🎬 CGAN-Based Movie Recommender System - Complete Code Explanation

This document provides a **beginner-friendly explanation** of the entire project, including PyTorch concepts, architecture details, and the complete flow of how the system works.

📖 Table of Contents

1. What is This Project?
2. Key Concepts Before We Start
3. Project Architecture Overview
4. Complete Code Flow
5. PyTorch Basics Used in This Project
6. Detailed Code Explanation
7. Training Process Explained
8. How Recommendations Are Generated
9. Evaluation Metrics
10. File Structure

🎯 What is This Project?

This is a **Movie Recommendation System** that suggests movies to users based on their past preferences.

The Traditional Approach

Most recommendation systems use **collaborative filtering** - finding users similar to you and recommending what they liked. Think of it like: "Users who liked Movie A also liked Movie B."

Our Approach: Collaborative GAN (CGAN)

We use a **Generative Adversarial Network (GAN)** - a type of AI where two neural networks compete against each other:

1. **Generator (G)**: Tries to "guess" which movies a user might like

2. **Discriminator (D)**: Tries to tell if a user-movie pair is a real interaction or a fake guess

Through this competition, both networks get better - the Generator learns to make realistic recommendations, and the Discriminator learns to identify real preferences.

🧠 Key Concepts Before We Start

What is PyTorch?

PyTorch is a Python library for building and training neural networks. Think of it as LEGO blocks for AI - you build models by connecting different pieces together.

What is a Neural Network?

A neural network is a series of mathematical operations that transform input data into useful outputs. It "learns" by adjusting its internal numbers (called **weights**) based on examples.

What is an Embedding?

An **embedding** is a way to represent something (like a user or movie) as a list of numbers (a vector).

Example: - User 1 might be represented as `[0.5, -0.2, 0.8, 0.1, ...]` - User 2 might be represented as `[0.3, 0.7, -0.4, 0.2, ...]`

Similar users will have similar number patterns. The network learns these patterns during training.

What is Implicit Feedback?

Instead of using star ratings directly (1-5 stars), we convert them to binary: - Rating ≥ 4 → User **liked** this movie (1) - Rating < 4 → Ignored (0)

This simplifies the problem to: "Did the user like this movie or not?"

🏗️ Project Architecture Overview

```
┌─────────────────────────────────────────────────────────────┐
│                         INPUT DATA                          │
│                   MovieLens 100K Dataset                    │
│            (user_id, movie_id, rating, timestamp)           │
```

```
└─────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────────┐
│                      DATA PREPROCESSING                          │
│  1. Convert ratings to implicit feedback (liked/not liked)       │
│  2. Create user and movie ID mappings                            │
│  3. Split into train/test sets                                   │
└─────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────────┐
│                       CGAN ARCHITECTURE                          │
│                                                                   │
│  ┌──────────────────────┐      ┌──────────────────────┐         │
│  │      GENERATOR       │      │   DISCRIMINATOR      │         │
│  │                      │      │                      │         │
│  │ Input:               │      │ Input:               │         │
│  │ - User Embedding     │      │ - User Embedding     │         │
│  │ - Random Noise       │      │ - Movie Embedding    │         │
│  │                      │      │                      │         │
│  │ Output:              │      │ Output:              │         │
│  │ - Scores for ALL     │      │ - Probability        │         │
│  │   movies             │      │   (real or fake?)    │         │
│  └──────────────────────┘      └──────────────────────┘         │
│              │                            ▲                      │
│              │    Adversarial Training    │                      │
│              └────────────────────────────┘                      │
└─────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────────┐
│                     RECOMMENDATION                               │
│  1. Generator produces scores for all movies                     │
│  2. Remove movies user already watched                           │
│  3. Return Top-K highest scoring movies                          │
└─────────────────────────────────────────────────────────────────┘
```

🔄 Complete Code Flow

Here's exactly what happens when you run `python main.py`:

Step 1: Setup & Configuration

```
 # Parse command line arguments
parser = argparse.ArgumentParser()
parser.add_argument("--epochs", type=int, default=10)  # How many times to train
parser.add_argument("--k", type=int, default=10)       # How many movies to recommend
# ... more arguments
```

