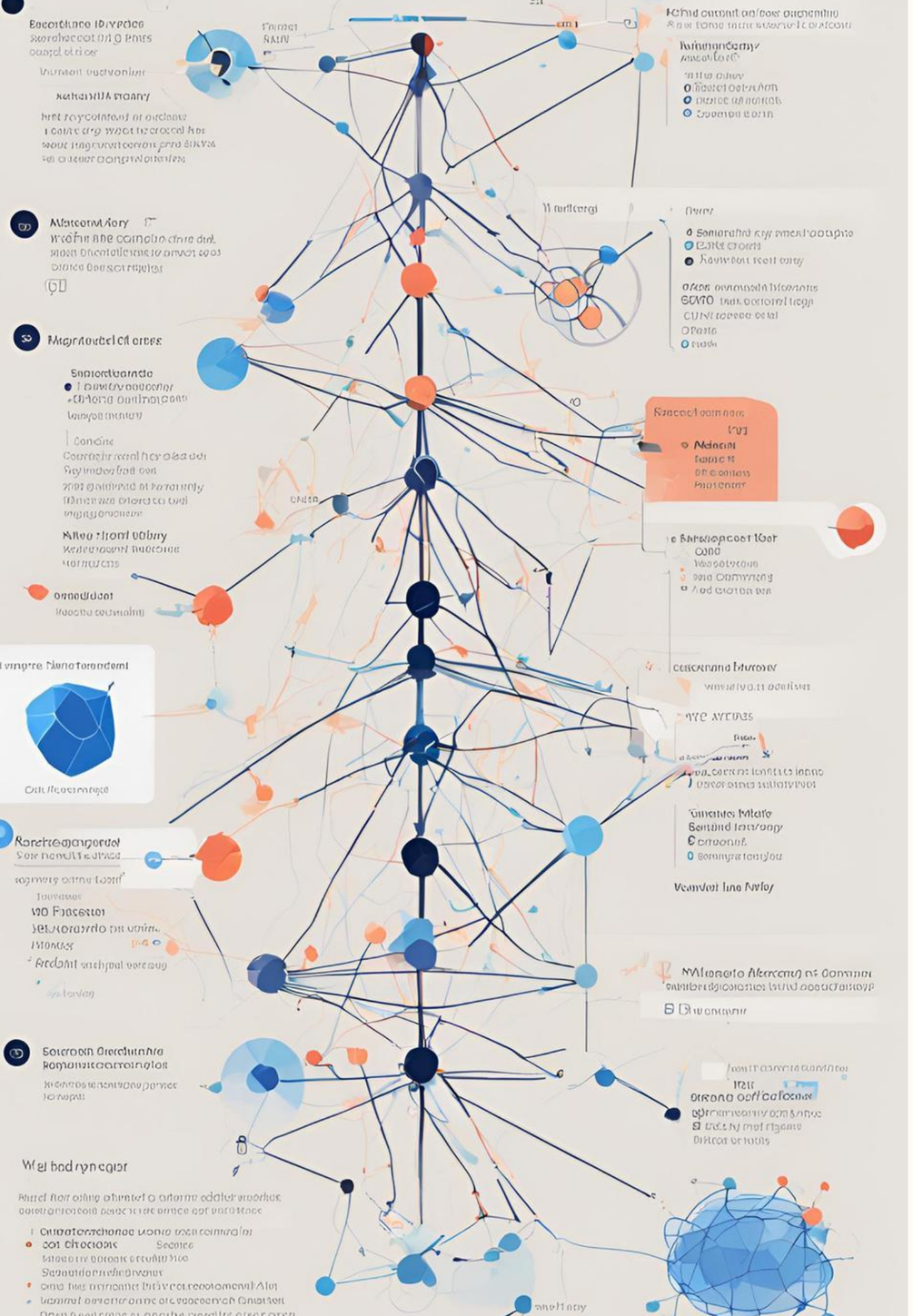
UNIMIB, UNIMI, UNIPV

AI Assignment

Ulysse Duvillier - Diego Corna - Andrea Barbieri

Master AI4ST



Introduction

How we understood and implemented those topics :

- Recommender system
- Explainable AI
- Graph representation learning
- Attention mechanism and transformers

Recommender System from Scratch

Dataset

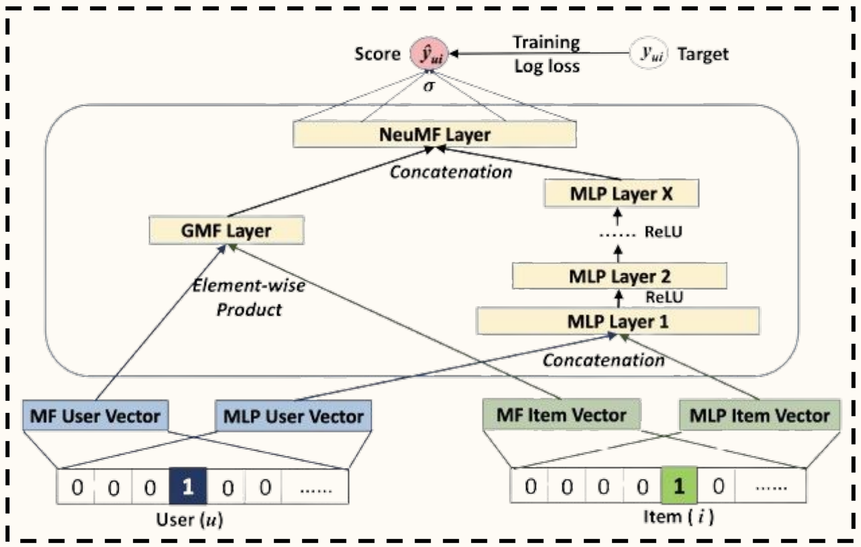
Movielens Dataset :

- 100,000 ratings
- from 1000 users on 1700 movies.

Goals

- Build a recommender system from scratch
- Implement GMF and MLP
- Implement Neural Collaborative Filtering

Structure



GMF

- Generally Matrix Factorization (MF) algorithms associates each user and item with a real-valued vector of latent features.

MLP

- The embedded latent vectors are fed into our multi-layer neural architecture, to map the latent vectors to the predicted probability scores.

NeuMF

- Perform MLP and MF separatly and then concatenate the output of the models.

GMF

Embedding :

1. user vector
2. item vector

Dot product over those two embedded vectors.

