Lab 1 – Feed Forward Neural Networks

Pierfrancesco Elia - s331497, Alessandro Meneghini - s332228,
Ankesh Porwal - s328746

Overview

The main goal of this lab was to explore how Feed Forward Neural Networks (FFNNs) can be applied to cybersecurity problems, specifically for intrusion detection. We worked with the CICIDS2017 dataset, a well-known benchmark in network security that includes a mix of normal and malicious traffic. Our task was to build and evaluate models capable of classifying different types of network behavior using PyTorch.
We began with preprocessing the dataset by cleaning missing values and duplicates, selecting key features, applying normalization, and encoding labels. The data was then split into training, validation, and test sets, using an hold-out technique. Special care was taken to ensure consistent preprocessing across all splits to avoid data leakage.
Our first experiments involved building a shallow FFNN with one hidden layer. Initially, the model performed poorly with a Linear activation function. When we switched to ReLU, performance improved significantly. This highlighted the importance of non-linearity in learning complex patterns. Early stopping was used to select the best-performing model based on validation loss.
We then explored how the model was influenced by specific features, especially *Destination Port*. We found that the model heavily relied on this field to detect Brute Force attacks. When we changed the port in the test set, performance dropped dramatically. This indicated that the feature introduces a bias. To resolve this, we removed the port field and retrained the model, this time also introducing class-weighted loss functions to address class imbalance.
Next, we expanded the architecture to deeper FFNNs with multiple hidden layers. These models achieved better generalization, but also showed signs of overfitting as complexity increased. We applied regularization techniques such as dropout, batch normalization, and weight decay to address this issue. These adjustments improved validation and test performance across the board.
Finally, we tested different batch sizes, activation functions, and optimizers. ReLU consistently provided the best results, and AdamW was the most effective optimizer in terms of both accuracy and training speed.
This lab helped us understand not only how to build neural networks, but also how to debug and improve them through careful analysis, tuning, and regularization — all within a realistic cybersecurity scenario.

1 Task 1

1.1 Data Preprocessing

The first step in developing a robust Machine Learning pipeline is data preprocessing. For the CICIDS2017 dataset, this involved ensuring data consistency, removing noisy or irrelevant features, and applying appropriate normalization and splitting procedures.
Figure 1 illustrates how we analyzed the distribution of each feature with respect to the label classes; this approach helps identify patterns or class-specific behaviors in the data.

1.1.1 Data Cleaning

We started by removing all `NaN` and `Inf` values as well as duplicate entries from the dataset. This helped eliminate inconsistencies and avoid misleading learning signals during training. A few features showed almost no variation (e.g., always zero or always one), such as `SYN Flag Count` and `Fwd PSH Flags`. These were dropped after confirming that they added no value for classification, specifically, they were always 1 for benign flows and therefore encoded a bias.

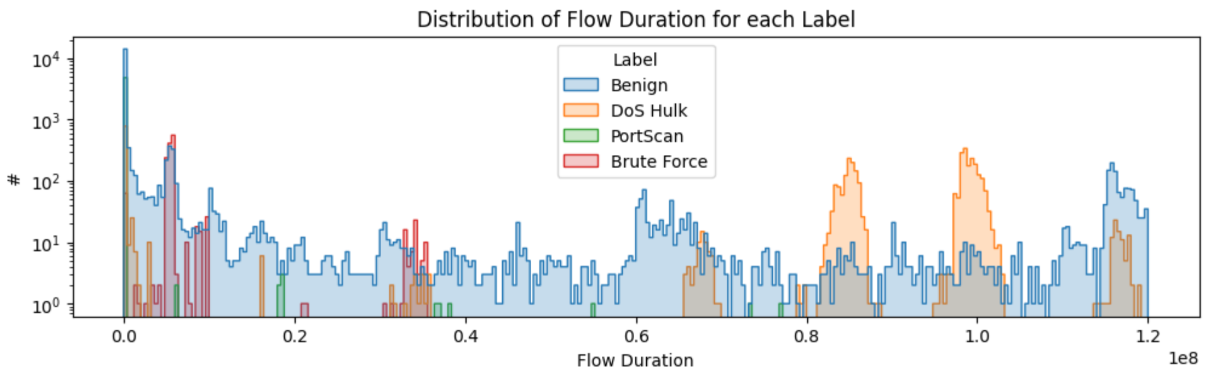

Figure 1: Distribution of Flow duration for each Label in the dataset.

