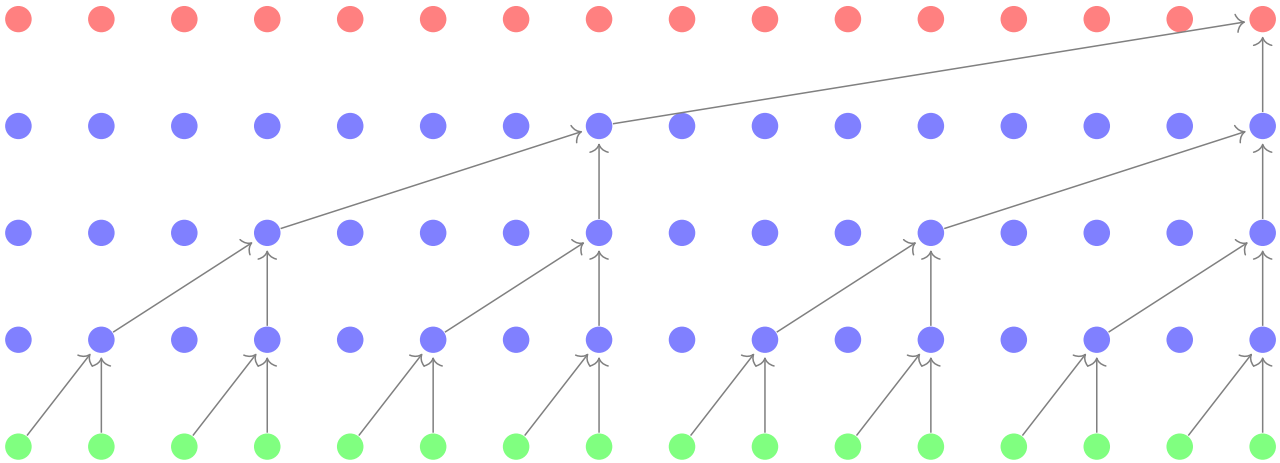


Convolutional Neural Networks for Sequence-Aware Recommender Systems

Master's thesis in Computer Science

Tim Kerschbaumer

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Convolutional Neural Networks for Sequence-Aware Recommender Systems

Tim Kerschbaumer

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Convolutional Neural Networks for
Sequence-Aware Recommender Systems
Tim Kerschbaumer

Supervisor: Aristide Tossou, Department of Computer Science and Engineering
Advisor: Max Berggren, Sellpy
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of a convolutional neural network using causal dilated convolutions, adapted from [1]. For more details, see section 2.3.4.

Typeset in LaTeX
Gothenburg, Sweden 2018

Convolutional Neural Networks for Sequence-Aware Recommender Systems
Tim Kerschbaumer
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Recommender systems are prominent components of many of today's web applications. Historically, the most successful recommender systems have been based on a matrix completion formulation. However, in some domains having sequence-aware recommender systems, i.e systems that take data's sequential nature into account, may be beneficial to capture user's short-term interests as well as long-term sequential patterns. The most successful methods for sequence-aware recommender systems have been based on recurrent neural networks. Recurrent neural networks, however, are often hard to train and suffer from several disadvantages in regard to speed and memory requirements. Several recent papers have suggested that convolutional neural networks can be used to process sequential data more efficiently and sometimes with better results than recurrent networks. In this thesis, we propose the use of convolutional neural networks for the task of sequence-aware recommendations. We present a two-stage deep learning approach to recommendations, where convolutional neural networks are used for sequence-aware candidate generation. Our results show that convolutional neural networks can achieve predictive performance comparable to state-of-the-art for sequence-aware recommendation tasks.

Keywords: Computer science, deep learning, machine learning, neural networks, recommender systems, sequence-aware, convolutional neural networks

Acknowledgements

First, I would like to thank my supervisor at Chalmers University of Technology, Aristide Tossou, whose support and feedback have been invaluable throughout the writing of this thesis.

Next, I would like to thank everyone at Sellpy for giving me the opportunity to write my thesis there. Especially I would like to thank my advisor at Sellpy, Max Berggren, whose ideas and contributions have been very helpful.

Finally, I would like to thank my examiner, Devdatt Dubhashi, for his feedback.

Tim Kerschbaumer, Gothenburg, August 2018

Contents

List of Figures

List of Tables

1
Introduction

This chapter introduces the topic of recommender systems as well as the background, problem statement, and delimitations of the thesis.

1.1 Background

Recommender systems are systems that suggest items that may be of interest to users. Such systems are prominent components of many of today's modern web applications, for example in e-commerce to personalise visitors' user experience by providing recommendations of items the customer will most likely buy. Recommender systems are and have been an important research area since the appearance of the first papers on collaborative filtering in the 1990s. Over the last two decades, there has been much work in both industry and academia to develop new approaches to recommender systems. The interest still remains high because it constitutes a problem-rich research area and because of the abundance of practical applications that help users deal with information overload and provide personalised recommendations, content, and services to them [2].

Historical data used to build recommender systems can be of two kinds, either *explicit feedback*, where users give items explicit scores that can be either positive or negative, or *implicit feedback*, where user behaviours such as clicks or dwell times are used as data. Historically, the vast majority of literature has focused on recommender systems for explicit feedback [3], although several methods to adapt explicit feedback systems to handle implicit data have been proposed, for instance [3] [4] [5]. Academic research on both explicit and implicit feedback recommender systems has historically often been based on a matrix completion problem formulation where each interaction, i.e user-item pair, is considered separately [6]. However, in many application domains, user-item interactions can be recorded over time. In analogy with [6], recommender systems that use user-item interactions recorded over time are called *sequence-aware recommender systems*. A number of recent works have shown that taking sequential data into account can yield richer recommendations and discover additional behavioural patterns [6]. Sequential interaction logs often contain useful information on both short-term user interests as well as long-term sequential patterns that can be central to the success of a recommender system [6]. To preserve data's sequential nature, approaches other than matrix completion must be used. Some of the most successful approaches for sequence-aware recommendations are based on recurrent neural networks, first introduced in the domain of recommendations by [7], and later improved on by [8].

Deep learning, a class of machine learning algorithms and techniques, has in recent years revolutionised several artificial intelligence tasks such as image analysis and natural language processing. Applying these techniques to recommender systems has also become an increasingly popular research area with no sign of stagnating [9]. Many of deep learning's successes have been based on two types of neural networks, *recurrent neural networks* and *convolutional neural networks*. Recurrent networks have been most successful in sequence modelling tasks, for example in natural language processing, while convolutional networks have been very successful in computer vision tasks, such as image classification. However, in recent years, specialised convolutional networks have been shown to also be successful and efficient in sequence modelling tasks [1] [10]. In some domains, convolutional networks have even exceeded the predictive performance of recurrent networks [11]. Apart from predictive performance, since convolutional networks do not have recurrent connections they allow more parallelisation and are thus often faster than recurrent networks, especially when input sequences are long [1].

1.2 Problem statement

Sellhelp AB (from this point forward referred to as *Sellpy*) is a company that re-sells items for people who have things to sell but would like to avoid the hassle of listing items, interacting with buyers, handle payments and delivery. Items that customers wish to sell are picked up by Sellpy at the customers' doorstep. Sellpy then handles the process from describing and photographing the items, to listing, selling, and shipping. All the customer has to do is order a bag and fill it with items. The items are listed on Sellpy's e-commerce platform, *Sellpy market*, where customers can browse for and buy items. With a growing number of items, navigating the product catalogue as a buyer is getting increasingly difficult. A recommender system displaying items that may be of interest to users as they browse Sellpy market would thus be beneficial both for the users and for the company by increasing sales.

Browsing on an e-commerce site, such as Sellpy market, is naturally a sequential task, therefore using a matrix completion formulation would neglect the sequential nature of the data. As user tastes on e-commerce sites can be both short-term, for example looking for a specific type of item during the current session, and long-term, the general taste of a user, using a sequence-aware recommender system could potentially increase capturing the short-term taste of a user. Using sequential user interactions is especially important when there are many new or anonymous users as no long-term historical data about the general tastes of these users is available.

Although convolutional neural networks have shown good results in sequence modelling tasks in other domains, investigating their usage in sequence-aware recommender systems remains an open topic of research. The main research question of this thesis is thus formulated as:

How can convolutional neural networks be used in sequence-aware recommender systems?

A natural sub-question is how such a system performs in comparison with other

techniques, so the first sub research question is:

How does the predictive performance of sequence-aware recommender systems based on convolutional neural networks differ from other sequence-aware models?

Further, since production recommender systems are rarely based on a single source of data, investigating how sequence-aware recommender systems can take additional sources of data into account is thus the second sub research question:

How can sequence-aware recommender systems account for more information than user-item interactions, such as content features and other sources of information?

1.3 Delimitations

The main research question emphasises the investigation of recommender systems that are sequence-aware. As most sequential data is on the form of implicit feedback, this thesis is limited from exploring recommender system for explicit feedback.

Recommender systems can act in several domains, such as news recommendations or friend recommendations. In this thesis, the domain is limited to items. Item recommender systems are common within e-commerce, where the goal of a system is to recommend items or products, to users browsing a large product catalogue.

1.4 Approach

To answer the research questions, a system that produces recommendations in two stages is presented. The first stage uses a convolutional neural network to make sequence-aware predictions of the next item a user will interact with, based on historical item interactions. The top-n items a user is most likely to interact with is then used as input in a second stage. In the second stage, a ranking neural network is used to find the best ordering of a user's top-n items. The ranking network incorporates item content information and other data sources on long-term user preferences to distinguish between the top-n items.

1.5 Contribution

The main contribution of this thesis is showing that convolutional neural networks can be successfully applied as the underlying model in sequence-aware recommender systems. The presented convolutional neural network outperforms several baselines as well as the best recurrent neural network proposed in [7], whose general architecture has been used to reach state-of-the-art results for session-based recommendations.

Further, the thesis shows how sequence-aware models can be used as candidate generation components in a two-stage approach. This two-stage approach can make use of convolutional neural networks or other sequence-aware models to account for collaborative information in a hybrid recommender system.

1.6 Outline

In the following chapter, a theoretical introduction to recommender systems and machine learning models later used in our approach or as baselines are presented. The theory chapter ends with a section on related work in the field.

Using concepts from the theory chapter, the next chapter introduces a sequence-aware two-stage recommender system. Beginning with a general overview of the two-stage approach to recommendations, the chapter continues by presenting the first stage of the recommender system, candidate generation based on a convolutional neural network. Finally, the second stage of the recommender system, a ranking network is presented.

The chapter on experimental setup introduces the Sellpy market dataset, benchmark datasets, baseline models, and evaluation metrics. Finally, details about the implementation are presented.

In the results chapter, experimental results on our model are introduced and compared to baseline results.

In the final chapters, discussion and conclusion, our findings are discussed and conclusions about our approach are drawn. The thesis is ended by discussing potential future work.

2

Theory

The theory chapter begins by presenting basic recommender system goals and types. Next, machine learning models used to achieve these goals, later used in our approach or as baseline models, are presented. Finally, related work in the field of recommender systems in general and deep recommender models in particular is presented.

2.1 Recommender Systems

Practical applications of recommender system vary from, for example, friend recommendations on Facebook to advertisement placement on Google search [12]. Although there are many practical applications, recommender systems are often based on the same fundamental methods.

The problem a recommender system tries to solve can be of two primary kinds [12]. The goal of the first, known as a *prediction problem*, is to predict a rating for a user-item pair or the next item a user will interact with given his history. However, in some domains, predicting the item ratings of users or predicting the most likely next item is not necessary to make recommendations. Instead, it makes more sense to view the problem as a *ranking problem* where recommending the top-k items for a specific user is considered.

Recommender systems are often designed for a specific type of data. Two general classes of data categorise recommender systems. *Explicit feedback* is data where users have given explicit ratings to items, both positive and negative. Not all domains enjoy this convenient setting and there has been much research in adapting recommender systems for domains where feedback is *implicit*. In such settings, user behaviour is observed and implicit feedback is created indirectly. Implicit feedback can be, for example, purchase history, browsing history, search patterns or mouse movements [3]. Typically, it is easier to build well-performing recommender systems for the more expressive explicit feedback but the data collection process is typically much more expensive, whereas log data, from which implicit feedback can be extracted, is one of the most ubiquitous forms of data available.

In the coming subsections, some commonly used recommender system methods are presented. These methods are general and variations of them can be applied to both prediction and ranking problems as well as with explicit or implicit feedback.

2.1.1 Collaborative filtering

One of the most successful types of recommender systems is collaborative filtering. The idea behind collaborative filtering is to identify pairs of items that tend to be rated or interacted with similarly, or like-minded users with a similar history of interacting, to deduce unknown relationships between users and items. A common way to concertise collaborative filtering problems is to use a matrix completion formulation. Assuming that data indicating user preferences for a subset of items is available, for m users and n items, the problem can be viewed as an incomplete $m \times n$ matrix R, where observed values are used to infer the unobserved values [12].

