[Team No: 31 - Matrix]

Collaborative Recurrent Neural Networks for Dynamic Recommender Systems

---

-

Abstract :

We are trying to construct a recommender system which can capture the contextual aspects of user behaviour revealed by temporal and recurrent patterns. These temporal and recurrent patterns are collected from the activity logs of the user in a non - intrusive way.

Preliminaries:

U = Set of users with cardinality |U|,

I = Set of items with cardinality |I|,

Each users activity history =$[x_1^u, x_2^u, \ldots x_{Tu}^u]$ where x_1 is item accessed at time stamp t_1.

(**Note** that this is not the absolute time stamp but just ordering of events.)

We defined the sequence $x_{<t}$ of items which is accessed before the timestamp t and $x_{>=t}$ is the sequence of items after timestamp t.

Our model report the quantization of likelihood at t = k, k+1, k+2, . . . given $x_{<t}$. So we define our model on the basis of probability as :

$$\mathbf{P(x^u / \theta) = \prod_1^{Tu} P(x^u_t / x^u_{<t}, \theta)}$$

Where θ is the model parameter.

The evaluation metric:

We reported the Average negative log likelihood which is computed for a held-out part of the sequence, For every user, we split x^u into $x^u_{<ru}$ and $x^u_{>=ru}$ at $1 < r_u \leq T_u$ and define the error as :

$$\mathbf{E(x^1_{>=r1}, \ldots, x^u_{>=ru} / \theta) = -1/U(\sum_{u \varepsilon U} (\log P(x^u_{>=ru} / \theta)) / (T_u - r_u + 1))}$$

Other Models:

i) n-gram Model:

An *n*-gram model is a type of probabilistic language model for predicting the next item when given n previous items.

When n=1(Unigram), then

$$\mathbf{C_i} = \sum_{u \varepsilon U} \sum_{1}^{Tu} \mathbf{1(x_t = i)}$$

And we defined unigram model as ,

$$\mathbf{P(x^u_t = i)} = \frac{ci+\varepsilon}{\sum_{k \varepsilon I}(ck + \varepsilon)}$$

And n=2 (Bigram),then

$$\mathbf{B_{ij}} = \sum_{u \varepsilon U} \sum_{2}^{Tu} \mathbf{1(x_t = i \, , \, x_{t-1} = j)}$$

And we defined bigram model as

$$\mathbf{P(x_t^u = i \, / \, x_{t-1}^u = j)} = \frac{bij+\varepsilon}{\sum_{k \varepsilon I} (bkj+\varepsilon)}$$

ii) Matrix Factorization:

Matrix factorization can be used to discover latent features underlying the interactions between two different kinds of entities.

Methods based on matrix factorization have become a widely used tool for collaborative filtering due to their early success for recommender system(ref: **(Koren et al., 2009)).**The intuition behind using matrix factorization to solve this problem is that there should be some latent features that determine how a user rates an item.

For example, two users would give high ratings to a certain movie if they both like the actors/actresses of the movie, or if the movie is an action movie, which is a genre preferred by both users. Hence, if we can discover these latent features, we should be able to predict a rating with respect to a certain user and

a certain item, because the features associated with the user should match with the features associated with the item.

we have a set U of users, and a set D of items. Let $\mathbf{R}$ of size $|U| \times |D|$ be the matrix that contains all the ratings that the users have assigned to the items. Also, we assume that we would like to discover K latent features. Our task, then, is to find two matrices matrices $\mathbf{P}$ (a $|U| \times K$ matrix) and $\mathbf{Q}$ (a $|D| \times K$ matrix) such that their product approximates $\mathbf{R}$:

$$\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^T = \hat{\mathbf{R}}$$

In this way, each row of $\mathbf{P}$ would represent the strength of the associations between a user and the features. Similarly, each row of $\mathbf{Q}$ would represent the strength of the associations between an item and the features. To get the prediction of a rating of an item d_j by u_i, we can calculate the dot product of the two vectors corresponding to u_i and d_j:

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj}$$

Now, we have to find a way to obtain $\mathbf{P}$ and $\mathbf{Q}$. One way to approach this problem is the first initialize the two matrices with some values, calculate how `different' their product is to $\mathbf{M}$, and then try to minimize this difference iteratively. Such a method is called gradient descent, aiming at finding a local minimum of the difference.

