Model Engineering

Alessandro Torrisi - s330732, Simone Sambataro - s331812,
Floralucy Cusano - s334142

The aim of this project was to experiment different Neural Network architectures (FFNN, RNN, and GNN) on the same dataset, applying appropriate pre processing techniques for each model and evaluating their performance and behavior.

1 Task 1: BoW Approaches

Before proceeding to network design, we performed a thorough inspection and cleaning of the dataset. First of all, we observed an imbalance in both training and test sets, with a larger proportion of samples labeled as *malware*. Moreover, each sample was originally identified by an MD5 hash computed over its API-call sequence. However, because the sequences were later randomly truncated, the hash no longer guaranteed uniqueness. Consequently, we dropped the `md5hash` column and eliminated all remaining duplicate rows. Fig. 1 shows the distribution of API-call sequence lengths in the training set. Sequences range roughly from 60 to 90 calls, with a slight skew toward longer traces, suggesting that malicious and benign behaviors may differ in typical call-sequence length.

API Call Frequency Comparison We also compare the top twenty most frequent API calls in train versus test. While the overall ranking remains consistent, the test set exhibits lower absolute counts, and certain calls (e.g. `RegOpenKeyExW` or `NtCreateFile`) show proportionally larger discrepancies.

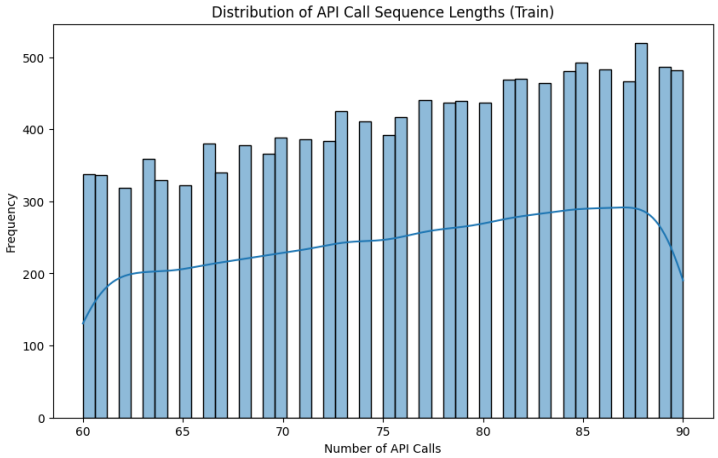


Figure 1: Distribution of API-call sequence lengths (Train).

Additional checks (e.g. for missing values) and summary statistics were also carried out and are documented in the accompanying notebook.
To transform these process-sequences (each treated as a "sentence" of API-call tokens) into a numerical feature space, we employed the CountVectorizer class. After fitting it on the training data to learn the vocabulary of API calls, we obtained a sparse matrix of token counts which was then used as input to our classifiers.

- **How many columns do you have after using the CountVectorizer? What does this number mean?** The number of columns in the resulting matrix is equal to the size of the vocabulary. In other words, each column represents one unique API call (token) that was seen in the training set. Specifically, our training data contains 253 unique API calls, so CountVectorizer

will generate 253 columns. Each column counts how many times a particular API call appears in a given sequence.

- **What does each row represent? Can you still track the order of the processes (how they were called)?** Each row in the resulting document-term matrix represents a single sample (i.e., a sequence of API calls). However, the Bag-of-Words (BoW) representation inherently discards any information about the order of the tokens. It captures only the frequency of each token within a sequence, not the sequence itself. Therefore, any temporal or structural information is lost in this representation.

- **Do you have any out of vocabulary from the test set? If yes, how many? How does CountVectorizer deal with them?** Looking more closely at the string containing the API calls, it was interesting to notice that some API calls appeared exclusively in either the training or the test set. Specifically, we found that 30 API calls were present only in the training set, and 5 only in the test set. However, the majority (223 out of 253) were present in both sets. Since the CountVectorizer builds its vocabulary from the training data, when applied to the test set, any API calls not present in the original vocabulary are treated as out-of-vocabulary (OOV) tokens. These tokens are ignored during transformation, meaning they are not included in the feature vector and so they are simply dropped. In our specific case the number of OOV in the test set is 5.

1.1 Try to fit a classifier. Report how you chose the hyperparameters of your classifier and what the final performance was on the test set.

