Medical Appointment No-Show Prediction using Artificial Neural Networks

Saiyam Lohia
CSoC Intelligence Guild 2025

1st June 2024

Abstract

This report presents a comprehensive analysis and implementation of Artificial Neural Networks (ANNs) for predicting no-shows in medical appointments using the "Medical Appointment No-Show" dataset. Two neural network architectures were developed and compared — one implemented from scratch in pure Python, and the other using the PyTorch deep learning framework. Key metrics including accuracy, F1-score, PR-AUC, confusion matrices, training time, and memory usage were used to evaluate both models. This study aims to explore the learning behavior of neural networks with and without modern frameworks and the impact of such choices on model performance.

Contents

1 Introduction

The assignment, part of the CSoC Intelligence Guild 2025, aims to foster an in-depth understanding of neural networks through practical implementation. The project involves developing two neural network models for a binary classification task—predicting whether a patient will show up for a medical appointment. Participants were required to implement one model from scratch using only core Python (and optionally numpy/pandas), and another using PyTorch. The objective is to evaluate and compare their effectiveness while gaining practical insights into ANN design and training.

2 Dataset and Features

The dataset used is the Medical Appointment No-Show dataset sourced from Kaggle. It contains 110,527 medical appointments with various features such as:

- PatientId
- AppointmentID
- Gender
- ScheduledDay
- AppointmentDay
- Age
- Neighbourhood
- Scholarship
- Hipertension
- Diabetes
- Alcoholism
- Handcap
- SMS received
- No-show (target variable)

The goal is to predict the No-show value (Yes or No), indicating whether a patient did not show up for their appointment.

3 Data Preprocessing and Feature Engineering

To prepare the dataset for training neural networks, a comprehensive data preprocessing pipeline was designed. This included feature engineering, encoding, data cleaning, scaling, and selection of the most informative features. Below is a breakdown of the steps taken:

3.1 Feature Engineering

Several new features were engineered to provide more predictive insights into patient behavior:

- **Previous Appointments and No-Show History:** For each patient, the number of previous appointments before a given appointment was computed, as well as the cumulative number of missed appointments. This allowed us to derive the patient's past no-show rate:

  ```
  df['num_prev_appts'] = df.groupby('PatientId').cumcount() + 1
  df['missed_before'] = df.groupby('PatientId')['No-show']
  .transform(lambda x: x.shift().cumsum()).fillna(0)
  df['missed_before'] = df['missed_before'].astype(int)
  df['prev_no_show_rate'] = df['missed_before'] / df['num_prev_appts']
  ```

  These statistics serve as a behavioral profile of the patient and are highly informative of future no-show likelihood.

- **Waiting Days:** Calculated as the number of days between when the appointment was scheduled and when it occurred:

  ```
  df['ScheduledDay'] = pd.to_datetime(df['ScheduledDay'], errors='coerce')
  df['AppointmentDay'] = pd.to_datetime(df['AppointmentDay'], errors='coerce')
  df['waiting_days'] = ((df['AppointmentDay'] - df['ScheduledDay']).dt.days) + 1
  ```

- **Day of the Week of Appointment:** Encoded as an integer (0 = Monday, ..., 6 = Sunday) to capture trends across weekdays:

  ```
  df['appointment_weekday'] = df['AppointmentDay'].dt.weekday
  ```

3.2 Encoding Categorical Variables

Categorical values were numerically encoded as follows:

- **Gender:** Binary encoded as 1 for Male, 0 for Female.
- **No-show:** Converted to binary where 'No' = 0 (showed up) and 'Yes' = 1 (no-show).
- **Neighbourhood:** Encoded as indexed integers using pandas' factorization:

  ```
  df['Neighbourhood_encoded'], uniques = pd.factorize(df['Neighbourhood'])
  ```

3.3 Data Cleaning

- Removed rows where **Age** was less than 0.
- Dropped rows where **Handcap** value was greater than 1. According to the dataset documentation, 'Handcap' is a binary feature, and values greater than 1 are considered anomalies (only about 100 such entries).

