Code Description

Contents

I. Batch Normalization

```
import torch
import torch.nn as nn
def demo():
    torch.manual_seed(0)
    dummy_input = torch.randn(5, 3) * 10 + 5
    print(dummy_input)
    print("Before Batch Norm:")
    print("Mean:\n", dummy_input.mean(dim=0))
    print("Variance:\n", dummy_input.var(dim=0, unbiased=False))
    batch_norm = nn.BatchNorm1d(num_features=3)
    output = batch_norm(dummy_input)
    print("After Batch Norm:")
    print("Mean:\n", output.mean(dim=0).detach())
    print("Variance:\n", output.var(dim=0,
unbiased=False).detach())
if __name__ == '__main__':
    demo()
```

Explanation:

```
torch.manual_seed(0)
```

Sets a fixed “seed” for the random number generator. This ensures that every time we run the code, we will get the exact same random numbers, making the results reproducible.

```
dummy_input = torch.randn(5, 3) * 10 + 5
```

Creates a tensor (matrix) of shape (5, 3).

5: The Batch Size (number of samples).

3: The Number of Features (e.g., Income, Age, Debt).

The Math: torch.randn generates numbers with a mean (μ) of 0 and variance (σ^2) of 1. By multiplying by 10 and adding 5, we simulate raw, unscaled data where the mean is roughly 5 and the variance is roughly 100.

```
print("Mean:\n", dummy_input.mean(dim=0))
print("Variance:\n", dummy_input.var(dim=0, unbiased=False))
```

`dim=0`: Tells PyTorch to calculate the statistics column-wise (across the 5 samples for each of the 3 features). This is how Batch Norm looks at.

`data.unbiased=False`: Forces PyTorch to calculate the population variance (dividing by n). This matches the internal mathematical formula used by the BatchNorm layer.

```
batch_norm = nn.BatchNorm1d(num_features=3)
```

Initializes the 1D Batch Normalization layer.

`num_features=3`: This must match the number of columns in input. It creates two learnable parameters: γ (Scale, initialized to 1) and $b\eta$ (Shift, initialized to 0).

II. Group Normalization

```
import torch
import torch.nn as nn
def demo():
    sample = torch.tensor([
        [20.4100, 2.0657, -16.7879],
        [10.6843, -5.8452, -8.9860],
        [9.0335, 13.3803, -2.1926],
        [0.9666, -0.9664, 6.8204],
        [-3.5667, 16.0060, -5.7119],
    ], dtype=torch.float32)
    print(sample)
    print("Before Group Norm")
    mean_before = sample.mean(dim=1)
    var_before = sample.var(dim=1, unbiased=False)
    print("Mean:\n", mean_before)
    print("Variance:\n", var_before)
    gn = nn.GroupNorm(num_groups=1, num_channels=3)
    output = gn(sample)
    print(output.detach())
    print("After Group Norm")
    print("Mean:\n", output.mean(dim=1).detach())
```

```
    print("Variance:\n", output.var(dim=1,
unbiased=False).detach())
if __name__ == '__main__':
    demo()
```

Explanation:

```
sample = torch.tensor([
    [20.4100, 2.0657, -16.7879],
    ...
], dtype=torch.float32)
```

We initialize a 5x3 matrix representing a mini-batch of 5 samples. Each sample contains 3 features with significantly different numerical scales. We use float32 to ensure the data is compatible with PyTorch's gradient descent engines.

```
mean_before = sample.mean(dim=1)
var_before = sample.var(dim=1, unbiased=False)
```

`dim=1`: While Batch Norm used `dim=0` (columns), Group Norm uses `dim=1` (rows).

Logic: GN calculates the mean and variance for each individual sample across its own channels. It doesn't care about other samples in the batch.

III. Standardization

```
import numpy as np
def construct():
    data = np.array([70, 80, 90, 100, 110], dtype=float)
    mean = np.mean(data)
    std_dev = np.std(data)
    z_scores = (data - mean) / std_dev
    print("Original data:", data)
    print("Mean:", mean)
    print("Standard Deviation:", std_dev)
    print("Z-scores:", z_scores)
if __name__ == "__main__":
    construct()
```

Explanation

`std_dev = np.std(data)`: it is used to calculate the standard deviation of the given array (70 80 90 100 110)

IV. Steps in Data Preprocessing

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA

data_main = {
    'CustomerID': [1, 2, 3, 4, 5],
    'Age': [25, 400, 22, 25, 45],
    'Income': [50000, 60000, -3, 50000, 80000]
}
df = pd.DataFrame(data_main)

print("Original Data")
print(df, "\n")

#Cleaning
df['Age'] = np.where((df['Age'] < 0) | (df['Age'] > 100), np.nan,
df['Age'])
df['Income'] = np.where(df['Income'] < 0, np.nan, df['Income'])
df['Age'] = df['Age'].fillna(df['Age'].mean())
df['Income'] = df['Income'].fillna(df['Income'].median())

print("After cleaning")
print(df, "\n")

#Integration
data_sales = {
    'CustomerID': [1, 2, 3, 4, 5],
    'Country': ['VN', 'US', 'JP', 'VN', 'JP'],
    'Total_Spent': [100, 250, 300, 150, 400]
}
df_sales = pd.DataFrame(data_sales)
df = pd.merge(df, df_sales, on='CustomerID', how='inner')

print("After integration (merge with sales data):")
print(df, "\n")

#Discretization
bins = [0, 18, 35, 55, 120]
labels = ['Child', 'Young Adult', 'Middle-Aged', 'Senior']
df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels)

print("After discretization (Age -> Age_Group):")
```

```
print(df[['Age', 'Age_Group']], "\n")

#Transformation
df = pd.get_dummies(df, columns=['Country', 'Age_Group'],
dtype=int)
scaler = MinMaxScaler()
df[['Income', 'Total_Spent']] = scaler.fit_transform(df[['Income',
'Total_Spent']])

print("After transformation (One-hot encoding và Min-Max
Scaling):")
print(df, "\n")

#Reduction
features = df.drop(columns=['CustomerID', 'Age'])
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(features)
df_final = pd.DataFrame(reduced_data, columns=['PC1', 'PC2'])
df_final['CustomerID'] = df['CustomerID']

print("After PCA reduction")
print(df_final)
```

Explanation:

`df = pd.DataFrame(data_main)`: it is used to create a 2d labeled array from inputted data $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 25 & 400 & 22 & 25 & 45 \\ 50000 & 60000 & -3 & 50000 & 80000 \end{pmatrix}$.

`df['Age'] = np.where((df['Age'] < 0) | (df['Age'] > 100), np.nan, df['Age'])`: If any value of the matrix related to "Age" is larger than 100 and smaller than 0, it will be nan value.

`df['Age'] = df['Age'].fillna(df['Age'].mean())`: The previous nan value will be replaced by the mean of the remaining "Age" value.

The remaining lines in `#Cleaning` have similar function to previously mentioned ones.

`df = pd.merge(df, df_sales, on='CustomerID', how='inner')`: it is used to merge two 2d arrays `df` and `df_sales`; `'CustomerID'` is the common identifier.

`df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels)`: A new 2d array named `Age_Group` is created with original values from `Age`, those values then are divided into groups depending on `pins` and `labels`.

`df = pd.get_dummies(df, columns=['Country', 'Age_Group'], dtype=int)`: `get_dummies` is used to convert table data into binary columns.

`df[['Income', 'Total_Spent']] = scaler.fit_transform(df[['Income', 'Total_Spent']])`: Data concerning to `Income` and `Total_Spent` now varies from 0 to 1.