Lab 1: Introduction to Deep Learning Using a Feed Forward Neural Network (FFNN)

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

1 Task 1: Data Preprocessing

Before delving into network architectures, we devoted a full phase to the sanitisation and scaling of the CIC-IDS 2017 subset used in this project. Here we summarise the key steps and findings.

1.1 Dataset overview

The original dataset contains **31 507** flows distributed across four classes:

Class	Instances	Share [%]
Benign	20 000	63.5
DoS Hulk	5 000	15.9
Port Scan	5 000	15.9
Brute Force SSH	1 507	4.7

The imbalance is moderate; we therefore kept the raw counts but monitored class-wise metrics in later stages.

1.2 Cleaning

1. **Infinite and missing values**. Two distinct pathologies were observed:
   - *Division-by-zero artefacts.* Twenty-seven flows had $\pm\infty$ in `Flow Bytes/s` and/or `Flow Packets/s` and all of them had `Flow Duration` $= 0$
   - *NaN.* We identified 20 rows containing NaN values, consequently replacing $\pm\infty \rightarrow$ NaN, all 47 rows were removed (0.15 % of the dataset).
2. **Negative placeholders**. Two rows contained -1 in time-based features and unphysical negative rates . These appear to be artefacts of an upstream capture script and were discarded.
3. **Duplicated flows**. The exact duplicates were pruned with `df.drop_duplicates`: 2 094 rows (6. 65 %) - mainly *DoS Hulk* - were removed, avoiding biasing both generative and discriminative training, and improving generalization and speeds up I/O.

Step	**Rows removed**	**Rows left**	**% cut**	**Main class affected**
Infinite/NaN	27	31 480	0.09	DoS Hulk (19)
Negative values	2	31 478	0.01	Benign (2)
Duplicates	2 094	29 384	6.65	DoS Hulk (1 125)
Total	2 123	29 384	6.74	–

Table 1: Cumulative effect of cleaning steps.

The final class counts are *Benign* = 19240, *Port Scan* = 4849, *DoS Hulk* = 3868, *Brute Force* = 1427.

1.3 Exploratory analysis

Label encoding Textual labels were mapped to integers via `LabelEncoder` (*Benign*→0, *Brute Force*→1, *DoS Hulk*→2, *Port Scan*→3). The port identifier is left unencoded at this stage; Sect. 3 shows that `Destination Port` behaves as a *dataset bias* and is later discarded.

Distributional shape

We performed a skewness analysis (Pearson's moment coefficient) that highlights strong right tails (here just a subset for space):

Feature	Skew	Family
Total Fwd Packets	152	counters
Flow Bytes/s	42	rates
Fwd Packet Length Max	12	lengths
Fwd IAT Std	8	timing

Binary flags such as `SYN Flag Count` and `Fwd PSH Flags` have skew ≈ 4.7 , and since a log-transform still leaves the flag with only two possible values, it does not reduce skewness or add useful granularity; therefore, the flags are kept in their native 0/1 scale.

Correlation structure

The plotted Correlation-Matrix reveals two perfect blocks ($|\rho| = 1.00$):

- `Fwd PSH Flags` ≡ `SYN Flag Count` (identical bit pattern in this subset). We keep `SYN Flag Count`—semantically richer for scan/flood detection—and drop its twin.
- `Subflow Fwd Packets` ≡ `Total Fwd Packets`. The clearer descriptor is retained.

All other pairs with $0.90 < |\rho| < 0.96$ are *kept* deliberately: deep networks cope well with moderate multicollinearity, and the nonlinear layers may capture interaction patterns that a manual purge would suppress. However, the choice contrasts with a shallow MLP baseline, where aggressive feature pruning is often a prerequisite for stable training, but it is done in order to compare the different results of the shallow and that of the deep networks.