Linear tranformation

```python
class GMF(nn.Module):
    def __init__(self, args, num_users, num_items):
        super(GMF, self).__init__()
        self.num_users = num_users
        self.num_items = num_items
        self.factor_num = args.factor_num

        self.embedding_user = nn.Embedding(num_embeddings=self.num_users, embedding_dim=self.factor_num)
        self.embedding_item = nn.Embedding(num_embeddings=self.num_items, embedding_dim=self.factor_num)

        self.affine_output = nn.Linear(in_features=self.factor_num, out_features=1)
        self.logistic = nn.Sigmoid()

    def forward(self, user_indices, item_indices):
        user_embedding = self.embedding_user(user_indices)
        item_embedding = self.embedding_item(item_indices)
        element_product = torch.mul(user_embedding, item_embedding)
        logits = self.affine_output(element_product)
        rating = self.logistic(logits)
        return rating.squeeze()

    def init_weight(self):
        pass
```

MLP

Embedding

1. user
2. item

Concatenation

1. user
2. item

fully connected layers

```python
class MLP(nn.Module):
    def __init__(self, args, num_users, num_items):
        super(MLP, self).__init__()
        self.num_users = num_users
        self.num_items = num_items
        self.factor_num = args.factor_num #n of neurons in the hidden layer
        self.layers = args.layers
        self.num_layers = len(self.layers)
        self.embedding_user = nn.Embedding(num_embeddings=self.num_users, embedding_dim=self.factor_num)
        self.embedding_item = nn.Embedding(num_embeddings=self.num_items, embedding_dim=self.factor_num)

        self.fc_layers = nn.ModuleList()
        for idx in range(self.num_layers):
            if idx == 0:
                input_size = self.factor_num * 2  # Initial input size
            else:
                input_size = self.layers[idx - 1]  # Adjust input size based on previous layer's output size
            self.fc_layers.append(nn.Linear(input_size, self.layers[idx]))

        self.affine_output = nn.Linear(in_features=self.layers[-1], out_features=1)
        self.logistic = nn.Sigmoid()
        # Initialize weights
        self.init_weights() # comment if training only MLP

    def init_weights(self):
        init.normal_(self.embedding_user.weight, std=0.01)
        init.normal_(self.embedding_item.weight, std=0.01)
        for fc_layer in self.fc_layers:
            init.xavier_normal_(fc_layer.weight)
            init.constant_(fc_layer.bias, 0.0)

    def forward(self, user_indices, item_indices):
        user_embedding = self.embedding_user(user_indices)
        item_embedding = self.embedding_item(item_indices)
        vector = torch.cat([user_embedding, item_embedding], dim=-1)  # the concat latent vector of user and item embeddings
        for idx, _ in enumerate(range(len(self.fc_layers))):
            vector = self.fc_layers[idx](vector)

            vector = nn.ReLU()(vector)
            # vector = nn.BatchNorm1d()(vector)
            # vector = nn.Dropout(p=0.5)(vector)

        logits = self.affine_output(vector)
        rating = self.logistic(logits)
        return rating.squeeze()
```

NeuMF

embedded user,movie for MLP and GMF in their own space.

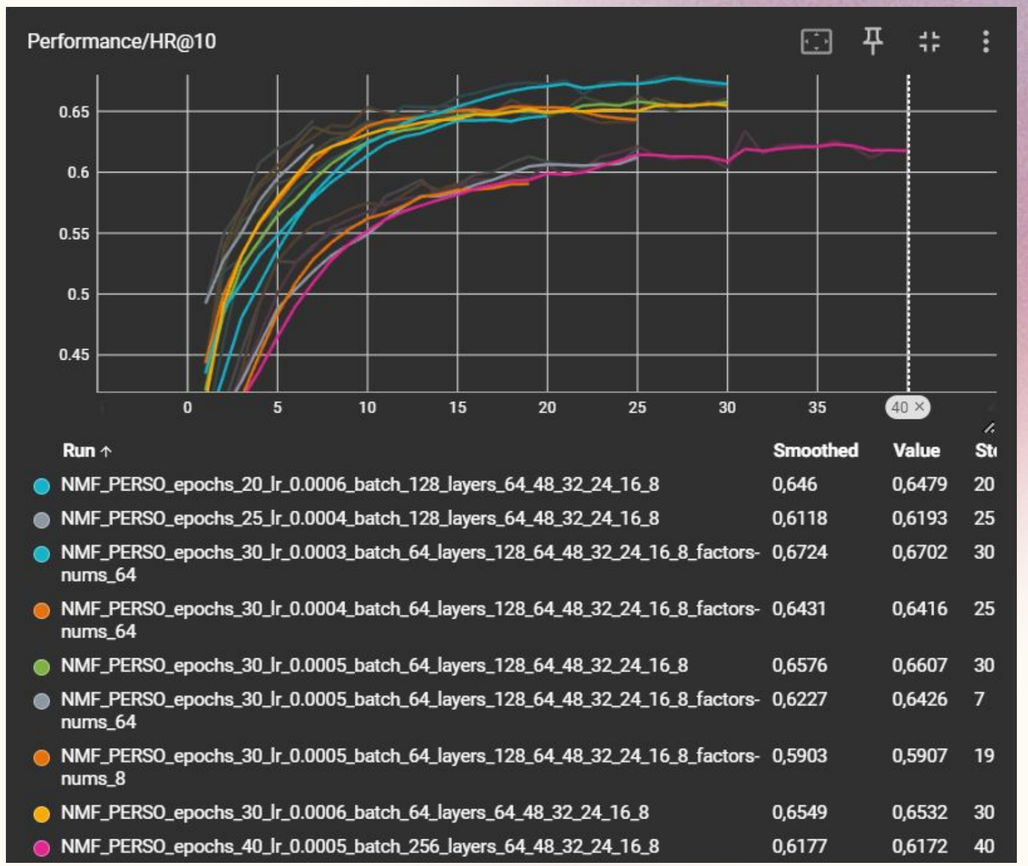
Then, **concatenate** the output of MLP and GMF and finally compute the sigmoid function.

note that : sigmoid function are not performed in GMF and MLP but only at the end of NeuMF.

```python
class NMF_perso(nn.Module):
    def __init__(self, args, num_users, num_items):
        super(NMF_perso, self).__init__()
        {...}


    def init_weight(self):
        {...}

    def forward(self, user_indices, item_indices):
        # Get embeddings for users and items for the MLP part
        user_embedding_mlp = self.embedding_user_mlp(user_indices)
        item_embedding_mlp = self.embedding_item_mlp(item_indices)
        # Get embeddings for users and items for the MF part
        user_embedding_mf = self.embedding_user_mf(user_indices)
        item_embedding_mf = self.embedding_item_mf(item_indices)

        # Apply MLP model
        mlp_vector = torch.cat([user_embedding_mlp, item_embedding_mlp], dim=-1)
        # Apply MF model
        mf_vector =torch.mul(user_embedding_mf, item_embedding_mf)

        # Pass the concatenated embeddings through the fully connected layers of the MLP model
        for idx, _ in enumerate(range(len(self.fc_layers))):
            mlp_vector = self.fc_layers[idx](mlp_vector)
            mlp_vector = nn.Dropout(p=self.dropout)(mlp_vector) # # Apply dropout to the output of each fully connected layer

        vector = torch.cat([mlp_vector, mf_vector], dim=-1) # Concatenate the output of the MLP and MF model
        logits = self.affine_output(vector)                 # Compute the Logits function
        rating = self.logistic(logits)                      # Apply the Logistic function to convert logits to ratings (probabilities)
```

