FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Recommendation using image data enriched by interaction data
Student:	Bc. Kamil Kader Agha
Supervisor:	Ing. Petr Kasalický
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

Explore the topic of Recommendation Systems with a focus on a content-based recommendation. Examine state-of-the-art methods for feature extraction from images, especially for the area of Recommendation Systems. Propose a method to incorporate interaction data into image data to improve the recommendation of never interacted items. Design a Recommendation System based on images and implement its prototype with your choice of tools. Select suitable non-trivial input data and perform an experiment to measure its predictive success (e.g., recall). Evaluate results and discuss possible future improvements of the suggested approach.

Electronically approved by Ing. Karel Klouda, Ph.D. on 31 January 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Recommendation using image data enriched by interaction data

Bc. Kamil Kader Agha

Department of Applied Mathematics
Supervisor: Ing. Petr Kasalický

May 4, 2022

Acknowledgements

I would like to thank my supervisor Ing. Petr Kasalický, for his professional guidance. Whenever I needed him, he was available to help me and willingly answered all my questions. Moreover, he spends several more hours correcting my work and discussing it. I would also like to thank the company Recombee s.r.o. for providing me the opportunity for cooperation, datasets, and equipment that was available to me during my research.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 4, 2022 . . . .

Citation of this thesis

Kader Agha, Kamil. *Recommendation using image data enriched by interaction data.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Cílem této diplomové práce je popsat doporučovací systémy založené na obsahu na základě obrazových dat, nejmodernější konvoluční neuronové sítě pro extrakci příznaků z obrázků a možnosti vytváření doporučení na základě více než jednoho obrázku jedné položky, tzv. multiple instance learning. Následně jsme navrhli prototyp doporučovacího systému založeném na obsahu položek, který využívá jeden a více obrázků jedné položky a začleňuje i interakce z chování uživatelů pro zlepšení doporučení. Dále jsme implementovali a porovnali několik prototypů doporučovacích systémů založených na konvolučních neuronových sítí a jejich schopnost extrahovat příznaky, které jsou důležité pro doporučení při cold-start problému. Nejlepší modely jsme poté trénovali na uživatelských interakcích. Také jsme implementovali dva modely multiple instance learning a porovnali všechny navržené modely v offline testech. Závěrem jsme čtyři modely porovnali v online A/B testu proti sobě. Výsledky ukázaly, že začlenění uživatelských interakcí a více obrázků do doporučovacího systému zlepšilo měřenou metriku míru prokliku.

Klíčová slova Konvoluční neuronové sítě, multiple instance learning, extrakce příznaků, content-based doporučovací systém, zpracování obrazu, maticová faktorizace, interakční embedding

Abstract

This master thesis aims to survey content-based recommendation systems based on image data, state-of-the-art convolutional neural networks for feature extraction from images, and possibilities of producing recommendations based on more than one image of a single item, so-called multiple instance learning. Subsequently, new content-based recommendation methods that use one and more images of a single item and incorporate interactions from users behavior to improve the recommendations were described. Several prototypes were implemented based on the state-of-the-art convolutional neural networks, and compared in offline tests in their ability to extract important features for recommendations on cold start problem. Finally, four of the models were compared in the online A/B test against each other. The results showed that the incorporation of user interactions and more images into the recommender system improved the measured click-through rate metric.

Keywords Convolutional neural networks, multiple instance learning, feature extraction, content-based recommendation, image processing, matrix factorization, interaction embedding

Contents

List of Figures

List of Tables

Introduction

Motivation

Nowadays, we spend more and more time on the Internet looking for an answer to our questions, for our favorite music, movie, and last but not least, our ideal product to buy of any kind. In this search, we can encounter thousands of results that we have to choose from, and the more options we have, the more time-consuming and more challenging it becomes to decide, as the choice paradox implies[1]. Selection can be so overwhelming that we totally give up on the search and perhaps never find the desired goal.

Therefore, there is a need for some mechanism to help us sort, filter, and find the perfect item for us, save our time, and minimize the overload of choices we experience. A recommender system can solve this task by comparing all available items and providing personalized user recommendations where only a few of the most relevant items are presented, which are hopefully what the user was looking for.

This work focuses on solving this problem for a specific use case of product recommendation based on image data. We will design and build a recommender system solution for an e-shop selling furniture based purely on the picture of the product, later enhanced with interaction data collected on the website. The recommender system will use machine learning and state-of-the-art image recognition and classification models. In the first part, we will use only one image of each product, and then we will try to increase the probability of successful recommendations by taking into account more than one image.

Such a system can save the time of customers searching for the ideal furniture, increase e-shop sales, and help keep their customers as they will be less

[1]The choice paradox can also be called choice overload or simply over choice. When there are many options available, it prevents us from making any decision resulting in no action.

likely to leave the e-shop and search elsewhere. This can bring a significant concurrency advantage to the e-shop, together with increased profits.

Outline

This work is organized as follows. First and foremost, in the two first chapters, we will describe the basics of recommender systems based on images of items and a brief introduction to artificial neural networks and their state-of-the-art image recognition models. Furthermore, we will research the problem of multiple instance learning.

In Chapter 3, we will propose a recommender system based on one and more images, where we incorporate interaction data to improve the recommendation of never-interacted items.

Next, in Chapter 4, we will implement the proposed solution. After that, in Chapter 5, recommender system on the real-world offline and online data will be evaluated, and the results discussed.

Finally, we will summarize the findings of this work in the last chapter and outline possible future work.

Chapter 1

Recommender systems

This chapter will introduce *recommender systems*, their different types and overall recommendations processes. Finally, we will describe how to evaluate our recommender system.

1.1 Recommender system and its usage

Recommender systems (RS) are information filtering tools that personalize the information that comes to a user based on his interests, behavior, relevance of the information to him, and the like. They are widely used to recommend movies, articles, places to visit, items to buy, etc. [16], [17].

Everyday Internet users can meet recommender systems quite often. According to [18] and [19] they are used mainly in:

- **E-commerce** - Internet shops want to sell as many products as possible, and the recommender system is one way to achieve this. E-Shops can customize their front page for each customer[2] with a purchasing history that can meet the needs of the user. It can make further recommendations on the product detail page, guessing what users may need for the specific item.
- **Media** - When we consume multimedia (movies, music, news, and others), we get recommendations on other content (another movie, article) according to our history.
- **Social network** - Social networks can propose to the user whom to follow, employ, subscribe to, etc.
- **Advertising** - We can recommend specific products in an e-shop and advertise an e-shop to a specific group of people who may be interested

[2]Unlike regular physical shops where usually the best seller is promoted or products with high discount.

in some of our products according to their behavior on the Internet. This approach is usually more efficient than advertising to all possible users.

We will call the recommendations objects (product, music, etc.) **items**. Furthermore, simply by the term **users** will mean all people who interact with *items* (they can buy, view, review items, and more) and to whom we want to make recommendations. The recommender system is then a collection of algorithms and tools that produce recommendations.

> "Formally, a recommender system then deals with a set of users U = $\{u_1, u_2, ..., u_n\}$ and a set of items I = $\{i_1, i_2, ..., i_m\}$. For each pair (u_i, i_j)[3], a recommender can compute a score $r_{i,j}$ that measures the expected interest of user u_i in item i_j (or the expected utility of object o_j for user u_i)" [20].

1.2 Recommender systems types

In general, there are two[4] [19], [21] different strategies for deriving a recommendation for users.

Each type of recommendation strategy expects different input data on which to make the recommendation. These input data consist of different sources of information about users and items. We can collect their ratings of items, visited pages, history of purchase, likes, time spent on specific pages, viewed content, mouse movement, and many more. On items, we already have some information that can be physical dimensions, duration of video or music, color, picture of the product, etc. [19].

The two main strategies are *Content-based* and *Collaborative filtering* recommender systems.

- In **Content-based** (CB), we recommend items based on a similarity between all items and items liked by a user based on item attributes, such as image, category, dimensions, color, description, and so forth, to make recommendations. We assume that users like items with similar attributes. Therefore, we focus only on item similarity [18].

  An example can be that the user purchased a chair in the Art Nouveau style; thus, we can recommend a table in the same style and not in the Renaissance style.

- **Collaborative filtering** (CF) takes a different approach and tries to predict the rating or preference that a user would give to an item by

[3] Authors note: u_i means i-th user and i_j means j-th item.

[4] In some other works, they refer to 4 or 3 types. We will discuss all of them later in this work.

matching his ratings with users with similar behavior [18], [22]. Two ways to do so are *item-based* and *user-based.* CF does not require item attributes, as an opposite to content-based recommenders.

For an example of the item-based system, if a user is looking for a night stand, there will probably be many users who bought a lamp with it; thus, despite having few or none of the same attributes (between lamp and night stand), it will recommend it to the user.

There can also be different systems such as *Knowledge-based*, *Demographics-based*, and a combination of all, referred to as a *Hybrid system* [23].

- **Knowledge-based** systems require active user interaction with the system in the form of a dialog and knowledge of the items [24]. It does not need to collect user information because its decisions are independent of individual tastes. Therefore, they work better at the beginning of their deployment, but if knowledge-based systems do not improve with time, they may become less relevant than other methods (such as CF) [18].

- **Demographics-based** systems distinguish different demographic niches and generate different recommendations among them. It can differentiate users by age, gender, profession, location, etc. Each group will combine the ratings of other users in the same group [18], [23].

- **Hybrid systems** combine all or some of the approaches discussed above. A hybrid system aims to use all available information about the users and the items and overcome possible disadvantages of a single system (namely, the cold start problem). The combination usually produces higher accuracy than the standalone system. [23] The outcome can be produced as a combination of results from a single system, or more systems can be combined into one algorithm internally with a single result [19].

1.3 Collaborative filtering recommendation

"System based on existing rating data and computing users (or items) similarity, according to the similarity search for the nearest neighbors of the target users (or items), thus according to the nearest neighbor prediction score to generate recommendation, which most commonly used were user-based collaborative filtering and item-based collaborative filtering" [25].

In **collaborative filtering** (CF) recommender systems, we need a history of user interactions with the items to produce a recommendation, so-called *user feedback.*

1.3.1 User feedback

In general, we have two types of user interactions: [26]

- **Explicit Feedback** - Users specify their opinion by direct actions, such as thumb-up or down, rating an item with stars, or giving some percentage (where higher is better). It can be positive or negative feedback.
- **Implicit Feedback** - The user's opinion is reflected and recorded indirectly by observing the user's actions and behavior, such as page views, search patterns, purchase history, clicks (even mouse movements), etc. They are easy to collect, but negative feedback is absent[5] [21].

We store user feedback in the so-called *Rating matrix.*

1.3.2 Rating matrix

Storing all possible interactions between all users and all items eventually leads to creating a matrix, where the rows identify the users $(user_1, user_2, ..., user_n)$ and the columns identify the items $(item_1, item_2, ..., item_m)$ as shown in Equation 1.1. This matrix is called a **Rating matrix** or simply R, sometimes referred to as a *User-item interaction matrix*, where $R \in \mathbb{R}^{n,m}$. When $user_1$ interacts with $item_2$ we will note a specific value r_{12} (usually between 0 and 1) in the corresponding column and row representing the rating given by $user_1$ to $item_2$. This matrix is usually very sparse, as users do not interact with all available items [19].

$$R = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ r_{21} & r_{22} & \cdots & r_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \cdots & r_{nm} \end{pmatrix} \tag{1.1}$$

1.3.3 Recommendation approaches

With collected user feedback in the Rating matrix, CF analyzes the relationships between other users and searches for interdependencies between products to identify new user-item associations [21].

Identifying associations can be done in two ways: [27]

- **Item-based** analyzes the relationships between the items. It assumes that if several users rated two items similarly, the other users would probably rate them likewise.

[5]We cannot say if the user dislikes an item he has not interacted with or simply the user did not know the item existed. On the other hand, buying a product as a gift may not mean that the user likes the product

- **User-based** relies on past preferences of users similar to a target user. The most similar users are found to a specific user and give recommendations based on what they liked the most. The more similar the other user opinion is, the more impact it will have on the final recommendation.

Similar users are users who are interacting with many similar items. In Rating matrix, we compare vectors in different rows. Moreover, similar items are items that are interacted with similar users; in the Rating matrix, we compare vectors in different columns. Since users and items are represented as vectors of n dimensions and m dimensions, similarity can be measured with standard methods such as *Cosine Similarity*[6], *Adjusted Cosine Similarity*, or *Pearson's Correlation Coefficient* [27].

1.3.4 Limitations

There are several limitations to the CF method: [28]

- **Noise** - it is hard to separate more users under one account.
- **Possibility of attacks** - some sellers can create many fake profiles prising their products.
- **Popularity bias** - popular items tend to be suggested, more over *long-tail* items[7]. This results in the recommendation of only a few of the most popular items over and over [29].
- **Cold start problem** - when new users or products are added to the system, we do not have any interaction with them. Therefore, we cannot find any similar users or items and make a recommendation. [30]

1.4 Content-based recommendation

"The system learns to recommend items that are similar to the ones that the user liked in the past. The similarity of items is calculated based on the features associated with the compared items" [18].

In **content-based** (CB) systems, we focus on item attributes apart from user behavior. It recommends items similar to previously rated items from an attribute point of view [17], [27]. The attributes of an item can be anything from the genre, length, color, image of a product[8], type, dimensions, and other metadata.

The advantages of content-based recommendations are: [18]

[6]We will present this method later in this work.
[7]Long-tail items do not have a lot of interactions(ratings/likes).
[8]In this work, we will focus on an image of the product.

- **User independence** - it is based on the ratings of a single user, but is not affected by other users (in contrast to CF).

- **Newly added items** - can be easily included in recommendations since it needs only item attributes (the CB recommender does not suffer from *Cold start* as CF).

- **Transparency** - it can explain how the recommendation was made by explicitly listing content features or descriptions that were important in the decision (CF are black boxes, where only users can explain their behavior).

However, there are also several limitations, mainly: [18]

- **Limited content analysis** - content may not be easily extractable (multimedia), or the attributes may not be distinguishable enough. Furthermore, captured attributes may not cover all aspects of the item, lacking some information (for example, the design of a website that may seduce the user into action more effortlessly than displeasing looking website, but items do not contain this information).

- **Overspecialization** - recommends similar items with a slight chance of recommending something novel[9] [17].

- **New user** - for a new user, the recommender system does not know user preferences and, therefore, what items to recommend. Some ratings or user information have to be collected to produce better recommendations.

1.5 Recommendation

In the previous section, we described different approaches and types of recommendation systems. This section will focus on the CF and the CB recommendations, where we need to measure the distance and similarities between users or items. Therefore, we need a vector representation of either users or items to measure similarities that the target users may like; we will call these vectors *embeddings*. After we have embeddings, the recommendation process becomes the same for both methods. We will discuss now where we can take embeddings in both methods and describe how the actual recommendation is made.

[9]This is also called the serendipity problem.

1.5.1 Processing of rating matrix

In CF, we already have vectors (in Rating matrix), and we can start making predictions.

However, Rating matrix is usually processed before being used in training a recommendation algorithm. There can be many misinterpretations when we collect user feedback, especially with implicit feedback. It can be challenging to guess user preferences and true motives as only positive feedback exists. For example, watching a video does not suggest user likes the video or forgets to switch the video off, or when the user does not buy some product, we cannot say he does not like the product or the user does not have enough money to buy it or does not even know such a product exists. Explicit feedback also has its ambiguities, such as shared accounts with many users, where more people rate different products and have different interests, etc. [21].

We should already think of these domain-specific cases when preparing the rating matrix. For example, buying the product is a more meaningful interaction for an e-shop than viewing only a product page or first played show is stronger interaction than fifth on autoplay mode.

Another thing is that Rating matrix is usually very sparse, as there can be thousands of items available, and even the active user does not visit most of them [27].

Here the *matrix factorization* comes into place. We can extract features from the rating matrix and improve our predictions [19]. The feature extraction will help us,

- reduce noise in data,
- reduce dimensionality and size of the matrix, [31], [21]
- find similar users or items, and find patterns in the whole matrix [32].

Subsequently, we can apply different feature extraction methods to produce two and more matrices where one is from $\mathbb{R}^{n,x}$ and the second is from $\mathbb{R}^{x,m}$, where x can be common for both matrices or equal to n and m, respectively, depending on the method of our choice. This factorization helps us with the problems mentioned earlier.

Matrix factorization

"Matrix factorization models map both users and items to a joint latent factor space of dimensionality f, such that user-item interactions are modeled as inner products in that space" [32].

We can associate each item i and user u with vectors, $q_i \in \mathbb{R}^f$ and $p_u \in \mathbb{R}^f$. When we do a dot product of these two vectors, we get the approximation of user u rating of item i as [32]

$$r_{ui} = q_i^T p_u.$$

The problem is to map each user-item rating into two matrices. This can be done with methods such as *Alternating Least Square* (ALS), *Singular Value Decomposition* (SVD), *Principal Component Analysis* (PCA), *Probabilistic Matrix Factorization* (PMF), and *Non-negative Matrix Factorization* (NMF) [31], [21].

We eventually want to minimize function

$$min_{q*,p*} \sum_{r_{ui} \in k} (r_{ui} - q_i^T p_u)^2 + \lambda(||q_i||^2 + ||p_u||^2) \tag{1.2}$$

[32], where k is known ratings, and $\lambda \in \mathbb{R}$ controls regularization.

Alternating Least Square (ALS)

ALS produces two matrices

$$R \approx X^T \times Y,$$

where $X \in \mathbb{R}^{m,n}$ and $Y \in \mathbb{R}^{n,m}$[10]. We produce these matrices by keeping one fixed while calculating (minimizing function 1.2) the other by solving a least squares problem and vice versa (fixing the other matrix and minimizing the first matrix) in several iterations as shown in code 1.

