UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Scalable Losses in Session-based Recommendation Systems with Deep Learning Architectures

Tobias Mesquita Silva da Veiga

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Tobias Mesquita Silva da Veiga

Scalable Losses in Session-based Recommendation Systems with Deep Learning Architectures

Dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Luís Gustavo Nonato

USP – São Carlos
February 2023

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

M582f

Mesquita Silva da Veiga, Tobias
Funções de custo escaláveis em sistemas de recomendação baseados em sessões com arquiteturas de Aprendizagem Profunda / Tobias Mesquita Silva da Veiga; orientador Luis Gustavo Nonato. -- São Carlos, 2023.
54 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2023.

1. Recomendações baseadas em sessões. 2. Aprendizagem Profunda. 3. Funções de custo. I. Nonato, Luis Gustavo, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Tobias Mesquita Silva da Veiga

Funções de custo escaláveis em Sistemas de Recomendação baseados em Sessões com Arquiteturas de Aprendizagem Profunda

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Luís Gustavo Nonato

USP – São Carlos
Fevereiro de 2023

RESUMO

DA VEIGA, T. M. S. **Funções de custo escaláveis em Sistemas de Recomendação baseados em Sessões com Arquiteturas de Aprendizagem Profunda**. 2023. 54 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Em sistemas de recomendação, o objetivo é melhorar a experiência do usuário, sugerindo conteúdos interessantes e fornecendo rapidamente o que procuram. Para ajudar nesta tarefa, Deep Learning provou ser uma ferramenta eficiente, especialmente em sistemas de recomendação onde as informações de identificação do usuário não são possíveis de serem atribuídas às sessões do usuário. A natureza de algumas ferramentas de Aprendizagem Profunda como Redes Recorrentes, Redes Neurais em Grafos e Mecanismo de Atenção os torna capazes de lidar com dados de tamanho variável em grande escala, o que é ideal para processar sessões de usuário. Nesse contexto, muitos trabalhos estado da arte de Aprendizagem Profunda têm a limitação de não serem escaláveis para grandes conjuntos de dados, onde o número de itens únicos a serem recomendados é muito grande. Neste trabalho, exploramos como funções de custo escaláveis podem modificar os resultados de trabalhos anteriores e apresentamos um novo conjunto de dados mais desafiador para testar se essas modificações realmente tornam os modelos escaláveis.

Palavras-chave: Recomendações baseadas em sessões, Aprendizagem Profunda, escalabilidade, Mecanismo de Atenção, Redes Neurais de Grafos.

ABSTRACT

DA VEIGA, T. M. S. **Scalable Losses in Session-based Recommendation Systems with Deep Learning Architectures**. 2023. 54 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

In recommendation systems the objective is to improve user experience by suggesting interesting contents and quickly providing what they look for. To help with this task, Deep learning has proven to be an efficient tool, especially in recommendation systems where user navigation sessions are anonymous. The nature of Deep Learning tools such as Recurrent Networks, Graph Neural Networks and Attention Mechanism makes them capable of dealing with variable length data in large scale, which is ideal for processing user sessions. In this context, many state of the art Deep Learning models have the limitation of not being scalable to large datasets, where the number of unique items to recommend is very large. In this work we explore how scalable loss functions can modify the results of previous works and we introduce a new challenging dataset to assess whether such modifications really make the models scalable.

Keywords: Session-based recommendation, Deep Learning, scalability, Attention Mechanism, Graph Neural Networks.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AM	Attention Mechanism
ConvGNN	Convolutional Graph Neural Networks
DL	Deep learning
GAT	Graph Attention Networks
GGSNN	Gated Graph Sequence Neural Network
GNN	Graph Neural Networks
GRU	Gated Recurrent Units
LSTM	Long Short-Term Memory
MLP	Multilayer Perceptron
NN	Neural Networks
RecGNNs	Recurrent Graph Neural Networks
RNN	Recurrent Neural Networks
SRS	Session-based Recommendation

LIST OF SYMBOLS

σ — sigmoid function

∇_θ — gradient of a function in respect to θ

ϕ — hyperbolic tangent function

$\odot$ — element-wise multiplication

CONTENTS

CHAPTER

1

INTRODUCTION

With the overwhelming amount of available information and efficient web searching platforms, the task of searching for a specific content is much easier than before. Yet it can still be hard to find what we want or to have the feeling that we have searched enough. However users might not be searching for something specific, but just looking for something new and interesting. The important point in this scenario is whether user can find relevant content without external help and with reduced browsing time.

Defining a content as relevant is subjective, but for research purposes some definitions are possible. For instance, a media content can be considered relevant depending on how long an user watches the content. A product can be relevant depending on how many consumers buy it.

Once we defined a measurement for relevance, the performance of a recommendation system becomes measurable and comparable. One important aspect that drastically impacts a recommendation performance is whether user identification is available to the recommendation system. When no user identification is available, we have the so-called Session-based Recommendation (SRS) scenario, where the recommendation system only has access to the activity log of the current active session of a user. A session in this context is a sequence of actions a user makes while using the search platform. In such scenario, it is not possible to associate the user in the current session with other user data, such as user preferences and even other recorded sessions made by the same user in the past.

For a long time, e-commerce companies stored users information to compare them and suggest items that similar users had interacted with. However, nowadays cookies restrictions and anonymous browsing prevent companies from storing user information. Furthermore, users intention might change drastically from one session to another. Therefore looking at a session individually rather than focusing on the user itself is not only essential but also a more general approach.

Classical techniques such as (LINDEN; SMITH; YORK, 2003; DAVIDSON *et al.*, 2010)

have limitations to handle sessions information due to the computational cost of processing large amounts of data and the inability to deal with sequential data information. Deep learning (DL), on the other hand, has been successfully applied to SRS problems, overcoming the limitations of classical methods.

However, while much effort has been done to improve the quality of recommendations, not so much has been done towards scalability. Most recent state of the art techniques (HIDASI *et al.*, 2016; LI *et al.*, 2017; LIU *et al.*, 2018; WU *et al.*, 2019) have limitations making them not suitable for datasets with large number of items (100k+).

In this work, we propose adaptations in Neural Networks (NN) architecture devoted to tackle SRS in order to make them able to handle larger datasets. The idea is to develop a methodology that can be used with small size datasets, as well as with large amounts of data, while preserving similar performance in both cases.

In the following chapters of this manuscript we present: 2) basic NN concepts that the state of the art techniques rely on for recommendation tasks; 3) the DL related works in the context of SRS; 4) the methodology of this work, including loss functions choices, gradient descent algorithm, datasets description and evaluation metrics); 5) description of experiments and results; 6) conclusions of this work.

CHAPTER

2

BASIC CONCEPTS

2.1 User sessions

A user session contains a lot of information and many different types of interactions. There is a wide range user interaction types such as "moves the mouse", "visits a product page", "uses the search tool", etc. In this work we are only interested in the interaction in which an user visits an item, e.g. visits a product page.

In e-commerces, a single item can a lot information associated with it. This information can be in the form of media, timestamp, tags, etc. These are all valid information to be used by a DL SRS. However, we will restrict our model to the simplest possible item data input input, which is keeping only and integer identifier for each item that ranges from 1 to N, N being the number of unique items in all sessions.

Therefore, for the purpose of this work, a user session is just sequence of integers ordered from the first visited item to the last visited item.

2.2 Neural Network approaches

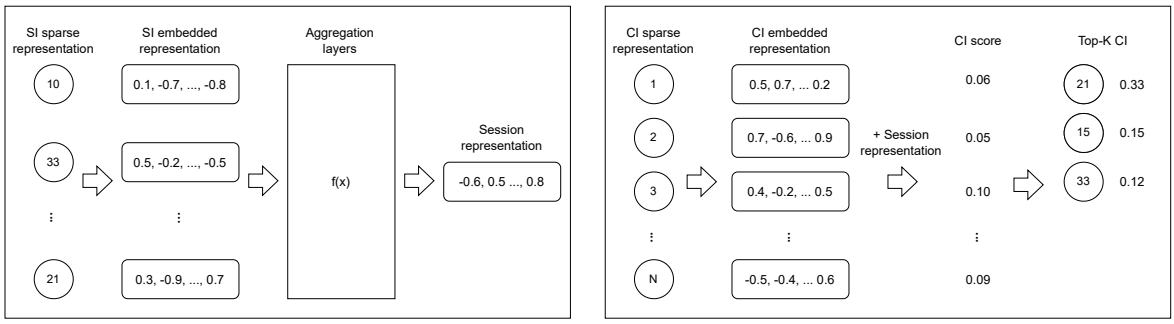
SRS require the existence of a function that receives a session, i.e. a sequence of item categories, and outputs a list of recommendations. An efficient way of modeling such functions is with NNs.