The difference here, usually called the error between the estimated rating and the real rating, can be calculated by the following equation for each user-item pair:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2$$

Here we consider the squared error because the estimated rating can be either higher or lower than the real rating.

To minimize the error, we have to know in which direction we have to modify the values of p_{ik} and q_{kj}. In other words, we need to know the gradient at the current values, and therefore we differentiate the above equation with respect to these two variables separately:

$$\frac{\partial}{\partial p_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(q_{kj}) = -2e_{ij}q_{kj}$$
$$\frac{\partial}{\partial q_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij}p_{ik}$$

Having obtained the gradient, we can now formulate the update rules for both p_{ik} and q_{kj}:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij}q_{kj}$$
$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij}p_{ik}$$

Here, α is a constant whose value determines the rate of approaching the minimum. Usually we will choose a small value for α, say 0.0002. This is because if we make too large a step towards the minimum we may run into the risk of missing the minimum and end up oscillating around the minimum.

$$E = \sum_{(u_i,d_j,r_{ij})\in T} e_{ij} = \sum_{(u_i,d_j,r_{ij})\in T} (r_{ij} - \sum_{k=1}^{K} p_{ik}q_{kj})^2$$

Regularization:

The above algorithm is a very basic algorithm for factoring a matrix. There are a lot of methods to make things look more complicated. A common extension to

this basic algorithm is to introduce regularization to avoid overfitting. This is done by adding a parameter β and modify the squared error as follows:

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2)$$

In other words, the new parameter β is used to control the magnitudes of the user-feature and item-feature vectors such that P and Q would give a good approximation of R without having to contain large numbers. In practice, β is set to some values in the range of 0.02. The new update rules for this squared error can be obtained by a procedure similar to the one described above. The new update rules are as follows.

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + \alpha(2 e_{ij} q_{kj} - \beta p_{ik})$$
$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + \alpha(2 e_{ij} p_{ik} - \beta q_{kj})$$

- RNN (LSTM & GRU) using tensorflow (ref: http://www.wildml.com/)
- Tensorflow tutorial: (ref: https://www.tensorflow.org/tutorials/)
- MatplotLib for plotting of graphs (ref: https://matplotlib.org/users/pyplot_tutorial.html)

CRNN with collaborative features with input from user-item activity matrix.

Steps:

1. Parsing the activity log data
2. Removing useless columns
3. Capping the data so that training can be done in feasible time and available systems (processed data is attached in github).
4. Training on RNN model is done as per Algorithm 2 and Algorithm 1 discussed in paper.

Algorithm 1 `processSequence()`

Input: Sequence $\boldsymbol{x}$, $\boldsymbol{\theta} = \{\boldsymbol{W}_{in}, \boldsymbol{W}_h, \boldsymbol{W}_{out}\}$, batch size B
Output: Updated parameters $\boldsymbol{\theta}$

$\mathcal{B} \leftarrow$ Split $\boldsymbol{x}$ into sub-sequences of length B
$\boldsymbol{h}_0 \leftarrow \epsilon\mathbf{1}$
for $\boldsymbol{b} \in \mathcal{B}$ **do**
  $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_B \leftarrow \text{RNN}(\boldsymbol{b}, \boldsymbol{h}_0; \boldsymbol{\theta})$ (Forward pass using Eq. 15)
  $\nabla_{\boldsymbol{\theta}} \log E \leftarrow \text{BPTT}(\boldsymbol{b}, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_B; \boldsymbol{\theta})$ (Backward pass, see Appendix A)
  $\boldsymbol{\theta} \leftarrow \text{LearningRule}(\nabla_{\boldsymbol{\theta}}, \boldsymbol{\theta})$
  $\boldsymbol{h}_0 = \boldsymbol{h}_B$
end for

