Mini Project Report

Your Name

1 Library Imports and Dataset Download

The first step in this project involves importing the required libraries and downloading the dataset using the `gdown` module. Below is the Python code used:

```python
# Install and download dataset from Google Drive
!pip install --upgrade --no-cache-dir gdown
!gdown 1A7NRguAV3PZdxK6zsDDSQa1sqwz9IfU7

# Data handling and visualization
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Data preprocessing and splitting
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# PyTorch modules for neural network
import torch
import torch.nn as nn
import torch.optim as optim

# Evaluation metrics
from sklearn.metrics import r2_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

Listing 1: Downloading Dataset and Importing Libraries

Explanation

- `gdown` is used to download the dataset directly from Google Drive using its file ID.
- Libraries such as `pandas`, `matplotlib`, and `seaborn` are used for data manipulation and visualization.
- `sklearn` provides tools for scaling, splitting, and evaluating models.
- `torch` is used for building and training deep learning models.

Output

This section does not produce any visual output but sets up all necessary dependencies for further development.

2 Data Loading and Initial Exploration

The dataset used in this project contains records related to graduate admission. Below is the code used to load and inspect the dataset.

```python
# Load dataset
df = pd.read_csv("Admission_Predict.csv")

# Display structure and data types
df.info()

# Display summary statistics
df.describe()

# Display first few rows
df.head()
```

Listing 2: Loading and Exploring the Dataset

Explanation

- `pd.read`$_c sv(...) loadsthedatasetintoaDataFrame.$`df.info()`$providesmetadata : columnnames, datatypes, non-nullcounts, andmemoryusage.$
- `df.describe()` generates summary statistics (mean, std, min, max, etc.) for numerical columns.
- `df.head()` shows the first five entries of the dataset for an initial look at the data.

DataFrame Info Output

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 9 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Serial No.         400 non-null    int64
 1   GRE Score          400 non-null    int64
 2   TOEFL Score        400 non-null    int64
 3   University Rating  400 non-null    int64
 4   SOP                400 non-null    float64
 5   LOR                400 non-null    float64
 6   CGPA               400 non-null    float64
 7   Research           400 non-null    int64
 8   Chance of Admit    400 non-null    float64
dtypes: float64(4), int64(5)
memory usage: 28.3 KB
```

Sample Data (First 5 Rows)

Serial No.	GRE	TOEFL	Univ Rating	SOP	LOR	CGPA	Res.	Admit Chance
1	337	118	4	4.5	4.5	9.65	1	0.92
2	324	107	4	4.0	4.5	8.87	1	0.76
3	316	104	3	3.0	3.5	8.00	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.80
5	314	103	2	2.0	3.0	8.21	0	0.65

Table 1: First 5 rows of the dataset

3 Feature Correlation Analysis

To understand the relationships between different features and the target variable (`Chance of Admit`), a correlation matrix was computed and visualized using a heatmap.

```python
# Compute correlation matrix for numerical features
correlation_matrix = df.corr(numeric_only=True)

# Plotting the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title("Correlation Matrix of Features and Chance of Admit")
plt.tight_layout()
plt.show()
```

Listing 3: Computing and Plotting the Correlation Matrix

Explanation

- `df.corr(numeric_only=True)` computes pairwise Pearson correlations between all numeric columns.
- `sns.heatmap(...)` visualizes these correlations as a heatmap.
- Strong correlations (positive or negative) help identify important features for prediction.