- **Removed features:** `SYN Flag Count`, `Fwd PSH Flags`
- **Shape after cleaning:** `(29386, 15)` $\rightarrow$ `(29386, 13)` + 1 label column

1.1.2 Feature Selection

Feature selection was carried out based on domain knowledge and exploratory data analysis. Features with extremely low variance and no discriminative value—such as `SYN Flag Count` and `Fwd PSH Flags`—were removed. These features were always active (value = 1) in benign traffic and inactive in attacks, making them trivially correlated with the label and not generalizable.
In addition, we prioritized keeping only numerical features that were suitable for neural network training. We assessed that highly sparse categorical features did not contribute meaningful information for our task. These variables were excluded to prevent introducing bias or noise into the model, and to ensure a more stable and generalizable learning process. This decision also reduced input dimensionality and computational cost, benefiting both performance and interpretability.

1.1.3 Standardization

Is the preprocessing the same for the test partition?
The same transformation is applied, but the fitting is only performed on the training set. Specifically, we applied `StandardScaler` to the features to standardize them (mean = 0, std = 1). However, we fit the scaler *only* on `X_train`, then used the same fitted scaler to transform `X_val` and `X_test`. This prevents information leakage and ensures that the test data remains unseen.

1.1.4 Label Encoding

Target labels such as `Benign`, `DoS Hulk`, `PortScan`, and `Brute Force` were encoded into numerical classes using `LabelEncoder`. This transformation was necessary for the model to process categorical labels as integer classes for the classification task. The mapping used was:

- `Benign`: 0
- `Brute Force`: 1
- `DoS Hulk`: 2
- `PortScan`: 3

1.1.5 Train/Validation/Test Split

We splitted the dataset into three partitions, using hold-out tecnique:

- 60% for training (`17,631 samples`)
- 20% for validation (`5,877 samples`)
- 20% for testing (`5,878 samples`)

Splitting was done using `train_test_split` with stratification to maintain label distribution across all subsets. This ensures consistency and reliable validation of the model's performance.

1.1.6 Outlier Handling

Visual inspection of the feature distributions revealed the presence of significant outliers, particularly in flow-based features. These were not removed directly, but normalization helped reduce their impact. Additional techniques, such as robust scaling or clipping, were considered but not adopted in the final implementation due to satisfactory model performance.

2 Task 2

2.1 Shallow Neural Network

The second task required the implementation of a shallow Feed-Forward Neural Network (FFNN) to serve as a baseline classifier for network intrusion detection. This simple architecture allows for the evaluation of the model's capacity to learn from data, before experimenting with more complex deep learning setups.

2.1.1 Model Architecture

We designed a neural network with a single hidden layer and tested it with different configurations by varying the number of neurons: `32`, `64`, and `128`. The output layer used the standard softmax formulation via `CrossEntropyLoss`, as the task involved multi-class classification into four categories: `Benign`, `Brute Force`, `DoS Hulk`, and `PortScan`.
Initially, the model was trained with a **Linear** activation function. As expected, this significantly limited the model's ability to learn nonlinear patterns in the data.

2.2 Training Process

The data was processed in mini-batches of size 64 using `DataLoader`. Training was performed with the following hyperparameters:

- **Loss function**: `CrossEntropyLoss`
- **Optimizer**: `AdamW`
- **Learning rate**: 0.0005
- **Epochs**: Up to 100 with **early stopping** (patience = 20)
- **Weight initialization**: Default (PyTorch)
- **Regularization**: None

The model with the lowest validation loss was saved and restored after training.

