Knowledge-Based Systems 206 (2020) 106372

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Recommender system using Long-term Cognitive Networks

Gonzalo Nápoles [a,b,*], Isel Grau [c], Yamisleydi Salgueiro [d]

[a] Faculty of Business Economics, Hasselt University, Belgium
[b] Department of Cognitive Science & Artificial Intelligence, Tilburg University, The Netherlands
[c] Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Belgium
[d] Department of Computer Science, Faculty of Engineering, Universidad de Talca, Campus Curicó, Chile

ARTICLE INFO

Article history:
Received 29 March 2020
Received in revised form 2 August 2020
Accepted 3 August 2020
Available online 7 August 2020

Keywords:
Recommender system
Prior knowledge
Long-term Cognitive Networks

ABSTRACT

In this paper, we build a recommender system based on Long-term Cognitive Networks (LTCNs), which are a type of recurrent neural network that allows reasoning with prior knowledge structures. Given that our approach is context-free and that we did not involve human experts in our study, the prior knowledge is replaced with Pearson's correlation coefficients. The proposed architecture expands the LTCN model by adding Gaussian kernel neurons that compute estimates for the missing ratings. These neurons feed the recurrent structure that corrects the estimates and makes the predictions. Moreover, we present an extension of the non-synaptic backpropagation algorithm to compute the proper non-linearity of each neuron together with its activation boundaries. Numerical results using several case studies have shown that our proposal outperforms most state-of-the-art methods. Towards the end, we explain how can we inject expert knowledge to the proposed neural system.

1. Introduction

Recommender Systems (RSs) play a key role in today's Business Intelligence as they allow companies to increase their revenue while also offering personalized suggestions. At the same time, when properly implemented, users often appreciate being guided through thousands of options. RSs address the information overload problem by using machine learning methods to offer users with suitable recommendations on items based on users' history and data [1]. For example, this can be done by considering the previous experience of each user and the preference of other users sharing similar interests.

When it comes to algorithms, existing RSs can be gathered into three broad categories [2]: context-aware [3,4], collaborative [5–7] and hybrid models [1,8,9]. In the first case, the RS learns a profile of new user's interests based on the features describing the items the user has rated. In the second case, the RS learns to recognize patterns in a user–item matrix to provide a recommendation. This approach is based on the assumption that users who agreed in the past will agree in the future, thus they will probably prefer items that users with similar preferences have liked in the past. Collaborative filtering (CF) methods can be further divided into two sub-categories: model-based [10–12] and memory-based models [13,14]. Finally, hybrid RSs combine two or more algorithms with the aim of improving the overall performance. This paper contributes to the CF approach, thus Section 2 will further elaborate on prominent CF methods reported in the literature.

A common limitation of existing RSs is that they do not allow for explicit knowledge injection. This means that human experts (e.g., company managers, marketing experts) cannot modify the model to adapt it when new trends – which are not reflected on the data – emerge. The same problem arises with the bias on historical data records: if we let the model learn only from data, the model will reflect what happened in the past, not necessarily what happens in the present. Both situations could be addressed by allowing for human–machine interaction. However, this is not trivial since we often rely on intelligent algorithms to find patterns that human beings cannot discover. This issue becomes more complex as most successful models often perform as black-boxes, so we barely understand where to insert the expert knowledge.

In this paper, we propose an RS using Long-term Cognitive Networks (LTCNs) [15] that operates on the user–item rating matrix. LTCNs are recurrent neural networks that allow the experts to inject prior knowledge through the weight matrix. Overall, LTCNs exhibit four key characteristics that distinguish them from other neural systems: (i) each neuron maps to a problem feature, thus explicit hidden neurons are not allowed, (ii) the weight matrix can be given by the domain experts, (iii) each neuron has its own transfer function, and (iv) the learning focuses on adjusting the proper non-linearity degree of each neuron instead of modifying the weight matrix. It is worth mentioning that, since our

* Corresponding author at: Department of Cognitive Science & Artificial Intelligence, Tilburg University, The Netherlands.
E-mail address: gonzalo.napoles@uhasselt.be (G. Nápoles).

https://doi.org/10.1016/j.knosys.2020.106372

research did not involve human experts, the prior knowledge will be replaced with Pearson's correlation coefficients. Section 7 will provide concrete guidelines on how to define the prior knowledge structures. Similarly, we will further discuss the positive impact of operating on previously defined knowledge structures and the remaining challenges to be addressed.

Overall, this paper brings two contributions to life. Firstly, we propose an LTCN-based neural architecture that expands the original model with Gaussian kernel neurons. These neurons compute estimates for the missing ratings in a user–item matrix, thus feeding the LTCN with complete (probably inexact) patterns. Next, these estimates are corrected by the LTCN when computing the final recommendations. As a second contribution, we propose a three-step learning procedure aimed at computing the weight matrix (in case that the prior knowledge is not available), training the Gaussian kernel neurons and adjusting the non-linearity of the remaining neurons. To do the latter, we propose a new variant of the non-synaptic backpropagation (NSBP) algorithm [15] that also optimizes the boundaries of the activation space of each LTCN neuron.

The rest of this paper is organized as follows. Section 2 revises the literature on CF methods, which do not use explicit information. Section 3 introduces the theoretical basis of the LTCN model. Section 4 presents the LTCN-based RS, while Section 5 describes the three-step learning procedure to compute the learnable parameters. Section 6 evaluates the performance of our model by using three popular case studies. Finally, Section 7 provides guidelines on how the define the prior knowledge structures, while Section 8 concludes the paper.

2. Related work on collaborative filtering

As previously mentioned, RSs aim at forecasting users' interests in order to suggest services or products. With the e-commerce extension and the data generation explosion, the application of RSs on the Internet has expanded, thus facilitating their use in several areas [16]. Likewise, such systems have become a key component within nowadays' business intelligence since they help companies better position their products or services [17]. In this section, we elaborate on the related works on RSs that relate to CF-based solutions.

CF methods are the most widely used approach for RSs [18]. This category comprises methods based on past interactions between users and items to produce new recommendations. Unlike content-based methods, CF is recommended in areas where no explicit information or "features" are available, or they are difficult to process, such as opinions [18].

There are two broad subcategories of CF methods: memory-based and model-based. In memory-based methods, users and items are represented directly by their interaction, and recommendations are made following the nearest neighbors' information (user–user or item–item). Therefore, these methods rely on computing similarities between users (or items) through their assigned ratings [19]. The k-nearest neighbor method [20] is the most used memory-based method due to its efficiency and implementation simplicity [21].

Within the model-based methods, the preference of users on a pool of items is encoded as a user–item interaction matrix. No further information about the users or items is used, thus recommendations are made following the model's information. Examples of these approaches are matrix factorization (MF) [22], singular value decomposition (SVD), autoencoders (AEs) [23], and variational autoencoders (VAEs) [24].

MF-based models transform both users and items into a joint latent factor space and use the inner product of this space to reflect user–item interactions [22]. Another type of MF for latent model generation is SVD [25].

Deep learning models [26] are integrated by multiple neural building blocks as a single differentiable function and trained end-to-end. These methods capture non-linear and non-trivial users–items' interactions allowing the encoding of more complex abstractions from data [1]. The use of deep learning-based RSs has become more popular due to several successes in providing high-quality recommendations [18].

AE-based models [23] are deep unsupervised neural networks trained to reconstruct a given pattern, which is the outcome of the output layer. The golden rule when designing an AE is that the size of the hidden layers decreases until reaching the desired number of latent features. After that, the size of the remaining hidden layers starts to increase until reaching the original number of features. They are powerful in handling noisy data, however, the training process is expensive due to the high parameter tuning [18]. VAE-based models are a recent advancement in the AE-based family. These neural networks are probabilistic generative models in the form of AEs whose training is regularized to avoid overfitting [24].

