Movie Recommendation System

Tiberiu Simion Voicu

Post graduate thesis project, ECS751P

Abstract

Recommender systems aim to provide users with personalized recommendations of items. This helps a lot in filtering irrelevant items from the big collection. It is usually achieved through collaborative filtering. Neural networks have proven to be highly effective in natural language processing, computer vision and speech recognition. Despite this, they haven't seen as much use in solving recommendation problems. This is partly because state of the art matrix factorization methods are already very effective for this problem. The system described in this paper will make use of multilayer perceptron networks to solve the recommendation problem using collaborative filtering. The design and implementation details of the recommendation engine will be discussed and the results compared with traditional matrix factorization methods. Furthermore, it will also go into detail about the choices that went into creating a scalable overall system and accompanying web application.

Contents

1. Introduction

Nowadays customers are faced with a vast choice in products, services and content. However, consumers don't want to waste time manually filtering through these choices. Recommender systems (RS) overcome this by providing personalized recommendations that suit a particular users' tastes. They are employed by tech companies in many distinct areas to predict ratings or preferences of their users. Spotify uses them to create personalized playlists. Amazon uses them to suggest products that consumers may wish to purchase. Facebook uses them to recommend pages and people to follow. In this paper I will discuss the best practices in deep learning and propose tower architecture network that incorporates extra content information into the model. I will also discuss the design choices that went into creation of a modern scalable web application to host and serve the recommendations. Later I will compare the effectiveness of the algorithm with state of the art matrix factorization methods. The experiments on the movieLens dataset show a decrease of 4.77% and 5.86% in RMSE compared to SVD and NMF factorization algorithms.

1.1. Dataset

The dataset used as the basis for the recommendation engine is the movie lens 20m dataset, put together by the Grouplens research group at the University of Minnesota (Harper and Konstan [6]). This is an explicit feedback dataset, where users manually assigned ratings to movies. The characteristics of this

dataset are the following.

- 20000263 ratings on a scale of 0.5-5 in 0.5 increments
- 27278 movies having been rated at least once
- 138493 users that have rated at least 20 movies
- 465564 tags about movies

The dataset is split into different files of which I am only using the ratings, movies and links files. They are all available in comma-separated value format.

- Ratings file - userId, movieId, rating, and timestamp
- Movies file - movieId, title, genre
- Links file - movieId, imdbId and tmdbId

The ratings file is the main dataset which will be used for collaborative filtering. The movies file is used to incorporate genre content data into the model. The links file provides useful ids from IMDB, a movie reviews website. These ids are used for webs-scrapping, to make available extra content for the demo web application..

2. Background & Related Work

The most common types of recommender systems can be split into collaborative filtering and content based. They can be further split into model-based, and memory-based.

2.1. Content Based

Content based recommendations require several features related to an item instead of historic user-item interactions. In the case of movie recommendations, these could be year, actors, genres, producers, writers, etc. This method relies on calculating the similarity between items using features such as the ones previously stated. The general idea is that if a user likes a given item he will also like items similar to it. One way of achieving this is by calculating term frequency (TF) and inverse document frequency (IDF) of the items. Then using the vector space model and a choice of similarity metrics such as cosine or pearson, we can compare different items (Pazzani and Billsus [17]). Another way of achieving this is to represent different content items as dense vectors that can be easily fed into a neural network architecture.

2.2. Collaborative Filtering

Collaborative filtering (CF) algorithms aim to recommend items to a user by combining the item interactions of a given user with item interactions of all other users. CF can be split into two categories. User-based where the aim is to measure the similarity of a given user and all other users (Zhao and

Shang [26]). Item-based where we aim to measure the similarity between the items a given user has interacted with and other items (Sarwar et al. [20]). The most widely used method of achieving this is through factorization of the very sparse user-item interaction matrix.

2.3. Matrix Factorization

The idea behind matrix factorization (MF) is to decompose the matrix R containing user-item interactions, into the product of two lower dimensional matrices P of size nXk and Q of size kXm. Matrix P is the user matrix where n is the number of users, k the number of latent factors and p_u is the latent vector of user u. The other matrix is the movie matrix with m number of movies, the same k latent factors and q_i is the latent vector of item i. The predicted rating $\hat{y}_{ui}$ can then be calculated by taking the dot product of those two vectors (Koren et al. [13]).

$$\hat{y}_{ui} = f(u, i|P_u Q_i) = P_u^T Q_i = \sum_{K=0}^{k} P_{uk} Q_{ik} \tag{1}$$

Singular value decomposition (SVD) and non-negative matrix factorization (NMF) are two techniques successfully applied in literature to achieve the decomposition (Koren [12]).

The resulting matrices are dense and have much lower dimension than the initial matrix R. By choosing a different number of latent factors we can include more or less abstract information. MF poses the recommendation problem as a regression optimization one. Two common optimization metrics

used for this are root mean squared error (RMSE) and mean absolute error (MAE). The RMSE and MAE can be calculated as follows given that, e_i is the difference between the actual and predicted value of rating i.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2} \tag{2}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n} \mid e_i \mid \tag{3}$$

Stochastic gradient descent (SGD) and alternating least square (ALS) are two optimization algorithm generally employed to learn a good approximation for the matrices. ALS is more often used for implicit dataset because its more efficient than SGD on such data. The algorithm is succinctly described below.

- 1. Set item factor matrix constant and adjust user factor matrix.
- 2. Set user factor matrix constant and adjust item factor matrix.
- Repeat until convergence.

2.4. Neural Networks & Deep Learning

Neural networks are universal approximators, typically organized in layers of neurons connected through weights and put through an activation function. The number of layers and neurons at each layer also called the depth and width of the network are variable and many different configurations seem to

work in practice. The simplest neural network is made up of 3 layers, input, hidden and output.