To evaluate how useful this BoW representation is for classification purposes, we fit different classifiers. We started with a simple Logistic Regression model, but this approach led to subpar performance on the underrepresented benign class (extremely low Recall), yielding a Macro-F_1 score of only 0.70. Then, we experimented:

- Logistic Regression training pipeline with 3-fold GridSearchCV to select the optimal regularization parameter C on the training set, then evaluated the best model on the held-out test set. achieved: Accuracy = 0.9712, Macro-$F_1 = 0.71$. However, performance on the minority (benign) class remains weak (indicating how misleading is the accuracy metric in case of unbalanced classes): Class 0 (Benign): Precision = 0.65, Recall = 0.34, $F_1 = 0.44$; for Class 1 (Malware): Precision = 0.98, Recall = 0.99, $F_1 = 0.99$

  We also perform an analysis of which API calls are most strongly correlated with malware or benign behavior, specifically for this specific case we obtain that the top positive coefficients (e.g. `GetAdaptersInfo`, `CryptCreateHash`) strongly signal malware, while the most negative coefficients (e.g. `WriteConsoleW`, `RegDeleteValueW`) indicate benign behavior.

- Logistic Regression training pipeline with 3-fold GridSearchCV with class_weight='balanced', but still the performances do not improve.

Additional Classifier Experiments

- **Decision Tree with 5-fold GridSearchCV.** We tuned `criterion`, `max_depth`, and `min_samples_split` using stratified folds.
  - **Best hyperparameters:** {criterion = "entropy", max_depth = $\perp$, min_samples_split = 2}
  - **Test performance:**
    - Accuracy = 0.9713
    - Class 0 (Benign): Precision = 0.59, Recall = 0.53, F_1=0.56
    - Class 1 (Malware): Precision = 0.98, Recall = 0.99, F_1=0.99
    - Macro-F_1 = 0.77

  The decision tree captures malware patterns very well, slightly improve to respect to the previous configurations, but still underperforms on the minority benign class.

- **TF–IDF Representation with Logistic Regression.** We applied a `TfidfVectorizer` (uni– to tri-grams, `min_df` = 5) and conducted a 5-fold GridSearchCV.

- Vocabulary size: 7 851 terms
- Overall performance: Accuracy = 0.9806, Macro-F_1 = 0.84
- Per-class performance:
  Benign: P/R/F_1 = 0.93/0.54/0.68;
  Malware: P/R/F_1 = 0.98/1.00/0.99

We can observe an important improvment to respect to the simple uso of CountVectorizer, this because TF–IDF enhances the raw count representation in several ways. First, by down-weighting extremely common API calls and up-weighting more informative, rarer n-grams, it focuses the model on discriminative features. Second, extending the vocabulary to include bi- and tri-grams captures short sequences of API calls—subtle patterns characteristic of malware—that a bag-of-unigrams approach cannot represent. Finally, applying a min_df = 5 threshold filters out very rare tokens that often introduce noise, ensuring that only robust, frequently observed patterns contribute to the model. Together, these improvements create a far more discriminative feature space, boosting macro-F_1 to 0.84 (from 0.78) and significantly improving the detection of the minority benign class (but still not perfect, as can be seen by the low value of Recall).

Note: that for the subsequent tasks, the original training set was further divided into 80% for training and 20% for validation.

2 Task 2: Feed Forward Neural Network

Each sample in our dataset contains a sequence of API calls. Transforming them into numerical representations is a fundamental step in preparing this type of data for machine learning models such as FFNN.

- **Do you have the same number of calls for each sample? Is the training distribution the same as the test distribution?** Our first step was to analyze the number of API calls per sample. Upon inspection, it became clear that we do not have the same number of API calls for each sample. The sequences vary in length. Specifically, the maximum sequence length in the training set is 90, while in the test set it goes up to 100, and as noted earlier, sequence lengths typically range from 60 to 90 API calls.
- **Suppose to use a simple Feed Forward Neural network to solve the problem. Can a Feedforward Neural Network handle a variable number of elements? And why?** FFNNs, by design, cannot handle variable length inputs because they require fixed size input vectors. Unlike recurrent models, FFNNs have a static architecture with fixed size weight matrices, that cannot dynamically adjust to input of varying lengths. Consequently, all input sequences must be transformed into fixed size representations before being fed into the network.
- **What technique do you use to get everything to a fixed size during training? What happens if you have more processes to process at test time?** Because we indexed API calls starting at 1 (reserving 0 for padding), we standardized every sequence to length 100 by rigth-padding shorter sequences with zeros and truncating any calls beyond the 100th. This way, at inference time any extra API calls are simply dropped and every input still matches the network's fixed input shape.

