Politecnico di Torino

AI and Cybersecurity

Project 2: Model Engineering

Use of Feed Forward Neural Network (FFNN), Recurrent Neural Network (RNN), and Graph Neural Network (GNN)

Angelo Barbera - s326432, Giorgia Moscato - s324647, Alessandro Genova - s330893

Table Of Contents

1 Task 1: BoW Approaches

1.1 Preprocessing

We preprocessed the dataset to ensure data quality and model robustness:

- **Dropped the `md5hash` column**: Removed as it was irrelevant in the modified dataset, no longer uniquely identifying sequences, reducing unnecessary complexity.
- **Removed duplicates**: Used `drop_duplicates` to eliminate redundant entries, preventing overfitting and improving generalization.
- **Dataset split**: Split `train.json` into 80% training and 20% validation sets, reserving `test.json` for testing to evaluate performance on unseen data.

1.2 Applying Vectorizer

Q: How many columns do you have after using the CountVectorizer? What does this number mean?
Applying `CountVectorizer.fit_transform` to the training set produced a document-term matrix with **249** columns, each representing a unique term in the vocabulary, capturing the frequency of terms across samples.

Q: What does each row represent? Can you still track the order of the processes (how they were called)?
Each row represents a sample, and has the number of words for each vocabulary entry (i.e. for each column). It is not possible to track the order of processes because the row keeps track only of the number of words, not their order.

Q: Do you have any out-of-vocabulary terms from the test set? If yes, how many? How does CountVectorizer deal with them?
The test set contained **26** out-of-vocabulary terms not seen during training. `CountVectorizer` ignores these terms, maintaining **249** columns in the document-term matrix to ensure robust feature representation consistent with training.

1.3 Classifier

Q: Try to fit a classifier (your choice, shallow, deep, or neural network). Report how you chose the hyperparameters of your classifier and what the final performance was on the test set.
We selected a logistic regression classifier for its efficiency and interpretability on large, imbalanced datasets, with hyperparameters:

- **Solver**: `saga`, chosen for its speed in handling large datasets.
- **Max iterations**: 2400, set to ensure convergence, as lower values led to failures.
- **Class weight**: `balanced`, to mitigate the impact of the imbalanced dataset by adjusting for the differing number of malware and benign samples.

The test accuracy was **86%**, with metrics:

Class	Precision	Recall	F1
Benign	0.17	0.83	0.28
Malware	0.99	0.86	0.92

Table 1: Test set performance metrics for logistic regression

The F1 performance of the classifier is very low (28%), especially of the Benign class due to low precision score (17%).

2 Task 2: Feed Forward Neural Network (FFNN)

Q: Do you have the same number of calls for each sample? Is the training distribution the same as the test distribution?
The dataset features variable-length API call sequences, with a mean of **76.13** calls ($\sigma = 8.86$, meaning most sequences vary roughly ± 9 calls around the mean) in the training set and **85.75** calls ($\sigma = 8.88$) in the test set, indicating slightly different distributions that could challenge model generalization.

Q: Suppose you really want to use a simple Feed Forward Neural Network to solve the problem. Can a Feed Forward Neural Network handle a variable number of elements? And why?
FFNNs cannot process variable-length inputs because their input layer has a fixed size, requiring a consistent number of inputs per neuron, which necessitates preprocessing to have fixed sequence length.

Q: 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?
Since from the lab specification we know that the max API call sequence is 100, we padded smaller sequences to **100**. If there are sequences longer than 100, they should be truncated.

Q: Report how you chose the hyperparameters of your final model and justify your choice.
We chose the hyperparameters of the final model by systematically testing different configurations, aiming to maximize the validation benign F1-score under strong class imbalance (malware greatly outnumbering benign samples). Comparing two FFNN variants, sequential IDs and embeddings, we found that the reported hyperparameters produced the best performance.

2.1 Sequential IDs

- **Architecture**: Input layer (size 100), two hidden layers (32 and 16 neurons), output layer (size 2), ReLU activation function, learning rate 0.0001, batch normalization and dropout to reduce overfitting.
- **Training**: Stopped at epoch 58 after 10 epochs without improvements in validation loss (max 100), time 23.14 seconds.
- **Results**: Validation accuracy is **74.92%**, while test accuracy is **75.83%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	10.01	74.19	**17.65**	9.37	70.40	**16.54**
Malware	98.72	74.95	85.21	98.65	76.02	85.87

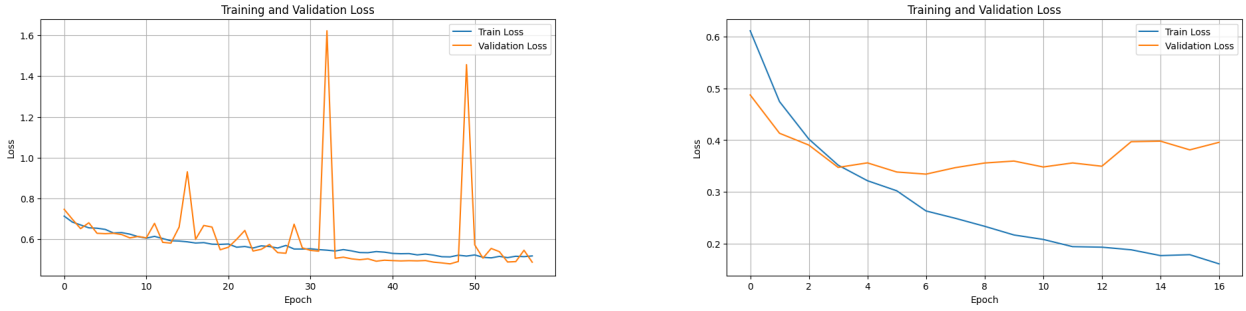
Table 2: FFNN with sequential IDs performance

2.2 Embeddings

- **Architecture**: Embedding layer (size 8), hidden layer (32 neurons), output layer (size 2), ReLU activation function, learning rate 0.0005, batch normalization and dropout.
- **Training**: Stopped at epoch 17 after 10 epochs without improvements in validation loss (max 100), time 6.53 seconds.
- **Results**: Validation accuracy is **91.90%**, test accuracy is **92.83%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	27.63	76.34	**40.57**	29.05	76.80	**42.15**
Malware	99.05	92.48	95.65	99.13	93.39	96.18

Table 3: FFNN with embeddings performance



(a) Sequential ID

(b) Embeddings

Figure 1: FFNN losses

Q: Can you achieve the same results for the two alternatives (sequential identifiers and learnable embeddings)?
No, we can see from the results in Tables 2 and 3 that the model performs better using learnable embeddings, because they capture latent semantic concepts and meaningful relationships within the data. Sequential identifiers, on the other hand, can imply misleading ordinal relationships that do not exist, potentially limiting the model's ability to learn actual patterns. The loss of the FFNN with sequential IDs shown in Fig. 1a, remains stable around 0.5 and it slightly improve. On the other hand, using embedding the loss in Fig. 1b improves only in the first 3 epochs reaching the value of 0.35. The best loss value is achieved using learnable embeddings.

3 Task 3: Recurrent Neural Network (RNN)

The dataset's imbalance, with a dominance of malware samples, led us to prioritize the F1 score of the benign class for model selection. Initial RNN runs with default hyperparameters produced poor benign F1-scores (10–20), leading us to perform a grid search over embedding sizes (4–256) and hidden dimensions (8–512), with learning rate 0.001 and weight decay 0.01. Grid search revealed that smaller embedding sizes (4–32) and moderate-to-large hidden dimensions (64–512) improved benign F1-scores, with LSTM benefiting most from larger hidden sizes due to its ability to capture long-term dependencies.

Q: Do you still need to pad your data? If yes, how?
Padding is required within mini-batches to ensure sequences in the same batch have equal lengths. We used `TimeSeriesDataset` with `collate_fn` in the `DataLoader` to apply `pad_sequence`, dynamically padding shorter sequences to the longest in each batch, minimizing memory usage compared to padding the entire dataset.

Q: Do you need to truncate the testing data? Justify your answer.
No truncation is needed for testing, as RNNs are inherently designed to handle variable-length sequences, allowing them to process test data of any length without modification, unlike FFNNs' fixed-size requirement.

Q: Is there any memory advantage to use an RNN over an FFNN when processing your dataset? And why?
RNNs offer a memory advantage over FFNNs by only padding sequences within mini-batches, not the entire dataset, using `pack_padded_sequence`. In addition to this, RNNs reuse the same weights across time steps, whereas FFNNs require separate weights for each input position. This makes RNNs more memory-efficient for long sequences.

