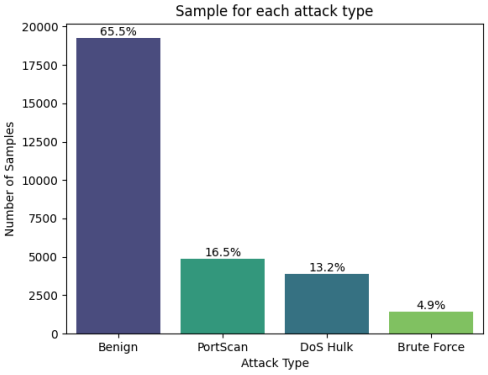
Project 1: Introduction to Deep Learning

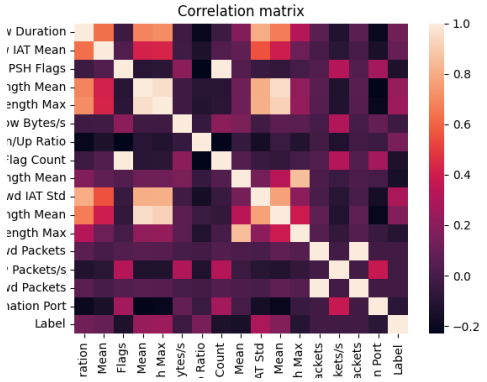
Scursatone Matteo - s332153, Lorenzato Fabio - s332186, Milani Alessandro - s332136

1 Task 1: Data Preprocessing

During this phase, we had to pre-process the dataset to make it suitable for the training phase, this means that we had to remove infinite and NaN values (27 entries in total), duplicate rows (2123) and any negative values (2) since they were not representative of any real world scenario, meaning that in total we lost 2123 sample rows, which is 6.74% of the total dataset. During the preprocessing phase, first thing we noticed was the dataset was really **imbalanced**, as you can see from figure 1a, with the most represented class was more than tenfold the size of the least represented one, but we decided to keep it as is at this step since we are required to address this one later on in the project. We also noticed that some of the features were **highly correlated**, as you can see from figure 1b, with the `Fwd PSH Flags` and `SYN Flag Count` features having a correlation index of 1, meaning they are perfectly correlated. This is also true for the `Subflow Fwd Packets` and `Total Fwd Packets`, and the correlation holds after splitting the dataset so we decided to drop one for each pair.



(a) Class distribution histogram



(b) Correlation matrix between features

For what concerns the **outliers**, as you can see from figure 2, there are many. In fact, almost 60% of each data partition is considered an outlier for at least one feature. Since we have so many of them, normalization was not an option so we decided to use **standardization**, which is less sensitive to outliers. At this end, we applied it at first to the training set, then we used the same parameters to standardize the validation and test sets. This is to **avoid leaking information** from the test set to the training set, which would be an unfair advantage for the model.

Furthermore, we decided to **split** the dataset into three partitions, a training set (60%), a validation set (20%) and a test set (20%). The split was done in a **stratified** way, meaning that the class distribution is preserved in each partition, which is important to avoid biasing the model towards a specific class. At last, since the true label is a categorical one, we used a simple **label encoding** to convert it into a numerical one. This choice was basically forced since most of the task require to use cross entropy as loss function, which does not support multi-class categorical labels.

2 Task 2: Shallow Neural Network

At this step, the **training process** was carried out using a shallow neural network with a singular hidden layer, as requested. The model was trained using the AdamW optimizer with a learning rate of 0.005 and batch size of 64. We did not implement any early stopping mechanism, since the training process only a handful of seconds. To test the different neurons per layer, we symply changed the number manually and