This factorization into two separate matrices better reflects the nature of the data and improves prediction accuracy with a more holistic approach that uncovers latent features, which explains the observed ratings better.

ALS is used to predict implicit data in collaborative filtering [32] [21].

Algorithm 1 The ALS algorithm [33]

.

```
Require: (R, k, λ; X, Y )
Ensure: X ← 0, Y ← random initial guess
1: while (reached max iterations) do
2:     for row u ← 1,m do
3:         $x_u \leftarrow (Y^T Y + \lambda I)^{-1} Y^T r_u$
4:     end for
5:     for column i ← 1,n do
6:         $y_i \leftarrow (X^T X + \lambda I)^{-1} X^T r_i$
7:     end for
8: end while
```

[10]X matrix is also called user-factor, and Y is called item-factor.

Singular Value Decomposition (SVD)

SVD creates three matrices that produce the original matrix

$$SVD(R) = U \times S \times V^T,$$

where $U \in \mathbb{R}^{m,r}$, $S \in \mathbb{R}^{r,r}$ and $V \in \mathbb{R}^{r,n}$. The matrices U and V are orthogonal, S is a diagonal matrix called the singular matrix[11] [31].

The final decomposition is a linear combination of vectors, and we can add or subtract vectors as needed. SVD can then provide the best low-ranking linear approximation of the original R matrix [34].

SVD is used to predict explicit data in collaborative filtering [35].

1.5.2 Processing of item attributes

As stated in Section 1.4 about CB, items can have various attributes and representations, and we need to measure the distance between two items. Therefore, we need to transfer these attributes into a vector space, where we can use a standard evaluation method such as *cosine similarity* [19].

Numerical features can be connected to a vector. We can use methods such as *one-hot-encoding* for categorical features. However, creating vector space from texts and multimedia usually requires unique methods.

For text attributes, we can use the method *bag of words* followed by *Term frequency and inverse document frequency* (TF-IDF) [36] a method that produces a table of words scores within a given text, where each score represents the relevancy of the word for the document. In other words, we produce embedding based on words that the given text contains for each document [37]. Alternatively, we use a more advanced model like *Bidirectional Encoder Representations from Transformers* (BERT) or other transformer[12] models [38].

With image attributes, we can generate vector embeddings, for example, with *image histogram*, *ORB*, and *artificial neural networks* which is examined in [39].

1.5.3 Producing a recommendation

When we have vectors representing given items or users processed as *ALS*, *SVD*, or attributes encoded in *embeddings*, we can easily differentiate items that are the most similar (and probably the most interesting) to some given item and suggest them to the target user. We can do it from *content-based* RS, where we recommend similar items, or in *collaborative filtering* RS, where we look at ratings given to similar items, as *item-based* recommendation, or

[11]Therefore, Singular Value Decomposition.

[12]A transformer is a deep learning model.

if we look for similar users, we recommend items that the most similar users liked the most, as *user-based* recommendation [40], [27].

This can be done with neighborhood-based methods where we find K the most similar items (or users), where K represents the number of items (or users) we want to recommend. Neighborhood methods can be *K-Nearest neighbors (KNN)*, *k-Means*, *k-d Trees*, and *Locality Sensitive Hashing* [41].

A KNN method is based on finding the top K nearest neighbors to an item or user and recommending the closest/most similar ones[13] [42].

We can do this with different similarity metrics between two vectors like *Pearson correlation*, *Mean Squared differences* (MSE), *Jaccard Similarity*, *Cosine similarity* (COS), and others.

MSE metric measures the average of the squares of the errors, i.e., the average squared difference between the estimated values and the actual value [41], [43].

1.5.4 Cosine similarity

Cosine similarity measures the similarity between two vectors of an inner product space irrespective of its size. It measures as a cosine of the angle between the two vectors as projected in a multi-dimensional space and determines whether two vectors are pointing in roughly the same direction. The smaller the angle, the higher the cosine similarity.

“Let x and y be two vectors for comparison. Using the cosine similarity as a similarity function, we compute it as

$$cos(x, y) = \frac{x * y}{||x|| * ||y||},$$

where $||x||$ is the Euclidean norm of vector $x = (x_1, x_2, ..., x_p)$, defined as $x = \sqrt{x_1^2 + x_2^2 + ... + x_p^2}$. Conceptually, it is the length of the vector. Similarly, $||y||$ is the Euclidean norm of vector y” [44].

The result is $cos(x, y) \in [-1, 1]$. A cosine value of -1 means that the two vectors are opposite, 1 means that the vectors are the same, and 0 indicates orthogonality (90 degrees to each other).

We can translate cosine similarity into cosine distance by subtracting cosine similarity from 1. Then 0 will indicate two precisely the same vectors (in contrast to 1 in cosine similarity) [45].

After we compute the similarity between all items (each one to all of the others), we can make a final step in a recommendation - prediction to a particular item by taking the first K items that are the most similar and showing them to the target user as an ordered list of these items. Alternatively, find the closes users and recommend items they liked the most [46].

[13] In CF, we can measure the similarity of two items if they have been both rated by the same users.

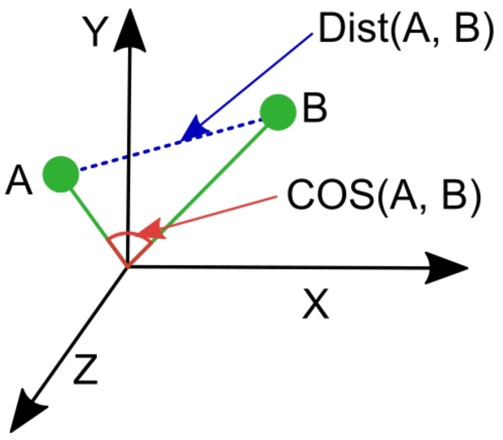

Figure 1.1: The difference between Euclidean distance and cosine similarity. Inspired by [1].

1.6 Evaluation of recommendations

When we build a recommender system, and we can provide recommendations, it is crucial to measure the accuracy of these recommendations. We have to set goals depending on what we want to achieve. That can be any interaction - user visit product detail page, purchase a product, review it, etc.

We can use *recall*, *precision* or *F1 measure* performance metrics to calculate accuracy using *confusion matrix*. The higher the resulting value, the more successful the recommendations are [27].

For this purpose we can use the *Leave-one-out cross-validation* method.

1.6.1 Recall

In the classification task, **recall**[14] refers to the percentage of relevant items correctly classified by the recommender system to a total number of relevant items available [47], [48].

$$recall = \frac{\text{\# of recommendations that are relevant}}{\text{\# of all the possible relevant items}}.$$

It would be trivial to achieve a recall of 100% by recommending all the items, but this recommendation would not be of any value to the user. Therefore, we measure the recall only on *top-K*[15] most relevant items. Recall at K or **Recall@k** is the proportion of relevant items found in the *top-k* (closest) recommendations [49].

Recall@k is defined as

[14]Recall can also be referred to as the true positive rate or sensitivity(in binary classification).

[15]K usually equals 1, 5, 10, and 25.

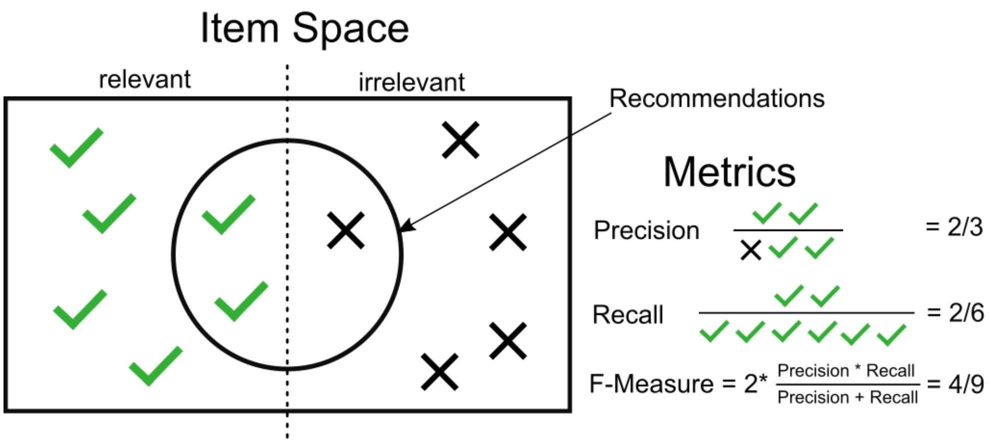

Figure 1.2: Illustration of recall, precision, and F1 score. Inspired by [2].

$$recall@k = \frac{\text{\# of recommended items @k that are relevant}}{\text{total \# of relevant items}}.$$

1.6.2 Precision

Precision[16] in recommender systems represents the percentage of relevant recommended items to the number of items selected (again, we speak about top-K items). Precision measures the recommender's ability to deny any nonrelevant items in the retrieved set [47], [48].

In our case, precision in the top-k recommendations is defined according to [50] as

$$precision@k = \frac{\text{\# of recommended items @k that are relevant}}{\text{total \# of recommended items @k}}.$$

1.7 Leave-one-out cross-validation

"Leave-one-out cross-validation is a special case of cross-validation where the number of folds equals the number of instances in the data set. Thus, the learning algorithm is applied once for each instance, using all other instances as a training set and using the selected instance as a single-item test set" [51].

Leave-one-out cross-validation (LOOCV) takes on input user-item matrix, where each user has interacted with at least two different items. Then 'hides' one item, and finds out if the recommender system would recommend this hidden item based on other visited items of the same user. In other words,

[16]Precision is also referred to as positive predictive value or confidence.

it will look at the nearest neighbors of the visible items, combines theirs results, and if they would recommend the hidden item in the top K positions, recommendation was successful and we can increase the recall value. This process repeats for all items, always leaving one item, therefore leave-one-out.

1.8 Summary of recommender system

We have described the possible workflow of the recommender system based on two approaches *Content-based* and *Collaborative filtering*. From data collection to the recommendation process and evaluation of our recommendations.

Chapter 2

Artifical Neural Networks

This chapter will introduce artificial neural networks and related topics, including architecture and basic functionality, learning, and optimization of a network focusing on convolutional neural networks. We will continue with a description of several state-of-the-art models, which are used in this work. At the end of this chapter, we will expand on multi-instance learning, which can be used to aggregate multiple features into one.

2.1 Introduction

Artificial Neural Networks (ANN) are computational models inspired by biological neural networks whose capabilities they attempt to mimic.

Compared to modern computers, the human brain can solve many complex real-world tasks that are still difficult or impossible to achieve with hard-coded algorithms [52]. Moreover, the human brain can learn and solve new, never seen before tasks or similar tasks he already acquired. Therefore, we try to mimic these abilities to learn, transfer our knowledge and solve complex tasks in computer science, where artificial neural networks can adapt to several different situations and provide a solution.

ANN has a rich history starting in 1958 with the introduction of an artificial neuron called perceptron [53] and deep neural networks with many layers and the ability to learn. Since then, many new concepts, models, and architectures have been introduced, such as *convolutional neural networks* (CNNs), *autoencoder* (AE), *deep belief networks* (DBNs), *restricted Boltzmann machines* (RBMs) [54].

Each of them focuses on different tasks, namely [52]

- **pattern classification** - assigning a class to an unknown pattern (i.e., name of an animal in image),
- **clustering** - discovering similarities or dissimilarities in a dataset and assigning them class,

- **function approximation (modeling)** - training ANN to fit input on output where we approximate some underlying function. (i.e., prediction of microbial growth),
- **forecasting** - prediction of time series (i.e. price of some asset in the future),
- **optimization** - solving nonlinear optimization problems (i.e., Knapsack problem[17]),
- **association** - develops a pattern (mask) that can be applied to noise-corrupted data (i.e., reconstruction of old photo),
- **control** - learn to control adaptive system (i.e., autonomous driving).

In this work, we will focus mainly on networks that are designed to deal with image recognition (pattern classification), which means *convolutional neural networks.*

2.2 Perceptron

Firstly, we will introduce the perceptron as it shows the main idea behind of ANNs. The perceptron can represents one neuron (node) in ANN and can learn and process simple linear problems [56]. It has inputs $x_1, x_2, ...x_n$, (i.e. sensor or dataset input) and each input has his corresponding weight $(w_1, w_2, ...w_n)$. There is also added one special input $x_0 = 1$ called *bias* with its weight w_0. Bias is a threshold the perceptron must reach before the output is produced. All inputs are then multiplied by their respective weights and then summed

$$z = \sum_{j=0}^{n} w_j x_j.$$

After summarization, the activation function $\varphi()$ is applied, which determines the output of the perceptron [57]

$$\hat{y} = \varphi(z).$$

We can use many activation functions according to our problem, but by default, the perceptron uses a simple threshold function [58]

$$\varphi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{Otherwise.} \end{cases}$$

[17]"Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack" [55].

Another widely used activation function is *sigmoid function*

$$S(z) = \frac{1}{1+e^{-z}},$$

where $S(z) \in [0,1]$, [59] and *Rectified Linear Unit* (ReLu)

$$\varphi(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0. \end{cases}$$

There are also *Leaky ReLU*, *TanH*, *Binary Step*, *Linear*, *SELU*, and others [60]. We can enable perceptron to solve non-linear problems and even speed up the learning process with different functions [61]. A suitable activation function has the first derivation for all points of its definition domain. However, in specific situations, partially differentiable activation functions are used [62].

2.3 Learning of perceptron

We will describe *supervised learning* of perceptron with a set of known inputs and known target output values that should be mapped by weights and the linear activation function as described in [63].

Learning is a process of minimizing the error, also known as *loss function*[18], between the output value from the perceptron and the target output value. This is achieved by changing the weights of the perceptron, which is usually determined by the descent gradient of the error. After the learning process is completed and successful, we should get the required output for the given input.

First, we define the error E between the output and target output values as a summation of quadratic differences.

$$E(w) = \frac{1}{2}\sum_{i=0}^{m}(y_i - \hat{y}_i)^2,$$

where m is a total number of train samples, y_i is a target output value, and $\hat{y}_i \in \mathbb{R}$ is the neuron's output.

Then, we calculate the change of weights against the direction of the gradient as

$$\Delta w = -\eta \nabla E(w),$$

where η is the learning rate and we get a partial derivation for every weight as

[18]Also called objective function, cost function (minimization) or fitness function (maximization).

$$\Delta w_j = -\eta \frac{\partial E}{w_j}.$$

The derivation of the error function is as follows

$$\begin{aligned}\frac{\partial E}{\partial w_j} &= \frac{1}{2}\sum_{i=0}^{m} \frac{\partial E}{w_j}(y_i - \hat{y}_i)^2 \\ &= \frac{1}{2}\sum_{i=0}^{m} 2(y_i - \hat{y}_i)\frac{\partial E}{\partial w_j}(y_i - \sum_{j=0}^{n} w_j x_{i,j}) \\ &= \sum_{i=0}^{m}(y_i - \hat{y}_i)(-x_{i,j}).\end{aligned}$$

Finally, the weights are updated as

$$\Delta w = \eta \sum_{i=0}^{m}(y_i - \hat{y}_i)x_{i,j}.$$

Training process is a cycle of 3 steps: feed-forward input, calculate the error, and change weights.

2.4 Deep neural networks

We can create multiple copies of perceptron and stack them vertically, creating one layer or horizontally creating more layers called *hidden layers*, which results in a bipartite graph. This can produce *Multilayer Perceptron Network* (MLP) or, in general *deep neural network*[19] (DNN) [54].

When we use our described perceptron as one node (with a sigmoid activation function to incorporate nonlinearity), we can create a *multilayer perceptron* (MLP) network where all nodes from one layer are connected to all nodes from a previous layer where the output of previous nodes becomes the input of the following layer. There are no connections between neurons in the same layer (this structure is also known as a *fully connected layer* or *dense layer*) as illustrated in image 2.1. The passing of data from one layer to the following layer defines a feedforward network as a finite acyclic graph.

Networks based on these principles contain an input and output layer and hidden layer [62]. Each layer may have any number of neurons, but at least one in the input layer and one in the output layer. Every node has its weights and can have different activation functions.

DNNs have multiple variants with different architectures defined by “the structure of the nodes, the topology of the network, and the learning algorithm used to find the weights of the network” [64]. For example, *convolutional*

[19]MLP is one type of DNN. There are many other variants of DNN networks.

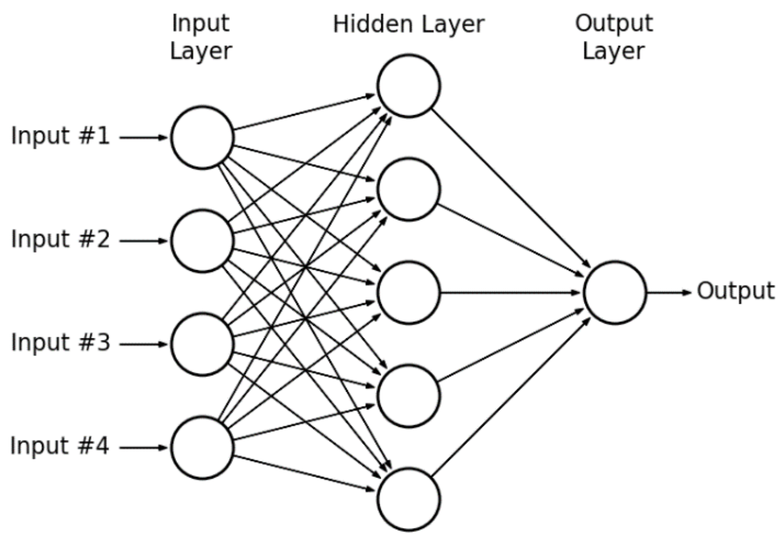

Figure 2.1: Example of Multilayer Perceptron Network [3].

neural networks (CNNs), which are not fully connected between layers, are suitable for image-related tasks, or *recurrent neural networks* (RNN), where the connections can be backward, are suitable for time-series related tasks.

