A Project Report

On

Recommendation System using Adversarial Deep Learning

BY

Nischith P

2021A7PS3220H

Under the supervision of

Dr. Aneesh Chivukula

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI,

HYDERABAD CAMPUS

Acknowledgements

I would like to express my heartfelt gratitude to my faculty advisor, Dr. Aneesh Chivukula, for his constant availability and invaluable guidance throughout my project. His insights in this field have been instrumental in shaping my work, and I have learned immensely from this experience. My sincere appreciation goes to BITS Pilani and the Computer Science department for providing me the opportunity to work on this project.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

Certificate

This is to certify that the project report entitled "**Recommendation System using Adversarial Deep Learning"** submitted by Mr. Nischith P (ID No. 2021A7PS3220H) in partial fulfillment of the requirements of the course Design Project, embodies the work done by him under my supervision and guidance.

Date: 06/12/2024 **Signature:**

Abstract

The aim of the project is to build a recommendation system using adversarial deep learning. Work involves implementation of GCN model for recommendation system using MovieLens dataset. There are three models namely GCNnormal, GCNmanipulated, GCNsecure. GCNnormal is trained and tested on the original dataset i.e, original interaction matrix. GCNmanipulated is trained on the original dataset and tested on adversarial examples. GCNsecure is trained and tested on perturbed dataset which involves the same attack used to create testing dataset for GCNmanipulated.

There are two methods used to create adversarial examples. Simulated Annealing and perturbation using modified GANs. In both the techniques successful attacks are made on the dataset in a game theory framework. The expected results are performance of GCNnormal > GCNmani and GCNsecure > GCNmani. Desirable results are obtained using the above adversarial techniques.

Contents

Introduction

A recommendation system is a type of information filtering tool designed to suggest relevant items to users by predicting their preferences based on various data inputs. It is widely used in many industries such as e-commerce, entertainment, social media, and more, where personalized suggestions can significantly enhance user experience

Recommendation systems offer several benefits, especially in industries where personalizing user experiences can significantly enhance engagement and customer satisfaction. Here are some key benefits:

- Recommendation systems tailor suggestions to individual users based on their preferences, browsing history, or interactions. This makes the user experience more engaging and relevant, improving user satisfaction.
- In e-commerce platforms, personalized recommendations can lead to higher conversion rates. Suggesting products that align with users' preferences increases the likelihood of purchases, which can drive significant revenue growth.
- OTT's like Netflix, Hotstar, Prime uses recommendations as a core feature to know the user's preferences, favorites and recommend new shows and movies accordingly.

Recommendation system

There are multiple approaches to build a recommendation system using machine learning, deep learning and natural language processing. Most widely used recommendation system models involves:

1. Collaborative Filtering:

   Collaborative filtering is one of the most commonly used techniques. It works by finding similarities between users or items. There are two main types:

   - User-based collaborative filtering: Recommends items that users with similar tastes have liked. This similar taste is learned by the model

using similarity metrics like cosine similarity where items of another user with maximum similarity score is recommended.

- Item-based collaborative filtering: Recommends items similar to the ones the user has previously enjoyed. If n users have similar scores for 2 items then the 2 items are very similar in nature because of which one might be recommended when the other is a part of interaction.

2. Content-Based Filtering:

   This approach recommends items based on the user's past preferences. It considers the attributes or features of the items that the user has interacted with and suggests similar items.

3. Matrix Factorization

   This technique is often used in collaborative filtering, especially with large-scale datasets like MovieLens. It breaks down a large matrix (user-item interactions) into smaller matrices to identify latent factors or hidden patterns in the data, allowing for better predictions.

4. Graph based approaches

   The interaction between users and items can be represented as a graph where each node represents user and item and the weight of the edge between 2 nodes indicates some metric. After constructing we can use popular based approaches like graph convolutional network or graph neural network to learn the latent feature vectors.
   The graph can be constructed dynamically according to the approach we are using to build a recommendation system. For example bipartite graph is a popular method where edges exist between only users and items between whom interaction has taken place. Then GCN/ GNN is applied to generate embedding based on which recommendation can be carried out.

Adversarial attacks

Adversarial attacks play an important role in machine learning while training a model for a specific task. When a model is trained on a dataset it can be vulnerable during the testing process to certain kinds of attacks if it is not well trained against

those attacks. Thus creating adversarial examples and including them in the training dataset helps the model to be more accurate and robust towards manipulations.

