Jonathan Erell
Dan Wilensky

Recommendation Systems Final Project

Anchor Paper

The anchor paper we chose is "Autoencoders Meet Collaborating Filtering", by a group of students from the Australian National University.

There are vast amounts of approaches for using CF for recommender systems, one of them is using autoencoding neural network, for generating a "missing" prediction by a user. The main objective of the paper is to prove how the new approach of using autoencoders as CF technique, outperforms the state of art algorithms for predicting users' preferences - such as content based, and popularity methods.

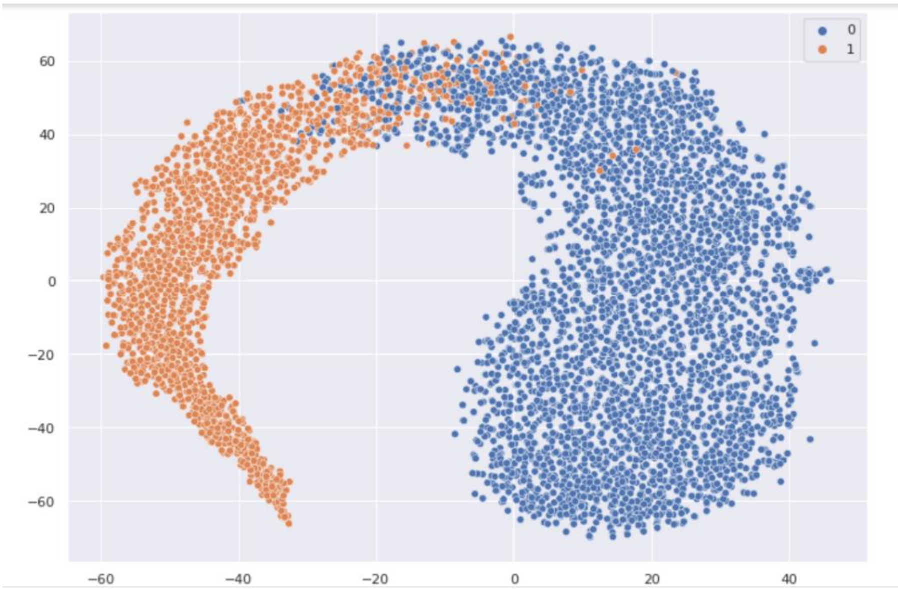
The authors of the paper presented a basic vanilla bottleneck autoencoder. The given data has been pre-processed as a rating matrix, of m users and n items, where $R_{i,j}$ is the rating of the i'th user on the j'th item. The stated approach is an item-based model, where each input's vector of the model will have a fixed dimension (as the number of users - m). The authors used a K dimensional (K<<n) latent dimension as the bottleneck, trained two matrices - $W_{m' \times K}, V_{K \times m'}$ $(m' \leq m)$ which represent the learning weights of the model. In addition to outperforming the methods mentioned above, the proposed method also outperforms another CF known method - 'RBM Based' by the number of learned parameters, which means AutoRec consumes less memory and tends less to overfit.

Improvement Suggestion

There are 2 improvement we have suggested:

1. Use users' profiles for different segments, and for each segment build and train its own AutoRec model.
2. Initialize the training rating matrix with 0's in places where a user hasn't yet rated a movie.

Since the MovieLens datasets are real world and not generated data sets, it is obvious that each user's preference is not generated from the same distribution. We wanted to explore such cases, and by using the users' profiles vector (of dim = N >>2), we reduced the dimension with TSN-E algorithm, to plot these vectors. The results were interesting:

