Project 1: Introduction to Deep Learning

Giovanni Bloise , Alessandro Mulassano , Manuel Firrera

In this project we are going to analyze the CICIDS2017 dataset, developed by the Canadian Institute for Cybersecurity (CIC) at the University of New Brunswick. It is used for evaluating intrusion detection systems (IDS) and intrusion prevention systems (IPS). Data are collected in five days, from 9 a.m. on Monday, July 3, 2017, to 5 p.m. on Friday, July 7, 2017. Each day focused on specific traffic patterns: on Monday it's only collected benign traffic, while for the other days it's a mixture between benign traffic and various scenarios. In the dataset 4 different attack vectors are present: benign, portscan, dos hulk and brute force.

1 Task 1: Data Preprocessing

The goal of the first task was to preprocess the dataset in order to then feed clean and consistent data to the neural network pipeline.

We started by addressing the quality of the raw dataset. Missing values and duplicate entries were removed at the beginning. Then, to ensure consistency, we checked for outliers. In particular, we found that some features, namely *Flow bytes/s* and *Flow packets/s*, contained infinite values. Initially, we considered replacing them with the largest finite value in the respective feature. However, we noticed that this would negatively affect later stages, especially feature standardization, as such extreme values would distort the mean. For this reason, we decided to remove all records with infinite values. In total, 2121 samples were discarded—approximately 6.7% of the dataset—which we considered an acceptable reduction.

Next, we applied label encoding to convert categorical labels into numerical values. Figure 1 shows the resulting class distribution: 19,242 benign samples, 4,849 portscan, 3,868 DoS Hulk, and only 1,427 brute-force entries. The brute-force class is therefore significantly underrepresented compared to the others.

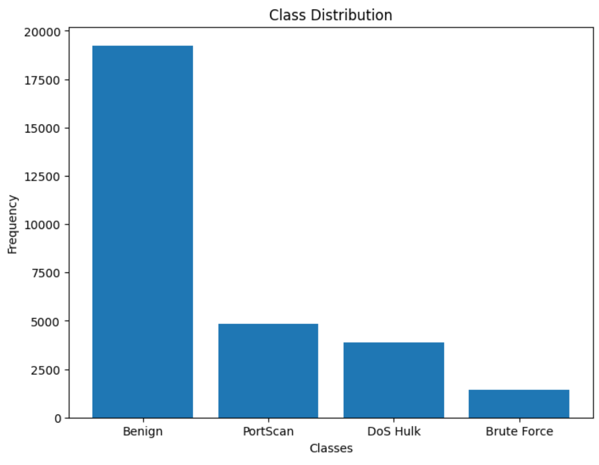


Figure 1: Class distribution of the dataset

The dataset was then split into three partitions: 60% for training, 20% for validation, and 20% for testing.

As the final step, we normalized the features. Due to the high variability in feature magnitudes, we chose *Standard Scaling* over Min-Max scaling. This choice was motivated by cases such as the *Total Fwd Packets* feature, which has a maximum value of 18,605 while its 75th percentile is only 5. In such cases, Min-Max scaling would compress the majority of values close to zero, reducing effectiveness. Standardization, instead, centers features around zero with unit variance, preserving relative distributions.

Q:Preprocessing of the test set. How do you preprocess the data? Is the preprocessing the same as for the training partition? It is crucial that the test set undergoes the *same preprocessing steps* as the training and validation sets to ensure consistency and comparability. This includes duplicate and missing value removal, filtering out infinite values, label encoding, and feature scaling. However, the key point is that the **Standard Scaler is fitted only on the training set**. The learned mean and variance are then applied to transform the validation and test sets. This prevents any data leakage from the validation or test partitions into the training process and ensures a fair evaluation of the model.

2 Task 2: Shallow Neural Network

The purpose of this second task was to design a shallow neural network with a single hidden layer and one output layer. We tested three different configurations of hidden layer sizes: 32, 64, and 128 neurons, all with a linear activation function.

To carry out the training process we leveraged the Colab GPU runtime to speed up computation. Data was converted into tensors and loaded on the T4 GPU VRAM in batches. Each model was trained and evaluated on the validation and test sets.

