Deep Learning- Task 5: Training on Binary Classifiers (Heart Disease Death Classification)

1. Task Overview

This task focuses on implementing a complete PyTorch-based pipeline for binary classification, using a heart failure clinical records dataset. It encompasses three key phases: importing essential libraries to lay the technical foundation, preprocessing the dataset to address class imbalance, skewed features, and feature scaling, and finally building, training, and evaluating an Artificial Neural Network (ANN) to predict the "DEATH_EVENT" target variable.

Each step is designed to ensure the dataset is model-ready and the ANN functions effectively—from setting up tools for data manipulation and neural network construction to refining data quality and assessing model performance via metrics like accuracy and loss. The task ties technical implementation to practical goals, ensuring the pipeline is both robust and interpretable for binary classification tasks.

2. Task Objective

The task objective is to successfully implement a PyTorch-based pipeline for binary classification with the heart failure dataset, ensuring each phase (library import, data preprocessing, ANN building/training/evaluation) is executed effectively to achieve reliable "DEATH_EVENT" prediction.

Objective Category	Sub - objective	Details
Ideological & Political Objectives	Ideological & Political Elements	Highlight the practical value of technical pipelines in medical data analysis, emphasizing the importance of ethical data handling (e.g., protecting patient privacy in clinical records) and responsible AI application in healthcare decision
Learning Objectives	Knowledge Objectives	Enable learners to master key knowledge such as PyTorch library functions for neural network construction, common data preprocessing techniques (handling class imbalance, feature scaling), and binary

		classification metrics (accuracy, loss). Understand the characteristics of heart failure clinical records datasets and their relevance to the "DEATH_EVENT" target.
	Skill Objectives	Develop learners' skills in importing and utilizing essential libraries (e.g., PyTorch, pandas, scikit-learn), preprocessing medical datasets to resolve issues like class imbalance and skewed features, and building, training, and evaluating an ANN using PyTorch. Cultivate the ability to debug pipeline steps and optimize model performance.
	Literacy Objectives	Enhance learners' literacy in technical documentation interpretation (e.g., PyTorch official docs), data-driven reasoning (linking preprocessing choices to model performance), and clear presentation of pipeline results (e.g., explaining loss/accuracy trends and model interpretability for clinical context).

3. Task Implementation

Importing Libraries

This snippet imports essential **Python libraries** required for **data manipulation, preprocessing, oversampling, scaling, and PyTorch-based modeling**. These libraries provide the foundation for preparing data, handling imbalances, and building neural networks for classification or regression tasks.

Code block

```
import pandas as pd
import numpy as np
import torch
from scipy.stats import boxcox
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from torch import nn
from torch.utils.data import Dataset, DataLoader
```

Explanation of each import:

- `pandas as pd` → For reading, processing, and manipulating tabular datasets.
- `numpy as np` → Provides support for numerical computations and array operations.

- `torch` → Core PyTorch library for building and training neural networks.
- `from scipy.stats import boxcox` → For **Box-Cox transformation** to stabilize variance and make data more Gaussian-like.
- `from imblearn.over_sampling import SMOTE` → Handles **class imbalance** by generating synthetic minority samples.
- `from sklearn.preprocessing import StandardScaler` → Scales features to zero mean and unit variance, improving model convergence.
- `from sklearn.metrics import accuracy_score` → Evaluates the performance of classification models.
- `from torch import nn` → Provides PyTorch modules to define neural network layers and architectures.
- `from torch.utils.data import Dataset, DataLoader` → Helps create custom datasets and load them in **mini-batches** for training neural networks.

This setup ensures the workflow is ready for **data preprocessing, balancing, scaling, and building a PyTorch-based machine learning or deep learning pipeline**.

Data Preprocessing

This snippet performs **end-to-end preprocessing** for a tabular dataset, preparing it for training a neural network. The steps include splitting data into training and testing sets, handling class imbalance with **SMOTE**, transforming skewed features using **Box-Cox**, and scaling all features for optimal model convergence.