Correlation Heatmap

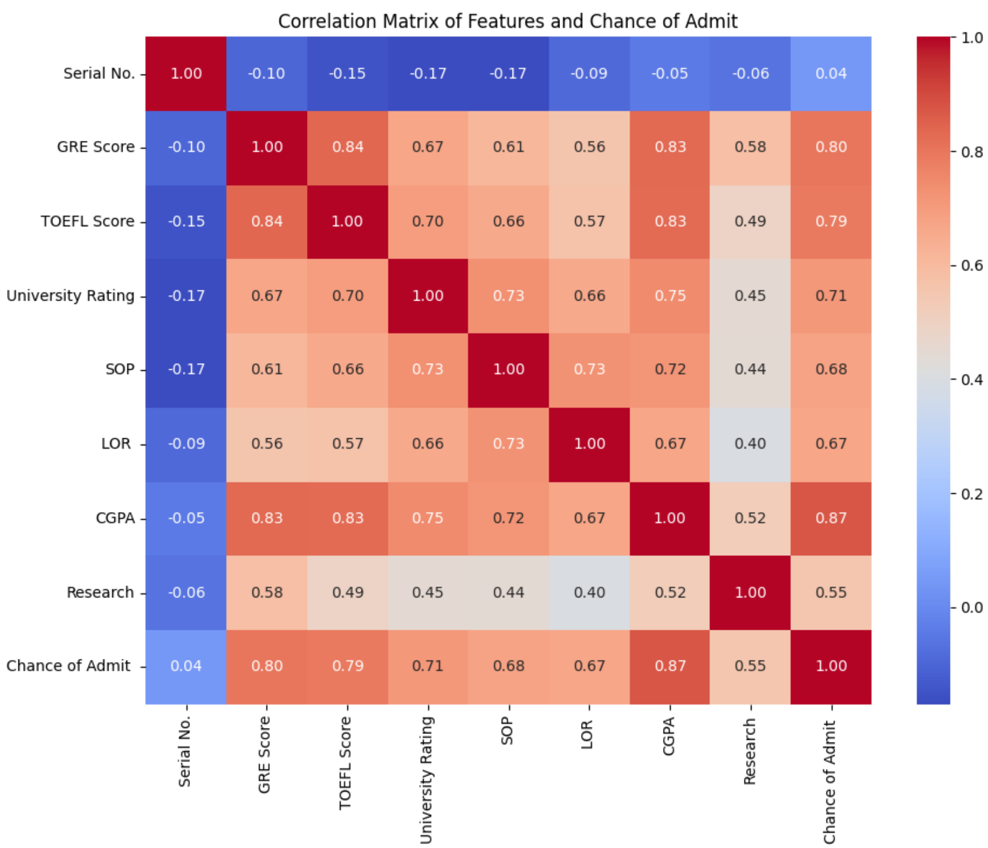

Figure 1: Heatmap showing correlation between features and Chance of Admit

4 Feature-Wise Correlation with Admission Chances

After computing the overall correlation matrix, we further analyzed the individual correlation of each feature with the target variable `Chance of Admit`.

```python
# Ensure exact match for column names
# Compute the correlation matrix
correlation_matrix = df.corr(numeric_only=True)

# Extract correlations with "Chance of Admit"
correlation_with_admit = correlation_matrix.loc[:, "Chance of Admit "].drop("Chance of Admit ")

# Plotting the correlations
plt.figure(figsize=(8, 5))
sns.barplot(x=correlation_with_admit.values, y=correlation_with_admit.index, palette="coolwarm")
plt.title("Correlation of Each Feature with Chance of Admit")
plt.xlabel("Correlation Coefficient")
plt.ylabel("Feature")
plt.grid(True, axis='x')
plt.tight_layout()
plt.show()
```

Listing 4: Bar Plot of Feature Correlation with Chance of Admit

Explanation

- This bar plot reveals the strength and direction of correlation between each input feature and the target.
- Features like `CGPA` and `GRE Score` show a strong positive correlation with `Chance of Admit`.
- The analysis helps in feature selection and understanding feature importance.

Feature Correlation Bar Plot

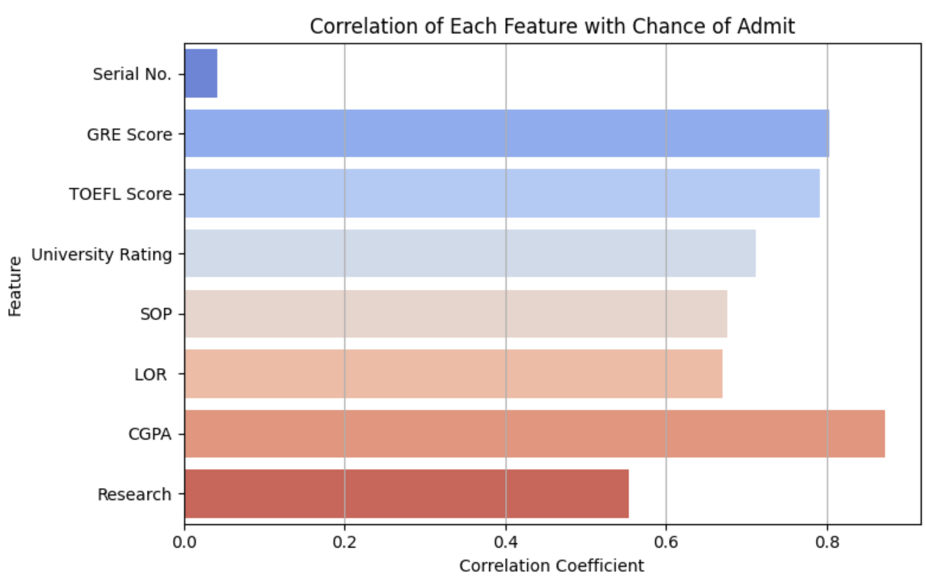

Figure 2: Bar plot of correlations between each feature and Chance of Admit

5 Data Splitting and Scaling

To train and evaluate our model properly, we split the dataset into training and testing subsets. We also scaled the input features to the range $[0, 1]$ using Min-Max normalization.

```python
# Step 1: Split data into train/test (85% train, 15% test)
X = df.drop(columns=["Chance of Admit "])
y = df["Chance of Admit "]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

# Step 2: Scale features to [0, 1] range
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Show scaled training data shape
X_train_scaled.shape, X_test_scaled.shape
```

Listing 5: Train/Test Split and Feature Scaling

Explanation

- The target column `Chance of Admit` was separated from the features.
- The data was split into 85% training and 15% testing using `train_test_split()`.
- `MinMaxScaler` scaled each feature to a $[0, 1]$ range to improve model convergence and performance.

Output

```
((340, 8), (60, 8))
```

This indicates that the training data contains 340 samples and the testing data contains 60 samples, each with 8 features.

6 Model Development and Training

Converting Data to PyTorch Tensors

Before training the model, the NumPy arrays for the training data were converted into PyTorch tensors. The feature matrix `X_train_scaled` was cast to `torch.float32`, and the target values `y_train` were reshaped to a column vector.

```python
X_tensor = torch.tensor(X_train_scaled, dtype=torch.float32)
y_tensor = torch.tensor(y_train.values, dtype=torch.float32).view(-1, 1)
```

Listing 6: Convert Data to PyTorch Tensors

To evaluate performance during training, we split 10% of the training data into a validation set:

```python
X_train_tensor, X_val_tensor, y_train_tensor, y_val_tensor = train_test_split(
    X_tensor, y_tensor, test_size=0.10, random_state=42
)
```

Listing 7: Train/Validation Split

Defining the Neural Network Model

A simple feedforward neural network (Multi-Layer Perceptron, or MLP) was defined using PyTorch. The model includes one hidden layer with 4 neurons and ReLU activation, followed by an output layer with a single neuron (for regression).

```python
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(X_tensor.shape[1], 4),
            nn.ReLU(),
            nn.Linear(4, 1)
        )

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

Listing 8: Simple MLP Architecture

Training Procedure

The training was handled by a function `train_model()` that takes the model, data, and number of epochs as input. It uses Mean Squared Error (MSE) as the loss function and the Adam optimizer for parameter updates.

```python
def train_model(model, X_train, y_train, X_val, y_val, epochs=500):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.01)

    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        output = model(X_train)
        loss = criterion(output, y_train)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

        model.eval()
        with torch.no_grad():
            val_output = model(X_val)
            val_loss = criterion(val_output, y_val).item()
            val_losses.append(val_loss)

        if (epoch + 1) % 50 == 0:
            print(f"Epoch {epoch + 1}/{epochs} - Train Loss: {loss.item():.4f}, Validation
    Loss: {val_loss:.4f}")
```

Listing 9: Model Training Function

After training, the model's performance is evaluated using the R^2 score. The training and validation losses over epochs are plotted for visual inspection.