Attack-oriented feature clues (raw space)

Figures in Cells `[ 124, 131, 32]` depict the *unscaled* data: a 4×4 scatter matrix, and classwise box/violin charts for the five most meaningful features. We observe :

- **DoS Hulk (label 2)**
  - Dominates the upper deciles of `Flow Duration`, `Packet Length Mean` and `Bwd Packet Length Mean` (see box- and violin-plots).
  - Points cluster at very *high* values of `Flow Bytes/s` and `Flow Packets/s` while the scatter grid shows almost vertical streaks (burst sending rate with little variation in packet size).
- **Port-Scan (label 3)**
  - Appears in the *long-duration / low-throughput* corner: the scatter cell `(Flow Duration, Flow Bytes/s)` contains a red cluster with duration $> 5{\times}10^7$ µs and byte-rate one order of magnitude below the DoS points.
- **Brute-Force (label 1)**
  - Median values are close to Benign for all five key metrics, but violin plots reveal a characteristic secondary mode in `Bwd Packet Length Mean` (e.g. server reply bursts during repeated authentications).
- **Benign (label 0)**

– Central bulk of every distribution, with very few points above the 95^{th} percentile except for `Flow Bytes/s`—heavy downloads create the long right tail visible in the box-plot.

– The scatter grid shows benign dots forming a dense nucleus; attack classes radiate outwards along one or two dimensions, indicating that a non-linear separator (e.g. an FFNN) will be required.

1.4 Dataset split

We partition the data into **training** (60 %), **validation** (20 %) and **test** (20 %) sets. A *stratified* sampling strategy is used at every step so that each subset preserves the original class priors.

1.5 Two normalisation pipelines

1. **Log (x+1) + RobustScaler** : adopted for the subsequest tasks.
2. **StandardScaler** : kept as a baseline for ablation.

As mentioned above, we adopted a two-step normalization process to mitigate the impact of outliers without discarding them. The rationale behind this choice stems from the nature of the dataset: outliers might carry **valuable semantic meaning** in the context of cyberattack detection. For instance, an unusually high `Flow Bytes/s` or `Flow Packets/s` value could be indicative of a **Denial-of-Service (DoS)** attack or the high value of `Bwd Packet Leangth Mean` useful to distinguish Brute-Force from Benign data. Simply removing or trimming these values would risk losing critical information relevant for accurate classification.

To preserve the potential significance of extreme values while reducing their negative impact on the learning process, we applied the following steps:

1. **Log Transformation**
   A natural logarithmic transformation (`log1p`, which computes $\log(x+1)$) was applied only to the features that, according to the skewness audit, showed the widest spread and the most pronounced extreme values:
   Total Fwd Packets, Fwd Packet Length Max, Fwd Packet Length Mean, Flow IAT Mean, Flow Bytes/s, Flow Packets/s, Fwd IAT Std, Bwd Packet Length Max, Bwd Packet Length Mean, Packet Length Mean, Flow Duration.
   This transformation reduces skewness and compresses the range of high values, making the data more suitable for downstream learning algorithms without eliminating outliers.

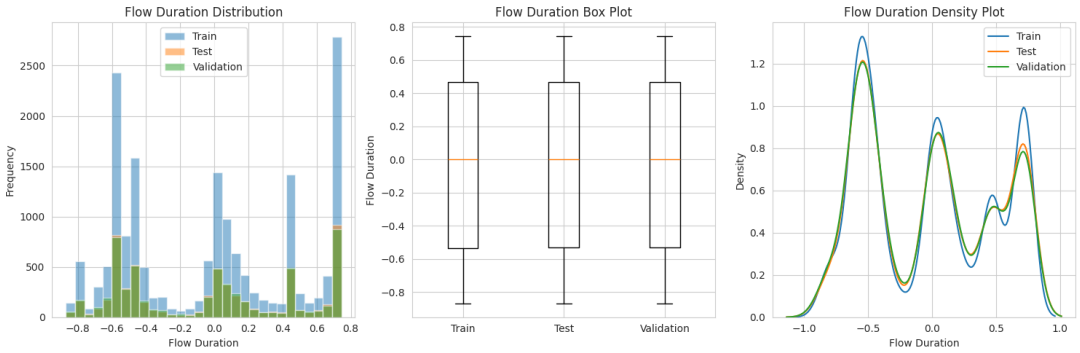
2. **Robust Scaling**
   Subsequently we normalised the data with `RobustScaler`. For each feature x the transformer computes
   $$x^\star = \frac{x - \text{median}(x)}{Q_3(x) - Q_1(x)},$$
   i.e. centring by the median and rescaling by the inter-quartile range $\text{IQR} = Q_3 - Q_1$. Because both statistics depend solely on the central 50 % of the order statistics, the resulting scale is virtually *insensitive* to extreme observations, outliers remain visible but no longer dominate the feature's dynamic range.

