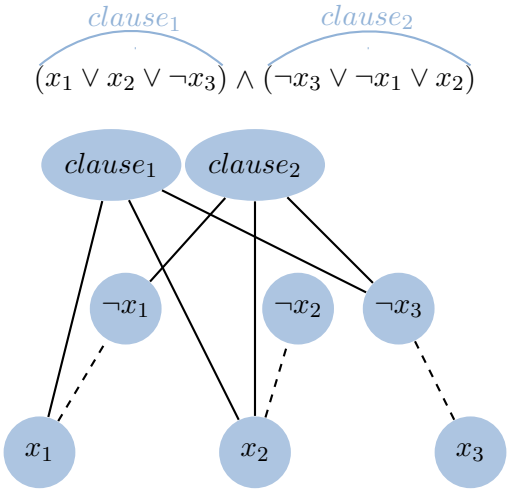


Figure 1: An example of modeling a 3SAT problem instance as an undirected graph

Note that another modeling we tried was introducing a third type of edges; one that connects the clauses with each other. In spite of that, this addition not only did not result in a better performance of the model, but produced very dense graphs that needed a large amount of memory to store and process.

2.2 Graph Transformer Networks for the 3SAT problem

Graph neural network (GNN) and *label propagation algorithm (LPA)* are both message passing algorithms, which have achieved superior performance in semi-supervised classification. GNN performs feature propagation by a neural network to make predictions, while LPA uses label propagation across graph adjacency matrix to get results. In this assignment we use a **Graph Transformer Network (GTN)** that can incorporate feature and label propagation at both training and inference time. In this section we briefly present these networks and explain the reason they were chosen for our classification task.

2.2.1 GNN

The GNN formalism [4–8] is a general framework for defining deep neural networks on graph data. The GNN's main idea is that it generates representations of nodes that depend on the structure of the graph, as well as any feature information that comes along with the problem. The defining feature of a GNN is that it uses a form of neural message passing in which vector messages are exchanged between neighboring nodes and updated using neural networks. Simply put, the input of the model is graph $G = (V, E)$, along with a set of node features and the generated output is the node embeddings.

2.2.2 LPA

The LPA [9–14] is a fast algorithm for finding communities in a graph. It detects these communities using the network's structure as its guide and does not require a pre-defined objective function or prior information about the communities. This algorithm propagates labels throughout the network and forms communities based on this process of label propagation. The intuition behind it is that a single label can quickly become dominant in a densely connected group of nodes, but will have trouble crossing a sparsely connected region. Labels will get

trapped inside a densely connected group of nodes and those nodes that end up with the same label when the algorithm finishes can be considered part of the same community.

2.2.3 Graph Transformers

A Transformer model is a neural network that learns context by tracking relationships in sequential data. Transformer models apply an evolving set of mathematical techniques, called attention, in order to detect subtle ways even distant data elements in a series influence and depend on each other. In [15], the authors infuse a technique (a layer), the vanilla multi-head attention proposed in [16, 17], into graph learning with taking into account the case of edge features. This layer is a *Graph Transformer*, that combines the label propagation by simply adding the node features and label vectors together as propagation information, which results in the unification of both label propagation and feature propagation within a shared message passing framework. Neural networks that use graph transformer layers are called **Graph Transformers Networks**.

2.2.4 Why GTNs for the 3SAT problem?

GNNs have been widely used in representation learning while achieving state-of-the-art performance in various tasks. However, most existing GNNs are designed to learn node representations on fixed and homogeneous graphs. Those limitations become problematic when learning representations on non-stationary or heterogeneous graphs that consist of various types of nodes and edges. GTNs, on the other hand, are capable of generating new graph structures, which involve identifying useful connections between unconnected nodes on the original graph, while learning effective node representation on the new graphs in an end-to-end fashion. *In our case, the edges connecting the literals with one another and the edges connecting the literals with the clauses differ from each other, making the graph of the problem instance heterogeneous*. For this reason, a typical GNN architecture is not optimal for predicting the satisfiability of a problem instance.

2.3 Classification

After describing the problem formulation, let us emerge into the training process that includes the data processing, the model's architecture and the selection of the optimal model via hyperparameter tuning.

2.3.1 Dataset

For this classification task, the acquired dataset is actually a benchmark dataset where classic solvers are tested on. It is found in SATLIB - Benchmark Problems. More specifically, the chosen dataset is the **"Uniform Random-3SAT"**. As described in the dataset's page, Uniform Random-3SAT is a family of SAT problems distributions obtained by randomly generating 3-CNF formulae in the following way: For an instance with n variables and k clauses, each of the k clauses is constructed from 3 literals which are randomly drawn from the $2n$ possible literals such that each possible literal is selected with the same probability. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological i.e. they contain $x_i \vee \neg x_i$ for any of the n variables. Each choice of n and k thus induces a distribution of Random-3SAT instances. Uniform Random-3SAT is the union of these distributions over all n and k. For generating satisfiable instances, one possibility is to randomly determine a variable assignment B at the beginning of the generation process. Then, the clauses are generated as before, but only clauses which have B as a model are accepted for the formula to be constructed.

