Lab-2
Group-5

Dennis D'Amico - s343841, Pierre Perrier - s350300,
Manuele Sforzini - s349555

0 Goal of the laboratory

The goal of this project is to learn how to perform model engineering. Students will use different architectures and preprocessing techniques to model the same problem and data.

In particular, this laboratory asks students to handle categorical data, preprocess the data, experiment different architectures.

1 Task 1: Frequency-based baseline

The first approach to convert a categorical data into a numerical data is the frequency-based method.

Q:How many unique API calls does the training set contain? How many the test set? The training vocabulary contains 259 unique API calls and the test vocabulary contains 232 unique API calls. The first step is to extract the vocabulary.

Q:Are there any API calls that appear only in the test set (but not in the training set)? If yes, how many? And which one are they? 3 elements of the test vocabulary are not in the training vocabulary, namely (`NtDeleteKey`, `WSASocketA`, `ControlService`). In total (training + test), there are 262 unique API calls.

Now, the dataset are converted into frequency-based data.

Q:Can you use the test vocabulary to build the new test dataframe? If not, how do you handle API calls in the test set that do not exist in the training vocabulary? We cannot use the test vocabulary, so we take the training vocabulary and add a new label `UNK` that will be used for the API calls unknown to the training set (eg: `NtDeleteKey`, `WSASocketA`, `ControlService`). Thus the training vocabulary size will be 260.

Q:One issue of this frequency-based approach is that it creates sparse vectors (i.e., vectors with many zeros per row): how many non-zero elements per row do you have on average in the training set? How many in the test set? What is the ratio with respect to the number of elements per row?

- Average number of non-zero values in training data: 21.94
- Average number of non-zero values in test data: 24.27
- Average ratio of non-zero values per row in training data: 0.08
- Average ratio of non-zero values per row in test data: 0.09

Q:The original API sequences were ordered. Is it still the case now (i.e., in the frequency-based dataframe, do you still know which API came first)? Why? The frequency-based dataframe are unordered now, it just contains the number of time an API have been called and does not maintain the order of the calls. This is a simple representation that loses the temporal information of the original set.

Q:Report how you chose the hyperparameters of your classifier, and the final performance on the test set. We use a simple shallow NN classifier with a sigmoid activation function and a Cross-entropy loss because they are suitable for a binary classifier. Since the classes are imbalanced, we also used a weighted random sampler. The parameters are described in Table 1.

Q: Is the final performance good - even ignoring the order of API calls and handling very sparse vectors? The performances are surprisingly good, as shown in Figures 1 and 2. The

# Hyperparameter	value
# Layers	1
# Neurons per Layer	64
Activation	ReLu (in the Hidden layer)
Weight Initialization	WeightedRandomSampler
Batch Size	64
Loss Function	Cross-Entropy
Optimizer	AdamW
Learning Rate	0.0005
Epochs & Early Stopping	100 & early stopping
Regularization	None

Table 1: Neural Network Hyperparameters

malware recall is 0.96 so there are few false negatives, but one may consider 0.96 recall not enough, as it means that 4% of the malwares are classified as goodwares and are not detected! Also, the recall of benign is 0.77, meaning that 23% of the goodwares are wrongly classified as malwares and won't be used. Finally, the benign class only yields a 41% precision, so taking a random program in the benign class leads to taking a malware 6 times over 10.
So even by ignoring the order of the API calls, we can have a quiet good model which however has limitations.