There are two main types of collaborative filtering, *user-based* and *item-based*. In user-based collaborative filtering similarities between users are considered. For each user, historical interactions are used to find like-minded users and recommend ratings or interactions by computing some weighted average of the interactions or ratings in this group. Similar users are found by computing a similarity function over rows in the matrix R [12]. In contrast, the idea in item-based collaborative filtering is to consider similarities between items. To make a prediction of a rating or likelihood of interaction of some item I for some user, a set of items similar to I is computed and used to do inference. Similar items are found by computing a similarity function over columns in the matrix R [12].

2.1.2 Content-based filtering

Differently from collaborative filtering that uses correlations in user patterns, content-based filtering uses item and/or user attributes to make predictions. The basic idea is that if a user likes an item with certain attributes, there is a good chance he may like items with similar attributes. The main components of a content-based recommender system can be grouped as follows [13]:

- **Preprocessing and feature extraction**: As content-based filtering can operate in a variety of domains, item attributes can take many forms. Often features must be extracted from raw data and transformed to vector-space representations before learning can take place.
- **Content-based learning**: A content-based model is specific to a certain user, thus a model based on previous user interactions and item features is learnt to predict user interest in other items. The resulting model relates user feedback to item features.
- **Recommendations**: At this stage, the learnt model is given an input of a user and an item and predicts, for example, a rating or if an item is selected.

2.1.3 Hybrid recommender systems

Both collaborative and content-based methods have some drawbacks. Collaborative filtering often suffers from cold-start problems, an unrated item can never be recommended to a user since it has no interactions. In contrast, content-based filtering often suffers from poor diversity in recommendations, a user may only be recommended items in the same genre as items he has previously rated highly or interacted

much with. For these reasons, production recommender systems are seldom based on a single approach.

Hybrid recommender systems combine two or more different recommender methods to gain better performance by overcoming some of the individual method's limitations. Hybrid systems can be classified into different categories as follows [14]:

- **Weighted**: The scores of a recommended item is computed from several different recommender systems.
- **Switching**: The system switches between recommendation techniques with some criterion.
- **Mixed**: In mixed recommender systems, recommendations from several different systems are presented at the same time.
- **Feature combination**: Features from different data sources are combined and used in one recommender system.
- **Cascade**: In a cascade hybrid, the output of one recommender system is the input to another. The second method refines the results of the first.
- **Feature augmentation**: The output of one recommender system is used to create input features for another one.
- **Meta level**: In a meta-level hybrid, the model generated by one recommender system is used as input to another. This differs from feature augmentation by considering models instead of features.

2.2 Latent factor models

Latent factor models are a set of models that relate observable variables to latent variables. The most successful latent factor models within recommender systems are *matrix factorisation techniques* [15]. Differently from the similarity based collaborative filtering models, called *neighbourhood models*, presented in section 2.1.1, matrix factorisation takes a learning approach to collaborative filtering. Although the neighbourhood models have several advantages related to their simplicity, they are often impractical in large-scale settings [16]. Matrix factorisation uses optimisation to train a machine learning model, the model is then used to predict missing values in a matrix of interactions or ratings [12].

As there are many algorithms for doing matrix factorisation, the focus here is on basic techniques, first for explicit feedback before the model is adapted to also handle implicit feedback.

2.2.1 Explicit feedback matrix factorisation

In matrix factorisation for explicit feedback, m users and n items are described by an $m \times n$ ratings matrix R. The matrix contains ratings for a subset of user-item pairs r_{ij}, for example $r_{ij} \in [1, 5]$. R is factorised approximately into an $m \times k$ matrix U and an $n \times k$ matrix V as

$$R \approx UV^T. \tag{2.1}$$

Each column in U and V is referred to as a latent component, whereas each row in either matrix is referred to as a latent factor. The number of latent components,

k, is a hyperparameter that needs to be determined by the practitioner. Row i in matrix U is the k-dimensional latent factor describing user i whereas row j in matrix V is the k-dimensional latent factor describing item j. A latent factor is often not semantically interpretable, however it does represent a correlation pattern in the ratings matrix [17]. As the matrix R is incomplete, the objective function cannot be directly defined but must be formulated in terms of the observed entries. For a set of observed user-item pairs $S = \{(i, j) : r_{ij} \text{ observed}\}$, the following unconstrained optimisation problem can be defined

$$\min \frac{1}{2} \sum_{(i,j)\in S} (r_{ij} - \sum_{s=1}^{k} u_{is} v_{js})^2. \tag{2.2}$$

Note that the objective function is defined only over observed values, but the entire matrix can be reconstructed as UV^T. Minimising the sum of squared errors objective function in equation 2.2 with respect to the unknowns u_{is} and v_{js} can be achieved with standard gradient descent methods [17].

2.2.2 Implicit feedback matrix factorisation

For explicit feedback it natural to see the unknown values as missing and exclude them from optimisation. For implicit feedback however, if the unknown values are treated as missing there is only positive feedback in the matrix. Following the work presented in [3], another matrix factorisation model is presented to handle this issue.

Following the notation from the previous section 2.2.1, let $r_{ij} \in [0, k]$ where 0 indicates no interaction and higher values indicate more or stronger interactions. Let a set of binary variables over user-item pairs ij be defined as

$$p_{ij} = \begin{cases} 1, & \text{if } r_{ij} > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

In other words, if a user interacts with an item then that is an indication that the user likes that item and $p_{ij} = 1$. If a user does not interact with an item, then no preference is believed and $p_{ij} = 0$. These preferences are associated with a confidence value. For instance, setting $p_{ij} = 0$ should have a low confidence value since it is unknown whether a user does not like an item or simply has not observed it. In general, for larger r_{ij}, the confidence should be higher. A possible choice for the confidence of a user-item pair is

$$c_{ij} = 1 + \alpha r_{ij} \tag{2.4}$$

where the rate of increase, α, is a hyperparameter.

Similar to explicit feedback matrix factorisation, the goal is to find the latent matrices U and V that factorise user preferences. The preference of a user-item pair is thus assumed to be the product of a latent user vector $\mathbf{u}_i$ and a latent item vector $\mathbf{v}_j$ such that

$$p_{ij} = \mathbf{u}_i^T \mathbf{v}_j = \sum_{s=1}^{k} u_{is} v_{js}. \tag{2.5}$$

The optimisation objective function can now be formulated as

$$\min \frac{1}{2} \sum_{(i,j) \in U \times V} c_{ij}(p_{ij} - \sum_{s=1}^{k} u_{is} v_{js})^2. \tag{2.6}$$

The objective function has several similarities with the explicit feedback matrix factorisation objective function in equation 2.2, with two important differences. First, varying confidence levels are accounted for by multiplying with c_{ij}. Second, optimisation is over all user-item pairs rater than just the observed interactions.

Optimising over all user-item pairs leads to a huge number of terms which prevents most direct optimisation techniques such as stochastic gradient descent, used for many explicit feedback datasets. Thus other optimisation techniques, such as the alternating least squares method presented in [3], must be used.

2.3 Neural networks

Neural networks are dynamic machine learning models that come in various forms and flavours. Because of their dynamic nature, these models can be constructed to solve a wide variety of tasks, including collaborative or content-based recommendations. In this section, we first give an introduction to the simplest possible neural network, a linear model. We then describe the quintessential model known as the *feedforward neural network* before we continue to present *recurrent neural networks*, *convolutional neural networks*, and more specialised operations.

2.3.1 Linear models

The linear model is one of the simpler machine learning models that can be applied to a variety of tasks in both classification, predicting one out of k classes, and regression, predicting real-valued outputs. Given one input sample and its vector of input features, i.e values describing it, $\mathbf{x} = (x_1, x_2, ..., x_m)$, the true output y is approximated via the model

$$\hat{y} = w_0 + \sum_{j=1}^{m} x_j w_j \tag{2.7}$$

or equivalently in matrix form, where the bias term w_0 is included in $\mathbf{w}$ and a constant 1 is included in $\mathbf{x}$, commonly known as *the bias trick* [18]

$$\hat{y} = \mathbf{x}^T \mathbf{w}. \tag{2.8}$$

This dot product can be seen as a simple neural network with inputs $\mathbf{x} = (x_1, x_2, x_3, x_4)$ in figure 2.1. As seen in the figure, each input is connected to the sole output $\hat{y}$ which is computed with equation 2.8 or equivalently 2.7 with $m = 4$.

The goal of learning is to set the parameters $\mathbf{w}$, known as the *weights*, in an optimal way given a collection of input-output pairs $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)$ known as the training data. To find optimal weights $\mathbf{w}$, some performance measure that describes how well the model is estimating the data must be used. This measure is

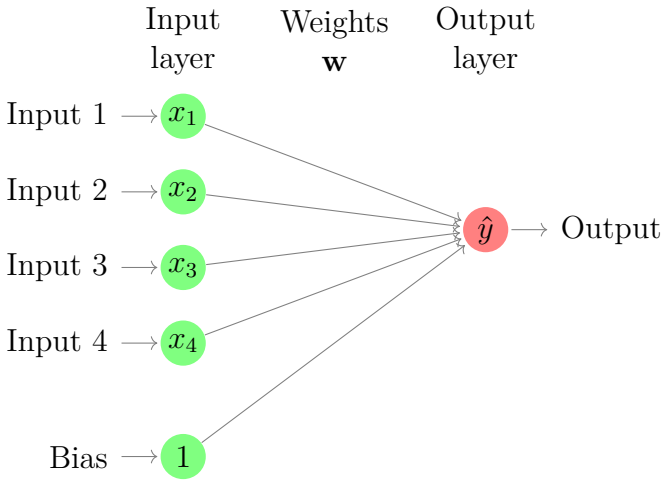


Figure 2.1: Linear neural network with 4 inputs, a bias, and 1 output unit.

usually called the *loss function* or *objective function*. One common loss function for regression tasks is the mean squared error loss, defined over all samples as

$$RSS(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) \tag{2.9}$$

where $\mathbf{y} \in \mathbb{R}^n$ is the vector of true outputs, i.e targets, and $\mathbf{X} \in \mathbb{R}^{n \times m}$ is the matrix of m-dimensional inputs in the training set. The problem now becomes an optimisation problem where the task is to find the weights that minimise the loss function. In this case, the normal equations which can be solved analytically are obtained by differentiating the loss w.r.t. $\mathbf{w}$ [19]

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\mathbf{w}) = 0 \implies \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{2.10}$$

2.3.2 Feedforward neural networks

The linear model presented in the previous section 2.3.1 defines the simplest neural network architecture that can be constructed. Problematically, the linear model can only represent linear dependencies between inputs and outputs. To represent non-linear dependencies, basis functions $\phi(\mathbf{x})$ that do non-linear transformations of inputs can be used. The transformed inputs are then used as inputs to a linear model, thus retaining the linearity with respect to the weights $\mathbf{w}$. Although this approach allows the learning algorithm to learn non-linear relationships, the question of how to choose the transformation ϕ arises. Prior to deep neural networks, manually engineering ϕ was a popular approach [20]. Another approach is to choose a very generic ϕ, for example an infinite dimensional mapping, such as *radial basis functions*. The strategy used by neural networks is instead to learn the transformation [20].

The term *network* in feedforward neural network gets its name from the chaining of many different functions [20]. A network is formed from the fact that the input to next function is the output from the previous one. For example, a chain of 3 *layers* forming a network are defined as $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$. The final layer f_3 is referred to as the *output layer* and the others, f_2 and f_1, as *hidden layers*. A

feedforward neural network with one hidden layer predicts outputs via the following computation

$$\hat{y} = \phi(\mathbf{x}, \mathbf{W})^T \mathbf{w}. \tag{2.11}$$

There are two weight matrices in this network, $\mathbf{W}$ which is used to learn ϕ and $\mathbf{w}$ that maps $\phi(\mathbf{x}, \mathbf{W})$ to the desired output. Here, ϕ is defined as a hidden layer. This architecture with 4 inputs and 5 hidden units can be seen in figure 2.2. The model can easily be extended to have more hidden layers by further chaining more functions of $\mathbf{x}$, the more of these functions the *deeper* the network becomes.

