Simple Neural Network Classifier in PyTorch

Overview

In this exercise you will build, train, and evaluate a **binary classification neural network** using **PyTorch**.

The business context is **customer purchase intention**: given a user's browsing behaviour on an e-commerce site, predict whether they will complete a purchase during that session.

By the end of this notebook you will have practiced:

- Loading and preprocessing a real-world tabular dataset
- Encoding categorical features and normalising numerical ones
- Building a feedforward neural network with `torch.nn.Module`
- Writing a training loop with a loss function and optimiser
- Evaluating model performance with accuracy, precision, recall, and F1-score
- Visualising training progress

Dataset: Online Shoppers Purchasing Intention

Source: UCI Machine Learning Repository

The dataset contains 12,330 sessions collected from an e-commerce website. Each row represents one user session and includes:

Feature group	Examples
Page visit counts	`Administrative`, `Informational`, `ProductRelated`
Time spent on pages	`Administrative_Duration`, `Informational_Duration`, `ProductRelated_Duration`
Google Analytics metrics	`BounceRates`, `ExitRates`, `PageValues`, `SpecialDay`
Session context	`Month`, `OperatingSystems`, `Browser`, `Region`, `TrafficType`
Visitor type	`VisitorType` (Returning / New / Other)
Timing flag	`Weekend` (Boolean)

Target variable: `Revenue` — `True` if the session ended in a purchase, `False` otherwise.

The dataset is **imbalanced**: roughly 84 % of sessions do not result in a purchase.

Getting the data

Run the cell below to download the CSV directly from the UCI repository.

In [1]:
```
import urllib.request
import os

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/00468/onlir
filename = "online_shoppers_intention.csv"

if not os.path.exists(filename):
    urllib.request.urlretrieve(url, filename)
    print("Dataset downloaded.")
else:
    print("Dataset already present.")
```

```
Dataset downloaded.
```

Step 1 — Imports

Import all libraries you will need for this exercise.

You will need at minimum:

- `pandas` and `numpy` for data handling
- `sklearn` utilities: `train_test_split`, `StandardScaler`, and classification metrics
- `torch`, `torch.nn`, `torch.optim`
- `torch.utils.data`: `TensorDataset`, `DataLoader`
- `matplotlib.pyplot` for plotting

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt
```

Step 2 — Load and Explore the Data

Load the CSV file into a pandas DataFrame.

Tasks:

1. Load the dataset and display the first few rows.

2. Check the shape, column names, and data types.
3. Check for missing values.
4. Inspect the class balance: how many sessions resulted in a purchase vs. not?

In [3]:
```
df = pd.read_csv("online_shoppers_intention.csv")
print(df.head())
print(f"\nShape: {df.shape}")
```

```
   Administrative  Administrative_Duration  Informational  \
0               0                      0.0              0
1               0                      0.0              0
2               0                      0.0              0
3               0                      0.0              0
4               0                      0.0              0

   Informational_Duration  ProductRelated  ProductRelated_Duration  \
0                     0.0               1                 0.000000
1                     0.0               2                64.000000
2                     0.0               1                 0.000000
3                     0.0               2                 2.666667
4                     0.0              10               627.500000

   BounceRates  ExitRates  PageValues  SpecialDay Month  OperatingSystems  \
0         0.20       0.20         0.0         0.0   Feb                 1
1         0.00       0.10         0.0         0.0   Feb                 2
2         0.20       0.20         0.0         0.0   Feb                 4
3         0.05       0.14         0.0         0.0   Feb                 3
4         0.02       0.05         0.0         0.0   Feb                 3

   Browser  Region  TrafficType        VisitorType  Weekend  Revenue
0        1       1            1  Returning_Visitor    False    False
1        2       1            2  Returning_Visitor    False    False
2        1       9            3  Returning_Visitor    False    False
3        2       2            4  Returning_Visitor    False    False
4        3       1            4  Returning_Visitor     True    False

Shape: (12330, 18)
```

In [4]:
```
print(df.dtypes)
print("\nMissing values:")
print(df.isnull().sum())
print("\nClass balance:")
print(df['Revenue'].value_counts(normalize=True))
```