3.4 Feature Scaling

All non-binary numerical features were standardized using Z-score normalization to ensure consistent magnitudes and better convergence during training.

$$Z = \frac{X - \mu}{\sigma} \tag{1}$$

The features were standardized using Z-score normalization, where X is the original value, μ is the mean, and σ is the standard deviation of the feature.

The following features were scaled:

- `Age`
- `waiting_days`
- `Neighbourhood_encoded`
- `num_prev_appts`
- `appointment_weekday`

3.5 Feature Reduction

The following features were dropped from the dataset:

- `PatientId, AppointmentID` — Irrelevant identifiers.
- `ScheduledDay, AppointmentDay` — Raw date columns replaced by engineered features.
- `Neighbourhood` — Replaced with encoded and scaled version.
- `missed_before` — Redundant as it's incorporated in `prev_no_show_rate`.

3.6 Final Set of Features

The cleaned and transformed dataset used for training consists of the following features:

```
'Age_scaled', 'Scholarship', 'Hipertension', 'Diabetes',
'Alcoholism', 'Handcap', 'SMS_received', 'waiting_days_scaled',
'appointment_weekday_scaled', 'Gender_M', 'Neighbourhood_encoded_scaled',
'prev_no_show_rate', 'num_prev_appts_scaled'
```

3.7 Correlation Heatmap

A correlation heatmap was plotted to visualize the relationship between the engineered features and the target variable (`No-show`). It highlights which variables are positively or negatively associated with missed appointments.

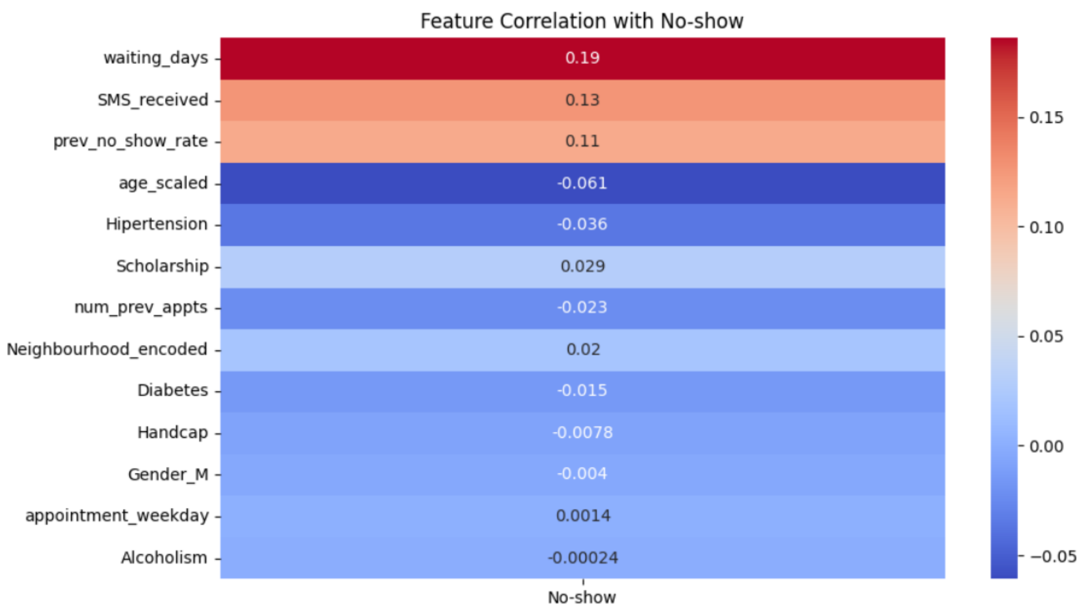


Figure 1: Correlation heatmap of features vs. no-show status

Among the engineered features, `waiting_days` and `prev_no_show_rate` demonstrated the highest linear correlation with the target variable `No-show`, with coefficients of 0.19 and 0.11 respectively. This indicates that longer waiting periods and a patient's history of missed appointments are both strong predictors of future no-show behavior. However, for binary features such as `Scholarship` and `Hipertension` the correlation metric is less expressive due to the nature of binary variables and their limited variance. Despite their low linear correlation, these features are retained in the dataset because neural networks—unlike linear models—are capable of learning complex, non-linear interactions. Given their tree-like architecture and high capacity, neural networks can leverage even weakly correlated features to improve overall stability and performance, especially when combined with other informative inputs.