One of the issues of CF-based recommenders is the data sparsity. This arises when there is insufficient data on user and item interactions, thus affecting the accuracy and efficiency of RSs [27]. One issue attached to this problem is how to extrapolate unknown ratings from the known ones. In some papers, the interaction matrix is transformed into a binary matrix. They treat observed and unobserved values as ones and zeros, respectively. This approach fails to accurately reflect the real world. It is challenging to describe users' preferences using just one or zero [11]. By binarizing the interaction matrix, this approach neglects the ratings of each user and their actual preferences while adding noise to the data. Also, by replacing the missing interactions with zero, the similarity measures treat these values as negative. In this context, another approach is the Pearson correlation, which treats missing values as average. This strategy allows handling both "difficult" and "easy" evaluators since it normalizes the items' evaluations to a certain extent.

In terms of efficiency, CF-based methods often struggle to process large datasets. The integration of hashing into CF-based methods is deemed an interesting solution to this problem. The rationale behind this technique is to learn a low-dimensional binary vector representation of the interaction matrix [28]. The binary code representation reduces the storage requirement for large-scale RSs (each element only needs one bit), thus reducing the cost of querying as the similarity calculation lies in the Hamming space. The efficiency improvement comes with the sacrifice of accuracy due to a large amount of information loss caused by a quantization procedure [29]. In recently published papers [28,30], the authors expand deep learning frameworks to learn binary code (hash) for CF solutions. However, they recognize that accuracy is still an issue when compared to traditional methods reported in the literature.

Despite their success, CF methods still have space for improvements. The cold-start problem, data sparsity, decisions' interpretability, and trade-off between bias and variance are among the open challenges. Moreover, most of these methods do not allow for human–machine interaction. Therefore, new approaches with fewer parameters, which are interpretable (to some extent), and able to handle missing data while maintaining the performance are welcome.

3. Long-term cognitive networks

In a nutshell, LTCNs are recurrent neural networks in which neurons have a well-defined meaning for the system being modeled [15]. This implies that neither explicit hidden layers nor neurons are allowed. However, the recurrent reasoning mechanism of LTCNs does produce a sort of *abstract layer* in each

iteration, each containing the activation values of neurons in that iteration. Notice however that the weights connecting the abstract layers do not change from an iteration to another. As a matter of fact, the weight matrix of LTCNs is expected to be provided by domain experts as prior knowledge, therefore allowing for the human–machine interaction.

Eq. (1) shows how to compute the activation value of the ith neural concept in each abstract layer (iteration in the recurrent reasoning) for the kth initial stimulus,

$$a_i^{(t+1)}(k) = f_i^{(t+1)}\left(\sum_{j=1}^{M} w_{ji} a_j^{(t)}(k)\right) \tag{1}$$

where M is the number of neurons, w_{ji} is the weight connecting the neuron C_j with C_i, $1 \leq t \leq T$ is the current abstract layer and T is the number of abstract layers. Eq. (2) shows the (generalized) sigmoid transfer function used to limit the neuron's activation value to the $[L_i, U_i]$ interval,

$$f_i^{(t)}(x) = L_i + \frac{U_i - L_i}{(1 + e^{-\lambda_i^{(t)}(x - h_i^{(t)})})^{1/v_i^{(t)}}} \tag{2}$$

such that $\lambda_i > 0$, $h_i \in \mathbb{R}$ and $v_i > 0$ are parameters to be adjusted during the nonsynaptic learning phase (discussed in Section 5.3). In this function, λ_i denotes the function slope, h_i stands for the sigmoid offset and v_i regulates toward which asymptote the maximum growth occurs.

4. LTCN-based recommender system

In this section, we present an LTCN-based RS where each neural entity denotes an item. The proposed model involves two types of neurons and connections. On one hand, we have Gaussian kernel neurons and sigmoid processing neurons. On the other hand, we have non-recurrent links connecting the Gaussian kernel neurons and recurrent links connecting the sigmoid neurons. Overall, the proposed LTCN-based RS involves two sub-networks that are connected sequentially such that the first network processes the rough input sequences while the second one produces the recommendations.

Fig. 1 shows the architecture of the proposed LTCN-based system for three items. The reader can notice that the fact that each neuron is mapped to a specific item makes our model quite interpretable. In the context of this paper, interpretability means that we understand where to insert the domain knowledge rather than elucidating how the recommendations were computed. Although such explanations can perfectly be extracted, it goes beyond the scope of this paper.

In this architecture, Gaussian kernel neurons receive the incomplete input sequences to be processed. The empty positions mark the items for which we need to compute the recommendations. Gaussian kernel neurons are computed using the ratings received by each item and the ratings provided by each user as explained in Section 5.2. These input neurons are connected through *non-recurrent links* so that they will be updated only once after being activated. Eq. (3) shows the information such granular neurons exchange.

The second sub-network consists of sigmoid neurons connected through *recurrent links* such that the neuron's activation values are updated in each iteration. Although the LTCN model in this sub-network could complete the sequence without the need for kernel neurons, it will likely need more iterations to produce acceptable outputs. The less information we have, the more iterations we would need for the patterns to emerge. Overall, increasing the number of iterations could cause problems when adjusting the non-linearity of the model (see Section 5.3) since

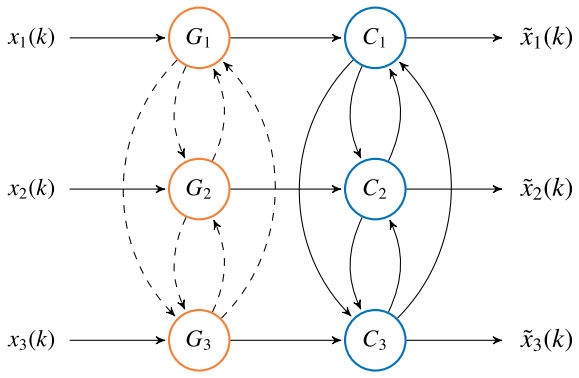


Fig. 1. LTCN-based RS for three items such that $x_i(k)$ is the known recommendation for the ith item according to the k user while $\tilde{x}_i(k)$ is the recommendation computed by the network for that item. Moreover, G_1, G_2 and G_3 are Gaussian kernel neurons connected with non-recurrent links (denoted with dashed arcs) whereas C_1, C_2 and C_3 are sigmoid neurons connected with recurrent links (denoted with solid arcs). If $x_i(k)$ is known, then the network should ensure that $x_i(k) \approx \tilde{x}_i(k)$ after performing the inference process.

the gradient tends to either vanish or explode when operating with very complex dependencies.

The reader can argue that, in the first sub-network, we could have transformed the non-recurrent links into recurrent ones. Such a strategy, however, would have implied to estimate the Gaussian kernels in each iteration, thus increasing the computational burden. Besides, we should keep in mind that this sub-network will not benefit from the non-synaptic learning and that the role of the second sub-network is to correct the rough estimates computed with the first one.

Each kernel neuron G_i involves two functions: a Gaussian kernel density estimator $\mathcal{G}_i(\cdot)$ and its inverse $\mathcal{G}_i^{-}(\cdot)$. The first function is used to determine the probability of the ith item of receiving a certain rating, while its inverse allows estimating the item's rating for a given probability. As mentioned, these functions are computed from data during the learning procedure described in the following Section 5.

Eq. (3) shows how to compute the activation value of a kernel neuron, which will be adopted to feed the sub-network containing the recurrent links,

$$a_i^{(0)}(k) = \mathcal{G}_i^{-}\left(\sum_{j=1}^{M} \frac{\delta_j(k)\mathcal{G}_j(x_j(k))}{N(k)}\right) \tag{3}$$

such that $\delta_j(k)$ is a binary function that returns one when the jth item was rated by the kth user (i.e., the $x_j(k)$ value is not missing), and $N(k)$ denotes the number of non-missing values in the kth instance. These kernel functions return values in the [0, 1] interval such that they compute estimates of the normalized ratings for each item being analyzed.