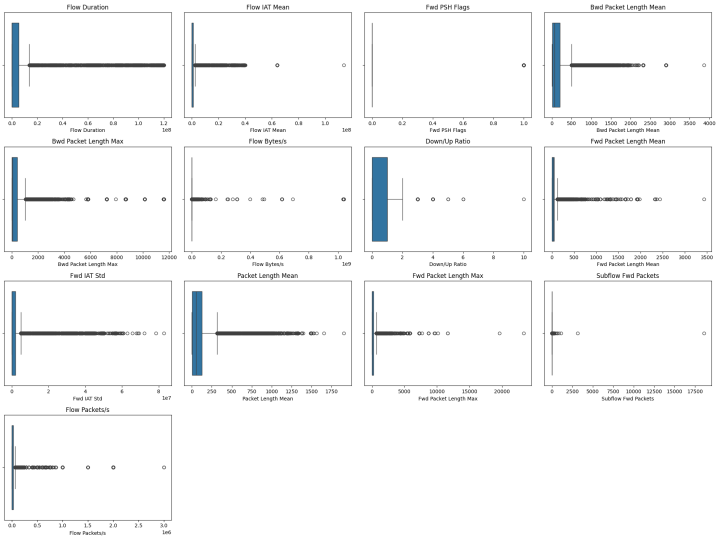
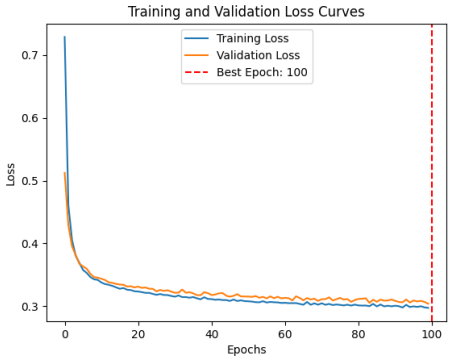
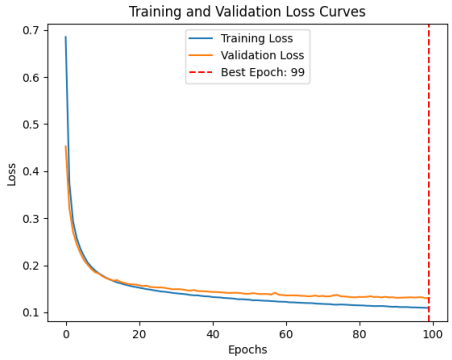


Figure 2: Outliers in the dataset

iterated the training process taking note of the results for each configuration. The best performing model was the one with 128 neurons in the hidden layer, with basically no difference in performance with the ones with 64 and 32 neurons (the weighted average F1 score and the Macro average F1 score were tied in all three cases), which is most likely due to the model being so simple, so the deciding factor was the validation loss, which was slightly lower for the 128 neurons model.



(a) Training and validation loss for the simple linear model (Linear Activation)



(b) Training and validation loss for the simple linear model (ReLU Activation)

The **training and validation loss curves** are shown in figure 3a. Both curves follow the same trend, with the validation curve not showing any sign of overfitting, most likely due to the fact that the number of epoch was limited to 100. The **best model** was **selected** based on the *validation loss*, which was the lowest at the very last epoch (100), for which we serialized via pytorch to load back afterward. The results of the training can be seen in table 1. The model **performs** as expected in both the **validation and test sets** overall (the weighted average F1 score is .87 on the test set), with better results on the most represented class, `Benign`, having an F1 score of .92 in the test set, and the worst on the least represented one, `Brute Force`, as it basically wasn't able to classify it at all. This kind of **poor perforce** is to be expected, since we did not address the class imbalance issue at this step, and also the model is not very complex, being only a linear hidden layer. This means it is not able to fit complex patterns in the data as well/or fitting the most represented classes too well and under representing the least represented one.

Then, we were asked to change the model to use a **ReLU activation function** and things started to change a bit. The training and validation loss curves are shown in figure 3b. The training and validation loss are a little lower (around .17 lower at the best epoch on the validation loss).

The trend of the curves is the same as the previous one, having the best parameter in term of loss reduction at epoch 99. It terms of general performance, the model performs better than the previous one, with a weighted average F1 score around .95 across all the sets. This time around, the model was able to classify the `Brute Force` class just fine, with an f1 score of .86 on the test set, also shown in table 1, which is a significant improvement over the previous model. This basically means that a simple linear

