Lab-1-FFNN
Group-7

Simone Capriolo - s<342664>, Salvatore Di Franco - s<346785>,
Alessandro Pispola - s<343720>

This report describes the design, implementation, and evaluation of various FFNN architectures on the CICIDS2017 dataset for network traffic classification. The work covers all aspects of the machine learning pipeline: from data preprocessing and handling dataset biases to experimenting with shallow and deep models, adjusting hyperparameters, and applying regularization. The main purpose was to understand how architectural choices, training strategy, and data characteristics influence model performance, considering common data analysis problems like class imbalance issues and minority attack detection. Through systematic experimentation, the report highlights the strengths and limitations of FFNNs in cybersecurity-oriented classification tasks.

1 Task 1 - Preprocessing

When building a ML pipeline, the first step to perform is data exploration and preprocessing. This operation is essential to better understand the underlying scenario and to prepare data for the subsequent learning tasks. Initially, the CICIDS2017 dataset presented **31507** rows and **17** features. Among these features, 16 were numerical (both float64 and int64 values) while the **Label** feature was categorical. **Label** feature represented the four classes the shallow and deep FFNNs aimed to predict: *Benign*, *Port Scan*, *DoS Hulk* and *Brute Force.*

Q: How many samples did you have before and after removing missing and duplicate entries? By inspecting the dataset, the Flow Bytes/s feature presented 20 NaN and 7 infinite values, while the Flow Packets/s feature presented 27 infinite values. Since these values represented a negligible part of the dataset and could potentially introduce noise, the corresponding samples were removed. Furthermore, 2094 duplicate samples were also removed to prevent data redundancy. So, before cleaning, the dataset had 31507 samples and after the removals, the final shape was 29386 samples and 17 features.

An initial preprocessing step was to encode the Label feature, since models can only process numerical values. To solve this issue, the **Label Encoding** technique was used considering both its simplicity, memory efficiency and the reduced number of possible values. As a result, the following mapping between classes and the corresponding encodings was obtained: Benign mapped with 0, Port Scan with 1, DoS Hulk with 2 and Brute Force with 3.

An additional preprocessing step was the dataset splitting. This phase is crucial, as it guarantees that the data are properly divided into training, validation, and test sets. As specified in the assignement, the dataset was divided in 60% of training and the remaining 40% was divided equally between validation and test set and to ensure the reproducibility, a fixed random seed was set so that data partitions remains constant considering different runs. Specifically, since the dataset represents the traffic from a university network, a strong class imbalance was expected:the Benign class accounts for 65.48% of the samples, while PortScan, DoS Hulk, and Brute Force represent 16.50%, 13.16%, and 4.86% respectively. To preserve the original class proportion across all splits avoiding some classes to be underrepresented or missing affecting performances, the stratified split was performed by using the *stratified* parameter in the *train_test_split* Python function.

Q: How did you normalize the data? Why did you choose it? Before training, it was necessary to analyze the distribution of numerical features to verify the presence of outliers. Outlier detection is an important step, since some unusually large or small values can distort feature scales negatively impacting on the model's ability to learn meaningful patterns. For the sake of visualization, for each feature (excluding label) the percentage of outlier values within its distribution was computed. As shown in Figure 1, the proportion of outliers was below 25%. More specifically, among the 16 numerical features, 8 exhibited more than 10% of outlier values.

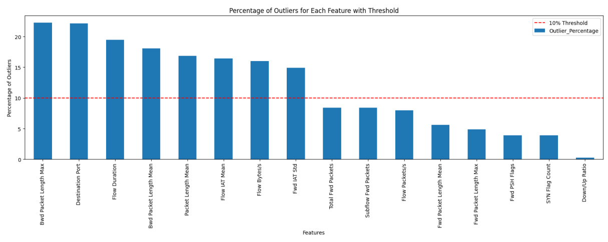


Figure 1: Outliers Histogram

Considering the previous considerations, the **StandardScaler** was applied as it standardizes each feature to zero mean and unit variance. This approach was preferred over the *Min–Max* scaling, which is highly sensitive to outliers and would have compressed data into a narrow range.

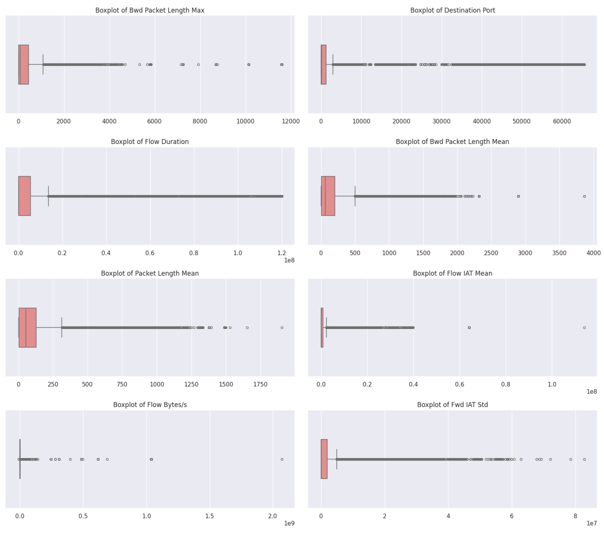


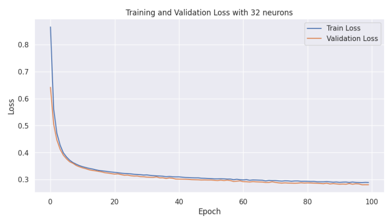
Figure 2: Boxplots of the eight features with the highest percentage of outliers

This decision is further supported by the boxplots shown in Figure 2, which highlight the high presence of outliers across the eight features with the highest percentage of outliers.

