Assignment 5

Parker Christenson

`Customer Segmentation Analysis with Boltzmann Machines Based on Online Retail Shopping Habits`

---

1. Load the Online Retail II dataset and clean the data by removing incomplete entries.
2. Preprocess the data by encoding categorical data and scaling numerical features to normalize the range of data values.
3. Transform the data into a suitable format where each customer's shopping habits over time are captured in a binary format—purchased or not purchased.
4. Train your Boltzmann machine using the training set with the goal of learning the underlying probability distribution of the data.

In [ ]:
```python
# imports

import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import polars as pl
from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

In [ ]:
```python
# Read first sheet
df1_polars = pl.read_excel('online_retail_II.xlsx')
df1_pandas = df1_polars.to_pandas()

# Read second sheet
df2_polars = pl.read_excel('online_retail_II.xlsx', sheet_name='Year 2010-2011')
df2_pandas = df2_polars.to_pandas()

# Combine both dataframes
df = pd.concat([df1_pandas, df2_pandas])
df.head()
```

Out[ ]:

	Invoice	StockCode	Description	Quantity	InvoiceDate	Price	Customer ID	Country
0	489434.0	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	2009-12-01 07:45:00	6.95	13085.0	United Kingdom
1	489434.0	79323P	PINK CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
2	489434.0	79323W	WHITE CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
3	489434.0	22041	RECORD FRAME 7" SINGLE SIZE	48	2009-12-01 07:45:00	2.10	13085.0	United Kingdom
4	489434.0	21232	STRAWBERRY CERAMIC TRINKET BOX	24	2009-12-01 07:45:00	1.25	13085.0	United Kingdom

In [ ]:
```
# null values
df.isnull().sum()
```

Out[ ]:
```
Invoice         19500
StockCode           0
Description      4382
Quantity            0
InvoiceDate         0
Price               0
Customer ID    243007
Country             0
dtype: int64
```

In [ ]:
```
# drop all rows with null values
df = df.dropna()
```

In [ ]:
```
# binary encoding
df['Purchased'] = 1
```

In [ ]:
```
# pivot the data
pivot_data = df.pivot_table(index='Customer ID', columns='StockCode', values='Purch
```

In [ ]:
```
# scale
scaler = StandardScaler()
scaled_pivot_data = scaler.fit_transform(pivot_data)
```

In [ ]:
```
# convert scaled into data frame
scaled_pivot_data = pd.DataFrame(scaled_pivot_data, index=pivot_data.index, columns

# another check for nulls
if pd.isna(scaled_pivot_data).any().any():
```

```
    raise ValueError("NaN detected in scaled input data")

# df.head
scaled_pivot_data.head()
```

Out[ ]:

StockCode	10002	10080	10109	10120	10123C	10123G	10124A	1012
Customer ID								
12346.0	-0.169371	-0.06266	-0.013041	-0.094451	-0.081706	-0.045218	-0.055408	-0.03
12347.0	-0.169371	-0.06266	-0.013041	-0.094451	-0.081706	-0.045218	-0.055408	-0.03
12348.0	-0.169371	-0.06266	-0.013041	-0.094451	-0.081706	-0.045218	-0.055408	-0.03
12349.0	-0.169371	-0.06266	-0.013041	-0.094451	-0.081706	-0.045218	-0.055408	-0.03
12350.0	-0.169371	-0.06266	-0.013041	-0.094451	-0.081706	-0.045218	-0.055408	-0.03

5 rows × 4631 columns

In [ ]:
```
# convert to tensor
scaled_pivot_data = scaled_pivot_data.values
scaled_pivot_data = torch.tensor(scaled_pivot_data, dtype=torch.float32)
```

In [ ]:
```
# define the model
class RBM(nn.Module):
    def __init__(self, n_visible, n_hidden):
        super(RBM, self).__init__()
        self.n_visible = n_visible
        self.n_hidden = n_hidden