Q: Start with a simple one-directional RNN. Is your network as fast as the FFNN? If not, where do you think the time-overhead comes from?
The monodirectional RNN (45.15 seconds) was significantly slower than the FFNN (6.53 seconds) due to its sequential processing nature, where each time step's output depends on the previous one, limiting parallelization compared to FFNNs' ability to process fixed-size inputs in parallel.

3.1 Simple one-directional RNN

- **Architecture**: Embedding layer, two-layer RNN with tanh nonlinearity and dropout (0.5) to reduce overfitting, followed by a linear layer using the final hidden state.
- **Configuration**: Embedding size 32, hidden size 64, learning rate 0.001, weight decay 0.01.
- **Training**: Stopped at epoch 25 after 10 epochs without improvements in validation loss (max 100), time 45.15 seconds.
- **Results**: Validation accuracy is **84.27%**, test accuracy is **81.60%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	15.82	77.42	**26.28**	12.62	74.40	**21.58**
Malware	99.01	84.53	91.19	98.91	81.86	89.58

Table 4: Monodirectional RNN performance

3.2 Bi-Directional RNN

- **Architecture**: Embedding layer, two-layer bidirectional RNN with tanh nonlinearity and dropout (0.5), concatenating final forward and backward hidden states, followed by a linear layer.
- **Configuration**: Embedding size 16, hidden size 64, learning rate 0.001, weight decay 0.01.
- **Training**: Stopped at epoch 22 after 10 epochs without improvements in validation loss (max 100), time 57.16 seconds.
- **Results**: Validation accuracy is **88.82%**, test accuracy is **88.13%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	19.42	79.57	**31.22**	19.33	78.40	**31.01**
Malware	99.13	87.60	93.01	99.15	88.48	93.51

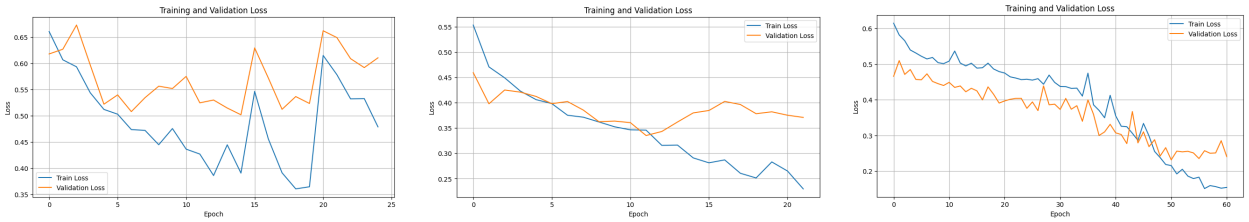
Table 5: Bidirectional RNN performance

3.3 LSTM

- **Architecture**: Embedding layer, two-layer bidirectional LSTM with dropout (0.5), concatenating final forward and backward hidden states, followed by a linear layer.
- **Configuration**: Embedding size 16, hidden size 512, learning rate 0.001, weight decay 0.01.
- **Training**: Stopped at epoch 61 after 10 epochs without improvements in validation loss (max 100), time 787.23 seconds.
- **Results**: Validation accuracy is **85.28%**, test accuracy is **91.94%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	42.42	90.32	**57.73**	27.73	85.20	**41.85**
Malware	99.62	95.39	97.46	99.44	92.18	95.67

Table 6: LSTM performance



(a) Monodirectional (b) Bidirectional (c) LSTM

Figure 2: RNN losses

Q: Can you see differences during their training? Can you see the same performance as the FFNN?
The training of the Monodirectional RNN is very unstable as show in Fig. 2a, the validation loss slightly improve reaching the lowest value of about 0.5. The training of the Bi-Directional RNN is more stable as can be seen in Fig. 2b raching the lowest value of about 0.35. The training of the LSTM shown in Fig. 2c is more unstable than the Bi-Directional RNN, but it reaches the lowest value with respect to the other architectures of about 0.25.
The FFNN with sequential IDs shows the worst performance of all the architectures. On the other hand, the FFNN with embeddings performs better than the Monodirectional and Bidirectional RNNs, achieving almost the same performance as the LSTM.

4 Task 4: Graph Neural Network (GNN)

The dataset is heavily unbalanced, with a dominance of malware samples, so we prioritized the F1 score of the benign class for model selection. Initial GNN runs showed low benign F1-scores (20-30), prompting a grid search over embedding sizes (4-32) and hidden dimensions (128-512), while keeping learning rate at 0.01 and weight decay at 0.01.