Table 1: Comparison of Classification Model Performance

		Linear Activation			ReLu Activation			Performance Change (%)		
Data Set	**Class**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec. Diff.**	**Recall Diff.**	**F1 Diff.**
Validation Set	Benign	0.88	0.96	0.92	0.97	0.97	0.97	+10.2%	+1%	+5.4%
	Brute Force	0.00	0.00	0.00	0.81	0.96	0.88	/	/	/
	DoS Hulk	0.98	0.86	0.92	0.98	0.94	0.96	0.0%	+9.3%	+4.3%
	PortScan	0.86	0.89	0.88	0.94	0.92	0.93	+9.3%	+3.4%	+5.7%
	Macro Avg	0.68	0.68	0.68	0.93	0.95	0.93	+36.8%	+39.7%	+36.8%
	Weighted Avg	0.85	0.89	0.87	0.96	0.96	0.96	+12.9%	+7.9%	+10.3%
	Accuracy		0.890			0.956			+7.4%	
Test Set	Benign	0.89	0.96	0.92	0.97	0.96	0.97	+9%	0.0%	+5.4%
	Brute Force	0.00	0.00	0.00	0.81	0.91	0.86	/	/	/
	DoS Hulk	0.98	0.87	0.92	0.97	0.95	0.96	−1%	+9.2%	+4.3%
	PortScan	0.83	0.90	0.87	0.92	0.94	0.93	+10.8%	+4.4%	+6.9%
	Macro Avg	0.68	0.68	0.68	0.92	0.94	0.93	+35.3%	+38.2%	+36.8%
	Weighted Avg	0.85	0.89	0.87	0.96	0.95	0.96	+12.9%	+6.7%	+10.3%
	Accuracy		0.889			0.955			+7.4%	

layer is not enough to capture the patterns in the data, which the ReLu function was able to do.

In figure 4 you can see the heatmap of the confusion matrix for all the data partitions. In the test set, the most common misclassifications are between `PortScan` and `DoS Hulk`, which are both commonly confused as `Benign`, which suggests that the model is not able to distinguish between the two classes in some cases.

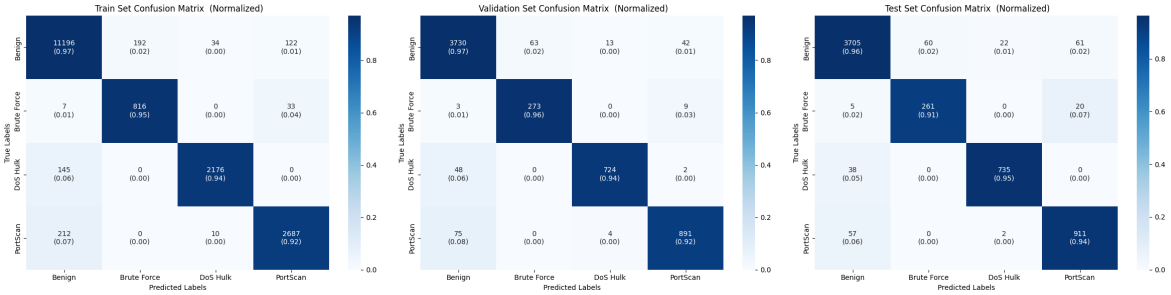


Figure 4: Confusion matrix heatmap for all data partitions (ReLU Activation)

3 Task 3: The impact of specific features

For this task, we had to evaluate the impact of specific features and biases on the model. We noticed that *all* the bruteforce attacks were directed towards the same port (80), which is the default port for HTTP traffic. This means that the model could learn a bias which associates this port with bruteforce attacks, which isn't ideal. This is pretty likely to happen since around 20% of the attacks which have as target the port 80 are bruteforce attacks, as you can see from table 2, which is still a relevant portion of the dataset.

Table 2: Distribution of samples with port 80 as destination port per class

Class	Count	Percentage
DoS Hulk	3868	52.55%
Benign	2058	27.96%
Brute Force	1427	19.39%
PortScan	8	0.11%

Table 3: Top 5 most common destination ports

Destination Port	Count
53	1626
80	1178
443	937
123	43
22	17

We were asked to **replace** the **port 80 with the 8080** for the bruteforce attacks in the test dataset, and by using the **pretrained model** the performance did in fact change. If we don't consider the class imbalance, the model performs *well enough overall*, having an accuracy of around 91% and a weighted average f1 score of .89, which are way worse results than the ones we got in the previous task, since even with non linearity the model is unable to predict correctly the `Brute Force` class again. This is much evident when looking at the confusion matrix in figure 5, which shows an huge bias of the model

towards predicting the `Benign` class for the `Brute Force` class. This shows that the model was not able to generalize the port changes well enough as was indeed overfitting to the port 80 feature. It likely that the model learned more biases than the port, since we would have expected to drift the result toward the most representative class for that port, which is the `DosHulk` class, which holds more than half of the records with port 80 as destination port.