NNs represent a large family of models that use a combination of linear and activation functions stacked in layers to approximate complex functions. A simple example is the Multilayer Perceptron (MLP) with one hidden layer, which can mathematically be defined as:

$$f(x) = \sigma(W^{(2)}(\sigma(W^{(1)}x)) \tag{2.1}$$

where $W^{(1)}$ and $W^{(2)}$ are the weights (parameters) in the hidden and output layers operating as linear functions, x is the input of the MLP and σ is an activation function.

The output $f(x)$, in the SRS context, is not directly a short list of items, instead it is a numerical vector representing a whole input session x. This session representation is compared with all items in the SRS, which are also represented as numerical vectors (see 2.2.1). By comparing the session representation with each item, we obtain a score for each of them. Finally, a post processing step is responsible for ranking and selecting the top-K items for recommendation. This process is illustrated in the Figure 1.



(a) (b)

Figure 1 – (a) Inference from session items (SI) to session representation. (b) Ranking process from all database candidate items (CI) to the top-K items selection.

The parameters of the linear functions are learned through variants of the gradient descent method, which can be stated as:

$$\theta = \theta - \alpha \nabla_\theta L(f(x), y) \tag{2.2}$$

where α defines a learning rate, i.e. the speed in which the parameters are updated, and ∇ is the gradient of the loss function L, which is a differentiable function that measures how close the model output $f(x)$ is to the real function value y.

The right choice of learning rate is important to avoid early convergence, for which it can be iteratively updated using sophisticated update rules.

In the context of SRS there are three important NN architectures, namely, Recurrent Neural Networks (RNN), Graph Neural Networks (GNN) and Attention Mechanism (AM). They all have the property of receiving variable-length data and reducing it to a lower dimension output, which is essential to SRS in order to obtain a representation of fixed size for a whole session.

The following subsections discuss each of these approaches with the intent of showing the importance of each while highlighting their limitations that open new avenues for future research. But as a prerequisite we need to cover how the input of the NNs is represented, hence we introduce embedding layers first.

2.2.1 Embedding layer

The input of a Neural Network is required to be numerical. Ideally, the input should be a vector with numbers in a limited range, e.g. a vector $\boldsymbol{v} = [v_1, ..., v_n], \forall i = 1, .., n, v_i \in [-1.0, 1.0]$.

In SRS, however, the most common input is a variable length sequence of items categories. There are several ways to manage variable length sequences with Neural Networks as we explain in the following subsections. However, the requirement that the elements of the sequence be numeric remains.

In order to represent item categories as numerical vectors, a common approach is to add and embedding layer to the Neural Network. An embedding layer is a map from a simplistic category representation, e.g. category as an integer, to a numerical vector.

After the whole sequence of items is mapped to the numerical representation, the NN aggregates the items representations and provides output scores for the available items in order to rank them. In this sense, the numerical representations must be optimal for the NN to perform at its best.

The process of finding an optimal embedding representation is the same as optimizing any other parameter in a NN. The embedding vector values are weights optimized by the gradient descent method.

2.2.2 Recurrent Neural Networks

RNNs iteratively process each input as a sequence of ordered elements and preserves information of previous elements in its hidden states.

The output of an RNN can be either the sequence of all generated hidden states at each iteration, representing a state for each input element, or just the last generated hidden state representing an aggregation of the whole sequence.

The hidden states are a numerical vectors updated according to the elements in the sequence, accounting for information from previously processed elements.

One standard approach for updating the hidden state is as follow:

$$\boldsymbol{h}_t = g(W\boldsymbol{x}_t + U\boldsymbol{h}_{t-1}) \tag{2.3}$$

where $\boldsymbol{h_t}$ is the hidden state at position t, $\boldsymbol{x_t}$ is the input element at position t, W and U are learnable wights and g is an activation function.

Although the above formula is simple and intuitive, it suffers from optimization problems of vanishing gradients and exploding gradients.

There are two well know approaches for solving these problems, Gated Recurrent Units (GRU) (CHO *et al.*, 2014) and Long Short-Term Memory (LSTM) (GERS; SCHMIDHUBER;

CUMMINS, 2000). In this work, we will focus on the GRU since it is the RNN used by works mentioned in this manuscript. GRU is mathematically defined as:

$$z_t = \sigma_g(W_x x_t + U_z h_{t-1} + b_z) \tag{2.4}$$

$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \tag{2.5}$$

$$\hat{h}_t = \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{2.6}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{2.7}$$

where x_t is the input, h_t is the output and also the hidden state, $\hat{h}_t$ is the candidate activation, z_t is the update gate , r_t is the reset gate, W, U and b are learnable weights, and ϕ_h is the hyperbolic tangent function, σ is the sigmoid function, $\odot$ is the element-wise multiplication. z_t controls the balance between the previous activation and the new candidate. r_t controls how much of the previous activation is used on the new candidate.

2.2.3 Attention Mechanism

The attention is a mechanism to give more weight to some inputs for a given set of context inputs. This allows the model to filter what is more relevant in long sequences of inputs and generate better representations that can further enhance the performance of the model.

More formally, attention maps a query vector and a set of key-value vectors pairs to an output. The query vector represent something that can be considered the attention context and each key and value vectors are representations for each of the inputs.

Weights for each input are computed by comparing query and keys. Higher matches between query and keys means an input will have a higher weight. These weights are then used to compute the output as the weighted average of the input values.

In other words, queries and keys are learnable parameters to compute what inputs are more important, while the value vector is also a learnable parameter representing what will be passed forward as being important.

In the "Scaled Dot-Product Attention" (VASWANI *et al.*, 2017), the dot product is used as matching function between queries and keys.

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.8}$$

where Q, K and V are matrices with sets of queries, keys and values vectors, d_k is the dimensions size of the key vectors. By packing the three attention components into matrices, one can effectively handle many queries at once. However, in the SRS studies presented here (see Chapter 3) a simpler formula is used:

$$\boldsymbol{s_q} = \sum_{i=1}^{n} \alpha_i * \boldsymbol{v}_i \tag{2.9}$$

,

where $\boldsymbol{s_q}$ denotes a session represented by the aggregation of the n unique items in the session given a query vector $\boldsymbol{q}$, $\boldsymbol{v}_i$ is the value vector of the ith item and α_i is the attention score for the ith item defined as:

$$\alpha_i = \boldsymbol{p}^\top \sigma(\boldsymbol{W}_0 \boldsymbol{v}_i + \boldsymbol{W}_1 \boldsymbol{q} + \boldsymbol{c}) \tag{2.10}$$

,

where $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$ are respectively projections of the item key representation $\boldsymbol{v}_i$ and the the query vector $\boldsymbol{q}$ into a latent space, $\boldsymbol{c}$ is a learnable bias vector, σ is a sigmoid function, $\boldsymbol{p}$ is a learnable vector that converts the resulting vector of the sigmoid function into an scalar.

This formulation is simplified because $\boldsymbol{v}_i$ is the embedding vector of the ith item and is representing both the key vector in 2.10 and the value vector seen in 2.9. However, this matching function is less intuitive than the dot product.

Attention is an important alternative to RNNs for several reasons. Computation is bounded to the size of the key and values dimensions, not to the sequence length. Computation is also easier to be done in parallel. And the most important, this method is more robust when there is a important dependency between distant inputs, because the distance does not affect the computations.

2.2.4 Graph Neural Networks

Sessions can be represented as graphs, in which nodes represent items and edges are the transitions from one item to another.

More formaly, a graph is a tuple (V,E), V is the set of vertices $\{v_1,...v_n\}$, n being the total number of vertices. E is the set of edges represented by tuples (v_i,v_j) indicating there is an edge from vertice v_i to vertice v_j. We also use a matrix A with size $n \times n$ as notation to refer to the adjacency matrix in which an element a_{ij} of A is 1 if an edge from v_i to v_j is present and 0 otherwise. To refer to the neighbours of a node v, we use the notation $N(v)$. These are the basic notations of a graph used to define its topology, but in SRS, items (or nodes) also contain important information that describes the item. So we also have X as the set of features of each item where each entry x_i indicates the features associated to the vertice v_i. In some special cases we can also have X^e as the set of features associated with each edge of the graph, in which x^e_{ij} represents the features of the edge from v_i to v_j.

We will consider two main types of GNNs: Recurrent Graph Neural Networks (RecGNNs) and Convolutional Graph Neural Networks (ConvGNN).

2.2.4.1 RecGNNs

RecGNNs have a hidden state for each node of the network, the hidden state is updated with the information from its neighbour nodes $N(v)$. This process is repeated for a finite number of iterations.

Gated Graph Sequence Neural Network (GGSNN) (LI *et al.*, 2016) presents an elegant formulation of this inspired by GRU:

$$h_v^t = GRU(h_v^{(t-1)}, \sum_{u \in N(v)} W h_u^{(t-1)}) \tag{2.11}$$

where h_v^t is the hidden state of a node v at iteration t, with h_v^0 as x_v, and GRU is the GRU function that updates the hidden state (first parameter) with the current information (second parameter).

