
Recommender systems with TensorFlow

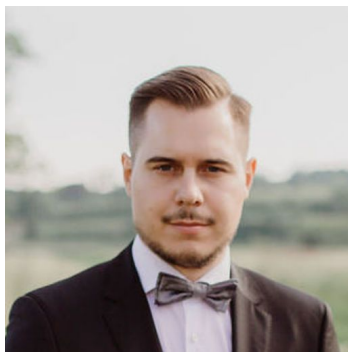
Martin Jakomin
Jan Hartman

About us

Data scientists



Dr. Martin Jakomin



Jan Hartman



What do we do?

- We develop and maintain the “brains” behind the purchasing of online ad space
- Choosing the right ad <-> recommender system!
- We use TensorFlow for real-time ML predictions (1 billion predictions per second)
- Involved in many other interesting things (conversion, bid win prediction, bid price control...)

Agenda, part 1

- Recommender systems paradigms and algorithms
 - Content-based filtering:
 - Item feature extraction
 - Similarity measures and distances
 - Collaborative filtering:
 - Memory based CF (user-user, item-item, hybrid)
 - Basic memory based algorithms (nearest neighbours, ...)
 - Model based CF - latent factors
 - Matrix Factorization
 - Factorization Machines
 - Deep learning for RS:
 - Deep Neural CF
 - Autoencoders & Deep autoencoders
 - DeepFM
 - Recommender systems evaluation and metrics
 - A short case study - YouTube recommendations
-

Agenda, part 2

- **TensorFlow:** basic concepts, tensors, autodiff
- Keras high-level API: layers, models, fitting & evaluation
- Matrix factorization implementation
- Deep autoencoder implementation
- Bonus challenge

Repo link: github.com/janhartman/recsystf

Recommender Systems

An Overview

Introduction

- We often make choices in life without sufficient personal experiences, thus we heavily rely on recommendations.
- Recommender systems (RS) are tools that assist and augment this natural decision making process.



Introduction

- RS are large-scale machine learning and knowledge discovery tools aimed at providing **personalized** recommendations.
 - Examples: videos, movies and songs, web pages, articles, ads, books, social media posts, products, services ...
 - In terms of **machine learning** problems:
 - estimation of user ratings (matrix completion or regression),
 - picking topN items (ranking),
 - picking new unfamiliar items (novelty, diversity).
-

Paradigms And Algorithms

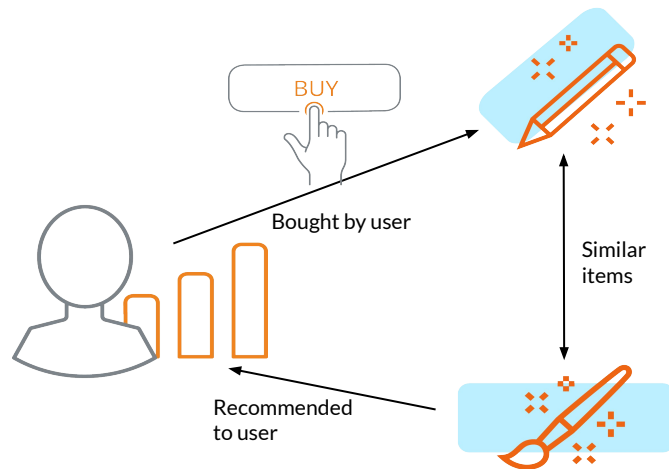
Typically we split RS into three categories:

- ***Content-based filtering*** (where the user is recommended items similar to the ones he preferred in the past),
- ***Collaborative filtering*** (where the user is recommended items that people with similar tastes and preferences liked in the past),
- ***Hybrid approaches*** (that combine collaborative and content-based methods).

Content Based Filtering

Content-based Filtering

- Recommendations are based on user choices made in the past (e.g. previous purchases), combined with rated items **descriptions**.
- The main idea: if you like a particular item in the past you will also like a **similar** item in the future.



Content-based Filtering

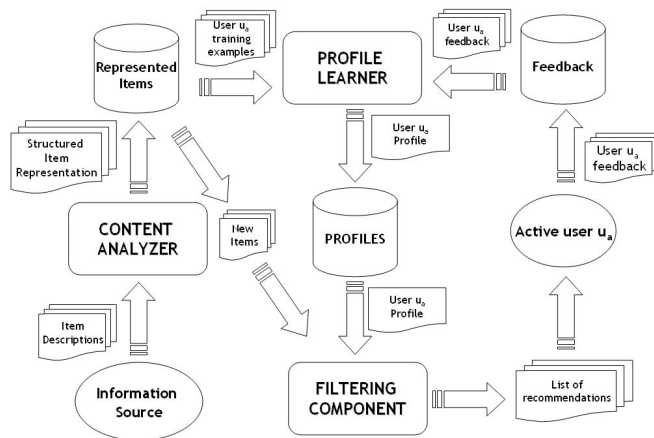
- **Similarity** between objects is therefore the basis for recommending new items.
 - We can view this as a user-specific classification problem of user's likes and dislikes based on an item's features.
 - CBF is best suited to situations where there is abundant data on items, but not for the users.
-

Content-based Filtering

- | | | |
|--|---|---|
| ★ Easy to implement | ↓ | Difficult information extraction |
| ★ Transparent, Explainable | ↓ | Limited content analysis |
| ★ User independent (no user data needed) | ↓ | Overspecialization (only recommending similar things) |
| ★ Scalable | ↓ | User cold-start |
| ★ No item cold-start | | |
-

Content-based Filtering

- The recommendation process is usually performed in 3 steps:
 - Content Analysis (extracting features),
 - Profile learning (extracting user preferences),
 - Filtering (extract relevant items).



CBF - Recommending

- Recommend simply by returning similar items to those which the user has rated in the past.
 - Weighting can be applied in case of real-number ratings.
 - More sophisticated approaches can use Bayesian Classifiers, cluster analysis, decision trees, and neural networks in order to estimate the probability that the user is going to like the item.
-

CBF - Item Similarity

- Item representation:
 - Structured (predefined sets of attributes and values),
 - Unstructured (raw text) - vector space models & TF-IDF,
 - Mixed.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j

df_i = number of documents containing i

N = total number of documents

CBF - Similarity Measures

- Euclidean distance:

$$\sqrt{\sum_{i=1}^n (q_i - p_i)^2}.$$

- Manhattan distance:

$$\sum_{i=1}^n |p_i - q_i|,$$

- Minkowski distance

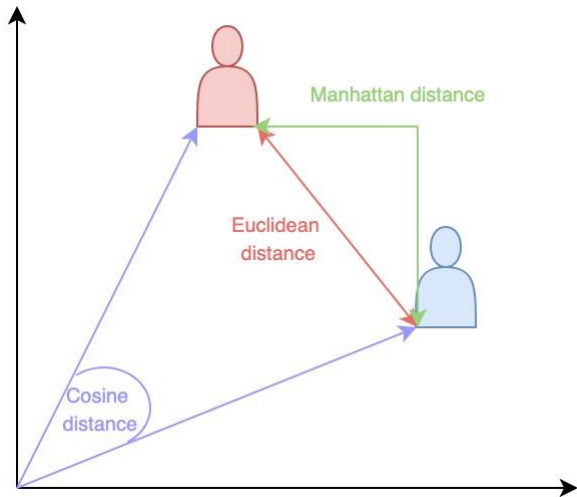
$$D(X, Y) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

- Jaccard distance

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

- Cosine distance

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}},$$



Collaborative Filtering

Collaborative Filtering

- Recommendations are usually based on user's **neighbours** choices made in the past (of similar items).
- The main idea: people who agreed in the past will **agree** in the future and will like similar kinds of items.