The pre-processing fit was performed **only on the training set** to avoid data leakage, and the learned transformations were consistently applied to the validation and test sets. Hence, the test flows never influence the parameters (median, IQR, log bias) learned from train, meeting the "identical pre-processing" requirement.

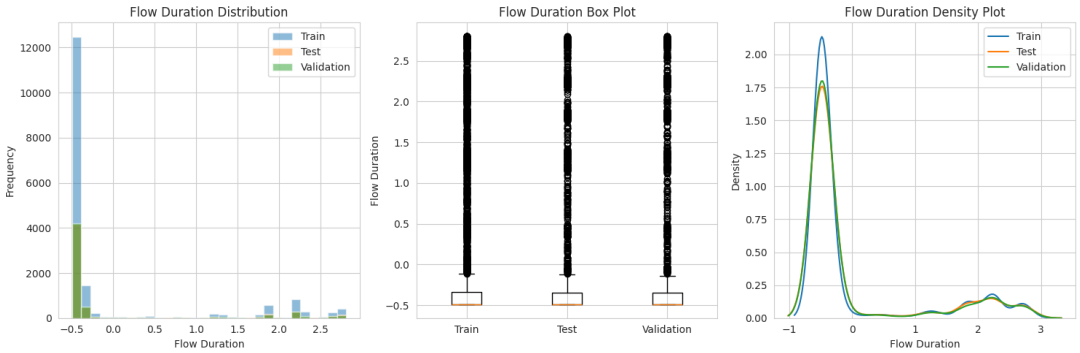
This strategy ensures that the model is trained on data that is more **uniformly scaled**, thereby improving learning stability and convergence, while also retaining the **discriminative power** of potential outlier values.

Demonstration of the effects produced by the two scaling strategies (Fig. 1).

- **Log (x+1) + RobustScaler** : The bulk of the data expands into three well-separated, quasi-Gaussian bumps, while extreme values remain visible without dominating the axis.



(a) Log (x+1) + RobustScaler



(b) StandardScaler (z-score)

Figure 1: Effect of the two scaling strategies on `Flow Duration`. Each panel displays, left to right, the stacked histogram (train–val–test), the IQR box-plot, and the KDE density.

- **StandardScaler**: Roughly 95 % of the samples collapse to a narrow spike at $z \approx -0.3$; genuine bursts are pushed several standard deviations away, yielding a highly unbalanced scale where a few outliers steer the gradients.

The very same pattern extends to *most* of the other features inspected, confirming that the choice of the 2-step pipeline has a systematic impact across the attribute space.

	% outliers (train / val)		KS–JS $_{\text{median}}$	
Feature group	Log + Robust	Standard	Log + Robust	Standard
Rates (Duration, Bytes/s, Pkts/s)	**0.3 / 0.3**	19.8 / 18.7	0.96 / 0.0004	0.75 / 0.037
Lengths (5 vars)	0.0 / 0.0	22.4 / 21.9	0.89 / 0.001	0.64 / 0.071
All 14 numeric	4.0 / 3.9	22.2 / 21.5	0.94 / 0.007	0.68 / 0.027

Table 2: IQR–based outlier prevalence and median two–sample metrics between **train** and **val** (p-value from Kolmogorov–Smirnov, JS divergence).

2. Outlier inspection and scaling choice

- **Raw data.** The initial IQR scan flagged 18–22 % of the training samples as extremes, with heavy concentration in rate-type variables and length maxima.
- **StandardScaler.** Z-score rescaling leaves the tail mass essentially unchanged (still ∼22 % of points are beyond 1.5 IQR) and lowers KS-p / raises JS, i.e. introduces residual train–val drift.
- **Log (x+1) + RobustScaler.** After the log compression and IQR-based scaling the global outlier rate collapses to **4 %** (train) and 3.9 % (val), yet class-specific burstiness is preserved: DoS-Hulk keeps 21 % extremes in `Flow Bytes/s`, while benign traffic remains at 6 %. All features now yield a KS $p > 0.20$ and JS< 0.05, indicating no detectable distributional shift across the dataset splits.