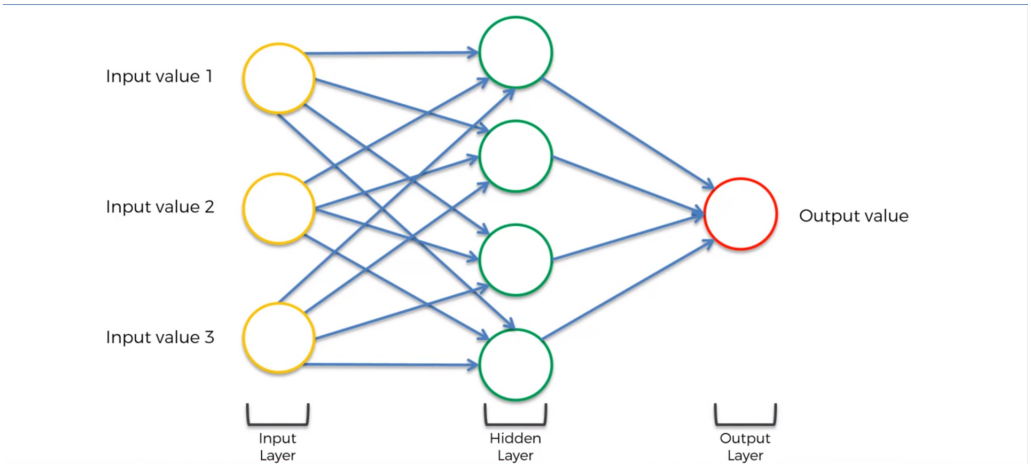


Figure 1: 3 Layer Neural Network

They can be used for both supervised and unsupervised learning and can be applied to classification and regression problems just as effectively. Deep neural networks (DNN) are NN of more than 3 layers, although in practice and state of the art implementations these are many layers deep and very wide as well. The main benefit of using NNs algorithms is that they don't require manual feature engineering. A big disadvantage is the lack of interpretability for the predictions, due to this NN are generally viewed as black boxes. This means that we are not aware of the 'why' and 'how' did the network product a certain output given some inputs. This could be especially detrimental in the context of recommendation systems because the users might wish to know the reasons behind their recommendations. Other memory and similarity based methods provide much more transparency in this regard.

At the core of DNNs success in modeling complex function is the back propagation algorithm. The back-propagation algorithm allows the network to learn by propagating the loss, calculated at the output layer, backwards to update the weights at each layer (Rumelhart et al. [19]). It uses the chain and product calculus rules to compute partial derivatives of the cost function for each node in the network.

Different classes of NN have emerged in literature including convolutional networks (Krizhevsky et al. [14]), autoencoders (Vincent et al. [24]), recurrent networks (Sherstinsky [21]), generative adversarial networks (Goodfellow et al. [5]), besides of simple fully connected multilayer perceptrons.

2.5. Related Work

Strub et al. [23] propose a hybrid recommender system based on auto-encoders. They use a loss function adapted to data with missing values. They also describe models that incorporate side information and find out these models deal better with the cold-start problem. He et al. [8] propose a neural collaborative filtering system trained using implicit data. They design a novel ensemble framework that joins together generalized matrix factorization (GMF) and multilayer perceptron (MLP). The inputs to the network are user and item ids which are embedded into dense low dimensional vectors. The embeddings are shared by both GMF and MLP. GMF is supposed to mimic MF techniques and works by taking the dot product of the item and user embeddings. The user and item embeddings are concatenated be-

fore being fed into MLP. They train the network using Adam optimizer and ReLU activation functions in the MLP. Their results show that more layers in the MLP leads to better performance while keeping the number of factors constant. Furthermore, they compare the ensembles effectiveness with similarity and model based state of the art models and report 4% improvement in hit ratio.

3. System Architecture

3.1. Technologies, tools and frameworks

Many different tools, libraries and frameworks were used in building the demonstration application. I will discuss the most important ones in greater detail.

3.1.1. React

React is a fast, declarative and efficient javascript library for creating web interfaces. It works around the concept of components. They are self-contained and composable blocks of code that encapsulates a part of user interface and its functionality. By putting together multiple small components it's possible to build complex user interfaces (UI). Components can be stateful or stateless. The library provides a virtual-dom similar to the browsers document object model (DOM). They are both node trees that list elements together with attributes and content as objects and properties. Updating the dom is rather slow which is why the virtual-dom is useful for efficiency. It allows react to optimize DOM updates under the hood to only happen when it's necessary.

3.1.2. Docker & Containers

Containers are self-contained pieces of code that can be run on any computer and operating system (OS). They contain the code and all of its necessary parts such as libraries, tools, and frameworks. They are similar to virtual machines but the main difference is in efficiency and application size. Containers

are more efficient and smaller because they run on the same underlying kernel as the operating system as opposed to virtual machines which runs an entirely different OS.

Docker is tool that allows creating, running and managing containers. First we must create a dockerFile, which defines all the dependencies needed to run our code 6.