Collaborative Filtering

- Oppose to CBF, in collaborative filtering (CF) we generally only use past **rating** information (though, other information/context can be incorporated into the models).
 - No “feature extraction” needed, models can learn abstract (latent) features on the fly.
 - This allows for serendipitous recommendations (items can be completely different from what was seen and liked in the past).
-

Collaborative Filtering

- ★ Superior modelling and accuracy
 - ★ Serendipity
 - ★ Efficient and scalable
 - ★ Justifiable and transparent*
 - ★ Robust and stable
 - ★ Doesn't need item information
- ↓ Black box models*
 - ↓ User and Item cold-start
 - ↓ Can be biased

* Depends on the model

Collaborative Filtering

CF methods are usually grouped in two general classes:

- **Memory-based / Neighborhood-based / Heuristic-based**
 - Where user/item ratings stored in the system are directly used for prediction
 - **Model-based / Latent factors**
 - Where user/item ratings are used to learn a predictive model
-

Memory-based CF

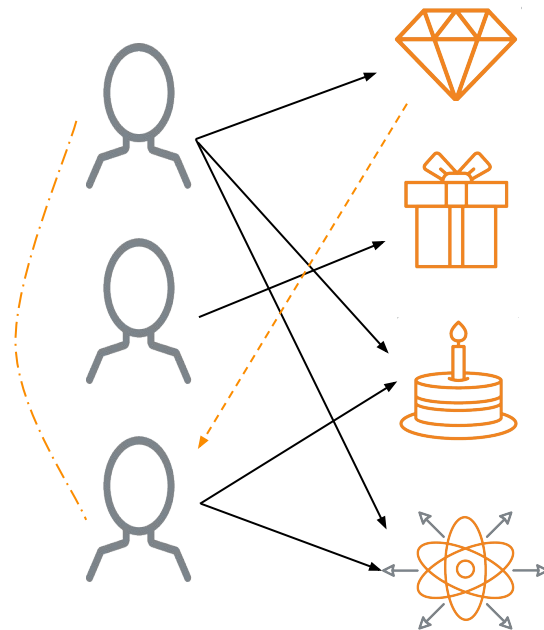
- Most basic approach where ratings are calculated using simple statistics and aggregations upon stored data.
- Very **simplistic**, but can often capture local associations in the data.
- Generally split into:
 - **user-based** (where users are recommended items from their neighbouring users with similar rating patterns),
 - **item-based** (where users are recommended items based on their ratings of the set of similar items).

(Memory) User-based CF

- Predict a rating for a user u for an item i using the ratings for the item i by users most similar to u .
- Weighted kNN (regression or classification).

$$\hat{r}_{ui} = \frac{\sum_{v \in \mathcal{N}_i(u)} w_{uv} r_{vi}}{\sum_{v \in \mathcal{N}_i(u)} |w_{uv}|}$$

Where \mathcal{N} denotes the neighboring users that rated the item, w the user similarity (weight) and r the rating.

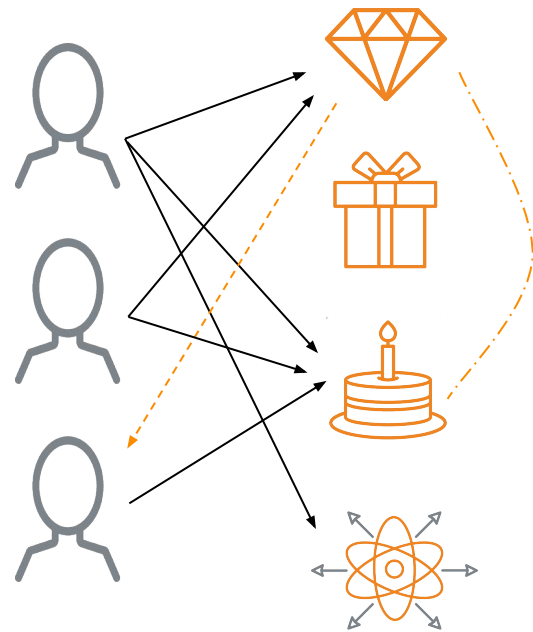


(Memory) Item-based CF

- Predict a rating for a user u for an item i using the ratings by the user u for items most similar to i .
- Weighted kNN (regression or classification).

$$\hat{r}_{ui} = \frac{\sum_{j \in \mathcal{N}_u(i)} w_{ij} r_{uj}}{\sum_{j \in \mathcal{N}_u(i)} |w_{ij}|}$$

Where \mathcal{N} denotes the neighboring items that were rated by the user, w the item similarity (weight) and r the rating.



Memory-based CF

- Memory-based CF methods are **simplistic**, transparent (explainable), efficient and allow for serendipitous predictions.
 - We can further extend MCF with various normalization (centering and biases), choosing different similarity metrics (Euclidean, Cosine, ...) or by applying various pre-filtering and post-filtering stages (neighbourhood selection), speedups (locality-sensitive hashing), ...
 - Finally, we can combine user-based and item-based approaches using Hybrid models.
-

Model-based CF

- Contrary, model-based approaches rather use ratings in order to learn a **predictive model** (which is then used to make new predictions).
 - Goal of modelling user-item interactions with factors representing their latent **characteristics**.
 - These factors are inferred solely from the ratings and might represent obvious dimensions (e.g. movie genres) or something completely uninterpretable to us.
-



Matrix Factorization

- Essential tool in machine learning, broadly used for dimensionality reduction, compression, clustering, classification, ...
 - Also, one of the most popular and most used CF model.
 - Simple and efficient way to model explicit data and various other side information (implicit ratings, biases, temporal effects, etc.).
-

Matrix Factorization

- User-Item ratings can be simply represented with a matrix.
- Recommendations now becomes a **matrix completion** problem.
- We can solve this by factorizing this matrix into a product of (two) smaller matrices.