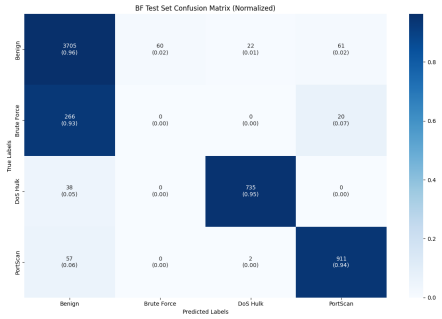


Figure 5: Confusion matrix heatmap for all data partitions (Brute Force Port Changed)

From this point on, we were asked to **remove the port feature** all together, which we will expect to have a negative impact on the model performance, since it is a very important feature for the classification task. After repeating the preprocessing tasks, the most affected class for duplicates dropped was the PortScan one, as you can see from figure 6, having lost more than 4000 samples (94% of its total, from 4849 to 285). This is most likely due to the fact that the only difference between most of the PortScan samples is the port, which also makes sense given the kind of traffic that scanning tools generate. Still, the class are **much imbalanced**, mostly toward the benign class, as it was previously. The result of the new training steps are reported in table 4. As you can see, performance are generally not the worst, with an average weighted F1 score of .95 on the test set, which is an improvement to the previous model, with good results on the previously worst performing class, `Brute Force`, which has now a F1 score of .87 on the test set. However, performance on the `PortScan` class are horrible, with an F1 score on the test set of .40, which is a significant drop from the previous model. This is even more evident when looking at the confusion matrices in figure 7, which clearly shows that most of the PostScans packets are misclassified as the most represented class, `Benign`, which is a clear sign of underfitting for this class.

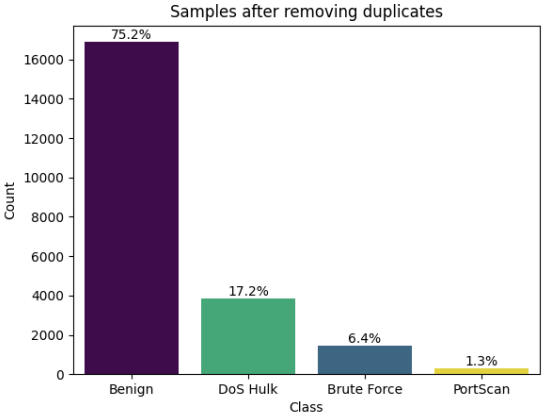


Figure 6: Distribution of the classes after port removal

To address class imbalance, we used the sklearn library to compute the **weights** to feed to the loss function, exactly as requested. When taking that into account, the performance change drastically, as you can see from table 4. The precision generally drops a bit in the least represented classes, but increases in the most represented ones, which generally balance the performance out. The real change in the class results is the recall, which is very much higher, especially in the `PortScan` class, which means that the model is better at identifying the samples of this class, since it's giving out more true positives and less false negatives for this class (the same is not true for all of the others, though). Generally, the weighed average F1 score stays basically the same, while the accuracy drops a little, from 95% to 92%, which means that the model is less accurate overall, but with better performance in the least represented class.

Furthermore, the confusion matrix in figure 8 shows that the model is now able to identify the

Table 4: Comparison of Classification Model Performance (PortScan Precision/Recall Tradeoff)

		No ports, No weights			No ports, Weights			Performance Change (%)		
Data Set	**Class**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec. Diff.**	**Recall Diff.**	**F1 Diff.**
Test Set	Benign	0.96	0.97	0.97	0.99	0.92	0.95	+3.1%	−5.2%	−2.1%
	Brute Force	0.81	0.95	0.87	0.78	0.96	0.86	−3.7%	+1.1%	−1.1%
	DoS Hulk	0.99	0.89	0.94	0.92	0.93	0.93	−7.1%	+4.5%	−1.1%
	PortScan	0.45	0.35	0.40	0.27	0.96	0.42	−40%	+174.3%	+5%
	Macro Avg	0.80	0.79	0.79	0.74	0.94	0.79	−7.5%	+19%	0.0%
	Weighted Avg	0.95	0.95	0.95	0.95	0.93	0.94	0.0%	−2.1%	−1.1%
	Accuracy		0.951			0.927			−2.5%	