Simulated Annealing (SA) is a powerful optimization algorithm inspired by the physical process of annealing, where metals are heated and slowly cooled to reduce defects and reach a stable state. In the context of my work, I used SA to introduce carefully designed perturbations to optimize solutions and generate adversarial examples. Its ability to escape local optima by occasionally accepting worse solutions during the early stages proved instrumental in fine-tuning adversarial attacks.

Generative Adversarial Networks (GANs) are used for generating synthetic data that closely resembles real data. By combining a generator and a discriminator in a min max environment GANs excel at creating examples that are nearly indistinguishable from the original data. In my experiments, modified GANs which include an autoencoder and a generator which acts as decoder are used to create adversarial examples of deliberate inputs designed to challenge and test the robustness of the underlying model.

The integration of Graph Convolutional Networks (GCNs), Simulated Annealing, and GANs in my work allowed me to explore diverse methods for generating and evaluating adversarial examples. This synergy not only enhanced the robustness testing of GCN models but also provided insights into optimization and adversarial strategies.

Literature survey

Dist-GAN: An Improved GAN using Distance Constraints

Dist-GAN is a new Generative Adversarial Network architecture that tackles two serious critical challenges, namely mode collapse and gradient vanishing usually encountered in GANs. The authors introduce the Autoencoders (AE) with GAN and propose two novel distance constraints:

Latent-Data Distance Constraint: The authors ensure that distances in latent space correspond meaningfully with distances in data space thus preventing mode collapse and providing better diversity in samples generation.

Discriminator-Score Distance Constraint: This ensures alignment of generated samples and real samples based on discriminator scores, hence leading the generator to minimize its loss and produce more realistic images.

In this paper auto encoder is used in the min max game framework of GAN. Auto encoder encodes images into latent representations and passes them to the generator where the generator acts like a decoder to bring back the image from latent representation and a generator to generate synthetic images for the discriminator.

Dist-GAN demonstrates mode diversity very well in the manner that it has shown through the ability to cover all modes in synthetic datasets, producing realistic high-quality images in real-world benchmarks, and robustness is established by improved evaluation metrics.

$$g(zi,zj)>\delta z \Longrightarrow f(xi,xj)>\delta x$$

This constraint introduces relation between the distances in latent space and the corresponding distances in data space. If two latent variables (zi and zj) are close (or far) in the latent space, their mapped data points (xi and xj) should also be close (or far) in the data space. g is a distance metric in latent space.f is a distance metric in data space. δz, δx are thresholds for the respective spaces.

$$\text{Min}_{w,\theta}\ L_R(\omega,\theta) + \lambda_r L_W(\omega,\theta)$$

$$\text{where, } L_W(\omega,\theta) = \| f(x, G_\theta(z)) - \lambda_w g(E_\omega(x), z) \|_2^2$$

$$L_R(\omega,\theta) = \| x - G_\theta(E_\omega(x)) \|_2^2$$

The above min function represents additional latent-data constraint along with AE objective. L_w is the final latent-data distance constraint added to the Autoencoder (AE) and L_r is the conventional AE objective function. G_θ is the generator (decoder), E_ω is the encoder, and λ_w is a scaling factor to match the dimensionality differences.

$$L_G(\theta) = | E_x[\sigma(D_\gamma(x))] - E_z[\sigma(D_\gamma(G_\theta(z)))] |$$

The objective minimizes the distance between the scores of real data and generated data. This is a distance constraint for the generator.

$$L_D(\omega,\theta,\gamma) = -(E_x[\log\sigma(D_\gamma(x))] + E_z[\log(1-\sigma(D_\gamma(G_\theta(z))))] + E_x[\log\sigma(D_\gamma(G_\theta(E_\omega(x))))] - \lambda_p E_x\hat{}[(\| \nabla x\hat{} D_\gamma(x) \|_2^2 - 1)^2])$$

The above discriminator objective contains reconstructed samples from the AE as real to slow down its convergence and prevent vanishing gradient problem.

Model

After the literature survey on GCN model the aim was to build a baseline GCN/ GNN model. For this I referred to various github repos consisting of GCN/ GNN models. Later I contacted a Phd student Danny Muzata as per Aneesh sir's instruction to get help regarding implementation of GCN model.