2.5 Learning and optimization algorithm

To begin with, we need to evaluate the error of our model with a *loss function* that compares two vectors. That can be done with *mean squared error* (MSE), *cosine distance*, and others. Our goal is to minimize the loss function. We can adjust the network weights to get better and better results until we achieve a convergence criterion or a local minimum.

With the weight update described in the learning of one perceptron, we can teach only one neuron, but we need a different method to teach hidden layers of DNN. This method is called **backpropagation** which can propagate the error backward as the gradient of the loss function to previous layers and attribute the error associated with each parameter and achieve the whole network learning.

In backpropagation, each weight is adjusted by taking a small step in the opposite direction in which the loss function decreases the most in its value i.e., the gradient of the function. The size of each step is controlled with η (learning rate). The whole process is based on the *Gradient Descent algorithm.* Nevertheless, we can use different training approaches such as *Adaptive Moment Estimation* (Adam), *Root Mean Square Propagation* (RMSProp), and not last *Adaptive Gradient Algorithm* (AdaGrad).

One of the first optimizers is gradient descent with weights update [65]

$$w_{t+1} = w_t - \eta \nabla f(w_t),$$

where f is a differentiable loss function. This method updates the weights w for each input individually, which is not very efficient.

In our work, we will use only the **Adam** optimizer [66]. Adam optimizer combines *Adaptive Gradients* (estimates of first and second moments of gradient) and *RMSProp* algorithms. Compared to the gradient descent algorithm, Adam does not use the entire dataset to calculate the actual gradient, but rather a randomly selected data subset to create a stochastic approximation. This makes Adam computationally efficient, and therefore, Adam requires less memory. Adam updates weights as

$$\begin{aligned}
m_t &\leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla w_t \\
v_t &\leftarrow \beta_2 v_{t-1} + (1-\beta_1)(\nabla w_t)^2 \\
\hat{m}_t &\leftarrow \frac{m_t}{(1-\beta_1^t)} \\
\hat{v}_t &\leftarrow \frac{v_t}{(1-\beta_2^t)} \\
w_{t+1} &\leftarrow w_t - \eta \frac{\hat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)},
\end{aligned}$$

where w_t are weights in time t, η is the learning rate, m_t is the exponential average of gradients along w_j, v_t exponential average of the squares of gradients along w_j, and β_1 and β_2 are hyperparameters. Suitable hyperparameters can be $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 0.002$.

Running average of the gradient m_t will ensure that we will not rely only on a current gradient and instead rely on the overall behavior of the gradients over many timestamps as the exponentially moving average. The third and fourth steps, $\hat{m}$ and $\hat{v}$, are bias corrections [67].

2.6 Convolutional Neural Network

Based on DNNs, we can change the architecture and get new types of networks called **Convolutional neural networks** (CNNs) utilized for image recognition, pattern recognition, and computer vision. By applying filters, also called *kernel*, together with *Hadamard product*, they identify patterns within an image in multiple layers of abstraction. Before we describe CNNs, we will first introduce the **ImageNet dataset**.

ImageNet Challenge or "*The ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) evaluates algorithms for object detection and image classification at large scale" [68] provides a dataset called *ImageNet* of more than

1 million images with 1000 human-annotated objects classes and more than 100,000 synonym sets. It was released in 2010 and became the base for training and evaluation of image recognition models. The accuracy of a model evaluated on the ImageNet dataset is measured as the top-1 and top-5 error rate, i.e., if the correct class of input image is not in the top-1 or top-5 output classes of a model, it gets a lower score.

Since 2012 when *AlexNet* (which is a CNN-based model) significantly improved score and won the ImageNet Challenge, convolutional neural networks have become widely used in computer vision.

The following description of CNN is based on [69]. A CNN has on input the whole image usually in format $width \times height \times depth$, where depth represents a number of image channels - 1 can mean grayscale image, 3 can mean RGB-colored image, and so on. The key difference from other networks is in specific layers, which are *not fully connected* but only to a small region of neurons (or image pixels) in the preceding layer, i.e., 2-dimensional input is in front of each neuron.

This architecture helps us to accept larger images on input. Otherwise, we would need significantly more computational power, and overfitting would happen, as well as the network would not be able to generalize enough, which would reduce the performance of the model.

Common CNN comes with three main types of layers. These are *convolutional layers*, *pooling layers*, and *fully-connected layers* or *dense layers* [70].

- **Convolutional layers** - They are not connected to each neuron in the previous layer. Instead, they focus only on a few corresponding neurons (or pixels if it is an input layer) in front of them of fixed size $n \times n$, where $n \in \mathbb{N}^+$ commonly equals 2 to 5. This focus on only a few neurons is called **kernel** or **kernel filter**. Each kernel is applied and can detect different low-level patterns such as diagonal or horizontal lines and others. The following layers form more complex patterns, creating abstract visualization of the original image. Several units of these kernels filters can be stacked in one layer (creating depth) where each has its filter but looks at the same area (i.e., input is identical).

  Instead of moving filters always by 1, we can move filters with more steps, reducing image size, and requiring fewer parameters. This step size is called **stride**.

  With each layer and increased stride, the image shrinks. We can add **padding** to the edges of the image (all 0 would be zero-padding), increasing the original image size, which will prevent the image from shrinking and keeps the information on the border. Padding allows us to have any number of convolutional layers without collapsing the image into a few pixels.

We illustrated the function of filter with stride 1 and padding 0 on 2d binary convolutional operation in Figure 2.2.

The final step is the non-linear activation function ReLU. The convolutional layer keeps local information about individual pixels (local spatial features) [71].

- **Pooling layers** - performs downsampling and reduces the spatial dimension of each feature map [72] in the given input and reduces the input size for further layers. It combines local spatial features with higher-order features. In CNN the most commonly used pooling is **Max-pooling**. Input is $n \times n$ area, and Max-pooling takes the maximum value from the selected area as output to the following layer. The pooling does not affect the number of filters. Other variants can also be *Sum-pooling* (which sums all input values), *Average-pooling* (which takes the average value from input), and others. Usually they have dimension 2×2 and a stride of 2.

- **Dense layers** - are at the end of a model, fully connected between each layer. They take high-level feature maps from previous layers as input and determine an image's final class score.

There is also **Flatten layer** in an implementation, which only changes the dimension after convolutional layers, for example, from the $2 \times 2 \times 4$ convolution layer to the $1 \times 1 \times 16$ flat layer, suitable as input into a dense layer.

Another commonly used layer is **dropout** which removes nodes randomly from each training process. Dropouts protect the network from overfitting and work as a regularization method [73], [70].

The kernel application is a **Hadamard product** ($\odot$) with a summation of its elements. Hadamard product of two matrices produces a new matrix as a dot product of the corresponding elements i, j. The result is a single scalar value c, as illustrated in Equation 2.1 [74].

$$\begin{aligned} \begin{bmatrix} x_{00} & x_{10} \\ x_{01} & x_{11} \end{bmatrix} \odot \begin{bmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \end{bmatrix} = \begin{bmatrix} x_{00} * w_{00} & x_{10} * w_{10} \\ x_{01} * w_{11} & x_{11} * w_{11} \end{bmatrix} = \\ = (x_{00} * w_{00}) + (x_{10} * w_{10}) + (x_{01} * w_{11}) + (x_{11} * w_{11}) = c \end{aligned} \tag{2.1}$$

Dense layers usually take the longest to train as they obtain many parameters concerning the rest of a network [70].

CNN architecture usually repeats *convolutional layers with pooling layers* (for example, repeating the pattern of 2 convolutional and one pooling layer), and at the end, there are dense layers as illustrated in Figure 2.3.

Nowadays, we have many different types and variants of CNN with improving recognition accuracy, namely *VGGNet*, *EfficientNet*, *DenseNet*, and

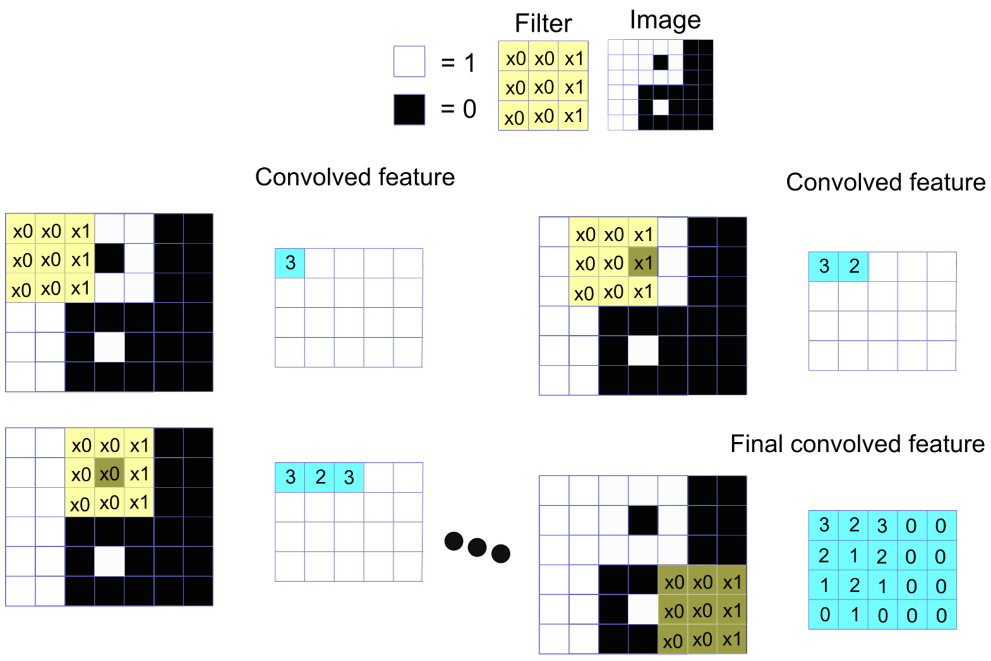

Figure 2.2: Visualization of 2d binary convolutional operation with stride 1 and padding 0 (we can see the resulting image is smaller without padding).

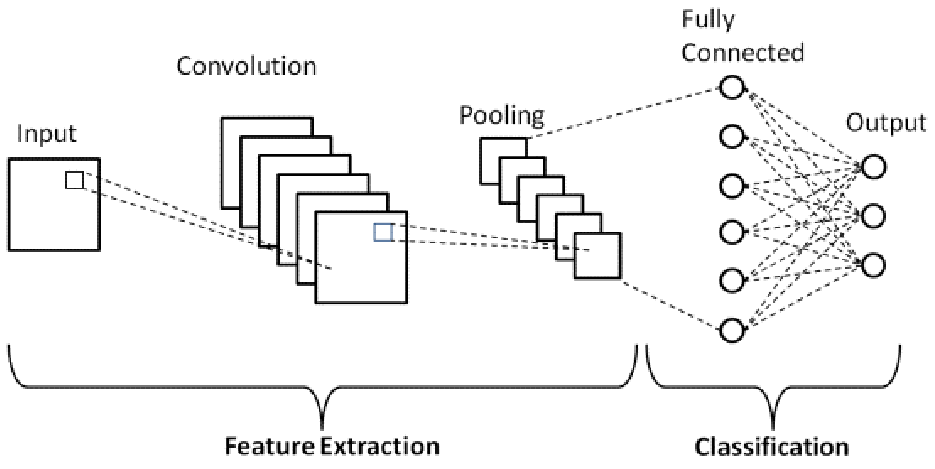

Figure 2.3: Visualisation of CNN architecture [4].

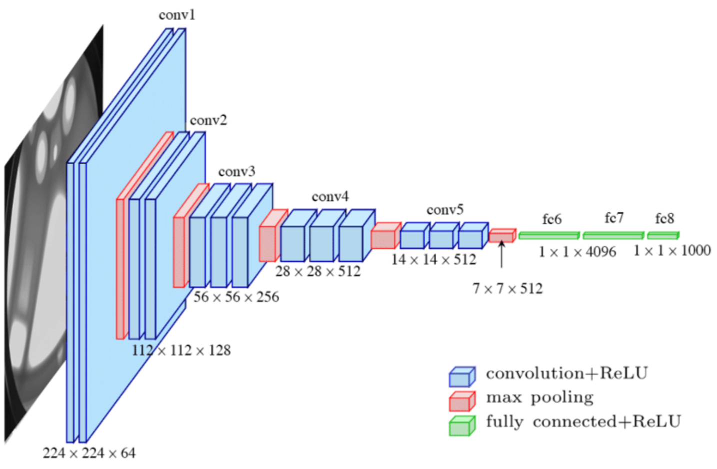

Figure 2.4: The standard VGG16 network architecture [5].

many more. Typically, these networks were trained and evaluated on the ImageNet dataset, and their output layer consists of $1 \times 1 \times n$, where n is the number of object classes. In the case of ImageNet, that is, 1000 neurons.

2.6.1 VGGNet

VGG stands for *Visual Geometry Group*, which published in 2014 several CNN networks with different configurations referred to as **VGGNet** [75]. VGGNet is trained and used for object and image recognition tasks. There are two basic models, called *VGG16* and *VGG19*. The difference is in the number of convolutional layers, where the number at the end of the model name refers to a number of weight layers (16 and 19, where we do not count pooling layers because they do not have any weights).

The input layer is $224 \times 224 \times 3$, which represents the RGB image of 224×224 pixels.

The network architecture follows the pattern of 2 or 3 convolution layers and one pooling layer. These segments repeat several times until the end, where is a flatten layer and 3 dense layers, where the first two have 4096 neurons and the last dense layer with 1000 neurons (channels), giving the final output. The architecture is illustrated in Figure 2.4.

The conv. layer uses 3x3 kernel-sized filters, the minimum required size to identify directions and center in a given segment. The pooling layer is max-pooling with a 2×2 pixel window, and stride 2. The activation function in all hidden layers is ReLu.

The total number of parameters is 138M for VGG16 and 144M for VGG19.

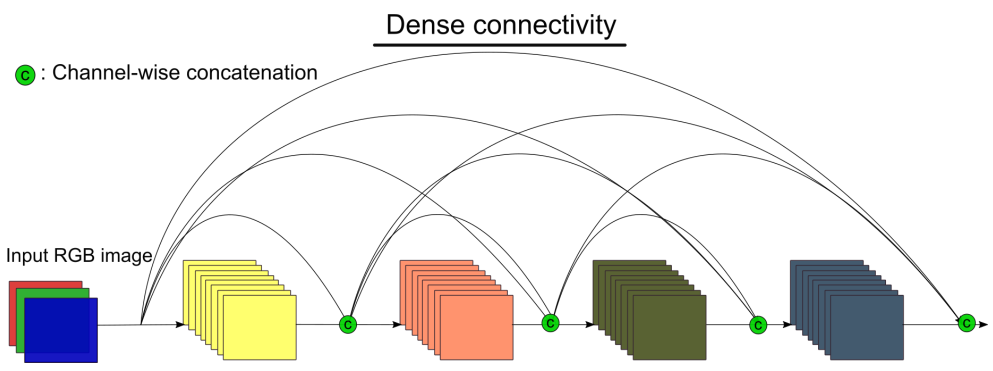

Figure 2.5: DenseNet connectivity with channel-wise concatenation in one block. Image based on [6].

2.6.2 DenseNet

Densely Connected Convolutional Networks, known as **DenseNet**, were published in 2016 [76]. DenseNet is a newer model than VGGNet and was also trained on ImageNet for the image classification task.

DenseNet network architecture tries to improve the problems with deep CNNs, where "information about the input or gradient passes through many layers, can vanish and *wash out* by the time it reaches the end (or beginning) of the network" [76]. This is accomplished by connecting all layers (with corresponding feature map sizes) directly and concatenating their features as illustrated in Figure 2.5. The concatenation is done by the convolution layer, pooling layer, batch normalization, and non-linear activation layer.

The network then has $L(L+1)/2$ connections between layers, where L is the number of layers.

However, this results in fewer parameters than the existing algorithms with comparable accuracy.[20]

There are different versions of the DenseNet, where the number in the name denotes the number of layers. The versions are *DenseNet121*, *DenseNet160*, *DenseNet201*, and *DenseNet264*. Compared to the VGGNet architecture with a maximum of only 19 layers, this is a significant enlargement.

The resulting architecture has from 8.1M (DenseNet121) parameters to 20.2M (DenseNet201) parameters. All DenseNet versions have the same input size as VGGNet - $224 \times 224 \times 3$ [77].

2.6.3 EfficientNet

Research on CNN has already reached the hardware limit for state-of-the-art CNN models, and adding only more layers is not sufficient. Moreover, there

[20]At least in 2016 when this model was released.

is an increasing demand to use CNNs on mobile devices with limited computational power. Therefore, researchers focus on different approaches and minimize hardware requirements and easy scaling of input size. The resulting family of networks is called **EfficientNet**, introduced in 2019 [78]. EfficientNets continues a process of scaling, increasing efficiency, learning speed, input size, and decreasing parameters for the similar result as other networks.

For the reasons mentioned above, a new method was developed called *compound coefficient*. This method scales all dimensions of a network (resolution[21], width, and depth[22]) with a constant ratio of 1.1 (width), 1.2 (depth), and 1.15 (resolution).