2.2.4.2 ConvGNNs

This type of GNN can be further divided into spectral and spatial based convolutions. Spectral-based have a solid mathematical foundation based on spectral graphs theory but these methods have several limitations such as not generalizing to new graph structures, high computational cost and don't work on directed graphs. We will leave this subtype out of the scope of the project and will focus only on spatial-based ConvGNNs that, as we will explain later, are more closely related with RecGNNs.

Spatial-based ConvGNNs overcome the mentioned limitations of the spectral-based counterparts. Instead of using costly graph signal operations, they operate directly on each node by aggregating its neighbours information. In this sense, they are similar to the RecGNNs. The difference is in how this process is accomplished. In each iteration the convoluted nodes generate a new layer of nodes in which the operation of convolution can be repeated again generating another layer.

GraphSAGE (HAMILTON; YING; LESKOVEC, 2017) was the first inductive ConvGNN algorithm, in the sense that it was not limited to only learning representations to the nodes that had been seen in training. The activation function can be described by the following equation:

$$h_v^k = \sigma(W^k \cdot \text{AGGREGATE}(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in N(v)\})) \tag{2.12}$$

where h_v^k is the representation of node v at layer k, $\cup$ is the concatenation operator and AGGREGATE indicates the aggregation function used, which can be either mean, max or LSTM.

GraphSAGE gives equal contribution to all nodes when aggregated. Graph Attention Networks (GAT) (VELIČKOVIĆ *et al.*, 2018) improves this and computes the importance of each node. In mathematical terms:

$$h_v^{(k)} = \sigma(\sum_{u \in N(v) \cup v} \alpha_{vu}^{(k)} W^{(k)} h_u^{(k-1)}) \tag{2.13}$$

where $h_v^{(0)} = x_v$, σ is a sigmoid function and $\alpha_{vu}^{(k)}$ is the attention weight between nodes u and v:

$$\alpha_{vu}^{(k)} = softmax(g(a^T [W^{(k)} h_v^{(k-1)} || W^{(k)} h_u^{(k-1)}])) \tag{2.14}$$

where g is LeakyReLU activation function and a is a vector of learnable parameters

2.3 The choice of loss function with NNs

Another important topic is the choice of loss function used in NN models. As we will show in section 3 many works use standard cross-entropy for computing the loss. This is not ideal in many SRS since the number of items can be considerably large, sometimes reaching millions of items, thus making the algorithm not scalable. Cross entropy has the following formula.

$$L(\hat{y}, y) = -\sum_{k}^{K} y^{(k)} \log \hat{y}^{(k)} \tag{2.15}$$

where K is total number of classes (i.e. items to recommended), $\hat{y}$ is the predicted class probability and y is 1 if k is the correct class and 0 otherwise. Since all but one element of this sum is zero, this computation is not proportional to K.

The scalability problem raises from the fact that NNs output scores, not probabilities. To convert scores into probabilities, the most standard mechanism is to use a Softmax function, defined as:

$$\hat{y}^{(i)} = \frac{\exp(s_i)}{\sum_k^K \exp(s_k)} \tag{2.16}$$

where s_k is the NN's output score for the kth class. The formula requires that the score for all possible K classes is calculated for each training sample. Therefore, the training time is directly proportional to the number of classes. In SRS where the number of classes is the number of possible items, which is usually very large, this easily becomes infeasible.

Finding solutions such as negative sampling or hierarchical sampling are more ideal for these problems, the number of scores calculated per sample is drastically reduced and in some studies better results are obtained (HIDASI *et al.*, 2016).

One alternative to the vanilla softmax is to use Hierarchical Softmax (MORIN; BENGIO, 2005), which organizes classes into a binary tree for fast computation of the error. This formula

computes in approximately log of K the error for each sample but in practice it is even more efficient because it uses Huffman encoding to make a efficient tree structure in which the more frequent classes are accessed faster.

A simpler approach that have provided better results is negative sampling. This loss was introduced in the word2vector model (MIKOLOV *et al.*, 2013) where the goal was to find embeddings for words given the structure they are organized in documents. In mathematical terms:

$$\log \sigma(v'^{\top}_{w_O} v_{w_I}) + \sum_{i=0}^{k} \mathbb{E}_{w_i \sim P_n(w)} [\log \sigma(-v'^{\top}_{w_i} v_{w_I})] \tag{2.17}$$

where w_I is an input word that is expected to have similar embedding to a w_O neighbour word, w_i is a word sample from a noise distribution $P_n(w)$, v_w is the input embedding of a word w and v'_w is the output embedding of a word w, σ is a sigmoid function. Intuitively this objective gives higher scores when the embedding of the input word is similar to the neighbour output word embedding while it is distinct from embeddings of words sampled from a noise distribution.

Another classical use of negative sampling was used in node2vec (GROVER; LESKOVEC, 2016) where the goal is to find embeddings for nodes in graphs given the graph structure:

$$\max_f \sum_{u \in V} [-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u)] \tag{2.18}$$

where $N_S(u)$ is a set of neighbour nodes from node u obtained from a random walk, f is the embedding function and $Z_u = \sum_{v \in V} \exp(f(u) \cdot f(v))$ with V being the set of all nodes. In practice Z_u is approximated using the negative sampling just as in word2vector.

The other scalable loss function is the Triplet Loss, initially introduced by (SCHROFF; KALENICHENKO; PHILBIN, 2015) and (WANG *et al.*, 2014). The idea of the loss is to make samples of the same class close while samples of different classes are kept distance. The distinction between close and distant is defined by the margin hyperparameter. First a sample is randomly selected and defines a reference (anchor sample), then another sample of the same class is selected (positive sample) and finally a sample of a different class is selected (negative sample). The loss formula is as following:

$$L(x_a, x_p, x_n) = \max(0, m + \|f(x_a - f(x_p))\| - \|f(x_a) - f(x_n)\|) \tag{2.19}$$

where x_a is the anchor sample embedding, x_p is the positive sample embedding, x_n is the negative sample embedding, $m \in$ is the margin.

This loss is more scalable than the vanilla cross-entropy since it is not necessary use all classes to compute the score of a sample. On the other hand it is more complicated to compute

because it needs to define the negative samples of all batches and samples at the beginning of each epoch.

One should notice that both negative sampling and triplet loss needs to define an embedding for the items they compare. This is a constraint to recommender systems that limits the system potential to be applied on unseen items that have no learned embedding.

CHAPTER

3

DEEP LEARNING BASED RECOMMENDATION MODELS

This section shows some of the relevant works that make use of the Neural Networks in SRS. The works are presented in their chronological order.

3.1 Gru4Rec

Gru4Rec (HIDASI *et al.*, 2016) introduced RNNs in the context of SRS. The architecture (see Figure 2) begins with input as one-hot representation, i.e. items are represented as zero vectors with the same size of the total number of unique items and with a single one in a unique vector position according to the item, followed by: embedding layer, stacked GRU layers, stacked feedforward layers, output as scores of each item. Also, they added skip connection from embedding layer to other GRU layers. And for training they used negative sampling.

Their batching scheme consisted in processing many sessions in parallel. Since the task was to predict the next item to be seen in a session, for each session the model is trained

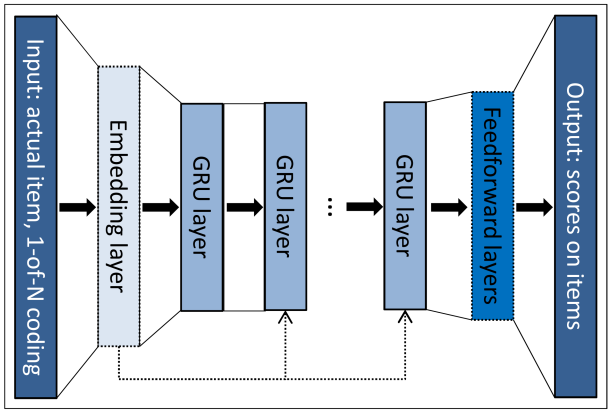


Figure 2 – GRU4REC model architecture (extracted from the original work (HIDASI *et al.*, 2016)).

iteratively, processing one item as input and predicting the next one, and then processing the next one and repeating the process. If a session has no more items than the item of a new session takes place and the hidden states of the respective model is reseted.

3.2 NARM

NARM (LI *et al.*, 2017) introduced Attention layers in the context of SRS. The attention was only a way to aggregate information from a previous RNN GRU layer (see Figure 3). The GRU layer serves as an encoding layer for each input item, such that the computed hidden state of each iteration represents an item.