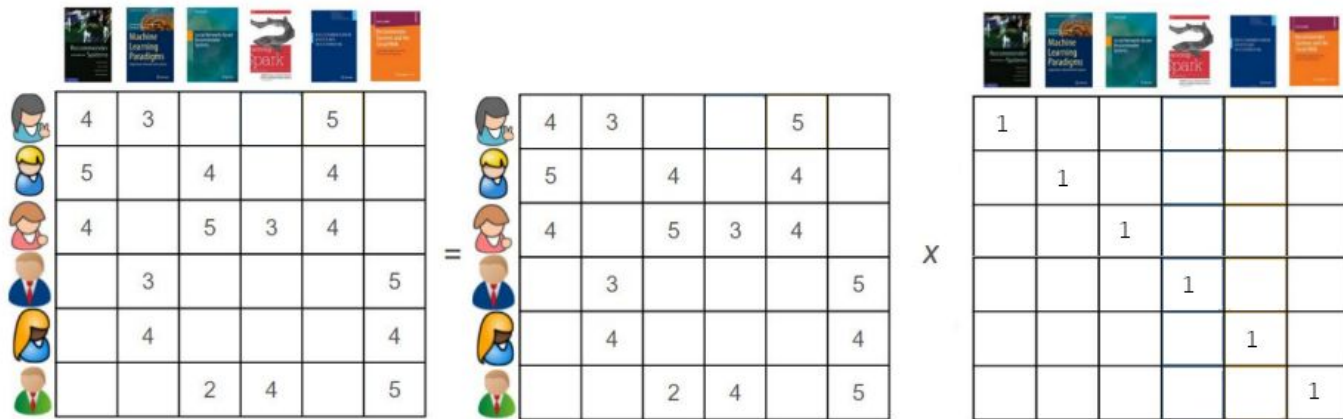
X
 $n \times m$



4	3		?	5	
5		4		4	
4		5	3	4	
	3				5
	4				4
		2	4		5

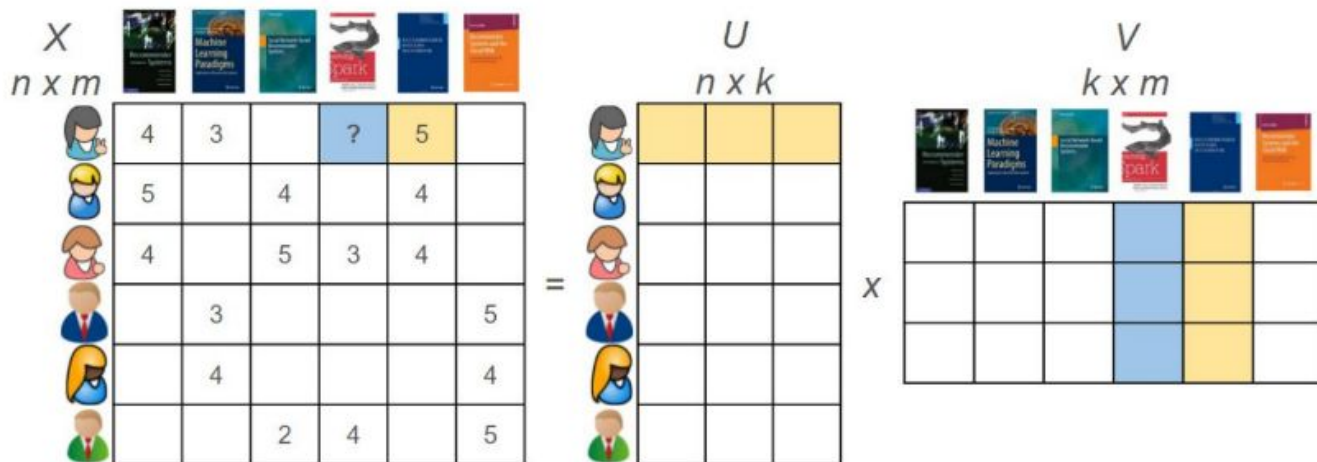
Matrix Factorization - Intuition

- A trivial factorization: $X = X * I$

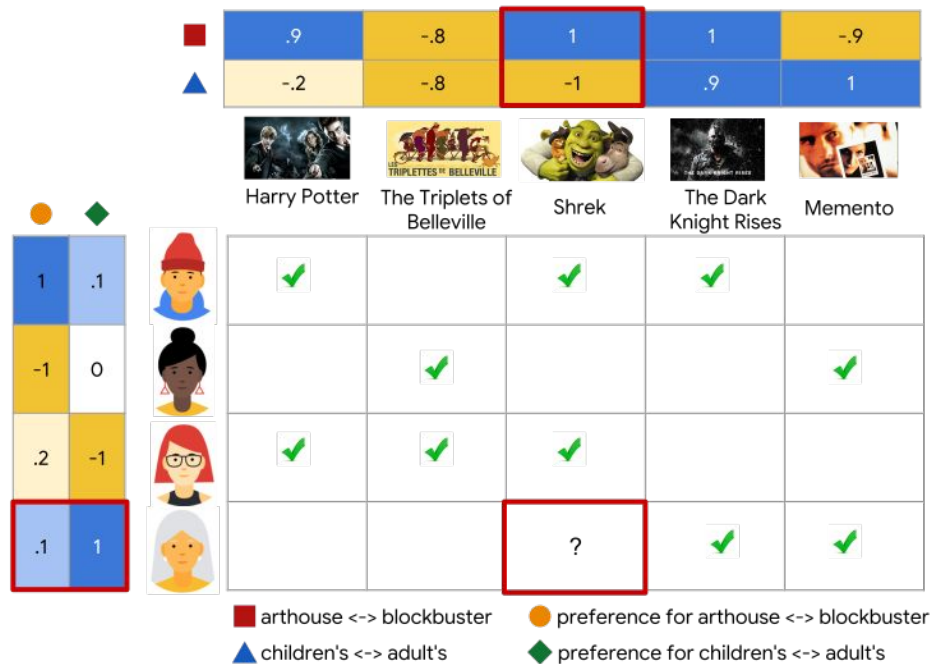
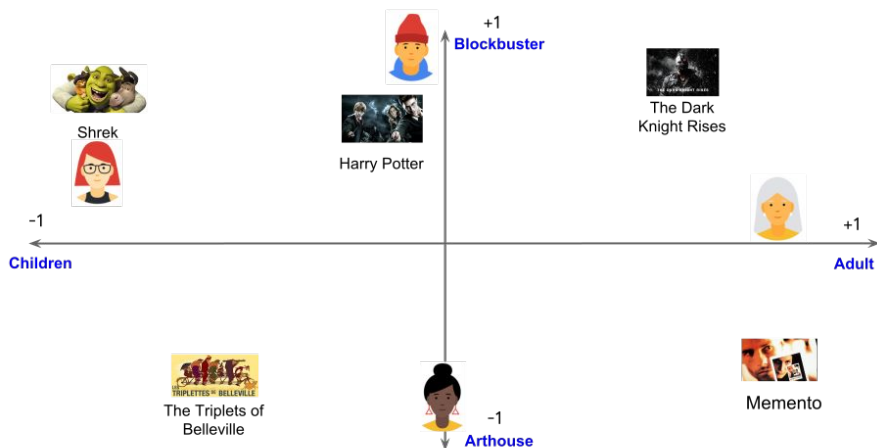