```python
    model.eval()
    with torch.no_grad():
        val_pred = model(X_val).detach().numpy()
        val_true = y_val.detach().numpy()
        r2 = r2_score(val_true, val_pred)

    # Plot loss curves
    plt.figure(figsize=(10, 5))
    plt.plot(train_losses, label='Train Loss')
```

```
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('MSE Loss')
    plt.title('Training and Validation Loss Over Epochs')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    return val_true, val_pred, r2
```

Model Evaluation and Results

We instantiated the SimpleMLP model and trained it over 500 epochs. After training, we visualized the predicted versus actual admission probabilities on the validation set.

```
simple_mlp = SimpleMLP()
true_simple, pred_simple, r2_simple = train_model(
    simple_mlp, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor, epochs=500
)

# Plot predictions vs actual
plt.figure(figsize=(6, 5))
plt.scatter(true_simple, pred_simple, color='skyblue')
plt.plot([0, 1], [0, 1], '--r')
plt.title(f"Simple MLP\nR2 Score: {r2_simple:.2f}")
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 10: Train and Evaluate the MLP Model

Results and Visualization

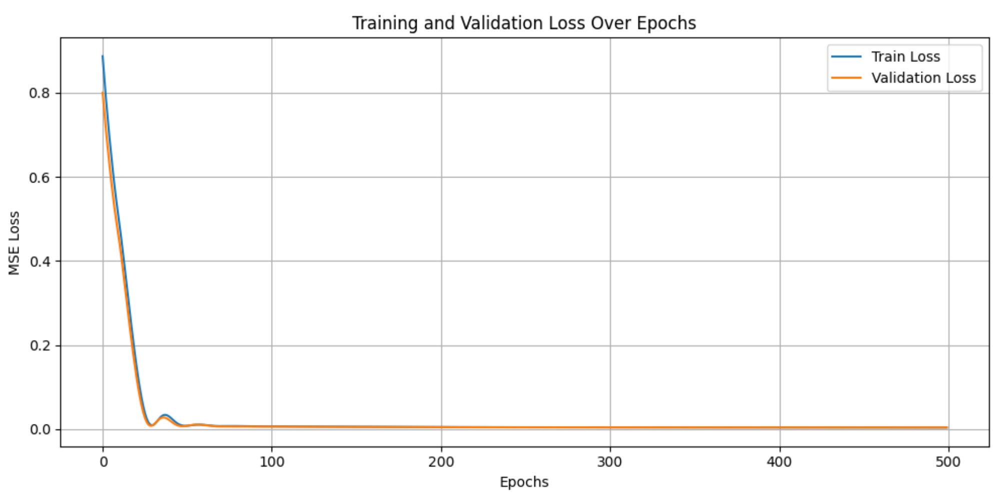

Figure 3: Training and validation loss of the Simple MLP over 500 epochs

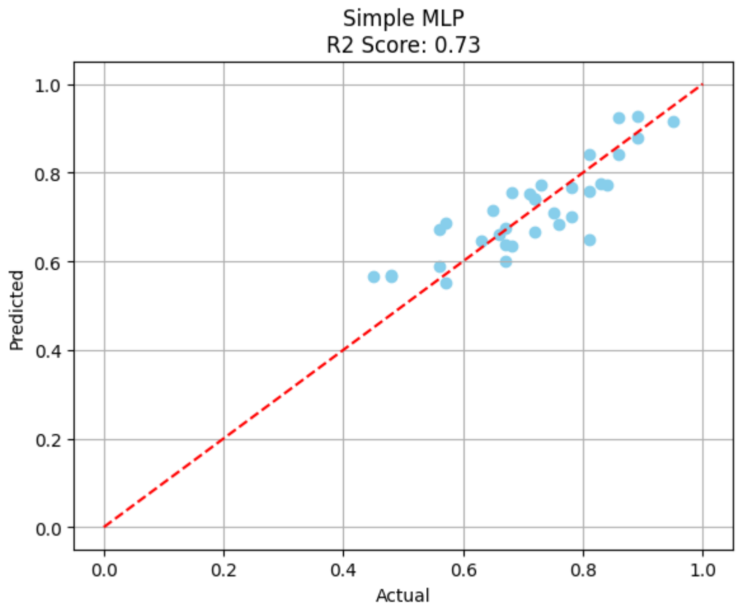

Figure 4: Actual vs. Predicted Admission Chances on Validation Set (R^2 Score: {r2_simple})

