Recurrent Neural Networks with Top-k Gains for Session-based Recommendations

Balázs Hidasi
Gravity R&D
Budapest, Hungary
balazs.hidasi@gravityrd.com

Alexandros Karatzoglou
Telefonica Research
Barcelona, Spain
alexk@tid.es

ABSTRACT

RNNs have been shown to be excellent models for sequential data and in particular for data that is generated by users in an session-based manner. The use of RNNs provides impressive performance benefits over classical methods in session-based recommendations. In this work we introduce novel ranking loss functions tailored to RNNs in the recommendation setting. The improved performance of these losses over alternatives, along with further tricks and refinements described in this work, allow for an overall improvement of up to 35% in terms of MRR and Recall@20 over previous session-based RNN solutions and up to 53% over classical collaborative filtering approaches. Unlike data augmentation-based improvements, our method does not increase training times significantly. We further demonstrate the performance gain of the RNN over baselines in an online A/B test.

KEYWORDS

recurrent neural networks; loss function; ranking; session-based recommendation; recommender systems

ACM Reference Format:
Balázs Hidasi and Alexandros Karatzoglou. 2018. Recurrent Neural Networks with Top-k Gains for Session-based Recommendations . In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18), October 22–26, 2018, Torino, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3269206.3271761

1 INTRODUCTION

Session-based recommendation is a very common recommendation problem that is encountered in many domains such as e-commerce, classified sites, music and video recommendation. In the session-based setting, past user history logs are often not available (either because the user is new or not logged-in or not tracked) and recommender systems have to rely only on the actions of the user in the current sessions to provide accurate recommendations. Until recently many of these recommendations tasks were tackled using relatively simple methods such as item-based collaborative filtering [17] or content-based methods. Recurrent Neural Networks (RNNs) have emerged from the deep learning literature as powerful methods for modeling sequential data. These models have been successfully applied in speech recognition, translation, time series forecasting and signal processing. In recommender systems RNNs have been recently applied to the session-based recommendation setting with impressive results [7].

The advantage of RNNs over traditional similarity-based methods for recommendation is that they can effectively model the whole session of user interactions (clicks, views, etc.). By modeling the whole session, RNNs can in effect learn the 'theme' of the session and thus provide recommendations with increased accuracy (between 20%-30%) over traditional methods.

RNNs have been adapted to the task of session-based recommendation. One of the main objectives in recommendation is to rank items by user preference; i.e. the exact ranking or scoring of items in the tail of the item list (items that the user will not like) is not that important, but it is very important to rank correctly the items that the user will like at the top of the list (first 5, 10 or 20 positions). To achieve this with machine learning, one has to typically utilize learning to rank techniques(see e.g. [3]) and in particular ranking objectives and loss functions. The current session-based RNN approaches use ranking loss functions and, in particular, pairwise ranking loss functions. As in most deep learning approaches the choice of a good ranking loss can have a very significant influence on performance. Since deep learning methods need to propagate gradients over several layers and in the case of RNNs 'back in time' over previous steps, to optimize the model parameters, the quality of these gradients originating from the loss function influences the quality of the optimization and the model parameters. Moreover the nature of the recommendation task, which typically entails large output spaces (due to large number of items), poses unique challenges that have to be taken into account as well when designing a proper ranking loss function. We will see that the way this large output space issue is tackled is very crucial in achieving good performance.

In this work we analyze ranking loss functions used in RNNs for session-based recommendations. This analysis leads to a new set of ranking loss functions that increase the performance of the RNN up to 35% over previous commonly used losses without significant computational overheads. We essentially devise a new class of loss functions that combines learnings from the deep learning and the learning to rank literature. Experimental results on several datasets coming from industry validate these improvements by showing significant increase in recommendation accuracy measured by Mean Reciprocal Rank (MRR) and Recall@20. With these improvements the difference between RNNs and conventional memory-based collaborative filtering jumps to 53% in terms of MRR and Recall@20,

CIKM '18, October 22–26, 2018, Torino, Italy
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6014-2/18/10...$15.00
https://doi.org/10.1145/3269206.3271761

demonstrating the potential that deep learning methods bring to the area of Recommender Systems.

1.1 Related Work

One of the main approaches that is employed in session-based recommendation and a natural solution to the problem of a missing user profile is the item-to-item recommendation approach [13, 17]. In this setting, an item-to-item similarity matrix is precomputed from the available session data, items that are often clicked together in sessions are deemed to be similar. This similarity matrix is then used during the session to recommend the most similar items to the one the user has currently clicked.

Long Short-Term Memory (LSTM) [10] networks are a type of RNNs that have been shown to solve the optimization issues that plague vanilla-type RNNs. LSTMs include additional gates that regulate when and how much of the input should be taken into account and when to reset the hidden state. A slightly simplified version of LSTM – that still maintains all their properties – is the Gated Recurrent Unit (GRU) [5], which we use in this work. Recurrent Neural Networks have been used with success in the area of session-based recommendations; [7] proposed a Recurrent Neural Network with a pairwise ranking loss for this task. [19] proposed data augmentation techniques to improve the performance of the RNN for session-based recommendations, however these techniques have the side effect of increasing training times as a single session is split into several sub-sessions for training. Session-based RNNs have been augmented [8] with feature information, such as text and images from the clicked/consumed items, showing improved performance over the plain models. RNNs have also been used in more standard user-item collaborative filtering settings where the aim is to model the evolution of the user and items factors [21],[6] where the results are less striking, with the proposed methods barely outperforming standard matrix factorization methods. This is to be expected as there is no strong evidence on major user taste evolution in a single domain in the timeframes of the available datasets and sequential modeling of items that are not 'consumed' in sessions such as movies might not bring major benefits.

Another area touched upon in this work are loss functions tailored to recommender systems requirements. This typically means ranking loss functions. In this area there has been work particularly in the context of matrix factorization techniques. One of the first learning to rank techniques for collaborative filtering was introduced in [20]. Essentially a listwise loss function was introduced along with an alternating bundle method for optimization of the factors. Further ranking loss function for collaborative filtering were introduced in [18] [15] and [12]. Note that the fact that these loss functions work well in matrix factorization does not guarantee in any way that they are an optimal choice for RNNs as backpropagation requirements are stronger than those posed by simple SGD. We will in fact see that BPR, a popular choice of loss function, needs to be significantly modified to extract optimal results in the case of RNNs for session-based recommendations. Another work related to sampling large output spaces in deep networks for efficient loss computations for language models is the 'blackout' method [11], where essentially a sampling procedure similar to the one used in [7] is applied in order to efficiently compute the categorical cross-entropy loss.