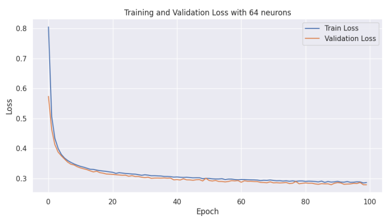
2 Task 2: Shallow Neural Network

Task 2 involved designing a single-layer neural network with 32, 64, and 128 neurons, respectively. All models were trained using a linear activation, AdamW optimizer (learning rate = 0.0005), batch size = 64, cross-entropy loss, no regularization, and up to 100.

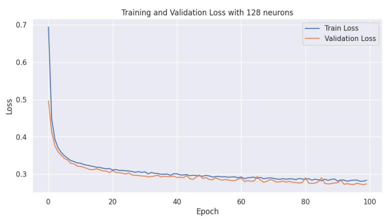
Q: Plot the loss curves during training on the training and validation set of the three models. What is their evolution? Do they converge?



(a) 32 neurons



(b) 64 neurons



(c) 128 neurons

Figure 3: Training and validation loss curves for the three models.

The loss curves for all three models showed a similar pattern. Initially, both the training and validation losses decreased quickly, indicating a rapid learning phase. By moving forward, the decreasing rate slowed and the curves begun to flatten, which suggested that all three models reached convergence. During the training process, the validation loss remained consistently close to the training loss, indicating stability and no significant overfitting or underfitting. The models with 32 and 64 neurons showed smooth and stable loss curves. In contrast, the model with 128 neurons displayed slightly more noise in the validation loss, indicating that its higher complexity may introduce a bit of variance in validation performance.

Q: How do you select the best model across epochs? The best model was selected by monitoring the validation loss and choosing the model from the epoch where this loss reached its minimum. This epoch represents the point of best generalization on unseen data, showing an optimal balance between learning the dataset patterns and avoiding overfitting.

Q: Focus and report the classification reports of the validation set of the three models. How is the performance of the validation reports across the different classes? Is the performance good or poor? Why? The complete classification reports for the validation set of the three models are available in the Jupyter Notebook, under the section titled *"Validation Classification Reports"*. In summary, all three models achieved similar overall performance, with good metrics on the *Benign* (an f1-score of 0.92, 0.92 and 0.93 for the 32, 64 and 128 models respectively), *DoS Hulk* (an f1-score of 0.93 for the 32, 64 and 128 models) and *Port Scan* (an f1-score of 0.89, 0.86 and 0.87 for the 32, 64 and 128 models respectively) classes. However, all three models completely failed to classify the *Brute Force* class correctly, as reflected by the zero precision, recall, and f1-score. This poor performance was a consequence of a significant class imbalance in the dataset, as indicated by the low support value in the classification reports. This can be considered normal when taking into account the simplicity of the model architectures, which consist of a single layer without any activation function to model non linearity or strategies to handle class imbalance.

Q: Now, focus on the best model you chose. Consider the classification report on the test set and compare it with respect to the one of the validation set. Is the performance similar? Since all models perform similarly on the validation set, the model with 64 hidden neurons was selected as a compromise, avoiding models that were either too simple (32 neurons may lead to underfitting) or too complex (128 may introduce the risk of overfitting). Comparing the classification report on the test set with the one on validation, it can be observed that the overall performance remained largely consistent. The accuracy decreased slightly from 0.89 (validation) to 0.88 (test), and the macro F1-score was almost unchanged (0.68 vs. 0.67), indicating that the model generalizes well to unseen data. Overall, the comparison showed that the model that the model generalized well all classes but the *Brute Force* one.

After selecting the best model, the next step was to introduce non-linearity by adding the ReLU, a non-linear activation function.The previously selected best model was retrained, evaluating how performance changed adding non linearity.

Q: Focus and report the classification report of the validation set. Does the model perform better in a specific class? The classification report for the validation set of the model with ReLU and 64 neurons showed a clear improvement compared to the previous linear model. This improvement was related to the introduction of a non-linear activation function. As already explained before, the previous models were entirely linear, therefore, regardless of the number of neurons, a single layer without any non-linear activation function could only separate data points using linear decision boundaries. As a result, all classes pointed out high precision and recall, with significant improvements for the previously problematic *Brute Force* class, which obtained a precision of 0.80, recall of 0.97, and F1-score of 0.88. Also the accuracy metric increased from 0.89 to 0.95. This confirmed that introducing non-linearity allowed the model to better capture the patterns of the *Brute Force*.