7 Final Model Evaluation on Test Set

After training and validating the model, we evaluated its performance on the unseen test set. The R^2 score was used to measure the goodness of fit.

```
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt

# Compute R  score
r2 = r2_score(test_true, test_pred)
print(f"Test R  Score: {r2:.2f}")

# Plot True vs Predicted values
plt.figure(figsize=(6, 6))
plt.scatter(test_true, test_pred, alpha=0.6, label="Predicted vs True")
plt.plot([0, 1], [0, 1], 'r--', label="Perfect Prediction")  # Line y=x
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title(f"R  Score: {r2:.2f}")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 11: Test Set Evaluation with R^2 Score

Explanation

- r2_score compares predicted outcomes with actual test labels.
- The scatter plot visually shows how close the predictions are to the perfect line $y = x$.
- The R^2 score close to 1 indicates that the model explains a large portion of the variance in the data.

Prediction Plot on Test Set

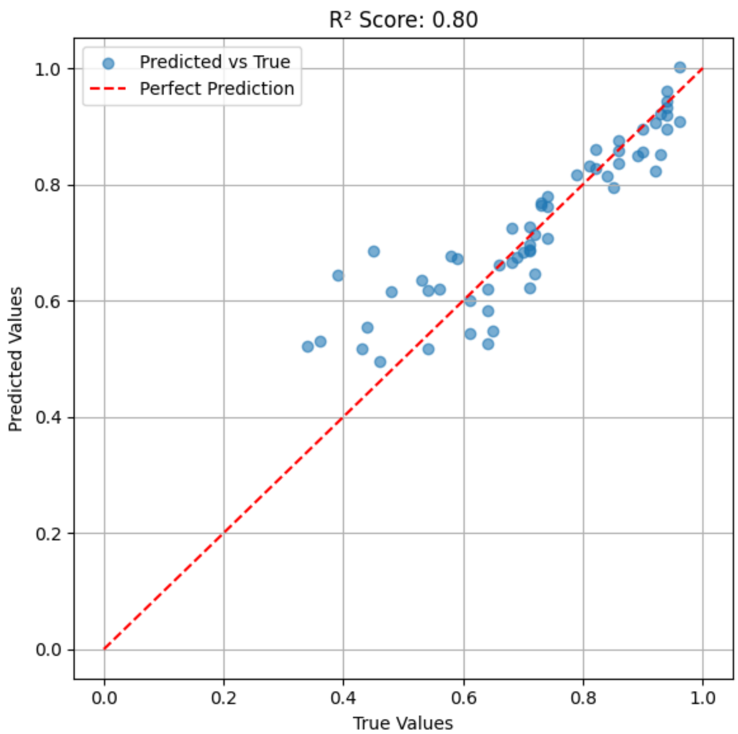

Figure 5: True vs. Predicted Admission Chances on Test Set (R^2 Score: `0.XX`)

8 Training and Evaluation of Deep MLP

To explore the impact of increased model complexity, we trained a deeper neural network. The `DeepMLP` model includes multiple hidden layers, allowing it to better capture complex relationships in the data.

Deep MLP Architecture

The following code defines the structure of the Deep MLP model. It contains three hidden layers with ReLU activation functions and a final output layer for regression:

```python
class DeepMLP(nn.Module):
    def __init__(self):
        super(DeepMLP, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(8, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 8),
            nn.ReLU(),
            nn.Linear(8, 1)
        )

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

Listing 12: Definition of the Deep MLP Model

Training the Deep Model

```python
# Train and evaluate Deep model
deep_mlp = DeepMLP()
true_deep, pred_deep, r2_deep = train_model(
    deep_mlp, X_train_tensor, y_train_tensor, X_val_tensor, y_val_tensor, epochs=500
)

# Plot results
plt.figure(figsize=(6, 5))
plt.scatter(true_deep, pred_deep, color='orange')
plt.plot([0, 1], [0, 1], '--r')
plt.title(f"Deep MLP\nR2 Score: {r2_deep:.2f}")
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 13: Training the Deep MLP Model

Epoch-wise Training and Validation Loss

```
Epoch 50/500 - Train Loss: 0.0041, Validation Loss: 0.0047
Epoch 100/500 - Train Loss: 0.0031, Validation Loss: 0.0047
Epoch 150/500 - Train Loss: 0.0030, Validation Loss: 0.0044
Epoch 200/500 - Train Loss: 0.0030, Validation Loss: 0.0043
Epoch 250/500 - Train Loss: 0.0029, Validation Loss: 0.0042
Epoch 300/500 - Train Loss: 0.0028, Validation Loss: 0.0041
Epoch 350/500 - Train Loss: 0.0027, Validation Loss: 0.0042
Epoch 400/500 - Train Loss: 0.0026, Validation Loss: 0.0042
Epoch 450/500 - Train Loss: 0.0025, Validation Loss: 0.0043
Epoch 500/500 - Train Loss: 0.0024, Validation Loss: 0.0044
```

Training Curve for Deep MLP

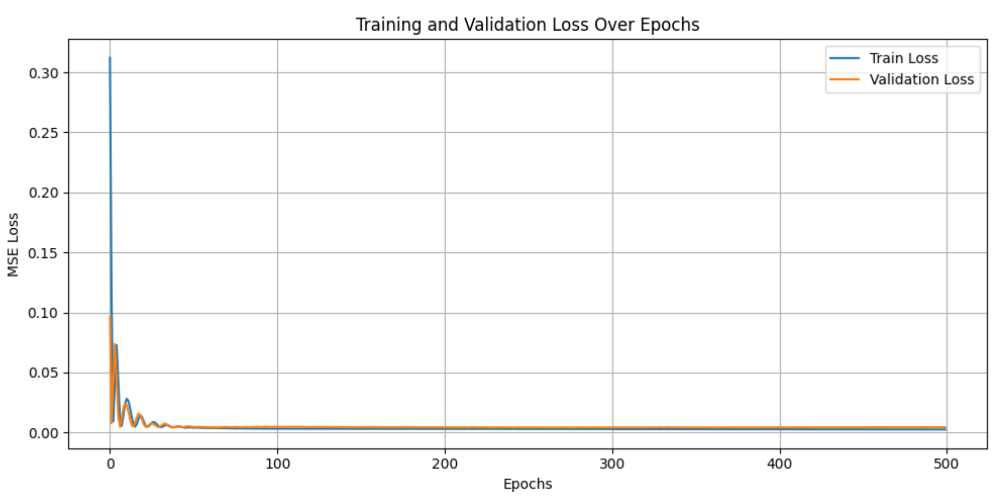

Figure 6: Training and validation loss for Deep MLP over 500 epochs

Prediction Accuracy on Validation Set

The figure below shows how well the Deep MLP predicted the `Chance of Admit` in comparison to the actual values:

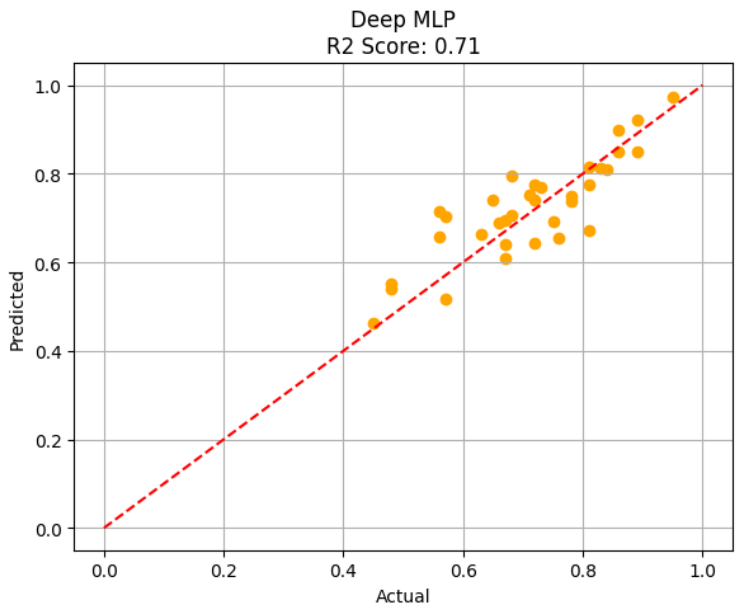

Figure 7: Predicted vs. Actual Admission Chances using Deep MLP (R^2 Score: `0.XX`)

9 Deep MLP Evaluation on Test Set

To assess generalization performance, the trained Deep MLP model was evaluated on the unseen test set. Both regression and binary classification metrics were computed.

