code

March 21, 2025

0.1 *Assignment 2*

[19]:
```
import torch
torch.cuda.empty_cache()
```

Import neccessary libraries and load the ratings

[20]:
```
import pandas as pd

# Cargar el archivo ratings.csv
ratings = pd.read_csv('ml-latest-small/ratings.csv')

# Ver las primeras filas
print(ratings.head())

# Revisar tamaño y columnas
print(ratings.shape)
print(ratings.columns)
```

```
   userId  movieId  rating  timestamp
0       1        1     4.0  964982703
1       1        3     4.0  964981247
2       1        6     4.0  964982224
3       1       47     5.0  964983815
4       1       50     5.0  964982931
(100836, 4)
Index(['userId', 'movieId', 'rating', 'timestamp'], dtype='object')
```

Preproccessing

[21]:
```
# Filtrar usuarios con menos de 10 ratings
user_counts = ratings['userId'].value_counts()
ratings = ratings[ratings['userId'].isin(user_counts[user_counts >= 10].index)]

# Filtrar películas con menos de 10 ratings
movie_counts = ratings['movieId'].value_counts()
ratings = ratings[ratings['movieId'].isin(movie_counts[movie_counts >= 10].
 ↪index)]

print(f"Usuarios después de filtrar: {ratings['userId'].nunique()}")
```

```
print(f"Películas después de filtrar: {ratings['movieId'].nunique()}")
```

```
Usuarios después de filtrar: 610
Películas después de filtrar: 2269
```

[22]:
```
# Obtener IDs únicos
unique_user_ids = ratings['userId'].unique()
unique_movie_ids = ratings['movieId'].unique()

print(f"Número de usuarios únicos: {len(unique_user_ids)}")
print(f"Número de películas únicas: {len(unique_movie_ids)}")

# Crear diccionarios de mapeo
userId_to_index = {user_id: idx for idx, user_id in enumerate(unique_user_ids)}
movieId_to_index = {movie_id: idx for idx, movie_id in␣
 ↪enumerate(unique_movie_ids)}

# Aplicar el mapeo al DataFrame
ratings['userIndex'] = ratings['userId'].map(userId_to_index)
ratings['movieIndex'] = ratings['movieId'].map(movieId_to_index)

# Comprobar
print(ratings.head())
```

```
Número de usuarios únicos: 610
Número de películas únicas: 2269
   userId  movieId  rating  timestamp  userIndex  movieIndex
0       1        1     4.0  964982703          0           0
1       1        3     4.0  964981247          0           1
2       1        6     4.0  964982224          0           2
3       1       47     5.0  964983815          0           3
4       1       50     5.0  964982931          0           4
```

[23]:
```
# Normalizamos ratings a [0, 1]
ratings['rating_norm'] = ratings['rating'] / 5.0
print(ratings[['rating', 'rating_norm']].head())
```

```
   rating  rating_norm
0     4.0          0.8
1     4.0          0.8
2     4.0          0.8
3     5.0          1.0
4     5.0          1.0
```

Split and Prepare

[24]:
```
from sklearn.model_selection import train_test_split

# Primero filtramos usuarios con al menos 3 ratings
```

```
user_counts = ratings['userId'].value_counts()
ratings_filtered = ratings[ratings['userId'].isin(user_counts[user_counts >= 3].
 ↪index)]

# Luego aplicamos el split
train_list = []
val_list = []
test_list = []

for user_id, group in ratings_filtered.groupby('userId'):
    user_train, user_temp = train_test_split(group, test_size=0.30,␣
 ↪random_state=42)
    user_val, user_test = train_test_split(user_temp, test_size=0.50,␣
 ↪random_state=42)

    train_list.append(user_train)
    val_list.append(user_val)
    test_list.append(user_test)

train_data = pd.concat(train_list).reset_index(drop=True)
val_data = pd.concat(val_list).reset_index(drop=True)
test_data = pd.concat(test_list).reset_index(drop=True)

print(f"Train size: {len(train_data)}")
print(f"Validation size: {len(val_data)}")
print(f"Test size: {len(test_data)}")
```

```
Train size: 56505
Validation size: 12164
Test size: 12447
```

