كلية العلوم و التقنيات بطنجة

ABDELMALEK ESSAADI UNIVERSITY
FACULTY OF SCIENCES AND TECHNIQUES
OF TANGIER
DEPARTMENT OF COMPUTER ENGINEERING

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

Workshop 1

PyTorch: DNN/MLP for Classification & Regression

Speciality

Master Security IT and Big Data

(Computer Engineering Department)

Supervised by

Pr ELAACHAK Lotfi

Preapared by

BOUASSAB Chaimae

Contents

1. Introduction

- **Lab Objective**

The main purpose of this lab is to **learn how to use the PyTorch library** by building **Deep Neural Networks (DNNs)**, also known as **Multi-Layer Perceptrons (MLPs)**, to solve common machine learning problems like **classification** and **regression**.

- **Datasets Overview**

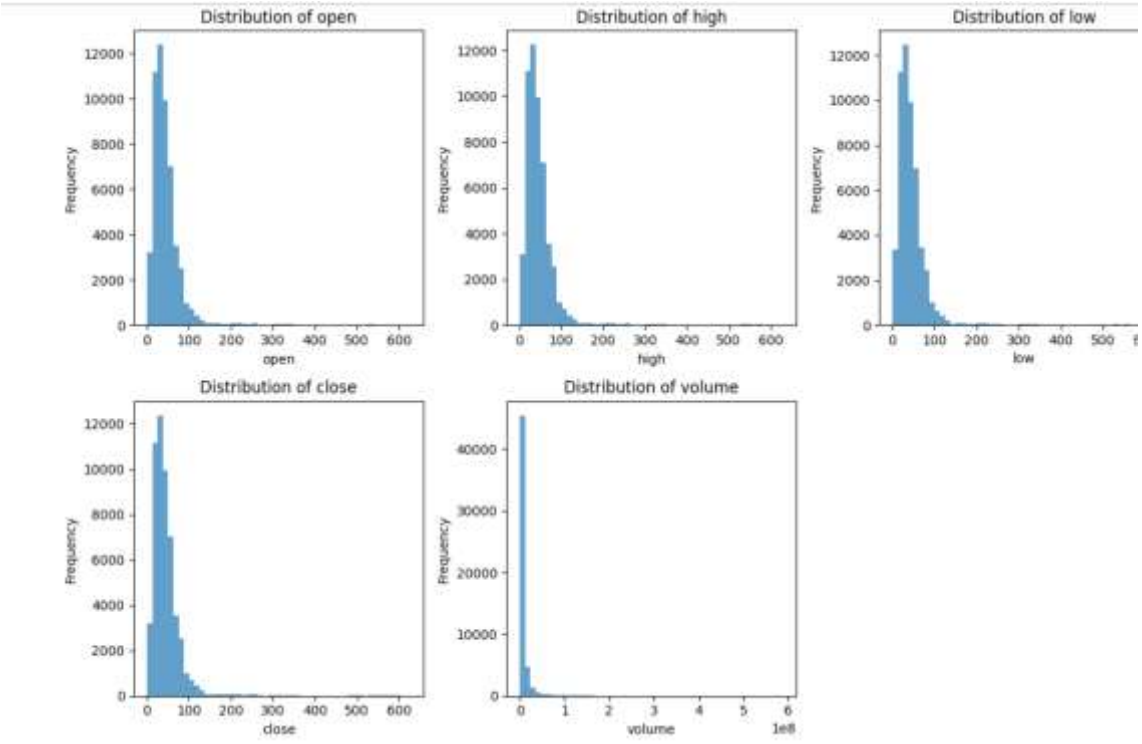
2. Part One: Regression Task (NYSE Dataset)

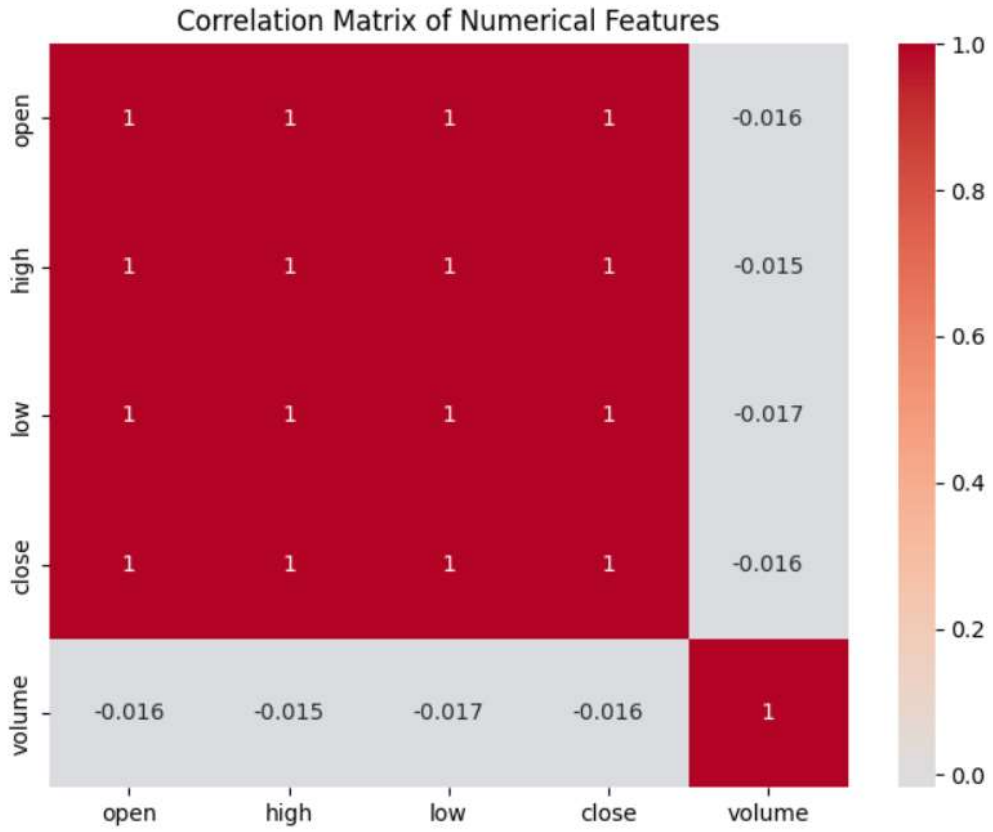
2.1 Exploratory Data Analysis

- **Statistical summary**

```
Statistical Summary:
                open         close           low          high        volume
count   53244.000000  53244.000000  53244.000000  53243.000000  5.324300e+04
mean       47.639114     47.654359     47.020299     48.221769  7.845860e+06
std        47.773543     47.718509     47.155959     48.292120  1.805251e+07
min         1.140000      1.130000      1.110000      1.170000  0.000000e+00
25%        25.059999     25.040001     24.719999     25.379999  1.628700e+06
50%        38.540001     38.570000     38.060001     39.020000  3.427000e+06
75%        56.072501     56.112500     55.410000     56.764999  7.249000e+06
max       627.181073    626.751061    624.241073    629.511067  5.890618e+08
```