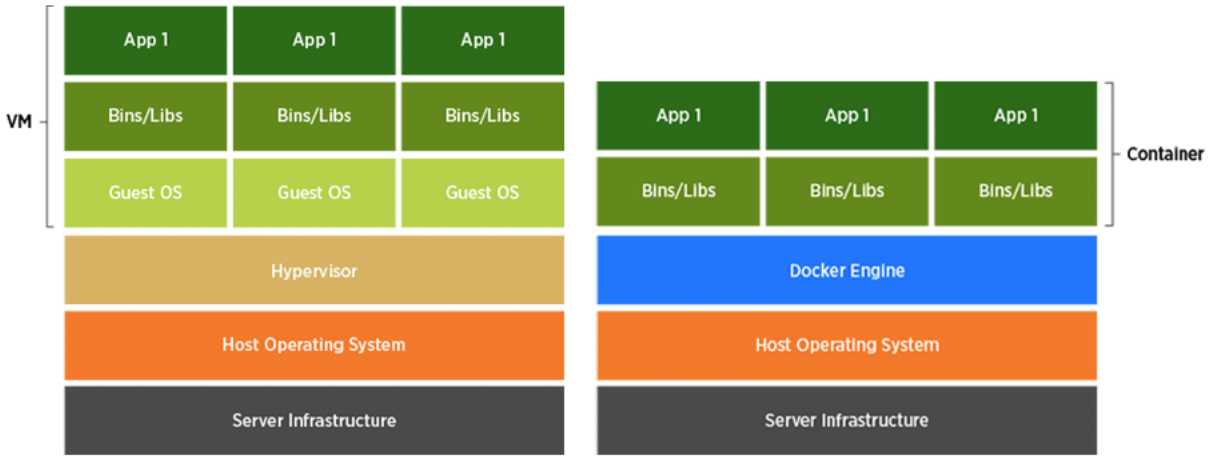


Figure 2: Container and virtual-machine comparison diagram

3.1.3. Kubernetes

Kubernetes is a container orchestration platform created by Google and open-sourced in 2014. It allows automation of deployment, scaling, and management of containerized applications. It groups the containers that make up a

multi micro-service application into logical units for easy discovery and management. It's built with scalability in mind and provides other useful features such as load-balancing, logging, and monitoring. Kubernetes allows running containers in a cluster of physical and virtual computers called nodes. This is especially easy when using a cloud provider because of the vast availability of resources. I decided to use GKE, which is googles implementation of kubernetes in the cloud. This is in part due to their experience with running it in production environments used by millions of people.

3.1.4. Mongodb & Mongoose

MongoDB is an open source document oriented database, which stores data in json like format. Its characteristics include high availability, high performance, high flexibility and easy scaling. It stores data internally as BSON, a binary representation of json A collection is a schema-less entity which contains multiple documents and is the counterpart of a relational database table.

Mongoose is a javascript object data modelling library, that provides an easy to use interface for working with mongoDB. It provides an easy way to translate between javascript objects and mongoDB document representations. While being schema-less at database layer provides great flexibility, in a modern application we still need to enforce it at the application layer. This can be achieved with mongoose models and schemas. The schema much like in relational databases defines fields and their types. A model is just a

wrapper around a schema. This model provides an extensive and flexible api for working with the underlying collections.

The application make uses a mongoDB atlas cloud instance with a separate database for each micro-service to store data. Due to the nature of mongoDB its possible too scale together with demand to more instances and shards.

3.1.5. Keras

Keras is a deep learning framework that provides an easy to use high level API. It provides efficient implementations of generally used layers, activations, loss functions and optimizers. The API is very intuitive and has a friendly implementation as its core principle. Furthermore, it is modular and easy to extend with custom functions and classes. Keras wraps around a different number of low-level libraries called backends. These are Theano, Cognitive Toolkit, and TensorFlow. I have decided to use it with the tensorflow backend, because of its performance and ease of multi-gpu training (Abadi et al. [1]).

3.2. Micro-services

The demo application follows a micro-services architecture composed of four applications 3. Each service has a specific role in the overall system.

- Gateway service is tasked with keeping track of user sessions and authentication.
- User service keeps track of user data including all their ratings.

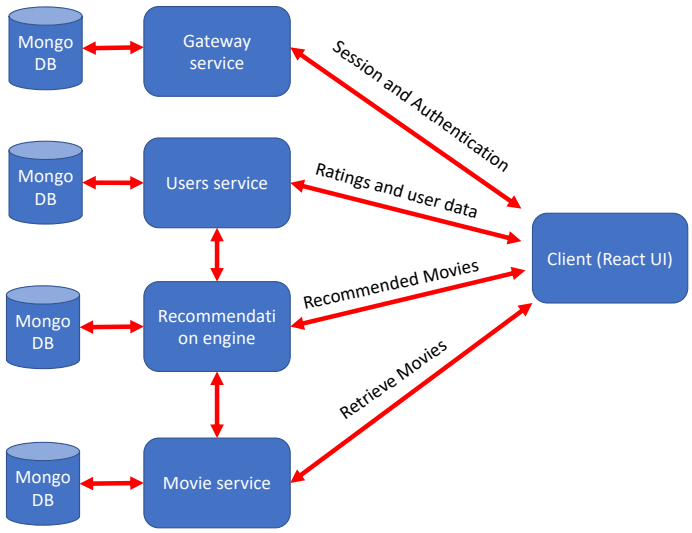


Figure 3: Architecture diagram

- Movie service manages movies and all data related to them.
- Engine service manages the recommendations model, including retraining and serving.

The gateway exposes endpoints for login, signup and creation of initial session. The movie service exposes endpoints for retrieving movies by id or search based on a query. In case of query retrieval it handles pagination of results. The user service exposes endpoints for adding a new rating for a movie, updating a movie rating with a new value, getting all the ratings of a user. The engine service exposes endpoints for retrieving top k recommendations of a certain user based on their id. Where k is a parameter in the request which allows serving different number of top recommendation results. It also exposes another endpoint which allows serving recommendations from

a subset of the movies as defined by the search query results described in the movie service api.

3.3. Web application

The demo application is created with ease of use in mind. It does not require creation of an account and instead uses the gateway service to provide a signed json-web-token (JWT) to identify users. This token is held in the browser local storage of each user. The frontend uses this token in authentication headers when making request to the backend services. If at any point later a user decides to create an account, the system automatically handles it by merging the current data with the newly created account 4.