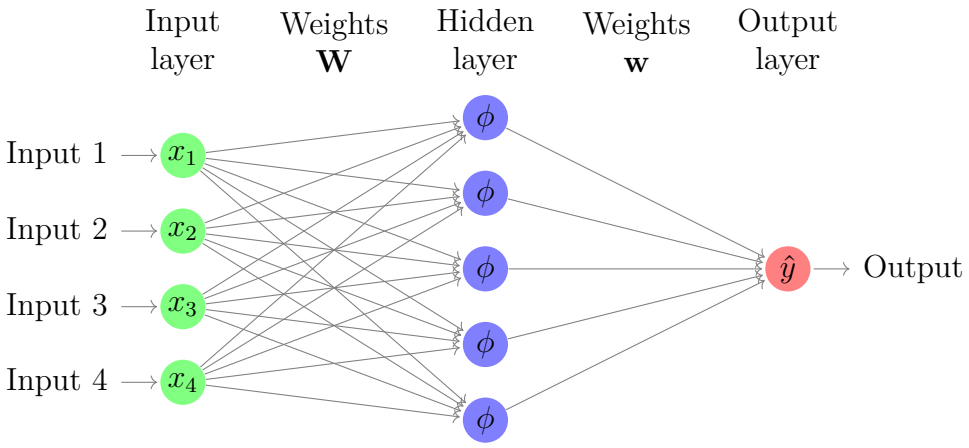


Figure 2.2: Neural network with 4 inputs, one hidden layer of 5 units, and an output layer with 1 unit. Bias units are omitted for clarity.

2.3.2.1 Activation functions

Consider a neural network with one hidden layer, this network can be described by two functions chained together $f(\mathbf{x}) = f^2(f^1(\mathbf{x}))$. Let $\mathbf{h} = f^1(\mathbf{x})$ and thus $f^2(\mathbf{h}) = \mathbf{h}^T\mathbf{w}$. If $\mathbf{h} = \mathbf{W}^T\mathbf{x}$ the neural network naturally becomes a linear model since can simply set $\mathbf{w}' = \mathbf{W}\mathbf{w}$. Thus, to allow a neural network to learn non-linear relationships, *activation functions* must be applied. If we instead take $\mathbf{h} = g(\mathbf{W}^T\mathbf{x})$ where g is a non-linear activation function, the network no longer reduces to a linear model.

In modern neural networks, and especially in networks with many hidden layers, a common choice is to use *rectified linear units*, *ReLU*, defined element-wise as [20]

$$g(z) = \max\{0, z\}. \tag{2.12}$$

Other choices include squashing functions such as the *sigmoid*

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.13}$$

or tanh,

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{2.14}$$

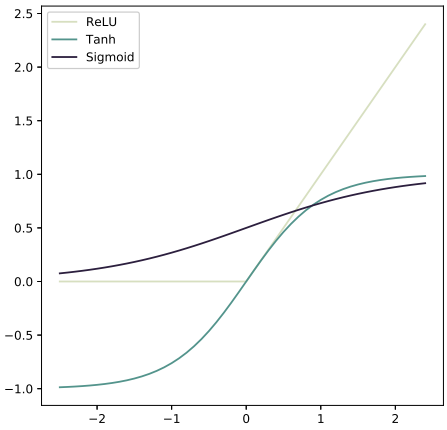


Figure 2.3: Activation functions on the interval $[-2.5, 2.5]$

The loss functions in equations 2.12, 2.13, and 2.14 on the interval $[-2.5, 2.5]$ can be seen in figure 2.3.

The universal approximation theorem states that a neural network with at least one hidden layer and any squashing activation function can approximate any continuous function arbitrarily well given that there are enough hidden units [20]. Although any function can be approximated with only one hidden layer, deep neural networks, with many hidden layers, have been shown to generalise much better to unseen data than wide networks, networks with many units but few hidden layers [20]. Squashing functions have also become less popular in favour of ReLU-like activation functions in deep networks. One reason for this is the saturation of the squashing functions, they saturate to a high value when the input is very positive and a low value when it is negative. This makes gradient-based learning difficult when inputs are not close to zero [20]. ReLU functions on the other hand have large and consistent gradients as long as a unit remains active, i.e is above zero [20].

2.3.2.2 Network outputs

A big difference between neural networks in general and the simple linear model presented in section 2.3.1 is that the non-linearity from introducing activation functions causes most loss functions to become non-convex [20]. The implication of non-convexity is that a closed form solution can no longer be derived, nor can convex optimisation techniques that guarantee global convergence be used [20]. Thus, neural networks are often trained by different forms of iterative gradient-based optimisers, from simple stochastic gradient descent [21] to more advanced techniques such as *Adam* [22]. The loss function that is optimised can take a variety of forms depending on what problem is solved, often the choice of the loss function is closely related to the choice of output layer.

When regression problems are considered, the output units are often taken as linear units without activation functions. Common loss functions include the *mean absolute error*, equation 2.15, or *mean squared error*, equation 2.16, of the prediction

$\hat{y}$ and the target y of dimensionality m.

$$\sum_{j=1}^{m} |\hat{y}_j - y_j| \tag{2.15}$$

$$\sum_{j=1}^{m} (\hat{y}_j - y_j)^2 \tag{2.16}$$

When considering classification tasks with k classes, the k output units are often taken through a *softmax function* to represent a probability distribution over the different classes as

$$\frac{\exp(\hat{y}_i)}{\sum_{j=1}^{k} \exp(\hat{y}_j)}. \tag{2.17}$$

The loss function to optimise, *cross-entropy loss*, is then taken as the negative logarithm of the softmax function.

Another common loss function in classification is the *hinge loss* defined as

$$\sum_{j=1 \wedge j \neq t}^{k} \max(1 - (\hat{y}_t - \hat{y}_j), 0) \tag{2.18}$$

where $\hat{y}_t$ is output of the unit that represents the correct class and $\hat{y}_j$ is the output of every other unit.

2.3.3 Recurrent neural networks

Recurrent neural networks, RNNs, are a family of neural networks that are specialised in processing sequential data. In these networks, information is persisted by using self-loops. A wide variety of recurrent network architectures can be designed but the focus here is on architectures that produce an output at each time step and have recurrent connections between hidden units, as described in [20]. As seen in figure 2.4, for input at time t, the network computes output $\hat{y}_t$. The loop in the hidden unit h_t allows information to be passed from one time step of the network to the next. A recurrent neural network can be seen as multiple copies of the same neural network, each passing a value to its successor. Unrolling the loop in figure 2.4 for four time steps produces the unrolled network in figure 2.5.

Importantly, each input at a time step shares weights with all other time steps [20]. In the architecture described here, there are three weight matrices parameterising the neural network. Input-to-hidden connections by a matrix $\mathbf{U}$, hidden-to-hidden connections by a matrix $\mathbf{W}$, and hidden-to-output connections by a matrix $\mathbf{V}$ [20]. In this architecture, the loss is computed for each time step as can be seen in figure 2.6, where L is some loss function that takes target y_t and prediction $\hat{y}_t$ at time t as input.

To compute the output of a recurrent neural network with the described architecture, the hidden state at time zero, h_0, is first initialised. Then, for each discrete time step $t = 1, t = 2, \ldots t = \tau$ the update equations are applied as [20]

$$\begin{aligned} \mathbf{a}_t &= \mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t \\ \mathbf{h}_t &= g(\mathbf{a}_t) \\ \hat{y}_t &= \mathbf{c} + \mathbf{V}\mathbf{h}_t \end{aligned} \tag{2.19}$$

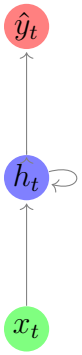


Figure 2.4: A recurrent neural network taking input x_t with hidden unit h_t producing output $\hat{y}_t$.

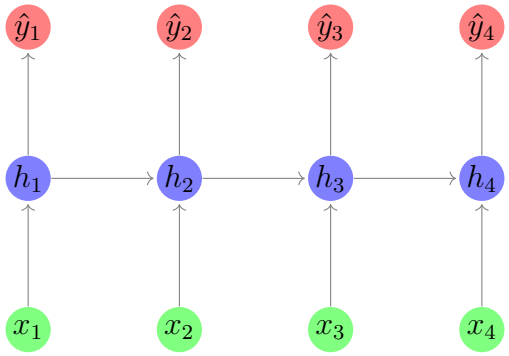


Figure 2.5: Recurrent neural network with unrolled self-loop for 4 inputs.

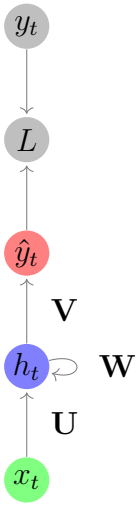


Figure 2.6: Loss computation of a recurrent neural network.

where $\mathbf{b}$ and $\mathbf{c}$ are the bias vectors and g is an activation function, for instance tanh. The total loss given a sequence of inputs $\mathbf{x}$ paired with a sequence of outputs $\mathbf{y}$, would then be the sum of all losses over all time steps [20].

The most effective recurrent neural networks for sequence modelling are called *gated RNNs* [20]. These models include *long short-term memory* (LSTM) [23] and *gated recurrent units* (GRU) [24]. A problem with traditional RNNs is that of

learning long-term information through time, usually the gradient vanishes when propagated over many states [20]. Gated RNNs allow networks to accumulate information over a long period of time whilst also allowing networks to decide when the accumulated information is used and then forget it [20]. On a high level, these models follow the architecture presented in this section but use custom layers and functions in the hidden units.

2.3.4 Convolutional neural networks

Feedforward neural networks use matrix multiplication with weights to describe the connection between layers, this implies that every output interacts with every input, hence the network is fully connected. Convolutional neural networks, CNNs, on the other hand, have local connectivity meaning that every output unit does not necessarily interact with every input unit. Unlike feedforward neural networks, convolutional networks have units arranged in several dimensions. For simplicity the 1-dimensional convolution is introduced here, but the convolution operation can be applied through more dimensions. The 1-dimensional convolutional model has two dimensions: length and depth. An example of such an input is a sequence with a length, that is every time step, and a depth, that is the values describing each entity. Each layer in a convolutional network consist of a set of learnable *kernels*, these are the weights of the convolutional network. These kernels are usually small spatially (in this case length), but extends through the full depth of the input. To compute an output, each kernel is convolved over the input. This is done by sliding each kernel across the spatial dimension and computing dot products between the kernel and the inputs at any position. For each convolved kernel, a 1-dimensional *activation map* is produced. The full output is then produced by stacking each activation map along the depth dimension.

The size of the output from a convolutional layer is controlled with three parameters. The first is the depth of the output which is equivalent to the number of kernels used. The second is the stride with which the kernel is slid over the spatial dimensions. For example, when the stride is 2, the kernels are moved two steps in the spatial dimension at a time. Having larger strides produces spatially smaller outputs. The third parameter is the amount of padding used around the border of the inputs. Padding allows control of the spatial size of the outputs. Most commonly zero padding is used. An example showing how different strides produce different spatial outputs is shown in figure 2.7.

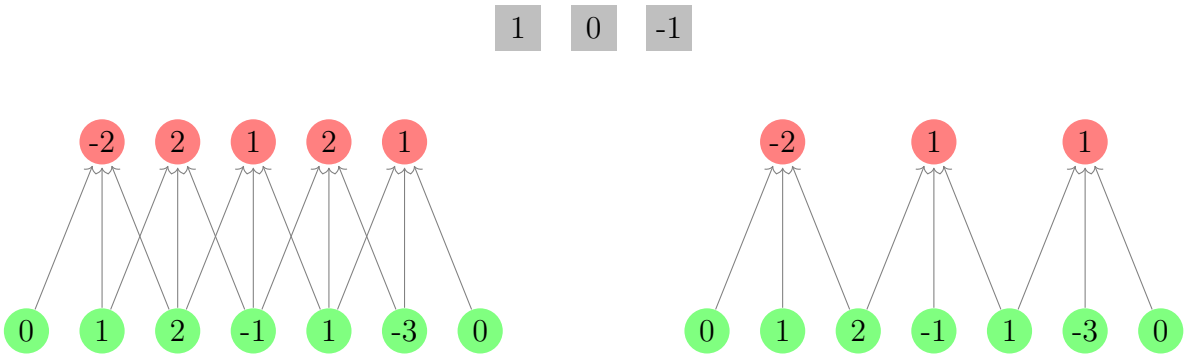


Figure 2.7: Example of spatial arrangement in one dimension. A kernel with size 3, grey, is convolved over inputs, green, to compute outputs, red. Zero padding is taken as 1. Stride is taken as 1 in the left network and as 2 in the right network.

2.3.4.1 Causal convolutions

Consider a sequence modelling task that given an input sequence $x_0, \dots x_t$, the task is to predict corresponding outputs $y_0, \dots, y_t$ at each time step. For each output y_i, we are constrained to only use inputs that have been observed at that point in time: $x_0, \dots x_i$. Formally the sequence modelling task is to learn a function $f : \mathcal{X}^{t+1} \to \mathcal{Y}^{t+1}$ that produces a mapping

$$\hat{y}_0, \dots \hat{y}_t = f(x_0, \dots, x_t) \tag{2.20}$$

with the constraint that $\hat{y}_i$ depends only on $x_0, \dots, x_i$. With a convolutional network that takes an input sequence $x_0, \dots, x_t$, this function can be approximated by using an output layer which is simply the input layer shifted by one time step [11]. A *causal convolution* is a convolutional layer where the next layer has the same length and we enforce that an output at time i is only convolved with elements from time i and earlier [11]. This is illustrated in figure 2.8.