- **Data visualization**





2.2 Data Preprocessing

- **Handling missing values**

```
Missing Values per Column:
date      364
symbol      0
open        0
close       0
low         0
high        1
volume      1
dtype: int64
```

2.3 DNN Model Architecture

- **Model design (layers, neurons, activation functions)**

```python
# Define MLP Architecture
class RegressionMLP(nn.Module):
    def __init__(self, input_size=4, hidden_sizes=[64, 32, 16]):
        super(RegressionMLP, self).__init__()
        layers = []
        prev_size = input_size
        for hidden_size in hidden_sizes:
            layers.extend([
                nn.Linear(prev_size, hidden_size),
                nn.ReLU(),
                nn.Dropout(0.2)  # Basic dropout for regularization
            ])
            prev_size = hidden_size
        layers.append(nn.Linear(prev_size, 1))  # Output layer for regression
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

- **Model implementation in PyTorch**

To build the model, we use the nn.Module class provided by PyTorch. The implemented architecture includes:

- **Input layer**: size equal to the number of selected features.
- **Three hidden layers** with progressively decreasing number of neurons (64 → 32 → 16).
- **ReLU activation function** after each linear layer to introduce non-linearity. ReLU(x)=max(0,x)
- **Dropout (0.2)** to reduce overfitting by randomly deactivating neurons during training.
- **Output layer** with a single neuron, since this is a regression problem

```
RegressionMLP(
  (network): Sequential(
    (0): Linear(in_features=4, out_features=64, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=64, out_features=32, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=32, out_features=16, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.2, inplace=False)
    (9): Linear(in_features=16, out_features=1, bias=True)
  )
)
Epoch [10/50], Loss: 0.0538
Epoch [20/50], Loss: 0.0554
Epoch [30/50], Loss: 0.0521
Epoch [40/50], Loss: 0.0501
Epoch [50/50], Loss: 0.0513
Test MSE: 0.0090
```

We used the Mean Squared Error (MSE) loss:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The loss values fluctuate slightly but decrease overall:

- From **0.0538 → 0.0501**, showing improvement
- The average prediction error is **very small**, meaning the model fits the data well.

2.4 Hyperparameter Tuning

- **GridSearch setup**

GridSearch helps automatically determine the best combination of hyperparameters such as:

- learning rate
- optimizer
- number of epochs
- hidden layer structure
- batch size

Since PyTorch models cannot be used directly with GridSearchCV, we created a wrapper class:

```
class PyTorchRegressor(BaseEstimator, RegressorMixin):
    def __init__(self, hidden_sizes=[64, 32, 16], lr=0.001, epochs=50,
                 optimizer='Adam', batch_size=32):
        self.hidden_sizes = hidden_sizes
        self.lr = lr
        self.epochs = epochs
        self.optimizer = optimizer
        self.batch_size = batch_size
        self.model = None
        self.criterion = nn.MSELoss()
```

- **Parameter selection (learning rate, optimizer, epochs, architecture)**

```
# OPTIMIZED: Reduced parameter grid
param_grid = {
    'lr': [0.001, 0.01],           # Reduced from 3 to 2
    'optimizer': ['Adam'],          # Only Adam (usually best)
    'epochs': [10, 20],             # Reduced from 3 to 2, removed 5
    'hidden_sizes': [[64, 32, 16]] # Only 1 architecture for speed
}
```

- **Best model selection**

```
Starting GridSearch (this will take a few minutes)...
Fitting 3 folds for each of 4 candidates, totalling 12 fits
[CV] END epochs=10, hidden_sizes=[64, 32, 16], lr=0.001, optimizer=Adam; total time=  17.3s
[CV] END epochs=10, hidden_sizes=[64, 32, 16], lr=0.001, optimizer=Adam; total time=  14.9s
[CV] END epochs=10, hidden_sizes=[64, 32, 16], lr=0.001, optimizer=Adam; total time=  15.0s
[CV] END epochs=10, hidden_sizes=[64, 32, 16], lr=0.01, optimizer=Adam; total time=  16.0s
[CV] END epochs=10, hidden_sizes=[64, 32, 16], lr=0.01, optimizer=Adam; total time=  14.7s
[CV] END epochs=10, hidden_sizes=[64, 32, 16], lr=0.01, optimizer=Adam; total time=  14.7s
[CV] END epochs=20, hidden_sizes=[64, 32, 16], lr=0.001, optimizer=Adam; total time=  29.6s
[CV] END epochs=20, hidden_sizes=[64, 32, 16], lr=0.001, optimizer=Adam; total time=  29.7s
[CV] END epochs=20, hidden_sizes=[64, 32, 16], lr=0.001, optimizer=Adam; total time=  30.8s
[CV] END epochs=20, hidden_sizes=[64, 32, 16], lr=0.01, optimizer=Adam; total time=  29.8s
[CV] END epochs=20, hidden_sizes=[64, 32, 16], lr=0.01, optimizer=Adam; total time=  30.7s
[CV] END epochs=20, hidden_sizes=[64, 32, 16], lr=0.01, optimizer=Adam; total time=  32.6s