It is relevant to mention that kernel neurons are only activated when the neuron receives a missing value, otherwise they will transfer the non-missing value to the recurrent architecture. This sub-network will correct the rough estimates computed by the kernel neurons using (i) the prior knowledge encoded into the weight matrix, and (ii) the non-linear sigmoid functions learned from the historical data. Moreover, the fact that kernels neurons are independent of each other suggests that we could easily paralyze their computation, thus notably boosting algorithm's speed when processing larger datasets.

5. Three-step learning procedure

In this section, we describe the necessary steps to train the LTCN-based recommender system.

5.1. Computing the weights matrix

The first learning step is devoted to estimating the weight matrix. Ideally, the domain experts should provide (some of) those weights to be used as prior knowledge. This would not only allow obtaining reliable predictions with less effort but also reasoning with pieces of knowledge that have not yet been observed in the historical data. If this knowledge is not available, then the LTCNs replace the weight matrix with the correlation coefficients associated with the correlation among the variables (i.e., the items to be recommended to the users).

Eq. (4) displays how to calculate the weights connecting the neurons, which are equivalent to Pearson's coefficients in multiple regression models,

$$w_{ji} = \frac{K\sum_k x_i(k)x_j(k) - \sum_k x_i(k)\sum_k x_j(k)}{K(\sum_k x_j(k)^2) - (\sum_k x_j(k))^2} \tag{4}$$

where $x_i(k)$ represents the value of the ith variable according to the kth instance, while K is the number of instances (that is to say the number of users) in the dataset.

Notice that this strategy might lead to dense networks, which might not be a realistic representation of the knowledge that domain experts would have provided. Moreover, having densely connected networks makes the sigmoid neurons to be more likely to be saturated (i.e., their activation values quickly move toward either L_i or U_i). To overcome this situation, we will prune the network by only considering those weights connecting highly-correlated items as indicated by a user-specified (absolute) correlation threshold $0 \leq \xi \leq 1$.

It should be stated that this training step is performed only once and that the computed weight matrix will not be altered in the subsequent learning steps.

5.2. Computing the kernel neurons

The second learning step consists in "training" the kernel neurons. In practice, this means that we are going to estimate the probability density function of each variable. This strategy attempts to feed the network with estimates that replace the missing values (e.g., non-rated items) in a given instance. This can be done by computing a Gaussian kernel density estimator – which is a non-negative function – for each variable. This non-parametric procedure is a fundamental data smoothing problem where inferences about the population are made, based on a finite data sample. Eq. (5) formalizes the Gaussian kernel associated with the ith neuron,

$$\mathcal{G}_i(y) = \sum_{x_i(k)\in X_i} e^{-\frac{(y-x_i(k))^2}{2s^4}} \tag{5}$$

such that X_i represents the set of users who have rated the ith item, while $s > 0$ is a smoothing parameter called the *bandwidth*, which establishes a trade-off between bias and variance. A large bandwidth leads to a density distribution with high bias, while a small bandwidth leads to a density distribution with high-variance. Since our neural system operates with normalized ratings, we have used a bandwidth equal to 0.05 in all simulations conducted in this paper.

As mentioned, when performing the reasoning process for a given instance, the known variable values are evaluated in their respective kernels such that we obtain a set of probability values. These values are averaged to obtain the average probability of that user liking any item. To obtain the values that replace the missing values in the given instance, we just evaluate the average probabilities that characterize the user's rating behavior in the inverse of each kernel function. Notice that several solutions might exist. Selecting the smallest one would imply that our neural system is being conservative while the greatest one would imply that our system is being optimistic. This paper will adopt the former approach.

5.3. Adjusting the non-linearity

This step is based on the nonsynaptic learning principle [31] which aims at adjusting the non-linearity degree of each sigmoid neuron. As mentioned, the NSBP algorithm [15] assumes that the weights are known, thus the target parameters are the ones controlling the sigmoid function shape.

If the weights are provided by the experts, then the NSBP would allow for the human–machine interaction. In contrast, if the weights are estimated based on the correlation among the items (which is the approach adopted in this paper) then the NSBP algorithm would still preserve the semantics behind the model. This is a valuable feature in RSs since users have the tendency of relying on systems that are able to explain how the recommendations were made.

Next, we present a variant of the NSBP algorithm that additionally optimizes the L_i and U_i parameters (see Eq. (2)). These parameters define the activation interval for the ith neuron. This modification improves the algorithm by adding more flexibility, thus alleviating the stringent (yet desirable) limitation of operating over fixed knowledge structures. Overall, the learning task consists in adjusting the shape of the transfer function associated with the ith neuron in each iteration. This can be done by computing the parameter set:

$$\Theta = \left\{\theta_i^{(t)} = \left(\lambda_i^{(t)}, h_i^{(t)}, v_i^{(t)}, L_i^{(t)}, U_i^{(t)}\right)\right\}. \tag{6}$$

Eq. (7) shows the loss (error) function to be minimized by the NSBP learning algorithm:

$$\mathcal{E}(\Theta) = \sum_{i=1}^{M} \frac{\delta_i(k)\left(x_i(k) - a_i^{(t)}(k)\right)^2}{2} \tag{7}$$

where $\delta_i(k)$ takes one when the ith variable of the kth training instance is not missing, otherwise it takes zero. In addition, $a_i^{(t)}(k)$ represents the activation value of the ith neuron in the tth abstract layer as calculated in Eq. (1).

Similarly to other backpropagation methods, the NSBP algorithm starts the parameter updating backwards. By doing that, we need to determine whether the current iteration t is the final one or not, thus leading to the following scenarios.

Case 1. When $t = T$, the partial derivative of the global error $\mathcal{E}(\Theta)$ is computed as follows:

$$\frac{\partial \mathcal{E}}{\partial a_i^{(t)}(k)} = -\delta_i(k)(x_i(k) - a_i^{(t)}(k)) \tag{8}$$

where $a_i^{(t)}(k)$ is the activation value of the ith neuron in the tth abstract layer, while $x_i(k)$ is the known value of the ith variable according to the kth training example.

Case 2. When $1 < t < T$, the partial derivative of the global error $\mathcal{E}(\Theta)$ is calculated as follows:

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial a_i^{(t)}(k)} &= \sum_{j=1}^{M} \frac{\partial \mathcal{E}}{\partial a_j^{(t+1)}(k)} \times \frac{\partial a_j^{(t+1)}(k)}{\partial a_i^{(t)}(k)} \\ &= \sum_{j=1}^{M} \frac{\partial \mathcal{E}}{\partial a_j^{(t+1)}(k)} \times \frac{\partial a_j^{(t+1)}(k)}{\partial \bar{a}_j^{(t+1)}(k)} \times \frac{\partial \bar{a}_j^{(t+1)}(k)}{\partial a_i^{(t)}(k)} \\ &= \sum_{j=1}^{M} \frac{\partial \mathcal{E}}{\partial a_j^{(t+1)}(k)} \times \frac{\partial a_j^{(t+1)}(k)}{\partial \bar{a}_j^{(t+1)}(k)} \times w_{ij}\end{aligned} \tag{9}$$

where $\bar{a}_j^{(t+1)}(k)$ is the raw value of the jth neuron,

$$\frac{\partial a_j^{(t+1)}(k)}{\partial \bar{a}_j^{(t+1)}(k)} = \frac{\left(U_j - L_j\right)\lambda_j^{(t+1)}\Gamma_j^{(t+1)}(k)}{v_j^{(t+1)}\left(1+\Gamma_j^{(t+1)}(k)\right)^{1+1/v_j^{(t+1)}}} \tag{10}$$

and

$$\Gamma_j^{(t+1)}(k) = e^{\lambda_j^{(t+1)}\left(-\bar{a}_j^{(t+1)}(k)+h_j^{(t+1)}\right)}. \tag{11}$$

Once $\partial\mathcal{E}/\partial a_i^{(t)}(k)$ have been calculated, we need to obtain the partial derivatives of the global error with respect to the target parameters $\theta_i^{(t)}(p) \in \Theta$,

$$\frac{\partial\mathcal{E}}{\partial\theta_i^{(t)}(p)} = \frac{\partial\mathcal{E}}{\partial a_i^{(t)}(k)} \times \frac{\partial a_i^{(t)}(k)}{\partial\theta_i^{(t)}(p)}, \tag{12}$$

such that p is the index of the parameter of the ith sigmoid function in the current abstract layer.