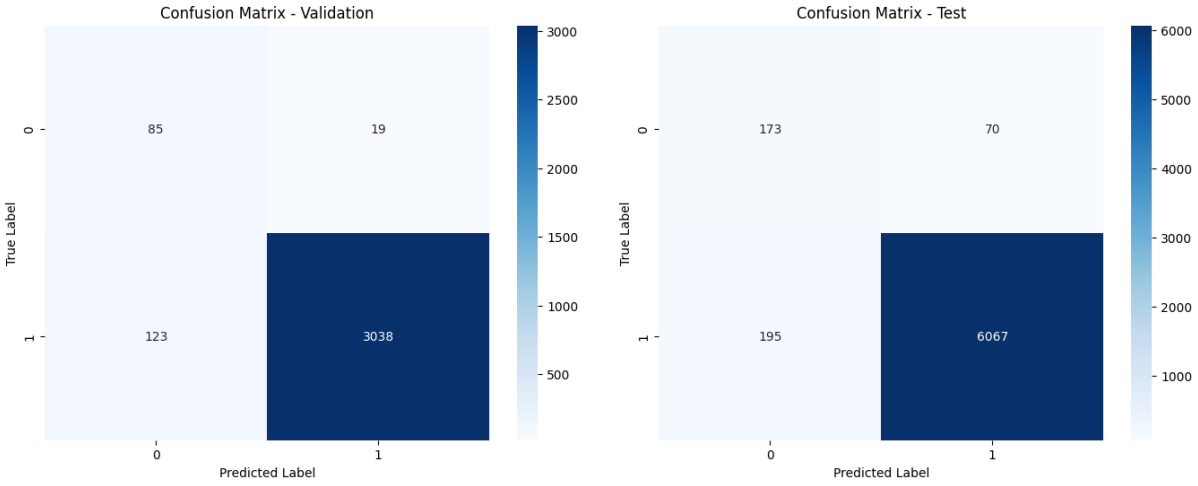


Figure 1: Confusion matrix for Validation and Test sets.

	precision	recall	f1-score
Benign	0.35	0.88	0.50
Malware	1.00	0.95	0.97
accuracy			0.94
macro avg	0.67	0.92	0.74
weighted avg	0.98	0.94	0.96

(a) Validation

	precision	recall	f1-score
Benign	0.41	0.77	**0.53**
Malware	0.99	**0.96**	**0.97**
accuracy			0.95
macro avg	0.70	0.86	0.75
weighted avg	0.97	0.95	0.96

(b) Test

Table 2: Classification reports for Validation and Test sets.

2 Task 2: Feed Forward Neural Network (FFNN)

Q: Do you have the same number of API calls for each sequence? If not, is the distribution of the number of API calls per sequence in the training set the same as the test one? No, we have sequences with different lengths in both the training and the test set. Also, the distribution of

the number of API calls per sequence is different among the sets: we have lengths between 60 and 90 calls in the training set and between 70 and 100 in the test set.

Statistic	Value
Min	60
Max	90
Mean	75.03
Std	8.95
Median	75.0

(a) Training set

Statistic	Value
Min	70
Max	100
Mean	86.33
Std	9.11
Median	87.0

(b) Test set

Table 3: Sequence length statistics for Training and Test sets.

Q: The first neural approach we learned involves FFNN. Can a FFNN handle a variable number of elements? If not, why? No, it can handle only inputs of a fixed size, in fact the input layer is composed of a fixed number of neurons: if we have an input that is smaller than the number of neurons, we will have some "holes", while with a bigger input we have no way to handle the extra dimensions.

Q: How to estimate a fixed-size candidate? What partition do you use to estimate it? We chose to use as a fixed-size candidate the sequence with the highest number of API calls in the training set. It is not possible to use the test set for these estimations, as it should represent the real world, and so it must only be used for testing.

Q: Given the estimate of the previous point, what technique could you use to obtain the same number of API calls per sequence? Using the estimation described above, we padded with 0s (0 is mapped to the `PAD` vocabulary word) the sequence of APIs, in order to obtain sequences of a fixed length.

Q: Suppose that at test time you have more API calls than the fixed-size you estimated. What do you do with the API calls exceeding such fixed-size? In this case, API calls exceeding the maximum length are truncated.

We used two approaches to map the features into a numerical space: sequential identifiers and learnable embeddings.