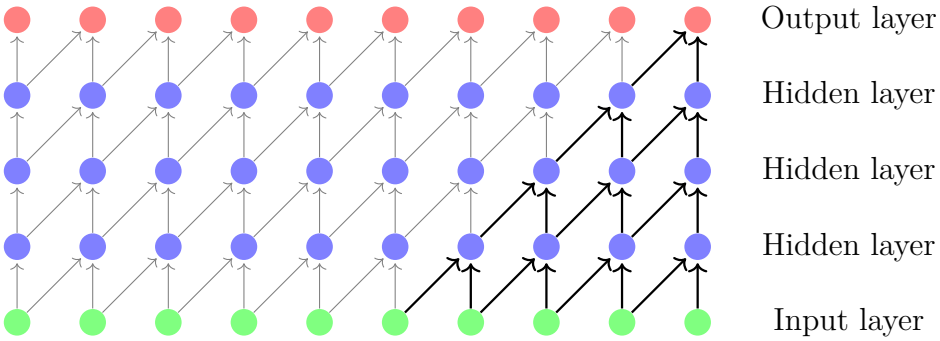


Figure 2.8: Illustration of causal convolutions where outputs only depend on inputs that occurred earlier in time. Adapted from [1].

2.3.4.2 Dilated convolutions

Apart from padding, stride, and kernel size, another hyperparameter that can be introduced on the convolution operation is known as a *dilation*. Differently from the

convolutions discussed previously, a dilated convolution has spaces between each cell in the kernel, called dilations. For example, for a kernel w of size 3, a 1-dilation on input $\mathbf{x}$ would compute

$$\mathbf{x}_0 w_0 + \mathbf{x}_1 w_1 + \mathbf{x}_2 w_2 \tag{2.21}$$

whereas a 2-dilation would compute

$$\mathbf{x}_0 w_0 + \mathbf{x}_2 w_1 + \mathbf{x}_4 w_2. \tag{2.22}$$

The difference between equation 2.21 and 2.22 is that there is a gap between applications of the convolution in the second. Formally, for a 1-dimensional input sequence $\mathbf{x} \in \mathbb{R}^n$ and a kernel $w : \{0, \ldots, k-1\} \to \mathbb{R}$, the dilated convolution on an element s of the sequence is defined as

$$F(s) = (\mathbf{x} *_d w)(s) = \sum_{i=0}^{k-1} w(i) \mathbf{x}_{s-di} \tag{2.23}$$

where d is the dilation factor and k is the size of the kernel. $s - di$ accounts for the direction of the past [11]. Taking $d = 1$ yields a standard convolution. Thus dilations allow merging of spatial information across inputs more aggressively and with fewer layers than regular convolutions. The effective receptive field can increase exponentially with exponentially increasing dilation factors for each layer, whilst the number of parameters still grows linearly [25]. The effective receptive field at a layer i can be computed with

$$f_i = f_{i-1} + (w - 1) \cdot d_i \tag{2.24}$$

where $f_0 = 1$. For the example network seen in figure 2.9, with kernel size $w = 2$ and dilation factors d_i increasing with a power of two, the effective receptive field at each layer is computed as

$$\begin{aligned} f_0 &= 1 \\ f_1 &= f_0 + 1 = 2 \\ f_2 &= f_1 + 2 = 4 \\ f_3 &= f_2 + 4 = 8 \\ f_4 &= f_3 + 8 = 16 \end{aligned} \tag{2.25}$$

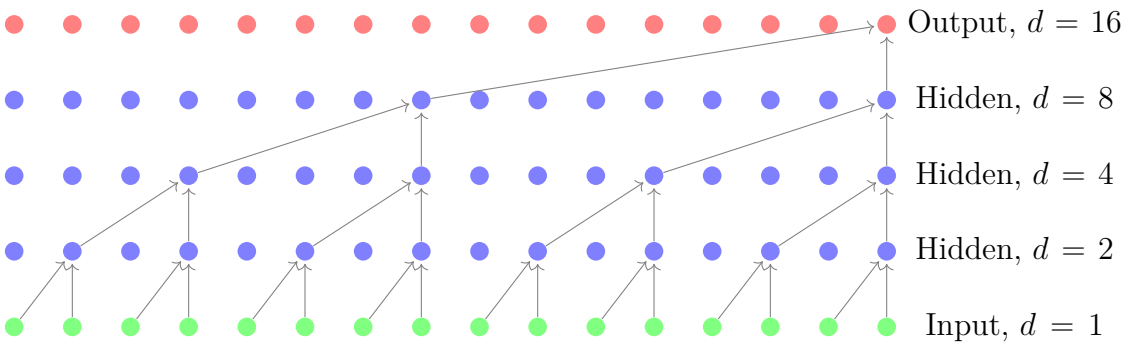


Figure 2.9: Illustration of exponentially increasing dilation factors. Adapted from [1].

2.3.5 Embedding layers

The idea of embeddings is similar to the latent factor models presented in section 2.2. There, each item and user is mapped to a latent factor of some predefined dimensionality k, here called the embedding dimensionality. This idea of mapping entities to some vector space can be reframed to a neural network setting by considering *embedding layers*. Like with latent factor models, for a vocabulary of n entities, we decide on an embedding dimensionality k and define a mapping from each entity to a latent vector with dimensionality k. Let an entity in a vocabulary be defined in one-hot encoded form, $\mathbf{v} \in \{0, 1\}^n$, the weights of an embedding layer is then a matrix $\mathbf{W} \in \mathbb{R}^{n \times k}$, and an embedding is produced by taking the matrix product $\mathbf{v}^T\mathbf{W} \in \mathbb{R}^k$. This matrix multiplication can be seen as a fully connected layer in a neural network, as seen in figure 2.10. Thus embeddings can be learnt jointly with other model parameters when optimising neural networks.

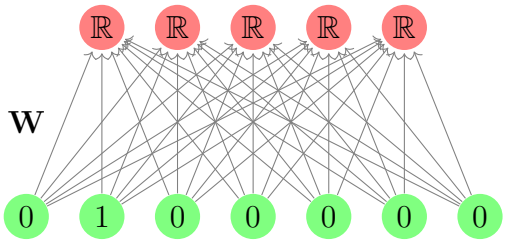


Figure 2.10: Illustration of an embeddding layer that maps entities in a vocabulary of $|\mathbf{v}| = 7$ entities, seen in green, to embeddings with dimensionality $k = 5$, seen in red.

2.4 Related work

In 2006, the company Netflix announced a public competition to improve their film recommender system. The first team that could improve the performance of the Netflix algorithm by at least 10 per cent would win a $1 million prise. The contest created a buzz within the area of recommender systems in general and within

collaborative filtering in particular, since the released dataset was magnitudes larger than other publicly available datasets at that time. The most successful models in the competition, as well as the winning team which reached the 10 per cent improvement in 2009, used variants of the matrix factorisation methods presented in section 2.2. Since then, matrix factorisation models have remained state-of-the-art in many recommendation domains [15].

Even though many matrix factorisation techniques were designed for explicit feedback in datasets like the Netflix prize, they can be used on implicit feedback with some modifications as presented by [3] and seen in section 2.2.2. The goal of implicit feedback is often not to produce a single value but a ranked list, however [3] does not directly optimise the model parameters for ranking. In 2009, a generic optimisation criterion for personalised ranking as well as a generic learning algorithm to optimise the criterion was presented in [5]. This generic method can be used to optimise implicit feedback matrix factorisation models directly on ranking. The main idea to optimise for ranking is to sample implicitly negative items and compute the loss pairwise between positive and sampled items, an idea that has been further researched and expanded on in [26] [27].

In recent years, deep learning has revolutionised several artificial intelligence tasks such as image analysis and natural language processing. Applying these techniques to recommender systems has become an increasingly popular research area with no sign of stagnating [9]. Recent recommender systems based on deep learning techniques have gotten significant attention by outperforming several conventional models, often based on matrix factorisation techniques. Deep models are able to capture complex non-linear relationships between users and items, something traditional models often struggle with [9]. Due to the recent advances in deep learning in general, many companies have resorted to deep learning for further enhancing the quality of their recommendations. Following is a summary of work within deep learning related to ours. For a more comprehensive review of deep learning in recommender systems we refer to [9].

A notable deep learning hybrid model for video recommendations on YouTube is [28]. In their model, implicit feedback is used to train a recommender system in two stages. The first stage, which is made up of a deep feedforward neural network reassembles a more complex non-linear matrix factorisation technique. The second stage, which is also based on a deep feedforward neural network incorporates content information as well. In their approach, the first stage serves as a candidate-generation network aimed at finding a set of candidate items in a large item corpus. The second stage is aimed at ranking the produced candidates. As the system takes both collaborative and content information into account it is essentially a hybrid recommender system based on deep learning.

The work of [7] investigates recommendations on sequential data, more specifically session-based recommendations. The task they are looking to solve is to predict the next item a user will click or buy in a session. Their approach is based on recurrent neural networks for next-item predictions. The follow-up work in [8] proposed several ideas to further improve the model and reach state-of-the-art predictive performance.

A different approach to make recommendations on sequential data is presented

in [29], where convolutional neural networks are used for the task of sequence-aware recommendations. [29] uses a 3-dimensional convolutional network aimed at integrating content features. Their network architecture does not use causal or dilated convolutions but instead treats the problem as a classification task where a fully connected layer is taken as the last layer before a softmax function is applied to get class scores. They further constrain their experiments to relatively short sequences of maximum 7 clicks.

On the general task of sequence modelling with neural networks, [11] compares recurrent neural networks with convolutional neural networks on a wide range of sequence modelling tasks. They present a general convolutional neural network called temporal convolutional network (TCN). Their results show that the TCN model, based on causal dilated convolutions, can outperform recurrent neural networks, which have widely been considered the go-to models for sequence modelling tasks. TCN has several similarities with other convolutional neural networks used in sequence modelling tasks. One example of a similar network architecture is WaveNet [1], a deep neural network for generating raw audio.

3

Two-stage sequence-aware recommender system

This chapter begins by giving a general overview of a two-stage approach to sequence-aware recommendations. The two stages are then presented in detail, first a candidate generation network and then a ranking network.

3.1 Approach

The amount of different methods and algorithms available for solving recommendation problems is huge. Since the problem domain here is to do sequence-aware recommendations from implicit feedback, the ultimate goal of such a recommender system is to provide a user with a ranked list of items he or she will most likely be interested in. To achieve this goal, our choice of recommender system architecture is based on a two-stage approach similar to [28]. Such a two-stage approach has been shown successful in implicit feedback domains with large item corpora, and it enables the capture of both short-term as well as long-term user preferences. The first stage is *candidate generation*, where the task is to find a general set of likely candidates a user may be interested in. The second stage is *ranking*, where the task is to take the candidates from the first stage and use a more precise machine learning model to rank them. As there are much fewer items in the second stage, additional sources of information about items and users can be used.

To solve the candidate-generation problem, a *candidate generation network* that is sequence-aware and based purely on collaborative information, user-item interactions, is considered. In this stage, a convolutional neural network is used as the underlying model. The items produced by the candidate generation network then form inputs to a *ranking network* that takes content features and additional sources of information into account. The underlying ranking model is based on a deep feedforward neural network. This two-stage approach forms a hybrid recommender system based on a cascade approach. The architecture can be seen in figure 3.1.

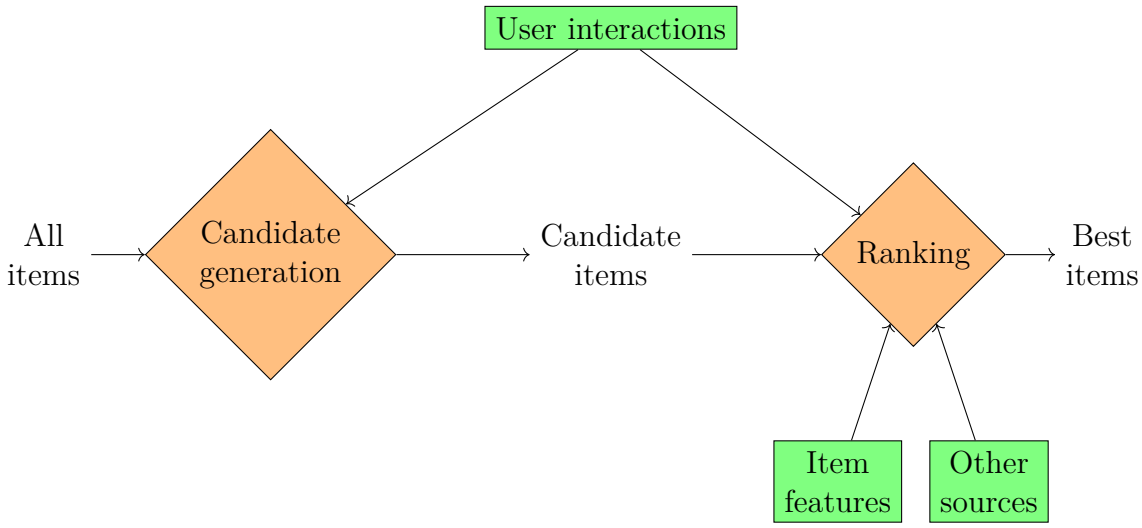


Figure 3.1: Two-stage recommender system architecture. Orange diamonds represent neural network models, taking inputs described in green rectangles.