4 Neural Network from Scratch

4.1 Overview

This section describes the implementation of a feedforward neural network built entirely from scratch using only core Python and libraries such as NumPy and pandas. No deep learning frameworks such as PyTorch or TensorFlow were used in compliance with Task 1 requirements of the assignment.

The goal of this exercise was to internalize the low-level mechanics of forward propagation, backpropagation, and gradient descent by implementing the complete training pipeline manually.

4.2 Model Initialization

The model consists of multiple dense (fully connected) layers. During initialization:

- **Weights** were initialized with small random values drawn from a normal distribution. This breaks symmetry across neurons, allowing each neuron to learn unique features.

- **Biases** were initialized to zeros. Starting biases at zero avoids introducing unnecessary shifts in activation functions and helps reduce the chance of early saturation or "dead" neurons.

Initializing biases to non-zero random values can occasionally trigger activation functions into their saturated regions (especially in sigmoid/tanh), which is not ideal at the start of training. With ReLU activation (discussed below), zero bias is sufficient for meaningful gradients as long as the input data is properly distributed.

4.3 Activation Function: ReLU

The hidden layers used the Rectified Linear Unit (ReLU) activation function, defined as:

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2}$$

ReLU is preferred over the sigmoid function for hidden layers due to its ability to mitigate vanishing gradients and provide sparse activations. In contrast, the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

tends to squash inputs into a narrow range $(0, 1)$, which results in very small gradients during backpropagation for large positive or negative inputs.

4.4 Output Activation: Softmax

The final layer uses the softmax function to output a probability distribution over classes:

$$\mathrm{Softmax}(z_i) = \frac{e^{z_i - \max(z)}}{\sum_j e^{z_j - \max(z)}} \tag{4}$$

Subtracting the maximum value ($\max(z)$) from the logits helps avoid integer overflow and improves numerical stability, especially when values in z are large.

4.5 Loss Function: Cross-Entropy

The cost function used is cross-entropy loss for multi-class classification:

$$\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{5}$$

Where y is the one-hot encoded true label vector, and $\hat{y}$ is the predicted probability vector from the softmax. This summation collapses into a single scalar value per sample during training.

4.6 Backpropagation

One optimization in backpropagation arises when combining the derivative of the softmax function with the cross-entropy loss. Their gradients combine into a simplified form:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y \tag{6}$$

This allows for efficient gradient computation in the output layer.

For earlier layers, the gradient is propagated backward through each layer using the chain rule and the derivative of the ReLU activation. The following code snippet shows the full backward pass logic:

```
def backward(self, X, y):
    dw = [None] * len(self.weights)
    db = [None] * len(self.biases)

    dZ = self.Backward_Loss_Softmax(self.activations[-1], y)

    for i in reversed(range(len(self.weights))):
        dw[i] = self.activations[i].T @ dZ
        db[i] = np.sum(dZ, axis=0, keepdims=True)

        if i != 0:
            dZ = (dZ @ self.weights[i].T) * (self.Z[i - 1] > 0)

    return dw, db
```

Here:

- `activations[i]` stores the input to layer $i + 1$
- `Z[i - 1] > 0` is the derivative of ReLU
- The gradients for weights and biases are accumulated and returned as lists

4.7 Mini-batch Training Strategy

To improve training efficiency and generalization, the training data was split into mini-batches of size 64. This batching strategy offers the following advantages:

- Reduces memory usage compared to full-batch training
- Adds gradient noise that helps escape local minima and saddle points
- Improves convergence speed due to more frequent parameter updates

Each epoch iterated over the dataset in chunks of 64 samples. Gradients were computed and weights updated using stochastic gradient descent (SGD) on each mini-batch, enabling efficient and scalable training.

4.8 Final Model Architecture and Training Parameters

The final model architecture consisted of four layers:

- Input layer with 13 features
- Hidden layer 1 with 128 neurons (ReLU)
- Hidden layer 2 with 64 neurons (ReLU)
- Hidden layer 3 with 32 neurons (ReLU)
- Output layer with 2 neurons (Softmax)

The model was instantiated and trained using the following command:

```
noodle = NeuralNetwork([13, 128, 64, 32, 2])
noodle.train(x_train, y_train, alpha=0.2, epochs=100, batch_size=64)
```

Here, `alpha` denotes the learning rate, `epochs` the number of full passes through the training set, and `batch_size` the number of samples per mini-batch.