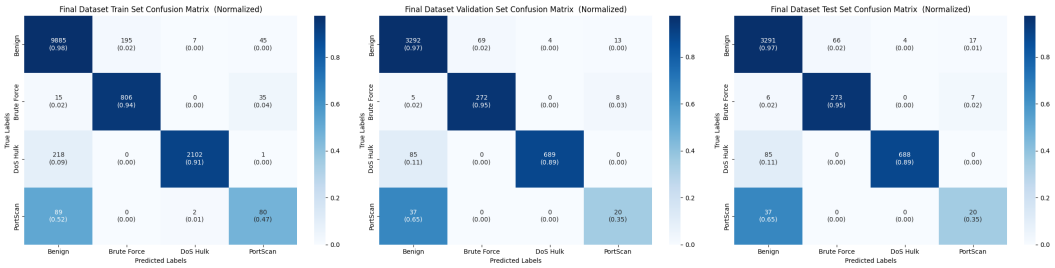


Figure 7: Confusion matrix for all data partitions (Port Removes, No Weights)

`PortScan` class far better.

4 Task 4: Deep Neural Network

For this task we had to select the best performing model given a set of parameters. To this end, an worked good enough, even if it was quite slow to carry out. To this end, we used the `GridSearchCV` function from the sklearn library. The only issue was that the minimum number of folds for the cross validation was 2, so we created a mask to use always the same split. The first thing to consider was the layers layout (results in the notebook), from which the best parameters were chosen using the weighted f1 score as reference, since we were not asked to address class imbalance further. After a general gridsearch with one fold, we carried out another one with 5 folds to not penalize performance on a single fold out of the best configurations. In the end, the best configuration was the one with 5 layers, with the following neurons per layer: (32, 16, 32, 32, 32).

The training and validation loss curves for the best performing model are shown in figure 9. The validation loss starts much lower than the training one and stays lower for the whole training process, without showing any sign of overfitting. Adding more neuron layers also helped to increase performances, allowing to achieve a weighted average F1 score of .96 on the test set, an improvement over the previous weightless model score of .95. The model performs better over the least represented classes, with an f1 score of .57 on the `PortScan` class, from the previous .40, and .94 on the `Brute Force` class, from the previous .87. This is also very evident by looking at the confusion matrix heatmap in figure 10, which shows that the model is now able to identify the `PortScan` class much better, but still misclassifying a lot of samples as `Benign`, exactly as before. At last, the accuracy also increased to 96.1% from the previous 95.1%.

After this, we had to evaluate the **batch size impact** on the model performance. The performances did indeed change, as expected since a smaller batch size means more updates to the weights, which should lead to a better performance.

The results of the comparison are shown in table 5. To choose the better performing model, we used the weighted average F1 score as reference and the best performing model is the one with a batch size of 32 (highlighted in green), due to the fact that it has a slightly higher weighted F1 score, as well as macro average one than the second best performing one (batch size 64). Generally, the batch size had an impact on the model performance: the lowest possible batch size (1) had one of the worst performances out of all, with a weighted average F1 score of .95 but a poor macro average F1 score of .78, due to underfitting on the least represented class, then topped at the next batch size (32) and started to drop again as the hyperparameter increased, as shown in figure 11. In the end, the model with the highest batch size (512) had the worst performance overall, being unable to classify the `PortScan` class at all. This is due to the fact that smaller batch sizes, while being slower to train, allow the model to update the weights more

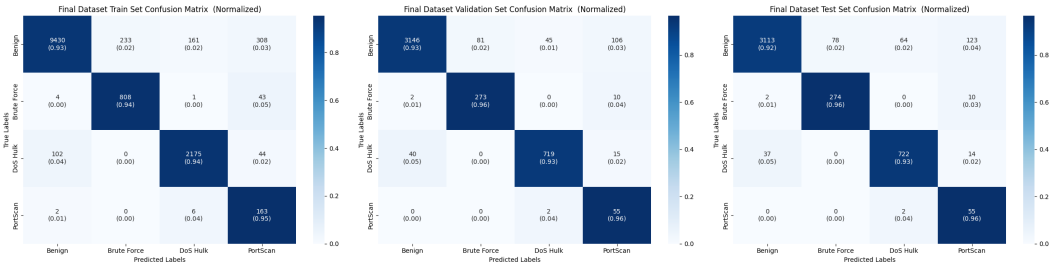


Figure 8: Confusion matrix for all data partitions (Port Removed, Weights)