The main building block consists of an **inverted bottleneck**, for example, denoted as *MBConv5 k3x3*, where 5 means the expansion factor after the bottleneck and 3×3 is the kernel size of the separable convolution, to which squeeze-and-excitation optimization is added. This is based on a block described in a MobileNetV2 network [79].

"Regular convolution is replaced by separable convolution, which consist of a **depthwise** separable convolution (for example, 3×3) acting on each channel separately (which reduces the time and space complexity of a regular convolution by a factor equal to the number of channels); a **pointwise** 1×1 convolution acting on each position independently (which reduces time and space complexity of a regular convolution by a factor of $3 \cdot 3$). There is no non-linear activation on the bottlenecks (it would lead to loss of information given the small capacity of bottlenecks)" [80, 1:24].

Furthermore, it expands into a high dimension, filtered with a lightweight depthwise convolution, and then it maps back to the low dimension. The high dimension acts as a non-linear transformation. Between these low-dimensional layers (bottlenecks) are residual connections that help propagate input and gradient. This block is called squeeze-and-excitation, "which learns to emphasize information channels and suppress less useful ones according to global information" [80, 1:24].

EfficientNet comes with different variations, which can be seen in Table 2.1. In Figure 2.6 is illustrated architecture of EffecientNetB0.

2.6.4 EfficientNetV2

There were several limitations in EfficientNet, such as "(1) training with very large image sizes is slow; (2) depthwise convolutions are slow in early layers. (3) equally scaling up every stage is sub-optimal" [81]. Therefore, a new version called **EfficientNetV2** was published in 2021. EfficientNetV2 improves mainly *training speed and decreases the size of the models.*

Research replaced some of the main building blocks with *Fused-MB Conv layers.* The Fused-MB Conv layer 'fused' the pointwise 1×1 convolution

[21]Resolution of the input image.

[22]Width is the number of channels, and depth is the number of layers.

Name	**Param.**	**Inp.**	**Name**	**Param.**	**Inp.**
EfficientNetB0	5.3M	224	EfficientNetB4	19M	380
EfficientNetB1	7.8M	240	EfficientNetB5	30M	456
EfficientNetB2	9.2M	260	EfficientNetB6	43M	528
EfficientNetB3	12M	300	EfficientNetB7	66M	600

Table 2.1: Inp. is input in one dimension of the square RGB image (for EfficientNetB0 that is $224 \times 224 \times 3$), Param. means the number of parameters [12].

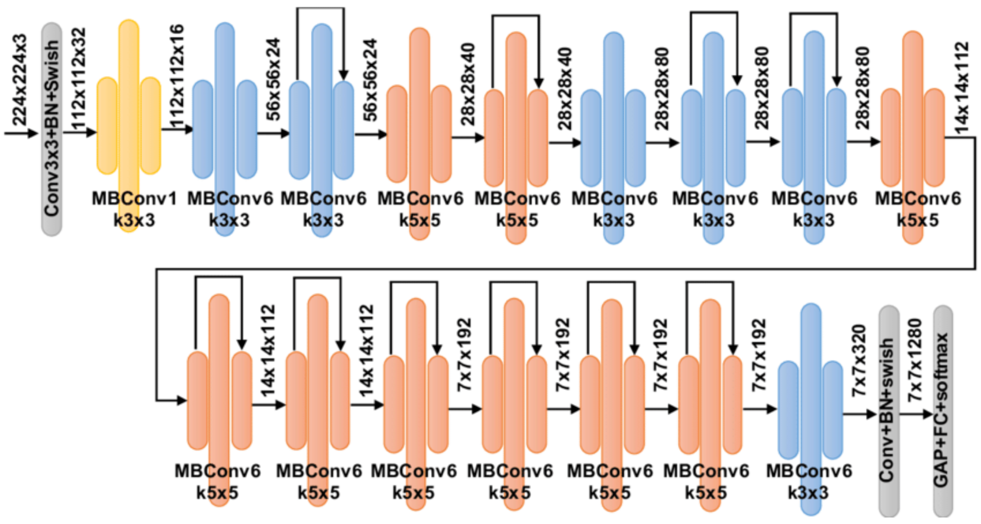

Figure 2.6: The EffecientNetB0 general architecture [7].

and a 3×3 depthwise convolution into a regular convolution as can be seen in Figure 2.7. The Fused-MB Conv layer has more parameters and requires more computation, but is executed faster due to the possibility of parallelization [80, 1:46]. This, together with a few more changes (smaller expansion ratio for the MBConv layers, smaller kernel sizes with more layers, removal of the stride-1 stage, introduction of dropouts and data augmentation during training), gives better results in a faster time [82].

EfficientNetV2 comes with different variations. Some of them are in the Table 2.2 and are currently one of the best CNNs available for image recognition.

2.6.5 Other networks

There are many different network architectures. We will briefly mention only a few that we will use in our work.

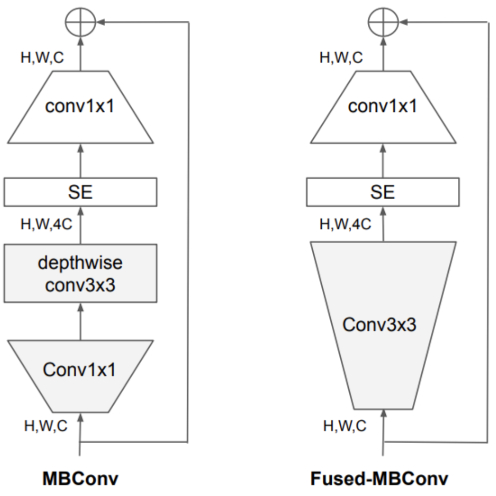

Figure 2.7: Comparision of Fused-MBConv and MBConv in EfficientNets [8].

Name	**Param.**	**Inp.**	**Name**	**Param.**	**Inp.**
EfficientNetV2B0	7.2M	224	EfficientNetV2S	21.6M	384
EfficientNetV2B1	8.2M	240	EfficientNetV2M	54.4M	480
EfficientNetV2B2	10.2M	260	EfficientNetV2L	119.0M	480
EfficientNetV2B3	14.5M	300			

Table 2.2: Inp. is input in one dimension of the square RGB image (for EfficientNetV2B0 that is $224 \times 224 \times 3$), Param. means the number of parameters [12].

ResNet was introduced before DenseNet and was one of the first architectures that tried to overcome the problem of input and gradient degradation in long convolutional networks. Its architecture consists of residual connections (also known as skip connections) between layers that connect with summation with only all preceding layers (in contrast to DenseNet, where all layers are connected to each other by concatenation) [83], [84].

InceptionV3 architecture consists of symmetric and asymmetric blocks of acyclic graphs, concatenated at the end. These blocks are easier to fit into memory [85], [86].

InceptionResNetV2 combines the architecture of Inception and ResNet into one architecture, which, due to residual connections, significantly accelerates the training of the Inception parts [87].

NASNetLarge architecture was found with an automatic search method called **Neural Architecture Search** (NAS, therefore, NAS-Net). It searched the possible space of options on a smaller dataset where the main building

Name	**# of parameters**	**Image size**	**Released**
ResNet-50	25.6M	224x224	2015
ResNet-101	44.7M	224x224	2015
ResNet-152	60.4M	224x224	2015
InceptionV3	23.9M	299x299	2015
InceptionResNetV2	55.9M	299x299	2016
Xception	22.9M	299x299	2016
NASNetLarge	88.9M	331x331	2017

Table 2.3: Comparison of different CNNs [12], [13], [14], [9], [15].

block is established and then transferred the block (and stacked more copies of it) to a final model on a large dataset. The new design is found after this process [88].

Xception means *Extreme Inception* since it is based on IncetionV3 net, where the standard convolution is replaced by *depthwise separable convolutions.* The resulting architecture gives slightly better results. Depthwise, separable convolutions operate in two phases. First, depthwise convolution occurs (which reduces width and height), and then pointwise convolution (which reduces channel dimension) [89].

We compared the top-1 and top-5 accuracy of these models in Figure 2.8, and Table 2.3 compares the number of parameters, the input image resolution, and the release year.

2.7 State-of-the-art image feature extraction

Historically, feature extractions from images were taken using different methods such as *color histogram*, *Oriented FAST and Rotated BRIEF* (ORB), *Scale-Invariant Feature Transform* (SIFT), and *Speeded Up Robust Features* (SURF) [39].

Nowadays, due to progress in CNNs, which showed their performance in the ImageNet challenge, state-of-the-art CNN models described in the previous section are being used for extracting features from images. We can use the final output, in fact, the output of any layer of each model as an embedding of the image, which can be further used for different tasks, including recommendations [80, 1:51].

2.8 Multiple Instance Learning

Many things in nature could be represented by a set of feature vectors of fixed length but with one label (representing one entity). A simple example can be a photo of an animal where only the rear view, where the tail is visible, can

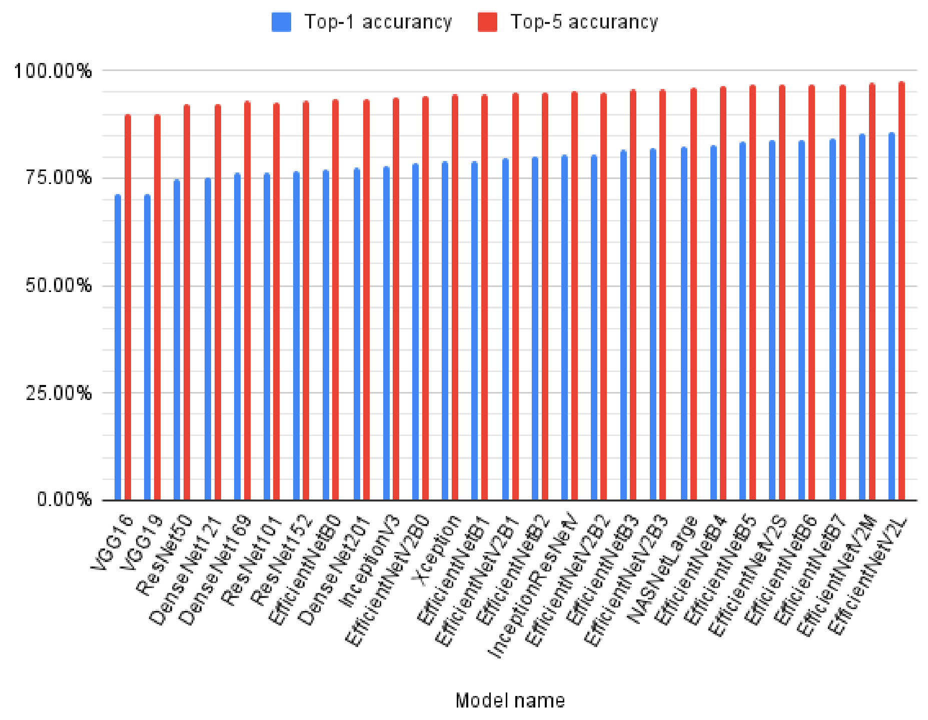

Figure 2.8: Comparison of top-1 and top-5 accuracy of selected CNN models on ImageNet validation dataset. They are sorted by top-1 accuracy [9].

reveal the correct species, and having just a front view would not be sufficient for correct labeling.

For similar cases, a method called **Multiple Instance Learning**[23] (MIL) was introduced, which can aggregate multiple features into one feature [90]. It expects samples to be composed of multiple lower-level instances with more than one set of fixed-sized vectors (together, these instances are called bags) and only one high-level label (of the bag). This can be represented as multiple photos and single name of a species compared to one fixed-sized vector (one photo), as is common in ANN models presented in previous sections. This should lead to higher accuracy as more information is present [11].

The example introduced by the authors of [91] demonstrates the usage of this system for the description of a person from a series of images. They take sub-images (instances) with different centers and sizes from an image (bag) and label them as positive if the person is present or negative if the person is missing.

[23]Or multi-instance learning.

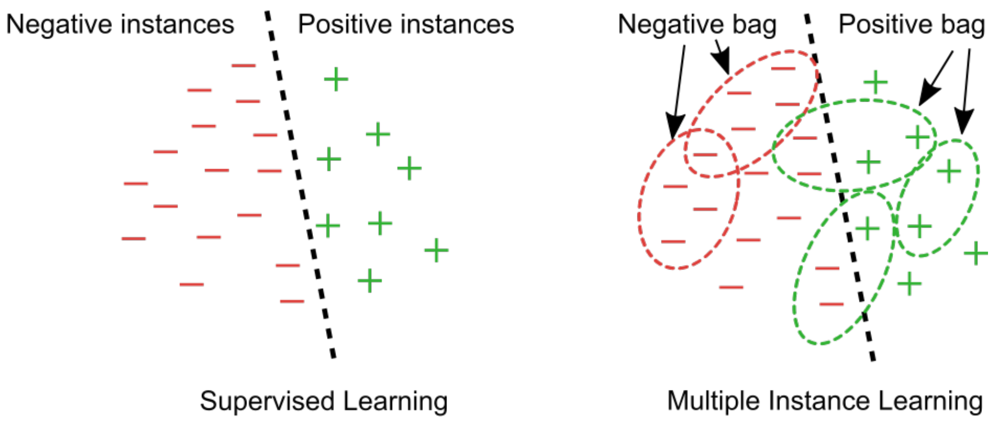

Figure 2.9: “Comparison of supervised learning and multi-instance learning. The classifier is learned over bags instead of instances in MIL” [10]. In this example target class is positive or negative, i.e., bag can be positively or negatively labeled. A bag is considered negative if it contains only negative instances and no positive instances, otherwise it is considered as positive.

In Figure 2.9 is another example of MIL problem illustrated.

In other real-life problems, we may not always have the same number of instances. Therefore, our model needs to be flexible and not only accept a constant number of instances, but dynamically adjust.

In view of the fact that the input data vary from classical machine learning approaches, that is *supervised learning*, *unsupervised learning*[24], and *reinforcement learning*[25] [92], the multi-instance learning was regarded as the fourth approach for machine learning and “should be placed in somewhere between supervised learning, whose training sets are with no ambiguity, and unsupervised learning, whose training sets are with maximal ambiguity” [92].

As in [11] MIL method can be defined as

$$\phi_i = g(\{k(x, \theta_i)\}_{x \in b}),$$

where b represents the bag, x is a single instance of the bag, k is a distance function (or kernel) over the bags, and the dictionary ($\Theta = \{\theta_i \in \chi | i \in \{1, ..., m\}\}$, χ is a non-empty instance space) containing bags. Moreover, g is a pooling function (e.g., minimum, maximum, or mean) that aggregates multiple instances from bags into one instance (ϕ).

The whole model can be represented as a neural network, where the distance function k can be the first part of a model (or the whole model itself),

[24]Unsupervised learning discovers patterns and similarities on its own.

[25]“The examples are with no labels but with delayed rewards that could be viewed as delayed labels.”

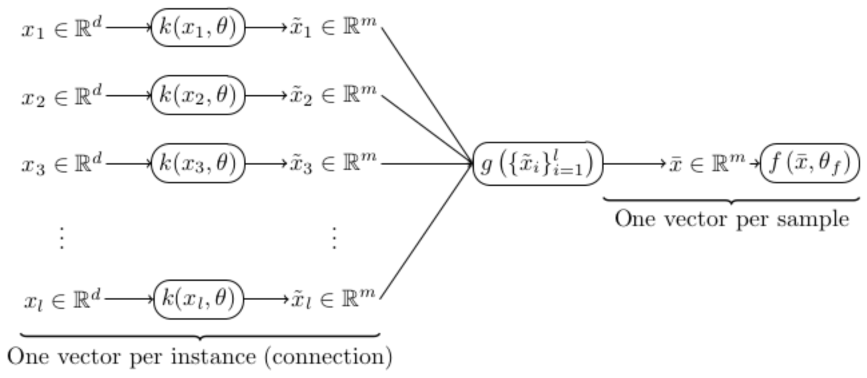
$x_1 \in \mathbb{R}^d$
$k(x_1, \theta)$
$\tilde{x}_1 \in \mathbb{R}^m$
$x_2 \in \mathbb{R}^d$
$k(x_2, \theta)$
$\tilde{x}_2 \in \mathbb{R}^m$
$x_3 \in \mathbb{R}^d$
$k(x_3, \theta)$
$\tilde{x}_3 \in \mathbb{R}^m$
$x_l \in \mathbb{R}^d$
$k(x_l, \theta)$
$\tilde{x}_l \in \mathbb{R}^m$
$g\left(\{\tilde{x}_i\}_{i=1}^l\right)$
$\bar{x} \in \mathbb{R}^m$
$f(\bar{x}, \theta_f)$
One vector per sample
One vector per instance (connection)

Figure 2.10: *"Sketch of the neural network optimizing the embedding in embedding-space paradigm."* [11]

followed by pooling layers g (after which we get one vector) and ended by the second part of a neural network model f to map the final vector to a bag label. Backpropagation algorithms can optimize the weights of the resulting architecture.

This process is summarized in Figure 2.10, where $(x_1, x_2, ..., x_l) \in \mathbb{R}^d$ are single instances of one bag mapped by second model f to one output vector.

Chapter 3

Analysis and design

In this chapter, we propose several methods for feature extraction from an image with the help of CNNs and evaluate extracted features on recommendation task with recall metrics.

3.1 Motivation

Visual presentation of a product on a website can take a large proportion of the space dedicated to the product demonstration. That can strongly influence customers when they decide whether to view and purchase the product or not. Therefore, we can use this fact and implement a recommender system based on the image representation.