2 SAMPLING THE OUTPUT

In the remainder of the paper we will refer to the RNN algorithm implemented in [7] as GRU4Rec, the name of the implementation published by the authors on GitHub [1]. In this section we revisit how GRU4Rec samples negative feedback on the output and discuss its importance. We extend this sampling with an option for additional samples and argue that this is crucial for the increased recommendation accuracy we achieve (up to 53% improvement).

In each training step, GRU4Rec takes the item of the current event in the session – represented by a one-hot vector – as an input. The output of the network is a set of scores over the items, corresponding to their likelihood of being the next item in the session. The training iterates through all events in the sequence. The complexity of the training with backpropagation through time is $O(N_E(H^2 + HN_O))$ where N_E is the number of training events, H is the number of hidden units and N_O is the number of outputs, for which scores are computed. Computing scores for all items is very impractical, since it makes the network unscalable[2]. Therefore GRU4Rec uses a sampling mechanism and computes the scores for only a subset of the items during training.

Instead of making a forward and backward pass with one training example only and then moving to the next, the network is fed with a bundle of examples and is trained on the mean gradient. This common practice is called *mini-batch training* and has several benefits, e.g. utilizing the parallelization capabilities of current hardware better, thus training faster, and producing more stable gradients than stochastic gradient training and thus converging faster. GRU4Rec introduced mini-batch based sampling [7]. For each example in the mini-batch, the other examples of the same mini-batch serve as negative examples (see Figure 1).[3] This method is practical from an implementation point of view and can be also implemented efficiently for GPUs.

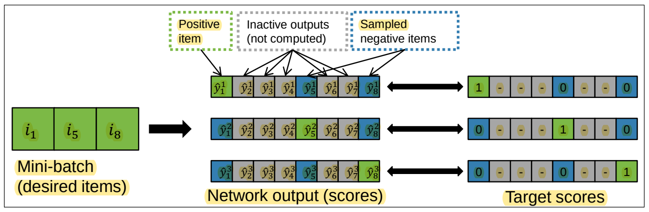

Figure 1: Mini-batch based negative sampling.

The network can be trained with one of three different listwise ranking loss functions (see Section 3). All loss functions require a score for the target item (i.e. for the item which was the actual next item) and score(s) for at least one negative sample (i.e. item other

[1]https://github.com/hidasib/GRU4Rec

[2]While it can still result in an acceptable training time for smaller datasets, especially if the number of items is only a few tens of thousand, algorithms scaling with the product of the number of events and items cannot scale up for larger datasets

[3]E.g. assume a mini-batch of 32 examples, with one desired output (target) for each example. Scores are computed for all 32 items for each of the 32 examples resulting in $32 \times 32 = 1024$ scores. Thus we have 31 scores of negative examples for each of the targets.

than the target). One property of ranking losses is that learning happens only if the score of the target item does not exceed that of the negative samples by a large margin, otherwise the items are already in the right order, so there is nothing to be learned. Therefore, when utilizing a sampling procedure, it is crucial that high scoring items are included among the negative samples. Whether an item has a high score, depends on the context (item sequence) for which the scores are actually computed for. Popular items generally score high in many situations, making popularity-based sampling a good sampling strategy. Mini-batch sampling is basically a form of popularity-based sampling, since the training iterates through all events, thus the probability of an item acting as a negative sample is proportional to its support. The problem with popularity-based sampling is that learning can slow down after the algorithm learns to (generally) rank target items above popular ones, and thus can still be inaccurate for ranking long tail high scoring items. On the other hand, uniform sampling slows down learning, due to the high number of low scoring negative samples, but might produce an overall more accurate model if trained indefinitely. In our experience, popularity-based sampling generally produces better results.

Tying sampling to the mini-batches has several practical benefits, but is too restrictive for three reasons. (1) Mini-batch sizes are generally small, ranging from few tens to few hundreds. If the number of items is large, the small sample size further hinders the chance of including all of the high scoring negative examples. (2) Mini-batch size has a direct effect on the training. E.g. we found that training with smaller mini-batch sizes (30-100) produces more accurate models, but training with larger ones is faster on the GPU due to parallelization. (3) The sampling method is inherently popularity-based, which is a good strategy generally, but might not be optimal for all datasets.

Therefore we extend the sampling of GRU4Rec with additional samples. We sample N_A items which are shared by the examples of the mini-batch, i.e. the same samples are used for each example[4]. These additional samples are used along with the $N_B - 1$ samples coming from the mini-batch based sampling (described above). The additional samples can be sampled in any way. We chose to sample proportional to supp_i^α, where supp_i is the support of the item and α is the parameter of the sampling ($0 \leq \alpha \leq 1$). $\alpha = 0$ and $\alpha = 1$ gives uniform and popularity-based sampling respectively.

Adding more samples naturally increases the complexity, since N_O increases from N_B to $N_A + N_B$. However, the computations are easily parallelizable, thus there is no actual increase in the training time on modern GPUs up to a certain sample size (see Section 4.1). The efficient implementation of this sampling however is not trivial. Sampling according to a distribution on GPUs is not well supported and thus slow, therefore it should be either handled by the CPU or requires some form of workaround. In the former case the sampled IDs should be given to the GPU along with the item IDs of the mini-batch. But sampling the distribution takes some time every time a new mini-batch is formed, thus GPU execution is frequently interrupted, making GPU utilization low and thus training slow. In the latter case (i.e. workaround on the GPU), sampling by distribution is translated to a sequence of multiple GPU operations, resulting in an overall faster execution than the built-in (one-step) sampling methods of the deep learning framework we use. In both cases, sampling a few items at once is less efficient than sampling lots of them. Therefore we also implemented a cache that pre-samples and stores lots of negative samples. Training uses up these samples and the cache is recomputed once it is empty. We found that pre-sampling 10-100 million item IDs significantly improves training speed when compared to using no cache at all.

[4]However, the scores of these samples will be still different per example, because of the differing item sequences they are based on.

3 LOSS FUNCTION DESIGN

In this section we examine the loss functions implemented in GRU4Rec and identify their weaknesses. We propose two ways to stabilize the numerical instability of the cross-entropy loss, we show how learning with the TOP1 and BPR pairwise losses degrades as we add more samples to the output, and propose a family of loss functions based on pairwise losses that alleviates this problem. We note that, while our aim is to improve GRU4Rec, the loss functions proposed in this section can be also used with other models, such as matrix factorization.