The *total number of instances are 6401*, where 3701 are satisfiable (positive class) and the rest 2700 are unsatisfiable (negative class), making the dataset slightly imbalanced.

2.3.1.1 Torch.geometric format

To implement the GTN in PyTorch, the data needs to be in the format described in torch_geometric.data.Data. Specifically, this structure constructs a homogeneous graph with nodes that have attributes and edges that are indexed and have features. For our dataset, since the nodes (literals and clauses) do not have characteristics (other than the nodes possibly having a truth value that we do not know) we made a simple choice : each

variable is assigned a value v from the uniform distribution in range $[-1, 1)$, the negation of the variable is assigned the value $-v$ and every clause-node has a value of 1. Concerning the edges, we mentioned before that we have two types of edges, the ones connecting literals and the ones connecting literals to clauses. However, the graph formatting we discuss does not support heterogeneous graphs. Luckily, though, it supports edge attributes, which means that different edges can be modelled by using two edge attributes a_1 and a_2 for every edge. If the edge connects the variable to its negation, then the values of its edge attributes are $a_1 = 1$ and $a_2 = 0$, otherwise its edge attributes are $a_1 = 0$ and $a_2 = 1$. A visual example of this format is provided in Figure 2 , where the problem instance in Figure 1 is altered to have the defined structure. Note that in this example the node values have been rounded at 2 decimal points.

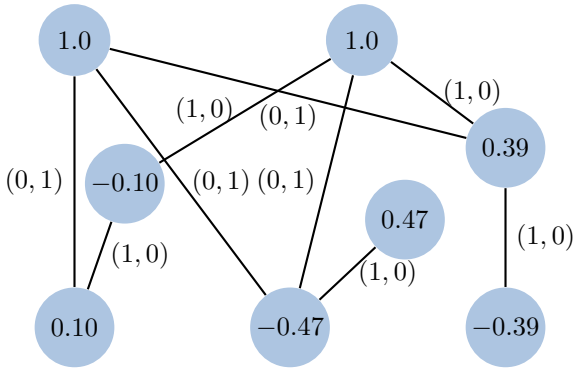


Figure 2: The example seen in Figure 1 as requested by torch_geometric

2.3.2 Model selection

The final selected model was the one with a specific architecture and training parameters tuned for the optimization of its performance. Here, we describe the process that led to selecting the presented model.

2.3.2.1 Model architecture

After experimenting with some architectures, the chosen one is the following.

- Firstly, there is a **repeated block** of 3 layers:
  1. A *graph transformer* (TransformerConv), followed by
  2. A *linear* layer that goes through a *ReLu* activation function, followed by
  3. A *batch normalization* layer
- After this sequence of blocks are 2 linear layers each one followed by a ReLu activation function and a final linear layer that produces the final output.

The number of repeated blocks, as well as the input and output size of each layer (except for the input and output layer) are decided after performing the hyperparameter tuning.

2.3.2.2 Training process

During the processing of the data, the dataset is randomly split into *train* (80%) and *evaluation* sets (20%). During the training process, the train set is split into validation (20%) and training (80%) set. The training set is used for training the algorithm, whereas the validation set is used in order to assure that the selected model is not overfitted and is more likely to generalize well for unseen data.

More accurately, the avoidance of overfitting is achieved using the regularization mechanism of *early stopping*. In early stopping, the training consists of *at most* max_epochs epochs. The training can be terminated when the model performance stops improving on the hold out validation dataset. More specifically, if the difference between the validation and the training loss is not reduced after $stopping\_counter$ number of epochs, then

the final model is the one with the minimum validation and training loss difference. In our implementation $max\_epochs = 50$ and $stopping\_counter = 15$. The $stopping\_counter$ is higher than normally (8-10 epochs), as the validation loss converges slowly. Figure 3 illustrates the aforementioned training and validation losses for the final selected model.

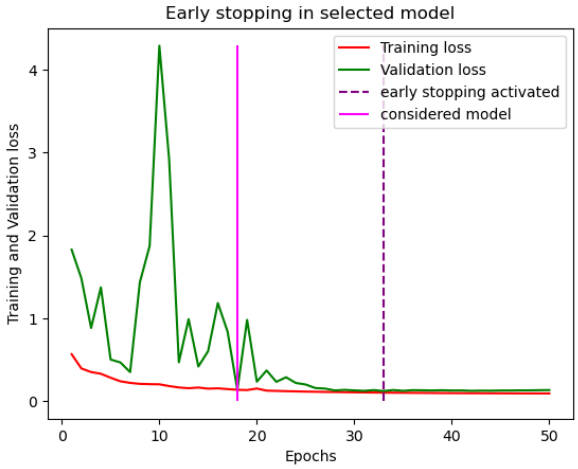


Figure 3: Training and Validation errors during training of GTN

As for the rest of the training requirements, we used the ones seen in Table 1. The l**loss function** we use is `BCEWithLogitsLoss` as it is numerically stable and combines a sigmoid layer and the binary cross entropy in one single class. It also models the dataset's imbalance through its argument *pos_weight*. The **optimizer** is the well-known Adam and the **scheduler** is `ExponentialLR`, which decays the learning rate of each parameter group by gamma every epoch.

