# CHAPTER – 6 IMPLEMENTATION

## CHAPTER 6

**IMPLEMENTATION**

#### 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:", data.shape)

* **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()

data[numerical\_features] = scaler.fit\_transform(data[numerical\_features])

* **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)', 'TARGET'])

* **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, random\_state=42)

**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()

optimizer = optim.Adam(model.parameters(), lr=0.001)

* **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, dtype=torch.float32), torch.tensor(y\_train.values, dtype=torch.float32))

train\_loader = DataLoader(train\_data, batch\_size=32, shuffle=True)

# 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 (R²) to verify accuracy and generalizability.

from sklearn.metrics import mean\_squared\_error, r2\_score

# Predict and evaluate on test data

model.eval()

with torch.no\_grad():

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)

print("R²:", r2)

**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()

return render\_template('result.html', prediction=prediction)

**6.5 Deployment**

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

# Save the trained model

torch.save(model.state\_dict(), 'drug\_response\_model.pth')

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

# Run the Flask application

flask run

'drug\_response\_model.pth')