2.3 Loss Curve Evolution

Figure 2 and Figure 3 illustrate how the training and validation loss evolved across epochs for the two activation functions tested.
With Linear activation, the loss converged slowly and plateaued at higher values (Train: 0.366, Val: 0.343). With ReLU, the loss decreased rapidly and stabilized at much lower values (Train: 0.079, Val: 0.085).

2.4 Model Selection Strategy

The best model was selected based on the lowest validation loss. During training, early stopping was triggered if the validation loss did not improve for 20 consecutive epochs. The model with the best weights was then used for final evaluation on both the validation and test sets.

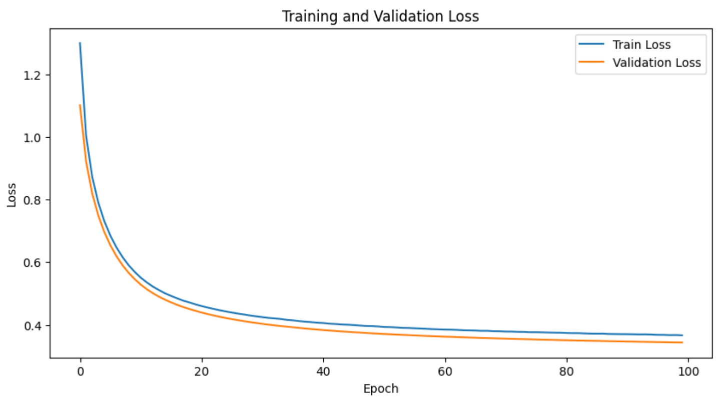

Figure 2: Training and Validation Loss – **Linear Activation**.

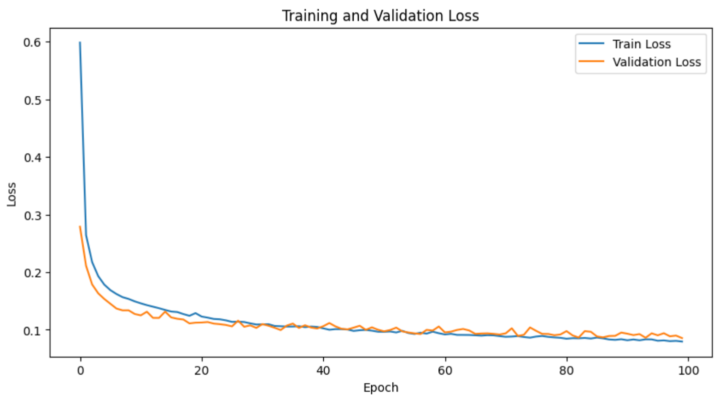

Figure 3: Training and Validation Loss – **ReLU Activation**.

2.5 Results and Observations

The model trained with a **Linear activation** struggled to learn non-linear decision boundaries and consistently underfit the data, regardless of the number of neurons. Notably, it failed to detect Brute Force attacks (class 1), achieving an F1-score of 0.00 across all dataset partitions. Despite an overall test accuracy of approximately **88.7%**, the model lacked the expressiveness needed to capture class-specific patterns.

When we switched the activation function to **ReLU**, performance improved significantly across all configurations. The ReLU-based models were better at separating the classes and captured more complex patterns in the feature space. In particular, the F1-score for Brute Force increased from **0.00** to **0.95** on the test set, and the overall test accuracy rose to **97.7%**.

Among the three tested configurations, the network with **128 neurons** in the hidden layer and ReLU activation achieved the best results and was chosen as baseline for the following experiments.