Q: Would it be correct to compare the results on the test set? It would not be correct to compare results on the test set at this stage. The reason is that the test set is intended to provide an evaluation of the final model in terms of generalization. In a machine learing and deep learning task, the test set is ment to verify how the model is performing over unseen data, comparing those performances with validation ones. Using it to compare models or make design decisions may introduce bias, as the model may indirectly overfit to the test data. Therefore, comparisons should primarily rely on the validation set, while the test set should only be used once the final model is selected and all hyperparameters are fixed.

3 Task 3: The impact of Specific Features

One important issue when dealing with a dataset consists in biases in the data that can force the model to learn non-effective patterns and lead to wrong inductions. In case of the CICIDS2017, all *Brute Force* attacks were destinated to port 80. Task 3 investigate the impact of the Destination Port feature over the entire dataset.

Q: Is this a reasonable assumption? This is not a reasonable assumption: *Brute Force* attacks can be destinated to any port, indipendently from the service exposed by a machine and so the service port itself. This bias in the dataset is affecting the model performance: it is likely to classify any flow with destination port 80 as a *Brute Force* attack. To further investigate this hypotesis, port 80 was replaced with port 8080 only for the *Brute Force* datapoints in the Test Set. The best model trained on Task 2 was retested over the modified Test set. The expected result was a significant drop in the classification performances over *Brute Force* class.

Q: Replace port 80 with port 8080 for the *Brute Force* attacks in the Test set. Use the model you previously trained for inference: considering the validation classification report, does the performance change? How does it change? Why? As expected, this change in the Test Set significantly impacted the model's ability in classifying datapoints belonging to this class. Considering the corresponding classification report, precision and recall for the *Brute Force* class dropped drastically (from 0.77 and 0.95 to 0.15 and 0.05 respectively). This indicates that the model was previously relying heavily on the 'Destination Port' value of 80 to identify *Brute Force* attacks, now struggles to correctly classify these attacks. Consequently, the f1-score for *Brute Force* showed a significant decrease (from 0.85 to 0.07), reflecting the poor performance on this class. The metrics related to the other classes (*Benign*, *PortScan*, *DoS Hulk*) remained relatively stable, suggesting that the model's reliance on the *Destination Port* feature was primarily for *Brute Force* classification. This technique effectively demonstrated that the presence of this specific bias in the original dataset forced the model to learn a non reasonable pattern in data resulting in wrong inductions.

To address the bias, the *Destination Port* feature was removed from the original dataset and all pre-processing steps were repeated. The resulting version of the dataset was then used for all subsequent steps.

Q: How many *PortScan* do you now have after preprocessing (e.g., removing duplicates)? How many did you have before? As already explained in the Task 1 section, the dataset was affected by a strong class imbalance. In particular, before addressing the Destination Port bias, the *PortScan* class had a support of 4849 samples, the second most populated class consisting in the 16.50% of the entire dataset. By removing the *Destination Port* feature, the total number of duplicated rows increased dramatically, from 2,094 to 9,011. After preprocesing (in particular removing duplicates), the total support of *PortScan* class dropped to 285 samples, being the least populated class (1.27% of the entire dataset).

Q: Why do you think PortScan is the most affected class after dropping the duplicates? This drastic increase in the number of duplicated rows is likely because *Port Scan* attack flows are highly similar in terms of all feature values except for the *Destination Port* feature. Once that column is removed, many data points become identical and are consequently dropped as duplicates.

Q: Are the classes now balanced? Both before and after removing the feature, classes weren't balanced at all: the majority of samples are *Benign* (75.17%), followed by *DoS Hulk* (17.21%), *Brute Force* (6.35%), and the minority *Port Scan* class (1.27%). As already expressed before, considering the scenario imbalance is expected: the dataset records network traffic from a university environment, which is hopefully predominantly benign. However, if not properly addressed, such class imbalance can lead to poor performance of deep learning models.

4 Task 4: The impact of the Loss Function

After the preprocessing performed in Task 3, which involved removing the 'Destination Port' feature, the task 4 started with the training on the modified dataset using the model with the ReLU activation function and 64 neurons.

Q: How does the performance change? Can you still classify the rarest class? The removal of the 'Destination Port' feature has a significant impact on performance, particularly on the rarest class, PortScan, which now constitutes only 1.27% of the dataset. The model, retrained on this new version of the dataset, showed a considerable deterioration in its ability to classify this specific type of attack. As shown in the classification report below (table 1), the model achieved a precision of only 0.32 and a recall of 0.28 for the PortScan class. The performance on the other classes (Benign, DoS Hulk, Brute Force) remained high, but the model effectively struggled to classify the minority class. This was an expected consequence, as the Destination Port feature was likely a key discriminator for identifying PortScan traffic.