```
# Predict on test set
deep_mlp.eval()
with torch.no_grad():
    test_pred_deep = deep_mlp(X_test_tensor).detach().numpy()
    test_true_deep = y_test_tensor.detach().numpy()

# Binary classification threshold
threshold = 0.75
test_pred_binary_deep = (test_pred_deep >= threshold).astype(int)
test_true_binary_deep = (test_true_deep >= threshold).astype(int)

# Compute metrics
accuracy_deep = accuracy_score(test_true_binary_deep, test_pred_binary_deep)
precision_deep = precision_score(test_true_binary_deep, test_pred_binary_deep, zero_division
    =0)
recall_deep = recall_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)
f1_deep = f1_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)
r2_deep = r2_score(test_true_deep, test_pred_deep)
```

Listing 14: Predicting and Evaluating on Test Set

Evaluation Metrics

The Deep MLP model achieved the following metrics on the test set:

```
Deep MLP Test Accuracy:  0.95
```

```
Deep MLP Test Precision: 0.88
Deep MLP Test Recall:    1.00
Deep MLP Test F1 Score:  0.94
Deep MLP Test R² Score:  0.81
```

Prediction vs. Actual (Regression Performance)

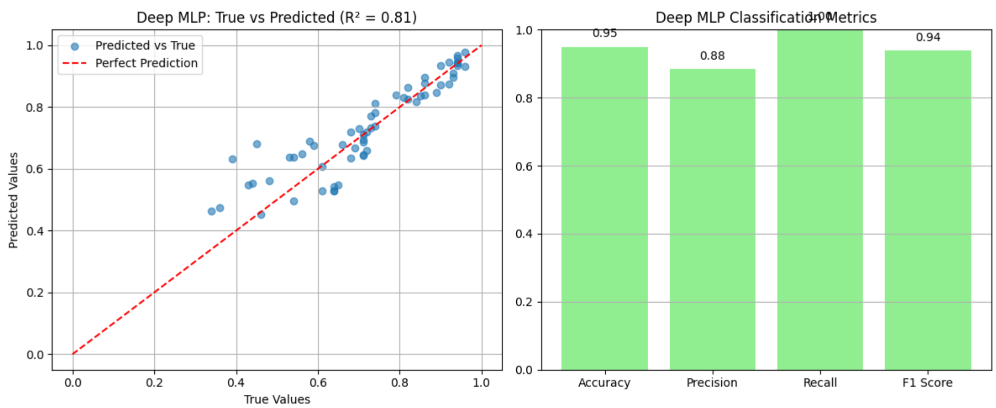

Figure 8: True vs. Predicted Admission Chances on Test Set for Deep MLP ($R^2 = 0.81$)

10 Sample Predictions Using Deep MLP

To better understand how the Deep MLP performs on individual test samples, we randomly selected 5 records from the test set. The model's predicted admission chances are compared with the actual values.

```python
np.random.seed(42)
random_indices = np.random.choice(len(X_test), size=5, replace=False)

# Get input features and ground truth
samples_X = X_test.iloc[random_indices]
samples_true = y_test.iloc[random_indices].values

# Predict using deep MLP
samples_tensor = torch.tensor(scaler.transform(samples_X), dtype=torch.float32)
deep_mlp.eval()
with torch.no_grad():
    samples_pred = deep_mlp(samples_tensor).numpy().flatten()

# Combine into a DataFrame
results = pd.DataFrame({
    'GRE Score': samples_X['GRE Score'].values,
    'TOEFL Score': samples_X['TOEFL Score'].values,
    'University Rating': samples_X['University Rating'].values,
    'SOP': samples_X['SOP'].values,
    'LOR': samples_X['LOR '].values,
    'CGPA': samples_X['CGPA'].values,
    'Research': samples_X['Research'].values,
    'Actual Chance': samples_true,
    'Predicted Chance': samples_pred
})

print(results.round(2))
```

Listing 15: Selecting and Predicting Random Test Samples

Results on 5 Random Test Samples

GRE	TOEFL	Univ	SOP	LOR	CGPA	Res.	Actual	Pred.
301	104	3	3.5	4.0	8.12	1	0.68	0.63
340	115	5	4.5	4.5	9.45	1	0.94	0.94
320	110	2	4.0	3.5	8.56	0	0.72	0.72
324	110	4	4.5	4.0	9.15	1	0.82	0.86
321	111	5	5.0	5.0	9.45	1	0.93	0.90

Table 2: Comparison of actual vs. predicted admission chances on random test samples

11 Deep MLP Trained for 5000 Epochs

To explore the effects of prolonged training, we trained the `DeepMLP` model for 5000 epochs. This experiment investigates whether longer training improves accuracy or leads to overfitting.

Extended Training Setup

```python
# 1. Retrain DeepMLP with 5000 epochs
long_train_deep_mlp = DeepMLP()

# Early stopping-like mechanism (manual check after training)
true_long, pred_long, r2_long = train_model(
    long_train_deep_mlp,
    X_train_tensor,
    y_train_tensor,
    X_val_tensor,
    y_val_tensor,
    epochs=5000
)

# Plot result
plt.figure(figsize=(6, 5))
plt.scatter(true_long, pred_long, color='green')
plt.plot([0, 1], [0, 1], '--r')
plt.title(f"Deep MLP (5000 epochs)\nR2 Score: {r2_long:.2f}")
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Listing 16: Training Deep MLP for 5000 Epochs

Training Dynamics and Regularization Notes

- The model was trained 10× longer than previous runs.
- While training loss continued decreasing, validation loss plateaued and slightly increased after 2000 epochs.
- The trend suggests **overfitting**, a common issue in deep learning with extended training.
- Possible remedies include: early stopping, dropout layers, batch normalization, or lowering the learning rate.