Dataset

Most papers on GCN/ GNN use one of the most popular datasets - **MovieLens** dataset. The MovieLens dataset is a widely-used dataset for evaluating recommendation systems. It is provided by GroupLens, a research group at the University of Minnesota, and contains user ratings of movies. Different versions of the dataset exist, with varying numbers of users, movies, and ratings. The most commonly used version, MovieLens 100K, consists of 100,000 ratings from 610 users on 9754 movies. The dataset includes userID, movieID, timestamp and ratings as columns. In some versions, the dataset includes additional information like movie genres and tags. This dataset has become a benchmark for developing and testing recommendation algorithms due to its scale, variety of metadata, and simplicity in modeling real-world movie preferences.

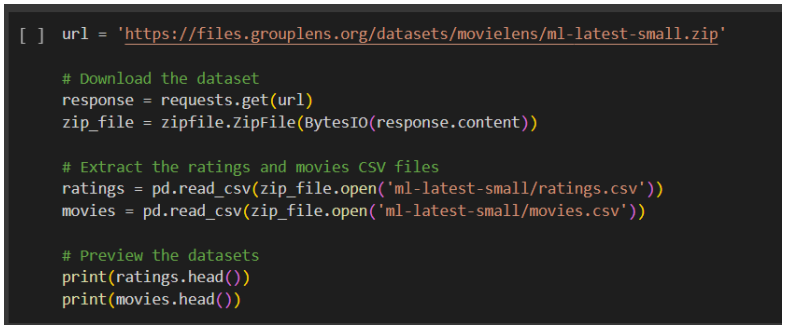
Algorithm

- First we download the MovieLens dataset and preprocess the dataset.
- We create an user item interaction matrix using this dataset.
- From the matrix we build a custom dataset which contains all the users, items and ratings.
- Before applying GCN we first create embeddings for each user and item of a certain dimension(hyper-parameter).
- We update these embeddings using matrix factorization method for n epochs. The embedding obtained after the last iteration acts as the initial/ start point for the GCN.

- Create a graph where each node is either an user or an item and represented by the embeddings obtained at the end of matrix factorization.
- Edge exists between an user and an item between whom an interaction is present in the dataset. This forms a bipartite graph between users and items.
- The weight of the edge is the dot product of the embedding of the user and transpose of the embedding of the item between which the edge exists.
- We train this model by applying GCN layers on this graph structure which helps in learning the embeddings by minimizing the loss.

Code

- Loading and preprocessing the movieLens dataset.



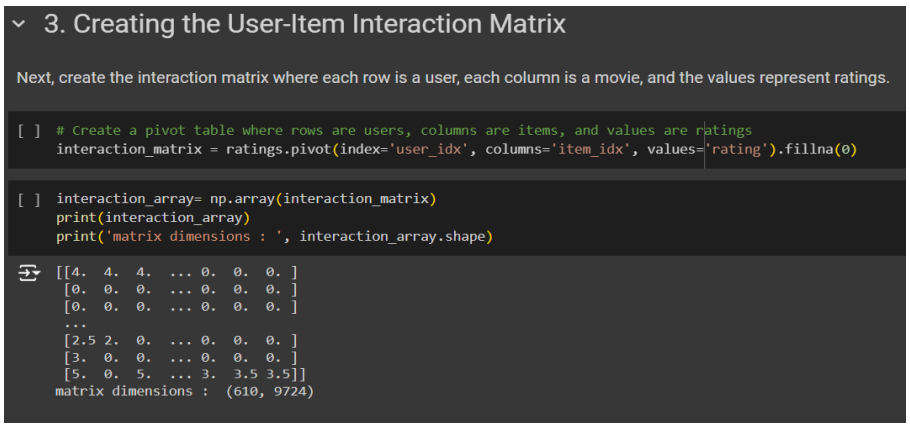
```
[ ] url = 'https://files.grouplens.org/datasets/movielens/ml-latest-small.zip'

    # Download the dataset
    response = requests.get(url)
    zip_file = zipfile.ZipFile(BytesIO(response.content))

    # Extract the ratings and movies CSV files
    ratings = pd.read_csv(zip_file.open('ml-latest-small/ratings.csv'))
    movies = pd.read_csv(zip_file.open('ml-latest-small/movies.csv'))

    # Preview the datasets
    print(ratings.head())
    print(movies.head())
```