Activation	**Dataset**	**Accuracy**	**Macro F1**	**Weighted F1**	**Brute Force F1 (class 1)**
Linear	Train	87.83%	0.67	0.86	0.00
	Validation	88.19%	0.67	0.86	0.00
	Test	88.72%	0.68	0.87	0.00
ReLU	Train	97.10%	0.96	0.97	0.93
	Validation	97.35%	0.96	0.97	0.95
	Test	97.65%	0.97	0.98	0.95

Table 1: Performance comparison of shallow FFNN models with Linear and ReLU activations.

2.6 Why Was the Initial Performance Poor?

The poor performance observed with the linear activation is expected. Without any non-linear transformation, the model behaves like a linear classifier, which is not powerful enough to capture complex patterns in the data. Additionally, the simple architecture (a single layer, no regularization) further limited its representational power.

2.7 Summary

This task demonstrated the importance of non-linear activation functions for neural networks. The switch from Linear to ReLU activation drastically improved the model's ability to classify even the minority classes, showing that a shallow architecture can still be effective when properly configured.

3 Task 3

3.1 The Impact of Specific Features

In this task, we analyzed how specific feature distributions—particularly the `Destination Port`—impacted the model's ability to learn generalizable patterns. We assessed whether certain features caused inductive biases and explored strategies to mitigate them.

3.1.1 Feature Bias: The Case of Destination Port

During initial analysis, we observed that **all Brute Force attacks occurred on port 80**. This is not a realistic assumption, as such attacks may target various ports in real-world scenarios. This forms an inductive bias: the model might associate class "Brute Force" exclusively with port 80 instead of learning general behavior.
To validate this, we created a modified version of the test set by replacing port 80 with **port 8080** for all Brute Force instances. After running inference on this new test set using the previously trained model (with port 80), performance on class 1 dropped drastically:

- **Precision, recall and F1-score** for class 1 (Brute Force): all **0.00**.
- **Overall accuracy**: dropped to **93.13%**.

This confirms that the model overfitted to port 80 as a proxy for Brute Force, failing to generalize when the port changes. This example clearly illustrates the importance of evaluating per-class metrics like precision, recall, and F1-score rather than relying solely on overall accuracy. Even though the model's accuracy remains high at 93.13%, it completely fails to detect class 1 (Brute Force). This emphasizes that high accuracy can mask poor performance on critical or minority classes, which may be unacceptable in real-world scenarios such as intrusion detection.

3.1.2 Feature Removal and Reprocessing

To mitigate this bias, we removed the `Destination Port` feature and repeated the full preprocessing pipeline:

- Dropped NaNs and duplicate rows.
- Re-applied `StandardScaler` (fitting only on training set).
- Encoded labels and split dataset (60%-20%-20%).

How many PortScan instances do you now have after preprocessing?

- Before: **4849**
- After removing `Destination Port`: **285**

Q: Why do you think PortScan is the most affected class after dropping duplicates?
Many PortScan entries differed only in their `Destination Port`, so removing that feature caused many samples to become duplicates and be discarded.
Are the classes now balanced?
No. After preprocessing, class imbalance worsened:

- Benign: 16889
- DoS Hulk: 3868
- Brute Force: 1427
- PortScan: 285

3.1.3 Model Retraining without Destination Port

We retrained the best-performing model (128 ReLU neurons) on the new dataset (13 features, no `Destination Port`). The model reached:

- **Test Accuracy: 96.40%**
- **F1-score (class 3 / PortScan)**: **0.44**

Despite overall high accuracy, the model struggled with PortScan, which became extremely rare.