Loss Function	`torch.nn.BCEWithLogitsLoss`
Optimizer	`torch.optim.Adam`
Scheduler	`torch.optim.lr_scheduler.ExponentialLR`

Table 1: Requirements for training

2.3.2.3 Hyperparameter tuning

In the previous sections, we presented the model's architecture and the training process. Both of them consist of many parameters that need to be tuned accordingly in order to achieve a good algorithm performance. In this assignment, we tried many combinations of parameters, for both *model's architecture* and *training process* and selected the one with the minimum validation loss. In Table 2 are presented the tunable hyperparameters and their explanation alongside with the values they are assigned with, as well as the final selected parameters that resulted in the minimal validation loss.

So, the architecture of the final selected model is the following:

```
GNN(
(conv1): TransformerConv(1, 64, heads=1)
  (transf1): Linear(in_features=64, out_features=64, bias=True)
  (bn1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_layers): ModuleList(
    (0): TransformerConv(64, 64, heads=1)
  )
  (transf_layers): ModuleList(
    (0): Linear(in_features=64, out_features=64, bias=True)
  )
  (bn_layers): ModuleList(
```

Model parameters			
Parameter	Possible Values	Selected Value	Explanation
embedding size	$\{64,\ 128\}$	64	Number of I/O samples passed through the Graph Transform network
attention heads	$\{1\}$	1	Number of multi-head-attentions in Graph Transform
number of layers	$\{2,\ 3,\ 4\}$	2	Number of repeated blocks
dropout rate	$\{0.1,\ 0.3,\ 0.5\}$	0.1	Dropout probability of the normalized attention coefficients in Graph Transform
dense neurons	$\{128,\ 256\}$	128	Number of I/O samples passed through the final linear layers
Training parameters			
Parameter	Possible Values	Selected Value	Explanation
batch size	$\{32,\ 64\}$	64	The number of training examples in one forward/backward pass
learning rate	$\{0.001,\ 0.01,\ 0.1\}$	0.01	Step size at each iteration
weight decay	$\{0.00001,\ 0.0001,\ 0.001\}$	$1e-05$	Weight decay (L2 penalty)

Table 2: Parameters to be tuned (GTN - same distribution)

```
    (0): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (linear1): Linear(in_features=128, out_features=128, bias=True)
  (linear2): Linear(in_features=128, out_features=64, bias=True)
  (linear3): Linear(in_features=64, out_features=1, bias=True)
)
```

2.3.3 Evaluation

The performance of the selected model was tested on a test set using several metrics.

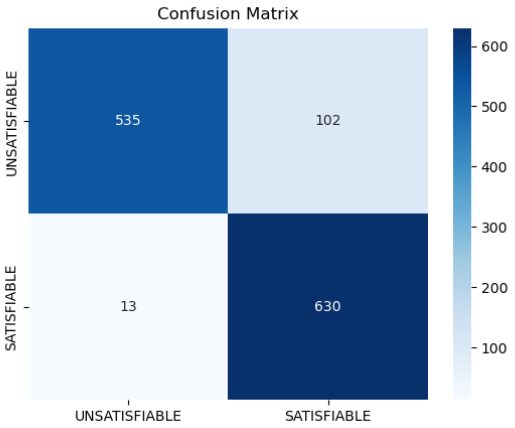
2.3.3.1 Test set from the same distribution

At first, the test set was the one described in Section 2.3.2.2 and comes from the same distribution as the train set. This means that, during its training the model had seen 3SAT problems of equal size and number of variables as the ones in the testing set. The algorithm's performance on the test set regarding the performance metrics is seen in Table 3. Here, the *test-set loss* is *0.131916*.

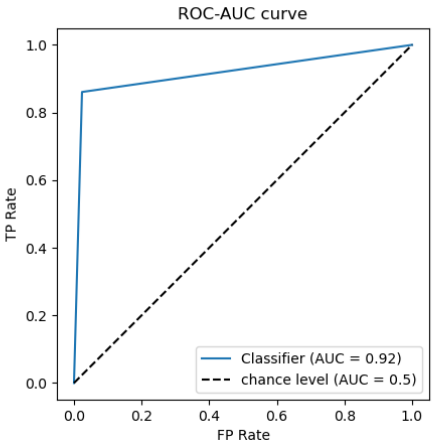
Performance	
Metric	**Value**
f1-score	0.9164
accuracy	0.9102
precision	0.9798
recall	0.8607
roc-auc	0.9185

Table 3: Performance on the test set (GTN - same distribution)

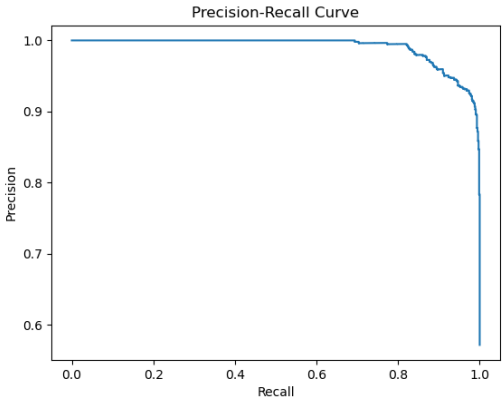
Additionally, in Figure 4 are illustrated the *confusion matrix*, *ROC curve* and the *Precision-Recall curve*.