3.2 Candidate generation stage

Solving a ranking problem directly on a large number of items imposes a problem: the computational complexity during training and prediction increases linearly with the number of items in the corpus. For the task of item recommendations, the number of items to rank is often synonyms with the number of available items and can thus be several hundreds of thousands. In order to reduce the number of items to rank, the use of a candidate generation network that extracts a subset of items from an item corpus can be used. The candidate generation network's task is to produce a set of item candidates given sequences of user interactions as input.

3.2.1 Learning setting

To do efficient candidate generation over many items, only collaborative information is considered in this stage. That is, the input is only based on a set of interactions made by a user. The interactions are viewed sequentially, taking the time of the interaction into account. Given the historical item interactions of a user $U = \{v_1, v_2, \ldots, v_t\}$, the learning task in the candidate generation stage is to predict the next item v_{t+1} a user will interact with.

Each item, represented by an id, is mapped to a latent representation using an embedding layer with dimensionality d, $\nu_i \in \mathbb{R}^d$. These latent representations, or embeddings, are jointly optimised with model parameters during training, thus allowing the model to learn similarities and relations between items. The task of the model is to learn an output embedding $\mu_t \in \mathbb{R}^d$ at time t as a function of user history such that the embedding can be used to discriminate among items.

3.2.2 Network architecture

The general neural network architecture can be seen in figure 3.2. The basis of the architecture is several layers of 1-dimensional causal dilated convolutions. Considering a sequence with t interactions, the first step taken by the network is to convert each of the items in the t interactions to their corresponding embedding by taking them through an embedding layer. For embeddings of dimensionality d, the input to the first convolutional layer is of shape $t \times d$. Stride is taken as 1 and padding is set so that the spatial output dimensionality is also t. d kernels are applied in each layer to produce d stacked activation maps, making the final output of each convolutional layer $t \times d$. By using causal convolutions and retaining the spatial dimensionality, the final convolutional layer produces a d-dimensional output embedding for each time step in the sequence, $\mu_1, \ldots, \mu_t$. The output embedding used to make a prediction over the whole sequence is thus in the last slice t, where all interactions have been seen by the network. Each hidden layer increases the dilations with a factor of 2 to exponentially increase the receptive field of the network. Residual connections, introduced in [30], are used between each layer. If a layer computes the function $f(\mathbf{x})$, then with a residual connection the effective output of the layer is $f(\mathbf{x}) + \mathbf{x}$. Using residual connections allow layers to learn modifications to the identity mapping instead of learning the full transformation [11]. These connections have repeatedly been shown to benefit deep networks by making optimisation easier [30].

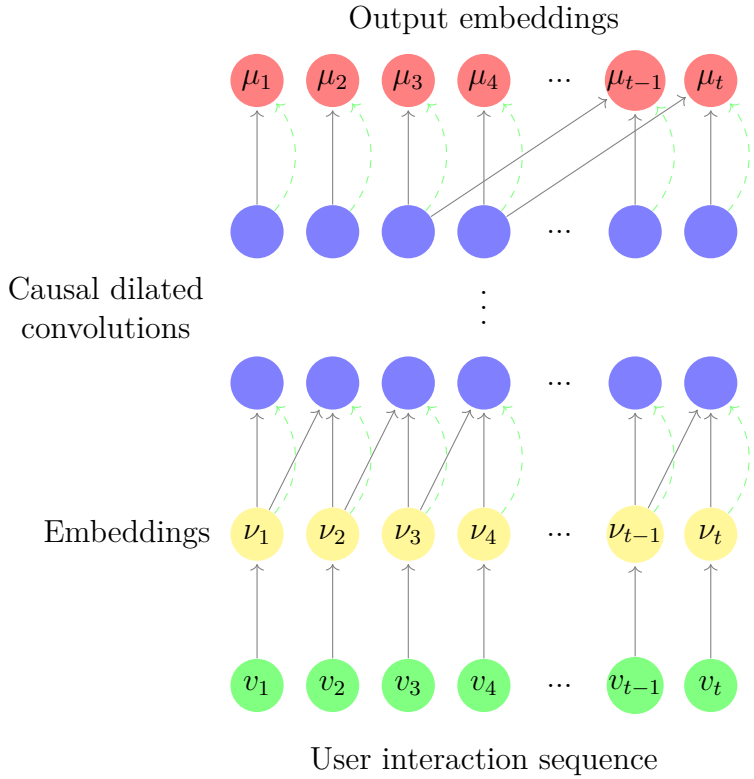


Figure 3.2: Candidate generation neural network taking a sequence of t interactions as input. Residual connections are showed with dashed green lines.

3.2.3 Efficient training

To compute a score for each item during training, the dot products between the output embedding and every item embedding in the corpus must be computed. When there are several thousands of items in the corpus this becomes very expensive to compute, therefore an adaptive hinge loss is used to handle this issue. First, the dot product between the output embedding and the embedding of the target item is computed to get a positive prediction. Second, a set of implicitly negative items, $\mathcal{N}$, is sampled, that is, items that a user has not interacted with. Next, the dot products between the output embedding and the item embeddings of the negative samples are computed. The negative item that has the highest score is then taken through a regular hinge loss, see 2.18. This loss does not only speed up the loss computation but it has also been shown that sampling implicitly negative items is a good way to train implicit feedback recommender systems, see [5]. Formally the loss function L_i at time i is defined as

$$\begin{aligned} p_i &= \nu_{i+1}^T \mu_i \\ n_i &= \max_{j \in \mathcal{N}} (\nu_j^T \mu_i) \\ L_i(p_i, n_i) &= \max(1 - (p_i - n_i), 0) \end{aligned} \tag{3.1}$$

where p_i is the positive prediction, i.e dot product between target item embedding and output embedding, and n_i is the maximum negative prediction among the sampled items, i.e the negative prediction violating the implicit rank the most.

To increase the size of a dataset of sequences, every sub-sequence of a sequence could be used as an input to the network and then simply padded with zeros when it is shorter than the desired sequence length. Although, since the presented architecture is fully convolutional, it computes an output for every time step. Thus instead of using every sub-sequence, the outputs can be used to compute the loss over all sub-sequences. This effectively resembles increasing the sample size without requiring additional computing power and memory to convert and process the new samples. Formally, equation 3.1 is computed for the output embedding at each time step and then averaged as

$$L = \frac{1}{t} \sum_{i=1}^{t} L_i(p_i, n_i). \tag{3.2}$$

3.2.4 Efficient predictions

The adaptive hinge loss described in the previous section 3.2.3 solves the problem of efficiently training the candidate generation network, but it does not speed up the prediction phase. Since the ultimate task is to select the k best candidates, computing dot products between the output embedding and every item embedding in the corpus would still be required during prediction. Instead, a nearest neighbour index, shown successful in [28], can be used in the prediction phase. After training, every learnt item embedding together with its corresponding id is added to an index. Thus during serving, instead of computing every dot product, the output embedding is used to query the index for the approximate k nearest neighbours in embedding space.

3.3 Ranking stage

With candidates from the candidate generation stage available, the next step is to learn another model to rank them. With much fewer items in this stage, the model can incorporate item metadata and other user interactions in addition to the user-item interactions used in the previous stage. Ranking is considered directly in the domain of item features, such that each item is described by a vector of features.

3.3.1 Learning setting

The task of learning to rank is to find an unknown scoring function for a joint feature map of inputs $f(\mathbf{x}, \mathbf{y})$. Here $\mathbf{x}$ are input features that describe context such as user history and $\mathbf{y}$ are item features of a candidate. The output of the function can be used as a ranking system S by simply sorting results of each item candidate $\mathbf{y}$ in the candidate set Y as

$$S(\mathbf{x}, Y) = \text{argsort}_{\mathbf{y} \in Y}\{f(\mathbf{x}, \mathbf{y})\}. \tag{3.3}$$

Let $\text{rank}(\mathbf{y}|S(\mathbf{x}, Y))$ be the position of $\mathbf{y}$ in the ranking $S(\mathbf{x}, Y)$. Since rank is a discontinuous step-function and thus can not be differentiated, the rank function is upper bounded with the hinge loss [31]

$$\begin{aligned} \text{rank}(\mathbf{y}_i|S(\mathbf{x}_i, Y_i)) - 1 &= \sum_{\mathbf{y} \in Y_i \wedge \mathbf{y} \neq \mathbf{y}_i} \mathbb{I}(f(\mathbf{x}_i, \mathbf{y}) - f(\mathbf{x}_i, \mathbf{y}_i) > 0) \\ &\leq \sum_{\mathbf{y} \in Y_i \wedge \mathbf{y} \neq \mathbf{y}_i} \max(1 - (f(\mathbf{x}_i, \mathbf{y}_i) - f(\mathbf{x}_i, \mathbf{y})), 0) \end{aligned} \tag{3.4}$$

applying a monotonically increasing function λ on both sides, the inequality still holds and the function upper bounds the average rank of the system S for n samples

$$\frac{1}{n} \sum_{i=1}^{n} \lambda(1 + \sum_{\mathbf{y} \in Y_i \wedge \mathbf{y} \neq \mathbf{y}_i} \max(1 - (f(\mathbf{x}_i, \mathbf{y}_i) - f(\mathbf{x}_i, \mathbf{y})), 0)). \tag{3.5}$$

One monotonically increasing function that could be used as λ is the *discounted cumulative gain*

$$\lambda(r) = \frac{-1}{\log(1 + r)} \tag{3.6}$$

or simply the average rank

$$\lambda(r) = r. \tag{3.7}$$

3.3.2 Network architecture

The task of the neural network is to approximate the scoring function $f(\mathbf{x}, \mathbf{y})$ used by the ranking system S in equation 3.3. The network architecture used to approximate the scoring function is seen in figure 3.3 where $NN(\phi(\mathbf{x}, \mathbf{y}))$ represents the real-valued scalar output of the neural network and $\phi(\mathbf{x}, \mathbf{y})$ is a joint feature map of context $\mathbf{x}$ and candidate item $\mathbf{y}$. All layers are feedforward and the network follows a common tower pattern where the input layer is the widest and following layers

decrease in width. ReLU activation functions are applied in all but the last layer. In equation 3.5, $f(\mathbf{x}, \mathbf{y})$ is replaced with the network output $NN(\phi(\mathbf{x}, \mathbf{y}))$ to get the loss function with respect to the network parameters as

$$\frac{1}{n}\sum_{i=1}^{n}\lambda(1+\sum_{\mathbf{y}\in Y_i \wedge \mathbf{y}\neq\mathbf{y}_i}\max(1-(NN(\phi(\mathbf{x}_i,\mathbf{y}_i))-NN(\phi(\mathbf{x}_i,\mathbf{y})),0)). \tag{3.8}$$

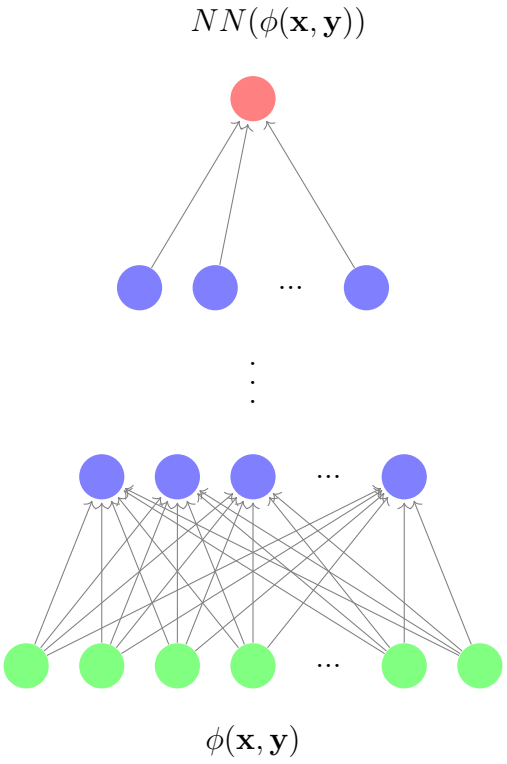


Figure 3.3: Feedforward neural network architecture with a variable number of hidden layers. The input consists of a joint feature map of the context and a candidate item $\phi(\mathbf{x}, \mathbf{y})$, the produced output is a scalar $NN(\phi(\mathbf{x}, \mathbf{y}))$.

3.3.3 Training strategy

Differently from standard neural network training, the loss function does not decompose into a sum over input-output pairs $(\mathbf{x}_i, \mathbf{y}_i)$ and the non-linear function λ is thus problematic. As proposed in [31], training is considered not in the domain of input-output pairs, but over interaction instances. An interaction instance consists of a context $\mathbf{x}_i$ and a set of candidates Y_i.