2.1 FFNN with sequential identifiers and learnable embeddings

Report how you chose the hyperparameters of your final model and justify your choice. Can you achieve the same results for the two alternatives? Explain (report on the results, whether one training was longer/more unstable than the other, etc.)

2.1.1 Sequence Identifiers: Hyperparameter Tuning and Final Model

We first trained a minimal FFNN with 2 hidden layers (256 64) using ReLU activations, the AdamW optimizer, and a OneCycleLR scheduler. Early stopping on validation loss halted training after 27 epochs. Although this model achieved 97% accuracy, it completely failed to capture the minority class,

with benign recall of only 17% and a macro-F_1 of 0.56—clearly demonstrating that accuracy is misleading on imbalanced data.

To improve the representation capacity, we tried different architectures, with different combinations of Batch Normalization, Dropout layers, weight decay, and finally we moved to a deeper FFNN (256 128 64 32 16) and added BatchNorm plus 20% dropout after each layer. The resulting learning curves demonstrated smoother convergence and the gap between training and validation loss narrowed, but overfitting persisted, indicating in the case of sequential identifiers a general unstable training process. The deeper network raised benign recall to 33% and boosted macro-F_1 to 0.71, yet its performance on the underrepresented class remains inadequate. These findings indicate that raw sequence identifiers alone do not capture the subtle patterns needed for reliable malware detection.

2.1.2 Learnable Embeddings and Comparison

As a final approach, we projected each API-call index into a 64-dimensional learnable embedding, mean pooled the sequence, and fed it into a deep FFNN (128→64→32→1) with BatchNorm and a high 50% Dropout rate to aggressively regularize. Training used AdamW with weight decay 10^{-2}, a OneCycleLR schedule, and early stopping (patience=15). We also swept the decision threshold on the validation fold to maximize macro-F_1, selecting $\tau \approx 0.75$ alongside with incorporating class-weighting in the loss function, to counteract the class imbalance and better align the model's behavior with real world operational requirements.

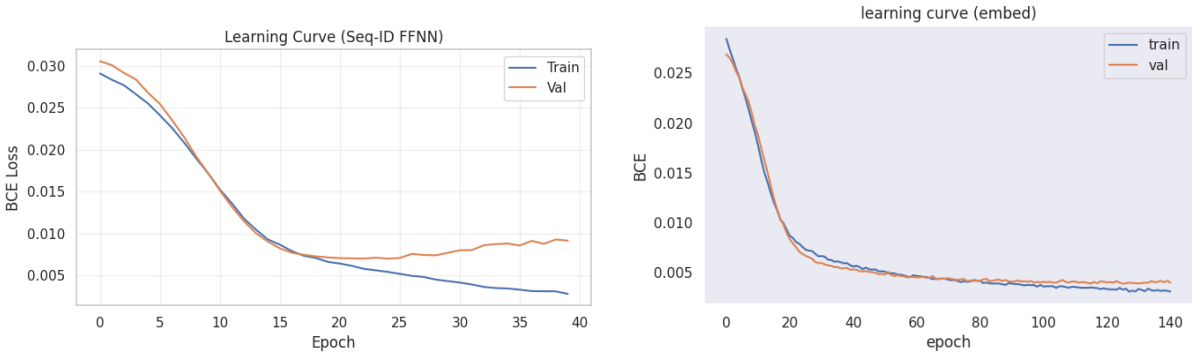
- **Training dynamics:**

  The BCE loss dropped precipitously in the first 20 epochs and then stabilized; training and validation curves remained close (Fig. 2b), indicating minimal overfitting despite the large model capacity. Total training time was about 1.51 minutes, and convergence required 141 epochs—longer than the 35 epochs of the Seq-ID model—but yielded far more stable generalization and smoother loss curves.