Dataloaders

[25]:
```
import torch
from torch.utils.data import Dataset, DataLoader

# Convertir a tensores los índices de usuario, película y ratings
train_user = torch.tensor(train_data['userIndex'].values, dtype=torch.long)
train_movie = torch.tensor(train_data['movieIndex'].values, dtype=torch.long)
train_rating = torch.tensor(train_data['rating_norm'].values, dtype=torch.
 ↪float32)

val_user = torch.tensor(val_data['userIndex'].values, dtype=torch.long)
val_movie = torch.tensor(val_data['movieIndex'].values, dtype=torch.long)
val_rating = torch.tensor(val_data['rating_norm'].values, dtype=torch.float32)

test_user = torch.tensor(test_data['userIndex'].values, dtype=torch.long)
test_movie = torch.tensor(test_data['movieIndex'].values, dtype=torch.long)
```

```
test_rating = torch.tensor(test_data['rating_norm'].values, dtype=torch.float32)
```

Dataset Personalizado

[26]:
```
class MovieLensDataset(Dataset):
    def __init__(self, users, movies, ratings):
        self.users = users
        self.movies = movies
        self.ratings = ratings

    def __len__(self):
        return len(self.ratings)

    def __getitem__(self, idx):
        return {
            'user': self.users[idx],
            'movie': self.movies[idx],
            'rating': self.ratings[idx]
        }
```

Dataloaders

[27]:
```
batch_size = 512

train_dataset = MovieLensDataset(train_user, train_movie, train_rating)
val_dataset = MovieLensDataset(val_user, val_movie, val_rating)
test_dataset = MovieLensDataset(test_user, test_movie, test_rating)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

Simple Model

[28]:
```
import torch.nn as nn
import torch.nn.functional as F

class NeuralCollaborativeFiltering(nn.Module):
    def __init__(self, num_users, num_movies, embedding_dim=64, dropout_rate=0.
 ↪3):
        super(NeuralCollaborativeFiltering, self).__init__()

        self.user_embedding = nn.Embedding(num_users, embedding_dim)
        self.movie_embedding = nn.Embedding(num_movies, embedding_dim)

        # MLP con Dropout
        self.fc1 = nn.Linear(embedding_dim * 2, 128)
        self.fc2 = nn.Linear(128, 64)
        self.output_layer = nn.Linear(64, 1)
```

```
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, user, movie):
        user_embedded = self.user_embedding(user)
        movie_embedded = self.movie_embedding(movie)

        x = torch.cat([user_embedded, movie_embedded], dim=1)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        out = self.output_layer(x)

        return out.squeeze()
```

Optimizer

[29]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Usando dispositivo: {device}")
```

```
Usando dispositivo: cuda
```

[30]:
```
num_users = len(userId_to_index)
num_movies = len(movieId_to_index)

model = NeuralCollaborativeFiltering(num_users=num_users,␣
 ↪num_movies=num_movies).to(device)
print("Modelo con Dropout creado correctamente")
```

```
Modelo con Dropout creado correctamente
```

[31]:
```
import torch.optim as optim
import torch.nn as nn

criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=0.0005, weight_decay=1e-5)
```

Trainning

[32]:
```
num_epochs = 10  # Puedes ajustar luego

for epoch in range(num_epochs):
    model.train()
    total_loss = 0

    for batch in train_loader:
        users = batch['user'].to(device)
```

```
        movies = batch['movie'].to(device)
        ratings = batch['rating'].to(device)  # ← ya es rating_norm

        preds = model(users, movies)
        loss = criterion(preds, ratings)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * len(ratings)

    avg_train_loss = total_loss / len(train_loader.dataset)

    # Validation
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for batch in val_loader:
            users = batch['user'].to(device)
            movies = batch['movie'].to(device)
            ratings = batch['rating'].to(device)

            preds = model(users, movies)
            loss = criterion(preds, ratings)
            val_loss += loss.item() * len(ratings)

    avg_val_loss = val_loss / len(val_loader.dataset)

    print(f"Epoch {epoch+1}: Train Loss = {avg_train_loss:.4f}, Val Loss =␣
↪{avg_val_loss:.4f}")
```

```
Epoch 1: Train Loss = 0.1400, Val Loss = 0.0432
Epoch 2: Train Loss = 0.0605, Val Loss = 0.0403
Epoch 3: Train Loss = 0.0547, Val Loss = 0.0388
Epoch 4: Train Loss = 0.0514, Val Loss = 0.0369
Epoch 5: Train Loss = 0.0493, Val Loss = 0.0359
Epoch 6: Train Loss = 0.0472, Val Loss = 0.0358
Epoch 7: Train Loss = 0.0458, Val Loss = 0.0358
Epoch 8: Train Loss = 0.0449, Val Loss = 0.0345
Epoch 9: Train Loss = 0.0434, Val Loss = 0.0345
Epoch 10: Train Loss = 0.0423, Val Loss = 0.0341
```