Best Params: {'epochs': 20, 'hidden_sizes': [64, 32, 16], 'lr': 0.001, 'optimizer': 'Adam'}
Best MSE: 0.0044
Test MSE: 1.9519
```

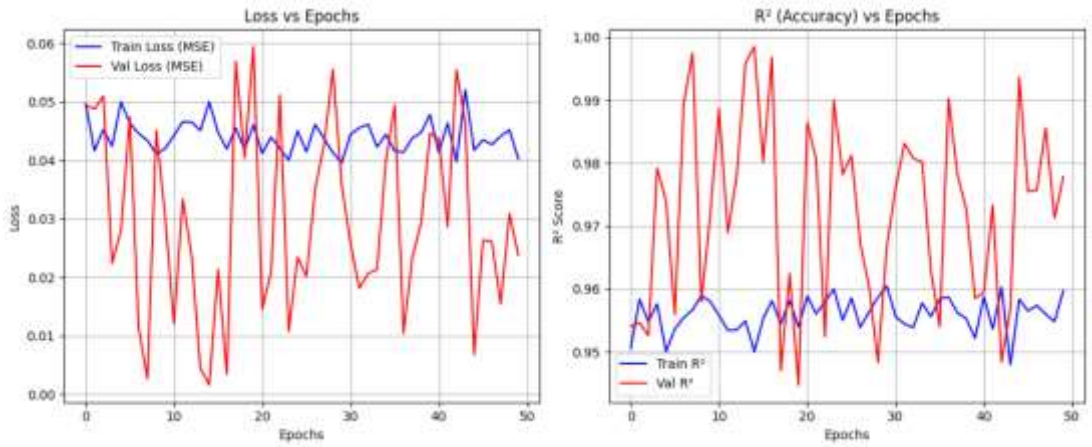
GridSearch enabled the evaluation of different combinations of neural network hyperparameters.

Each configuration was evaluated using 3-fold cross-validation to ensure a reliable performance estimate.

Best MSE = 0.0044 ➡ Cela indique une erreur très faible sur les données d'entraînement/validation.

During the final evaluation on the test data, the model achieved an MSE of 1.9519, indicating a slight increase in error, probably related to a difference in distribution between the training and test data, or to a slight overfitting phenomenon.

2.6 Regularization Techniques



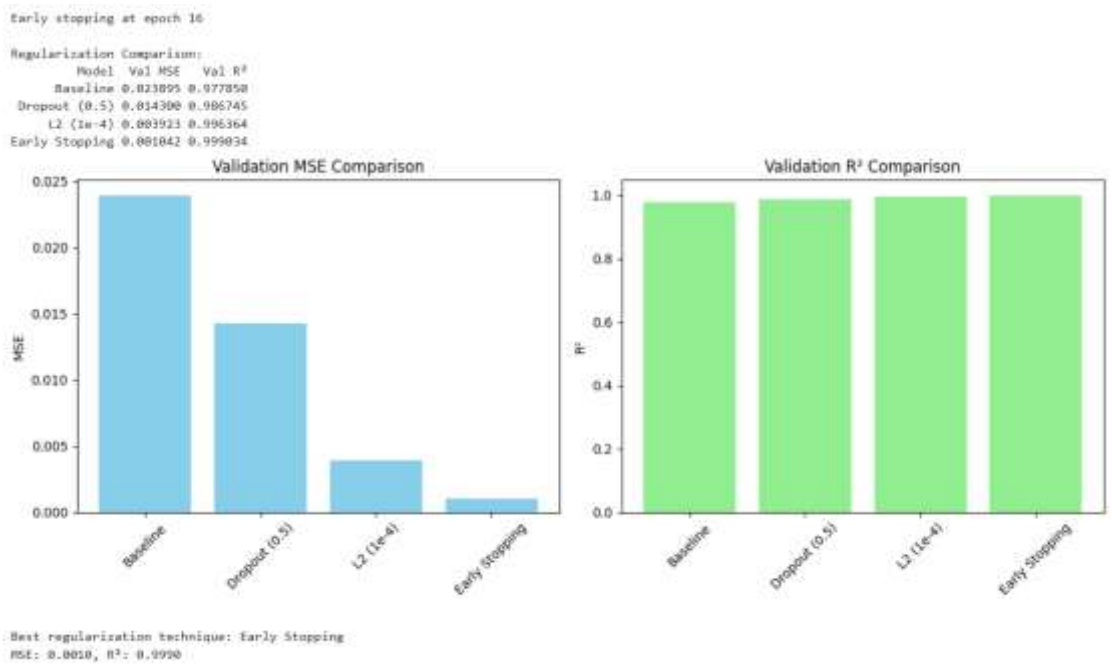
Training curves fluctuate ⇒ the model lacks regularization and stability.

This is exactly where dropout, L1/L2, and batch-norm help.

Dropout Helps reduce overfitting by forcing neurons not to rely on each other.

To improve stability and reduce overfitting, we recommend implementing regularization techniques such as:

- **Dropout: to reduce co-adaptation and improve robustness**
- **L1/L2 regularization: to constrain weight growth and simplify the model**
- **Batch Normalization: to stabilize activations and accelerate convergenc**



- Early stopping is the best, with extremely low error

Early Stopping, which halted training at epoch 16, achieved the best performance overall (MSE = 0.0010, R^2 = 0.9990).

The first model is the baseline MLP architecture, consisting of three fully connected layers (64 → 32 → 16) with ReLU activation and no regularization. This model serves as the reference point against which all other regularized models (Dropout, L2, Early Stopping) are compared.

3. Part Two: Multi-Class Classification (Predictive Maintenance Dataset)

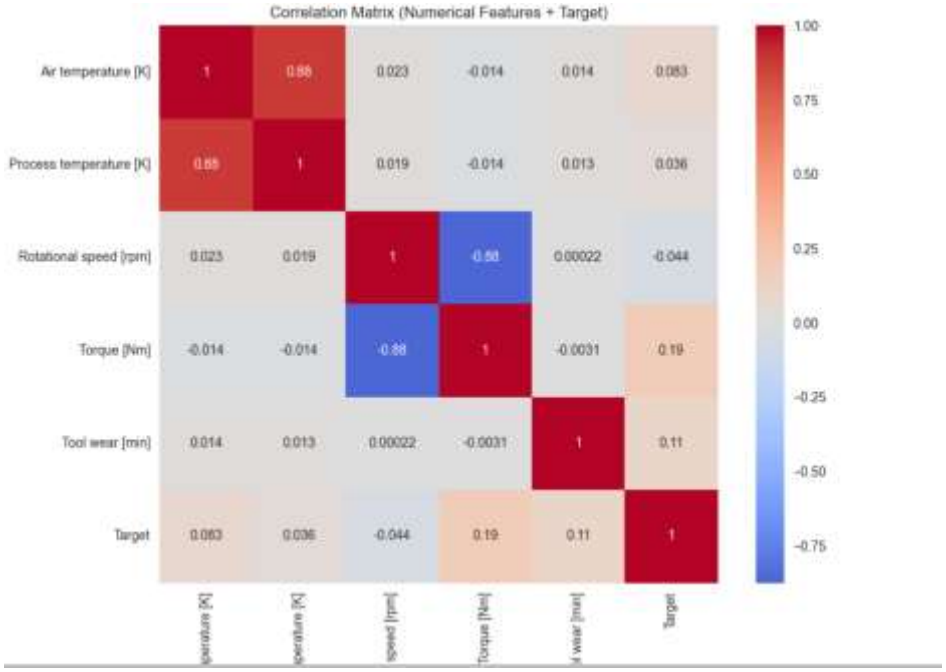
3.1 Data Preprocessing

- Data cleaning
- Standardization/Normalization
- Encoding categorical variables

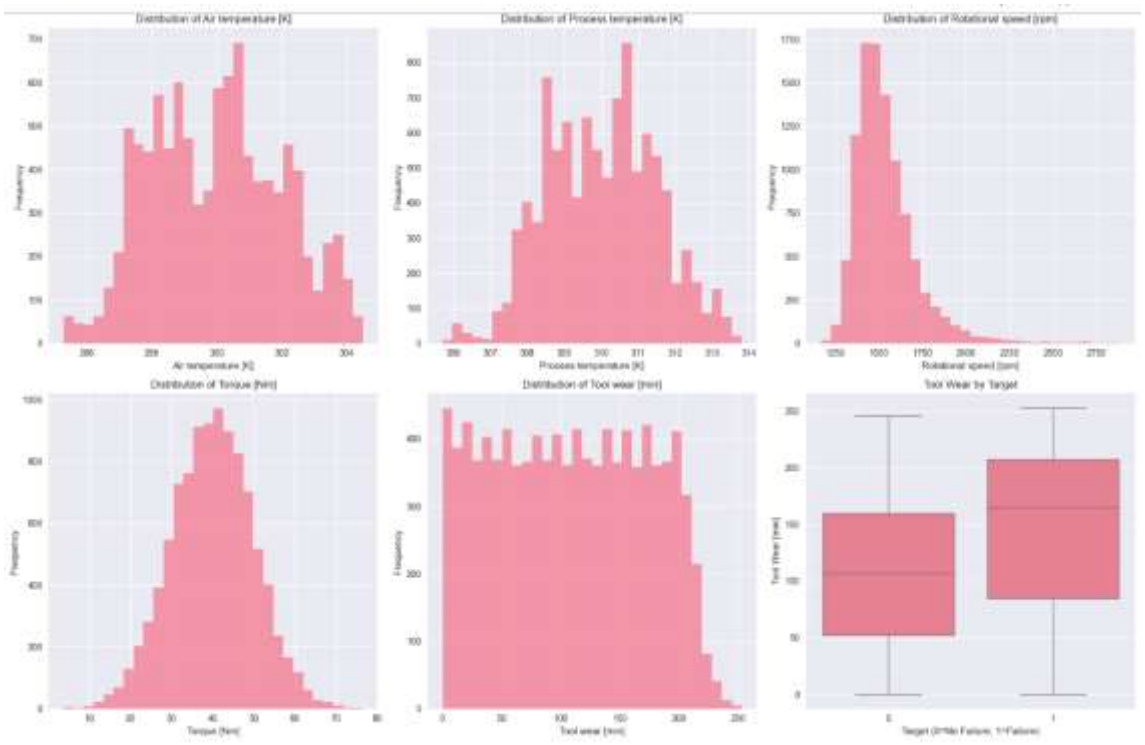
```