Step 2: Load the Dataset

```
 # Extract the MovieLens zip file
ml_dir = maybe_extract_zip(zip_path, data_dir)

# Read movie titles (e.g., "Toy Story (1995)")
movieid_to_title = read_u_item(ml_dir)

# Read user-movie-rating data
train_raw, test_raw = read_movielens_split(ml_dir, split="u1")
```

Step 3: Preprocess Data

```
 # Create mappings: original IDs → sequential indices (0, 1, 2, ...)
user2idx, item2idx = build_id_maps(train_raw, test_raw)

# Convert ratings to implicit feedback
# Result: {user_idx: {set of liked movie indices}}
train_pos, test_pos = to_implicit_sets(train_raw, test_raw, ...)
```

Step 4: Train the CGAN

```
 G, D = train_cgan(train_pos, users_train, num_users, num_items, device, cfg)
```

Step 5: Evaluate & Recommend

```
 # Calculate metrics (Recall, NDCG, HitRate)
evaluate(G, train_pos, test_pos, users_eval, ...)
```

```
# Generate recommendations for users
recs = recommend_for_user(G, user_id, num_items, seen_movies, k=10, ...)
```

🔧 PyTorch Basics Used in This Project

1. Tensors

Tensors are multi-dimensional arrays (like NumPy arrays but can run on GPU).

```
import torch

# Create a tensor
x = torch.tensor([1, 2, 3])  # 1D tensor

# Create a random tensor
noise = torch.randn(32)  # 32 random numbers from normal distribution
```

2. nn.Module (Building Neural Networks)

All neural networks in PyTorch inherit from `nn.Module`:

```
import torch.nn as nn

class MyNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        # Define layers here
        self.layer1 = nn.Linear(10, 20)  # 10 inputs → 20 outputs

    def forward(self, x):
        # Define how data flows through the network
        return self.layer1(x)
```

3. nn.Embedding (Looking Up Vectors)

Embeddings convert IDs into learned vector representations:

```
# Create an embedding layer for 100 users, each represented by 64 numbers
user_embedding = nn.Embedding(num_embeddings=100, embedding_dim=64)
```

```
# Look up embedding for user 5
user_5_vector = user_embedding(torch.tensor([5]))  # Returns a 64-dim vector
```

4. nn.Linear (Fully Connected Layer)

Transforms input of one size to output of another size:

```
 # Transform 64 inputs to 128 outputs
layer = nn.Linear(64, 128)

# Apply it
output = layer(input_tensor)  # (batch_size, 64) → (batch_size, 128)
```

5. Activation Functions

Add non-linearity so the network can learn complex patterns:

```
 nn.ReLU()       # max(0, x) - zeros out negatives
nn.LeakyReLU()  # allows small negative values
nn.Sigmoid()    # squashes to range (0, 1)
```

6. Loss Functions

Measure how wrong the network's predictions are:

```
 bce = nn.BCELoss()  # Binary Cross-Entropy for classification
loss = bce(predictions, actual_labels)
```

7. Optimizers

Update network weights to reduce loss:

```
 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training loop
optimizer.zero_grad()  # Clear previous gradients
loss.backward()        # Calculate gradients
optimizer.step()       # Update weights
```

8. torch.no_grad()

Disable gradient tracking for inference (faster, less memory):

```
 @torch.no_grad()
def recommend_for_user(...):
    # No gradients calculated here
    scores = G(user_tensor)
```