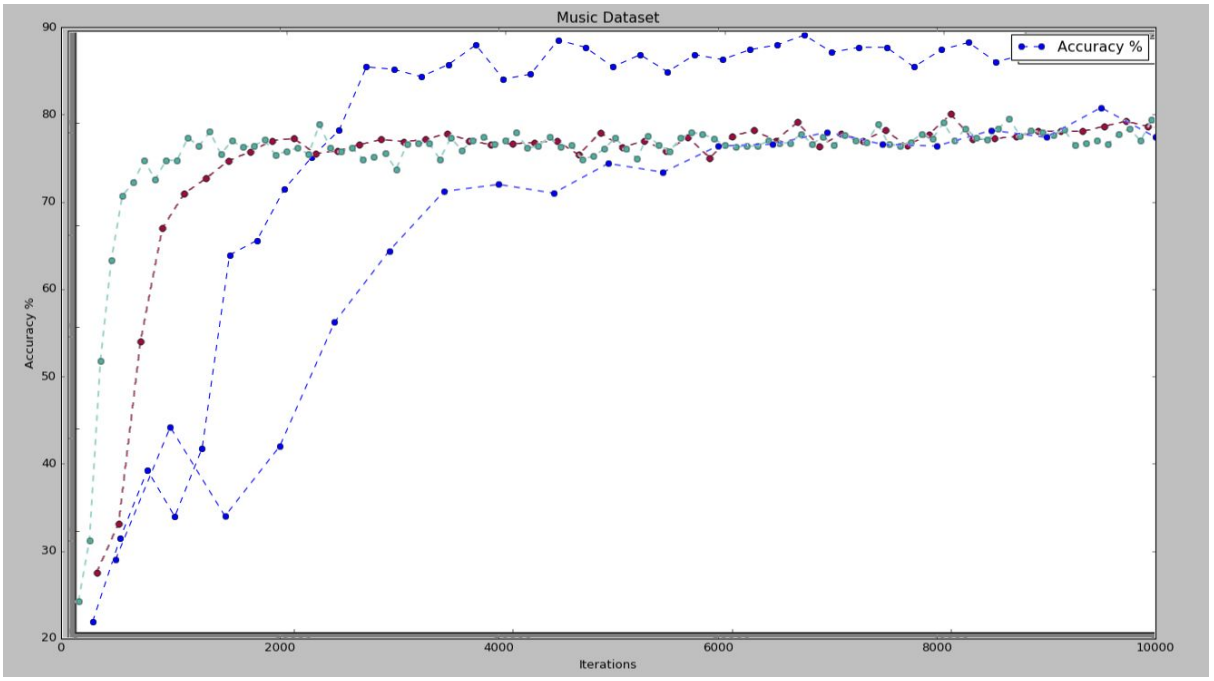
Algorithm 2 Collaborative RNN Training

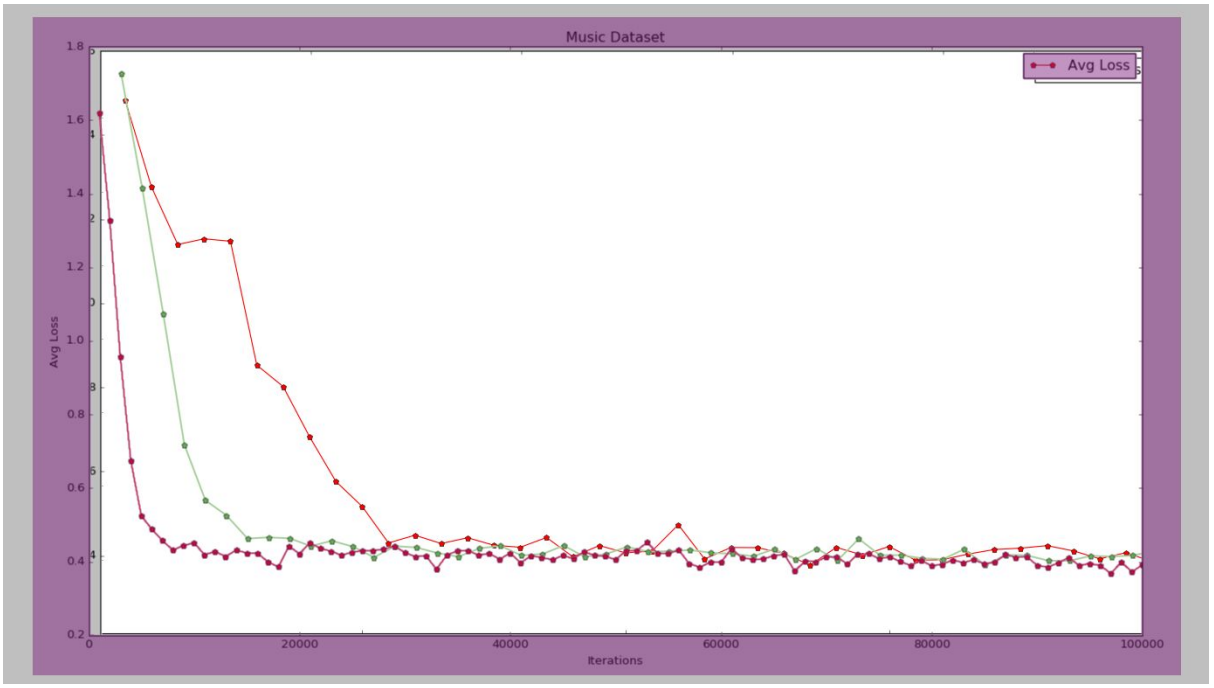
Input: Sequences $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(U)}$, Batch size B

Init $\boldsymbol{\theta} = \left\{\boldsymbol{W}_{in}, \boldsymbol{W}_{out}, \boldsymbol{W}_h^{(1)}, \ldots, \boldsymbol{W}_h^{(U)}\right\}$
repeat
  $\mathcal{R} = randperm(U)$ (process users in random order)
  for $u \in \mathcal{R}$ **do**
    $\boldsymbol{\theta}_u \leftarrow \left\{\boldsymbol{W}_{in}, \boldsymbol{W}_{out}, \boldsymbol{W}_h^{(u)}\right\}$
    $\boldsymbol{W}_{in}, \boldsymbol{W}_{out}, \boldsymbol{W}_h^{(u)} \leftarrow$ `processSequence` $\left(\boldsymbol{x}^{(u)}, \boldsymbol{\theta}_u, B\right)$ (Algorithm 1)
  end for
until Early Stopping Criterion holds

Execution & Output:

1. Graph for Collaborative RNN (code attached in github) using processed & suppressed data set of user activity for music recommender system, **finding accuracy and loss** after various epoch training.





2. **Recommended music track from trained model.**

```
Epoch Size= 10000, Avg_Loss= 0.395665, Avg_Accuracy= 80.00%
['14', '14', '98'] - [84] vs [84]
Optimization DOne!
Toatl Elapsed time in Training:  14.2688658237 sec
last 3 activity used: '14 14 99'
Next Track to recommend: 180
```

3. Comparison of performance with various RNN model viz LSTM / GRU GRU turns out to be best performer.

Shortcomings / Drawbacks:

- ~3GB data set to train on.
- Data parsing of such large set.
- Didn’t achieve accuracy more than 85% even with large epoch.
- Training on local systems for huge data set.
- Stripping and capping of useless data from given data set.
- Handling new items is not taken care of by this proposed recommendations system.