Class	Precision	Recall	F1-score	Support
Benign	0.97	0.97	0.97	3378
PortScan	0.32	0.25	0.27	57
DoS Hulk	0.98	0.93	0.96	774
Brute Force	0.81	0.94	0.87	285
Accuracy			0.95	4494
Macro avg	0.77	0.77	0.77	4494
Weighted avg	0.95	0.95	0.95	4494

Table 1: Classification report on the test set (no weighted loss).

Class	Precision	Recall	F1-score	Support
Benign	0.99	0.91	0.95	3378
PortScan	0.23	0.88	0.37	57
DoS Hulk	0.89	0.96	0.92	774
Brute Force	0.78	0.95	0.85	285
Accuracy			0.92	4494
Macro avg	0.72	0.92	0.77	4494
Weighted avg	0.95	0.92	0.93	4494

Table 2: Classification report on the test set (weighted loss).

A possible solution to improve the classification performance for PortScan samples is to use a Weighted Loss function, where misclassifications related to underrepresented classes have a higher cost than errors in majority classes. This forces the model to pay more attention in classifying samples of the minority classes during training, reducing the tendency to bias predictions toward the dominant class.
Q: Which partition do you use to estimate the class weights? To compute the class weights, we used the y_train partition and so the label of the training set. The goal of using a weighted loss is to balance the influence of classes during training. Therefore, the weights must reflect the distribution of classes that the model "sees" while learning, i.e., the training set. Using the training set ensures that the information from the validation and test sets does not influence the training process, maintaining their integrity for an unbiased evaluation.

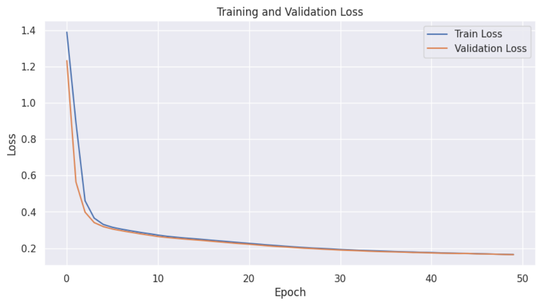
Q: How does the performance change per class and overall? In particular, how does the accuracy change? How does the f1 score change? By introducing class weights into the Cross-Entropy Loss function, the model was re-trained. The new approach forces the model to give more importance to correctly classifying the PortScan class. The resulting performance shows a clear trade-off.
Considering the Classification report (table 2), as a result the model is encouraged to classify samples as PortScan even when they are not. This behavior is reflected in the precision and recall values for the PortScan class: precision drops to a low value of 0.23 because the number of false positives increases (the model predicts PortScan when it is not), while recall becomes significantly higher at 0.88, since very few actual PortScan samples are missed. The f1-score for the PortScan class, being a harmonic mean of precision and recall, improves from 0.27 to 0.37, indicating a better balance in identifying this specific class.
However, this targeted improvement comes at a cost to the overall performance. The overall accuracy decreases from 0.95 to 0.92, and similarly, the weighted avg f1-score falls from 0.95 to 0.93. This decline is a direct consequence of the increased false positives for PortScan. Nevertheless, in a cybersecurity context, this trade-off is often desirable: a higher False Positive rate is generally preferred over a False Negative (missing an actual attack like PortScan).

5 Task 5: Deep Neural Network

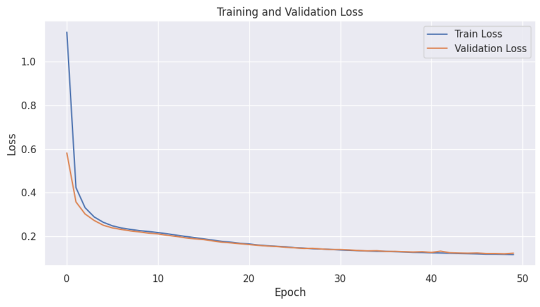
After investigating the impact of the loss function over the performance of shallow learning FFNNs, deep neural networks were deployed. In particular, each architecture was characterized by a number of layers ranging from 3 to 5 while the number of hidden neurons varies across each layer. As concerns the input layer and the output layer, here the number of neurons was fixed and depended on the number of input features for each datapoint (15 in total) and the number of output class (4 in total).

The six tested configurations were as follows: for the 3-layer networks, (15, 8, 16, 4, 4) and (15, 32, 16, 8, 4); for the 4-layer networks, (15, 4, 16, 2, 4, 4) and (15, 32, 16, 8, 4, 4); and for the 5-layer networks, (15, 8, 4, 4, 8, 2, 4) and (15, 32, 32, 32, 16, 16, 4). All models were trained for 50 epochs, with early stopping basing on a convergence criteria, AdamW optimizer, learning rate of 0.0005, batch size of 64 and Cross-Entropy loss function. Moreover, each layer used ReLU as activation function. No explicit regularization techniques were used, and default weight initialization was adopted.