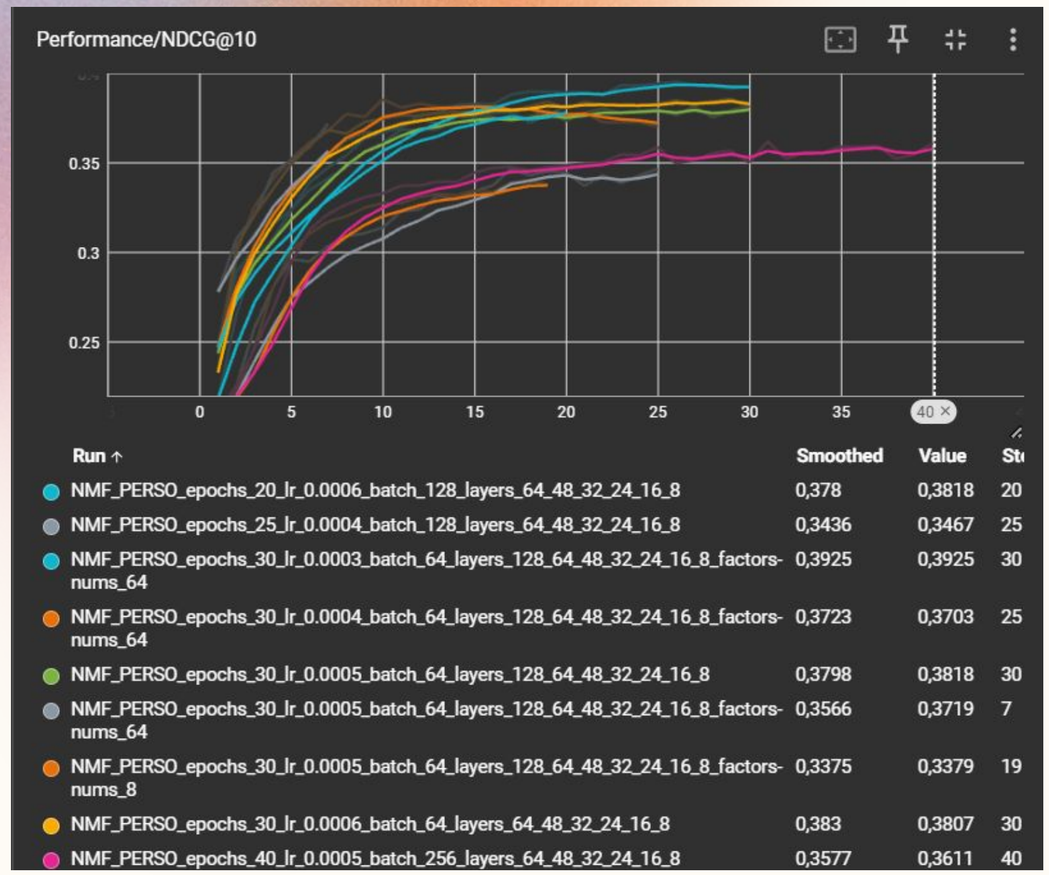
Performance results

HR



Run	Smoothed	Value	St
NMF_PERSO_epochs_20_lr_0.0006_batch_128_layers_64_48_32_24_16_8	0,646	0,6479	20
NMF_PERSO_epochs_25_lr_0.0004_batch_128_layers_64_48_32_24_16_8	0,6118	0,6193	25
NMF_PERSO_epochs_30_lr_0.0003_batch_64_layers_128_64_48_32_24_16_8_factors-nums_64	0,6724	0,6702	30
NMF_PERSO_epochs_30_lr_0.0004_batch_64_layers_128_64_48_32_24_16_8_factors-nums_64	0,6431	0,6416	25
NMF_PERSO_epochs_30_lr_0.0005_batch_64_layers_128_64_48_32_24_16_8	0,6576	0,6607	30
NMF_PERSO_epochs_30_lr_0.0005_batch_64_layers_128_64_48_32_24_16_8_factors-nums_64	0,6227	0,6426	7
NMF_PERSO_epochs_30_lr_0.0005_batch_64_layers_128_64_48_32_24_16_8_factors-nums_8	0,5903	0,5907	19
NMF_PERSO_epochs_30_lr_0.0006_batch_64_layers_64_48_32_24_16_8	0,6549	0,6532	30
NMF_PERSO_epochs_40_lr_0.0005_batch_256_layers_64_48_32_24_16_8	0,6177	0,6172	40

NDCG



Run	Smoothed	Value	St
NMF_PERSO_epochs_20_lr_0.0006_batch_128_layers_64_48_32_24_16_8	0,378	0,3818	20
NMF_PERSO_epochs_25_lr_0.0004_batch_128_layers_64_48_32_24_16_8	0,3436	0,3467	25
NMF_PERSO_epochs_30_lr_0.0003_batch_64_layers_128_64_48_32_24_16_8_factors-nums_64	0,3925	0,3925	30
NMF_PERSO_epochs_30_lr_0.0004_batch_64_layers_128_64_48_32_24_16_8_factors-nums_64	0,3723	0,3703	25
NMF_PERSO_epochs_30_lr_0.0005_batch_64_layers_128_64_48_32_24_16_8	0,3798	0,3818	30
NMF_PERSO_epochs_30_lr_0.0005_batch_64_layers_128_64_48_32_24_16_8_factors-nums_64	0,3566	0,3719	7
NMF_PERSO_epochs_30_lr_0.0005_batch_64_layers_128_64_48_32_24_16_8_factors-nums_8	0,3375	0,3379	19
NMF_PERSO_epochs_30_lr_0.0006_batch_64_layers_64_48_32_24_16_8	0,383	0,3807	30
NMF_PERSO_epochs_40_lr_0.0005_batch_256_layers_64_48_32_24_16_8	0,3577	0,3611	40

Explainable AI

Dataset

Movielens Dataset :

- 100,000 ratings
- from 1000 users on 1700 movies.

Goals

- Implement MLP model with more features.

- Explain the model using LIME and/or SHAP.

Challenges

Data

- Process the data.
- Correct the merging error and adapt the data structure for each column.

MLP with more features

- Reconstruct the functions and adapt the code for more features.
- Build a new MLP model with more features.

Explainable AI

- Explain with LIME and SHAP the contribution of the each features to the model.