(a) CM



(b) ROC curve



(c) PR curve

Figure 4: confusion matrix (CM), ROC curve and Precision-Recall(PR) curve (GTN)

Overall, the model performs very well on the test set. The only metric that is evaluated less than 0.90 is recall. Let us remind that *precision* identifies the proportion of positive identifications that was actually correct, whereas *recall* identifies the proportion of actual positives that was identified correctly. In this application what interests us is high precision as we do not want to label unsatisfiable problem instances as satisfiable, since no matter how hard we look we will not find any truth values that classify this problem as solvable. Given the precision and recall tradeoff (Figure 4c), in order to have the aforementioned high precision we do not mind classifying some satisfiable instances as unsatisfiable. Note that the *baseline* for the PR-curve is about 0.58.

It is worth mentioning that given the *roc-auc* metric and the plot 4b, the model learns to separate the two problem-classes well.

2.3.3.2 Test set from a different distribution

Starting this report we mentioned that the 3SAT problem is NP-complete. NP-complete is a class of problems for which no efficient solution algorithm has been found. In simpler words, this means that the current algorithms cannot solve large problem instances . So, it makes sense to train neural networks to solve this problem, given that these models generalize well for unseen inputs and, maybe, can solve problem instances larger than those solved by standard algorithms with efficient heuristics.

In order to explore this possibility, from the initial dataset we excluded the largest problems (250 vars and 1065 clauses) and trained the model with the rest data. With this change, the size of the train set is 6200 and the size of the test set is 200. The training and tuning process was the same one followed earlier and resulted in a model

with parameters as seen in Table 4.

Model parameters	
Parameter	Selected Value
embedding size	64
attention heads	1
number of layers	3
dropout rate	0.1
dense neurons	128
Training parameters	
Parameter	Selected Value
batch size	32
learning rate	0.001
weight decay	0.0001

Table 4: Tuned parameters (GTN - different distribution test set)

The evaluation of the model on the latter test set using the same metrics as before is seen in Table 5. The confusion matrix is presented in 5. The *test-set loss* is *0.573288*.

Performance	
Metric	**Value**
f1-score	0.5525
accuracy	0.5950
precision	0.6173
recall	0.5000
roc-auc	0.5950

Table 5: Performance on the test set (GTN - different distribution)

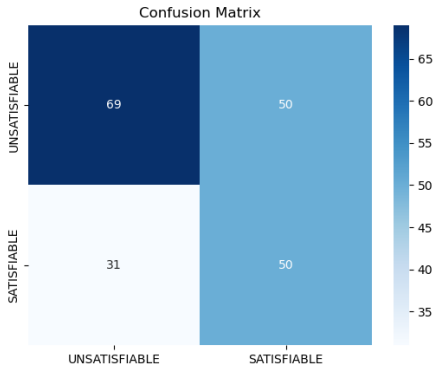


Figure 5: Confusion matrix (GTN - different distribution test set)

The performance of the selected model on the unseen data was not optimal. Most metrics are near the baseline, i.e. accuracy is 0.5950 when the baseline is *0.58*.

3 Solving the 3SAT problem using a LSTM

In the previous sections, we formulated the problem as a graph and, therefore, were able to use a neural network that handles graph representations to solve the classification problem satisfactorily. However, there is another way of viewing the data : as a **sequence**. This modeling allows the problem to be solved using a **recurrent neural network**. So, here we present this new formulation, the process of training and tuning the model and the results of this approach.

3.1 Sequential formulation of the problem

Since the sequential modeling of the problem is possible due to the dataset in our hands, let us formulate the problem in alignment to the dataset.

The used dataset was the same one described in Section 2.3.1. As referred by the Dataset Documentation, for all problem instances there is a clause number-threshold after which the problem is highly unlikely to be satisfiable. Before this threshold, the problem is most likely to be satisfiable. Hence, we are provided a theoretical way to transform our static data into a sequence: for any problem instance with k-clauses one can create a k-sized timeseries corresponding to that instance, by starting from one clause (3 literals) and labelling it as satisfiable and gradually adding clauses until the threshold is reached and the label is changed to unsatisfiable. Of course this data augmentation process is not infallible, but it is an accepted one.