3.1 Categorical cross-entropy

Categorical cross-entropy measures the distance of a proposed (discrete) probability distribution q from the target distribution p as defined by (1).

$$H(p,q) = -\sum_{j=1}^{N} p_j \log q_j \tag{1}$$

This loss is often used in machine learning and deep learning in particular for multi-class classification problems. Next item recommendation can be interpreted as classification, where the class labels are the items in the system and item sequences need to be assigned with the label of the item that follows. In a single-label scenario – such as next item recommendation – the target distribution is a one-hot vector over the set of items, with the coordinate corresponding to the target item set to 1. The proposed distribution consists of the scores assigned to the items by the algorithm. The output scores need to be transformed to form a distribution. It is common practice to use the *softmax* transformation (2), which is a continuous approximation of the *max* operation. This naturally aligns with the sentiment that the label with the highest score is assigned to the sequence.

$$s_i = \frac{e^{r_i}}{\sum_{j=1}^{N} e^{r_j}} \tag{2}$$

Cross-entropy in itself is a pointwise loss, as it is the sum of independent losses defined over the coordinates. Combining it with softmax introduces listwise properties into the loss, since the loss now cannot be separated over coordinates. Putting them together we get the following loss function over the scores (assuming that the target item is indexed by i):

$$L_{\text{xe}} = -\log s_i = -\log \frac{e^{r_i}}{\sum_{j=1}^{N} e^{r_j}} \tag{3}$$

Fixing the instability: One of the losses available in GRU4Rec was cross-entropy with softmax scores. [7] reported slightly better results than with other losses, but deemed the loss to be unstable for a large fraction of the hyperparameter space and thus advised against its use. This instability comes from the limited numerical precision. Assuming that there is a k for which $r_k \gg r_i$, s_i becomes very small and rounded to 0, because of the limited precision. The loss then computes $\log 0$, which is undefined. Two ways to circumvent this problem are as follow: (a) compute $-\log(s_i + \epsilon)$, where ϵ is a very small value (we use 10^{-24}); (b) compute $-\log s_i$ directly as $-r_i + \log \sum_{j=1}^{N} e^{r_j}$. The former introduces some noise, while the latter does not allow the separated use of the transformation and the loss, but both methods stabilize the loss. We did not observe any differences in the results of the two variants.

3.2 Ranking losses: TOP1 & BPR

GRU4Rec offers two loss functions based on pairwise losses. Pairwise losses compare the score of the target to a negative example (i.e. any item other than the target). The loss is high if the target's score is higher than that of the negative example. GRU4Rec computes scores for multiple negative samples per each target, and thus the loss function is composed as the average of the individual pairwise losses. This results in a listwise loss function, which is composed of pairwise losses.

One of the loss functions is coined TOP1 (4). It is a heuristically put together loss consisting of two parts. The first part aims to push the target score above the score of the samples, while the second part lowers the score of negative samples towards zero. The latter acts as a regularizer, but instead of constraining the model weights directly, it penalizes high scores on the negative examples. Since all items act as a negative score in one training example or another, it generally pushes the scores down.

$$L_{\text{top1}} = \frac{1}{N_S} \sum_{j=1}^{N_S} \sigma(r_j - r_i) + \sigma(r_j^2) \tag{4}$$

j runs over the N_S sampled negative (non-relevant) items, relevant items are indexed by i.

The other loss function (5) is based on the popular Bayesian Personalized Ranking (BPR) [16] loss. Here the negative log-probability of the target score exceeding the sample scores is minimized (i.e. the probability of target scores being above sample scores is maximized). The non-continuous $P(r_i > r_j)$ is approximated by $\sigma(r_i - r_j)$.

$$L_{\text{bpr}} = -\frac{1}{N_S} \sum_{j=1}^{N_S} \log \sigma(r_i - r_j) \tag{5}$$

3.2.1 Vanishing gradients. Taking the average of individual pairwise losses has an undesired side effect. Examining the gradients for the TOP1 and BPR losses w.r.t. the target score r_i, ((6) and (7) respectively) reveals that under certain circumstances gradients vanish and thus learning stops. With pairwise losses, one generally wants to have negative samples with high scores, as those samples produce high gradients. Or intuitively, if the score of the negative sample is already well below that of the target, there is nothing to learn from that negative sample anymore. For this discussion we will denote samples where $r_j \ll r_i$ *irrelevant*. For an irrelevant sample $\sigma(r_j - r_i)$ in ((6) and $1 - \sigma(r_i - r_j)$ (7) will be close to zero. Therefore, any irrelevant sample adds basically nothing to the total gradient. Meanwhile the gradient is always discounted by the total number of negative samples. By increasing the number of samples, the number of irrelevant samples increases faster than that of including relevant samples, since the majority of items are irrelevant as negative samples. This is especially true for non-popularity-based sampling and high sample numbers. Therefore the gradients of these losses start to vanish as the number of samples increases, which is counterintuitive and hurts the full potential of the algorithm.[5][6]

$$\frac{\partial L_{\text{top1}}}{\partial r_i} = -\frac{1}{N_S} \sum_{j=1}^{N_S} \sigma(r_j - r_i)\left(1 - \sigma(r_j - r_i)\right) \tag{6}$$

$$\frac{\partial L_{\text{bpr}}}{\partial r_i} = -\frac{1}{N_S} \sum_{j=1}^{N_S} \left(1 - \sigma(r_i - r_j)\right) \tag{7}$$

Note, that TOP1 is also somewhat sensitive to relevant examples where $r_j \gg r_i$, which is an oversight in the design of the loss. While this is unlikely to happen, it cannot be outruled. For example, when comparing a niche target to a very popular sample – especially during the early phase of learning – the target score might be much lower than the sample score.

We concentrated on the gradients w.r.t. the target score, but a similar issue can be observed for the gradients on the negative scores. The gradient w.r.t. the score of a negative sample is the gradient of the pairwise loss between the target and the sample divided by the number of negative samples. This means that even if all negative samples would be relevant, their updates would still diminish as their number grows.

3.3 Ranking-max loss function family

To overcome the vanishing of gradients as the number of samples increase, we propose a new family of listwise loss functions, based on individual pairwise losses. The idea is to have the target score compared with the most relevant sample score, which is the maximal score amongst the samples. The general structure of the loss is described by (8).

$$L_{\text{pairwise-max}}\left(r_i, \{r_j\}_{j=1}^{N_S}\right) = L_{\text{pairwise}}(r_i, \max_j r_j) \tag{8}$$

The maximum selection is non-differentiable and thus cannot be used with gradient descent. Therefore we use the softmax scores to preserve differentiability. Here, the softmax transformation is only used on the negative examples (i.e. r_i is excluded), since we are looking from the maximum score amongst the negative examples. This naturally results in loss functions where each negative sample is taken into account proportional to its likelihood of having the maximal score. Based on this general idea, we now derive the TOP1-max and BPR-max loss functions.

TOP1-max: The TOP1-max loss is fairly straightforward. The regularizing part does not necessarily need to be only applied for the

[5]Simply removing the discounting factor does not solve this problem, since it is equivalent of multiplying the learning rate by N_S. This would destabilize learning due to introducing high variance into the updates.