Martin Pavlicek already tried this in his work [39], where he compared several methods of producing an embedding from an image and, after that, a recommendation. Pavlicek concludes that VGG16 outperforms other methods not based on CNNs (such as color histogram and ORB). However, he did not compare VGG16 with other CNNs, and since the release of his work in 2018, several new architectures have been introduced that could outperform even the VGG16 model.

Since training our own CNN is not possible with our resources, we will retain the standard practice of using pre-trained models and only *fine-tune* their weights (or only their dense layers) for our use case.

We will focus on three situations. Firstly, on a cold start problem where we do not have any interaction data but only pictures of items. We will follow up with Pavlicek and compare VGG16 with other newer CNNs models. The second case will be when we have interaction data in the form of *ALS embeddings* of items, and we can train our model on these embeddings with one given image of the item. Lastly, we will propose a method to use more images of one item with one ALS embedding based on the description of *MIL* in Section 2.8 and see if this can increase the recall.

3.2 Model selection and Default recall

As we described in the previous chapter, *state-of-the-art CNNs* can encode the image of an item into an embedding, where we can take the output of any CNN layer and consider it as an embedding. We can then compare these embeddings with some distance metrics and find similar items to a selected item. The similar items are then suitable for a recommendation, and we can calculate the recall of these embeddings. However, embeddings generated from different layers will produce different recalls. Therefore, we will compare the recall of different layers of trained but not fine-tuned models. Only embeddings generated by dense layers or whole CNN without any dense layers will be compared for simplification.

Since the recall calculation does not depend on the length of the embedding[26], we can calculate the recall on the output of any layer with the same method and compare the results.

We will call **Default recall** the highest recall of each CNN model of these *not* fine-tuned models. This is the cold start problem situation, where we will mainly compare VGG16 to newer CNNs. Later, we will try to improve this *Default recall.*

For the reasons mentioned above, we will create an *evaluation framework*, where we will calculate the recall of a given model.

3.3 Model evaluation

When we have a given model, we need to evaluate its performance. We will create an *evaluation framework* with four steps.

1. *Train* the model. (optional)

2. Using a (non)trained model, *predict embeddings* of all images.

3. Find *K Nearest Neighbors* (KNN) to all items we predicted.

4. *Evaluate* the embeddings, i.e., calculate the recall with item-KNN from step 4.

In the third step, we predict embeddings of not-yet-seen items on a given model (optionally) trained on items with known ALS embeddings.

With a *leave-one-out cross-validation* evaluation, we will calculate the recall of our recommender model.

[26]Cosine similarity or MSE can be calculated on a vector of any size.

3.4 Search for the best hyperparameters and One-to-one training

After collecting Default recalls from all models and their different layers, we will select the *top-10* best performing models and fine-tune them further with the goal of increasing the recall.

We will consider every model only once, with the layer which achieved the highest Default recall. This will help us discover and compare more models, which we prefer more than only comparing different outputs of a single model.

Since all used models were pre-trained on the object classification task, their architecture may not be optimal for the recommendation task. Therefore, models may need more (or less) dense layers on top and perhaps even different widths of dense layers compared to the original architecture used to adjust to a new problem.

For the reasons above, we will search for the best hyperparameters of these *top-10* models: the number of dense layers after the best performing layer and the width of these layers.

The final step is to train and evaluate the resulting architectures with the best hyperparameters of all *top-10* models using our framework and calculate a new (hopefully better) recall. This is called *transfer learning* when we try to reuse a trained network for a different task it was originally designed. We will call this step **One-to-one training** as we map one image to one ALS embedding, i.e., the embedding based on the user's interactions.

The learning (or training) process means *fine-tuning* model output to the item ALS embedding and minimizing the *loss function.* In our case, map the first image (and later images) of the item (input) to the ALS embedding of its item (target output). However, as already stated in [93], training models on ALS embeddings are not in monotonic relation, and minimizing loss function may not improve the quality of recommendation algorithms. Nevertheless, the training should improve final recall. In other words, the model tries to learn connections based on user interactions encoded in ALS embeddings of the items.

3.5 Many-to-one training

Until now, training was done using only one image of an item to the ALS embedding of the same item. At this moment, we will take advantage of multiple images we have from one item and try to train all top-10 models with their best hyperparameters on more than one image to one single ALS embedding and see if recall improves.

During training, we will proportionally lower the importance of an image if it shares the ALS embedding with some different images of the same item. If, for example, we would train on 1000 images of one item and the other items would have only one image, the model could overfit to this one item with 1000 images. The reduction of the importance of each image should prevent this.

As we provide more information to a model, we hope to get a better recall. We will call this phase **Many-to-one training** as we map one and more images to one ALS embedding.

As a result of these experiments, the best model is found, which we will call the **best model**.

3.6 MIL training

We propose two more implementations for the use case when we have two and more embeddings (representing several images of one item) with only one corresponding ALS embedding. As we described *MIL* in the previous chapter, we assume that the MIL model should be able to combine *multiple embeddings of one item* into a single embedding, which can then be fine-tuned on the ALS embedding.

We will use the *best model* found in previous experiments (i.e., one with the highest recall) as our base model to produce embeddings from images in our MIL implementation (as a function k according to the previous description in Figure 2.10).

We propose two methods of MIL. A naive solution we will call *Simple MIL*, and a solution based on [11] we will call *Ragged MIL*.

In the **Simple MIL** method, we will take the embeddings produced by the best model, and then we will perform operation maximum (max), minimum (min), or mean on these embeddings of one item. For example, if there are two embeddings (from two different pictures of one item) of size l, i.e., $x_1, x_2, ..., x_l \in \mathbb{R}^l$ and $y_1, y_2, ..., y_l \in \mathbb{R}^l$, the result of the max operation is

$$MAX(x_1, y_1), MAX(x_2, y_2), ...MAX(x_l, y_l)$$

embedding. We will fit the resulting embedding of this operation with our custom model, which contains only dense layers, to the ALS embedding of the corresponding item.

In **Ragged MIL**, we will also produce embeddings from images with the best model, and then we will use neural networks to fit a variable number of images to one final ALS embedding.

We will always find the best hyperparameters before mapping our embedding(s) to the ALS embedding.

3.7 Workflow summary

To wrap up everything, we will first calculate the *Default recall* of each model and its various outputs. Then, we will select the top-10 different models and search for the best hyperparameters for our task. Train each of these top-10 models with its best hyperparameters as *One-to-one* and *Many-to-one*. Lastly, we will implement two solutions to the MIL problem, *Simple MIL* and *Ragged MIL*. The result of these experiments should be the best CNN-based model for recommendations based on an image of items.

Chapter **4**

Realisation

This chapter will describe our implementation of the proposed solution with a short description of the used technologies.

4.1 Used technologies

Firstly, we will briefly describe the technologies and tools that we will use during development. The implementation is written in the *Python3.8* programming language in the *Jupyter Notebook* environment. For our experiments, we have been provided with a *250 GB RAM* computer with *processor Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz with 40 threads, and Tesla V100-DGXS 32 GB graphic card.*

4.1.1 TensorFlow

TensorFlow (tf) is an open-source platform for machine learning. It contains many state-of-the-art models that have already been trained, same as an interface to build and modify models, and many tools to prepare data, tune hyperparameters, and evaluation.

The *tf.data.Dataset API* (we will call it shortly tf.Dataset) provides input pipelines for training a model. As in our case, dataset can have several GB of data, which is not always possible to load into memory. Tf.Dataset provides a simple interface to preprocess input data and split them into batches, which can be streamed one by one into a graphic card where we trained the model.

Part of the TensorFlow is the *Keras Tuner* library, which contains state-of-the-art methods for searching for hyperparameters, such as number of layers, neurons in one layer, learning rate, the best optimizer, and many more. We will use the *Hyperband* tuning algorithm, which trains more networks at once in just a few epochs and carries forward only part of the best to further training. Moreover, Hyperband uses adaptive resource allocation and early stopping during the search process and is suitable for our use case.

4.1.2 Jupyter Notebook

We will use *Jupyter Notebooks* for the development environment, which allows interactive development and avoids repeatedly loading large amounts of data. It allows checking the results on the go, which saves time.

4.1.3 Other libraries

Other essential libraries that we will use are

- *Numpy* - simplifies and significantly improves the speed of mathematic operations such as work with multidimensional arrays (matrices) and shape manipulation,
- *Pandas* - helps with data manipulation, presentation, and modifications,
- *Pickle* - serialize and de-serialize data which helps store them in a file,
- *Matplotlib* - a library that helps with data visualization,
- *Logger* - a library that helps with logging information,
- *Sklearn* - similarly to TensorFlow, Sklearn provides tools for machine learning; we will use it for searching for nearest neighbors.

4.2 Folder description

The code is divided into several Python files and three main Jupyter Notebooks. In the first, we cleaned the data; in the second, we implemented model selection - the *Default recall*, *One-to-one*, and *Many-to-one* training. In the last notebook, we implemented both MIL methods - *simple* and *ragged*. Furthermore, several supporting Python files contain classes that we import into the Notebooks.

Moreover, in a file called *CONFIG*, we can change some constant values such as path to an image, path for logs, path for saved models, and more.

In class *DatasetsHandler*, we implemented functions for loading datasets into pandas dataframes, processing them into tf.Dataset and other dataset-related functions.

4.3 Evaluation framework

Framework for evaluation of the model is implemented in class *ModelEvaluator* with its main function *train_eval_model* as can be seen in Algorithm 2. First, we can train a model and then evaluate it. For tasks related to models (loading model, compiling model, and Keras Tuner implementation), we created the class *CModels*, and we are using the TensorFlow library, to use already implemented models.

Training is implemented with *ModelCheckpoint* and *EarlyStopping* callbacks. The EarlyStopping ensures that the model will automatically stop after no improvement over a few epochs. ModelCheckpoint ensures that the best model (on validation dataset) is saved on disk for later usage.

For the computation of the nearest neighbors, we used a function called *NearestNeighbors* from the *sklearn* library. Moreover, for the computation of Recall, we used a class called *RecallMeter* and its *measure_leave_one_out* function.

Algorithm 2 The pseudocode of the train_eval_model function.

```
Require: (model, trainDSs, imgesWithoutEmbeddings, rawInteractions)
Ensure: resultRecall
1: if trainModel then
2:     trainModel(model, trainDSs)
3: end if
4: computedEmbeddings ← predictEmbeddings(model, imgesWithoutEmbeddings)
5: similarItems ← findKNNItems(computedEmbeddings)
6: resultRecall ← computeRecall(similarItems, rawInteractions)
```

We can calculate *Default recall* by passing a model and the required datasets into our framework with a number of $epochs = 0$, which means that the model will be only evaluated and not trained.

4.4 Hyperparameters and models

After Default recall is computed, we can take the top-10 models and search for the best end layers with the help of *Keras Tuner* and the *Hyperband* search method. We created one function, *get_all_models*, which takes as parameter name of the model in format *MODEL_NAME_X*, where X represents a number of omitted dense layers at the end of a model. For example, if $X = 3$ in the VGG16 model, we will not include any of the last layers. Furthermore, the function *get_all_models* also returns the required image size for each model. Therefore, before starting the Keras Tuner, we create tf.Dataset with corresponding image sizes.

We would note that during training, the convolutional parts of our models are frozen[27]. We train only the top dense layers.

[27]The weights of frozen layers do not change during training.

4.5 MIL implementations

In a Notebook called *MIL*, we implemented the last two proposed methods (Ragged and Simple MIL, where users can switch between modes). In *MILClass* are necessary functions to prepare datasets, define Keras Tuners hyperparameters, and manipulate with the models of both methods. *MILRagged* class in the same file contains the crucial part of the Ragged MIL method.

There are several hyperparameters in the MIL notebook that help to accelerate experiments, such as

- *MIL_VARIANT* - switch between *Ragged* and *Simple MIL*, where *True* = Simple MIL, *False* = Ragged MIL.
- *MIL_OPERATION* - switch between underlying operation; options are *MEAN*,*MIN*, and *MAX*.
- *LAST_LAYER* - determines whether we cut the model in the first phase of its last layer, which can give better results, where *True* = no, use last layer, *False* = yes, cut the last layer.
- *TOP_K_IMGS* - the maximum number of images the model will work with from one item.
- *IMG_SIZE* - the size of the input image in pixels.
- *PHASE1_BATCH* - the batch size of phase 1.
- *PHASE2_BATCH* - the batch size of phase 2.
- *EPOCHS_KT* - Number of epochs in Keras Tuner.

Other hyperparameters were found again with the help of Keras Tuner.

We are implementing a simplified version, where we do *not* train end-to-end (i.e., the base model in MIL, which produces embeddings from images, is not being trained). Therefore, we first extract the embeddings and then try to map them to ALS embeddings. An important part is the difference between the mapping in Simple and Ragged MIL.

In the case of *Ragged MIL*, we predict embeddings of images called *embeddings_big* in the dataframe. We need to process them before using them as input into the second model, which will map them into ALS embedding. The embeddings must be in format n, k, l, where n is a number of items, k means a maximum number of images on input, and l is the size of *embeddings_big*. This will become input into the second model, where if an item has less than the maximum number of images, we fill missing items with zeros. We can finally use Keras Tuner to find a model with the best hyperparameters and train and evaluate the final model with our evaluation framework.

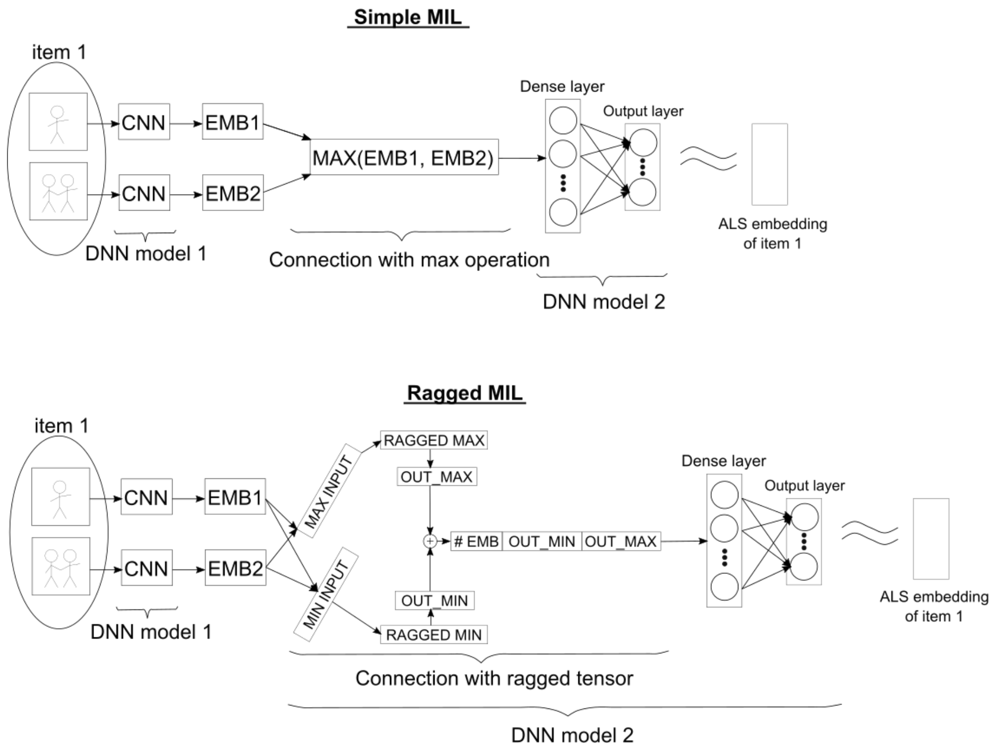

Figure 4.1: Comparison of Simple MIL and Ragged MIL.

On the other hand, *Simple MIL* performs the selected operation (min, max, mean) on extracted embeddings from images and creates one embedding for one item. Therefore, the second model contains only dense layers and is trained on items with only one input and one output embedding. Once more, we use Keras Tuner to find the best hyperparameters of the second model. The architecture of both MILs is shown in Figure 4.1.

Chapter 5

Experiments

Experiments proposed in Chapter 3 were verified using our implementation and provided datasets. The results of these experiments will be presented in this chapter.

We measured recall on $k \in 5, 25, 50, 100$, but since we are mostly interested in *recall@5*, we will present only this value. The rest of the results can be seen on the attached medium in the folder named **results**.

5.1 Datasets

We have available a dataset consisting of around 700,000 path to images from a furniture e-shop based in Australia and around 1.3 million interactions for our experiments. The dataset used during this work is confidential and was only provided to us for research purposes. Therefore, the dataset is not included in the attached medium.

First, this data set needs to be pre-processed, which we do in Notebook called *data_preprocess*. That means we need to ensure that all images are working (we can read them from disk and use them as input). All the dataframes are in *pickle* format on disk, and we work with them using the *Pandas* library, where we expect a column with a path to the image.

The cleaning process consists of more tests. Namely, we check if the value is not empty, is a legitimate path, and has the correct file extension (*.jpg* or *.png*). Next, we check the encoding of a file and try to load the image into the TensorFlow library, which will be essential for our future work. All entries that do not comply with only one of these tests are deleted.

The dataframe contains the following columns:

- *item_id* - unique ID of the product,
- *property* - which can have two values, *main_image* which is the main picture of the product and *additional_images*, which symbolizes an ad-

ditional image of the product (that can be seen, for example, in product detail),

- *filename* - path to the image on disk.

After this process, we have 350,000 unique paths to images of approximately 70,000 different items (in the dataframe *only_functional_paths*), which we will use for the evaluation process. We also have 66,000 unique items with their ALS embeddings with the size of 1024 (in dataframe *final_merge*), which will be used during training. Furthermore, we split the dataframe with ALS embeddings into a train and a valid dataframes, where the training dataframe will be used during training, and the valid dataframe will be used after every epoch for validation (the valid dataframe will prevent our models from overfitting).