Eqs. (13) to (17) portray the partial derivatives of the neuron's activation value with respect to each sigmoid function parameter in the current iteration,

$$\frac{\partial a_i^{(t)}(k)}{\partial\theta_i^{(t)}(1)} = \frac{\left(U_i - L_i\right)\Gamma_i^{(t)}(k)\left(\bar{a}_i^{(t)}(k) - h_i^{(t)}\right)}{v_i^{(t)}\left(1+\Gamma_i^{(t)}(k)\right)^{1+1/v_i^{(t)}}}, \tag{13}$$

$$\frac{\partial a_i^{(t)}(k)}{\partial\theta_i^{(t)}(2)} = \frac{-\left(U_i - L_i\right)\Gamma_i^{(t)}(k)\lambda_i^{(t)}}{v_i^{(t)}\left(1+\Gamma_i^{(t)}(k)\right)^{(1+1/v_i^{(t)})}}, \tag{14}$$

$$\frac{\partial a_i^{(t)}(k)}{\partial\theta_i^{(t)}(3)} = \frac{\left(U_i - L_i\right)\log\left[1+\Gamma_i^{(t)}(k)\right]}{\left(v_i^{(t)}\right)^2\left(1+\Gamma_i^{(t)}(k)\right)^{1/v_i^{(t)}}}, \tag{15}$$

$$\frac{\partial a_i^{(t)}(k)}{\partial\theta_i^{(t)}(4)} = 1 - \left(\Gamma_i^{(t)}(k)\right)^{-1/v_i^{(t)}}, \tag{16}$$

$$\frac{\partial a_i^{(t)}}{\partial\theta_i^{(t)}(5)} = \left(\Gamma_i^{(t)}(k)\right)^{-1/v_i^{(t)}}. \tag{17}$$

Eq. (18) shows the gradient vector used to updated the sigmoid function parameters attached to each neural processing entity in the tth abstract layer,

$$\nabla_\Theta^{(t)}\mathcal{E} = \left(\frac{\partial\mathcal{E}}{\partial\theta_1^{(t)}(1)}, \ldots, \frac{\partial\mathcal{E}}{\partial\theta_i^{(t)}(p)}, \ldots, \frac{\partial\mathcal{E}}{\partial\theta_M^{(t)}(P)}\right). \tag{18}$$

With respect to the sigmoid parameter initialization, we use the following values for all layers: $\lambda_i = v_i = 1.0$, $U_i = max_i\{M_i\}$ and $L_i = -U_i$ with M_i being the number of incoming connections to the ith neuron after performing the network pruning, while h_i is given as follows:

$$h_i = \sum_{j=1}^{M} \frac{\gamma_\xi(x_j, x_i)\phi_{ji}}{M_i} \tag{19}$$

where $\gamma_\xi(x_j, x_i)$ is a binary function that returns one if the absolute correlation value between variables x_j and x_i exceeds the correlation threshold ξ, and

$$\phi_{ji} = \frac{\sum_k x_j(k)^2 \sum_i x_i(k) - \sum_k x_i(k)x_j(k)\sum_k x_j(k)}{K(\sum_k x_j(k)^2) - (\sum_k x_j(k))^2}. \tag{20}$$

This method needs to establish some constraints to produce feasible parameter values. For example, we need to ensure that $\lambda_i^{(t)} > 0$ and $v_i^{(t)} > 0$, and that $L_i^{(t)} \leq U_i^{(t)}$. The easiest way to enforce these constraints is to preserve the actual parameter value if the new one does not fulfill the constraints. Of course, other alternatives are also possible.

6. Numerical simulations

In this section, we will explore the overall accuracy of our system on three case studies.

6.1. State-of-the-art algorithms

In this subsection, we describe the state-of-the-art algorithms used for comparison purpose. Aiming at performing fair comparisons, we have selected methods that compute the recommendations based on the user–item rating matrix, thus they do not use explicit (context) information. The selected algorithms are: the k-nearest neighbor (kNN), a matrix factorization (MF), the SVD algorithm, a neural network autoencoder (AE), and a variational autoencoder (VAE). No algorithm performs hyperparameter tuning since it would increase significantly the time required to build an optimal model.

The simulations conducted in this section use the top-50, the top-200 and the top-500 items for all problems. The parameter settings described next take the dataset size into consideration. The number of latent features in both the MF and SVD models is set as the 20% of the number of items. The neural autoencoders use three hidden layers such that the number of hidden neurons in each layer is set as the 40%, 20% and 40% of the number of items, respectively. The number of neighbors k in the kNN algorithm is set to 10, while the cosine distance has been adopted as the dissimilarity functional.

In the case of the LTCN model, we used three abstract layers ($T = 3$) while the absolute correlation threshold ξ establishing whether or not a weight is deemed relevant was set to 0.3. We ensure that no neuron is isolated, thus each neuron will receive at least one incoming connection (the one having the largest absolute correlation value). Moreover, all optimization-based algorithms uses the Mean Squared Error (MSE) as the loss function, with exception of the VAE which additionally includes a penalization component based on the Kullback–Leibler divergence. Finally, we use the ADAM method [32] with learning rate equal to 0.002, such that the batch size equal to 128 and the number of epochs is equal to 50. It goes without saying that other parameter settings are also possible.

6.2. Case study 1: anime

The first case study concerns with anime recommendation.[1] This dataset contains the ratings (integers between 1 and 10) of 73,516 users on a pool of 12,294 anime. However, a closer inspection to the data shows that there are only 500 anime which are frequently evaluated. Fig. 2 shows the relative (decreasing) frequency of those top-500 anime.

After normalizing the ratings, we create three datasets with the top-50, top-200 and top-500 anime. This will serve to evaluate the scalability of our model as we expect the LTCNs' accuracy to deteriorate as the number of items increase (after all, the inner knowledge of our model relies on the Pearson's correlation). Also, we removed the users who have less than 10 ratings, thus the number of users in each dataset is 34,764, 47,503 and 51,824, respectively. The training set in all cases will contain 80% of the users while the remaining 20% will be used to test the accuracy of the recommendations.

Tables 1, 2 and 3 display both the training and test errors reported by each model on the anime-50, anime-200 and anime-500 datasets, respectively. The lowest test errors is highlighted in bold. Notice that our model notably outperforms the remaining ones for the anime-50 and the anime-500 datasets, while it

[1] https://www.kaggle.com/CooperUnion/anime-recommendations-database.