To improve detection of the minority class, we applied a custom classification threshold on the GNN outputs: instead of selecting the class with maximum predicted probability (which was almost always malware due to class imbalance), the threshold defines the minimum probability required for predicting the benign class. By increasing the threshold (e.g. 0.7-0.9), the model is less likely to classify samples as malware, which improves precision and F1 for the benign class. This threshold adjustment was effective only for GNNs, because their

graph-level embeddings provide more detailed representations, unlike RNNs or FFNNs.

The grid search showed that smaller embedding sizes (16-32) combined with larger hidden dimensions (128-512) improved overall performance. Attempts to further enhance benign detection using class weight, focal loss, or SMOTE augmentation were unsuccessful.

Q: Do you still need to pad your data? If so, how?
No padding is required for GNNs, as PyTorch Geometric's internal batching automatically handles variable graph sizes and topologies, eliminating manual preprocessing needs, unlike FFNNs and RNNs, which require padding for fixed or batch-aligned inputs.

Q: Do you need to truncate the testing data? Justify your answer with your understanding of why this is or is not the case.
GNN aggregation handles graph of any size, so no truncation is needed. FFNNs, by contrast, require a fixed-size inputs.

Q: What is the advantage of modelling your problem with a GNN compared to FFNN and RNN? Are there any disadvantages?
GNNs are particularly well suited at modeling graph-structured data by capturing node and edge relationships, offering permutation invariance, unlike FFNNs (tabular) or RNNs (sequential). However, deep GNNs risk over-smoothing, where node embeddings lose distinctiveness, and face scalability challenges due to computational costs of graph operations. GraphSAGE, by working with just a subset of neighbors, can achieve greater scalability retaining sufficient representational capacity.

Q: Create a simple GCN and train/test it on CPU and GPU. How long does it take? How does it differ from FFNN and RNN? Why?
We trained a Simple GCN on GPU (57.13 seconds, 40 epochs) and CPU (145.60 seconds, 50 epochs). GNNs are slower than FFNNs (30.66 vs 6.53 seconds) due to complex neighbor aggregation and message passing, and comparable to RNNs (45.15–787.23 seconds), as their computational overhead arises from graph-based operations rather than sequential processing.

4.1 Simple GCN

- **Architecture**: Two GCNConv layers with ReLU, batch normalization, dropout (0.4), and a final linear layer, using global mean pooling to aggregate node features.
- **Configuration**: Embedding size 16, hidden size 128, learning rate 0.01, weight decay 0.01.
- **Training**: Stopped at epoch 40 after 10 epochs without improvements in validation loss (max 50), time 57.13 seconds (GPU).
- **Best Threshold**: 0.9.
- **Results**: Validation accuracy is **96.57%**, test accuracy is **96.49%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	53.97	36.56	**43.59**	52.83	30.11	**38.36**
Malware	97.64	98.83	98.23	97.42	98.99	98.20

Table 7: Simple GCN performance

4.2 GraphSAGE

- **Architecture**: Two SAGEConv layers with ReLU, dropout (0.4), and a final linear layer, using global mean pooling.

- **Configuration**: Embedding size 16, hidden size 256, learning rate 0.01, weight decay 0.01.
- **Training**: Stopped at epoch 20 after 10 epochs without improvements in validation loss (max 50), time 29.33 seconds (GPU).
- **Best Threshold**: 0.9.
- **Results**: Validation accuracy is **97.47%**, test accuracy is **96.57%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	67.95	56.99	**61.99**	53.01	47.31	**50.00**
Malware	98.39	98.99	98.69	98.03	98.42	98.23