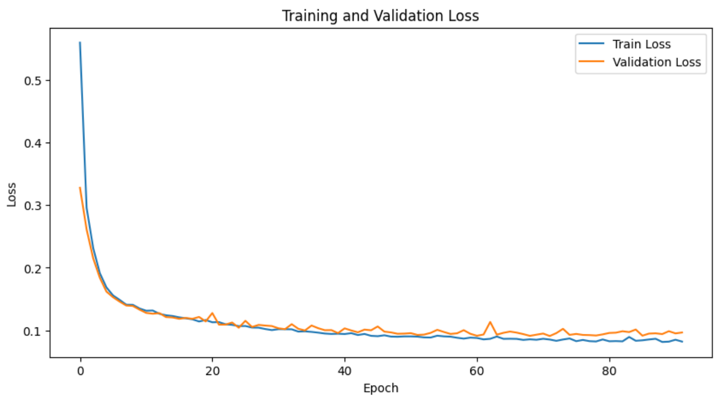

Figure 4: Training and validation loss without `Destination Port` feature.

3.1.4 Weighted Loss for Class Imbalance

To address the imbalance, we computed class weights using `compute_class_weight(balance=True)` and passed them to `CrossEntropyLoss`. This reweighted loss function allowed the model to better handle rare classes.

Performance after weighting

- **Test Accuracy: 93.10%** (↓)
- **F1-score (class 3 / PortScan): 0.36** (↓ from 0.44, but **recall: 0.94**)
- **Precision (class 3): 0.22** (↓ due to false positives)

The weighted loss function improved recall for rare classes, especially PortScan (from 0.44 to 0.94), but at the cost of lower precision (↑ false positives). The overall F1-score and accuracy slightly decreased, reflecting the expected trade-off between class balance and global performance.

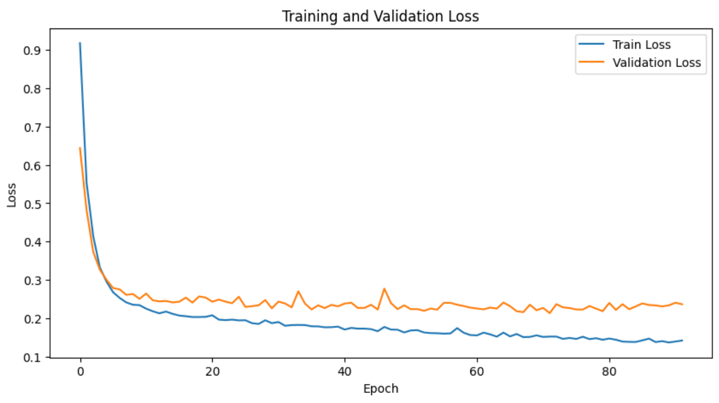

Figure 5: Training and validation loss using class-weighted loss.

Configuration	Test Acc.	F1-score Cl. 3	Recall Cl. 3	Precision Cl. 3
Without port, no weights	96.40%	0.44	0.44	0.44
Without port, with weights	93.10%	0.36	0.94	0.22

Table 2: Performance comparison for PortScan class (class 3).

3.1.5 Conclusion

This task illustrated how strong feature correlations can mislead the model (e.g., Brute Force $\rightarrow$ port 80). Removing the biased feature and applying weighted loss allowed the model to generalize better and emphasize rare but critical classes. However, it also revealed the trade-offs between precision and recall in imbalanced classification.

4 Task 4

4.1 Deep Neural Network

In this task, we extended our model architecture from a shallow neural network to a deeper Feed Forward Neural Network (FFNN), aiming to capture more complex patterns and improve classification performance. We also systematically tested the effects of different hyperparameters and architectural choices, including batch size, activation function, and optimizer.

4.1.1 Deep Network Design

We experimented with networks composed of 2 to 5 hidden layers. The number of neurons per layer varied across configurations, using values such as {2, 4, 8, 16, 32}. All hidden layers used the ReLU activation function unless otherwise stated. The output layer remained unchanged, using softmax via `CrossEntropyLoss` for multiclass classification.
Early stopping was applied in all cases to prevent overfitting, and model selection was based on the lowest validation loss. The best-performing deep network had four hidden layers with 16, 32, 32, and 16 neurons respectively. This configuration provided a good balance between model complexity and generalization.