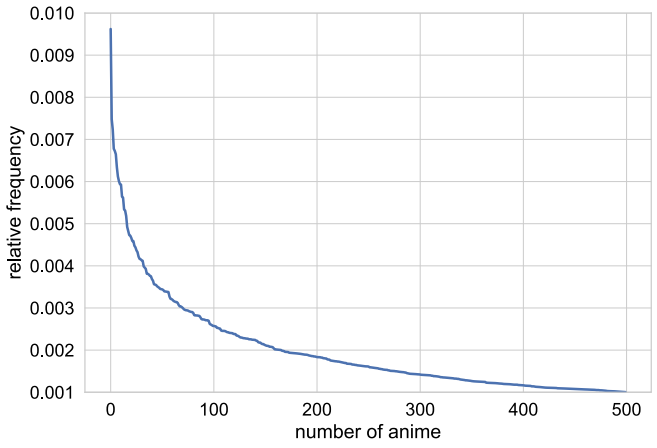


Fig. 2. Relative frequency of the top-500 anime.

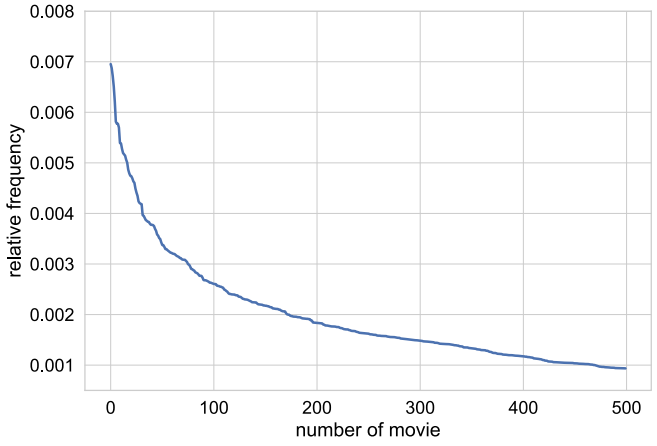


Fig. 3. Relative frequency of the top-500 movies.

Table 1
Training and test MSE on the anime dataset with the top-50 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0187	0.0934	0.0136	0.0182	0.0087
Test	0.0387	0.0349	0.1480	0.0136	0.0179	**0.0085**

Table 2
Training and test MSE on the anime dataset with the top-200 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0263	0.0895	0.0124	0.0188	0.0134
Test	0.0903	0.0397	0.1466	0.0137	0.0190	**0.0132**

Table 3
Training and test MSE on the anime dataset with the top-500 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0354	0.0860	0.0110	0.0191	0.0148
Test	0.1333	0.0513	0.1463	0.0159	0.0192	**0.0145**

performs comparably to the AE for anime-200. There is also a distinction on the number of trainable parameters (excluding the kernel neurons). For example, in the case of anime-50 the AE involves 2500 parameters while our model needs to adjust only 750 parameters. In the case of the MF, the number of parameters is 260,060, even when the parameter setting seems to be reasonable for a system with 50 items.

For this case study, we can conclude that our model performs better than the selected state-of-the-art methods on the top-50 and top-500 anime, while being competitive in performance with the AE model for anime-200.

Table 4
Training and test MSE on the anime dataset with 50 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0100	0.0869	0.0147	0.0175	0.0031
Test	0.0656	0.0416	0.1532	0.0152	0.0181	**0.0032**

Table 5
Training and test MSE on the anime dataset with 200 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0238	0.0838	0.0122	0.0191	0.0095
Test	0.1246	0.0456	0.1528	0.0139	0.0195	**0.0097**

Table 6
Training and test MSE on the anime dataset with 500 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0784	0.0333	0.0110	0.0192	0.0130
Test	0.1530	0.1503	0.0469	0.0157	0.0196	**0.0132**

However, operating on the top-*k* items could accentuate the cold-start problem. Therefore, it would be interesting to evaluate the performance on randomly selected items. For the next experiments, we randomly select the items from a distribution based on their frequency.

Tables 4, 5 and 6 display both the training and test errors reported by each model on the anime datasets, with 50, 200, and 500 randomly selected items, respectively. Similarly to the previous experiment, we removed the users who have less than 10 ratings, thus the number of users in each dataset is 10,054, 36,371 and 47,582, respectively.

The results show that our model outperforms the state-of-the-art for all datasets when using randomly selected items, even obtaining better results than the AE model.

6.3. Case study 2: movielens-10M

The second case study is movielens-10M[2] which contains 10 million ratings applied to 10,000 movies by 72,000 users. Similarly to the previous case study, a closer inspection to the data shows that there are about 500 movies which are often rated by the users. Fig. 3 shows the relative (decreasing) frequency of those top-500 movies.

Aiming at creating the datasets with the top-50, top-200 and top-500 movies, we normalized the ratings and removed the users with less than 10 ratings. This resulted in three datasets (movie-50, movie-200 and movie-500) with 44,830, 62,241 and 67,847 users, respectively. The training sets and the test sets were created as explained before.

Tables 7, 8 and 9 show both the training and test errors reported by each method on the movie-50, movie-200 and movie-500 datasets, respectively. The best results are highlighted in bold. In this case study, the proposed model stands as the best-performing algorithm for the movie-50 dataset, while ranking second for movie-200 and movie-500.

This case study confirms the superiority of our model when operating on the top-50 items. However, the results do not deteriorate as much as expected for larger item sets as the LTCN reports the best results after the AE model.

Similarly to the previous case study, we replicate the experiment on randomly selected 50, 200, and 500 items. After

[2] https://grouplens.org/datasets/movielens/.

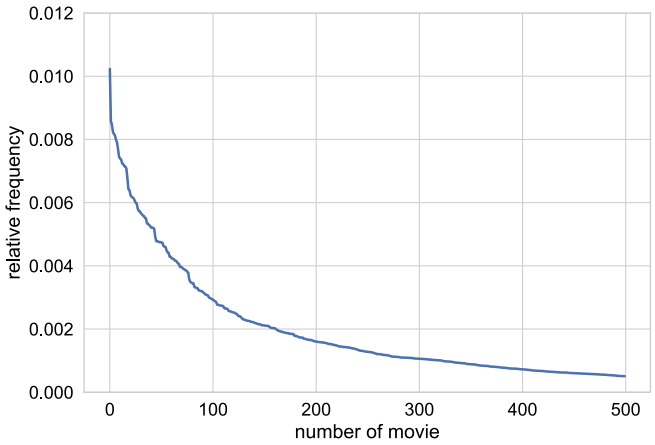


Fig. 4. Relative frequency of the top-500 netflix movies.

Table 7
Training and test MSE on the movielens dataset with the top-50 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0109	0.0634	0.0030	0.0082	0.0044
Test	0.0081	0.0160	0.1002	0.0050	0.0082	**0.0041**

Table 8
Training and test MSE on the movielens dataset with the top-200 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0151	0.0633	0.0056	0.0085	0.0082
Test	0.0155	0.0199	0.0985	**0.0058**	0.0084	0.0081

Table 9
Training and test MSE on the movielens dataset with the top-500 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0211	0.0620	0.0055	0.0086	0.0081
Test	0.0221	0.0271	0.0986	**0.0068**	0.0087	0.0082

Table 10
Training and test MSE on the movielens dataset with 50 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0052	0.0579	0.0037	0.0084	0.0027
Test	0.0122	0.0175	0.1077	0.0037	0.0084	**0.0027**

Table 11
Training and test MSE on the movielens dataset with 200 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0137	0.0574	0.0050	0.0086	0.0051
Test	0.0207	0.0181	0.1028	0.0053	0.0087	**0.0051**

the preprocessing, we obtain datasets containing the rates from 10,101, 48,285, and 62,721 users, respectively. Tables 10, 11, and 12 show that our model outperforms the other algorithm for movie-50 while performing comparably to AE model for movie-200. In the case of movielens dataset with 500 randomly selected items, the AE is the best-performing algorithm followed by the proposed model.