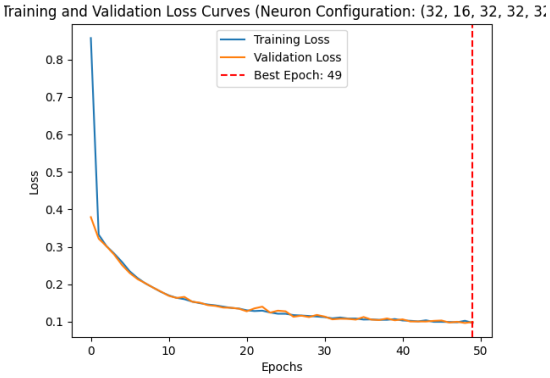


Figure 9: Training and validation loss for the best performing model

frequently, and better generalize the patterns in the data, while larger batch sizes tend to expose the model to less micro variations in the data, which makes learning patterns harder, especially for the least represented classes. The same figure also shows the fitting time for each batch size, with the highest one at batch size 1 (which took almost 10 minutes to fit), to rapidly decrease as the batch size increased to be almost irrelevant.

Table 5: Model Performance Comparison by Batch Size (Validation and Test Sets)

		Batch Size: 1			**Batch Size: 32**			**Batch Size: 64**			**Batch Size: 128**			**Batch Size: 512**		
Data Set	**Class**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**
Validation Set	Benign	0.96	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98	0.97	0.98	0.98	0.95	0.98	0.96
	Brute Force	0.79	0.95	0.86	0.94	0.95	0.94	0.94	0.95	0.95	0.93	0.95	0.94	0.79	0.95	0.86
	DoS Hulk	0.96	0.90	0.93	0.94	0.94	0.94	0.96	0.93	0.94	0.97	0.93	0.95	0.99	0.86	0.92
	PortScan	0.54	0.26	0.35	0.52	0.58	0.55	0.52	0.58	0.55	0.39	0.26	0.32	0.00	0.00	0.00
	Macro Avg	0.81	0.77	0.78	0.84	0.86	0.85	0.85	0.86	0.85	0.82	0.78	0.80	0.68	0.70	0.69
	Weighted Avg	0.95	0.95	0.95	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.93	0.94	0.94
	Accuracy		0.9470			0.9637			0.9642			0.9635			0.9430	
Test Set	Benign	0.96	0.96	0.96	0.98	0.97	0.97	0.98	0.97	0.97	0.97	0.97	0.97	0.95	0.98	0.96
	Brute Force	0.80	0.96	0.87	0.93	0.96	0.94	0.92	0.96	0.94	0.93	0.95	0.94	0.79	0.96	0.87
	DoS Hulk	0.94	0.90	0.92	0.91	0.94	0.93	0.92	0.93	0.93	0.93	0.93	0.93	0.99	0.87	0.93
	PortScan	0.34	0.19	0.25	0.48	0.56	0.52	0.44	0.56	0.50	0.35	0.25	0.29	0.00	0.00	0.00
	Macro Avg	0.76	0.75	0.75	0.83	0.86	0.84	0.82	0.86	0.83	0.79	0.78	0.78	0.68	0.70	0.69
	Weighted Avg	0.94	0.94	0.94	0.96	0.96	0.96	0.96	0.96	0.96	0.95	0.96	0.95	0.94	0.95	0.94
	Accuracy		0.9419			0.9564			0.9557			0.9557			0.9453	

In the end, we settled for a **batch size** of **32** going forward in the task, since it yielded the better performance overall, while being the most accure, with an almost negligible increase in fitting time compared to the batch size of 64. For what concerns the evaluation of the **different activation functions**, we found out that performance did in fact change. A linear activation function yielded the worst performance of all, with a weighted average F1 score of .86 and a macro average F1 score of .48 in the test set, which are somehow worse results that our very first model of this report (with only 1 layer), due to the fact that the model is unable to classify the `PortScan` class at all. This also resulted in having the noisiest loss curves out of all the activation functions tested . Those result were also in line with what we've got while using a *Sigmoid* activation function, with basically identical results for both the validation and test sets. While for the linear activation function the performance issues could be attributed to the fact that the model is unable to learn any non-linear patterns in the data, the same is not true for the sigmoid activation function, in which case the performance issues could be attributed to vanishing gradients, which are a common issue with this activation function, which is possibly confirmed