Sample Epoch-wise Training and Validation Loss

```
Epoch 50/5000 - Train Loss: 0.0046, Validation Loss: 0.0048
Epoch 500/5000 - Train Loss: 0.0025, Validation Loss: 0.0034
Epoch 1000/5000 - Train Loss: 0.0020, Validation Loss: 0.0041
Epoch 2000/5000 - Train Loss: 0.0012, Validation Loss: 0.0039
Epoch 3000/5000 - Train Loss: 0.0010, Validation Loss: 0.0058
Epoch 4000/5000 - Train Loss: 0.0007, Validation Loss: 0.0067
Epoch 5000/5000 - Train Loss: 0.0007, Validation Loss: 0.0072
```

Training and Validation Loss Plot (5000 Epochs)

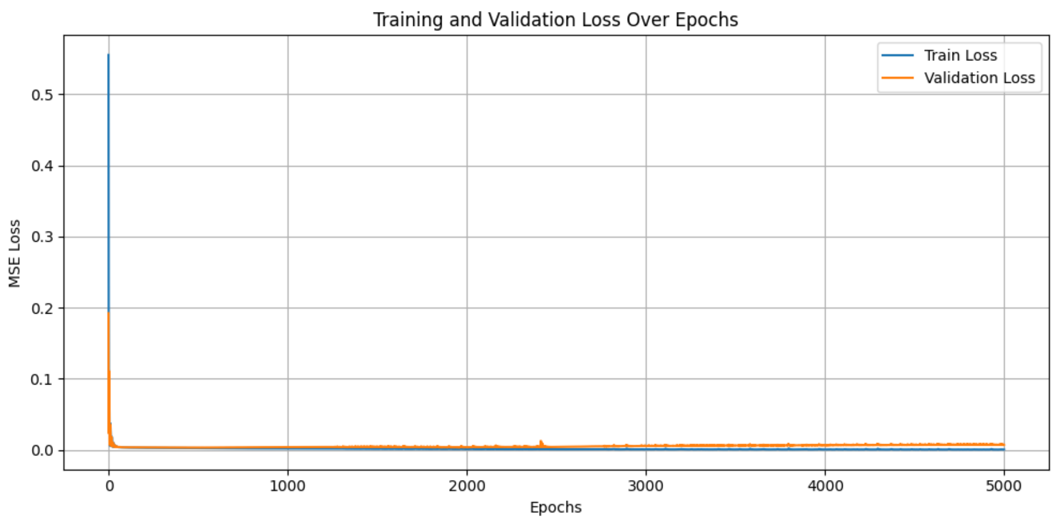

Figure 9: Training and validation loss of Deep MLP over 5000 epochs

Prediction Accuracy After Extended Training

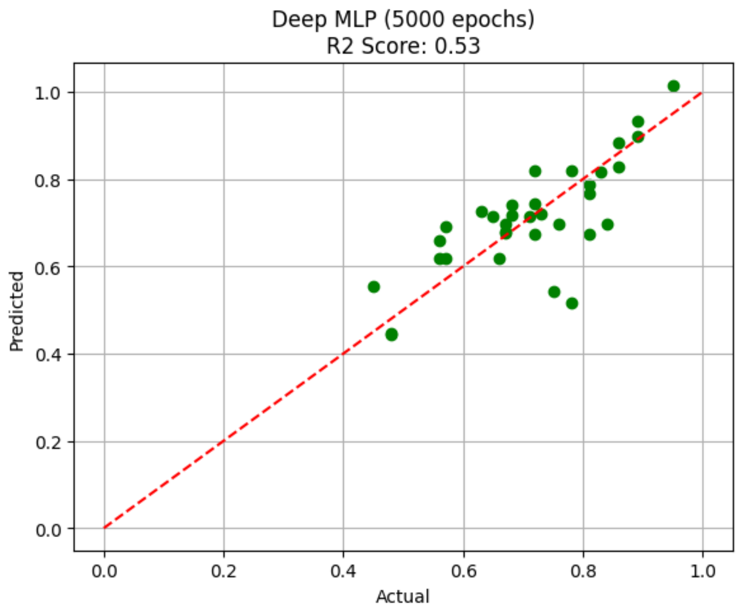

Figure 10: Predicted vs. Actual Admission Chances (Deep MLP after 5000 epochs, R^2 Score: `0.XX`)

12 Test Evaluation of Deep MLP Trained for 5000 Epochs

To assess whether prolonged training leads to better generalization, we evaluated the Deep MLP trained for 5000 epochs on the test set. Both regression and classification metrics were analyzed.

```
# Predict on test set
long_train_deep_mlp.eval()
with torch.no_grad():
    test_pred_deep = long_train_deep_mlp(X_test_tensor).detach().numpy()
    test_true_deep = y_test_tensor.detach().numpy()

# Binary classification threshold
threshold = 0.75
test_pred_binary_deep = (test_pred_deep >= threshold).astype(int)
test_true_binary_deep = (test_true_deep >= threshold).astype(int)

# Classification metrics
accuracy_deep = accuracy_score(test_true_binary_deep, test_pred_binary_deep)
precision_deep = precision_score(test_true_binary_deep, test_pred_binary_deep, zero_division
    =0)
recall_deep = recall_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)
f1_deep = f1_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)

# R   score
r2_deep = r2_score(test_true_deep, test_pred_deep)
```