How does the loss curve evolve during training on the training and validation set? For every configuration, the training and validation loss curves followed a similar trend. There was a steep descent in the first 10–15 epochs, followed by a much slower decrease. Interestingly, the training loss consistently remained slightly higher than the validation loss. For all three models, both training and validation losses settled between 0.3 and 0.4, which is suboptimal.

This suggests that the models are *underfitting.* The elbow point of the curves indicates that more epochs would not significantly improve the fit. Early stopping is not required in this case. Moreover, the slow descent after the elbow point suggests that a linear model is not an adequate approximation for this dataset.

How do you select the best model across epochs? Which model do you use for validation and test? The optimal epoch is chosen where both the training and validation losses are at their lowest and the gap between them is minimized, aiming for a balance between underfitting and overfitting. For validation and testing, the same trained model is used, but with weight updates disabled. In this phase, the goal is purely evaluation, not further training.

What is the overall classification performance in the validation and test datasets and considering the different classes? To evaluate performance, we used the *Macro F1-score* metric, since it accounts for the class imbalance described in Task 1.

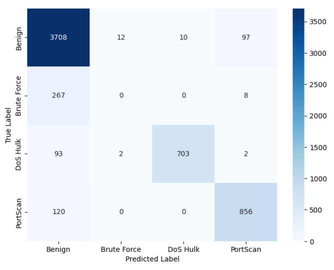
Table 1: Macro F1-scores for different hidden layer sizes (linear activation).

Hidden Neurons	Validation F1 (%)	Test F1 (%)
32	67.99	68.14
64	67.84	68.20
128	**68.59**	**68.44**

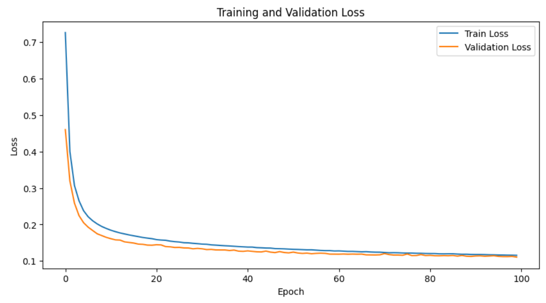
We also generated confusion matrices for each model to better visualize performance. For example, Figure 2a shows the confusion matrix for the 128-neuron model on the test set. It is clear that class imbalance heavily impacts performance: the *benign* class is predicted relatively well due to its abundance, while the *brute force* class is almost never predicted correctly due to its scarcity.

Why is the performance of the model so poor? As mentioned, a linear model is not well suited for our dataset. The F1-scores of all configurations remain below 70%, which confirms that the model fails to capture the complex decision boundaries required. Among the tested models, we selected the 128-neuron configuration since it provided the best (though still limited) performance.

Introducing non-linearity. To overcome the limitations of the linear model, we modified the hidden layer activation function from *linear* to *ReLU*. This simple change drastically improved performance. The new model achieved a test F1-score of **93.71%**, which is very promising and suggests that we are indeed facing a non-linear classification problem.



(a) Confusion matrix of the 128-neuron linear model on the test set.



(b) Loss plot for 128-neuron non-linear model.

The loss plots(Figure 2b) for training and validation are nearly overlapping, indicating a well-fitted model without noticeable underfitting or overfitting. Moreover, the new confusion matrix shows a significant improvement in the detection of the *brute force* class, which was previously ignored by the linear models.

3 Task 3: The Impact of Specific Features

As discussed in the lecture, biases in data collection can carry over to the model and become inductive biases. For instance, in our dataset all Brute Force attacks originate from port 80.

Is this a reasonable assumption? Yes, but it is problematic. The dataset contains four different types of attacks. For the Brute Force attack, the destination port is always 80. This creates a bias factor: the neural network inevitably assigns disproportionate weight to the `Destination Port` feature when classifying Brute Force attacks. This means that the model is not truly learning the general characteristics of the attack, but rather exploiting a shortcut. If future Brute Force traffic were to use a different port, there would be a high risk of misclassification.

Does the performance change? How does it change? Why? To verify this, we swapped the destination port value from 80 to 8080 in the test set and fed it to our pre-trained non-linear model. Performance dropped sharply, with the test F1-score decreasing to **72.86%**. The confusion matrix confirmed our expectation: the model was heavily biased toward recognizing Brute Force traffic only when it occurred on port 80, and failed to generalize to different ports.