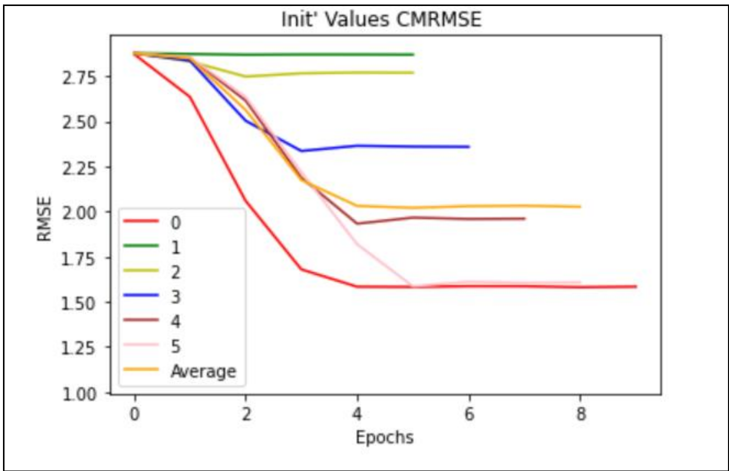


Jonathan Erell
Dan Wilensky

As we can see, the users' preferences are not generated from the same distribution, and some of them are not similar at all. We calculated the weighted average number of ratings per user, created a split where the label of each user with less than average is orange, otherwise blue. The best imaginary method is obviously building a model for each user, but it's not practical since there is never enough data for each user, and training resources will never suffice. How about using 2 different models for 2 different groups of users? Maybe 3 or 4… Such an idea may improve the model's performance, since otherwise, there may be a group of users which are heavily "damaged" by the majority of the users, and the performance of the majority might get "damaged" by the minority. For example, what if 99% of the users' rating were to Horror/Adventure movies, the model will underfit and the predictions for other genres won't be that great.

For the second suggestion, we treated the matrix in the same manner we treated the learned W, V weights matrices, because we want to learn each user's rating. Initializing weights is generally a major aspect of ML, so we tried all possible initial ratings – values between 0-5, and average value per user.

These are the results, sorted by best to worst init value (RMSE), from top to bottom:



	Model	InitRating	CMRMSE
0	AutoRec	0	1.360034
1	AutoRec	4	1.368921
2	AutoRec	5	1.595909
3	AutoRec	Average	2.040143
4	AutoRec	3	2.346179
5	AutoRec	2	2.741149
6	AutoRec	1	2.788876

As we can see, there are at least 4 better init values for the rating matrix than the value 3, which the paper stated.

<u>Presenting the 3 algorithms:</u>

1. **MF-** the baseline algorithm, we chose the vanilla matrix factorization.

The model is built as follows:

```
def get_MF_model(num_users, num_items, latent_dim):

    user_input = Input(shape=[1], name = 'user_input')
    item_input = Input(shape=[1], name = 'item_input')

    MF_Embedding_User = Embedding(input_dim = num_users, output_dim = latent_dim, name = 'user_embedding', input_length=1)
    MF_Embedding_Item = Embedding(input_dim = num_items, output_dim = latent_dim, name = 'item_embedding', input_length=1)

    user_latent = Flatten()(MF_Embedding_User(user_input))
    item_latent = Flatten()(MF_Embedding_Item(item_input))

    prediction = keras.layers.dot([user_latent,item_latent], axes=1,normalize=False)


    model = Model(inputs=[user_input, item_input], outputs=prediction)

    return model
```

Jonathan Erell
Dan Wilensky

Pseudo MF:

$Input$: $train\ matrix\ V$, $latent$ dim K, $learning\ rate\ \eta$
$Output$: $new\ prediction\ matrix, resulted\ as\ the\ matrix\ multiplication\ of\ the\ latent\ dims$

1. $initialize\ matrices\ W_{m\times K}, H_{K\times n}$
2. $while\ not\ converged$:

    2.1 $for\ a\ random\ V_{i,j} \in V$:

        2.1.1 $set\ error = W_i H_j - V_{i,j}$

        2.1.2 $W_i = W_i - \eta\left(error \cdot H_j^T + W_i\right)$

        2.1.3 $H_j = H_j - \eta(error \cdot W_i^T + H_j)$

    2.2 $if\ converged, break$

2. **Autoencoder**- the paper algorithm, a 1 hidden layer bottleneck for dimensionality reduction. The model is built as follows:

```
def AutoRec_Model(X, reg, first_activation, last_activation, hidden_neurons=500):

  input_layer = Input(shape=(X.shape[1],), name='UserRating')
  x = Dense(hidden_neurons, activation=first_activation, name='LatentSpace', kernel_regularizer=regularizers.l2(reg))(input_layer)
  output_layer = Dense(X.shape[1], activation=last_activation, name='UserScorePred', kernel_regularizer=regularizers.l2(reg))(x)
  model = Model(input_layer, output_layer)

  return model
```

Pseudo AutoRec:

$Input$: $train\ set\ matrix\ D_{d\times n}, regularization\ term\ \lambda, encoding(f) and\ decoding(g)$

$activation\ function,\quad k\ hidden\ layer's\ neurons$

$Output$: $predicted\ matrix\ D'_{d\times n}$

1. $initialize\ \theta(W_{b\times k}, V_{k\times b}, \mu, b)$,

the weight matrices, biases of the encoder and decoder respectively

2. $for\ i\ in\ range(epochs)$:

    2.1 $sample\ batch\ size\ b \ll d \to D_{b\times n}$

    2.2 $encode\ and\ regularize = \lambda f[(D_{b\times n} \cdot W + \mu)]$

    2.3 $decode\ and\ regularize = \lambda g[encode \cdot V + b]$

    2.4 $compute\ loss\ grad\ w.r.t\ \theta$

    2.5 $update\ \theta = \theta - \eta\nabla_\theta L$

3. $return\ W, V, \mu, b$

3. **Improved Autoencoder**- As explained before, a 1 hidden layer bottleneck autoencoder, with 2 additions: a 0 initialized training ratings matrix, and users' preference segments models.

```
class Improved_AutoRec_Model():

  def __init__(self, k, reg, hidden_neurons, first_activation, last_activation):
    self.hidden_neurons = hidden_neurons
    self.first_activation = first_activation
    self.last_activation = last_activation
    self.reg = reg
    self.k = k
    self.models = {}

    self.users_item_train = df_to_numpy(train_df, num_users, num_movies, default = 0)
    divided_dataset, _ = users_to_clusters(users_profiles, self.users_item_train, users_item_val, users_item_test, train_df, k=self.k)
    self.new_dataset = get_new_dataset(divided_dataset)

    for i in range(self.k):
      self.models[i] = self.get_model(self.new_dataset[(i,'train')])

  def get_model(self, X):

    input_layer = Input(shape=(X.shape[1],), name='UserRating')
    x = Dense(self.hidden_neurons, activation=self.first_activation, name='LatentSpace', kernel_regularizer=regularizers.l2(self.reg))(input_layer)
    output_layer = Dense(X.shape[1], activation=self.last_activation, name='UserScorePred', kernel_regularizer=regularizers.l2(self.reg))(x)
    model = Model(input_layer, output_layer)

    return model
```

Pseudo code:

$Input: train\ set\ matrix\ D_{d\times n}, regularization\ term\ \lambda, encoding(f) and\ decoding(g)$

$activation\ function,\quad z\ hidden\ layer's\ neurons, users'\ profiles, k\ segments$

$Output: predicted\ matrix\ D'_{d\times n}$

1. $set\ group_0, group_1, \dots, group_k\ as\ the\ result\ groups$

   $of\ running\ Kmeans\ on\ users'\ profiles$

2. $initialize\ \theta_i\left(W_{i_{b\times z}}, V_{i_{z\times b}}, \mu_i, b_i\right) for\ 0 \le 1 \le k,$

   the weight matrices, biases of the encoder and decoder respectively

3. $initialize\ D, as\ 0's\ where\ the\ user\ i\ didn't\ rate\ item\ j$

4. $for\ each\ group, train\ and\ fit\ its\ corresponding\ model\ as\ the\ paper's\ algorithm:$

   4.1 $for\ epoch\ in\ range(epochs):$

   4.1.1 $sample\ batch\ size\ b \ll d \rightarrow D_{b\times n}$

   4.1.2 $encode\ and\ regularize = \lambda f[(D_{b\times n} \cdot W_i + \mu_i)]$

   4.1.3 $decode\ and\ regularize = \lambda g[encode \cdot V_i + b_i]$

   4.1.4 $compute\ loss\ grad\ w.r.t\ \theta_i$

   4.1.5 $update\ \theta_i = \theta_i - \eta \nabla_{\theta_i} L$

   4.1.6. $return\ W_i, V_i, \mu_i, b_i$

5. $return\ \theta_i\left(W_{i_{b\times z}}, V_{i_{z\times b}}, \mu_i, b_i\right) for\ 0 \le 1 \le k$

Performance and Evaluation

The paper uses RMSE as the evaluation metric, and since there is no such thing as a rating lower than 0 or higher than 5, we used clipped masked RMSE (CMRMSE) as evaluation, and masked RMSE (MRSE) as loss function.