6.4. Case study 3: netflix

The third case study concerns with the Netflix Prize dataset[3] which contains 480,189 users and 17,770 integer ratings between

[3] https://www.kaggle.com/netflix-inc/netflix-prize-data.

Table 12
Training and test MSE on the movielens dataset with 500 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0199	0.0539	0.0050	0.0085	0.0069
Test	0.0262	0.0249	0.1006	**0.0059**	0.0086	0.0070

Table 13
Training and test MSE on the netflix dataset with the top-50 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0148	0.0698	0.0024	0.0092	0.0027
Test	0.0066	0.0185	0.1088	**0.0023**	0.0092	0.0028

Table 14
Training and test MSE on the netflix dataset with the top-200 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0176	0.0677	0.0058	0.0094	0.0049
Test	0.0124	0.0211	0.1083	0.0059	0.0093	**0.0048**

Table 15
Training and test MSE on the netflix dataset with the top-500 items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0236	0.0643	0.0057	0.0095	0.0054
Test	0.0218	0.0274	0.1095	0.0061	0.0095	**0.0053**

Table 16
Training and test MSE on the netflix dataset with 50 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0141	0.0630	0.0009	0.0086	0.0017
Test	0.0098	0.0200	0.1060	**0.0010**	0.0086	0.0017

1 and 5. Given the large size of this dataset, we selected the first 80,000 users to perform the simulations. For that sample, the data also show that there are only 500 movies that are often rated by the user. Fig. 4 shows the relative (decreasing) frequency of those top-500 movies.

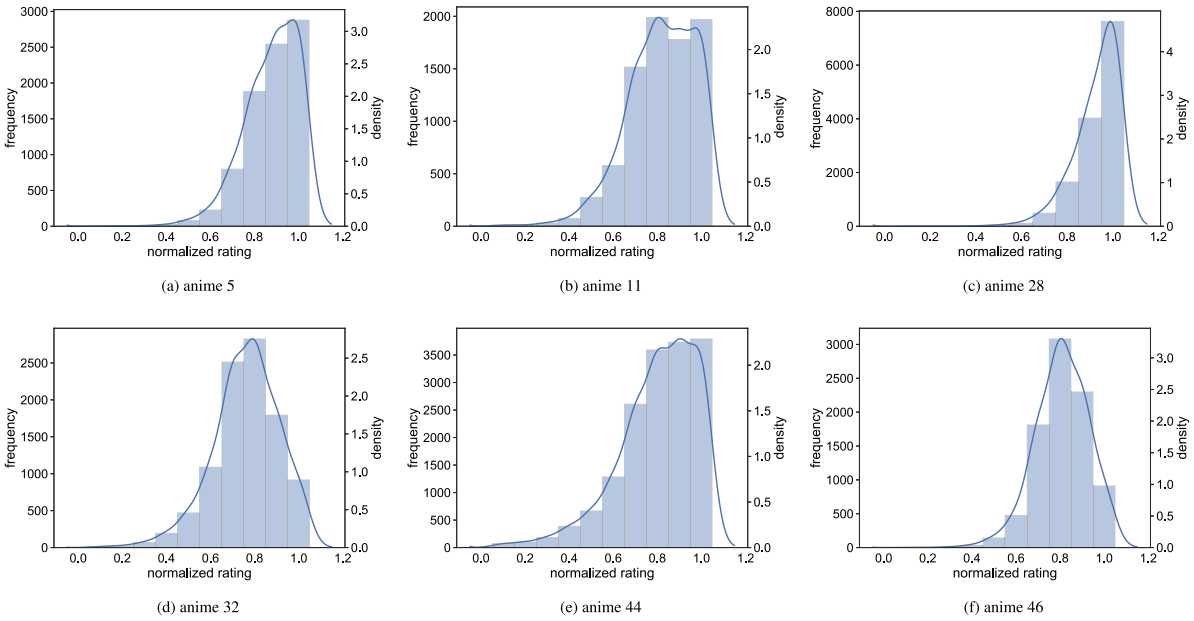
Tables 13, 14 and 15 show both the training and test errors obtained by each model on the netflix-50, netflix-200 and netflix-500 datasets, respectively. The lowest test errors are highlighted in bold. Our model reports the lowest errors for all datasets except for netflix-50, followed by the AE, while the SVD model reported the highest test errors.

In this case study, we also analyze the impact of choosing random items instead of the top-*k* ones. After the preprocessing steps described above, we obtain datasets with the ratings of 50, 200, and 500 random movies from 80,000 users. The Tables 16, Tables 17 and 18 show similar results to those obtained in the top-*k* experiment. Our model outperforms the state-of-the-art for 200 and 500 randomly selected items, while for 50 items it ranks second after the AE model.

Overall, the numerical simulations show the LTCN is capable of produce high-quality recommendations when learning from popular items, even when it involves much less learnable parameters than the other models. Surprisingly, a mix of random and frequently rated items results in better overall performance that selecting the top items, even when the LTCN's weights depend on the correlation matrix.

6.5. The inner workings of the LTCN algorithm

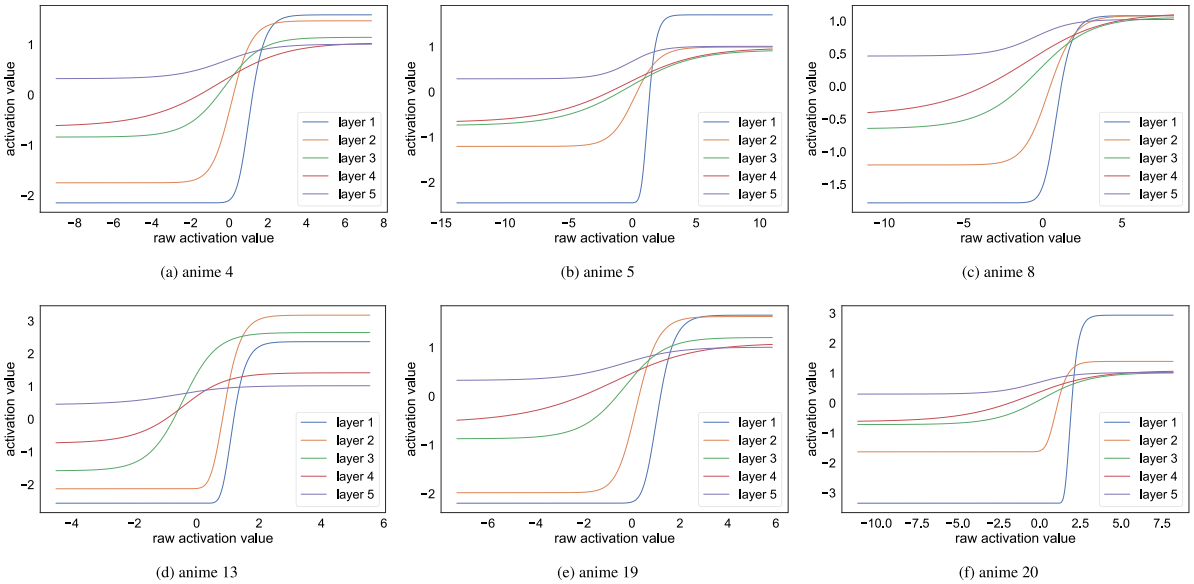
In this subsection, we will briefly illustrate the inner workings of the LTCN algorithm. More specifically, we will show how the kernel neurons and the adjusted sigmoid transfer functions look



(a) anime 5 (b) anime 11 (c) anime 28

(d) anime 32 (e) anime 44 (f) anime 46

Fig. 5. Gaussian kernel neurons for selected anime. The x-axis denotes the normalized rating each anime has received (according to the training data), the left y-axis is the frequency while the right y-axis denotes the probability density. Kernel neurons are used to complete the missing values with estimates that combine the preference of all users for the anime with the rating behavior of the user.