However, since it is computationally very costly to perform the aforementioned sequencing of the data, in this assignment we did not create k-length-sequence data, but 2-sequence-length data as follows :

1. For each instance is provided a satisfiable smaller version of it (the instance below the threshold) and
2. For each unsatisfiable instance is provided another unsatisfiable instance (the instance just above the threshold)

For example, consider the **satisfiable** instance,

$$S_1 = (\neg x_{31} \vee x_4 \vee x_5) \wedge \cdots \wedge (x_{45} \vee \neg x_{25} \vee \neg x_{89}) \wedge (x_{57} \vee x_{90} \vee x_{23})$$

Firstly, every variable is assigned a value v from the uniform distribution in range $[-1, 1)$, the negation of the variable is assigned the value $-v$ and the connectives are ignored. The datum is now displayed as a simple sequence of numbers,

$$S_1 = 0.9596, 0.2322, 0.6809, 0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, -0.9862, -0.3984$$

Then, we compute the threshold after which the instance is probably unsatisfiable and marked it with cyan. The green area is the area were the instance is satisfiable given the threshold.

$$S_1 = 0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, -0.9862, -0.3984$$

Given the threshold we now compute the series T_1 of sequence length two:

$$\begin{aligned} T_1 = (T_{11}, T_{12}) = ((&0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, 0.0, 0.0, 0.0), \\ &(0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, -0.9862, -0.3984)) \end{aligned}$$

where both T_{11} and T_{12} are satisfiable, making the series T_1 satisfiable.

Now, let's suppose the **unsatisfiable** instance,

$$S_2 = (\neg x_{31} \vee x_4 \vee x_5) \wedge \cdots \wedge (x_{45} \vee \neg x_{25} \vee \neg x_{89}) \wedge (x_{57} \vee x_{90} \vee x_{23})$$

Following the same process and marking the probably unsatisfiable region red,

$$S_2 = 0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, -0.9862, -0.3984$$

we can now create two series T_2 and T_3 of sequence length two each.

$$\begin{aligned} T_2 = (T_{21}, T_{22}) = ((&0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, 0.0, 0.0, 0.0), \\ &(0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, 0.0, 0.0)) \end{aligned}$$

where T_{21} is satisfiable and T_{22} is unsatisfiable (making the series T_2 unsatisfiable) and

$$\begin{aligned} T_3 = (T_{31}, T_{32}) = ((&0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, 0.0, 0.0), \\ &(0.0628, 0.9996, 0.8762, \ldots, 0.5678, 0.8305, 0.4795, -0.9489, -0.9862, -0.3984)) \end{aligned}$$

where both T_{31} and T_{32} are unsatisfiable, making the T_3 unsatisfiable.

3.2 Classification

In the next sections we present the model that was used for the prediction of the satisfiability by analyzing its architecture and the training process along with the hyperparameter tuning. After that we evaluate the model on a test set and comment on its performance.

3.2.1 Dataset

The dataset and its use was thoroughly explained earlier in this report. In order to train and evaluate the model, the dataset is split randomly at the preprocessing stage into training (80 %), validation (10 %) and evaluation (10 %) sets. The total number of satisfiable instances (series of sequence-length 2) is 3701 instances, whereas the total number of unsatisfiable instances is 5400, making the dataset imbalanced.

3.2.2 Model selection

Since, we modelled the data as a series, for its handling we will be using a network designed to handle these data; specifically a Long Short-Term Memory Network (LSTM). Again, for the selection of this model many networks were trained using specific hyperparameters and the final one was selected due to its minimal validation set error. The next paragraphs are dedicated to presenting the aforesaid procedure.

3.2.2.1 Model architecture

LSTM is a deep learning, sequential neural network designed by Hochreiter and Schmidhuber in [18], that allows information to persist. It is a special type of *Recurrent Neural Network (RNN)* which is capable of handling the vanishing gradient problem faced by RNN.

In this project, the network's architecture was based on one or more LSTM layers. Two architectures were tried, however the simpler one was utilized, as the data in our disposal was not substantial in amount and a more complex model was a highly overfitted model.

So, the selected model has the following structure,

- One or more stacked LSTM layers, followed by
- A *linear* layer to produce the output

The number of stacked LSTMs and the input and output size of each layer (except for the input and output) are selected during the hyperparameter tuning.

3.2.2.2 Training process

The training process is the same one described in Section 2.3.2.2. In this implementation $max\_epochs = 50$ and $stopping\_counter = 10$. Also, some models were overfitted from the start, so we decided to only accept the models that have been trained for at least 4 epochs. In Figure 6 are illustrated the training and validation errors along with the final selected model.

3.2.2.3 Hyperparameter tuning

Just like in Section 2.3.2.3, the tunable parameters were the ones regarding the model's architecture, as well as the ones relevant to the training process. In Table 6 are shown and explained the tuned hyperparameters, the values we tried and the parameters that resulted in the final model.
So, the architecture of the final selected model is the following:

```
ShallowLSTM(
  (lstm): LSTM(3205, 32, batch_first=True)
  (linear): Linear(in_features=32, out_features=1, bias=True)
)
```