Q: Plot and analyze the losses. Do the models converge? Starting from the 3-layers architectures, both models converged as shown in figure 4a and 4b.



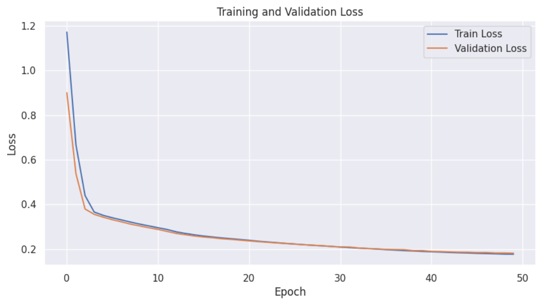
(a) Loss curves for the first 3-layer architecture



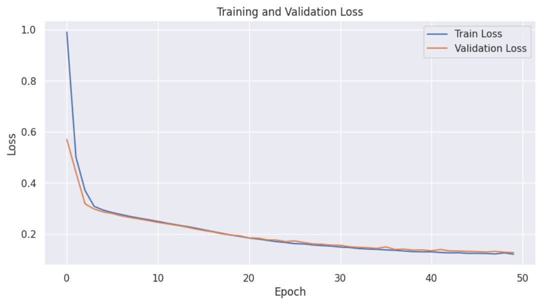
(b) Loss curves for the second 3-layer architecture

Figure 4: Losses curves trends for the 2 3-layer architectures.

The training and validation losses resulted respectively in low values: 0.1767 and 0.1756 for the first configuration , 0.1263 and 0.1299 for the second model. In both cases, during training the curves remained tightly grouped, indicating model stability and the absence of both overfitting and underfitting, until they reached the final plateau. Considering the 4-layers architectures, the first model reached a train loss of 0.1893 and a validation loss equal to 0.1930. Moreover, both training and validation curves seemed not to reach a plateau as shown in figures 5a and 5b.



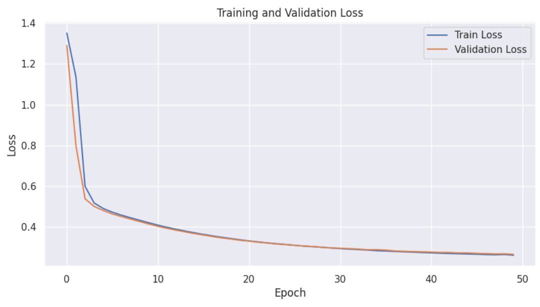
(a) Loss curves for the first 4-layer architecture



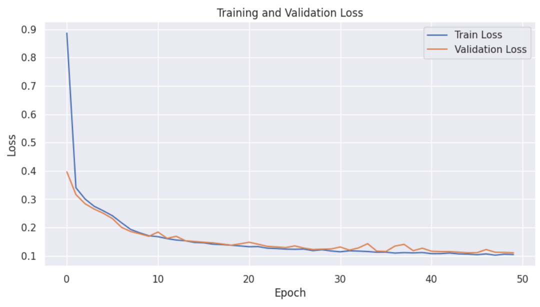
(b) Loss curves for the second 4-layer architecture

Figure 5: Losses curves trends for the 2 4-layer architectures.

So, increasing the number of epochs (fixed by the assignment) might lead to lower values of both losses. The second model showed a similar trend but reached lower losses (Train: 0.1303, Val: 0.1380), indicating that more epochs could further enhance performance. As concern the 5 layers architectures, here performances were different. Train and validation curves for the first model start from an high value and reach a flat shape around 0.27 for both losses (Train loss: 0.2728, Val Loss: 0.2764). The second model, instead, has a better performance, converging with a train loss of 0.1118 and a validation loss of 0.1271 as shown in figures 6a and 6b.



(a) Loss curves for the first 5-layer architecture



(b) Loss curves for the second 5-layer architecture

Figure 6: Losses curves trends for the 2 5-layer architectures.

Q: Calculate the performance in the validation set and identify the best-performing architecture. How do you select one? Table 3 summarizes the performance of the tested architectures on the validation set. The analysis of the classification reports revealed a clear trend regarding the model's ability to handle class imbalance.

Model Configuration	Macro F1	Benign F1	DoS Hulk F1	Brute Force F1	PortScan F1
3-Layers (Small)	0.68	0.96	0.92	0.85	0.00
3-Layers (Large)	0.71	0.97	0.94	0.94	0.00
4-Layers (Small)	0.68	0.96	0.92	0.83	0.00
4-Layers (Large)	0.70	0.97	0.90	0.94	0.00
5-Layers (Small)	0.47	0.94	0.93	0.00	0.00
5-Layers (Large)	**0.92**	**0.98**	**0.94**	**0.93**	**0.82**

Table 3: Comparison of different architectural configurations on the validation set.