We also have two more dataframes, representing the *rating matrix* of interactions - *raw_interaction_valid* and *raw_interaction_test*, which we will use for the evaluation of the recall of our models as well. As the names suggest, the first (valid) dataset will be used during training, and the second (test) dataset will be used only once for the final evaluation of our models. Both dataframes have the following columns:

- *item_id* - unique ID of the product,
- *user_id* - unique ID of the user,
- *interaction_type* - type of interaction (detail_views, cart_additions, bookmarks, purchases),
- *timestamp* - timestamp when interaction happened,
- *value* - value of interaction (number).

5.2 Keras Tuner configuration

We used the TensorFlow library implementation of all models with pre-trained weights on the ImageNet dataset. We used Adam as an optimizer and a custom loss function inspired by [94] as a *ratio* between *cosine similarity* (COS) and *mean squared error* (MSE)

$$loss = \alpha COS + \beta MSE + 1,$$

where $\alpha = 1$ and $\beta = 0.01$. When cosine similarity is calculated, it normalizes the vectors, giving us no information about the magnitude of the vector. Our loss functions ensure that the vectors have a similar angle and a similar magnitude.

In general, we use Keras Tuner to find the optimal architecture of the top layers. In the *One-to-one method*, we set the hyperparameters to append $0, 1, 2$ dense layers on top of the model, each with $1, 2, 3$ times 1024 neurons and a learning rate in $(1e-5, 1e-2)$.

For *Ragged MIL*, we set the output of each operation as hyperparameters and the possible values are $1024, 1536, 2048, 3072$. For example, if Keras Tuner selects value 2048 and configures Ragged MIL to append output from operations min and max, we get output consisting of 2*2048 values (+1 representing a number of embeddings in the input as suggested in [95]). Moreover, we search for other hyperparameters similar to those in the One-to-one method, namely, how many layers to append to output (0-3), each containing 1-3 times 1024 neurons and the learning rate in $(1e-5, 1e-2)$.

Finally, for *Simple MIL*, we searched a number of layers on top (0-3), neurons in each layer (1-3 times 1024), and the learning rate in $(1e-5, 1e-2)$.

In each Keras Tuner search of hyperparameters, we append the output layer consisting of 1024 neurons, which corresponds to the size of our ALS embedding.

5.3 Results of Default recall

We compared all these models: *ResNet* versions 50, 101, and 152, *InceptionV3*, *InceptionResNetV2*, *DenseNet* versions 121, 169, and 201, *NASNetLarge*, and *Xception*, where we measured recall on 2 layers, *EfficientNet* versions B0, B1, B2, B3, B4, B5, B6, B7, and *EfficientNetV2* versions L, M, S, B0, B1, B2, B3, where we measured recall on 3 layers and finally *VGG19 and VGG16*, where we measured recall on 4 layers. In total $10 \times 2 + 15 \times 3 + 2 \times 4 = 73$ output layers of 27 different models.

We consider the output of VGG16 as our baseline as we follow the results of [39], which we hope to surpass. We also measured the recall of given ALS embeddings, which we use for training. Recall of ALS embeddings should correspond to the ideal recall we would like to achieve.

The results can be seen in Table 5.1, where we compared the best resulting outputs of each model. In Figure 5.1, we compare only the best output layers of each model. We experimentally verified that the best performing embeddings were found in the layer placed right after the convolutional part in all cases. The model with the **highest recall is DenseNet201**, which achieved approximately half of the recall of ALS embeddings and performed approximately 15% better than the best recall of VGG16. All three versions of DenseNet (201, 169, and 121) are in the top-3 positions, followed by EfficientNetV2 and its versions B1, B3, and B0. VGG16, EfficientNetV2B2, VGG19, and EfficientNetV2L ended the top-10 best performing models. We will use all these models without their dense layers in the following experiments (in Keras Tuner and the following).

Model name	Rec@5 (%)	Model name	Rec@5 (%)
ALS embeddings	3.62%	ResNet152_1	1.03%
DenseNet201_1	1.82%	ResNet50_1	1.03%
DenseNet169_1	1.76%	ResNet101_1	0.98%
DenseNet121_1	1.73%	NASNetLarge_1	0.84%
EfficientNetV2B1_2	1.70%	EfficientNetV2S_2	0.80%
EfficientNetV2B3_2	1.70%	InceptionV3_1	0.77%
EfficientNetV2B0_2	1.68%	EfficientNetB5_2	0.57%
VGG16_3	1.57%	EfficientNetB7_2	0.51%
EfficientNetV2B2_2	1.55%	EfficientNetB6_2	0.42%
VGG19_3	1.54%	EfficientNetB4_2	0.42%
EfficientNetV2L_2	1.37%	EfficientNetB3_2	0.37%
EfficientNetV2M_2	1.31%	EfficientNetB1_2	0.32%
Xception_1	1.29%	EfficientNetB2_2	0.29%
InceptionResNetV2_1	1.14%	EfficientNetB0_2	0.26%

Table 5.1: Table of Default recall@5 in % of the best output layer of different models and ALS embeddings (the higher, the better).

On the other hand, the first version of EfficientNet had the worst recall, filling the lowest positions on the chart. Additionally, the ResNets family architectures did not perform well either.

5.4 Results of One-to-one and Many-to-one

We continued our experiments by running Keras Tuner on all *top-10* models with a maximum of 50 epochs and a batch size of 128. Keras Tuner needed around **25 days** to find the best hyperparameters. Significantly longer time was needed for both VGG16 and VGG19, about 5 days, and other models usually needed around 2 days to complete the search process.

Except for one model, all preferred to append only *one more dense layer* with 3*1024 neurons inside[28]. In Table 5.2 we can see the result of the loss function.

After finding the best hyperparameters, we trained the networks on our items with *ALS embeddings* and evaluated them in our framework. Surprisingly, VGG16 retreated to the top with the *highest-scoring recall*, about 30% worse than the recall of ALS embeddings, and 35% increased than non-trained DenseNet201 (winner or previous experiment). In second place is VGG19, followed by EfficientNetV2B3. We can tell that **fine-tuning output of our models to ALS embeddings significantly improved the recall**.

[28]More details can be seen in attached medium in file *One-to-one Keras Tuner.txt* inside *result* folder.

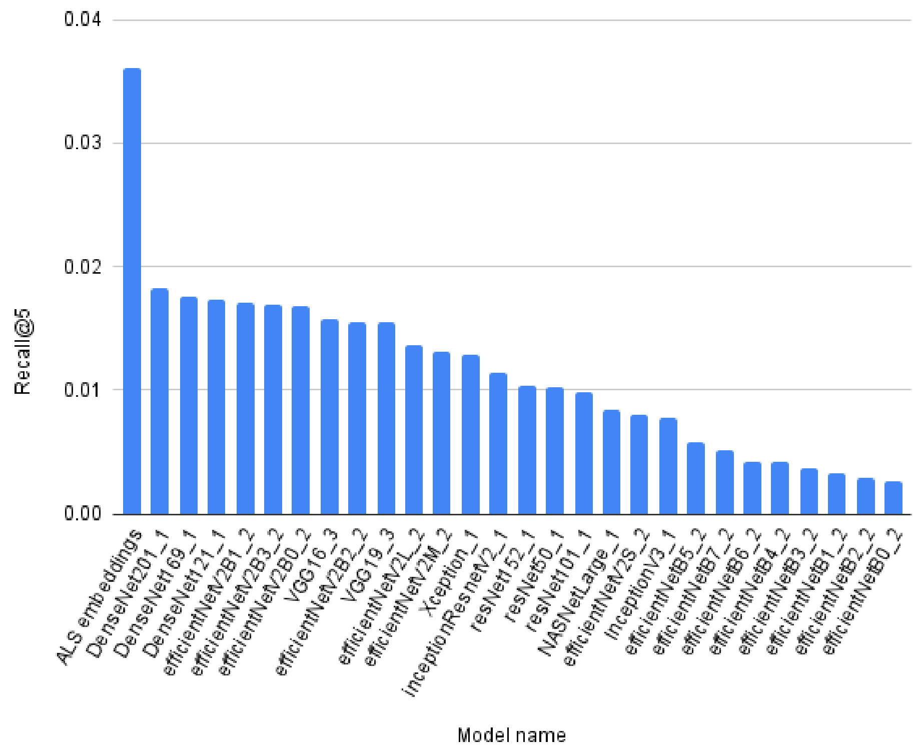

Figure 5.1: Default recall@5 of the best output layer of different models and ALS embeddings.

Model name	Loss value	Model name	Loss value
EfficientNetV2B1_2	0.7193	DenseNet121_1	0.7313
DenseNet201_1	0.7233	EfficientNetV2B2_2	0.7316
DenseNet169_1	0.7241	EfficientNetV2L_2	0.7389
EfficientNetV2B3_2	0.7249	VGG16_3	0.7796
EfficientNetV2B0_2	0.7287	VGG19_3	0.7800

Table 5.2: Loss value of models after Keras Tuner trained and searched for the best hyperparameters (the lower, the better).

Model name	1 image	2 images	3 images	4 images
VGG16_3	2.44%	2.42%	2.36%	2.32%
VGG19_3	2.33%	2.22%	2.20%	2.12%
EfficientNetV2B3_2	2.30%	2.14%	2.11%	2.10%
EfficientNetV2B1_2	1.99%	1.92%	1.88%	1.87%
EfficientNetV2B2_2	1.89%	1.84%	1.82%	1.82%
DenseNet201_1	1.84%	1.81%	1.82%	1.79%
EfficientNetV2B0_2	1.76%	1.78%	1.75%	1.73%
EfficientNetV2L_2	1.76%	1.73%	1.71%	1.69%
DenseNet121_1	1.71%	1.68%	1.68%	1.73%
DenseNet169_1	1.52%	1.53%	1.54%	1.54%

Table 5.3: Table of recall@5 in % of trained models on 1-4 images to one ALS embedding.

Interestingly, not all networks improved their recall after training. Namely, the DenseNet family did not perform well after the training process, and DenseNet121 and DenseNet169 ended up with the worst recall than without training and DenseNet201 with only a small improvement. This may be because the network weights would need significantly more time to escape from local optima.

We also tried to train models with the hyperparameters found with Keras Tuner on more images. In almost every case, the more images we provided, the lower the recall was. The reason for this could be that despite training on multiple images, we evaluate the model only on one. Perhaps, if we used some more advanced technique of weighting recommended items for multiple different images of one item, we could achieve different results.

The results can be seen in Figure 5.2 and Table 5.3. Interestingly, when trained on One-to-one embedding, VGG16, which scored almost the *worst in the loss value, had the highest recall.* It is worth comparing the order of the models sorted by the best loss metric and the models sorted by the best recall in Tables 5.2 and 5.3. We can observe that **loss and recall are not in monotonic relation, and the best loss does not translate into the best recall**.

5.5 Results of MIL

We will first describe all experiments with Simple MIL and then Ragged MIL as the experiments need more description than the previous ones. We always changed baseline architecture, then ran Keras Tuner, and the resulting model with searched hyperparameters we trained and measured recall.

In *Simple MIL*, we set up the architecture to produce embeddings (model with the best recall - trained VGG16 on one image, but few experiments were

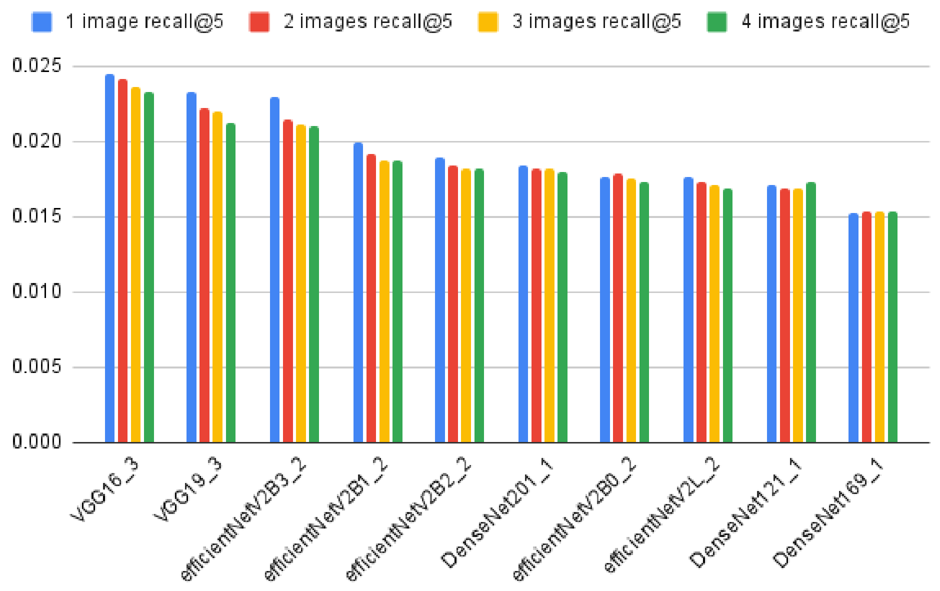

Figure 5.2: Recall@5 of trained models on 1-4 images to one ALS embedding.

done on VGG19 and EfficientNetV2B3) on the last and penultimate layers and also on operations min, max, mean. We will make a name convention of *Sim_X_Y_C_Z*, where X is the size of the embedded that is produced from the first model (in the case of VGG16, that is 1024 or 3072 if we do not use the last layer), Y is operation (min, max, mean), C can have values EF meaning EfficientNetV2B3 as baseline model or VGG19 baseline model (optional), and Z number of images on input (optional). By default, we used 2 images in our MIL model and VGG16 trained on one image as our baseline model. Results can be seen in Figure 5.3.

Similarly, with *Ragged MIL*, we performed several experiments with different hyperparameters. We first used Keras Tuner to search for the best hyperparameters, then trained and evaluated the resulting model. We will use a similar name convention for our model *Rag_X_Y_C_Z_M*, where X (size), C (different base model, new value is 2VGG which is VGG16 trained on two images), and Z (number of images) remain the same, but Y (operation) may also contain a combination MAX-MIN, MIN-MEAN, MEAN-MAX, and MMM, which means MIN-MAX-MEAN.

Figure 5.3 shows only the *top* of the graph from the range 0.021 to better visualize the difference between experiments, which may not be that apparent otherwise.

We were systematically filtering the best hyperparameters with the follow-

Model Name	**Rec@5**	**Model Name**	**Rec@5**
Sim_3072_MAX	2.460	Sim_1024_MAX	2.367
Sim_3072_MEAN	2.435	Sim_1024_MIN	2.362
Rag_1024_MAX	2.424	Rag_1024_MIN-MAX	2.355
Rag_1024_MAX-MEAN	2.421	Rag_1024_MIN-MEAN	2.325
Rag_1024_MAX-MAX	2.415	Rag_1024_MMM_2VGG	2.277
Rag_3072_MAX	2.414	Rag_3072_MMM_2VGG	2.272
Rag_1024_MAX_3PIC	2.412	Rag_1024_MIN	2.252
Rag_1024_MEAN	2.406	Sim_3072_MAX_VGG19	2.241
Rag_1024_MAX_4PIC	2.405	Rag_1024_MAX_VGG19	2.239
Sim_3072_MAX_3PICS	2.398	Rag_1024_MAX_EF	2.215
Sim_3072_MAX_4PICS	2.397	Sim_1024_MEAN_EF	2.171
Sim_1024_MEAN	2.396	Sim_1024_MAX_EF	2.147
Rag_1024_MMM	2.380	Sim_1024_MIN_EF	2.143
Rag_3072_MMM	2.380	Sim_3072_MIN	1.603

Table 5.4: Table of recall@5 in % of MIL with different architectures and hyperparameters.

ing observations. 1) Taking the *final layer* (1024) gives a *better recall* than the second last in Ragged MIL. In Simple MIL, this is *vice versa* - removing the final layer gives better recall. 2) Performing *max operation* gives the *best results*, and on the contrary *min operation* gives the *worst results.* 3) The *more input image* we use, the *worse recall* we get.

Many experiments took place, but we were unable to beat the recall of VGG16 trained on one image on offline tests. The Simple MIL with 3072 input embedding and VGG16 baseline network using two images was the best performing model. We also tried different models, more pictures, and a combination of min, max, and mean in Ragged MIL.

Several illustrations of the recommendations produced can be seen in Figures 5.4, 5.5, 5.6 and 5.7. In each figure, recommendations of four models are shown to the item on the left (first column), with descending importance of recommended items (items on right are less relevant). The models are Rag_1024_MAX, Sim_3072_MAX, trained VGG16 on One-to-one and not trained VGG16 in this order from the first row to the last row. We can see that not trained VGG16 recommends more based on shapes. However, after the model is fine-tuned on ALS embeddings, it recommends more based on semantic similarities of items.