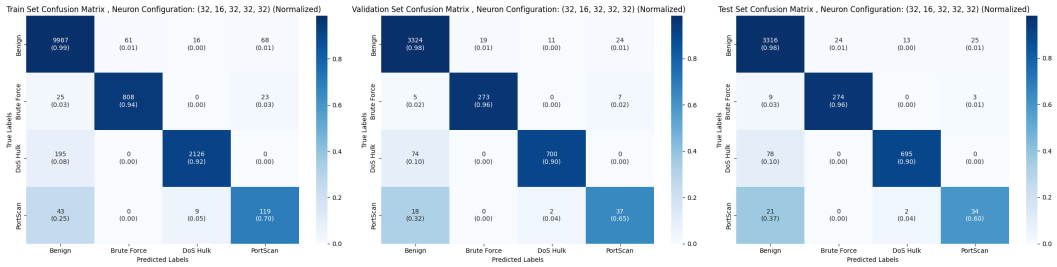


Figure 10: Confusion matrix heatmap for all data partitions (Best Configuration)

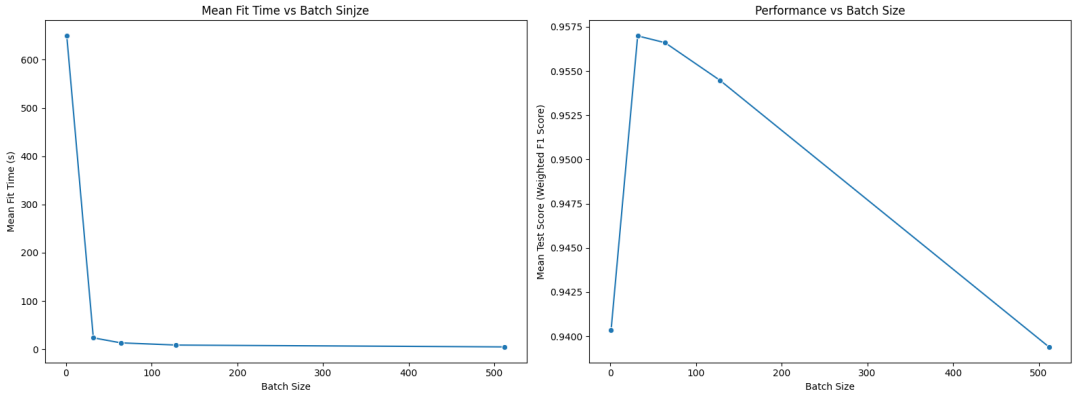


Figure 11: Batch Size vs Performance Comparison

by the fact that the loss curves for this activation function, after a few epochs, are basically flat, with little to no change in the loss values. As expected, the **ReLU activation function** consistently yielded the best performance of all, while also being the most accurate by a mile, with a weighted average F1 score of .97 in the validation set, with a little drop in the test set, and a macro average F1 score of .88 in the validation set and .85 in the test set (due to poorer performance on the `PortScan` class).

For the last part of the task, we had to **compare** the performance of the impact of **different optimizers**. The loss curves for the different optimizers are shown in figure 12.