Both 3-layer and 4-layer models struggled significantly with the minority class *PortScan*, achieving precision and recall values of 0.00. While these models performed well on the majority *Benign* class and maintained acceptable scores for *DoS Hulk* and *Brute Force*, the insufficient representational capacity limited their capacity to capture rare attack patterns. The first 5-layer model (the narrower one) performed the worst, with a macro-average F1-score of 0.47, failing to classify both *PortScan* and *Brute Force* instances. This suggests that the architecture was poorly designed for this depth, lacking the representational capacity to capture patterns in minority classes.
In contrast, the larger and wider 5-layer model achieved a macro-average F1 score of 0.92. As shown in Table 3, this was the only architecture capable of correctly classifying the *PortScan* class (Precision: 0.82, Recall: 0.82). To verify the generalization capability, the model was tested on the test set, obtaining performance metrics similar to the validation set as depicted in table 4.

Class	Precision	Recall	F1-score	Support
Benign	0.98	0.98	0.98	3378
PortScan	0.85	0.79	0.82	57
DoS Hulk	0.97	0.94	0.95	774
Brute Force	0.91	0.95	0.93	285
Accuracy			0.97	4494
Macro avg	0.93	0.91	0.92	4494
Weighted avg	0.97	0.97	0.97	4494

Table 4: Classification report for the second model with 5 layer architecture on the test set.

This demonstrated that increasing both width and depth significantly enhanced the network's ability to learn rare patterns and generalize across all attack types. Consequently, this architecture was selected as the best one. After finding this optimal configuration, the focus shifted to exploring model performance considering the following batch sizes: 4, 64, 256, 1024.

Q: Does performance change? And why? Report the validation results. Model performance was strongly affected by the batch size, particularly regarding the ability to classify classes with low support. Table 5 summarizes the validation results.

Batch Size	Macro F1	Benign F1	DoS Hulk F1	Brute Force F1	PortScan F1
4	0.88	0.98	0.95	0.94	0.66
64	**0.91**	**0.98**	**0.94**	**0.94**	**0.79**
256	0.72	0.98	0.95	0.94	0.00
1024	0.68	0.96	0.92	0.85	0.00

Table 5: Impact of Batch Size on validation metrics.

Small Batch Sizes (4, 64): With a batch size of 4, the model achieved high accuracy but showed instability. The validation loss curve was extremely noisy, showing random spikes throughout the epochs. The architecture achieved a precision of 0.59 and a recall of 0.74, indicating that while it correctly identified most instances of this class, it also misclassified some. The batch size of 64 proved to be the optimal configuration. It maintained the high accuracy of the smaller batch but with a smooth, stable convergence (Train Loss: 0.1099, Val Loss: 0.1095). Crucially, it achieved the best balance for the minority class *PortScan* (F1: 0.79).

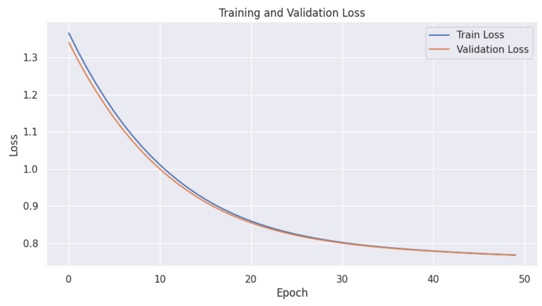
Large Batch Sizes (256, 1024): Increasing the batch size to 256 and 1024 led to a significant performance drop on minority classes. As shown in Table 5, the model with batch size 256 failed completely to detect *PortScan* instances (F1: 0.00). With the largest batch size of 1024, the performance dropped further, affecting other attack classes as well (Macro F1 decreased to 0.68). This occurs because larger batches reduce the number of iterations per epoch (fewer gradient updates). Consequently, the model has fewer opportunities to adjust its weights for rare samples. Furthermore, the gradient computed over a large batch tends to be dominated by the majority class (*Benign*), "averaging out" the signal from rare attacks and leading the model to converge towards a local minimum that favors the majority class.

Q: How long does it take to train the models depending on the batch size? And why? Looking at the execution outputs: using a batch size of 4 required approximately 358 seconds, while batch sizes of 64, 256, and 1024 reduced it to about 33.4, 15.7, and 13.2 seconds respectively.

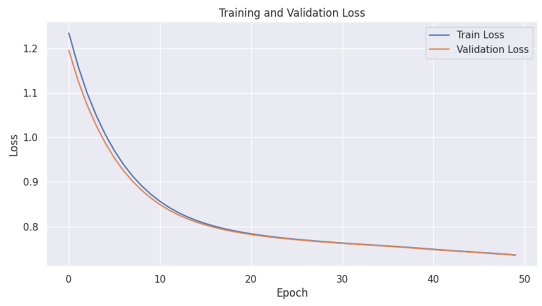
Data shows a clear trend: training time decreases as soon as the batch size increases. This is normal and happens because a larger batch size implies that the model processes more data before recomputing and updating the gradient. This means that in case of a small batch size, gradient updates are more frequent but iterations are less time consuming. Despite that, there are only few iterations required to go through the entire dataset when using a large batch size, while smaller batch sizes require many more iterations to process the entire dataset. This leads to a longer overall time even though each iteration is faster.