To train over interaction instances, the neural network loss computation scheme proposed by [31] is used. The accumulated forward pass for an interaction instance consists of several forward passes through the neural network, once for each candidate $\mathbf{y} \in Y_i$. After each candidate has been fed through the neural network, the accumulated loss is computed by combining outputs with a pairwise hinge loss between the target item and every other candidate item. The resulting loss with

$|Y_i| = m$ candidate items and $\mathbf{y}_t$ as the target item is thus computed as

$$\begin{aligned} L(h) &= \lambda(1 + \sum_{j=1}^{m-1} h_j) \\ h_j &= \max(1 - (NN(\phi(\mathbf{x}_i, \mathbf{y}_t)) - NN(\phi(\mathbf{x}_i, \mathbf{y}_j))), 0) \end{aligned} \tag{3.9}$$

where h_j is the pairwise hinge loss between the target $\mathbf{y}_t$ and a candidate $\mathbf{y}_j$. The process of computing the accumulated loss is illustrated in figure 3.4.

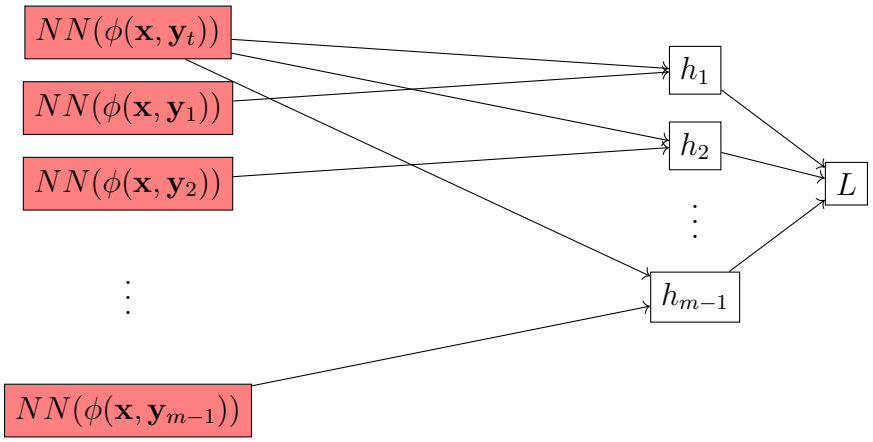


Figure 3.4: Accumulated loss computation scheme for an interaction instance where $\mathbf{y}_t$ is the target item and $\mathbf{y}_1 \dots \mathbf{y}_{m-1}$ are the other candidate items. The pairwise hinge loss between the target's output and every other candidate's output is first computed. Then, the pairwise hinge losses serve as input in L, equation 3.9.

4
Experimental setup

In this chapter, the datasets used for experiments are first presented. Then, baselines and evaluation metrics used in experiments are introduced before the implementation is discussed.

4.1 Sellpy market dataset

The main dataset used to evaluate the two-stage recommender system was the *Sellpy market dataset*. The dataset is comprised of items viewed by users. These views were converted to sequences by sorting the interactions of each user by time. The user-item views were split into a train set 90% and a test set 10%. The split is time-based, meaning that any interaction in the test set is at least as late as the latest interaction in the train set. The advantage of this split is that it reassembles a more realistic setting than randomly splitting user-item views which would break the sequential nature of the data. Further, sequences of length less than 2 were removed from the dataset since predictions can not be made from empty sequences. From the nature of collaborative filtering, items that were in the test set but not in the training set were also removed. Since the Sellpy market dataset has many items in contrast to the number of interactions, two datasets with a reduced number of items were also considered. In the *clustered* dataset, items are clustered into groups based on their category, size, type, and brand. In the *reduced* dataset, items are clustered into groups based only on their category and size. These datasets were preprocessed in the same way as the regular, *full*, dataset. The datasets after preprocessing are summarised in table 4.1.

Table 4.1: Sellpy market datasets summary.

Dataset	Type	Users	Items/Clusters	Interactions
Full	Train	9602	232,997	897,691
Full	Test	2941	37,326	69,441
Clustered	Train	9587	20,066	882,597
Clustered	Test	3287	9381	96,269
Reduced	Train	9587	200	882,597
Reduced	Test	3297	197	97,493

4.1.1 Item features

In the ranking phase, features of user-item interactions are considered as inputs. Every item on Sellpy market is described by a set of categorical and numerical attributes, summarised in table 4.2. To make features that can be used as inputs in a ranking model, the item attributes were fed through a pipeline of preprocessing steps as follows:

- **Vectorisation**: This step converts all categorical attributes into sparse one-hot encoded features.
- **Scaling**: This step scales features to the range $[0, 1]$.
- **Variance clipping**: Features that have the same value in more than 99.9% of the samples are removed.
- **Normalisation**: Features are normalised to have unit variance.
- **Dimensionality reduction**: Features are transformed to another lower dimensional space where 90% of the original variance is retained.

Table 4.2: Item attributes available in the Sellpy market dataset.

Name	Type	Description
Category	Categorical	Primary category of item
Subcategory	Categorical	Subcategory of item
Subsubcategory	Categorical	Subsubcategory of item
Current value	Float	Current value
Estimate bid	Float	Estimated likelihood of bid
Condition	Categorical	Item condition
Colour	Categorical	Item colour
Brand	Categorical	Item brand
Size	Categorical	Size of item
Material	Categorical	Item material
Defect	Categorical	Item defect
Original price	Float	Purchase price

4.1.2 Other sources

As described in section 3.1, the ranking network can also take additional sources of information. To capture long-term user tastes, order information was taken as an additional source of information in the ranking stage. This was done by considering, for each sequence of interactions, the latest items ordered by that user at that point in time. The items in these orders were converted to features, following the same pipeline described above in section 4.1.1. Finally, the order features were averaged to make an additional input signal in the ranking network.

4.2 Benchmark datasets

Two benchmark datasets were considered for evaluation and comparisons.

The first benchmark dataset is *RecSys Challenge 2015* [32] which consists of sessions of click-streams from an e-commerce site that sometimes end in purchase events. Only the provided training set was used and no difference was made if a sequence of interactions ended with a purchase. The dataset was split into a train set and a test set with the same procedure described in [7]. The test set is made up of sessions of the subsequent day. The dataset was split so that a session is either completely in the train set or test set. Any sessions containing items that are in the test set but not in the train set were filtered out. The minimum session length was taken as 2, removing sessions with only one interaction. Due to memory requirements, sessions longer than 80 interactions were also filtered out. Filtering out the long sequences left the test set unaffected but removed around 1000 sessions from the train set, a small amount considering the size of the dataset. Sequences with fewer than 80 interactions were padded with zeros. A summary of this dataset can be seen in table 4.3.

Table 4.3: RecSys dataset summary.

Type	Sessions	Items	Interactions
Train	7,965,295	37,483	31,532,116
Test	15,324	6751	71,222

The second benchmark dataset is the *Movielens* dataset [33]. This dataset consists of user-movie ratings that were converted to sequences of implicit feedback by, for each user, sorting the times of movie ratings and then only considering the movie, not the actual rating. The dataset was split into a train set 80%, and a test set 20%. Any items that were only in the test set were removed. A time-split was employed such that interactions in the test set are always at least as late as the latest interaction in the train set. The minimum session length was taken as 2, filtering out users that have only interacted with one item. A summary of this dataset can be seen in table 4.4.

Table 4.4: Movielens dataset summary.

Type	Users	Items	Interactions
Train	5400	3662	799,998
Test	1739	3467	199,844

4.3 Baseline models

A set of baseline models were considered for comparison with our models. Different baselines were used on different datasets and models, following is a summary of all baseline models used in the experiments.

- **POP**: A simple popularity model, always recommending the most popular items.
- **MF**: Standard matrix factorisation for implicit feedback, trained under ranking loss. See [5].
- **POOL**: A neural network that is similar to the candidate generation network, but instead of using convolutions, averages item embeddings over each sequence to compute the output embedding. See [28].
- **LSTM**: A neural network that is similar to the candidate generation network, but computes the output embedding with a long short-term memory recurrent neural network instead of with a convolutional network. See [23].
- **RNN**: A recurrent neural network for session-based sequence prediction presented in [7].

4.4 Evaluation metrics

For evaluation of the candidate and ranking networks as well as for the baseline models, the three metrics summarised below were considered. The term entity is used to denote either a specific item or a cluster class. All metrics are defined for k samples and $\text{rank}(i)$ denotes the rank of entity i.

- **Prec@n** $\in [\mathbf{0}, \mathbf{1}]$: Percent of correct next entity in the top n. Formally defined in equation 4.1. Higher value is better.
- **AvgRank** $\in [\mathbf{1}, \mathbf{m}]$: The average rank of the correct entities among a corpus of m entities. Formally defined in equation 4.2. Lower value is better.
- **MRR@n** $\in [\mathbf{0}, \mathbf{1}]$: The mean reciprocal rank at n. Formally defined in equation 4.3. Higher value is better.

$$\textbf{Prec@n} = \frac{1}{k}\sum_{i=1}^{k}\begin{cases}1, & \text{rank}(i) \leq n. \\ 0, & \text{otherwise.}\end{cases} \tag{4.1}$$

$$\textbf{AvgRank} = \frac{1}{k}\sum_{i=1}^{k}\text{rank}(i) \tag{4.2}$$

$$\textbf{MRR@n} = \frac{1}{k}\sum_{i=1}^{k}\begin{cases}\frac{1}{\text{rank}(i)}, & \text{rank}(i) \leq n. \\ 0, & \text{otherwise.}\end{cases} \tag{4.3}$$

4.5 Implementation

All implementation was done in *Python 3.6.5*. Python was selected as programming language due to it being fast to prototype machine learning models with. The main data processing was made with *Numpy*, *Scipy*, and *Pandas* libraries, due to their

efficient C implementations of numerical calculations and matrix operations. Preprocessing was performed with methods from the general machine learning library *scikit-learn*.

The neural networks were implemented in *PyTorch*, a Deep learning framework for Python. PyTorch was preferred ahead of other Deep learning frameworks such as *TensorFlow* or *Theano* due to its flexibility from dynamically building the neural network structure. For the candidate generation stage, *Spotlight*, a library built on top of PyTorch was used due to its implementations of sequence models with embeddings [34].

The use of PyTorch enables the machine learning models to be run on GPU rather than CPU. Since neural networks are essentially matrix products, their operations can be parallelised. Generally, GPUs have many more cores than CPUs which enables more parallelisation, which in turn makes neural network training faster. Thus an *Nvidia Tesla K80* GPU with 12GB of RAM was used to run the experiments.

5
Results

In this chapter, results on baseline models and our recommender system are presented. Results are presented for two types of experiments, the first is experiments focusing only on convolutional neural networks for sequence-aware recommendations. The second is experiments on the two-stage recommender system.

5.1 CNN for sequence-aware recommendations

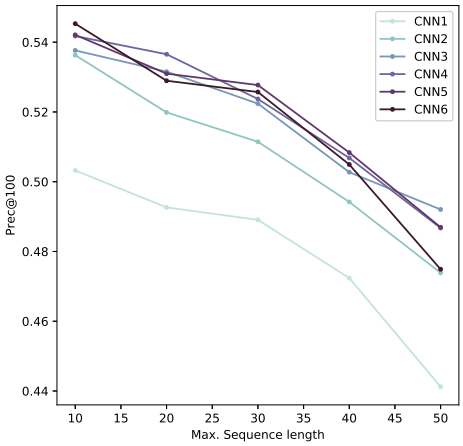
For experiments on convolutional neural networks used directly for sequence-aware recommendations, the two benchmark datasets were considered to quickly run experiments and easily compare results with other models.

5.1.1 Varying hyperparameters

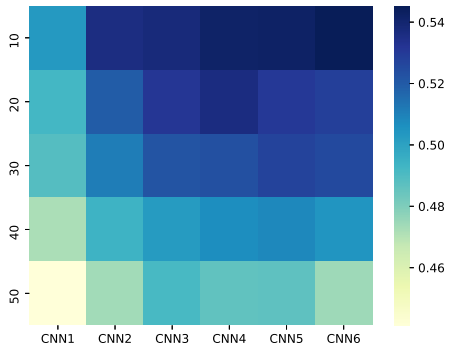
To investigate hyperparameter effects of convolutional neural networks for sequence prediction tasks, the Movielens dataset was used. Since it is a relatively small and general dataset, experiments could be run quickly. For all experiments on this dataset, if nothing else is mentioned, we trained the networks for 15 epochs with the gradient-based optimiser *Adam*, see [22], using initial learning rate $1 \cdot 10^{-3}$, embedding dimensionality 64, batch size 1024, L2-regularisation $1 \cdot 10^{-6}$. ReLU was taken as the activation function for the convolutional neural networks. The maximum sequence length was set to 30. Longer sequences were split into several sequences and shorter sequences were padded with zeros.