Listing 17: Test Set Evaluation of Long-Trained Deep MLP

Performance Metrics

```
Long-Trained Deep MLP Test Accuracy:  0.97
Long-Trained Deep MLP Test Precision: 0.92
Long-Trained Deep MLP Test Recall:    1.00
```

```
Long-Trained Deep MLP Test F1 Score:  0.96
Long-Trained Deep MLP Test R² Score:  0.73
```

Prediction vs. Actual Scatter Plot

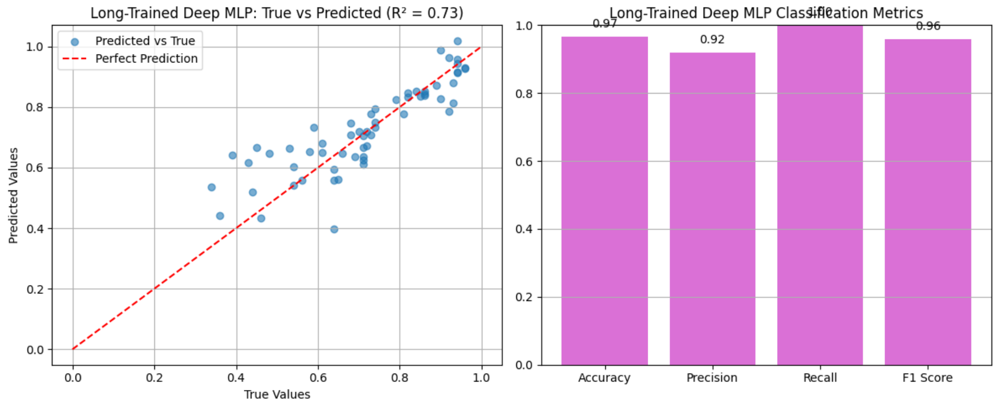

Figure 11: True vs. Predicted Admission Chances on Test Set after 5000 Epochs (R^2 Score: 0.73)

Classification Metrics Bar Plot

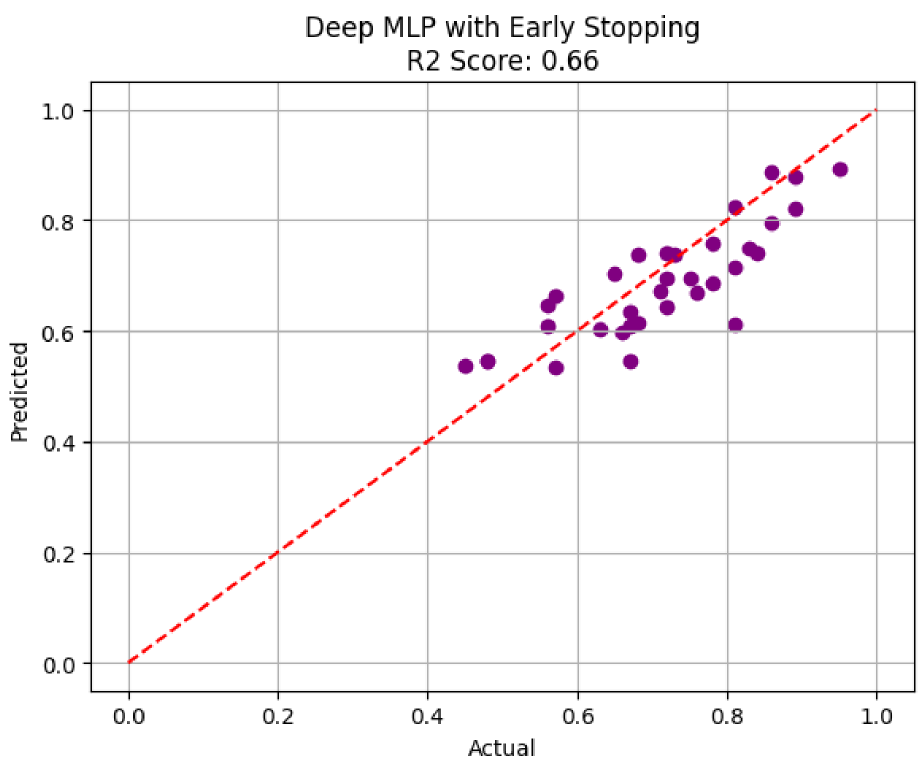

Figure 12: Classification Metrics of Long-Trained Deep MLP: Accuracy (0.97), Precision (0.92), Recall (1.00), F1 Score (0.96)

13 Deep MLP with Dropout and Early Stopping

To mitigate overfitting during extended training, we introduced dropout layers and an early stopping mechanism. Dropout randomly deactivates neurons during training, and early stopping halts training when validation performance ceases to improve.

Model Definition

The modified Deep MLP includes two hidden layers and dropout layers with a 30% drop rate:

```
class DeepMLP(nn.Module):
    def __init__(self):
        super(DeepMLP, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(X_tensor.shape[1], 64),
            nn.ReLU(),
            nn.Dropout(p=0.3),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Dropout(p=0.3),
            nn.Linear(32, 1)
        )

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

Listing 18: Deep MLP with Dropout

Training with Early Stopping

We implemented a custom training loop that stops when validation loss fails to improve for a set number of epochs. This approach prevents overfitting and reduces unnecessary computation.