Following the batch size analysis, the impact of different optimizers was evaluated on the best 5-layer model. The comparison included Stochastic Gradient Descent (SGD), SGD with Momentum (0.1, 0.5, 0.9), and AdamW. All tests utilized a fixed learning rate of 0.0005 and the optimal batch size of 64 determined in the previous step.

Q: Is there a difference in the trend of the loss functions? As concern classification performance there is a significant difference in the trend of the loss functions considering several optimizers. In case of SGD (without and with 0.1 and 0.5 momentum) the training and validation losses decreased very slowly remaining high throughout the epochs of the training process, starting from values around 1.2 and reaching values around 0.8 as shown in figures 7a and 7b.



(a) Loss curves for SGD with 0.1 momentum



(b) Loss curves for SGD with 0.5 momentum

Figure 7: Losses curves trends for SGD

This indicates that the model is struggling to converge and learn effectively pattern in data. The curves trend in all of the cases suggest that performance can be enhanced by increasing the number of epochs. Considering SGD with momentum 0.9, the loss decreases at a faster rate, but it still appears not to reach a plateau in 50 epochs as illustrated in figure 8.

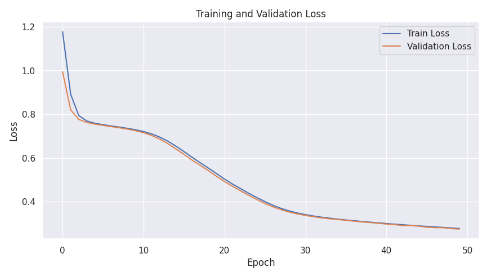


Figure 8: Loss Cuvers for SGD with 0.9 momentum

Moreover it seems to be converging slower and to a higher loss (Train Loss: 0.3032, Val Loss: 0.3011) value than the last optimizer, AdamW.

With AdamW the training and validation loss decreases much more rapidly and reaches a lower value (Train Loss: 0.1032, Val Loss: 0.1031) within the same number of epochs as shown in figure 9.

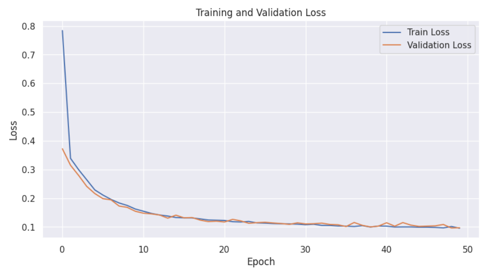


Figure 9: Loss Cuvers for AdamW optimizer

The loss curves show a smoother and more consistent downward trend, indicating efficient convergence. The difference in loss trends highlights that AdamW is much more effective at minimizing the loss function for this specific model architecture and dataset compared to all the tested SGD variants.

Q: How long does it take to train the models with the different optimizers? And why? The training time for each optimizer with 50 epochs and a batch size of 64 varied as follows: SGD required 21.93 seconds, SGD with momentum 0.1 took 25.3 seconds, with momentum 0.5 took 25.9 seconds, with momentum 0.9 required 24.4 seconds, and AdamW took the longest at 34.5 seconds.

From a temporal efficiency point of view, all the SDG variants are quicker than AdamW. This gap in time performances can be explained by computational complexity of the internal operations of each optimizer. The SDG is the fastest since it simply performs gradients updates with a relatively low computational cost and overhead. Despite the value, adding the concept of momentum slightly increases the time since additional calculation are required. Despite that, the impact remains modest. In contrast, AdamW is the slowest optimizer among the tested ones: it computes and updates both the first and the second moments estimates for each parameter, leading to an higher computational cost. Therefore, while AdamW often provides faster convergence in terms of epochs, it requires more time per epoch due to its more complex update rules.

Q: Now, focus on the architecture with the best optimizer. Evaluate the effects of the different learning rates and epochs. Report the test results for the best model. The second 5 layer model with AdamW as optimizer was tested, evaluating the performances through an hyperparameter grid where three different learning rates (0.001, 0.0005, 0.0001) and number of epochs (50, 100) were tested. A trend was noticed: as the learning rate decreased, the curves became smoother and less noisy, although training progressed more slowly; moreover, increasing the number of epochs made reaching a stable plateau much more likely, ultimately leading to improved overall performance.