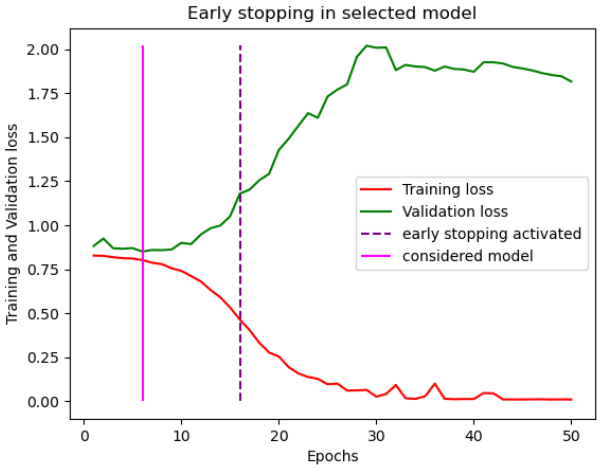


Figure 6: Training and Validation errors during training of LSTM

Model parameters			
Parameter	Possible Values	Selected Value	Explanation
hidden units	$\{8,\ 16,\ 32,\ 64\}$	32	Number of features in the hidden state of the LSTM layers
number of layers	$\{1,\ 3,\ 5\}$	1	Number of recurrent layers - stacked LSTMs
dropout rate	$\{0.0,\ 0.3,\ 0.5,\ 0.8\}$	0.0	Introduces a Dropout layer on the outputs of each LSTM layer except the last one
Training parameters			
Parameter	Possible Values	Selected Value	Explanation
batch size	$\{16,\ 32,\ 64\}$	16	The number of training examples in one forward/backward pass
learning rate	$\{0.001,\ 0.01,\ 0.05,\ 0.1\}$	0.05	Step size at each iteration
weight decay	$\{0.00001,\ 0.0001,\ 0.001\}$	0.001	Weight decay (L2 penalty)

Table 6: Parameters to be tuned (LSTM)

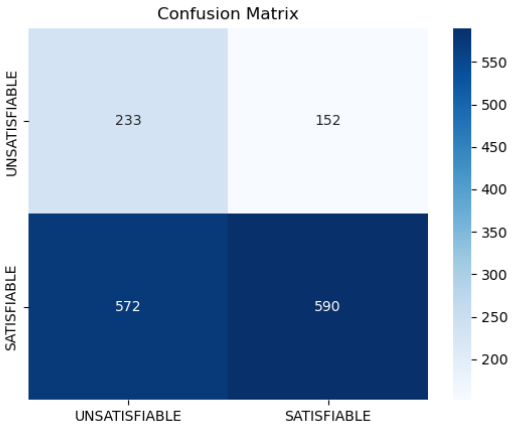
3.2.3 Evaluation

The performance of the selected model was tested on a test set using several metrics. The algorithm's performance on this evaluation set regarding the performance metrics are seen in Table 7. Here, the *test-set loss* is *0.840221*.

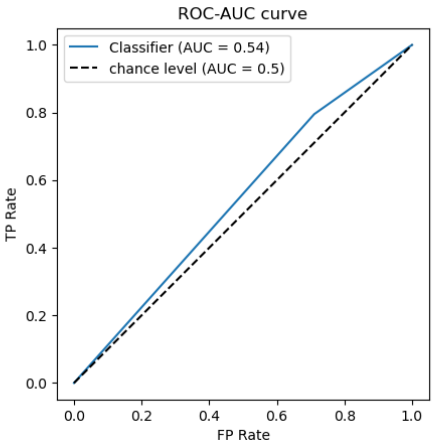
Additionally, in Figure 7 are illustrated the *confusion matrix*, *ROC curve* and the *Precision-Recall curve*.

Performance	
Metric	**Value**
f1-score	0.6197
accuracy	0.5320
precision	0.5077
recall	0.7951
roc-auc	0.5423

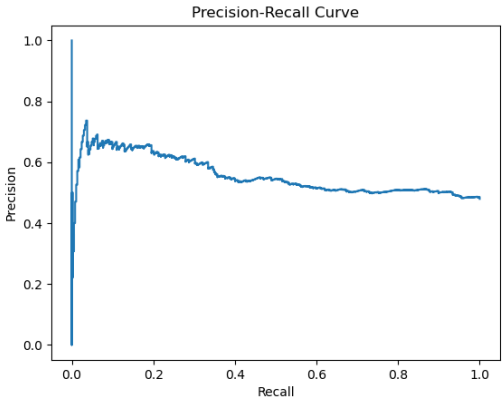
Table 7: Performance on the test set (LSTM)



(a) CM



(b) ROC curve



(c) PR curve

Figure 7: confusion matrix (CM), ROC curve and Precision-Recall(PR) curve (LSTM)

As illustrated by the above figures in 7, the LSTM-based architecture did not perform well. It surely performs better than random-guessing, as the baseline classifier would perform with accuracy *0.4* (this is also the baseline for the PR curve 7c), but its results cannot be compared with those of the GTN. This behavior is expected, as the LSTM needs way more data during its training, given the high dimensionality of the data.

Since the results are not very promising using this architecture under the current circumstances, the model is not tested on an evaluation from a different data distribution.