Original shape: (10000, 10)
Missing values:
 UDI                        0
Product ID                  0
Type                        0
Air temperature [K]         0
Process temperature [K]     0
Rotational speed [rpm]      0
Torque [Nm]                 0
Tool wear [min]             0
Target                      0
Failure Type                0
dtype: int64
Data types after cleaning:
 UDI                          int64
Product ID                   object
Type                       category
Air temperature [K]         float64
Process temperature [K]     float64
Rotational speed [rpm]      float64
Torque [Nm]                 float64
Tool wear [min]             float64
Target                        int64
Failure Type                 object
dtype: object
Sample after cleaning:
   UDI Product ID Type  Air temperature [K]  Process temperature [K]  \
0    1     M14860    M                298.1                    308.6
1    2     L47181    L                298.2                    308.7
2    3     L47182    L                298.1                    308.5
3    4     L47183    L                298.2                    308.6
4    5     L47184    L                298.2                    308.7
```

3.2 Exploratory Data Analysis

- Data distribution analysis
- Feature correlations



- **Class imbalance visualization**



Tool Wear is a strong indicator of failure. Failures generally occur when the tool wear is in the upper range (median and IQR are higher for Target=1). The model should utilize this feature heavily.

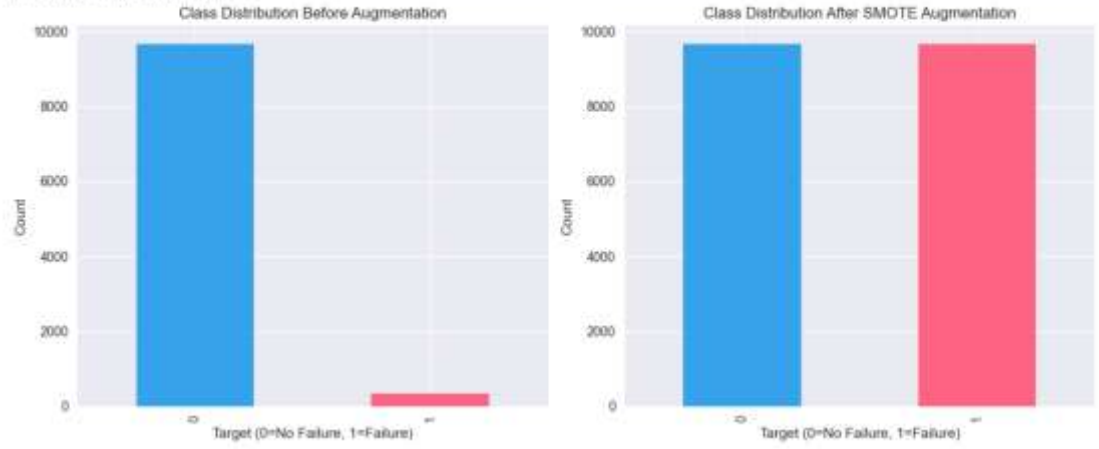
3.3 Data Augmentation

Data augmentation serves to **increase the diversity and quantity of the training data** without actually collecting new, real-world data. It helps Prevent Overfitting and Improve Model Robustness balancing the class distribution, the model is forced to learn the distinguishing features of the rare failure events, leading to better **Recall (Sensitivity)** and **F1-scores** for the classes that matter most.

- **Balancing techniques (SMOTE, oversampling, undersampling)**