Decision. Log + Robust simultaneously neutralizes the majority of non-informative outliers, retainingng cyber-relevant extremes. Therefore, it is selected as the normalization pipeline for all subsequent modeling steps; StandardScaler is kept only as a comparison baseline. The standard scaler version of the label is in the *Lab_1_only_SS.ipynb*, while other tests are in the *Lab1_final_FFNN_Robust.ipynb*

2 Task 2: Shallow Neural Network

In this section, we focus exclusively on the best-performing model of the three single-layer variants, each configured with a different neuron count, the winner being the configuration with 32 neurons.

- The network is trained on the chosen compute device for *up to 100 epochs*, using early stopping (patience = 30), mini-batches of 64 samples, and the AdamW optimizer.
- **single-layer with 32 hidden neurons:** it achieves the highest F_1 on the most under-represented class (Class 1), keeps the overall accuracy at 96% with an almost identical macro-F_1 respect to the 128 model, and does so with a significantly lighter compute and memory footprint.

At every epoch:

1. **Training phase** – With the model in `train` mode, gradients are reset, the loss for each mini-batch is back-propagated, and the optimizer updates the parameters. The mean training loss over the full training set is recorded.
2. **Validation phase** – Switching to `eval` mode and disabling gradient tracking, the mean loss is computed on the validation set.

An *early-stopping* rule terminates training when the validation loss fails to improve for *30 consecutive epochs*, mitigating over-fitting and unnecessary computation. Epoch-level training and validation losses are logged and plotted on completion to illustrate convergence, and the wall-clock time of the entire training run is reported.

Figure 2 presents the epoch-wise loss for the training and validation splits. Both curves fall sharply during the first three epochs and then level off, indicating rapid convergence followed by incremental refinement. The validation loss closely follows—and in the final epoch marginally undercuts—the training loss (~ 0.187 vs. ~ 0.186), signalling good generalisation.

Sensitivity to normalisation. The outstanding scores are achieved only with our two–step normalisation pipeline (log transform + *RobustScaler*). Training the same network with a *StandardScaler* instead results in a sharp drop in performance: both the 64- and 128-unit variants perform significantly worse, and even the 32-unit configuration—while still the best among the Standard-scaled models—shows a much lower macro-F_1 and, more importantly, inferior metrics on the least-represented class. The discrepancy stems from employing a single linear activation: this proves sufficient in combination with log + robust normalisation, but is inadequate when paired with standard scaling, confirming that careful data preparation is essential for high performance.

The network parameters that achieve the *lowest validation loss* are retained for final evaluation:

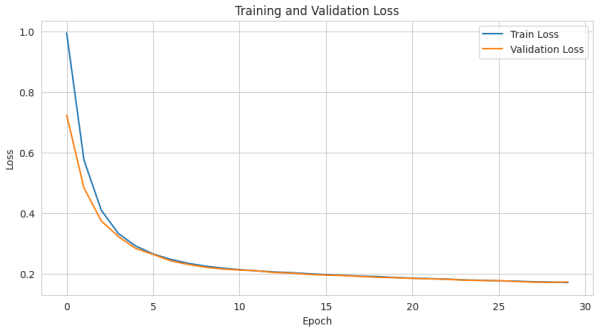


Figure 2: Training Curve 32 neurons

1. After every epoch, the mean loss on the validation set $\mathcal{L}_{\text{val}}^{(e)}$ is computed.

2. If $\mathcal{L}_{\text{val}}^{(e)} < \mathcal{L}_{\text{best}}$, the current weights $\theta^{(e)}$ are deep-copied and stored as θ^*; the early-stopping counter is reset.

3. Otherwise, the counter is incremented. Training stops when no improvement is observed for *patience* = 30 consecutive epochs.

At completion, θ^*—i.e. the weights from the epoch that minimised the validation loss—is reloaded and used for all test-set inference and subsequent deployment, ensuring that the reported performance corresponds to the best-generalising model.

