Compressed Sensing Defence

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

Balazs Hidasi and Alexandros Karatzoglou

Bastien Billiot and Joël Garde

29 Mars 2021

Table of Contents

Table of Contents

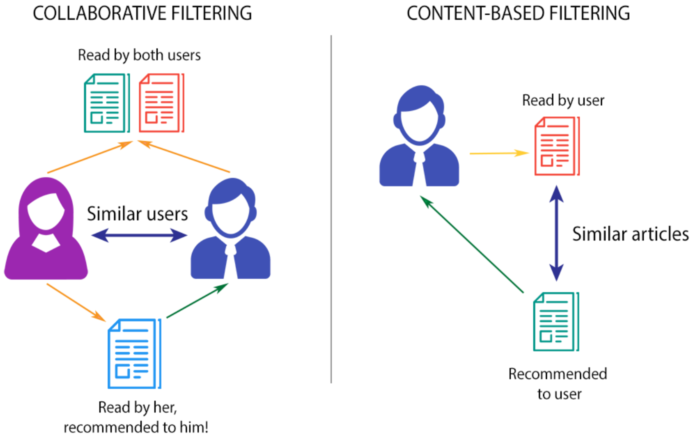
Context : Recommendation system

- Digitization and boom of online services and products providers leading to an increase importance of customer retention and satisfaction
- Thus recommendation system has become an ever increasing tool to that goal

Definition

Recommenders systems are built to predict what users might like, especially when there are lots of choices available

Two major types of Recommenders Systems



Compressed Sensing Approach : Matrix completion and Recommender Systems

- Collaborative-filtering recommendation system : matrix completion with homogeneous group → Netflix Prize
- Problem Formulation

  Rebuild $A^* \in \mathbb{R}^{u \times v}$ with the available data of the matrix $Y_i = (< A^*, X_i >)_{i=1,\ldots,m}$, knowing that A^* is low-rank

- Rank-minimization procedure (NP hard) :

$$\min_A (rg(A) :< A, X_i >=< A^*, X_i >, i = 1, ...m)$$

- Nuclear Norm Minimization problem :

$$\min_A (||A||_{S_1} :< A, X_i >= Y_i, i = 1..., m)$$

- Semi Definitite Programming : projected or proximal gradient descent

Session Based Recommandation Systems and Paper Introduction

- Emergence of unknown users $\rightarrow$ Session-Based Recommender Systems
- GRU4REC : Session based reommender system proposed by two authors in a previous 2016 paper
- Studied Paper : Sampling and loss improvements

Table of Contents

Problem formulation

- Ongoing session $s \in S$, sequence of events $a_i \in V$, $s = [a_1, a_2, \ldots, a_t]$
- Provide recommendation a_{t+1}
- Ranking over all V items (probability of being chosen) : k highest recommended items
- Grundtruth item among k highest and as high as possible
- Challenges : different sequences length, high cardinality, ranking evaluation metrics

Model Architecture

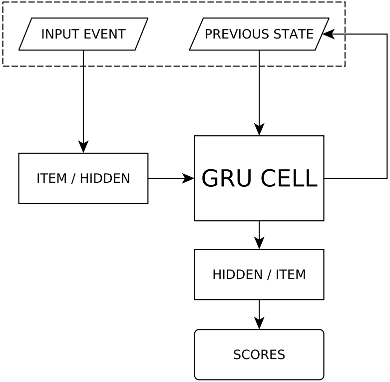
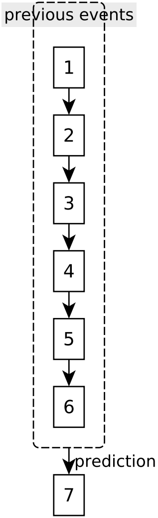


Figure: General GRU4REC architecture.

Training procedure

Next event prediction

Given $[a_0, \ldots, a_i]$ predict a_{i+1}



Back-Propagation-Through-Time

Mini-batching.

Use the parallel computing facilities of GPU to accelerate optimization.

***BPTT*(1, 1).**

Process size 1, step size 1.

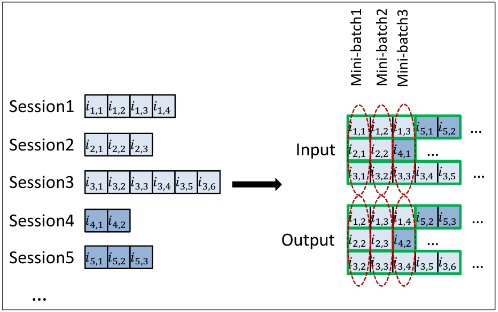


Figure: Mini-batch, taken from the paper.

Table of Contents

Why do we sample?