[6]For BPR, there is the option of maximizing the sum of individual pairwise probabilities $\sum_{j=1}^{N_S} P(r_i > r_j)$, i.e. minimizing $-\log \sum_{j=1}^{N_S} \sigma(r_i - r_j)$. However, this loss has even worse properties.

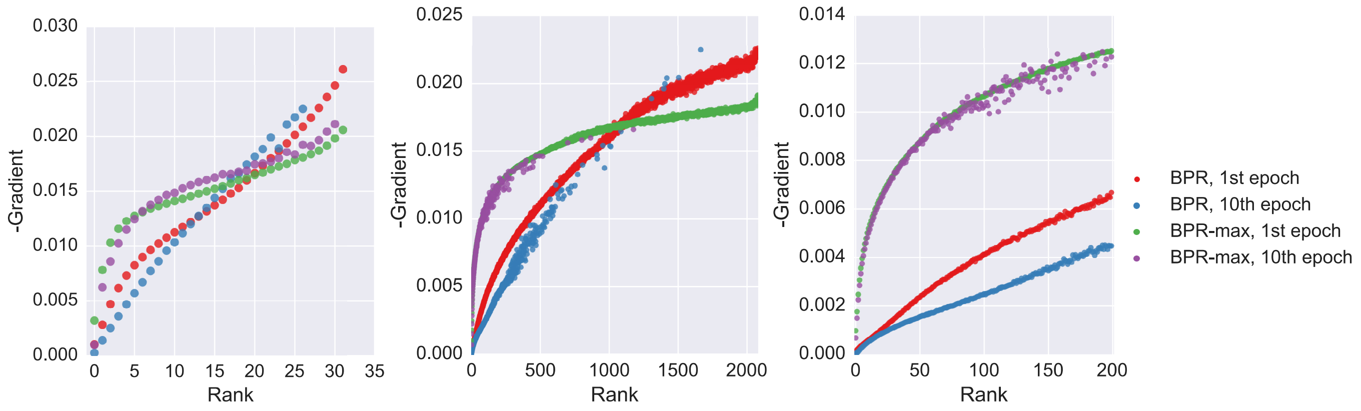

Figure 2: Median negative gradients of BPR and BPR-max w.r.t. the target score against the rank of the target item. Left: only minibatch samples are used (minibatch size: 32); Center: 2048 additional negative samples were added to the minibatch samples; Right: same setting as the center, focusing on ranks 0-200.

maximal negative score, however we found that this gave the best results, thus we kept it this way. The continuous approximation to the maximum selection entails summing over the individual losses weighted by the corresponding softmax scores s_j, giving us the TOP1-max loss (9).

$$L_{\text{top1-max}} = \sum_{j=1}^{N_S} s_j \left(\sigma(r_j - r_i) + \sigma(r_j^2) \right) \tag{9}$$

The gradient of TOP1-max (10) is the softmax weighted average[7] of individual pairwise gradients. If r_j is much lower than the maximum of negative scores, its weight will be almost zero and more weight will be placed on examples with scores close to the maximum. This solves the issue of vanishing gradients with more samples, because irrelevant samples will be just ignored, while the gradient will point towards the gradient of the relevant samples. Of course, if all samples are irrelevant, the gradient becomes near zero, but this is not a problem, since if the target score is greater than all sample scores, there is nothing to be learned. Unfortunately, the sensitivity to large sample scores of TOP1 is still an issue as it is the consequence of the TOP1 pairwise loss and not the aggregation.

$$\frac{\partial L_{\text{top1-max}}}{\partial r_i} = -\sum_{j=1}^{N_S} s_j \sigma(r_j - r_i) \left(1 - \sigma(r_j - r_i)\right) \tag{10}$$

BPR-max: Going back to the probability interpretation of BPR, the goal is to maximize the probability of the target score being higher than the maximal sample score $r_{\max} = \max_j r_j$. This can be rewritten using conditional probabilities:

$$P(r_i > r_{\max}) = \sum_{j=1}^{N_S} P(r_i > r_j | r_j = r_{\max}) P(r_j = r_{\max}) \tag{11}$$

$P(r_i > r_j)$ and $P(r_j = r_{\max})$ is approximated by $\sigma(r_i - r_j)$ (as in the original BPR loss) and the softmax score s_j respectively. We then want to minimize the negative log-probability, which gives us the loss:

$$L_{\text{bpr-max}} = -\log \sum_{j=1}^{N_S} s_j \sigma(r_i - r_j) \tag{12}$$

The gradient of BPR-max (13) is the weighted average of individual BPR gradients, where the weights are $s_j \sigma(r_i - r_j)$. The relative importance of negative samples j and k is $\frac{\sigma(r_i - r_j)s_j}{\sigma(r_i - r_k)s_k} = \frac{e^{r_j} + e^{-r_i + r_j + r_k}}{e^{r_k} + e^{-r_i + r_j + r_k}}$, which behaves like softmax weights if $r_i \gg r_j + r_k$ or if both r_i and r_k are small. Otherwise it is a smoothed softmax. This means that while r_i is small, the weights are distributed more evenly, yet clear emphasis will be given to higher sample scores. As r_i becomes higher, the focus shifts quickly to the samples with high scores. This is an ideal behaviour.

$$\frac{\partial L_{\text{bpr-max}}}{\partial r_i} = -\frac{\sum_{j=1}^{N_S} s_j \sigma(r_i - r_j) \left(1 - \sigma(r_i - r_j)\right)}{\sum_{j=1}^{N_S} s_j \sigma(r_i - r_j)} \tag{13}$$

The gradient w.r.t. a negative sample – with both the BPR-max and TOP1-max – is proportional to the softmax score of the example, meaning that only the items, near the maximum will be updated. This is beneficial, because if the score of a negative sample is low, it doesn't need to be updated. If the score of a sample is much higher than that of the others it will be the only one updated and the gradient will coincide with the gradient of the pairwise loss between the target and the sample score. In a more balanced setting the gradient is between the aforementioned gradient and 0. For example the gradient of BPR-max w.r.t. a negative sample's score is as follows:

$$\frac{\partial L_{\text{bpr-max}}}{\partial r_k} = s_k - \frac{s_k \sigma^2(r_i - r_k)}{\sum_{j=1}^{N_S} s_j \sigma(r_i - r_j)} \tag{14}$$

Figure 2 depicts how the gradients of BPR and BPR-max behave given the rank of the target item[8]. The rank of the target is the number of negative scores exceeding it, e.g. rank 0 means that the target score is higher than all sample scores. Lower rank means

[7] $\sum s_j = 1$

[8] Similar trends can be observed when comparing TOP1 and TOP1-max, even though the shape of the curves is quite different from that of the BPR.