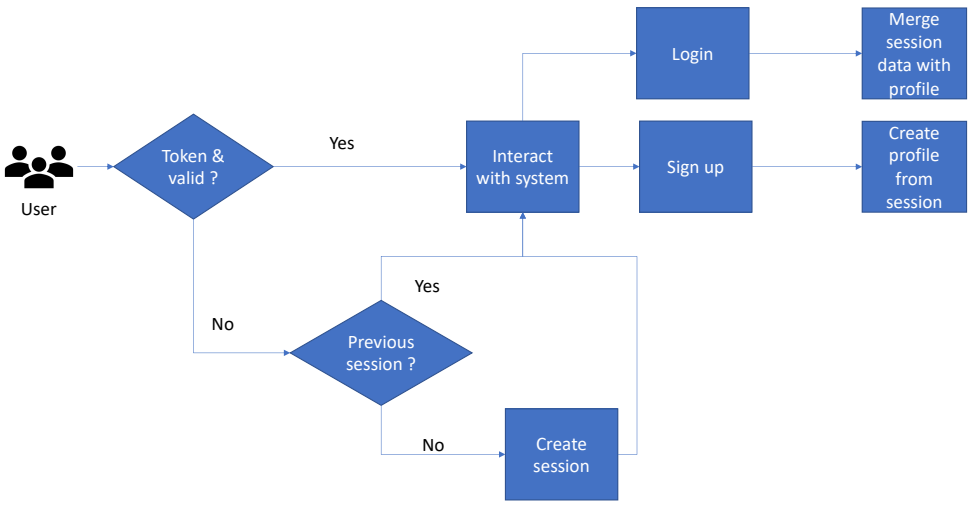


Figure 4: User flow diagram

The demo app displays a list of movies that can be rated on scale of 0.5 to 5 in 0.5 increments. The app was built with user interface responsiveness in

mind from the beginning, this means the layout is self-adjusting to display nicely on different size displays and also on mobile.

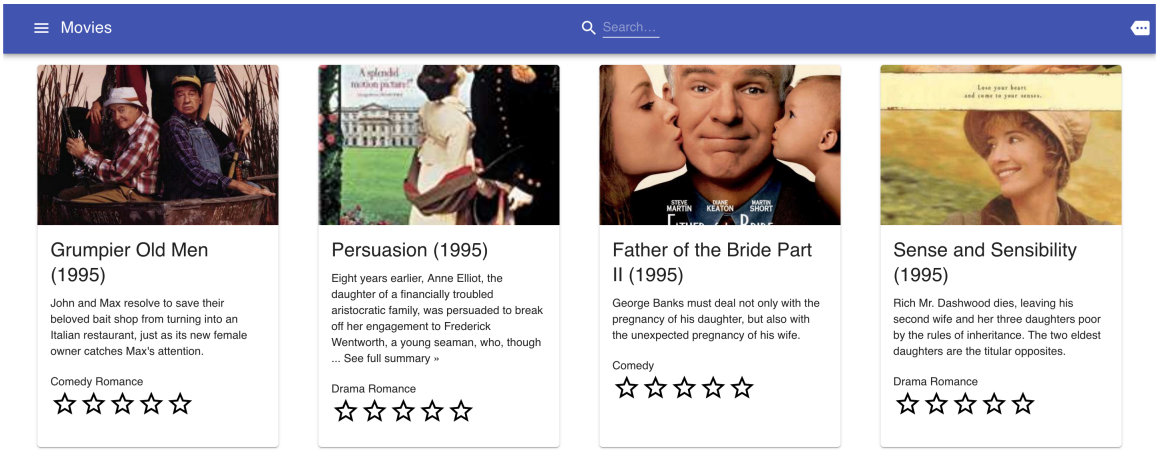


Figure 5: Movie list desktop

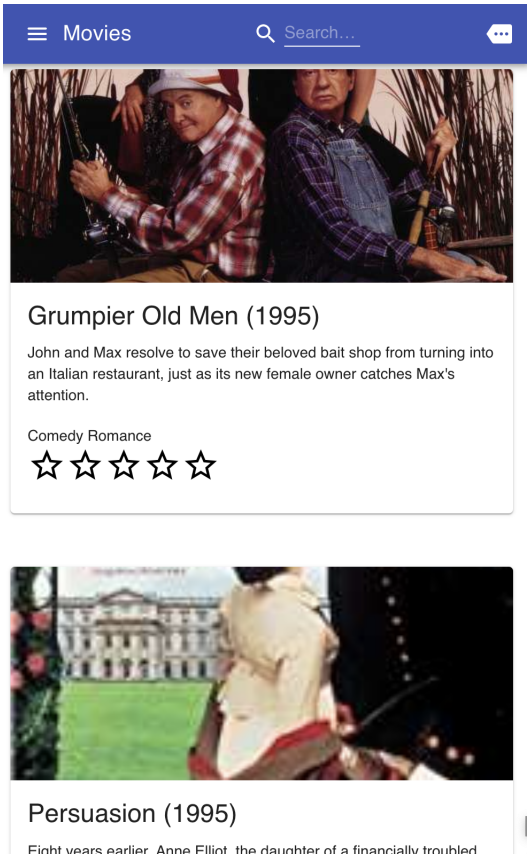


Figure 6: Movie list mobile

4. Recommendation Engine

4.1. Embeddings

Neural networks do not deal well with categorical variables. One hot encoding is a way to handle this. It allows us to represent categorical data as sparse vectors of zeroes and a single 1 representing the specific category. This method has two main drawbacks. Firstly, the dimensionality of the vector representation size grows with the corpus. It can become unmanageable very quick. Secondly, each vector is equidistant from every other vector. This means 'similar' categories are not represented close together in the vector space.

A better way of handling categorical data is through embeddings. When using embeddings we can project categories in a low dimensional latent space and represent them as dense continuous vectors. They are learned parameters and due to this, similar items are projected close together in the latent space. Therefore it is useful in the context of recommendations as we are trying to model user-item similarities. They are also well studied and highly applied in literature for natural language processing to represent words (Mikolov et al. [15]). Embeddings can be pre-trained and adapted to be used in a model or learned end-to-end with the other parameters.

The main inputs to the model consist of user, item and genre embeddings learned iteratively with the rest of the model parameters.

Genres are handled slightly different than the other two because a movie can

have more than one genre. The basis of the embedding is a multi hot encoding, meaning the vector has a value of one for each category that describes it.