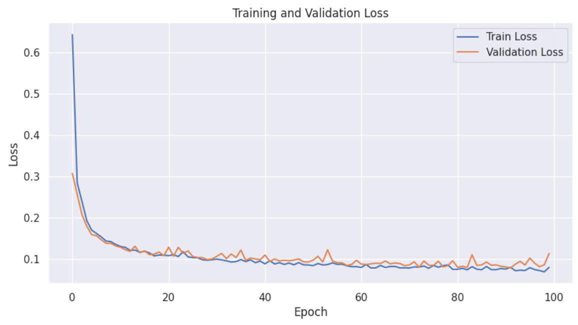


Figure 10: Best Deep Neural Network train and validation curves (lr: 0.001, epochs: 100)

Class	Precision	Recall	F1-score	Support
Benign	0.98	0.99	0.98	3378
PortScan	0.94	0.82	0.88	57
DoS Hulk	0.98	0.93	0.95	774
Brute Force	0.93	0.97	0.95	285
Accuracy			0.97	4494
Macro avg	0.96	0.93	0.94	4494
Weighted avg	0.97	0.97	0.97	4494

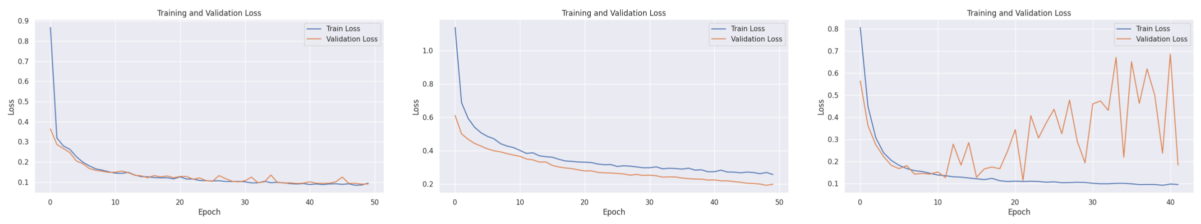
Figure 11: Best model classification report on validation set.

The best Deep Feed Forward Neural Network was the one with the 5 layer architecture: an input layer of 15 neurons, three hidden layers of 32 neurons each, two hidden layers with 16 neurons each, and a final output layer with 4 neurons. The batch size used was 64, the optimizer AdamW and loss the Cross Entropy since it's a multi-class classification problem. As concerns the learning rate and epochs, the best performance was achieved with 0.001 and 100 respectively. Despite the slightly noisy trend, figure 10 shows that both train and validation loss reached a plateau indicating model convergence. Table 11 highlights good performances over all classes, with a macro-average f1 score of 0.94. Performance was solid on the benign class, lower recall as concerns *PortScan* and *DoS Hulk* in favour of an higher precision and an opposite trand in case of *BruteForce* class. The classification report over the test set that is available in the Jupiter Notebook, under the section titled *"The impact of lr and epochs"*, shows very similar performances with respect to table 11, meaning the model performance mantained solid also considering the unseen data of the test set.

6 Task 6: Overfitting and Regularization

In task 6, the performance of a deeper network architecture and the impact of different regularization techniques were evaluated. The baseline model was a Feed-Forward Neural Network (FFNN) with 6 hidden layers, with 256, 128, 64, 32, 16 and 16 neurons each. The non-linear activation function used was the ReLU with the default weight initialization. The model was trained over batches of size 128, minimizing the Cross-Entropy loss. The number of training epochs was 50 (with early stopping) with a learning rate of 0.0005 and optimizer AdamW. Initially, no particular regularization techniques were deployed.
First, the model was trained to establish a performance baseline and to analyze its learning behavior.

Q: What do the losses look like? Is the model overfitting? Despite a deep neural network with a high number of layers and neurons per layers, like the one defined, is prone to overfitting the obtained result was surprisingly stable even without applying any specific regularization techniques (figure 12a.



(a) Loss Cuvers for FFNN without regularization techniques.

(b) Loss Cuvers for FFNN with dropout techniques.

(c) Loss Cuvers for FFNN with batch normalization.

Figure 12: Regularization techniques impact.

Q: What impact do the different normalization techniques have on validation and testing performance? To mitigate the overfitting of the deep model and improve its generalization capabilities, three different regularization techniques were implemented and evaluated: Dropout, Batch Normalization, and Weight Decay (L2 Regularization).
Impact of Dropout: Dropout yielded to a worst performance improvements compared to the baseline. As shown in figure 12b, the validation curve consistently remained below the training curve, a clear indication of underfitting. Moreover, both validation and test scores were worst (Macro F1 dropped from 0.85 to 0.68). Crucially, the model failed to detect the *PortScan* class (F1 = 0.00), and the *Brute Force* class suffered from low precision despite high recall, indicating persistent false positives. Overall, dropout proved ineffective in mitigating overfitting or improving generalization for the minority classes in this specific architecture.

Impact of Batch Normalization: Batch Normalization achieved an high instability in terms of loss curves (12c. In terms of classification report, it achieved slightly better overall performance (Macro F1: 0.71, Weighted F1: 0.95) with respect to dropout, yet the *PortScan* class remained undetected (F1 = 0.00). Results on *Brute Force* and *Benign* classes were again strong and consistent across validation and test sets.
Impact of Weight Decay (L2): As reported on the Jupiter Notebook under the section *"Impact of Weight Decay"*, this technique proved to be the most effective strategy. All decay values offered the most balanced results similarly to the situation without regularization techniques.