Given an input sequence of items $\boldsymbol{x} = [x_1, ..., x_t], \forall i, x_i \in N$, the NARM model has two data paths. The first path is the global encoder that takes into account the last computed hidden state h_t of a GRU layer on the input sequence. This is referred as the sequential behavior feature $c_t^g = h_t$. The second path is the local encoder that takes the computed hidden state of each iteration of a GRU layer over the input sequence $\boldsymbol{x}$ and apply an attention layer over these hidden states. The output of the attention layer is defined as:

$$\boldsymbol{c}_t^l = \sum_{j=1}^{t} \alpha_{tj} \boldsymbol{h} j \tag{3.1}$$

where α is the attention weighting factor, that is,

$$\alpha_{tj} = \boldsymbol{v}^\top \sigma(\boldsymbol{A}_1 \boldsymbol{h}_t + \boldsymbol{A}_2 \boldsymbol{h}_j) \tag{3.2}$$

where $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are matrices of learnable weights, σ is a sigmoid function. In this approach the attention score is computed having the last computed hidden representation as context information.

The two data paths are joined by concatenating their outputs to generate $\boldsymbol{c}_t = [\boldsymbol{c}_t^g; \boldsymbol{c}_t^l]$. Then a similarity layer compares $\boldsymbol{c}_t$ to each candidate item embedding:

$$S_i = emb_i^\top \boldsymbol{B} \boldsymbol{c}_t \tag{3.3}$$

where $\boldsymbol{B}$ projects $\boldsymbol{c}_t$ into emb_i dimension space and S_i is the similarity score for an item x_i.

They use cross-entropy as the loss function with standard mini batches but processing each sequence separately instead of doing as Gru4Rec, since the attention layer needs to process everything together and they only aim to predict labels generated before training.

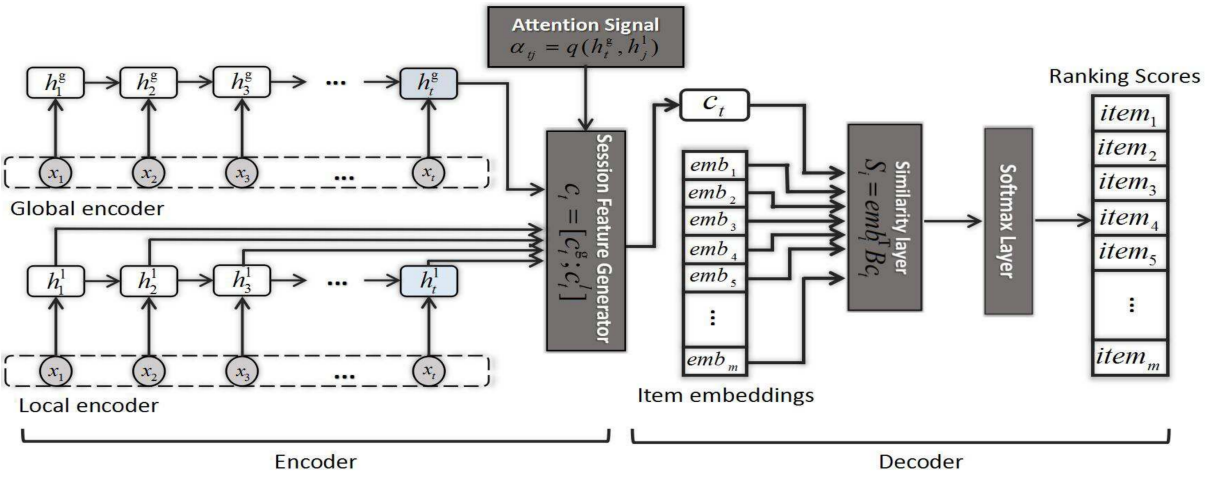


Figure 3 – NARM model architecture (extracted from the original work (LI *et al.*, 2017)).

3.3 STAMP

STAMP (LIU *et al.*, 2018) used only attention without RNN or GNN layers (see Figure 4).

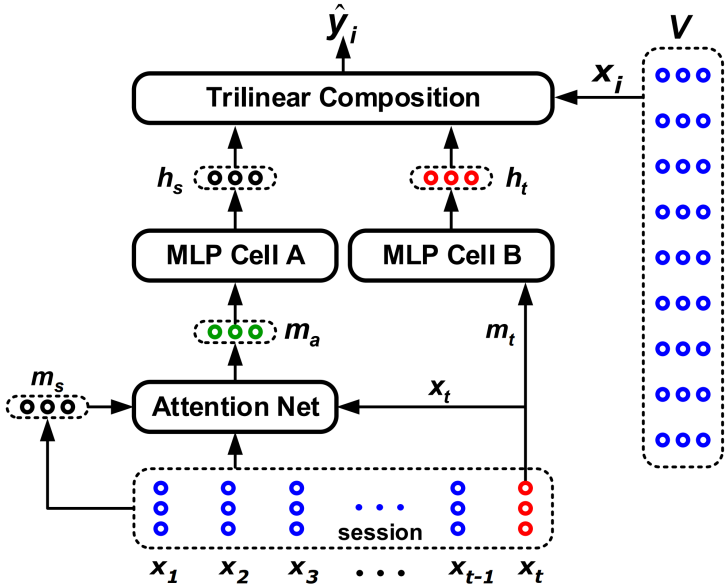


Figure 4 – STAMP model architecture (extracted from the original work (LIU *et al.*, 2018)).

The attention weight for the ith item for a session with length t is computed as:

$$\alpha_i = \boldsymbol{W}_0 \sigma(\boldsymbol{W}_1 \boldsymbol{x}_i + \boldsymbol{W}_2 \boldsymbol{x}_t + \boldsymbol{W}_3 \boldsymbol{m}_s + \boldsymbol{b}_a) \tag{3.4}$$

where $x_i \in R^d$ denotes the ith item, $x_t \in R$ denotes the last item, $\boldsymbol{W}_0 \in R^{1\times d}$ is a weighting vector, $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, $\boldsymbol{W}_3$, $\in R^{d\times d}$ are weightning matrices, $\boldsymbol{b}_a$ is a bias vector, $\boldsymbol{m}_s$ is the average embedding of the session, σ is a sigmoid function.

α_i is used as the weight for averaging out the item embeddings:

$$\boldsymbol{m}_a = \sum_{i=1}^{t} \alpha_i * \boldsymbol{x}_i \tag{3.5}$$

Then $\boldsymbol{m}_a$ and $\boldsymbol{m}_t = \boldsymbol{x}_t$ go through a feedforward layer each resulting in $\boldsymbol{h}_s$ and $\boldsymbol{h}_t$ respectively.

The score $\hat{z}_i$ for a given candidate item $\boldsymbol{x}_i$ from the set of all items is defined as:

$$\hat{z}_i = \sigma(< \boldsymbol{h}_s, \boldsymbol{h}_t, \boldsymbol{x}_i >) \tag{3.6}$$

where σ is a sigmoid function and $< .,.,. >$ is the trilinear product of three vectors defined as:

$$< a,b,c >= \sum_{i=1}^{d} a_i b_i c_i \tag{3.7}$$

Finally, they used cross-entropy for computing the loss.

3.4 SR-GNN

The SR-GNN proposed by (WU *et al.*, 2019) uses a recurrent GNN from GGNN (LI *et al.*, 2016) (see Figure 5). The GGNN layer is recursive and at each iteration receives two elements: 1) the hidden items representations $[\boldsymbol{v}_1^{t-1}, ..., \boldsymbol{v}_n^{t-1}]$, where $\boldsymbol{v}_i^0$ is the item embedding of an item i present in a session s with n distinct items , 2) a special adjacency matrix $\boldsymbol{A}_s \in R^{n \times 2n}$, where $\boldsymbol{A}_s = [\boldsymbol{A}_s^{(in)}; \boldsymbol{A}_s^{(out)}]$ is the concatenation of the adjacency matrix of incoming and outgoing edges normalized by the in-degree and out-degree respectively, outputting a new hidden state representation for each item $[\boldsymbol{v}_1^t, ..., \boldsymbol{v}_n^t]$. It is decomposed by the following equations:

$$\boldsymbol{a}_{s,i}^t = \boldsymbol{A}_{s,i:}[\boldsymbol{v}_1^{t-1}, ..., \boldsymbol{v}_n^{t-1}]^\top \boldsymbol{H} + \boldsymbol{b} \tag{3.8}$$

$$\boldsymbol{z}_{s,i}^t = \sigma(\boldsymbol{W}_z \boldsymbol{a}_{s,i}^t + \boldsymbol{U}_z \boldsymbol{v}_i^{t-1}) \tag{3.9}$$

$$\boldsymbol{r}_{s,i}^t = \sigma(\boldsymbol{W}_r \boldsymbol{a}_{s,i}^t + \boldsymbol{U}_r \boldsymbol{v}_i^{t-1}) \tag{3.10}$$

$$\tilde{\boldsymbol{v}}_i^t = \tanh(\boldsymbol{W}_o \boldsymbol{a}_{s,i}^t + \boldsymbol{U}_o(\boldsymbol{r}_{s,i}^t \odot \boldsymbol{v}_i^{t-1})) \tag{3.11}$$

$$\boldsymbol{v}_i^t = (1 - \boldsymbol{z}_{s,i}^t) \odot \boldsymbol{v}_i^{t-1} + \boldsymbol{z}_{s,i} \odot \tilde{\boldsymbol{v}}_i^t \tag{3.12}$$