```
Administrative              int64
Administrative_Duration    float64
Informational               int64
Informational_Duration     float64
ProductRelated              int64
ProductRelated_Duration    float64
BounceRates                float64
ExitRates                  float64
PageValues                 float64
SpecialDay                 float64
Month                          str
OperatingSystems            int64
Browser                     int64
Region                      int64
TrafficType                 int64
VisitorType                   str
Weekend                      bool
Revenue                      bool
dtype: object

Missing values:
Administrative             0
Administrative_Duration    0
Informational              0
Informational_Duration     0
ProductRelated             0
ProductRelated_Duration    0
BounceRates                0
ExitRates                  0
PageValues                 0
SpecialDay                 0
Month                      0
OperatingSystems           0
Browser                    0
Region                     0
TrafficType                0
VisitorType                0
Weekend                    0
Revenue                    0
dtype: int64

Class balance:
Revenue
False    0.845255
True     0.154745
Name: proportion, dtype: float64
```

Step 3 — Preprocess the Data

The dataset contains a mix of numerical and categorical columns. You need to prepare it before passing it to a neural network.

Tasks:

1. **Encode categorical columns.** The columns `Month`, `VisitorType`, and `Weekend` are not numeric. Use one-hot encoding (e.g., `pd.get_dummies`) or label encoding as appropriate. Drop the original columns after encoding.
2. **Encode the target.** Convert the `Revenue` column (boolean) to integer (0/1) and separate it from the features.
3. **Split the data** into training and test sets (80 % / 20 %). Use `random_state=42` and `stratify=y` to preserve class proportions.
4. **Normalise numerical features** using `StandardScaler`. Fit the scaler on the training set only, then transform both train and test sets.

> **Why stratify?** The dataset is imbalanced. Stratified splitting ensures both train and test sets have the same proportion of positive examples as the full dataset.

In [5]:
```
# One-hot encode categorical columns
df_encoded = pd.get_dummies(df, columns=['Month', 'VisitorType'], drop_first
df_encoded['Weekend'] = df_encoded['Weekend'].astype(int)

# Encode target
y = df_encoded['Revenue'].astype(int).values
X = df_encoded.drop(columns=['Revenue']).values.astype(np.float32)

print(f"Features shape: {X.shape}")
print(f"Target shape: {y.shape}")
print(f"Feature columns: {df_encoded.drop(columns=['Revenue']).columns.tolis
```

```
Features shape: (12330, 28)
Target shape: (12330,)
Feature columns: ['Administrative', 'Administrative_Duration', 'Informationa
l', 'Informational_Duration', 'ProductRelated', 'ProductRelated_Duration',
'BounceRates', 'ExitRates', 'PageValues', 'SpecialDay', 'OperatingSystems',
'Browser', 'Region', 'TrafficType', 'Weekend', 'Month_Aug', 'Month_Dec', 'Mo
nth_Feb', 'Month_Jul', 'Month_June', 'Month_Mar', 'Month_May', 'Month_Nov',
'Month_Oct', 'Month_Sep', 'VisitorType_New_Visitor', 'VisitorType_Other', 'V
isitorType_Returning_Visitor']
```

In [6]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

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

print(f"Train size: {X_train.shape}, Test size: {X_test.shape}")
```

```
Train size: (9864, 28), Test size: (2466, 28)
```

Step 4 — Create PyTorch Datasets and DataLoaders

PyTorch models consume data through `DataLoader` objects, which handle batching and shuffling automatically.

Tasks:

1. Convert your NumPy arrays `X_train`, `X_test`, `y_train`, `y_test` to `torch.FloatTensor` (features) and `torch.FloatTensor` (labels — keep as float for `BCELoss` compatibility).
2. Wrap each pair into a `TensorDataset`.
3. Create a `DataLoader` for the training set with `batch_size=64` and `shuffle=True`, and one for the test set with `shuffle=False`.
4. Print the number of batches in each loader to verify.

In [7]:
```
X_train_t = torch.FloatTensor(X_train)
X_test_t  = torch.FloatTensor(X_test)
y_train_t = torch.FloatTensor(y_train)
y_test_t  = torch.FloatTensor(y_test)

train_ds = TensorDataset(X_train_t, y_train_t)
test_ds  = TensorDataset(X_test_t,  y_test_t)

train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_ds,  batch_size=64, shuffle=False)

print(f"Train batches: {len(train_loader)}, Test batches: {len(test_loader)}")
```

```
Train batches: 155, Test batches: 39
```

Step 5 — Define the Neural Network

Build a feedforward neural network by subclassing `torch.nn.Module`.