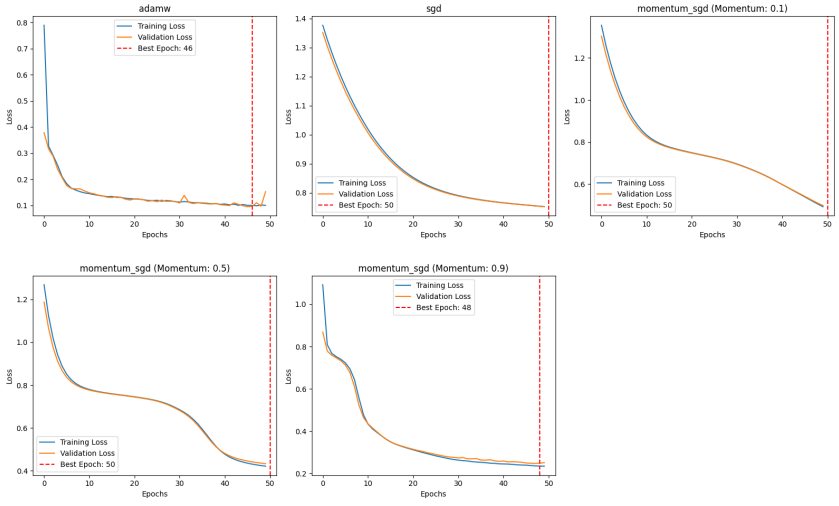


Figure 12: Training and Validation Loss Curves for Different Optimizers

The behavior for the curves varies a lot between the different optimizers. Stochastic Gradient Descent (SGD) showed a very *smooth* loss curve with the lowest delta between the loss at the end of the training and the loss at the beginning, which is expected since sgd relies only on local gradients, which makes it very slow to converge. Adding momentum helps to increase the convergence speed, and thus the performances, while also dampening the oscillations in the loss curve. The convergence speed, in this case, is higher the higher the momentum, with the highest momentum (0.9) being the best performing one out of the three. Only at the highest momentum it is possible that convergence is

reached, since the loss increases a little at the end of the training. In the end, the AdamW optimizer yielded the best performance of all by a significant amount. The combination of adaptive learning rate and weight decay allows the model to achieve better results in a much shorter time that sgd with momentum (about half the time, around 11s for AdamW), but a little longer than plain sgd (about 2 seconds).

After having successfully identified the best hyperparameters for our model, we had to test the effect of different **epochs** and **learning rates** on the model performance. The results of this comparison are shown in table 6. We tried out three different learning rates (0.0005, 0.001 and 0.01) and two different epochs (50, same as the previous experiments and 100). This should allow us to see if the epochs were too low and if the learning rate was not high enough to allow the model to converge. The loss curves shows (visible in the notebook) that the model is able to converge converge in all of the cases. This is most likely due to some initialization of the weights in the instances where the learning rate is lower, but, in the cases where the learning rate is higher, it is actually unable to reach convergence, since the loss keeps oscillating and the losses are higher, which is also reflected in the performance metrics. The worst performing models were in fact the ones with the highest learning rate (0.01), which suggests that the value was too high. About the epochs, even tough the lowest value for the losses was lower that the last epoch in the 50 epochs case, it was actually higher in the 100 epochs case, which highlights that the model is not able to actually converge in shorter epochs. In those cases the models shows the same trend as the one with lower epochs, with a learning rate of 0.01 being too high but showing that a **learning rate** of **0.001** (double the lowest one) is able to yield **better performances**, with good f1 scores in all the classes and the highest accuracy of all the models, with a value of 97.04% in the test set.

Table 6: Model Performance Comparison by Learning Rate and Epochs

		LR=0.0005, E=50			**LR=0.0005, E=100**			**LR=0.001, E=50**			**LR=0.001, E=100**			**LR=0.01, E=50**			**LR=0.01, E=100**		
Data Set	**Class**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**
Test Set	Benign	0.98	0.97	0.98	0.98	0.98	0.98	0.97	0.98	0.98	0.97	0.99	0.98	0.96	0.96	0.96	0.97	0.97	0.97
	Brute Force	0.92	0.95	0.94	0.92	0.96	0.94	0.92	0.96	0.94	0.93	0.95	0.94	0.79	0.95	0.87	0.86	0.95	0.90
	DoS Hulk	0.95	0.93	0.94	0.96	0.94	0.95	0.99	0.90	0.94	0.97	0.91	0.94	1.00	0.87	0.93	0.98	0.87	0.92
	PortScan	0.49	0.67	0.57	0.54	0.75	0.63	0.52	0.74	0.61	0.98	0.77	0.86	0.39	0.72	0.50	0.46	0.70	0.56
	Macro Avg	0.84	0.88	0.86	0.85	0.91	0.88	0.85	0.89	0.87	0.96	0.91	0.93	0.78	0.88	0.81	0.82	0.88	0.84
	Weighted Avg	0.96	0.96	0.96	0.97	0.97	0.97	0.97	0.96	0.96	0.97	0.97	0.97	0.95	0.94	0.94	0.96	0.95	0.95
	Accuracy		0.9624			0.9666			0.9635			0.9704			0.9417			0.9506	