- Creating user item interaction matrix



```
3. Creating the User-Item Interaction Matrix

Next, create the interaction matrix where each row is a user, each column is a movie, and the values represent ratings.

[ ] # Create a pivot table where rows are users, columns are items, and values are ratings
    interaction_matrix = ratings.pivot(index='user_idx', columns='item_idx', values='rating').fillna(0)

[ ] interaction_array= np.array(interaction_matrix)
    print(interaction_array)
    print('matrix dimensions : ', interaction_array.shape)

    [[4.  4.  4.  ... 0.  0.  0. ]
     [0.  0.  0.  ... 0.  0.  0. ]
     [0.  0.  0.  ... 0.  0.  0. ]
     ...
     [2.5 2.  0.  ... 0.  0.  0. ]
     [3.  0.  0.  ... 0.  0.  0. ]
     [5.  0.  5.  ... 3.  3.5 3.5]]
    matrix dimensions :  (610, 9724)
```

- Using matrix factorization method to create initial embeddings of users and items for GCN.

```python
#Matrix Factorization Model for Embedding Users and Items

class MFModel(nn.Module):
    def __init__(self, num_users, num_items, embedding_size):
        super(MFModel, self).__init__()
        # Create embedding layers for users and items
        self.user_embedding = nn.Embedding(num_users, embedding_size)
        self.item_embedding = nn.Embedding(num_items, embedding_size)

    def forward(self, user_ids, item_ids):
        # Get user and item embeddings
        user_embedding = self.user_embedding(user_ids)
        item_embedding = self.item_embedding(item_ids)
        # Compute the dot product between user and item embeddings
        dot_product = (user_embedding * item_embedding).sum(dim=1)
        return dot_product

# Initialize the model with number of users, items, and the embedding size
num_users = len(users)
num_items = len(items)
embedding_size = 50  # This is a tunable hyperparameter

mf_model = MFModel(num_users, num_items, embedding_size).to(device)
```

```python
#Train Matrix Factorization Model

optimizer = optim.Adam(mf_model.parameters(), lr=0.001)  # Adam optimizer
loss_fn = nn.MSELoss()  # Loss function (Mean Squared Error)

def train_mf_model(model, train_loader, optimizer, criterion, num_epochs=2):
    model.train()  # Set the model to training mode
    for epoch in range(num_epochs):
        total_loss = 0
        for batch in train_loader:
            user_ids = batch['user_idx'].to(device)
            item_ids = batch['item_idx'].to(device)
            ratings = batch['rating'].to(device)

            optimizer.zero_grad()  # Zero the gradients
            preds = model(user_ids, item_ids)  # Forward pass
            loss = criterion(preds, ratings)  # Compute loss
            loss.backward()  # Backpropagation
            optimizer.step()  # Gradient descent step
            total_loss += loss.item()  # Accumulate loss

        print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {total_loss / len(train_loader):.4f}')

train_mf_model(mf_model, train_loader, optimizer, loss_fn)
```

- Creating graph data for GCN. This involves creating nodes(users n items) and edges(interactions) between nodes.

```python
# Create edges between users and items based on interactions (ratings)
user_item_edges = ratings[['user_idx', 'item_idx']].values.T  # Create edges between user-item pairs

user_item_edges[1] += num_users

# Create edge index (format required by torch_geometric)
edge_index = torch.tensor(user_item_edges, dtype=torch.long)

# Concatenate user and item embeddings to form node features
node_features = torch.cat([torch.tensor(user_embeddings, dtype=torch.float), torch.tensor(item_embeddings, dtype=torch.float)], dim=0)

# Create the PyTorch Geometric data object (x: node features, edge_index: graph edges)
train_graph_data = Data(x=node_features, edge_index=edge_index)
```

