Project Report - ECE 176

Giacomo Visentin
A18158031

Abstract

This project is about the usage of neural networks in constructing a collaborative filtering movie recommender system using Python, Keras, and TensorFlow. The experiment uses the MovieLens database, which includes over 100,000 ratings from 600 people. The goal is to create a neural network architecture that predicts the ratings of unrated movies and provides users with a top-ten movie suggestion list. After creating the deep learning model, its performance is compared with that obtained from a recommender system based on a more traditional algorithm namely Matrix factorization to test the potential of neural networks.

1 Introduction

Many modern applications and websites, both for computers and mobile devices, employ recommender systems to provide personalized suggestions to users, enhancing their experience. These systems, relying on Machine Learning algorithms, are widely used in various contexts ranging from entertainment and social media to online sales platforms.

Recommender systems are often implemented using machine learning algorithms such as Matrix Factorization. However, There have been recent attempts to create algorithms using deep learning, but there are not many studies highlighting the performance supremacy of these new algorithms over traditional ones. Therefore, I would like to try creating a neural network to analyze the quality of the results obtained.

The simplest and most immediate way to create such a system is to recommend the most popular items, i.e., those that are most purchased or viewed. For example, YouTube features a "Trending" section based on this elementary concept, while Netflix suggests the most-watched movies in collections like "Trending Now" or "Top 10 Series in Italy Today".

This type of recommendation is particularly useful for new users who haven't interacted with any items yet. However, by increasing the complexity of the model, personalized recommendations can be created for each user, yielding more precise and effective results.

We can divide personalized recommendation systems into two broad categories plus the hybrid one:

- Collaborative Filtering:
  - User-Based: Recommends items similar to those liked by users with similar tastes.
  - Item-Based: Recommends items similar to those already appreciated by the user.
- Content-Based Filtering:
  Analyzes the features of recommended items and suggests those similar to those already liked by the user, based on attributes such as keywords, categories, etc.
- Hybrids:
  Combines collaborative filtering and content-based approaches to achieve more accurate and robust recommendations.

I want to build simple movie recommender system based on collaborative filtering, implemented in Python using the Keras and TensorFlow libraries and then compare the results obtained with a Matrix

Factorization based model trained with the same dataset. The aim is to develop a model that is able to predict the user's rating for an unrated item. My model will then return to the user a list of ten movie recommendations that are the ten highest prediction of unrated movies.

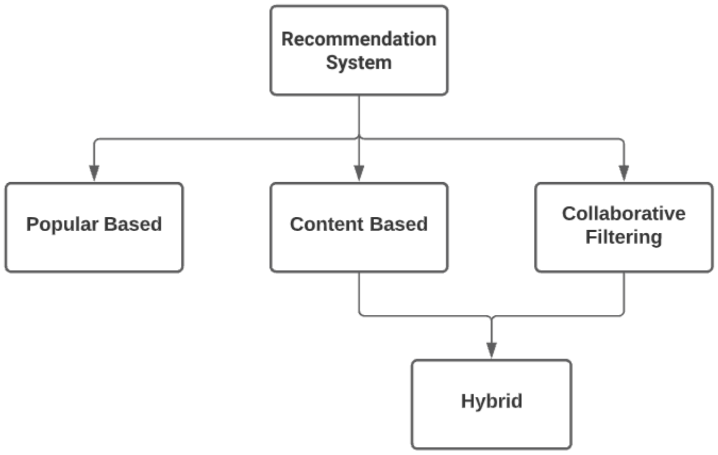

Figure 1: Different types of recommender systems

2 Related Work

The first useful paper I mention is "systematic review and research perspective on recommender systems"[1] provides an overview of the various types of recommendation systems.

"Deep Learning based Recommender System: A Survey and New Perspectives"[2] is a paper about the explanation of some possible neural network architectures to use for recommendation systems. It turned out to be a very important paper because it provided me with the initial architecture that I then modified to adapt it to the specific problem I intended to solve. I also used it to understand better the advantages and disadvantages of Recommender System models based on neural network compared to the traditional ML model.

2.1 Matrix Factorization

This paper "Matrix Factorization Techniques for Recommender Systems"[3] was very useful for creating the traditional model. It provides an explanation of how Matrix Factorization works. Matrix factorization is a technique used to decompose a complex matrix into two simpler matrices. The goal is to find a compact representation of the data. Essentially, the original matrix is approximated by multiplying two matrices of reduced dimensions, which can be interpreted as latent representations of the data.