```
# Step 3: Apply Data Augmentation with SMOTE (Oversampling Minority Class)
smote = SMOTE(random_state=42, k_neighbors=5)  # Synthetic samples based on nearest neighbors
X_resampled, y_resampled = smote.fit_resample(X, y)

# Convert back to DataFrame for ease
X_resampled = pd.DataFrame(X_resampled, columns=X.columns)
y_resampled = pd.Series(y_resampled, name='Target')

print("\nAugmented X shape:", X_resampled.shape)
print("Augmented y distribution:\n", y_resampled.value_counts(normalize=True).round(3))

# Step 4: Visualize Class Balance Before/After
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
```

```
Augmented X shape: (19322, 7)
Augmented y distribution:
 Target
0    0.5
1    0.5
Name: proportion, dtype: float64
```



The successful application of SMOTE is a critical step for the predictive maintenance classification:

1. Enables Learning: The model can now learn the distinct features and patterns of the Failure class (1).
2. Improves Key Metrics: This step is expected to improve metrics, such as Recall (Sensitivity) and the F1-Score, ensuring the model actually detects rare failure events rather than just ignoring them.

```
=== AUGMENTATION COMPLETE ===
Technique: SMOTE (Synthetic Minority Over-sampling Technique)
- Generates synthetic samples for the minority class (failures).
- New balance: ~50/50 split.
- Ready for balanced modeling (e.g., no need for class weights).
```

```
#4. Establish a Deep Neural network Architecture by using PyTorch library to handle the multicl
```

3.4 DNN Model Architecture

- **Model design for classification**
- **PyTorch implementation**

```python
# Step 4: Define DNN Architecture for Multi-Class Classification
class PredictiveMaintenanceDNN(nn.Module):
    def __init__(self, input_size, num_classes=6, hidden_sizes=[128, 64, 32]
        super(PredictiveMaintenanceDNN, self).__init__()
        layers = []
        prev_size = input_size

        # Hidden layers with ReLU and Dropout
        for hidden_size in hidden_sizes:
            layers.extend([
                nn.Linear(prev_size, hidden_size),
                nn.ReLU(),
                nn.Dropout(0.3)
            ])
            prev_size = hidden_size

        # Output layer
        layers.append(nn.Linear(prev_size, num_classes))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
```

nn.CrossEntropyLoss, already combines Softmax and the negative log-likelihood loss into one numerically stable operation.

```
Model architecture:
PredictiveMaintenanceDNN(
  (network): Sequential(
    (0): Linear(in_features=7, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=64, out_features=32, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.3, inplace=False)
    (9): Linear(in_features=32, out_features=6, bias=True)
  )
)
Epoch [10/50], Loss: 0.1041, Accuracy: 96.86%
Epoch [20/50], Loss: 0.0865, Accuracy: 97.39%
Epoch [30/50], Loss: 0.0727, Accuracy: 97.81%
Epoch [40/50], Loss: 0.0715, Accuracy: 97.90%
Epoch [50/50], Loss: 0.0639, Accuracy: 98.10%
```

This layer transforms the input from $\mathbb{R}^7$ to $\mathbb{R}^{128}$.

- **Linear Transformation:** The input $\mathbf{x}$ is multiplied by a weight matrix $\mathbf{W}_1 \in \mathbb{R}^{128\times 7}$ and a bias vector $\mathbf{b}_1 \in \mathbb{R}^{128}$ is added.

$$\mathbf{a}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$

- **Activation:** The result is passed through the Rectified Linear Unit (ReLU) function.

$$\mathbf{h}_1 = \max(0, \mathbf{a}_1)$$

- **Regularization (Dropout):** A portion ($p = 0.3$) of the resulting vector $\mathbf{h}_1$ is randomly set to zero during training to prevent overfitting.

$$\mathbf{h}_1' = \mathbf{h}_1 \odot \mathbf{M}_1, \quad \text{where } M_{1,i} \sim \text{Bernoulli}(1-p)$$

  - $\mathbf{h}_1' \in \mathbb{R}^{128}$ is the output of the first hidden block.

5. Output Layer (Layer 9)

This final layer produces the output logits $\mathbf{z} \in \mathbb{R}^6$ (raw scores for the 6 classes).

- **Linear Transformation:**

$$\mathbf{z} = \mathbf{W}_4\mathbf{h}_3' + \mathbf{b}_4, \quad \text{where } \mathbf{W}_4 \in \mathbb{R}^{6\times 32}$$

Summary of Architecture

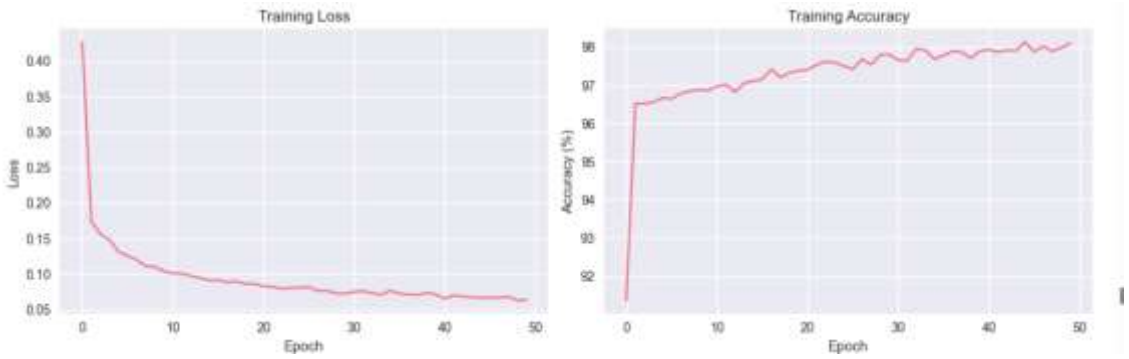
This is a **4-layer Deep Neural Network** (3 hidden layers + 1 output layer) characterized by the sequence:

$$\mathbb{R}^7 \xrightarrow[\text{ReLU, Dropout}]{\text{Linear}} \mathbb{R}^{128} \xrightarrow[\text{ReLU, Dropout}]{\text{Linear}} \mathbb{R}^{64} \xrightarrow[\text{ReLU, Dropout}]{\text{Linear}} \mathbb{R}^{32} \xrightarrow{\text{Linear}} \mathbb{R}^6$$

The final output $\mathbf{z}$ is then used in conjunction with the **Softmax** function (implicitly handled by PyTorch's `nn.CrossEntropyLoss`) to determine the class probabilities $\hat{\mathbf{y}}$ for multi-class classification.