- GCN model and training

```python
class GCNModel(nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GCNModel, self).__init__()
        # First graph convolutional layer
        self.conv1 = GCNConv(in_channels, hidden_channels)
        # Second graph convolutional layer
        self.conv2 = GCNConv(hidden_channels, out_channels)

    def forward(self, data):
        # Forward pass through the first graph convolutional layer
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = torch.relu(x)  # Apply ReLU non-linearity
        # Forward pass through the second graph convolutional layer
        x = self.conv2(x, edge_index)
        return x

# Initialize the GCN model
gcn_model = GCNModel(in_channels=embedding_size, hidden_channels=64, out_channels=32).to(device)
```

```python
gcn_optimizer = optim.Adam(gcn_model.parameters(), lr=0.01)
gcn_loss_fn = nn.MSELoss()

def train_gcn_model(model, train_graph, optimizer, criterion, num_epochs=100):
    model.train()  # Set model to training mode
    for epoch in range(num_epochs):
        optimizer.zero_grad()  # Zero the gradients
        output = model(train_graph)  # Forward pass through the GCN


        #print('output dimension',output.shape)
        # Assuming user_idx and item_idx are indices of user-item pairs
        user_indices = ratings['user_idx'].unique()  # Indices for users
        item_indices = ratings['item_idx'].unique()  # Indices for items


        #print('user indices dimension check',user_indices.shape)
        #print('item indices dimension check',item_indices.shape)
        # Get embeddings for the relevant user-item pairs
        user_embeddings = output[user_indices]  # Shape: (N, embedding_size)
        item_embeddings = output[item_indices + num_users]  # Shift by num_users for items

        # Compute predicted ratings
        predicted_ratings = torch.matmul(user_embeddings, item_embeddings.T) # Dot product

        # Get target ratings from interaction matrix
        interaction_tensor = torch.tensor(interaction_matrix.values, dtype=torch.float32)
        target= interaction_tensor
        #target = interaction_tensor[user_indices, item_indices].view(-1)  # Flatten to match

        # Compute loss
        loss = criterion(predicted_ratings, target)
        loss.backward()  # Backpropagation
        optimizer.step()  # Gradient descent step

        print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item():.4f}')


train_gcn_model(gcn_model, train_graph_data, gcn_optimizer, gcn_loss_fn)
```

- GCN model evaluation

```python
def evaluate_gcn_model(model, graph_data, interaction_matrix):
    model.eval()  # Set model to evaluation mode
    with torch.no_grad():
        output = model(graph_data)  # Forward pass

        user_indices = ratings['user_idx'].unique()  # Indices for users
        item_indices = ratings['item_idx'].unique()  # Indices for items

        user_embeddings = output[user_indices]  # Shape: (N, embedding_size)
        item_embeddings = output[item_indices + num_users]  # Shift by num_users for items

        predicted_ratings = torch.matmul(user_embeddings, item_embeddings.T)
        interaction_tensor = torch.tensor(interaction_matrix.values, dtype=torch.float32)
        target= interaction_tensor

        rmse = np.sqrt(mean_squared_error(target, predicted_ratings))  # Compute RMSE
        print(f'RMSE: {rmse:.4f}')

# Evaluate on training data
evaluate_gcn_model(gcn_model, train_graph_data, interaction_matrix)
```

```
RMSE: 0.4263
```

- Predictions for new user:

  When a new user is introduced to the system with initial interactions half of the user interaction with the items is masked and the other half is used to create the user embeddings. Weighted mean of these item embeddings is used to generate the user embeddings. Here the weights are the interaction between the user and that particular item (rating given by the user to that particular movie).

```python
def predict_new_user_rating(item_embeddings, masked_array, num_users=num_users):
    item_embeddings = item_embeddings.detach().numpy()
    masked_array = np.array(masked_array, dtype=np.float32)

    masked_array = masked_array.reshape(-1, 1)  # Shape: (num_items, 1)
    weighted_sum = np.sum(item_embeddings * masked_array, axis=0)
    sum_of_weights = np.sum(masked_array)
    new_user_embedding = weighted_sum / sum_of_weights
    predicted_ratings = np.dot(item_embeddings, new_user_embedding)

    return predicted_ratings
```

- Testing the GCN model:

  To test the performance of 3 baseline models GCNnormal, GCNmani, GCNsecure we use n new users to find their embeddings and predicted ratings. Using this rating we find rmse for each of n users. We use the mean of all the rsme found as our comparison metric for the models.
  The below code explains the above:

```python
def prediction_test(num_users, interaction_array, item_embeddings):
    metric=0
    for i in range(len(interaction_array) - num_users, len(interaction_array)):
        normal_test = interaction_array[i]
        non_zero_indices = np.nonzero(normal_test)[0]
        num_values_to_keep = len(non_zero_indices) // 2
        selected_indices = np.random.choice(non_zero_indices, size=num_values_to_keep, replace=False)
        masked_array = np.zeros_like(normal_test)
        masked_array[selected_indices] = normal_test[selected_indices]
        prediction = predict_new_user_rating(item_embeddings, masked_array)
        prediction= np.clip(prediction, 0, 5)
        rmse = np.sqrt(np.mean((prediction - normal_test) ** 2))
        metric+= rmse
    return metric/num_users
```

This function takes a number of new users to be tested, an interaction matrix(normal or perturbed) and item embeddings as the input.