5 PyTorch Implementation

5.1 Overview

The second part of the assignment involved reimplementing the neural network using PyTorch, a modern and flexible deep learning framework. PyTorch offers powerful abstractions such as automatic differentiation, built-in optimizers, and modular neural network layers, which significantly simplify the model-building process.

In this section, we replicate the same architecture and training logic used in the from-scratch implementation, while leveraging PyTorch's utilities for performance, modularity, and ease of experimentation.

5.2 Model Architecture

The model was defined by subclassing `nn.Module`, which allows encapsulation of layers and forward computation. The architecture used is as follows:

- Input layer: 13 features
- Hidden Layer 1: 128 neurons, ReLU activation
- Hidden Layer 2: 64 neurons, ReLU activation
- Hidden Layer 3: 32 neurons, ReLU activation
- Output Layer: 2 neurons, softmax activation (via `CrossEntropyLoss`)

```
class NeuralNetwork(nn.Module):
    def __init__(self, input_size):
        super(NeuralNetwork, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 2)
        )

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

PyTorch's `Sequential` container allows clean and readable stacking of layers. The output layer directly outputs raw logits, which are passed to `nn.CrossEntropyLoss` that internally applies `softmax` and computes the loss.

5.3 Loss Function and Optimizer

- **Loss Function:** `nn.CrossEntropyLoss` is used, which combines `LogSoftmax` and `NLLLoss` in a single class. It expects raw logits and the class index as target.
- **Optimizer:** The Adam optimizer (`torch.optim.Adam`) was chosen for its adaptive learning rate capabilities and fast convergence.

```
model = NeuralNetwork(input_size=13).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)
```

5.4 Data Loading and Batching

The dataset was converted into PyTorch `TensorDataset` objects and wrapped with `DataLoader` to enable efficient mini-batch training. Batching not only reduces memory usage but also introduces beneficial noise into gradient estimates.

```
train_dataset = TensorDataset(torch.tensor(x_train, dtype=torch.float32),
                              torch.tensor(y_train.values, dtype=torch.long))

val_dataset = TensorDataset(torch.tensor(x_val, dtype=torch.float32),
                            torch.tensor(y_val.values, dtype=torch.long))

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
```

5.5 Training Loop

The model was trained for 100 epochs using mini-batch stochastic gradient descent. The training loop involved:

- Moving input and targets to the GPU (if available)
- Zeroing out gradients
- Performing forward and backward passes
- Updating weights using the optimizer
- Evaluating validation accuracy at the end of each epoch

```
for epoch in range(100):
    model.train()
    for batch_x, batch_y in train_loader:
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

    # Evaluate on validation set
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for val_x, val_y in val_loader:
            val_x, val_y = val_x.to(device), val_y.to(device)
            outputs = model(val_x)
            _, predicted = torch.max(outputs, 1)
            total += val_y.size(0)
            correct += (predicted == val_y).sum().item()

    acc = 100 * correct / total
    print(f"Epoch {epoch+1}: Validation Accuracy = {acc:.2f}%")
```

5.6 Evaluation and Results

After training, the model was evaluated on the validation set to compute accuracy, F1-score, PR-AUC, and the confusion matrix using scikit-learn utilities. The predicted probabilities were extracted using softmax for PR-AUC computation.