We therefore removed the `Destination Port` feature from the dataset. This step affects all subsequent tasks in this laboratory activity.

How many PortScan samples do you now have after preprocessing? How many did you have before? After repeating all preprocessing steps, we noticed a dramatic reduction in the number of PortScan samples due to duplicate removal. Out of 5000 PortScan entries, **4715 were removed**, leaving PortScan as the new rarest class with 285 samples left.

Why do you think PortScan is the most affected class after dropping duplicates? Port scanning consists of probing multiple ports on a target host. The only relevant feature that changes between scans is the `Destination Port`, while many other features remain identical. Since we removed `Destination Port`, most PortScan samples became duplicates and were consequently dropped, leaving very few unique examples.

Are the classes now balanced? No. Even after preprocessing, the class imbalance remains. As shown in Figure 3, the *Benign* class is still heavily predominant compared to the other three classes, with *PortScan* now being the rarest.

How does the performance change? Can you still classify the rarest class? We retrained the model after removing the `Destination Port` feature. The loss plot shows no significant overfitting, as the validation loss is only slightly higher than the training loss. Validation and training accuracies are also similar.

From the confusion matrix (Figure 4), we observe that performance improved for the Brute Force class, which is no longer tied to port 80. However, the *PortScan* class now suffers significantly, both because of the feature removal and the reduced number of samples. The overall performance on the test set improved, as the F1-score is now **82.46%**. The other evaluation metrics are shown in table 2

To address the imbalance, we retrained the model using a weighted cross-entropy loss, where minority classes were assigned higher weights.

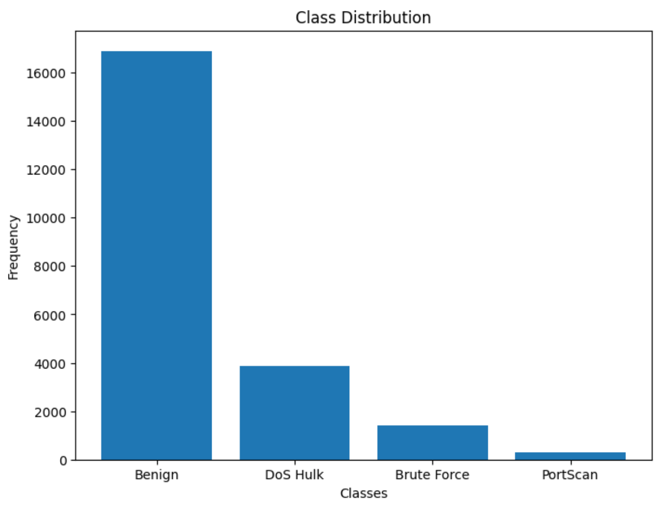


Figure 3: Class distribution after removing `Destination Port` and duplicates.

Table 2: Model performance after removing `Destination Port`.

Metric	Value
Train Loss (epoch 100/100)	0.1051
Validation Loss	0.1300
F1-score (Test)	82.46 %
Recall (Test)	84.12 %
Accuracy (Test)	95.24 %
Precision (Test)	81.19 %

How does the performance change per class and overall? In particular, how does the accuracy change? How does the f1 score change? With weighted loss, we observed a drop in global F1-score and accuracy(see Table 3). However, the per-class analysis (from the confusion matrix) showed that recall for the rarest class (*PortScan*) slightly improved, while precision on the majority class (*Benign*) decreased. This is expected:

- Weighted loss increases the importance of minority classes.
- The model sacrifices accuracy on majority classes to improve recall for rare ones.
- As a result, overall performance metrics (accuracy, F1) appear worse, even if minority detection is improved.

Moreover, the loss and accuracy curves indicated signs of overfitting, likely due to the limited number of samples in the rarest class. In conclusion, the model became slightly better at predicting *PortScan* attacks with weighted loss, but at the cost of significantly reduced overall performance.