- *Overall performance.* Both validation and test splits achieve an accuracy of 0.96 and a macro–averaged F_1 score of 0.92 - 0.93 , confirming that the model generalises well beyond the training data.

- *Class–specific behaviour.* Classes 0 and 3 exhibit near–perfect performance ($F_1 \geq 0.97$), whereas Classes 1 and 2 show comparatively lower recall, yielding $F_1 \approx 0.84$–0.89. The discrepancy is consistent across splits and reflects the class–imbalance (Class 1 represents only $\sim$ 5% of the samples).

Table 3: Per–class and overall metrics on the validation and test splits.

	Validation			**Test**		
Class	Precision	Recall	F_1	Precision	Recall	F_1
0	0.97	0.98	0.97	0.96	0.98	0.97
1	0.82	0.87	0.84	0.86	0.84	0.85
2	0.94	0.88	0.91	0.93	0.89	0.91
3	0.98	0.96	0.97	0.99	0.96	0.98
Accuracy		0.96			0.96	
Macro avg F_1		0.92			0.93	
Weighted avg F_1		0.96			0.96	

Impact of the activation function. Replacing the linear activation with ReLU boosts the best model's performance to an overall **98% accuracy** and an F_1 **score of 0.97** on the test set. When the same ReLU activation is applied in the configuration that uses a *StandardScaler*, the macro-F_1 likewise jumps from 0.86 (linear activation) to 0.91. Even so, the two-step *log + RobustScaler* pipeline still outperforms the Standard-scaled variant. It should be emphasised, however, that these exceptional results are partly driven by a systematic bias—the inclusion of the *Destination Port* feature, which serves as a shortcut for the model.

3 Task 3: The impact of specific features

Brute-Force attacks in the dataset originate from port 80.

- **Is this a reasonable assumption?** Yes. Port 80 is the default port for HTTP and thus one of the most widely exposed services on the Internet in order to guess credentials and access the service exposed by port 80. Because web servers are attractive targets, many brute-force attacks concentrate on contacting this port; however, brute forcing the service on port 80 is not the only possible attack. We may also try to brute force SSH in order to gain a shell on the end user, or all the ports related to servers like MySQL and so on, and if we don't have similar network traffic that tries to attack those services, we will not be able to spot them.

- Replace port 80 with port 8080 for the Brute-Force attacks in the *test* set and run inference with the previously trained model:

  - **Does the performance change? How does it change, and why?**

    * **Log + RobustScaler.** Performance on *Class 1* collapses entirely: on the test set the model fails to identify a single instance, resulting in an F_1 score of 0. Meaning that with the 2-step pipeline the model heavily relied on the shortcut.
    * **StandardScaler.** The outcome is only marginally better: the F_1 for *Class 1* drops to just 0.09. Evidently the network has over-fit to the *destination-port shortcut* (it was trained exclusively on traffic with port 80); when evaluated on data using a different port, it can no longer recognise the pattern.