Both users and items are mapped to a joint latent factor space of dimensionality k. The matrix R denote the interaction matrix has dimension of of size (m, n) with n users and m items. The purpose of the algorithm is to find a matrix P of size (n, k) and a matrix Q of size (m, k) such that the predicted rating is $R = PQ^T$. The learning rate is α, λ denotes the regularization rate.

To learn each entry of the matrices P and Q, we want to minimize the mean squared error between predicted rating scores and real rating scores:

$$\min_{q^\star, p^\star} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\|q_i\|^2 + \|p_u\|^2)$$

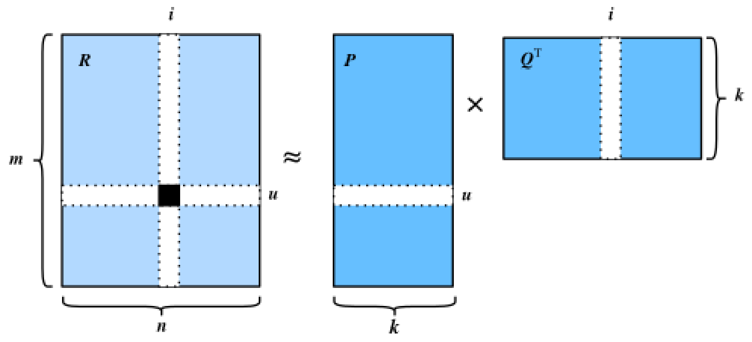

Figure 2: Dimensions of matrices in Matrix Factorization

so the error in the prediction is:

$$e_{ui} \overset{def}{=} r_{ui} - q_i^T p_u$$

We can then update P and Q with this equations:

- $q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i)$
- $p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$

We can also have biases: bias b_u for each user, a bias b_i for each item, and a global bias μ:

$$b_{ui} = \mu + b_i + b_u$$

and so the function to minimize is:

$$\min_{p^\star, q^\star, b^\star} \sum_{(u,i) \in \kappa} (r_{ui} - \mu - b_u - b_i - {p_u}^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2 + {b_u}^2 + {b_i}^2)$$

3 Method

To create the collaborative filtering recommendation model, I decided to use the Python programming language along with the Keras and Tensorflow libraries, having used them in the past.

The implementation of the recommendation model was divided into several phases. Initially, the data preprocessing phase was executed, where various operations and transformations were applied to the raw data in the datasets to make them suitable for use by the model. Subsequently, the architecture was developed, which was then subjected to the training and evaluation phase.

3.1 Data preprocessing

The "ratings" and "movies" datasets were loaded into a Colab notebook, and the "timestamp" column was removed from the former as it is not relevant for the model being developed. Then, the two datasets were merged into a single dataset by combining the information based on the "movieId" column. Through this procedure, each review (corresponding to a row in the rating dataset) was enriched with the columns from the movies dataset related to the film with the same movieId. This step may seem insignificant, but it actually allows for identifying and removing some films that have never received reviews from users. Those films, not having reviews, cannot be considered in a collaborative filtering-based model and thus must be excluded.

Through this simple procedure, the films present in the movies dataset that are not present in the rating dataset were eliminated, reducing the total number of films used in the model from 9742 to 9724.

The opposite situation, where some films reviewed by users were not present in the movies dataset, could not be excluded a priori. In such a case, it would have been necessary to eliminate the review because we would not have had all the information related to the film (such as title, genre, and release year). However, the number of reviews remained constant, as did the number of users (610), implying that all reviewed films were present in the movies dataset.

Accessing specific films in the dataset was not straightforward because the movieIds assigned were random numbers and did not follow a sequential order. Therefore, it was necessary to map them to a sequence from 0 to 9723, and the LabelEncoder[w1] library from scikit-learn was used for this purpose.

I then created two separate datasets for the training and testing phases using the train_test_split[w2] function from the scikit-learn library. With this procedure, the data preprocessing phase was concluded.

3.2 Architecture

Initially, I created a simple neural network with few layers, and based on the results obtained from that network, I adjusted its complexity by increasing or decreasing it.

The architecture of the model consists of two branches, one for the movie and one for the user. Both branches take a numerical ID as input and pass it through an Embedding Module[w3], which converts the ID into a feature vector. The embedding modules are key layers in this architecture.

The two branches are then merged into a single one by concatenating the two feature vectors. They are then passed through a series of fully connected layers. The last layer is a single neuron that produces the predicted movie score.

3.3 Training

To train the model, I used the fit function of Keras. The number of epochs is kept low to prevent overfitting.