Matrix Factorization

- Low-Rank factorization: $X = U * V$



Matrix Factorization



Matrix Factorization



Matrix Factorization

- **Solve** matrix factorization problem: $X \approx \hat{X} = UV^T$
 - **Predict** with dot product (linear combinations) between latent user matrix and latent item matrix.
 - There are (infinitely) many possible factorizations - how to find the best one?
 - Define the matrix completion as an **optimization problem**.
-

Matrix Factorization

- Define the objective (loss) function, based on your needs.
- One of the most common is the squared loss (Frobenius

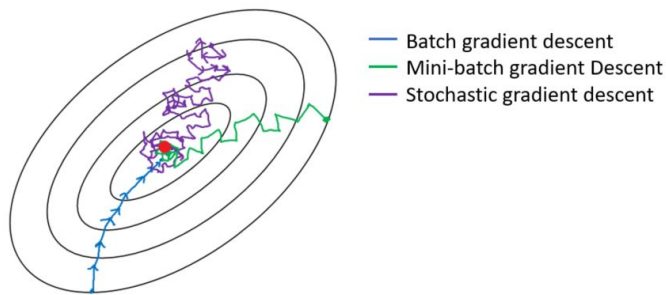
norm):
$$||X - \hat{X}||_F^2 = ||X - UV^T||_F^2 = \sum_{i=1}^n \sum_{j=1}^m (x_{ij} - u_i v_j^T)^2$$

- Problem of huge **sparsity** (typically over 99%).
- We can use weighted (masked) matrix factorization:

$$W \circ X \approx W \circ \hat{X} = W \circ (UV^T)$$

Matrix Factorization

- This kind of optimization problems are usually solved using various gradient descent methods: GD, SGD, MUR, PGD, CD ...
- Alternatively, we can use ALS (convex optimization).
- With SGD we can **omit** the weighted part and optimize only on the **known** (non-zero) ratings and thus tackle the sparsity problem very efficiently.



Matrix Factorization - SGD

$$L = \sum_{u,i} (r_{ui} - (\mu + b_u + b_i + \mathbf{x}_u^\top \cdot \mathbf{y}_i))^2 + \lambda_{xb} \sum_u \|b_u\|^2 + \lambda_{yb} \sum_i \|b_i\|^2 + \lambda_{xf} \sum_u \|\mathbf{x}_u\|^2 + \lambda_{yf} \sum_i \|\mathbf{y}_i\|^2$$

$$\frac{\partial L}{\partial b_u} = 2(r_{ui} - (\mu + b_u + b_i + \mathbf{x}_u^\top \cdot \mathbf{y}_i))(-1) + 2\lambda_{xb}b_u$$

$$\frac{\partial L}{\partial b_u} = 2(e_{ui})(-1) + 2\lambda_{xb}b_u$$

$$\frac{\partial L}{\partial b_u} = -e_{ui} + \lambda_{xb}b_u$$

$$b_u \leftarrow b_u + \eta(e_{ui} - \lambda_{xb}b_u)$$

$$b_i \leftarrow b_i + \eta(e_{ui} - \lambda_{yb}b_i)$$

$$\mathbf{x}_u \leftarrow \mathbf{x}_u + \eta(e_{ui}\mathbf{y}_i - \lambda_{xf}\mathbf{x}_u)$$

$$\mathbf{y}_i \leftarrow \mathbf{y}_i + \eta(e_{ui}\mathbf{x}_u - \lambda_{yf}\mathbf{y}_i)$$

Matrix Factorization

- MF allows for a simple inclusion of various **side information** (like implicit ratings, tags), **constraints** (penalties and rewards), **regularization**, **biases** and **temporal dynamics**.
- Contextual information about the rating process (circumstances) can also be incorporated - CARS:
 - Contextual pre-filtering, post-filtering and modelling
 - $R : User \times Item \times Context \rightarrow Rating$
 - Tensor factorization

Matrix Factorization

- Constrained Matrix Factorization:
 - NMF (only additive linear combinations induce parts-based representation)
 - Sparse MF (constrained number of combinations)
 - Orthogonal MF (uniqueness, clustering)
- Data Fusion

$$\begin{array}{c} R \\ \begin{bmatrix} * & R_{12} & \dots & R_{1r} \\ R_{21} & * & \dots & R_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ R_{r1} & R_{r2} & \dots & * \end{bmatrix} \end{array} \approx \begin{array}{c} G \\ \begin{bmatrix} G_1 & & & \\ & G_2 & & \\ & & \ddots & \\ & & & G_r \end{bmatrix} \end{array} \begin{array}{c} S \\ \begin{bmatrix} * & S_{12} & \dots & S_{1r} \\ S_{21} & * & \dots & S_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ S_{r1} & S_{r2} & \dots & * \end{bmatrix} \end{array} \begin{array}{c} G^T \\ \begin{bmatrix} G_1^T & & & \\ & G_2^T & & \\ & & \ddots & \\ & & & G_r^T \end{bmatrix} \end{array}$$

Factorization Machines

- Factorization machines can be looked as a generalization of linear regression.
- Moreover, they are a generalization of support vector machines (SVM) with a polynomial kernel.
- Except, with FM we do **not** need to calculate every feature interaction (especially if they are sparse): $k \ll n$.

1) *Model Equation*: The model equation for a factorization machine of degree $d = 2$ is defined as:

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j \quad (1)$$

where the model parameters that have to be estimated are:

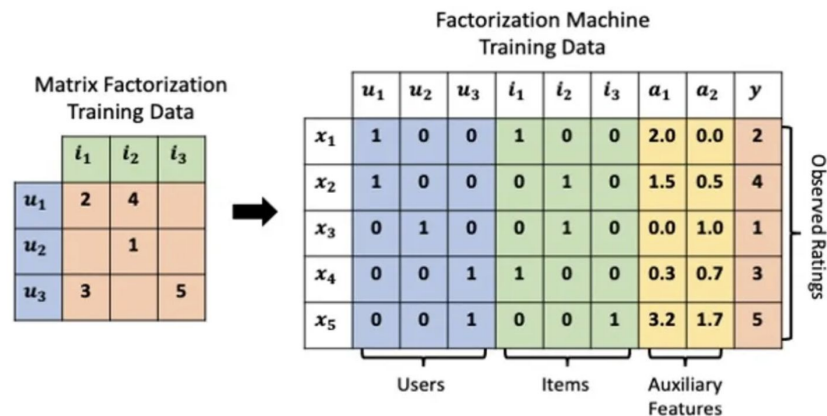
$$w_0 \in \mathbb{R}, \quad \mathbf{w} \in \mathbb{R}^n, \quad \mathbf{V} \in \mathbb{R}^{n \times k} \quad (2)$$

And $\langle \cdot, \cdot \rangle$ is the dot product of two vectors of size k :

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle := \sum_{f=1}^k v_{i,f} \cdot v_{j,f} \quad (3)$$

Factorization Machines

- Allows for **quadratic** (second-order) feature interactions or even higher (d hyperparameter) with variable number of **latent dimensions** (k hyperparameter).
- Designed to capture interactions between features within high dimensional **sparse** datasets
- Still very fast - can be trained in a **linear** time.