After a pre-defined K number of iterations, the new items representations are combined into the global embedding, a vector $\boldsymbol{s}_g$, using a soft attention mechanism that takes the last seem item $\boldsymbol{v}_n$ as context information:

$$\boldsymbol{s}_g = \sum_{i=1}^{n} \alpha_i * \boldsymbol{v}_i \tag{3.13}$$

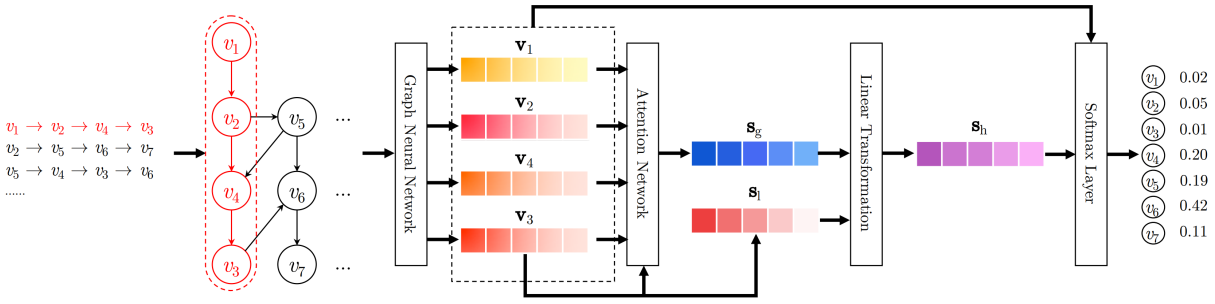


Figure 5 – SRGNN model architecture (extracted from the original work (WU *et al.*, 2019)).

$$\alpha_i = \boldsymbol{q}^\top \sigma(\boldsymbol{W}_1 \boldsymbol{v}_n + \boldsymbol{W}_2 \boldsymbol{v}_i + \boldsymbol{c}) \tag{3.14}$$

where α_i is the attention score, σ is a sigmoid function.

Then the global embedding $\boldsymbol{s}_l$ is concatenated with the last clicked item embedding and projected into the item embedding space again.

$$\boldsymbol{s}_h = \boldsymbol{W}_3 [\boldsymbol{s}_l; \boldsymbol{s}_g] \tag{3.15}$$

where $\boldsymbol{s}_l = \boldsymbol{v}_n$ and the projection $\boldsymbol{s}_h$ is the hybrid embedding.

Finally they compute the score $\hat{z}_i$ for each candidate item $\boldsymbol{v}_i \in V$ by multiplying its embedding $\boldsymbol{v}_i$ by the session representation $\boldsymbol{s}_h$:

$$\hat{\boldsymbol{z}}_i = \boldsymbol{s}_h^\top \boldsymbol{v}_i \tag{3.16}$$

and the score is used in cross-entropy loss to optimize the model.

3.5 Models' comparison

In this section we compare the models described above 1. Two small datasets are used for comparison *Yoochoose* [1] and *Diginetica* [2]. Yoochoose is further divided into two version where only 1/64 and 1/4 of the most recent samples are used. Two different metrics are used, P@20 and MRR@20. P@20 is the proportion of samples in which the correct item was within the top 20 predicted items with highest score. MRR@20 is the average over the reciprocal ranks of all samples, it is a metric that penalizes more if the correct item has a lower rank within the top 20 highest score classes. The reciprocal rank, in MRR@20, of a sample in which the correct class has rank $rank_i$ is $\frac{1}{rank_i}$ if $rank_i < 20$ and 0 otherwise. Table 1 summarizes the performance of the models regarding these metrics.

[1] <https://2015.recsyschallenge.com/challege.html>
[2] <http://cikm2016.cs.iupui.edu/cikm-cup>

Model	Yoochoose 1/64		Yoochoose 1/4		Diginetica	
	P@20	MRR@20	P@20	MRR@20	P@20	MRR@20
GRU4REC	60.64	22.89	59.53	22.60	29.45	8.33
NARM	68.32	28.63	69.73	29.23	49.70	16.17
STAMP	68.74	29.67	70.44	30.00	45.64	14.32
SR-GNN	**70.57**	**30.94**	**71.36**	**31.89**	**50.73**	**17.59**

Table 1 – Performance of the models over the Yoochoose (1/64 and 1/4) and Diginetica datasets. The results were obtained from (WU *et al.*, 2019). The best results of each metric and dataset are highlighted in bold.

From the four presented models, three of them used standard cross-entropy. Gru4Rec is the exception that uses an approach similar to negative sampling. In the study, the authors also reported that their loss outperforms the Cross Entropy while being more stable for training. The other three studies were limited to only using Cross Entropy in their experiments.

The three last models, NARM, STAMP and SR-GNN, also have in common that they use attention and they had the best results (see Table 1). AM is becoming increasingly more common with DL and much have been developed around this subject. It is interesting, however, that in these three models, AM is used in a limited way, with the attention formula being mostly the same: the context information is the last session item or the average item representation and only one attention mechanism is used per model. Nevertheless, these three models outperform the GRU4REC model with using simple AMs.

The best model in the Table 1 is the SR-GNN, indicating that using attention together with GNN is the best approach for recommendation tasks. Also, the superior result of the GNN to instead of the RNN as aggregating layer as utilized in the NARM model indicates that spending more processing on local information, i.e. relations between items that were close in the session sequence, is more important than processing all items in a sequential manner.

CHAPTER

4

METHODS

In this chapter, we present the contributions of our work, with the main contribution being the loss functions modifications to achieve scalable loss behavior. We also present the gradient descent modification and the datasets used. Lastly, we specify the training and the validation process used in our experiments.

4.1 Loss function modifications

4.1.1 Negative Sampling

We propose a simple modification to the Negative Sampling formula, resulting in the following expression:

$$\log \sigma(v'^{\top}_{w_O} v_{w_I}) + \frac{1}{k} \sum_{i=0}^{k} \mathbb{E}_{w_i \sim P_n(w)} [\log \sigma(-v'^{\top}_{w_i} v_{w_I})] \tag{4.1}$$

, where $\frac{1}{k}$ is the normalizing constant for the noise term, consequently the right part of the sum becomes the mean noise term. The mean noise term differs from the original formula in the sense it is a mean value for all the noise samples while in the original formula it was a total value for all the noise samples. In this new formulation, one can assume that noise and the correct class have the same weight for the gradient.

4.1.2 Triplet

While the original triplet formula compares the representation of a sample with the representation of two other samples (a positive and a negative), we choose to compare the samples representation with a positive and a negative label.

$$L(x_a, y_p, y_n) = \max(0, m + \|f(x_a) - y_p\| - \|f(x_a) - y_n\|), \tag{4.2}$$

where x_a is the anchor sample embedding, y_p is the positive label embedding, i.e. x_a's label, y_n is the negative label embedding, i.e. a random label different from x_a's label, $m \in$ is the margin.

In, SRS terms, this loss modification means that a session (sample) is compared with items (labels) instead of other sessions (samples). As a consequence of this change, the learned item embeddings share the embedding space of the inferred session embeddings. This embedding unification idea is better illustrated in the Figure 6.