4.2. Activation function

In a neural network activation functions are mathematical equations, applied to each neuron, that determine if it should activate or not based on its inputs. This function must be computationally efficient to calculate, differentiable and will generally be non-linear. The last part is very important because without non-linearities a NN would just behave like a single-layer perceptron and would not be able to model complex functions. One exception to this is the output layer for a regression NN which will have a linear activation to allow the prediction of any real value.

Early neural networks were using hyperbolic tangent (tanH) and sigmoid activation functions. Sigmoid also known as logistic function is S-shaped and bounded by 1 and 0 4. It suffers from vanishing gradient for very high or low values of x which can result in the network refusing to learn after some point. TanH function is similar to sigmoid but suffers less from the vanishing gradients problem 5.

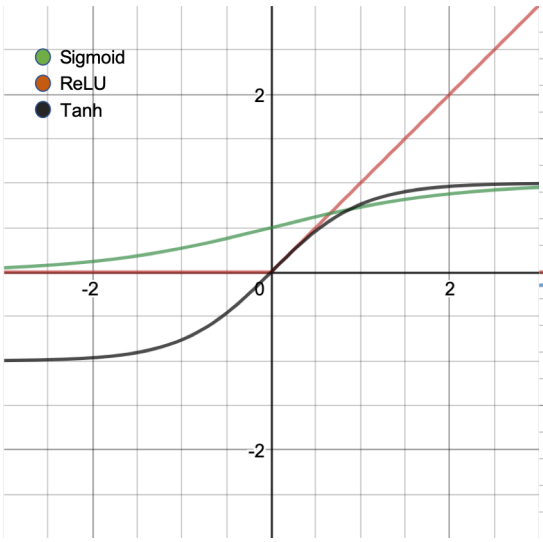
$$\begin{aligned} f(x) &= \frac{1}{1+e^{-x}} \\ f'(x) &= f(x)(1-f(x)) \end{aligned} \tag{4}$$

$$\begin{aligned} f(x) &= \frac{2}{1+e^{-2x}} - 1 \\ f'(x) &= 1 - f(x)^2 \end{aligned} \tag{5}$$

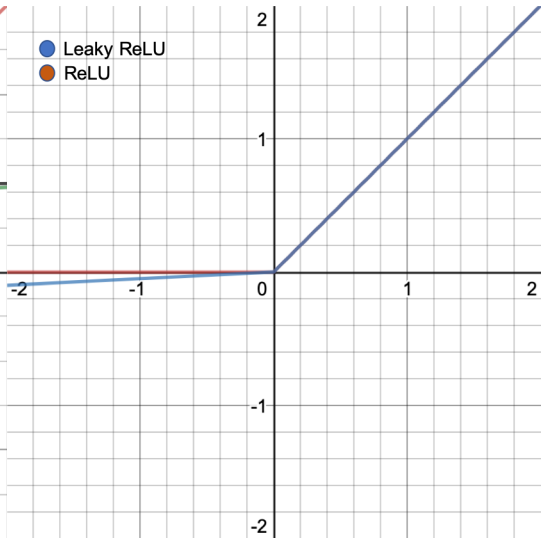
Rectified Linear Unit (ReLU) was first introduced by Nair and E. Hinton [16] in 2010 and is nowadays highly used in deep learning. It introduces sparsity in the network because it outputs 0 for negative numbers 6. Moreover, reLU is more computationally efficient than the other two activations. However, a major drawback of using ReLU activations is the "dying ReLU" problem. A ReLU is said to be 'dying' when its stuck outputting 0. The leakyReLU 7 is an adaptation of the original. It allows a small value as output for negative samples. This fixes the problem of "dying ReLU" leading to a more robust model (Xu et al. [25]).

$$
\begin{aligned}
f(x) &= \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \\
f'(x) &= \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
\tag{6}
$$

$$
\begin{aligned}
f(x) &= \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x, & \text{otherwise} \end{cases} \\
f'(x) &= \begin{cases} 1, & \text{if } x \geq 1 \\ 0.01, & \text{otherwise} \end{cases}
\end{aligned}
\tag{7}
$$



(a) Activation functions



(b) ReLU vs Leaky ReLU

4.3. Regularization

One of the most common problems in training neural networks is over-fitting. A model is said to be over-fitting when it performs well on training data but poorly on validation data. Intuitively, it means the model is unable to generalize well and only memorizes the training examples. This generally occurs if the model is too complex or the size of the training dataset is too small. Regularization is a method of reducing over-fitting through small modifications of the learning algorithm. The most common types of regularization are l1 and l2. L1 regularization adds the absolute value of the coefficients as penalty term to the cost function. This leads feature sparsity, which is desireable in some cases. L2 is similar but adds the squared value of coefficients

as penalty term to the cost function.

$$
\begin{aligned}
l1penalty &= \frac{\lambda}{2m} * \sum || \ w \ || \\
l2penalty &= \frac{\lambda}{2m} * \sum || \ w \ ||^2
\end{aligned}
\tag{8}
$$

In the equations lambda is the hyperparameter controlling the regularization strength. The parameter is usually found using cross-validation or through experimentation. I have experimented with values between $1e^{-4}$ and $1e^{-8}$. The final model uses l2 weight regularization on embedding and dense layers with lambda of value $1e^{-7}$.

Dropout is a special way to regularize the network. This changes the normal training routine by dropping random neurons with a probability p. The edges of a dropped neuron are also removed for that epoch. Therefore, at each epoch we train on a smaller model. This stops the NN from depending too much on any specific neurons (Srivastava et al. [22]). During testing or prediction phase dropout is disabled. The probability p of dropping neurons is another hyperparameter that has to be selected similar to λ in l1 and l2. I have tried values of p between 0.1 - 0.5 in 0.1 increments. The best values of 0.2 and 0.5 are applied in the final model on embedding and dense layers respectively.