Deep Learning For RS

Deep Learning for RS

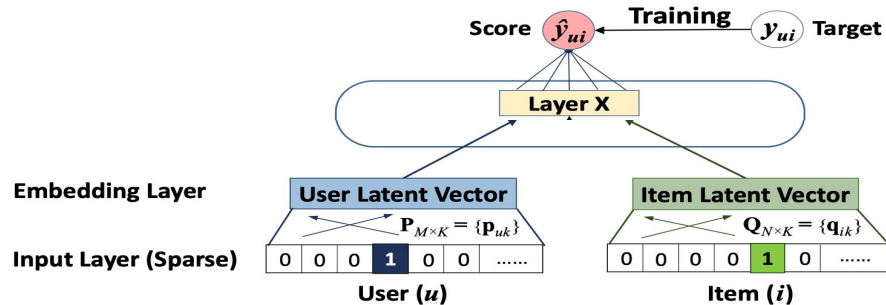
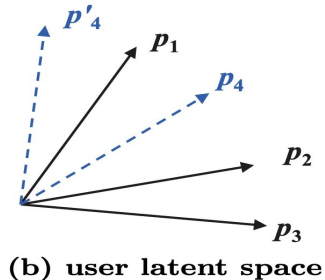
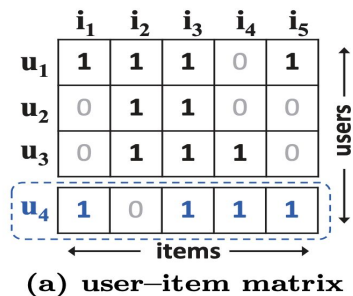
- Nowadays, deep learning is everywhere - RS are not an exception.
- Data is becoming more abundant and available.
- Main Idea - the user/item interaction may be **non-linear**.
- Deep models and Hybrids are SOTA.

Deep RS

- ★ Non-linear transformations  Black box models - bad interpretability
 - ★ Representation learning and better generalization  Data requirement and sparseness
 - ★ Sequence modelling (RNN)  Extensive Hyperparameter tuning
 - ★ Flexibility
 - ★ Superior accuracy
 - ★ Unified representation and data fusion
-

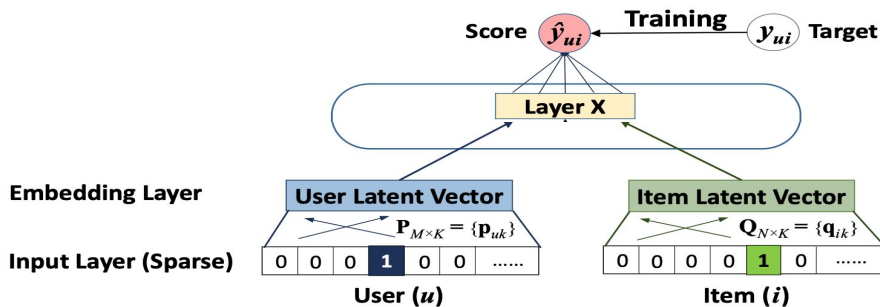
Deep Neural CF

- User and Item embeddings (with one-hot input).
- Dense layers and single output.



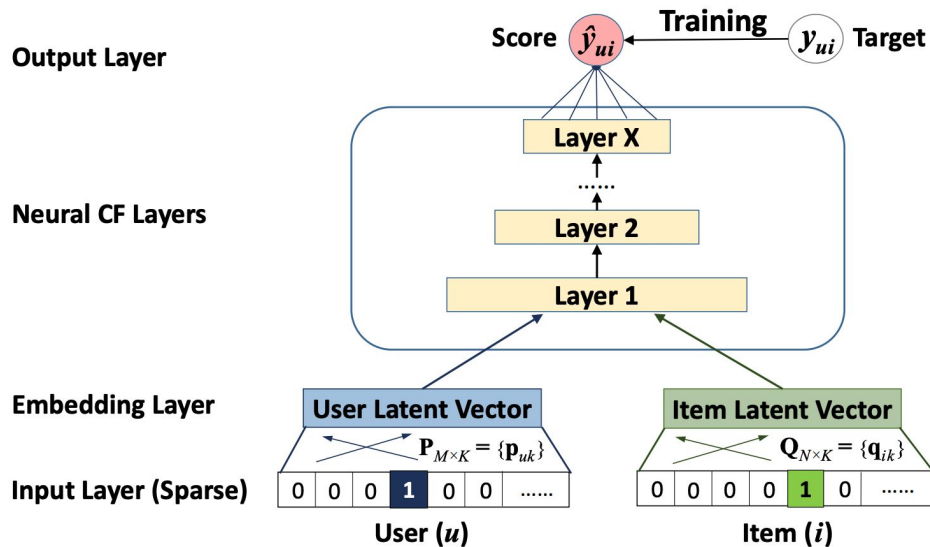
Deep Neural CF

- This is a generalization of the matrix factorization (by using the element-wise product layer).



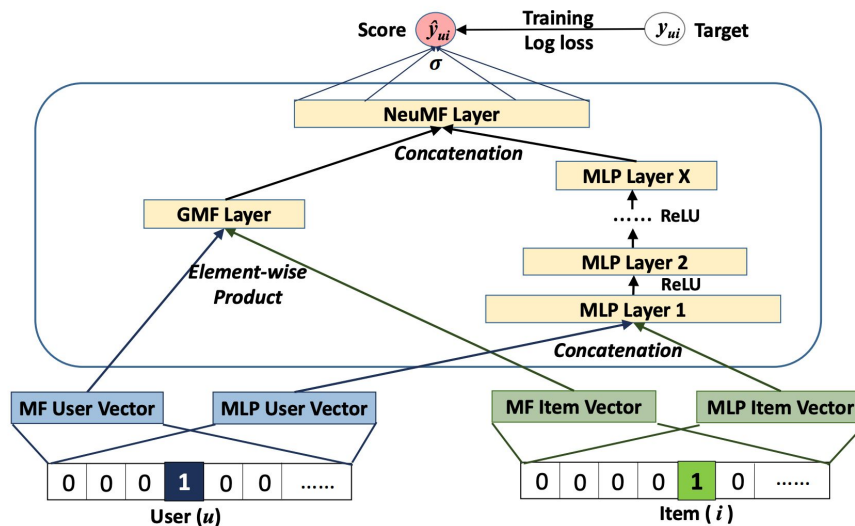
Deep Neural CF

- We can then add multiple dense layers.