        # Xavier Initialization <-- I started to use lots of initalization methods,
        self.W = nn.Parameter(torch.randn(n_hidden, n_visible) * torch.sqrt(torch.t
        self.h_bias = nn.Parameter(torch.zeros(n_hidden))
        self.v_bias = nn.Parameter(torch.zeros(n_visible))

    def sample_from_p(self, p):
        return torch.bernoulli(p)

    def v_to_h(self, v):
        p_h = torch.sigmoid(torch.matmul(v, self.W.t()) + self.h_bias)
        return p_h, self.sample_from_p(p_h)

    def h_to_v(self, h):
        p_v = torch.sigmoid(torch.matmul(h, self.W) + self.v_bias)
        return p_v, self.sample_from_p(p_v)

    def forward(self, v):
        p_h, h = self.v_to_h(v)
        p_v, v = self.h_to_v(h)
        return v
```

```
    def free_energy(self, v):
        v_term = torch.matmul(v, self.v_bias)
        w_x_h = torch.matmul(v, self.W.t()) + self.h_bias
        h_term = torch.sum(torch.log(1 + torch.exp(w_x_h)), dim=1)
        return -v_term - h_term
```

In [ ]:
```
# define the model
n_visible = scaled_pivot_data.shape[1]
n_hidden = 256
rbm = RBM(n_visible, n_hidden)

# training the model
n_epochs = 10
batch_size = 64
learning_rate = 0.001  # Further reduced learning rate

# setting the optimizer
optimizer = optim.SGD(rbm.parameters(), lr=learning_rate)
```

In [ ]:
```
# custom loop with lots of debugging statements
for epoch in range(n_epochs):
    train_loss = 0
    for i in range(0, len(scaled_pivot_data), batch_size):
        batch = scaled_pivot_data[i:i+batch_size]
        if len(batch) != batch_size:
            continue

        # postive phase
        v0 = batch
        ph0, h0 = rbm.v_to_h(v0)

        # check for nans in pos phase
        if torch.isnan(ph0).any() or torch.isnan(h0).any():
            print("NaN detected in positive phase")
            break

        # negative phase
        vk = v0
        for k in range(1):
            _, hk = rbm.v_to_h(vk)
            _, vk = rbm.h_to_v(hk)

        phk, _ = rbm.v_to_h(vk)

        # check to see if the negative phasae has NaNs
        if torch.isnan(phk).any() or torch.isnan(vk).any():
            print("NaN detected in negative phase")
            break

        positive_phase = torch.matmul(h0.t(), v0)
        negative_phase = torch.matmul(phk.t(), vk)

        # update gradients
        rbm.W.grad = (positive_phase - negative_phase) / batch_size
```

```
        rbm.v_bias.grad = torch.sum(v0 - vk, dim=0) / batch_size
        rbm.h_bias.grad = torch.sum(ph0 - phk, dim=0) / batch_size

        # clip the gradients to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(rbm.parameters(), max_norm=1)

        optimizer.step()

        train_loss += torch.mean(rbm.free_energy(v0)) - torch.mean(rbm.free_energy(

    # nans in train loss causes loop to eand early
    if torch.isnan(train_loss).any():
        print("NaN detected in train_loss")
        break

    print(f'Epoch {epoch+1}/{n_epochs}, Loss: {train_loss.item()}')
```

```
Epoch 1/10, Loss: 3741.68505859375
Epoch 2/10, Loss: 6398.607421875
Epoch 3/10, Loss: 9253.9833984375
Epoch 4/10, Loss: 12266.40234375
Epoch 5/10, Loss: 15445.40234375
Epoch 6/10, Loss: 18800.091796875
Epoch 7/10, Loss: 22314.486328125
Epoch 8/10, Loss: 25941.3984375
Epoch 9/10, Loss: 29682.236328125
Epoch 10/10, Loss: 33581.9453125
```

In [ ]: