CHAPTER – 6

IMPLEMENTATION

CHAPTER 6

IMPLEMENTATION

6.1 Module Implementation

The implementation of our drug response prediction model followed a structured approach encompassing data preprocessing, model development, and deployment within a user accessible web interface. The objective was to create an efficient tool for predicting drug sensitivity in oncology using a deep learning model. This chapter provides a comprehensive description of each step, from data processing and model training to the development of a web interface using Flask, allowing real-time predictions and visualizations.

6.2 Data Preprocessing

The GDSC dataset required extensive preprocessing to ensure compatibility with our PyTorch based Artificial Neural Network (ANN) model. Key data preprocessing steps included:

- **Data Loading and Initial Checks:** The GDSC dataset was loaded and inspected for missing values and inconsistencies. Initial data cleaning involved removing rows with missing or incomplete values, ensuring a robust dataset for training and evaluation.

```
import pandas as pd
# Load the dataset
data = pd.read_csv('GDSC_DATASET.csv')
print(data.info())
# Drop rows with missing values
data.dropna(inplace=True)
print("Data shape after dropping missing values:",
```

- **Feature Selection:** Feature selection focused on identifying the most relevant genomic features, such as gene mutations, copy number alterations (CNAs), and tissue descriptors, using domain knowledge and feature ranking techniques.

- **Normalization and Scaling:** Continuous variables were normalized and scaled to ensure consistency across features, enhancing model learning efficiency and accuracy.

```
from sklearn.preprocessing import StandardScaler
# Define numerical features for scaling
numerical_features = ['AUC', 'Z_SCORE',
'TARGET_PATHWAY']
# Initialize and apply the scaler
scaler = StandardScaler()
```

- **Encoding Categorical Variables:** Categorical features such as tissue descriptors and cancer types were one-hot encoded to create binary vectors, making them suitable for input into the ANN model.

```
# Apply one-hot encoding to categorical columns
data = pd.get_dummies(data, columns=['GDSC Tissue
descriptor 1', 'Cancer Type (matching TCGA label)',
```

- **Data Splitting:** The preprocessed dataset was split into training and testing subsets to evaluate the model's performance and generalization.

```
from sklearn.model_selection import train_test_split
# Define features and target variable
X = data.drop(columns=['LN_IC50'])
y = data['LN_IC50']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
```

6.3 Model Development

The ANN model was designed to analyze complex genomic data and predict IC50 values for drug sensitivity.

- **Model Architecture:** The ANN model comprised an input layer, multiple hidden layers with ReLU activation, and an output layer for regression. This architecture was chosen to capture the nonlinear relationships between genomic features and drug response data.

```
import torch
import torch.nn as nn
import torch.optim as optim
# Define the ANN model architecture
class DrugSensitivityModel(nn.Module):
    def __init__(self, input_dim):
        super(DrugSensitivityModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 1)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

- **Model Compilation:** The model was compiled with Mean Squared Error (MSE) as the loss function (appropriate for regression tasks) and the Adam optimizer for efficient.

```
model =
DrugSensitivityModel(input_dim=X_train.shape[1])
criterion = nn.MSELoss()
```

- **Model Training:** The model was trained with early stopping to prevent overfitting, using batch training and validation.

```
from torch.utils.data import DataLoader,
TensorDataset
import torch
# Convert data to PyTorch tensors
train_data =
TensorDataset(torch.tensor(X_train.values,
# Training loop with early stopping
num_epochs = 50
patience = 5
best_loss = float('inf')
patience_counter = 0
for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    epoch_loss = running_loss / len(train_loader)
    print(f'Epoch {epoch+1}, Loss: {epoch_loss}')
    # Early stopping
    if epoch_loss < best_loss:
        best_loss = epoch_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping")
            break
```

- **Model Evaluation:** Model performance was assessed using Root Mean Squared Error (RMSE) and R-squared (R^2) to verify accuracy and generalizability.

```
from sklearn.metrics import mean_squared_error,
r2_score
# Predict and evaluate on test data
model.eval()
```

```
predictions = model(torch.tensor(X_test.values,
dtype=torch.float32)).squeeze().numpy()
rmse = mean_squared_error(y_test, predictions,
squared=False)
r2 = r2_score(y_test, predictions)
print("RMSE:", rmse)
```

6.4 Web Interface Development

A user-friendly Flask-based web interface was created to facilitate interaction with the model, allowing users to input genomic data and receive IC50 predictions.

- **Setting Up Flask:** Flask was configured as the backend framework to handle user input, data processing, and model integration.

```
from flask import Flask, request, render_template
app = Flask(__name__)
@app.route('/')
def home():
    return render_template('index.html')
```

- **Frontend Design:** HTML and CSS were used to design a user-friendly input form and result display, ensuring an intuitive experience.

```
<-- HTML structure for input form -->
<form action="/predict" method="post">
    <label for="AUC">AUC:</label>
    <input type="number" id="AUC" name="AUC" required>
    <button type="submit">Predict</button>
</form>
```

- **Model Integration with Flask:** The model was integrated with Flask to process user inputs, generate predictions, and display them in real time**.**

```
@app.route('/predict', methods=['POST'])
def predict():
    auc = float(request.form['AUC'])
    prediction = model(torch.tensor([[auc]],
dtype=torch.float32)).item()
```

6.5 Deployment

- **Model Serialization:** The trained model was saved for efficient loading and deployment.

```
# Save the trained model
torch.save(model.state_dict(),
```

- **Server Deployment:** The application was hosted locally using Flask, with future plans for cloud deployment.

```
# Run the Flask application
flask run
```