- **Test performance:**

  $$\begin{aligned} &\text{Benign: } P/R/F_1 = 0.57/0.66/0.61, \\ &\text{Malware: } P/R/F_1 = 0.99/0.98/0.99, \\ &\text{Macro-}F_1 = 0.80, \quad \text{Accuracy} = 0.97. \end{aligned}$$

  The confusion matrix obtained confirms a substantial increase in benign recall (66%) versus the Seq-ID FFNN (33%), reducing false negatives and correctly recovering more than half of the benign samples.



(a) BCE loss over epochs - SeqID.

(b) BCE loss over epochs - Learnable Embeddings.

Figure 2: Training process comparison

3 Task 3: Recurrent Neural Network

In this third part, we implemented and trained various types of architecture of Recursive Neural Networks (RNNs) to model the same classification problem used previously with FFNNs.

3.1 Preprocessing: padding and truncation

Do you still need to pad your data? If yes, how? Because RNNs require that every sequence in the same mini-batch have identical lengths (even though different mini-batches may vary), we zero-pad shorter inputs. In our straightforward implementation, each API-call sequence is encoded as a vector of positive integer IDs and then padded with zeros to the maximum sequence length observed in the test set—while still retaining each sequence's true length. A more flexible approach would be to define a custom `TensorDataset` and supply a `collate_fn` that, for each batch, dynamically finds the longest sequence and pads all others (e.g. using `pad_sequence`).
Do you need to truncate the test sequences? Justify your answer with an understanding of why this is or is not the case. In our implementation, we set the global maximum length to the longest sequence in the test set, so no test example ever exceeds it, hence truncation is unnecessary. We simply zero-pad every sequence up to that fixed length and rely on `pack_padded_sequence` to ensure the RNN processes only the real tokens.
More generally, if the RNN is allowed to handle truly variable lengths via packing, we never need to truncate at test time—just pad each sequence to its own length. The only scenario that mandates truncation is when you deliberately cap every training sequence at some fixed L, in that case, we should have to truncate any test sequences longer than L to maintain a consistent input distribution.

3.2 Memory advantage using RNN

Is there any memory advantage to using an RNN over an FFNN when processing your dataset? And why? It is broadly correct to say that RNNs offer a memory advantage over FFNNs at inference time. RNNs process a sequence step by step, carrying forward a fixed-size hidden state that captures temporal dependencies without storing all past inputs explicitly. In contrast, a FFNN must flatten the entire sequence into one large input vector, whose dimension grows linearly with the number of time steps—leading to much higher memory usage for long sequences.
However, during training, RNNs still incur an $O(T)$ memory cost for back-propagation through time (BPTT), since frameworks must keep the activations of all T steps to compute gradients. Only at inference can an RNN operate with constant memory complexity in the sequence length.

3.3 Training Speed and Overhead

In this step we started with a simple one-directional RNN as a baseline.
Is your network as fast as the FFNN? If not, where do you think the time-overhead comes from? Compared to the FFNN, the RNN was not as fast in training. This is expected: RNNs process sequences step-by-step, updating the hidden state at each time step, which introduces sequential operations that are not easily parallelizable. In contrast, FFNNs perform computations in parallel across all inputs, making them faster.
The overhead in RNN training time mainly comes from this recurrent nature and the backpropagation through time algorithm, which is more computationally intensive than standard backpropagation. The extra time is also due to the additional cost of handling variable length sequences (e.g., using pack_padded_sequence). In addition, bidirectional and LSTM architectures have extra computations (e.g., for backward passes and gating mechanisms).

3.4 Training and Performance - Variants

We trained and evaluated several RNN-based architectures to understand how different configuration performed.
Can you see differences during their training? Can you see the same performance as the FFNN? Report the training details, your choices of hyperparameters, the test results.
The three main families of models explored were:

- **Simple one-directional RNN**: several tests have been conducted. We tested models with 1 and 2 layers, with different combination of Dropout (e.g., dropout(0.2) added or removed), pos_weight (to address class imbalance). They are all listed in the notebook. An embedding layer (embedding dim=50). As a trial, also an extra nonlinear layer was introduced before the final output layer, but the single-layer RNN (hidden size=64) was ultimately chosen.

The best performance for this configuration was achieved without dropout and without weights. Although we experimented with the same architecture augmented by early stopping and positive-class weighting which delivered excellent metrics on the minority class, it nevertheless produced an unacceptably high number of false negatives. Consequently, we adopted the plain configuration: despite its lower recall on the minority class (fewer true positives and more false positives), it achieves a very low false negative rate, a critical requirement since overlooking a malware instance is far more costly. However, evaluating metrics against the test set, the model performs well for the Malware class (precision 0.98, recall 0.98) but misclassifies the minority Benign class (precision 0.76, recall 0.48) with a marco-F_1 of 0.79.