We will show each performance, of each model, on both datasets – 1M MovieLens, and 10M MovieLens:

- **MF**:
  Hyperparameters:
    - Learning rates $[5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]$
    - Hidden dim $[4, 8, 16, 32, 64]$

  Performance:

Best 3 hyperparameters w.r.t CMRMSE

1M

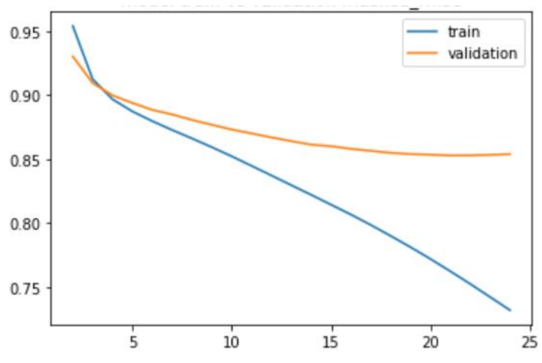
	Model	LearningRate	HiddenDim	CMRMSE
0	MF	0.0005	64	0.847963
1	MF	0.0001	64	0.848445
2	MF	0.0005	32	0.849232

10M

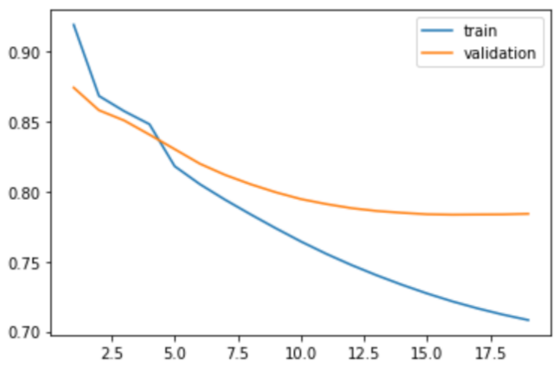
	Model	LearningRate	HiddenDim	CMRMSE
0	MF	0.0005	16	0.794671
1	MF	0.0005	32	0.795678
2	MF	0.0010	16	0.798809

Train and Validation CMRMSE vs Epochs

1M

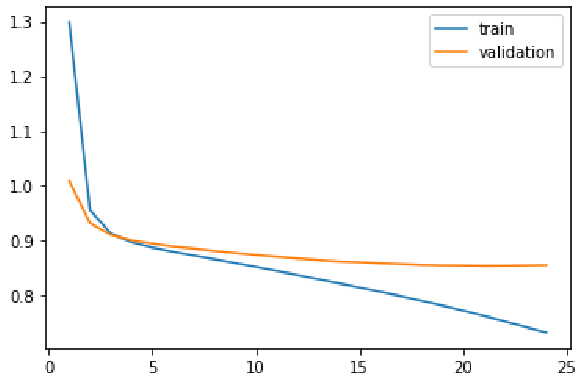


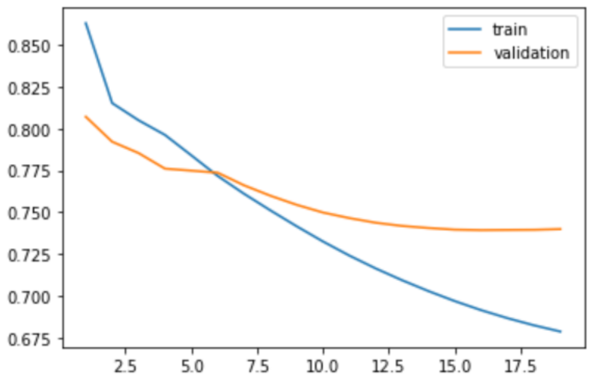
10M



Jonathan Erell
Dan Wilensky

Train and Validation Loss vs Epochs





- **Autoencoder**:
  Hyperparameters:
  - Learning rates $[5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]$
  - Hidden dim $[50, 100, 300, 500, 1000, 1500]$
  - regs = $[1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1]$
  - Activate functions = $[(sigmoid, linear), (sigmoid, elu), (elu, linear), (sigmoid, sigmoid), (elu, elu)]$

  Performance:

Best 3 hyperparameters w.r.t CMRMSE

1M

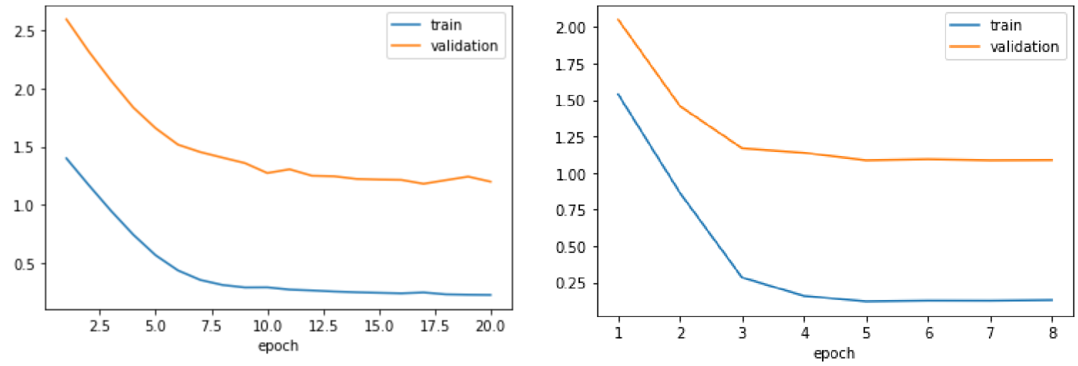
	Model	Regs	HiddenDim	LearningRate	FirstActivate	SecActivate	CMRMSE
0	AutoRec	0.10	1000	0.005	sigmoid	elu	1.199986
1	AutoRec	0.01	300	0.005	sigmoid	elu	1.203488
2	AutoRec	0.01	300	0.005	sigmoid	linear	1.204364

20M

	Model	Regs	HiddenDim	LearningRate	FirstActivate	SecActivate	CMRMSE
0	AutoRec	0.10	1000	0.005	sigmoid	linear	1.091146
1	AutoRec	0.01	1000	0.005	sigmoid	linear	1.093594
2	AutoRec	0.01	1000	0.005	sigmoid	elu	1.095210

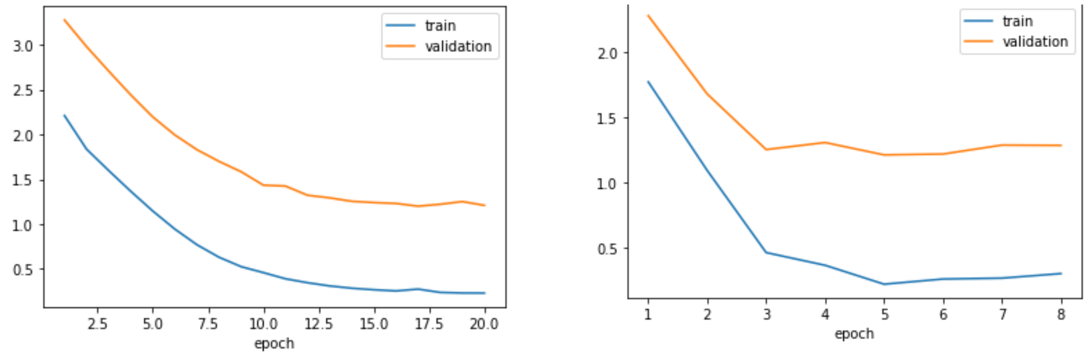
Train and Validation CMRMSE vs Epochs

1M 10M



Train and Validation Loss vs Epochs

1M 10M



- **<u>Improved Autoencoder</u>**:

  Hyperparameters:
  - Learning rates $[5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5]$
  - Hidden dim $[50,100,300,500,1000,1500]$
  - regs = $[1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1]$
  - Activate functions = $[(sigmoid, linear), (sigmoid, elu), (elu, linear), (sigmoid, sigmoid), (elu, elu)]$
  - K different segments = $\{2,3,4\}$

  Performance:

Best 3 hyperparameters w.r.t CMRMSE

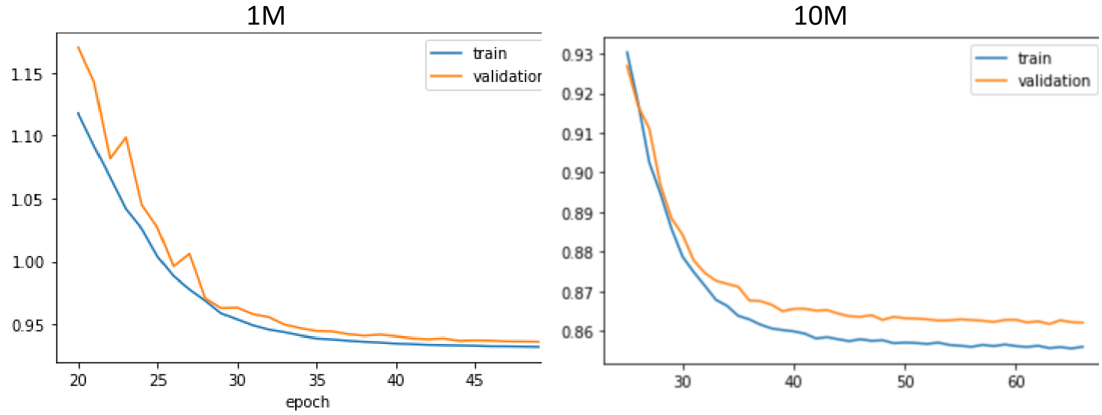
1M

	Model	#Teams	Regs	HiddenDim	LearningRate	FirstActivate	SecActivate	CMRMSE
0	ImprovedAutoRec	2	0.5	1000	0.005	elu	linear	0.944779
1	ImprovedAutoRec	2	0.5	1500	0.005	elu	linear	0.944963
2	ImprovedAutoRec	2	0.1	1000	0.005	sigmoid	linear	0.945562

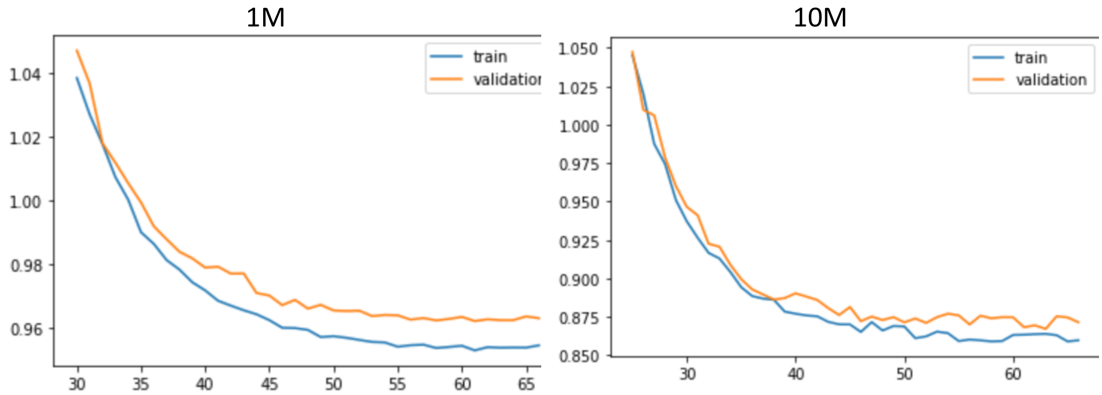
10M

	Model	#Teams	Regs	HiddenDim	LearningRate	FirstActivate	SecActivate	CMRMSE
0	ImprovedAutoRec	2	0.5	1000	0.005	elu	linear	0.859746
1	ImprovedAutoRec	2	0.5	1500	0.005	elu	linear	0.867552
2	ImprovedAutoRec	2	0.1	1500	0.005	sigmoid	linear	0.880318

Train and Validation CMRMSE vs Epochs



Train and Validation Loss vs Epochs



- **Overall Performance**

1M

	Model	#Teams	Regs	HiddenDim	LearningRate	FirstActivate	SecActivate	CMRMSE
0	MF	NaN	NaN	64	0.0005	NaN	NaN	0.847963
1	AutoRec	NaN	0.1	1000	0.0050	sigmoid	elu	1.199986
2	ImprovedAutoRec	2	0.5	1000	0.0050	elu	linear	0.944779

10M

	Model	#Teams	Regs	HiddenDim	LearningRate	FirstActivate	SecActivate	CMRMSE
0	MF	NaN	NaN	16	0.0005	NaN	NaN	0.794671
1	AutoRec	NaN	0.1	1000	0.0050	sigmoid	linear	1.091146
2	ImprovedAutoRec	2	0.5	1000	0.0050	elu	linear	0.859746

Conclusion

- Our proposed improved algorithm ***outperformed the paper's algorithm by*** $\mathbf{21.26\%}$ ***over the 1M dataset, and by*** $\mathbf{23.14\%}$ ***over the 10M dataset!***
- 1,000 hidden neurons as a latent dimension performed the best among all methods.
- The AutoRec models, both the paper's and ours, tend to overfit, and act best as the regularization gets larger.
- Tweaking the init values for the training set had the major impact – the paper's algorithm CMRMSE value changed from 2.7 to 1.3 within that range ($\sim 50\%$ improvement).

Future work

- We are sure that constructing a deeper network would decrease the error. In addition, we could use such a deep NN inside a variational autoencoder, in the same manner of our proposed improvement, by segmenting the users' preferences.
- Setting the init values has a a huge impact as we saw. There may be some better values to init the train set matrix, for example using real values and not 0..5 integers, and maybe using an item-to-item content base first for cold start.