Q: Use a FFNN in both cases. Report how you selected the hyper-parameters of your final model, and justify your choices. For the sequential identifier FFNN (Table 4), the input size is the longest sequence in the training dataset (90). We decided to put 5 layers continuously decreasing in dimension to try to force the model to aggregate information and create a meaningful representation of what is a malware. We use ReLU as an activation function in the hidden layers. The loss function for this binary classification problem is the Cross-Entropy function. We use a weighted random sampler to compensate for the class imbalance.

# Hyperparameter	Value
Input Layer	
Input Size	90
Hidden Layers	
Number of neurons	[64, 64, 32, 32, 16]
Dropout	0.2
Activation	ReLU
Output Layer	
Output Dimension	2
Training	
Batch Size	64
Loss Function	Cross-Entropy
Optimizer	Adamax
Learning Rate	0.0005
Epochs	100 and early stopping
Class Balancing	Weighted Random Sampler

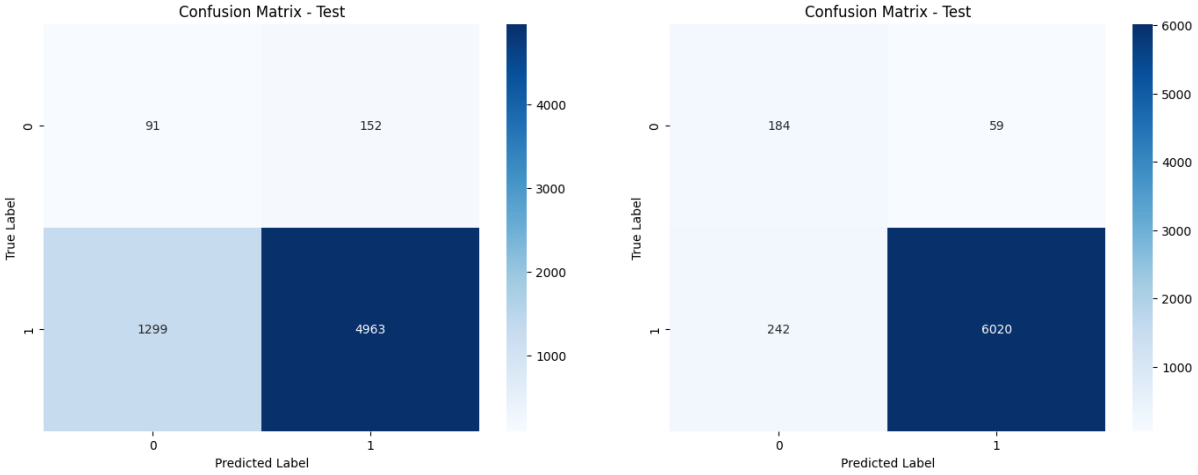
Table 4: Hyperparameters for the sequential identifier FFNN Model

For the embedded classifier (Table 5), we keep the same structure but choose an embedding of dimension 64, as tests showed that a higher dimension leads to a poor generalization (learning the data rather than the pattern) and a lower dimension simply loses too many information and the model doesn't learn anything.

Hyperparameter	**Value**
Embedding	
Vocabulary Size	260
Embedding Dimension	64
Hidden Layers	
Number of neurons	[64, 64, 32, 32, 16]
Dropout	0.3
Activation	ReLU
Output Layer	
Output Dimension	2
Training	
Batch Size	64
Loss Function	Cross-Entropy
Optimizer	AdamW
Learning Rate	0.0005
Weight Decay	0.0001
Epochs	100 and early stopping
Class Balancing	Weighted Random Sampler

Table 5: Hyperparameters for the Embedded-Classifier Model

Q: Can you obtain the same results for the two alternatives (sequential identifiers and learnable embeddings)? If not, why? No, as we can see in Figure 2a, we have very bad performance with sequential identifiers, in particular with normal samples, because with sequential ids the FFNN learns only the id, but there is no correlation between two different ids. The model is, in fact, forced to learn biased information based on the id (it may learn that api 5 is "closer" to api 6 than api 54, but they should be at the same distance). Instead, embeddings allow the model to achieve an internal meaningful representation, starting from a random mapping (so without bias), which helps it to perform better in the test set, as we can see in Figure 2b. In particular, the test report for the Embedded-Classifier FFNN, which will be used to do comparisons with other models, is reported in Table 7a.