First, we ran experiments with 6 convolutional neural networks: CNN1...CNN6. The networks share all hyperparameters except network depth and dilation factor. The first network, CNN1, comprises one layer with dilation factor 1, i.e a standard convolutional network with a single layer. For each network with increasing index, one more hidden layer was added with dilation factor times two. So CNN2 has dilation factors $1, 2$ and CNN3 has $1, 2, 4$, and so forth. Each CNN was trained on sequences of interactions with maximum length varying from 10 to 50, increasing by 10. By setting a higher maximum sequence length, the number of samples is decreased but each sample is more expressive as the model receives a longer history to base its predictions on. The CNN results were compared with two baseline models, POOL and LSTM, that were trained with the same hyperparameters. Results on the Movielens test set are shown in figure 5.1.

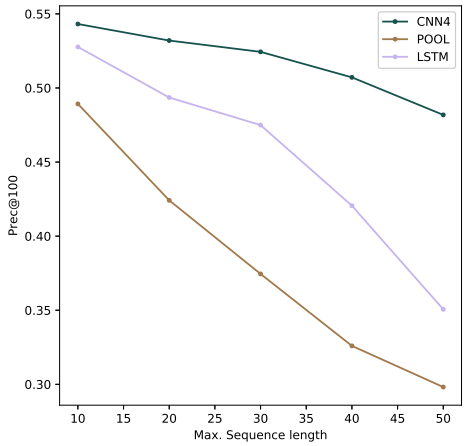
In figure 5.1a and 5.1b we note that the precision is generally better for shorter sequence lengths. We can also note that as depth and dilation factors increase, the



(a) Comparison of Prec@100 between 6 CNNs with different dilation factors as sequence length increases.



(b) Heat plot of Prec@100 for varying sequence lengths, 10 to 50, and networks with varying dilation factors, CNN1 to CNN6.



(c) Comparison of Prec@100 between CNN4, POOL, and LSTM as sequence length increases.

Figure 5.1: Prec@100 for varying maximum sequence lengths on the Movielens dataset.

performance on Prec@100 also increases in almost all cases. We note from figure 5.1c that CNN4 with dilation factors $1, 2, 4, 8$ outperforms both the POOL and the LSTM model for this dataset and parameter setting. We also see that as the maximum sequence length increases, implying that the number of samples decreases, the predictive performance goes down. From the figures we also see that different models decay in performance at different rates. Deep CNN models decay at a lower

rate than shallow CNN models as well as POOL and LSTM models.

We also experimented with embedding dimensionality, keeping all other parameters fixed and varying the dimensionality from 16 to 256 increasing by a factor of 2. Both the POOL and CNN4 models were tested, as seen in figure 5.2. We note that varying the embedding dimensionality has a large effect on the predictive performance of both models.

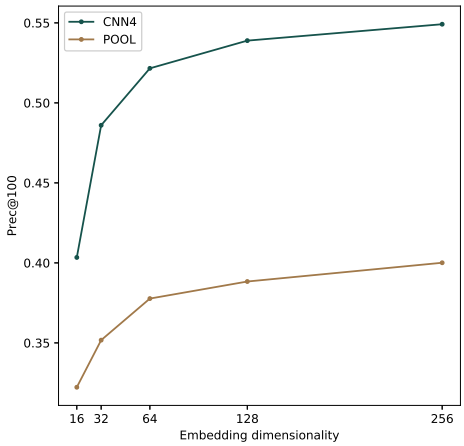
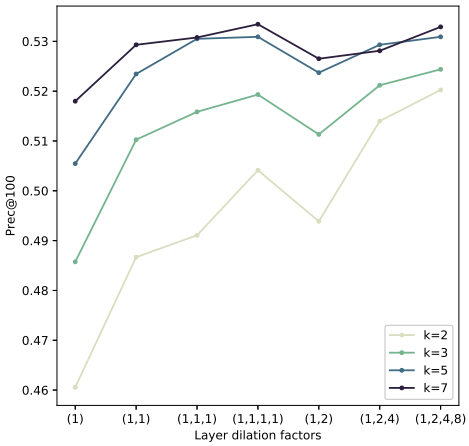
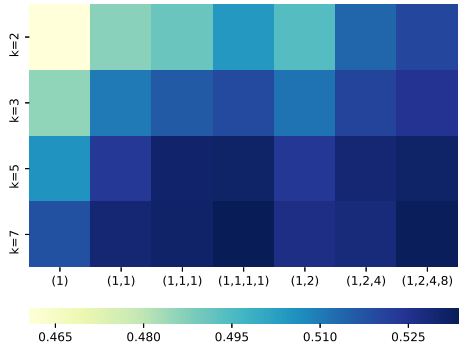


Figure 5.2: Comparison of Prec@100 between POOL and CNN4 models when embedding dimensionality increases on the Movielens dataset.

Finally, keeping all other parameters fixed, we experimented with kernel size and dilation factor, thus seeing the effect of changing the receptive field of the network in two ways. We varied kernel size for $k \in \{2, 3, 5, 7\}$, and layers by both increasing dilation factors with powers of two and without increasing dilation factors. The results can be seen in figure 5.3. From the figures we see that increasing the receptive field, by either using wider kernels or higher dilation factors, has a large effect on the network's predictive performance up to the point where the whole sequence is covered.



(a) Comparison of Prec@100 for CNN models with different kernel sizes as the dilation factor changes.



(b) Heat plot of Prec@100 for kernel sizes $k = 2, 3, 5, 7$ and different dilation factors.

Figure 5.3: Prec@100 for different kernel sizes and changing dilation factors.

5.1.2 Comparing with state-of-the-art

To compare the convolutional neural network for sequence prediction with state-of-the-art, the RecSys dataset was considered. As this dataset was used in [7] with recurrent neural networks, we can easily compare our model to their models and baselines.

We ran a hyperparameter search over POOL and CNN models by randomly selecting combinations of hyperparameters from a predefined sampling space. The sampling space was defined over learning rates (LR), L2-regularisation strength (L2-Reg), activation function (AF), as well as network depth and dilation factors for the CNN network. The batch size was fixed at 1024 if the network depth was smaller than 6 layers and 512 otherwise due to memory constraints. The embedding dimensionality was kept fixed at 64 for memory requirements. Kernel size was taken as 3, thus only increasing the receptive field by having more layers and larger dilation factors. *Adam* [22] was used for optimisation. The hyperparameters of the best performing model on a portion of the training set not used for training, the validation set, were selected and a new model was trained on the full training set. Results on the test set compared to baseline models are summarised in table 5.1.

As seen in table 5.1, the CNN model outperforms POOL in all four metrics. Interestingly, the CNN model also outperforms the RNN model from [7] by a small margin on Prec@20 as well as on MRR@20.

Table 5.1: Baseline results from [7], and in bold results from our experiments on the RecSys dataset.

Model	Prec@20	MRR@20	Prec@100	LR	L2-Reg	AF	Dilations
POP	0.005	0.0012	-	-	-	-	-
MF	0.2574	0.0618	-	-	-	-	-
RNN	0.6322	0.2693	-	-	-	-	-
POOL	0.5070	0.2138	0.7389	$5 \cdot 10^{-4}$	$1 \cdot 10^{-6}$	-	-
CNN	0.6505	0.3089	0.8211	$5 \cdot 10^{-4}$	$1 \cdot 10^{-7}$	tanh	1,2,4,8,16,32

5.2 Two-stage recommender system

For evaluation of the two-stage recommender system, the Sellpy market datasets were the only datasets considered. For the ranking stage, only the full Sellpy market dataset was used as it was the only dataset that had content information, required in the ranking stage.

5.2.1 Candidate generation stage

To select the best performing candidate generation network we ran a hyperparameter search over POOL and CNN models on all Sellpy market datasets by randomly sampling combinations of hyperparameters from a predefined space. We fixed the maximum sequence length to 30, splitting longer sequences and padding shorter sequences with zeros. The sampling space was defined over learning rates (LR), L2-regularisation strength (L2-Reg), embedding dimensionality (ED), as well as network depth, dilation factors, and activation functions (AF) for the CNN. Kernel size was taken as 3. Batch size was taken as 2048 and *Adam* [22] was used as the optimiser. The best performing models on a portion of the training set not used for training, the validation set, were selected. The results of the best models on the test set are shown in table 5.2, their associated parameters are summarised in table 5.3.

As seen in table 5.2, the predictive performance is significantly increased for both models when using the clustered dataset. POOL outperforms CNN by a large margin on the full Sellpy market dataset but the gap becomes smaller when the number of entities is smaller. In the reduced dataset, CNN outperforms POOL on both metrics.

Table 5.2: Candidate generation results for a POOL baseline and a CNN model on the Sellpy market datasets. Ratio is the number of interactions divided by the number of items or clusters, i.e the average number of interactions for each item or cluster.

Dataset	Model	Prec@100	Prec@20	Prec@5	Prec@1	Ratio
Full	POOL	0.3138	0.2249	-	-	≈ 3.85
Full	CNN	0.1824	0.1384	-	-	≈ 3.85
Clustered	POOL	0.5881	0.4047	-	-	≈ 44
Clustered	CNN	0.5429	0.3533	-	-	≈ 44
Reduced	POOL	-	-	0.7188	0.3802	≈ 4412
Reduced	CNN	-	-	0.7457	0.4139	≈ 4412

Table 5.3: Model parameters on the Sellpy market datasets.

Dataset	Model	LR	L2-Reg	ED	AF	Dilations
Full	POOL	$1 \cdot 10^{-2}$	$1 \cdot 10^{-7}$	128	-	-
Full	CNN	$5 \cdot 10^{-3}$	$1 \cdot 10^{-7}$	64	ReLU	1,2,4,8
Clustered	POOL	$1 \cdot 10^{-2}$	$1 \cdot 10^{-7}$	128	-	-
Clustered	CNN	$5 \cdot 10^{-3}$	$1 \cdot 10^{-7}$	64	ReLU	1,2,4,8
Reduced	POOL	$5 \cdot 10^{-3}$	$1 \cdot 10^{-7}$	64	-	-
Reduced	CNN	$1 \cdot 10^{-2}$	$1 \cdot 10^{-7}$	128	tanh	1,2,4,8,16,32

5.2.2 Ranking stage

In the ranking stage, we only considered the full Sellpy market dataset as clusters have no metadata. The input to the ranking stage of the recommender system was based on two sources of data. Recall section 3.3 where the input to the network is defined as the joint feature map of context $\mathbf{x}$ and candidate $\mathbf{y}$. Here we define the context as the average of the item features of the 29 latest item views (the 30th item is the target item), concatenated with the average of the 10 latest item orders at the time of each view. $\mathbf{y}$ is the item features of the candidate item. We make the joint feature map $\phi(\mathbf{x}, \mathbf{y})$ by concatenating $\mathbf{x}$ and $\mathbf{y}$. The dataset was produced by, for each sub-sequence of interactions, retrieving a candidate set of 100 items from the nearest neighbour index of the best candidate generation model, which was shown to be a POOL model. Target items that were not part of the 100 generated candidates were added to the candidate set.

The ranking network was tuned by running a hyperparameter search in a predefined space. Parameters that were varied was network depth and hidden layer sizes as well as learning rate (LR) and L2-regularisation strength (L2-Reg). Batch size was kept fixed at 2048. Discounted cumulative gain was applied as λ on the hinge

loss function, see section 3.3. The best performing model was selected on a portion of the training set not used for training, the validation set. The results on the test set for the best model found during hyperparameter search is summarised in table 5.4.

Table 5.4: Results of the best performing ranking network on the full Sellpy market dataset.

Model	AvgRank	LR	L2-Reg	Hidden layers
Ranking network	26.25/100	$5 \cdot 10^{-3}$	$1 \cdot 10^{-6}$	512,256,128

The best ranking network was compared directly with the candidate generation network. By only considering the $\approx 31\%$ of interactions in the test set where the target item was included in the 100 candidates, we could compare the average rank of the candidate generation network with the ranking network. To produce a ranking with the candidate generation network, the network's output scores of the 100 candidates were sorted. The average rank comparison on the subset of the test set can be seen in table 5.5.

Table 5.5: Average rank comparison between candidate and ranking networks for a subset of the test set.

Model	AvgRank
Candidate network	19.04/100
Ranking network	26.76/100

6
Discussion

This chapter discusses the experimental results presented in the previous chapter.