Best Architecture: `[16, 32, 32, 16]`
Validation Accuracy: 96.35%
Test Accuracy: 96.31%

4.1.2 The Impact of Batch Size

We evaluated the performance of the best model using different batch sizes: 1, 32, 64, 128, 512.

- **Smaller batch sizes** (e.g., 1, 32) produced noisy but fine-grained updates, yielding better generalization but with much longer training times.

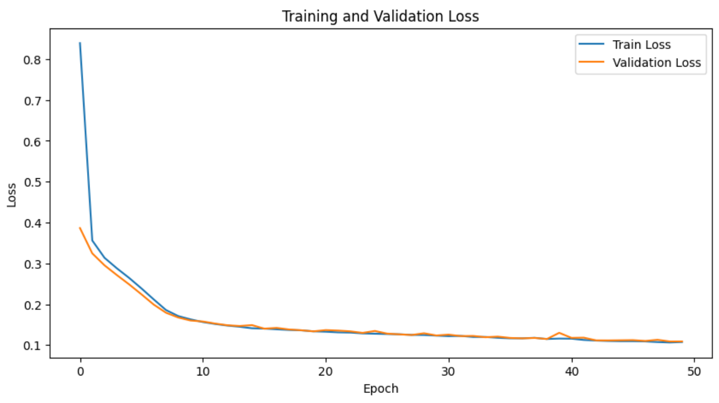

Figure 6: Training and Validation Loss for best architecture [16, 32, 32, 16].

- **Larger batch sizes** (e.g., 128, 512) reduced training time but converged to suboptimal minima and increased the risk of overfitting.

Summary:

Batch Size	Val Accuracy	Test Accuracy	Training Time (s)
1	95.55%	95.44%	1432.37
32	95.55%	95.55%	58.11
64	**96.13%**	**96.19%**	35.33
128	96.08%	95.95%	22.22
512	94.08%	93.88%	14.91

Table 3: Effect of Batch Size on Model Performance.

Best batch size: 64 — optimal trade-off between generalization and training efficiency.

4.1.3 The Impact of the Activation Function

We tested the following activation functions in the hidden layers:

- **Linear**: underfitting, no non-linearity introduced.
- **Sigmoid**: vanishing gradients, slower convergence.
- **ReLU**: best performance overall.

Summary:

Activation	Val Accuracy	Test Accuracy
Linear	89.81%	89.10%
Sigmoid	90.21%	89.68%
ReLU	**96.24%**	**96.08%**

Table 4: Comparison of Activation Functions.

Explanation: ReLU avoids vanishing gradients, speeds up convergence, and handles sparse activations efficiently.

4.1.4 The Impact of the Optimizer

We tested several optimizers:

- SGD

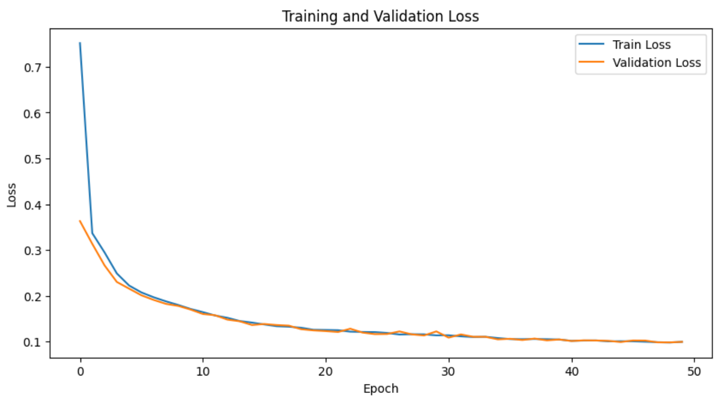

Figure 7: Loss Curve using ReLU Activation.

- SGD + Momentum (0.1, 0.5, 0.9)
- AdamW

Summary:

Optimizer	Val Accuracy	Test Accuracy
SGD	94.86%	94.68%
SGD + Momentum 0.1	93.99%	93.55%
SGD + Momentum 0.5	94.50%	94.24%
SGD + Momentum 0.9	**95.77%**	**96.02%**
AdamW	95.26%	94.95%

Table 5: Effect of Optimizers on Model Performance.