Preprocessing

Merging the data

- Drop unnecessary columns.
- One hot encoding for the occupation and genres columns.
- Merging in one big dataset.

```python
u_item.rename(columns={'movie_id':'item_id'},inplace=True)

# Merge u_user and u_data
data_merged = pd.merge(u_item, u_data, on="item_id", how="left")

# Merge u_user, u_data and u_item
data_merged = pd.merge(data_merged, u_user, on='user_id', how="left")
data_merged.drop(columns = ["video_release_date","IMDb_URL","unknown", "movie_title", 'release_date'],
inplace =True)

data_dummies = pd.concat([data_merged,
                          pd.get_dummies(data_merged['occupation'],  dtype = int),
                          ], axis=1)
data_dummies.drop(columns=["occupation"],inplace=True)

data_dummies
```

Dataset

Reconstruct the dataset and function

We had to readapt all the functions regarding the construction of the dataset.

Example : get_train_instance

Variables considered

users, items, genders, ages, genres, occupations.

```python
def get_train_instance(self):
      users, items, genders, ages, ratings, genres, occupations= [], [], [], [], [], [], []
      # merge together self.train_ratings and self.negatives.
      genre_list= ['Action', 'Adventure', 'Animation', "Childrens", 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy', 'FilmNoir', 'Horror', 'Musical', 'Mystery', 'Romance', 'SciFi', 'Thriller', 'War', 'Western']
      occupation_list = ['administrator','artist','doctor','educator','engineer','entertainment','executive','healthcare','homemaker','lawyer','librarian','marketing','none','other','programmer','retired','salesman','scientist','student','technician','writer']

      train_ratings = pd.merge(self.train_ratings, self.negatives[['user_id', 'negative_items']], on='user_id')
      train_ratings['negatives'] = train_ratings['negative_items'].apply(lambda x: random.sample(x, self.num_ng))
      for row in train_ratings.itertuples():
          users.append(int(row.user_id))
          items.append(int(row.item_id))
          genders.append(int(row.gender))
          ages.append(int(row.age))
          genre_vector = [int(row.__getattribute__(column)) for column in genre_list]
          genres.append(genre_vector)
          occupation_vector = [int(row.__getattribute__(column)) for column in occupation_list]
          occupations.append(occupation_vector)
          ratings.append(float(row.rating))
          for i in range(self.num_ng):
              users.append(int(row.user_id))
              items.append(int(row.negatives[i]))
              genders.append(int(row.gender))
              ages.append(int(row.age))
              genre_vector = [int(row.__getattribute__(column)) for column in genre_list]
              genres.append(genre_vector)
              occupation_vector = [int(row.__getattribute__(column)) for column in occupation_list]
              occupations.append(occupation_vector)
              ratings.append(float(0))  # negative samples get 0 rating
      # dataset contain the information of the user
      dataset = Rating_Datset( # Rating_Dataset
          user_list=users,
          item_list=items,
          gender_list=genders,
          age_list = ages,
          rating_list=ratings,
          genre_matrix = genres,
          occupation_matrix = occupations
          )
      return DataLoader(dataset, batch_size=self.batch_size, shuffle=True, num_workers=4)
```

Model MLP

MLP with all the features

Embedding :

1. user
2. item
3. gender
4. age

Merge and concatenate the embeddings and the matrix of one hot encoding.

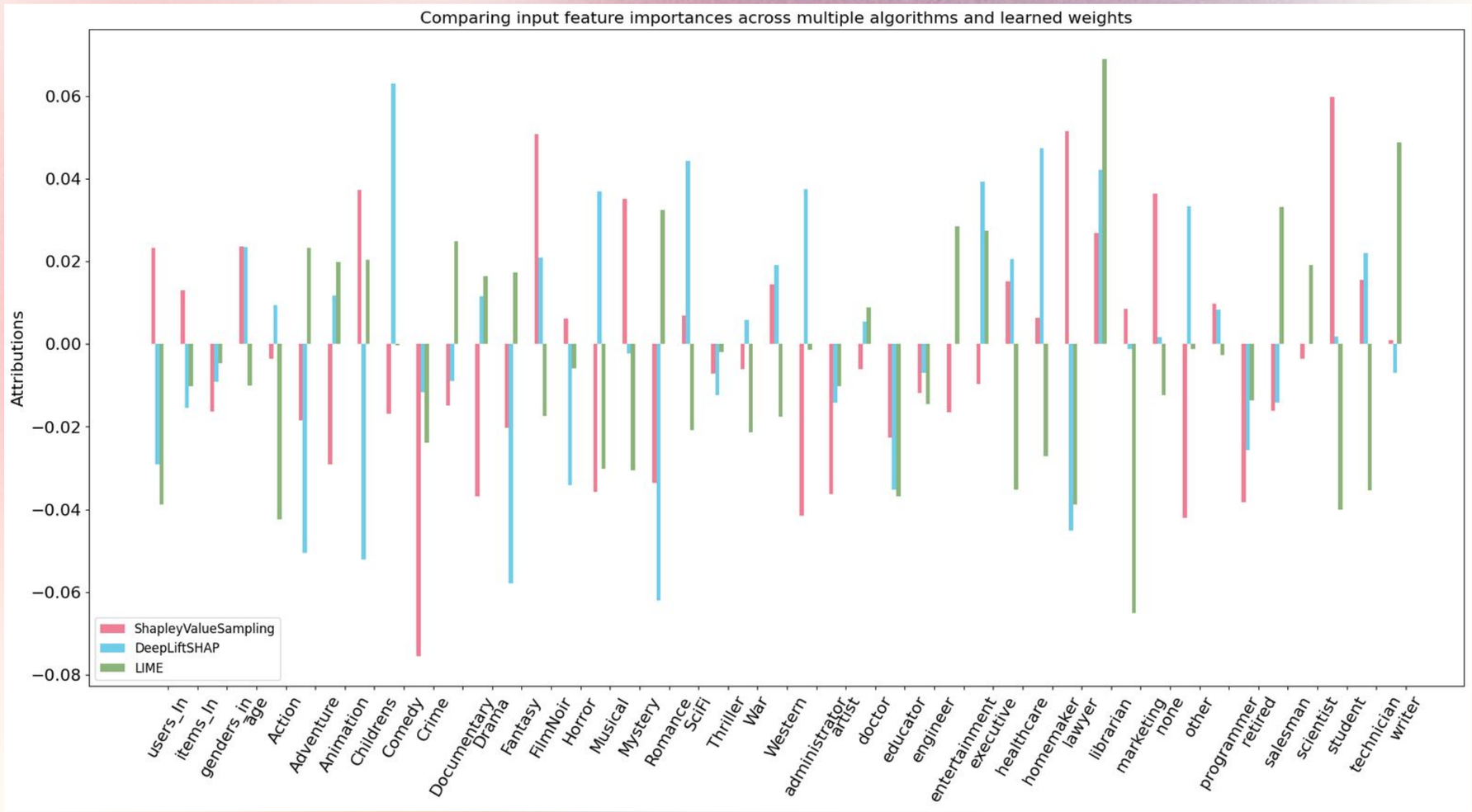
```python
class MLPcustom(nn.Module):
    def __init__(self, args, num_users, num_items, num_gender, num_age, num_genres, num_occupations):
        super(MLPcustom, self).__init__()
        self.num_users = num_users
        self.num_items = num_items
        self.num_gender = num_gender  # New parameter for the number of gender categories
        self.factor_num = args.factor_num
        self.layers = args.layers
        self.num_age = num_age
        self.num_genres = num_genres
        self.num_occupations = num_occupations
        self.dropout = args.dropout

        self.embedding_user = nn.Embedding(num_embeddings=self.num_users, embedding_dim=self.factor_num)
        self.embedding_item = nn.Embedding(num_embeddings=self.num_items, embedding_dim=self.factor_num)
        self.embedding_gender = nn.Embedding(num_embeddings=self.num_gender,
embedding_dim=self.factor_num)  # Embedding for gender
        self.embedding_age = nn.Embedding(num_embeddings=self.num_age, embedding_dim=self.factor_num)

        # Fully connected layers
        self.fc_layers = nn.ModuleList()
        for idx, (in_size, out_size) in enumerate(zip(self.layers[:-1], self.layers[1:])):
            self.fc_layers.append(nn.Linear(in_size, out_size))

        # Output layer
        self.affine_output = nn.Linear(in_features=self.layers[-1], out_features=1)
        self.logistic = nn.Sigmoid()



    def forward(self, user_indices, item_indices, gender_indices, age_indices, genre_vectors,
occupation_vectors):  # Add gender_indices to the forward pass
        # Lookup user, item, and gender embeddings

        user_embedding = self.embedding_user(user_indices)
        item_embedding = self.embedding_item(item_indices)
        gender_embedding = self.embedding_gender(gender_indices)  # Lookup gender embeddings
        age_embedding = self.embedding_age(age_indices)
        genre_matrix = torch.stack([torch.tensor(genre_vector) for genre_vector in genre_vectors])
        occupation_matrix = torch.stack([torch.tensor(occupation_vector) for occupation_vector in
occupation_vectors])

        # Concatenate user, item, and gender embeddings along the last dimension
        vector = torch.cat([user_embedding, item_embedding, gender_embedding, age_embedding, genre_matrix,
occupation_matrix], dim=-1)

        # Pass the concatenated vector through each fully connected layer followed by ReLU activation
        for idx, _ in enumerate(range(len(self.fc_layers))):
            vector = self.fc_layers[idx](vector)
            vector = nn.ReLU()(vector)
            vector = nn.Dropout(p=self.dropout)(vector)

        # Pass the resulting vector through the output layer to obtain logits
        logits = self.affine_output(vector)
        rating = self.logistic(logits)
        return rating.squeeze()
```