📝 Detailed Code Explanation

The Generator Class

```
class Generator(nn.Module):
    def __init__(self, num_users: int, num_items: int, embed_dim=64, noise_dim=32, hi
        super().__init__()
        self.num_items = num_items
        self.noise_dim = noise_dim
        self.embed_dim = embed_dim

        # Embedding layer: converts user ID → vector of size embed_dim
        self.user_emb = nn.Embedding(num_users, embed_dim)

        # Neural network layers
        self.net = nn.Sequential(
            nn.Linear(embed_dim + noise_dim, hidden),  # Combine user + noise → hidde
            nn.ReLU(),                                  # Activation function
            nn.Linear(hidden, num_items)                # hidden → score for each mov
        )

    def forward(self, user_ids: torch.Tensor) -> torch.Tensor:
        b = user_ids.size(0)  # batch size (how many users at once)

        # Step 1: Get user embeddings
        u = self.user_emb(user_ids)  # Shape: (batch_size, embed_dim)

        # Step 2: Generate random noise
        z = torch.randn(b, self.noise_dim, device=user_ids.device)  # Shape: (batch_s

        # Step 3: Concatenate user embedding + noise
        x = torch.cat([u, z], dim=1)  # Shape: (batch_size, embed_dim + noise_dim)
        # Step 4: Pass through neural network
        logits = self.net(x)  # Shape: (batch_size, num_items)

        return logits  # Raw scores for each movie
```

What the Generator does: 1. Takes a user ID 2. Looks up their learned embedding (a vector of numbers representing the user) 3. Adds random noise (for variety in recommendations) 4. Passes through neural network layers 5. Outputs a score for EVERY movie in the catalog

The Discriminator Class

```
class Discriminator(nn.Module):
    def __init__(self, num_users: int, num_items: int, embed_dim=64, hidden=256):
        super().__init__()
        self.embed_dim = embed_dim

        # Separate embeddings for users and items
        self.user_emb = nn.Embedding(num_users, embed_dim)
        self.item_emb = nn.Embedding(num_items, embed_dim)

        # Neural network to classify real vs fake
        self.net = nn.Sequential(
            nn.Linear(embed_dim * 2, hidden),  # user + item embeddings
            nn.LeakyReLU(0.2),                 # Activation (allows small negatives)
            nn.Linear(hidden, 1),              # → single output
            nn.Sigmoid()                       # Squash to probability (0-1)
        )

    def forward(self, user_ids: torch.Tensor, item_ids: torch.Tensor) -> torch.Tensor
        u = self.user_emb(user_ids)  # User embedding
        i = self.item_emb(item_ids)  # Item embedding
        x = torch.cat([u, i], dim=1) # Combine them
        return self.net(x)           # Output: probability this is a REAL interaction
```

What the Discriminator does: 1. Takes a (user, movie) pair 2. Looks up embeddings for both 3. Combines them and passes through neural network 4. Outputs a probability: "Is this a real user-movie interaction?"

🎓 Training Process Explained

The training happens in the `train_cgan` function. Here's the step-by-step process:

Training Loop Overview

```
for epoch in range(1, epochs + 1):
    for batch in dataloader:
        # 1. Train Discriminator
        # 2. Train Generator
```

Step 1: Train the Discriminator

The Discriminator learns to distinguish: - **Real interactions**: (user, movie) pairs where the user actually liked the movie - **Fake interactions**: (user, movie) pairs generated by the Generator

```
 # --- Train Discriminator ---

# Get a batch of REAL (user, movie) pairs from training data
u_real, i_real = batch

# Discriminator predicts "real" for actual interactions
pred_real = D(u_real, i_real)
loss_real = bce(pred_real, torch.ones_like(pred_real))  # Should predict 1 (real)

# Generator creates FAKE movie predictions
fake_logits = G(u_real)
i_fake = sample_items_from_logits(fake_logits)  # Pick movies based on Generator scor

# Discriminator predicts "fake" for generated interactions
pred_fake = D(u_real, i_fake.detach())  # .detach() = don't update Generator here
loss_fake = bce(pred_fake, torch.zeros_like(pred_fake))  # Should predict 0 (fake)

# Total Discriminator loss
loss_D = loss_real + loss_fake
loss_D.backward()
opt_D.step()
```

Step 2: Train the Generator

The Generator learns to fool the Discriminator:

```
 # --- Train Generator ---

# Generate fake movie predictions
fake_logits = G(u_real)
i_fake = sample_items_from_logits(fake_logits)

# Ask Discriminator: "Is this real?"
pred = D(u_real, i_fake)

# Generator wants Discriminator to think it's REAL (predict 1)
loss_G = bce(pred, torch.ones_like(pred))
loss_G.backward()
opt_G.step()
```