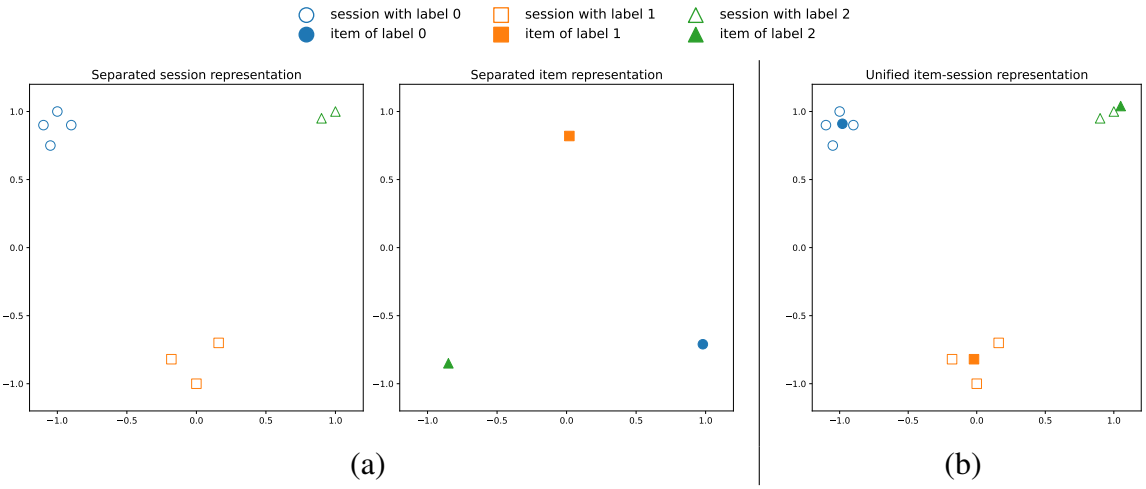
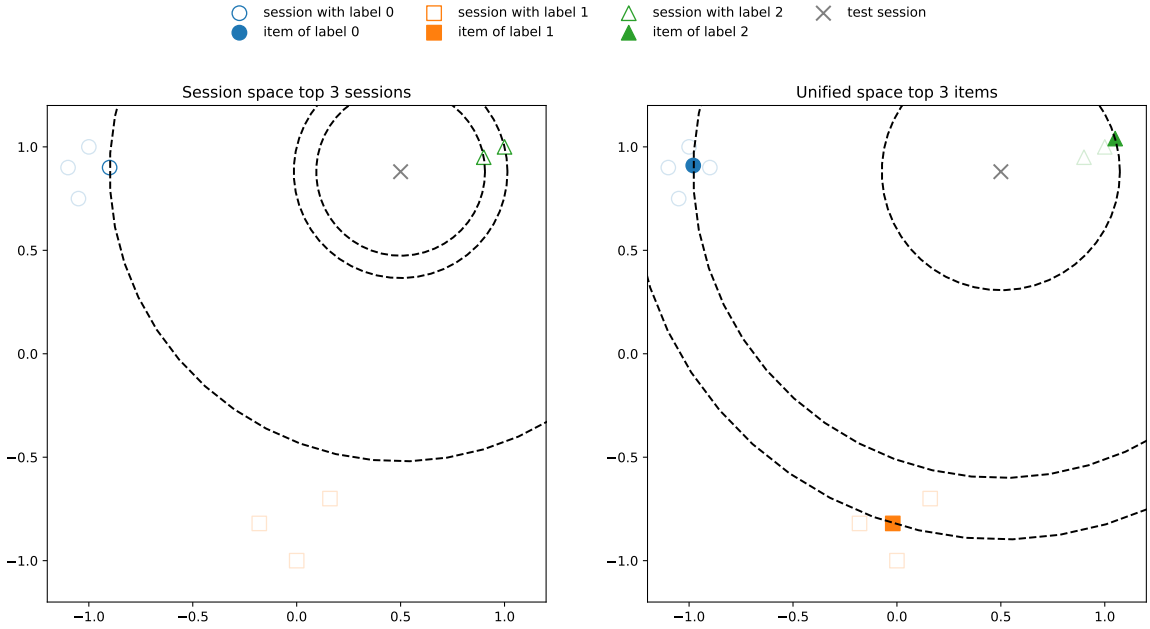


Figure 6 – Embedding spaces where coordinates of an element represent its embedding vector. (a) Two embedding spaces, one for the session and the other for the items and we cannot infer an item embedding from the corresponding sessions embeddings. (b) Unified embedding space for both items and sessions where we expect to see session embeddings close to their respective items after training a model.

This modification also simplifies the process of obtaining the best item suggestion for a given session embedding. The original method required searching for a number of similar sessions embeddings within all the session embeddings from the training split and then selecting the most similar items given the labels of these most similar sessions. With our approach, we directly search for the most similar items, which is possible because, items and sessions share the same embedding space in our approach (see Figure 7).

Performance is also affected by this choice, since the asymptotic time complexity the original approach is bounded by the number of sessions in the database while in our approach it is bounded by the number of unique items in the database. In practice if a dataset has less unique items than sessions, which is the most common scenario with SRS datasets, then this modification is more effective.



(a) (b)

Figure 7 – Embedding spaces where coordinates of an element represent its embedding vector. A target test session is indicated by the X symbol. A dashed circle is used to indicate distance from the target session to another element; small circles indicate the element is closer to the target session. (a) Session space without being unified with the item space, where the top-3 closest elements for a given target session must be three other sessions which is not enough to provide us with the top three unique best items, since the two closest sessions have the same label. (b) Unified session and item space where we can directly select the three unique closest items.

4.2 L2 normalization of session embeddings

In order to achieve decent recommendation performance with the scalable loss functions, we had to normalize the models generated session embedding. The same normalization was done in the FaceNet work (SCHROFF; KALENICHENKO; PHILBIN, 2015) where they also used the triplet loss.

The L2 normalization constrains the session embedding to live on the d-dimensional hypersphere, i.e. $||f(x)||^2 = 1$, where x is the session input and $f(x)$ is the inferred session emebedding.

Regarding the items embeddings, they were not normalized in the model inference phase before the aggregation layers, since we have found no improvement when testing it. However we did normalize them before any operation requiring them to be compared with session embeddings, including computing the triplet loss, computing the cross entropy loss and computing the validation metrics.

4.3 Sparse gradients

The models described in Chapter 3 used the Adam (KINGMA; BA, 2014) method for gradient descent weight optimization. The Adam algorithm is described as follows:

Algorithm 1 – Adam optimizer

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective) λ (weight decay)
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment)

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda \theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \varepsilon\right)$

return θ_t

The Adam algorithm updates the moments and the gradient of all parameters for each batch iteration. Each unique item in the dataset has a vector of parameters corresponding to its embedding, therefore the Adam algorithm time complexity for one batch is bounded by the number of items in the dataset.

To address this problem while minimizing the changes in the original architectures, we kept the Adam optimizer for all NN layers except the embedding layer. In the embedding layer we assigned the SparseAdam from the Pytorch package (PASZKE *et al.*, 2019).

The SparseAdam behaves similar to the Adam optimizer. The difference is in only updating the parameters and moments where the gradient was computed, which correspond exclusively to the items that appear in the sessions of a batch. This is possible given that a sparse embedding layer implementation is used, e.g. Pytorch embedding layer with sparsity option (PASZKE *et al.*, 2019). The algorithm 2 shows the modified algorithm, with the addition of the operator INBATCH that receives all parameters and returns only parameters with computed gradients in a given batch.

Although the formulation still remains similar to the original Adam algorithm, in practice the behavior of the algorithm changes with the sparse formulation. It is not hard to verify that, with the original Adam algorithm, if the gradient for a parameter is zero, then the moments of

Algorithm 2 – SparseAdam optimizer

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective) λ (weight decay)
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment)

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\text{INBATCH}(\theta_{t-1}))$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\, \text{INBATCH}(\theta_{t-1})$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  $\text{INBATCH}(\theta_t) \leftarrow \text{INBATCH}(\theta_{t-1}) - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \varepsilon\right)$

return θ_t

this parameter might not be zero and the parameter would still be updated with a different value. Therefore the algorithms are similar in formulation but behave differently.

This method drastically reduce the expected time complexity, however due to the cost to access memory with non-contiguous data this might not be completely effective in practice.

4.4 Summary of model modifications

The Table 2 summarizes the changes we propose with our methods and compares with the traditional approaches from the related deep learning works (Chapter 3).

Modification type	Traditional method	Proposed method
Loss function	Cross entropy	Triplet or negative sampling
Session embedding processing	No normalization	L2 normalization
Item embeddings optimization	Standard Adam optimizer	Adam optimizer

Table 2 – Summary of modification comparing traditional methods with ours (proposed) method.

In addition, we restricted the set of models utilized in this experiment to GRU4REC, STAMP and SRGNN. However for the GRU4REC model, we simplified its architecture to single GRU layer in the aggregation phase.

4.5 Dataset preprocessing

4.5.1 New dataset

We made use of a more challenging dataset, MeLi2020, to better explore the capabilities of the proposed approaches. The original datasets were small and even with the most demanding model, SRGNN, the mean epoch training time was only 15 min (WU *et al.*, 2019).

The MeLi dataset has similar number of sessions compared to the Yoochoose 1/4, however the number of unique items is two order of magnitudes higher (see Table 3).

In traditional SRS training methods batch training time is directly proportional to the number of unique items. Therefore, we expect a large increase in training time with traditional training methods and a small increase with the proposed approaches.

4.5.2 Train and test split

Train and test split were changed for Diginetica and Yoochoose. Before, the date of session was the date of the first seem item in the session. In the new implementation, the date of a session becomes the last seem item date in the session. This change enforces that any augmented sub-session, i.e. same session starting at different item, is in the same split as the original session. This is more ideal to avoid label data leaking from train to test.

4.5.3 Maximum session length

The maximum session length was changed from 20 to 25. This increment was made with the objective maintaining pf some equivalence with all datasets. The original datasets are up to 5.1 items long in average while the new MeLi2020 dataset has a 21.7 average session length. Therefore the increment keeps the threshold above the average session length for all datasets.

4.5.4 Preprocessing steps

Below are common preprocessing steps we used in our preprocessing pipeline:

1. Remove sessions with length equal to 1
2. Remove items that appear less than 5 times
3. Remove sessions with length equal to 1
4. Split sessions into train and test in way that train sessions have maximum date lower than a threshold and test have greater or equal maximum date than the threshold
5. Remove test session items that do no appear in train

6. Remove test sessions with length equal to 1
7. Trim sampled sessions to the last 25 items
8. Augment samples by splitting the sessions with a sliding window changing the session first item while maintaining the last item

4.5.5 Datasets statistics

The resulting datasets statistics after preprocessing can be found in the Table 3.