```
Classification Report:
                          precision    recall  f1-score   support

Heat Dissipation Failure       0.70      0.73      0.71        22
              No Failure       0.98      1.00      0.99      1930
      Overstrain Failure       1.00      0.38      0.55        16
           Power Failure       0.89      0.84      0.86        19
         Random Failures       0.00      0.00      0.00         4
       Tool Wear Failure       0.00      0.00      0.00         9

                accuracy                           0.98      2000
               macro avg       0.59      0.49      0.52      2000
            weighted avg       0.97      0.98      0.98      2000
```

These poor results on the rarest classes, despite the use of SMOTE during training, suggest that the synthetic data generated for these minority classes may not be representative or that the model requires further optimization (e.g., deeper architecture, advanced regularization, or a different oversampling strategy) to successfully capture their complex features.



Since the training accuracy is very high (98\%) and the training loss is very low (0.06), there is a **high risk of overfitting**, especially given the use of SMOTE augmentation and the poor performance seen on the minority classes in the Classification Report (F1-scores of 0.00).

3.5 Hyperparameter Tuning

- **GridSearch configuration**

```python
# MLP and param grid (tuned for efficiency)
mlp = MLPClassifier(random_state=42)
param_grid = {
    'hidden_layer_sizes': [(50,), (100,)],
    'activation': ['relu'],
    'solver': ['adam'],
    'alpha': [0.0001, 0.001],
    'learning_rate_init': [0.001, 0.01],
    'max_iter': [100, 200]
}

# GridSearch
grid_search = GridSearchCV(mlp, param_grid, cv=3, scoring='accuracy', n_jobs=-1, verbose=1)
grid_search.fit(X, y)

# Output
print("Best parameters:", grid_search.best_params_)
print("Best CV score:", grid_search.best_score_)
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X)
print("\nClassification Report:\n", classification_report(y, y_pred, target_names=le.classes_))
```

- **Optimal parameters selection**

```
X shape: (10000, 7)
Unique classes: ['Heat Dissipation Failure' 'No Failure' 'Overstrain Failure'
 'Power Failure' 'Random Failures' 'Tool Wear Failure']
Fitting 3 folds for each of 16 candidates, totalling 48 fits
Best parameters: {'activation': 'relu', 'alpha': 0.001, 'hidden_layer_sizes': (50,), 'learning_rate_init': 0.001, 'max_iter': 100, 'solver': 'adam'}
Best CV score: 0.9088935288249881

Classification Report:
                       precision    recall  f1-score   support

Heat Dissipation Failure       0.95      0.65      0.77       112
              No Failure       0.99      1.00      0.99      9652
      Overstrain Failure       0.84      0.63      0.72        78
         Power Failure         0.93      0.80      0.86        95
      Random Failures          0.00      0.00      0.00        18
      Tool Wear Failure        0.00      0.00      0.00        45