We also evaluated a deeper variant by introducing a second hidden layer, with different combination of dropout and positive-class weighting, but this modification actually degraded performance. Likewise, inserting an extra non-linear activation immediately before the output layer failed to yield any improvement. A plausible explanation is that increasing the network's depth and non-linearity introduced excess complexity for our relatively small and imbalanced dataset, leading to overfitting and less stable gradient propagation, which in turn harmed generalization.

The loss curves of the chosen model (Fig. 3) shows sudden fluctuations, with frequent spikes, indicating instability. Moreover, there is a clear gap between training and validation. However, the overall trend remains decreasing and does not diverge. A possible explanation of these abrupt spikes in both the training and validation loss curves are indicative of unstable weight updates in a vanilla RNN. In a single-layer RNN, gradients must be back-propagated across many time steps, and when sequences (or batches) are especially long these gradients can explode. Over the following epochs the network readjusts and the loss returns to its prior trajectory, hence the brief "spikes" we can observe.

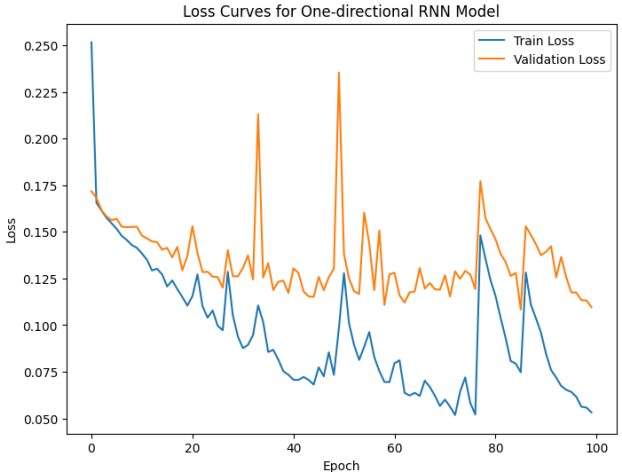


Figure 3: Loss curve for the simple one-directional 1 layer RNN

- **Bi-Directional RNN**: moving from a unidirectional RNN to a bidirectional one, shows an initial improvement in performance. The model was created with the same hyperparameters of the previous one, but sequences are now processed in both direction, doubling the effective hidden state size. As result on the test set, the majority Malware class continues to be well classified, and in this case, the precision for the Benign class increases to 0.87. However, recall remains low: there are many false positives, but an incredibly low number of false negative (only 17).

  The loss curve (Fig. 4) follows a smoother behavior than the one-directional RNN indicating less instability. The performance improved because a bidirectional RNN processes each sequence both forwards and backwards. In doing so it has access to "future" as well as "past" context around every timestep enabling the model to base its prediction not only on what came before a given call but also on what follows it.

- **Long Short Term Memory (LSTM)**: converged more slowly, requiring on the order of 60–90 seconds versus roughly 45–50 s for the simple RNN, but nevertheless achieved markedly better performance. Its internal gating mechanisms enable it to retain and propagate long range dependencies, yielding both greater training stability and higher final accuracy. Moreover, we opted for

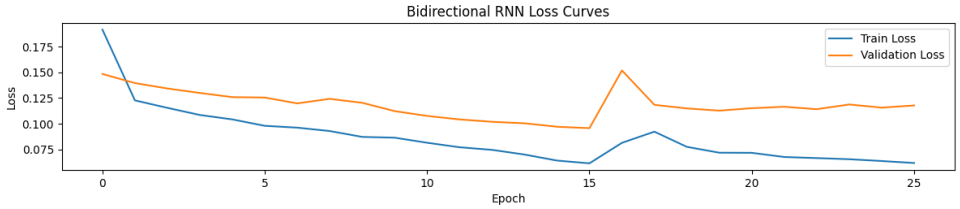


Figure 4: Bi-RNN Loss Curve

a *bidirectional* LSTM in our setting, since we have access to entire API call sequences and can therefore exploit both forward and backward context; a unidirectional LSTM, by contrast, is more appropriate for streaming or real time prediction scenarios.