(a) anime 4 (b) anime 5 (c) anime 8

(d) anime 13 (e) anime 19 (f) anime 20

Fig. 6. Adjusted sigmoid transfer functions for selected neurons (using $T = 5$ abstract layers). The reader can notice that the transfer functions in the first abstract layers have a higher degree of non-linearity, which decreases as long as the LTCN model goes deeper. This could be a result of the gradient traveling back through functions with different non-linearity degrees.

like. In order to perform these simulations, we will use the anime-50 dataset and an LTCN with five abstract layers. The remaining parameter settings hold.

Fig. 5 shows some Gaussian kernel neurons computed during the second learning step (see Section 5.2). These neurons contain valuable pieces of information on the ratings received by each anime. Besides, they provide rough estimates for the missing values for a given instance, which are later on corrected by the recurrent reasoning layer.

Fig. 6 displays the shape of selected sigmoid transfer functions (using $T = 5$ abstract layers) that result from the third learning phase. Notice that the functions in the first abstract layers have a higher degree of non-linearity when compared with the functions estimated in the last abstract layers. This behavior is expected somehow if we consider that most parameters optimized by the NSBP algorithm control the non-linearity of an exponential-based function. On the other hand, when the gradient goes backward from the deeper layers to the first ones, it needs to travel through the partial derivatives that change their non-linearity from a layer to another.

Figs. 7–9 show the overall effect of increasing in one unit the arguments of $\partial f_i(x)/\partial v_i$, $\partial f_i(x)/\partial \lambda_i$ and $\partial f_i(x)/\partial h_i$, respectively. Observe that the shape of both $\partial f_i(x)/\partial v_i$ and $\partial f_i(x)/\partial \lambda_i$ changes significantly, while $\partial f_i(x)/\partial h_i$ is shifted. It is worth mentioning that, in these plots the remaining parameters are fixed for the sake of simplicity. However, this is not the reality when training an LTCN-based model since all sigmoid function parameters change simultaneously.

Table 17
Training and test MSE on the netflix dataset with 200 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0176	0.0612	0.0050	0.0094	0.0030
Test	0.0180	0.0212	0.1126	0.0050	0.0095	**0.0031**

Table 18
Training and test MSE on the netflix dataset with 500 randomly selected items.

	*k*NN	MF	SVD	AE	VAE	LTCN
Training	NA	0.0233	0.0594	0.0054	0.0095	0.0046
Test	0.0220	0.0271	0.1121	0.0058	0.0096	**0.0046**

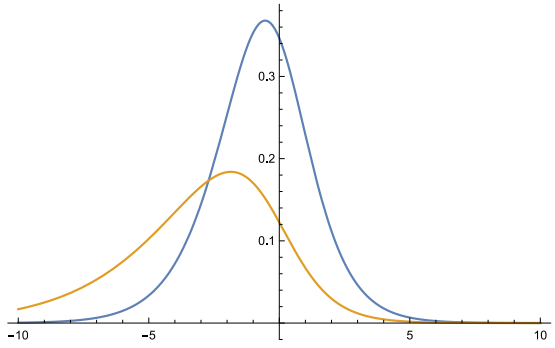


Fig. 7. Result of increasing the argument of $\partial f_i(x)/\partial v_i$ in one unit, such that $h_i = L_i = 0$ and $\lambda_i = U_i = 1$.

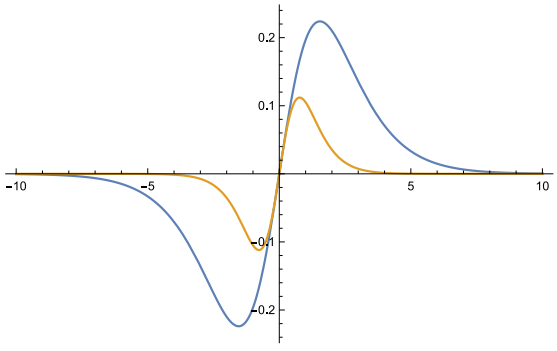


Fig. 8. Result of increasing the argument of $\partial f_i(x)/\partial \lambda_i$ in one unit, such that $h_i = L_i = 0$ and $v_i = U_i = 1$.

A suitable alternative that could be explored in next research studies is to include a regularization component to minimize the variability of the sigmoid transfer functions attached to the same neuron across different abstract layers.

7. Further discussion

Although our model performs well in terms of accuracy when compared with other methods, it seems worth reiterating that its main power still relies on its capability of reasoning with prior knowledge structures. Besides, the literature already includes abundant prediction models producing outstanding results when it comes to accuracy. For example, we could obtain better results with a neural autoencoder if we increase the number of hidden layers and neurons, or if we use other transfer functions. However, the model would also become less interpretable and more difficult to be trained.

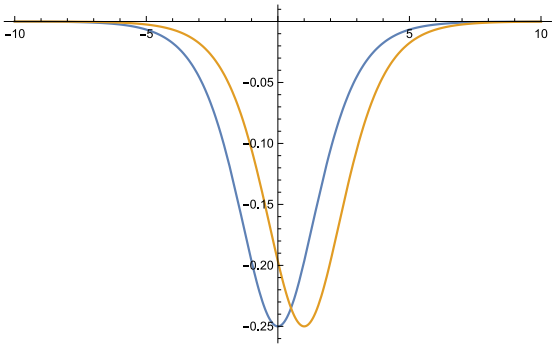


Fig. 9. Result of increasing the argument of $\partial f_i(x)/\partial h_i$ in one unit, such that $L_i = 0$ and $\lambda_i = v_i = U_i = 1$.

In this paper, we have claimed that our model can be deemed interpretable to some extent since its components have a well-defined meaning. This brings some advantages that are difficult to obtain with the solutions reported in the literature. Firstly, we could elucidate how some recommendations are done, which is often appreciated by the user. Although our model certainly allows for that, in presence of networks comprised of hundreds of items, we would need some post-hoc methods to derive concise explanations. Secondly, if the model is transparent enough, then we could inject domain knowledge without changing the theoretical formalism surrounding the model. Reasoning with prior knowledge structures does not only allow producing more consistent results with less effort but also correcting the bias present in the historical data. In that way, the reasoning will not be entirely controlled by what the data dictate.

The reader can fairly question to which extent it would be realistic to ask business analysts or marketing experts to define the *whole* weight matrix. This is however not the intention but the enable the human–machine interaction. Being more explicit, we should request human intervention when the experts suspect that there are new trends that have not yet been observed in the data. This means that the experts can be requested to provide the relationship among a few items, the remaining ones can be effortlessly estimated from the data.

For example, let us suppose that a company included a new item to its stock, thus they have no record on the preference of users on that item. In this case, the correlation coefficients attached to this item could not be derived directly from data. However, an expert could envisage that the new item will likely be preferred in a similar way to X already established items. In the same way, the expert could envisage that the preference of users on the new item will oppose to items provided by competing companies. Then, it would suffice to set up large positive weights to links connecting the new item with the ones associated with it, and large negative weights to items that are opposed to it. The remaining weights can be computed from historical records. Observe that, unlike models that use explicit features, our approach does not require to define new features every time that a new piece of knowledge is available (which would probably lead to a new learning problem).

Finally, it is worth mentioning that such cognitive processes can be automated to some extent. Our perception that experts are often unable to formalize their mental processes would only reflect that we have failed to overcome that problem. This does not imply that such a task is not possible or unrealistic. For example, in [33,34] the authors proposed an automated knowledge engineer to formalize mental representations related to the travel behavior of people in Belgium. Later on, such structures were automatically translated into fuzzy cognitive maps in order to

perform simulations and predictions. Such a result can certainly serve a starting point to accomplish something similar in the context of recommender systems.