Explainable AI

Features

- users_In
- items_In
- genders_in
- age_in
- genre_list
- occupation_list

genre_list and occupation_list are displayed as their values.



Graph Representation Learnin with Node2Vec

Dataset

Movielens Dataset :

- 100,000 ratings
- from 1000 users on 1700 movies.

Goals

- Graphs representation for movie
- Find the most similar movies to a target movie

Challenges

Graph

- construct and understand the graph based on our dataset.
- Build graph exploration functions : Next_step, random_walk.
- Build a suitable training dataset.

Train a Word2Vec model

- define and train the Word2Vec model that learns word (movie) embeddings by predicting context words (movies).

Represent the 5 most similar movies according to our model

- compute the distance with cosine similarity and euclidian distance.
- PCA reduction to obtain a representable dimension space.

Training data

Skipgram

Generate skipgrams for the the context target movies with window size = 2.

Negative sampling

Generates a set of 4 random samples among all the movies, as negative sampling.

```python
# Function to generate skip-grams from a given sequence
def generate_skipgrams(sequence):
    """
    Generates skip-gram pairs from the input sequence.

    Args:
        sequence (list): A list of tokens or words representing the input sequence (movie ids).

    Returns:
        list: A list of skip-gram pairs, where each pair is a tuple of two tokens/words.
    """
    # Initialize an empty list to store the skip-gram pairs
    skip_grams = []

    # Iterate over each word in the sequence, excluding the first and last words
    for i in range(1, len(sequence) - 1):
        # Define the input word (target word)
        input = sequence[i]

        # Define the context words as the previous and next words in the sequence
        context = [sequence[i - 1], sequence[i + 1]]

        # Create skip-gram pairs and append them to the list
        for w in context:
            skip_grams.append((input, w))

    # Return the list of generated skip-gram pairs
    return skip_grams



def negative_sampling(true_class, num_sampled, vocab_size):
    """
    Generates negative samples for the Noise Contrastive Estimation (NCE) loss.

    Args:
        true_class (int): The index of the true class (positive sample).
        num_sampled (int): The number of negative samples to generate.
        vocab_size (int): The size of the vocabulary.
        seed (int, optional): The random seed for reproducibility. Defaults to None.

    Returns:
        list: A list of indices of the negative samples.
    """

    # Ensure uniqueness of negative samples
    negatives = set()
    while len(negatives) < num_sampled:
        neg_sample = random.randint(0, vocab_size - 1)
        if neg_sample != true_class:
            negatives.add(neg_sample)

    return list(negatives)
```

Training data

The training data is generated by returning :

- the target movie from the walks generated with random_walks.
- its context couples (true and negative samples).
- labels (1 for true and 0 for negative).

```python
def generate_training_data(sequences, num_ns, vocab_size):
    """
    Generate training data for the word2vec model using the skip-gram approach.

    Args:
        sequences (list of list of ints): List of tokenized sentences (sequences), in our case the random walks.
        num_ns (int): Number of negative sampling examples to generate for each positive context word.
        vocab_size (int): Size of the vocabulary.
        seed (int): Random seed for reproducibility.

    Returns:
        targets (list of ints): List of target words.
        contexts (list of torch.Tensor): List of context word vectors (positive and negative samples).
        labels (list of ints): List of labels indicating if the context word is a positive or negative sample.
    """
    targets, contexts, labels = [], [], []

    # Iterate over all sequences (sentences) in the dataset.
    for sequence in tqdm(sequences):
        # Generate positive skip-gram pairs for a sequence (sentence).
        # This function extracts word pairs within the specified window size.
        positive_skip_grams = generate_skipgrams(sequence)

        # Iterate over each positive skip-gram pair to produce training examples
        # with a positive context word and negative samples.
        for target_word, context_word in positive_skip_grams:
            # Convert the context word to a tensor and add an extra dimension for batch processing.
            context_class = torch.tensor([context_word], dtype=torch.int64)
            context_class = context_class.unsqueeze(1)

            # Generate negative sampling candidates for the context word.
            # `negative_sampling` randomly samples `num_ns` words from the vocabulary.
            negative_sampling_candidates = torch.tensor(
                negative_sampling(context_word, num_ns, vocab_size), dtype=torch.int64)

            # Build context and label vectors (for one target word)
            # Context vector contains the positive context word followed by negative sampling candidates.
            context = torch.cat((context_class.squeeze(1), negative_sampling_candidates), 0)
            # Label vector indicates if the context word is a positive or negative sample.
            label = torch.tensor([1] + [0] * num_ns, dtype=torch.int64)

            # Append each element from the training example to global lists.
            targets.append(target_word)
            contexts.append(context)
            labels.append(label)

    return targets, contexts, labels