3.3.1 Loss Function

I decided to use mean squared error as the loss function. Since my goal was to create a recommendation system that suggests ten movies, it's easy to understand that the aim was to obtain predictions that were sufficiently accurate to identify whether a movie might please a user or not, rather than aiming for extremely precise individual predictions at the expense of others largely inaccurate. The mean squared error allowed minimizing the larger errors.

3.4 Test

To test the model and during the Hyperparameter tuning phase, I didn't solely rely on the value of the loss function. It was the most important evaluation metric, but I also assessed, for example, the number of reviews that appeared as outliers. The rating range of movies in the dataset used is from 0.5 to 5, so values lower than 0.5 or greater than 5 were considered outliers to me. Values that approached 0 or slightly exceeded 5 were not considered problematic, but some tested architectures also produced predictions close to 6. I always sought to favor models with hyperparameters that produced predictions within the correct range

3.5 Advantages and disadvantages of deep learning models

The disadvantages of deep learning models compared to traditional ones are[2]:

- Interpretability: Despite its success, deep learning is notorious for behaving like black boxes, making it challenging to provide explainable predictions. A common critique of deep neural networks is that the hidden weights and activations are typically non-interpretable, which limits their explainability;
- Data Requirement: Another potential limitation is that deep learning is known to be data-hungry, requiring a sufficient amount of data to fully support its rich parameterization;
- Extensive Hyperparameter Tuning: A third well-established argument against deep learning is the need for extensive hyperparameter tuning, which I will discuss in more detail later and which has consumed much of the time devoted to this project;