that there are fewer negative samples that are relevant. The figure depicts the median negative gradient w.r.t. the target score in two cases, measured on a dataset sample during the 1^{st} and 10^{th} epochs (i.e. beginning and end of the training): (left) no additional samples were used, only the other examples from a mini-batch of size 32; (middle & right) 2048 additional negative samples were added. The rightmost figure focuses on the first 200 ranks of the figure in the middle. The gradient is slightly higher for BPR when there are more relevant samples (i.e. high ranks). This is natural, since BPR-max focuses on samples closest to the maximum value and ignores other still relevant samples. This entails slightly slower learning for BPR-max when the target item is ranked at the end of the list, but the difference is not really significant. On the other hand, the gradient of BPR quickly vanishes as the number of relevant samples decrease (i.e. low ranks). The point of vanishing is relative to the total sample size. With small sample size, BPR's gradient starts vanishing around rank 5 (the BPR-max does not vanish until rank 0); meanwhile, with more samples, the BPR gradient is very low, even for rank 100-500 (again, the gradient BPR-max starts decreasing significantly later). This means that BPR can hardly push target scores up in the ranking after a certain point, which comes earlier as the number of sample size increases. BPR-max, on the other hand, behaves well and is able to improve the score all the way.

3.3.1 BPR-max with score regularization. Even though we showed that the heuristic TOP1 loss is sensitive to relevant samples with very high scores, it was found to be performing better than BPR in [7]. According to our observation, the same is true for the relation of TOP1-max and BPR-max. Part of the reasons lies in the rare occurrence of $r_j \gg r_i$ while $r_j \approx 0$ simultaneously. If only the first condition is met, the gradient w.r.t. r_i might vanish, but the regularizing part of TOP1 makes sure that r_j is moved towards zero, which might even make the update possible for r_i next time (e.g. if r_j was negative, moving it towards zero decreases the difference with r_i). The score regularization in TOP1 is very beneficial to the overall learning process, so even though the loss might not be theoretically optimal, it can achieve good results. GRU4Rec support two forms of regularization with every loss: dropout and ℓ_2 regularization of the model parameters. The regularization of TOP1 is used on the top of these. According to our experiments, the ℓ_2 regularization of model parameters decreases the model performance. Our assumption is that some of the model weights – such as the weight matrices for computing the update and reset gate – should not be regularized. Penalizing high output scores takes care of constraining the model, even without explicitly regularizing the weights.

Therefore we added score regularization to the BPR-max loss function as well. We tried several ways of score regularization. In the best performing one we conditioned the sample scores on independent, zero mean Gaussians with variance inversely proportional to the softmax score (15). This entails stronger regularization on scores closer to the maximum, which is ideal in our case.

$$P\left(r_i > r_{\max} | \{r_j\}_{j=1}^{N_S}\right) \prod_{j=1}^{N_S} P(r_j) = P\left(r_i > r_{\max} | \{r_j\}_{j=1}^{N_S}\right) \prod_{j=1}^{N_S} \mathcal{N}\left(0, \frac{c}{s_j}\right) \tag{15}$$

We minimize the negative log-probability and do continuous approximations as before, resulting in the final form of the BPR-max loss function (16). The regularization term is a simple, softmax weighted ℓ_2 regularization over the scores of the negative samples. λ is the regularization hyperparameter of the loss.

$$L_{\text{bpr-max}} = -\log \sum_{j=1}^{N_S} s_j \sigma(r_i - r_j) + \lambda \sum_{j=1}^{N_S} s_j r_j^2 \tag{16}$$

4 EXPERIMENTS

Experimental setup: We evaluated the proposed improvements – fixed cross-entropy loss, ranking-max loss functions & adding additional samples – on four dataset. RSC15 is based on the dataset of RecSys Challange 2015[9], which contains click and buy events from an online webshop. We only kept the click data. VIDEO and VIDXL are proprietary datasets containing watch events from an online video service. Finally, CLASS is a proprietary dataset containing item page view events from an online classified site. Datasets were subjugated to minor preprocessing then split into train and test sets so that a whole session either belongs to the train or to the test set. The split is based on the time of the first event of the sessions. The datsets and the split are exactly the same for RSC15 as in [7]; and for VIDXL and CLASS as in [8]. VIDEO is of the same source as in [7], but a slightly different subset. Table 1 overviews the main properties of the datasets.

Table 1: Properties of the datasets.

Data	Train set		Test set		Items
	Sessions	Events	Sessions	Events	
RSC15	7,966,257	31,637,239	15,324	71,222	37,483
VIDEO	2,144,930	10,214,429	29,804	153,157	262,050
VIDXL	17,419,964	69,312,698	216,725	921,202	712,824
CLASS	1,173,094	9,011,321	35,741	254,857	339,055

Evaluation is done under the next item prediction scenario, that is we iterate over test sessions and events therein. For each event, the algorithm guesses the item of the next event of that session. Since the size of the VIDXL test set is large, we compare the target item's score to that of the 50,000 most popular items during testing, similarly to [8]. While this evaluation for VIDXL overestimates the performance, the comparison of algorithms remain fair [2]. As recommender systems can only recommend a few items at once, the actual item a user might pick should be amongst the first few items of the list. Therefore, our primary evaluation metric is recall@20 that is the proportion of cases having the desired item amongst the top-20 items in all test cases. Recall does not consider the actual rank of the item as long as it is amongst the top-N. This models certain practical scenarios well where there is no highlighting of recommendations and the absolute order does not matter. Recall also usually correlates well with important online KPIs, such as click-through rate (CTR)[9, 14]. The second metric used in the experiments is MRR@20 (Mean Reciprocal Rank). That is the average of reciprocal ranks of the desired items. The reciprocal rank is set

[9]http://2015.recsyschallenge.com

to zero if the rank is above 20. MRR takes into account the rank of the item, which is important in cases where the order of recommendations matter (e.g. the lower ranked items are only visible after scrolling).

The natural baseline we use is the original GRU4Rec algorithm, upon which we aim to improve. We consider the results with the originally proposed TOP1 loss and *tanh* activation function on the output to be the baseline. The hidden layer has 100 units. We also indicate the performance of item-kNN, a natural baseline for next item prediction. Results for RSC15, VIDXL and CLASS are taken directly from corresponding papers [7, 8] and measured with the optimal hyperparameters in [7] for VIDEO. We do separate hyperparameter optimization on a separate validation set for the proposed improvements.

The methods are implemented under the Theano framework [1] in python. Experiments were run on various GPUs, training times were measured on an unloaded GeForce GTX 1080Ti GPU. Code is available publicly on GitHub[10] for reproducibility.