Statistics	Yoochoose 1/4	Yoochoose 1/64	Diginetica	MeLi2020
# unique items	33,280	17,511	104,419	1,279,791
# train session	1,925,647	117,178	186,670	247,834
# train session (augmented)	5,922,115	370,130	819,081	4,675,443
avg. session length	4.70	5.07	4.66	21.74

Table 3 – Relevant statistics to estimate the training time performance on each dataset.

4.6 Training and evaluation

Given the large number of experimental settings training was kept simple. In all settings, training was performed for 8 epochs with no early stopping and with batches of size 128 samples. Most models would converge between 2 and 5 epochs, so the choice of 8 epochs was meant to give some extra margin for convergence. Each experimental setting was executed only once.

No strict method was used for finding the optimal hyper-parameters. Most hyper-parameters values correspond to slight changes in the hyper-parameters reported in the related works. Along with this, each model had the same hyper-parameters used in all datasets it was trained.

For each epoch, we evaluated the test set using the MRR@20 metric and we kept the best evaluation value of all epochs as the final result of each setting.

CHAPTER

5

RESULTS

In this chapter, we present the results of three experiments aimed at measuring the importance and the efficiency of the methods proposed.

5.1 Experiment I - Performance baseline with traditional methods

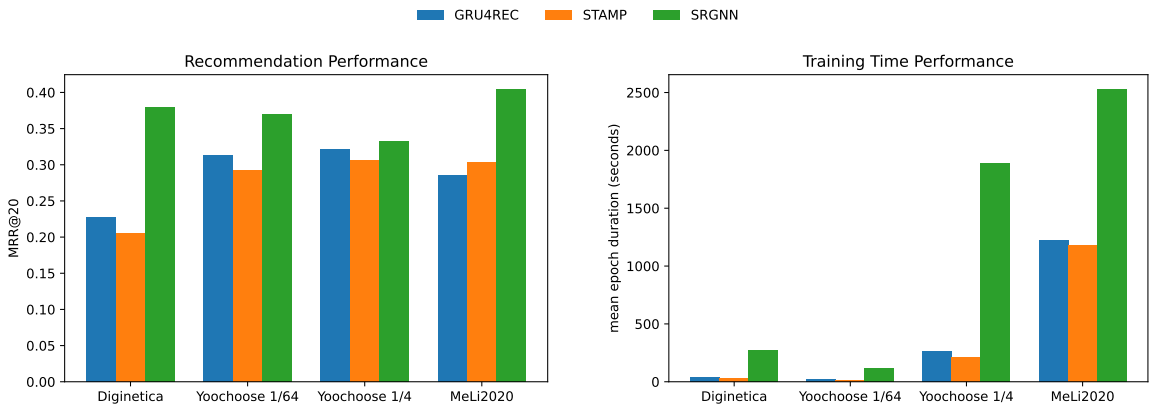
This experiment compares recommendation performance and training time performance with the traditional approach (cross entropy and no sparse gradients) over multiple settings of datasets and models.

Two objectives are covered: 1) to have a performance baseline revealing if the proposed model training results in satisfactory recommendation performance and whether they are more time effective, 2) to evaluate if a larger dataset (MeLi2020) evidences the necessity of more scalable algorithms given the required training time.

The experimental setting includes the GRU4REC, STAMP and SRGNN models described in the Chapter 3 applied to the four datasets described in the Section 4.5. The experiment results are shown in the Figure 8.

Recommendation performance shows that SRGNN out performs the other models while STAMP and GRU4REC show similar performance. Given the similar performance of STAMP and GRU4REC, we removed the latter from the list of models in the next experiments.

Training time increases by at least one order of magnitude from the smaller datasets (Diginiteca and Yoochoose 1/64) to the larger datasets (Yoochoose 1/4 and MeLi2020). MeLi2020 is the slowest dataset to process, taking up to 2500s to train only one epoch. This growth matches the expected asymptotic curve of time complexity. In the traditional architectures epoch training is $O(nm)$, n and m being respectively the number of samples and the number of unique items.



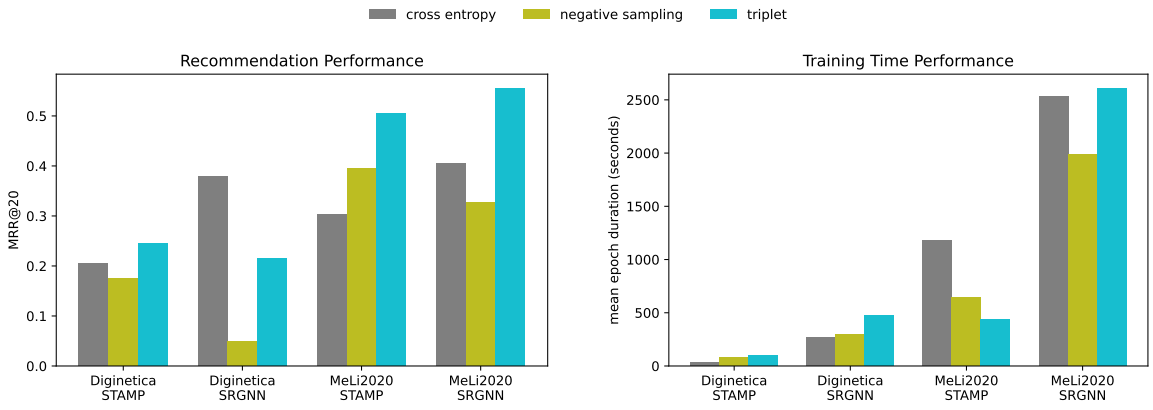
(a) (b)

Figure 8 – Recommendation (a) and training time (b) performance on each dataset for each model.

The training time for the SRGNN stands out as being much slower than the other models. On the larger datasets (Yoochoose 1/4 and MeLi2020), it did not change as much as the other two models from Yoochoose 1/4 to MeLi2020. This might indicate implementation problems and requires further investigation.

5.2 Experiment II - Performance with the proposed methods

This experiment investigate the recommendation performance and the training time performance of STAMP and SRGNN using the proposed modifications detailed in Chapter 4. We execute the training of models using triplet and negative sampling loss on the datasets Diginetica and MeLi2020 and compared the results against the traditional method baseline (cross entropy) for the same models.



(a) (b)

Figure 9 – Recommendation (a) and training time (b) performance on each dataset for each loss function. Cross entropy stands for the traditional methods results obtained in the experiment I.

Figure 9 (a) shows recommendation performance, where higher is better. Triplet training

over performs cross entropy recommendation performance in three out of four cases. Negative sampling out performs cross-entropy recommendation performance in one case and keeps a competitive result in other two cases.

Figure 9 (b) shows training time performance, where lower is better. The proposed approaches are slower when trained on the smaller Digiteca dataset with both models, however they show better performance on the larger MeLi2020 dataset. With the STAMP model on the MeLi2020 dataset, both proposed approaches are faster taking half of the time or less to train. With the SRGNN model on the MeLi2020 dataset, the results are not as sharp, negative sample is faster but only slightly in proportion and triplet is slightly slower, but proportionally close indicating improvement compared to training on the smaller Diginetica dataset.

These results show that the scalable approaches are similar in recommendation performance to the traditional non-scalable approaches, while having a trend to perform faster when training on larger datasets, depending on the model architecture.

5.3 Experiment III - Performance contribution of L2 normalization on session inference

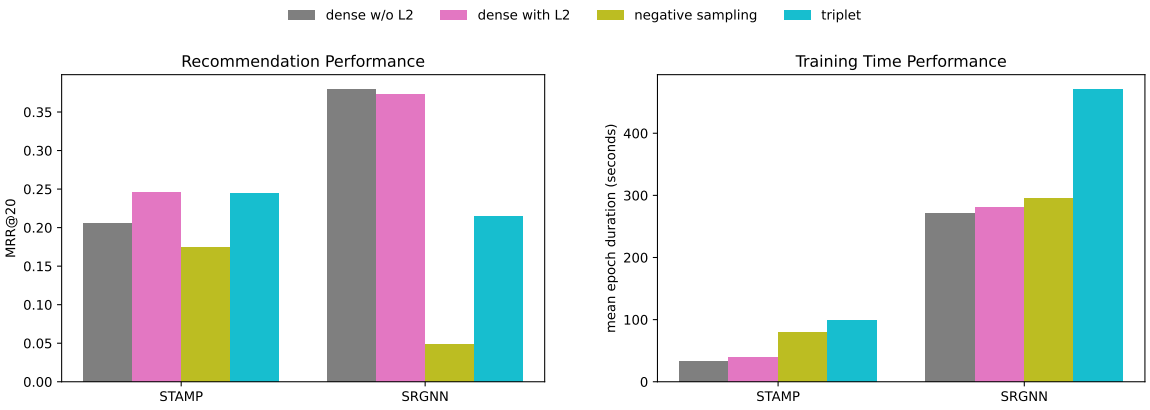
The proposed approaches required L2 normalization to have competitive recommendation performance. Without it the results were extremely low. However, since L2 normalization is only present in the proposed approaches, it must be tested whether using it with the traditional approaches provide performance improvements, which could result in the proposed methods performance from the Experiment II not being competitive anymore.