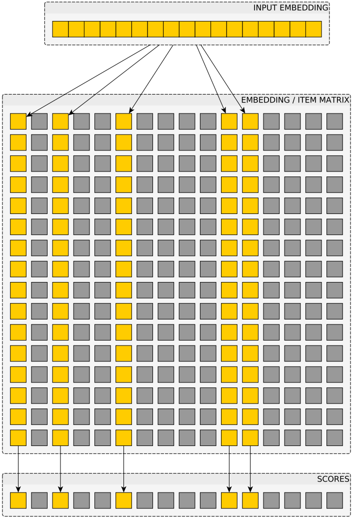
- $|V| \sim 10^5$

Relevant samples

High-scoring samples.

Power-raized unigram distribution

$Q(i) = supp(i)^{\alpha}$



Scores for mini-batch negative sampling

		scores				
session	item	a_{i+1}	b_{j+1}	c_{k+1}	...	e_{m+1}
a	a_i	1	0	0	...	0
b	b_j	0	1	0	...	0
c	c_k	0	0	1	...	0
...	...	0	0	0	...	0
e	e_m	0	0	0	...	1

Scores for full negative sampling

		scores									
session	item	a_{i+1}	b_{j+1}	c_{k+1}	...	e_{m+1}	n_1	n_2	...	...	n_{N_A}
a	a_i	1	0	0	...	0	0	0	0	0	0
b	b_j	0	1	0	...	0	0	0	0	0	0
c	c_k	0	0	1	...	0	0	0	0	0	0
...	...	0	0	0	...	0	0	0	0	0	0
e	e_m	0	0	0	...	1	0	0	0	0	0

Previous losses

- **Cross-entropy**:

$$CE(\hat{y}, y) = -\sum_{k=1}^{N} \hat{y}_k log(y_k)$$

- **Bayesian Personalized Ranking (BPR)**:

$$L_{BPR} = -\frac{1}{N_s} \sum_{j=1}^{N_s} log[\sigma(r_i - r_j)]$$

- **TOP1** :

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

Vanishing Gradient

$$\frac{\partial L_{top1}}{\partial r_i} = -\frac{1}{N_S}\sum_{j=1}^{N_S} \sigma'(r_j - r_i) \text{ and } \frac{\partial L_{BPR}}{\partial r_i} = -\frac{1}{N_S}\sum_{j=1}^{N_S}(1 - \sigma(r_i - r_j))$$

- Irrelevant negative samples : vanishing gradient
- Number of irrelevant samples increases faster than relevant ones
- Gradient vanishes as the number of samples increases

Proposed losses

Ranking-Max loss family functions

$$L_{pairwise-max}(r_i, \{r_j\}_{j=1}^{N_S}) = L_{pairwise}(r_i, \max_{j=1,\ldots,N_s}(r_j))$$

- $L_{TOP1-max} = \sum_{j=1}^{N_s} s_j[\sigma(r_j - r_i) + \sigma(r_j^2)]$
- $L_{BPR-max} = -log \sum_{j=1}^{N_s} s_j \sigma(r_i - r_j)$

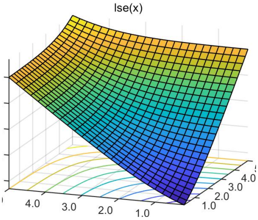
Table of Contents

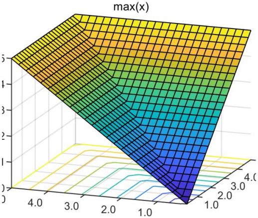
Importance sampling

- Sampled loss $\neq$ full loss
- $\nabla p\left(a_t|a_{<t},x\right) = \nabla\mathcal{E}(a_t) - \mathbb{E}_{p\left(a_t|a_{<t},x\right)}[\mathcal{E}(a)]$[1]
- $\mathbb{E}_{p\left(a_t|a_{<t},x\right)}[\mathcal{E}(y)] \approx \sum_{j=1}^{N_s} w(a_j)\mathcal{E}(a)$
- $w(a_j) = softmax_{[1,N_s]}(E(a_j) - log\ Q(a_j))$

[1] Sébastien Jean et al. *On Using Very Large Target Vocabulary for Neural Machine Translation*. 2015. arXiv: 1412.2007 [cs.CL].

LogSumExp





2

- $lse((x)_i) = log \sum_i e^{x_i}$
- $CE = -r_i + lse(r_j)$

[2] *What Is the Log-Sum-Exp Function?*
https://nhigham.com/2021/01/05/what-is-the-log-sum-exp-function/.

Connections to other papers

- Word2Vec[3]: non-recurrent
- Blackout:[4] Importance sampling + Q_α
- re-weighting:[5] according to rank.

[3] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[4] Shihao Ji et al. *BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies*. 2016. arXiv: 1511.06909 [cs.LG].

[5] Wei Chen et al. "Ranking Measures and Loss Functions in Learning to Rank.". In: Jan. 2009, pp. 315–323.