4.4. Optimizer

Optimizers are a very important parameter in NN configuration. Nowadays there is a vast choice of good optimizers. At its core, an optimizer is an it-

erative method of optimizing for a cost function. At each optimization step, the weights in the NN will be updated based on the negative of the gradient of the cost function. Stochastic gradient descent (SGD) is a variation of gradient descent in that the optimization happens after each training example. In practice, this usually implies mini-batches of between 32 and 1024 data points (Bengio [2]). Batch gradient descent involves updating the weights based on the gradient over the whole dataset. SGD optimizes based on an approximation of the gradient, unlike batch gradient descent. This turns out to be useful as it introduces noise in the network which leads to better generalization. It is also more scalable as the whole dataset does not have to be kept in memory. (Bengio [2])

$$\theta = \theta - \alpha \Delta_\theta J(\theta; x^{(i)} y^{(i)}) \tag{9}$$

More advanced optimizers are built on top of SGD and include things such as adaptive learning rates and momentum to increase convergence speed and overall stability.

One such algorithm is adaptive moment estimation (Adam). Its widely used in literature and converges much faster than SGD. Adam can be seen as a combination of RMSProp and momentum (Kingma and Ba [11]). Adam uses estimations of first and second order moments of the gradient to adapt the learning rate for each weights in the network.

Nesterov adaptive moment estimation (NAdam) combines adam with nes-

terov momentum which improve convergence (Dozat [3]).

I have decided to use NAdam for training following experiments that proved its the best choice for this specific problem and on this dataset. I am using it with learning rate of $2e^{-4}$, epsilon value of $1e^{-4}$, and the other parameters are keras defaults for this optimizer.

4.5. Training

Weight initialization plays an important role in training a neural network. Ideally, we wish the initial weights to be random but not too small and too big, otherwise, it will lead to problems of vanishing or exploding gradients. Xavier normal initialization is one technique that constricts the weights to these characteristics. It works by drawing from a truncated random distribution centered on 0 and with a standard deviation of $\frac{2}{n_{in}+n_{out}}$, where n_{in} and n_out represent the number of inputs and outputs (Glorot and Bengio [4]). This is the default initializer used in keras. It is best suited for use in NN that employ tanh activation functions. He normal initialization works better for relu activation. Its similar to xavier, but the standard deviation is $\frac{2}{n_{in}}$ (He et al. [7]).

```
xavier_w = np.random.rand((n_in, n_out)) * np.sqrt(2 / (n_in + n_out))
he_w = np.random.rand((n_in, n_out)) * np.sqrt(2 / n_in)
```

Too speed up the training I have implemented a special generator class extending keras *sequence*. This enables multiprocessing execution of the batch generation algorithm and more importantly it ensures safety and single use

of each training example per epoch. After creating the generator its possible to use the keras *fit_generator* function with the number of workers and max processing queue size instead of *fit*. This achieves a good speed up in training as the GPU does not have to wait for the CPU to prepare the training batch. More importantly, with these changes it's possible to train on multiple GPUs using data parallelization. In this case the batch will be divided equally among the GPUs. Moreover, I have enabled the keras callback *EarlyStopping* with patience of six. This acts as regularization method, because it stop training after the model validation loss has not decreased for 6 consecutive epochs. Moreover, it also rewinds the model parameters to the epoch that had the best loss.

Batch Normalization is a technique proposed by Ioffe and Szegedy [10], which aims to reduce the amount by which hidden unit values shift. It normalizes the input to a layer by subtracting the batch mean and dividing by the batch standard deviation. It turns out this leads to faster convergence and more stability in training. Furthermore, the batch normalization layers also have a small regularization effect which helps the network generalize better.

Finally, the model was trained using 12 workers, a queue size of 200, and batch of 512 data points split evenly between 4 GPUs.

4.6. Architecture

The NN architecture follows a common tower pattern, where layers near the top are widest and progressively decrease in width. The inputs to the

network consist of user, movie and genre embeddings and average movie and user ratings. They are then concatenated fed into the first hidden layer. The activation functions employed at the hidden layers are leakyReLU. Each hidden and embedding layer is followed by a dropout layer and a batch normalization layer. The output is made up of a single neuron with linear activation.

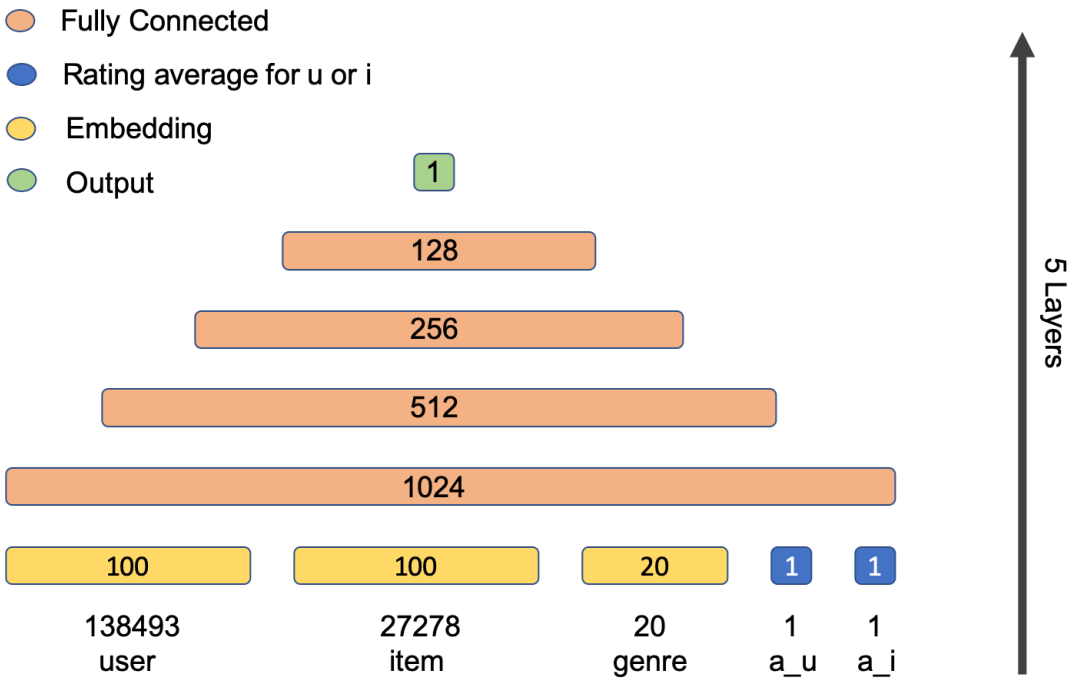


Figure 7: Network architecture

5. Results

I have trained singular value decomposition and non-negative matrix factorization models on the same dataset using python surprise library in order to compare with my implementation (Hug [9]). The models were optimized for 100 epochs each and 3 fold cross-validated. The SVD model was run λ of 0.05 for regularization and learning rate of 0.001. The NMF model trained used biases instead of baselines and regularization λ of 1.