4 Task 4: Deep Neural Network

For this task, we generated 30 random combinations of layers and a random number of neurons per layer, then trained them for a limited number of epochs to save compute time. From these, we selected the 3 models with the best F1-score on the validation set and trained them thoroughly for 50 epochs, evaluating their performance on the test set. The final model was chosen based on the best F1-score on the validation set. The selected architecture was a **2-hidden-layer deep neural network with 32 neurons per layer**.

4.1 Loss Analysis

At around epoch 40, the training and validation losses started to diverge, which is a strong indication of overfitting:

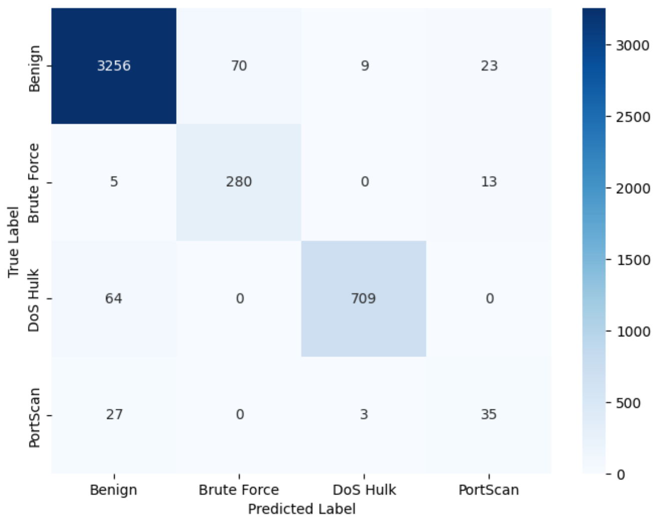


Figure 4: Confusion matrix after removing `Destination Port`.

Table 3: Performance comparison with and without weighted loss.

Metric	No Weights	With Weights
F1-score (Test)	82.46 %	78.00 %
Accuracy (Test)	95.24 %	92.00 %

- Epoch 40/50: Train Loss = 0.1928, Val Loss = 0.2967
- Epoch 20/50: Train Loss = 0.2295, Val Loss = 0.2957

Plotting the training and validation losses (see Figure 5) confirmed this observation: while the training loss continued to decrease, the validation loss plateaued and began to rise, a classical sign that the model was starting to memorize training data rather than generalize. To mitigate this, we applied early stopping, selecting epoch 15 as the best stopping point, since both training and validation losses were still close at that stage. The validation and test results are summarized in Table 4.

Table 4: Validation and Test Performance

Metric	Validation	Test
F1-score	77.4049	79.3463
Recall	92.4097	91.1169
Accuracy	90.9212	91.3663
Precision	70.8381	73.2299

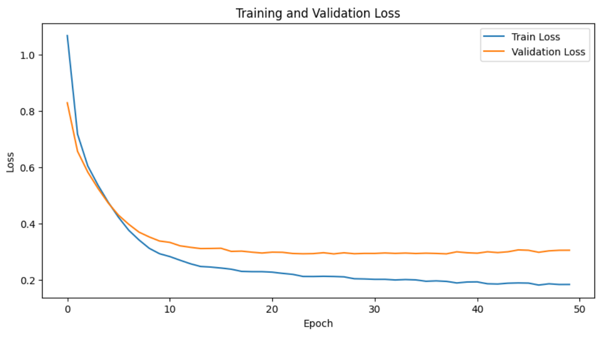


Figure 5: Train and Validation Loss for the best Deep FFNN.

4.2 Impact of Batch Size

Performance change: Table 5 shows validation and test F1-scores for different batch sizes.

Table 5: F1-score for Different Batch Sizes

Batch Size	Validation F1	Test F1
1	80.9851	81.1147
32	78.3417	80.3138
64	73.3021	75.6354
128	72.7672	74.6601
512	72.3576	74.1800

Q: Does the performance change? And why?