This experiment isolates the L2 normalization effect by applying it on the dense traditional methods as well, allowing us to estimate how much of the proposed methods improvement can be explained by the L2 normalization. Therefore the traditional approach is splitted into two evaluation groups: dense without L2 and dense with L2.

The experimental setting includes the Diginetica dataset along with the STAMP and SRGNN models.

Dense with L2 has better recommendation performance with the STAMP model, but the opposite happens with the SRGNN model. This means that using the L2 normalization can't guarantee better results. Also, even if the approximate improvement of 0.5 with the STAMP model would generalize to other settings not tested in this experiment, it would still not be enough to change the conclusions from the Experiment II. Therefore this small evidence indicates the L2 normalization is not an unfair performance boost against the traditional methods. On the other hand, more experiments are necessary to quantify precisely the L2 normalization effect on recommendation.

In the STAMP model condition, when comparing the traditional (dense) approaches,



(a) (b)

Figure 10 – Recommendation (a) and training time (b) performance with STAMP and SRGNN models on the Diginetica dataset.

using L2 normalization shows improvement. Given that adding L2 normalization has a small extra training time cost, it is worth considering it as potential improvement on any architecture that does not use it.

5.4 Summary of the results

The results of this chapter can be summarized as the following:

- **Experiment I:** Larger dataset can take up to 1h to train one epoch, meaning that faster and more scalable approaches are a important.
- **Experiment II:** The proposed approaches with scalable losses have similar recommendation performance compared to the traditional approaches, sometimes under performing them and in other times over performing them. Regarding training time performance, the proposed approaches are slower for small datasets, however they are faster or similar on larger datasets, in three out of four training settings. Given that the training time in larger datasets is the most important aspect of this work, the proposed methods are a success.
- **Experiment III:** The L2 normalization is a necessity for the proposed approaches to have decent results, however it is not something that the traditional approaches can always benefit from.

CHAPTER

6

CONCLUSIONS

In this work, we investigated two different loss function approaches to improve the scalability of SRS DL models. Our experiments showed our approach can produce models with similar or faster training time on large datasets, while maintaining similar performance to traditional approaches.

As expected, we observed that our method performed better on datasets with a large number of unique items, which suggests that it could be particularly useful for real-world applications with tens of millions of items or more. However, we also noted that further research is needed to address memory management issues in systems with a large number of items.

Overall, our work contributes to the development of single DL model SRS systems aimed at making suggestions from a large number of options. This could lead to better software development experiences and improved recommendations for end-users.

As for future work, we propose several directions that could improve our approach. First, we suggest exploring different pre-processing techniques to increase the number of unique items on the larger datasets and potentially enhance the efficiency of our proposed approaches. Second, we recommend performing more precise model fine-tuning to better evaluate the recommendation performance of our proposed approaches. Third, we suggest running more experiments with L2 normalization, testing on additional datasets, and comparing the performance of our proposed approaches with and without L2 normalization. Fourth, we suggest investigating the possible impact of graph data structure computation overhead on the performance of our SRGNN model training time. Finally, we recommend exploring other training time metrics, such as time to convergence or time to reach a given baseline, to provide a more comprehensive evaluation of our proposed approaches.

BIBLIOGRAPHY

CHO, K.; MERRIËNBOER, B. V.; GULCEHRE, C.; BAHDANAU, D.; BOUGARES, F.; SCHWENK, H.; BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. **arXiv preprint arXiv:1406.1078**, 2014. Citation on page 25.

DAVIDSON, J.; LIEBALD, B.; LIU, J.; NANDY, P.; VLEET, T. V.; GARGI, U.; GUPTA, S.; HE, Y.; LAMBERT, M.; LIVINGSTON, B. *et al.* The youtube video recommendation system. In: **Proceedings of the fourth ACM conference on Recommender systems**. [S.l.: s.n.], 2010. p. 293–296. Citation on page 21.

GERS, F. A.; SCHMIDHUBER, J.; CUMMINS, F. Learning to forget: Continual prediction with lstm. **Neural Computation**, v. 12, n. 10, p. 2451–2471, 2000. Citation on page 26.

GROVER, A.; LESKOVEC, J. node2vec: Scalable feature learning for networks. In: **Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.: s.n.], 2016. p. 855–864. Citation on page 30.

HAMILTON, W.; YING, Z.; LESKOVEC, J. Inductive representation learning on large graphs. In: GUYON, I.; LUXBURG, U. V.; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (Ed.). **Advances in Neural Information Processing Systems**. Curran Associates, Inc., 2017. v. 30. Available: <https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf>. Citation on page 28.

HIDASI, B.; KARATZOGLOU, A.; BALTRUNAS, L.; TIKK, D. **Session-based Recommendations with Recurrent Neural Networks**. 2016. Citations on pages 9, 22, 29, and 33.

KINGMA, D. P.; BA, J. **Adam: A Method for Stochastic Optimization**. arXiv, 2014. Available: <https://arxiv.org/abs/1412.6980>. Citation on page 42.

LI, J.; REN, P.; CHEN, Z.; REN, Z.; LIAN, T.; MA, J. Neural attentive session-based recommendation. In: **Proceedings of the 2017 ACM on Conference on Information and Knowledge Management**. [S.l.: s.n.], 2017. p. 1419–1428. Citations on pages 9, 22, 34, and 35.

LI, Y.; TARLOW, D.; BROCKSCHMIDT, M.; ZEMEL, R. Gated graph sequence neural networks. In: **International Conference on Learning Representations**. [S.l.: s.n.], 2016. Citations on pages 28 and 36.

LINDEN, G.; SMITH, B.; YORK, J. Amazon.com recommendations: item-to-item collaborative filtering. **IEEE Internet Computing**, v. 7, n. 1, p. 76–80, 2003. Citation on page 21.

LIU, Q.; ZENG, Y.; MOKHOSI, R.; ZHANG, H. Stamp: short-term attention/memory priority model for session-based recommendation. In: **Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining**. [S.l.: s.n.], 2018. p. 1831–1839. Citations on pages 9, 22, and 35.

MIKOLOV, T.; SUTSKEVER, I.; CHEN, K.; CORRADO, G.; DEAN, J. Distributed representations of words and phrases and their compositionality. **arXiv preprint arXiv:1310.4546**, 2013. Citation on page 30.

MORIN, F.; BENGIO, Y. Hierarchical probabilistic neural network language model. In: CITESEER. **Aistats**. [S.l.], 2005. v. 5, p. 246–252. Citation on page 29.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L.; DESMAISON, A.; KOPF, A.; YANG, E.; DEVITO, Z.; RAISON, M.; TEJANI, A.; CHILAMKURTHY, S.; STEINER, B.; FANG, L.; BAI, J.; CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In: WALLACH, H.; LAROCHELLE, H.; BEYGELZIMER, A.; ALCHé-BUC, F. d'; FOX, E.; GARNETT, R. (Ed.). **Advances in Neural Information Processing Systems 32**. Curran Associates, Inc., 2019. p. 8024–8035. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>. Citation on page 42.

SCHROFF, F.; KALENICHENKO, D.; PHILBIN, J. Facenet: A unified embedding for face recognition and clustering. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 815–823. Citations on pages 30 and 41.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L. u.; POLOSUKHIN, I. Attention is all you need. In: GUYON, I.; LUXBURG, U. V.; BENGIO, S.; WALLACH, H.; FERGUS, R.; VISHWANATHAN, S.; GARNETT, R. (Ed.). **Advances in Neural Information Processing Systems**. Curran Associates, Inc., 2017. v. 30. Available: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>. Citation on page 26.

VELIčKOVIć, P.; CUCURULL, G.; CASANOVA, A.; ROMERO, A.; LIò, P.; BENGIO, Y. Graph attention networks. In: **International Conference on Learning Representations**. [s.n.], 2018. Available: <https://openreview.net/forum?id=rJXMpikCZ>. Citation on page 28.

WANG, J.; SONG, Y.; LEUNG, T.; ROSENBERG, C.; WANG, J.; PHILBIN, J.; CHEN, B.; WU, Y. Learning fine-grained image similarity with deep ranking. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2014. p. 1386–1393. Citation on page 30.

WU, S.; TANG, Y.; ZHU, Y.; WANG, L.; XIE, X.; TAN, T. Session-based recommendation with graph neural networks. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2019. v. 33, n. 01, p. 346–353. Citations on pages 9, 13, 22, 36, 37, 38, and 44.


ICMC
USP
SÃO CARLOS