Simulated Annealing

One of the ways to create adversarial examples is by the Simulated Annealing algorithm. Simulated Annealing (SA) is a probabilistic optimization algorithm inspired by the annealing process in metallurgy, where controlled cooling helps attain a system's minimum energy state. In the context of generating adversarial examples, SA is employed to perturb an input sample such that it fools a machine learning model while keeping the perturbation minimal and imperceptible. Algorithm involves many by parts (functions) such as-

- Anneal function
- Rmse score
- Fitness function
- Twoplayergame_simulated annealing

Anneal function -

```
def anneal(alpha, mask_a, d=2, lower_bound=20, upper_bound=10):
    alpha = alpha.copy()
    mask_b = np.random.choice([True, False], size=alpha.shape)
    mask = mask_a ^ mask_b
    step = np.random.randint(0, d+1, size=alpha.shape)/225.
    start_h = 0
    end_h = 1
    start_w = np.random.randint(0, lower_bound)
    end_w = np.random.randint(len(alpha) - upper_bound, len(alpha))
    masksliced = np.zeros(alpha.shape, dtype=bool)
    masksliced[start_w:end_w] = mask[start_w:end_w]
    alpha[masksliced] += step[masksliced]
    return alpha
```

The anneal function is inspired by the annealing process, where gradual adjustments are made to reach an optimal state. In this context, it modifies the input array alpha by introducing small, random changes. It uses a combination of random masks and segment selection to decide which parts of the array to modify. Within a randomly chosen range, the values are incremented by a small step, helping the system explore different possibilities and escape local optima. A random boolean mask (mask_b) is generated and XORed with mask_a to create a new mask (mask). A

random step size is generated using d, normalized by dividing by 225. Random start and end indices for slicing (start_w and end_w) are calculated within the specified bounds. Since we are perturbing a single user interaction it is a 1D array (only single row) so the start_h is 0 and end_h is 1. A boolean slice mask (mask sliced) is created to apply modifications within the segment [start_w:end_w]. The alpha array is modified by incrementing the elements within the masked region by the computed step size.

RMSE score -

```
[ ]  def rmse_score(model, user_ind,user_embeddings, item_embeddings, alpha):
         model.eval()  # Set model to evaluation mode
         with torch.no_grad():
             alpha= alpha.reshape(-1)
             predicted_ratings = torch.matmul(user_embeddings[user_ind], item_embeddings.T)
             interaction_tensor = torch.tensor(alpha, dtype=torch.float32)
             target= interaction_tensor
             rmse = np.sqrt(mean_squared_error(target, predicted_ratings))  # Compute RMSE

         return rmse
```

The rmse_score function calculates the Root Mean Squared Error (RMSE) to evaluate the accuracy of the model's predictions for a specific user's interactions with items. The function computes predicted ratings by taking the dot product of the user's embedding and the transpose of the item embeddings. The actual interaction data, alpha is used as the target for comparison. Finally, the RMSE is used as a metric to quantify the difference between the predicted and actual values, providing a measure of prediction accuracy.

Fitness function -

```
[ ]  def fitness(user_embeddings, item_embeddings, alpha_population, model, lambda_value = 0.1):
         fitness_values = []
         for alpha_ind, alpha in alpha_population:
             # here alpha represents that one pparticular user's interaction with all the items
             # alpha_population represents manipulated interaction matrix for some users

             error = lambda_value * rmse_score(model, alpha_ind ,user_embeddings, item_embeddings, alpha)
             alpha_fitness = 1 + error - np.linalg.norm(alpha)
             fitness_values.append(abs(np.max(alpha_fitness)))
         return fitness_values
```

The fitness function evaluates the quality of each candidate solution (alpha) in a population of manipulated interaction matrices. For each candidate, it computes an error term by scaling the RMSE (obtained using the rmse_score function) with a regularization parameter.