The result are in 1 and show that even without side-information my model achieves a 4.17% decrease in RMSE compared to SVD model and 5.26% when compared to NMF model. Adding genres information reduces RMSE and MAE by 0.32% and 0.97% respectively when compared to basic NNCF. The final model incorporates genres and average ratings for movie and user. It improves on NNCF by 0.62% RMSE and 1.66% MAE.

Model	Test RMSE	Test MAE
SVD	0.8152	0.6190
NMF	0.8246	0.6287
NNCF	0.7812	0.5960
NNCF+genres	0.7787	0.5902
NNCF+genres+avg	0.7763	0.5861

Table 1: Results Comparison (RMSE & MAE)

6. Conclusion

In this paper I have covered the best practices in deep learning and recommender systems. These include regularization, activation functions and principles of fast and stable multi-gpu training in keras. Furthermore I have implemented a demonstration web application that allows rating movies and serving recommendations. It is designed to be scalable and easy to extend in the future. I have also compared the DNN performance with traditional matrix factorization algorithms. The results show that deep learning can be used for solving the recommendation problem and it achieves better evaluation performance than the MF methods.

6.1. Future Work

Possible improvements to the system include the following.

- Incorporate more content information into the system such as movie title, summary, and cast. This could be achieved using pre-trained word embeddings such as word2Vec or glove (Pennington et al. [18]).
- Add recurrent layers such as GRU or LSTM.
- Train a convolutional network on movie posters. Then incorporate said network into the CF model.
- Use a deeper and wider architecture for network.

7. Acknowledgements

Firstly, I would like to thank my supervisor, Dr Nikos Tzevelekos, for his support, guidance, and mentoring throughout the duration of this project. I also wish to thank the lecturers on the Big Data course at Queen Mary for supporting the development off skills and knowledge that were necessary to undertake the writing of this paper and implementation of accompanying project.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X. [2015], 'TensorFlow: Large-scale machine learning on heterogeneous systems'. Software available from tensorflow.org.
URL: *https://www.tensorflow.org/*

[2] Bengio, Y. [2012], 'Practical recommendations for gradient-based training of deep architectures', *CoRR* **abs/1206.5533**.
URL: *http://arxiv.org/abs/1206.5533*

[3] Dozat, T. [2016], Incorporating nesterov momentum into adam.

[4] Glorot, X. and Bengio, Y. [2010], Understanding the difficulty of training deep feedforward neural networks, *in* Y. W. Teh and M. Titterington, eds, 'Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics', Vol. 9 of *Proceedings of Machine Learning Research*, PMLR, Chia Laguna Resort, Sardinia, Italy, pp. 249–256.
URL: *http://proceedings.mlr.press/v9/glorot10a.html*

[5] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. [2014], Generative adversarial nets, *in* Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence and K. Q. Weinberger, eds, 'Advances in Neural Information Processing Systems 27', Curran Associates, Inc., pp. 2672–2680.
URL: *http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf*

[6] Harper, F. M. and Konstan, J. A. [2015], 'The movielens datasets: History and context', *ACM Trans. Interact. Intell. Syst.* **5**(4), 19:1–19:19.
URL: *http://doi.acm.org/10.1145/2827872*

[7] He, K., Zhang, X., Ren, S. and Sun, J. [2015], 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification', *CoRR* **abs/1502.01852**.
URL: *http://arxiv.org/abs/1502.01852*

[8] He, X., Liao, L., Zhang, H., Nie, L., Hu, X. and Chua, T.-S. [2017], 'Neural collaborative filtering', *ArXiv* **abs/1708.05031**.

[9] Hug, N. [2017], 'Surprise, a Python library for recommender systems', `http://surpriselib.com`.

[10] Ioffe, S. and Szegedy, C. [2015], 'Batch normalization: Accelerating deep network training by reducing internal covariate shift', *CoRR*

abs/1502.03167.

URL: *http://arxiv.org/abs/1502.03167*

[11] Kingma, D. P. and Ba, J. [2014], 'Adam: A method for stochastic optimization'. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

URL: *http://arxiv.org/abs/1412.6980*

[12] Koren, Y. [2008], Factorization meets the neighborhood: A multifaceted collaborative filtering model, pp. 426–434.

[13] Koren, Y., Bell, R. and Volinsky, C. [2009], 'Matrix factorization techniques for recommender systems', *Computer* **42**(8), 30–37.

[14] Krizhevsky, A., Sutskever, I. and Hinton, G. E. [2012], Imagenet classification with deep convolutional neural networks, *in* F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, eds, 'Advances in Neural Information Processing Systems 25', Curran Associates, Inc., pp. 1097–1105.

URL: *http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf*

[15] Mikolov, T., Chen, K., Corrado, G. S. and Dean, J. [2013], 'Efficient estimation of word representations in vector space', *CoRR* **abs/1301.3781**.

[16] Nair, V. and E. Hinton, G. [2010], Rectified linear units improve restricted boltzmann machines vinod nair, Vol. 27, pp. 807–814.