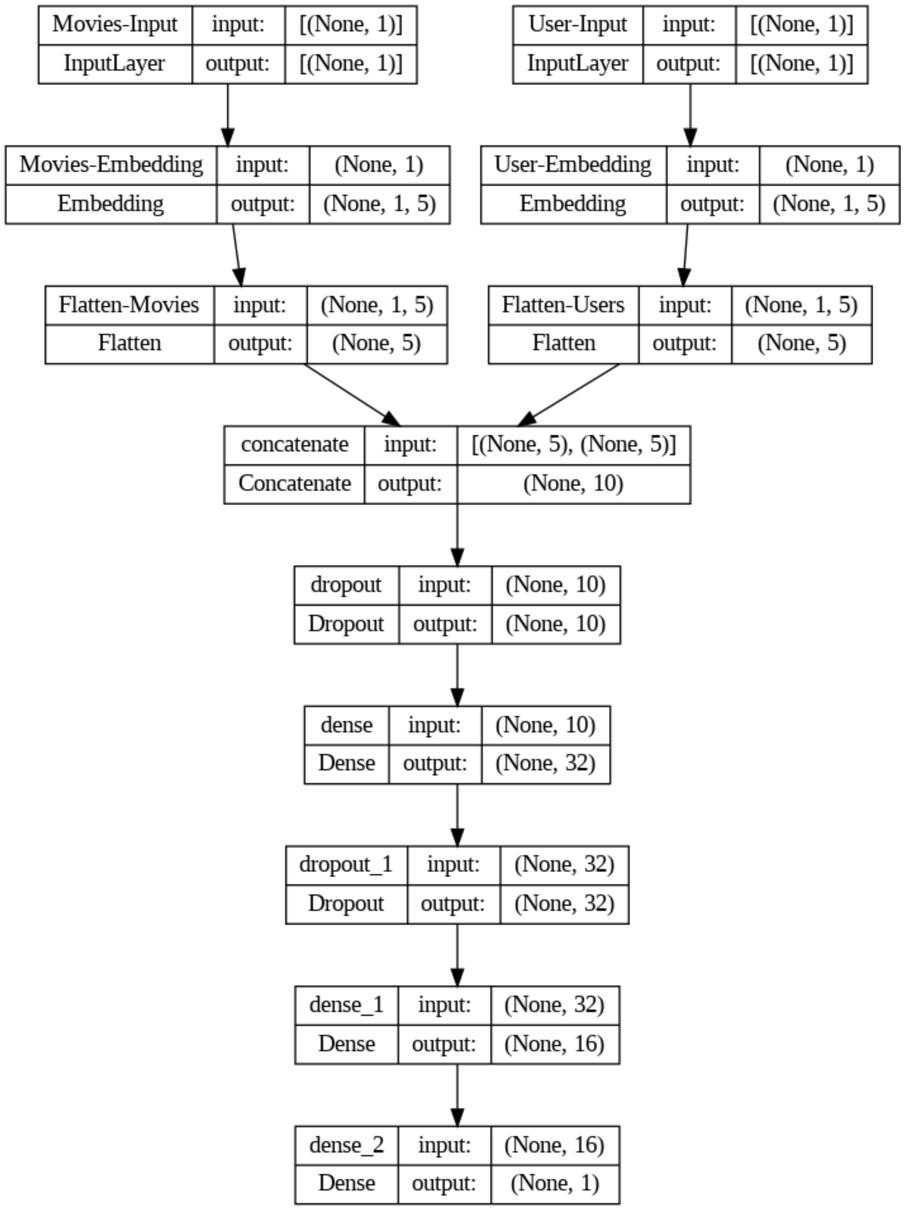

Figure 3: Model Architecture

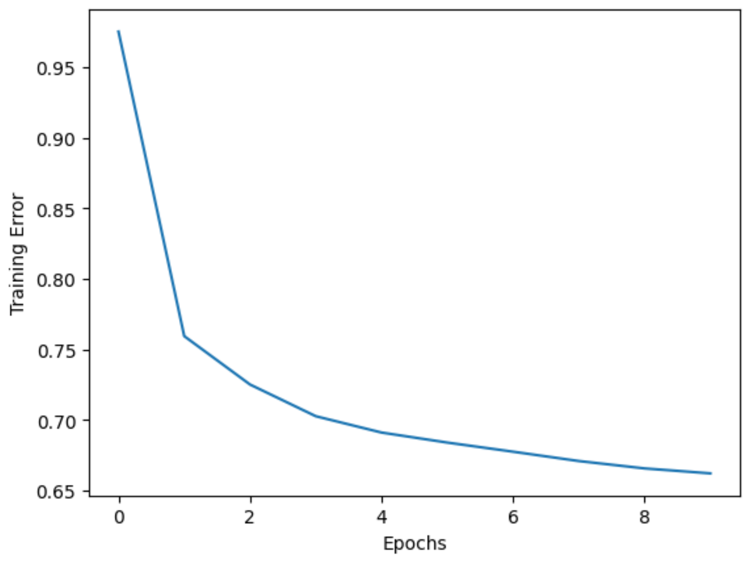

Figure 4: training error graph during the model training as a function of epochs

On the other hand, the advantages are:

- Nonlinear Transformation: Unlike linear models, deep neural networks are capable of modeling the nonlinearity in data with nonlinear activations such as ReLU, sigmoid, tanh, etc. This property enables them to capture complex and intricate user-item interaction patterns. Conventional methods such as matrix factorization, factorization machine, and sparse linear models are essentially linear;
- Representation Learning: Deep neural networks are effective in learning the underlying explanatory factors and useful representations from input data. In general, a large amount of descriptive information about items and users is available in real-world applications. I leveraged this advantage minimally in my project due to my choice to create a model based solely on reviews and without using additional features;
- Flexibility: Deep learning techniques offer high flexibility, especially with the emergence of popular deep learning frameworks such as TensorFlow, Keras, PyTorch, etc. Most of these tools are developed in a modular way and have an active community;

4 Experiments

4.1 Datasets

I chose the MovieLens[w4] dataset as the database for my project. This database was specifically created by a research group at the University of Minnesota with the aim of enhancing and refining recommendation systems. Furthermore, the MovieLens database undergoes constant updates and comes in different versions that mainly differ in the sizes of the data they contain. For my project, I chose to adopt the latest available version (September 2018) with a limited number of movies and reviews. It includes 100,000 movie ratings made by approximately 600 users and features a catalog of 9,000 movies.

Downloading the MovieLens database yields four different tables:

- links.csv, which also contains IMDb information;
- movies.csv, containing movie data;
- ratings.csv, containing ratings;
- tags.csv, which includes some tags for the movies such as the presence of a famous actor or topic.

The "links" and "tags" datasets were not utilized. Although the "tags" dataset could potentially improve the accuracy of recommendation systems, I chose to focus on creating a model based solely on user reviews.

The "movies" dataset contains 3 columns:

- movieId: an integer representing the movie. It's worth noting that the numbers do not follow a continuous order;
- title: a string containing the movie title;
- genres: a string with the movie's genres. This column was not used for the model creation for the same reason as the "tags" dataset.

The "ratings" dataset contains 4 columns:

- userId: an integer uniquely identifying a user in the dataset. UserIDs are sequential and range from 1 to 610;
- movieId: the movie identifier identical to that in the "movies" dataset;
- rating: the evaluation assigned by a user to a movie, ranging from 0.5 to 5;
- timestamp: an integer uniquely representing the review, unnecessary for the recommendation model I intended to create.