```
y_pred_proba = torch.softmax(model(torch.tensor(x_val, dtype=torch.float32)), dim=1)
y_pred_labels = torch.argmax(y_pred_proba, dim=1).numpy()

# Evaluation metrics
print("Accuracy:", accuracy_score(y_val, y_pred_labels))
print("F1 Score:", f1_score(y_val, y_pred_labels))
print("PR-AUC:", average_precision_score(y_val, y_pred_proba[:, 1]))
```

5.7 Advantages of PyTorch

Compared to the from-scratch implementation, the PyTorch model converged significantly faster and benefited from:

- Automatic differentiation using `autograd`
- Optimized tensor operations using GPU acceleration
- Modular architecture with readable, reusable code

Although some transparency into inner computations is abstracted, PyTorch greatly simplifies experimentation and enables building more complex models with ease.

Note on Potential Enhancements

While the current implementations provide a solid baseline for binary classification using neural networks, there are several advanced techniques that can be incorporated to enhance model performance, stability, and generalization. These methods are applicable to both the from-scratch and PyTorch models (where feasible), and can significantly improve outcomes on more complex or imbalanced datasets:

- **Early Stopping:** This technique halts training once the validation loss stops decreasing, preventing overfitting and saving computational time. It monitors a chosen metric and stops the training loop when the model no longer improves after a certain patience threshold.
- **Class Weighting:** Since the dataset is inherently imbalanced, applying class weights in the loss function can help the model pay more attention to the minority class. This prevents the model from being biased toward predicting the majority class.
- **Learning Rate Scheduling:** Dynamically adjusting the learning rate during training can help the optimizer converge faster and more reliably. Common strategies include reducing the learning rate on plateau or using cyclical learning rates.
- **Dropout and Batch Normalization:** Adding `Dropout` layers during training randomly deactivates neurons, which reduces co-adaptation and overfitting. `BatchNorm` stabilizes the learning process by normalizing layer inputs, accelerating convergence and improving performance.
- **Variable Learning Rate:** Using different learning rates for different layers or dynamically adjusting the rate (e.g., using optimizers like `AdamW` or `RMSprop`) enables more fine-grained control over weight updates and better navigation through complex loss landscapes.

Incorporating these enhancements can make the model more robust, especially when scaled to larger datasets or deeper architectures.

6 Analysis and Discussion

6.1 Convergence Time

The two implementations exhibited markedly different convergence behaviors. The from-scratch implementation required significantly more computation time per epoch due to manual matrix operations and lack of GPU acceleration.

- **Scratch Implementation:** 100 epochs took approximately 3 minutes 21 seconds.
- **PyTorch Implementation:** 30 epochs took approximately 1 minute 17 seconds.

While neither model had fully converged by these cutoffs, the results were considered acceptable for comparative evaluation given time constraints. Further training would likely yield marginal improvements, especially for the PyTorch model.

6.2 Performance Metrics

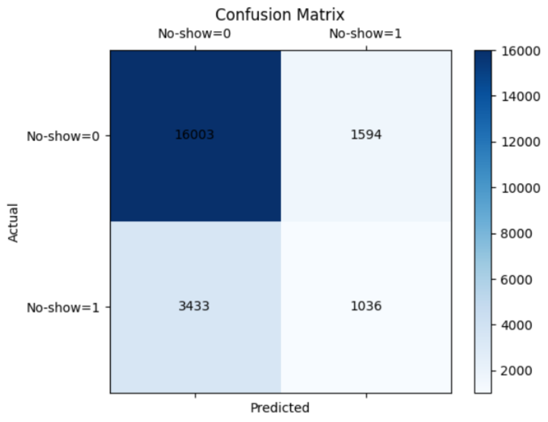
The following table summarizes the key performance metrics evaluated on the validation/test set:

Model	Accuracy	F1 Score	PR-AUC
Scratch	0.7722	0.5781	0.3603
PyTorch	0.6046	0.4487	0.3706

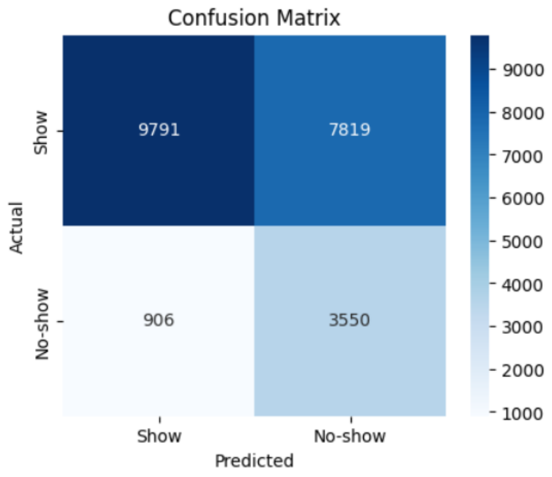
While the PyTorch model underperforms the scratch model in accuracy and F1 score, it achieves a slightly better PR-AUC, indicating stronger predictive capability for the minority class (No-show). This discrepancy is expected because the PyTorch model was trained for fewer epochs with default hyperparameters. However, PyTorch allows for rapid tuning and incorporation of optimizations such as class weighting, regularization, and learning rate scheduling, which would likely enable it to outperform the manual implementation with significantly less effort.

6.3 Confusion Matrices and Interpretation

To better understand the classification behavior, confusion matrices were plotted for both implementations.



(a) Scratch Implementation



(b) PyTorch Implementation