```

Word2Vec Models

Embedded words

Create a embedding for words (movies) contexts and targets.

Forward pass

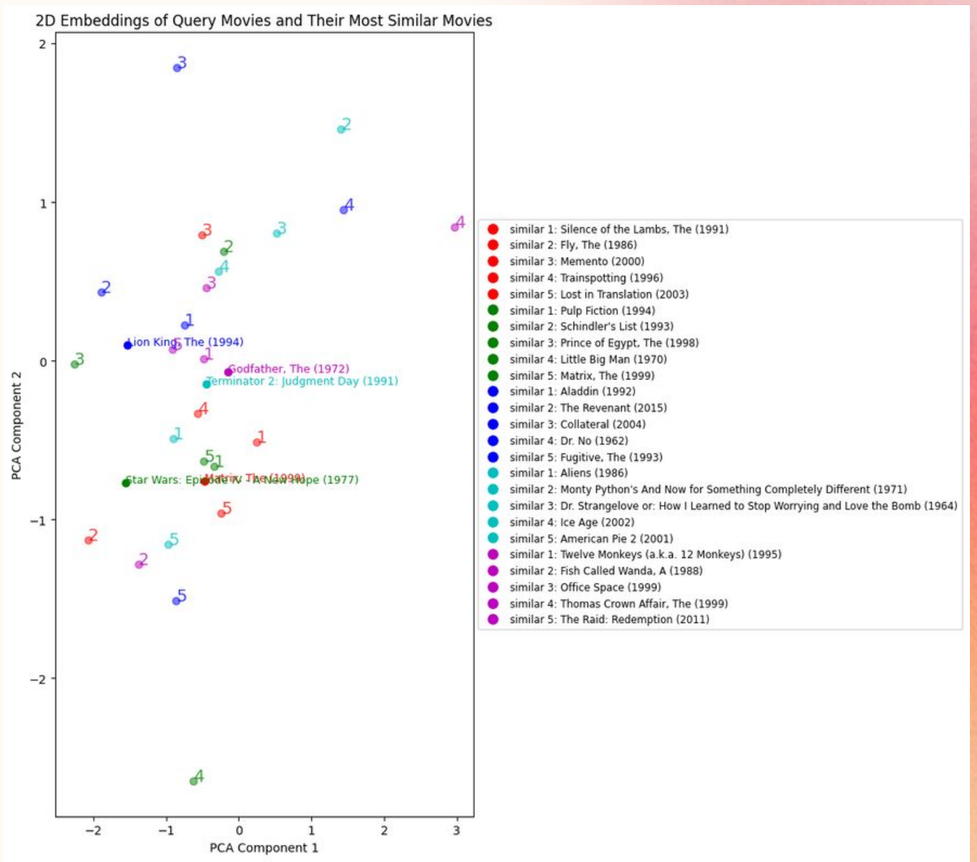
Computes the dot products between the target and context word embeddings, producing a matrix of shape `(batch, context)

- 'be' shape for word embeddings
- 'bce' shape for context embeddings
- 'bc' Desired output shape

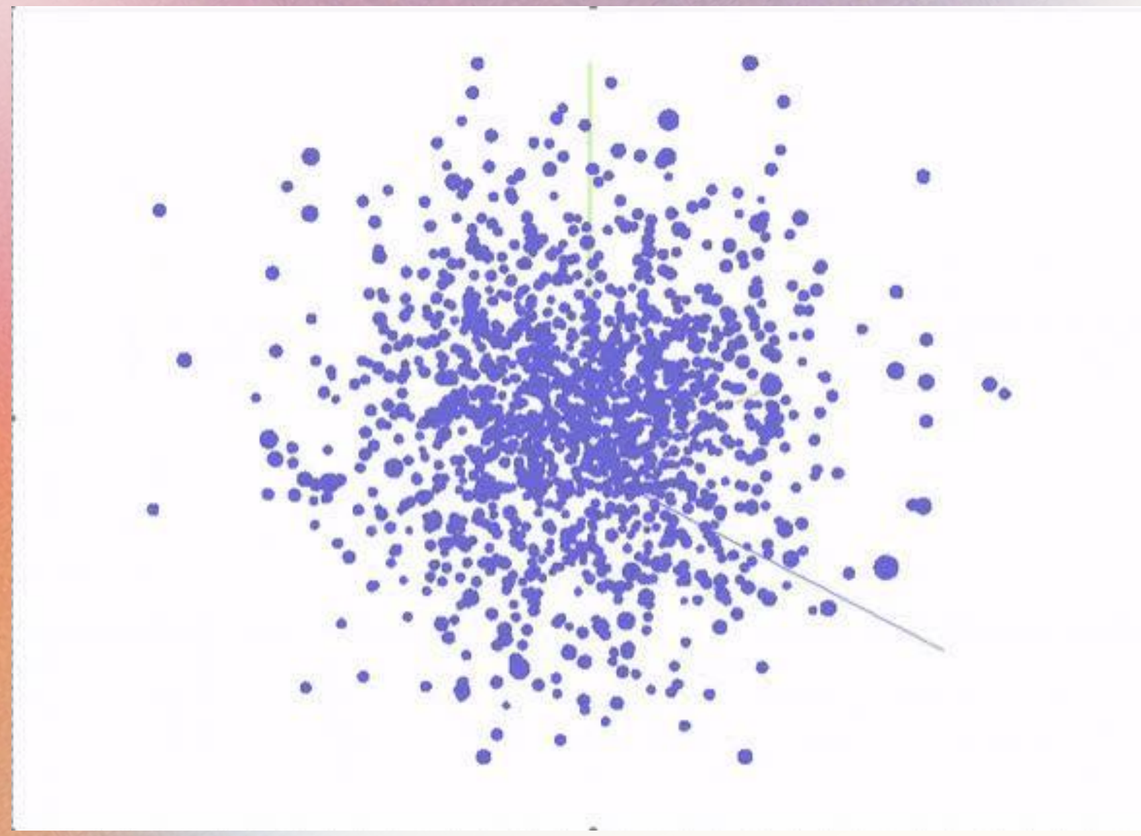
```python
class Word2Vec_new(nn.Module):
    """
    Word2Vec model that learns word (movie) embeddings by predicting context words (movies).
    """
    def __init__(self, vocab_size, embedding_dim):
        """
        Initialize the Word2Vec model.

        Args:
            vocab_size (int): Size of the vocabulary.
            embedding_dim (int): Dimensionality of the embeddings.
        """
        super(Word2Vec_new, self).__init__()
        # Create two embedding layers for target and context words
        self.target_embedding = nn.Embedding(vocab_size, embedding_dim)
        self.context_embedding = nn.Embedding(vocab_size, embedding_dim)


    def forward(self, pair):
        """
        Forward pass of the Word2Vec model.

        Args:
            pair (tuple): A tuple containing target and context word indices.

        Returns:
            torch.Tensor: Dot products between target and context embeddings.
        """
        target, context = pair

        word_emb = self.target_embedding(target)  # (batch, embed)
        context_emb = self.context_embedding(context)  # (batch, context, embed)

        """
        Explanation :
        Compute dot products between target embeddings and context embeddings
        The result is a matrix of shape (batch, context) where each element is the dot product
        between the embedding of the target word and the embedding of its corresponding context word
        """
        dots = torch.einsum('be,bce->bc', word_emb, context_emb)

        return dots