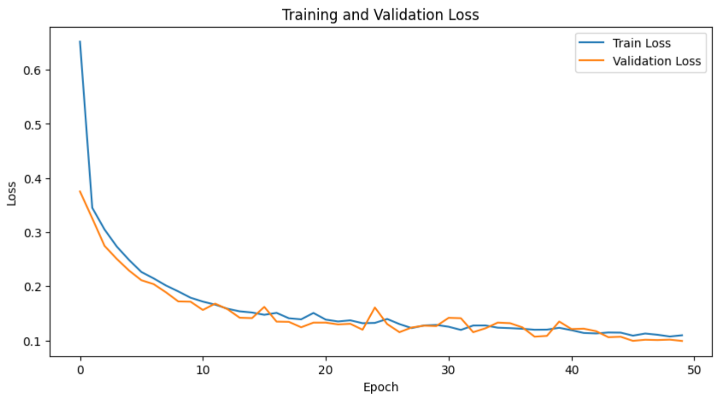

Figure 8: Best Optimizer: SGD with Momentum 0.9.

Conclusion:

- Optimizers significantly impact convergence and final accuracy.
- Momentum helps SGD escape local shallow minima; 0.9 yielded best results.
- AdamW had faster convergence but slightly lower test performance.

4.1.5 Final Observations

Deeper architectures significantly improved the model's ability to capture non-linear and subtle patterns. However, as model capacity increased, overfitting became more likely. This justified the application of regularization strategies, which are discussed in Task 5.

5 Task 5

5.1 Overfitting and Regularization

In this task, we explored the behavior of an overparameterized Feed Forward Neural Network (FFNN) and tested regularization strategies to mitigate overfitting.

5.1.1 Initial Model without Regularization

We trained a 6-layer FFNN with architecture `[256, 128, 64, 32, 16]` using ReLU activation, batch size 128, and AdamW optimizer. The model quickly achieved low training loss, but validation loss plateaued early and test accuracy stagnated, indicating overfitting.
Training Accuracy: 96.14% **Validation Accuracy:** 96.35% **Test Accuracy:** 96.31%
Q: Is the model overfitting?
Yes. The training loss continued to decrease while validation loss stagnated, revealing a gap typical of overfitting.

5.1.2 Dropout Regularization

We introduced `Dropout(0.5)` after each hidden layer. This forced the network to rely less on individual neurons and improved generalization.
Test Accuracy: Improved robustness; slight gain in generalization, reduced overfitting variance.

5.1.3 Batch Normalization

We inserted `BatchNorm1d` layers before activations in selected layers. Combined with dropout, it stabilized training and provided smoother loss curves.
Validation Accuracy: Slight improvement over dropout-only version. Training became more stable.

5.1.4 Weight Decay (L2 Regularization)

We evaluated different `weight_decay` values in AdamW: 0.0, 0.01, 0.02, 0.1. The best result was obtained with 0.1.

Weight Decay	Train Acc.	Val Acc.	Test Acc.
0.0	94.35%	94.37%	94.21%
0.01	94.47%	94.33%	94.17%
0.02	94.53%	94.39%	94.21%
0.1	**94.60%**	**94.46%**	**94.39%**

Table 6: Impact of Weight Decay on Performance

5.1.5 Conclusions

- **Dropout** improved generalization by reducing neuron co-adaptation.
- **Batch Normalization** stabilized training and helped faster convergence.
- **Weight Decay** reduced model complexity and boosted validation performance slightly.

Q: What impact do the regularization techniques have?
All three techniques reduced overfitting. The best generalization was achieved using dropout + batch normalization + weight decay (0.1), which led to a more balanced validation and test accuracy while stabilizing training behavior.