After removing the destination port and repeating all the preprocessing steps

- How many *PortScan* samples remain after preprocessing, and how many were present beforehand?
  - Before preprocessing there were 5000 samples of PortScan, after the pre-prepocessing 4849 and then after the drop Duplicates we only get 285 port scan (both distribuited in train validation and test)
- Why is *PortScan* the class most affected after duplicate removal?
  - During a port scan the attacker issues a large number of almost identical probes, throttling their rate just enough to avoid triggering anomaly-detection thresholds. Across these probes the sole field that consistently varies is the destination port, because each request targets a different service hosted on a different port. Consequently, if we drop the destination-port feature from the dataset we eliminate the only attribute that distinguishes one probe from the next, rendering all port-scan traffic virtually indistinguishable and hampering any attempt at fine-grained analysis or classification.
- Are the classes now balanced?
  - No the classes are now even more unbalanced then before, we have class that now only rappresents 1% of our dataset (PortScan), and this might mean that the model will have an hard time at detecting it
- Repeat the training with the best architecture identified in the previous step:
  - How does the performance change?
    * **Robust + Log**: Performance have a slight drop and as expected especially for Class 3, but oveall the model is still good.
    * **StandardScaler.** The model now struggles to detect *Class 3*, attaining a recall of only 0.07 because this class has become the scarcest in the dataset; performance on the remaining classes is largely unchanged. The resulting macro-F_1 plummets by roughly 20 percentage points compared with the earlier, biased-dataset setting.
  - Can the model still correctly classify the rarest class?
    * **Robust + log scaler**: yes we are still able to detect the least present class (port scan), and checking the test set performance we are able to reach an 88% F1 score with a 92% accuracy over it.
    * **StandardScaler.** The model fails to recognise any *Class 3* instances. A plausible explanation is the scarcity of samples for this class; moreover, z-score standardisation disperses their feature values instead of emphasising them, making the underlying patterns harder for the network to learn.
- Use weight loss to "improve" the performance of the model
  - How does the performance change per class and overall? In particular, how does the accuracy change? How does the f1 score change?
    * **Log scaler + Robust**: If we add class weight the thing that we noticed is that the accuracy over the 2 least present classes (PortScan and BruteForce), this is caused by the fact that the model is rewarded if it predicts one of these 2 classes, and so we have a higher number of FP on this, with an increase also on the recall. In general this makes the Macro F1 score go from 0.92 to 0.81, and we also have a huge decrease in the F1 score of PortScan (from 0.88 to 0.51).

* **Standard Scaler** : we see a general increase in the performance over class 1 (that now has a 31% F1 score compared to the 12 %). This is caused by the fact that adding weights to the classes we are trying to increase the recall of class that are less present in the dataset.
  The Standard-scaled network still captures most *Class 3* instances ($49/57 \approx 86\%$ recall), yet it generates markedly more false positives: 208 samples from other classes are misclassified as Class 3 (179 from Class 0, 14 from Class 1, and 15 from Class 2), compared with only 102 in the Log + Robust model. In other words, Standard scaling trades precision for recall, whereas the Log + Robust pipeline achieves a better balance by keeping both false positives and false negatives low (even though applying class weights slightly decreased performance compared with the unweighted setting).

4 Task 4: Deep Neural Network

4.1 First Deep Neural Network

In order to find the best architecture, we created a simple parametrized model that let us decide how many layers and the size of each layer, and after comparing different results, the best architecture came up to be a 4–layer FFNN with hidden sizes [32, 32, 32, 32]. It preserves the highest overall accuracy ($\approx 97\%$) and macro-F_1 (0.93),while achieving the highest F_1 in the rarest class (0.89 for Class 3), offering the best trade-off between global performance and sensitivity of minority classes (model 051 in the notebook).

– **Analyze the losses**

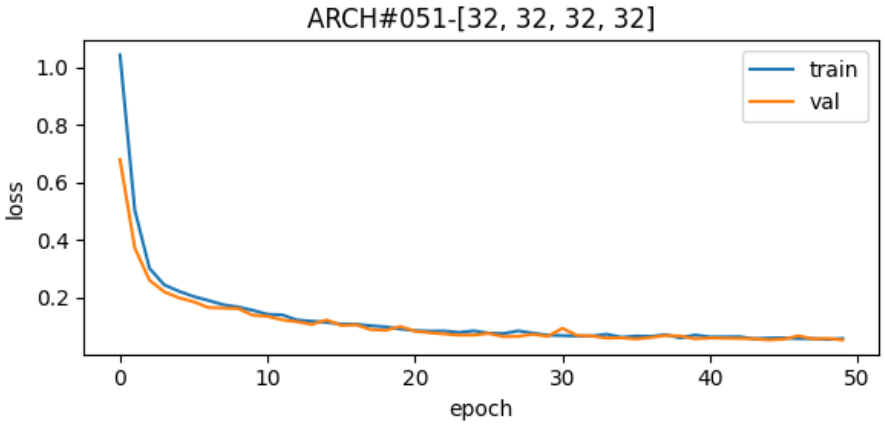


Figure 3: Loss for the Model with 4 layer with hidden sizes ([32,32,32,32]

* from the learning curve (Figure 3) we notice that the early stop is triggered around the 48 epoch in order to avoid a plateaou region. In general looking at the curve there are no particular sign of over fitting, noticing also that the validation and training curve decreases approximately at the same rate

– **Evaluate the performances on test and validation set**

* Performance over this model are reported in Table 4. In general the important things to notice are related to the least- present classes, that both achieves high precision, with respect to the element that have be used in training. Overall we are able to achieve Macro average F1 score of 93%, with still room to improve.