8. Concluding remarks

In this paper, we have presented a neural recommender system based on LTCNs operating on the user–item interaction matrix. The proposed model exploits the Pearson's correlation in the user–item matrix while using Gaussian kernel neurons and sigmoid neurons with varying non-linearity degrees. The simulation results using several case studies have shown that our model outperforms most state-of-the-art models in terms of recommendation error. We have also observed that our model performs very well when operating on randomly selected items from a distribution based on their frequency. This makes sense since the inner knowledge of LTCNs relies on the correlation coefficients. However, we believe that the most attractive feature of the LTCN-based model is that it allows experts to inject knowledge into the network. Therefore, the next step of our research will be concerned with exploiting this feature with the aid of an automated knowledge engineer. At the same time, we could use the context information about the items to define the LTCNs' weight matrix. This can be done prescriptively with low computational effort, thus leading to an efficient context-aware approach.

CRediT authorship contribution statement

Gonzalo Nápoles: Conceptualization, Methodology, Writing - original draft, Supervision . **Isel Grau:** Data curation, Software, Validation, Visualization. **Yamisleydi Salgueiro:** Writing - review & editing, Investigation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable and constructive feedback. This paper was partially supported by the Program FONDECYT de Postdoctorado, Chile through the project 3200284.

References

[1] S. Zhang, L. Yao, A. Sun, Y. Tay, Deep learning based recommender system: A survey and new perspectives, ACM Comput. Surv. 52 (1) (2019) 1–38.
[2] P.B. Thorat, R. Goudar, S. Barve, Survey on collaborative filtering, content-based filtering and hybrid recommendation system, Int. J. Comput. Appl. 110 (4) (2015) 31–36.
[3] S. Raza, C. Ding, Progress in context-aware recommender systems — An overview, Comp. Sci. Rev. 31 (2019) 84–97.
[4] N.M. Villegas, C. Sánchez, J. Díaz-Cely, G. Tamura, Characterizing context-aware recommender systems: A systematic literature review, Knowl.-Based Syst. 140 (2018) 173–200.
[5] J. Bobadilla, F. Ortega, A. Hernando, J. Alcalá, Improving collaborative filtering recommender system results and performance using genetic algorithms, Knowl.-Based Syst. 24 (8) (2011) 1310–1316.
[6] Y. Shi, M. Larson, A. Hanjalic, Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges, ACM Comput. Surv. 47 (1) (2014) 1–45.
[7] M. Nilashi, O. Ibrahim, K. Bagherifard, A recommender system based on collaborative filtering using ontology and dimensionality reduction techniques, Expert Syst. Appl. 92 (2018) 507–520.
[8] A.A. Kardan, M. Ebrahimi, A novel approach to hybrid recommendation systems based on association rules mining for content recommendation in asynchronous discussion groups, Inform. Sci. 219 (2013) 93–110.
[9] F. Strub, R. Gaudel, J. Mary, Hybrid recommender system based on autoencoders, in: Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, 2016, pp. 11–16.
[10] F. Zhuang, Z. Zhang, M. Qian, C. Shi, X. Xie, Q. He, Representation learning via dual-autoencoder for recommendation, Neural Netw. 90 (2017) 83–89.
[11] Y. Pan, F. He, H. Yu, A novel enhanced collaborative autoencoder with knowledge distillation for top-n recommender systems, Neurocomputing 332 (2019) 137–148.
[12] T.V. Himabindu, V. Padmanabhan, A.K. Pujari, Conformal matrix factorization based recommender system, Inform. Sci. 467 (2018) 685–707.
[13] J. Bobadilla, F. Ortega, A. Hernando, G. Glez-de Rivera, A similarity metric designed to speed up, using hardware, the recommender systems k-nearest neighbors algorithm, Knowl.-Based Syst. 51 (2013) 27–34.
[14] Y. Park, S. Park, W. Jung, S.-g. Lee, Reversed cf: A fast collaborative filtering algorithm using a k-nearest neighbor graph, Expert Syst. Appl. 42 (8) (2015) 4022–4028.
[15] G. Nápoles, F. Vanhoenshoven, R. Falcon, K. Vanhoof, Nonsynaptic error backpropagation in long-term cognitive networks, IEEE Trans. Neural Netw. Learn. Syst. 31 (3) (2020) 865–875.
[16] J. Bobadilla, F. Ortega, A. Hernando, A. Gutiérrez, Recommender systems survey, Knowl.-Based Syst. 46 (2013) 109–132.
[17] Y. Duan, J.S. Edwards, Y.K. Dwivedi, Artificial intelligence for decision making in the era of big data - evolution, challenges and research agenda, Int. J. Inf. Manage. 48 (January) (2019) 63–71.
[18] A. Dau, N. Salim, Recommendation system based on deep learning methods: a systematic review and new directions, Artif. Intell. Rev. 53 (4) (2020) 2709–2748.
[19] G.R. Lima, C.E. Mello, A. Lyra, G. Zimbrao, Applying landmarks to enhance memory-based collaborative filtering, Inform. Sci. 513 (2020) 412–428.
[20] V. Subramaniyaswamy, R. Logesh, Adaptive knn based recommender system through mining of user preferences, Wirel. Pers. Commun. 97 (2) (2017) 2229—2247.
[21] X. Ning, C. Desrosiers, G. Karypis, A comprehensive survey of neighborhood-based recommendation methods, in: Recommender Systems Handbook, second ed., Springer Science & Business Media, 2015, pp. 37–76.
[22] D. Liu, X. Ye, A matrix factorization based dynamic granularity recommendation with three-way decisions, Knowl.-Based Syst. 191 (2020) 105243.
[23] G. Zhang, Y. Liu, X. Jin, A survey of autoencoder-based recommender systems, Front. Comput. Sci. 14 (2) (2020) 430–450.
[24] D. Kim, B. Suh, Enhancing vaes for collaborative filtering: Flexible priors & gating mechanisms, in: RecSys 2019 - 13th ACM Conference on Recommender Systems, 2019, pp. 403–407.
[25] Y. Gu, X. Yang, M. Peng, G. Lin, Robust weighted svd-type latent factor models for rating prediction, Expert Syst. Appl. 141 (2020) 112885.
[26] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
[27] N. Idrissi, A. Zellou, A systematic literature review of sparsity issues in recommender systems, Soc. Netw. Anal. Min. 10 (1) (2020) 1–23.
[28] Y. Li, S. Wang, Q. Pan, H. Peng, T. Yang, E. Cambria, Learning binary codes with neural collaborative filtering for efficient recommendation systems, Knowl.-Based Syst. 172 (2019) 64–75.
[29] Y. Zhang, D. Liu, G. Yang, L. Hu, Quantization-based hashing with optimal bits for efficient recommendation, Multimedia Tools Appl. (2006) (2020).
[30] Y. Zhang, J. Wu, H. Wang, Neural binary representation learning for large-scale collaborative filtering, IEEE Access 7 (2019) 60752–60763.
[31] G. Nápoles, F. Vanhoenshoven, K. Vanhoof, Short-term cognitive networks, flexible reasoning and nonsynaptic learning, Neural Netw. 115 (2019) 72–81.
[32] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015), 2015.
[33] M. León, G. Nápoles, M.M. García, R. Bello, K. Vanhoof, Two steps individuals travel behavior modeling through fuzzy cognitive maps pre-definition and learning, in: Mexican International Conference on Artificial Intelligence, Springer, 2011, pp. 82–94.
[34] M. León, G. Nápoles, R. Bello, L. Mkrtchyan, B. Depaire, K. Vanhoof, Tackling travel behaviour: an approach based on fuzzy cognitive maps, Int. J. Comput. Intell. Syst. 6 (6) (2013) 1012–1039.