Twoplayergame_simulated annealing -

The twoplayergame_sa function implements a two-player game optimization process combined with simulated annealing to refine interaction matrices and maximize a payoff function. It begins by splitting the interaction matrix into three populations: candidate (ac), active (ag), and inactive (an). Fitness scores are computed for these populations to evaluate their quality. The active group (ag) is iteratively improved by comparing its fitness with the candidate group and updating it using the simulated annealing process. The simulated annealing step introduces perturbations to the candidate group (ac) using the anneal function, exploring new solutions (an). If the new solutions show improved fitness, then replace it with the current candidate solutions. The process continues until the payoff converges or the maximum temperature threshold is reached. Finally, the best-performing solution from the active group is the result of the optimization.

```
def adversarial_manipulation(user_embeddings, item_embeddings, interaction_matrix, model,M):
    A_s = []
    for i in range(1, M+1):
        a_i = twoplayergame_sa(user_embeddings, item_embeddings, interaction_matrix, model)
        A_s.append(a_i)

    interaction_matrix_manipulated= generate_manipulated_data(interaction_matrix, A_s)
    return interaction_matrix_manipulated
```

The adversarial_manipulation function generates a manipulated interaction matrix by introducing adversarial perturbations. It iteratively calls the twoplayergame_sa function M times to create adversarial interaction vectors for specific users, storing these in a list (A_s). Once all adversarial vectors are generated, the generate_manipulated_data function combines the original interaction matrix with the manipulated data. The generated_manipulated_data is appended to the original interaction matrix.

GAN

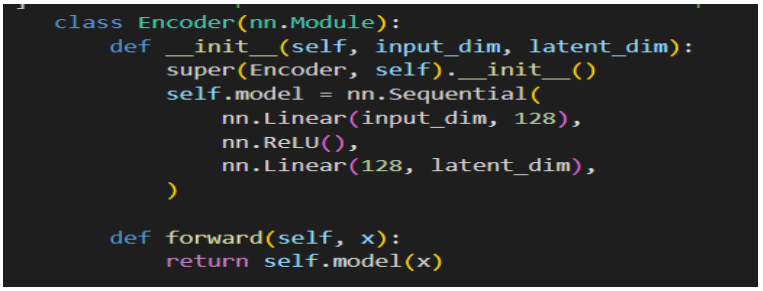
Another method to create perturbations is by using one of the widely used deep learning models GAN. GAN is an architecture created by stacking up generator and a discriminator. Generator is used to produce synthetic data which is very similar to real data and acts as input to discriminator and Discriminator tries to classify the images as real or fake. Both the models try to maximize their payoffs/ accuracies and loss of the other model.

In the architecture I have implemented there is a denoising auto encoder and a generator which acts as both generator and decoder. Both the models are trained back to back for a few iterations to create perturbations in the interaction matrix.

The main components are -

- Encoder
- Generator

Encoder -



```
class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Encoder, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, latent_dim),
        )

    def forward(self, x):
        return self.model(x)
```

The Encoder class is designed to map an interaction matrix to a latent space representation. It has a fully connected architecture with input features first passed through a hidden layer of 128 units with ReLU activation, which introduces non-linearity in order to capture complex patterns. Then the output of this layer goes through another linear layer that produces the final latent representation with the specified dimensionality, latent_dim.

Generator (Decoder) -

```
[ ]  # Generator: Generates perturbed interactions
     class Generator(nn.Module):
         def __init__(self, latent_dim, output_dim):
             super(Generator, self).__init__()
             self.model = nn.Sequential(
                 nn.Linear(latent_dim, 128),
                 nn.ReLU(),
                 nn.Linear(128, output_dim),
                 nn.Tanh(),  # Output perturbations in range [-1, 1]
             )

         def forward(self, z):
             return self.model(z)
```

The Generator(Decoder) is designed to create perturbed interactions from the latent space representation generated by the encoder. The first layer is a linear layer which expands the latent space through 128 units with a ReLU activation function, introducing non-linearity. The output is then transformed into the desired interaction space (output_dim) through another linear layer and passed through a Tanh activation, ensuring the generated perturbations are scaled within the range [-1, 1]. This design enables the generator to produce realistic and controlled perturbations.

Later the above model is trained for a certain number of epochs based on the mse loss the output generates with respect to the original interaction matrix.