– **Final Bi-LSTM Model with Optuna Tuning**
We selected as our best model a two-layer bidirectional LSTM whose hyperparameters were optimized via Optuna over 20 trials (embedding_dim=50, hidden_size=64). The training pipeline was as follows:
  * **Benign (minority) class**: Precision = 0.82, Recall = 0.72, F_1 = 0.77
  * **Malware (majority) class**: Precision = 0.99, Recall = 0.99, F_1 = 0.99
  * **Overall**: Accuracy = 0.99, Macro-F_1 = 0.88

  The confusion matrix (Fig. 5) shows only 38 false negatives, our primary concern, while accepting 71 false positives. Although the other architectures implemented achieved higher benign recall, they did so at the cost of many more false negatives. In contrast, this Bi-LSTM strikes the best balance by minimizing malware mislabeled as benign) even if it incurs more benign samples as malware, since as we said above in security applications false negatives are far more costly.

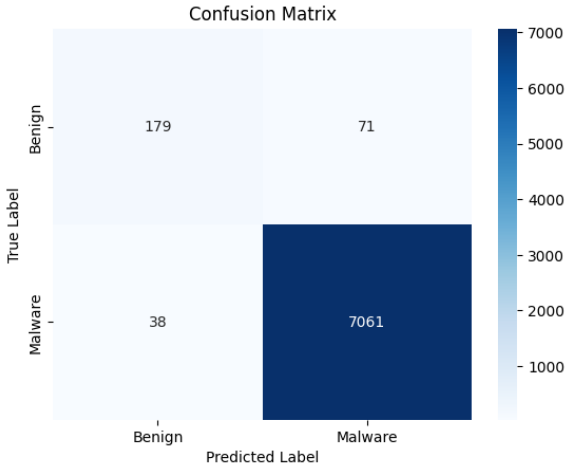


Figure 5: Test-set confusion matrix final Bi-LSTM.

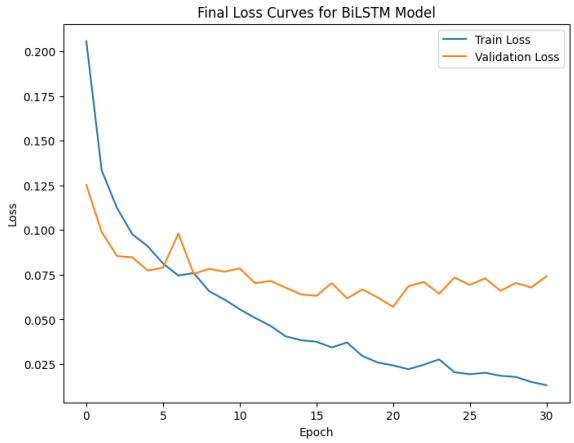


Figure 6: BCE loss curves final Bi-LSTM.

Overall, recurrent architectures—particularly the bidirectional LSTM, outperform the FFNN on sequential API call data by explicitly modeling temporal dependencies in both directions. This gain in accuracy, however, requires substantially more training time and memory. In the next phase, we turn to Graph Neural Networks (GNNs) to capture richer, non-linear structural relationships that lie beyond simple sequence order.

4 Task 4: Graph Neural Network

In this step, we represented each sequence of API calls as a graph, where each unique API call is a node and the edges represent how calls follow or relate to each other. We tried 2 different

approach, we used tried 2 diffrent ways to create the feature matrix

- We used direclty the APIs IDs
- We create a one hot encoding of all the possible values (and used UNK for value that were not present in the training dataset)

In the notebook we have different tests to see which was the best model, testing the best architecture (best hidden size, use of dropout layer and weighted class to account for the unbalanced dataset).

In the end we tested that the best thing to do is to pass the one hot encoding of the APIs, as clearly evident in the notebook.

4.1 Padding and truncation

Do you still need to pad your data? If so, how? Since GNNs naturally handle variable-size input graphs, there is no need to pad the data to a fixed length, unlike RNNs or FFNNs.

Do you need to truncate the testing data? Justify your answer with your understanding of why this is or is not the case. Truncating the testing data is not necessary either, as GNNs are capable of processing graphs of different sizes. Truncating would discard structural information and harm generalization. Keeping full sequences ensures the model captures the full context of the behavior pattern encoded in the graph.