```

Similarities Visualization

2D Embeddings



3D Tensorboard Latent space



Attention-based Recommender System

Dataset

Movielens Dataset :

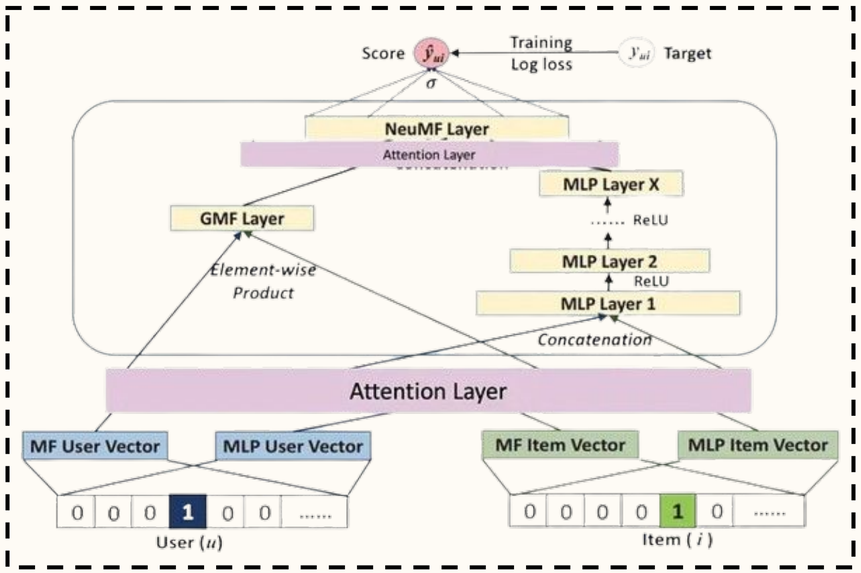
- 100,000 ratings
- from 1000 users on 1700 movies.

Goals

- build a multi head attention layer that could enhance our model performances.

- Compare the models performances with and

Why we put the attention layer there ?



After embedding

- We want the enhance the embedding of the data with the multi head attention layer.

After computation

- We add an attention layer after the "computation" layer in order to use the attention layer as a weighted separator. This layer will prefer the representation of GMF or MLP depending on their quality of representation.

MHSelfAttention

Increase the dimension of the space to represent features from our movies.

Concatenate the different heads for the output representation.

The goals is to enhance the model's ability to focus on different parts of the input sequence simultaneously.

```python
class MHSelfAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super(MHSelfAttention, self).__init__()

        # initialize MHSelfAttention
        self.num_heads = num_heads
        self.embed_dim = embed_dim
        self.head_dim = embed_dim // num_heads
        assert self.head_dim * num_heads == embed_dim, "Embedding dimension must be divisible by number of heads"

        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)
        self.out = nn.Linear(embed_dim, embed_dim)

    def forward(self, x):
        batch_size = x.size(0)

        # Linear projections for Query, Key and value
        Q = self.query(x)
        K = self.key(x)
        V = self.value(x)

        # Split into heads
        Q = Q.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.head_dim, dtype=torch.float32))
        attn = F.softmax(scores, dim=-1)
        context = torch.matmul(attn, V)

        # Concatenate heads
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, self.embed_dim)

        return self.out(context)

```

4. Attention-based Recommender System

Attention NeuMF

We **added the attention layer** after the initial embeddings in GMF and MLP and after the concatenation of their outputs.

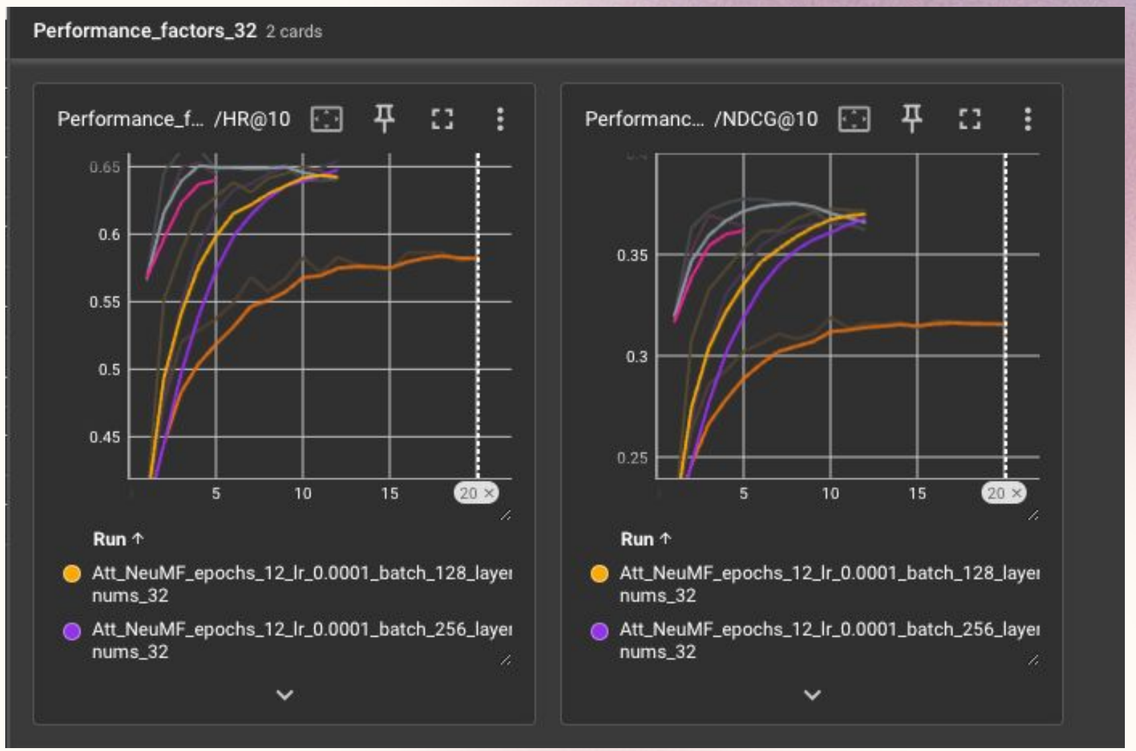
We put **heads attention = 8** for the 1st attention layers.