- **Small batch sizes (e.g., 1):** With very small batches, each gradient update is based on extremely limited information (just one sample in the case of batch size 1). This makes gradient estimates very noisy, which injects a large amount of stochasticity into the training process. While this instability usually makes optimization harder, it can sometimes act as a form of implicit regularization by helping the optimizer escape sharp minima. This explains why in some cases (like ours) small batches may achieve slightly higher F1-scores, although the results are unstable and vary significantly across runs. In practice, such improvements are not consistent and come with a high computational cost.
- **Medium batch sizes (32–64):** Medium-sized batches strike the right balance between noisy updates and stability. The gradients are averaged over enough samples to smooth out extreme fluctuations, but they still retain some stochasticity, which is beneficial for generalization. This explains why batch size 32 often gives reliable and robust F1-scores in small datasets, making it a good compromise between training stability, generalization ability, and computational efficiency.
- **Large batch sizes (128–512):** When the batch size becomes large, gradients are averaged over many samples, leading to very smooth gradient estimates. While this improves optimization stability, it reduces the natural regularizing effect of stochastic noise in SGD, which can make the model overfit to the training data. On small datasets, this effect is amplified: the model essentially "sees" most of the dataset at every update, reducing the variability of training signals. Consequently, large batches tend to converge faster in terms of loss but result in lower validation and test performance.

The results align with the aforementioned hyposeses.
Q: How long does it take to train the models depending on the batch size? And why?

Table 6: Training Time for Different Batch Sizes (seconds)

Batch Size	**Training Time (s)**
1	1206.86
32	44.76
64	26.35
128	15.62
512	13.72

Training time decreases significantly with larger batch sizes. This happens because larger batches reduce the number of parameter updates per epoch, while also allowing better parallelism on modern hardware such as GPUs. Very small batches (e.g., 1) require a separate update for every single sample, leading to an enormous number of updates and consequently very long wall-clock times. Although each update is cheap, the overhead accumulates. On the other hand, extremely noisy gradients from small batches can also slow down effective convergence, as the optimizer "jumps around" more before settling into a good minimum. Based on this trade-off, batch size 32 represents the best compromise: it trains in a reasonable time, provides stable gradients, and generalizes better than both very small and very large batches.

4.3 Impact of Activation Functions

Different activation functions were tested:

- Only Sigmoid
- Sigmoid + ReLU
- Only Linear
- ReLU + Sigmoid at the output

Table 7: Performance with Different Activation Functions

Activation	Validation F1	Test F1
Sigmoid only	71.3386	73.4595
Sigmoid + ReLU	73.5588	74.8594
Linear only	62.5187	64.8231
ReLU + Sigmoid output	77.9809	79.7511

Q: Does the performance change? Why does it change?

From Table 7, we observe clear differences in performance across activation functions. As expected, the linear-only model performed the worst, while the ReLU + Sigmoid combination performed the best. This can be explained by the following reasons:

- **Linear activations:** Stacking purely linear layers is equivalent to a single linear transformation. Such a network is incapable of capturing nonlinear decision boundaries, which are required in real-world classification tasks. This fundamental limitation explains the poor performance.
- **Sigmoid activations:** Introducing Sigmoid functions adds nonlinearity, making the network capable of modeling more complex relationships. However, Sigmoid suffers from the vanishing gradient problem: for very large or very small input values, the gradients become extremely small, slowing down learning in deeper layers. This makes training inefficient and limits performance.
- **ReLU in hidden layers:** ReLU (Rectified Linear Unit) overcomes the vanishing gradient problem for positive activations. It allows sparse representations, speeds up training, and enables hidden layers to learn more expressive features. This is why adding ReLU activations significantly improves performance over Sigmoid alone.
- **Sigmoid at the output layer:** Using a Sigmoid at the output is appropriate for probabilistic binary or multi-label classification. It ensures the outputs lie in the range $[0, 1]$, which can be directly interpreted as probabilities.
- **ReLU + Sigmoid combination:** This combination leverages the strengths of both functions: ReLU in the hidden layers facilitates efficient training and expressive feature learning, while Sigmoid at the output provides calibrated probability scores. This explains why the ReLU + Sigmoid configuration achieves the highest F1-scores.