              accuracy                           0.98     10000
             macro avg         0.62      0.51      0.56     10000
          weighted avg         0.98      0.98      0.98     10000
```

The optimal configuration found by GridSearchCV that resulted in a Cross-Validation score of 90.88%

Efficiency: The model with the single hidden layer of 50 neurons is highly computationally efficient and achieved a high overall Accuracy (98%).

Effectiveness (Failure Detection): Despite being the "best" choice according to the overall accuracy metric used by GridSearch, this simple architecture was ineffective at detecting the rarest failure types (Random Failures and Tool Wear Failure), where its F1-Score was 0.00

Model Architecture: 1 hidden layer with 50 neurons — simplest and most efficient setup tested.
Optimizer: Adam — fast and reliable convergence.
L2 Regularization: $\alpha = 0.001$ — effective at reducing overfitting without limiting learning.
Learning Rate: 0.001 — stable and commonly used starting point.

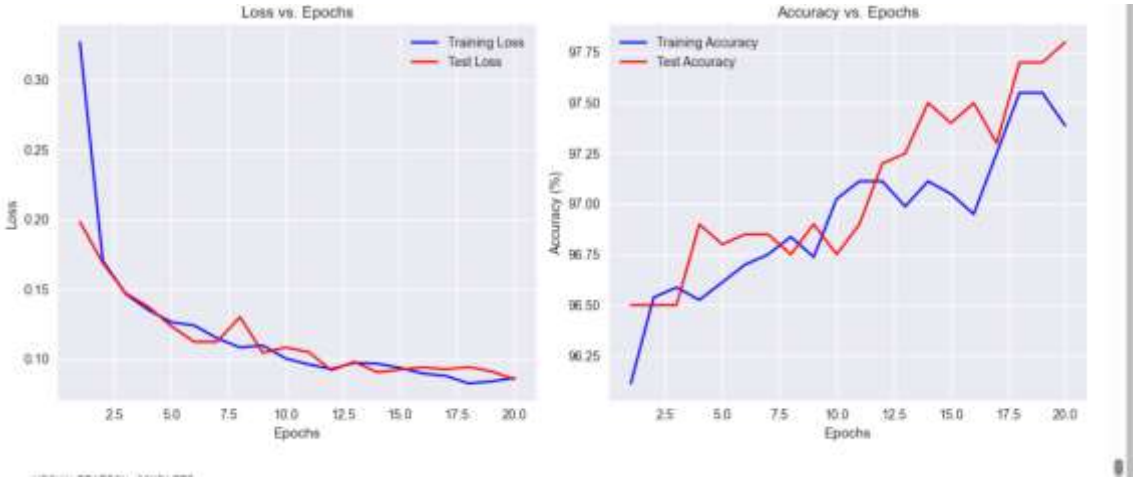
Max Epochs: 100 — model converged well within this range.
Activation: ReLU — efficient and avoids vanishing gradients.

Recall: 0.63A significant improvement in recall from 0.38, meaning the optimized model now detects more of these failures.

collecting more real data would be necessary to achieve full effectiveness.

3.6 Model Evaluation

- **Loss vs Epochs visualization & Accuracy vs Epochs visualization**



- **Results interpretation**

Left graph : The synchronized decrease in both losses confirms that the model is learning without suffering from significant overfitting.

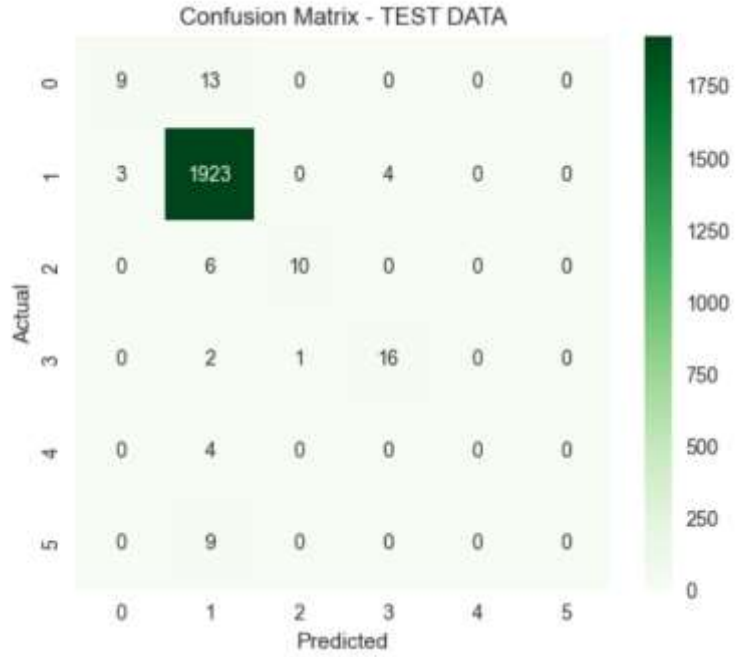
The training loss starts higher but quickly drops, showing fast early learning.

Right graph: Test accuracy often slightly outperforms training accuracy, which is normal with regularization (L2) and suggests the model is not memorizing the training data.

The model steadily improves and reaches high and stable accuracy, showing strong generalization.

3.7 Performance Metrics

- **Accuracy, Sensitivity, F1-Score**
- **Confusion matrix**



The confusion matrices show that while the model is good at recognizing normal machine behaviour, it struggles with the critical task: identifying failure types.
This highlights the need for regularization, data balancing techniques (SMOTE, class weights), and improved architecture to ensure reliability in failure detection.

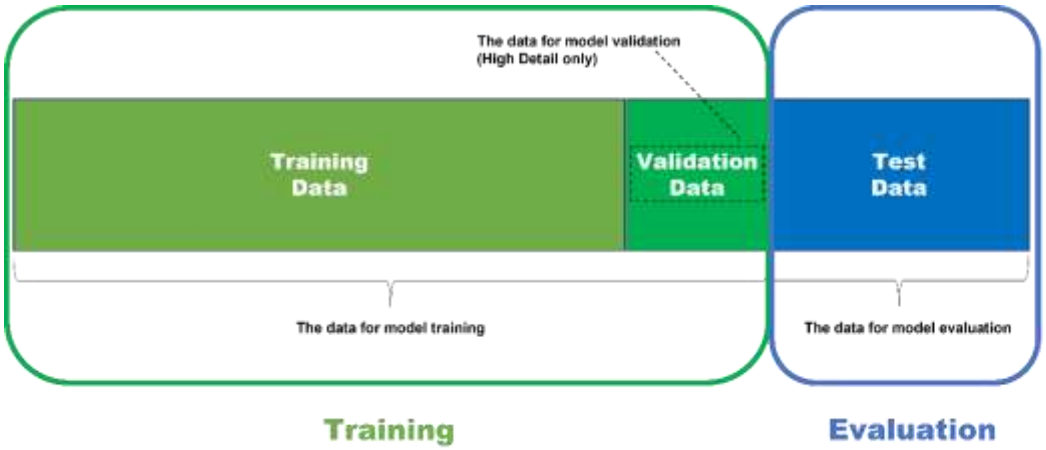
```
=== TRAIN METRICS ===
Accuracy: 0.941125
Precision: 0.16457828693017204
Recall: 0.16684387737019316
F1: 0.16567843539104873

=== TEST METRICS ===
Accuracy: 0.979
Precision: 0.573619563029374
Recall: 0.47876153820727047
F1: 0.5133528909186329
```

Although the model achieves high accuracy, this is mainly due to the overwhelmingly large number of "No Failure" samples. When evaluating using macro-averaged metrics—which treat all failure types equally—the performance is much lower.
This indicates that the model does not correctly detect rare failure types, leading to poor recall and F1-scores for those classes.
Additional techniques such as class weighting, oversampling, or more balanced training data are required to improve failure-type prediction performance.

- **Training vs Test comparison**

Difference Between Training and Testing the Model

- Training:
  The model learns patterns from labeled data by adjusting its internal weights using an optimization algorithm. This phase minimizes the loss and improves the model's ability to recognize relationships in the data.
- Testing:
  The trained model is evaluated on unseen data to measure how well it generalizes. No learning or weight updates occur here; only performance metrics such as accuracy, recall, and F1-score are calculated.