3.3 Comparison of the GTN model with the LSTM

It is obvious that the GTN architecture outperformed the LSTM one in the same classification task. This was expected for two main reasons. First of all, from the start we did not have high hopes for the LSTM architecture. As already mentioned, the LSTM needed to be trained with more data given the data's high dimensionality. Also, in order to achieve better results we could have followed the computationally costly process of data augmentation described in Section 3.1 in order to retrieve longer sequences. However, in retrospect these changes would not matter as the GTN performs very well with less effort and we would choose it over the LSTM nonetheless.

This leads us to the second reason we expected the graph based approach to be better: **the GTN architecture enforces both permutation invariance and negation invariance**. Permutation invariance essentially means that in the graph formulation of the problem, the semantically identical CNF clauses $C_1 = (x_2 \vee \neg x_1 \vee x_3) \wedge (\neg x_3 \vee \neg x_2 \vee x_1)$ and $C_2 = (\neg x_2 \vee \neg x_3 \vee x_1) \wedge (\neg x_1 \vee x_3 \vee x_2)$ result in the same

graph arrangement, despite that the clauses and the literals inside them are permuted. This does not hold true for their sequence modeling. Negation invariance indicates that the satisfiability of a formula is also not affected by negating every literal corresponding to a given variable, e.g. negating all occurrences of x_1 in $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ to yield $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$.

4 Solving the graph-coloring problem using the pre-trained Graph Transformer Network

In the previous sections, we efficiently predicted the label of many instances of the 3SAT problem. You may recall that in the beginning of this report we mentioned that there are other problems that belong to the same class as the SAT3 ones, i.e. the NP-complete class, and that there exists an efficient algorithm that transforms each problem to the other. One of those problems, is the **graph coloring problem**. So, since we do have a neural network that performs well on the 3SAT problem, in this section we try to use this model for predicting if a graph can be colored using at most 3 colors, by applying transfer learning.

4.1 Translating the 3-colorable graph problem to the SAT3 problem

The most effective way for us to explain the problem translation of the 3-colorable graph problem to the SAT3 problem is through an example. Suppose we have the graph in Figure 8 and we want to check if it is 3-colorable i.e. every vertex can have a different color from its neighbors and the total number of colors that is used is at most 3.

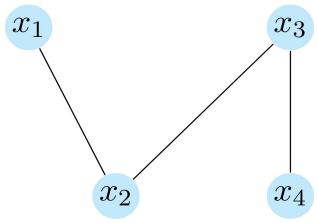


Figure 8: Graph to be colored

In order to transform this problem to a satisfiability one we follow the next steps by order:

1. We assign *3 logic variables for every vertex* that represent whether this vertex is colored using the corresponding color. For example, for vertex x_1 we have: x_{11} that is true if the vertex x_1 is colored using the first color and false otherwise, x_{12} that is true if the vertex x_1 is colored using the second color and false otherwise and x_{13} that is true if the vertex x_1 is colored using the third color and false otherwise.

2. Given these new variables, we have to write the logical expressions that will be used to evaluate the colorability of the graph. More specifically,

   - A vertex should be colored with *at least one color*. For example, for vertex x_1 this statement translated to the following expression,

     $$x_{11} \vee x_{12} \vee x_{13}$$

   - A vertex *cannot be colored with all 3 colors at the same time*. For example, for vertex x_1 this statement translated to the following expressions,

     $$\neg(x_{11} \wedge x_{12}) \wedge \neg(x_{11} \wedge x_{13}) \wedge \neg(x_{12} \wedge x_{13}) \overset{DeMorgan}{\equiv} (\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{11} \vee \neg x_{13}) \wedge (\neg x_{12} \vee \neg x_{13})$$

   - A vertex *cannot be colored with the same color as its neighbor*. For example, for vertex x_1 this statement translated to the following expressions,

     $$\neg(x_{11} \wedge x_{21}) \wedge \neg(x_{12} \wedge x_{22}) \wedge \neg(x_{13} \wedge x_{23}) \overset{DeMorgan}{\equiv} (\neg x_{11} \vee \neg x_{21}) \wedge (\neg x_{12} \vee \neg x_{22}) \wedge (\neg x_{13} \vee \neg x_{23})$$

3. Since all these expressions must hold true for all variables (x_1, x_2, x_3, x_4), we combine them using the logical-and connective in order to create a CNF clause. This final CNF is now the representation of the 3-colorable problem as a satisfiability one. It is not a 3SAT one by nature, but we can make it one, by simply adding tautologies to the clauses that have less than 3 variables.

4.2 Dataset

For this classification task, we used another benchmark dataset from the same repository, SATLIB - Benchmark Problems. This time, the chosen dataset is the **"'Flat' Graph Colouring"**. However, this dataset contains only 1700 (actually 1699) satisfiable instances and no unsatisfiable ones. Thence, we augmented the dataset and created unsatisfiable problem instances from the existing satisfiable ones simply by adding cliques of size 4 to each instance. Concretely, for every 3-colorable graph we randomly selected 4 vertices and connected them to one another, creating a clique of size 4. By doing so, the graph now needs at least 4 colors to be colored in a way that every vertex has a different color from its neighbors, which makes the 3-colorable hypothesis not satisfiable. After the data augmentation, the dataset became balanced and was handled the same way as in Paragraph 2.3.1.1.