The Adversarial Game

```
 ┌─────────────────────────────────────────────────────────────────┐
│                                                                 │
│   Generator: "I'll generate movie suggestions"                  │
│        ↓                                                        │
│   Discriminator: "That's fake! Real users don't like those"     │
│        ↓                                                        │
│   Generator: "Let me learn and try again..."                    │
│        ↓                                                        │
│   Discriminator: "Hmm, this looks more real..."                 │
│        ↓                                                        │
│   (Repeat thousands of times)                                   │
│        ↓                                                        │
│   Generator: *produces realistic recommendations*               │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

🎯 How Recommendations Are Generated

After training, the Generator can recommend movies:

```
 @torch.no_grad()  # No gradient calculation needed
def recommend_for_user(G, user_idx, num_items, seen_movies, k=10, noise_samples=5):
    G.eval()  # Set to evaluation mode

    # Create user tensor
    u_tensor = torch.tensor([user_idx])

    # Generate scores multiple times with different noise
    # (averaging reduces randomness from the noise)
    scores_sum = None
    for _ in range(noise_samples):
        logits = G(u_tensor)
        if scores_sum is None:
            scores_sum = logits
        else:
            scores_sum = scores_sum + logits

    # Average the scores
    scores = scores_sum / noise_samples

    # Remove movies the user has already seen
    for movie_idx in seen_movies:
        scores[movie_idx] = -infinity  # Extremely low score

    # Return top-K movies with highest scores
    top_k_indices = torch.argsort(scores, descending=True)[:k]
    return top_k_indices
```

📊 Evaluation Metrics

Recall@K

"Of all the movies the user actually liked in the test set, what percentage did we recommend?"

$$\text{Recall@K} = \frac{|\text{Recommended}_K \cap \text{Actually Liked}|}{|\text{Actually Liked}|}$$

Example: - User liked 10 movies in test set - We recommended 10 movies - 3 of our recommendations were movies they actually liked - Recall@10 = 3/10 = 0.30 (30%)

NDCG@K (Normalized Discounted Cumulative Gain)

"Did we rank the good recommendations at the top of the list?"

Higher positions get more weight. If a liked movie is #1, it contributes more than if it's #10.

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{rel_i}{\log_2(i+1)}$$

HitRate@K

"Did we recommend at least one movie the user liked?"

$$\text{HitRate@K} = \begin{cases} 1 & \text{if any recommendation was liked} \ 0 & \text{otherwise} \end{cases}$$

📁 File Structure

```
rs-project/
│
├── main.py                     # Core logic: data loading, models, training, evaluati
│
├── app.py                      # Streamlit web interface
│
├── requirements.txt            # Python dependencies (numpy, torch)
│
├── README.md                   # Project overview and how to run
│
├── PROJECT_EXPLANATION.md      # This file - detailed code explanation
│
└── ml-100k/                    # MovieLens dataset folder
    ├── u.item                  # Movie metadata (ID, title, genres)
    ├── u1.base                 # Training data for split 1
    ├── u1.test                 # Test data for split 1
    ├── u2.base, u2.test        # Split 2
    ├── u3.base, u3.test        # Split 3
    ├── u4.base, u4.test        # Split 4
    └── u5.base, u5.test        # Split 5
```

🚀 Summary

1. **Data Processing**: MovieLens ratings are converted to binary "liked/not liked" signals

2. **Generator**: Learns to predict which movies each user might like
3. **Discriminator**: Learns to tell real preferences from fake ones
4. **Training**: They compete, making each other better
5. **Recommendation**: Generator produces scores, we pick the top-K unseen movies
6. **Evaluation**: We measure how well our recommendations match actual user preferences

The beauty of this approach is that the Generator learns complex patterns about user preferences without being explicitly programmed with rules - it discovers them through the adversarial training process!