[17] Pazzani, M. J. and Billsus, D. [2007], The adaptive web, Springer-Verlag, Berlin, Heidelberg, chapter Content-based Recommendation Systems, pp. 325–341.
URL: *http://dl.acm.org/citation.cfm?id=1768197.1768209*

[18] Pennington, J., Socher, R. and Manning, C. [2014], Glove: Global vectors for word representation, *in* 'Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)', Association for Computational Linguistics, Doha, Qatar, pp. 1532–1543.
URL: *https://www.aclweb.org/anthology/D14-1162*

[19] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. [1986], 'Learning representations by back-propagating errors', *Nature* **323**(6088), 533–536.
URL: *https://doi.org/10.1038/323533a0*

[20] Sarwar, B., Karypis, G., Konstan, J. and Riedl, J. [2001], Item-based collaborative filtering recommendation algorithms, *in* 'Proceedings of the 10th International Conference on World Wide Web', WWW '01, ACM, New York, NY, USA, pp. 285–295.
URL: *http://doi.acm.org/10.1145/371920.372071*

[21] Sherstinsky, A. [2018], 'Fundamentals of recurrent neural network

(RNN) and long short-term memory (LSTM) network', *CoRR* **abs/1808.03314**.
URL: *http://arxiv.org/abs/1808.03314*

[22] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. [2014], 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research* **15**, 1929–1958.
URL: *http://jmlr.org/papers/v15/srivastava14a.html*

[23] Strub, F., Mary, J. and Gaudel, R. [2016], 'Hybrid recommender system based on autoencoders', *CoRR* **abs/1606.07659**.
URL: *http://arxiv.org/abs/1606.07659*

[24] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y. and Manzagol, P.-A. [2010], 'Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion', *J. Mach. Learn. Res.* **11**, 3371–3408.
URL: *http://dl.acm.org/citation.cfm?id=1756006.1953039*

[25] Xu, B., Wang, N., Chen, T. and Li, M. [2015], 'Empirical evaluation of rectified activations in convolutional network', *CoRR* **abs/1505.00853**.
URL: *http://arxiv.org/abs/1505.00853*

[26] Zhao, Z. and Shang, M. [2010], User-based collaborative-filtering recommendation algorithms on hadoop, *in* '2010 Third International Conference on Knowledge Discovery and Data Mining', pp. 478–481.

Listing 1: Generator class and usage

```
class Generator(Sequence):
    def __init__(self, data, user_avg_ratings,
            movie_avg_ratings, batch_size):
        self.data = data
        self.user_avg_ratings = user_avg_ratings
        self.movie_avg_ratings = movie_avg_ratings
        self.batch_size = batch_size

    def __len__(self):
        return int(np.floor(len(self.data) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch = self.data.take(np.arange(idx * self.batch_size, (idx + 1) * self.batch_size, 1), 0)
        userIds = batch.loc[:]['userEmbeddingId'].values
        movieIds = batch.loc[:]['movieEmbeddingId'].values
        genreEmbeddings = np.array(list(map(lambda x: np.array(x), batch.loc[:]['genreEmbedding'].values)))
        ratings = batch.loc[:]['rating'].values

        userAvgRatings = np.array([self.user_avg_ratings.get(i) for i in userIds.tolist()])
        movieAvgRatings = np.array([self.movie_avg_ratings.get(i) for i in movieIds.tolist()])

        return [[userIds, userAvgRatings, movieIds, movieAvgRatings, genreEmbeddings], ratings]

train_gen = Generator(self.train, self.user_avg_ratings, self.movie_avg_ratings, self.batch_size)
validate_gen = Generator(self.test, self.user_avg_ratings, self.movie_avg_ratings, self.batch_size)

history = self.model.fit_generator(
    generator=train_gen,
    validation_data=validate_gen,
    epochs=epochs,
    callbacks=_callbacks,
    workers=12,
    max_queue_size=200,
    shuffle=True,
    use_multiprocessing=True
)
```

Listing 2: Movie Schema

```
const MovieSchema = new Schema ({
    movieId: {
        type: String,
        unique: true,
        required: [true, 'Id is required!'],
        trim: true,
```

```
    },
    title: {
        type: String,
        required: [true, 'Title is required!'],
        trim: true,
    },
    genres: {
        type: String,
        required: [true, 'Genres is required!'],
        trim: true,
        get: genreToArray,
    },
    imdbId: {
        type: String,
        unique: true,
    },
    tmdbId: {
        type: String,
        unique: true,
    },
    posterUrl: {
        type: String,
    },
    production: {
        type: Schema.Types.ObjectId,
        ref: 'Production'
    },
    castMembers: {
        type: [Schema.Types.ObjectId]
    },
    summary: {
        type: String,
    },

}, {autoIndex: false})

export default mongoose.model('Movie', MovieSchema)
```

Listing 3: Movie api routes

```
const routes = (app) => {
    app.route('/health').get(asyncMiddleware(health))
    app.route('/:id').get(asyncMiddleware(getMovie))
    app.route('/search').post(asyncMiddleware(searchMovies))
}
```

Listing 4: Kubernetes Deployment Yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: gateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gateway
  progressDeadlineSeconds: 120
  template:
    metadata:
      name: gateway
      labels:
        app: gateway
    spec:
      hostAliases:
      - ip: "10.0.2.2"
        hostnames:
          - "api.gateway.com"
      containers:
      - name: gateway
        resources:
          requests:
            memory: "64Mi"
          limits:
            memory: "256Mi"
        image: authentication:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

Listing 5: Kubernetes Service Yaml

```
apiVersion: v1
kind: Service
metadata:
  name: gateway
spec:
  type: NodePort
  selector:
    app: gateway
  ports:
  - name: http
    port: 8080
```

```
    targetPort: 8080
    protocol: TCP
```

Listing 6: DockerFile Example

```
FROM node:10-alpine

WORKDIR /usr/src/app

RUN npm install --global yarn

COPY package*.json ./

RUN apk add --no-cache --virtual .gyp \
        python \
        make \
        g++ \
    && yarn install \
    && apk del .gyp


COPY . .

EXPOSE 8080

RUN yarn run build

CMD [ "yarn", "serve" ]
```