Deep Neural CF

- Or combine both models - **Deep & Wide** architecture.

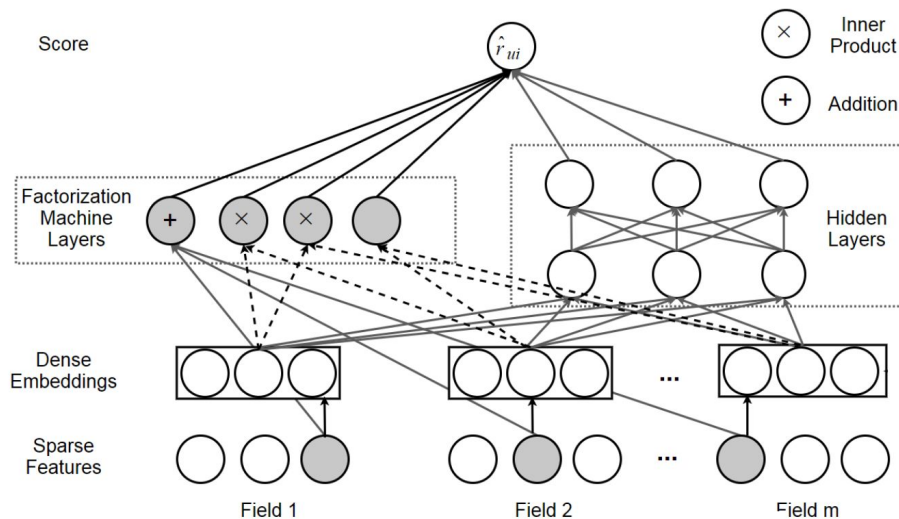


Deep Factorization Machines

- Main idea: Learn both, **low-order** (FM) and **high-order** (DNN) feature interactions.

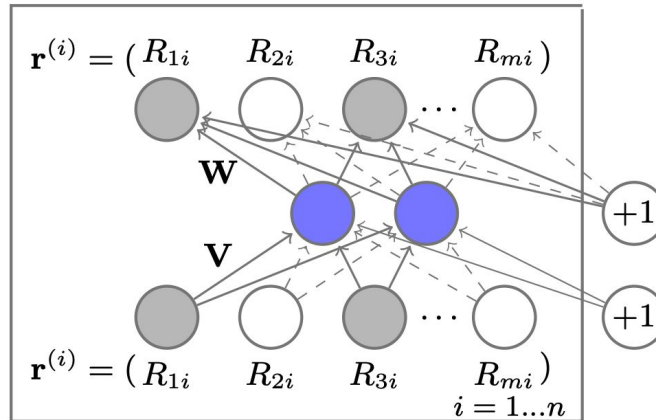
	User				Item				Categories				History			
x^1	1	0	0	...	1	0	0	...	1	0	1	...	1	0	1	...
x^2	1	0	0	...	0	1	0	...	0	2	1	...	0	0	1	...
x^3	0	1	0	...	1	0	0	...	3	0	14	...	1	0	0	...
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
x^n	0	0	0	...	0	0	1	...	0	1	5	...	0	0	1	...

Quantity	
2	y^1
4	y^2
5	y^3
\vdots	\vdots
1	y^n



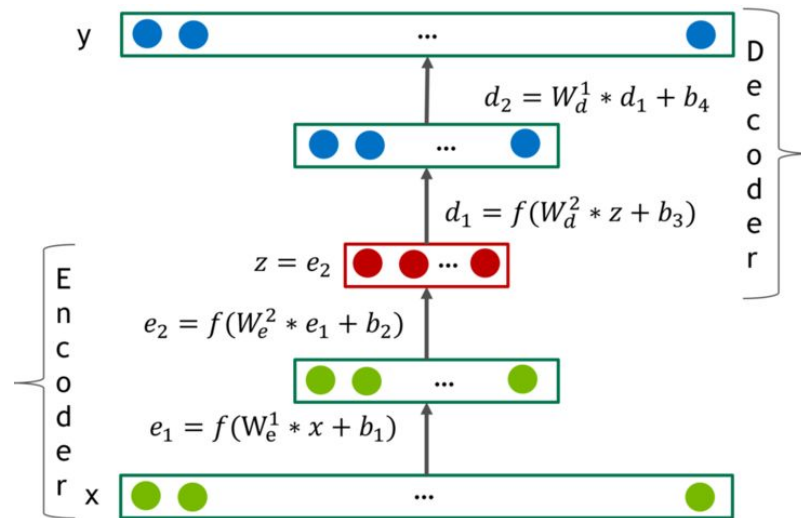
Autoencoders & AutoRec

- Generalization of the PCA.
- Aims to reconstruct the input on the output level.
- Either User-based or Item-based (by passing in the partial vectors of either user ratings or item ratings).



Deep Autoencoders & DeepRec

- Either User-based or Item-based.
- Based on the AutoRec but much deeper with a high dropout rate.
- It uses iterative output re-feeding during the training for better results.



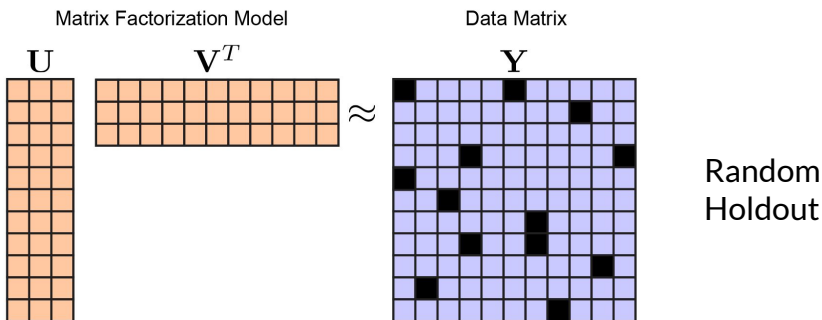
Evaluating RS

Evaluating Recommender Systems

- Inherently difficult, highly sparsity, bias and many outliers.
 - How to measure success? User **satisfaction** is hard to define!
 - How to optimize for novelty and serendipity?
 - In general, we try to optimize the pre-defined loss function (which does not guarantee best predictions or best user satisfactions).
 - Offline results do not guarantee the success in production environment.
-

Evaluating Recommender Systems

- We commonly use an offline holdout set (sub-matrix, random sample) on which we apply predictive accuracy metrics, such as RMSE, MAE, ...
- For topN recommendations we typically use coverage measures, such as precision and recall (and F1).