Architecture requirements:

- **Input layer:** size equal to the number of features after preprocessing.
- **Hidden layer 1:** 64 neurons, ReLU activation.
- **Hidden layer 2:** 32 neurons, ReLU activation.
- **Output layer:** 1 neuron, Sigmoid activation (outputs a probability between 0 and 1).

Tasks:

1. Define the class `CustomerClassifier(nn.Module)` with an `__init__` method that builds the layers and a `forward` method that defines the data flow.
2. Instantiate the model and print it to verify the architecture.

> **Tip:** `nn.Sequential` can help you chain layers cleanly inside `__init__`.

In [8]:
```python
input_dim = X_train.shape[1]

class CustomerClassifier(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.network(x).squeeze(1)

model = CustomerClassifier(input_dim)
print(model)
print(f"\nInput dimension: {input_dim}")
```

```
CustomerClassifier(
  (network): Sequential(
    (0): Linear(in_features=28, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=32, bias=True)
    (3): ReLU()
    (4): Linear(in_features=32, out_features=1, bias=True)
    (5): Sigmoid()
  )
)

Input dimension: 28
```

Step 6 — Define Loss Function and Optimiser

For binary classification with a sigmoid output, the standard choice is **Binary Cross-Entropy loss**.

Tasks:

1. Instantiate `nn.BCELoss()` as your loss function.
2. Instantiate `torch.optim.Adam` with a learning rate of `0.001` as your optimiser, passing `model.parameters()`.

> **Why Adam?** Adam adapts the learning rate per parameter and generally converges faster than plain SGD on tabular data.

In [9]:
```python
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Step 7 — Write the Training Loop

Train the model for **30 epochs**. For each epoch you should:

1. Set the model to training mode with `model.train()`.
2. Iterate over batches from `train_loader`.
3. For each batch:
   - Zero the gradients with `optimizer.zero_grad()`.
   - Run a **forward pass** to get predictions.
   - Compute the **loss**.
   - Run a **backward pass** with `loss.backward()`.
   - Update the weights with `optimizer.step()`.
4. After all batches, record the average epoch training loss.
5. Run a **validation pass** (no gradient computation) over `test_loader` and record the average test loss.
6. Print the losses every 5 epochs.

Store training and test losses in lists so you can plot them in the next step.

In [10]:
```
NUM_EPOCHS = 30
train_losses = []
test_losses  = []

for epoch in range(1, NUM_EPOCHS + 1):
    model.train()
    epoch_loss = 0.0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        preds = model(X_batch)
        loss  = criterion(preds, y_batch)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item() * len(X_batch)
    train_losses.append(epoch_loss / len(train_loader.dataset))

    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            preds    = model(X_batch)
            val_loss += criterion(preds, y_batch).item() * len(X_batch)
    test_losses.append(val_loss / len(test_loader.dataset))

    if epoch % 5 == 0:
        print(f"Epoch {epoch:3d} | Train Loss: {train_losses[-1]:.4f} | Test
```

```
Epoch   5 | Train Loss: 0.2388 | Test Loss: 0.2612
Epoch  10 | Train Loss: 0.2203 | Test Loss: 0.2574
Epoch  15 | Train Loss: 0.2125 | Test Loss: 0.2549
Epoch  20 | Train Loss: 0.2028 | Test Loss: 0.2644
```