4.2 The impact of batch size

After finding the best architecture, we tested the different batch sizes during training 1, 32, 64, 128, 512

Table 4: Classification Report on Validation and Test Sets

Class	Validation Set				Test Set			
	Precision	Recall	F1-Score	Support	Precision	Recall	F1-Score	Support
Class 0	1.00	0.96	0.98	2702	1.00	0.96	0.98	3378
Class 1	0.84	0.99	0.91	228	0.83	0.99	0.90	285
Class 2	0.91	0.99	0.95	619	0.93	0.99	0.96	774
Class 3	0.81	1.00	0.89	46	0.83	0.95	0.89	57
Accuracy		**0.97**		3595		**0.97**		4494
Macro Avg	0.89	0.99	0.93	3595	0.90	0.97	0.93	4494
Weighted Avg	0.97	0.97	0.97	3595	0.97	0.97	0.97	4494

- **Does the performance change? And why? Report both the validation and test results**
  More detailed results can be found in the notebook. With a *single-sample batch* (B=1) the network attains the best headline numbers —macro-F_1 = 0.96 and 98 % accuracy—yet the learning curve is jagged and the wall-clock time balloons to roughly 28minutes (1 717s to reach early stopping). Increasing the batch to **32** smooths the loss trajectory, shortens the run to around one minute (58s), and still closely aligns the top accuracy (macro-$\boldsymbol{F_1}$ **= 0.93**, 97 %). At $\boldsymbol{B}$**=64** the performance are quite similar. Beyond this point the benefits vanish: very large batches ($\boldsymbol{B}\geq$**128**) push the macro-$\boldsymbol{F_1}$ below 0.89 despite even faster runtimes. The pattern suggests that larger batches, although computationally efficient, reduce gradient stochasticity and curtail the model's exposure to diverse micro-patterns, ultimately harming generalization. Batch size 32 therefore represents the best overall compromise.
- **How long does it take to train the models depending on the batch size? And why?**
  The smaller the batch size, the longer it takes to train the model. In fact, the slowest training occurred with a batch size of 1, which took around 1717 seconds to complete. This happens because with such a small batch, the model needs to perform many more gradient computations and weight updates. As a result, the convergence towards the optimal solution is slower and less efficient.

4.3 The impact of the activation function

We tried to see what the differences were in using 3 different Activation functions, *Relu*, *Sigomid* and *Linear*

- **Does the performance change?** *Yes—dramatically.* Switching the hidden–layer activation from `linear` to `ReLU` boosts the macro-F_1 on the test set from **0.74** to **0.93** (accuracy rises from 89 % to 97 %); adopting `sigmoid` instead achieve **0.85** of F_1. Validation loss mirrors this trend: $\text{val\_loss}_{\text{linear}} \approx 0.270$, $\text{val\_loss}_{\text{sigmoid}} \approx 0.247$, while $\text{val\_loss}_{\text{ReLU}}$ drops to 0.047.
- **Why does it change?** `Linear` activations leave the network equivalent to a single logistic layer, limiting capacity; `sigmoid` suggests that the activations remain within a range that does not cause significant saturation, yet still results in poor performance on the minority class `ReLU` avoids saturation, preserves gradient flow, and enables the model to capture non-linear patterns—hence the sharp improvement.

4.4 The impact of the optimizer

AdamW still offers the best trade-off: in 63.5s it achieves the lowest and steadiest validation loss (0.0489), the highest test accuracy (97.51%) and macro-F_1 (0.95).

SGD with momentum 0.1 is a close second—57.7 s, val-loss 0.0755, accuracy 96.84% and macro-F_1 0.93—while increasing momentum to 0.5 (58.4 s, val-loss 0.0638) tightens the loss but yields no accuracy gain (96.66%, macro-F_1 0.92). Plain SGD without momentum converges more slowly with occasional peaks (56.3 s, val-loss 0.0724) and scores 96.48% accuracy with macro-F_1 0.92.