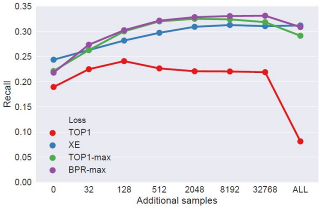
Table of Contents

Presentation : Datasets and Metrics

- 4 datasets : RSC15, VIDEO and VIDXL, CLASS
- Metrics : Recall@20 and MRR@20
- $REC_{20}(y_h) = \mathbb{I}_{rank(y_h)>20}$
- $MRR_{20}(y_h) = \frac{1}{rank(y_h)}\mathbb{I}_{rank(y_h)>20}$

Presentation : Results

- Additional samples :



- Top performance : combination of additional samples and new loss *BPR-max* +25% to +55% vs. baseline kNN and +18% to +37.5% vs. GRU4REC
- Mitigated results for *TOP1-max* vs. XE (with or without additional samples)

Batching

```
def __iter__(self):
  r = np.arange(len(self.offsets)-1) # r is a permutation array. we acces
  if self.random:
      np.random.shuffle(r)
  n_done = 0 # how many sessions have been fully seen.
  n_new = self.bs # how many new sessions to fetch to form the current ba
  reset = np.ones(self.bs, dtype=bool) # true if the current element if t
  CUR = np.zeros(self.bs, dtype=np.intp) # pointers to the current events
  END = np.zeros_like(CUR) # pointers to the last events of the current s
  ft = np.arange(2, dtype=np.intp).reshape((-1,1)) # [0,1] array, used as

  while n_new + n_done + 1 < r.shape[0]: # as long as we can have full ba
    CUR[reset] = self.offsets[r[n_done:n_done+n_new]] # where we finished
    END[reset] = self.offsets[r[n_done:n_done+n_new] + 1] - 1 #offsets[x+
    batch = CUR + ft # form a batch of [features, targets] by looking for
    b = self.data[batch] # form a batch of [features, targets] by looking
    reset = batch[1] == END # if the target is the last event of the sess
    n_new = reset.sum() # compute how many to get for the new one
    n_done += n_new
    CUR += 1 # go to next element in all current sessions.
    yield b, reset
```

Batching

```
class RandomDataset(torch.utils.data.IterableDataset):
  def __init__(self,  data:SessionData, n_neg=1024, alpha=1.0):
    self.n_neg = n_neg
    self.alpha = alpha
    self.p = np.power(data.freq, alpha)# alpha-power raised support
    self.squash = 8 # distortion / speed trade-off by dividing the support.
    self.p = np.ceil(self.p / self.squash).astype(np.int) # converting to i
    self.expanded = np.repeat(np.arange(len(self.p), dtype=np.long), self.p
  def __iter__(self):
      while True:
        yield np.random.choice(self.expanded,size=self.n_neg) # fast sampli
```

Model

```
def __init__(self, input_size, hidden_size):
    """
    parameters:
    ------
    input_size: number of words. (events)
    hidden_size: hidden dimention.
    """
    super().__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size

    # minified GRU cell with only 4 weights matrices
    self.Wx = nn.Parameter(torch.empty((input_size, hidden_size)))
    self.Wr = nn.Parameter(torch.empty((hidden_size, hidden_size)))
    self.Wh = nn.Parameter(torch.empty((hidden_size, hidden_size)))
    self.Wy = self.Wx # tied weights: the output projection matrix is the t

    self.bh = nn.Parameter(torch.empty((hidden_size,)))
    self.by = nn.Parameter(torch.empty((input_size,)))

    nn.init.xavier_normal_(self.Wx)
    nn.init.xavier_normal_(self.Wr)
    nn.init.orthogonal_(self.Wh) #orthogonal initialisation for the recurre
    nn.init.uniform_(self.bh)
    nn.init.uniform_(self.by)
```

Model

```
def forward(self, inputs, carry, targets=None):
    # gru
    s = self.Wx[inputs]
    v = s + self.bh
    r = torch.sigmoid(torch.addmm(v, carry, self.Wr))
    h = torch.tanh(torch.addmm(v, r * carry, self.Wh))
    h = (1-r) * carry + r * h
    if targets is not None: #partial score computation for using negative s
        y = torch.addmm(self.by[targets], h, self.Wy[targets].T)
    else: # full score computation for testing.
        y = torch.addmm(self.by, h, self.Wy.T)
    return h, y
```

general remarks

- focus on speed
- unusual GRU
- `y = y - self.logq * T.log(self.P0)`

Table of Contents

Opening

Comparison with other models:

- Always a top contender with overall decent results on every dataset
- Not better than baseline KNN models when much more complex

Evaluation of Session based recommender systems:

- Difficult to build a state-of-the-art model for all business applications (SR for music vs KNNs for e-commerce)
- No reference method (A/B, fixed dataset), metric or dataset

To go further: Cross-domain recommendation systems

Conclusion

- Improvements of a performing model : GRU4REC with additional samples and new losses
- Top performer in various applications but still quiet expensive computationnaly for the improvements over baseline models
- Ranking-max loss : unsufficient explanations and first parts do not introduce new concepts
- Evaluation methods, metrics and datasets and other modelling (cross domain)