YouTube - Case Study

YouTube - A short case study

- Largest platform for creating, sharing and discovering video content.
- And also one of the **largest** recommender system in industry!
- Main Goal: Maximize the **time** users spend on the platform (and thus maximize the number of served ads).
- Vast majority of views come from recommendations (more than 70 %).
- However, YouTube is not fully transparent how their algorithms work.

History

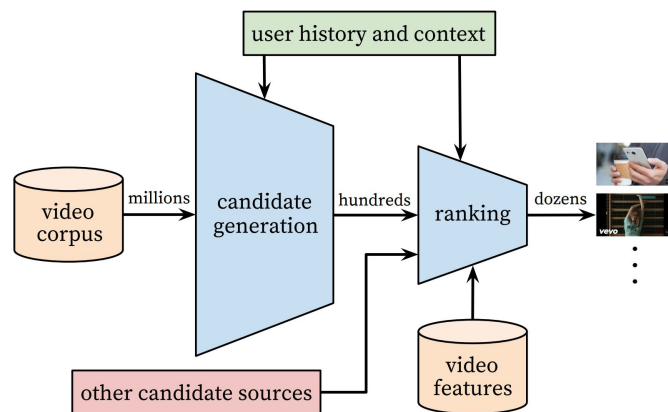
- In the past YouTube recommended videos according to **click-rate**, but this incentivized users to create click-baity videos.
 - YouTube then changed their strategy to recommend videos based on overall **watch time**.
 - Emphasis on user satisfaction and quality watch time (hard to measure).
 - Search results are also personalized.
 - Offline metrics, such as precision, recall, ranking loss are widely used, however the most important metric is the live **A/B testing**.
-

Deep Neural Networks for Youtube Recommendations

- Used matrix factorization in the past.
- One of first to use deep neural network for recommendations .
- Built upon TensorFlow.
- Two-stage design (with two deep neural networks), one for candidate generation and one for ranking.
- This two stage approach allows for recommendation from millions of different videos.

The two-stage design

- The candidate generation uses collaborative filtering in order to provide **broad personalization** (using various features) with a high precision.
- Ranking network scores each video according to the desired objective function and thus provides **fine-level representation** among candidates with a high recall.

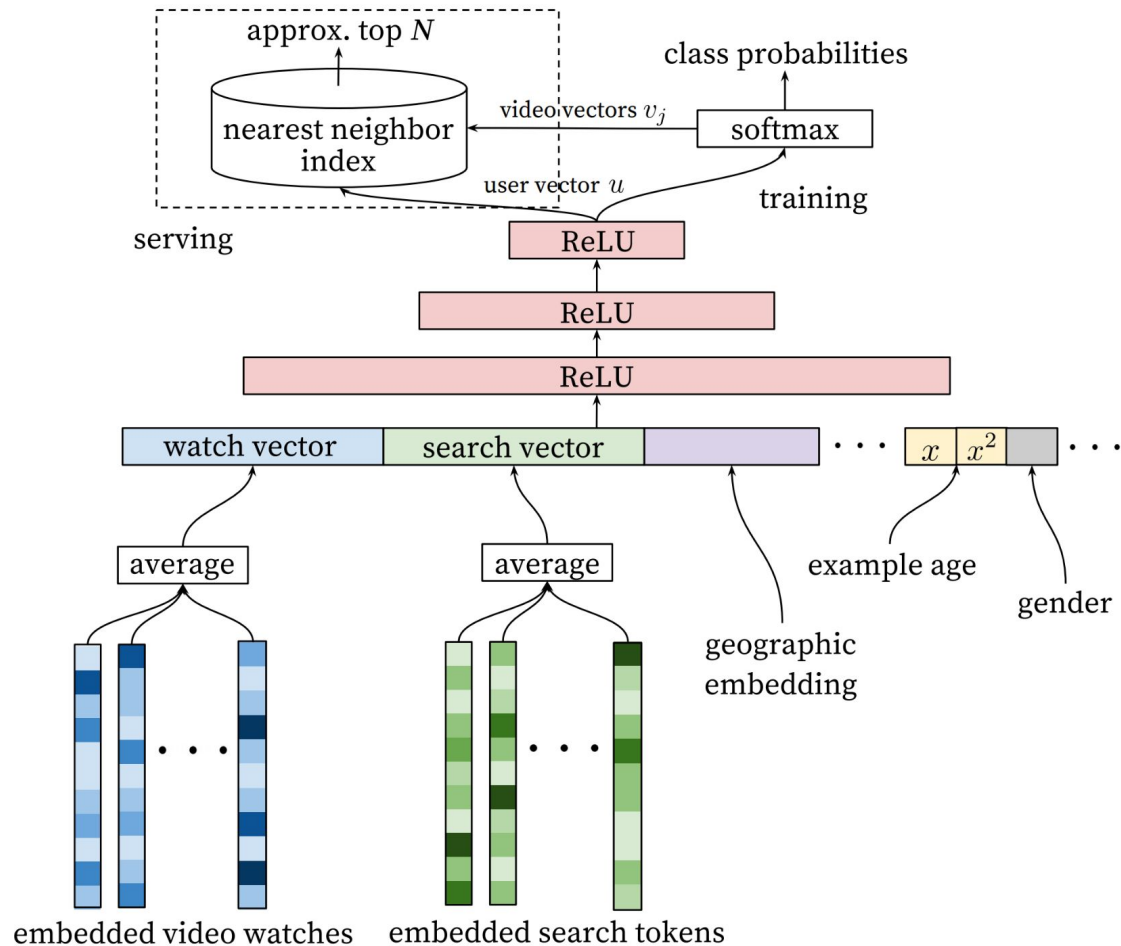


Candidate generation

- Neural networks as a generalization of matrix factorization (for embedding user's behaviours).
 - Recommendation is viewed as an extreme multiclass (1M) classification.
 - Explicit data is very sparse, but **implicit** (such as video views) is more abundant.
 - User watch history is represented by a sequence of sparse video IDs (which is then embedded).
 - Arbitrary continuous and categorical features can be easily added to the model.
-