4.2 Hyperparameter tuning

The main problem encountered was overfitting. Even relatively simple models suffered from significant overfitting, so much of the work was focused on tuning hyperparameters. During the testing phase, the model with basic hyperparameter values proved to be extremely inefficient. Experimental results showed a clear correlation between the presence of overfitting and the output dimension of the embeddings, with 5 being the best value, as well as a high number of Dense layers.

Another significant improvement was achieved by adjusting the parameters of the fit function, which handles model training. The first parameter modified was epochs, which determines the number of training iterations. A higher number of epochs implies a more thorough training that could potentially reduce both losses (training and test), provided that the model doesn't start suffering from overfitting. After various attempts, 10 was experimentally found to be the best value for decreasing the test loss and simultaneously reducing the difference between the two losses.

A similar discussion applies to the batch_size parameter, which represents the number of samples used in a single iteration during the training process. The experimentally best value was found to be 8 after testing it with values of 64, 32, 16, 8, and 4. However, the improvement in the test loss value was very modest.

Further slight improvement was introduced by simultaneously reducing both the test loss and the difference between the losses. This result was achieved by adding Dropout layers. Dropout layers work by randomly deactivating some units (neurons) within the layer during each training pass and were inserted between the Dense layers.

4.3 Results

Let's take the example of a classification model that determines whether an image contains a dog or a cat. This model is trained on images with precise, defined, and certain labels. In contrast, a recommendation model is trained using user ratings, which are subjective and changeable. A single user can express completely different opinions about the same movie, depending on when they rate it (a rating immediately after watching may differ from one expressed a day or a month later). Additionally, opinions can vary depending on mood, emotions, or whether it's the first or second time the movie is watched. Consequently, a rating does not uniquely represent an individual's opinion on a particular movie, unlike an image where it's clearly identifiable if it contains a dog or a cat. Moreover, it's important to consider the rating scale used in reviews. Some people might rate a movie negatively if it receives a 3-star rating out of 5, while others might consider it negative only with ratings of 1 or 2 stars. Recommendation systems cannot consider all these aspects; they attempt to predict reviews through algorithms and mathematical calculations, which, although advanced, have limitations. Even the most accurate system will never be error-free because it cannot take into account all the factors listed above. The number of reviews also plays a crucial role. A significant amount of data is needed to create an accurate model, and the one created uses a database of approximately 100,000 reviews. Although this is a significant number, it's not enough to create a truly accurate collaborative filtering-based system. Another crucial aspect affecting the quality of results is that the created recommendation system does not leverage additional movie information such as the year of release, genre, director, or actors. If the system used such information along with reviews, a more sophisticated model could be developed to understand hidden relationships between users and, consequently, become more specific and efficient in its recommendation function.

The created model has many limitations and many opportunities for improvement, such as creating a hybrid system or addressing some of the limitations listed above (e.g., choosing a larger database or considering features other than just numerical ratings). However, despite this, the results obtained are truly satisfying. The Mean Squared Error is approximately 0.74, and the Mean Absolute Error is even lower at around 0.65, indicating that the system can determine whether a movie might be liked by a user. Furthermore, comparing the performance with that of the Matrix Factorization model, we notice that the deep learning model is more accurate. The traditional model has an MSE of approximately 0.98, significantly higher than the 0.74 of the first model.

We can therefore conclude that the deep learning model not only outperforms the traditional algorithm but also has many opportunities for improvement.

5 Supplementary Material

References

5.1 Papers

[1] Roy, D., Dutta, M. A systematic review and research perspective on recommender systems. J Big Data 9, 59 (2022). https://doi.org/10.1186/s40537-022-00592-5

[2] Shuai Zhang, Lina Yao, Aixin Sun, Yi Tay. Deep Learning based Recommender System: A Survey and New Perspectives. 24 Jul 2017. https://doi.org/10.48550/arXiv.1707.07435

[3] Y. Koren, R. Bell and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," in Computer, vol. 42, no. 8, pp. 30-37, Aug. 2009, doi: 10.1109/MC.2009.263

5.2 Websites

[w1] sklearn.preprocessing.LabelEncoder. url: https://scikit-learn.org/stable/modules/ generated/sklearn.preprocessing.LabelEncoder.html.

[w2] sklearn.model_selection.train_test_split. url: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[w3] tf.keras.layers.Embedding. url: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding

[w4] MovieLens. url: https://grouplens.org/datasets/movielens/