4.1 Using additional samples

The first set of experiments examines the effect of additional negative samples on recommendation accuracy. Experiments were performed on the CLASS and the VIDEO datasets. Since results are quite similar we excluded the VIDEO results to save some space. Figure 3 depicts the performance of the network with TOP1, cross-entropy, TOP1-max and BPR-max losses. Recommendation accuracy was measured with different number of additional samples, as well as in the case when all scores are computed and there is no sampling. As we discussed earlier, this latter scenario is a more theoretical one, because it is not scalable. As theory suggests (see Section 3), the TOP1 loss does not cope well with lots of samples. There is a slight increase in performance with a few extra samples, as the chance of having relevant samples increases; but performance quickly degrades as sample size grows, thus lots of irrelevant samples are included. On the other hand, all three of the other losses react well to adding more samples. The point of diminishing returns is around a few thousand of extra samples for cross-entropy. TOP1-max starts to slightly lose accuracy after that. BPR-max improves with more samples all the way, but slightly loses accuracy when all items are used.

Adding extra samples increases computational cost, yet due to easy parallelization on modern GPUs most of this cost is alleviated. Figure 4 shows the training times at different sample sizes. Please note the logarithmic scale. The actual training time depends on not just the dataset, but on the model parameters as well (especially mini-batch size) and how certain operators used for computing the loss are supported by the framework. The trend, however, is similar for all losses. For example, the full training of the network is around 6-7 minutes, which does not increase with even 512 extra samples. At the point of diminishing returns, i.e. at 2048 extra samples, training time is still around 7-8 minutes, which is only a slight increase. After that, training times grow quickly, due to exceeding the parallelization capabilities of the GPU we used. The trend is similar on the VIDEO dataset, with training times starting around 20 minutes, starting to increase at 2048 extra samples (to 24 minutes) and quickly above thereafter. This means that the proposed method can be used with zero to little additional cost in practice, unlike data augmentation methods. It is also clear that GRU4Rec can work just as well with a few thousands of negative examples as with the whole itemset, thus it can be kept scalable.

[10]https://github.com/hidasib/GRU4Rec

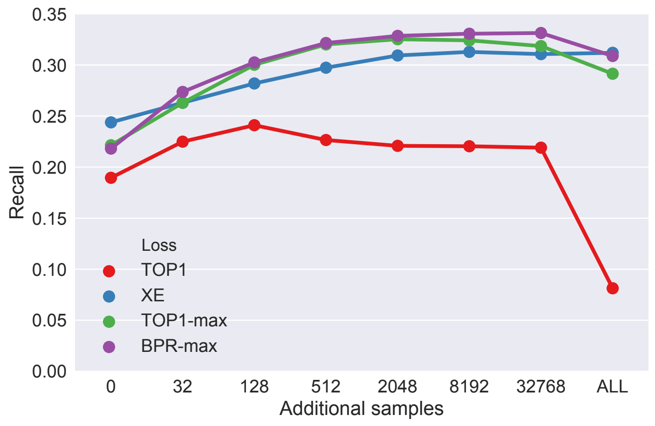

Figure 3: Recommendation accuracy with additional samples on the CLASS dataset. "ALL" means that there is no sampling, but scores are computed for all items in every step.

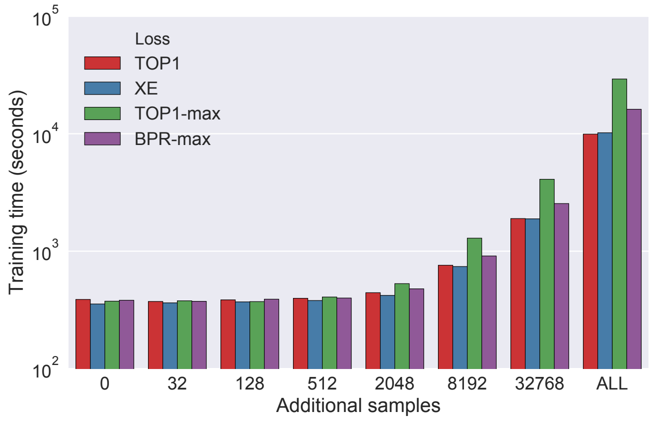

Figure 4: Training times with different sample sizes on the CLASS dataset.

In the next experiment we perform a parameter sensitivity analysis of the α parameter that controls the sampling. Figure 5 depicts the performance over different α values for the cross-entropy, TOP1-max and BPR-max losses. Cross-entropy favors higher α values with low sample sizes and low α values for large samples. This is inline with our discussion in Section 2: popular samples are useful when the sample size is very limited and at the beginning of the training, but might be exhausted quickly, thus switching to a more balanced sampling can be beneficial if we have the means to (e.g. large enough sample size). Also, the uniform sampling in this case is supplemented by the few popularity based samples of the mini-batch sampling. The ranking-max losses, on the other hand, seem to prefer the middle road with a slight preference towards higher

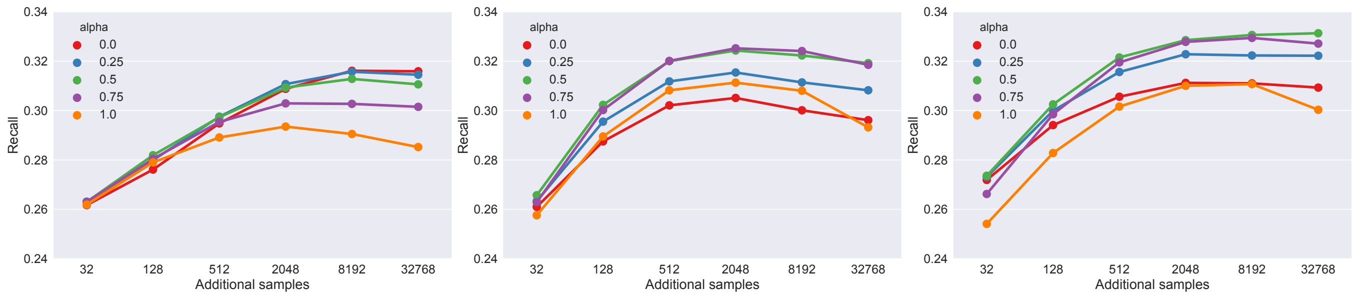

Figure 5: The effect of the alpha parameter on recommendation accuracy at different sample sizes on the CLASS dataset. Left: cross-entropy loss; Middle: TOP1-max loss; Right: BPR-max loss.

Table 2: Recommendation accuracy with additional samples and different loss functions compared to item-kNN and the original GRU4Rec. Improvements over item-kNN and the original GRU4Rec (with TOP1 loss) results are shown in parentheses. Best results are typeset bold.