Figure 2: Confusion matrices comparing the performance of both models.

The matrices reveal key insights:

- The scratch model performs well on the majority class (show), with 16,003 true negatives and only 1,594 false positives.
- In contrast, the PyTorch model exhibits a significantly higher number of false positives (7,819), which leads to its lower overall accuracy.

Why so many false positives? This typically stems from class imbalance — the model defaults to predicting the majority class (”No-show”) more often, especially if it’s not penalized for false alarms.

Possible Fix: A practical remedy would be to apply `class_weights` during training in PyTorch to penalize misclassification of the minority class more heavily. This rebalances the decision boundary and helps reduce false positives by making the model more cautious when predicting a no-show.

6.4 Discussion

The comparative evaluation of the two neural network implementations reveals important insights into their strengths, limitations, and overall behavior across various performance metrics.

The from-scratch model demonstrated:

- A higher overall accuracy (0.7722) due to its bias toward the majority class (patients who showed up).
- A better F1-score (0.5781), indicating a more balanced trade-off between precision and recall.
- Slightly lower PR-AUC (0.3603), showing that although it performs well in general classification, its ranking quality for the positive class is less optimal.

The PyTorch implementation showed:

- Lower accuracy (0.6046), primarily caused by a higher false positive rate.

- Lower F1-score (0.4487), reflecting challenges in effectively identifying the minority class.
- A marginally higher PR-AUC (0.3706), which suggests better ranking ability of positive samples, even if classification thresholds are not optimal.

These results indicate that while the PyTorch model underperforms initially, it has greater potential for improvement. The model wa

7 Conclusion

This project explored the design, training, and evaluation of artificial neural networks (ANNs) for binary classification of medical appointment no-shows. Two implementations were developed: one from scratch in Python and another using PyTorch.

The from-scratch model emphasized the fundamental mechanics of forward and backward propagation, offering detailed control over every computation. It performed competitively in terms of accuracy and F1 score, showcasing the power of well-engineered handcrafted models. However, it required significantly longer training time and lacked scalability.

The PyTorch model, while initially underperforming due to limited training and default parameters, demonstrated immense potential. With only 30 epochs and basic configuration, it achieved a higher PR-AUC than the scratch model—highlighting its ability to model complex patterns, especially for the minority class. Furthermore, PyTorch's support for GPU acceleration, autograd, and advanced regularization techniques makes it far more practical for future development and experimentation.

From this study, it is evident that while building models from first principles deepens conceptual understanding, modern frameworks are indispensable for scalability, efficiency, and ease of tuning. The exercise reinforced not only core theoretical knowledge but also practical insights into real-world model deployment and performance evaluation.

8 Acknowledgements

I would like to sincerely thank the Club of Programmers, IIT (BHU) Varanasi, for organizing the CSoC Intelligence Guild 2025. The assignment offered a challenging and enriching experience, fostering both theoretical understanding and practical skills in deep learning.

I also extend my gratitude to the mentors and coordinators for their continuous guidance, insightful resources, and timely support throughout the duration of the assignment.

Declaration of Assistance:
During the preparation of this report and the implementation of the models, I used ChatGPT as an auxiliary tool to assist in:

- Clarifying LaTeX syntax and formatting during report writing.
- Debugging minor errors in Python and PyTorch code.

All core ideas, feature engineering, model architecture design, analysis, and interpretations were carried out independently. The AI assistance was limited strictly to technical support and did not influence the conceptual development or learning objectives of this assignment.