Too much momentum ($m = 0.9$) causes oscillations (58.0 s, val-loss 0.0660) and still produces just 96.86% accuracy with macro-F_1 0.93, without beating AdamW.

So the recommended configuration is a five–layer FFNN with hidden sizes [32, 32, 32, 32], trained with a batch size of 32, ReLU activations, and the AdamW optimizer.

When we swap in a *StandardScaler* instead of our two–step *log + RobustScaler*, the only change in the final network is its architecture, which collapses to three layers of 32 neurons each ($[32, 32, 32]$), while the optimizer (AdamW), activation (ReLU) and batch size (32) remain identical. Despite this minimal adjustment, the Standard-scaled model trails by more than 14 percentage points in macro-F_1 compared to the log + Robust variant. The root cause is its amplified sensitivity to class imbalance: it over-priorities recall for minority classes at the cost of overall precision, yielding lower balanced performance.

5 Task 5: Overfit and regularization

This final task involved creating a model with specific characteristics, including a 6-layer architecture ($256 \to 128 \to 64 \to 32 \to 16$) and analyzing the loss curve. Despite the relatively deep architecture, training loss drops precipitously—from about 1.0 to below 0.05 within the first five epochs and then flattens out; the validation loss tracks it almost perfectly, never deviating by more than 10^{-3} and showing no upward drift (see Figure 4). Because the two curves remain virtually superimposed throughout and the final macro-F_1 on the test set is still 0.94, **there is no evidence of overfitting**; the model generalizes well despite the absence of explicit regularisers. This behavior may be due to the preprocessing step, where features with extreme correlation values were dropped. Additionally, the way the dataset was split into training, validation, and test sets could have influenced the model's performance. Finally, differences in hardware might also lead to variations in the results.

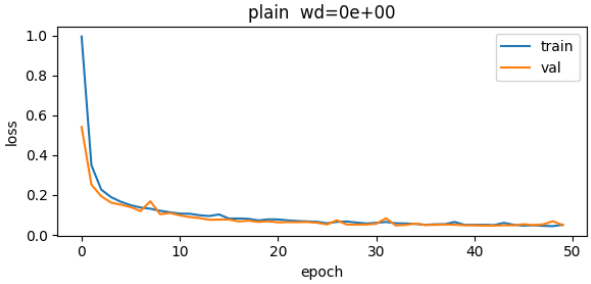


Figure 4: Learning curve of model for Task 5

However, various normalization techniques were applied, such as dropout and batch normalization, along with weight regularization using AdamW's weight decay. We have observed that:

Impact of Normalization and Weight Decay. Examining the loss curves produced, we observe that inserting *Batch Normalization* after each hidden layer yields the smoothest, fastest decline in validation loss—reaching a nadir of 0.0436 at weight decay 10^{-3}—and a peak test accuracy of 97.30%. Despite this, its macro-F_1 (0.93) still falls short of our simplest baseline.

In contrast, adding *Dropout* anywhere in the network consistently raises the best validation loss (to around 0.0549) and trims macro-F_1 by 1–2 percentage points. The loss plots show noticeably larger oscillations when Dropout is used, suggesting that with large training volumes and class weighting, Dropout chiefly injects noise rather than curbing over-fitting.

Combining both techniques (*BatchNorm + Dropout*) only compounds these drawbacks: training slows by roughly 25 %, the loss curves flatten later, and the best macro-F_1 never exceeds 0.92.

Finally, exploring *AdamW*'s weight decay confirms that on the plain six-layer, no decay ($wd = 0$) delivers the lowest validation loss (0.0467) and highest test accuracy (97.37 %). Under BatchNorm a mild decay (10^{-3}) marginally smooths the loss descent but fails to improve F1.

We can conclude that Batch-Norm can smooth optimization and match the baseline, but the simplest configuration—*plain, no weight decay, no dropout*—remains the champion, balancing speed, simplicity and top generalization (97 % acc., macro-$F_1 = 0.94$), since the model with the 2-step pipeline normalization (log + RobustScaler) already generalize well.