Code block

```
dataset = pd.read_csv("D:\BOOK\Deep_learning_book\ANN\datatset\heart_failure_clinical_records_dataset.csv")


def train_test_split(dataset: pd.DataFrame, train_size, prediction_column: str):
    size = int(train_size * len(dataset))
    train_dataset = dataset.sample(size)
    test_dataset = dataset.drop(train_dataset.index, axis=0)
    X_train = train_dataset.drop([prediction_column], axis=1)
    y_train = train_dataset[prediction_column]
    X_test = test_dataset.drop([prediction_column], axis=1)
    y_test = test_dataset[prediction_column]
    return X_train, X_test, y_train, y_test


```

```
X_train, X_test, y_train, y_test = train_test_split(dataset, 0.8,
'DEATH_EVENT')

smote = SMOTE(random_state=42)
X_train, y_train = smote.fit_resample(X_train, y_train)

skewed_columns = ['creatinine_phosphokinase', 'platelets', 'serum_creatinine',
'serum_sodium']

# Store lambdas for each column
lambdas_dict = {}

# Apply Box-Cox on training data
for col in skewed_columns:
    # Box-Cox requires positive values
    X_train[col] = X_train[col] + 1e-6
    X_train[col], lambdas_dict[col] = boxcox(X_train[col])

# Apply the same transformation to test data using the stored lambdas
for col in skewed_columns:
    X_test[col] = X_test[col] + 1e-6
    X_test[col] = boxcox(X_test[col], lmbda=lambdas_dict[col])

scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Step-by-step explanation:

1. **Loading Dataset:**

Code block

```
dataset = pd.read_csv("...csv")
```

1. Reads the dataset into a Pandas DataFrame for further processing.
2. **Train-Test Split:**

Code block

```
def train_test_split(...):
```

- Splits the dataset into training and testing sets based on a given ratio.

- Separates features (`X_train` , `X_test`) and target (`y_train` , `y_test`).

1. **Handling Class Imbalance:**

```
smote = SMOTE(...)
X_train, y_train = smote.fit_resample(...)
```

- Applies **SMOTE** (Synthetic Minority Oversampling Technique) to generate synthetic samples for the minority class.
- Ensures the training dataset is balanced to prevent model bias.

1. **Box-Cox Transformation for Skewed Features:**

```
X_train[col], lambdas_dict[col] = boxcox(X_train[col])
```

- Corrects skewed distributions in features like `creatinine_phosphokinase` and `serum_creatinine` .
- Stores the lambda values used in training to **apply the same transformation to the test set**.

1. **Feature Scaling:**

```
scaler = StandardScaler()
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

- Standardizes features to have **zero mean and unit variance**.
- Ensures all features contribute equally during training and accelerates convergence for gradient-based models.

This preprocessing pipeline ensures that the dataset is **balanced, normalized, and ready for neural network training**, minimizing issues like skewed distributions, feature scale differences, or class imbalance.

Building, Training, and Evaluating a PyTorch ANN for Binary Classification

This snippet defines an **Artificial Neural Network (ANN)** for a binary classification task (predicting `DEATH_EVENT`) using **PyTorch**, trains it with a **binary cross-entropy loss**, and evaluates its performance on a test set. The workflow includes forward propagation, loss computation, backpropagation, and accuracy calculation.

Code block

```
class ANNModel(nn.Module):
    def __init__(self):
        super(ANNModel, self).__init__()
        self.fc1 = nn.Linear(12,128)
        self.fc2 = nn.Linear(128,60)
        self.fc3 = nn.Linear(60,1)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))
        return x

model = ANNModel().to("cuda")

criterian = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
epochs = 30

def evaluate(test_loader):
    with torch.no_grad():
        test_loss_add = 0.0
        test_true = []
        test_predictions = []
        for test_data, test_label in test_loader:
            test_data, test_label = test_data.to("cuda"), test_label.to("cuda")
            test_pred = model(test_data)
            test_pred = test_pred.squeeze(1)
            test_loss = criterian(test_pred, test_label)
            test_true.append(test_label.cpu().numpy())
            test_predictions.append(test_pred.cpu().numpy())
            test_loss_add += test_loss.item()

        average_test_loss = test_loss_add / len(test_loader)
        ty_true = np.concatenate(test_true)
        ty_pred = np.concatenate(test_predictions)
        ty_pred_class = (ty_pred > 0.5).astype(int)
        test_accuracy = accuracy_score(ty_true, ty_pred_class)
    return test_accuracy, average_test_loss

for epoch in range(epochs):
```

```
        train_loss = 0.0
        true_label = []
        pred_label = []
        model.train()
        for data, label in train_loader:
            optimizer.zero_grad()
            data, label = data.to("cuda"), label.to("cuda")
            pred_train = model(data)
            loss = criterian(pred_train.squeeze(1), label)
            loss.backward()
            optimizer.step()

            true_label.append(label.cpu().numpy())
            pred_label.append(pred_train.squeeze(1).cpu().detach().numpy())
            train_loss += loss.item()

        y_true = np.concatenate(true_label)
        y_pred = np.concatenate(pred_label)
        y_pred_class = (y_pred > 0.5).astype(int)

        accuracy = accuracy_score(y_true, y_pred_class)
        average_loss = train_loss / len(train_loader)
        test_accuracy, test_loss = evaluate(test_loader)
        print(f' Epoch {epoch + 1}: Train Loss: {average_loss}, Train Accracy:
    {accuracy}, Test Accuracy: {test_accuracy}, test loss: {test_loss} ')
```