4.2 Advantages and Disadvantages using GNN

What is the advantage of modelling your problem with a GNN compared to FFNN and RNN? Are there any disadvantages? The main advantage of using GNNs compared to FFNNs and RNNs is their ability to capture topological relationships and non-sequential dependencies within the data. For instance, it's possible to capture relationships between API calls that are not adjacent. While FFNNs ignore order and RNNs are limited to temporal structure, through GNN we have a better representation of a program behavior. Since programs inherently have a graph-like structure, now we are able to analyze them in a way that better reflects their nature. As disadvantages, we mainly lose on simplicity and interpretability of the model: we need to create a graph representation of our data and that could be computationally more expensive and may require memory managment.

4.3 Training speed on CPU and GPU

We started with a simple Graph Convolutional Network (GCN) trained on both CPU and GPU.

How long does it take to train and test in each configuration? How is it different from previous architectures? Can you guess why? On the GPU it took around 189 seconds, while on the CPU the time was 1052 seconds (almost 10 times the GPU time). Graph models like GCNs involve sparse matrix multiplications, which are computationally more intensive and well-optimized on GPUs using CUDA libraries. Compared to FFNNs and RNNs, GCNs are generally slower to train due to neighborhood aggregation and sparse graph operations. FFNNs are the fastest thanks to full parallelization. RNNs are slower than FFNNs because of their sequential nature, but still faster than GCNs when using short sequences. The time overhead in GCNs increases with graph complexity and depth.

4.4 Training and Performance - Variants

As the final step we extended our evaluation to three different GNN architectures:

- Simple Graph Convolutional Network
- GraphSAGE
- Graph Attention Network GAT

Can you see any differences in your training?

The goal was to compare their performance and training time, and to understand how different message-passing and aggregation strategies impact the model's ability to classify sequences of API calls represented as graphs.

The first difference observed between the 3 architectures was the **training time**. Training was relatively fast for GCN and GraphSAGE (approximately 234s and 204s on GPU respectively), but higher for GAT, which required around 298s. This was expected: GAT introduces an attention mechanism that performs additional computations per edge to weigh neighbor contributions dynamically.

All three **learning curves** (Figure 7) show a slight overfitting. This behavior might be due to the limited size of the dataset, so the models have limited data to learn from and haven't fully generalized yet.

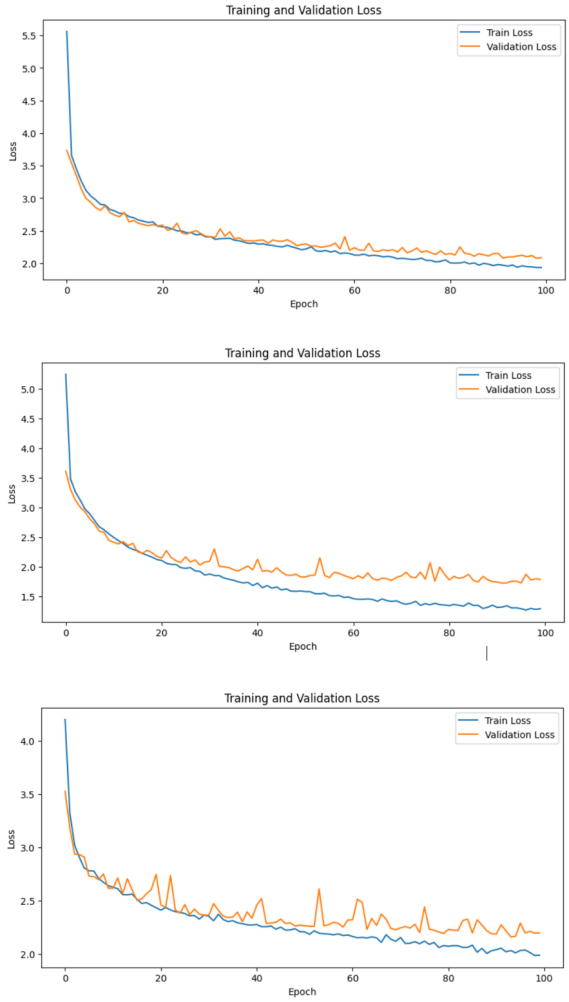


Figure 7: Learning curves for Simple GCN, GraphSAGE and GAT training

The **classification report** (referring to the test set) is summarized in the Table 4.4 and the confusion matrices are reported in Figure 8.