```
Epoch  25 | Train Loss: 0.1940 | Test Loss: 0.2741
Epoch  30 | Train Loss: 0.1861 | Test Loss: 0.2756
```

Step 8 — Visualise Training Progress

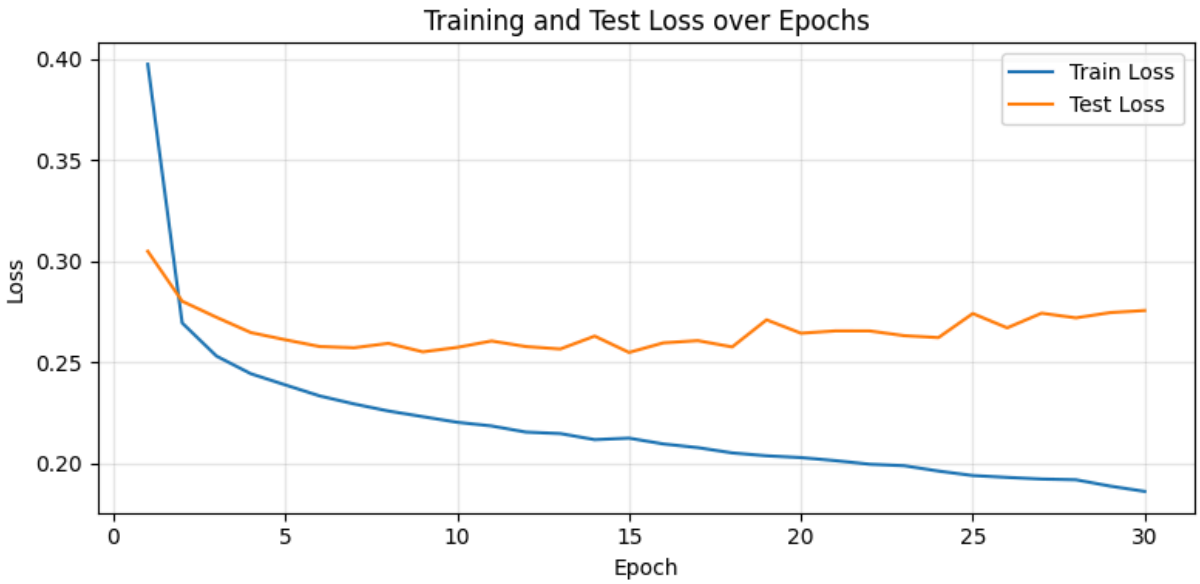
Plot the training and test loss curves over epochs.

Tasks:

1. Create a line plot with epochs on the x-axis and loss on the y-axis.
2. Show both training loss and test loss on the same plot with a legend.
3. Add axis labels and a title.

> **What to look for:** If training loss decreases but test loss plateaus or rises, the model may be overfitting.

In [11]:
```
plt.figure(figsize=(8, 4))
plt.plot(range(1, NUM_EPOCHS + 1), train_losses, label='Train Loss')
plt.plot(range(1, NUM_EPOCHS + 1), test_losses,  label='Test Loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training and Test Loss over Epochs")
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



Step 9 — Evaluate the Model

A single accuracy number is not enough for an imbalanced dataset. Evaluate using a full set of metrics.

Tasks:

1. Set the model to evaluation mode with `model.eval()`.
2. Run inference over the entire test set (disable gradient tracking with `torch.no_grad()`).
3. Convert predicted probabilities to binary labels using a threshold of 0.5.
4. Compute and print:
   - **Accuracy**
   - **Precision**
   - **Recall**
   - **F1-score**
5. Print a **classification report** using `sklearn.metrics.classification_report`.

> **Tip:** Use `sklearn.metrics` functions. Remember to move tensors to CPU and convert to NumPy before passing to sklearn.

In [12]:
```
model.eval()
all_preds = []
all_true  = []

with torch.no_grad():
    for X_batch, y_batch in test_loader:
        probs = model(X_batch)
        preds = (probs >= 0.5).float()
        all_preds.append(preds.cpu().numpy())
        all_true.append(y_batch.cpu().numpy())

all_preds = np.concatenate(all_preds)
all_true  = np.concatenate(all_true)

acc  = accuracy_score(all_true, all_preds)
prec = precision_score(all_true, all_preds)
rec  = recall_score(all_true, all_preds)
f1   = f1_score(all_true, all_preds)

print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print("\nClassification Report:")
print(classification_report(all_true, all_preds, target_names=['No Purchase'
```

```
Accuracy : 0.8913
Precision: 0.7036
Recall   : 0.5157
F1-Score : 0.5952

Classification Report:
              precision    recall  f1-score   support

 No Purchase       0.92      0.96      0.94      2084
    Purchase       0.70      0.52      0.60       382

    accuracy                           0.89      2466
   macro avg       0.81      0.74      0.77      2466
weighted avg       0.88      0.89      0.88      2466
```