4.3 Transfer learning

Given the direct connection of the 3SAT problem to the 3-coloring problem in theoretical computer science, the idea of **transfer learning** seems very relevant. Transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task. So, since we have trained a model that predicts pretty well the satisfiability of 3SAT problem instances, let us try to use this model to predict the satisfiability of the 3-coloring ones.

4.3.1 Using the pre-trained model directly

Performance	
Metric	**Value**
f1-score	0.0531
accuracy	0.5279
precision	0.3750
recall	0.0286
roc-auc	0.4937

Table 8: Pre-trained GTN for the graph coloring problem without re-training

If we test the already trained GTN model on a test set we excluded from the dataset, its performance in illustrated in Table 8. The loss is 0.911732.

Given the fact that the baseline for this data is 0.5, the results are not promising.

4.3.2 Re-training the model

Because the problems are similar and not identical using the pre-trained model did not result in a good performance. Nonetheless, in the concept of transfer learning the re-purposed model is rarely used without alteration, i.e. re-training some of its layers. More specifically, depending on the amount of data for the new problem and the similarities between both problems, in transfer learning we re-train layers of the model starting from the final one and going backwards. The layers that are not altered, are called "frozen layers".

In our case, we froze layers gradually, tuned the learning parameters, trained the model each time and logged the results. Regarding the results, the best ones are achieved when only re-training the final layer. If we unfreeze any other layer other that the final linear one and retrain them, the model has a suboptimal performance. Indicatively, if we unfreeze only the 2 final linear layers, the model achieves $f1 - score = 0.6332$, $accuracy = 0.4632$, $precision = 0.4632$, $recall = 1.0$ and $roc - auc = 0.5$.

In Table 9 are presented the tunable training parameters and the selected ones in the re-training of the final linear layer of the classifier. The results after the transfer learning are shown in Table 10. The loss here is equal to 0.695687.

Training parameters		
Parameter	Possible Values	Selected Value
batch size	$\{16,\ 32\}$	16
learning rate	$\{0.001,\ 0.005,\ 0.01,\}$	0.001
weight decay	$\{0.00001,\ 0.0001,\ 0.001\}$	0.001

Table 9: Tunable learning parameters in transfer learning

Performance	
Metric	**Value**
f1-score	0.5115
accuracy	0.5309
precision	0.4941
recall	0.5302
roc-auc	0.5308

Table 10: Pre-trained GTN for the graph coloring problem after re-training

Although the classifier predicts slightly better that a dummy one (that predicts randomly), these results were surprising given the similarity between the two problems and the performance of the classifier on the SAT3 problem. There are many reasons as to why this may have occurred. First of all, the amount of data provided for re-training may not be sufficient. Also, the data augmentation we performed in order to create unsatisfiable instances may not be of good quality. Thirdly, in the SAT3 dataset, all clauses have no tautologies or repetition of the same literal, whereas in order to translate our graph coloring satisfiability problem into a 3SAT one we added tautologies into the clauses. Last but not least, each one of the datasets were created using different distributions of variables and clauses leading to different graphs with different topologies, making the problem quite different in practice that in theory.

References

[1] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. Moura, and D. Dill, "Learning a sat solver from single-bit supervision," 02 2018.

[2] M. O. R. Prates, P. H. C. Avelar, H. Lemos, L. Lamb, and M. Vardi, "Learning to solve np-complete problems - a graph neural network for decision tsp," 2018.

[3] B. Bünz and M. Lamm, "Graph neural networks and boolean satisfiability," 2017.

[4] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[5] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.

[6] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2019.

[7] R. Liao, Z. Zhao, R. Urtasun, and R. S. Zemel, "Lanczosnet: Multi-scale deep graph convolutional networks," 2019.

[8] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 5453–5462.

[9] X. Zhu, Z. Ghahramani, and J. D. Lafferty, "Semi-supervised learning using gaussian fields and harmonic functions," in *International Conference on Machine Learning*, 2003.

[10] X. Zhang and W. Lee, "Hyperparameter learning for graph based semi-supervised learning algorithms," in *Advances in Neural Information Processing Systems*, B. Schölkopf, J. Platt, and T. Hoffman, Eds., vol. 19. MIT Press, 2006.

[11] F. Wang and C. Zhang, "Label propagation through linear neighborhoods," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 1, pp. 55–67, 2008.

[12] M. Karasuyama and H. Mamitsuka, "Manifold-based similarity adaptation for label propagation," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013.

[13] C. Gong, D. Tao, W. Liu, L. Liu, and J. Yang, "Label propagation via teaching-to-learn and learning-to-teach," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, pp. 1–14, 04 2016.

[14] Y. Liu, J. Lee, M. Park, S. Kim, E. Yang, S. J. Hwang, and Y. Yang, "Learning to propagate labels: Transductive propagation network for few-shot learning," 2019.

[15] Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun, "Masked label prediction: Unified message passing model for semi-supervised classification," 2021.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.