Dataset	Item kNN	GRU4Rec original	GRU4Rec XE	GRU4Rec with additional samples TOP1	XE	TOP1-max	BPR-max
Recall@20							
RSC15	0.5065	0.5853	0.5781	0.6117 (+20.77%, +4.51%)	0.7208 (+42.31%, +23.15%)	0.7086 (+39.91%, +21.07%)	**0.7211 (+42.37%, +23.20%)**
VIDEO	0.5201	0.5051	0.5060	0.5325 (+2.40%, +5.43%)	0.6400 (+23.06%, +26.72%)	0.6421 (+23.46%, +27.12%)	**0.6517 (+25.31%, +29.03%)**
VIDXL	0.6263	0.6831	0.7046	0.6723 (+7.35%, -1.58%)	0.8028 (+28.19%, +17.53%)	0.7935 (+26.70%, +16.16%)	**0.8058 (+28.66%, +17.97%)**
CLASS	0.2201	0.2478	0.2545	0.2342 (+6.41%, -5.50%)	0.3137 (+42.54%, +26.61%)	0.3252 (+47.75%, +31.22%)	**0.3337 (+51.61%, +34.66%)**
MRR@20							
RSC15	0.2048	0.2305	0.2375	0.2367 (+15.61%, +2.69%)	0.3137 (+53.16%, +36.08%)	0.3045 (+48.70%, +32.08%)	**0.3170 (+54.78%, +37.52%)**
VIDEO	0.2257	0.2359	0.2609	0.2295 (+1.69%, -2.73%)	0.3079 (+36.42%, +30.52%)	0.2950 (+30.72%, +25.05%)	**0.3089 (+36.87%, +30.95%)**
VIDXL	0.3740	0.3847	0.4343	0.3608 (-3.53%, -6.21%)	0.5031 (+34.52%, +30.78%)	0.4939 (+32.05%, +28.39%)	**0.5066 (+35.45%, +31.68%)**
CLASS	0.0799	0.0949	0.0995	0.0870 (+8.83%, -8.36%)	0.1167 (+46.08%, +22.99%)	0.1198 (+49.93%, +26.25%)	**0.1202 (+50.40%, +26.63%)**

values, while the extremes perform the worst. We assume that this is mostly due to (a) being based on pairwise losses, where popular samples are usually desired; (b) and the score regularization: with popularity based sampling the scores of the most popular items would be decreased beyond what is desirable.

4.2 Loss-functions

We measure the performance gain of the proposed improvements over the baselines. The big accuracy improvement comes from the combination of additional samples and the loss functions (fixed cross-entropy, TOP1-max and BPR-max). Table 2 showcases our most important results. Besides the original version of GRU4Rec and the item-kNN, we included results with cross-entropy (XE) loss without additional sampling to confirm that the fixed cross-entropy loss still performs just slightly better than TOP1. The increase with sampling and the proper loss function is stunning as the best results exceed the accuracy of the original GRU4Rec by 18 − 37.5% and that of item-kNN by up to 55%. BPR-max performs similarly (2 of 4) or better (2 of 4; +2 − 6% improvement) than cross-entropy when extra samples are used for both method.

On RSC15, [19] reported ∼ 0.685 and ∼ 0.29 in recall@20 and MRR@20 respectively[11] using data augmentation. Unlike our solutions, data augmentation greatly increases training times. Data augmentation and our improvements are not mutually exclusive, thus it is possible that by combining the two methods, even better results can be achieved. A recent paper [4] proposes the Bayesian version of GRU4Rec and reports ∼ 0.61 and ∼ 0.25 in recall@20 and MRR@20 when using 100 units[12]. Therefore our GRU4Rec version is the current best performer so far.

4.3 Unified item representations

Previous experiments did not find any benefits of using an embedding layer before the GRU layers. The role of the embedding layer is to translate item IDs into the latent representation space. In the recommender systems terminology, item embeddings correspond to "item feature vectors". The network has another "item feature matrix" though in the form of the output weight matrix. By unifying the representations, i.e. sharing the weight matrix between the embedding layer and the output layer, we learn better

[11]Read from figure4. Unfortunately, the results in table1 are for networks trained on various subsets of the training set.
[12]Based on figure1. Their best results (0.6507 and 0.3527) are achieved using 1500 units, which is highly impractical. Even though, our version still performs better w.r.t. recall when compared to this much bigger network.

Table 3: Results with unified embeddings. Relative improvement over the same network without unified item representation is shown in the parentheses.

Dataset	Recall@20	MRR@20
RSC15	0.7244 (+0.45%)	0.3125 (-1.42%)
VIDEO	0.6598 (+1.25%)	0.3066 (-0.74%)
VIDXL	0.8110 (+0.65%)	0.5099 (+0.66%)
CLASS	0.3962 (+18.74%)	0.1555 (+29.44%)

item representations quicker. Preliminary experiments (Table 3) show additional slight improvements in recall@20 and slight decrease in MRR@20 for most of the datasets. However, for the CLASS dataset both recall and MRR are increased significantly when unified embeddings are used (+18.74% and +29.44% in recall and MRR respectively, compared to the model trained without embeddings). Unified embeddings have the additional benefit of reducing the overall memory footprint and the model size by a factor of ∼ 4.

4.4 Online tests

With the improvements proposed in this paper, it became technically feasible to also evaluate GRU4Rec in a large scale online A/B test on an online video portal. Recommendations are shown on each video page and are available as soon as the page loads. The site has an autoplay function, similar to the one on Youtube. If the video is accessed from the recommendation queue by either the user clicking on one of the recommended items or by autoplay, no new recommendation is calculated, thus the user can have multiple interactions with one recommended set of items. The user can also access videos directly, and then a new set of video recommendations is generated for her.

Experimental setup: GRU4Rec was compared to a fine tuned complex recommendation logic for a duration of 2.5 months. Users are divided into three groups. The first group is served by the baseline logic. The second group is served by GRU4Rec in "next best" mode, meaning that the algorithm shows recommendations that are very likely to be the next in the user's session. The third group is served by GRU4Rec in "sequence mode" where GRU4Rec generates a sequence of items based on the session so far. The sequence is generated greedily, i.e. the algorithm assumes that its next best guesses were correct so far. Bots and power users are filtered from the measurement as they can skew the results. Bots are filtered based on the user agent. Unidentified bots and power users (e.g. users leaving videos playing for a very long time) and users with unrealistic behaviour patterns are also filtered on a daily basis. The proportion of non-bot users affected by filtering is very low (∼ 0.1%).

KPIs: We used different KPIs to evaluate performance. Each KPI is relative to the number of recommendation requests. *Watch time* is the total number of seconds of videos watched by the group. *Video plays* is the number of videos watched for at least a certain amount of time by the group. *Clicks* is the number of times the group clicked on the recommendations.