Architecture	Accuracy	F1-score for Benign	F1-score for Malware
Simple GCN	0.98	0.50	0.99
GraphSAGE	0.98	0.73	0.99
GAT	0.97	0.50	0.99

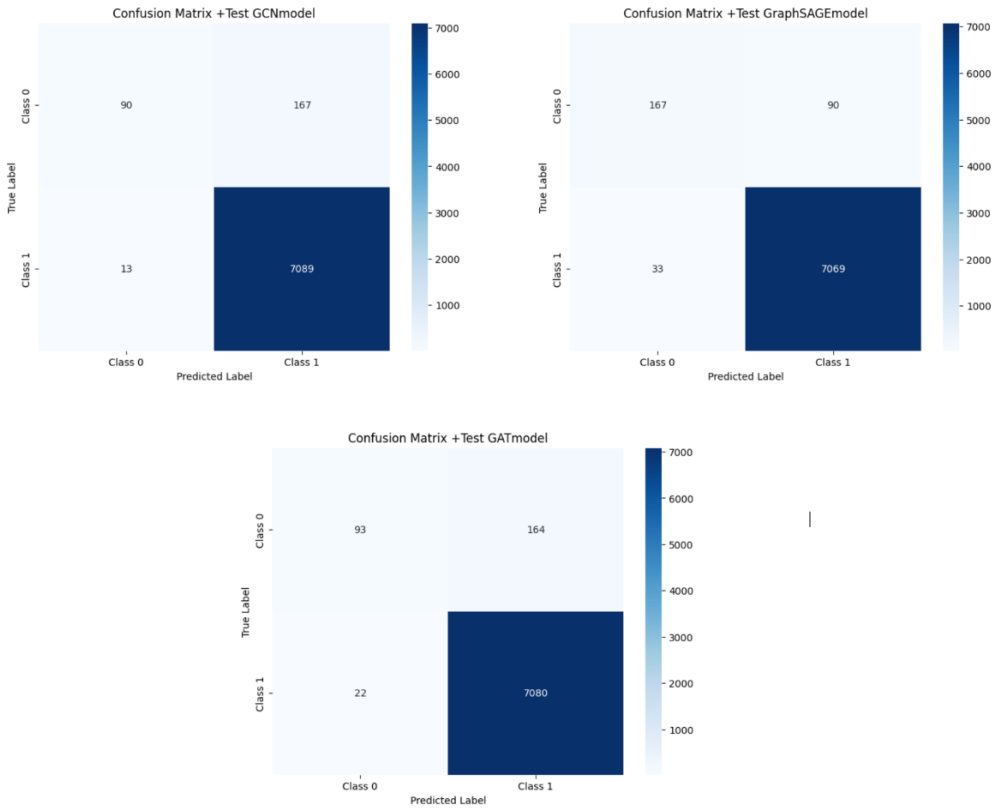


Figure 8: Confusion matrices for Simple GCN, GraphSAGE and GAT architectures

One observation is that the **recall values** for the benign class remain consistently low across all models (see the notebook for more details). The **precision scores** for all three architectures are relatively high, ranging between 0.81 and 0.87, indicating that when the models predict the benign class, they are usually correct. However, the recall scores, which measure how many actual benign cases are correctly identified, are much lower: approximately 0.3 for both the first (GCN) and third (GAT) architectures, while **GraphSAGE achieves a significantly better recall** of around 0.65. This suggests that GraphSAGE is more effective at detecting benign samples compared to the other models.

This is likely because it uses a more flexible way to combine information from neighbor nodes. Instead of treating all neighbors the same, GraphSAGE can learn different ways to gather and summarize information, which helps it better recognize the subtle patterns in benign API call sequences.

On the other hand, GCN averages all neighbor information equally, which might make it harder to focus on important details.

We expected GAT to perform better because its attention mechanism can focus on the most important neighbors. However, its recall was still low. Sometimes it can focus too much on a few neighbors and miss others, which could explain this low recall value.

Can you obtain the same performance as with the previous architectures?

Compared to the previous architectures, none of the GNNs completely matched the F1 score of the best-performing BiLSTM, but GraphSAGE came very close. This suggests that GNNs are good at capturing structural relationships in data, and models like GraphSAGE can nearly match RNNs in modeling sequential patterns.