We put **heads attention = 3** for the 2nd attention layers based on the size of of the input vectors.

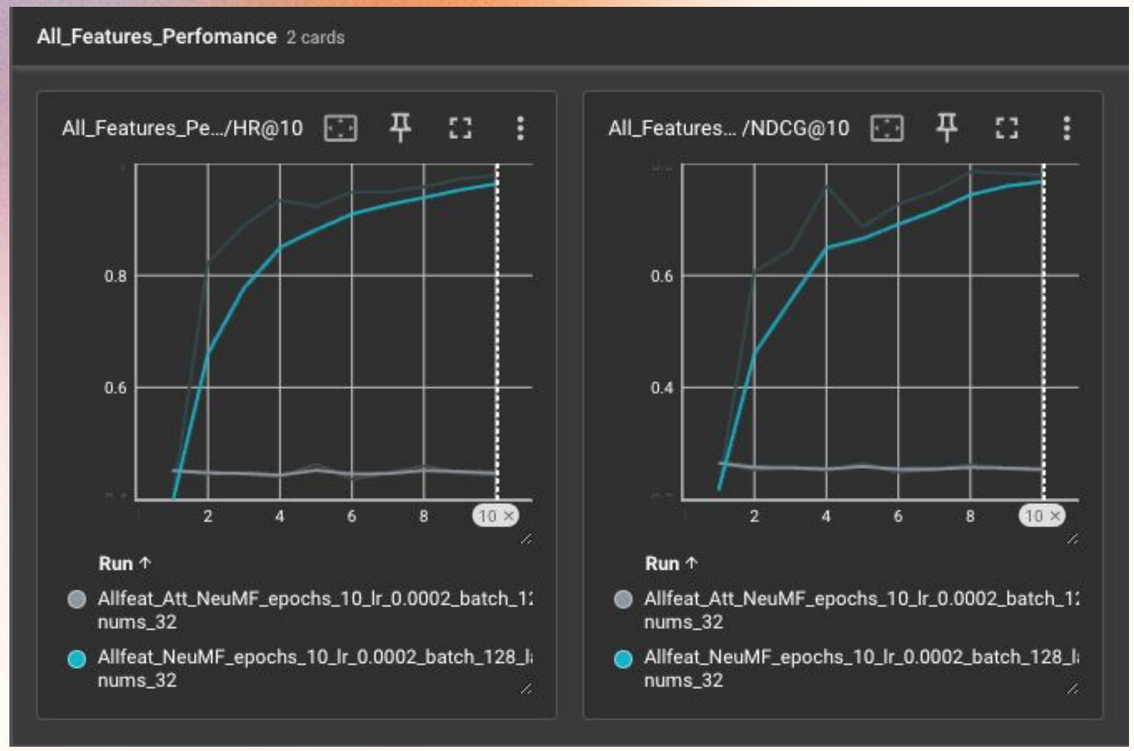
```python
class Att_NeuMF_All(nn.Module):
    def __init__(self, args, num_users, num_items, num_gender, num_age, num_genres, num_occupations, heads=8):
        super(Att_NeuMF_All, self).__init__()

        {...}

        # Attention MLP
        self.attention = MHSelfAttention(self.factor_num_mlp, heads)
        self.attention_mf = MHSelfAttention(self.factor_num_mf, heads)

        {...}

        # Define combined dimension and multi-head self-attention for combined vector
        combined_dim = in_features=args.layers[-1] + input_dim_mlp
        self.combination_attention = MHSelfAttention(combined_dim, 3)

        {...}

    def init_weight(self):
        {...}

    def forward(self, user_indices, item_indices, gender_indices, age_indices, genre_vectors, occupation_vectors):

        {...}

        # Apply attention mlp
        user_embedding_mlp = self.attention(user_embedding_mlp.unsqueeze(1)).squeeze(1)
        item_embedding_mlp = self.attention(item_embedding_mlp.unsqueeze(1)).squeeze(1)
        gender_embedding_mlp = self.attention(gender_embedding_mlp.unsqueeze(1)).squeeze(1)
        age_embedding_mlp = self.attention(age_embedding_mlp.unsqueeze(1)).squeeze(1)

        # Apply attention mf
        user_embedding_mf = self.attention_mf(user_embedding_mf.unsqueeze(1)).squeeze(1)
        item_embedding_mf = self.attention_mf(item_embedding_mf.unsqueeze(1)).squeeze(1)
        gender_embedding_mf = self.attention_mf(gender_embedding_mf.unsqueeze(1)).squeeze(1)
        age_embedding_mf = self.attention_mf(age_embedding_mf.unsqueeze(1)).squeeze(1)

        {...}

        # Combine GMF and MLP vectors using multi-head self-attention
        combined_vector = torch.cat([mf_vector, mlp_vector], dim=-1)
        combined_vector = self.combination_attention(combined_vector.unsqueeze(1)).squeeze(1)
        '''
        Why Use Unsqueeze and Squeeze?
          Input Shape for Attention Mechanism
          The multi-head self-attention mechanism (MHSelfAttention) typically expects the input tensor to have the following shape:
          (batch_size, seq_length, embed_dim)
          Here, seq_length is the sequence length, and embed_dim is the embedding dimension.

          In the context of natural language processing, seq_length would represent the number of tokens in a sentence.
          For our use case, we want to apply attention to the combined GMF and MLP vector, which can be thought of as a single "sequence" of embeddings.
        '''

        logits = self.affine_output(combined_vector)
        rating = self.logistic(logits)
        return rating.squeeze()

```

Performance comparison

2 Features Models



All Features Models



UNIMIB, UNIMI, UNIPV

Thank you for attention !

Ulysse Duvillier - Diego Corna - Andrea Barbieri

Master AI4ST