6.1 CNN for sequence-aware recommendations

Experiments on the Movielens dataset empirically show how predictive performance of CNN models is affected by varying hyperparameters. As seen in figure 5.1a and 5.1b, very shallow networks that are not able to cover the full input sequence perform considerably worse than deeper models with dilations. However, increasing depth and dilation factors only yields an increase in precision to a certain point, an expected result since larger expressiveness increases the likelihood of overfitting. The comparison between CNN4, POOL, and LSTM for varying sequence lengths, figure 5.1c, shows that CNN4 outperforms the other models for all sequence lengths on this dataset. That the POOL model performs worst is expected since it ignores the sequential nature of the data and simply averages the sequences. Thus, as the size of the dataset decreases, the predictive performance of the POOL model drops rapidly. CNN4s predictive performance decays at a slower rate than the other models, suggesting that the network is good at learning from long sequences. That LSTM is also outperformed by CNN4 on all sequence lengths is more surprising and may indicate that the LSTM requires a larger dataset than CNN4. It is also possible that the LSTM parameters could be tuned to reach performance comparable with the CNN.

Experiments on embedding dimensionality, figure 5.2, suggest that increasing the dimensionality yields higher predictive performance for both POOL and CNN models. As higher embedding dimensionality allows models to describe items with more parameters, these results are expected. It should however be noted that higher embedding dimensionality significantly increases training time as well as memory requirements. Since each item is mapped to a vector in the embedding dimensionality, the memory requirement is a product of the number of items in the corpus and the embedding dimensionality. Further, since the embedding dimension is the depth dimension in all convolutional layers, the number of kernels in each layer is also the same as the embedding dimensionality.

The receptive field of the CNN is changed with two parameters in figure 5.3. The impact of using dilation factors to rapidly increase the receptive field without adding more layers, especially for smaller kernels, can be empirically seen in the predictive performance. Differently from recurrent neural networks where memory is built up as the sequence evolves, this design allows control of how much memory

the model is taking up and adds flexibility to control the size of the receptive field which can be beneficial in several settings [11].

Experiments on the RecSys dataset, table 5.1, shows that convolutional neural networks can reach comparable results with state-of-the-art for sequence-aware recommendations. The convolutional network reached better performance than the best recurrent neural network from [7] in both metrics. As convolutional neural networks based on causal dilated convolutions have been shown successful in other sequence-modelling tasks [11], this result further strengthens the use of convolutional neural networks as an alternative to recurrent neural networks for sequence modelling and shows that they can be successfully applied as sequence-aware recommender systems as well.

6.2 Two-stage recommender system

Experiments in the candidate generation phase on the Sellpy market datasets, table 5.2, reveal that the CNN performs worse than the POOL baseline on the full as well as the clustered dataset, directly contradicting the results found on the benchmark datasets. Only for the reduced dataset does CNN achieve better performance than POOL. The poor performance of the CNN model may be explained by the large item vocabulary of the Sellpy market dataset. On the full dataset, the number of interactions in the training set is only $\approx 800,000$ whilst the number of items is $\approx 200,000$. For the clustered dataset the number of entities to choose from is reduced to $\approx 20,000$, which is still relatively high with respect to the number of interactions. When the number of entities to choose from is reduced, the performance gap between POOL and CNN models also decreases, which may suggest that the CNN model requires more data than the POOL model to learn good representations.

The ranking network was compared with the candidate generation network by considering the implicit rank of sequences where the target was included in the 100 predicted candidates, table 5.5. The implicit rank computed with the candidate generation network beat the ranking network with almost 8 positions on average. These results are surprising since the ranking network is trained to rank between a much smaller set of items than the candidate generation network. A possible explanation is that the item features are not descriptive enough for the ranking network to distinguish between similar candidates. The candidate generation network does not suffer from this problem since the embeddings describing the items are learnt during training. To avoid this issue, the embeddings from the candidate generation step could be added as features.

The ranking network is harder to evaluate than the candidate generation network since all that is known is that the rank of the clicked item should be highest. The ranks of all other items do not contribute to the evaluation since users preferences on them are unknown. As the typical task is to display the top-n ranked items to a user, online A/B testing would give a better estimate of the model's performance.

7
Conclusion

Based on the results and the discussion in the previous chapter, this chapter relates back to the research questions and draws conclusions about the findings. Finally, suggestions for potential future work are discussed.

7.1 Research questions

To answer the research questions, a two-stage recommender system was designed, built, and tested. The model provides recommendations in two stages, in the first with a convolutional neural network for sequence-awareness, and in the second with a feedforward neural network for candidate ranking.

The thesis is concluded by reflecting back to the research questions. The main research question was

How can convolutional neural networks be used in sequence-aware recommender systems?

With our two-stage approach, we showed how convolutional neural networks can be used to incorporate sequential data in recommender systems. Although used for candidate generation in the two-stage approach, the results indicate that convolutional networks can perform well as stand-alone recommender systems as well, for instance in session-based recommendations.

The first sub research question was

How does the predictive performance of sequence-aware recommender systems based on convolutional neural networks differ from other sequence-aware models?

The predictive performance of convolutional neural networks was compared to two other models on the Movielens dataset, namely POOL and LSTM. The convolutional neural network was shown to outperform both models for varying sequence lengths on this dataset, suggesting that for some datasets, convolutional neural networks can outperform sequence-aware models based on feedforward or recurrent architectures.

The predictive performance of convolutional neural networks was also compared with a custom recurrent neural network for the task of session-based recommendations on the RecSys dataset. These experiments showed that the convolutional neural network outperformed the recurrent neural network, which architecture has shown state-of-the-art results for session-based recommendations. Thus concluding

that convolutional neural networks can achieve results comparable to state-of-the-art for sequence-aware recommender system tasks.

For domains with very large item corpora, such as in the full Sellpy market dataset, convolutional neural networks showed worse performance than the POOL baseline. The performance of the CNN increased and eventually exceeded the POOL baseline as the number of entities to predict decreased. For sequence-aware recommendations in domains with very large item corpora with respect to the size of the dataset, convolutional neural networks may not be the best approach unless preprocessing such as clustering is applied to reduce the number of entities in the corpus.

Our second sub research question was

> *How can sequence-aware recommender systems account for more information than user-item interactions, such as content features and other sources of information?*

Using a two-stage approach, our architecture showed that content features, as well as additional information, can be incorporated easily in sequence-aware recommender systems by using a second ranking stage. However, the results from the experiments with the ranking network were mixed since the ranking network was not able to outperform the implicit ranking created with the candidate generation network.

7.2 Future work

The convolutional neural network architecture used in this thesis is relatively simple. It could be further extended on by including more advanced deep learning techniques such as dropout and weight normalisation, shown successful in other sequence modelling domains [11]. Adding more advanced techniques could further improve the predictive performance of the network.

Whilst the current candidate generation network only considers collaborative information, the network could also be extended to incorporate item features directly by having embeddings for metadata as in [35]. Using such an architecture would reduce the need for a second ranking network. On the negative side, using such an architecture would also increase the model complexity as well as memory requirements and running time.

In order for a ranking network to produce good results in a production recommender system, a different optimisation objective or dataset than the one used in this thesis would possibly need to be used. As the network's task is to rank while incorporating long-term relationships, a possibility would be to optimise for next buy instead of next item view.

Although theory suggests that convolutional networks are faster than recurrent networks, empirically evaluating the time performance for sequence-aware recommendations during both training and prediction would be an interesting study and could further motivate the use of convolutional networks instead of recurrent networks in time-critical domains such as recommender systems.

Bibliography

[1] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "WaveNet: A generative model for raw audio.", in *SSW*, 2016, p. 125.

[2] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions", *IEEE Transactions on Knowledge & Data Engineering*, no. 6, pp. 734–749, 2005.

[3] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets", in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, Ieee, 2008, pp. 263–272.

[4] D. W. Oard and J. Kim, "Implicit Feedback for Recommender Systems", *Proceedings of the AAAI workshop on recommender systems*, pp. 81–83, 1998.

[5] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "BPR: Bayesian personalized ranking from implicit feedback", in *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*, AUAI Press, 2009, pp. 452–461.

[6] M. Quadrana, P. Cremonesi, and D. Jannach, "Sequence-aware recommender systems", *arXiv preprint arXiv:1802.08452*, 2018.

[7] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk, "Session-based recommendations with recurrent neural networks", *arXiv preprint arXiv:1511.06939*, 2015.

[8] Y. K. Tan, X. Xu, and Y. Liu, "Improved recurrent neural networks for session-based recommendations", in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, ACM, 2016, pp. 17–22.

[9] S. Zhang and L. Yao, "Deep Learning based Recommender System: A Survey and New Perspectives", *ACM J. Comput. Cult. Herit. Article*, vol. 1, no. 35, pp. 1–36, 2017, ISSN: 15232867. DOI: 10.1145/nnnnnnn.nnnnnnn.

[10] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. v. d. Oord, A. Graves, and K. Kavukcuoglu, "Neural Machine Translation in Linear Time", *Arxiv*, pp. 1–11, 2016, ISSN: 1610.10099. [Online]. Available: http://arxiv.org/abs/1610.10099.

[11] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling", *arXiv preprint arXiv:1803.01271*, 2018.

[12] C. C. Aggarwal, *Recommender Systems*. Cham: Springer International Publishing, 2016, ISBN: 978-3-319-29657-9. DOI: 10.1007/978-3-319-29659-3.

[Online]. Available: http://link.springer.com/10.1007/978-3-319-29659-3.

[13] ——, "Content-Based Recommender Systems", in *Recommender Systems*, Cham: Springer International Publishing, 2016, pp. 139–166. DOI: 10.1007/978-3-319-29659-3{_}4. [Online]. Available: http://link.springer.com/10.1007/978-3-319-29659-3_4.

[14] R. Burke, "Hybrid web recommender systems", in *The adaptive web*, Springer, 2007, pp. 377–408.

[15] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems", *Computer*, vol. 42, no. 8, pp. 30–37, 2009, ISSN: 00189162. DOI: 10.1109/MC.2009.263.

[16] C. C. Aggarwal, "Neighborhood-Based Collaborative Filtering", in *Recommender Systems*, Cham: Springer International Publishing, 2016, pp. 29–70. DOI: 10.1007/978-3-319-29659-3{_}2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-29659-3_2.

[17] ——, "Model-Based Collaborative Filtering", in *Recommender Systems*, Cham: Springer International Publishing, 2016, pp. 71–138. DOI: 10.1007/978-3-319-29659-3{_}3. [Online]. Available: http://link.springer.com/10.1007/978-3-319-29659-3_3.

[18] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 0387310738.

[19] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, 10. Springer series in statistics New York, NY, USA: 2001, vol. 1.

[20] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[21] J. Kiefer and J. Wolfowitz, "Stochastic Estimation of the Maximum of a Regression Function", *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952, ISSN: 0003-4851. DOI: 10.1214/aoms/1177729392. [Online]. Available: http://projecteuclid.org/euclid.aoms/1177729392.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] K. Cho, B. V. Merrienboer, D. Bahdanau, and Y. Bengio, "On the Properties of Neural Machine Translation : Encoder – Decoder Approaches", *Ssst-2014*, pp. 103–111, 2014. DOI: 10.3115/v1/W14-4012.

[25] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions", *arXiv preprint arXiv:1511.07122*, 2015.

[26] J. Weston, S. Bengio, and N. Usunier, "WSABIE: Scaling up to large vocabulary image annotation", in *IJCAI International Joint Conference on Artificial Intelligence*, 2011, pp. 2764–2770, ISBN: 9781577355120. DOI: 10.5591/978-1-57735-516-8/IJCAI11-460.

[27] S. Rendle and C. Freudenthaler, "Improving pairwise learning for item recommendation from implicit feedback", in *Proceedings of the 7th ACM international conference on Web search and data mining - WSDM '14*, 2014, pp. 273–

282, ISBN: 9781450323512. DOI: 10.1145/2556195.2556248. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2556195.2556248.

[28] P. Covington, J. Adams, and E. Sargin, "Deep Neural Networks for YouTube Recommendations", in *Proceedings of the 10th ACM Conference on Recommender Systems - RecSys '16*, 2016, pp. 191–198, ISBN: 9781450340359. DOI: 10.1145/2959100.2959190. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2959100.2959190.

[29] T. X. Tuan and T. M. Phuong, "3D Convolutional Networks for Session-based Recommendation with Content Features", in *Proceedings of the Eleventh ACM Conference on Recommender Systems - RecSys '17*, 2017, pp. 138–146, ISBN: 9781450346528. DOI: 10.1145/3109859.3109900. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3109859.3109900.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[31] A. Agarwal, I. Zaitsev, and T. Joachims, "Counterfactual Learning-to-Rank for Additive Metrics and Deep Models", *arXiv preprint arXiv:1805.00065*, 2018.

[32] D. Ben-Shimon, A. Tsikinovsky, M. Friedmann, B. Shapira, L. Rokach, and J. Hoerle, "Recsys challenge 2015 and the yoochoose dataset", in *Proceedings of the 9th ACM Conference on Recommender Systems*, ACM, 2015, pp. 357–358.

[33] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context", *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, p. 19, 2016.

[34] M. Kula, *Spotlight*, https://github.com/maciejkula/spotlight, 2017.

[35] ——, "Metadata embeddings for user and item cold-start recommendations", *arXiv preprint arXiv:1507.08439*, 2015.