A comparison between original and perturbed interaction matrix -

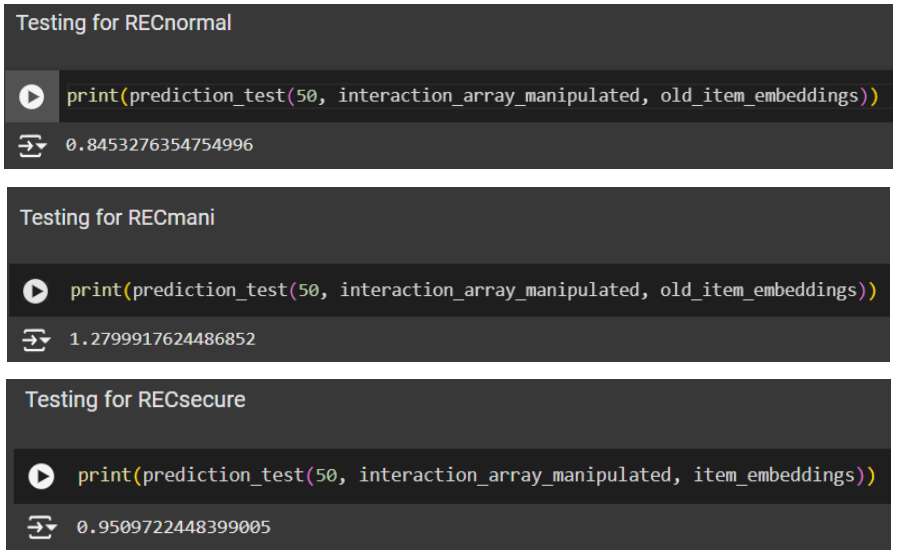
```
Original Interaction Matrix:
tensor([[4.0000, 4.0000, 4.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        ...,
        [2.5000, 2.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [3.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [5.0000, 0.0000, 5.0000,  ..., 3.0000, 3.5000, 3.5000]])
Perturbed Interaction Matrix:
tensor([[4.0027e+00, 4.0025e+00, 3.9960e+00,  ..., 5.5158e-03, 1.5042e-02,
         0.0000e+00],
        [2.6638e-03, 2.4732e-03, 0.0000e+00,  ..., 5.5158e-03, 1.5042e-02,
         0.0000e+00],
        [1.3525e-02, 9.0535e-03, 0.0000e+00,  ..., 1.1342e-02, 1.2437e-02,
         1.3604e-03],
        ...,
        [2.5027e+00, 2.0025e+00, 0.0000e+00,  ..., 5.5158e-03, 1.5042e-02,
         0.0000e+00],
        [3.0027e+00, 2.4732e-03, 0.0000e+00,  ..., 5.5158e-03, 1.5042e-02,
         0.0000e+00],
        [5.0000e+00, 2.4732e-03, 4.9960e+00,  ..., 3.0055e+00, 3.5150e+00,
         3.4997e+00]])
```

Results

The main objective of this experiment is to generate results for the 3 baseline models mentioned above and compare their performance. As mentioned before we use RMSE as our metric to compare the performances. The expected results are RMSE of GCNnormal < GCNmani and GCNsecure <GCNmani.

Below are the results for the Simulated Annealing and GAN approach:

Simulated Annealing



GAN

Final result table-

Method	Model	Metric(MRMSE)
Simulated Annealing	GCNnormal	0.8453
	GCNmani	1.2799
	GCNsecure	0.9509
GAN	GCNnormal	0.9315
	GCNmani	1.2530
	GCNsecure	0.7437

As we can see the performance GCNnormal > GCNmani and GCNsecure > GCNmani for both Simulated Annealing and GAN based adversarial attacks.

References

1. Dist-GAN: An Improved GAN using Distance Constraints https://arxiv.org/abs/1803.08887
2. Generative Adversarial Autoencoder Networks https://arxiv.org/abs/2111.13282
3. Adversarial Autoencoders https://arxiv.org/abs/1511.05644
4. Conditional Autoencoders with Adversarial Information Factorization https://arxiv.org/abs/1711.05175
5. RecGAN: recurrent generative adversarial networks for recommendation systems https://dl.acm.org/doi/10.1145/3240323.3240383
6. Collaborative Denoising Auto-Encoders for Top-N Recommender Systems https://dl.acm.org/doi/10.1145/2835776.2835837
7. Adversarial Deep Learning Models with Multiple Adversaries https://ieeexplore.ieee.org/document/8399545
8. Game Theoretical Adversarial Deep Learning With Variational Adversaries https://doi.org/10.1109/TKDE.2020.2972320