4.4 Impact of Optimizers

Table 8: Performance and Training Time with Different Optimizers

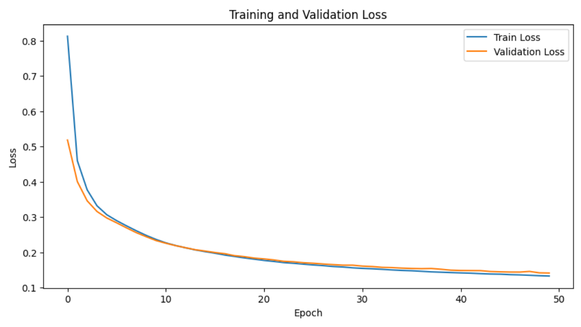
Optimizer	Validation F1	Test F1	Training Time (s)
SGD	21.5893	21.3831	22.62
SGD + Momentum 0.1	21.5893	21.3831	21.69
SGD + Momentum 0.5	45.2641	45.1669	21.65
SGD + Momentum 0.9	46.3804	46.4231	21.28
AdamW	78.3042	78.2730	23.20

The validation and test results show how different optimizers affect classification performance. Plain SGD was by far the worst, while AdamW was the best by a wide margin.

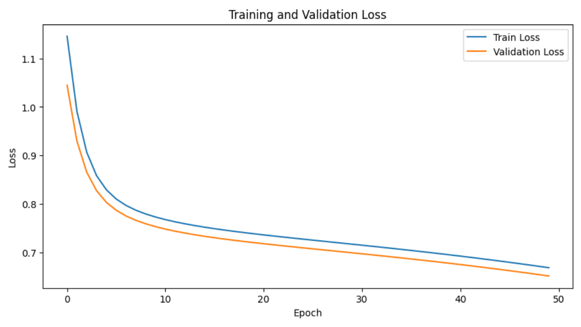
Q: Is there a difference in the trend of the loss functions? Yes. The differences can be explained as follows:

- **Plain SGD:** Requires carefully tuned learning rates. If the learning rate is too high, training diverges; if too low, convergence is extremely slow. In practice, this results in unstable or flat loss curves, reflecting poor optimization ability.
- **SGD + Momentum:** Momentum accumulates past gradients to smooth out updates. For imbalanced datasets, this helps amplify gradient contributions from minority classes that might otherwise vanish. Low momentum values (0.1, 0.5) have little effect, but high momentum (0.9) preserves useful gradient information across updates, leading to faster convergence and better performance. Loss curves become smoother and steeper compared to plain SGD.
- **AdamW:** AdamW adapts learning rates per parameter by tracking both the first (mean) and second (variance) moments of the gradients. Additionally, it decouples weight decay from gradient updates, which improves generalization. This makes AdamW robust to noisy gradients and insensitive to poor learning rate choices. The loss curve shows rapid and stable convergence, reaching significantly lower values than SGD-based methods.

Plotting the losses (Figure 6) for the best optimizer (AdamW) and the worst (SGD) highlights these differences: AdamW quickly converges to a loss lower than 0.2, while SGD lingers at much higher loss values, showing almost no progress.



AdamW Loss Curve



SGD Loss Curve

Figure 6: Comparison of training and validation loss curves for AdamW (left) and SGD (right).

Q: How long does it take to train the models with different optimizers? And why? Surprisingly, the models took similar wall-clock times to train, despite the theoretical differences in computational complexity. This unexpected result may be due to the Google Colab environment dynamically allocating computational resources depending on overall server load. Nevertheless, some trends can still be observed:

- **SGD:** Per update, plain SGD is the cheapest computationally. However, it often requires many more epochs to converge, which cancels out the efficiency advantage.
- **SGD with momentum:** The extra computations from maintaining the velocity tensor make it slightly more expensive per step, but high momentum accelerates convergence and reduces the number of required epochs.
- **AdamW:** Each update is more computationally expensive, since it maintains moving averages of both the first and second moments of the gradients. However, because it adapts step sizes dynamically, AdamW converges much faster in practice, making it the most effective optimizer overall in terms of time-to-solution.

4.5 Learning Rate Tuning with AdamW

Finally, we tuned the learning rate for AdamW to evaluate sensitivity. Four different learning rates were tested:

Table 9: AdamW Learning Rate Sensitivity

Learning Rate	Validation F1	Test F1
0.0001	76.2450	75.9581
0.0005	77.8981	77.7155
0.001	78.6309	77.7968
0.005	78.7664	77.7527

The performances were very similar across learning rates. This behavior is expected: AdamW internally scales parameter updates using its estimates of the first and second moments of the gradients. This adaptive nature makes it far less sensitive to the choice of global learning rate than plain SGD. In practice, even if the learning rate is set higher or lower, AdamW adjusts individual step sizes accordingly, maintaining stability and generalization. Training times were unaffected and remained comparable to the optimizer tests above.