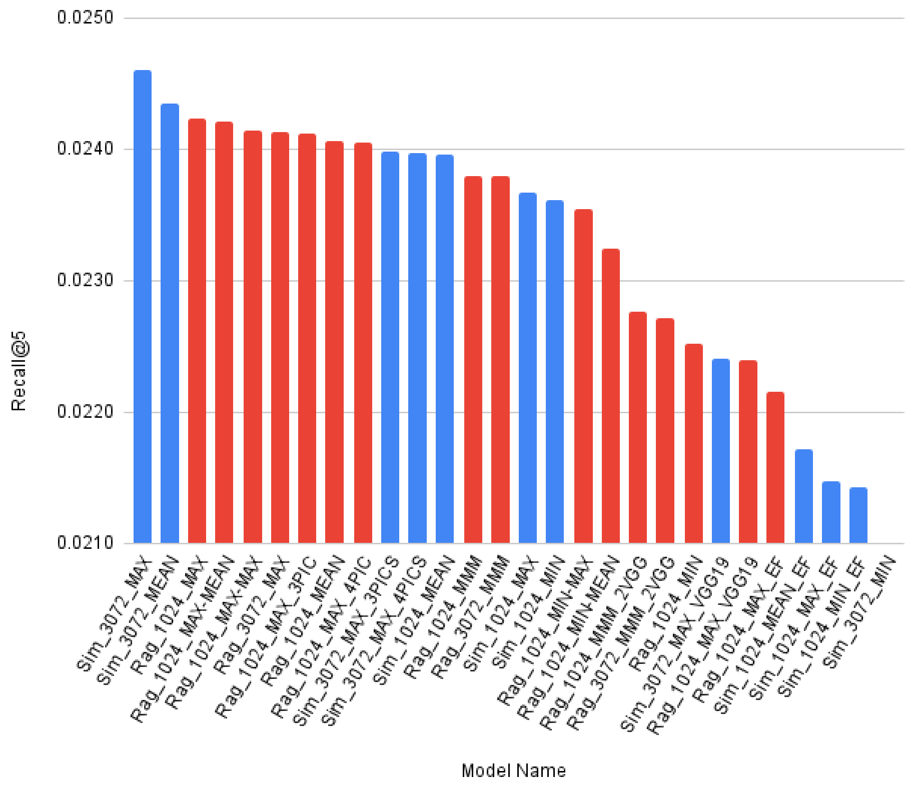

Figure 5.3: Recall@5 of MIL with different architecture and hyperparameters (blue is Simple MIL, red is Ragged MIL). Y-axis starts at 0.021 to stress out the difference between individual versions.

5.6 Online tests

From April 25, 2022, to March 2, 2022, we compared 4 models in online **A/B test**. The 4 models are *non-trained VGG16* as our baseline, then the best performing models in their category on offline tests, thus *trained VGG16* on one image (the best model we found on One-to-one and Many-to-one training), and both best performing MILs - *Sim_3072_MAX* and *Rag_1024_MAX*. In total, we get 31,362 participants randomly and equally divided between all models.

We measured the click-through rate of each model recommendation as the average click-through of all users in the test group.

The results can be seen in Table 5.5. The observed conversion rate of **7.64% of Rag_1024_MAX is 1.43% higher** than trained VGG16, and we

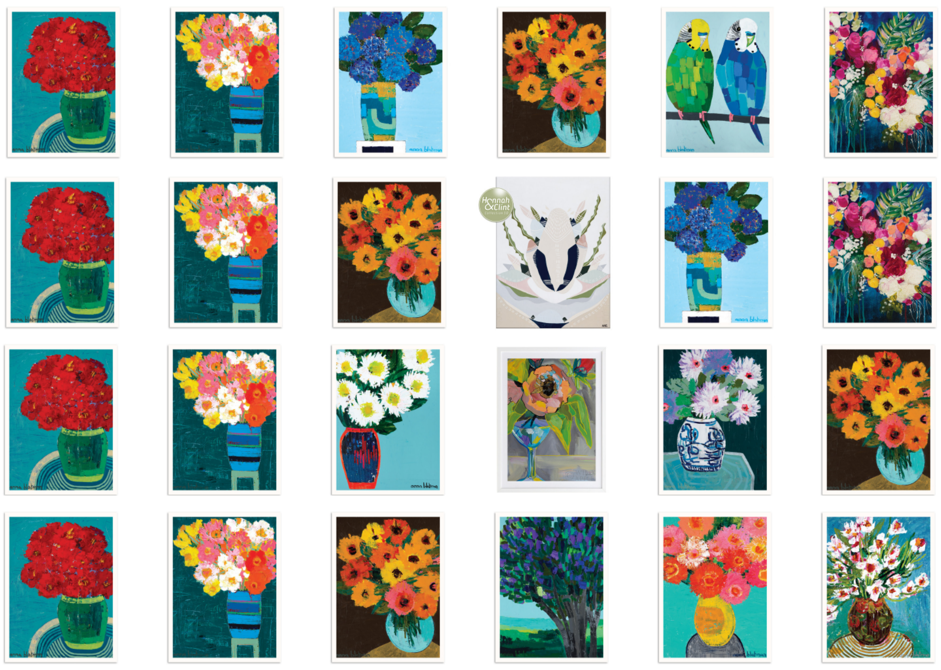

Figure 5.4: Showcase of recommendations (to the first image in each row). First row recommendations are based on Rag_1024_MAX, second row on Sim_3072_MAX, then trained VGG16 on One-to-one and not trained VGG16.

Model Name	**# of recomm.**	**Clicked recomm.**	**CTR (%)**
Rag_1023_MAX	302,977	23,143	7.63853
Sim_3072_MAX	306,840	23,635	7.70271
Trained VGG16	302,239	22,761	7.53079
non-trained VGG16	293,885	19,744	6.71827

Table 5.5: Resulting CTR of models in the online A/B test.

can conclude that Rag_1024_MAX is better than trained VGG16 with **94.3% confidence** (p-value = 0.0567). An even better result obtained the model *Sim_3072_MAX* with a conversion rate of **7.70%, which is 2.28% better than trained VGG16** and we can also conclude that Sim_3072_MAX is better than trained VGG16 with **99% confidence**. And non-trained VGG16 ended up last with the worst recall.

From the results, we can come to the conclusion that **Sim_3072_MAX is the best model**, and making recommendations based on **more than one image can increase the click-through rate**. We would point out that the model which achieved the highest recall (trained VGG16) is different from the

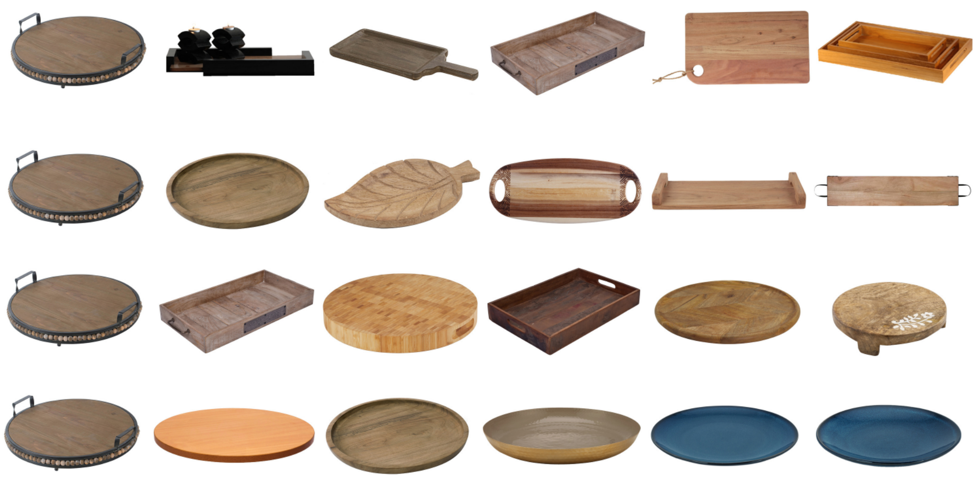

Figure 5.5: Showcase of recommendations (to the first image in each row). First row recommendations are based on Rag_1024_MAX, second row on Sim_3072_MAX, then trained VGG16 on One-to-one and not trained VGG16.

model that achieved the highest click-through rate (Sim_3072_MAX) in online A/B tests.

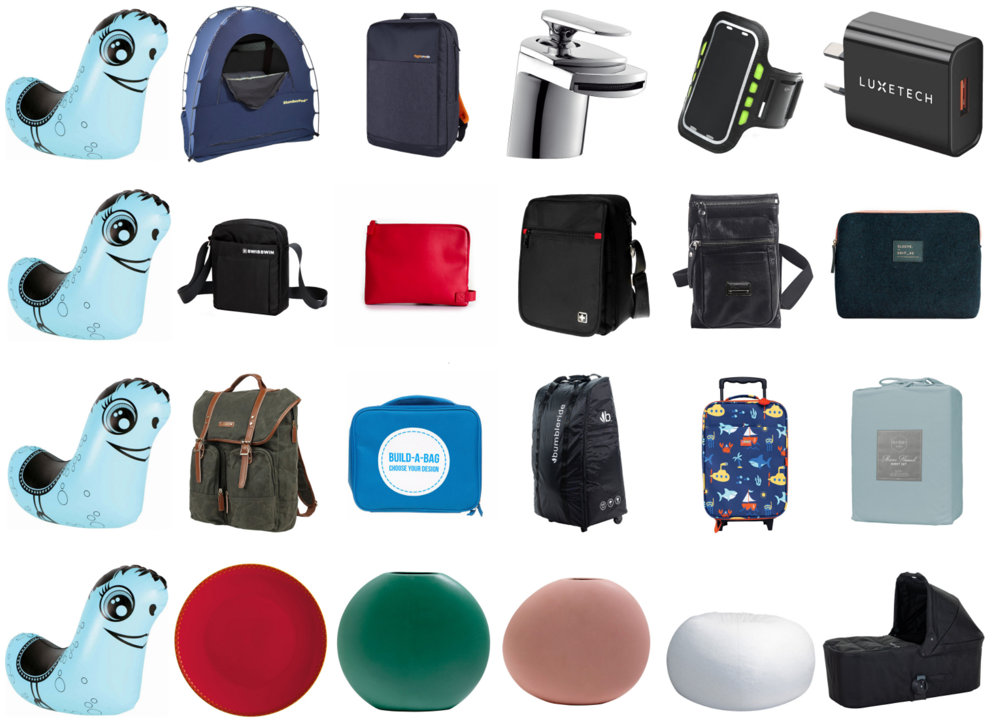

Figure 5.6: Showcase of recommendations (to the first image in each row). First row recommendations are based on Rag_1024_MAX, second row on Sim_3072_MAX, then trained VGG16 on One-to-one and not trained VGG16.

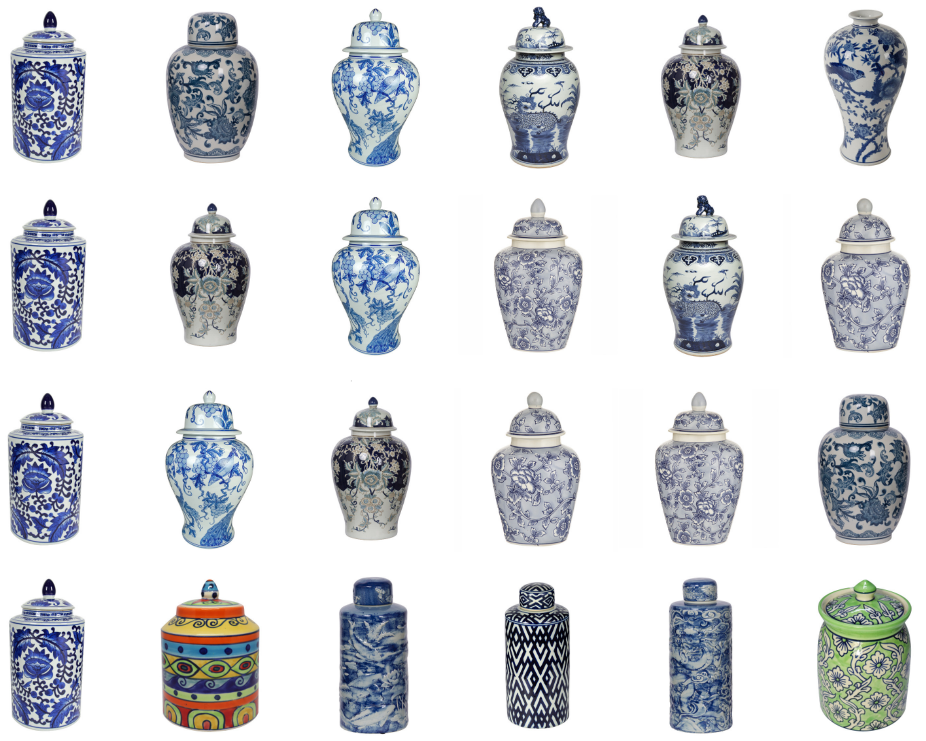

Figure 5.7: Showcase of recommendations (to the first image in each row). First row recommendations are based on Rag_1024_MAX, second row on Sim_3072_MAX, then trained VGG16 on One-to-one and not trained VGG16.

Conclusion

The goal of this thesis was to compare state-of-the-art convolutional neural networks in their ability to extract important features for recommendation, propose a method to incorporate interaction data into image data, and implement a prototype of such a recommender system which can produce recommendations based on more than one image of a single item. We presented several methods and experiments using one and more images from a single item, and we successfully enhanced the recall with a user's interaction data.

The experiments showed that if we do not have any interaction data, it is better to use a newer CNN model than VGG16, such as DenseNet201, which gives better recommendations. However, we can train the models on a collection of interactions and achieve an even higher score in most cases.

Interestingly, trained VGG16 on interactions got the highest score of all models in offline evaluation despite having the smallest accuracy score in the ImageNet dataset. This can be due to overfitting or too narrow focus of these other networks on the ImageNet dataset, and their representations of objects in the image may be too concrete. In other words, say EfficientNetV2L can exactly name the object in the image (therefore, achieving higher accuracy on ImageNet). However, such information is not enough to capture small features critical to the production of a good recommendation. On the other hand, VGG16 cannot distinguish so well between objects (i.e., smaller accuracy on ImageNet) and identify a more general feature representation of items that are more useful and critical for a better recommendation.

Moreover, we showed that using more than one image for training on classical CNNs with a nonflexible number of inputs does not result in improved recall, but rather the opposite.

Furthermore, we also implemented two versions of the MIL problem and trained our models on more than one image, which did not outperform VGG16 in offline tests. On the other hand, our two MIL implementations (namely Rag_1024_MAX and Sim_3072_MAX) in online tests achieved a higher score of click-through rate of 1.43% and 2.28%, and we can conclude that they are better than trained VGG16 with confidence levels of 94.3% and 99% compared to trained VGG16. Therefore, it was manifested that using more images of a single product leads to better recommendations and could be further researched and implemented in recommender systems.

In future work, we propose to implement MIL as one model, which can be trained end-to-end and see if it can further improve recall. Another option is to use a more advanced evaluation technique of the classic CNNs trained on multiple images. Furthermore, the last suggestion for future work is to use Visual Transformers for image recognition.

Bibliography

[1] Wang, L.; Chen, Z.; et al. An Opportunistic Routing for Data Forwarding Based on Vehicle Mobility Association in Vehicular Ad Hoc Networks. *Information*, volume 8, 11 2017: p. 140, doi:10.3390/info8040140.

[2] Calero Valdez, A.; Ziefle, M.; et al. Recommender Systems for Health Informatics: State-of-the-Art and Future Perspectives. *Lecture Notes in Computer Science*, volume 9605, 11 2016, doi:10.1007/978-3-319-50478-0_20.

[3] Hassan, H.; Negm, A.; et al. Assessment of artificial neural network for bathymetry estimation using high resolution satellite imagery in Shallow lakes: case study el Burullus lake. *International Water Technology Journal*, volume 5, 12 2015.

[4] Sharma, P. Basic introduction to convolutional neural network in Deep Learning. Mar 2022. Available from: `https://www.analyticsvidhya.com/blog/2022/03/basic-introduction-to-convolutional-neural-network-in-deep-learning/`

[5] Ferguson, M.; ak, R.; et al. Automatic localization of casting defects with convolutional neural networks. 12 2017, pp. 1726–1735, doi:10.1109/BigData.2017.8258115.

[6] Huaxiao, M. DenseNet implemented by tensorflow. Jun 2018. Available from: `https://mohuaxiao.github.io/2018/06/09/tensorflow/`

[7] Alhichri, H.; Alsuwayed, A.; et al. Classification of Remote Sensing Images Using EfficientNet-B3 CNN Model with Attention. *IEEE Access*, volume PP, 01 2021: pp. 1–1, doi:10.1109/ACCESS.2021.3051085.

[8] Ha, C. Convergence of SOTA CV models. Sep 2021. Available from: `https://medium.com/@hac541309/convergence-of-sota-cv-models-ad985a597173`

[9] Team, K. *Keras Documentation: Keras Applications.* Available from: https://keras.io/api/applications/

[10] Kumar, J.; Pillai, J.; et al. Document Image Classification and Labeling Using Multiple Instance Learning. 10 2011, pp. 1059 – 1063, doi:10.1109/ICDAR.2011.214.

[11] Pevný, T.; Somol, P. Using Neural Network Formalism to Solve Multiple-Instance Problems. In *Advances in Neural Networks - ISNN 2017*, edited by F. Cong; A. Leung; Q. Wei, Cham: Springer International Publishing, 2017, ISBN 978-3-319-59072-1, pp. 135–142.

[12] Team, K. Keras, efficientnet_v2.py at v2.8.0 Keras Team, Keras. Oct 2021. Available from: https://github.com/keras-team/keras/blob/v2.8.0/keras/applications/efficientnet_v2.py

[13] Team, K. *Keras Documentation: Resnet and RESNETV2.* Available from: https://keras.io/api/applications/resnet/

[14] Team, K. *Keras Documentation: NasNetLarge and NasNetMobile.* Available from: https://keras.io/api/applications/nasnet/

[15] Team, K. *Keras Documentation: Xception.* Available from: https://keras.io/api/applications/xception/

[16] Mahmood, T.; Ricci, F. Improving Recommender Systems with Adaptive Conversational Strategies. In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, HT '09, New York, NY, USA: Association for Computing Machinery, 2009, ISBN 9781605584867, p. 73–82, doi:10.1145/1557914.1557930. Available from: https://doi.org/10.1145/1557914.1557930

[17] Kim, B. M.; Li, Q.; et al. A new approach for combining content-based and collaborative filters. *Journal of Intelligent Information Systems*, volume 27, no. 1, 2006: pp. 79–91.