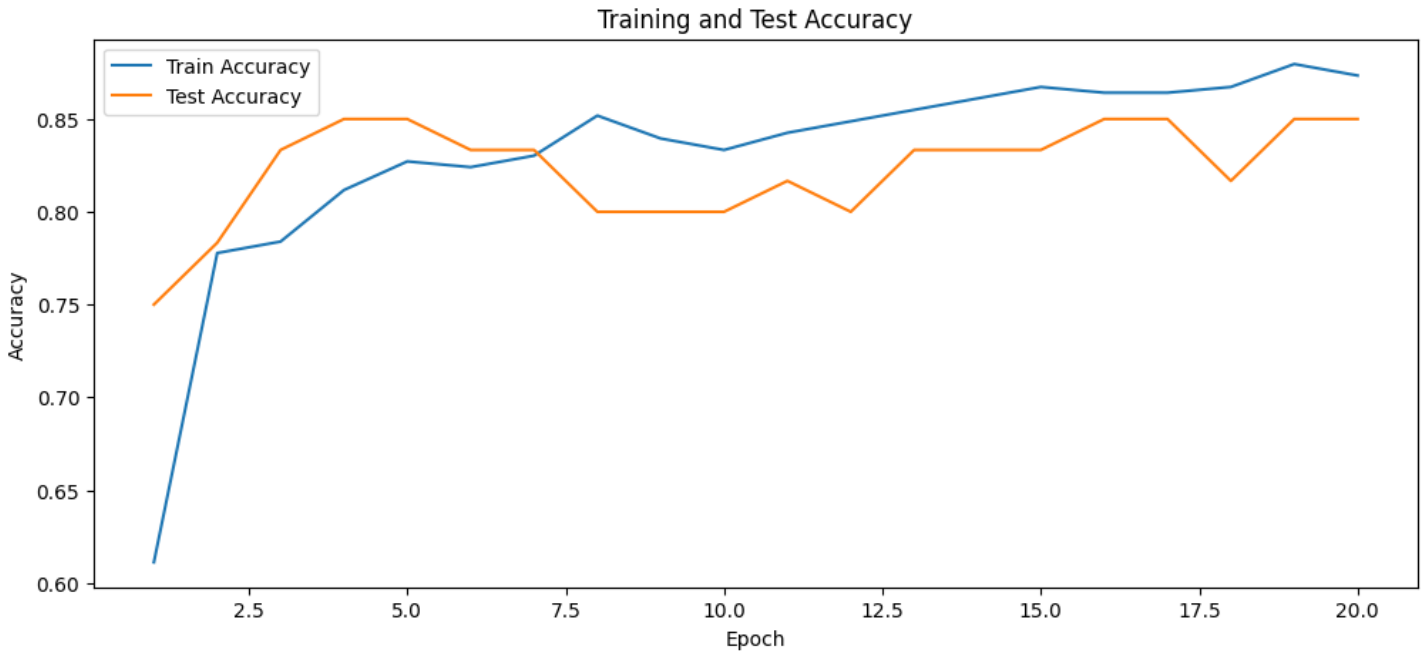
Step-by-step explanation:

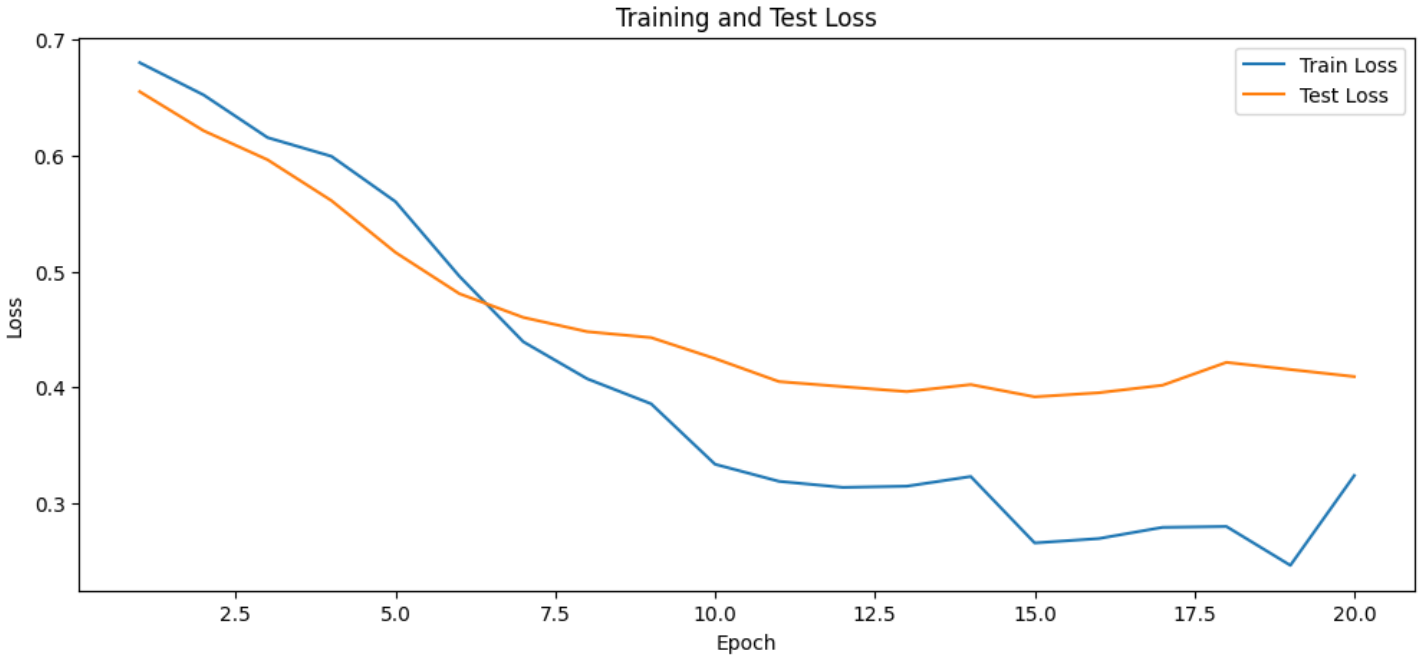
1. **Defining the ANN model:**
   - `ANNModel` has **3 fully connected layers**:
     - Input layer: 12 features → 128 neurons
     - Hidden layer: 128 → 60 neurons
     - Output layer: 60 → 1 neuron with **sigmoid activation** (for binary classification).
   - ReLU activation is applied in hidden layers to introduce non-linearity.
2. **Setting up loss, optimizer, and device:**
   - `BCELoss` (Binary Cross-Entropy) is used for binary classification.
   - `Adam` optimizer updates weights using gradients efficiently.
   - Model is moved to **GPU** using `.to("cuda")` for faster computation.
3. **Evaluation function:**
   - Computes **test loss** and **accuracy**.

- Squeezes predictions to 1D, converts probabilities to binary classes using a threshold of 0.5.

4. **Training loop:**
   - Iterates over epochs and batches:
     - Performs forward pass (`pred_train = model(data)`)
     - Computes loss and performs **backpropagation**
     - Updates weights using `optimizer.step()`
   - Collects predictions and true labels for **train accuracy** computation.
   - After each epoch, evaluates the model on the test set using the `evaluate()` function.
5. **Metrics reporting:**
   - Prints **train loss, train accuracy, test accuracy, and test loss** for each epoch, allowing monitoring of overfitting and model performance.

This code completes a **binary classification pipeline** using PyTorch, incorporating modern practices like GPU acceleration, batch-wise training, and proper evaluation.




Training and Test Loss
Train Loss
Test Loss
Loss
Epoch
0.7
0.6
0.5
0.4
0.3
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0