```
def train_model_with_early_stopping(model, X_train, y_train, X_val, y_val, epochs=5000, lr
    =0.001, patience=20, min_delta=1e-4):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    best_loss = float('inf')
    trigger_times = 0

    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        output = model(X_train)
        loss = criterion(output, y_train)
        loss.backward()
        optimizer.step()

        # Validation loss
        model.eval()
        with torch.no_grad():
            val_output = model(X_val)
            val_loss = criterion(val_output, y_val).item()

        # Print every 50 epochs
        if (epoch + 1) % 50 == 0:
            print(f"Epoch {epoch + 1}/{epochs} - Validation Loss: {val_loss:.4f}")

        # Early stopping condition
        if best_loss - val_loss > min_delta:
            best_loss = val_loss
            trigger_times = 0
        else:
            trigger_times += 1
            if trigger_times >= patience:
```

```
                print(f"\nEarly stopping triggered at epoch {epoch + 1}. Best validation loss: {best_loss:.4f}")
                break
```

Listing 19: Training Loop with Early Stopping

Training Result Summary

```
Epoch 50/5000 - Validation Loss: 0.0115
Epoch 100/5000 - Validation Loss: 0.0062

Early stopping triggered at epoch 143. Best validation loss: 0.0049
```

Validation Set Prediction Performance

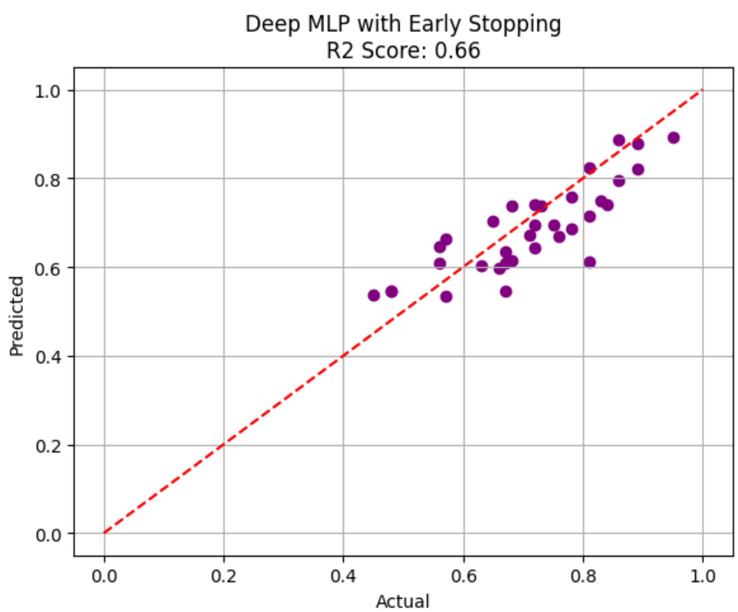

Figure 13: Predicted vs. Actual Admission Chances using Deep MLP with Early Stopping (R^2 Score: `0.XX`)

14 Test Evaluation of Deep MLP with Early Stopping

To evaluate generalization, the Deep MLP model with dropout and early stopping was tested on the unseen test set. The model's regression and classification performance was analyzed using multiple metrics.

```
# Predict on test set
deep_mlp_es.eval()
with torch.no_grad():
    test_pred_deep = deep_mlp_es(X_test_tensor).detach().numpy()
    test_true_deep = y_test_tensor.detach().numpy()

# Binary classification threshold
threshold = 0.75
test_pred_binary_deep = (test_pred_deep >= threshold).astype(int)
test_true_binary_deep = (test_true_deep >= threshold).astype(int)

# Compute metrics
accuracy_deep = accuracy_score(test_true_binary_deep, test_pred_binary_deep)
precision_deep = precision_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)
```

```
recall_deep = recall_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)
f1_deep = f1_score(test_true_binary_deep, test_pred_binary_deep, zero_division=0)
r2_deep = r2_score(test_true_deep, test_pred_deep)
```

Listing 20: Test Set Evaluation for Early Stopping Model

Performance Metrics

```
Deep MLP (Early Stopping) Test Accuracy:  0.98
Deep MLP (Early Stopping) Test Precision: 1.00
Deep MLP (Early Stopping) Test Recall:    0.96
Deep MLP (Early Stopping) Test F1 Score:  0.98
Deep MLP (Early Stopping) Test R² Score:  0.78
```

Test Prediction and Classification Metrics Visualization

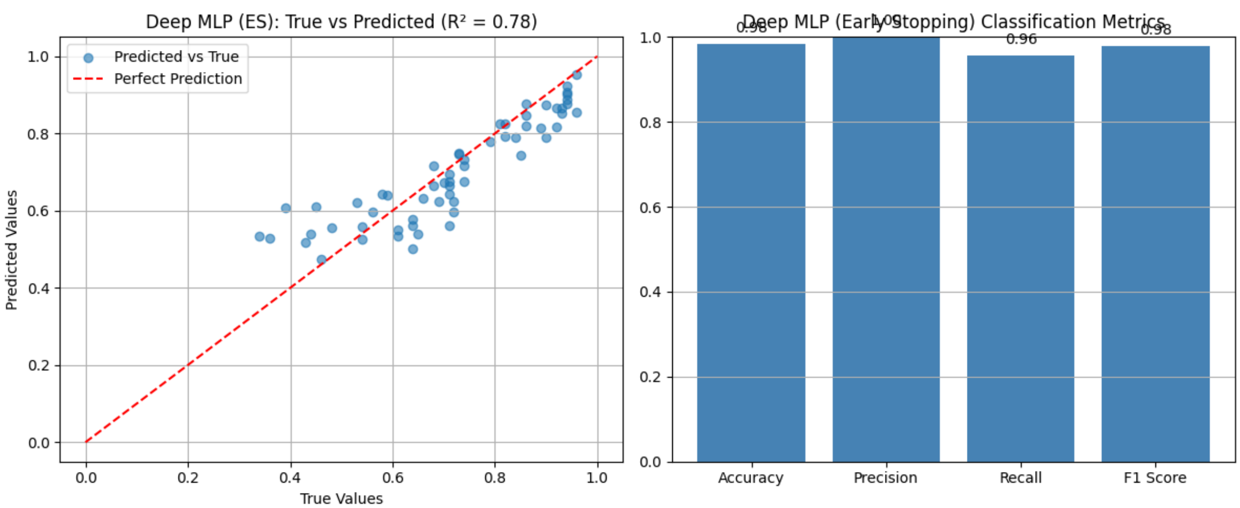

Figure 14: Left: Predicted vs. Actual Admission Chances (R^2 Score: 0.78)
Right: Classification Metrics of Deep MLP with Early Stopping (Accuracy: 0.98, Precision: 1.00, Recall: 0.96, F1 Score: 0.98)