[18] Ricci, F.; Rokach, L.; et al. *Introduction to Recommender Systems Handbook.* Boston, MA: Springer US, 2011, ISBN 978-0-387-85820-3, pp. 1–35, doi:10.1007/978-0-387-85820-3_1. Available from: https://doi.org/10.1007/978-0-387-85820-3_1

[19] Holeňa, M.; Pulc, P.; et al. *Classification Methods for Internet Applications.* Springer, 2020.

[20] Albanese, M.; d'Acierno, A.; et al. A Multimedia Recommender System. *ACM Trans. Internet Technol.*, volume 13, no. 1, nov 2013, ISSN 1533-5399, doi:10.1145/2532640. Available from: https://doi.org/10.1145/2532640

[21] Hu, Y.; Koren, Y.; et al. Collaborative Filtering for Implicit Feedback Datasets. In *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 263–272, doi:10.1109/ICDM.2008.22.

[22] Brusilovski, P.; Kobsa, A.; et al. *The adaptive web: methods and strategies of web personalization*, volume 4321. Springer Science & Business Media, 2007.

[23] Geetha, G.; Safa, M.; et al. A Hybrid Approach using Collaborative filtering and Content based Filtering for Recommender System. *Journal of Physics: Conference Series*, volume 1000, apr 2018: p. 012101, doi:10.1088/1742-6596/1000/1/012101. Available from: `https://doi.org/10.1088/1742-6596/1000/1/012101`

[24] Trewin, S. Knowledge-based recommender systems. *Encyclopedia of library and information science*, volume 69, no. Supplement 32, 2000: p. 180.

[25] Chen, Q.; Li, W.; et al. Collaborative filtering algorithm based on item attribute and time weight. In *The 2016 International Conference on Automatic Control and Information Engineering*, 2016, pp. 12–15.

[26] Jawaheer, G.; Weller, P.; et al. Modeling User Preferences in Recommender Systems: A Classification Framework for Explicit and Implicit User Feedback. *ACM Trans. Interact. Intell. Syst.*, volume 4, no. 2, jun 2014, ISSN 2160-6455, doi:10.1145/2512208. Available from: `https://doi.org/10.1145/2512208`

[27] Jalili, M.; Ahmadian, S.; et al. Evaluating Collaborative Filtering Recommender Algorithms: A Survey. *IEEE Access*, volume 6, 2018: pp. 74003–74024, doi:10.1109/ACCESS.2018.2883742.

[28] Pelanek, R. Collaborative Filtering. 3 2022. Available from: `https://www.fi.muni.cz/~xpelanek/PV254/slides/collaborative-filtering.pdf`

[29] Yalcin, E.; Bilge, A. Investigating and counteracting popularity bias in group recommendations. *Information Processing & Management*, volume 58, no. 5, 2021: p. 102608, ISSN 0306-4573, doi: https://doi.org/10.1016/j.ipm.2021.102608. Available from: `https://www.sciencedirect.com/science/article/pii/S0306457321001047`

[30] Lika, B.; Kolomvatsos, K.; et al. Facing the cold start problem in recommender systems. *Expert Systems with Applications*, volume 41, no. 4, Part 2, 2014: pp. 2065–2073, ISSN 0957-4174, doi:https://doi.org/10.1016/j.eswa.2013.09.005. Available from: `https://www.sciencedirect.com/science/article/pii/S0957417413007240`

[31] Mehta, R.; Rana, K. A review on matrix factorization techniques in recommender systems. In *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, 2017, pp. 269–274, doi:10.1109/CSCITA.2017.8066567.

[32] Koren, Y.; Bell, R.; et al. Matrix Factorization Techniques for Recommender Systems. *Computer*, volume 42, no. 8, 2009: pp. 30–37, doi:10.1109/MC.2009.263.

[33] Chen, J.; Fang, J.; et al. Efficient and Portable ALS Matrix Factorization for Recommender Systems. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 409–418, doi:10.1109/IPDPSW.2017.91.

[34] Vozalis, M.; Margaritis, K. Applying SVD on item-based filtering. In *5th International Conference on Intelligent Systems Design and Applications (ISDA'05)*, 2005, pp. 464–469, doi:10.1109/ISDA.2005.25.

[35] Sarwar, B.; Karypis, G.; et al. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth international conference on computer and information science*, volume 1, Citeseer, 2002, pp. 27–8.

[36] Wang, D.; Liang, Y.; et al. A content-based recommender system for computer science publications. *Knowledge-Based Systems*, volume 157, 2018: pp. 1–9, ISSN 0950-7051, doi:https://doi.org/10.1016/j.knosys.2018.05.001. Available from: `https://www.sciencedirect.com/science/article/pii/S0950705118302107`

[37] Ullman, J. D. Data Mining. Available from: `http://i.stanford.edu/~ullman/mmds/ch1.pdf`

[38] Devlin, J.; Chang, M.-W.; et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018, doi:10.48550/ARXIV.1810.04805. Available from: `https://arxiv.org/abs/1810.04805`

[39] Pavlicek, M. Recommendation Models Based on Images. 2018.

[40] Desrosiers, C.; Karypis, G. *A Comprehensive Survey of Neighborhood-Based Recommendation Methods*. 01 2011, pp. 107–144, doi:10.1007/978-0-387-85820-3_4.

[41] Dommeti, R. Neighborhood based methods for collaborative filtering. *A Case Study, I*, 2009: pp. 1–5.

[42] Su, X.; Khoshgoftaar, T. M. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, volume 2009, 2009.

[43] Melville, P.; Sindhwani, V. Recommender systems. *Encyclopedia of machine learning*, volume 1, 2010: pp. 829–838.

[44] Sarwar, B.; Karypis, G.; et al. Item-Based Collaborative Filtering Recommendation Algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, New York, NY, USA: Association for Computing Machinery, 2001, ISBN 1581133480, p. 285–295, doi:10.1145/371920.372071. Available from: `https://doi.org/10.1145/371920.372071`

[45] Prabhakaran, S. Cosine similarity - understanding the math and how it works (with python codes). Oct 2018. Available from: `https://www.machinelearningplus.com/nlp/cosine-similarity/`

[46] Hernández del Olmo, F.; Gaudioso, E. Evaluation of recommender systems: A new approach. *Expert Systems with Applications*, volume 35, no. 3, 2008: pp. 790–804, ISSN 0957-4174, doi:https://doi.org/10.1016/j.eswa.2007.07.047. Available from: `https://www.sciencedirect.com/science/article/pii/S0957417407002928`

[47] Arguello, J. Evaluation Metrics. 3 2013. Available from: `https://ils.unc.edu/courses/2013_spring/inls509_001/lectures/10-EvaluationMetrics.pdf`

[48] Herlocker, J. L.; Konstan, J. A.; et al. Evaluating Collaborative Filtering Recommender Systems. *ACM Trans. Inf. Syst.*, volume 22, no. 1, jan 2004: p. 5–53, ISSN 1046-8188, doi:10.1145/963770.963772. Available from: `https://doi.org/10.1145/963770.963772`

[49] Malaeb, M. Recall and Precision at k for Recommender Systems. 8 2017. Available from: `https://medium.com/@m_n_malaeb/recall-and-precision-at-k-for-recommender-systems-618483226c54`

[50] Powers, D. M. W. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Technical report, School of Informatics and Engineering, 2007. Available from: `https://web.archive.org/web/20191114213255/https://www.flinders.edu.au/science_engineering/fms/School-CSEM/publications/tech_reps-research_artfcts/TRRA_2007.pdf`

[51] Sammut, C.; Webb, G. I. (editors). *Leave-One-Out Cross-Validation.* Boston, MA: Springer US, 2010, ISBN 978-0-387-30164-8, pp. 600–601, doi:10.1007/978-0-387-30164-8_469. Available from: `https://doi.org/10.1007/978-0-387-30164-8_469`

[52] Basheer, I.; Hajmeer, M. Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods*, volume 43, no. 1, 2000: pp. 3–31, ISSN 0167-7012, doi:https:

//doi.org/10.1016/S0167-7012(00)00201-3, neural Computting in Micrbiology. Available from: https://www.sciencedirect.com/science/article/pii/S0167701200002013

[53] Brain, I. T.; Rosenblatt, F. The perceptron: a probabilisticmodel for information storage and organization.

[54] Liu, W.; Wang, Z.; et al. A survey of deep neural network architectures and their applications. *Neurocomputing*, volume 234, 2017: pp. 11–26, ISSN 0925-2312, doi:https://doi.org/10.1016/j.neucom.2016.12.038. Available from: https://www.sciencedirect.com/science/article/pii/S0925231216315533

[55] GeeksforGeeks. 0-1 Knapsack problem: DP-10. Mar 2022. Available from: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

[56] Agatonovic-Kustrin, S.; Beresford, R. Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research. *Journal of Pharmaceutical and Biomedical Analysis*, volume 22, no. 5, 2000: pp. 717–727, ISSN 0731-7085, doi:https://doi.org/10.1016/S0731-7085(99)00272-1. Available from: https://www.sciencedirect.com/science/article/pii/S0731708599002721

[57] Begum, A.; Fatima, F.; et al. Implementation of Deep Learning Algorithm with Perceptron using TenzorFlow Library. In *2019 International Conference on Communication and Signal Processing (ICCSP)*, 2019, pp. 0172–0175, doi:10.1109/ICCSP.2019.8697910.

[58] Noriega, L. Multilayer perceptron tutorial. *School of Computing. Staffordshire University*, 2005.

[59] Balaji, S. A.; Baskaran, K. Design and Development of Artificial Neural Networking (ANN) system using sigmoid activation function to predict annual rice production in Tamilnadu. 2013, doi:10.48550/ARXIV.1303.1913. Available from: https://arxiv.org/abs/1303.1913

[60] Baheti, P. 12 types of neural networks activation functions: How to choose? Apr 2022. Available from: https://www.v7labs.com/blog/neural-networks-activation-functions

[61] Karlik, B.; Olgac, A. V. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, volume 1, no. 4, 2011: pp. 111–122.

[62] Da Silva, I. N.; Spatti, D. H.; et al. Artificial neural networks. *Cham: Springer International Publishing*, volume 39, 2017.

[63] Rojas, R. Perceptron learning. In *Neural Networks*, Springer, 1996, pp. 77–98.

[64] Feldman, J.; Rojas, R. *Neural Networks: A Systematic Introduction*. Springer Berlin Heidelberg, 2013, ISBN 9783642610684, 77-98 pp. Available from: `https://books.google.cz/books?id=4rESBwAAQBAJ`

[65] Bock, S.; Weiß, M. A Proof of Local Convergence for the Adam Optimizer. In *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8, doi:10.1109/IJCNN.2019.8852239.

[66] Kingma, D. P.; Ba, J. Adam: A Method for Stochastic Optimization. 2014, doi:10.48550/ARXIV.1412.6980. Available from: `https://arxiv.org/abs/1412.6980`

[67] Khapra, M. M. Deep Learning(CS7015): Lec 5.9 (Part-2) Bias Correction in Adam. 2002. Available from: `https://www.youtube.com/watch?v=-0ZMU-gnm2g`

[68] Russakovsky, O.; Deng, J.; et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, volume 115, no. 3, 2015: pp. 211–252, doi:10.1007/s11263-015-0816-y.

[69] O'Shea, K.; Nash, R. An Introduction to Convolutional Neural Networks. 2015, doi:10.48550/ARXIV.1511.08458. Available from: `https://arxiv.org/abs/1511.08458`

[70] Albawi, S.; Mohammed, T. A.; et al. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6, doi:10.1109/ICEngTechnol.2017.8308186.

[71] Marais, W.; Holz, R.; et al. Leveraging spatial textures, through machine learning, to identify aerosols and distinct cloud types from multispectral observations. *Atmospheric Measurement Techniques*, volume 13, 10 2020: pp. 5459–5480, doi:10.5194/amt-13-5459-2020.

[72] Singh, P.; Raj, P.; et al. EDS pooling layer. *Image and Vision Computing*, volume 98, 2020: p. 103923, ISSN 0262-8856, doi: https://doi.org/10.1016/j.imavis.2020.103923. Available from: `https://www.sciencedirect.com/science/article/pii/S026288562030055X`

[73] Srivastava, N. Improving neural networks with dropout. *University of Toronto*, volume 182, no. 566, 2013.

[74] Zhang, Y.; Robertson, J.; et al. All-optical neuromorphic binary convolution with a spiking VCSEL neuron for image gradient magnitudes. *Photon. Res.*, volume 9, no. 5, May 2021: pp. B201–B209,

doi:10.1364/PRJ.412141. Available from: http://opg.optica.org/prj/abstract.cfm?URI=prj-9-5-B201

[75] Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2014, doi:10.48550/ARXIV.1409.1556. Available from: https://arxiv.org/abs/1409.1556

[76] Huang, G.; Liu, Z.; et al. Densely Connected Convolutional Networks. 2016, doi:10.48550/ARXIV.1608.06993. Available from: https://arxiv.org/abs/1608.06993

[77] Abadi, M.; Agarwal, A.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015, software available from tensorflow.org. Available from: https://www.tensorflow.org/api_docs/python/tf/keras/applications/densenet/DenseNet121

[78] Tan, M.; Le, Q. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, volume 97, edited by K. Chaudhuri; R. Salakhutdinov, PMLR, 09–15 Jun 2019, pp. 6105–6114. Available from: https://proceedings.mlr.press/v97/tan19a.html

[79] Sandler, M.; Howard, A.; et al. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[80] Milan, S. Convolutional Neural Network II. Available from: https://lectures.ms.mff.cuni.cz/video/rec/npfl114/2122/npfl114-05-czech.mp4

[81] Tan, M.; Le, Q. EfficientNetV2: Smaller Models and Faster Training. In *Proceedings of the 38th International Conference on Machine Learning*, *Proceedings of Machine Learning Research*, volume 139, edited by M. Meila; T. Zhang, PMLR, 18–24 Jul 2021, pp. 10096–10106. Available from: https://proceedings.mlr.press/v139/tan21a.html

[82] Ibrahim, M. Google releases EfficientNetV2 - a smaller, faster, and better EfficientNet. Apr 2021. Available from: https://towardsdatascience.com/google-releases-efficientnetv2-a-smaller-faster-and-better-efficientnet-673a77bdd43c

[83] Zhang, C.; Benz, P.; et al. ResNet or DenseNet? Introducing Dense Shortcuts to ResNet. 2020, doi:10.48550/ARXIV.2010.12496. Available from: https://arxiv.org/abs/2010.12496

[84] He, K.; Zhang, X.; et al. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[85] Cloud, G. Advanced guide to inception V3 — cloud TPU — google cloud. Available from: `https://cloud.google.com/tpu/docs/inception-v3-advanced`

[86] Szegedy, C.; Vanhoucke, V.; et al. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[87] Szegedy, C.; Ioffe, S.; et al. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[88] Zoph, B.; Vasudevan, V.; et al. Learning Transferable Architectures for Scalable Image Recognition. 2017, doi:10.48550/ARXIV.1707.07012. Available from: `https://arxiv.org/abs/1707.07012`

[89] Chollet, F. Xception: Deep Learning With Depthwise Separable Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[90] Yang, C.; Lozano-Perez, T. Image database retrieval with multiple-instance learning techniques. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, 2000, pp. 233–243, doi:10.1109/ICDE.2000.839416.

[91] Maron, O.; Lozano-Pérez, T. A Framework for Multiple-Instance Learning. In *Advances in Neural Information Processing Systems*, volume 10, edited by M. Jordan; M. Kearns; S. Solla, MIT Press, 1997. Available from: `https://proceedings.neurips.cc/paper/1997/file/82965d4ed8150294d4330ace00821d77-Paper.pdf`

[92] Zhou, Z.-H.; Zhang, M.-L. Neural networks for multi-instance learning. In *Proceedings of the International Conference on Intelligent Information Technology, Beijing, China*, 2002, pp. 455–459.

[93] Cremonesi, P.; Koren, Y.; et al. Performance of recommender algorithms on top-N recommendation tasks. 01 2010, pp. 39–46, doi:10.1145/1864708.1864721.

[94] Martinek, L. Recommendations Model Based on Recurrent Neural Networks. 2020.

[95] Vančura, V. Neural Basket Embedding for Sequential Recommendation. In *Fifteenth ACM Conference on Recommender Systems*, 2021, pp. 878–883.

Appendix **A**

Acronyms

AdaGrad Adaptive Gradient Algorithm

Adam Adaptive Moment Estimation

ANN Artifical neural network

CB Content-based

CF Collaborative filtering

COS Cosine similarity

CNN Convolutional neural network

CTR Click-through rate

MIL Multiple Instance Learning

MLP Multilayer perceptron

MSE Mean Squared Error

ReLu Rectified Linear Unit

RMSProp Root Mean Square Propagation

RNN Recurrent neural network

RS Recommender System

Appendix B

Contents of enclosed CD

```
readme.txt ....................... the file with CD contents description
results.........................results of our experiments in text form
src.....................................the directory of source codes
  implementation ........................... implementation sources
  thesis..............the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
  DP_Kader_Agha_Kamil_2022.pdf ....... the thesis text in PDF format
```