(a) Sequential identifier FFNN

(b) Embedded-Classifier FFNN

Figure 2: Confusion matrices of the FFNNs

3 Task 3: Recursive Neural Network (RNN)

Q: With RNNs, do you still have to pad your data? If yes, how? We still have to pad the data, but not all the data must have the same size: the padding depends on the batch. We rank the data by length, and for each batch we pad all the data to the length of the longest sequence in that batch.

Q: Do you have to truncate the testing sequences? Justify your answer with your understanding of why it is/it is not the case. We don't have to truncate the sequence. RNNs can handle arbitrary length sequences, the only restriction is that each batch must have the same length, this is why we use padding.

Q: Is the RNN padding - if any - more memory efficient with respect to the FFNN's one? Why? The RNN padding is more memory efficient than the FFNN's one because it depends on the batch longest sequence, so less memory is wasted. Also during the training, we can *mask* the padding, this means that the NN does not train on padding and goes to the next input.

Q: Start with a simple one-directional RNN. Is your network fast as the FFNN? If not,where do you think that time-overhead comes from? Our RNN (cf. Table 6) is much slower than our FFNN. RNN is sequential and must compute and pass an internal state before going to the next sequence, so the parallelization is limited and the model is significantly slower. Also, back-propagation is more complex than the one from FFNN as the data go through multiple NN.

# Hyperparameter	Value
Embedding	
Vocabulary Size	260
Embedding Dimension	100
RNN	
Hidden Dimension	64
Number of Layers	2
Directionality	One-directional
Fully Connected Layer	
Output Dimension	1 (with Sigmoid)
Training	
Batch Size	64
Loss Function	Binary Cross-Entropy
Optimizer	Adam
Learning Rate	0.0001
Epochs	20

Table 6: Hyperparameters for the RNN Classifier Model

Q: Train and test three variations of networks We decided to train an RNN (cf. Table6), a BiRNN (cf. Table 8) and an LSTM (cf. Table 9). We provide the loss functions (Figure 3) and the test reports for each RNN (Figure 4 and Tables 7b, 7c, 7d).

	precision	recall	f1-score
Benign	0.43	0.76	0.55
Malware	0.99	0.96	**0.98**
accuracy			0.95
macro avg	0.71	0.86	0.76
weighted avg	0.97	0.95	0.96

(a) Embedded-Classifier FFNN

	precision	recall	f1-score
Benign	0.63	0.38	**0.48**
Malware	0.98	**0.99**	**0.98**
accuracy			0.97
macro avg	0.80	0.69	0.73
weighted avg	0.96	0.97	0.96

(b) RNN

	precision	recall	f1-score
Benign	0.46	0.51	**0.48**
Malware	0.98	**0.98**	**0.98**
accuracy			0.96
macro avg	0.72	0.75	0.73
weighted avg	0.96	0.96	0.96

(c) BiRNN

	precision	recall	f1-score
Benign	0.56	0.57	**0.57**
Malware	0.98	**0.98**	**0.98**
accuracy			0.97
macro avg	0.77	0.78	0.78
weighted avg	0.97	0.97	0.97