```
Using device: cpu

=== TRAINING STARTED ===
Epoch 1/20 | Loss: 0.3583
Epoch 2/20 | Loss: 0.1608
Epoch 3/20 | Loss: 0.1492
Epoch 4/20 | Loss: 0.1353
Epoch 5/20 | Loss: 0.1214
Epoch 6/20 | Loss: 0.1138
Epoch 7/20 | Loss: 0.1148
Epoch 8/20 | Loss: 0.1069
Epoch 9/20 | Loss: 0.1085
Epoch 10/20 | Loss: 0.1013
Epoch 11/20 | Loss: 0.0981
Epoch 12/20 | Loss: 0.0932
Epoch 13/20 | Loss: 0.0929
Epoch 14/20 | Loss: 0.0961
Epoch 15/20 | Loss: 0.0936
Epoch 16/20 | Loss: 0.0880
Epoch 17/20 | Loss: 0.0918
Epoch 18/20 | Loss: 0.0847
Epoch 19/20 | Loss: 0.0858
Epoch 20/20 | Loss: 0.0880
=== TRAINING COMPLETE ===

=== TRAINING METRICS (Macro Average) ===
Accuracy: 0.9463
Precision: 0.1684
Recall (Sensitivity): 0.1670
F1-Score: 0.1671

=== TESTING METRICS (Macro Average) ===
Accuracy: 0.9755
Precision: 0.5879
Recall (Sensitivity): 0.3756
F1-Score: 0.4420

=== WEIGHTED METRICS (Handles Class Imbalance) ===
Train - Precision: 0.9322 | Recall: 0.9463 | F1: 0.9391
Test  - Precision: 0.9672 | Recall: 0.9755 | F1: 0.9690
```

```
=== CLASSIFICATION REPORT (TRAIN) ===
                           precision    recall  f1-score   support

Heat Dissipation Failure       0.05      0.02      0.03        90
              No Failure       0.97      0.98      0.97      7722
      Overstrain Failure       0.00      0.00      0.00        62
           Power Failure       0.00      0.00      0.00        76
         Random Failures       0.00      0.00      0.00        14
       Tool Wear Failure       0.00      0.00      0.00        36

                accuracy                           0.95      8000
               macro avg       0.17      0.17      0.17      8000
            weighted avg       0.93      0.95      0.94      8000


=== CLASSIFICATION REPORT (TEST) ===
                           precision    recall  f1-score   support

Heat Dissipation Failure       0.80      0.36      0.50        22
              No Failure       0.98      1.00      0.99      1930
      Overstrain Failure       0.83      0.31      0.45        16
           Power Failure       0.92      0.58      0.71        19
         Random Failures       0.00      0.00      0.00         4
       Tool Wear Failure       0.00      0.00      0.00         9

                accuracy                           0.98      2000
               macro avg       0.59      0.38      0.44      2000
            weighted avg       0.97      0.98      0.97      2000
```

Metrics Summary Table:

	Metric	Train	Test
0	Accuracy	0.9462	0.9755
1	Precision (Macro)	0.1684	0.5879
2	Recall (Macro)	0.1670	0.3756
3	F1 (Macro)	0.1671	0.4420

3.8 Regularization Techniques

- **Implementation and comparison**
- **Model performance analysis**

```
# Run experiments
baseline = train_and_evaluate(dropout_rate=0.3, weight_decay=1e-5, patience=None)
high_reg = train_and_evaluate(dropout_rate=0.5, weight_decay=0.001, patience=None)
batchnorm = train_and_evaluate(use_bn=True, dropout_rate=0.3, weight_decay=1e-5, patience=None)
early_stop = train_and_evaluate(dropout_rate=0.3, weight_decay=1e-5, patience=5)

# Results table
results_df = pd.DataFrame({
    'Model': ['Baseline (Dropout 0.3, L2 1e-5)', 'High Reg (Dropout 0.5, L2 1e-3)', 'BatchNorm + Dropout 0.3', 'Early Stopping (patience=5)'],
    'Accuracy': [baseline['accuracy'], high_reg['accuracy'], batchnorm['accuracy'], early_stop['accuracy']],
    'F1 Macro': [baseline['f1_macro'], high_reg['f1_macro'], batchnorm['f1_macro'], early_stop['f1_macro']],
    'Epochs Used': [baseline['epochs_used'], high_reg['epochs_used'], batchnorm['epochs_used'], early_stop['epochs_used']]
})
print(results_df.round(4))
```

```
                          Model  Accuracy  F1 Macro  Epochs Used
0  Baseline (Dropout 0.3, L2 1e-5)    0.9785    0.5246           50
1  High Reg (Dropout 0.5, L2 1e-3)    0.9760    0.4550           50
2          BatchNorm + Dropout 0.3    0.9770    0.4959           50
3      Early Stopping (patience=5)    0.9785    0.5156           50
```

1 Baseline Model (Dropout = 0.3, L2 = 1e-5) — ✔ GOOD

- Accuracy: 0.9785
- F1 Macro: 0.5246
- This configuration gave the best overall results.
- Balanced dropout + light L2 regularization worked effectively.
- Serves as the reference model.

Technique	Effect	Result
Baseline (Dropout 0.3 + L2 1e-5)	Best performance	GOOD
High Regularization (Dropout 0.5 + L2 1e-3)	Underfitting	✘ BAD
BatchNorm + Dropout	Slight improvement in stability, not accuracy	NEUTRAL
Early Stopping	Prevents overfitting but no improvement	✔ OKAY

4. Conclusion

- **Key learnings synthesis**
- **Comparative analysis**

The experiments show that regularization must be tuned carefully.
Too little regularization risks overfitting, while too much causes underfitting.
In this lab, a moderate level of regularization (Dropout 0.3 + L2 1e-5)

provided the best balance between accuracy and generalization, especially for an imbalanced dataset.

During this lab, several important concepts and practical skills were learned:

1. Understanding Train vs Test Workflow

- The **training set** is used to teach the model by adjusting weights.
- The **test set** is used **only for evaluation** to measure generalization.
- A model can show **high accuracy but poor recall** when classes are imbalanced.

2. Effect of Class Imbalance

- Our dataset contained a very large majority class ("No Failure"), which caused:
  - High accuracy
  - But low macro precision, recall, and F1 for rare failure types
- Metrics such as **macro F1** and confusion matrices are essential to reveal these issues.

3. Regularization Techniques

- **Dropout**
- **L2 weight decay**
- **Batch Normalization**
- **Early Stopping**

Each method helps control overfitting in different ways.

4. Using Metrics & Confusion Matrices Correctly

- Accuracy alone was misleading because the dataset is imbalanced.
- Macro metrics allowed us to evaluate the model equally across all classes.
- Confusion matrices showed exactly which classes the model struggles to detect.

5. Practical Skills Developed

- Implementing a DNN architecture in PyTorch
- Applying regularization methods and adjusting hyperparameters