Results: Figure 6 shows the relative performance gain of GRU4Rec over the complex logic. The error bars represent the confidence

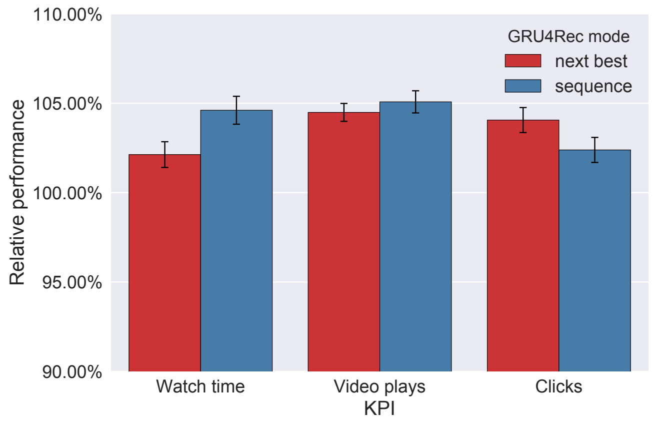

Figure 6: Performance of GRU4Rec relative to the baseline in the online A/B test.

interval at $p = 0.05$. GRU4Rec outperforms the baseline in both prediction modes. The improvement is approximately 5% in watch time, 5% in video plays and 4% in clicks. Sequential predictions perform better than next best guess based predictions w.r.t. watch time and number of video plays, but not in clicks. This is due to sequential predictions being more appropriate for the autoplay functionality, thus resulting in less clicks from the users. On the other hand, while next best guess based predictions are relevant, they are also more diverse and it is more likely for the user to skip videos in the recommendation queue. Sequential predictions are more appropriate for video series and other closely knit videos. We also noticed that as the two prediction modes run simultaneously and learn from each other's recommendations through the feedback loop, the differences in watch time and video plays slowly start to disappear.

5 CONCLUSION

In this paper we focused on session-based recommendations, which is becoming one of the most important recommendation scenarios in practice for many domains, including video, music and general e-commerce and even for novel applications such as energy saving recommendations. We introduced a new class of loss functions that together with an improved sampling strategy have provided impressive top-k gains for RNNs for session-based recommendations. We believe that these new losses could be more generally applicable and along with the corresponding sampling strategies also provide top-k gains for different recommendations settings and algorithms such as e.g. matrix factorization or autoencoders. It is also conceivable that these techniques could also provide similar benefits in the area of Natural Language Processing a domain that shares significant similarities to the recommendation domain in terms of machine learning (e.g. ranking, retrieval) and data structure (e.g. sparse large input and output space). We also showed that by using these improvements along with other minor ones – such as unified embeddings – GRU4Rec can outperform previous solutions in a live online setting according to important business KPIs.

ACKNOWLEDGMENTS

The authors would like to thank István Pilászy for his valuable support in executing the online A/B tests.

REFERENCES

[1] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016).

[2] Alejandro Bellogin, Pablo Castells, and Ivan Cantador. 2011. Precision-oriented Evaluation of Recommender Systems: An Algorithmic Comparison. In *RecSys'11: 5th ACM Conf. on Recommender Systems*. 333–336. https://doi.org/10.1145/2043932.2043996

[3] Chris J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview.* Technical Report. https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/

[4] Sotirios Chatzis, Panayiotis Christodoulou, and Andreas S Andreou. 2017. Recurrent Latent Variable Networks for Session-Based Recommendation. *arXiv preprint arXiv:1706.04026* (2017).

[5] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *SSST-8: 8th Workshop on Syntax, Semantics and Structure in Statistical Translation*. 103–111.

[6] Robin Devooght and Hugues Bersini. 2016. Collaborative filtering with recurrent neural networks. *arXiv preprint arXiv:1608.07400* (2016).

[7] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. 2016. Session-based Recommendations with Recurrent Neural Networks. *International Conference on Learning Representations* (2016). http://arxiv.org/abs/1511.06939

[8] Balázs Hidasi, Massimo Quadrana, Alexandros Karatzoglou, and Domonkos Tikk. 2016. Parallel Recurrent Neural Network Architectures for Feature-rich Session-based Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys '16)*. ACM, New York, NY, USA, 241–248. https://doi.org/10.1145/2959100.2959167

[9] B. Hidasi and D. Tikk. 2012. Fast ALS-based tensor factorization for context-aware recommendation from implicit feedback. In *ECML-PKDD'12, Part II*. Number 7524 in LNCS. Springer, 67–82.

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[11] Shihao Ji, SVN Vishwanathan, Nadathur Satish, Michael J Anderson, and Pradeep Dubey. 2016. Blackout: Speeding up recurrent neural network language models with very large vocabularies. *ICLR* (2016).

[12] Yehuda Koren and Joe Sill. 2011. OrdRec: An Ordinal Model for Predicting Personalized Item Rating Distributions. In *Proceedings of the Fifth ACM Conference on Recommender Systems (RecSys '11)*. ACM, New York, NY, USA, 117–124.

[13] G. Linden, B. Smith, and J. York. 2003. Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE* 7, 1 (2003), 76–80.

[14] Qiwen Liu, Tianjian Chen, Jing Cai, and Dianhai Yu. 2012. Enlister: Baidu's Recommender System for the Biggest Chinese Q&A Website. In *RecSys-12: Proc. of the 6th ACM Conf. on Recommender Systems*. 285–288.

[15] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI '09)*. 452–461.

[16] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *UAI'09: 25th Conf. on Uncertainty in Artificial Intelligence*. 452–461.

[17] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *WWW:01: 10th Int. Conf. on World Wide Web*. 285–295.

[18] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Nuria Oliver, and Alan Hanjalic. 2012. CLiMF: Learning to Maximize Reciprocal Rank with Collaborative Less-is-more Filtering. In *Proceedings of the Sixth ACM Conference on Recommender Systems (RecSys '12)*. 139–146. https://doi.org/10.1145/2365952.2365981

[19] Yong Kiam Tan, Xinxing Xu, and Yong Liu. 2016. Improved Recurrent Neural Networks for Session-based Recommendations. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems (DLRS 2016)*. ACM, New York, NY, USA, 17–22. https://doi.org/10.1145/2988450.2988452

[20] Markus Weimer, Alexandros Karatzoglou, Quoc Viet Le, and Alex Smola. 2007. COFIRANK Maximum Margin Matrix Factorization for Collaborative Ranking. In *Proceedings of the 20th International Conference on Neural Information Processing Systems (NIPS'07)*. 1593–1600.

[21] Chao-Yuan Wu, Amr Ahmed, Alex Beutel, Alexander J. Smola, and How Jing. 2017. Recurrent Recommender Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. ACM, New York, NY, USA, 495–503.