4.6 Epoch Reduction Experiment

Even though our final model did not exhibit strong overfitting, we tested the effect of reducing training epochs from 50 to 30 while keeping the learning rate at 0.0005 (baseline model). As expected, the training time was almost halved. However, the model began to underfit, with validation and test F1-scores dropping to 67.77 and 68.14 respectively. This confirms that while early stopping is a powerful regularization technique, reducing epochs too aggressively can prevent the model from fully learning the data distribution.

5 Task 5: Overfitting and Regularization

In this final task, we tested a Feed-Forward Neural Network (FFNN) to evaluate the effects of overfitting and regularization techniques.

Baseline experiment. Q: What do the losses look like? Is the model overfitting? We first ran the model without any regularization to establish a performance benchmark. The training and validation losses indicate that the model already performs well, although a slight overfitting is visible (training loss is consistently lower than validation loss).

- Epoch 20/50: Train Loss = 0.1225, Validation Loss = 0.1408
- Epoch 40/50: Train Loss = 0.0902, Validation Loss = 0.1057
- Validation F1-score = 85.96, Test F1-score = 86.46

FFNN architecture. To test different normalization and regularization methods, we implemented a configurable FFNN:

```
class LastFFNN(nn.Module):
    def __init__(self, input_size, layer_sizes, output_size, use_batchnorm, dropout_rate):
        super(LastFFNN, self).__init__()
        layers = []
        layers.append(nn.Linear(input_size, layer_sizes[0]))
        layers.append(nn.ReLU())
        for i in range(1, len(layer_sizes)):
            layers.append(nn.Linear(layer_sizes[i - 1], layer_sizes[i]))
            if use_batchnorm:
                layers.append(nn.BatchNorm1d(layer_sizes[i]))
            layers.append(nn.ReLU())
            if dropout_rate > 0.0:
                layers.append(nn.Dropout(dropout_rate))
        layers.append(nn.Linear(layer_sizes[-1], output_size))
        self.model = nn.Sequential(*layers)
    def forward(self, x):
        return self.model(x)
```

Effect of regularization techniques. We systematically applied dropout, batch normalization, and weight decay (AdamW optimizer). Macro F1-score was used as the main evaluation metric.

Table 10: Validation and Test F1-scores with different regularization techniques.

Technique	Validation F1 (%)	Test F1 (%)
Baseline (no reg.)	85.96	86.46
Dropout 0.2	82.10	79.96
Dropout 0.3	71.20	71.15
Dropout 0.5	68.85	69.46
Batch Normalization (all layers)	70.56	70.57
Weight Decay 1e−4	83.52	82.05
Weight Decay 1e−3	**86.76**	**87.07**
Weight Decay 1e−2	86.84	85.93

Q: What impact do the different normalization techniques have on validation and testing performance?

- *Dropout:* Adding dropout layers after each non-linear layer consistently decreased performance. Stronger dropout values (0.3, 0.5) led to underfitting, confirmed by the loss plots, as the model required more than 50 epochs to compensate for dropped neurons.
- *Batch Normalization:* Applying batch normalization to all layers unexpectedly harmed performance, introducing noise in the loss curve (especially after epoch 20). We hypothesize this was due to *over-normalization* in smaller hidden layers, which erased useful signals. Indeed, applying batch normalization only to the first two layers restored stability and improved results, confirming the hypothesis.
- *Weight Decay:* Small weight decay (1e−4) had little effect. Increasing it to 1e−3 improved generalization: validation and test F1-scores surpassed the baseline, and training/validation losses were almost overlapping. The confusion matrix also showed better detection of the minority class (*PortScan*), with correct predictions increasing from 44 (baseline) to 50 samples. However, setting weight decay too high (1e−2) decreased performance again, as strong regularization prevented weights from growing sufficiently, leading to underfitting.

In summary, dropout and batch normalization (when applied indiscriminately) harmed performance, while weight decay at the right scale (1e−3) proved to be the most effective regularization method, slightly improving both generalization and minority class classification.