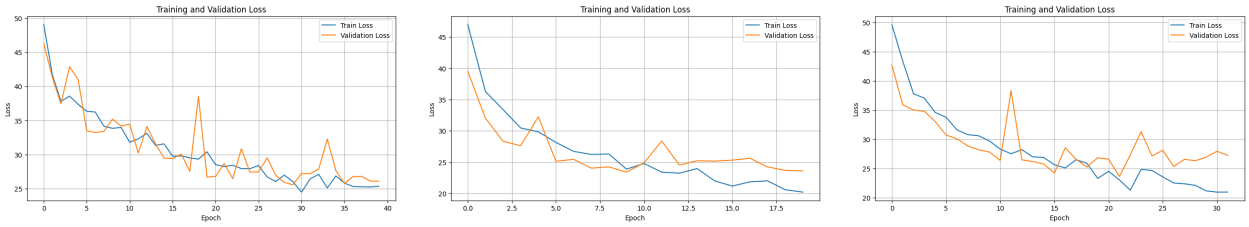
Table 8: GraphSAGE performance

4.3 GAT

- **Architecture**: Two GATConv layers with ELU (4 heads for the first layer, 1 for the second), and a final linear layer, using global mean pooling.
- **Configuration**: Embedding size 16, hidden size 16, learning rate 0.01, weight decay 0.01.
- **Training**: Stopped at epoch 32, time 50.99 seconds (GPU).
- **Best Threshold**: 0.8.
- **Results**: Validation accuracy is **96.07%**, test accuracy is **94.43%**.

Class	Validation			Test		
	Prec.	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
Benign	46.87	64.51	**54.29**	35.63	66.66	**46.44**
Malware	98.64	97.25	97.94	98.70	95.48	97.06

Table 9: GAT performance



(a) GCN (b) GraphSAGE (c) GAT

Figure 3: GNN losses

Q: Can you see any differences in your training? Can you obtain the same performance as with the previous architectures?
The SimpleGCN validation loss shown in Fig. 3a decreases for the first 30 epochs reaching about the value of 25. The GraphSAGE validation loss shown in Fig. 3b decreases for the first 9 epochs reaching the value of 25. The training of the GAT improves for the first 20 epochs and reaches the lowest loss value of about 21, as shown in Fig. 3c.

GNN variants differ in training dynamics: GAT's attention mechanism increases computational cost but enhances feature representation, while GraphSAGE's neighbor aggregation method improves efficiency other than performances. With threshold tuning, GNNs surpassed FFNN performance and matched or exceeded RNNs, particularly in benign F1-scores, by leveraging graph structures for better minority class detection.

4.4 Training Time

Model	Time (s)	Epochs	Epoch/s
GPU			
FFNN (Embeddings)	6.53	17	**2.60**
Mono RNN	45.15	25	**0.55**
Bi RNN	57.16	22	**0.38**
LSTM	787.23	61	**0.08**
Simple GCN	57.13	40	**0.70**
GraphSAGE	29.33	20	**0.68**
GAT	50.99	32	**0.63**
CPU			
Simple GCN	204.92	50	0.41

Table 10: Models training time

The number of epochs required for convergence varied significantly across models: epoch counts are highly case-dependent (depends on initialization and initial randomness) and less reliable for comparing model efficiency. A better metric is epochs per second, which highlights computational efficiency. From this perspective, feed-forward networks achieve the highest throughput (2.60 epoch/s), while GNNs, despite requiring heavier per-epoch computations due to graph operations, maintain reasonable efficiency (0.6–0.7 epoch/s). Recurrent models, in contrast, are slower (0.4–0.5 epoch/s), with the LSTM standing out for its particularly high computational cost (787.23 seconds, 0.08 epoch/s). On CPU, Simple GCN's longer training time (204.92 seconds) and lower epoch-per-second rate (0.41) reflect slower graph computations, yet it maintained competitive performance.

5 Conclusion

The loss curves show irregularities across FFNNs, RNNs, and GNNs, with spikes and oscillations instead of smooth plateau convergence. Several factors may contribute to this behavior: the strong class imbalance combined with mini-batch variability, but also potentially high learning rates. Despite these instabilities, the overall decreasing tendency is preserved, and model selection was ultimately driven by the validation F1 of the benign class, which directly reflects performance on the underrepresented and more challenging category.

Grid search and threshold tuning (0.5–0.9) for RNNs and GNNs identified GraphSAGE as the top model (validation benign F1-score 61.99, test 50.00 at 0.9), followed by GAT (54.29, 46.44) and LSTM (57.73, 41.85). GNNs are generally better because of their ability to leverage graph structures, capturing complex relationships between nodes. Monodirectional (26.28, 21.58) and bidirectional RNNs (31.22, 31.01) were less effective. FFNN with embeddings (40.57, 42.15) outperformed sequential IDs FFNN (17.65, 16.54) due to richer representations and faster training (6.53 seconds vs. 23.14 seconds).