(d) LSTM

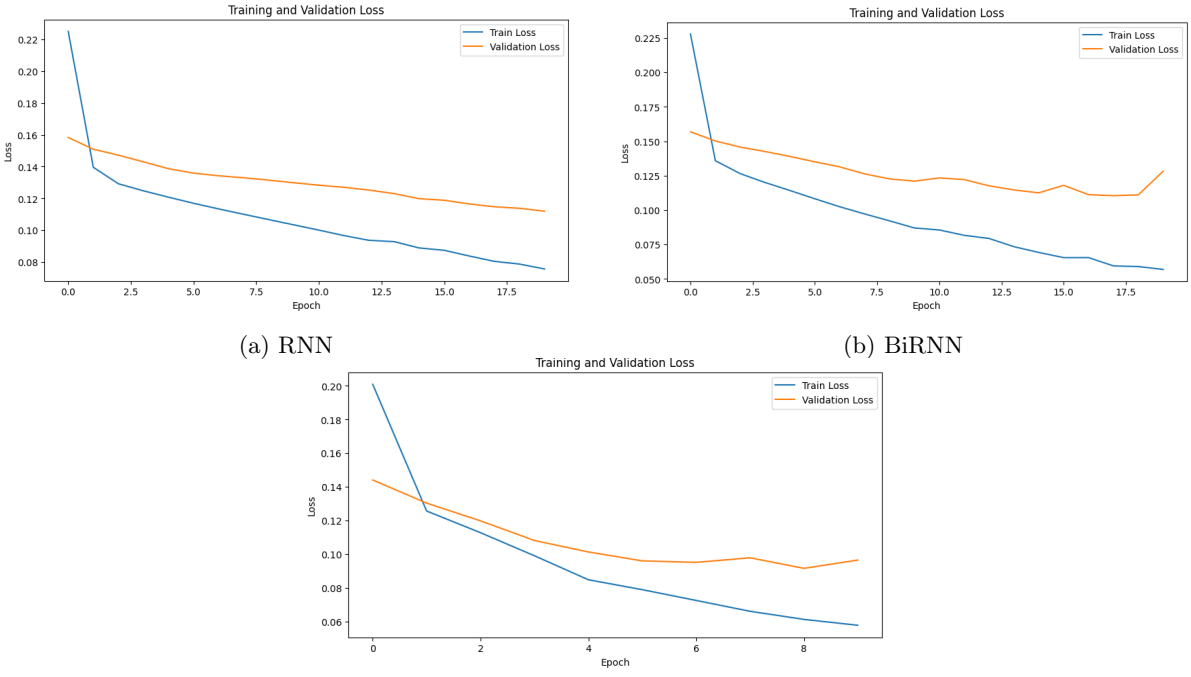
Table 7: Tests results for different RNNs

Table 8: Hyperparameters for the BiRNN Classifier Model

# Hyperparameter	Value
Embedding	
Vocabulary Size	260
Embedding Dimension	100
BiRNN	
Hidden Dimension	64
Number of Layers	2
Directionality	Bidirectional
Fully Connected Layer	
Output Dimension	1 (with Logits)
Training	
Batch Size	64
Loss Function	Binary Cross-Entropy (with Logits)
Optimizer	Adam
Learning Rate	0.0001
Epochs	20

# Hyperparameter	Value
Embedding	
Vocabulary Size	260
Embedding Dimension	128
LSTM	
Hidden Dimension	128
Number of Layers	2
Directionality	Bidirectional
Dropout	0.3 (between layers)
Fully Connected Layer	
Output Dimension	1 (with Logits)
Training	
Batch Size	64
Loss Function	Binary Cross-Entropy (with Logits)
Optimizer	Adam
Learning Rate	0.0001
Epochs	10