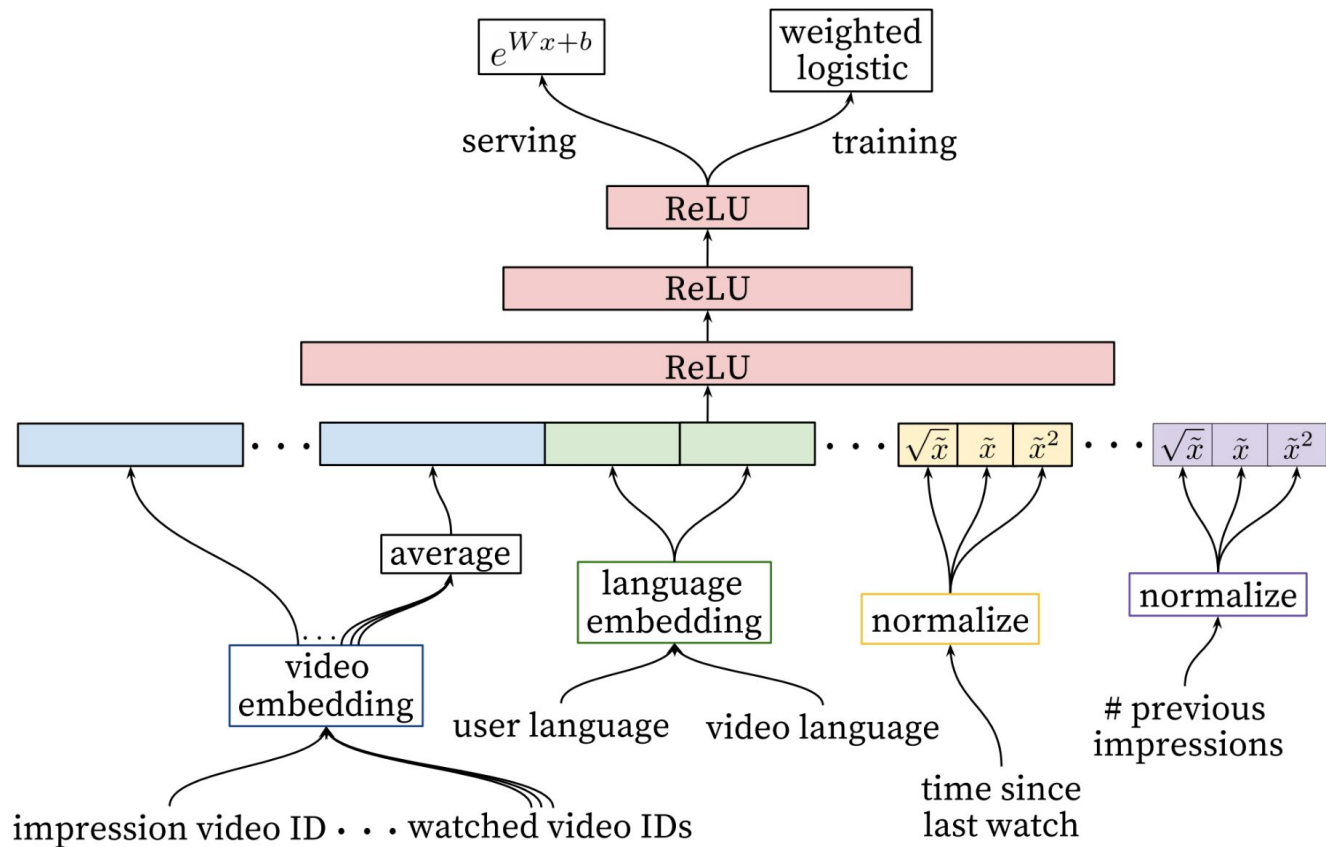
Candidate generation

- Age of the video is very important feature in order to recommend “fresh” examples.
 - Candidates are found by nearest neighbour search from the last layer of the network.
 - Limited number of training example per user (to avoid heavy raters).
 - Adding features and layers significantly improves the accuracy.
-



Ranking

- A similar deep neural network is used as for candidate selection.
 - More video features (such as thumbnails, etc.) are used.
 - **Feature engineering** is very important (although deep learning alleviates this problem).
 - Shared embedding for similar features and normalization of continuous features.
 - Adding features and layers significantly improves the accuracy.
-



Add-ons

- Such design is still not optimal (often only relevant/similar items are recommended and users can get bored).
 - Google then introduced REINFORCE algorithm (reinforcement learning) with goal to maximize users' engagement over time by predicting which recommendations would expand their tastes.
 - YouTube is **constantly updating** their machine learning algorithms and has multiple A/B tests in production.
-

Conclusion

Conclusion - Key Takeaways

- Offline metrics are a good indicator, but they **do not** always map perfectly to the online setting.
 - A/B testing in production is a **must**.
 - Continuous improvements and optimizations.
 - Although deep networks help, manual **feature engineering** can still vastly improve the performance.
 - Adding features and layers usually improve the accuracy (but has higher infrastructure costs and/or diminishing returns).
 - Optimize for **user satisfaction**.
-

TensorFlow



What is TensorFlow?

- *"An end-to-end open source machine learning platform"*
 - At its core: general purpose ML framework, focused on DL
 - Core implemented in C / C++ / Python, wrappers for Java / Go / Swift
 - Peripherals: TF Serving (microservice for serving models), TensorBoard (model visualization / profiling), etc.
-

Pros & cons of TF

- + A lot of implemented operations and models
 - + Optimized for speed, CPU, GPU support
 - + Ecosystem (users, libraries, end-to-end platform) - most widely used DL framework
 - Complexity
 - Poor documentation + awkward API in some places
-

Code time

- Repository: github.com/janhartman/recsystf
 - Option 1: run the notebooks on Google Colab
 - Colab: <https://colab.research.google.com/>
 - Click GitHub and enter *janhartman/recsystf*
 - All dependencies are preinstalled
 - The colab notebooks are stored on your GDrive
 - Option 2: run the notebooks locally
 - Clone the repo and open it
 - Create virtualenv and install dependencies
 - Run *jupyter-lab*
-

Stuff we left out

- Saving and loading models
 - TF Serving, production usage
 - TF in other programming languages
 - Custom op creation
 - Estimators and tf.data pipelines
-

Sources (TF part)

- [TensorFlow guide](#)
 - [Deep Learning With Keras: Recommender Systems](#)
 - [Collaborative Filtering for Movie Recommendations](#)
 - [NVIDIA/DeepRecommender: Deep learning for recommender systems](#)
 - [An implementation of DeepRecommender in Tensorflow & Keras](#)
-

Challenge

Data

- Data: [Goodreads book ratings](#), filtered
 - 84k train, 21k test examples
 - *train.csv* & *test.csv*: user ID, book name & rating
 - Bonus: additional book data in *books.csv* (match by book name) - try to find a way to add it to the model
 - *books.csv*: authors, publish date, publisher, rating, number of reviews, number of pages
-

Challenge

- Goal: implement a recommender system in TF to accurately predict a user's rating of a book
 - Use mean squared error as your loss function & a validation dataset
 - Play around with different algorithms, add extra book data into the model...
 - Try to achieve the lowest MSE on the test set (mine: 0.84)
 - Make sure to use additional data from *books.csv*
-

Conclusion



We are hiring!

- Always looking for interesting candidates
 - Various positions are open
 - Hard and interesting problems
 - Big data
 - outbrain.com/careers
-

Thank you for your attention!



Martin Jakomin

mjakomin@outbrain.com

[linkedin.com/in/martin-jakomin](https://www.linkedin.com/in/martin-jakomin)

Jan Hartman

jan.hartman7@gmail.com

[linkedin.com/in/jan-hartman](https://www.linkedin.com/in/jan-hartman)