[15]:
```
torch.save(model.state_dict(), "ncf_model_current.pth")
print("Modelo guardado correctamente.")
```

```
Modelo guardado correctamente.
```

[ ]:
```
model = NeuralCollaborativeFiltering(num_users, num_movies)  # misma clase
model.load_state_dict(torch.load("ncf_model_current.pth"))
model.to(device)  # si estabas en GPU
```

Model evaluation

[33]:
```
model.eval()
test_loss = 0
with torch.no_grad():
    for batch in test_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        ratings = batch['rating'].to(device)

        preds = model(users, movies)
        loss = criterion(preds, ratings)
        test_loss += loss.item() * len(ratings)

avg_test_loss = test_loss / len(test_loader.dataset)
print(f"Test Loss = {avg_test_loss:.4f}")
```

```
Test Loss = 0.0350
```

[34]:
```
# Evaluación en test
model.eval()
test_loss = 0
with torch.no_grad():
    for batch in test_loader:
        users = batch['user'].to(device)
        movies = batch['movie'].to(device)
        ratings = batch['rating'].to(device)

        preds = model(users, movies)
        loss = criterion(preds, ratings)
        test_loss += loss.item() * len(ratings)

avg_test_loss = test_loss / len(test_loader.dataset)

# RMSE (recordar que está normalizado, multiplicamos por 5)
import numpy as np
rmse = np.sqrt(avg_test_loss) * 5
print(f"Test RMSE = {rmse:.4f}")
```

```
Test RMSE = 0.9358
```

[35]:
```
def calculate_accuracy(model, data_loader):
    model.eval()
    correct = 0
    total = 0
```

```
    with torch.no_grad():
        for batch in data_loader:
            users = batch['user'].to(device)
            movies = batch['movie'].to(device)
            ratings = batch['rating'].to(device) * 5  # Desnormalizamos a␣
↪escala 0.5 - 5

            preds = model(users, movies) * 5  # También desnormalizamos
            preds_rounded = torch.round(preds * 2) / 2  # Redondeamos a␣
↪múltiplos de 0.5

            correct += torch.sum(preds_rounded == torch.round(ratings * 2) / 2).
↪item()
            total += ratings.size(0)

    accuracy = correct / total
    print(f"Accuracy: {accuracy:.4f}")

# Ejemplo en test:
calculate_accuracy(model, test_loader)
```

```
Accuracy: 0.2082
```

1 Definición del Mejor Modelo para el Sistema de Recomendación

1.1 Objetivo

Desarrollar un sistema de recomendación capaz de predecir con alta precisión las calificaciones (ratings) que un usuario daría a una película, y utilizar esta predicción para recomendar nuevas películas que probablemente le gusten.

1.2 Arquitectura del Modelo

1.2.1 1 Embeddings Aprendidos para Usuarios y Películas

- Cada usuario y cada película es representada por un vector (embedding) de dimensiones elevadas (por ejemplo, 200 o 300).
- Estos embeddings capturan características latentes sobre las preferencias de los usuarios y las propiedades de las películas.

1.2.2 2 Concatenación de Embeddings

- En lugar de calcular simplemente el producto punto entre los embeddings de usuario y película, **concatenamos ambos vectores**.

- Esto permite conservar toda la información individual de usuario y película antes de pasarlos por la red neuronal.

1.2.3 3 Red Neuronal Multicapa (MLP)

- La concatenación se pasa por varias capas densas (fully connected) con activaciones **ReLU**.
- Esto permite que el modelo aprenda relaciones **no lineales y complejas** entre usuarios y películas.
- Ejemplo: Dos capas ocultas con 128 y 64 neuronas respectivamente.

1.2.4 4 Regularización

- Uso de **Dropout** entre capas para prevenir overfitting.
- Aplicación de **Weight Decay** en el optimizador (Adam) para controlar el crecimiento excesivo de los pesos.

1.2.5 5 Información de Contenido (Opcional para mejora futura)

- Inclusión de información adicional como:
  - **Géneros** de las películas (usando one-hot encoding o embeddings).
  - **Tags** generados por usuarios (requiere procesamiento adicional).
- Estos datos se pueden concatenar junto a los embeddings de usuario y película para enriquecer la entrada al modelo.

1.2.6 6 Función de Pérdida

- Se utilizará **MSELoss (Mean Squared Error Loss)** ya que estamos prediciendo ratings continuos en un rango de 0.5 a 5 estrellas.