Table 9: Hyperparameters for the LSTM Classifier Model

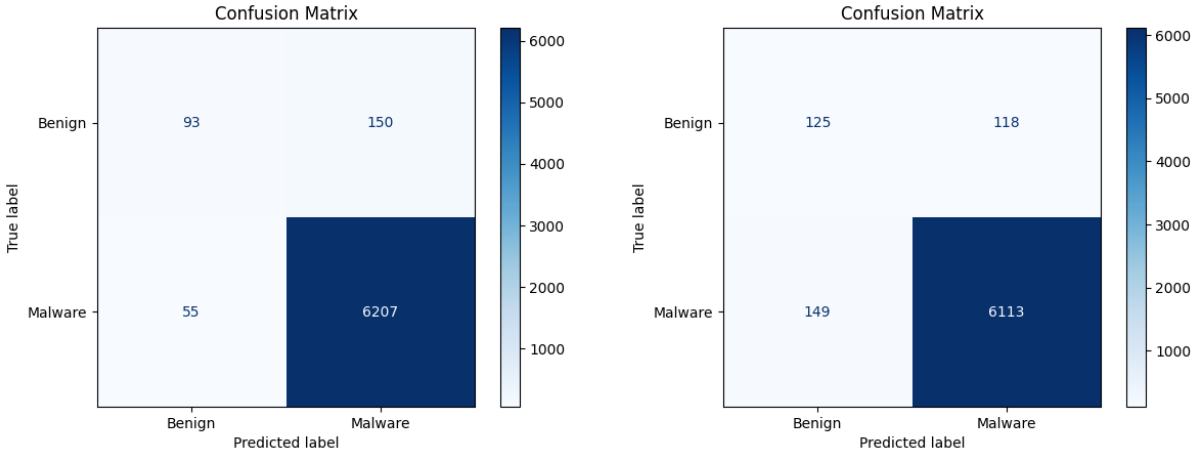


(a) RNN

(b) BiRNN

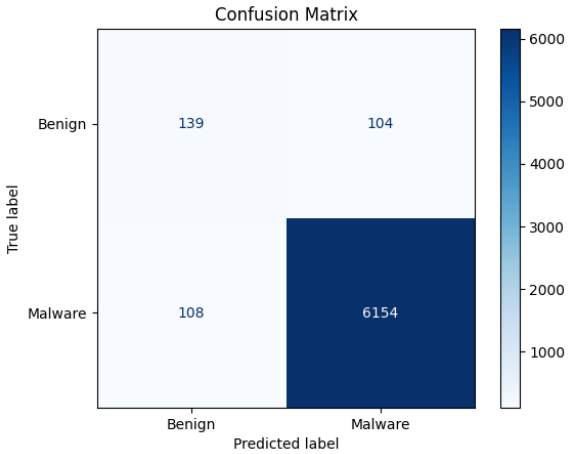
(c) LSTM

Figure 3: RNNs loss functions



(a) RNN

(b) BiRNN



(c) LSTM

Figure 4: RNNs test confusion matrices

Q: Is the RNNs training as stable as the FFNN's one? As we can see in Figure 3, the RNNs training is less stable than the training of FFNN, we must use a smaller learning rate and a lower number of epochs.

Q: How does your model's performance compare to the simple frequency baseline, given that you now account for the sequence of API calls and use a significantly more complex network? The LSTM model outperforms the previous FFNN model in general without being extremely more efficient. We have a slightly better f1-score for both classes, a higher recall for malware and a better precision for benign so fewer false negative, at the cost of having a bit more false positive, indeed a slightly lower malware precision and lower benign recall means that more benign are classified as malware. The other two RNNs have worse performance than the LSTM, with more precision in the benign class with respect to the FFNN but worse recall and f1-score and with no improvements in the malware class.

4 Task 4: Graph Neural Network (GNN)

Q: Do you still have to pad your data? If yes, how? No, GNNs don't require padding, as their algorithms don't make assumptions on the number of vertices and on the number of edges in the graph.

Q: Do you have to truncate the testing sequences? Justify your answer with your understanding of why it is/it is not the case. No, for the same reason as the one of the previous question: in GNN, we don't have to make assumptions on the length of each sequence, which will be correctly handled independently of its length.

Q: What is the advantage of modeling your problem with a GNN with respect to an RNN in this scenario? However, what do you lose? Graphs captures structural information really well, they can better understand the relationship between two API calls. Also, they don't need any sort of padding or truncation, which could lose information. On the other hand, we lose the order of the API calls in the sequence, there is no order/direction in the sequence.

Q: Finally train and tune variations of GNN considering different message and aggregation functions and architectures: Simple GCN, GraphSAGE and GAT. We trained a GCN model (Table 10), a GraphSAGE model (Table 11) and a GAT model (Table 12). We report the loss functions (Figure 5) and the test reports (Table 13), along with the related confusion matrices (Figure 6).

# Hyperparameter	Value
Embedding	
Vocabulary Size	260
Embedding Dimension	64
Graph Convolutional Layers	
Hidden Dimension	64
Number of Layers	3
Dropout	0.3
Fully Connected Layer	
Output Dimension	2
Training	
Batch Size	64
Loss Function	Cross-Entropy
Optimizer	Adam with weight decay = 0.0001
Learning Rate	0.0005
Epochs	50

Table 10: Hyperparameters for a GCN Model

# Hyperparameter	Value
Embedding	
Vocabulary Size	260
Embedding Dimension	32
GraphSAGE Convolutional Layers	
Hidden Dimension	32
Number of Layers	3
Dropout (Intermediate)	0.3
Dropout (Final)	0.4
Fully Connected Layer	
Output Dimension	2
Training	
Batch Size	64
Loss Function	Cross-Entropy
Optimizer	Adam with weight decay = 0.0001
Learning Rate	0.0005
Epochs	50

Table 11: Hyperparameters for a GraphSAGE Model

# Hyperparameter	Value
Embedding	
Vocabulary Size	260
Embedding Dimension	64
Graph Attention Layers	
Hidden Dimension	32
Number of Attention Heads (First Layer)	8
Number of Attention Heads (Second Layer)	1
Concatenation (First Layer)	True
Concatenation (Second Layer)	False
Dropout	0.3
Fully Connected Layer	
Output Dimension	2
Training	
Batch Size	64)
Loss Function	Cross-Entropy
Optimizer	Adam with weight decay = 0.0001
Learning Rate	0.0001
Epochs	50

Table 12: Hyperparameters for the GAT Model

(a) GCN

	precision	recall	f1-score
Benign	0.38	0.88	0.53
Malware	1.00	0.94	0.97
accuracy			0.94
macro avg	0.69	0.91	0.75
weighted avg	0.97	0.94	0.95

(b) GraphSage

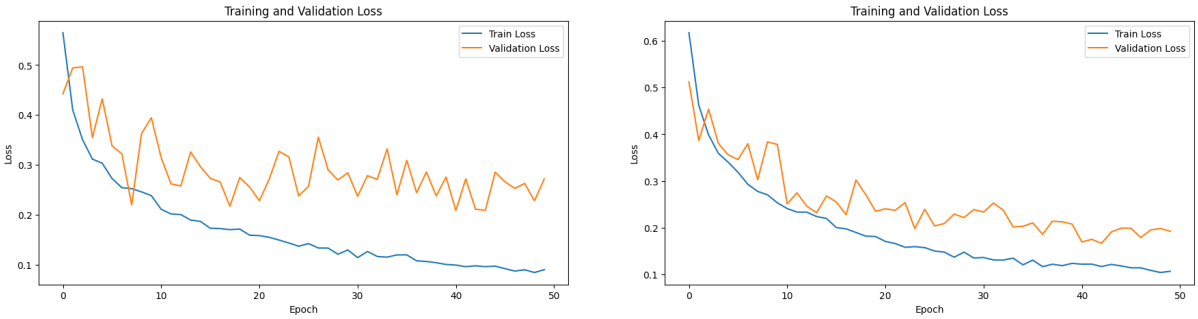
	precision	recall	f1-score
Benign	0.45	0.87	**0.59**
Malware	0.99	**0.96**	**0.98**
accuracy			0.95
macro avg	0.72	0.92	0.78
weighted avg	0.97	0.95	0.96

(c) GAT

	precision	recall	f1-score
Benign	0.42	0.83	0.56
Malware	0.99	0.96	0.97
accuracy			0.95
macro avg	0.71	0.89	0.77
weighted avg	0.97	0.95	0.96

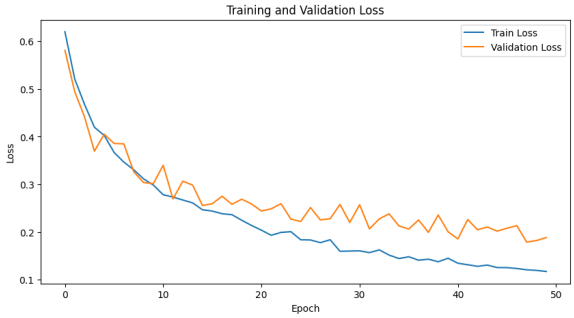
Table 13: Tests results for different GNNs

Q: How does each model perform with respect to the previous architectures? Can you beat the baseline? Each model has a really good f1-score for malware, but a slightly lower recall and higher precision, which means there is a bit more false negative. But there is also much less false positive, indeed recall for benign is greater than 0,8. So if we accept slightly more false negative (non detected malware), we can have much less false positive (benign classified as malware). The overall performances of the model are not perfect but seem good enough.
We can say that the GNN models achieve better results than the baseline, improving each metric in the Benign class and keeping the same very good performance in the Malware class.



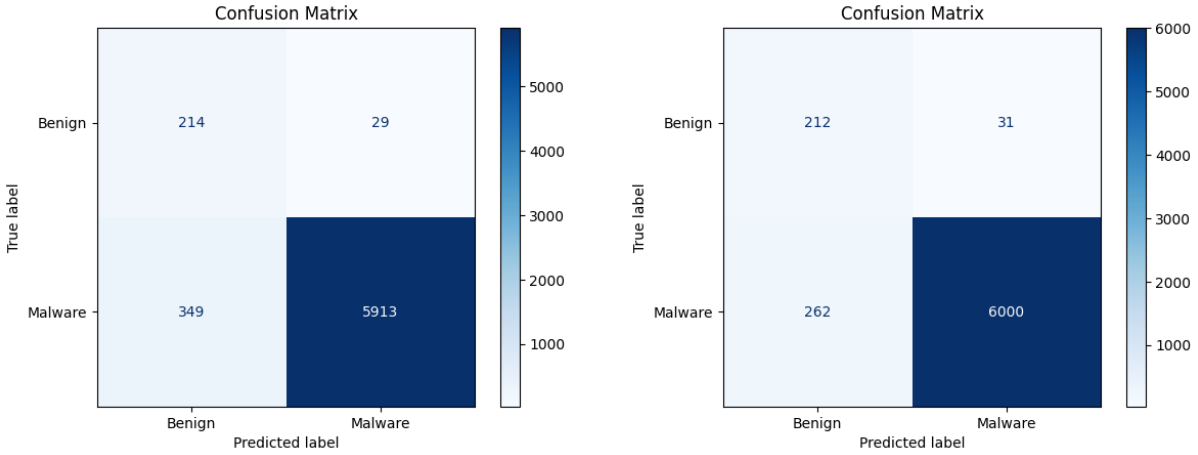
(a) GCN

(b) GraphSage



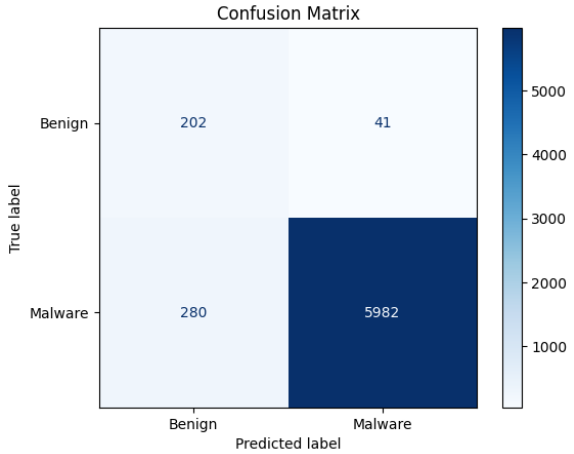
(c) GAT

Figure 5: GNNs loss functions



(a) GCN

(b) GraphSage



(c) GAT

Figure 6: GNNs test confusion matrices